Functional Programming & Kotlin

Pick the **Best** - **Skip** the **Rest**

by Urs Peter

LinkedIn: bit.ly/urs-peter-linked-in

Email: upeter@xebia.com

Blog: xebia.com/blog/

About Me:

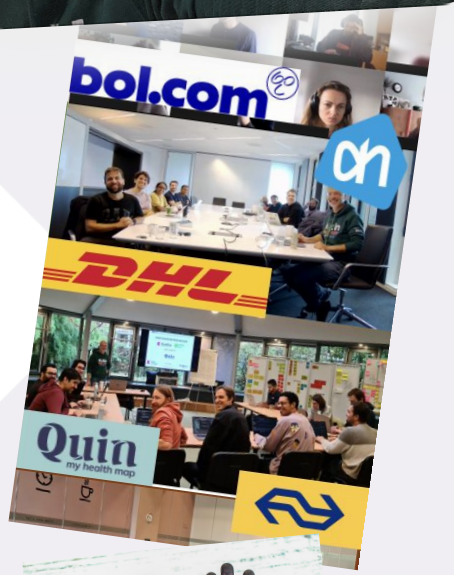
Kotlin Training
Certified by JetBrains

Xebia Academy

About Me:

Kotlin Training
Certified by JetBrains

Xebia Academy



Kotlin

Scala

Java

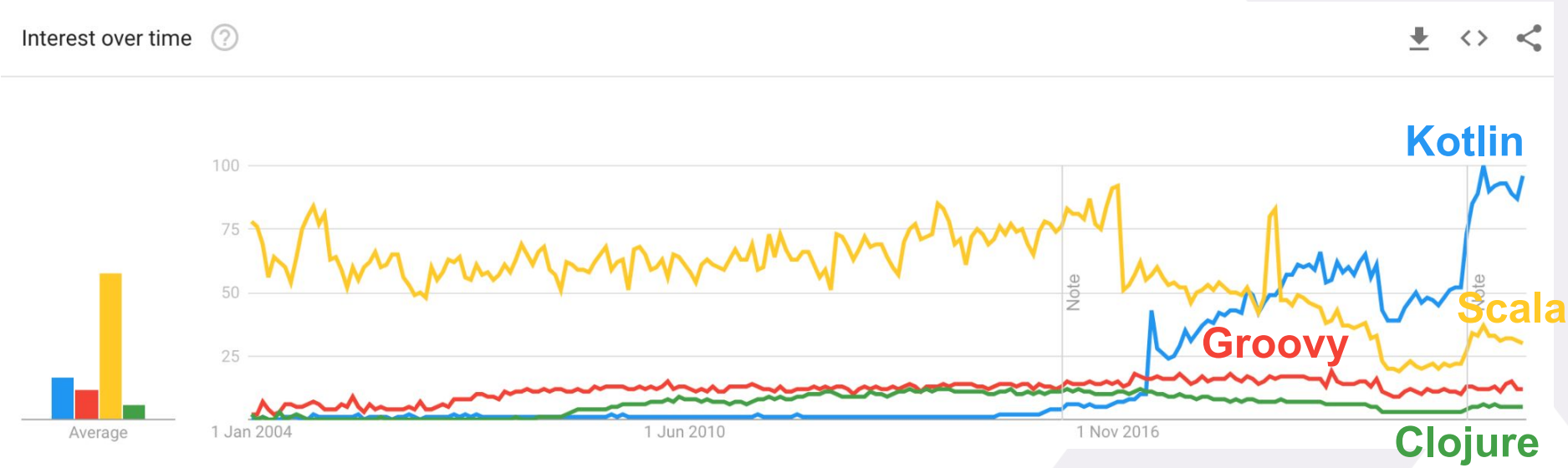
Nice to meet:

Why this talk?

Why?

Kotlin

Scala



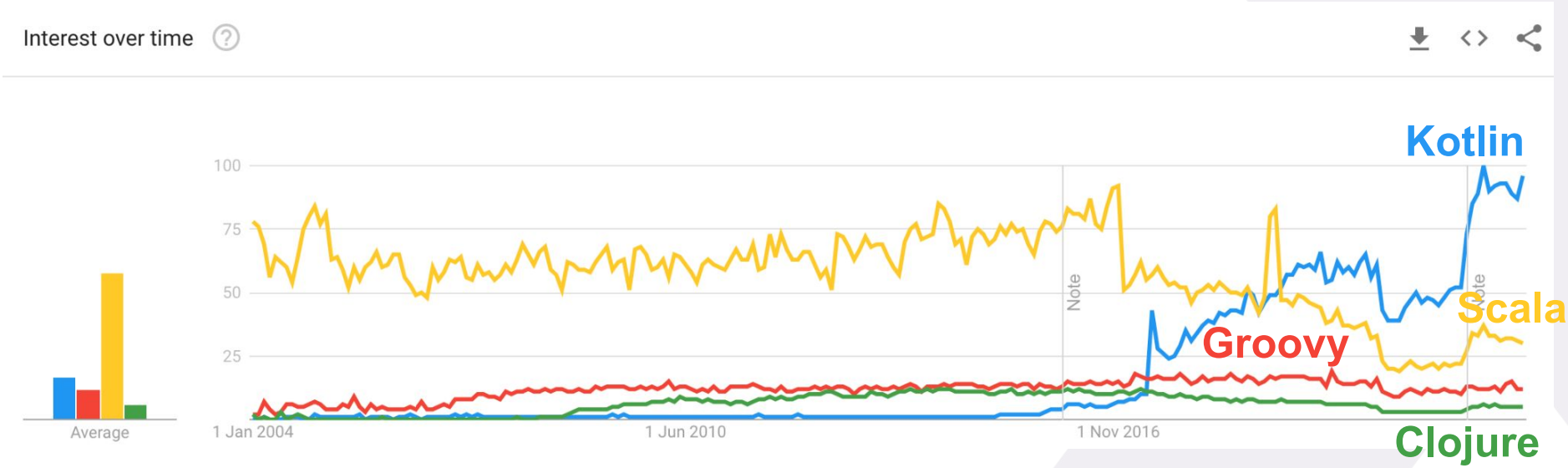
https://trends.google.com/trends/explore?date=all&q=%2Fm%2F0_lcrx4,%2Fm%2F02js86,%2Fm%2F091hdj,%2Fm%2F03yb8hb

Why this talk?

Introduced ***disproportionate complexity*** for the problems at hand mainly due to: **Functional Programming**

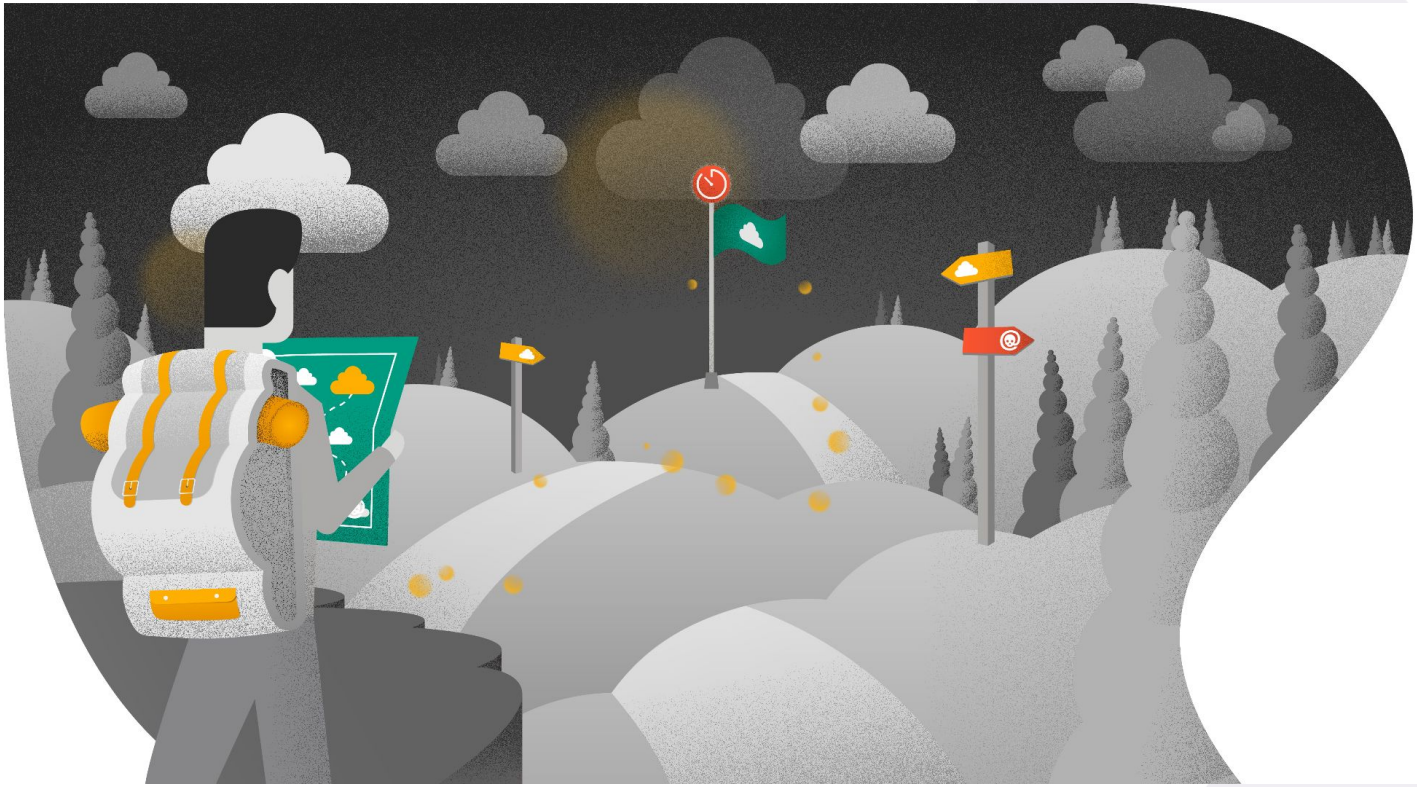
Kotlin ↗

Scala ↘



https://trends.google.com/trends/explore?date=all&q=%2Fm%2F0_lcrx4,%2Fm%2F02js86,%2Fm%2F091hdj,%2Fm%2F03yb8hb

Let’s get ready for our functional journey!



Declaring and Calling Functions

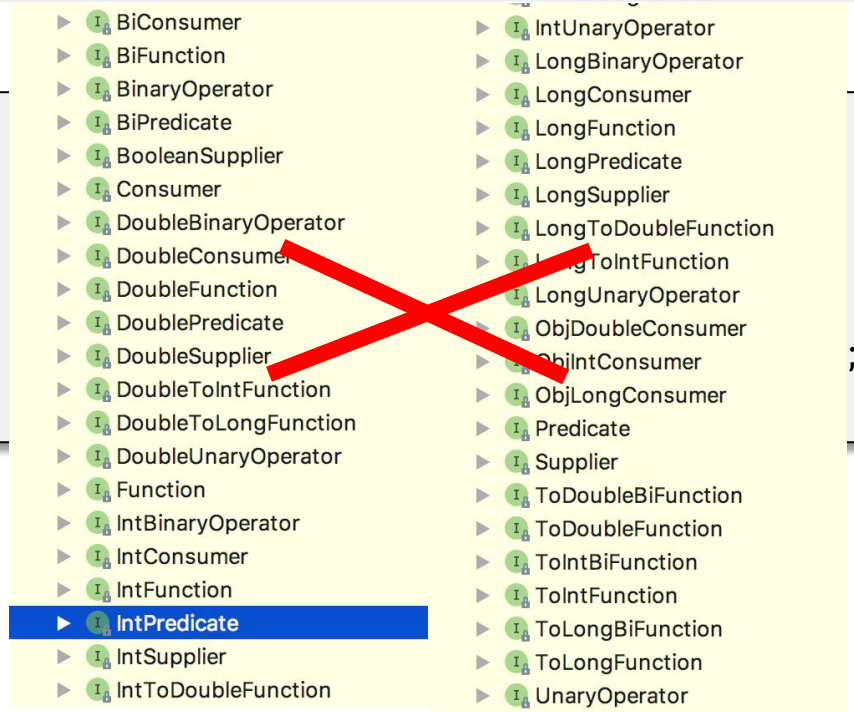
In Kotlin Functions are *first class citizens*. Unlike Java, there is syntax support for *declaring* functions

Java

```
public String doWithText(
        File file,
        Function<String, String> f
) throws IOException {
 return
     f.apply(readFileToString(file));
}
```

Kotlin

```
fun doWithText(
  file: File,
  f: (String) -> String
): String =
 f(readFileToString(file))
```




```
fun needsProcessing(
  file: File,
  predicate: (Long) -> Boolean
): Boolean =
 predicate(file.length())
```

...which makes function declarations easier to *write* and *read* (also because Kotlin has no primitives).

