To throw or not to throw

That is the question

Introduction

```
@PostMapping
fun doSomeMagic(@RequestBody magicInput: MagicInput): ResponseEntity<Unit>
{
    return try {
        magicService.doMagic(magicInput.data())
        ResponseEntity.status(HttpStatus.CREATED).build()
    } catch (ex: ValidationException) {
        ResponseEntity.ok().build()
    }
}
```

Exceptions - basics.

Classification of exceptions:

- checked - checked at compile time
- unchecked - checked at runtime
- errors - critical situations from which there is no way out, for example, out of memory.

Exceptions - java vs kotlin

Java:

```
public void someMethod() throws IOException{...}
```

Kotlin

```
@Throws(IOException::class)
fun create() {...}
```

Exceptions - java vs kotlin

1. Kotlin does not have checked exceptions.
2. All exceptions in Kotlin are unchecked.
3. Error (OutOfMemoryError) occurs in both Kotlin and Java.

It is possible to call a Java method that throws a checked exception from Kotlin code without the need to handle the exception.

To make Java aware of a checked exception thrown by Kotlin code, it is necessary to add the @Throws annotation with information about the exception.

Is it an real exception or just a quick way out?

What is an exceptional situation and what is just a "clever" way to jump out of the code?

- external service is not working
- incorrect SQL query
- data does not exist in the database but it should
- data exists in the database, but should it?
- data validation error
- all kinds of business errors

Bad habits

- We learned to throw unchecked exceptions because it's easy.
  - We don't always remember to handle them.
  - Sometimes the exception will just fly up to the client.
- We create many specific exceptions instead of one generic exception.
- We swallow the message contained in the exception (empty catch block or just empty log message).

Is there a different way?

Is throwing an exception the only way to tell the calling method that something went wrong?

- Java has proposed the vavr library to us, which provides the Either class and the Try class.
- In Kotlin, a lazy approach with nullable types can be used to avoid throwing exceptions.
- Although this approach may lack detailed information about what went wrong.
- The syntax of the language also allows us to quick implement a wrapper class similar to Either.

Is there a different way?

Sealed class/interface

```
sealed interface ProcessResult
data class SuccessProcessing<T>(val input: T) : ProcessResult
data class SkipProcessing(val message: String) : ProcessResult
```

Is there a different way?

Sealed class/interface

```
sealed class Outcome<out T : Any> {
    data class Success<out T : Any>(val value: T) : Outcome<T>()
    data class Error(val message: String) : Outcome<Nothing>()
}
```

Kotlin and sealed classes/interfaces

Pattern matching

```
fun extract(input: String): Outcome<String>

when (val result = extract(input)) {
    is Outcome.Error -> SkipProcessing(result.message)
    is Outcome.Success -> processSecondStep(result.value, input)
}
```

An real life example

```
fun doMagic(input: String) {
    val data = getData(input)
    validateData(input, data)
    saveData(input)
}

private fun getData(input: String): String {
    if (input.length < 3) throw ValidationException("")
    return input.uppercase()
}

private fun validateData(input: String, data:String) {
        if (input.length < 4) throw ValidationException("")
}

private fun saveData(input: String) {
    if (input.length < 5) throw ValidationException("")
    //do some more processing
}
```

An real life example

```
private fun getData(input: String): String {
    if (input.length < 3) throw ValidationException("")
    return input.uppercase()
}
```

```
private fun getDataWithWrapper(input: String): Outcome<String> {
    return if (input.length < 3) Outcome.Error("Less than 3")
    else Outcome.Success(input.uppercase())
}
```

An real life example

```
private fun validateDataWithWrapper(input: String): Outcome<Unit> {
    return if (input.length < 4) Outcome.Error("less than 4")
    else Outcome.Success(Unit)
}


private fun saveDataWithWrapper(input: String): ProcessResult {
    return if (input.length < 5) SkipProcessing("less than 5") else {
        //more processing
        SuccessProcessing("")
    }
}
```

An real life example

```
fun processInputWithWrapper (input: String): ProcessResult {
    return when (val result = getDataWithWrapper(input)) {
        is Outcome.Error -> SkipProcessing( result.message)
        is Outcome.Success -> processFetchedData( result.value)
    }
}

private fun processFetchedData (input: String): ProcessResult {
    return when (val result = validateDataWithWrapper(input)) {
        is Outcome.Error -> SkipProcessing( result.message)
        is Outcome.Success -> saveDataWithWrapper(input)
    }
}
```

An real life example

```
@PostMapping
fun doSomeMagic(@RequestBody magicInput: MagicInput): ResponseEntity<Unit>
{
    return try {
        magicService.doMagic(magicInput)
        ResponseEntity.status(HttpStatus.CREATED).build()
    } catch (ex: ValidationException) {
        ResponseEntity.ok().build()
    }
}
```

