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/

KOTLIN CONF'23

About Me:

Kotlin Training
Certified by JetBrains

Xebia Academy

Kotlin

Scala

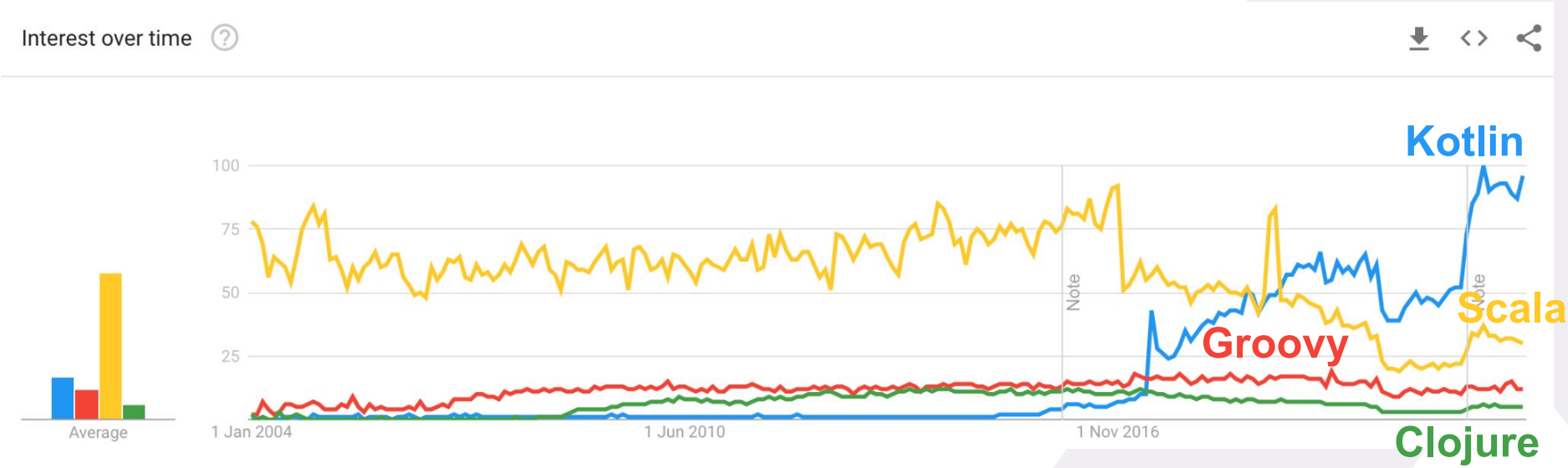
Java

Why this talk?

Why?

Kotlin

Scala



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

Nice to meet:

Let’s start a ‘transforming’ journey

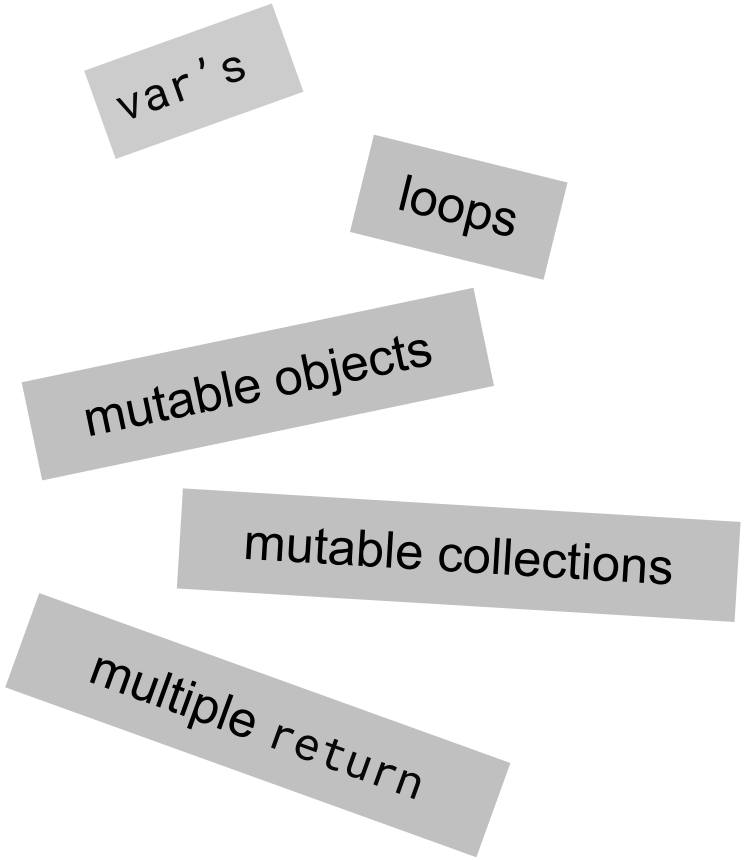
Starting point:
The swamps of Imperative Programming