Declaring and Calling Functions

Java

```
public String doWithText(
        File file,
        Function<String, String> fun
) throws IOException {
 return
     fun.apply(readFileToString(file));
}
```

Kotlin

```
fun doWithText(
  file: File,
  f: (String) -> String
): String =
 f(readFileToString(file))
```

```
doWithText(new File("/text.txt"),txt ->
  txt.toUpperCase()
);

doWithText(new File("/text.txt"),
  String::toUpperCase);
```

```
doWithText(File("/text.txt")){txt ->
  txt.uppercase()
}

doWithText(File("/text.txt"),
  String::uppercase)


doWithText(File("/text.txt")){
  it.uppercase()
}
```

Declaring and Calling Functions

Java

```java
public void writeToFile(
        File file,
        Supplier<String> block
) throws IOException {
  if(file.isFile)
    writeStringToFile(file, block.get());
}
```

Kotlin

```kotlin
fun writeToFile(
  file:File,
  block: () -> String
) {
  if(file.isFile)
    writeStringToFile(file, block())
}
```

```java
writeToFile(new File("/out.txt"), () ->
  String.join("",
    Collections.nCopies(
      1000,
      "Hello world! "
    )
  )
);
```

```kotlin
writeToFile(File("/out.txt")){
  "Hello world! ".repeat(1000)
}
```

Declaring and Calling Functions

Java

```
public String doWithText(
        File file,
        Function<String, String> fun
) throws IOException {
 return
     fun.apply(readFileToString(file));
}
```

Kotlin

```
fun doWithText(
  file: File,
  f: String.() -> String
): String =
 f(readFileToString(file))
```

With a slightly different function declaration syntax…

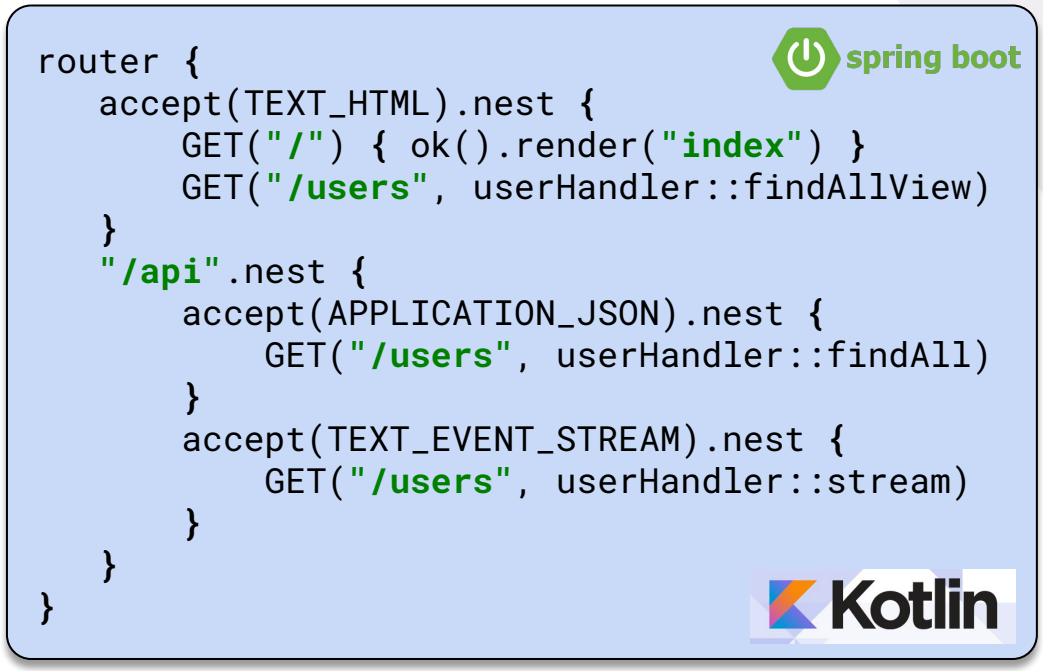
```
doWithText(new File("/text.txt"),txt ->
  txt.toUpperCase()
);

doWithText(new File("/text.txt"),
  String::toUpperCase);
```

```
doWithText(File("/text.txt")){
  this.uppercase()
}
```

...the lambda parameter of type `String` is *implicitly available* in the function body via `this`. This is a very powerful feature used to create advanced builder.

Functional Literal with Receiver in Action

Here an example of Spring's Routing DSL solely built on Extensions and Function Literals with Receivers:



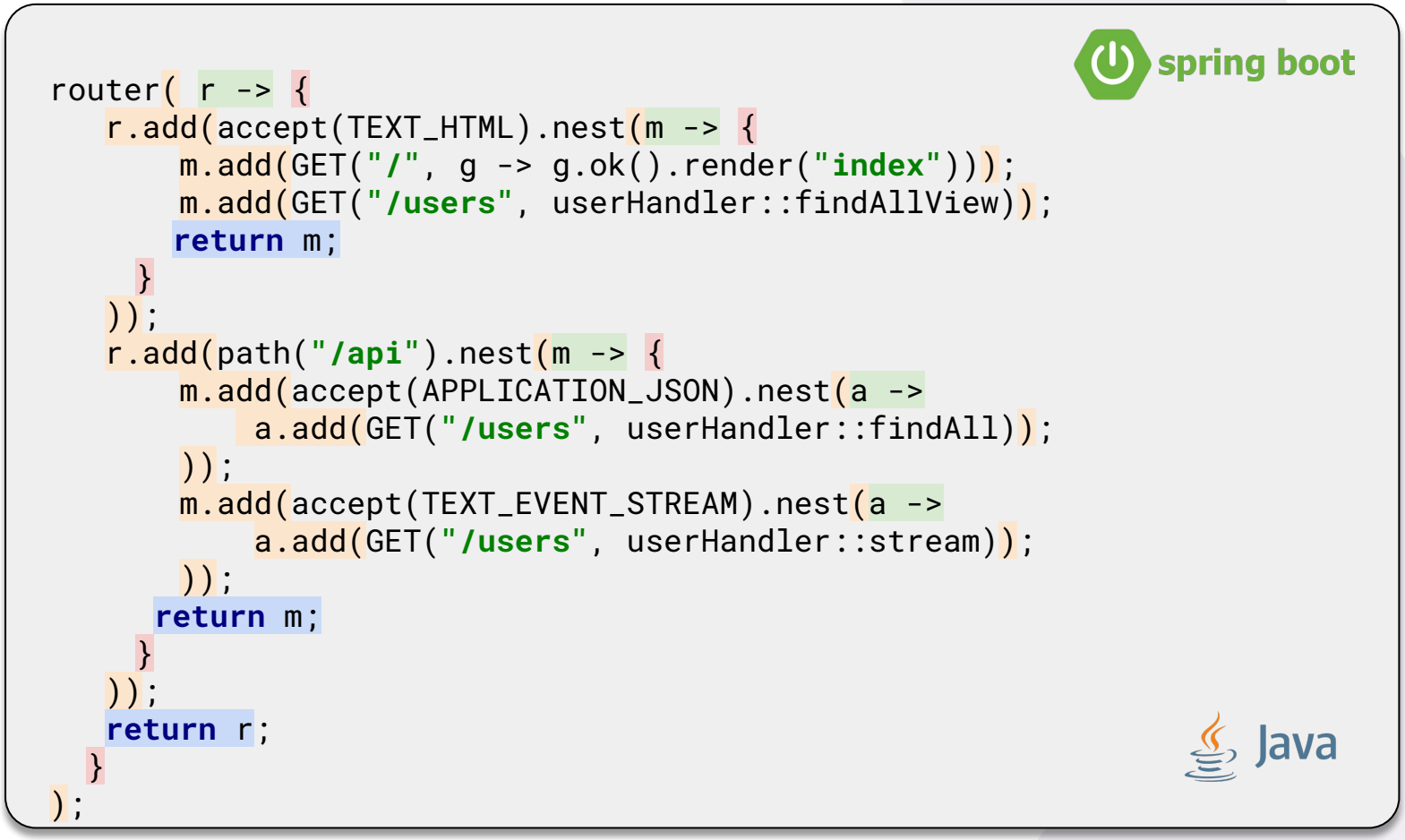
Functional Literal with Receiver in Action

Here an example of Spring's Routing DSL solely built on Extensions and Function Literals with Receivers:

```
router { this:RouterBuilder
    this.accept(TEXT_HTML).nest { this:MethodBuilder
          this.GET("/") { ok().render("index") }
          this.GET("/users", userHandler::findAllView)
    }
    "/api".nest { this:MethodBuilder
        this.accept(APPLICATION_JSON).nest { this:MethodBuilder
          this.GET("/users", userHandler::findAll)
        }
        this.accept(TEXT_EVENT_STREAM).nest { this:MethodBuilder
          this.GET("/users", userHandler::stream)
        }
    }
}
```

Functional Literal with Receiver in Action

Since Java lacks this feature, such as DSL would look as follows:



Let’s start a ‘transforming’ journey

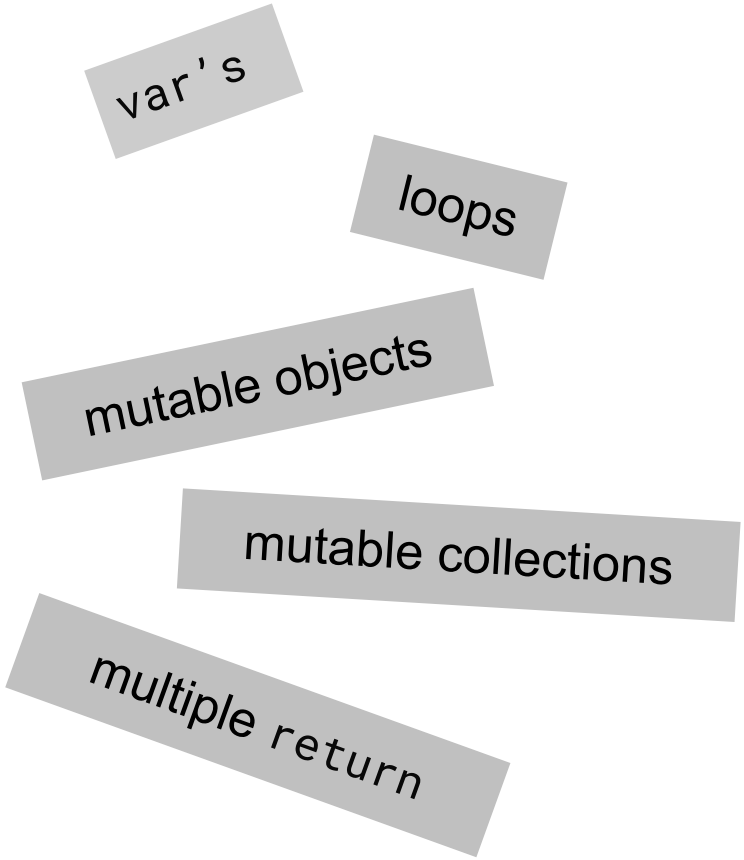
Starting point:
The swamps of Imperative Programming

Programming Styles

Imperative programming

Imperative programming relies on *declaring variables* that are *mutated* along the way.

```
fun findBestDev(lang: String): Developer? {
    var devs:List<Developer>
    try {
        devs = get<Developer>()
    } catch (ex:Exception) {
        return null
    }
    var result = mutableListOf<Developer>()
    for(dev in devs) {
      if(dev.languages.contains(language))
         result.add(dev)
    }
    if(result.isEmpty()) {
        return null
    }
    result.sortBy { it.experience }
    return result[0]
}
```



Programming Styles

Imperative programming

Imperative programming relies on *declaring variables* that are *mutated* along the way.

```
fun findBestDev(lang: String): Developer? {
    var devs:List<Developer>
    try {
        devs = get<Developer>()
    } catch (ex:Exception) {
        return null
    }
    var result = mutableListOf<Developer>()
    for(dev in devs) {
      if(dev.languages.contains(language))
        result.add(dev)
    }
    if(result.isEmpty()) {
        return null
    }
    result.sortBy { it.experience }
    return result[0]
}
```

Expression oriented programming

Expression oriented programming relies on thinking in *functions* where every *input* results in an *output*:

```
fun findBestDev(lang: String): Developer? =
  try {
     get<Developer>()
       .filter{ it.languages.contains(lang) }
       .maxByOrNull{ it.experience }
    } catch (ex:Exception) {
       null
    }
}
```

immutable features like `data classes` and `val's`

immutable collections

expression constructs

higher-order functions

Programming Styles

Imperative programming

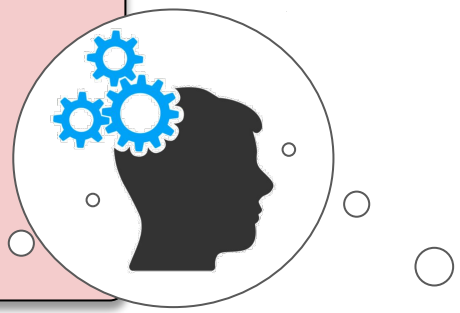
Imperative programming relies on *declaring variables* that are *mutated* along the way.

```
fun findBestDev(lang: String): Developer? {
    var devs:List<Developer>
    try {
        devs = client.getAll<Developer>()
    } catch (ex:Exception) {
        return null
    }
    var result = mutableListOf<Developer>()
    for(dev in devs) {
      if(dev.languages.contains(language))
        result.add(dev)
    }
    if(result.isEmpty()) {
        return null
    }
    result.sortBy { it.experience }
    return result[0]
}
```

Expression oriented programming

Expression oriented programming relies on thinking in *functions* where every *input* results in an *output*:

```
fun findBestDev(lang: String): Developer? =
  try {
    client.getAll<Developer>()
     .filter{ it.languages.contains(lang) }
     .maxByOrNull{ it.experience }
   } catch (ex:Exception) {
      null
   }
}
```



Thinking in terms of:

Mutating data, side-effects

Thinking in terms of:

Transforming data, Input/Output

Programming Styles

Imperative programming

Imperative programming relies on *declaring variables* that are *mutated* along the way.

```
fun findBestDev(lang: String): Developer? {
    var devs:List<Developer>
    try {
        devs = client.getAll<Developer>()
    } catch (ex:Exception) {
        return null
    }
    var result = mutableListOf<Developer>()
    for(dev in devs) {
      if(dev.languages.contains(language))
        result.add(dev)
    }
    if(result.isEmpty()) {
        return null
    }
    result.sortBy { it.experience }
    return result[0]
}
```

Expression oriented programming

Expression oriented programming relies on thinking in *functions* where every *input* results in an *output*:

```
fun findBestDev(lang: String): Developer? =
  try {
    client.getAll<Developer>()
     .filter{ it.languages.contains(lang) }
     .maxByOrNull{ it.experience }
  } catch (ex:Exception) {
     null
  }
}
```

Results in conciser, expressive, deterministic, better testable and clearly scoped code that is easier to reason about compared to the imperative style.

