Result Implementation for Kotlin 1.5

The `Result` class can be seen as specialized version of the `Either` monad. This monad by conventions allows the caller of function to return either the actual result, or the error. By convention this monad is declared (in general) terms:

```
abstract class Either<out Left,out Right>
```

By conventions the left side (denoted by `Left`) indicates the error value, followed by the right side for the result. Only one value can be present. Hence the 'either' moniker.

Design considerations of the Result library

This project is an experiment to implement a more fluent and natural implementation of the `Result` pattern by:

1. Introducing errors handling to a domain in an explicit and deliberate way.
2. Providing a rich `Result` type which can be returned by APIs and consumed by callers.
3. Providing a procedural (non Object Orientated) style of handling exceptions(and errors).
4. Provide a more rigorous handling of errors from a functional perspective.
5. At the same time not forcing developers which is more comfortable with the try-catch style of handling exceptions to adopt a new functional style.

Why Not ...?

Both Java and Kotlin has some mechanisms which can be exploited to handle error conditions in an API serfuce. Lets explore some reasons of why neither of these standard library API fails to deliver.

Use `kotlin.Result`

Java World

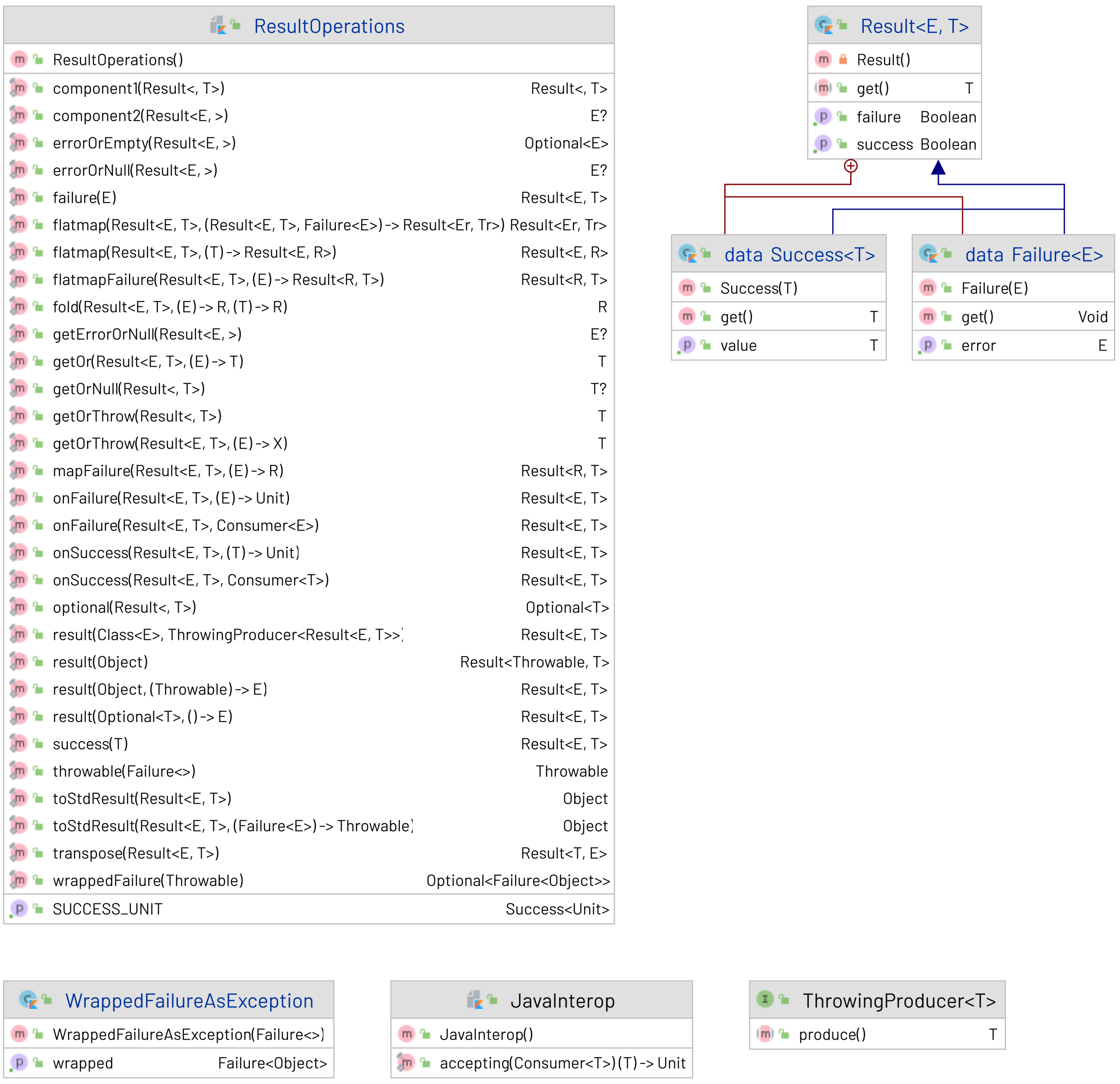
Use Java's `Optional<T>`

Java's optional is not intended to handle errors, rather it used to handle `null` in very explicit manner. Specifically an optional only indicate that a value is present, or not.

Use of domain exceptions

High Level Overview

The Result library implementation can summarized by the following UML diagrams:


ResultOperations
ResultOperations()
component1(Result<, T>) Result<, T>
component2(Result<E, >) E?
errorOrEmpty(Result<E, >) Optional<E>
errorOrNull(Result<E, >) E?
failure(E) Result<E, T>
flatmap(Result<E, T>, (Result<E, T>, Failure<E>) -> Result<Er, Tr>) Result<Er, Tr>
flatmap(Result<E, T>, (T) -> Result<E, R>) Result<E, R>
flatmapFailure(Result<E, T>, (E) -> Result<R, T>) Result<R, T>
fold(Result<E, T>, (E) -> R, (T) -> R) R
getErrorOrNull(Result<E, >) E?
getOr(Result<E, T>, (E) -> T) T
getOrNull(Result<, T>) T?
getOrThrow(Result<, T>) T
getOrThrow(Result<E, T>, (E) -> X) T
mapFailure(Result<E, T>, (E) -> R) Result<R, T>
onFailure(Result<E, T>, (E) -> Unit) Result<E, T>
onFailure(Result<E, T>, Consumer<E>) Result<E, T>
onSuccess(Result<E, T>, (T) -> Unit) Result<E, T>
onSuccess(Result<E, T>, Consumer<T>) Result<E, T>
optional(Result<, T>) Optional<T>
result(Class<E>, ThrowingProducer<Result<E, T>>) Result<E, T>
result(Object) Result<Throwable, T>
result(Object, (Throwable) -> E) Result<E, T>
result(Optional<T>, () -> E) Result<E, T>
success(T) Result<E, T>
throwable(Failure<>) Throwable
toStdResult(Result<E, T>) Object
toStdResult(Result<E, T>, (Failure<E>) -> Throwable) Object
transpose(Result<E, T>) Result<T, E>
wrappedFailure(Throwable) Optional<Failure<Object>>
SUCCESS_UNIT Success<Unit>
Result<E, T>
Result()
get() T
failure Boolean
success Boolean
data Success<T>
Success(T)
get() T
value T
data Failure<E>
Failure(E)
get() Void
error E
WrappedFailureAsException
WrappedFailureAsException(Failure<>)
wrapped Failure<Object>
JavaInterop
JavaInterop()
accepting(Consumer<T>)(T) -> Unit
ThrowingProducer<T>
produce() T

At this point note that:

1. `ResultOperations` represent a common library of operations applicable to`Result` types.
2. These operations exposes a high level API which enables the user to leverage both traditional *try-catch* semantics as well as *higher functions* via lambdas.
3. The `UnhandledFailureAsException` class along with the `tryUnwrappingFailure()` function is used to bridge between functional style of error handling, and the more traditional OO style of using a `try-catch`
4. Notice also that `Success.get()` returns an actual success value, while `Failure.get()` actually does not return anything. *In fact* – the latter throws an exception if called.
5. The `Result<E,T>` class hierarchy is sealed. This means that at compile- and runtime there can only be, either an `Success<T>`, or `Failure<E>` instance for given `Result` instance.

Library Use Case Patterns

Improved API Design

Pure Functional Error Handling

Traditional Object Orientated Error Handling

Hybrid Approach of Error handling