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

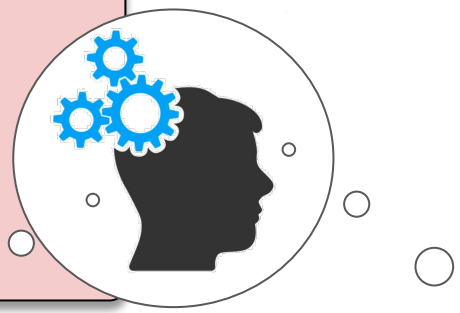
Imperative programming relies on *declaring variables* that are *mutated* along the way.

```
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*:

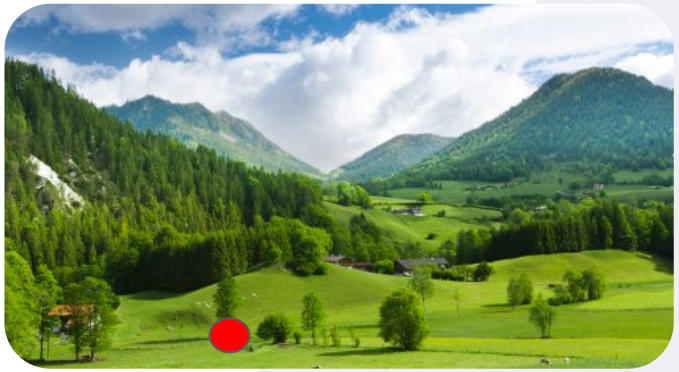
```
fun findBestDev(lang: String): Developer? =
  try {
    client.getAll<Developer>()
     .filter{ it.languages.contains(lang) }
     .maxByOrNull{ it.experience }
  } catch (ex:Exception) {
     null
  }
}
```

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

Welcome to:

The swamps of imperative programming

The fruitful valley of **Expression Oriented Programming**



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

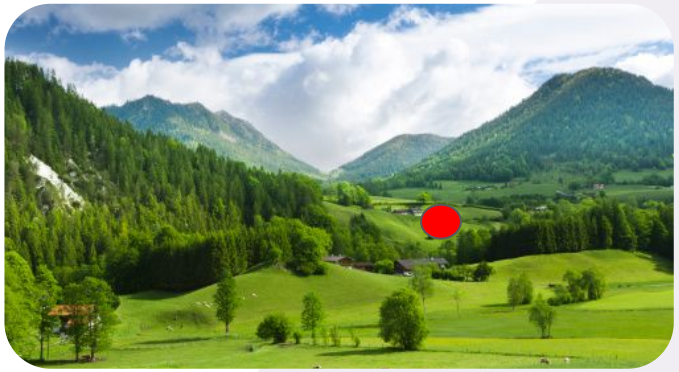
File Block

```
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())
        }
     }
  }
```

Welcome to:

The swamps of imperative programming

The fruitful valley of **Expression Oriented Programming with clearly scoped code**



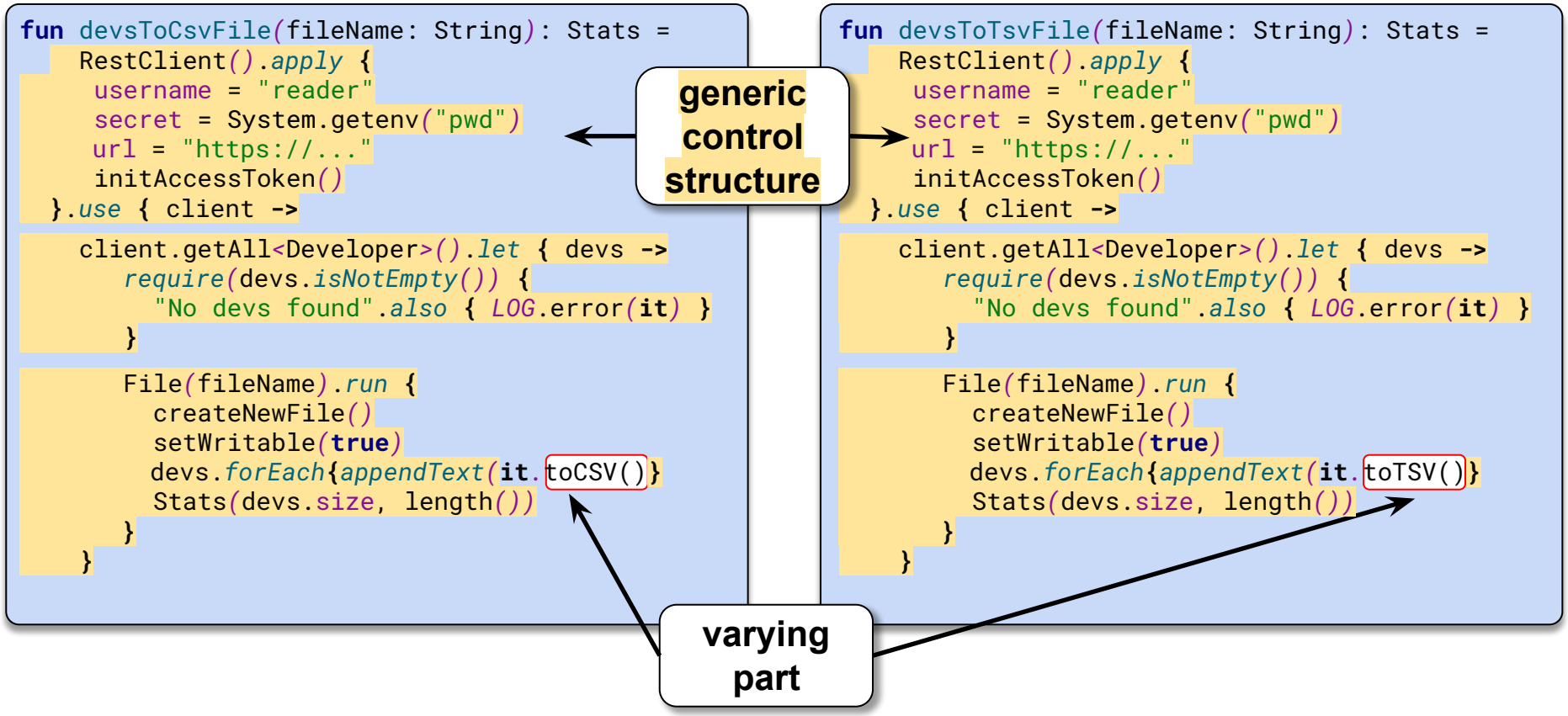
How to re-use Control Structures?

```
fun devsToCsvFile(fileName: 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(it.toCSV())}
         Stats(devs.size, length())
       }
    }
 }
```

```
fun devsToTsvFile(fileName: 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(it.toTSV())}
         Stats(devs.size, length())
       }
    }
 }
```

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): 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())
        }
     }
  }
```

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???()}
```

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)
   }
}
```

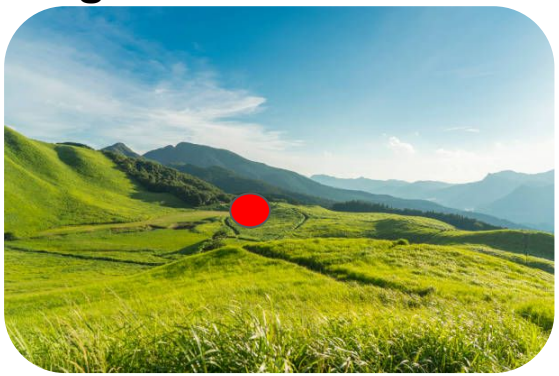
Higher-Order Functions also shine in tests

```
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
        ...
}
```

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?

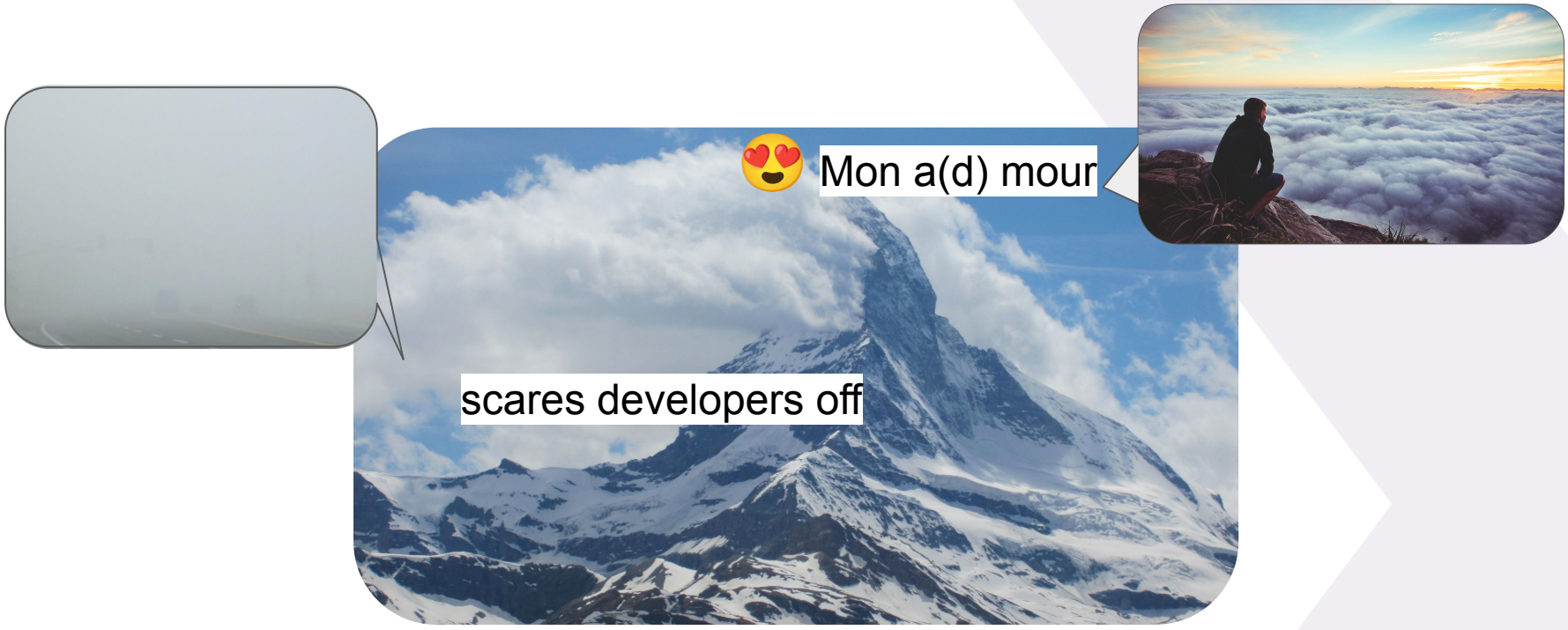
-> Can we leverage them beyond Collections?

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)
```

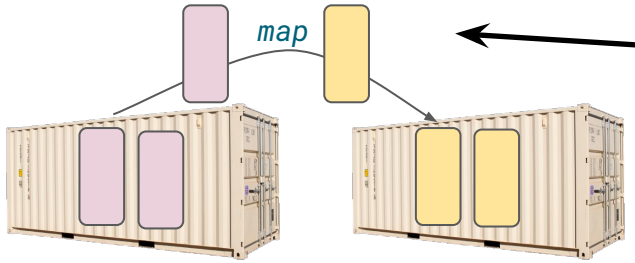
When we can combine *types* that results in a *new type* containing the elements of the initial ones, we are talking about a: **Monoid**

```
interface Monoid<T> {
    fun empty():Monoid<T>
    fun combine(other:Monoid<T>):Monoid<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>
}
```

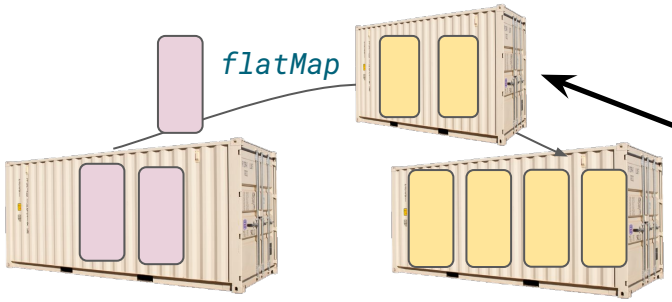
We all know Monads - and use them daily!

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

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

```
//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 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**':

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
```

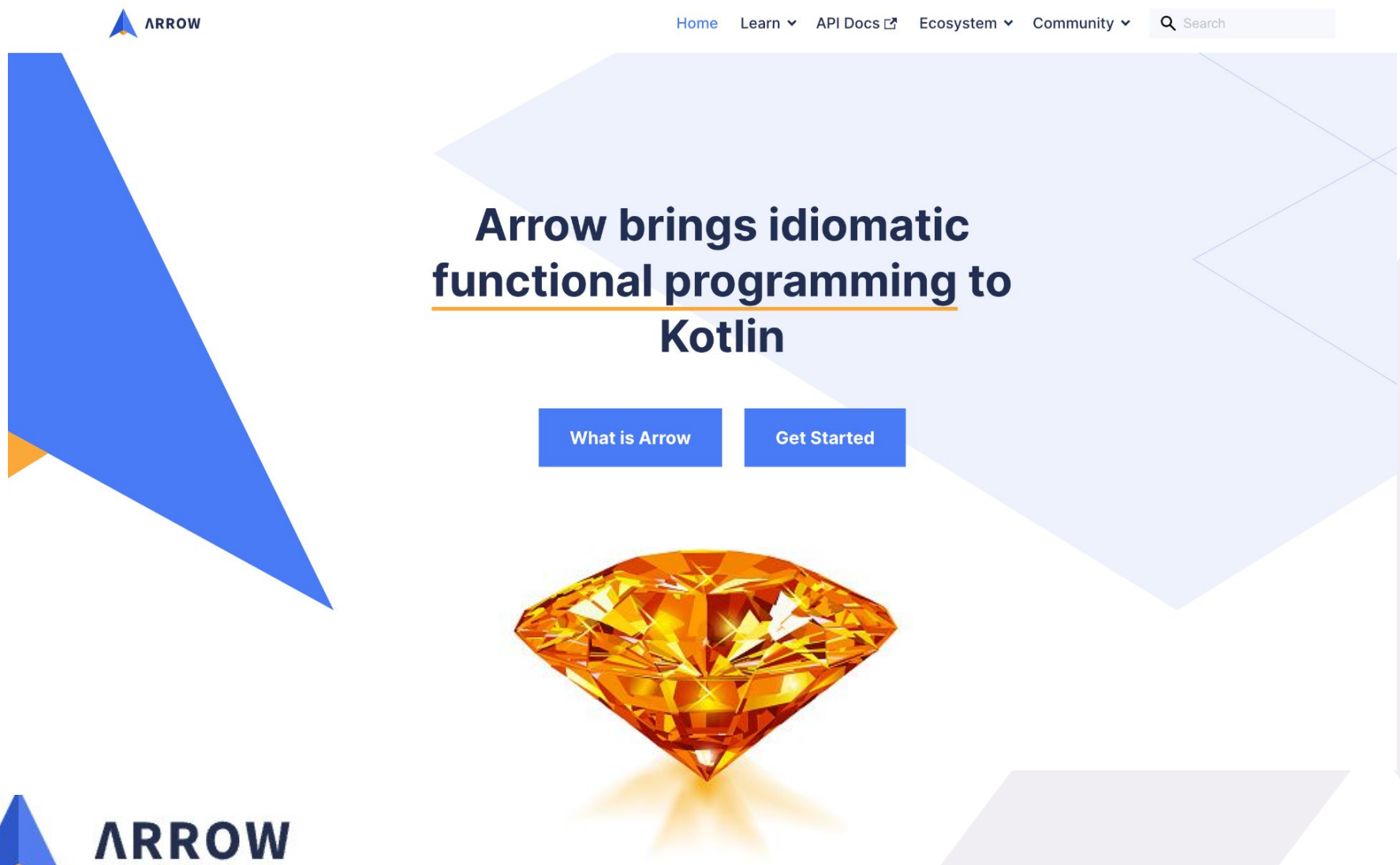
```
fun getDevByName(name:String):Result<Developer>
fun selectBest(langs:List<Language>):Result<Language>
fun getStatsFor(lang:Language):Result<LanguageStats>
```

```
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.

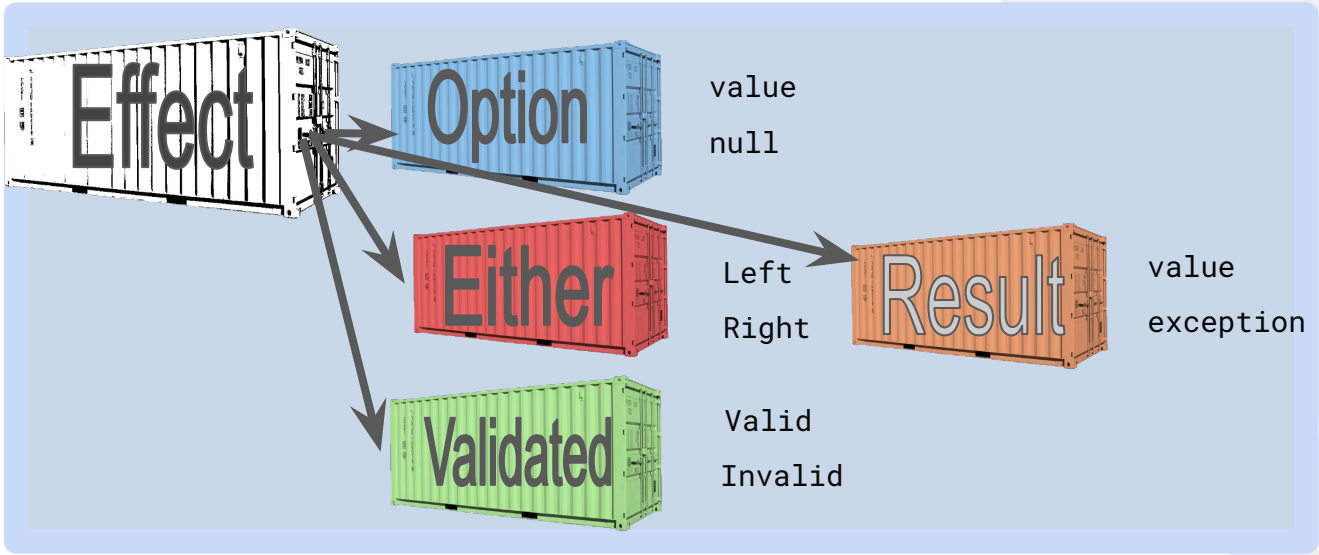
Is there no better way?

Welcome to Arrow



ARROW

Some Fancy Arrow Monads



ARROW

Arrow has a nice solution to avoid flatMap

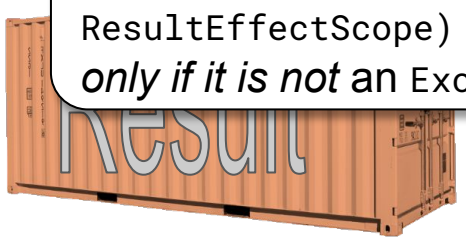
Which are: **Monad Comprehensions**

conceptually ported from functional languages like Scala and Haskell

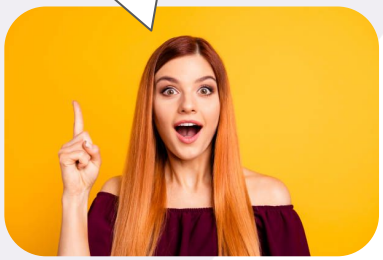
```
fun statsOfBestLanguageOf(name:String):Result<LanguageStats>=
   client.getDevByName(dev1.name).flatMap { dev ->
       client.selectBest(dev.languages).flatMap { language ->
           client.getStatsFor(language)
       }
   }
```

```
suspend fun statsOfBestLanguageOf(name:String):Result<LanguageStats>=
    result { this:ResultEffectScope
        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 `ResultEffectScope`) 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(EffectScope<Throwable>)
suspend 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
}
```

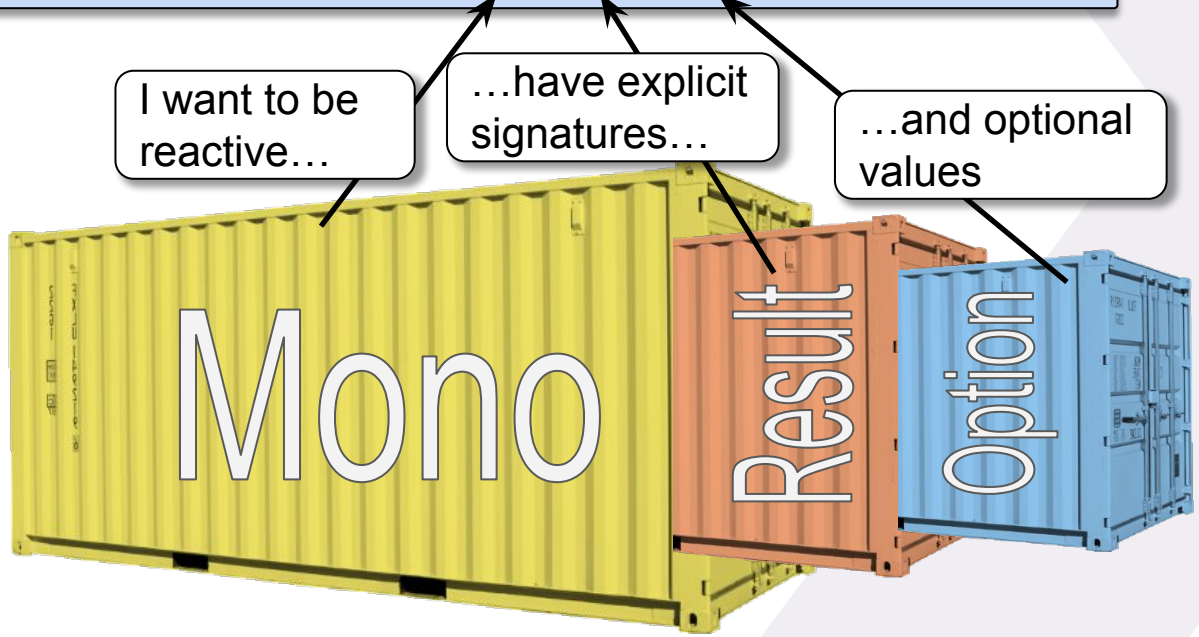
Let's Compose the Composables!

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>>>
```



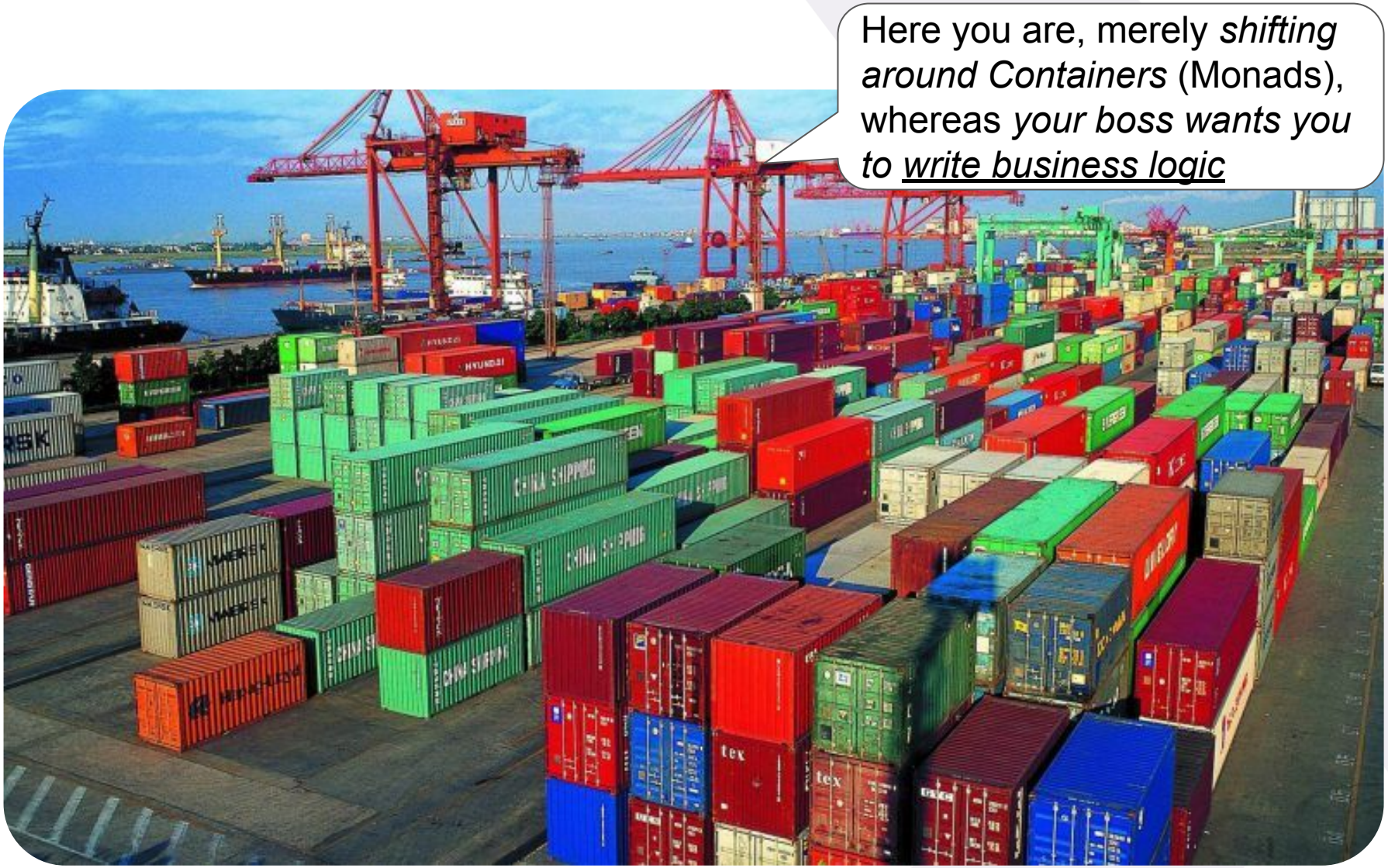
Let’s Compose the Composables???

