FUNCTIONAL - REACTIVE

WITH CORE JAVA 8 - 11 AND BEYOND

@SVENRUPPERT

vaadin}>

SVEN RUPPERT

Developer Advocate @ Vaadin

CODING JAVA SINCE 1996

DISTRIBUTED SYSTEMS SINCE 2002

CONSULTING WORLD WIDE

JOINED VAADIN 2017

ORACLE Developer Champion

Private Sector: Automotive / Aerospace / SMB /

Public Sector: Military / Government

NonProfit / NonGov: World Bank / UN / YPARD / CGIAR

vaadin}>

INTRO

DIFFERENCES BETWEEN OO AND FP IN JAVA

Java8 - Optional<T>

@SvenRuppert

```
final boolean present = stringOptional.isPresent();
stringOptional.ifPresent(s1 -> System.out.println("stored = " + s1));


final String orElseThrow = stringOptional.orElseThrow(() -> new RuntimeException("to bad.."));


final Optional<String> s1 = stringOptional.filter(Objects::isNull); // only with value if Predicate true
```

```
final Optional<Integer> flatMapOptional = stringOptional.flatMap(new Function<String, Optional<Integer>>() {
  @Override
  public Optional<Integer> apply(final String s) {
    if (Objects.isNull(s)) return Optional.empty();
    return Optional.of(s.length());
  }
});
```

the border between OO and FP in Java is not 100% clear.

