ANYTHING YOU CAN DO I CAN DO BETTER.

[brandon] Hello! I'm Brandon, and [I'm] Lisa, we're engineers on the native team at Kickstarter, doing both iOS and Android. We're a pretty small team of engineers, ranging from junior to senior, and we've been on a journey for the past 2 years of unifying our foundations across iOS and Android.

[brandon] For example, I was originally hired as an iOS engineer at Kickstarter, but jumped onto Android when we started working on our 1.0

[brandon] My esteemed colleague Lisa on the other hand was hired as an Android engineer, but happily started writing Swift when we started on our rewrite and re-architecture.

[brandon] But we always had a few core ideas that we held closely so that we could share knowledge while working on two platforms even though we couldn't necessarily share code.

[brandon] However, that doesn't mean there isn't still a bit of identity involved. I would still say I primarily identify as an iOS engineer in my day-to-day, and Lisa would probably identify as an android engineer, even though she's given talks at more iOS conferences than Android and has made a guest appearance on Chris and Florian's swift talks :D

[brandon] So, Lisa and I would like to give you a tour of our platforms of choice to show off their strengths and weaknesses, but still hoping to convey that really we're all just doing the same thing if you embrace a few core ideas

OPEN SOURCED

HTTPS://GITHUB.COM/KICKSTARTER/IOS-OSS
HTTPS://GITHUB.COM/KICKSTARTER/ANDROID-OSS

[Lisa] Oh also worth noting that all of our code for our iOS and Android apps is open sourced on Github, so you can go see how we use all of the things we are going to talk about.

KOTLIN & SWIFT

[Lisa] Let's start with the language we use for our respective platforms.

[Lisa] So everyone here is clearly already familiar with Swift, but you may not have heard of Kotlin.

KOTLIN

- JVM LANGUAGE
- BUILT BY JETBRAINS
- 100% INTEROP WITH JAVA
- OOP WITH A BIT OF FP
- VERY EXPRESSIVE

[Lisa] It's a JVM language...

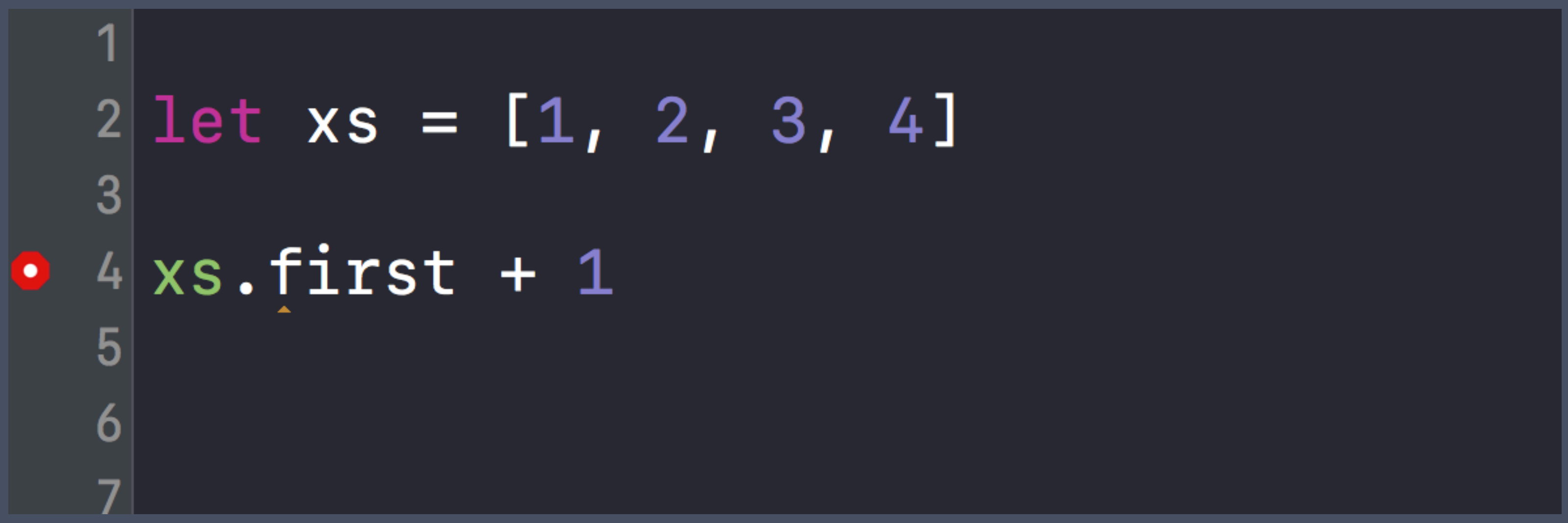
[lisa] ...that is built by JetBrains, the makers of Android Studio, the most popular IDE for android, and IntelliJ.

[Lisa] Its aim is to have 100% interop with Java, which is a bit different from Swift. They want all Kotlin code to be reachable from Java. This is a great thing, but sometimes this holds kotlin back a bit.

[Lisa] It has a similar philosophy as Swift in that it's primarily an OOP language but has given first-class support to a few small features from functional programming.

[Lisa] And it is *very* expressive in some really beautiful ways that we'll get into soon.