PICK

Expression Oriented Programming

SKIP

Imperative Programming

Transforming data over *Mutating data*

Input / Output over *Side-effects*

- `val`'s,
- immutable objects and collections
- `data` classes
- expression constructs
- (higher-order) functions
- functional collections

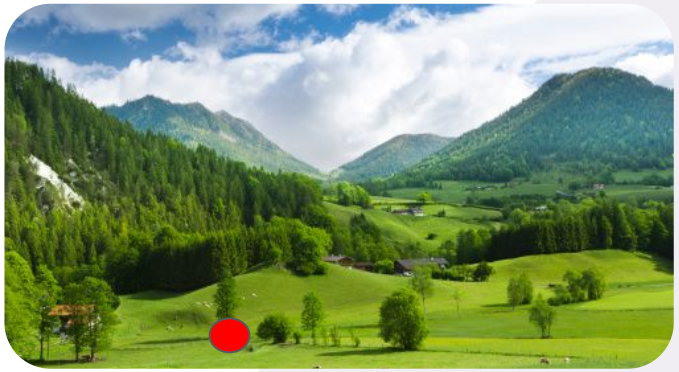
over

- `var`'s
- loops
- mutable objects
- mutable collections
- multiple `return`'s

Welcome to:

The swamps of imperative programming

The fruitful valley of **Expression Oriented Programming**



Comparing OO between Java >= 14 & Kotlin

```java
public record Person(String name, int age) {

    public Person(String name) {
        this(name, 0);
    }

    public String sayHi() {
        return "Hi";
    }

    public void plus(int y) {
      //no re-assignment to final age possible
    }

}
```

```kotlin
data class Person(val name: String,
                  var age: Int = 0) {

  fun sayHi() = "Hi"

  operator fun plus(y:Int):Unit {
     age += y
  }
}
```

Comparing OO between Java >= 14 & Kotlin

```java
public record Person(String name, int age) {

    public Person(String name) {
        this(name, 0);
    }

    public String sayHi() {
        return "Hi";
    }

    public void plus(int y) {
      //no re-assignment to final age possible
    }

}
```

```kotlin
data class Person(val name: String,
                  var age: Int = 0) {

  fun sayHi() = "Hi"

  operator fun plus(y:Int):Unit {
     age += y
  }
}
```

```java
final var jack = new Person("Jack", 42);
jack: Person[name=Jack, age=5]
```

```kotlin
val jack = Person("Jack", 42)
jack: Person(name=Jack, age=42)
```

Comparing OO between Java >= 14 & Kotlin

```java
public record Person(String name, int age) {

    public Person(String name) {
        this(name, 0);
    }

    public String sayHi() {
        return "Hi";
    }

    public void plus(int y) {
      //no re-assignment to final age possible
    }

}
```

```kotlin
data class Person(val name: String,
                  var age: Int = 0) {

  fun sayHi() = "Hi"

  operator fun plus(y:Int):Unit {
     age += y
  }
}
```

```java
final var jack = new Person("Jack", 42);
jack: Person[name=Jack, age=5]

jack.equals(new Person("Jack", 42));
res1: true
```

```kotlin
val jack = Person("Jack", 42)
jack: Person(name=Jack, age=42)

jack == Person("Jack", 42)
res1: true
```

Comparing OO between Java >= 14 & Kotlin

```java
public record Person(String name, int age) {

    public Person(String name) {
        this(name, 0);
    }

    public String sayHi() {
        return "Hi";
    }

    public void plus(int y) {
      //no re-assignment to final age possible
    }

}
```

```kotlin
data class Person(val name: String,
                  var age: Int = 0) {

  fun sayHi() = "Hi"

  operator fun plus(y:Int):Unit {
     age += y
  }
}
```

```java
final var jack = new Person("Jack", 42);
jack: Person[name=Jack, age=5]

jack.equals(new Person("Jack", 42));
res1: true

final var fred = new Person("Fred", jack.name);
//copy(...) does not exist :-(
```

```kotlin
val jack = Person("Jack", 42)
jack: Person(name=Jack, age=42)

jack == Person("Jack", 42)
res1: true

val fred = jack.copy(name = "Fred")
fred: Person(name=Fred, age=42)
```

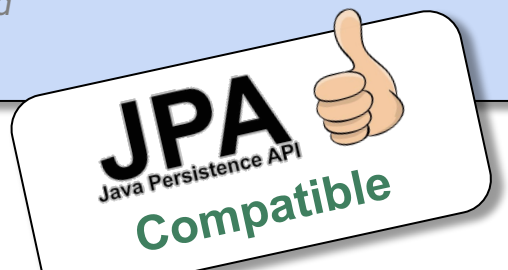
Comparing OO between Java >= 14 & Kotlin

```java
public record Person(String name, int age) {

    public Person(String name) {
        this(name, 0);
    }

    public String sayHi() {
        return "Hi";
    }

    public void plus(int y) {
      //no re-assignment to final age possible
    }

}
```

```kotlin
data class Person(val name: String,
                  var age: Int = 0) {

  fun sayHi() = "Hi"

  operator fun plus(y:Int):Unit {
     age += y
  }
}
```

```java
final var jack = new Person("Jack", 42);
jack: Person[name=Jack, age=5]

jack.equals(new Person("Jack", 42));
res1: true

final var fred = new Person("Fred", jack.name);
//copy(...) does not exist :-(

//destructuring either
```


```kotlin
val jack = Person("Jack", 42)
jack: Person(name=Jack, age=42)

jack == Person("Jack", 42)
res1: true

val fred = jack.copy(name = "Fred")
fred: Person(name=Fred, age=42)

val (name, age) = fred
name: "Fred"
age: 42
```



Functional Programming & DDD go hand-in-hand

…or how to put the compiler at work

Take1

```
data class Meetup(
   val title: String,
   val description: String,
   val date: Instant = Instant.now()
)
```

```
Meetup(
   "With GraalVM, the JVM now also ...",
   "JVM goes Native"
   )
```

Functional Programming & DDD go hand-in-hand

Can *named arguments* help?

Take1b

```
data class Meetup(
   val title: String,
   val description: String,
   val date: Instant = Instant.now()
)
```

```
Meetup(
  description = "With GraalVM, the JVM...",
  title = "JVM goes Native"
  )
```

Better, but still not safe…

Functional Programming & DDD go hand-in-hand

Value classes to the rescue:

Take2

```
@JvmInline
value class Title(val value: String)

@JvmInline
value class Description(val value: String)

data class Meetup(
   val title: Title,
   val description: Description,
   val date: Instant = Instant.now()
)
```

```
Meetup(
  Title("JVM goes Native"),
  Description("With GraalVM, the JVM...")
 )
```

Value classes are wrappers that only exist in the *source code,* at runtime they are gone (no overhead!)

That looks *descriptive*, *safe* and *efficient* to me

Functional Programming & DDD go hand-in-hand

Value classes to the rescue:

Take2

```
@JvmInline
value class Title(val value: String)

@JvmInline
value class Description(val value: String)

data class Meetup(
   val title: Title,
   val description: Description,
   val date: Instant = Instant.now()
)
```

```
Meetup(
  Title("JVM goes Native"),
  Description("With GraalVM, the JVM...")
 )
```

Value classes are wrappers that only exist in the *source code*, at runtime they are gone (no overhead!)

Ok, now we have two kinds of meetups: on-premise and online*, with different attributes. How can I model that?*

Functional Programming & DDD go hand-in-hand

Take3

nullable on-premise attributes

```
data class Meetup(
    val title: Title,
    val description: Description,
    val date: Instant = Instant.now(),
    val address: Address? = null,
    val rsvpRequired: Boolean? = null,
    val streamingUrl: URL? = null,
)
```

nullable online attributes

```
fun handleMeetup(m: Meetup) = when {
        m.address != null && m.rsvpRequired != null ->
            println("On premise: ${m.address},
                                rsvp:${m.rsvpRequired}")
        m.streamingUrl != null ->
            println("Online: ${m.streamingUrl}")
        else -> throw IllegalStateException("...")
    }
```

Wait a minute, this code is very error-prone!

Functional Programming & DDD go hand-in-hand

sealed classes create great domain models and put the compiler at work!

Take4

```
sealed interface Meetup {
    val title: Title
    val description: Description
    val date: Instant
}

data class OnPremiseMeetup(
  override val title: Title,
  override val description: Description,
  override val date: Instant,
  val address: Address,
  val rsvpRequired: Boolean
) : Meetup

data class OnlineMeetup(
  override val title: Title,
  override val description: Description,
  override val date: Instant,
  val streamingUrl: URL
) : Meetup
```

mandatory on-premise attributes

mandatory online attributes

```
fun handleMeetup(m: Meetup) = when(m) {
        is OnPremiseMeetup ->
            println("On premise: ${m.address},
                              rsvp:${m.rsvpRequired}")
        is OnlineMeetup ->
                    println("Online: ${m.streamingUrl}")
    }
}
```

No more `else` case required, because the list is *exhaustive*.

Hey, now I can discard all those unit-tests that checked all the *null combinations*!

Functional Programming & DDD go hand-in-hand

sealed classes create great domain models and put the compiler at work!

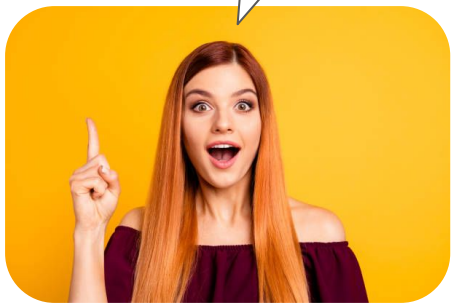
Take5 🙌

```
sealed interface Meetup {
   val title: Title
   val description: Description
   val date: Instant

   data class OnPremiseMeetup(
     override val title: Title,
     override val description: Description,
     override val date: Instant,
     val address: Address,
     val rsvpRequired: Boolean
   ) : Meetup

   data class OnlineMeetup(
     override val title: Title,
     override val description: Description,
     override val date: Instant,
     val streamingUrl: URL
   ) : Meetup

   data class HackatonMeetup(
     override val title: Title,
     override val description: Description,
     override val date: Instant,
     val sourcesUrl: URL
   ) : Meetup
}
```

```
fun handleMeetup(m: Meetup) = when(m) {

       is OnPremiseMeetup ->
           println("On premise: ${m.address},
                           rsvp:${m.rsvpRequired}")

       is OnlineMeetup ->
                  println("Online: ${m.streamingUrl}")
   }
}
```

…adding a new type, results in a compilation error

…which is exactly what I want!



