# COMPOSABLE ERROR HANDLING

## STOJAN ANASTASOV

@s_anastasov

# FUNCTIONAL ERROR HANDLING

STOJAN ANASTASOV

@s_anastasov

## SOLVING PROBLEMS

As developers we solve complex problems.

## ANDROID ACTIVITY LIFECYCLE

## SOLVING COMPLEX PROBLEMS

## SOLVING COMPLEX PROBLEMS

- Split problem into smaller problems

## SOLVING COMPLEX PROBLEMS

- Split problem into smaller problems
- Write code solving the small problems

## SOLVING COMPLEX PROBLEMS

- Split problem into smaller problems
- Write code solving the small problems
- Combine the solutions of the small problems

## DATA VALIDATION

The problem of user sign up with data:

- Email
- First Name
- Last Name
- Date of Birth

## VALIDATION RULES

- Email must contain @
- First Name and Last Name can't be blank. Max length 50 (DB limit).
- Date of Birth must be formatted as YYYY-MM-DD

## THE DTO

```kotlin
data class UserDto(
    val email: String?,
    val firstName: String?,
    val lastName: String?,
    val dateOfBirth: String?
)
```

## THE DTO

```kotlin
data class UserDto(
    val email: String?,
    val firstName: String?,
    val lastName: String?,
    val dateOfBirth: String?
)
```

*Postel's law*: Be conservative in what you do, be liberal in what you accept from others.

## THE DOMAIN

## THE DOMAIN

```kotlin
data class Email(val email: String) { companion object }
```

## THE DOMAIN

```kotlin
data class Email(val email: String) { companion object }
```

```kotlin
data class String50(val value: String) { companion object }
```

# THE DOMAIN

```kotlin
data class Email(val email: String) { companion object }
```

```kotlin
data class String50(val value: String) { companion object }
```

```kotlin
import java.time.LocalDate

data class User(
    val email: Email,
    val firstName: String50,
    val lastName: String50,
    val dateOfBirth: LocalDate
) { companion object }
```

## VALIDATING EMAIL

Email must contain @

```
fun validateEmail(email: String?): Boolean =
    email != null && email.contains('@')
```

## VALIDATING FIRST/LAST NAME

First Name and Last Name can't be blank. Max length 50 (DB limit).

```kotlin
fun validateName(name: String?): Boolean =
    !name.isNullOrBlank() && name.length < 50
```

## VALIDATING DOB

Date of Birth must be formatted as YYYY-MM-DD

```
import java.time.LocalDate
import java.time.format.DateTimeParseException

fun validateDateOfBirth(dob: String?): Boolean =
    try {
        LocalDate.parse(dob)
        true
    } catch (e: DateTimeParseException) {
        false
    }
```

## VALIDATING USER

```
import java.time.LocalDate
import java.time.format.DateTimeParseException

fun validateUser(
    email: String?,
    firstName: String?,
    lastName: String?,
    dob: String?
): Boolean = validateEmail(email)
        && validateName(firstName)
        && validateName(lastName)
        && validateDateOfBirth(dob)
```

## USAGE (A) -> BOOLEAN

```
validateUser("stojan", null, "", "August")
// false
```

## USAGE (A) -> BOOLEAN

```
validateUser("stojan", null, "", "August")
// false
```

## BOOLEANS

- Composes well
- Bad error messages

Boolean -> True | False

## EXCEPTIONS

## EXCEPTIONS

```
fun validateEmailBool(email: String?): Boolean {
    require(email != null && email.contains('@'))
    { "Email must contain @, found: '$email'" }
    return true
}
```

IllegalArgumentException if the predicate is false

## RETURN VALUE IS ALWAYS TRUE

```kotlin
fun validateEmailUnit(email: String?): Unit =
    require(email != null && email.contains('@'))
    { "Email must contain @, found: '$email'" }

fun validateNameUnit(name: String?): Unit =
    require(!name.isNullOrBlank() && name.length < 50)
    { "Name must be between 1 and 50 chars, found: '$name'" }

fun validateDateOfBirthUnit(dob: String?): Unit {
    LocalDate.parse(dob)
}
```

