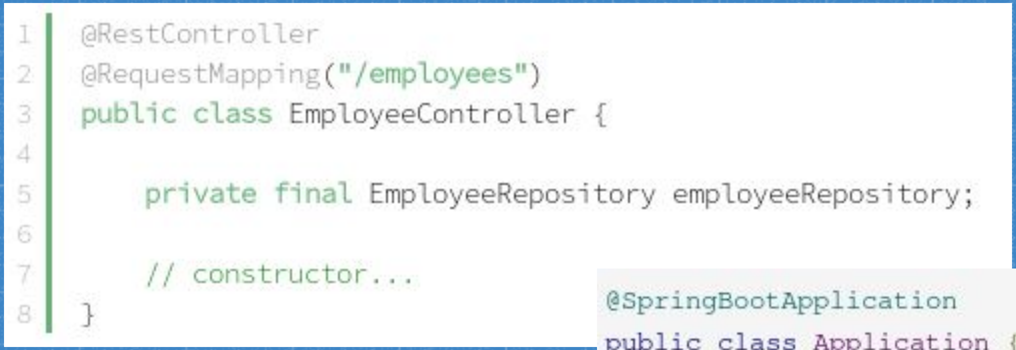
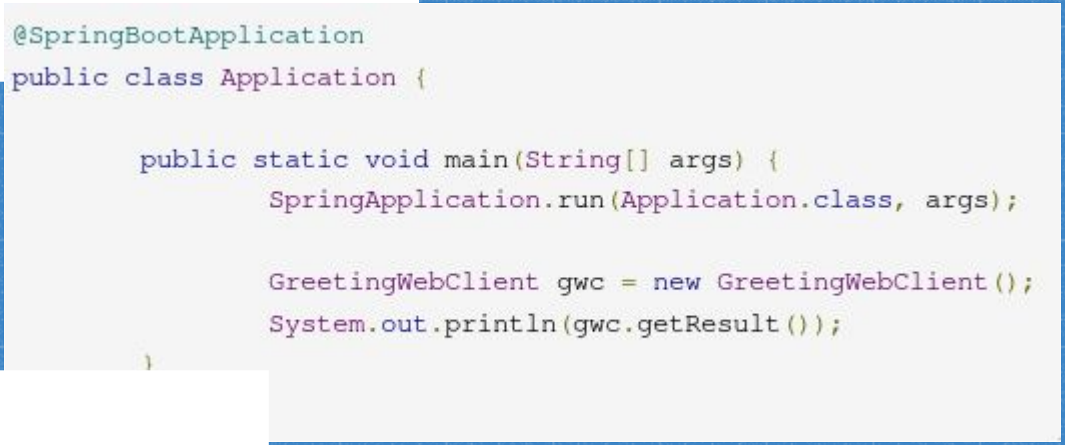
Functional Answers to Object-Oriented Questions

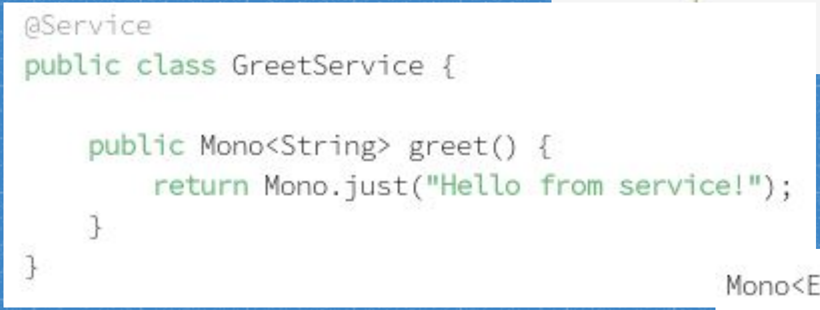
Springframework and Spring Boot = <3



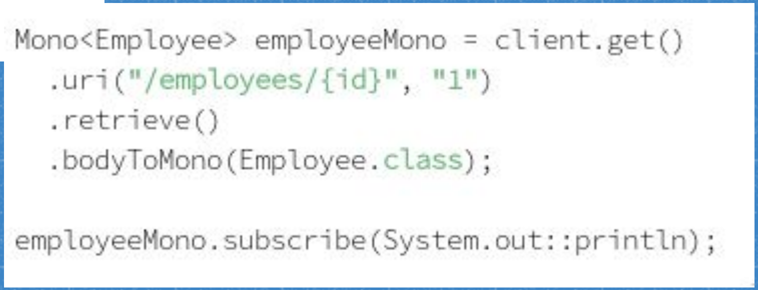
```
@RestController
@RequestMapping("/employees")
public class EmployeeController {

    private final EmployeeRepository employeeRepository;

    // constructor...
}
```



```
@SpringBootApplication
public class Application {

        public static void main(String[] args) {
                SpringApplication.run(Application.class, args);

                GreetingWebClient gwc = new GreetingWebClient();
                System.out.println(gwc.getResult());
        }
```



```
@Service
public class GreetService {

    public Mono<String> greet() {
        return Mono.just("Hello from service!");
    }
}
```