Functional Interfaces - basics

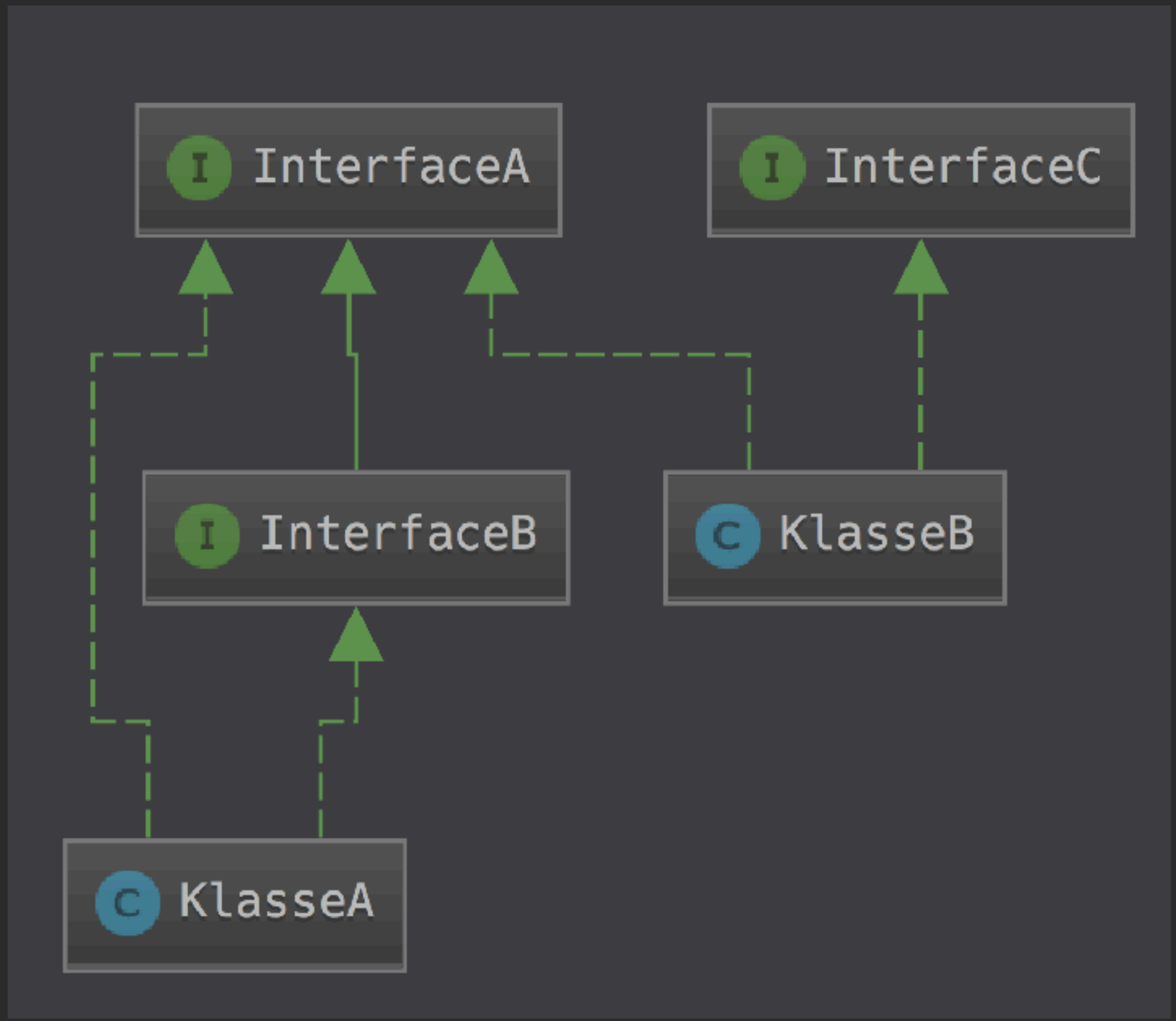
SINCE JAVA8

@SvenRuppert

```java
@FunctionalInterface
public interface DemoInterface {
    public void doSomething();
}
```

```java
@FunctionalInterface
public interface InterfaceA {
    public String doMoreB();

    public default String doMoreA(){
        return "aaeettsch";
    }
}
```

Java8 - Functional Interfaces



Java8 - Functional Interfaces

@SvenRuppert

```
public interface InterfaceA {

    public static void doMore() { System.out.println(" doMore "); }

    public default void doSomething() { System.out.println(" InterfaceA -> doSomething"); }
}

public interface InterfaceB extends InterfaceA {
    public default void doSomething() { System.out.println(" InterfaceB -> doSomething"); }
}

public interface InterfaceC {
    public  default void doSomething() { System.out.println(" InterfaceC -> doSomething"); }
}
```

```
public class KlasseA implements InterfaceA, InterfaceB {

}
```

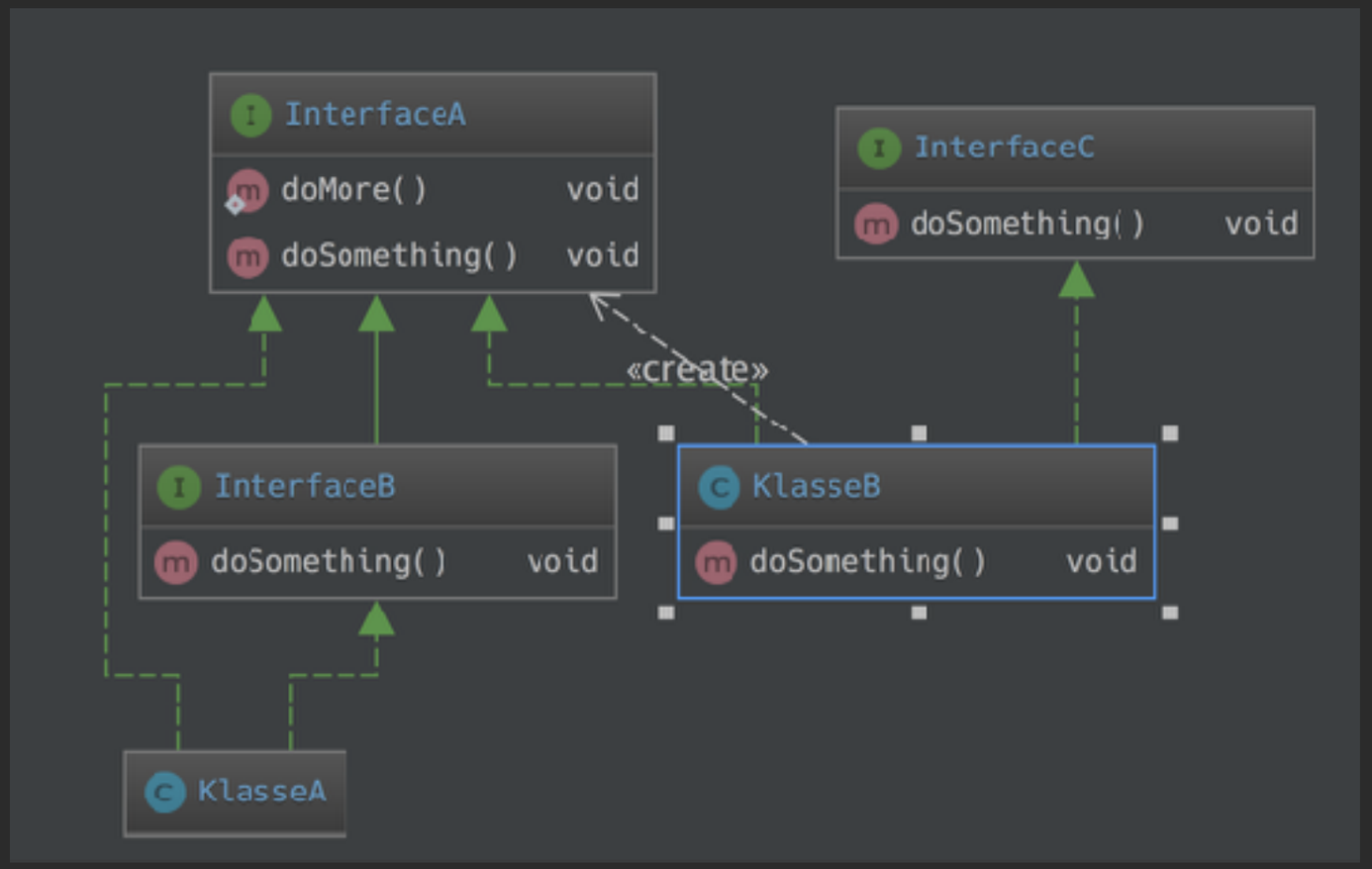
```
public class KlasseB implements InterfaceA, InterfaceC {

    @Override
    public void doSomething() {
        InterfaceA a = new InterfaceA() {};
        a.doSomething();
    }
}
```

```
KlasseA a = new KlasseA();
a.doSomething();
//KlasseA.doMore();

final KlasseB b = new KlasseB();
b.doSomething();

InterfaceA.doMore();
```