But how to deal with mutable API's?

```
fun devsToFile(fileName:String):Stats {
    val client = RestClient()
    client.username = "xyz"
    client.secret = System.getenv("pwd")
    client.url = "https://..."
    client.initAccessToken()
    try{
        val file = File(fileName)
        file.createNewFile()
        file.setWritable(true)
        val devs = client.getAll<Developer>()
        require(devs.isNotEmpty()){
            val msg = "No devs found"
            LOG.error(msg)
            msg
        }
        devs.forEach { file.appendText(it.toCSV()) }
        return Stats(devs.size, file.length())
    }
    finally {
        client.close()
    }
}
```

Something with a `RestClient`

Something with a `File`

Something with Developers

Again something with `File` and Developers

Again something with `RestClient`

Looks a bit messy to me, but I can't help it because `RestClient` *and* `File` *are mutable by nature…*

With clearly scoped code!

```
fun devsToFile(fileName:String):Result {
  val client = RestClient()
  client.username = "reader"
  client.secret = System.getenv("pwd")
  client.url = "https://..."
  client.initAccessToken()
  try{
    val file = File(fileName)
    file.createNewFile()
    file.setWritable(true)
    val devs = client.getAll<Developer>()
    require(devs.isNotEmpty()){
       val msg = "No devs found"
       LOG.error(msg)
       msg
    }
    devs.forEach{file.appendText(it.toCSV())}
    return Result(devs.size,file.length())
  }
  finally {
    client.close()
  }
}
```

Client Block

Developer Block

File Block

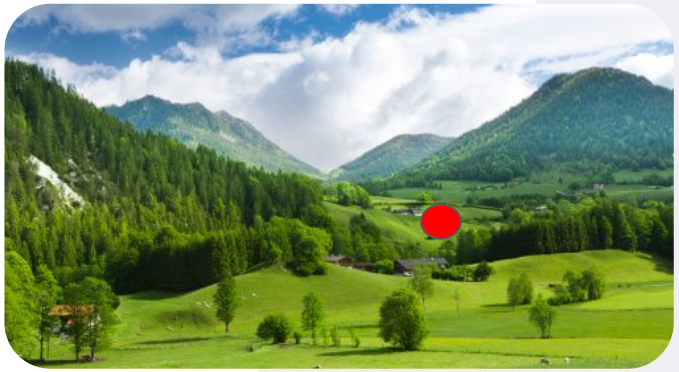
```
fun devsToFile(fileName: String): Result =
  RestClient().apply {
     username = "reader"
     secret = System.getenv("pwd")
     url = "https://..."
     initAccessToken()
  }.use { client ->
     client.getAll<Developer>().let { devs ->
        require(devs.isNotEmpty()) {
          "No devs found".also { LOG.error(it) }
        }
        File(fileName).run {
          createNewFile()
          setWritable(true)
          devs.forEach{appendText(it.toCSV())}
          Result(devs.size, length())
        }
     }
  }
```

PICK		SKIP
Scope Methods		***Frequent Variables & Assignments***
Group related operations on an object together	over	***Free floating variables***
Restrict variable access as much as possible		
Isolate mutable code	over	***Scattered side-effects***
'Tag' side-effects (mainly `apply`, `also`***)***		

Welcome to:

The swamps of imperative programming

The fruitful valley of **Expression Oriented Programming with rich, safe and expressive domain models**



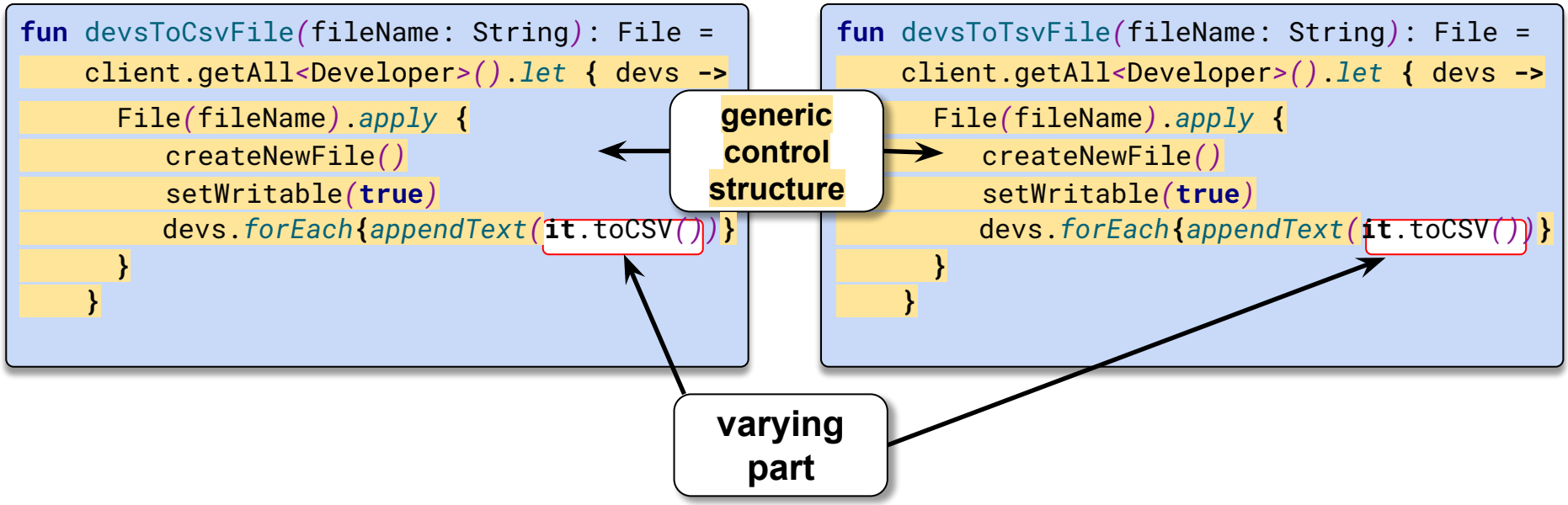
How to re-use Control Structures?

```
fun devsToCsvFile(fileName: String): File =
    client.getAll<Developer>().let { devs ->
      File(fileName).apply {
         createNewFile()
         setWritable(true)
         devs.forEach{appendText(it.toCSV())}
      }
    }
```

```
fun devsToTsvFile(fileName: String): File =
    client.getAll<Developer>().let { devs ->
      File(fileName).apply {
         createNewFile()
         setWritable(true)
         devs.forEach{appendText(it.toTSV())}
      }
    }
```

So much duplication only because I want a different format?

How to re-use Control Structures?



…with Higher-Order Functions

Turn the **varying part** into a function

```
fun devsToFile(fileName: String, toLine:(Developer) -> String): File =
    client.getAll<Developer>().let { devs ->
      File(fileName).apply {
          createNewFile()
          setWritable(true)
          devs.forEach{appendText(toLine(it))}
      }
    }
```

Let the **generic control structure** use it at the designated point

```
devsToFile("devs.csv"){it.toCSV()}
```

```
devsToFile("devs.tsv"){it.toTSV()}
```

```
devsToFile("devs.???"){it.to???()}
```

…with Higher-Order Functions

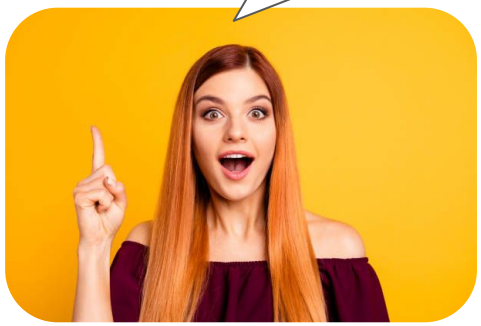
```
fun devsToFile(fileName: String, toLine:(Developer) -> String): Stats =
  RestClient().apply {
     username = "reader"
     secret = System.getenv("pwd")
     url = "https://..."
     initAccessToken()
  }.use { client ->
    client.getAll<Developer>().let { devs ->
       require(devs.isNotEmpty()) {
         "No devs found".also { LOG.error(it) }
        }

       File(fileName).run {
         createNewFile()
         setWritable(true)
         devs.forEach{appendText(toLine(it)}
         Stats(devs.size, length())
       }
    }
 }
```

Cool, but what if I need to save the `Developers` *not only to* `File` *but also to other storage types like s3, db etc.???*

```
devsToFile("devs.csv"){it.toCSV()}
```

```
devsToFile("devs.tsv"){it.toTSV()}
```

```
devsToFile("devs.???"){it.to???()}
```



Generic Control Structure vs Varying Part: you define the split

```
fun <T> devsTo(process:(List<Developer>) -> T): T =
  RestClient().apply {
     username = "reader"
     secret = System.getenv("pwd")
     url = "https://..."
     initAccessToken()
  }.use { client ->
    client.getAll<Developer>().let { devs ->
       require(devs.isNotEmpty()) {
         "No devs found".also { LOG.error(it) }
       }
       process(devs)
  }
```

```
devsTo{ devs ->
  File("devs.csv").run {
       createNewFile()
       setWritable(true)
       devs.forEach{
          appendText(it.toCSV())
       }
       Stats(devs.size, length())
  }
}
```

```
devsTo{ devs ->
  devsRepository.saveAll(devs)
}
```

Higher-Order Functions also shine in tests

```
@Test
fun `should fetch all developers`() {
   val saved = saveAll((1..10).map { DevSample.create(it) })
   try {
      val fetchedDevs = client.getAll<Developer>()
      fetchedDevs.size shouldBe devs.size
   } finally {
       deleteAll(saved)
   }
}

@Test
fun `should fetch developers by id`() {
   val saved = saveAll((1..1).map { DevSample.create(it) })
   try {
      val fetchedDev = client.get<Developer>(id = 1)
      devs shouldContain fetchedDev
   } finally {
       deleteAll(saved)
   }
}
```

Higher-Order Functions also shine in tests

```
@Test
fun `should fetch all developers`() {
   val saved = saveAll((1..10).map { DevSample.create(it) })
   try {
      val fetchedDevs = client.getAll<Developer>()
      fetchedDevs.size shouldBe devs.size
   } finally {
      deleteAll(saved)
   }
}

@Test
fun `should fetch developers by id`() {
   val saved = saveAll((1..1).map { DevSample.create(it) })
   try {
      val fetchedDev = client.get<Developer>(id = 1)
      devs shouldContain fetchedDev
   } finally {
      deleteAll(saved)
   }
}
```

Higher-Order Functions also shine in tests

```
fun <T> Repository<Developer>.doWith(count: Int, doTest: (List<Developer>) -> Unit) {
    val saved = saveAll((1..count).map { DevSample.create(it) })
    try {
        doTest(saved)
    } finally {
        deleteAll(saved)
    }
}
```

```
@Test
fun `should fetch all developers`() = devRepository.doWith(10) { devs ->
        val fetchedDevs = client.getAll<Developer>()
        fetchedDevs.size shouldBe devs.size
        ...
}


@Test
fun `should fetch one developers`() = devRepository.doWith(1) { devs ->
        val fetchedDev = client.get<Developer>(id = 1)
        devs shouldContain fetchedDev
        ...
}
```

Higher-Order Functions also shine in tests

```
@Test
fun `should fetch all developers`() {
    val saved = saveAll((1..10).map { DevSample.create(it) })
    try {
        val fetchedDevs = client.getAll<Developer>()
        fetchedDevs.size shouldBe devs.size
    } finally {
        deleteAll(saved)
    }
}

@Test
fun `should fetch developers by id`() {
    val saved = saveAll((1..1).map { DevSample.create(it) })
    try {
        val fetchedDev = client.get<Developer>(id = 1)
        devs shouldContain fetchedDev
    } finally {
        deleteAll(saved)
    }
}
```

PICK

Higher-Order Functions

Leverage your code by dividing a control structure in a generic part that executes the specific via a function

Higher-Order functions for collections

over

over

SKIP

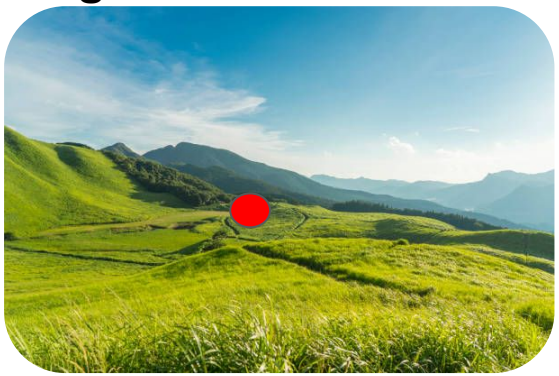
Code-Duplication

Copy-Paste

for loops

Welcome to:

The abundant heights of
Higher-Order Functions



The swamps of imperative programming

The fruitful valley of Expression Oriented Programming

Composable Constructs

The Higher-Order Functions we created previously lack an abstraction so they can be *combined* with other *similar building blocks* to create *new valuable results*.

In other words: they *are not composable*

```
fun findBestDev(lang: String): Developer? =
    getAll<Developer>()
     .filter{ it.languages.contains(lang) }
     .maxBy{ it.experience }
}
```

Apparently, the Collection *abstraction* allows us to manipulate / transform its data in very flexible and powerful ways

So, the ‘characteristics’ of the Collection *Container* in combination with *Higher-Order-Functions* apparently makes this possible.

-> What are the fundamental aspects of these characteristics?