An real life example

```
@PostMapping
fun doSomeMagic(@RequestBody magicInput: MagicInput): ResponseEntity<Unit>
{
    return when (val result = magicService.doMagic(magicInput)) {
        is SkipProcessing -> {
            logger.warning(result.message)
            ResponseEntity.status(OK).build()
        }
        is SuccessProcessing<*> -> ResponseEntity.status(CREATED).build()
    }
}
```

Benchmarks

Benchmark	Mode	Cnt	Score	Error	Units
ControllerBench.doNotThrowException	thrpt	5	6,101	± 0,181	ops/ms
ControllerBench.doThrowException	thrpt	5	0,167	± 0,003	ops/ms
ControllerBench.doThrowExceptionWithStack	thrpt	5	0,030	± 0,001	ops/ms
ControllerBench.doNotThrowException	avgt	5	0,182	± 0,004	ms/op
ControllerBench.doThrowException	avgt	5	6,027	± 0,214	ms/op
ControllerBench.doThrowExceptionWithStack	avgt	5	33,239	± 0,121	ms/op

The world of functional programming.

1. When programming functionally, we try to avoid exceptions at all costs.
2. Exceptions in functional code can lead to unpredictable behavior.
3. Functional programming languages have a number of classes dedicated to handling exceptions.

The world of functional programming.

Scala:

Option - is used to model an object that may not have a value (instead of null).

Try - similar to Option but used for defensive programming when we anticipate that an exception may occur.

Either - used in cases when we want to receive a value or information about why we did not receive the value. Either never returns both.

The world of functional programming - Option

Scala:

```
val name: Option[String]  = None

val name2: Option[String] = Some( "Value")

val name3: Option[String] =  Option(null)
```

The world of functional programming - Option

Pattern matching

Scala:

```
val opt = Option(null)
opt match {
 case None => println("Missing value")
 case Some(x) => println(s"Value is: ${x}")
}
```

The world of functional programming - Option

Kotlin approach - nullable type

Kotlin:

```
val data: Int? = try (10/0) catch(e:RuntimeException) { null }
when (data) {
   null -> println("Missing value. Reason: unknown" )
   else -> println("Value is: $data")
}
```

The world of functional programming - Try

Scala:

```
val tr: Try[Int] = Try(1 / 0)
tr match {
 case Failure(exception) => println(s"Invalid value:
{exception.getMessage} ")
 case Success(value) => println(s"Value is: ${value}")
}
```

The world of functional programming - Try

Kotlin does not have an equivalent of the Try class, but we can use the language syntax to build a similar solution

Kotlin:

```
val throwFunction: Result<Int> =  runCatching { 1 / 0 }

throwFunction
    .fold(
        onSuccess = { println("Result: $it") },
        onFailure = { println("Invalid value:${it.message}") }
    )
```

The world of functional programming - Either

Scala:

```
val l = Left("Invalid data")
val r = Right(1)

val data: Either[String, Int] = try Right(1 / 0) catch {
  case e: ArithmeticException => Left( "Divide by zero")
}

data match {
  case Left(value) => println(s"Error occurred: $value")
  case Right(value) => println(s"Result is: $value")
}
}
```

The world of functional programming - Either

Scala: for comprehensions (imperative look)

```scala
val first = Right(2)
val second = Right(2)
val third = Left("Wrong value"): Left[String, Int]

val result: Either[String, Int] = for {
 x <- first
 y <- second
 z <- third
} yield x + y + z

result match {
 case Left(value) => println(s"Something went wrong: $value")
 case Right(value) => println(s"Result is $value")
}
```

The world of functional programming - Either

Kotlin doesn't have a built-in Either type, but its syntax allows us to implement such a class very easily.

Sealed class/interface

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

The world of functional programming - Either

What we had at the beginning:

```
fun doMagic(input: String) {
    val data = getData(input)
    validateData(input, data)
    saveData(input)
}
```

The world of functional programming - Either

First refactor with sealed classes/interfaces:

```
fun processInput(input: String): ProcessResult {
    return when (val result = getDataWithWrapper(input)) {
        is Outcome.Error -> SkipProcessing(result.message)
        is Outcome.Success -> processFetchedData(result.value)
    }
}

private fun processFetchedData(input: String): ProcessResult {
    return when (val result = validateDataWithWrapper(input)) {
        is Outcome.Error -> SkipProcessing(result.message)
        is Outcome.Success -> saveData(input)
    }
}
private fun saveData(input: String): ProcessResult {
    return if (input.length < 5) SkipProcessing("less than 5") else {
        //more processing
        SuccessProcessing("")
    }
}
```

Can we refactor it once again?

```
fun processInput(input: String): EitherK<MyError, String> {
    return when (val result = getDataWithWrapper(input)) {
        is EitherK.Left-> result
        is EitherK.Right -> processFetchedData(result.value)
    }
}

private fun processFetchedData(input: String): EitherK<MyError, String>{
    return when (val result = validateDataWithWrapper(input)) {
        is EitherK.Left -> result
        is EitherK.Right -> saveData(input)
    }
}
private fun saveData(input: String): EitherK<MyError, String>{
    return if (input.length < 5) EitherK.Left(MyError("less than 5"))
            else {
        //more processing
        EitherK.Right("")
    }
}
```

Wait... nothing has changed

Monads - oh no...

```
inline fun <A, B, C> EitherK<A, B>.map(fn: (B) -> C) = when (this) {
    is EitherK.Left -> this
    is EitherK.Right -> EitherK.Right(fn( this.value))
}

inline fun <A, B, C> EitherK<A, B>.flatMap(fn: (B) -> EitherK<A, C>) = when
(this) {
    is EitherK.Left -> this
    is EitherK.Right -> fn( this.value)
}
```

The world of functional programming - Either

```
fun processInput(input: String): EitherK<MyError, String> {
    return getData(input).map { data ->
        validateData(data).map { _ ->
            saveData(data)
                error
                expected EitherK<MyError, String>
                found EitherK<MyError, EitherK<MyError, EitherK<MyError, String>>>
        }
    }
}
```

The world of functional programming - Either

```
fun processInput(input: String): EitherK<MyError, String> {
    return getData(input).flatMap { data ->
        validateData(data).flatMap { _ ->
            saveData(data)
        }
    }
}
fun getData(input: String): EitherK<MyError, String> = TODO()
fun validateData(data: String): EitherK<MyError, Unit> = TODO()
fun saveData(data: String): EitherK<MyError, String> = TODO()
```

arrow-kt to the rescue

```
fun processInput(input: String): Either<MyError, String> {
    return getData(input).flatMap { data ->
        validateData(data).flatMap { _ ->
            saveData(data)
        }
    }
}
fun getData(input: String): Either<MyError , String> = TODO()
fun validateData(data: String): Either<MyError, Unit> = TODO()
fun saveData(data: String): Either<MyError, String> = TODO()
```

https://old.arrow-kt.io/docs/patterns/error_handling/

arrow-kt to the rescue

```
fun processInput(input: String): Either<MyError , String> {
    return either {
        val data = getData(input).bind()
        validateData(data).bind()
        saveData(data).bind()
    }
}
```

https://old.arrow-kt.io/docs/patterns/error_handling/

arrow-kt - builders

```kotlin
private suspend fun getData(input: String): Either<MyError, String> = either {
    ensure(input.length > 3) { MyError("input is less than 3") }
    input.uppercase()
}

private suspend fun validateData(input: String, data: String) = either {
    ensure(input.length > 4) { MyError("input is less than 4") }
    ensure(data.length > 4) { MyError("input is less than 4") }
}

private suspend fun saveData(input: String) =
    if (input.length <= 5) {
        MyError("input is less or equal 5").left()
    } else {
        input.right()
    }
```

arrow-kt - builders

```
suspend fun main() {
    when (val result = doSomeValidation("input")) {
        is Either.Left -> println("Some error: ${result.value.message}")
        is Either.Right -> println("Valid result: ${result.value}")
    }
}

private suspend fun doSomeValidation(input: String): Either<MyError, String> = either {
    val data = getData(input).bind()
    validateData(input, data).bind()
    saveData(data).bind()
}
```

arrow-kt - validation

```
data class Person(val name: String, val age: Int)

fun validateName(name: String): Either<String, String> {
    return either {
        ensure(name.length > 3) { "Smaller than 3" }
        name.uppercase()
    }
}

fun validateAge(age: Int): Either<String, Int> {
    return either {
        ensure(age > 18) { "Smaller than 18" }
        age
    }
}
```

arrow-kt - validation

```
fun validate(name: String, age: Int): Either<Nel<String>, Person> =  either {
    zipOrAccumulate(
        { validateName(name).bind() },
        { validateAge(age).bind() }
    ) { name, age ->
        Person(name, age)
    }
}

val result = validationSerivice.validate("jo", 12)
result.fold(
    { println("Error: $it") },
    { println("Success: $it") })

Error: NonEmptyList(Smaller than 5, Smaller than 18)
```

arrow-kt - a little piece of magic

```
fun parse(listOf: List<String>): Either<MyError, List<Int>> {
    return either {
        listOf.map {
            parseNumber(it)
        }
    }
}

fun parseWithEither(listOf: List<String>): Either<List<MyError>, List<Int>> {
    return listOf.mapOrAccumulate {
        parseNumber(it)
    }
}

fun Raise<MyError>.parseNumber(i: String): Int {
    return try {
        i.toInt()
    } catch (ex: NumberFormatException) {
        raise(MyError("Not a number"))
    }
}
```