Combinator als funktionales Entwurfsmuster

...

In Java 8


Combinator Pattern
...
In Java 8

About Me

Gregor Trefs

31 years old

Organizer of **@majug**

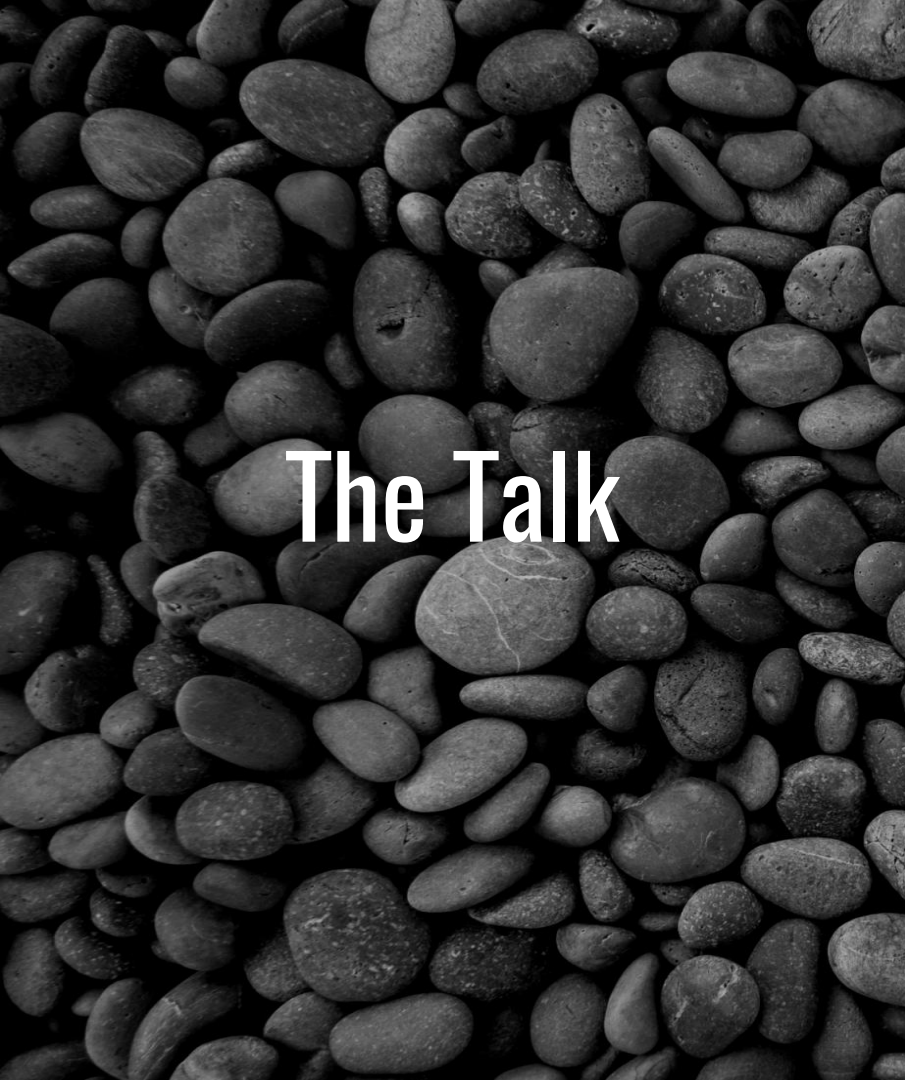
Achievement 2017: Acting lessons

twitter/github: **gtrefs**

About You

Who knows what

- a function is?
- a lambda expression is?
- a combinator is?
- the combinator pattern is?