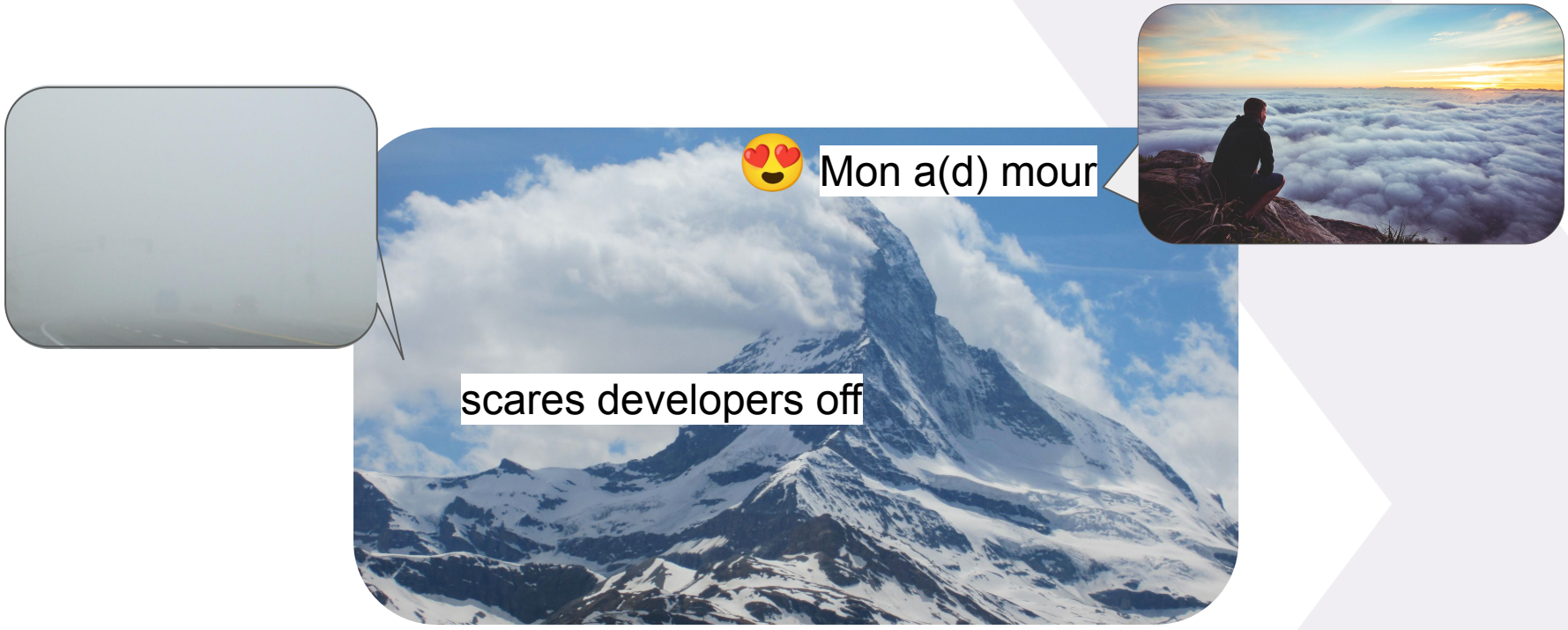
-> Can we leverage them beyond Collections?

Welcome to Category Theory

Welcome to the **danger zone**

A Monad is just a monoid in the category of endofunctors

The academic world managed to describe a powerful programming paradigm in such a difficult and inaccessible way that it either - most of the time:



In both cases we fail to unlock promising opportunities.

So, highest time for a demystification so that we all can profit from this powerful paradigm - ***where best applicable***.

We all know Monads - and use them daily!

Monads are all about **Containers**:

Let's look at the 'Container' we all know: **Collections**!

From a *Container Perspective*: what kind of operations makes a Collection useful?

We all know Monads - and use them daily!

1. We can *combine* two Collections, resulting in a new Collection:

```
listOf(1, 2) + listOf(3) == listOf(1, 2, 3)
```

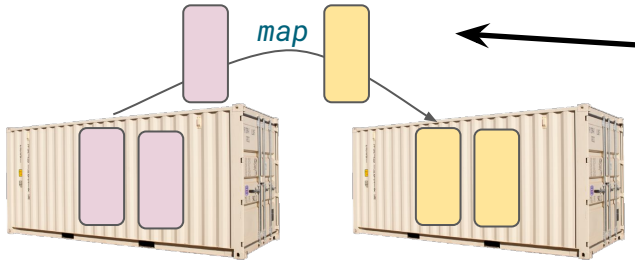
When we can combine instances of a *type* that results in a new instance of the *same type* containing the elements of the initial ones, we are talking about a: **Monoid**

```
interface Monoid<T> {
    fun empty():T
    fun combine(a:T, b:T):T
}
```

We all know Monads - and use them daily!

2. We can *transform* the elements of the Collection

```
listOf(1, 2).map{it.toString()} == listOf("1", "2")
```



When a type offers a method to map over, it's called a **Functor**

```
interface Functor<A> {
   fun <B> map(transform:(A) -> B):Functor<B>
}
```

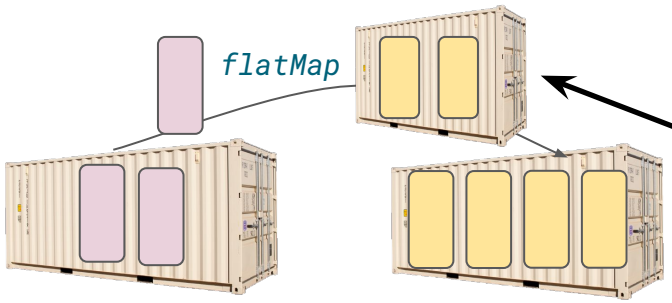
We all know Monads - and use them daily!

3. If our elements are Collections too, we can *transform them without additional nesting*

```
data class Developer(val name: String, val languages: List<String>)
```

```
//map results in nesting
listOf(dev1,dev2).map{it.languages} == listOf(listOf("Kotlin","Scala"),listOf("Python"))
```

```
//flatMap maps AND flattens
listOf(dev1,dev2).flatMap{it.languages} == listOf("Kotlin","Scala","Python")
```



If - besides combining (*Monoid)*, mapping (Functor) - I can convert the elements of a *Container* that are *Containers too and have no additional nesting*, it's a **Monad**

```
class Monad<T>(val value:T) {
    fun empty(): Monad<T> = …
    operator fun plus(other: Monad<T>): Monad<T> = …
    fun <O> map(transform: (T) -> O): Monad<O> = …
    fun <O> flatMap(transform:(O) -> Monad<T>):Monad<O> = …
}
```

Monad (Composable)

Monoid (Combinable)

Functor (Mappable)

Not only Collections are Monads - there are many more you know

```
class Monad<T>(val value:T) {

 fun empty(): Monad<T> = …

 fun <O> map(m:(T)->O): Monad<O> = …

 fun <O> flatMap(fm:(O)->Monad<T>):Monad<O> = …

}
```

```
class List<T>(...)

emptyList()

listOf(dev1,dev2).map{it.name}
     == listOf("Jack", "Joe")

listOf(dev1,dev2).flatMap{it.languages}
     == listOf("Kotlin","Scala","Python")
```

Not only Collections are Monads - there are many more you know

```
class Monad<T>(val value:T) {

 fun empty(): Monad<T> = …

 fun <O> map(m:(T)->O): Monad<O> = …

 fun <O> flatMap(fm:(O)->Monad<T>):Monad<O> = …


}
```

```
class Optional<T>(...)

Optional.empty()

Optional.of(dev1).map{it.name}
     == Optional("Jack")

Optional.of(dev1).flatMap{
  it.languages.firstOrNull()?.let{
     Optional.of(it)} ?: Optional.empty()
  }   == Optional.of("Kotlin")
```

Not only Collections are Monads - there are many more you know

Swap Mono out for every other reactive abstraction like: `CompletableFuture` (standard library), `Observable` (RxJava), `Uni` (Quarkus)

```
class Monad<T>(val value:T) {

 fun empty(): Monad<T> = …

 fun <O> map(m:(T)->O): Monad<O> = …

 fun <O> flatMap(fm:(O)->Monad<T>):Monad<O> = …


}
```

```
class Mono<T>(...)

Mono.empty()

getDeveloperByName(dev1.name).map{it.name}
      == Mono.just("Jack")

getDeveloperByName(dev1.name).flatMap{
  selectBest(it.languages)
}     == Mono.just("Kotlin")
```


```
fun getDeveloperByName(name:String):Mono<String> =
      //some remote API/DB call

fun selectBest(languages:List<String>):Mono<String> =
      //some remote API call
```

So, what are these Monads/Containers good for?

What do all these abstractions and confusing terminology bring us?

- The confusing terminology not much - unless you are a scientist ;-)
- The abstraction, however, does have value. What we get is a special '**ability**':

How Composability does NOT look like

```
class RestClient {
      fun <T> getAll():List<T> = …


}
```

```
fun <T> devsTo(fn:(List<Developer>) -> T): T =
    client.getAll<Developer>().let { devs ->
       require(devs.isNotEmpty()) {
         "No devs found".also { LOG.error(it) }
        }
        fn(devs)
 }
```

How Composability does NOT look like

Let's use a non-blocking, thus more resource efficient way to fetch data via our RestClient by means of *Callbacks*

```
class RestClient {

      fun <T> getAll():List<T> = …
      fun <T> getAllAsync(success:(List<T>) -> Unit, failure:(Throwable) -> Unit):Unit = …

}
```

```
fun <T> devsTo(fn:(List<Developer>) -> T): T =
    client.getAll<Developer>().let { devs ->
       require(devs.isNotEmpty()) {
         "No devs found".also { LOG.error(it) }
       }
       fn(devs)
 }
```

```
fun <T> devsTo(fn:(List<Developer>) -> Unit): Unit =
   client.getAllAsync<Developer>(
     success = { devs ->
       require(devs.isNotEmpty()) {
           "No devs found".also { LOG.error(it) }
       }
       fn(devs)
     },
     failure = {ex ->
        LOG.error( "Error fetching devs",ex)
     })
```

Phu, how am I going to test this???

How Composability *DOES* look like

Instead of Callbacks, let's use the 'Reactive Monad': `Mono<T>` as return type

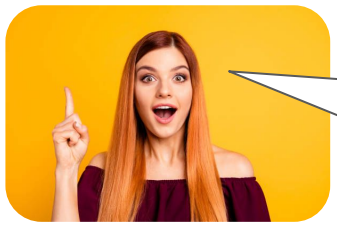
```
class RestClient {
      fun <T> getAll():List<T> = …
      fun <T> getAllAsync(success:(List<T>) -> Unit, failure:(Throwable) -> Unit):Unit = …
      fun <T> getAllReactive():Mono<List<T>> = …
}
```

```
fun <T> devsTo(fn:(List<Developer>) -> Unit):Unit =
   client.getAllAsync<Developer>(
     success = { devs ->
       require(devs.isNotEmpty()) {
           "No devs found".also { LOG.error(it) }
       }
       fn(devs)
     },
     failure = {ex ->
        LOG.error( "Error fetching devs",ex)
     })
```

```
fun <T> devsTo(fn:(List<Developer>) -> T):Mono<T> =
  client.getAllReactive<Developer>().flatMap{devs ->
    if(devs.isEmpty()) Mono.error{
     IllegalArgumentException("No devs found".also {
       LOG.error(it)})
    } else Mono.just(fn(it))
  }
```

```
devsTo{devs -> println(devs)}
```

```
devsTo{devs -> devs}
      .map{...}
      .onErrorMap{...}
      .onErrorContinue{...}
```



I get it: *when I let methods return Monads they become composable*

So, now let's go composing 😀😀😀

How about composable error handling?

```
fun mostPopularLanguageOf(name:String):Language {
    val dev = try {
        client.getDevByName(name)
    } catch (ex: IOException) {
        throw ApplicationException("Oeps", ex)
    }
    return try {
        client.getMostPopular(dev.languages)
    } catch (ex: Exception) {
       Language("Kotlin")
    }
}
```

Mmh, how could I make this `try - catch` more 'composable'?

Welcome to functional Error Handling

You want composability? You need a Monad!

Which is already part of the standard library:


```
class Result<T>(...){
   fun getOrNull(): T?
   fun exceptionOrNull():Throwable?
   fun map(convert:(value: T)->R):Result<R>
   fun flatMap(convert:(value:T)->Result<R>):Result<R>
}
```

```
fun mostPopularLanguageOf(name:String):Language{
   val dev = try {
       client.getDevByName(name)
   } catch (ex: IOException) {
       throw ApplicationException("Oeps", ex)
   }
   return try {
       client.getMostPopular(dev.languages)
   } catch (ex: Exception) {
      Language("Kotlin")
   }
}
```

```
fun mostPopularLanguageOf(name:String):Language {
   runCatching { client.getDevByName(name) }
    .onFailure { throw ApplicationException("Oeps",it)}
    .map { client.selectBest(it.languages) }
    .getOrElse { Language("Kotlin") }
}
```

Looks good, BUT: how do I know that I have to catch an Exception in the first place? It's not in the method signature…

Give me my ~~Un~~checked Exceptions back!

Convert every return type `T` into a `Result<T>` to make it very *explicit* that this method can also return an `Exception`

```
fun getDevByName(name:String):Developer
fun selectBest(langs:List<String>):Language
```

```
fun getDevByName(name:String):Result<Developer>
fun selectBest(langs:List<Language>):Result<Language>
```

```
fun bestLanguageOf(name:String):Result<Language> =
   client.getDevByName(name)
     .recoverCatching{throw ApplicationException("Oeps", it) }
     .flatMap{dev-> client.selectBest(dev.languages)
                          .mapCatching { Language("Kotlin") }
           }
```

A larger example

```
fun getDevByName(name:String):Developer
fun selectBest(langs:List<Language>):Language
fun getStatsFor(lang:Language):LanguageStats
```

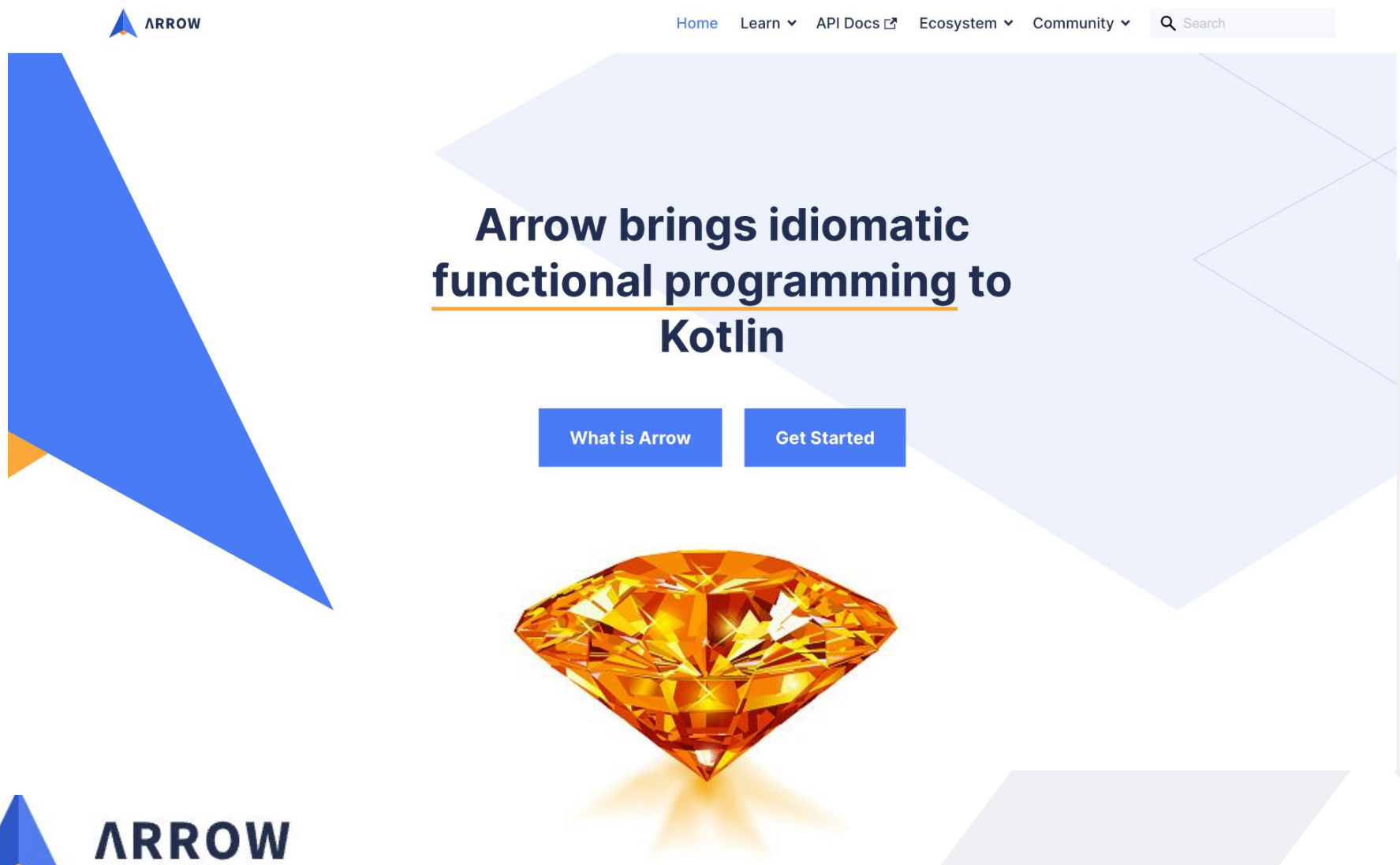
```
fun getDevByName(name:String):Result<Developer>
fun selectBest(langs:List<Language>):Result<Language>
fun getStatsFor(lang:Language):Result<LanguageStats>
```

```
fun statsOfBestLanguageOf(name:String):Result<LanguageStats> =
    client.getDevByName(name).flatMap { dev ->
        client.selectBest(dev.languages).flatMap { language ->
            client.getStatsFor(language)
        }
    }
```

Though these Monads compose nicely, I don't know if I - or my colleagues - like all this `flatMap`-ing.

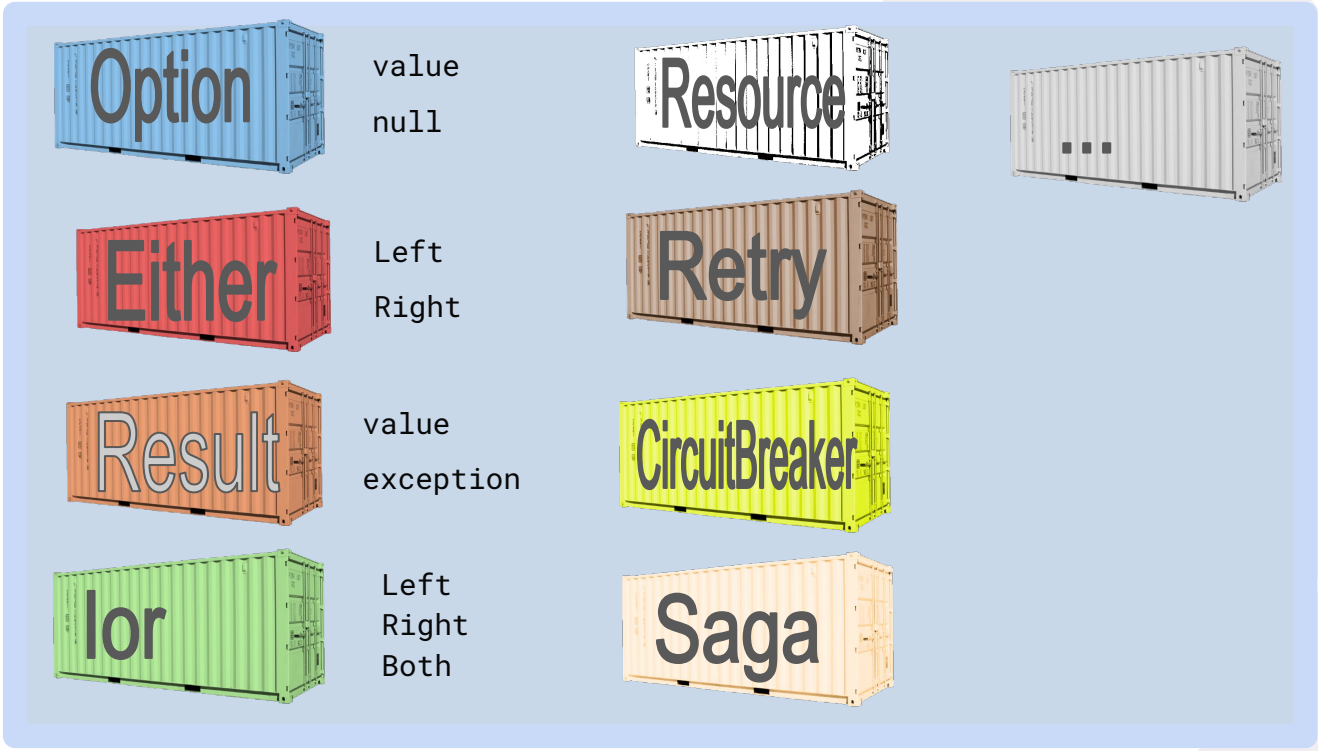
Is there no better way?

Welcome to Arrow



ARROW

Some Fancy Arrow Monads



ARROW

Arrow has a nice solution to avoid flatMap

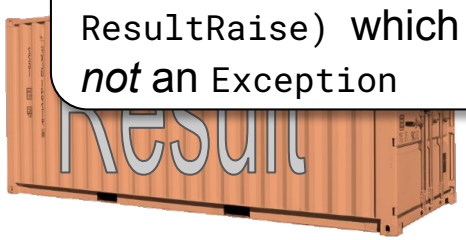
Which are: **Monad Comprehensions**

conceptually ported from functional languages like Scala and Haskell

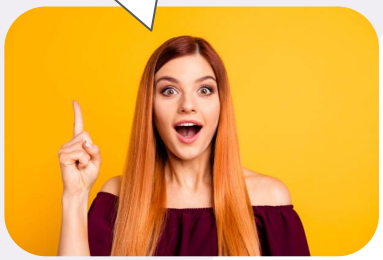
```
fun statsOfBestLanguageOf(name:String):Result<LanguageStats>=
   client.getDevByName(dev1.name).flatMap { dev ->
       client.selectBest(dev.languages).flatMap { language ->
           client.getStatsFor(language)
       }
   }
```

```
fun statsOfBestLanguageOf(name:String):Result<LanguageStats>=
    result { this:ResultRaise
      val dev = client.getDevByName(name).bind()
      val language = client.selectBest(dev.languages).bind()
      val stats = client.getStatsFor(language).bind()
      stats
    }
```

With the Monad Comprehension `result {...}`, `bind()` becomes available on a `Result<T>` (via `ResultRaise`) which returns it's value *only if it is not* an `Exception`



Even though I find this `bind()` a bit weird, I like it much better than these nested `flatMap's`



Arrow has a nice solution to avoid flatMap

Which are: **Monad Comprehensions**

conceptually ported from functional languages like Scala and Haskell

```
fun statsOfBestLanguageOf(name:String):Result<LanguageStats>=
   client.getDevByName(dev1.name).flatMap { dev ->
       client.selectBest(dev.languages).flatMap { language ->
           client.getStatsFor(language)
       }
   }
```

```
context(ResultRaise)
fun statsOfBestLanguageOf(name:String):LanguageStats {
    val dev = client.getDevByName(name).bind()
    val language = client.selectBest(dev.languages).bind()
    val stats = client.getStatsFor(language).bind()
    return stats
}
```

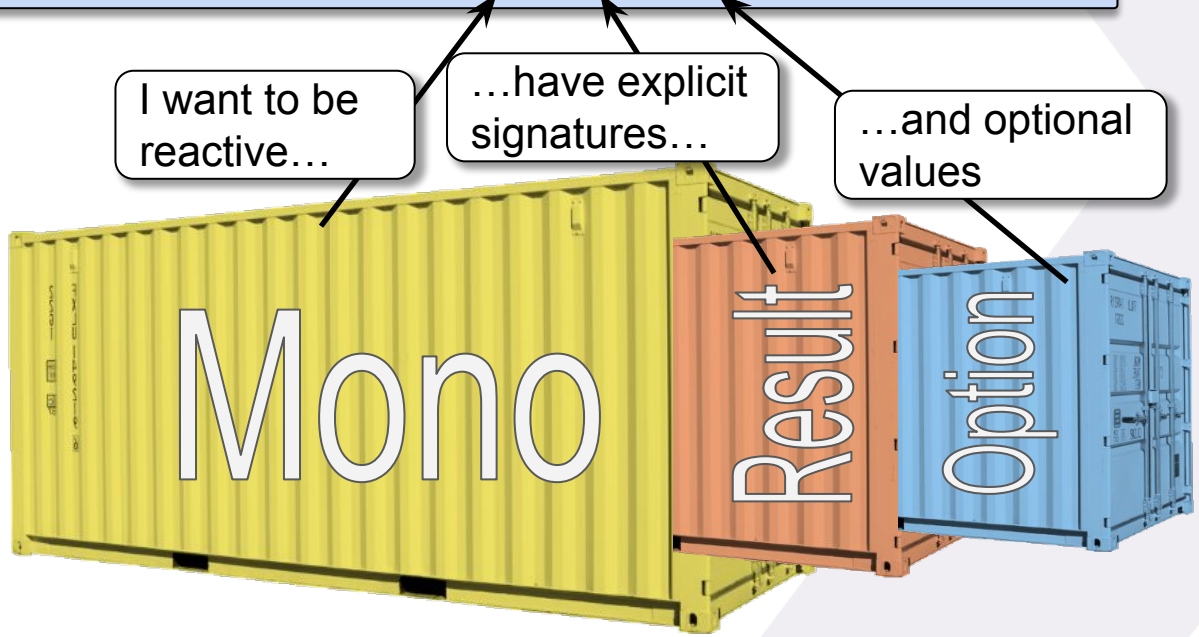
Let's Compose the Composables!

So far we only had to deal with *a single* Monad/Container.