## COMPOSING

```
fun validateUserUnit(
    email: String?,
    firstName: String?,
    lastName: String?,
    dob: String?
): Unit {
    validateEmailUnit(email)
    validateNameUnit(firstName)
    validateNameUnit(lastName)
    validateDateOfBirthUnit(dob)
}
```

## USAGE (A) -> UNIT + EXCEPTION

```
validateUserUnit("stolea@gmail.com", "Stojan", "An", "1995-10-10"
```

## USAGE (A) -> UNIT + EXCEPTION

```
validateUserUnit("stolea@gmail.com", "Stojan", "An", "1995-10-10"
```

```
val result = try {
    validateUserUnit("stojan", null, "", "August")
    "Valid"
} catch (e: Exception) {
    e.message!!
}
result
// Email must contain @, found: 'stojan'
```

We only get the first error!

# ACCUMULATE ERRORS

```kotlin
fun validateUserAccumulateErrors(
    email: String?,
    firstName: String?,
    lastName: String?,
    dob: String?
): Unit {
    val errors = mutableListOf<String>()

    try {
        validateEmailUnit(email)
    } catch (e: IllegalArgumentException) {
        errors.add(e.message!!)
    }

    // TODO: firstName, lastName, dob
```

Glue code to the MAX!

## PROBLEMS WITH EXCEPTIONS

- Boilerplate code
- Throwing Exceptions is expensive on JVM
- Dos not fit on a slide

## ERRORS AS VALUES

```
typealias ErrorMsg = String
```

## ERRORS AS VALUES

```
typealias ErrorMsg = String
```

```
fun validateEmail(email: String?): ErrorMsg? =
    if (email != null && email.contains('@')) null
    else "Email must contain @, found: '$email'"
```

# ERRORS AS VALUES

```
typealias ErrorMsg = String
```

```
fun validateEmail(email: String?): ErrorMsg? =
    if (email != null && email.contains('@')) null
    else "Email must contain @, found: '$email'"
```

```
fun validateName(name: String?): ErrorMsg? =
    if (!name.isNullOrBlank() && name.length < 50) null
    else "Name must be between 1 and 50 chars, found: '$name'"

fun validateDateOfBirth(dob: String?): ErrorMsg? = TODO()
```

## COMPOSING VALUES

```
fun validateUser(
    email: String?,
    firstName: String?,
    lastName: String?,
    dob: String?
): ErrorMsg? {
    val errorMsg = listOfNotNull(
        validateEmail(email),
        validateName(firstName),
        validateName(lastName),
        validateDateOfBirth(dob)
    ).joinToString()
    return if (errorMsg.isEmpty()) null else errorMsg
}
```