Java8 - Functional Interfaces

@SvenRuppert

```
KlasseA a = new KlasseA();
a.doSomething();
//KlasseA.doMore();

final KlasseB b = new KlasseB();
b.doSomething();

InterfaceA.doMore();
```




```
InterfaceB -> doSomething
InterfaceA -> doSomething
doMore
```

Java9 - JEP213

```
@FunctionalInterface
public interface Service {

  default void workOnDefault() {
    workA();
    workB();
    workC();
    //somethingHidden();
  }

  void workA();

  private void workB() { System.out.println("workB"); }

  private static void workC() { System.out.println("workC"); }

  //private void somethingHidden(); // declaration only not possible
}
```

EXERCISES

Functional Interfaces - basics

SINCE JAVA8

@SvenRuppert

Optional<T>

SINCE JAVA8

@SvenRuppert

Optional<T>

@SvenRuppert

```
final Optional<Object> empty = Optional.empty();
final Optional<String> hello = Optional.of("Hello");
final Optional<Object> nothing = Optional.ofNullable(null);
```

```
// since 8
final boolean present = hello.isPresent();
final String value = hello.get();

//since 9
final Stream<String> stream = hello.stream();
```

Optional<T>

@SvenRuppert

```
Optional
        .of(1)
        .ifPresentOrElse(
                value -> System.out.println("integer = " + value) ,
                () -> System.out.println("nothing = " + nothing));

hello.ifPresent(System.out::println);
```

```
final Optional<String> or = hello.or(() -> Optional.of("something else"));

final String orElse = hello.orElse( other: "something else");
final String orElseThrow = hello.orElseThrow(() -> new RuntimeException("and go.."));
```

Optional<T>

@SvenRuppert

```
final Optional<Integer> mappedA = hello.map(input -> Integer.valueOf(input));
final Optional<Integer> mappedB = hello.map(Integer::valueOf);

final Optional<String> filtered = hello.filter(s -> s.contains("H"));

final Optional<Integer> flatMaped = hello
    .flatMap(s -> Optional.of(Integer.parseInt(s)));
```

Optional<T> - since JDK10

@SvenRuppert

```
public T orElseThrow() {
    if (value == null) {
        throw new NoSuchElementException("No value present");
    }
    return value;
}
```