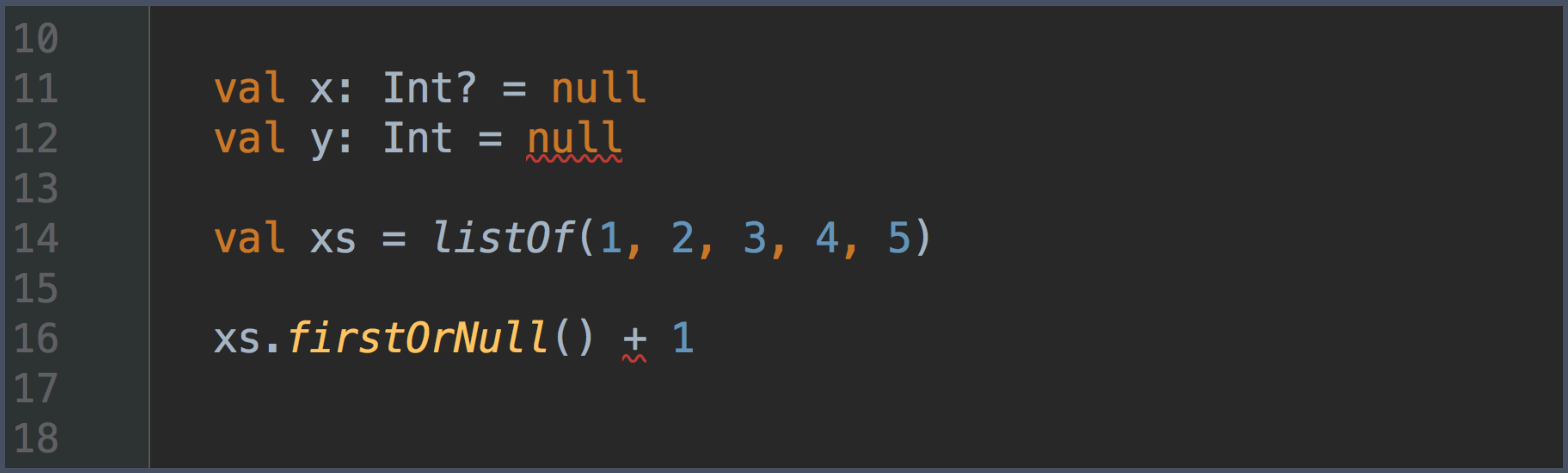
OPTIONALS



```
let xs = [1, 2, 3, 4]

xs.first + 1
```

[brandon] Ok, so one of the nice features of swift is the optional type. It allows you to safely express the idea of the absence of a value. So here, I have an array of integers and I want to add one to the first element. Swift is stopping me from doing this because it cannot know that the array of `xs` is not empty. I have to explicitly handle the case that the array is empty and `first` returns `nil`.

OPTIONALS

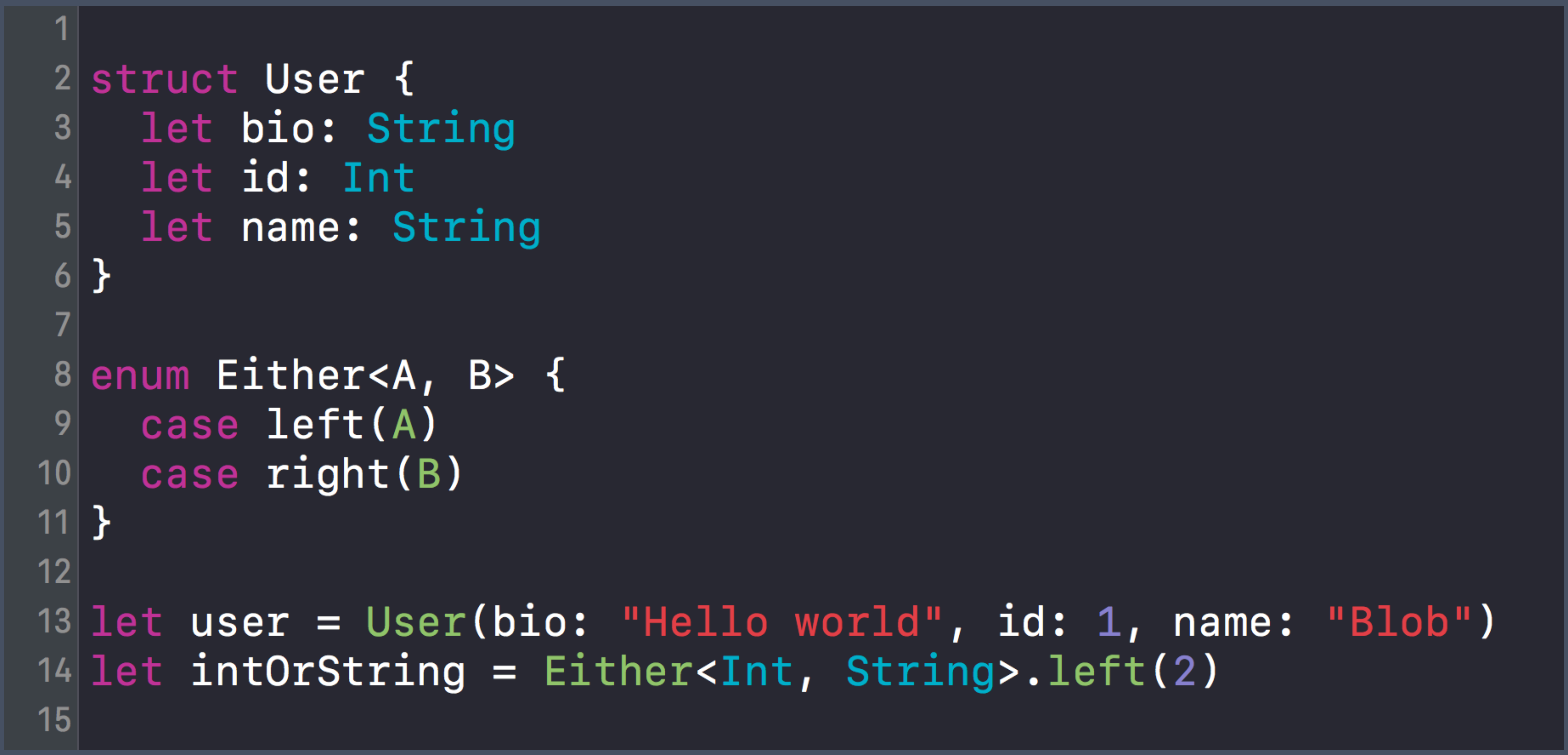


```
val x: Int? = null
val y: Int = null

val xs = listOf(1, 2, 3, 4, 5)

xs.firstOrNull() + 1
```

[Lisa] Yeah, optionals and null-safety are great. Fortunately Kotlin has made this a first-class concern. Here we see how we have to explicitly tell kotlin that `x` can hold a `null` value, using the same `?` notation as in Swift, and in the case of `y` Kotlin has prevented us from storing `null` since we have marked its type as a non-nullable `Int`.