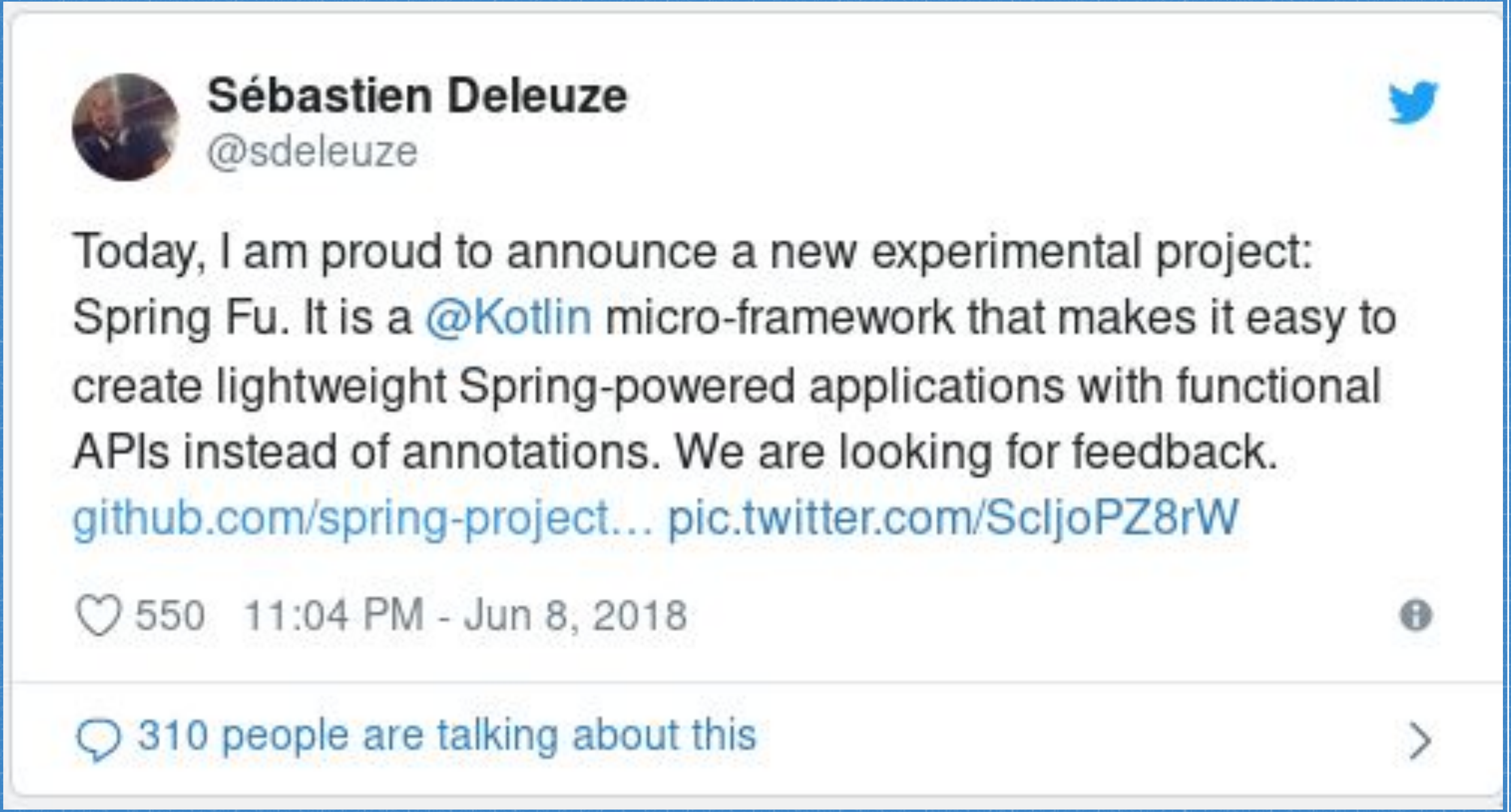


```
Mono<Employee> employeeMono = client.get()
  .uri("/employees/{id}", "1")
  .retrieve()
  .bodyToMono(Employee.class);

employeeMono.subscribe(System.out::println);
```

Not my real production code

I stopped seeing this

And started seeing this! </3



Sébastien Deleuze
@sdeleuze

Today, I am proud to announce a new experimental project: Spring Fu. It is a @Kotlin micro-framework that makes it easy to create lightweight Spring-powered applications with functional APIs instead of annotations. We are looking for feedback. github.com/spring-project… pic.twitter.com/ScljoPZ8rW

550 11:04 PM - Jun 8, 2018

310 people are talking about this

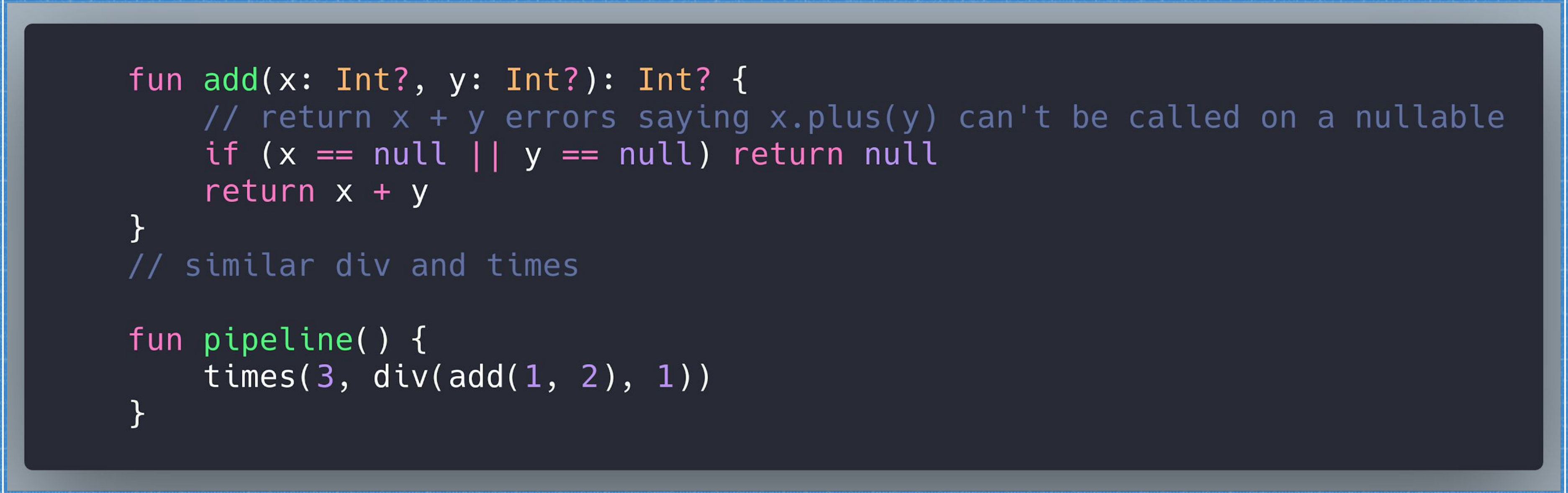
And I wasn’t the only one feeling this way

A Functional Approach

Values, Data, and Structures

Values...

