

# How to Kotlin Arrow

01.02.2024 Daniel Tiefenauer



# Agenda

- Functional Programming in a nutshell
- Introduction to Arrow
- Nullability vs. Exceptions vs. Kotlin's Result Type
- Introduction to monads by example of Arrow's Either monad
- Introduction to Arrow's Optics and Lenses



# Why Kotlin Arrow (yet another framework)



# FP vs. Imperative programming

handle objects  
and mutate state



Compose  
functions to  
make a program



# Advantages of pure FP

**Explicitness**  
What goes in?  
What comes out

**Statelessness**  
Everything is  
immutable

**Simplicity**  
No caveats

**No side effects**  
focus on function

**Testability**  
Program  
Verification

**Shift left**  
Runtime  
Exceptions  
become Compile-  
time Errors

easier to read  
and debug  
less bugs



# Disadvantages of pure FP

- Steep learning curve
- Some concepts might be hard to grasp at first (especially coming from OO)



# FP concepts

- Functions are first class citizens → no @FunctionalInterface
- High Order Functions → currying and partial application
- Pure Functions and Referential Transparency
- Recursion
- **Typed Errors**
- Algebraic Data Types: e.g. Union Types, Intersection Types,
- **Immutability**
- Statelessness/no side effects
- Declarative Programming
- **Monads**

Leveraged by



(and this presentation)



# The idea behind Arrow

- **Goal:** make FP paradigms easy and accessible for everyone
  - Fill the gaps between Scala/Haskell + Kotlin
- Uniform + idiomatic FP paradigms for every Kotlin developer
  - FP has a steep learning curve, is hard to explain if you're not familiar with the concepts
- First commit >10y ago
  - Merge of three libraries trying to fill the gap between Scala+Kotlin
    - MarioAriasC/funKTionale
    - FinceCinnamon/Katz → Kategory
    - nomisRev/optikal → optics



# Arrow's History



March 2013

**MarioAriasC/funKTionale**



March 2017

**FineCinnamon/Katz**



June 2017

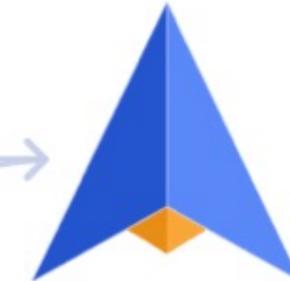
**Kategory**

Oct 2017



Sept 2017

**nomisRev/optikal**



Dec 2017

**arrow-kt**



# Practical example: A simple function

- Parse a string containing a number and return its reciprocal Int-value as string
  - Both String-representations of Integers and Doubles should be accepted
  - When passing the String-representation of a Double, it should always be rounded up (ceiling) to get the int-value
  - When passing the String-representation of a Double, both comma and dot should be accepted as fractional delimiter
- Examples:

Input	Output
“42”	1/42
“42.1”	1/43
“42.5”	1/43
“42.6”	1/43
“42,1”	1/43



## The function signature

# toReciprocal: String -> Double

What is the result  
if I enter an  
empty String?

What is the result  
if I enter “0”?

What is the result  
if I enter “abc”?





# The naïve (Java-ish) approach

- Function signature:

String -> Double?

- Kotlin implementation:

```
/**  
 * Converts a string-representation of a number to its reciprocal double value.  
 * Fractional values will always be rounded up.  
 * Both '.' and ',' are accepted as separators for the fractional part  
 */
```

```
fun toReciprocal(string: String): Double?
```



I don't like  
NPE's



Why is it  
null?



# DEMO TIME!!!





# A slightly better approach (using Exceptions)

- Function signature:

String -> Double

- Kotlin implementation:

```
/*
 * Converts a string-representation of a number to its reciprocal double value.
 * Fractional values will always be rounded up.
 * Both '.' and ',' are accepted as separators for the fractional part
 * @throws IllegalArgumentException if the string-representation of the number is not within the interval [Int.MIN_VALUE, Int.MAX_VALUE]
 * @throws IllegalArgumentException if the string-representation of the number is zero
 * @throws NumberFormatException if the string is not a valid string-representation of a number
 */
fun toReciprocal(string: String): Double
```

Even when handling the Exceptions,  
there is no type-safety (I would have  
to parse the Exception message)

Yay! No more NPEs!



I hope the creator of  
this functions took the  
time to list all the  
possible Exceptions...



I need to read the  
Javadoc (or  
reverse-engineer)



How can I find out  
what caused an  
IllegalArgumentException?



I should probably handle  
the exceptions – but I'm  
not forced to (my code will  
compile without exception  
handling)





# DEMO TIME!!!





# Kotlin's Result<T> to the rescue!

- Function signature

String -> Result<Double>

- Kotlin implementation

```
/**
```

```
 * Converts a string-representation of a number to its reciprocal double value.
```

```
 * Fractional values will always be rounded up.
```

```
 * Both '.' and ',' are accepted as separators for the fractional part
```

```
*/
```

```
fun toReciprocal(string: String): Result<Double>
```

Yay! No more NPEs and  
also no Exceptions! AND I  
can use pattern matching!



I still don't know why  
something went wrong without  
checking the type or the  
message...





# DEMO TIME!!!





# There must be a better way!

- Spoiler alert: there is!
- Expectations from the better solution:
  - Null-safety
  - Type-safe error handling at compile (!) time



# Enter monads



# What is a monad?

- I asked Wikipedia: [https://en.wikipedia.org/wiki/Monad\\_\(functional\\_programming\)](https://en.wikipedia.org/wiki/Monad_(functional_programming))

In [functional programming](#), a **monad** is a structure that combines program fragments ([functions](#)) and wraps their [return values](#) in a [type](#) with additional computation. In addition to defining a wrapping **monadic type**, monads define two [operators](#): one to wrap a value in the monad type, and another to compose together functions that output values of the monad type (these are known as **monadic functions**). General-purpose languages use monads to reduce [boilerplate code](#) needed for common operations (such as dealing with undefined values or fallible functions, or encapsulating bookkeeping code). Functional languages use monads to turn complicated sequences of functions into succinct pipelines that abstract away [control flow](#), and [side-effects](#).<sup>[1][2]</sup>



- I asked a mathematician

## What is a monad?

A monad is a triple  $(T, \eta, \mu)$  where  $T$  is an endofunctor  $T: X \rightarrow X$   
and  $\eta: I \rightarrow T$  and  $\mu: T \times T \rightarrow T$  are 2 natural transformations  
satisfying these laws:

Identity law:  $\mu(\eta(T)) = T = \mu(T(\eta))$

Associative law:  $\mu(\mu(T \times T) \times T) = \mu(T \times \mu(T \times T))$

In other words: "*a monad in X is just a monoid in the category of endofunctors of X, with product  $\times$  replaced by composition of endofunctors and unit set by the identity endofunctor*"

## What's the problem?

Source: <https://thomasandolf.medium.com/write-a-monad-in-java-seriously-50a9047c9839>



# What is a monad?

- I asked ChatGPT



You

What is a monad?



ChatGPT

A monad is a design pattern used in functional programming to handle computations with side effects, such as input/output or state changes, in a structured and composable way. The concept of monads originated from category theory, a branch of abstract mathematics.

In programming, a monad is a type that represents a computation or a sequence of computations. It provides a set of operations that allow you to chain computations together, while also handling side effects in a controlled manner. The key features of a monad include:

1. **Unit/Return:** There is a function (often called `unit` or `return`) that lifts a value into the monad, allowing you to start a computation with a pure value.
2. **Bind/FlatMap:** There is a bind operation (often represented by `>>=` or `flatMap`) that takes a value in the monad and a function that produces another value in the same monad. This allows you to sequence computations and thread values through them.

Monads are known for their ability to encapsulate and control side effects while providing a clean and composable way to express complex computations. They are widely used in functional programming languages such as Haskell, Scala, and F#, but the concept has also influenced the design of libraries in other languages, including JavaScript (e.g., using Promises) and Python (e.g., using the `async/await` syntax for asynchronous programming).



# What is a monad?

- For the sake of simplicity: A monad is a wrapper (data + context)





# Monads in Java

- `Optional<T>`
- `Stream<T>`
- `CompletableFuture<T>`
- `Try/catch`



# Arrow's Either monad



# Arrow's Optics (Lenses)

- Add the KSP plugin

```
plugins {  
    id("com.google.devtools.ksp") version "1.9.21-1.0.16"  
}
```

- Add the Arrow dependencies

```
dependencies {  
    implementation(libs.arrow.optics)  
    ksp(libs.arrow.optics.ksp.plugin)  
}
```



# Enter Arrow: The Either monad

```
dependencies {  
    implementation("io.arrow-kt:arrow-core:1.2.0")  
}
```

- Either<Left, Right>
  - Left is context
  - Right is data (“Right is right”)
- Kotlin implementation in Arrow:

```
sealed class Either<out A, out B>  
data class Left<out A>(val value: A) : Either<A, Nothing>()  
data class Right<out B>(val value: B) : Either<Nothing, B>()
```



## Back to our example

`toReciprocal : String -> Either<ParseError, Double>`

- Either the string could be parsed (Right) or there was some error (Left)
- If everything went right: the data is the parsed reciprocal as double
- If something went wrong: the context is the (typed!) error



# DEMO TIME!!!





- Code will fail and we have to deal with it
- Arrow's Either monad introduces type-safe error handling as an alternative to throwing exceptions
- Just like Kotlin introduced null-safety at compile-time, Arrow introduces error handling at compile-time instead of runtime exceptions (shift-left)



# Optics



# Arrow's Optics (Lenses)

- Add the KSP plugin

```
plugins {  
    id("com.google.devtools.ksp") version "1.9.21-1.0.16"  
}
```

- Add the Arrow dependencies

```
dependencies {  
    implementation(libs.arrow.optics)  
    ksp(libs.arrow.optics.ksp.plugin)  
}
```



# A sample structure

```
{  
  "policyNumber": "123456",  
  "insurant": {  
    "firstName": "John",  
    "lastName": "Doe",  
    "address": {  
      "street": "Sesame Street",  
      "streetNumber": "42", ← I want to update this!  
      "zip": "1234",  
      "city": "Muppettown"  
    }  
  }  
}
```



# Dealing with immutable objects

- Immutability is good! ...
- ... but is sometimes difficult to handle

```
data class InsurancePolicy(val policyNumber: String, val insurant: Insurant, val vehicle: Vehicle)
data class Insurant(val firstName: String, val lastName: String, val address: Address)
data class Vehicle(val make: String, val model: String, val horsePower: Int)
data class Address(val street: String, val streetNumber: String, val zip: String, val city: String)
```

```
// an immutable InsurancePolicy
val policy = InsurancePolicy(..., insurant = Insurant(..., address = Address(..., streetNumber = "42")))
```

```
// alternative pyramid of doom
val updatedPolicy = policy.copy(
    insurant = policy.insurant.copy(
        address = policy.insurant.address.copy(
            streetNumber = "43"
        )
    )
)
```