[Lisa] We also have an array of integers and this `firstOrNull` method, which behaves like swift's `first` method, and similarly kotlin is preventing us from adding `1` to an optional integer.

STRUCTS AND ENUMS



```swift

struct User {
  let bio: String
  let id: Int
  let name: String
}

enum Either<A, B> {
  case left(A)
  case right(B)
}

let user = User(bio: "Hello world", id: 1, name: "Blob")
let intOrString = Either<Int, String>.left(2)

```

[brandon] An important part of functional programming is structs and enums, also known as product types and sum types, or even product and coproducts if you wanna go really deep.

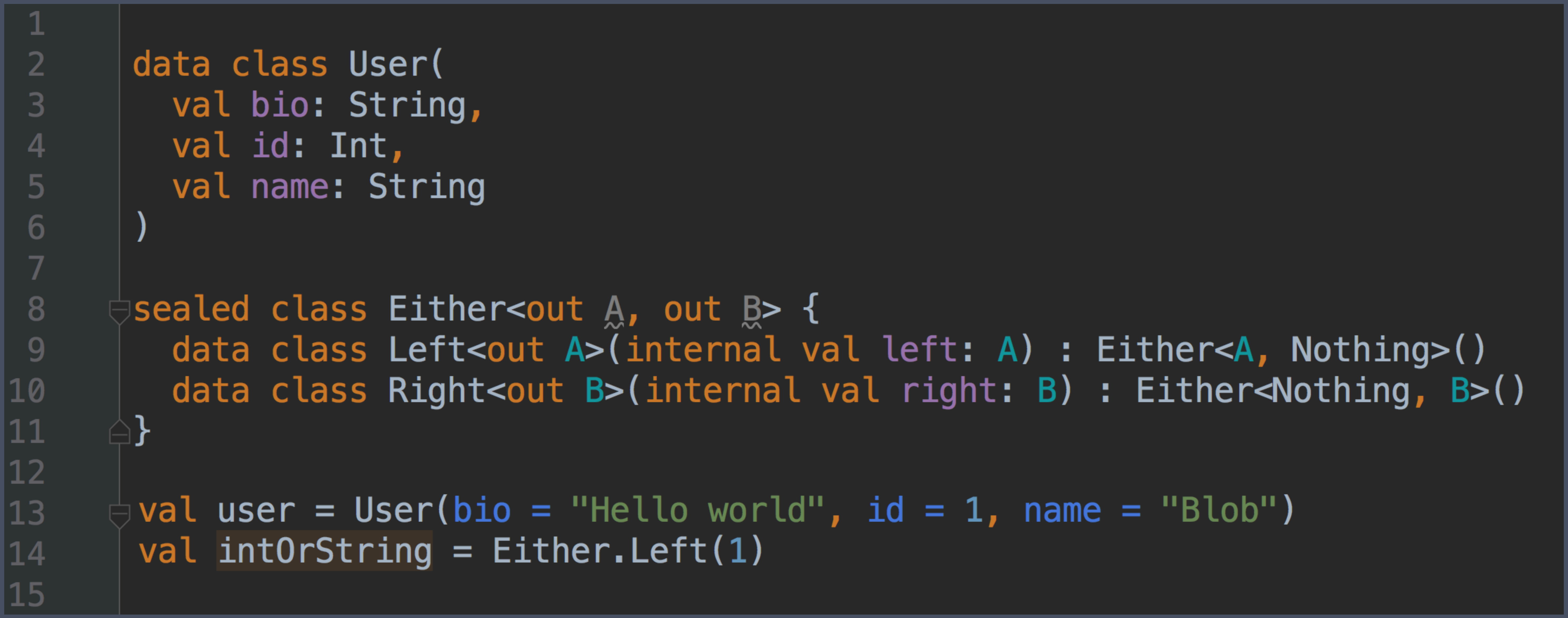
[brandon] these types express the idea of having many values at once, or having one choice of many types of values.

[brandon] they are also well suited for immutability and statelessness

[brandon] Here we have a `User` type that has three fields, and an `Either` type that expresses having either a value of type `A` or type `B`, which we use quite a bit in our code base.

[brandon] and we've instantiated a few values here so that we can see how it's used

DATA CLASSES AND SEALED CLASSES



```kotlin

data class User(
  val bio: String,
  val id: Int,
  val name: String
)

sealed class Either<out A, out B> {
  data class Left<out A>(internal val left: A) : Either<A, Nothing>()
  data class Right<out B>(internal val right: B) : Either<Nothing, B>()
}

val user = User(bio = "Hello world", id = 1, name = "Blob")
val intOrString = Either.Left(1)