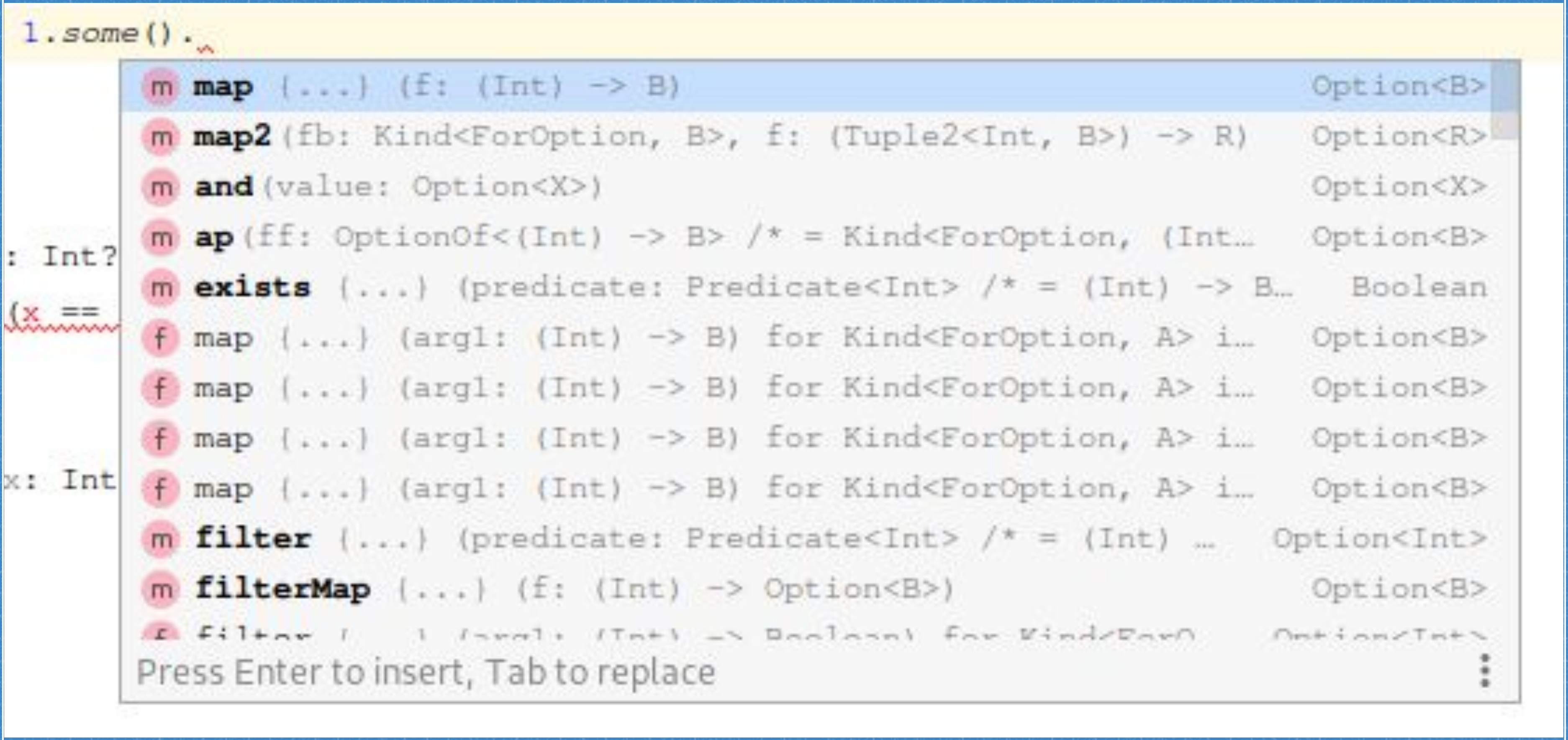
More Than a Feeling



```kotlin
fun add(x: Int?, y: Int?): Int? {
    // return x + y errors saying x.plus(y) can't be called on a nullable
    if (x == null || y == null) return null
    return x + y
}
// similar div and times

fun pipeline() {
    times(3, div(add(1, 2), 1))
}
```

```kotlin
fun add(x: Int, y: Int): Int {
    return x + y
}
// similar div and times

fun pipeline() {
    nullableNumer?.let { x ->
        nullableOtherNumber?.let { y ->
            add(x, y)
        }
    }?.let {
        div(it, 1)
    }?.let {
        times(it, 3)
    }
}
```

```
fun pipeline() {
    optionNumX
        .map2(optionNumY, { (x, y) -> add(x, y) })
        .map{ div(it, 1 }
        .map { times(it, 3) }
}
```

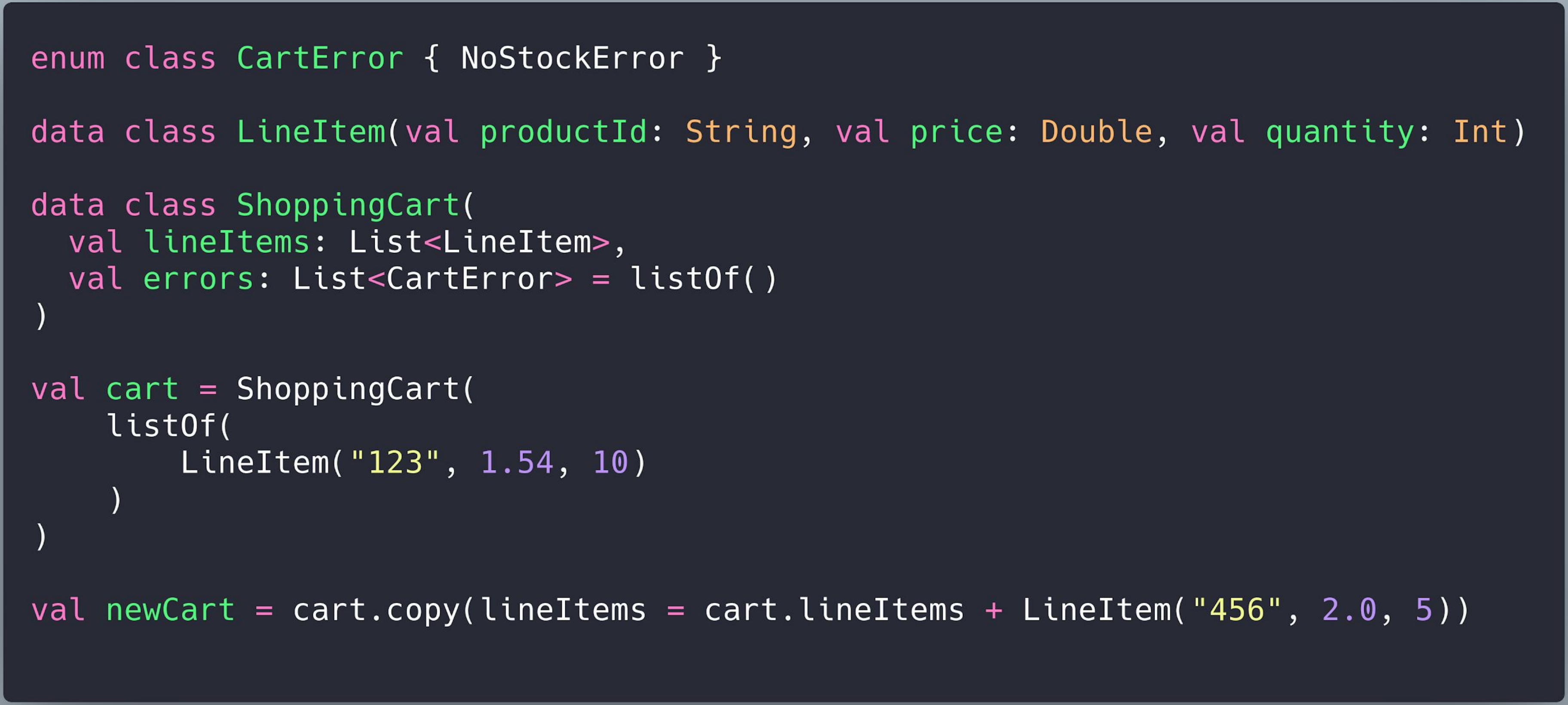


```
1.some().
```

```
m map (...) (f: (Int) -> B)                                              Option<B>
m map2 (fb: Kind<ForOption, B>, f: (Tuple2<Int, B>) -> R)                Option<R>
m and (value: Option<X>)                                                 Option<X>
m ap (ff: OptionOf<(Int) -> B> /* = Kind<ForOption, (Int…                Option<B>
m exists (...) (predicate: Predicate<Int> /* = (Int) -> B…                 Boolean
f map (...) (arg1: (Int) -> B) for Kind<ForOption, A> i…                 Option<B>
f map (...) (arg1: (Int) -> B) for Kind<ForOption, A> i…                 Option<B>
f map (...) (arg1: (Int) -> B) for Kind<ForOption, A> i…                 Option<B>
f map (...) (arg1: (Int) -> B) for Kind<ForOption, A> i…                 Option<B>
m filter (...) (predicate: Predicate<Int> /* = (Int) …                 Option<Int>
m filterMap (...) (f: (Int) -> Option<B>)                                Option<B>
f filter (...) (arg1: (Int) -> Boolean) for Kind<ForO…                 Option<Int>
```

Add Behaviour...

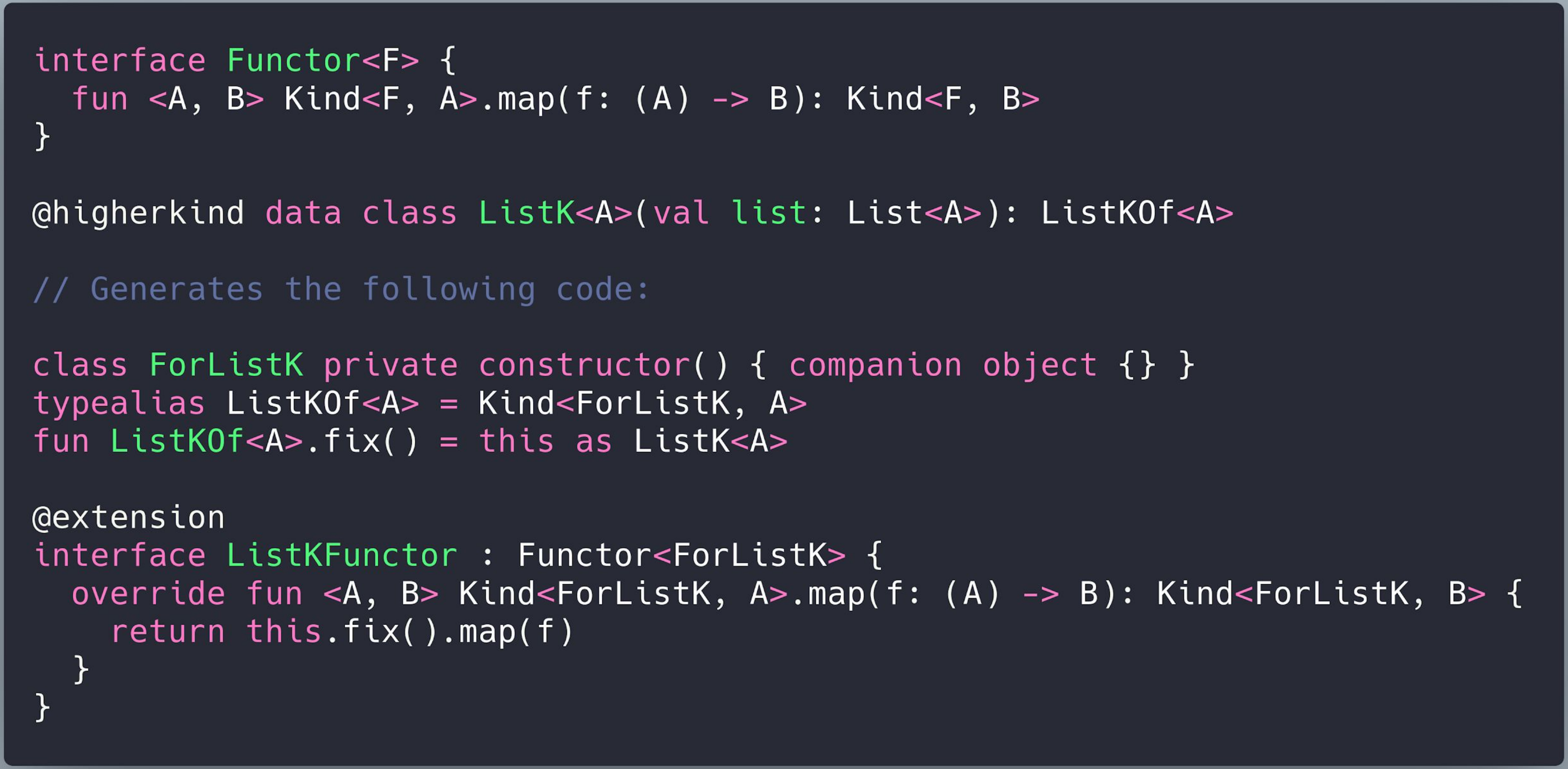
Cast polymorph



```kotlin
enum class CartError { NoStockError }

data class LineItem(val productId: String, val price: Double, val quantity: Int)

data class ShoppingCart(
  val lineItems: List<LineItem>,
  val errors: List<CartError> = listOf()
)

val cart = ShoppingCart(
    listOf(
        LineItem("123", 1.54, 10)
    )
)

val newCart = cart.copy(lineItems = cart.lineItems + LineItem("456", 2.0, 5))
```

```kotlin
fun combineCarts(cart1: ShoppingCart, cart2: ShoppingCart): ShoppingCart {
  return cart1.copy(
    cart1.lineItems + cart2.lineItems,
    cart1.errors + cart2.errors
  )
}
```

```kotlin
interface ShoppingCartMonoid : Monoid<ShoppingCart> {
    override fun empty() = ShoppingCart(listOf(), listOf())

    override fun ShoppingCart.combine(b: ShoppingCart): ShoppingCart {
        return ShoppingCart(lineItems + b.lineItems, errors + b.errors)
    }
}

fun ShoppingCart.Companion.monoid(): Monoid<ShoppingCart> =
    object : ShoppingCartMonoid {}

fun <F> useMonoid(M: Monoid<F>, f0: F, f1: F): F {
  // f0.combine(f1) doesn't work
  // fails to compile with f0.combine is undefined

  // instead have to
  M.run { f0.combine(f1) }
}

useCartMonoid(ShoppingCart.monoid(), cart0, cart1)
```



```kotlin
interface Functor<F> {
  fun <A, B> Kind<F, A>.map(f: (A) -> B): Kind<F, B>
}

@higherkind data class ListK<A>(val list: List<A>): ListKOf<A>

// Generates the following code:

class ForListK private constructor() { companion object {} }
typealias ListKOf<A> = Kind<ForListK, A>
fun ListKOf<A>.fix() = this as ListK<A>

@extension
interface ListKFunctor : Functor<ForListK> {
  override fun <A, B> Kind<ForListK, A>.map(f: (A) -> B): Kind<ForListK, B> {
    return this.fix().map(f)
  }
}
```

Nested Mutation...

Immutably

```kotlin
@optics data class Street(val number: Int, val name: String)
@optics data class Address(val city: String, val street: Street)
@optics data class Company(val name: String, val address: Address)
@optics data class Employee(val name: String, val company: Company?)

val street = Street(42, "lambda street")
val address = Address("Functional city", street)
val company = Company("Kategory", address)
val employee = Employee("John Doe", company)

employee.copy(
  company = employee.company.copy(
    address = employee.company.address.copy(
      street = employee.company.address.street.copy(
        name = employee.company.address.street.name.capitalize()
      )
    )
  )
)

// vs

val optional: Optional<Employee, String> = Employee.company.address.street.name
optional.modify(employee, String::toUpperCase) // sets street to Lambda Street
```

Look Ma...

No Magic Annotations

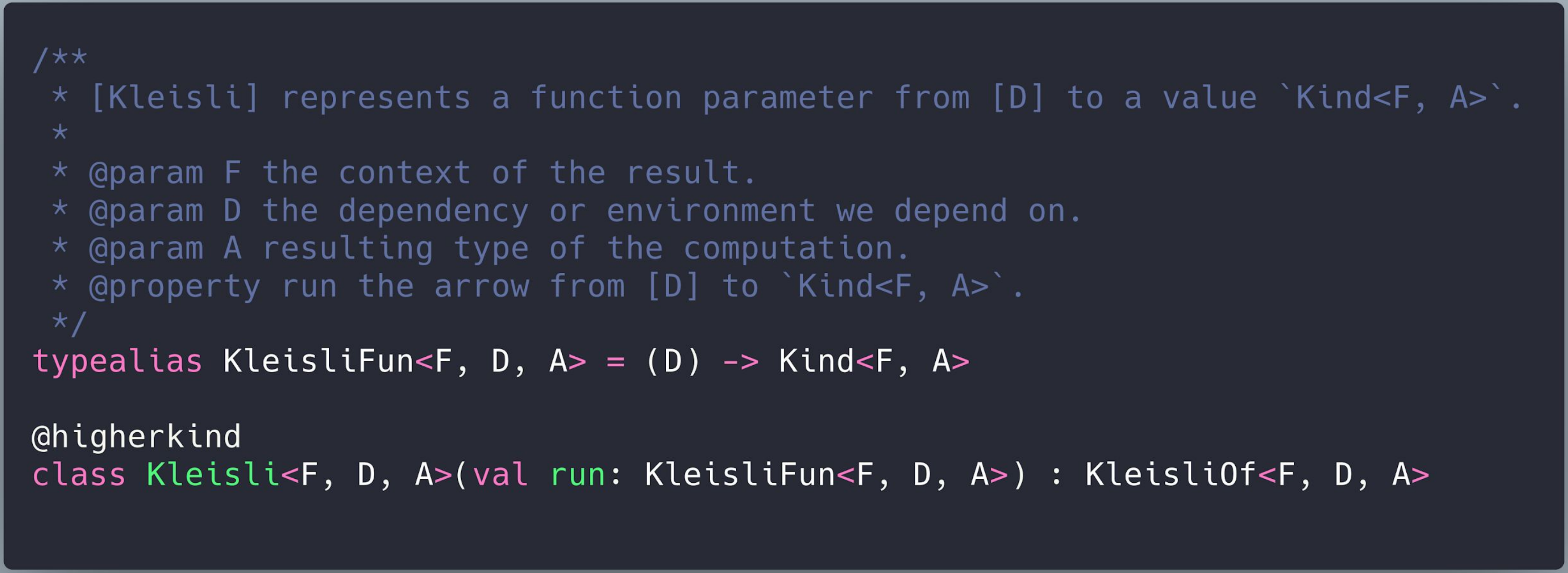
```kotlin
fun applicationFactory(env: Env<ForMonoK>, pport: Int): KofuApplication {
    return application(WebApplicationType.REACTIVE) {
        webFlux {
            port = pport
            router {
                GET("/") { discountHandler(env, it) }
            }
        }
    }
}
```

```kotlin
class Discount(val id: String, val productId: String, val discount: Double)

interface Repository<T, D> { fun findAll(): Kind<T, List<D>> }

interface Repositories<T> { val discountRepository: Repository<T, Discount> }

interface WarehouseService<T> { fun checkStock(productId: String, quantity: Double):
Kind<T, Pair<String, Boolean>> }

interface ExternalServices<T> { val warehouseService: WarehouseService<T> }

@higherkind
data class Env<R>(
        val log: (String) -> IO<Unit>,
        val repositories: Repositories<R>,
        val externalServices: ExternalServices<R>
) : EnvOf<R> { companion object }
```

```kotlin
@higherkind
data class Env<R>(
    val log: (String) -> IO<Unit>,
    val repositories: Repositories<R>,
    val externalServices: ExternalServices<R>
) : EnvOf<R> {
    companion object
}

@extension
interface EnvHasRepositories : HasRepositories<ForEnv> {
    override fun <A> Kind<ForEnv, A>.getRepositories() =
      fix().repositories
}

@extension
interface EnvHasWarehouseService : HasWarehouseService<ForEnv> {
    override fun <A> Kind<ForEnv, A>.getWarehouseService() =
      fix().externalServices.warehouseService
}
```

```kotlin
fun main() {
    val env = Env(
            log = { s -> IO { println(s) } },
            repositories = repositories,
            externalServices = externalServices
    )
    applicationFactory(env, 8080).run()
}
```

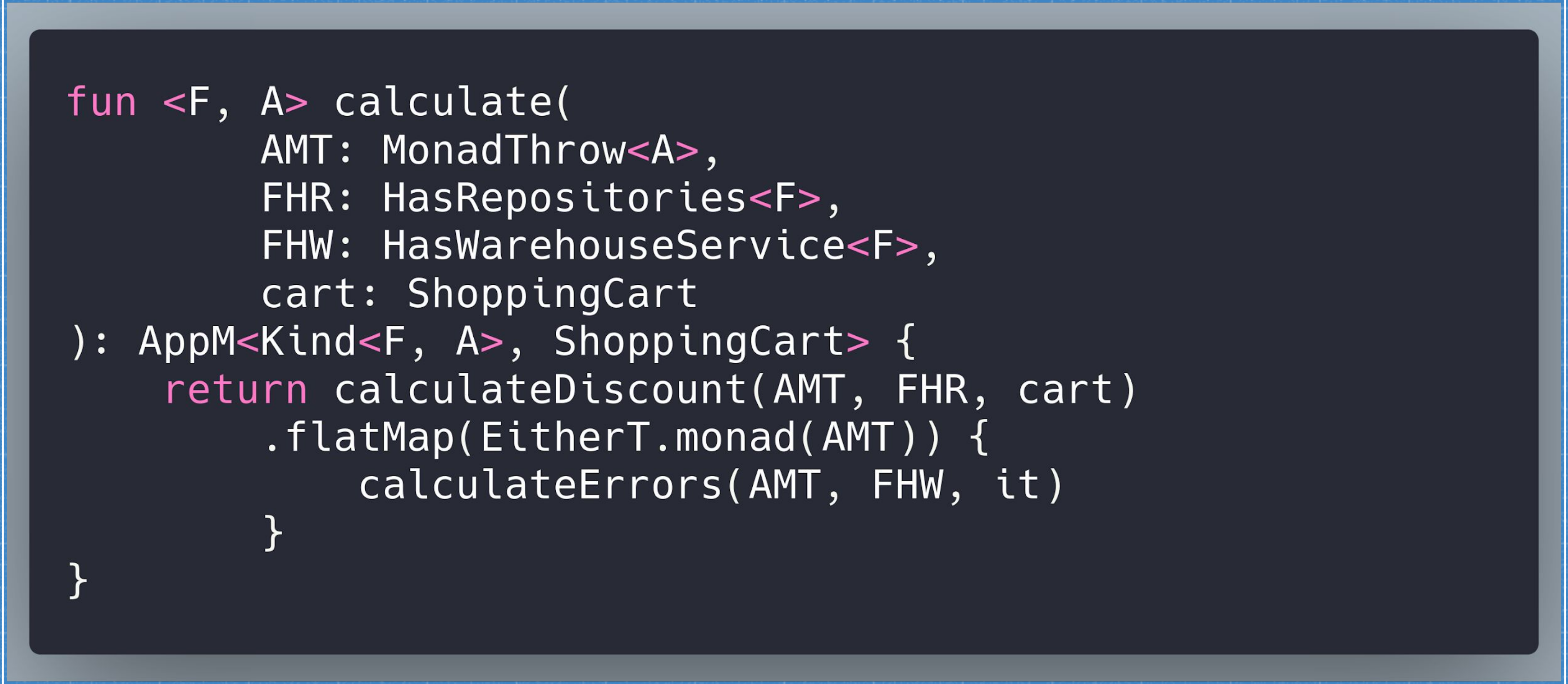
```kotlin
object repositories : Repositories<ForMonoK> {
    override val discountRepository = object : Repository<ForMonoK, Discount> {
        override fun findAll() = /* access database */
    }
}

object externalServices : ExternalServices<ForMonoK> {
    override val warehouseService = object : WarehouseService<ForMonoK> {
        override fun checkStock(productId: String, quantity: Double):
MonoK<Pair<String, Boolean>> {
            /* access warehouse API */
        }
    }
}
```



```kotlin
/**
 * [Kleisli] represents a function parameter from [D] to a value `Kind<F, A>`.
 *
 * @param F the context of the result.
 * @param D the dependency or environment we depend on.
 * @param A resulting type of the computation.
 * @property run the arrow from [D] to `Kind<F, A>`.
 */
typealias KleisliFun<F, D, A> = (D) -> Kind<F, A>

@higherkind
class Kleisli<F, D, A>(val run: KleisliFun<F, D, A>) : KleisliOf<F, D, A>
```

```kotlin
typealias AppM<F, O> = Kleisli<EitherTPartialOf<F, ServerError>, Kind<ForEnv, F>, O>

fun <O> handler(
    env: Env<ForMonoK>,
    request: ServerRequest,
    block: (ServerRequest) -> AppM<ForMonoK, O>
): Mono<ServerResponse> {
    return block(request)
            .run(env).value()
            .map { it.fold({ it }, { it }) }.mono
            .flatMap { x -> ok().syncBody(x.toString()) }
}
```

```kotlin
typealias AppM<F, O> = Kleisli<EitherTPartialOf<F, ServerError>, Kind<ForEnv, F>, O>

fun <O> handler(
    env: Env<ForMonoK>,
    request: ServerRequest,
    block: (ServerRequest) -> AppM<ForMonoK, O>
): Mono<ServerResponse> {
    return block(request)
            .run(env).value()
            .map { it.fold({ it }, { it }) }.mono
            .flatMap { x -> ok().syncBody(x.toString()) }
}

fun discountHandler(env: Env<ForMonoK>, req: ServerRequest): Mono<ServerResponse> {
    return req.bodyToMono(ShoppingCart::class.java).flatMap { cart ->
        handler<ShoppingCart>(env, req) {
            calculate(
                    MonoK.monadThrow(),
                    Env.hasRepositories(),
                    Env.hasWarehouseService(),
                    cart
            )
        }
    }
```