## ERRORMSG

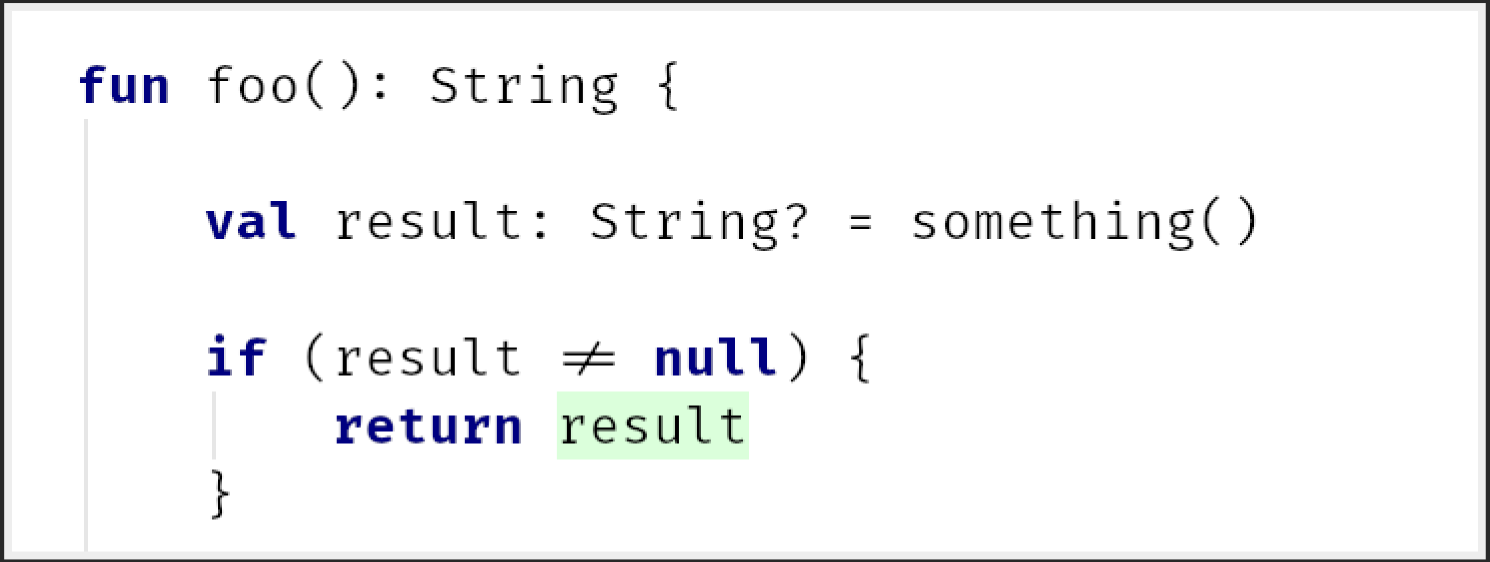
- Composable
- Good error messages
- Developer friendly ?

## ERROR PRONE

```
val email: String? = "stolea@gmail.com"
val emailErr: ErrorMsg? = validateEmail(email)
if (emailErr == null) {
    Email(email!!) // <-- Error prone
}
```

validateEmail already does a null check

## SMART CAST



```
fun foo(): String {

    val result: String? = something()

    if (result ≠ null) {
        return result
    }
```

## CAN WE DO BETTER

## VALRES

```kotlin
sealed class ValRes<out E, out A> {
    data class Valid<A>(val a: A) : ValRes<Nothing, A>()
    data class Invalid<E>(val e: E) : ValRes<E, Nothing>()
}
```

# VALRES

```
sealed class ValRes<out E, out A> {
    data class Valid<A>(val a: A) : ValRes<Nothing, A>()
    data class Invalid<E>(val e: E) : ValRes<E, Nothing>()
}
```

```
fun <A> valid(a: A): ValRes<Nothing, A> = ValRes.Valid(a)

fun <E> invalid(e: E): ValRes<E, Nothing> = ValRes.Invalid(e)
```

## VALRES IN THE SMALL

```kotlin
fun validateEmail(email: String?): ValRes<String, Email> =
    if (email != null && email.contains('@')) valid(Email(email))
    else invalid("Email must contain @, found: '$email'")
```

## VALRES IN THE SMALL

```
fun validateEmail(email: String?): ValRes<String, Email> =
    if (email != null && email.contains('@')) valid(Email(email))
    else invalid("Email must contain @, found: '$email'")
```

```
fun validateName(name: String?): ValRes<String, String50> =
    if (!name.isNullOrBlank() && name.length < 50) valid(String50
    else invalid("Name must be between 1 and 50 chars, found: '$n
```

# VALRES IN THE SMALL

```
fun validateEmail(email: String?): ValRes<String, Email> =
    if (email != null && email.contains('@')) valid(Email(email))
    else invalid("Email must contain @, found: '$email'")
```

```
fun validateName(name: String?): ValRes<String, String50> =
    if (!name.isNullOrBlank() && name.length < 50) valid(String50
    else invalid("Name must be between 1 and 50 chars, found: '$n
```

```
fun validateDateOfBirth(dob: String?): ValRes<String, LocalDate>
```

# COMBINING VALRES

```
typealias Valid<A> = ValRes.Valid<A>
typealias Invalid<E> = ValRes.Invalid<E>

fun <E, A, B> tupled(
            a: ValRes<E, A>,
            b: ValRes<E, B>,
            combine: (E, E) -> E
        ): ValRes<E, Pair<A, B>> =
            if (a is Valid && b is Valid) valid(Pair(a.a, b.a))
            else if (a is Invalid && b is Invalid) invalid(combin
            else if (a is Invalid) invalid(a.e)
            else if (b is Invalid) invalid(b.e)
            else throw IllegalStateException("This is impossible"
```

## USAGE

```
validateEmail("stolea@gmail.com")
// Valid(a=Email(email=stolea@gmail.com))
```

## USAGE

```
validateEmail("stolea@gmail.com")
// Valid(a=Email(email=stolea@gmail.com))
```

```
validateEmail("email")
// Invalid(e=Email must contain @, found: 'email')
```

## USAGE

```
validateEmail("stolea@gmail.com")
// Valid(a=Email(email=stolea@gmail.com))
```

```
validateEmail("email")
// Invalid(e=Email must contain @, found: 'email')
```

```
tupled(
    validateEmail("stojan"),    //invalid
    validateName(null)          //invalid
) { e1, e2 -> "$e1, $e2" }
// Invalid(e=Email must contain @, found: 'stojan', Name must be
```

```kotlin
data class Triple<A, B, C>(val a: A, val b: B, val c: C)

fun <E, A, B, C> tupled(
            a: ValRes<E, A>,
            b: ValRes<E, B>,
            c: ValRes<E, C>,
            combine: (E, E) -> E
        ): ValRes<E, Triple<A, B, C>> = TODO()
```

## ARROW-KT

Functional companion to Kotlin's Standard Library

arrow-kt.io

## VALIDATED

```kotlin
sealed class Validated<out E, out A> {
    data class Valid<out A>(val a: A) : Validated<Nothing, A>()
    data class Invalid<out E>(val e: E) : Validated<E, Nothing>()
}
```

## VALIDATED

```
sealed class Validated<out E, out A> {
    data class Valid<out A>(val a: A) : Validated<Nothing, A>()
    data class Invalid<out E>(val e: E) : Validated<E, Nothing>()
}
```

```
import arrow.core.*

typealias ValidatedNel<E, A> = Validated<Nel<E>, A>
```

## VALIDATIONRESULT

```
typealias ValidationResult<A> = ValidatedNel<String, A>
```

## VALIDATIONRESULT

```
typealias ValidationResult<A> = ValidatedNel<String, A>
```

```
fun Email.Companion.create(email: String?): ValidationResult<Emai
    if (email != null && email.contains('@')) Email(email).valid(
    else "Email must contain @, found: '$email'".invalidNel()
```