```

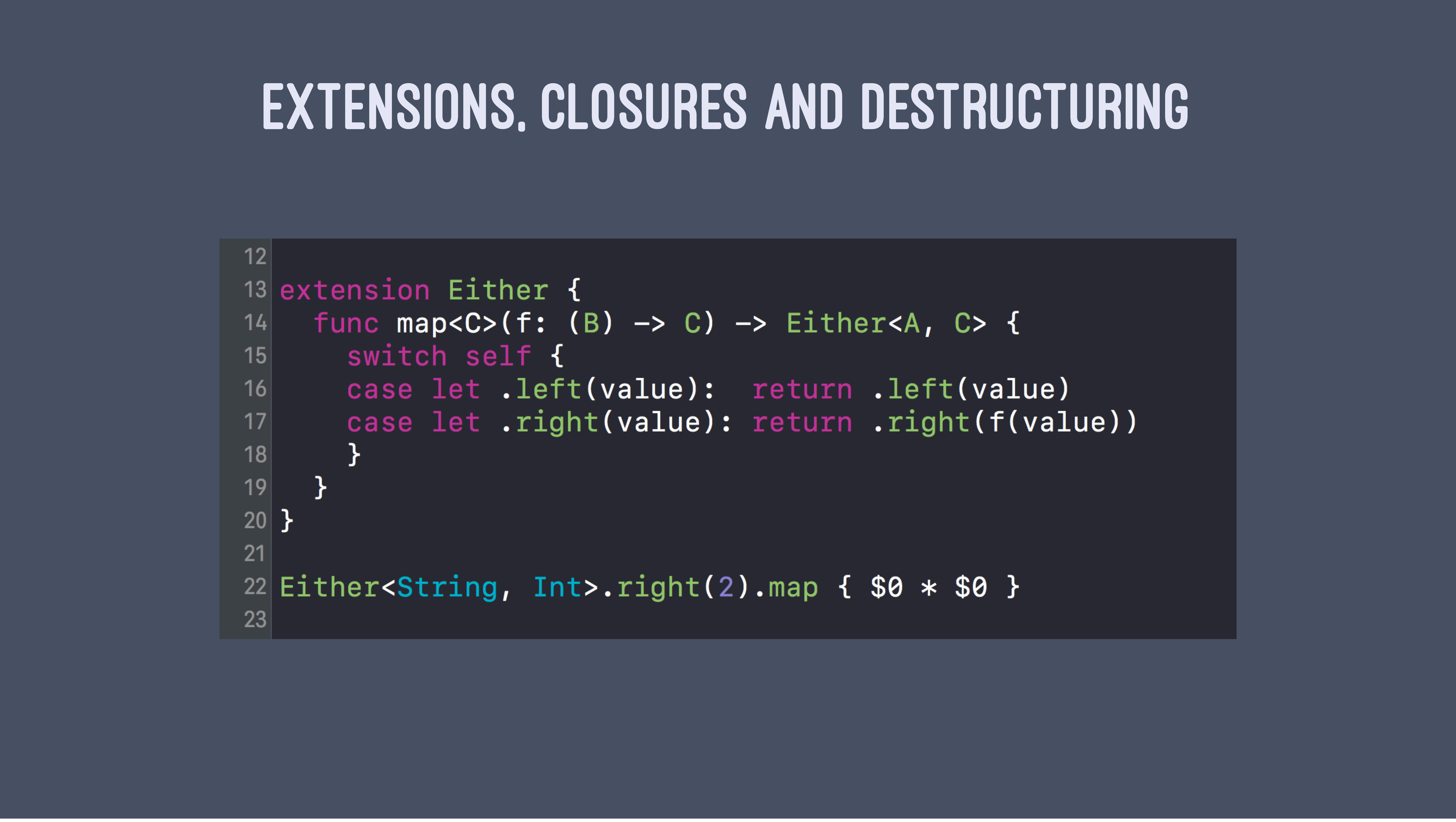
[lisa] Over in the kotlin world, stucts and enums are called `data classes` and `sealed classes`.

[lisa] The data classes have pretty similar style to structs in Swift, and they work pretty much the same.

[lisa] Sealed classes are how we achieve enum-like functionality in kotlin, and they look a bit different. Essentially, we create an un-instantiable type called either, the sealed class, and then have two inner subclasses for modeling the left and right values. it's basically an OOP way to do enums.

[lisa] the amazing part is that this is 100% interoperable with java, so we can use the `Either` type in our java code, and we do.

[lisa] whereas in swift the `Either` enum is not accessible from objective-c at all.



EXTENSIONS, CLOSURES AND DESTRUCTURING

```swift

extension Either {
  func map<C>(f: (B) -> C) -> Either<A, C> {
    switch self {
    case let .left(value):  return .left(value)
    case let .right(value): return .right(f(value))
    }
  }
}

Either<String, Int>.right(2).map { $0 * $0 }

```

[brandon] Here we are showing off a couple of cool things in Swift.

