Error handling in Kotlin

so ... Kotlin in the backend 🤔

Kotlin does not have checked exceptions!

Why this might not be the best idea ?

Often used for flow control

=> violates the **Principle Of Least Astonishment**

Most exceptions are
accepted and / or expected use cases !

How is Kotlin different ?

We can use sealed classes

to represent restricted class hierarchies

… basically our result cases

```
sealed class ActionResult {

     data class Success(val result: MyValue) : ActionResult()

     data class Error(val message: String) : ActionResult()

}
```

```
sealed class ActionResult {

     data class Success(val result: MyValue) : ActionResult()

     data class Error(val message: String) : ActionResult()

}


 ...

 val result = when(someAction()) {

      is ActionResult.Success -> "GOOD"

      is ActionResult.Error -> "NOT GOOD"

 }
```

```
fun <T> mapTo(json: JsonObject, clazz: Class<T>): Outcome<T> =

          try {

               Outcome.Success(json.mapTo(clazz))

          } catch (e: Exception) {

               val message = "Cannot map to ${clazz.simpleName} due to missing or
invalid attribute: ${e.message}"

               Outcome.Failed(message, e)

          }
```

```kotlin
sealed class Outcome(out T) {

    data class Success<out T>(val message: T) : Outcome<T>()

    data class Failed(val message: String, val e: Exception) : Outcome<Nothing>()

}
```

```
when(action1()) {
    is Action1Result.Success -> { res ->
        when(action2(res)) {
            is Action2Result.Success -> { when(..) }
            is Action2Result.Error -> "NOT GOOD"
        }
    }
    is ActionResult.Error -> "NOT GOOD"
}
```

Arrow library

We know that functions can and will fail

… hence we make it explicit in the data type we return

Either

Is used to short-circuit a computation upon the first error.

The right hand side holds the successful values

```kotlin
fun <T> mapTo(json: JsonObject, clazz: Class<T>): Either<String, T> =

          try {

               Either.Right(json.mapTo(clazz))

          } catch (e: Exception) {

               val message = "Cannot create ${clazz.simpleName} due to missing or
invalid attribute: ${e.message}"

               Either.Left(message)

          }
```

```
mapTo(jsonObject, MyRequest::class.java))
     .fold(
          ifLeft = {
               // return 400
          }
          ifRight = {
               // do something with my object
          }
     )
```

Either

Let’s enumerate explicitly the things that can go wrong in our program.

```
fun <T> mapTo(json: JsonObject, clazz: Class<T>): Either<JsonMapError, T> =
    try {
        Either.Right(json.mapTo(clazz))
    } catch (e: Exception) {
        val error = if (condition)
            JsonMapError.Missing()
        else
            JsonMapError.Invalid()
        Either.Left(error)
    }
```

```kotlin
mapTo(jsonObject, MyRequest::class.java)
     .fold(
          ifLeft = { jsonMapError ->
               when(jsonMapError) {
                    is JsonMapError.Missing -> { /* report 400 and log this */ }
                    is JsonMapError.Invalid -> { /* report 400 and log that */ }
               }
          },
          ifRight = { /* do something with my object */ }
     )
```

```kotlin
mapTo(jsonObject, MyRequest::class.java)
     .map { myRequest -> "do some more work with ${myRequest.x}" }
     .fold(
          ifLeft = {
               when(it) {
                    is JsonMapError.Missing -> { /* report 400 and log this */ }
                    is JsonMapError.Invalid -> { /* report 400 and log that */ }
               }
          },
          ifRight = { /* do something with my object */ }
     )
```

```
extractCustomerId(requestContext) // Either<Problem, UUID>
     .flatMap { uuid -> getSomeOverview(uuid) } // <Problem, SomeOverviewResult>
     .fold(
          ifLeft = { problem -> ... },
          ifRight = { overview -> ... },
     )
```

To transform (map) the problem cases
according to the domain
at the boundaries of the layers

```
loginService.getLoginToken(username)
    .mapLeft { unknownUser: UnknownUser ->
        logger.info("No users found for username $username")
        Problem(title = "Unknown user", status = 404)
    }
    .map { loginToken -> ... }
```

```
binding<Problem, Pair<UUID, UUID>> {

          val (customerId) = extractCustomerId(ctx) // Either<Problem, UUID>

          val (detailId) = extractDetailId(ctx) // Either<Problem, UUID>

          Pair(customerId, detailId)

     }.flatMap { (customerId, detailId) ->

          getDetail(customerId, detailId) // Either<Problem, DetailResult>

     }.fold(

          ifLeft = { ... },

          ifRight = { ... },

     )
```

Try

= computation which can result in

- a result
- an exception if something went wrong

We can even combine Try with Either

```
Try {

          service.getDetail(detailId, customerId).also

     }.toEither {

               logger.error("Failed to get the detail with id $detailId: ", it)

               Problem(status = 500) // present with the left part

          }
```