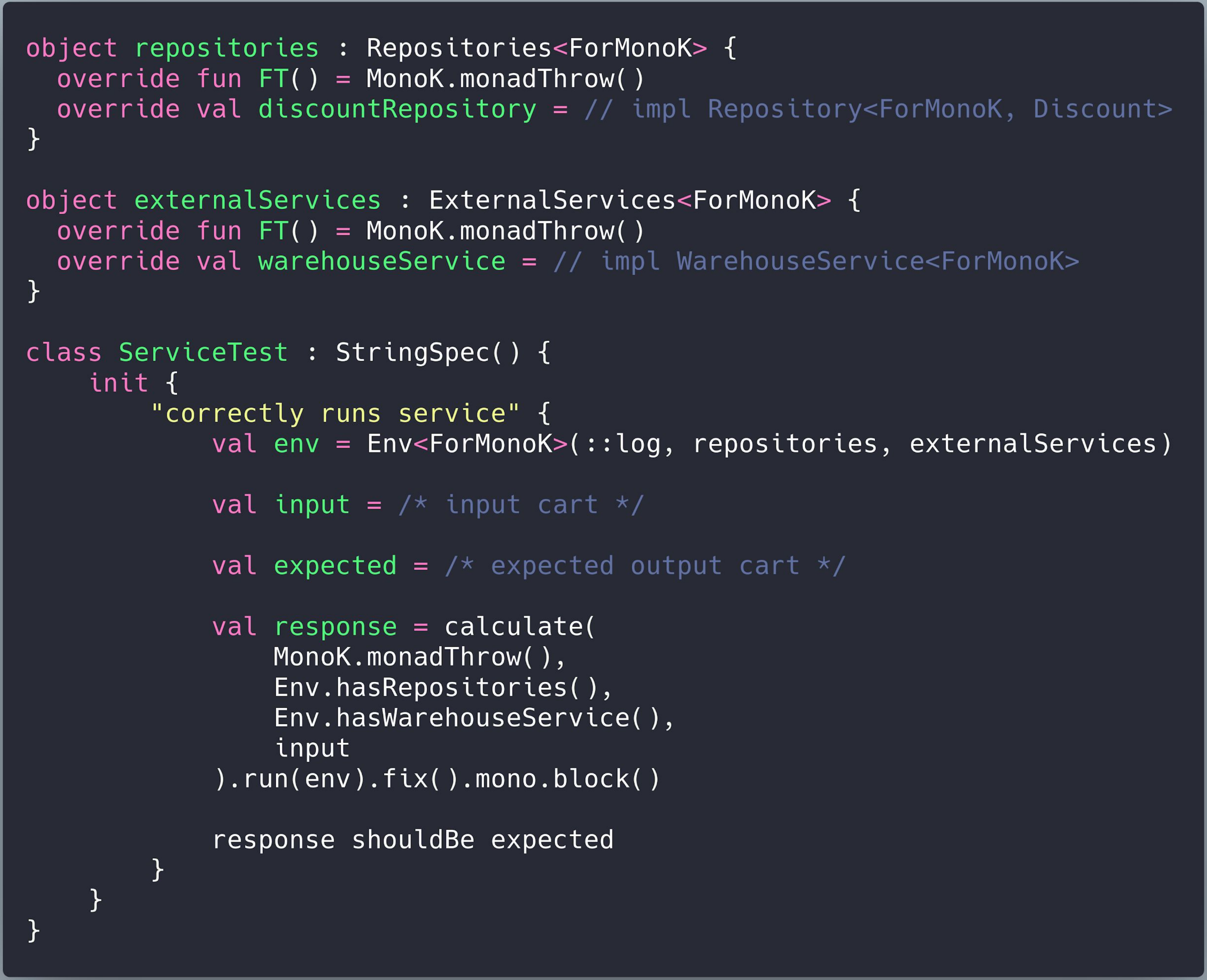
```kotlin
fun <F, A> calculate(
        AMT: MonadThrow<A>,
        FHR: HasRepositories<F>,
        FHW: HasWarehouseService<F>,
        cart: ShoppingCart
): AppM<Kind<F, A>, ShoppingCart> {
    return calculateDiscount(AMT, FHR, cart)
        .flatMap(EitherT.monad(AMT)) {
            calculateErrors(AMT, FHW, it)
        }
}
```

```kotlin
fun <F, A> calculateErrors(
        AMT: MonadThrow<A>,
        FHW: HasWarehouseService<F>,
        cart: ShoppingCart
): Kleisli<EitherTPartialOf<A, ServerError>, Kind<F, A>, ShoppingCart> {
    return Kleisli { env ->
        val warehouseService = FHW.run { env.getWarehouseService() }
        val x = with(AMT) {
            cart.lineItems
                    .map { warehouseService.checkStock(it.productId, it.price) }
                    .structureErrors()
                    .map { ShoppingCart.errors.set(cart, it) }
        }
        EitherT.liftF(AMT, x)
    }
}
```

```kotlin
fun applicationFactory(env: Env<ForMonoK>, pport: Int): KofuApplication {
    return application(WebApplicationType.REACTIVE) {
        webFlux {
            port = pport
            router {
                GET("/") { discountHandler(env, it) }
            }
        }
    }
}
```

Test cases...