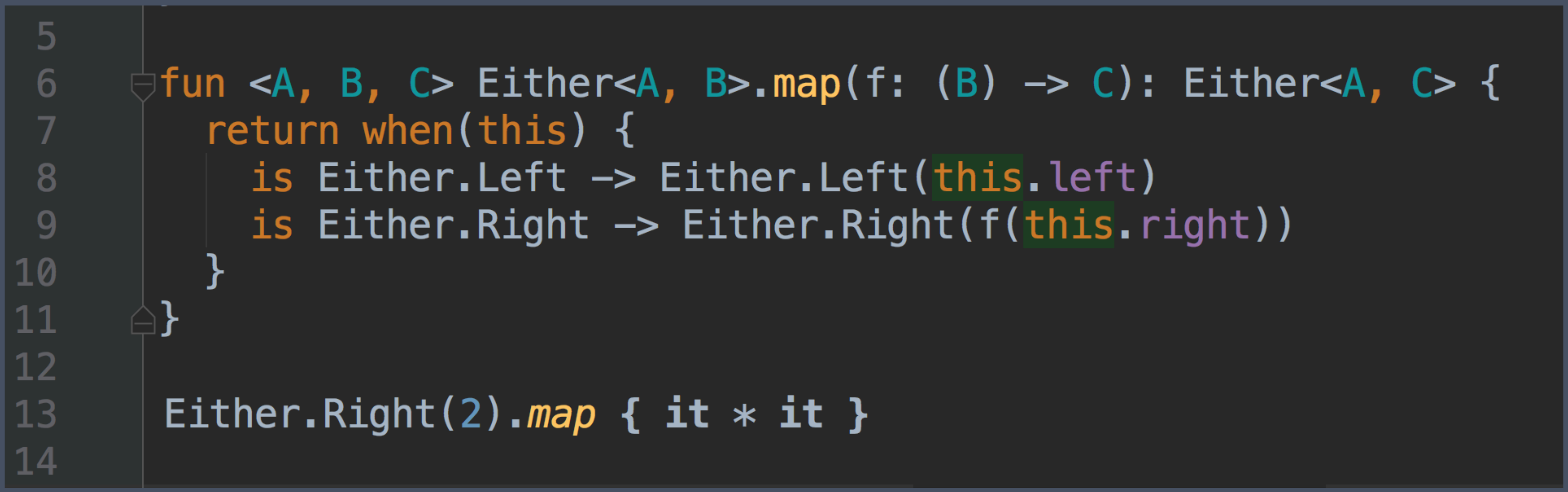
[brandon] First, we can open up the `Either` type and add functions to it. Here I've added a *map* function, which allows one to transform the type on the right B to a different type C.

[brandon] Second, functions can take functions as arguments, which allows us to pass in this transformation as an argument.

[brandon] Third, we have `switch` for destructuring an `Either` in order to process the left and right separately, and getting compile time guarantees that we handled all the cases properly.

[brandon] and then we can use it by constructing a right value, and mapping it

EXTENSIONS, CLOSURES AND DESTRUCTURING



```kotlin
fun <A, B, C> Either<A, B>.map(f: (B) -> C): Either<A, C> {
  return when(this) {
    is Either.Left -> Either.Left(this.left)
    is Either.Right -> Either.Right(f(this.right))
  }
}

Either.Right(2).map { it * it }
```

[lisa] first, to extend a type you just define a new function on the type itself using dot `.`.

[lisa] also, functions are supported as values in kotlin so that we can provide the transformation function as an argument to `map`. this is called a "higher order function"

[lisa] finally, we can use `when` to destructure the `Either` into each of its inner subclasses.

[lisa] With `when`, we have compile time safety that we handled both the left and the right cases.

[lisa] and finally, we can use it much the same way as we did with swift. big difference here is that instead of `$0` we

EXTENSIONS, CLOSURES AND DESTRUCTURING

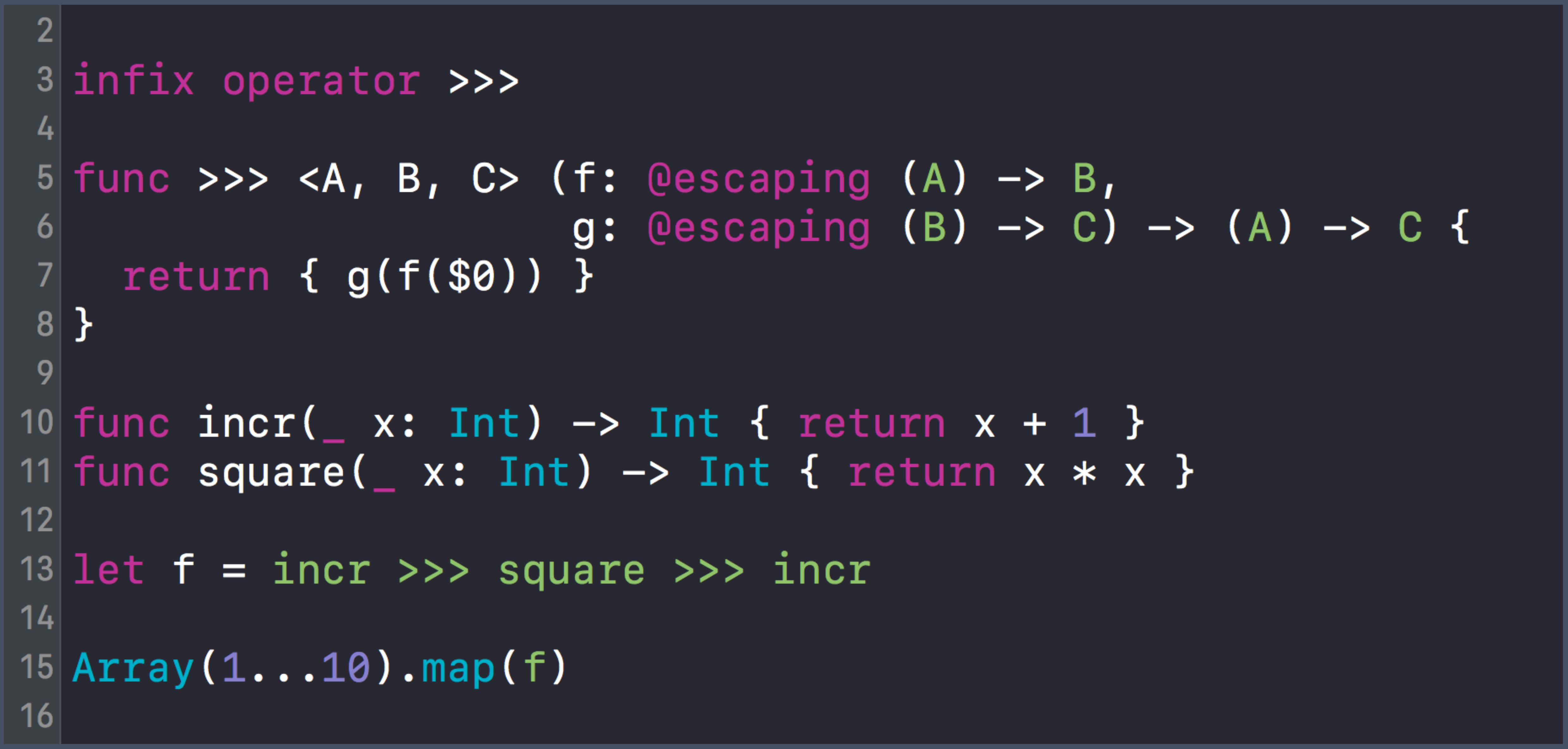
EVEN BETTER...

```kotlin

fun <A, B, C> Either<A, B>.map(f: (B) -> C): Either<A, C> = when(this) {
  is Either.Left -> Either.Left(this.left)
  is Either.Right -> Either.Right(f(this.right))
}

```

[lisa] And, anything you can do I can do better, we can write this function as an expression. We can use this syntax in Kotlin because `when` is treated as an expression.

[lisa] and just to remind everyone, this is fully interoperable with java. we can construct `Either` values, call kotlin functions that accept and return `Either`s, all from Java.

OPERATORS



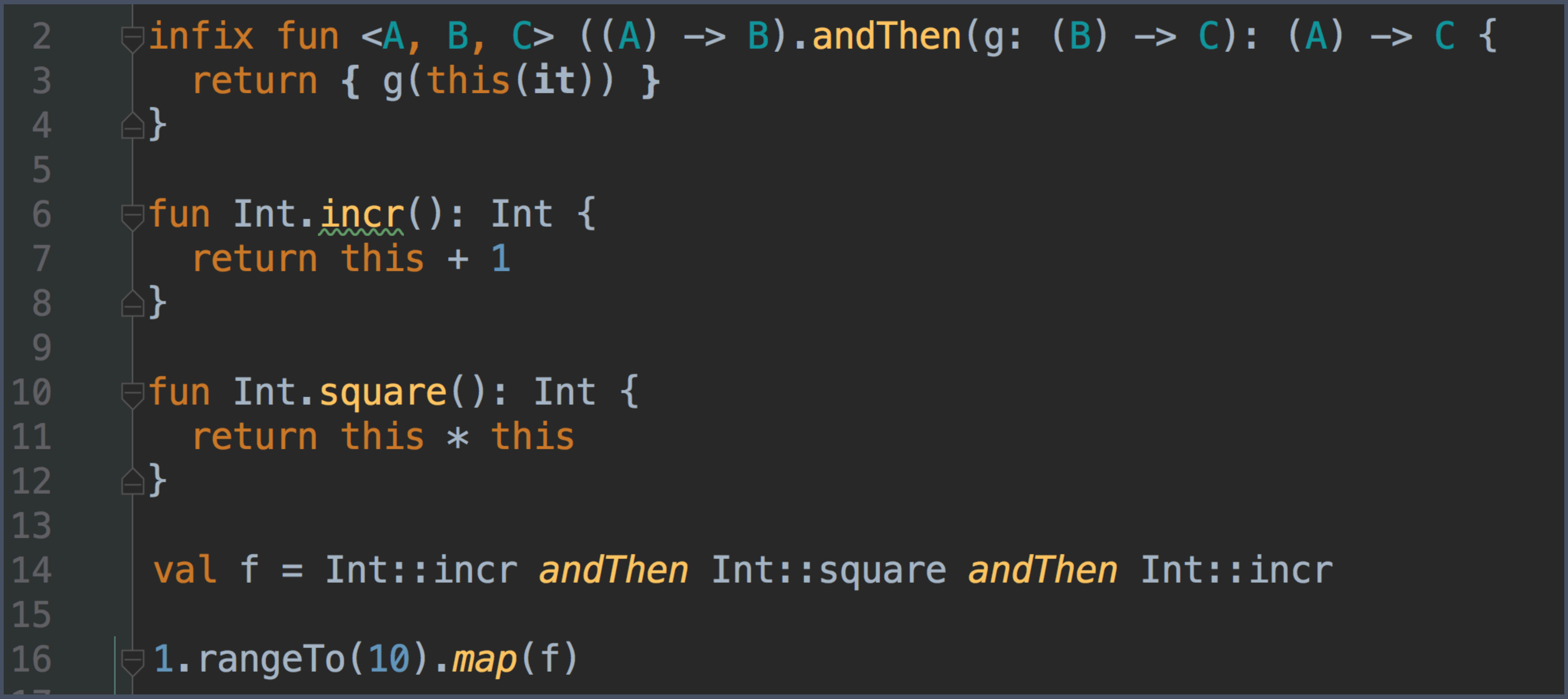
```swift

infix operator >>>

func >>> <A, B, C> (f: @escaping (A) -> B,
                    g: @escaping (B) -> C) -> (A) -> C {
  return { g(f($0)) }
}

func incr(_ x: Int) -> Int { return x + 1 }
func square(_ x: Int) -> Int { return x * x }

let f = incr >>> square >>> incr

Array(1...10).map(f)

```

[brandon] Swift has support for operators which allows us to write expressive code with nice algebraic properties. Here we have defined an arrow operator to represent forward composition of functions.

[brandon] We can then take a couple of lil pure functions, `incr` and `square`, and derive new functions from composition.

[brandon] Then, we can use that function to *map* an array of integers to a new array of integers.