But: the real world demands more:

```
fun getDevByName(name:String):Result<Developer>
fun selectBest(langs:List<Language>):Result<Language>
fun getStatsFor(lang:Language):Result<LanguageStats>
```

```
fun getDevByName(name:String):Mono<Result<Option<Developer>>>
fun selectBest(langs:List<Language>):Mono<Result<Option<Language>>>
fun getStatsFor(lang:Language):Mono<Result<Option<LanguageStats>>>
```



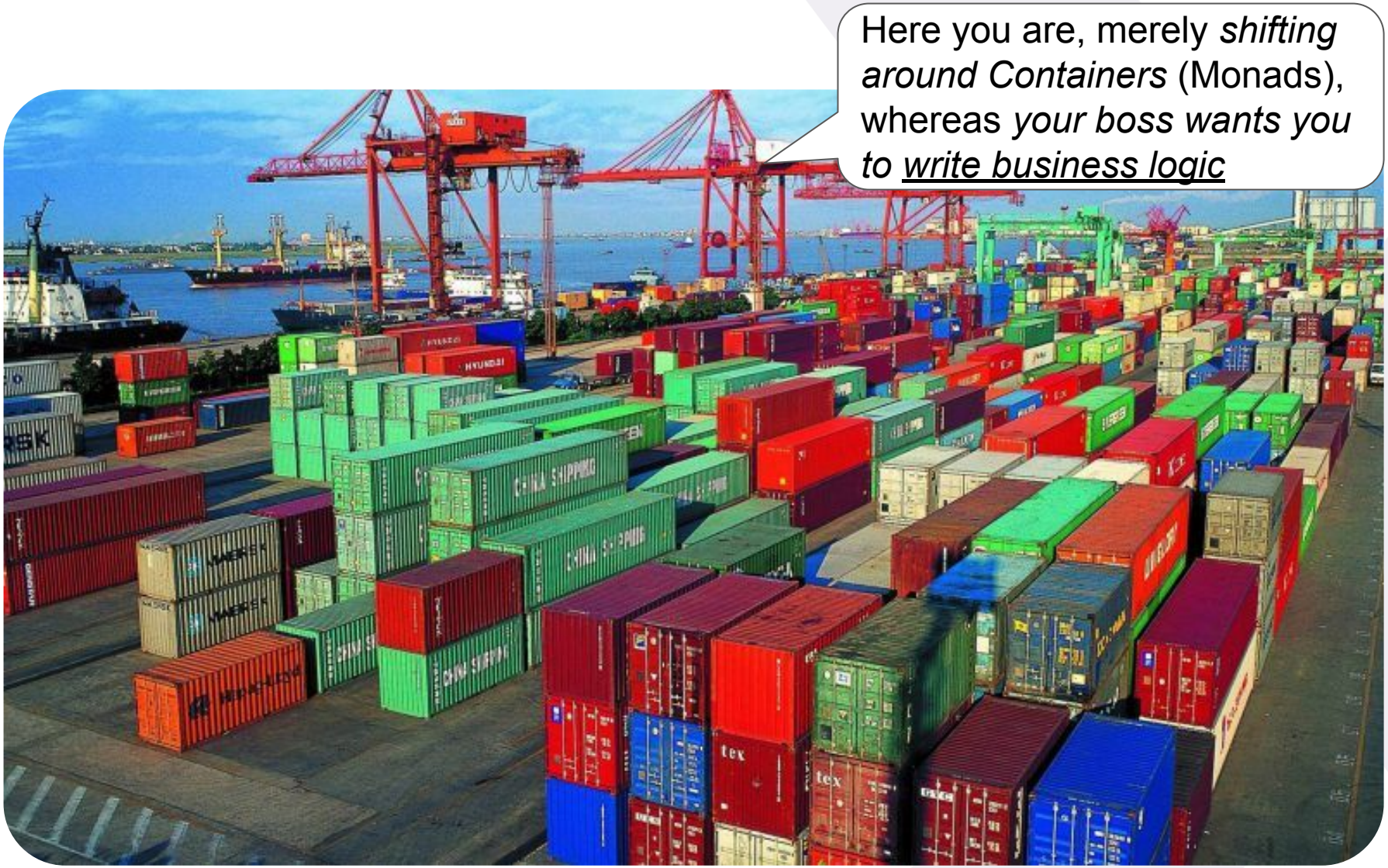
Let’s Compose the Composables???

```
fun statsOfBestLanguageOf(name:String):Mono<Result<Option<LanguageStats>>> =
   client.getDevByName(name).flatMap { devResOpt ->
         devResOpt.map {
             it.map { dev ->
                 client.getMostPopular(dev.languages).flatMap { langResOpt ->
                     langResOpt.map {
                         it.map { language ->
                             client.getStatsForLanguage(language)
                         }.getOrElse { Mono.just(Result.success(None)) }
                     }.getOrElse { Mono.just(langResOpt.map{None}) }
                 }
             }.getOrElse { Mono.just(Result.success(None)) }
         }.getOrElse { Mono.just(devResOpt.map { None }) }
}
```

OMG *- and we haven’t implemented any serious business logic yet…*

Congratulations

You just managed to convert your code-base into a: **Container terminal**



Let’s Compose the Composables???

```
fun statsOfBestLanguageOf(name:String):Mono<Result<Option<LanguageStats>>> =
   client.getDevByName(name).flatMap { devResOpt ->
         devResOpt.map {
             it.map { dev ->
                 client.selectBest(dev.languages).flatMap { langResOpt ->
                     langResOpt.map {
                         it.map { language ->
                             client.getStatsForLanguage(language)
                         }.getOrElse { Mono.just(Result.success(None)) }
                     }.getOrElse { Mono.just(langResOpt.map{None}) }
                 }
             }.getOrElse { Mono.just(Result.success(None)) }
         }.getOrElse { Mono.just(devResOpt.map { None }) }
}
```

But wait, don’t we have these *Monad Comprehensions* to make this code more readable?

Nope, sorry…

Monad Comprehensions - in Kotlin - only work for a *single* Monad

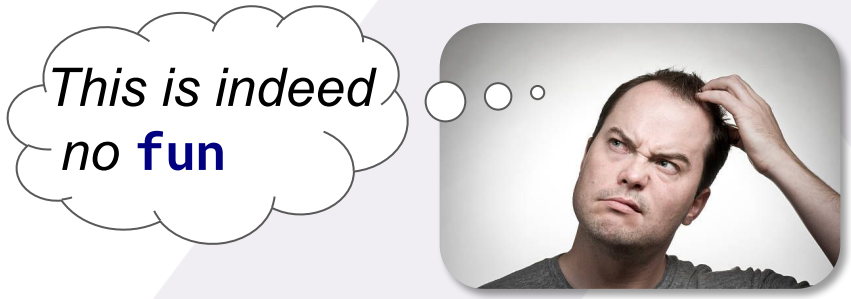
Nope, sorry…

Monad Comprehensions - in Kotlin - only work for a *single* Monad

Monad Transformers to the rescue?

- *They only work for two levels of Monads*
- *They cannot be used in Monad Comprehensions since Kotlin misses languages features for them to work (Higher Kinded Types)*

```
fun <A, B> Mono<Result<A>>.mapT(f:(A) -> B):Mono<Result<B>> = ...
fun <A, B> Mono<Result<A>>.flatMapT(f:(A) -> Mono<Result<A>>):Mono<Result<B>> = ...
fun <A, B> Mono<Result<A>>.flatMapTOuter(f:(A) -> Mono<A>):Mono<Result<B>> = ...
fun <A, B> Mono<Result<A>>.flatMapTInner(f:(A) -> Result<A>):Mono<Result<B>> = ...
```



So, are these Monads no good at all?

As long as you can keep your signature to a *single* Monad you're fine:

```
fun getDevByName(name:String):Mono<Result<Option<Developer>>>
```

```
suspend fun getDevByName(name:String):Result<Option<Developer>>
```

```
suspend fun getDevByName(name:String):Result<Developer?>
```

But I *need three*, so how can I reduce it to a *single* one?

Nullability

```
class Booking(val destination:Destination? = null)
class Destination(val hotel:Hotel? = null)
class Hotel(val name:String, val stars:Int? = null)
```

Kotlin

Since nullability is part of the *type system* no wrapper is needed: The required type or `null` can be used.

```
val booking:Booking? = Booking(Destination(Hotel("Sunset Paradise", 5)))
val stars = "*".repeat(booking?.destination?.hotel?.stars ?: 0) //-> "*****"

if(booking != null) {
    println(booking.destination)
}

booking?.let{
    println(it.destination)
}
```

To safely access *nullable types* the `?` can be used with `?:` for the alternative case.

After checking for *not null* a type is ‘smart casted’ to its non-null type: here from *Booking?* to *Booking*

Alternatively, the *?.let{...}* syntax can be used.

Nullability vs Optional

```kotlin
val booking:Booking? = Booking(Destination(Hotel("Sunset Paradise", 5)))
val stars = "*".repeat(booking?.destination?.hotel?.stars ?: 0) //-> "*****"

if(booking != null) {
   println(booking.destination)
}
```

Kotlin

```java
final Optional<Booking> booking = Optional.of(new Booking(
    Optional.of(new Destination(Optional.of(
        new Hotel("Sunset Paradise", 5))))));

var stars = "*".repeat(booking.flatMap(Booking::getDestination)
                  .flatMap(Destination::getHotel)
                  .map(Hotel::getStars).orElse(0)); //-> "*****"

if(booking.isPresent()) {
   println(booking.get().getDestination());
}
```

Java

Nullability vs Optional

```kotlin
val booking:Booking = null //Compilation error 😃
val booking:Booking? = null

booking.destination //Compilation error 😃
booking?.destination
```

Nullability is enforced by the compiler in Kotlin

Kotlin

```java
Booking booking = null; //Fine...
Optional<Booking> booking = null; //Fine...

booking.map(Destination::destination); //Runtime error 😱

booking.destination; //Runtime error 😱
```

Java

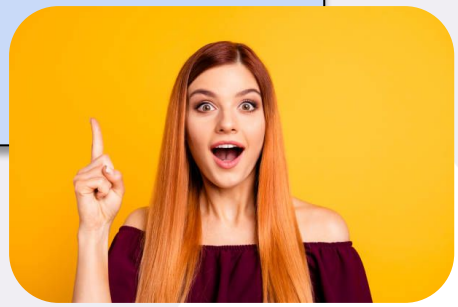
So, are these Monads no good at all?

As long as you can keep your signature to a *single* Monad you are fine

```
fun statsOfBestLanguageOf(name:String):Mono<Result<Option<LanguageStats>>> =
   client.getDevByName(name).flatMap { devResOpt ->
         devResOpt.map {
             it.map { dev ->
                 client.selectBest(dev.languages).flatMap { langResOpt ->
                     langResOpt.map {
                         it.map { language ->
                             client.getStatsFor(language)
                         }.getOrElse { Mono.just(Result.success(None)) }
                     }.getOrElse { Mono.just(langResOpt.map{None}) }
                 }
             }.getOrElse { Mono.just(Result.success(None)) }
         }.getOrElse { Mono.just(devResOpt.map { None }) }
}
```

Except for the `bind()` that's just 'normal' Kotlin code

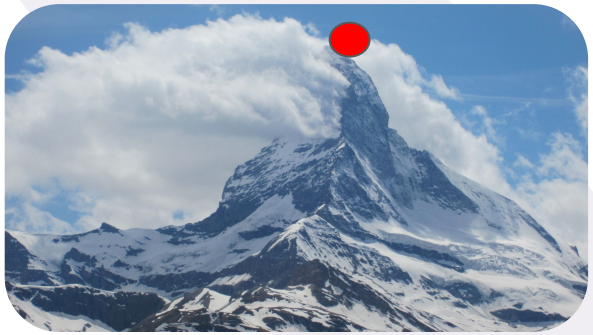
```
suspend fun statsOfBestLanguageOf(name:String):Result<LanguageStats?> = result {
 client.getDevByName(name).bind()?.let { dev ->
   client.selectBest(dev.languages).bind()?.let { lang ->
      client.getStatsFor(lang).bind() }
   }
```



PICK		SKIP
A single Monad		***Nested Monads***
Coroutines `(suspend)`	over	***Reactive Monads:*** `Mono/Uni/Observable/`***etc.***
Nullable Types `(?)`	over	`Option`
Arrow's Monad Comprehension `effect { … }`	over	`flatMap`

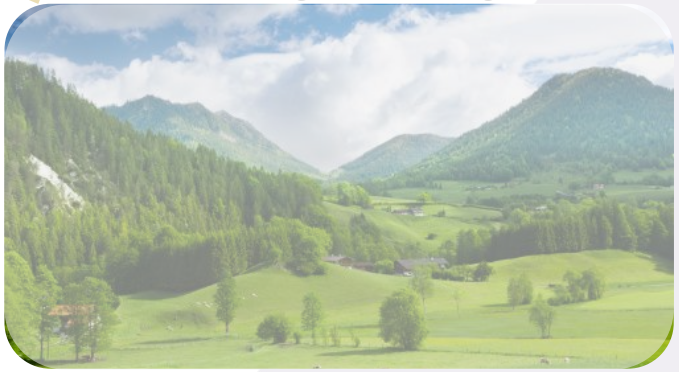
Welcome to:

The lonely peak of
Monadic Mist



The abundant heights of
Higher-Order Functions

The fruitful valley of
Expression Oriented
Programming



The swamps of imperative
programming

Which amount of Monads is ok? …and which too much?