EXERCISES

Optional<T>

SINCE JAVA8

@SvenRuppert

Result<T>

SINCE JAVA8

@SvenRuppert

Optional<T>

@SvenRuppert

Not symmetric

Not async - blocking only

Not functional enough

Declared final - no inheritance

Having ValueTypes in mind

We need something

Result<T>

Must be easy to transform Optional <-> Result

Should work as an Drop-In-Replacement

Should connect to the Reactive-World

Should be symmetric

Must fit functional style / oo style

Result<T>

@SvenRuppert

Transform from and to an Optional

```
final Optional<String> s = helloResult.toOptional();
final Result<String> result = Result.fromOptional(s);
```

pos. and neg. available

```
    final Result<String> helloResult = Result.success("Hello");
    final Result<Integer> failed = Result.failure("ups..");
```

Result<T>

@SvenRuppert

Symmetric

```
helloResult.ifAbsent(() -> System.out.println("nothing here"));
helloResult.ifPresent((v) -> System.out.println(v));
```

Handle both ways

```
helloResult.ifPresentOrElse(
    success -> System.out.println("success = " + success) ,
    failure -> System.out.println("failure = " + failure)
);
```

Result<T>

@SvenRuppert

Handle both ways

```
helloResult.ifPresentOrElse(
    success -> System.out.println("success = " + success) ,
    failure -> System.out.println("failure = " + failure)
);
```

Async if needed

```
helloResult.ifPresentOrElseAsync(
    success -> System.out.println("success = " + success) ,
    failure -> System.out.println("failure = " + failure)
);
```

Result<T>

@SvenRuppert

Combine with Value and Transformation

```
final Result<String> next = helloResult
    .thenCombine( value: "next" ,
                  (s1 , s2) -> Result.success(s1 + s2));
```

```
final CompletableFuture<Result<String>> nextAsyncA
= helloResult.thenCombineAsync( value: "next" ,
                                (s1 , s2) -> Result.success(s1 + s2));
```

Async if needed

Result<T>

@SvenRuppert

Combine with Value / Function

```
final CompletableFuture<Result<String>> nextAsyncA
    = helloResult.thenCombineAsync( value: "next" ,
                                    (s1 , s2) -> Result.success(s1 + s2));
```

Async if needed

```
final CompletableFuture<Result<Integer>> cfFuncB = helloResult
    .thenCombineAsync((Function<String, Integer>) Integer::parseInt ,
                      (s , f) -> Result.success(f.apply(s)));
```

Combine with a function

EXERCISES

Result<T>

SINCE JAVA8

@SvenRuppert

Exceptions

SINCE JAVA8

@SvenRuppert

Exceptions

@SvenRuppert

```
// could crash ;-)
try {
    final int parseInt = Integer.parseInt( s: "uuppss");
} catch (NumberFormatException e) {
    e.printStackTrace();
}
```

```
final int upsA = Stream
        .of("1" , "2" , "3" , "ups" , "4")
        //will crash
        .mapToInt(Integer::parseInt)
        .sum();
```

Exceptions

@SvenRuppert

```
final int upsB = Stream
        .of("1" , "2" , "3" , "ups" , "4")
        //will crash
        .mapToInt(s -> {
            try {
                return Integer.parseInt(s);
            } catch (NumberFormatException e) {
                e.printStackTrace();
                return 0; // works only for sum()
            }
        })
        .sum();
```

Exceptions


```
final int upsC = Stream
        .of("1" , "2" , "3" , "ups" , "4")
        .map(s -> {
            try {
                return Optional.of(Integer.parseInt(s));
            } catch (NumberFormatException e) {
                e.printStackTrace();
                return Optional.<Integer>empty();
            }
        })
        .filter(Optional::isPresent)
        .mapToInt(Optional::get)
        .sum();
```

Exceptions

@SvenRuppert

```
final int upsD = Stream
    .of("1" , "2" , "3" , "ups" , "4")
    .map(s -> {
      try {
        return Optional.of(Integer.parseInt(s));
      } catch (NumberFormatException e) {
        e.printStackTrace();
        return Optional.<Integer>empty();
      }
    })
    .flatMap(Optional::stream)
    .mapToInt(i->i)
    .sum();
```