OPERATORS



```
infix fun <A, B, C> ((A) -> B).andThen(g: (B) -> C): (A) -> C {
  return { g(this(it)) }
}

fun Int.incr(): Int {
  return this + 1
}

fun Int.square(): Int {
  return this * this
}

val f = Int::incr andThen Int::square andThen Int::incr

1.rangeTo(10).map(f)
```

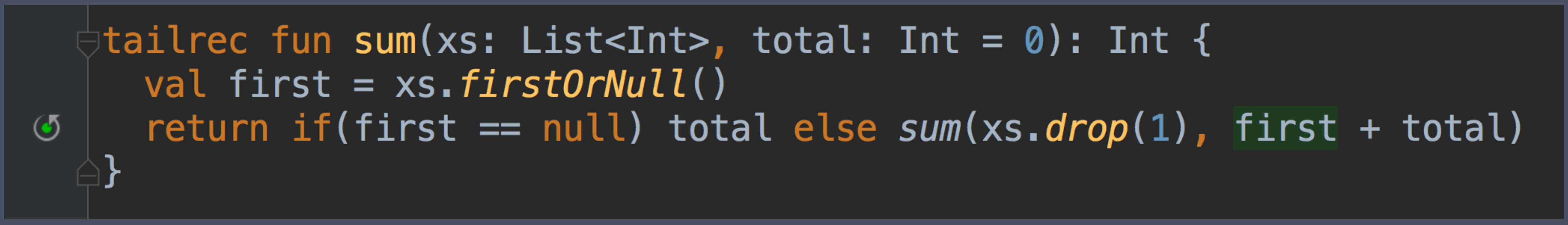
[lisa] Kotlin doesn't have support for custom operators, but it does allow you to define `infix` functions. This means you can define a function that takes two arguments, but use it in an infix manner.

[lisa] for example, here we have defined an `andThen` function that takes a function from `A` to `B` on the left, and a function `B` to `C` on the right, and returns a function from `A` to `C`. This allows us to chain the increment and square functions together in any way we want.

[lisa] and finally, we can feed that function to `map` to transform an array of integers to a new array of integers.

TAIL RECURSION

KOTLIN



```kotlin
tailrec fun sum(xs: List<Int>, total: Int = 0): Int {
  val first = xs.firstOrNull()
  return if(first == null) total else sum(xs.drop(1), first + total)
}
```

[lisa] Here's a cool feature of Kotlin that allows us to specify when a recursive function can take advantage of tail recursion. Recursion is an important part of functional programming because it allows for us to focus on the structure of data.

[lisa] Let's remind ourselves that a recursive function is said to be in "tail form" if the return statement of the function contains only a call to the function itself, and nothing else. such recursive functions can be optimized by unrolling the recursion into a loop.

[lisa] kotlin has direct support for this optimization. if you can write your recursive function in tail form, you can annotate the function with the `tailrec` keyword and kotlin will optimize the function to be a plain ol' `for` loop. it'll even raise a compiler warning if you use the modifier on a function that is not properly in tail form.

[lisa] here I have a `sum` function that shows how to recursively define the sum of a list of integers as the sum of the head plus the sum of the tail. since it's easy enough to write this in tail form, i can now sum a list of thousands of integers without worrying about blowing up the stack.

TAIL RECURSION

SWIFT 😭

```swift

func sum(_ xs: ArraySlice<Int>, total: Int = 0) -> Int {
  guard let first = xs.first else { return total }
  return sum(xs.dropFirst(), total: first + total)
}

```

[brandon] So this is a bit sad for Swift. We have no tail call guarantees. It could happen, but you can't count on it.

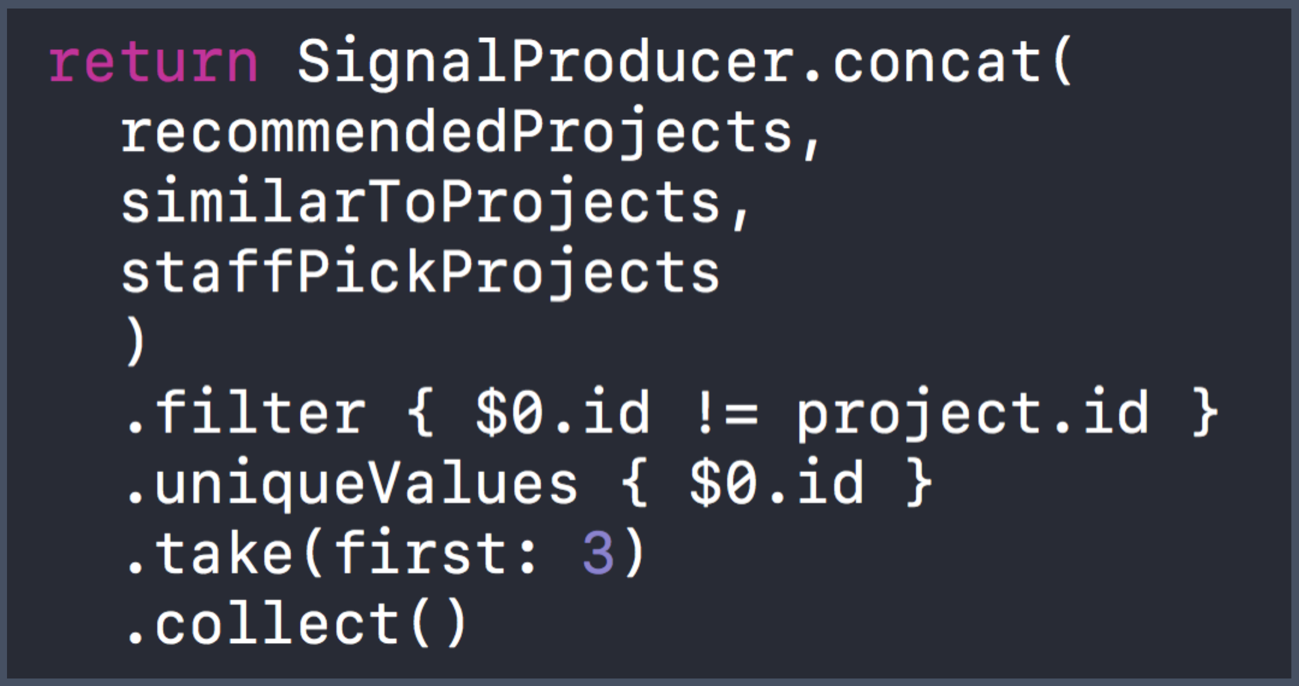
[brandon] Here I have defined the `sum` function that Lisa defined, but this could very easily blow up the stack since it cannot be guaranteed to be optimized.

