
Monadische Transaktionen
...
In Java


Monadic transactions
...
In Java

Gregor Trefs

33 years old

Team Lead **@LivePerson**

Organizer of **@majug**

Founder of **@swkrheinneckar**

twitter/github: **gtrefs**

12 FEB

Mittwoch, 12. Februar 2014

Java 8 Streams API

5 APR

Dienstag, 5. April 2016

Frege - konsequent funktionale Programmierung für die JVM

7 APR

Freitag, 7. April 2017

Special Day : Funktionale Programmierung

12 APR

Mittwoch, 12. April 2017

JVM Functional Language Battle

19 OKT

Donnerstag, 19. Oktober 2017

Vavr - Objekt-funktionales Java ohne Schmerzen

23 MAI

Donnerstag, 23. Mai 2019

Functional Core für einen seiteneffektfreien Anwendungskern

Who knows what

- a function is?
- a lambda expression is?
- a higher order function is?
- a monad is?

```java
public void save(T entity) {
    beginTransaction();
    entityManager.persist(entity);
    commitTransaction();
}

private void beginTransaction() {
    try {
        entityManager.getTransaction().begin();
    } catch (IllegalStateException e) {
        rollBackTransaction();
    }
}

private void commitTransaction() {
    try {
        entityManager.getTransaction().commit();
    } catch (IllegalStateException | RollbackException e) {
        rollBackTransaction();
    }
}
```

A typical repository

Move a scattered and/or repeated responsibility into one single method

Execute around method pattern

```java
public void remove(int id) {
   transactional(em -> em.remove(find(id)));
}

public void update(int id, Consumer<T>... updates) throws Exception {
   T entity = find(id);
   transactional(em -> Arrays.stream(updates).forEach(up -> up.accept(entity)));
}

private void transactional(Consumer<EntityManager> action) {
   try {
       entityManager.getTransaction().begin();
       action.accept(entityManager);
       entityManager.getTransaction().commit();
   } catch (RuntimeException e) {
       entityManager.getTransaction().rollback();
       throw e;
   }
}
```