JDK9

Exceptions

@SvenRuppert

```
final int upsE = Stream
    .of("1" , "2" , "3" , "ups" , "4")
    .map(new CheckedFunction<String, Integer>() {
      @Override
      public Integer applyWithException(String s) throws Exception {
        return Integer.parseInt(s);
      }
    })
    .flatMap(Result::stream)
    .mapToInt(i->i)
    .sum();


final int upsG = Stream
    .of("1" , "2" , "3" , "ups" , "4")
    .map((CheckedFunction<String, Integer>) Integer::parseInt)
    .flatMap(Result::stream)
    .mapToInt(i->i)
    .sum();
```

Exceptions

@SvenRuppert

```
Function<String, Result<Integer>> checkedFuncA = new CheckedFunction<String, Integer>() {
  @Override
  public Integer applyWithException(String s) throws Exception {
    return Integer.parseInt(s);
  }
};
```

```
Function<String, Result<Integer>> checkedFuncB
    = (CheckedFunction<String, Integer>) Integer::parseInt;
```

Exceptions

@SvenRuppert

```
Function<String, Result<Integer>> checkedFuncB
        = (CheckedFunction<String, Integer>) Integer::parseInt;
```

```
@FunctionalInterface
public interface CheckedFunction<T, R> extends Function<T, Result<R>> {
  @Override
  default Result<R> apply(T t) {
    try {
      return Result.success(applyWithException(t));
    } catch (Exception e) {
      return Result.failure(message().apply(e));
    }
  }

  R applyWithException(T t) throws Exception;

}
```

Exceptions


```
@FunctionalInterface
public interface CheckedConsumer<T> extends CheckedFunction<T, Void> {


}
```

Exceptions

@SvenRuppert

```
@FunctionalInterface
public interface CheckedExecutor extends Function<Void, Result<Void>> {

  default Result<Void> execute() { return apply( t: null); }


  @Override
  default Result<Void> apply(Void t) {
    try {
      applyWithException();
      return Result.success(null);
    } catch (Exception e) {
      return Result.failure(message().apply(e));
    }
  }

  void applyWithException() throws Exception;

}
```

Exceptions

@SvenRuppert

```
@FunctionalInterface
public interface CheckedSupplier<T> extends Supplier<Result<T>> {

  @Override
  default Result<T> get() {
    try {
      return Result.success(getWithException());
    } catch (Exception e) {
      return Result.failure(message().apply(e));
    }
  }

  T getWithException() throws Exception;

  default T getOrElse(Supplier<T> supplier) { return get().getOrElse(supplier); }
}
```

Exceptions

different exceptions

EXERCISES

Exceptions

SINCE JAVA8

@SvenRuppert

Functions - Basics

HOW TO USE FUNCTIONS…

@SvenRuppert

Functions - Java8

@SvenRuppert

```
Function<Integer, Integer> funcAdd = (x) -> x + 2;
```

a function is a transformation

input -> output

(input - type, input value) → (output - type, output value)

Functions - Java8


```
Function<Integer, Integer> funcA = (x) -> x + 2;
Function<Integer, Integer> funcB = (x) -> x + 10;
Function<Integer, Integer> funcC = (x) -> x + 5;

Integer a = funcA.apply( t: 2);
Integer b = funcB.apply(a);
Integer c = funcB.apply(b);

funcA.apply(funcB.apply(funcC.apply( t: 2)));
funcC.apply(funcB.apply(funcA.apply( t: 2)));
```

Functions - Java8

@SvenRuppert

```
default <V> Function<V, R> compose(Function<? super V, ? extends T> before) {
    Objects.requireNonNull(before);
    return (V v) -> apply(before.apply(v));
}
```

```
Function<Integer, Integer> func = (x) -> x + 2;
int result = func
    .compose((Function<Integer, Integer>) x -> x + 10)
    .compose((Function<Integer, Integer>) x -> x + 5)
      .compose(x -> x + 10)
      .compose(x -> x + 5)
    .apply( t: 2);
```

Functions - Java8

@SvenRuppert

```
Function<Integer, Integer> func = (x) -> x + 2;
int result = func
    .compose((Function<Integer, Integer>) x -> x + 10)
    .compose((Function<Integer, Integer>) x -> x + 5)
      .compose(x -> x + 10)
      .compose(x -> x + 5)
    .apply( t: 2);
```

```
public static <T, U, V> Function<Function<U, V>, Function<Function<T, U>, