* Predicate Calculus Inferences: States / Order.
* Contexts:
* Resources : Data Statements (CSPO);
* Kinds : Schema Patterns (C,SK,PK,OK);
* Contexts : Transforms Patterns (C, (P, SK), (SK,OK), (P, OK));
* Selectors (context roles). Bind, unit. Map, flatMap. Transforms (domain / range) interaction flows. Functional DCI: Wrappers hierarchy (i.e.: root DOM / Resource / Kind / Contexts).
* DCI: actions dynamic DSL. Parse domain / roles (contexts selectors). DataFlows from models Contexts Transforms domains / range.
* DCI Roles: Monads. Type classes. Traits. Implicits.
* DCI Contexts / Interactions: Roles Resolution (ctx init / dialog / prompts), Functional DataFlow.
* @FunctionalInterface Parameterization. Higher Kinds Types. Case Matching: Kind (class) Role (metaclass) functions contexts arguments resolution.
* A trait, which is the type class.
* Type class instances, which are implicit values.
* Type class usage, which uses implicit parameters.
* Role streams, observable. RxJava. Vert.x.
* Higher Kinded Types. Free Monads / DSLs from quads / triples (wrapper, wrapped, ids: encoding / matching, transformations). DOM / Parse Models.
* Models: OGM DOM Domain Facade
* Functor Contexts / Transforms (Models, Ontology)
* Services / Augmentations DataFlows.
* Data, Context, Interaction DataFlows.
* HATEOAS / HAL Model, View, Controller
* DDD CDI Connectors (Protocols)
* Data, Information, Knowledge:
* OGM DOM. MVC / DCI. CDI DDD.
* Model / Data: OGM / DOM (Resource, Instance).
* Use Cases / Contexts / Controller. MVC / DCI (Class, Metaclass).
* Roles / Views / Interactions: CDI / DDD (Occurrence, Role).
* Use Cases / Data Model / Roles Interactions:  ResourceURN aggregated role Statements pairs.
* Data: Resource (actor), Instance (occurrence).
* Contexts: Class (player), Metaclass (role).
* Interactions: Occurrence, Role.
* In DCI, a Context holds the mappings from roles in an interaction to specific object instances, and also have the interactions that can be performed on this mapping.
* Finally, interactions are implemented as simple methods in the context. They may take arguments, and then look up mapped roles in the context map, and fire off a domain method. The important point here is that the interaction methods should match whatever actions you have in your user interface, so that the code matches the users mental model.
* We have three entities, Project, User, and Task. Tasks implement the Assignable role. Project and User implement the Assignments role. Users can be Assignees. We then have a main context with interactions called InboxContext.
* Both Users and Projects have an Inbox with tasks, and it is these that we want to assign to Users. But they can be "owned" either by the User itself, or by a particular Project, hence the need for a separate role Assignments so that our notion of "assignment" is not tied to "given a users list of tasks, one of them can be assigned to the user" but rather "given a Assignments collection of Assignables, pick one and assign to an Assignee". This way the interaction and context is entirely separated from the actual classes, and the only thing we need are the appropriate roles.
* Executing an interaction
* Let's see what happens as we walk through an execution of the "assign" interaction. The first thing we need to do is set up the context:
* InteractionContext map;
* RootContext context;
* InboxContext inboxContext;
* I create a new context map for this interaction, and instantiate a new RootContext. This symbolizes the root of all contexts in my application, and will mainly hold methods for getting to subcontexts. I pass in the map so that the RootContext can pass it on during the user() call, which will create the subcontext UserContext that has all the interactions and subcontexts for working with a selected user.
* I pass in the userId so that the user() method can do the lookup. If this DCI implementation is used in a REST API setting that context lookup will basically map to the URL, so the "userId" will be one part of the URL being referenced. You can imagine the above being mapped to "/administrator/inbox" in a URL.
* The user() method will add the given user to the context map, and then create a new subcontext with the extended map. Here's what it looks like:
* public class RootContext   extends Context
* public UserContext user(String id)
* "Context" is a baseclass that has the InteractionContext in a variable "context", and a "subContext" method for easily instantiating new contexts with that context map. What I do here is to look up the UserEntity with the given id from the Qi4j UnitOfWork, and then register it in the map with the given roles. The object already has those roles, so the only thing that happens here is that the map knows that if someone asks for the object playing the "Assignee" role, it knows what to return.
* Once the context has been looked up it is time to invoke the interaction, with arguments:
* [inboxContext.assignTo](http://inboxcontext.assignto)( task );
* Since the context has access to the context map the above method doesn't need to know who to assign it to. That is given by the context map! The implementation of assignTo() is as follows:
* public class InboxContext   extends Context {
* public void assignTo( Assignable assignable )
* {
* [context.role](http://context.role)( [Assignments.class](http://assignments.class)).assignTo( assignable, [context.role](http://context.role)( [Assignee.class](http://assignee.class) ));
* }
* }
* The assignTo() interaction uses the context map to look up the objects bound to the roles Assignments and Assignee, and invokes the assignTo method given these objects. In the above you therefore see exactly how the context, interaction and roles interact to implement a given usecase.
* class AssignmentsMixin  implements Assignments
* @This AssignmentsData data;
* public void assignTo( Assignable assignable, Assignee assignee )
* {
* [assignable.assignTo](http://assignable.assignto)( assignee );
* [data.assignments](http://data.assignments)().add( assignable );
* }
* public Iterable<Assignable> assignments() {
* return [data.assignments](http://data.assignments)();
* }
* The AssignmentsMixin implementation assigns the Assignable to the Assignee, and then adds it to the list of assignments. Nowhere in this code do we see that we are talking about Users, Tasks or Projects. It is all related to the roles related to handling assignment. This allows us to focus on one thing at a time, and makes it clear what the boundaries are between various algorithms and roles that interact in our system as a whole.
* The @This injection is what provides the private mixin support. The field will be injected with a reference to "this object", cast to the  "AssignmentsData", which is a mixin that holds the data for managing  assignments. This cannot be reached from the outside of the entity, however. What we want is to ensure that all access to data of entities are accessed through our roles. This helps keep our state encapsulated.
* If another algorithm also needs to use the same state, then all it has to  do is perform the same injection. This way that other functionality, for some other usecase, can be kept separate from this AssignmentsMixin, so that each mixin deals with one thing, and one thing only.
* Now that you know how to assign a task to a user, let's try switching things around: a task will be assigned to a user, but within the context of a specific project. The code to do this looks like this:
* InteractionContext stack = new InteractionContext();
* RootContext context = [assembler.objectBuilderFactory](http://assembler.objectbuilderfactory)().newObjectBuilder( [RootContext.class](http://rootcontext.class) ).use( stack ).newInstance();
* [context.user](http://context.user)( [user.identity](http://user.identity)().get() ).project( [project.identity](http://project.identity)().get() ).inbox().assignTo( task2 );
* In this case, instead of letting the user play both the Assignee and  Assignments roles, the user will only be used for the Assignee. The  project which is looked up in the project() call will be bound to  Assignments, so that once we get to assignTo() in the InboxContext, the  algorithm will essentially say: "Assign the Task to the User within the  Projects assignment". And to do this we did not have to change any code in the assignments handling. The only thing that changed was what  objects were bound to what roles! Let me hear you say: "SWEET!"
* What we have seen here is a simple example of how DCI can be implemented in Qi4j, and which provides all the key ingredients needed: Roles, Data, Contexts and Interactions.
* Interactions
* So where does the use-case-specific code go? The answer is: in the object roles. The concept of roles is pretty unique to DCI – I haven’t seen it in any other major architectural style. The roles are supposed to dynamically extend the data object’s behavior with the use case-specific functionality. Since such functionality might want to operate on object’s state (like the printer above), an object role should preferably have access to the object’s internals e.g. via accessors or simple methods
* Class Book
* Role PrintedBook
* Book justData : [books.findByID](http://books.findbyid)(id);
* PrintedBook youCanPrintMe : justData extendedBy PrintedBook;
* Second, an object role can be played by objects of different classes, as long as they contain data and methods necessary to fulfill the role. In our example, we could print a magazine the same way we print a book (we should adjust the role‘s name then).
* Context
* The place where data objects are retrieved and roles assigned is called the context. This would be a rough equivalent of an application service or Clean Architecture’s use case interactor. Why only rough? Because ideally, a context should provide the roles with references to all collaborators in a use case, call one of the roles and do nothing else.
* Think about a bank transfer. We have two accounts – source account and destination account. Obviously, the account is a data object, while source account and destination account are roles. When the source account decreases its own balance, it wants to let the destination account know that it should increase its balance.
* DCI assumes that object roles should know what collaborators they have based on the context in which they execute.
* All Together
* Let’s walk through things the way control flows in the application. A user presses a button that sends a use-case-related request. An application receives that request, instantiates an appropriate context, and passes the request to it. The context retrieves all data objects necessary to fulfill the use case‘s goal and assigns appropriate roles to them. Then, it sends a message to an object role that begins a series of interactions between the objects. If there is a need to communicate something to the user, it’s also done by the roles. Once it’s complete, the user’s goal should be achieved.