# Lenses to the rescue!

- A lens focuses on one particular property of an object

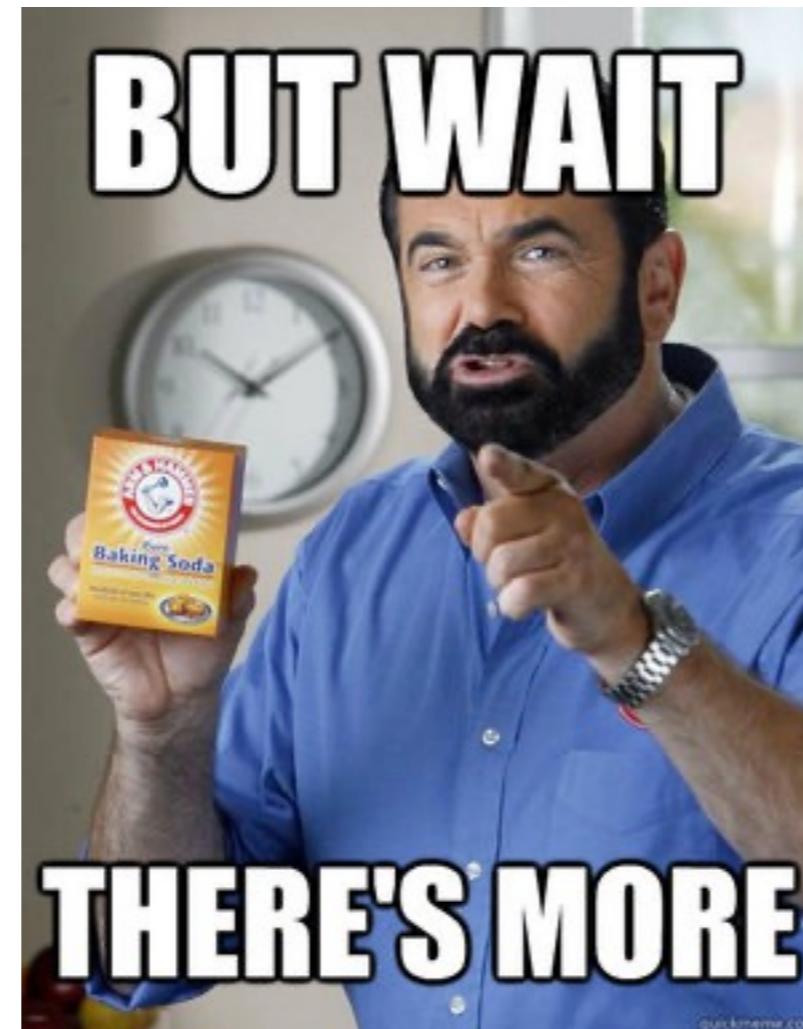
```
// InsurancePolicy > Insurant
private val insurantLens: Lens<InsurancePolicy, Insurant> = Lens(
    get = { it.insurant },
    set = { insurancePolicy, insurant -> insurancePolicy.copy(insurant = insurant) }
)
// Insurant > Address
val addressLens: Lens<Insurant, Address> = Lens(...)
// Address > streetNumber
val streetNumberLens: Lens<Address, String> = Lens(...)
```

- Lenses can be combined...

```
val insurancePolicyInsurantAddressStreetNumberLens = insurantLens
    .compose(addressLens)
    .compose(streetNumberLens)
```

- ... to update a single property

```
insurancePolicyInsurantAddressStreetNumberLens.modify(policy) { "43" }
```





# Auto-generated Lenses (Gradle only)

Add this

```
@optics data class InsurancePolicy(val policyNumber: String, val insurant: Insurant, val vehicle: Vehicle) {companion object}  
@optics data class Insurant(val firstName: String, val lastName: String, val address: Address) {companion object}  
@optics data class Vehicle(val make: String, val model: String, val horsePower: Int) {companion object}  
@optics data class Address(val street: String, val streetNumber: String, val zip: String, val city: String) {companion object}
```

And this

```
InsurancePolicy.insurant.address.streetNumber.modify(policy) { "43" }
```

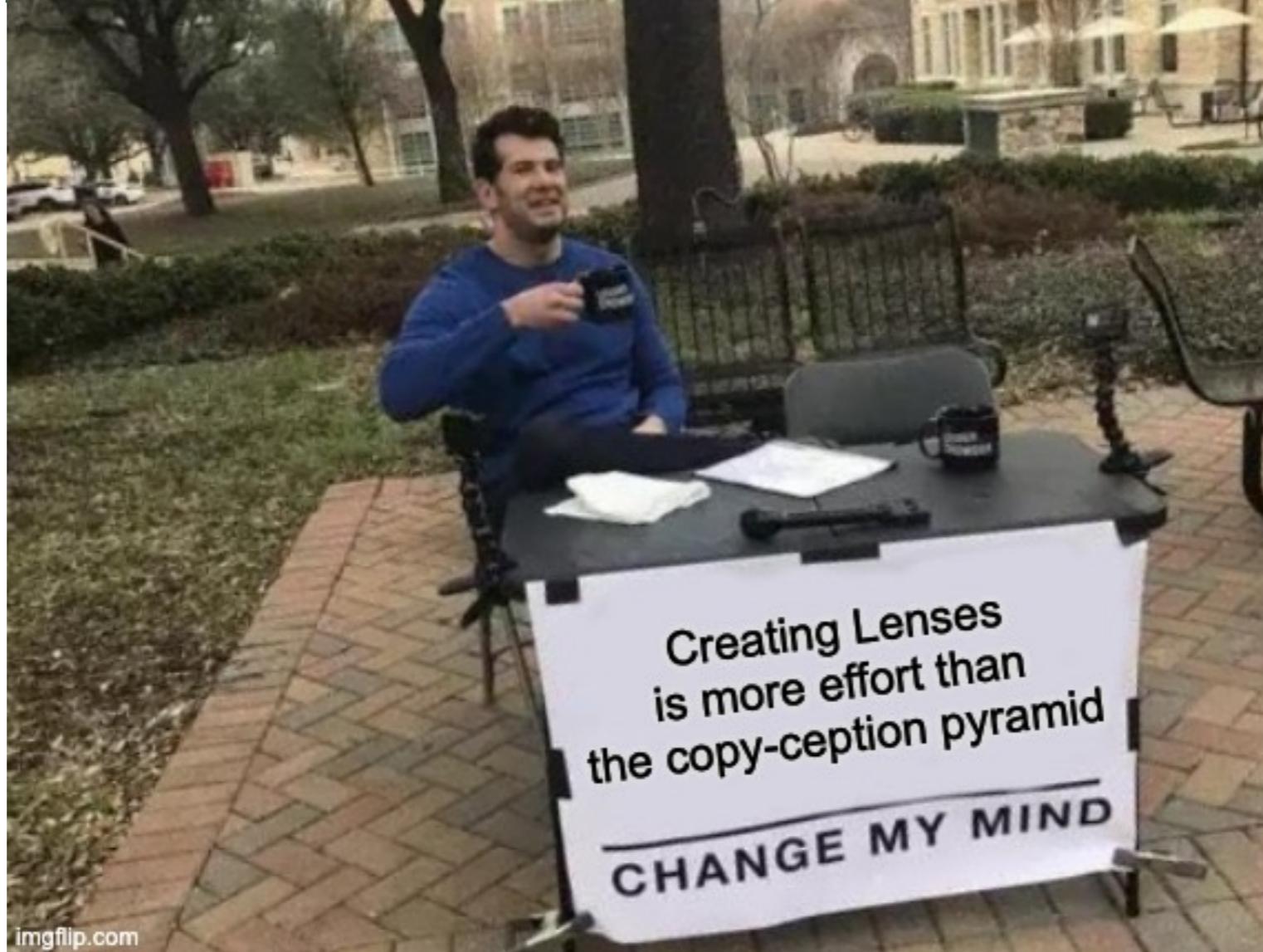


# DEMO TIME!!!





## DEMO TIME!!!





# Further reading



# Links

- Code examples: <https://github.com/mptechnology/mp-techlunch-kotlin-arrow>
- Arrow: <https://arrow-kt.io/>
- Very nice introduction into Monads by example of Kotlin/Arrow:  
<https://medium.com/@albert.llousas/monads-explained-in-kotlin-4126ac0cb7f2>
- Very good introduction into Arrow by Ties Van de Ven:  
<https://www.youtube.com/watch?v=eFheAErqJzA>
- Arrow on Slack: <https://slack-chats.kotlinlang.org/c/arrow>
- FP programming in general: Learn you a Haskell for Great Good!  
<https://learnyouahaskell.com>

# LOONEY TUNES



*"That's all Folks!"*