KotlinConf

Architecture using Functional Programming concepts <+>

Jorge Castillo
@JorgeCastilloPr

GoMore

Kotlin and Functional Programming

- FP means concern separation (declarative computations vs runtime execution), purity, referential transparency, push state aside…
- Many features are also found on FP languages.
- Kotlin still lacks important FP features (HKs, typeclasses…)

kategory.io

- Functional datatypes and abstractions over Kotlin
- Inspired by typelevel/cats, Scalaz
- Open for public contribution

Let’s use it to solve some key problems for many systems👌

- Modeling error and success cases
- Asynchronous code + Threading
- Side Effects
- Dependency Injection
- Testing

Error / Success cases

Vanilla Java approach: Exceptions + callbacks

Return type cannot reflect what you get in return

```
public class GetHeroesUseCase {

  public GetHeroesUseCase(HeroesDataSource dataSource, Logger logger) {
    /* … */
  }

  public void get(int page, Callback<List<SuperHero>> callback) {
    try {
      List<SuperHero> heroes = dataSource.getHeroes(page);
      callback.onSuccess(heroes);
    } catch (IOException e) {
      logger.log(e);
      callback.onError("Some error");
    }
  }
}
```

Breaks referential transparency: Error type?

Catch + callback to surpass thread limits

Alternative 1: Result wrapper (Error + Success)

```
public Result(ErrorType error, SuccessType success) {
  this.error = error;
  this.success = success;
}

public enum Error {
  NETWORK_ERROR, NOT_FOUND_ERROR, UNKNOWN_ERROR
}

public class GetHeroesUseCase {

  /*...*/

  public Result<Error, List<SuperHero>> get(int page) {
    Result<Error, List<SuperHero>> result = dataSource.getHeroes(page);
    if (result.isError()) {
      logger.log(result.getError());
    }
    return result;
  }
}
```

Wrapper type

We are obviously tricking here. We are ignoring async, but at least we have a very explicit return type.

Alternative 2: RxJava


```
public class GetHeroesUseCaseRx {

  public Single<List<SuperHero>> get() {
    return dataSource.getHeroes()
        .map(this::discardNonValidHeroes)
        .doOnError(logger::log);
  }

  private List<SuperHero> discardNonValidHeroes(List<SuperHero> superHeroes) {
    return superHeroes;
  }
}
```

Threading is easily handled using Schedulers

Both result sides (error / success) fit on a single stream

```
public class HeroesNetworkDataSourceRx {

  public Single<List<SuperHero>> getHeroes() {
    return Single.create(emitter -> {
      List<SuperHero> heroes = fetchSuperHeroes();
      if (everythingIsAlright()) {
        emitter.onSuccess(heroes);
      } else if (heroesNotFound()) {
        emitter.onError(new RxErrors.NotFoundError());
      } else {
        emitter.onError(new RxErrors.UnknownError());
      }
    });
  }
}
```

Alternative 3: Either<Error, Success>

```
sealed class CharacterError {
  object AuthenticationError : CharacterError()
  object NotFoundError : CharacterError()
  object UnknownServerError : CharacterError()
}
```

Sealed hierarchy of supported domain errors

```
/* data source impl */
fun getAllHeroes(service: HeroesService): Either<CharacterError, List<SuperHero>> =
    try {
      Right(service.getCharacters().map { SuperHero(it.id, it.name, it.thumbnailUrl, it.description) })
    } catch (e: MarvelAuthApiException) {
      Left(AuthenticationError)
    } catch (e: MarvelApiException) {
      if (e.httpCode == HttpURLConnection.HTTP_NOT_FOUND) {
        Left(NotFoundError)
      } else {
        Left(UnknownServerError)
      }
    }
```

Transform outer layer exceptions on expected domain errors

```
fun getHeroesUseCase(dataSource: HeroesDataSource, logger: Logger): Either<Error, List<SuperHero>> =
    dataSource.getAllHeroes().fold(
      { logger.log(it); Left(it) },
      { Right(it) })
```

We fold() over the Either for effects depending on the side

Alternative 3: Either<Error, Success> 

- Presentation code could look like this:

```
fun getSuperHeroes(view: SuperHeroesListView, logger: Logger, dataSource: HeroesDataSource) {
  getHeroesUseCase(dataSource, logger).fold(
      { error -> drawError(error, view) },
      { heroes -> drawHeroes(heroes, view) })
}

private fun drawError(error: CharacterError,
    view: HeroesView) {
  when (error) {
    is NotFoundError -> view.showNotFoundError()
    is UnknownServerError -> view.showGenericError()
    is AuthenticationError -> view.showAuthenticationError()
  }
}

private fun drawHeroes(success: List<SuperHero>, view: SuperHeroesListView) {
  view.drawHeroes(success.map {
    RenderableHero(
        it.name,
        it.thumbnailUrl)
  })
}
```

But still, what about Async + Threading?! 😱

Asynchronous code + Threading

Alternatives

- Vanilla Java: **ThreadPoolExecutor** + **exceptions** + **callbacks**.
- RxJava: **Schedulers** + **observable** + **error subscription**.
- KATEGORY: 
  - IO to wrap the IO computations and **make them pure.**
  - Make the computation **explicit in the return type**

IO<Either<CharacterError, List<SuperHero>>>

- IO wraps a computation that can return either a CharacterError or a List<SuperHero>, **never both**.

```
/* network data source */
fun getAllHeroes(service: HeroesService, logger: Logger):
IO<Either<CharacterError, List<SuperHero>>> =
    runInAsyncContext(
        f = { queryForHeroes(service) },
        onError = { logger.log(it); it.toCharacterError().left() },
        onSuccess = { mapHeroes(it).right() },
        AC = IO.asyncContext()
    )
```

Very explicit result type

We run the task in an async context using kotlinx coroutines. It returns an IO wrapped computation.

IO<Either<CharacterError, List<SuperHero>>>

```
/* Use case */
fun getHeroesUseCase(service: HeroesService, logger: Logger):
IO<Either<CharacterError, List<SuperHero>>> =
    getAllHeroesDataSource(service, logger).map { it.map { discardNonValidHeroes(it) } }


/* Presentation logic */
fun getSuperHeroes(view: SuperHeroesListView, service: HeroesService, logger: Logger) =
    getHeroesUseCase(service, logger).unsafeRunAsync { it.map { maybeHeroes ->
      maybeHeroes.fold(
          { error -> drawError(error, view) },
          { success -> drawHeroes(success, view) })}
    }
```

- Effects are being applied here, but that's not ideal!

Problem

- Ideally, we would perform unsafe effects on the **edge of the system**, where our frameworks are coupled. On a system with a frontend layer, it would be the view impl.

Solutions

- Lazy evaluation. Defer all the things!

- Declare the whole execution tree based on returning functions

- By returning functions at all levels, you **swap proactive evaluation with deferred execution**.

```
presenter(deps) = { deps -> useCase(deps) }

useCase(deps) = { deps -> dataSource(deps) }

dataSource(deps) = { deps -> deps.apiClient.getHeroes()  }
```

- But passing dependencies all the way down at every execution level can be painful 😓.
- Can’t we **implicitly inject / pass** them in a simple way to avoid passing them manually?

Dependency Injection / passing

Discovering the Reader Monad

- Wraps a computation with type (D) -> A and enables composition over computations with that type.
- D stands for the Reader “context” (dependencies)
- Its operations **implicitly pass** in the context to the next execution level.
- Think about the context as the dependencies needed to run the complete function tree. (dependency graph)

Discovering the Reader Monad

- It solves both concerns:
  - Defers computations at all levels.
  - Injects dependencies by automatically passing them across the different function calls.

Reader<D, IO<Either<CharacterError, List<SuperHero>>>> 🤒

- We start to die on types a bit here. We'll find a solution for it!

```
/* data source could look like this */
fun getHeroes():
Reader<GetHeroesContext, IO<Either<CharacterError, List<SuperHero>>>> =
    Reader.ask<GetHeroesContext>().map({ ctx ->
      runInAsyncContext(
          f = { ctx.apiClient.getHeroes() },
          onError = { it.toCharacterError().left() },
          onSuccess = { it.right() },
          AC = ctx.threading
      )
    })
```

Explicit dependencies not needed anymore

Reader.ask() lifts a Reader { D -> D } so we get access to D when mapping

Reader<D, IO<Either<CharacterError, List<SuperHero>>>>

```
/* use case */
fun getHeroesUseCase() = fetchAllHeroes().map { io ->
  io.map { maybeHeroes ->
    maybeHeroes.map { discardNonValidHeroes(it) }
  }
}
/* presenter code */
fun getSuperHeroes() = Reader.ask<GetHeroesContext>().flatMap(
{ (_, view: SuperHeroesListView) ->
  getHeroesUseCase().map({ io ->
    io.unsafeRunAsync { it.map { maybeHeroes ->
        maybeHeroes.fold(
            { error -> drawError(error, view) },
            { success -> drawHeroes(view, success) })
      }
    }
  })
})
```

Reader<D, IO<Either<CharacterError, List<SuperHero>>>>

- Complete computation tree deferred thanks to Reader.
- Thats a completely pure computation since effects are still not run.
- When the moment for performing effects comes, you can simply run it passing the context you want to use:

```
/* we perform unsafe effects on view impl now */
override fun onResume() {
  /* presenter call */
  getSuperHeroes().run(heroesContext)
}
```

Returns a Reader (deferred computation)

- On testing scenarios, you just need to **pass a different context** which can be providing fake dependencies for the ones we need to mock.

How to improve the nested types “hell”?

- Monads do not compose gracefully.
- Functional developers use Monad Transformers to solve this.
- Monad Transformers wrap monads to gift those with other monad capabilities.

How to improve the nested types “hell”?

- We want to achieve ReaderT<EitherT<IO>>
- EitherT (Either Transformer) gives Either capabilities to IO.
- ReaderT (Reader Transformer) gives Reader capabilities to EitherT<IO>
- We create an alias for that composed type, for syntax:

```
typealias AsyncResult = ReaderT<EitherT<IO>>
```

AsyncResult<D, A>

- Takes care of error handling, asynchrony, IO operations, and dependency injection.

```
/* data source */
fun <D : SuperHeroesContext> fetchAllHeroes(): AsyncResult<D, List<SuperHero>> =
    AsyncResult.monadError<D>().binding {
        val query = buildFetchHeroesQuery()
        val ctx = AsyncResult.ask<D>().bind()
        runInAsyncContext(
            f = { fetchHeroes(ctx, query) },
            onError = { liftError<D>(it) },
            onSuccess = { liftSuccess(it) },
            AC = ctx.threading<D>()
        ).bind()
    }
```

bindings are part of Monad comprehensions.

Code sequential async calls as if they were sync.

- Monad bindings return an already lifted and flatMapped result to the context of the monad.

AsyncResult<D, A>

```
/* use case */
fun <D : SuperHeroesContext> getHeroesUseCase(): AsyncResult<D, List<CharacterDto>> =
    fetchAllHeroes<D>().map { discardNonValidHeroes(it) }

/* presenter */
fun getSuperHeroes(): AsyncResult<GetHeroesContext, Unit> =
    getHeroesUseCase<GetHeroesContext>()
        .map { heroesToRenderableModels(it) }
        .flatMap { drawHeroes(it) }
        .handleErrorWith { displayErrors(it) }

/* view impl */
override fun onResume() {
  getSuperHeroes().unsafePerformEffects(heroesContext)
}
```

- Again on testing scenarios, you just need to **pass a different context** which can be providing fake dependencies for the ones we need to mock.

Extra bullets

- Two advanced FP styles can be implemented using Kategory.
  - Tagless-Final
  - Free Monads

Tagless-Final

- Remove concrete monad types from your code (IO, Either, Reader) and depend just on behaviors defined by typeclasses.
- Run your program later on passing in the implementations you want to use for those behaviors on this execution.
- tagless-final gradle module on sample repo + PR: github.com/JorgeCastilloPrz/KotlinAndroidFunctional/pull/2

Free Monads

- Separates concerns about declaring the AST (abstract syntax tree) based on Free<S, A> in a pure way, and interpreting it later on using an interpreter.
- Free is used to decouple dependencies, so it also replaces the need for dependency injection. Remember this when defining the algebras.
- free-monads gradle module + PR: github.com/JorgeCastilloPrz/KotlinAndroidFunctional/pull/6

Some conclusions

- The patterns we learned today to solve DI, asynchrony, decoupling… etc, are shared with any other FP languages. That helps us to share all the concepts and glossary with frontend and backend devs inside the company.
- On FP its common to fix problems once and use the same solution for further executions, programs or systems.

Samples for every style explained

- Four grade modules on repo github.com/JorgeCastilloPrz/KotlinAndroidFunctional
  - nested-monads (Monad Stack)
  - monad-transformers
  - Tagless-Final
  - Free Monads
- https://medium.com/@JorgeCastilloPr/

Thank you!

Jorge Castillo
@JorgeCastilloPr

#kotlinconf17