FUNCTIONAL PROGRAMMING

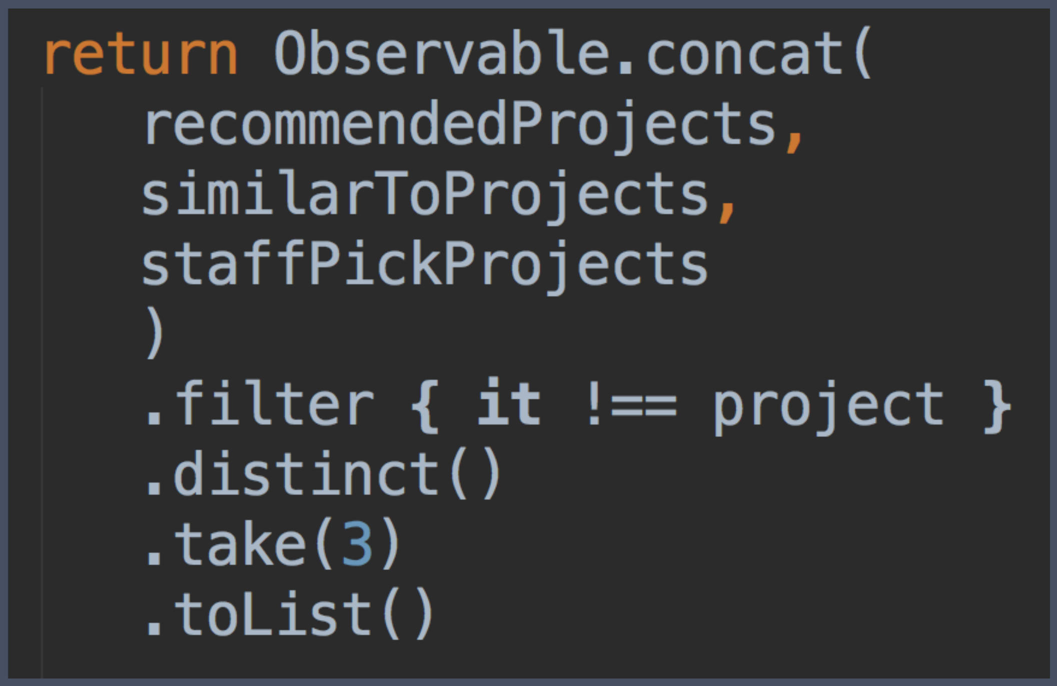
[lisa] In case you haven't already guessed, we like functional programming because it allows for us to leverage functions, immutability, and minimal side effects to write composable and testable code.

[lisa] Swift and Kotlin are not functional languages, but they do offer first-class support for many functional features such as map and filter operators. They are good foundations for building functional frameworks.

FUNCTIONAL PROGRAMMING



```swift
return SignalProducer.concat(
  recommendedProjects,
  similarToProjects,
  staffPickProjects
  )
  .filter { $0.id != project.id }
  .uniqueValues { $0.id }
  .take(first: 3)
  .collect()
```



```kotlin
return Observable.concat(
    recommendedProjects,
    similarToProjects,
    staffPickProjects
    )
    .filter { it !== project }
    .distinct()
    .take(3)
    .toList()
```

ReactiveSwift on the left and RxJava on the right.

[lisa] This code is run on the thanks screen after backing a project. We want to show you 3 projects, preferably 3 projects recommended for you, but in case you dont have enough recommendations we will pull 3 similar projects to the one you just backed, and there are still arent 3 projects we will fallback to just some staff curated projects.

[lisa] So usually to get these projects we would have to perform multiple API requests to get all the types of projects, concat them together, and then take 3. But, working with signals and observables, the take(3) you see here will complete the entire chain, preventing us from doing more API requests than needed. For example, if the API request for recommended projects returned 3 projects, we wouldn't even execute the other two requests to fetch projects.

[lisa] This is an example of lazy evaluation which actually saves us from making excess API requests.

[lisa] The rest of the logic filters out the project you just backed and that there are no repeats in projects.

[lisa] It's cool to see how such complicated logic can be written in nearly an identical way on both platforms, and two languages, without appealing to platform specific tools like URLSession or async task.

ANYTHING YOU CAN DO WE CAN DO TOGETHER

So, after two years of developing with a team where everyone is expected to contribute to both ios and android, we've come to the conclusion that we want to develop our foundation on the ideas of functional programming, and then implement them into our respective platforms. We invest a lot of time in experimenting with ideas, for example we are quite deep in the world of lenses for working with immutable data and applicative parsing for safely handling blobs of data, so we want to be able to share those findings with each other, regardless of platform.

Further, it feels good to wipe away the divide between android and ios and the "anything you can do i can do better" attitude. we benefit tremendously from encouraging everyone to be fluent in all ios/android/java/swift/kotlin, sometimes in ways that are not completely obvious. for example, it is certainly true that i write more swift than java/kotlin these days, but i also review all of lisa's android pull requests. so we are able to distribute a lot of work and understanding, beyond just the act of writing code.

@MBRANDONW
@LUOSER

thanks everyone