This article was initially an appendix in our Reactive Programming with RxJava book. However, an introduction to monads, albeit very much related to reactive programming, didn't suit that very well. So I decided to take it out and publish this separately as a blog post. I am aware that "my very own, half correct and half complete explanation of monads" is the new "Hello, world" on programming blogs. Yet the article looks at functors and monads from a specific angle of Java data structures and libraries. Thus I thought it's worthwhile to share.

RxJava was designed and built on top of very fundamental concepts like functors, monoids, and monads. Even though Rx was modeled initially for imperative C# language and we are learning about RxJava, working on top of a similarly imperative language, the library has its roots in functional programming. You should not be surprised after you realize how compact the RxJava API is. There are pretty much just a handful of core classes, typically immutable, and everything is composed using mostly pure functions.

With a recent rise of functional programming (or functional style), most commonly expressed in modern languages like Scala or Clojure, monads became a widely discussed topic. There is a lot of folklore around them:

A monad is a monoid in the category of endofunctors, what's the problem?

James Iry

The curse of the monad is that once you get the epiphany, once you understand - "oh that's what it is" - you lose the ability to explain it to anybody.

Douglas Crockford

The vast majority of programmers, especially those without a functional programming background, tend to believe monads are some arcane computer science concept, so theoretical that it can not possibly help in their programming career. This negative perspective can be attributed to dozens of articles and blog posts being either too abstract or too narrow. But it turns out that monads are all around us, even in a standard Java library, especially since Java Development Kit (JDK) 8 (more on that later). What is absolutely brilliant is that once you understand monads for the first time, suddenly several unrelated classes and abstractions, serving entirely different purposes, become familiar.

Monads generalize various seemingly independent concepts so that learning yet another incarnation of monad takes very little time. For example, you do not have to learn how CompletableFuture works in Java 8 - once you realize it is a monad, you know precisely how it works and what can you expect from its semantics. And then you hear about RxJava which sounds so much different but because Observable is a monad, there is not much to add. There are numerous other examples of monads you already came across without knowing that. Therefore, this section will be a useful refresher even if you fail to actually use RxJava.

Functors

Before we explain what a monad is, let's explore simpler construct called a functor . A functor is a typed data structure that encapsulates some value(s). From a syntactic perspective a functor is a container with the following API:

import java.util.function.Function;

interface Functor<T> {

<R> Functor<R> map(Function<T, R> f);

}

But mere syntax is not enough to understand what a functor is. The only operation that functor provides is map() that takes a function f. This function receives whatever is inside a box, transforms it and wraps the result as-is into a second functor. Please read that carefully. Functor<T> is always an immutable container, thus map never mutates the original object it was executed on. Instead, it returns the result (or results - be patient) wrapped in a brand new functor, possibly of different type R. Additionally functors should not perform any actions when identity function is applied, that is map(x -> x). Such a pattern should always return either the same functor or an equal instance.

Often Functor<T> is compared to a box holding instance of T where the only way of interacting with this value is by transforming it. However, there is no idiomatic way of unwrapping or escaping from the functor. The value(s) always stay within the context of a functor. Why are functors useful? They generalize multiple common idioms like collections, promises, optionals, etc. with a single, uniform API that works across all of them. Let me introduce a couple of functors to make you more fluent with this API:

interface Functor<T,F extends Functor<?,?>> {

<R> F map(Function<T,R> f);

}

class Identity<T> implements Functor<T,Identity<?>> {

private final T value;

Identity(T value) { this.value = value; }

public <R> Identity<R> map(Function<T,R> f) {

final R result = f.apply(value);

return new Identity<>(result);

}

}

An extra F type parameter was required to make Identity compile. What you saw in the preceding example was the simplest functor just holding a value. All you can do with that value is transforming it inside map method, but there is no way to extract it. This is considered beyond the scope of a pure functor. The only way to interact with functor is by applying sequences of type-safe transformations:

Identity<String> idString = new Identity<>("abc");

Identity<Integer> idInt = idString.map(String::length);

Or fluently, just like you compose functions:

Identity<byte[]> idBytes = new Identity<>(customer)

.map(Customer::getAddress)

.map(Address::street)

.map((String s) -> s.substring(0, 3))

.map(String::toLowerCase)

.map(String::getBytes);

From this perspective mapping over a functor is not much different than just invoking chained functions:

byte[] bytes = customer

.getAddress()

.street()

.substring(0, 3)

.toLowerCase()

.getBytes();

Why would you even bother with such verbose wrapping that not only does not provide any added value, but also is not capable of extracting the contents back? Well, it turns out you can model several other concepts using this raw functor abstraction. For example starting from Java 8 Optional is a functor with the map() method. Let us implement it from scratch:

class FOptional<T> implements Functor<T,FOptional<?>> {

private final T valueOrNull;

private FOptional(T valueOrNull) {

this.valueOrNull = valueOrNull;

}

public <R> FOptional<R> map(Function<T,R> f) {

if (valueOrNull == null)

return empty();

else

return of(f.apply(valueOrNull));

}

public static <T> FOptional<T> of(T a) {

return new FOptional<T>(a);

}

public static <T> FOptional<T> empty() {

return new FOptional<T>(null);

}

}

Now it becomes interesting. An FOptional<T> functor may hold a value, but just as well it might be empty. It's a type-safe way of encoding null. There are two ways of constructing FOptional - by supplying a value or creating an empty() instance. In both cases, just like with Identity,FOptional is immutable and we can only interact with the value from inside. What differsFOptional is that the transformation function f may not be applied to any value if it is empty. This means functor may not necessarily encapsulate exactly one value of type T. It can just as well wrap an arbitrary number of values, just like List... functor:

import com.google.common.collect.ImmutableList;

class FList<T> implements Functor<T, FList<?>> {

private final ImmutableList<T> list;

FList(Iterable<T> value) {

this.list = ImmutableList.copyOf(value);

}

@Override

public <R> FList<?> map(Function<T, R> {

ArrayList<R> result = new ArrayList<R>(list.size());

for (T t : list) {

result.add(f.apply(t));

}

return new FList<>(result);

}

}

The API remains the same: you take a functor in a transformation - but the behavior is much different. Now we apply a transformation on each and every item in the FList, declaratively transforming the whole list. So if you have a list of customers and you want a list of their streets, it's as simple as:

import static java.util.Arrays.asList;

FList<Customer> customers = new FList<>(asList(cust1, cust2));

FList<String> streets = customers.

.map(Customer::getAddress)

.map(Address::street);

It's no longer as simple as saying customers.getAddress().street(), you can't invokegetAddress() on a collection of customers, you must invoke getAddress() on each individual customer and then place it back in a collection. By the way, Groovy found this pattern so common that it actually has a syntax sugar for that: customer\*.getAddress()\*.street(). This operator, known as spread-dot, is actually a map in disguise. Maybe you are wondering why I iterate over list manually inside map rather than using Streams from Java 8:list.stream().map(f).collect(toList())? Does this ring a bell? What if I told youjava.util.stream.Stream<T> in Java is a functor as well? And by the way, also a monad?

From Functors to Monads

I assume you understand how functors work and why are they a useful abstraction. But functors are not that universal as one might expect. What happens if your transformation function (the one passed as an argument to map()) returns functor instance rather than simple value? Well, a functor is just a value as well, so nothing bad happens. Whatever was returned is placed back in a functor so all behaves consistently. However imagine you have this handy method for parsing Strings:

FOptional<Integer> tryParse(String s) {

try {

final int i = Integer.parseInt(s);

return FOptional.of(i);

} catch (NumberFormatException e) {

return FOptional.empty();

}

}

Exceptions are side-effects that undermine type system and functional purity. In pure functional languages, there is no place for exceptions. After all, we never heard about throwing exceptions during math classes, right? Errors and illegal conditions are represented explicitly using values and wrappers. For example tryParse() takes a String but does not simply return an int or silently throw an exception at runtime. We explicitly tell, through the type system, that tryParse() can fail, there is nothing exceptional or erroneous in having a malformed string. This semi-failure is represented by an optional result. Interestingly Java has checked exceptions, the ones that must be declared and handled, so in some sense, Java is purer in that regard, it does not hide side-effects. But for better or worse checked exceptions are often discouraged in Java, so let's get back to tryParse(). It seems useful to compose tryParse with String already wrapped in FOptional:

FOptional<String> str = FOptional.of("42");

FOptional<FOptional<Integer>> num = str.map(this::tryParse);

That should not come as a surprise. If tryParse() would return an int you would getFOptional<Integer> num, but because map() function returns FOptional<Integer> itself, it gets wrapped twice into awkward FOptional<FOptional<Integer>>. Please look carefully at the types, you must understand why we got this double wrapper here. Apart from looking horrible, having a functor in functor ruins composition and fluent chaining:

FOptional<Integer> num1 = //...

FOptional<FOptional<Integer>> num2 = //...

​

FOptional<Date> date1 = num1.map(t -> new Date(t));

​

//doesn't compile!

FOptional<Date> date2 = num2.map(t -> new Date(t));

Here we try to map over the contents of FOptional by turning int into +Date+. Having a function of int -> Date we can easily transform from Functor<Integer> to Functor<Date>, we know how it works. But in case of num2 the situation becomes complicated. What num2.map()receives as input is no longer an int but an FOoption<Integer> and obviouslyjava.util.Date does not have such a constructor. We broke our functor by double wrapping it. However having a function that returns a functor rather than simple value is so common (liketryParse()) that we can not simply ignore such requirement. One approach is to introduce a special parameterless join() method that "flattens" nested functors:

FOptional<Integer> num3 = num2.join()

It works but because this pattern is so common, special method named flatMap() was introduced. flatMap() is very similar to map but expects the function received as an argument to return a functor - or monad to be precise:

interface Monad<T,M extends Monad<?,?>> extends Functor<T,M> {

M flatMap(Function<T,M> f);

}

We simply concluded that flatMap is just a syntactic sugar to allow better composition. ButflatMap method (often called bind or >>= from Haskell) makes all the difference since it allows complex transformations to be composed in a pure, functional style. If FOptional was an instance of monad, parsing suddenly works as expected:

FOptional<String> num = FOptional.of("42");

FOptional<Integer> answer = num.flatMap(this::tryParse);

Monads do not need to implement map, it can be implemented on top of flatMap() easily. As a matter of fact flatMap is the essential operator that enables a whole new universe of transformations. Obviously just like with functors, syntactic compliance is not enough to call some class a monad, the flatMap() operator has to follow monad laws, but they are fairly intuitive like associativity of flatMap() and identity. The latter requires that m(x).flatMap(f) is the same asf(x) for any monad holding a value x and any function f. We are not going to dive too deep into monad theory, instead let's focus on practical implications. Monads shine when their internal structure is not trivial, for example Promise monad that will hold a value in the future. Can you guess from the type system how Promise will behave in the following program? First, all methods that can potentially take some time to complete return a Promise:

import java.time.DayOfWeek;

Promise<Customer> loadCustomer(int id) {

//...

}

Promise<Basket> readBasket(Customer customer) {

//...

}

Promise<BigDecimal> calculateDiscount(Basket basket, DayOfWeek dow) {

//...

}

We can now compose these functions as if they were all blocking using monadic operators:

Promise<BigDecimal> discount =

loadCustomer(42)

.flatMap(this::readBasket)

.flatMap(b -> calculateDiscount(b, DayOfWeek.FRIDAY));

This becomes interesting. flatMap() must preserve monadic type therefore all intermediate objects are Promises. It is not just about keeping the types in order - preceding program is suddenly fully asynchronous! loadCustomer() returns a Promise so it does not block. readBasket() takes whatever the Promise has (will have) and applies a function returning another Promise and so on and so forth. Basically, we built an asynchronous pipeline of computation where the completion of one step in the background automatically triggers next step.

Exploring flatMap()

It is very common to have two monads and combining the value they enclose together. However, both functors and monads do not allow direct access to their internals, which would be impure. Instead, we must carefully apply transformation without escaping the monad. Imagine you have two monads and you want to combine them:

import java.time.LocalDate;

import java.time.Month;

Monad<Month> month = //...

Monad<Integer> dayOfMonth = //...

​

Monad<LocalDate> date = month.flatMap((Month m) ->

dayOfMonth

.map((int d) -> LocalDate.of(2016, m, d)));

Please take your time to study the preceding pseudo-code. I don't use any real monad implementation like Promise or List to emphasize the core concept. We have two independent monads, one of type Month and the other of type Integer. In order to build LocalDate out of them, we must build a nested transformation that has access to the internals of both monads. Work through the types, especially making sure you understand why we use flatMap in one place andmap() in the other. Think how you would structure this code if you had a third Monad<Year> as well. This pattern of applying a function of two arguments (m and d in our case) is so common that in Haskell there is special helper function called liftM2 that does exactly this transformation, implemented on top of map and flatMap. In Java pseudo-syntax it would look somewhat like this:

Monad<R> liftM2(Monad<T1> t1, Monad<T2> t2, BiFunction<T1, T2, R> fun) {

return t1.flatMap((T1 tv1) ->

t2.map((T2 tv2) -> fun.apply(tv1, tv2))

);

}

You don't have to implement this method for every monad, flatMap() is enough, moreover, it works consistently for all monads. liftM2 is extremely useful when you consider how it can be used with various monads. For example, listM2(list1, list2, function) will apply function on every possible pair of items from list1 and list2 (Cartesian product). On the other hand, for optionals it will apply a function only when both optionals are non-empty. Even better, for a Promise monad a function will be executed asynchronously when both Promises are completed. This means we just invented a simple synchronization mechanism (join() in fork-join algorithms) of two asynchronous steps.

Another useful operator that we can easily build on top of flatMap() is filter(Predicate<T>)which takes whatever is inside a monad and discards it entirely if it does not meet certain predicate. In a way, it is similar to map but rather than 1-to-1 mapping we have 1-to-0-or-1. Again filter()has the same semantics for every monad but quite amazing functionality depending on which monad we actually use. Obviously, it allows filtering out certain elements from a list:

FList<Customer> vips =

customers.filter(c -> c.totalOrders > 1\_000);

But it works just as well e.g. for optionals. In that case, we can transform non-empty optional into an empty one if the contents of the optional do not meet some criteria. Empty optionals are left intact.

From List of Monads to Monad of List

Another useful operator that originates from flatMap() is sequence(). You can easily guess what it does simply by looking at type signature:

Monad<Iterable<T>> sequence(Iterable<Monad<T>> monads)

Often we have a bunch of monads of the same type and we want to have a single monad of a list of that type. This might sound abstract to you, but it is impressively useful. Imagine you wanted to load a few customers from the database concurrently by ID so you used loadCustomer(id) method several times for different IDs, each invocation returning Promise<Customer>. Now you have a list of Promises but what you really want is a list of customers, e.g. to be displayed in the web browser. The sequence() (in RxJava sequence() is called concat() or merge(), depending on use-case) operator is built just for that:

FList<Promise<Customer>> custPromises = FList

.of(1, 2, 3)

.map(database::loadCustomer);

​

Promise<FList<Customer>> customers = custPromises.sequence();

customers.map((FList<Customer> c) -> ...);

Having an FList<Integer> representing customer IDs we map over it (do you see how it helps that FList is a functor?) by calling database.loadCustomer(id) for each ID. This leads to a rather inconvenient list of Promises. sequence() saves the day, but once again this is not just a syntactic sugar. The preceding code is fully non-blocking. For different kinds of monads sequence() still makes sense, but in a different computational context. For example, it can change FList<FOptional<T>> into FOptional<FList<T>>. And by the way, you can implementsequence() (just like map()) on top of flatMap().

This is just the tip of the iceberg when it comes to the usefulness of flatMap() and monads in general. Despite coming from rather an obscure category theory, monads proved to be extremely useful abstraction even in object-oriented programming languages such as Java. Being able to compose functions returning monads is so universally helpful that dozens of unrelated classes follow monadic behavior.

Moreover, once you encapsulate data inside a monad, it is often hard to get it out explicitly. Such an operation is not part of the monad behavior and often leads to non-idiomatic code. For example, Promise.get() on Promise<T> can technically return T, but only by blocking, whereas all operators based on flatMap() are non-blocking. Another example is FOptional.get(), but that can fail because FOptional may be empty. Even FList.get(idx) that peeks particular element from a list sounds awkward because you can replace for loops with map() quite often.

I hope you now understand why monads are so popular these days. Even in an object-oriented(-ish) language like Java, they are quite a useful abstraction.