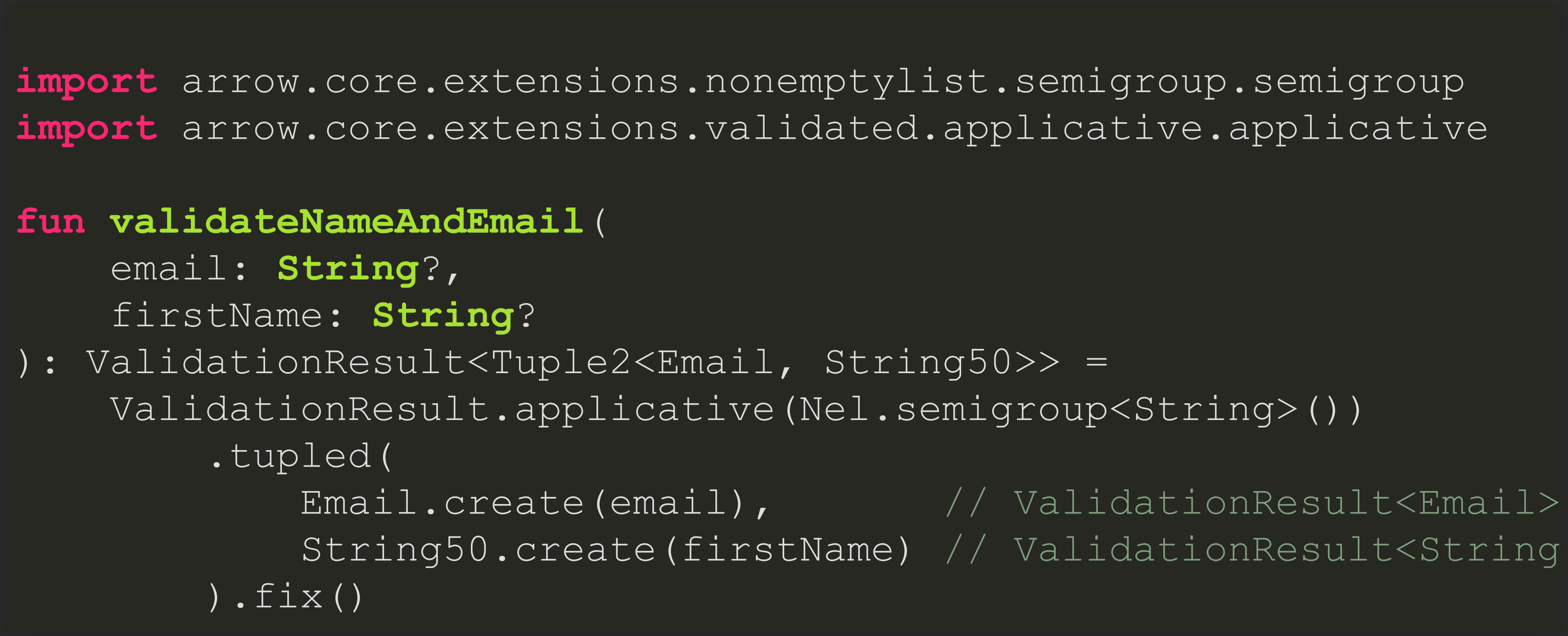
## VALIDATING IN THE SMALL

```
fun String50.Companion.create(name: String?): ValidationResult<St
    if (!name.isNullOrBlank() && name.length < 50) String50(name)
    else "Name must be between 1 and 50 chars, found: '$name'".in
```

# VALIDATING IN THE SMALL

```
fun String50.Companion.create(name: String?): ValidationResult<St
    if (!name.isNullOrBlank() && name.length < 50) String50(name)
    else "Name must be between 1 and 50 chars, found: '$name'".in
```

```
fun validateDateOfBirth(dob: String?): ValidationResult<LocalDate
    try {
        LocalDate.parse(dob).valid()
    } catch (e: DateTimeParseException) {
        "Date of Birth must be a valid date, found: '$dob'".inval
    }
```



```kotlin
import arrow.core.extensions.nonemptylist.semigroup.semigroup
import arrow.core.extensions.validated.applicative.applicative

fun validateNameAndEmail(
    email: String?,
    firstName: String?
): ValidationResult<Tuple2<Email, String50>> =
    ValidationResult.applicative(Nel.semigroup<String>())
        .tupled(
            Email.create(email),        // ValidationResult<Email>
            String50.create(firstName) // ValidationResult<String
        ).fix()
```

## APPLICATIVE

Combine values from multiple independent computations that can potentially fail.

`tupled` 2-X arguments returns `Tuple2`-`TupleX`

## SEMIGROUP

A semigroup for some given type A has a single operation (which we will call combine), which takes two values of type A, and returns a value of type A. This operation must be guaranteed to be associative.

```
interface Semigroup<A> {

    fun combine(a: A, b: A): A
}
```

## VALIDATING NAME AND EMAIL

```
validateNameAndEmail("stolea@gmail.com", "Stojan")
// Valid(a=Tuple2(a=Email(email=stolea@gmail.com), b=String50(val
```