```
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

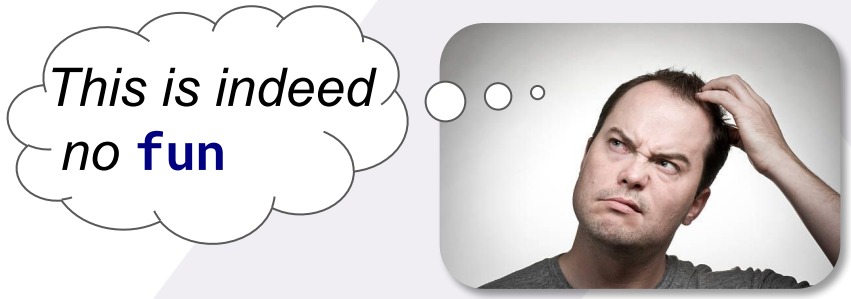
Nope, sorry…

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?

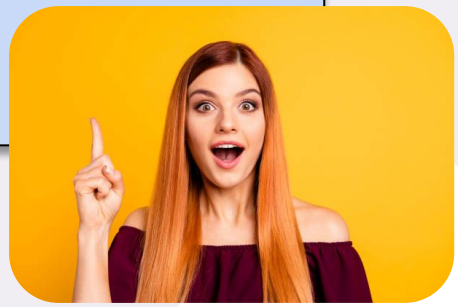
So, are these Monads no good at all?

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 }) }
}
```

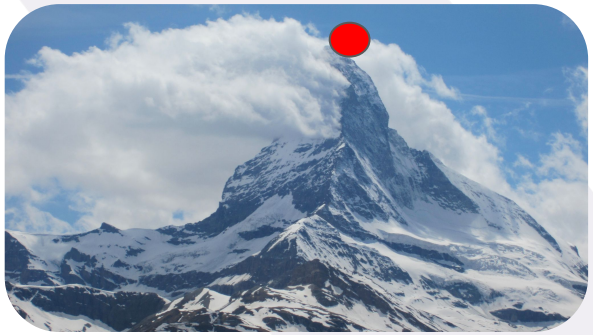
Except for the `bind()` that's just 'normal' Kotlin code