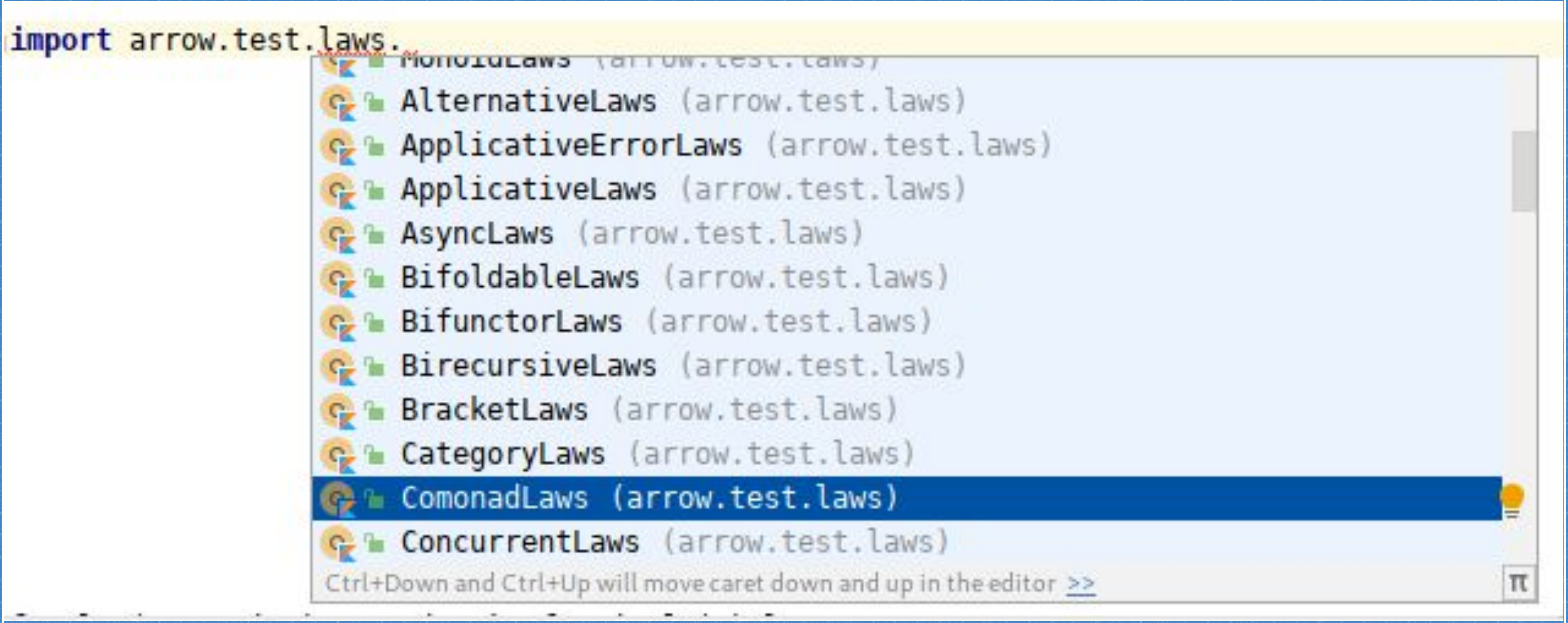
That don’t test your patience



```kotlin
object repositories : Repositories<ForMonoK> {
  override fun FT() = MonoK.monadThrow()
  override val discountRepository = // impl Repository<ForMonoK, Discount>
}

object externalServices : ExternalServices<ForMonoK> {
  override fun FT() = MonoK.monadThrow()
  override val warehouseService = // impl WarehouseService<ForMonoK>
}

class ServiceTest : StringSpec() {
    init {
        "correctly runs service" {
            val env = Env<ForMonoK>(::log, repositories, externalServices)

            val input = /* input cart */

            val expected = /* expected output cart */

            val response = calculate(
                MonoK.monadThrow(),
                Env.hasRepositories(),
                Env.hasWarehouseService(),
                input
            ).run(env).fix().mono.block()

            response shouldBe expected
        }
    }
}
```

```
fun <T> List<T>.immutableReverse() = this.fold(listOf<T>(), { p, c -> p + c })

class ListSpec : StringSpec() {
    init {
        forAll(Gen.list(Gen.int())) { l: List<Int> ->
            l.immutableReverse().immutableReverse() == l
        }
    }
}
```

```kotlin
object ShoppingCartGenerator : Gen<ShoppingCart> {
    override fun constants() = emptyList<ShoppingCart>()

    override fun random() = generateSequence {
        ShoppingCart(
            Gen.list(LineItemGenerator).random().first(),
            Gen.list(Gen.pair(
              Gen.string(),
              Gen.constant(CartError.NoStockError)
            )).random().first()
        )
    }
}
```



```
import arrow.test.laws.
MonoidLaws (arrow.test.laws)
AlternativeLaws (arrow.test.laws)
ApplicativeErrorLaws (arrow.test.laws)
ApplicativeLaws (arrow.test.laws)
AsyncLaws (arrow.test.laws)
BifoldableLaws (arrow.test.laws)
BifunctorLaws (arrow.test.laws)
BirecursiveLaws (arrow.test.laws)
BracketLaws (arrow.test.laws)
CategoryLaws (arrow.test.laws)
ComonadLaws (arrow.test.laws)
ConcurrentLaws (arrow.test.laws)
Ctrl+Down and Ctrl+Up will move caret down and up in the editor >>
```

```kotlin
class ShoppingCartSpec : StringSpec() {
    init {
        "obeys the monoid laws" {
            MonoidLaws.laws(ShoppingCart.monoid(), ShoppingCartGenerator, Eq.any())
        }
    }
}
```

Go check out arrow-kt.io

You can find me at:

@bassjacob
me@bassjacob.com