Taming Complexity: Functional Error Handling with Arrow.kt

An impromptu webinar around fascination

Peter Pilgrim, Java Champion, JAVA TECH COACH
November 2025

EVERYONE OF US, WE WANT SUCCESS!

Thanks to **Priscilla Du Preez** for sharing their work on Unsplash.
https://unsplash.com/@priscilladupreez

GET IN TOUCH WITH PETER

Email:	peter.pilgrim@gmail.com
Website:	https://www.xenonique.co.uk/
LinkedIn:	https://www.linkedin.com/in/peterpilgrim2000/
Twitter/X:	**@peter_pilgrim**
Bluesky:	https://bsky.app/profile/peterpilgrim.bsky.social
Mastodon:	https://techhub.social/@peter_pilgrim
Github:	https://github.com/peterpilgrim/

Remote work in **Milton Keynes**, Bucks, England
Approaches about Outside IR35 Contracts UK appreciated
Timezone: GMT (BST)

Welcome to the webinar

- Beyond Imperative
  - Curious Java Developer
  - Heard about FP blocked by intimidation or eco-system
- For the Problem-Solver developer
  - How Arrow.kt can make your Kotlin code safer, cleaner and fun!
- Direct and Clear
  - Moving from Java to Kotlin and achieving clarity

Setting Up and Tools

- JDK 24+, Kotlin 2.2
- Gradle, SDKman
- IntelliJ IDEA (or preferred IDE)
- GitHub repo (starter code provided)

Kotlin - Language Design & Expressiveness

Aspect	**Kotlin**	**Java 24 beyond**
Boilerplate (WET, DRY)	Much less - data classes, named parameters, default parameters, type inference	Verbose still, records & var and simpler main() are welcome improvements
Null Safety	Build-in via nullable types (String?)	Optional, but still runtime risk
Extension Functions	Extended existing classes cleanly without inheritance	Not supported directly - workaround sealed records (see Venkat S)
Smart Casts	Automatic type inference after compiler checks	Partial support via pattern matching incoming, deconstruction pending
Functional Style	Lambdas, map/filter, coroutines, inline functions	Streams & Lambdas, no coroutines, however virtual threads

Kotlin - Concurrency & Async

Aspect	**Kotlin**	**Java 24 beyond**
Coroutines	Lightweight async model, structured concurrency	Project Loom (Virtual Threads) - more recent, however syntax is still heavy
Suspend Functions	Native async semantics without callback	Project Loom helps but there is no suspension and code still looks imperative in nature
Ecosystem Support	Mature coroutine libraries exists in 2025: Spring, Ktor and Reactor	Project Loom is still new to consider - however Kotlin may still leverage virtual threads in the future

Interoperability

- Kotlin is 100% JVM-compatible
  - You can mix Kotlin and Java code in the same project
- Allows developer to migrate Java microservices to Kotlin
- The Kotlin compiler produces normal `.class` files;
  - Ergo: ✅ Zero lock-in - start with Kotlin incrementally

Arrow.kt

- Arrow is a collection of self-contained Kotlin libraries
- Each library either
  - extends or improves single common-used Kotlin library
  - enhances a Kotlin language feature
  - focuses on tasks or work item
- Latest working version is Arrow-kt 2.1.2 => 2.2.0 (October 2025)
  - Some libraries in Arrow-kt 1.x are merged together / replaced
  - Read the migration upgrade notes
  - At the time of writing, Kotlin does not support Java 25 byte codes!

Overview of Arrow.kt Libraries

Library	Feature
`arrow-core`	Extends the feature of Kotlin’s standard library Raise, Either, Ior and Option
`arrow-fx-coroutines`	High-level concurrency and resource management
`arrow-optics`	Utilities for working with immutable data
`arrow-resilience`	Application service resilience patterns
`arrow-fx-stm`	Software Transactional Memory (STM)

Let’s Go to The Code

Problematic Java-Style Code

```
fun sqrtFromString(str: String): Double {
    val number = str.toDouble()
    return sqrt(number)
}
```

What else do you see?

Problematic Java-Style Code (1-Line Version)

```
fun sqrtFromString(str: String)
    = sqrt( str.toDouble )

// Double return type is inferred
```

What else do you see?

Problematic Java-Style Code

```
fun sqrtFromString(str: String): Double {
    val number = str.toDouble()
        // Throws NumberFormatException!
    return sqrt(number)
        // Fails for negative numbers!
}
```

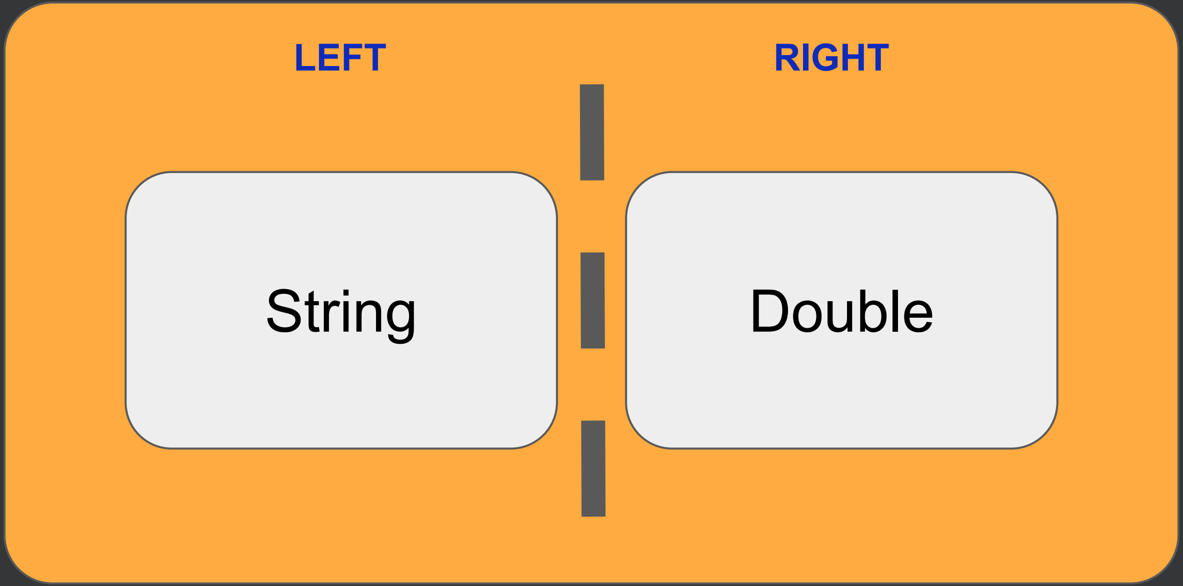
Kotlin Improvement (Nullable Types)

```
fun kotlinSqrtFromString(str: String): Double? {
    val number = str.toDoubleOrNull()
          ?: return null
    return if (number >= 0)
          sqrt(number) else null
}
```

Arrow.kt Solution (Explicit Error Handling)

```
fun arrowEitherSqrt(str: String):
    Either<String, Double> = either {
     val number = str.toDoubleOrNull() ?:
       raise("'str' is not a valid number')
     ensure(number >= 0) {
       "Cannot calculate square root of negative number:
    $number" }
     sqrt(number)
}
```

Either<E, A>



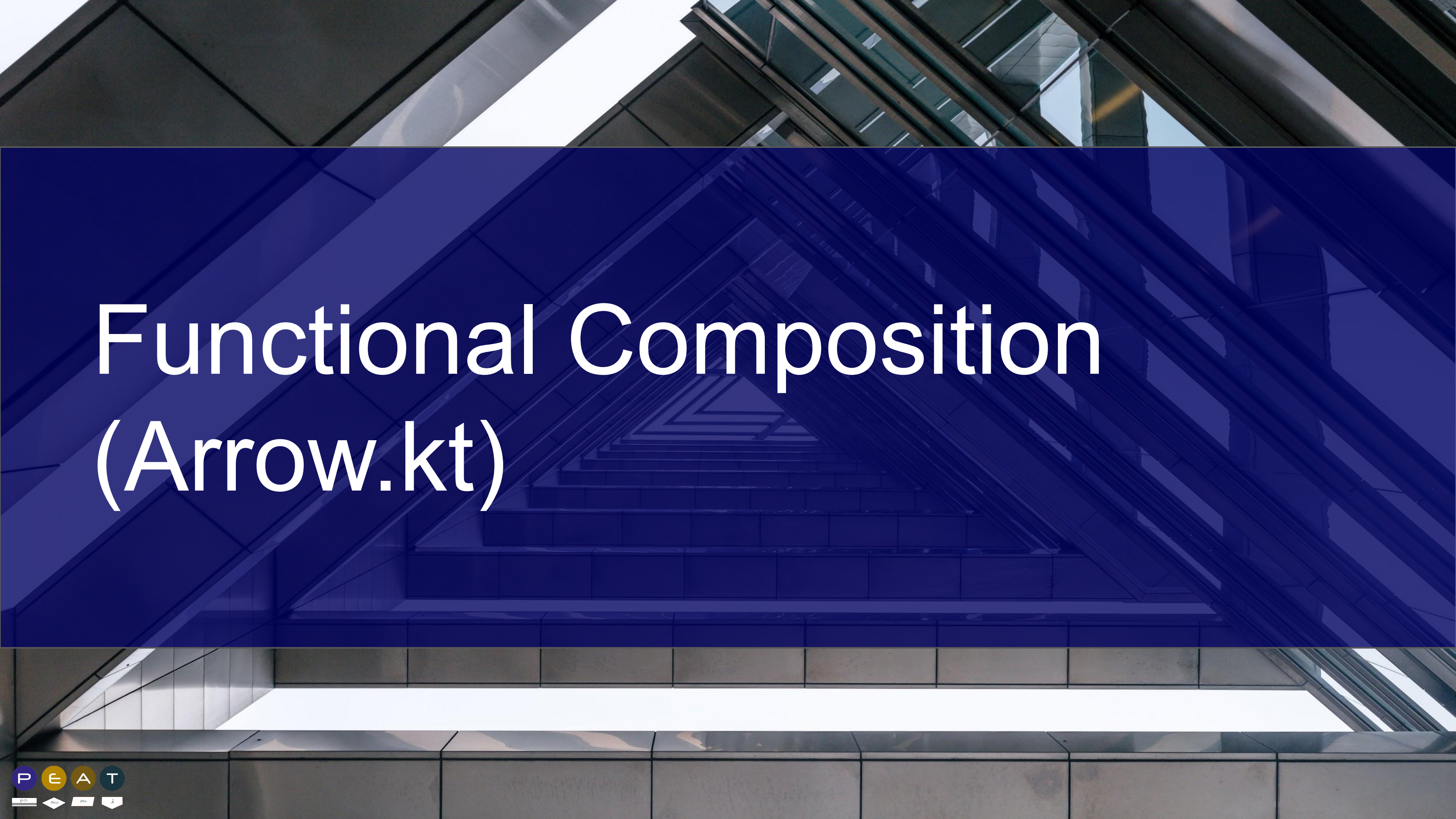
Second Function - Sealed Types - Arrow.kt

```
// potential error inputs

sealed interface ErrorValue {
    object InvalidNumber: ErrorValue
    object NegativeValue: ErrorValue
}
```

Second Function - Formatting Detailed #2 - Arrow.kt

```kotlin
fun formatResultDetailed(result: Either<ErrorValue, Double>): String =
result.fold(
    ifLeft = { error ->
        when {
            InvalidNumber -> "🔢 Input error: $error"
            NegativeValue -> "📉 Math error: $error"
            else -> "⚠️ Unknown error: $error"
        }
    },
    ifRight = { value ->
        "🎯 Result: ${"%.4f".format(value)}"
    }
)
```



Functional Composition (Arrow.kt)

Let’s Compose #1 - Arrow.kt

```kotlin
fun parseNumber(str: String): Either<String, Double> = either {
    str.toDouble() ?: raise("'$str' is not a valid number")
}

fun validatePositive(number: Double): Either<String, Double> = either {
    if (number >= 0) number else raise("Number must be positive: $number")
}

fun calculateSqrt(number: Double): Either<String, Double> = either {
    sqrt(number)
}
```

Let’s Compose #2 - Arrow.kt

```kotlin
fun formatResult(result: Either<String, Double>): String =
  result.fold(
    ifLeft = { error -> "❌ Error: $error" },
    ifRight = { value -> "✅ Result: ${"%.3f".format(value)}" }
)
```

Let’s Compose #3 - Arrow.kt

```
fun processAndFormat(str: String): String = either {
    val num = parseNumber(str).bind()
    val positiveNum = validatePositive(num).bind()
    calculateSqrt(positiveNum).bind()
}.let {
    formatResult(it)  // Format the final result
}
```

What is this “Let” function?

- Let is a Scope function that executes a block of code on the same object using a temporary scope.
- Let is example of Scope function in Kotlin.

```
CustomerAccount("Andrew","Clarke","123S8642").let {
    val amount = GasSupplier.standingOrder(1)
    it.balanceCheckAndThenDirectDebit( amount)
    it.auditCheck()
}
```

- Kotlin standard library contains several functions (`let`, `run`, `with`, `apply` and `also`), which allow a executable block of code to operate on an object context

Ior<E, A> ?

- Understanding flow logic in an `Either<E, A>` block
  - Execute each line, `bind()` a `Left` or find `raise()`, stop and return that value
  - At end of the block, wrap the result in a `Right`.
- `Ior<E, A>` provides a third option,
  - A special type `Both` to represent potential errors
  - `Ior` is rarely used

Ior<E, A> - Advanced Example #1

```
object SimpleIorDemo {
    // Simple case with String warnings (easier to combine)
    fun processData(input: String): Ior<String, Int> = ior(String::plus) {
        val warnings = StringBuilder()
        val step1 = if (input.length < 5) {
            warnings.append("Input too short. ")
            input.padEnd(5, 'X')
        } else { input }

        val step2 = if (!step1.contains("data", ignoreCase = true)) {
            warnings.append("Missing 'data' keyword. ")
            "data_$step1"
        } else { step1 }

        (   if (warnings.isNotEmpty()) {
            Ior.Both(warnings.toString(), step2.length)
           } else {
            Ior.Right(step2.length)
           } ).bind()
    }
// . . .
```

Ior<E, A> - Advanced Example #2

```
object SimpleIorDemo {
    fun processData(input: String): Ior<String, Int> = ior(String::plus) ...

    fun demonstrate() {
        println("=== Simple Ior Example ===")

        listOf("hi", "datatest", "hello", "Data").forEach { input ->
            val result = processData(input)
            println("\nInput: '$input' -> $result")

            result.fold(
                { warn ->     println(" ⚠️ Warnings only: $warn") },
                { success -> println(" ✅ Pure success: $success") },
                { warn, success -> println(" ✅ Success: $success with ⚠️ warnings: $warn") }
            )
        }
    }
}
```

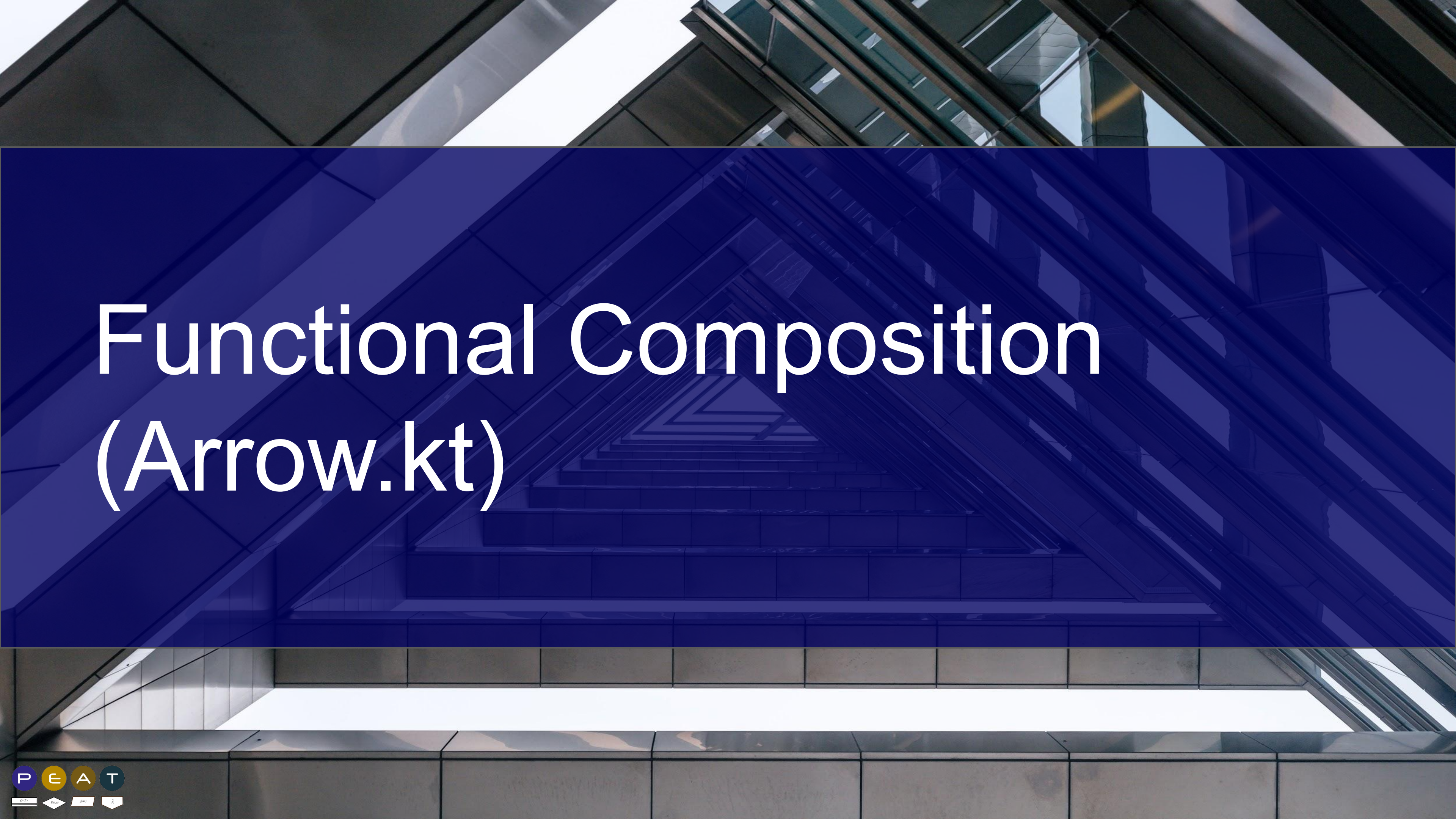
Executive Summary

Wrap-Up & Next Steps

- Recap: Introduction Arrow Core library
- Resources: Typed Errors
- Share code repo & cheat sheet

Questions & Answers

- Questions
- Discussion
- Feedback



Functional Composition (Arrow.kt)

Second Function - Formatting Simple - Arrow.kt

```
fun formatResult(result: Either<String, Double>):
    String = result.fold(
     ifLeft =
        { error -> "❌ Calculation failed: $error" },
     ifRight =
        { value -> "✅ Success! Square root is:
    ${"%.2f".format(value)}" }
)
```

Second Function - Formatting Detailed - Arrow.kt

```kotlin
fun formatResultDetailed(result: Either<String, Double>): String =
result.fold(
    ifLeft = { error ->
        when {
            error.contains("not a valid number") -> "🔢 Input error:
$error"
            error.contains("negative") -> "📉 Math error: $error"
            else -> "⚠️ Unknown error: $error"
        }
    },
    ifRight = { value ->
        "🎯 Result: ${"%.4f".format(value)}"
    }
)
```