```
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() }
   }
```



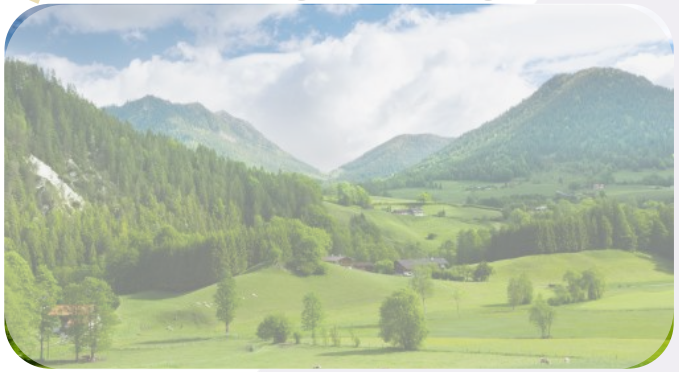
Welcome to:

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.

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

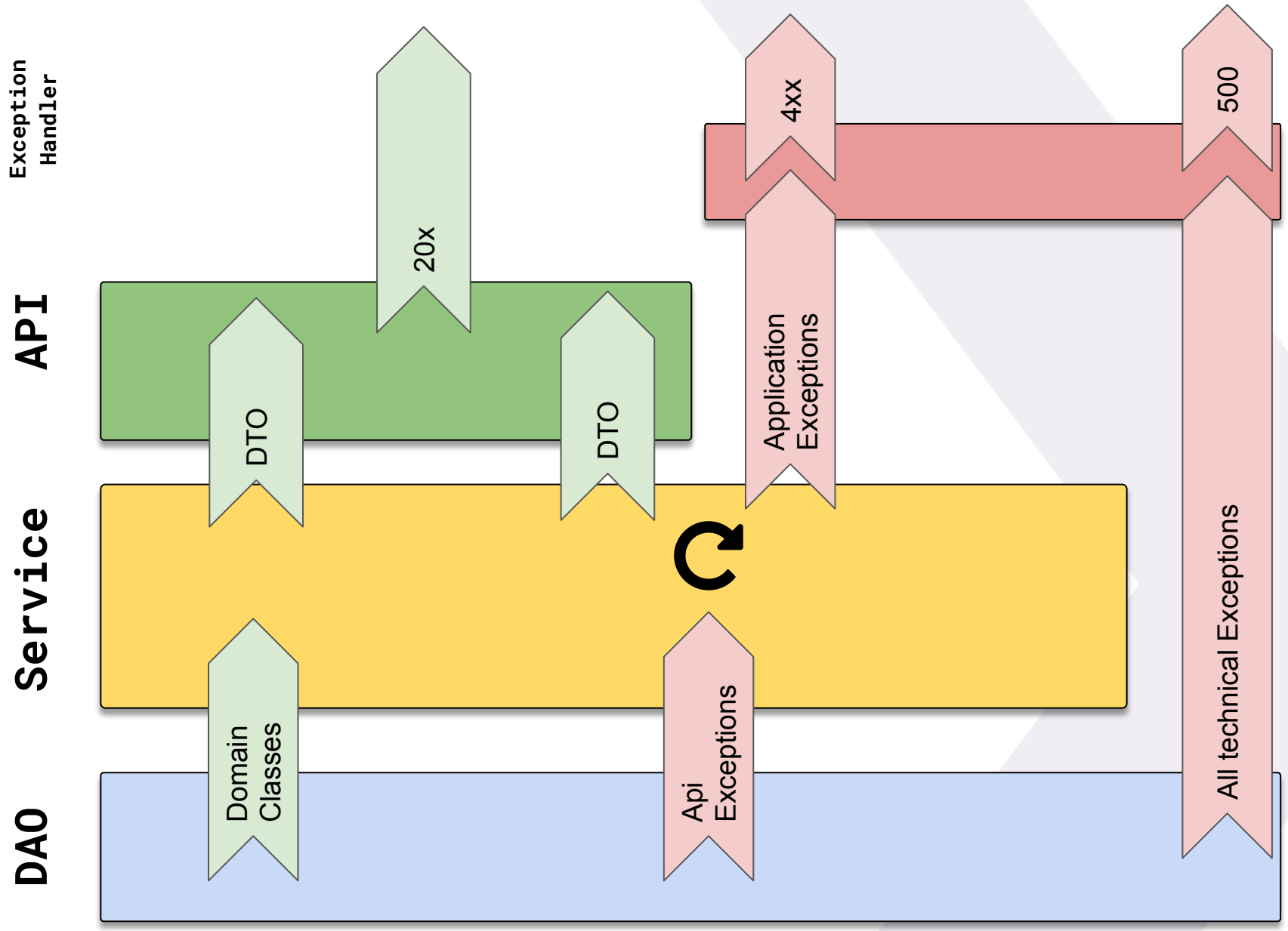
```
class DevService(val devDao:DevDao, val langApi:LangApi) {
   suspend fun getBestLanguageOf(name:String):Language =
       devDao.getDevByName(name)?.let {
           langApi.selectBest(it.languages)
       }  ?: Language("Kotlin")
   }
```

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()).handleErrorWith {
            when(it.code) {
               ERROR_LANG_NOT_FOUND -> Language("Kotlin").right()
               ERROR_TOKEN_EXPIRED -> …
               else -> it.left()
            }
      }.map { it ?: Language("Kotlin") }
}.getOrHandle { throw  ApplicationException(it.code.toString()) }
```

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

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

DAO

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

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

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

How can I prevent these tedious `null` checks?

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

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

With this approach in mind: Cherry Pick the best!

#2

```
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 {}}
```

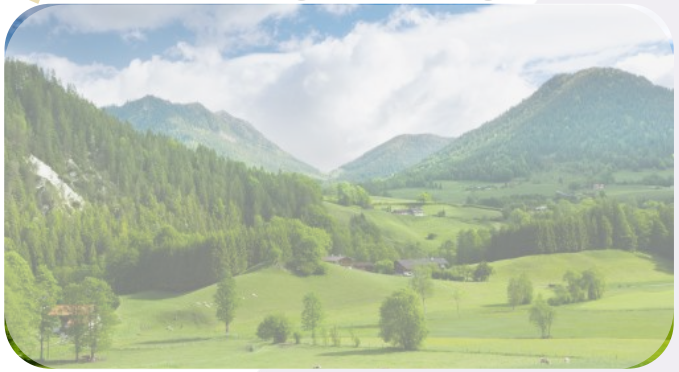
Welcome to:

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

Kotlin Training

Certified by JetBrains

Xebia Academy

Urs Peter

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

Email: upeter@xebia.com

Blog: xebia.com/blog/