Recap: Functions

Primitives and Combinators

Return value reasoning

Benefits and Disadvantages

When to use it

```
add :: Num a => a -> a -> a
  add = \x -> \y -> x + y
```

Recap: Functions

```
int addOne(int i){ return i+1; }
```

Recap: Functions

Deduction Context

```
Function<Integer, Integer> addOne;
addOne = i -> i + 1;
```

Lambda expression

Recap: Functions

Higher order

```
int compute(int i, Function<Int, Int> f)
```

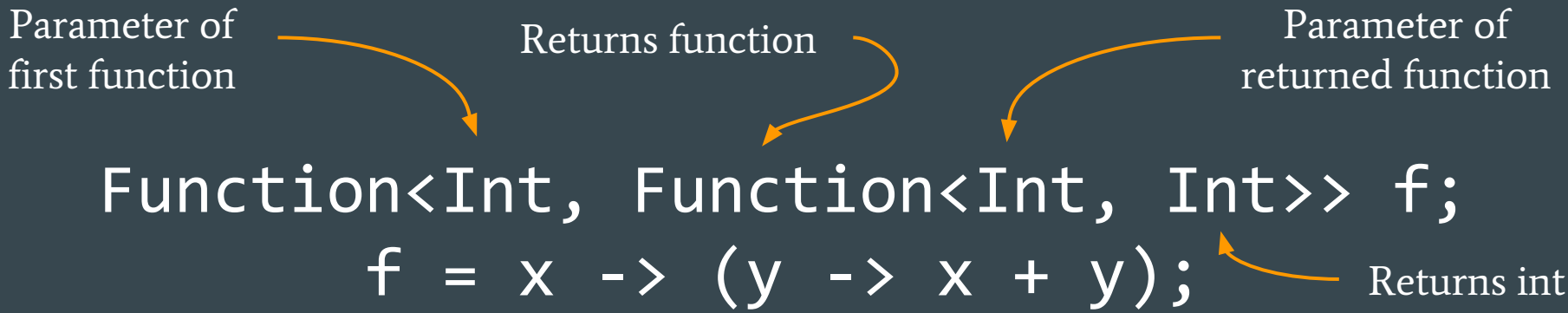
First order

Recap: Functions


```
Function<Int, Int> makeAdder(int i){
        return x -> x + i;}

Function<Int, Int> addOne = makeAdder(1)
```

Recap: Functions




```
Function<Integer, Integer> addOne =
    f.apply(1);
```

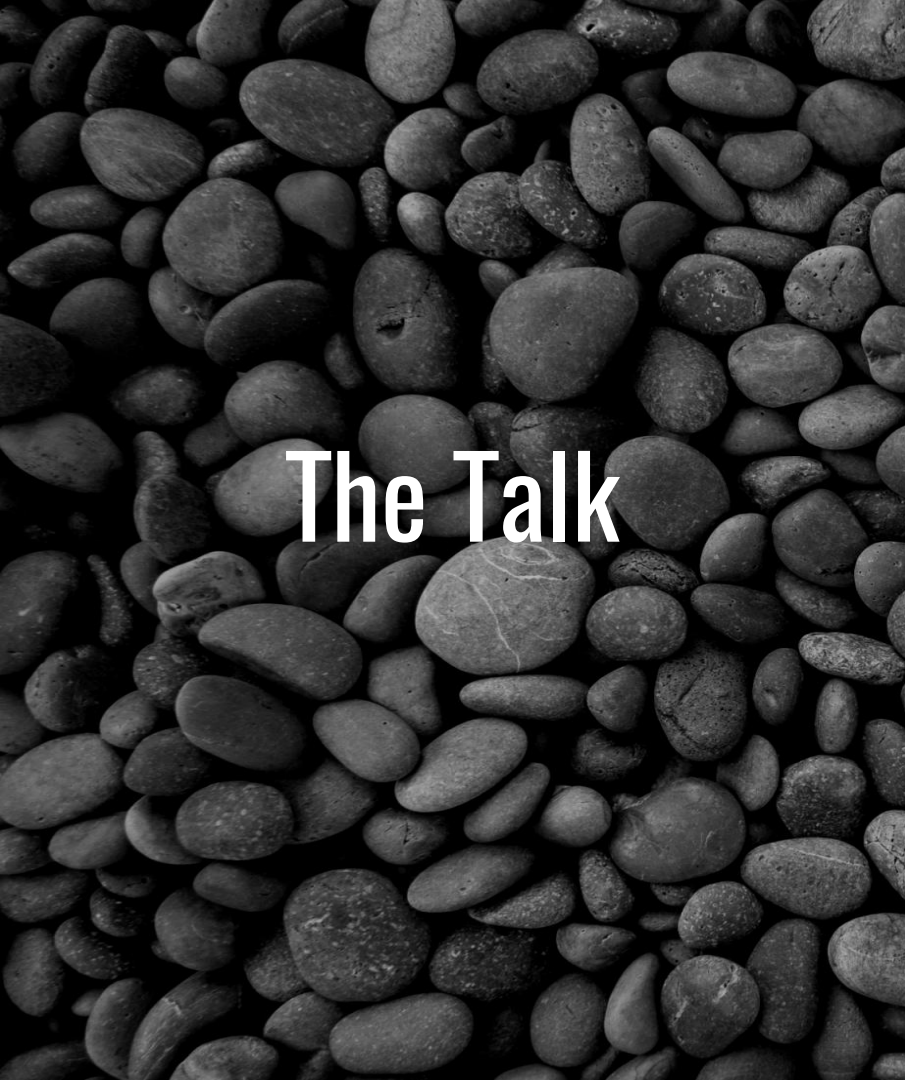
Recap: Functions

```
addOne = f.apply(1);
addTwo = f.apply(2);

Function<Int, Int> addThree =
x -> addTwo.apply(addOne.apply(x));

addFive = addThree.compose(addTwo);
```

Recap: Functions



Recap: Functions

Primitives and Combinators

Return value reasoning

Benefits and Disadvantages

When to use it

Combine primitives into more complex structures

Primitives and Combinators

Primitives are the simplest elements within a domain

Combinators compose primitives and/or domain structures into more complex domain structures

Use Case: User Validation

```
Function<User, Boolean>
```

```java
@Test
public void yield_valid_for_user_with_email_and_non_empty_name(){
   User gregor = new User("Gregor Trefs", 31, "mail@mailinator.com");

   UserValidation validation = todo();

   assertThat(validation.apply(gregor), is(true));
}

interface UserValidation extends Function<User, Boolean> {
}
```

Primitives and Combinators

```java
@Test
public void yield_valid_for_user_with_email_and_non_empty_name(){
   User gregor = new User("Gregor Trefs", 31, "mail@mailinator.com");

   UserValidation nameIsNotEmpty = user -> !user.name.trim().isEmpty();
   UserValidation mailContainsAtSign = user -> user.email.contains("@");

   UserValidation validation;
   validation = user -> nameIsNotEmpty.apply(user) && mailContainsAtSign.apply(user);

   assertThat(validation.apply(gregor), is(true));
}

interface UserValidation extends Function<User, Boolean> {
}
```

Primitives and Combinators

```java
@Test
public void yield_valid_for_user_with_email_and_non_empty_name(){
    final User gregor = new User("Gregor Trefs", 31, "mail@mailinator.com");
    final UserValidation validation = nameIsNotEmpty.and(mailContainsAtSign);

    assertThat(validation.apply(gregor), is(true));
}

public interface UserValidation extends Function<User, Boolean> {
    UserValidation nameIsNotEmpty = user -> !user.name.trim().isEmpty();
    UserValidation mailContainsAtSign = user -> user.email.contains("@");

    default UserValidation and(UserValidation other){
        return user -> this.apply(user) && other.apply(user);
    }

    default UserValidation or(UserValidation other){
        return user -> this.apply(user) || other.apply(user);
    }
}
```

Primitives and Combinators

Embedded domain specific language: Primitives and combinators from the validation domain

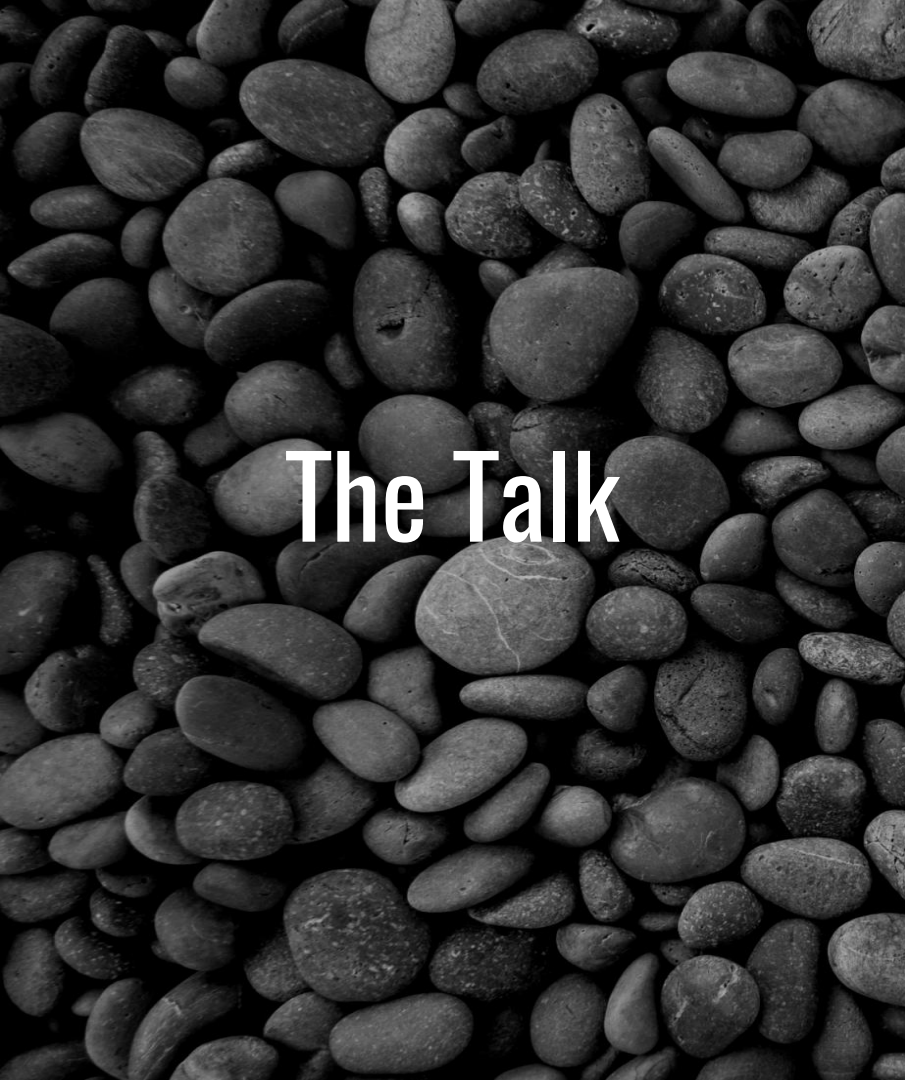
Return value reasoning

Separation of validation description and execution

Return value reasoning

Validation has no shared mutable state

Return value reasoning



Recap: Functions
Primitives and Combinators
Return value reasoning
Benefits and Disadvantages
When to use it

`Boolean` is bad for representing validation results

Return value reasoning

Hard to determine which rules invalidated the result

Return value reasoning

Semantic is implicit and context specific

Return value reasoning

Type for representing the validation result is needed

Return value reasoning

```java
@Test
public void yield_invalid_for_user_without_email(){
   User gregor = new User("Gregor Trefs", 31, "");

   ValidationResult result = nameIsNotEmpty.and(eMailContainsAtSign).apply(gregor);

   assertThat(result.getReason().get(), is("E-Mail is not valid."));
}

public interface UserValidation extends Function<User, ValidationResult> {
   UserValidation nameIsNotEmpty = todo();
   UserValidation eMailContainsAtSign = todo();

   default UserValidation and(UserValidation other){
     return todo();
   }
}
```

Return value reasoning

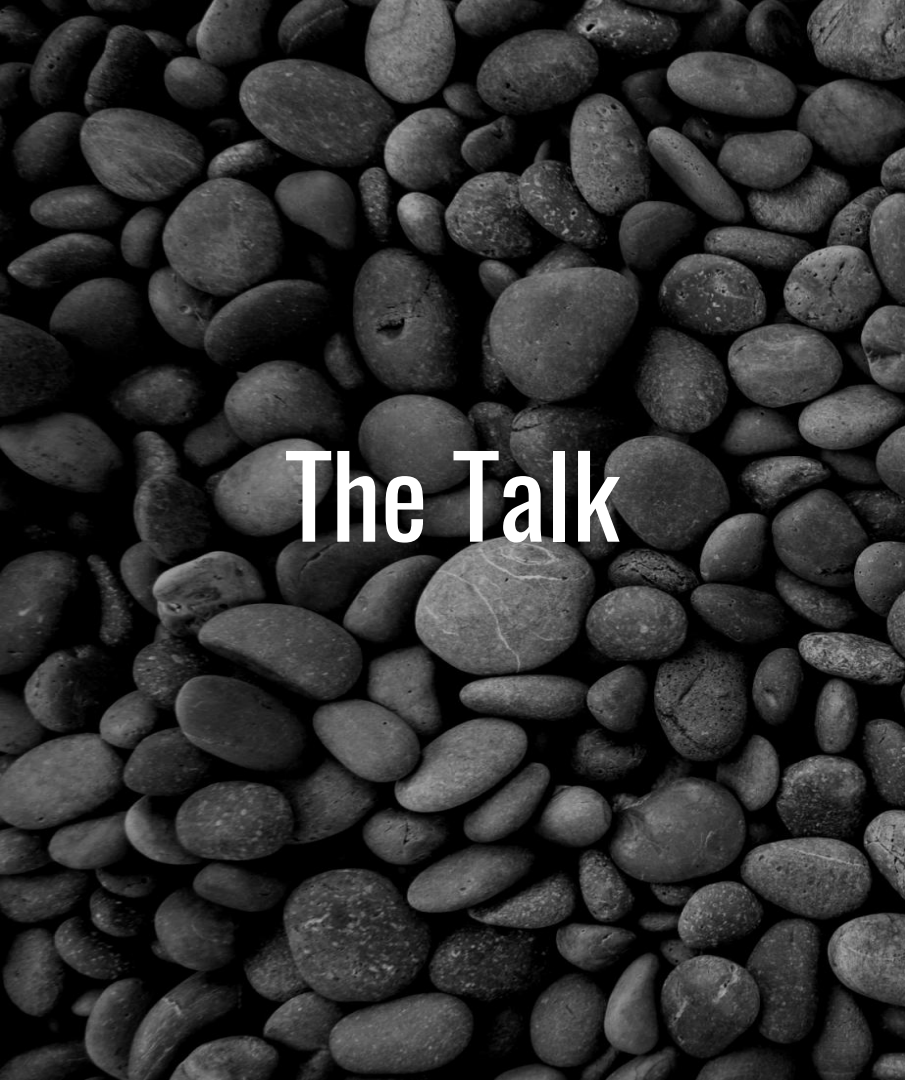
```
@Test
public void yield_invalid_for_user_without_email(){
   User gregor = new User("Gregor Trefs", 31, "");

   ValidationResult result = nameIsNotEmpty.and(eMailContainsAtSign).apply(gregor);

   assertThat(result.getReason().get(), is("E-Mail is not valid."));
}
public interface UserValidation extends Function<User, ValidationResult> {
   UserValidation nameIsNotEmpty =
     user -> !user.name.trim().isEmpty()?valid():invalid("User name is empty");
   UserValidation eMailContainsAtSign =
     user -> user.email.contains("@")?valid():invalid("E-Mail is not valid.");

   default UserValidation and(UserValidation other){
     return user -> {
         ValidationResult result = this.apply(user);
         return result.isValid() ? other.apply(user) : result;
     };
   }
}
```

Return value reasoning



Recap: Functions
Primitives and Combinators
Return value reasoning
Benefits and Disadvantages
When to use it

Domain specific approach

Benefits and Disadvantages

Implicit information is modelled explicit

Benefits and Disadvantages

Separation of concerns in primitives and composability with combinators

Benefits and Disadvantages

Extensibility by using the context

```
UserValidation ext;
ext = nameIsNotEmpty.and(u -> ...)
```

Benefits and Disadvantages

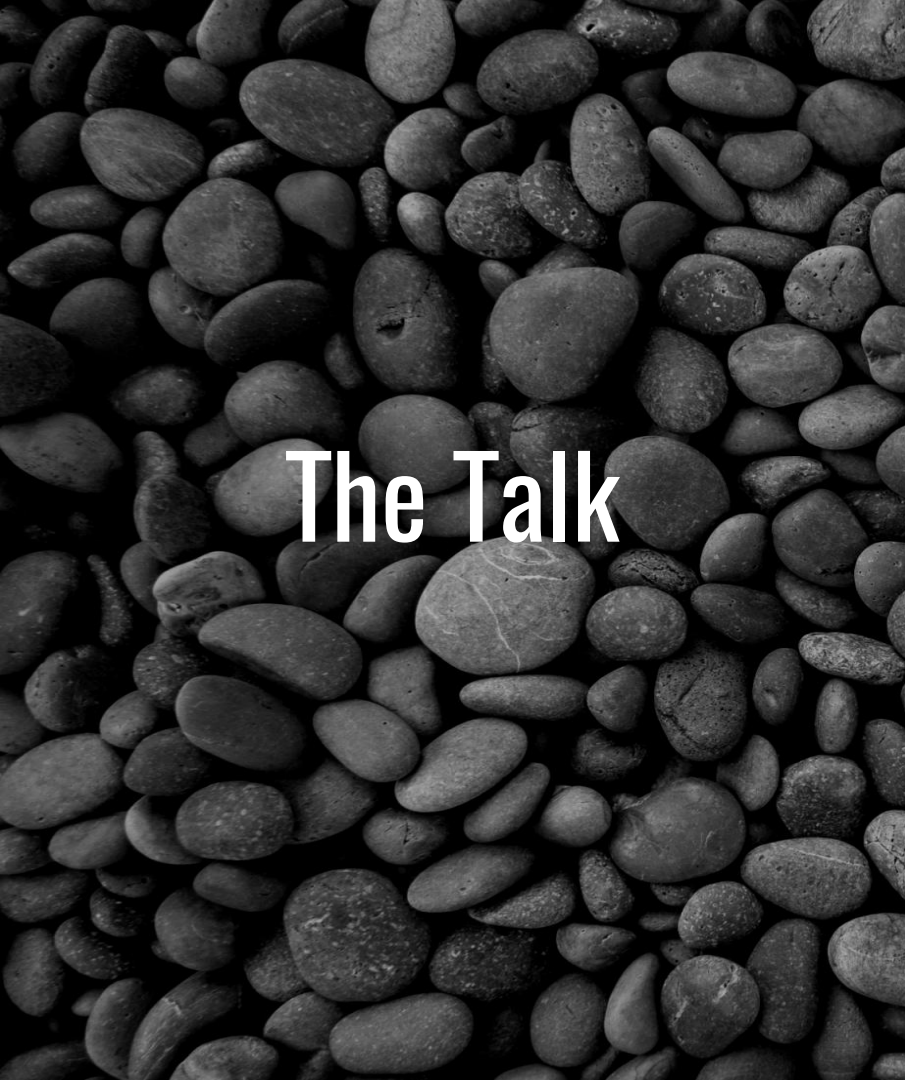
Reusability: Once described, a function can be applied manyfold

Does my team understand the concepts of functional programming?

Benefits and Disadvantages

How do I determine primitives and combinators in my domain?

Benefits and Disadvantages



Recap: Functions
Primitives and Combinators
Return value reasoning
Benefits and Disadvantages
When to use it

Design your API in a composable way

`Comparator`

When to use it

Strategy pattern: Combine your strategies

When to use it

Command pattern: Combine your commands

When to use it

Similar to the composite pattern, but:
Composes behaviour instead of structure

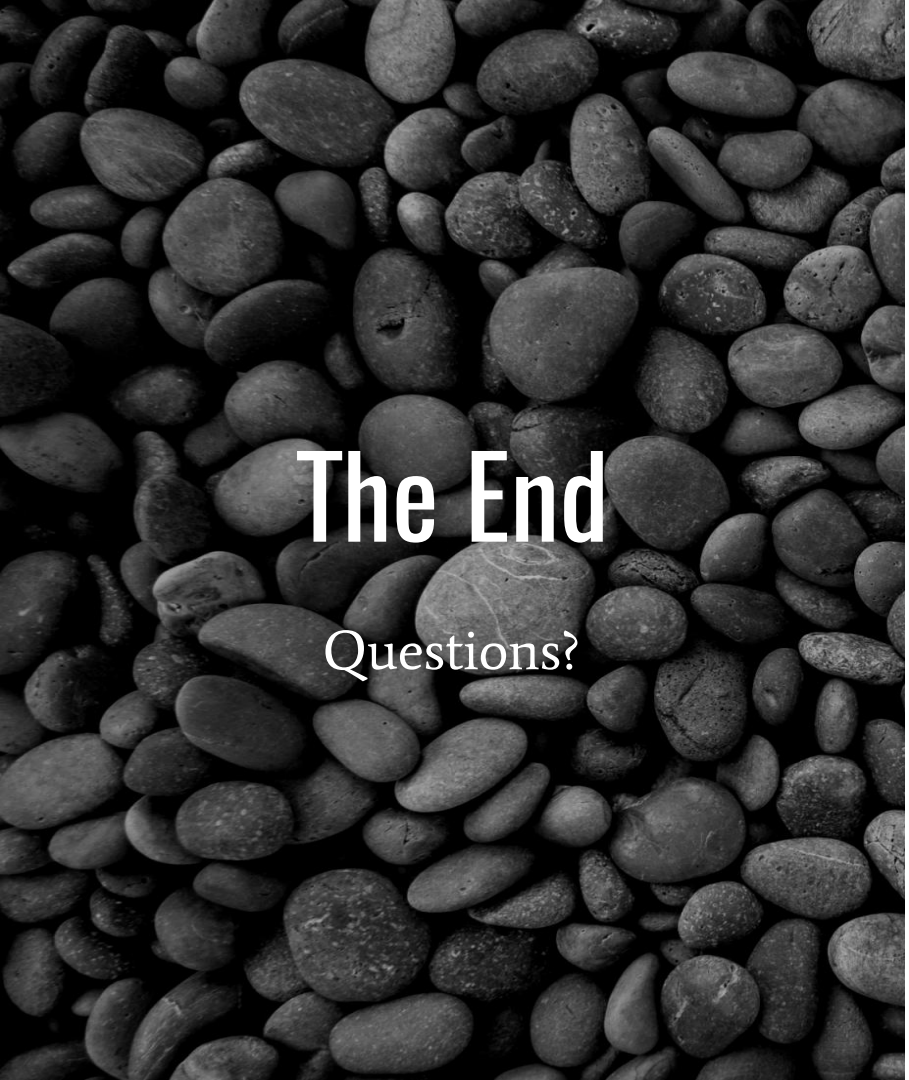
When to use it

Whenever a function is the basic concept

When to use it

```
FizzBuzz: Number -> Word
Validation: User -> ValidationResult
Parsing: String -> AST
Projection: EventStream -> Aggregate
Serialization: Object -> JSON
```

When to use it



Hire me for

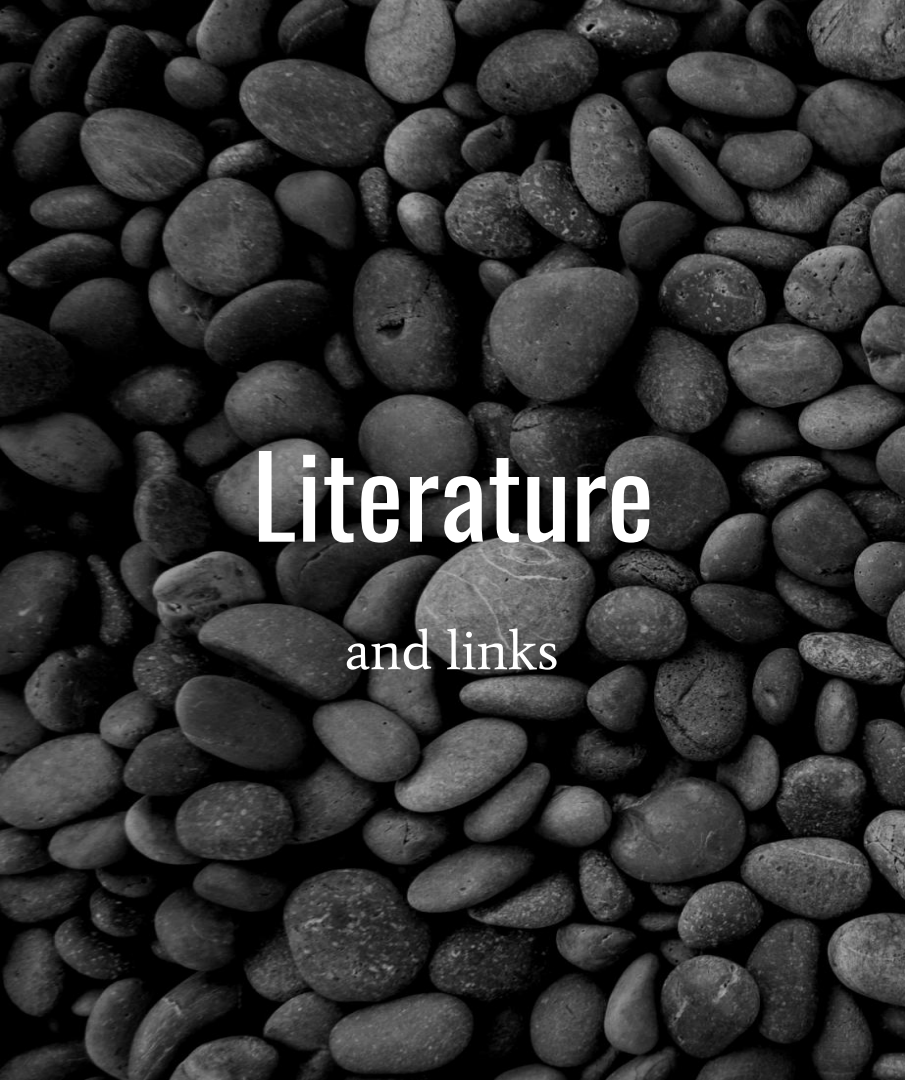
development

courses on functional Java

Contact

Gregor.Trefs@gmail.com

linkedin.com/in/gregor-trefs



Literature

and links

- My blog post about the topic
  http://bitly.com/2lryGxJ
- Functional Programming in
  Scala (The red one)
  Java (The blue one)
- Background picture by
  John Salzarulo