## VALIDATING NAME AND EMAIL

```
validateNameAndEmail("stolea@gmail.com", "Stojan")
// Valid(a=Tuple2(a=Email(email=stolea@gmail.com), b=String50(val
```

```
validateNameAndEmail("Not an email", "     ")
// Invalid(e=NonEmptyList(all=[Email must contain @, found: 'Not
```

```kotlin
import arrow.core.extensions.nonemptylist.semigroup.semigroup
import arrow.core.extensions.validated.applicative.applicative

fun User.Companion.create(
    email: String?,
    firstName: String?,
    lastName: String?,
    dob: String?
): ValidationResult<User> =
    ValidationResult.applicative(Nel.semigroup<String>())
        .tupled(
            Email.create(email),
            String50.create(firstName),
            String50.create(lastName),
            validateDateOfBirth(dob)
```

## CREATE USER (VALID)

```
User.create(
     email = "stolea@gmail.com",
     firstName = "Stojan",
     lastName = "Anastasov",
     dob = "1991-10-10"
)
// Valid(a=User(email=Email(email=stolea@gmail.com), firstName=St
```

## CREATE USER (INVALID)

```
User.create(
    email = "",
    firstName = "   ",
    lastName = "a",
    dob = "10.10.1992"
)
// Invalid(e=NonEmptyList(all=[Date of Birth must be a valid date
```

# EXTRACTING THE VALUE

```
val user = User.create(
    email = "stolea@gmail.com",
    firstName = "Stojan",
    lastName = "Anastasov",
    dob = "1991-10-10"
)

user.fold(
    { e: Nel<String> -> TODO() },
    { validUser: User -> validUser }
)
// User(email=Email(email=stolea@gmail.com), firstName=String50(v
```

## BENEFITS OF VALIDATIONRESULT

- It composes well
- Good error messages
- Developer friendly
- Composition functions are written and tested