Execute around method pattern

Concise code

Execute around method pattern

Lowers the risk of incorrect transaction management

Execute around method pattern

Transaction execution is a side effect

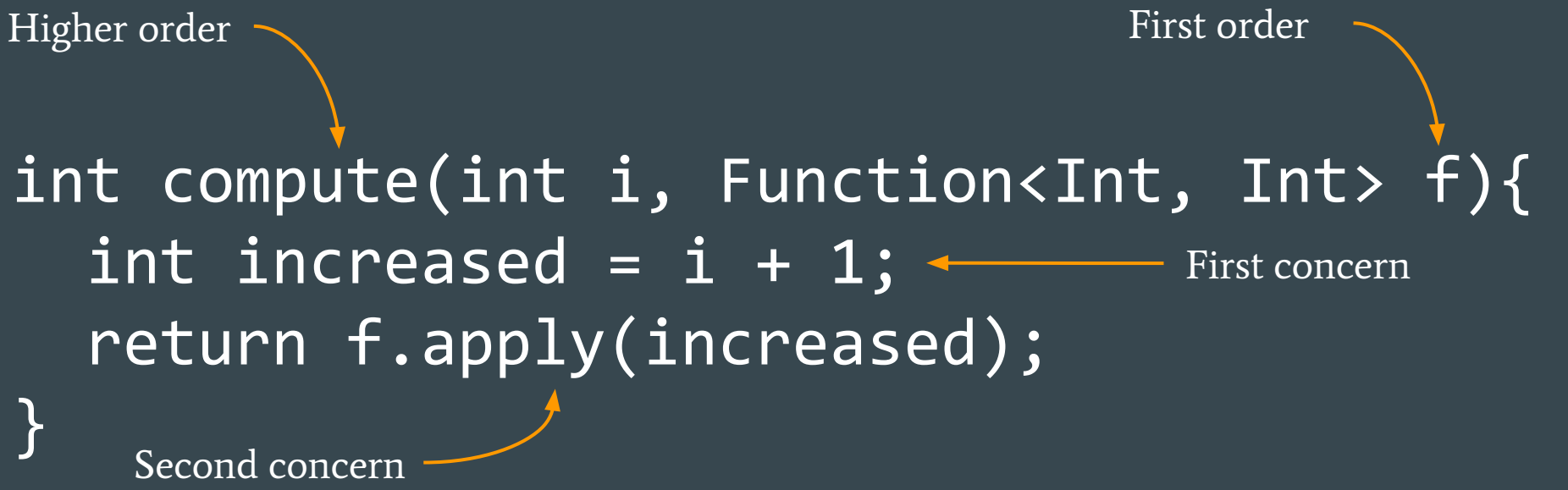
Execute around method pattern

Difficult code reuse: update and remove in the same transaction?

Execute around method pattern

```java
public void remove(int id) {
   transactional(em -> em.remove(find(id)));
}

public void update(int id, Consumer<T>... updates) throws Exception {
   T entity = find(id);
   transactional(em -> Arrays.stream(updates).forEach(up -> up.accept(entity)));
}

private void transactional(Consumer<EntityManager> action) {
   try {
       entityManager.getTransaction().begin();
       action.accept(entityManager);
       entityManager.getTransaction().commit();
   } catch (RuntimeException e) {
       entityManager.getTransaction().rollback();
       throw e;
   }
}
```

Execute around method pattern



Higher order functions

Describe a transaction and delay its execution

Higher order functions

```java
public T convert(int id, UnaryOperator<T> converter){
    Function<EntityManager, T> find = em -> em.find(entityType, id);
    Function<EntityManager, T> transaction = transactional(find.andThen(converter));
    return transaction.apply(entityManager);
}

private <U> Function<EntityManager, U> transactional(Function<EntityManager, U> action){
    return em -> {
        try {
            em.getTransaction().begin();
            final U result = action.apply(em);
            em.getTransaction().commit();
            return result;
        } catch (RuntimeException e) {
            em.getTransaction().rollback();
            throw e;
        }
    };
}
```

Higher order functions

Compose descriptions without side effects

Higher order functions

Run the description with an entity manager

Database is determined upon execution

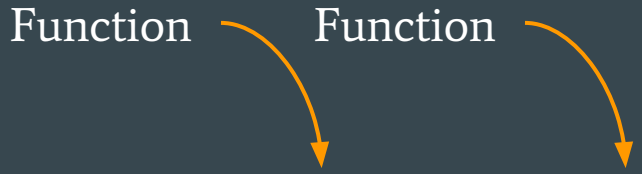
```
public Function<EntityManager, T> convert(int id, UnaryOperator<T> converter){
   return transactional(find(id).andThen(converter));
}

public Function<EntityManager, T> find(int id){
   return em -> em.find(entityType, id);
}

private <U> Function<EntityManager, U> transactional(Function<EntityManager, U> action){
   return em -> {
       try {
           em.getTransaction().begin();
           final U result = action.apply(em);
           em.getTransaction().commit();
           return result;
       } catch (RuntimeException e) {
           em.getTransaction().rollback();
           throw e;
       }
   };
}
```

Higher order functions

`Function` does not care about transactions

```
public Function<EntityManager, T> transactionalConvert(int id, UnaryOperator<T> converter){
   return transactional(transactional(find(id)).andThen(converter));
}

public Function<EntityManager, T> find(int id){
   return em -> em.find(entityType, id);
}

private <U> Function<EntityManager, U> transactional(Function<EntityManager, U> action){
   return em -> {
       try {
           em.getTransaction().begin();
           final U result = action.apply(em);
           em.getTransaction().commit();
           return result;
       } catch (RuntimeException e) {
           em.getTransaction().rollback();
           throw e;
       }
   };
}
```

Higher order functions



A `Transaction` type to put an action in, to combine it with others and to run it

Is `Transaction` a monad?

Functional languages are based on the lambda calculus

Applying conversions on expressions

$\lambda x.x$

How to integrate side effects and stay pure?

Functional programming languages

Don’t: Lisp (Clojure, Scheme), Standard ML

```
(println (read-line))
```

Functional programming languages

“Or I could use a monad” -- Philip Wadler

Functional programming languages

“‘In order to understand monads you first need to learn category theory’ is like saying ‘In order to understand Pizza you first need to learn Italian.’” -- Mario Fusco (Italian)

Functional programming languages

A monad of type `M` represents some computation: I/O, potential absent values, values available in the future, lists, etc.

A function to turn a value into a computation that produces the value

```
M<T> of(T value)
```

A function to combine computations

```
M<U> flatMap(M<T> m, Function<T, M<U>> f)
```

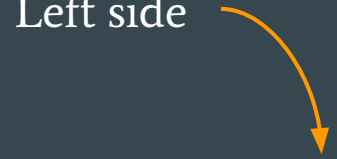
```
M<String> hello = of("hello");

Function<String, M<String>> world = str -> of(str + "World");

M<String> helloWorld = flatMap(hello, world);
```

Monads

A monad is a tuple `(M, of, flatMap)`

Monad laws describe how operations relate to each other and, thus, make reasonable assumptions about their behavior

Identity laws state that the the value put into the computation should not be changed




```
flatMap(of(v), f) == f.apply(v)
  flatMap(m, v -> of(v)) == m
```

The associativity law states that the order of combining monads should not matter

```
flatMap(flatMap(m, f), g) ==
flatMap(m,v -> flatMap(f.apply(v), g))
```

Monads

Laws do not provide a mental model what a monad is or what a monad means

We are used to perceive interfaces as a generalization of specific representations

Interface `List` is a generalization of `ArrayList` and `LinkedList`

Monad does not generalize one type or another

A type is monadic if it has operations which satisfy the laws

The monad operations are often just a small fragment of the full API for a given type that happens to be a monad

The monad contract does not specify what is happening between the lines, only that whatever is happening satisfies the laws

CompletableFuture
Stream

```
completedFuture("hello").thenCompose(v -> completedFuture(v + "world"));

completedFuture("hello").thenComposeAsync(v -> completedFuture(v + "world"));

Stream.of("hello").flatMap(v -> Stream.of(v + "world"));
```

Monads in Java

```
userRepository.convert(10, clone).run(entityManager);
final UnaryOperator<User> clone = user -> new User(user.getName(), user.getEmail());

public Transaction<T> convert(int id, UnaryOperator<T> converter){
   return find(id).flatMap(entity -> Transaction.of(converter.apply(entity)));
}

public Transaction<T> find(int id) {
   return findById(id).apply(entityClass);
}

private Function<Class<T>, Transaction<T>> findById(int id) {
   return clazz -> Transaction.of(em -> em.find(clazz, id));
}
```

Towards a transaction monad

Group and reuse transactions

Towards a transaction monad

```java
public interface CrudTransactions<T> {

    default Transaction<Void> saveEntity(T entity) {
        return transactional(em -> em.persist(entity));
    }

    default Transaction<Void> removeEntity(T entity) {
        return transactional(em -> em.remove(entity));
    }

    default Transaction<Void> transactional(Consumer<EntityManager> action){
        return Transaction.withoutResult(action);
    }

    default Function<Class<T>, Transaction<T>> findById(int id) {
        return clazz -> Transaction.of(em -> em.find(clazz, id));
    }
}
```

Towards a transaction monad

```java
public class Repo<T> implements EntityRepository<T>, CrudTransactions<T> {
    // some code is left out
    public Transaction<T> find(int id) {
        return findById(id).apply(entityClass);
    }

    public Transaction<T> convert(int id, UnaryOperator<T> conv){
        return find(id).flatMap(entity -> Transaction.of(conv.apply(entity)));
    }

    public Transaction<Void> save(T entity) {
        return saveEntity(entity);
    }

    public Transaction<Void> remove(int id) {
        return find(id).flatMap(this::removeEntity);
    }
}
```

Towards a transaction monad

Build up reusable vocabularies to talk to the databases

Towards a transaction monad

```
UserDetails implements
Read<User>,Count<User>
```

Towards a transaction monad

Code of Transaction<T>

Towards a transaction monad

Type `Transaction` is monadic*

Knowing a type is monadic, let us reason about its behavior

For example: Exploit the associativity law and rearrange the function chaining

```
completedFuture("hello")
       .thenCompose(v -> completedFuture(v + "world"))
       .thenCompose(v -> completedFuture(v + "2017"));

completedFuture("hello").thenCompose(v ->
       completedFuture(v + "world")
       .thenCompose(w -> completedFuture(w + "2017"))
);
```

Benefits of monads

Java is a poor tool for monads

Not so cool in Java

```
describe("Combination of crud methods"){
 it("should combine findById with update"){
   val em = ???
   val user = new User("Test", "test")
   findAndUpdate(user.getId, classOf[User], _.setEmail("mail")).run(em)

   def findAndUpdate(id:Int,clazz:Class[User],updates:Consumer[User]) = for {
     entity <- findById(id)(clazz)
     update <- updateEntity(entity, updates)
   } yield update

 }
}
```

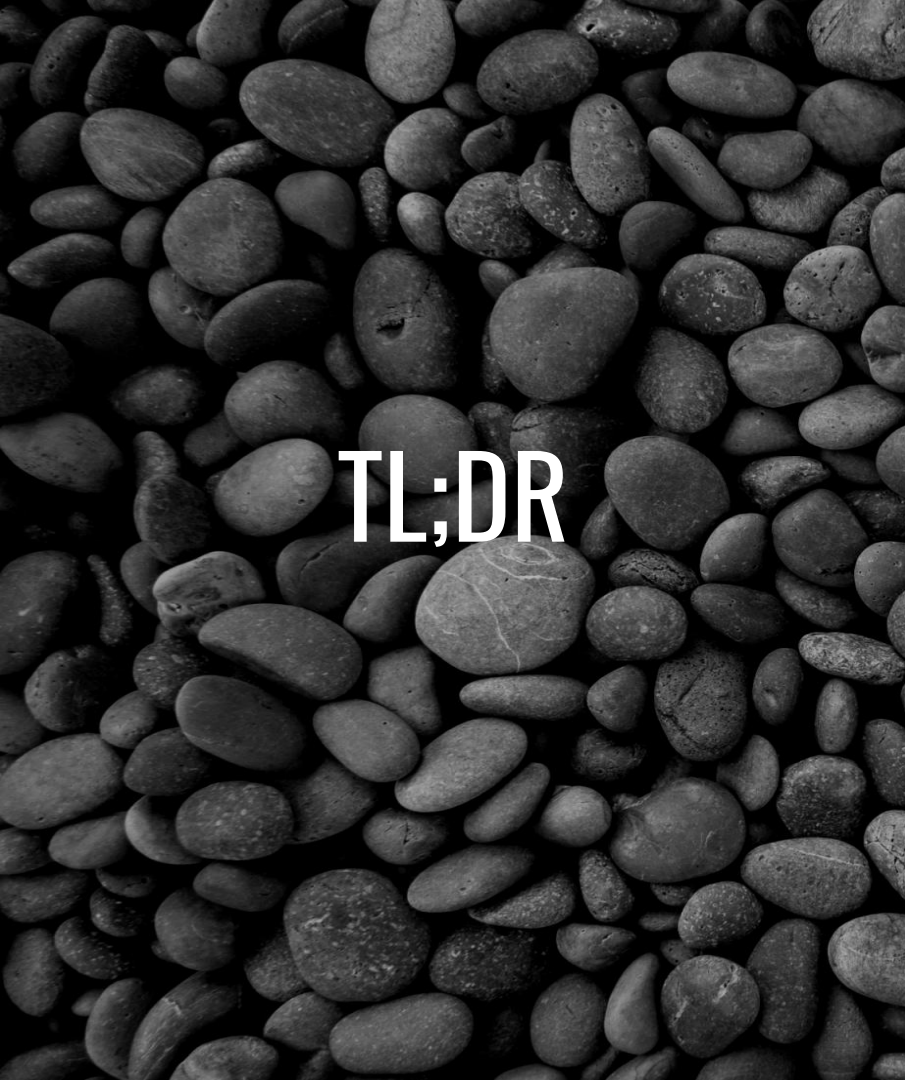
Not so cool in Java

Monads are a part of a solution to a problem that never existed in Java

Not so cool in Java

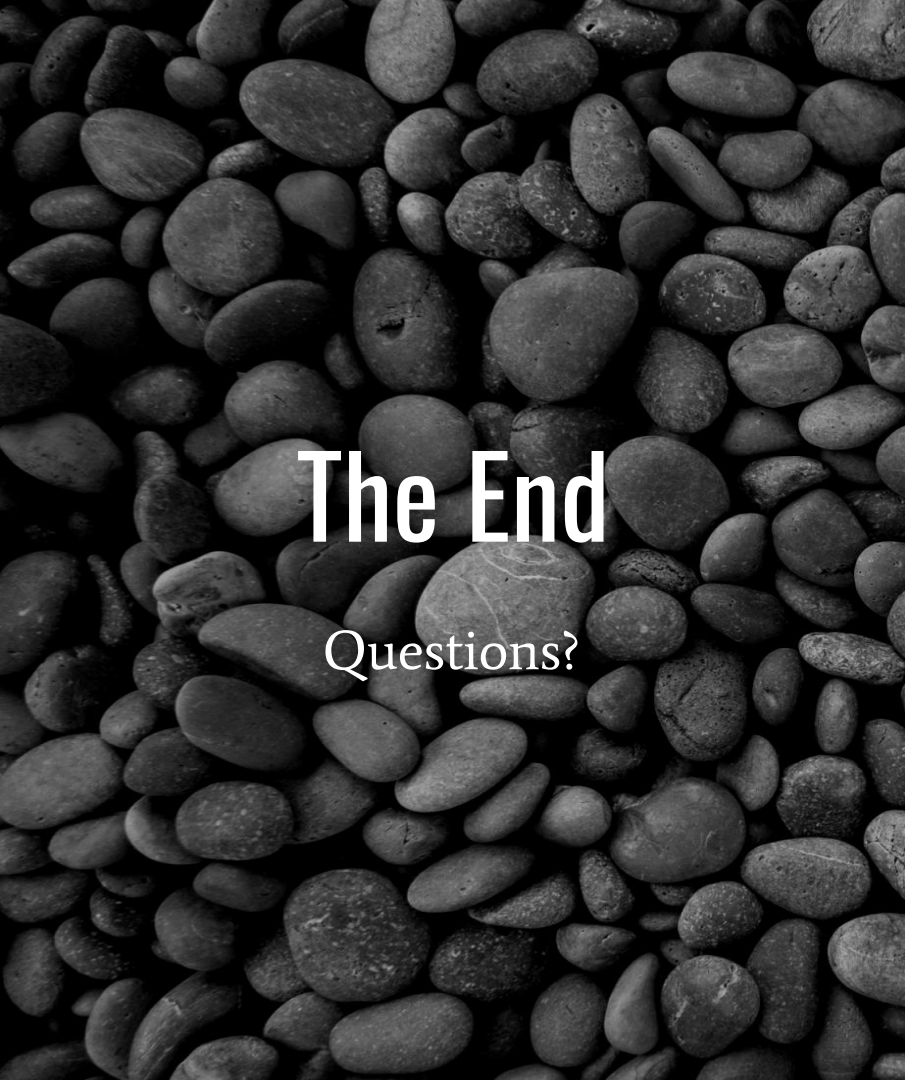
Though monadic types help us dealing with side effects in a predictable way

Not so cool in Java



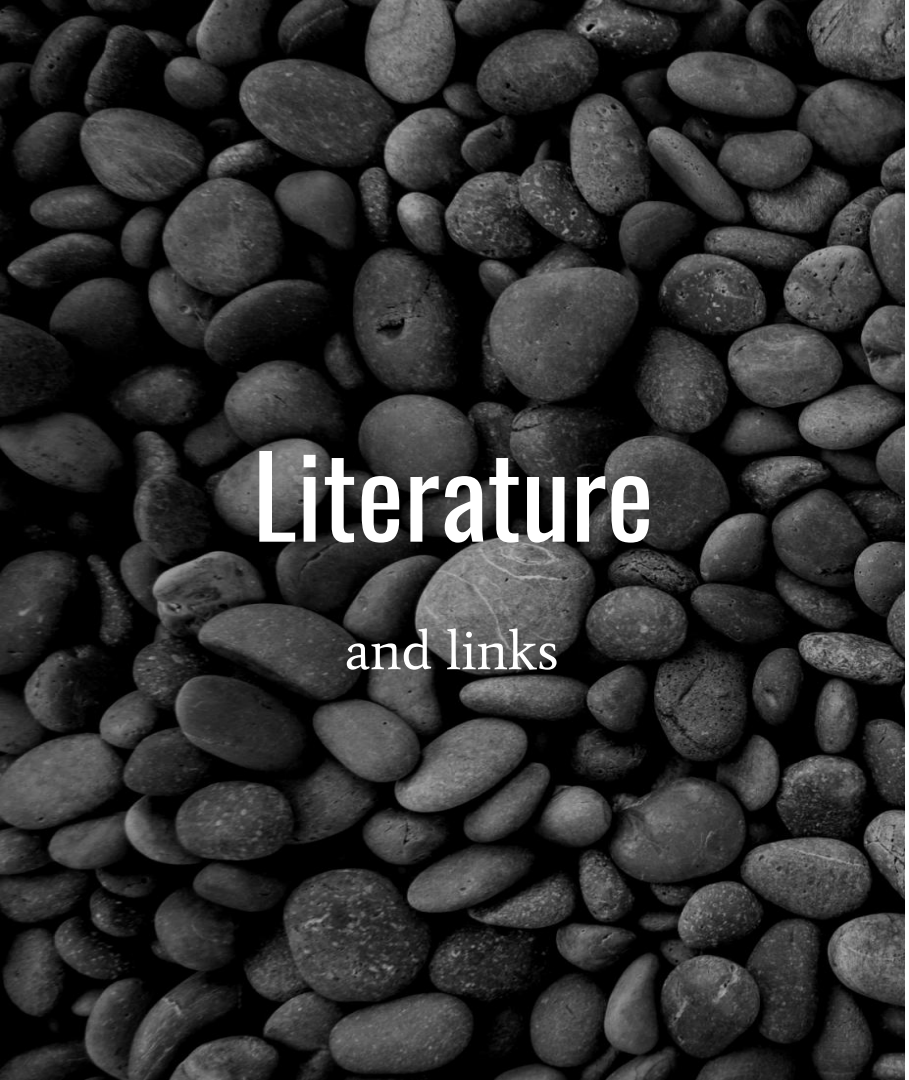
A monad
- is a triple (M, of, flatMap)
- adheres to laws
- is a *self-containing* interface
- is not well supported in Java

Transaction
- is a monadic type
- defines reusable transactions
- is a specialization of Reader



Contact

Gregor.Trefs@gmail.com
linkedin.com/in/gregor-trefs



Literature

and links

- My blog
  - https://gtrefs.github.com
- FP in Scala
  - Paul Chiusano
  - Rúnar Bjarnason
- The essence of FP
  - Philip Wadler
- Background picture
  - by John Salzarulo