…which brings us to a wider question:

what is ‘good’ code?

Generally spoken ‘good’ code:

…reflects the domain of the ‘bounded context’ / the business

Code of an Investment Bank

…has a minimum of non-domain abstractions.
These abstractions should serve a specific, useful purpose.

…should be understandable by a domain expert.

In other words: If your boss understands your code…

…everything is fine

Code that looks like a container terminal does not meet this definition

Unless your company *IS* the container terminal 😉

Why use Monads through your call chain…

Exception Handler

```
class GlobalExceptionHandler {
     suspend fun handleUncaught(ex:Throwable) = …
}
```

Finally produces ErrorResponse

API

```
class DevController(val devService: DevService) {
   @GetMapping("/api/devs")
   @ResponseBody
   suspend fun getBestLanguageOf(@RequestParam("name") name: String):Language =
       devService.getBestLanguageOf(name)
           .mapError { throw ApplicationException(ErrorType.Server, it) }
           .getOrThrow()
}
```

Service

```
class DevService(val devDao:DevDao, val langApi:LangApi) {
   suspend fun getBestLanguageOf(name:String):Result<Language> = result {
       devDao.getDevByName(name).bind()?.let {
           langApi.selectBest(it.languages).bind()
       }  ?: Language("Kotlin")
   }
```

Mmh, actually I don't do anything with `Result<T>`, so why is it there?

DAO

```
class DevDao {
   suspend fun getDevByName(name:String):Result<Developer?> = …
}
class LangApi {
   suspend fun selectBest(lang:List<Language>):Result<Language?> = …
}
```

… if in 95% of all cases you don't use them anyway

Exception Handler

```
class GlobalExceptionHandler {
     suspend fun handleUncaught(ex:Throwable) = …
}
```

API

```
class DevController(val devService: DevService) {
   @GetMapping("/api/devs")
   @ResponseBody
   suspend fun getBestLanguageOf(@RequestParam("name") name: String):Language =
       devService.getBestLanguageOf(name)
}
```

Service

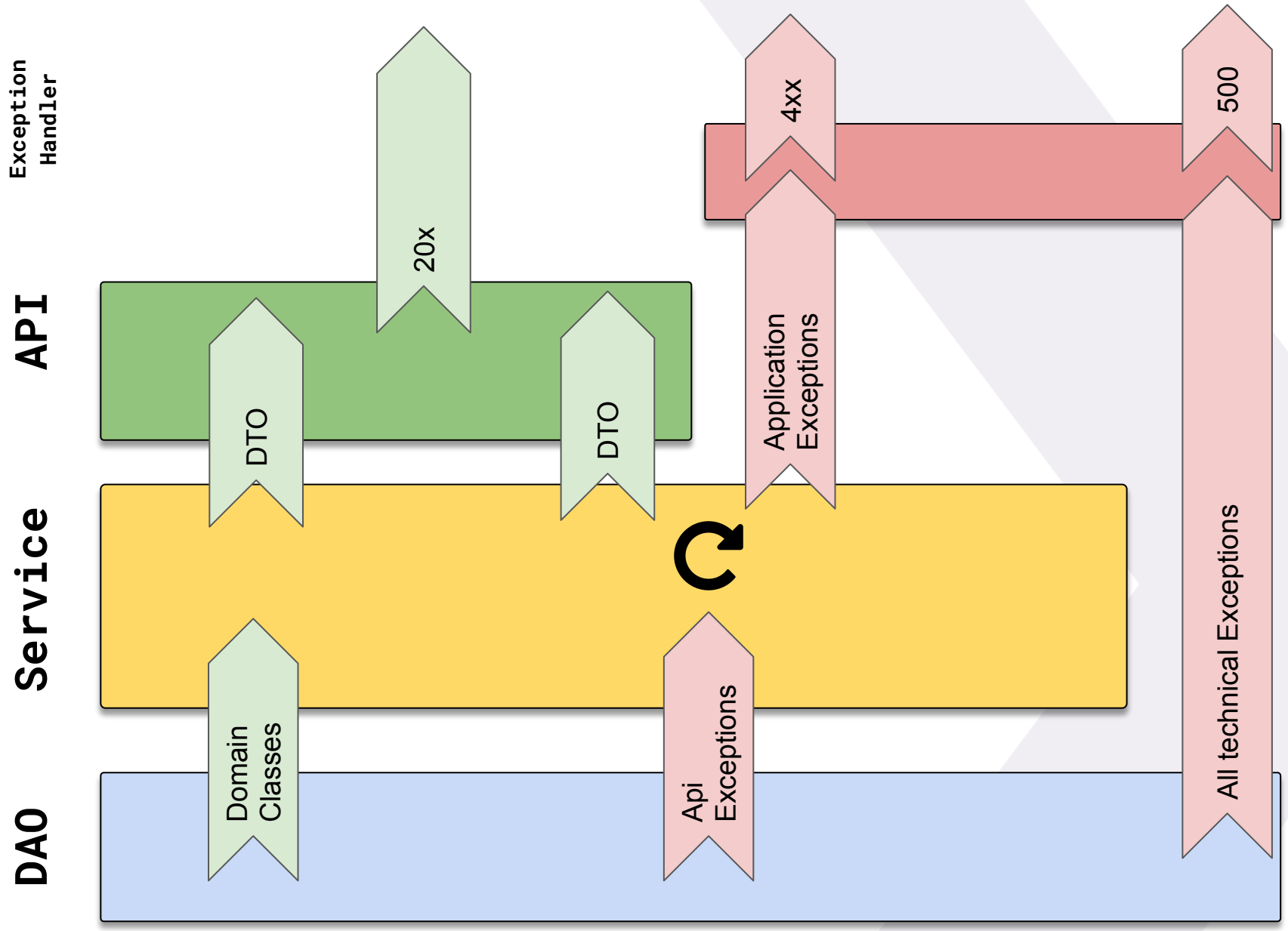
```
class DevService(val devDao:DevDao, val langApi:LangApi) {
   suspend fun getBestLanguageOf(name:String):Language =
       devDao.getDevByName(name)?.let {
           langApi.selectBest(it.languages)
       }  ?: Language("Kotlin")
   }
```

DAO

```
class DevDao {
   suspend fun getDevByName(name:String):Developer? = …
}
class LangApi {
   suspend fun selectBest(lang:List<Language>):Language? = …
}
```

Only domain logic, no 'additional' abstractions. Looks good to me!

But how about the other 5%?



Use Monads *selectively* where needed and useful

Exception Handler

```
class GlobalExceptionHandler {
     suspend fun handleUncaught(ex:Throwable) = …
}
```

API

```
class DevController(val devService: DevService) {
   @GetMapping("/api/devs")
   @ResponseBody
   suspend fun getBestLanguageOf(@RequestParam("name") name: String):Language =
       devService.getBestLanguageOf(name)
}
```

Service

```
class DevService(val devDao:DevDao, val langApi:LangApi) {
   suspend fun getBestLanguageOf(name:String):Language =
      devDao.getDevByName(name).let { dev ->
        langApi.selectBest(dev?.languages.orEmpty()).recover {
            when(it.code) {
               ERROR_LANG_NOT_FOUND -> Language("Kotlin")
               ERROR_TOKEN_EXPIRED -> …
               else -> raise(it)
            }
      }.map { it ?: Language("Kotlin") }
}.getOrElse { throw  ApplicationException(it.code.toString()) }
```

DAO

```
class DevDao {
   suspend fun getDevByName(name:String):Developer? = …
}
class LangApi {
   suspend fun selectBest(lang:List<Language>):Either<ApiError, Language?> = …
}
```

Which is win win:
1. explicit contract
2. easy access to error reply

To trigger error replies, I still can rely on my `ExceptionHandler`

For DB stuff, I only need entities

For external API's `Either<A,B>` makes the contract explicit

With this approach in mind: Cherry Pick the best!

#1 handle multiple null values

```kotlin
fun createKotlinDeveloper(name:String?, age:String?, languages: List<Language>): KotlinDeveloper? {
    val kotlin = languages.firstOrNull{it.name == "Kotlin"}
    return if(name != null && age.toIntOrNull() != null && kotlin != null) {
        KotlinDeveloper(
            name = name,
            age = age.toInt(),
            otherLanguages =  languages - kotlin
        )
    } else null
}
```

How can I prevent these tedious `null` checks?

```kotlin
fun createKotlinDeveloper(name:String?, age:String?, languages: List<Language>): KotlinDeveloper? =
    nullable {
        val kotlin = languages.firstOrNull{it.name == "Kotlin"}.bind()
        KotlinDeveloper(
            name = name.bind(),
            age = age.toIntOrNull().bind(),
            otherLanguages = languages - kotlin
        )
}
```

…with Arrow's `nullable{ }` comprehension!

With this approach in mind: Cherry Pick the best!

#2 composable validation with accumulated errors

```kotlin
fun createKotlinDeveloper(name:String?, age:String?, languages: List<Language>): KotlinDeveloper {
    val errors = mutableListOf<ValidationError>()
    if(name.isNullOrEmpty()) errors.add(NameInvalid)
    if(age?.toIntOrNull()?.let{it > 0} != false) errors.add(AgeInvalid)
    if(languages.isNotEmpty()) errors.add(NoLanguage)
    return if(errors.isEmpty())
        KotlinDeveloper(name!!, age!!.toInt(), languages)
    else
        throw ValidationsException(errors)
}
```

Value conversion done twice - possible maintenance issue

Throw Exception - instead of having an explicit return type for validation errors

Explicit signature for validation errors

```kotlin
context(Raise<List<ValidationError>>)
fun createKotlinDeveloper(name:String?, age:String?, languages: List<Language>):KotlinDeveloper =
    zipOrAccumulate(
        { ensureNotNull(name){ NameInvalid }},
        { ensureNotNull(age?.toIntOrNull()){ AgeInvalid }},
        { ensure(languages.isNotEmpty()){ NoLanguage }}
    ) {validName, validAge, validLangs -> KotlinDeveloper(validName, validAge, validLangs)}
}
```

Better: Use `zipOrAccumulate{ }` to *validate* and *accumulate errors*

Combined *conversion* and *verification*.

Converted and validated values are passed to conversion function

With this approach in mind: Cherry Pick the best!

#3 optics

```
val dev = Developer(
    name = "John",
    age = 32,
    primaryLanguage = Language("Kotlin", LanguageStats(popularity = 9)))
)
```

```
val devChanged = dev.copy(
    primaryLanguage = dev.primaryLanguage.copy(
        stats = dev.primaryLanguage.stats.copy(popularity = 10)
    )
)
```

Changing data in immutable nested data-structures is quite tedious with `copy()`...

But not with Arrow's `optics{ }`!

```
val devChanged = Developer.Companion.primaryLanguage.stats.modify(dev){it.copy(popularity = 10)}
```

Note: this requires Arrow annotations and a KSP compiler plugin to be configured in you build.

```
@optics
data class Developer(val name:String,val age:Int,val primaryLanguage:Language){companion object{}}

@optics
data class Language(val name:String, val stats: LanguageStats){companion object {}}

@optics
data class LanguageStats(val popularity:Int){companion object {}}
```

PICK		SKIP
Monads were useful		***Monads everywhere***
Code primarily reflecting your domain	over	*Code looking like a Monad/Container terminal*
`Either` *where the error part of a return is relevant*	over	*Using* `Exceptions` *for fine-grained business logic*
Other useful Monads like `Validation`, `Resource`, `Ior`, *-> Arrow*	over	*Writing these abstractions yourself*

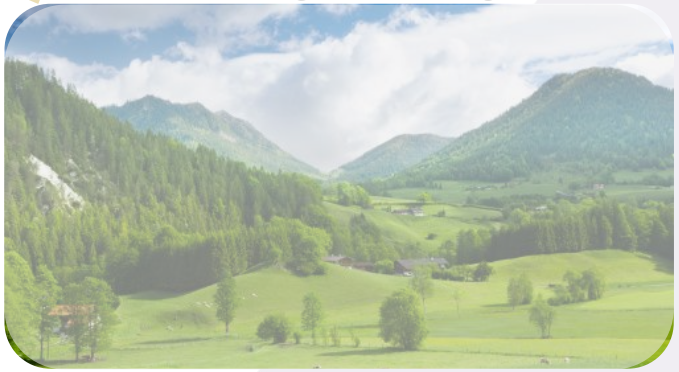
Welcome to:

The prosperous alpine meadows of **Functional Common Sense**

The lonely peak of Monadic Mist

The abundant heights of Higher-Order Functions

The fruitful valley of Expression Oriented Programming



The swamps of imperative programming

Making things simple is hard

Keeping things simple is even harder

Use Functional Programming

to make things *simpler*

Kotlin Training

Certified by JetBrains

Xebia Academy

...and check out our Kotlin courses ;-)

Urs Peter

LinkedIn: bit.ly/urs-peter-linked-in

Email: upeter@xebia.com

Blog: xebia.com/blog/