## LEARN MORE

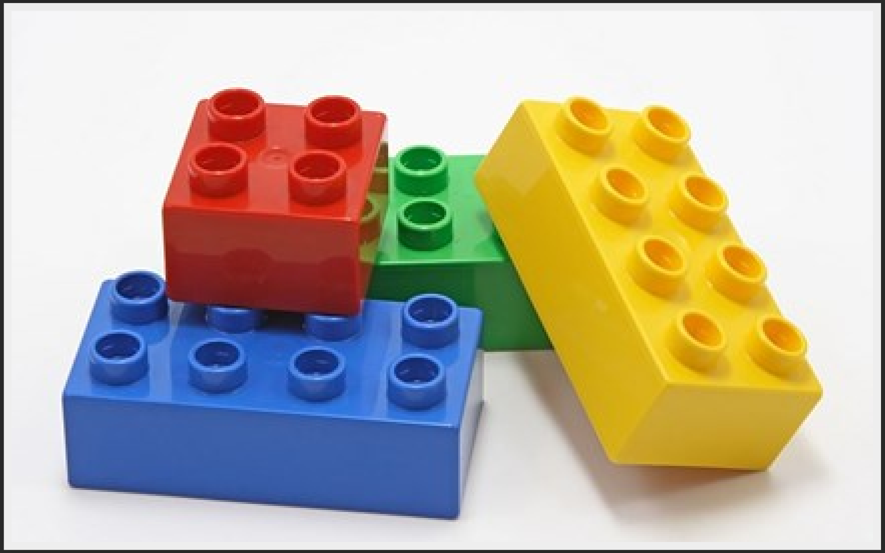
arrow-kt.io

Patterns -> Error Handling

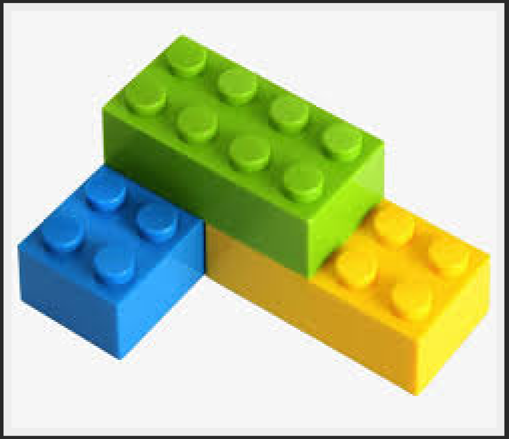
## COMPOSITION

How do things compose

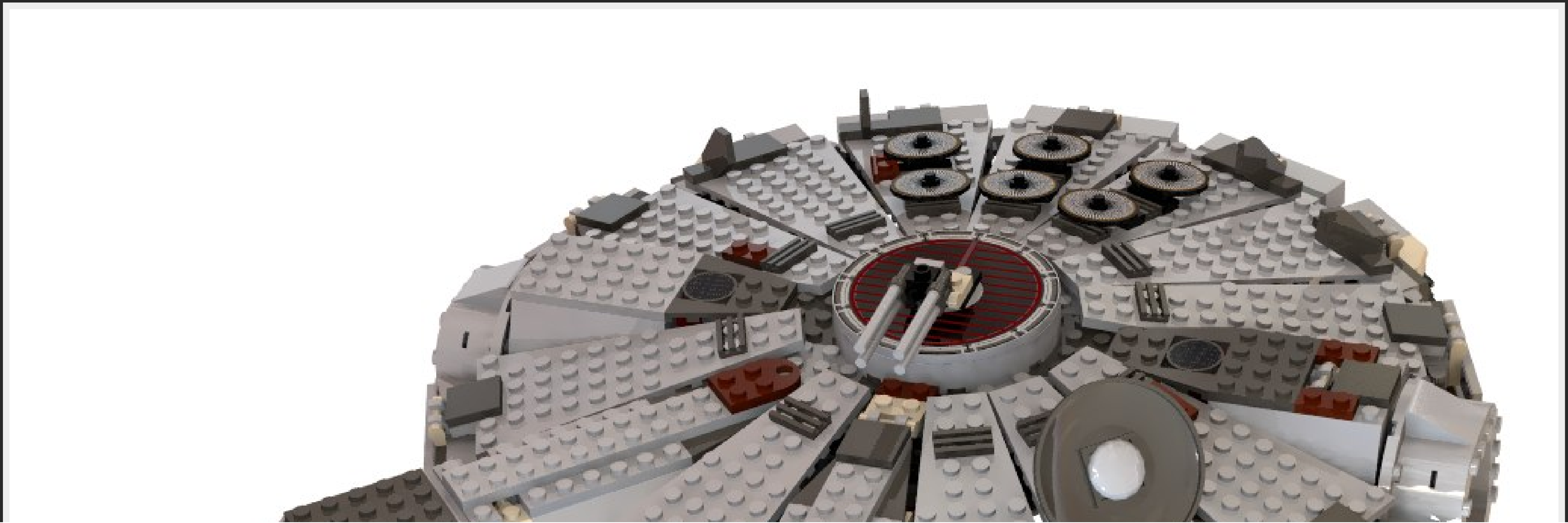
## LEGO BLOCK



## LEGO COMBINED



## LEGO FALCON



## COMBINING FUNCTIONS

```kotlin
// (String?) -> ValidationResult<Email>
fun validateEmail(email: String?): ValidationResult<Email> = TODO
```

## COMBINING FUNCTIONS

```
// (String?) -> ValidationResult<Email>
fun validateEmail(email: String?): ValidationResult<Email> = TODO
```

```
// (String?, String?, String?, String?) -> ValidationResult<User>
fun validateUser(
    email: String?,
    firstName: String?,
    lastName: String?,
    dob: String?): ValidationResult<User> = TODO()
```

## SUMMARY

We can validate data using different technics

- (A) -> Boolean

## SUMMARY

We can validate data using different technics

- (A) -> Boolean
- (A) -> Unit + Exceptions

## SUMMARY

We can validate data using different technics

- (A) -> Boolean
- (A) -> Unit + Exceptions
- (A) -> ErrorMsg?

## SUMMARY

We can validate data using different technics

- (A) -> Boolean
- (A) -> Unit + Exceptions
- (A) -> ErrorMsg?
- (A) -> ValRes

## SUMMARY

We can validate data using different technics

- (A) -> Boolean
- (A) -> Unit + Exceptions
- (A) -> ErrorMsg?
- (A) -> ValRes
- (A) -> Validated (arrow-kt)

## THANK YOU

### QUESTIONS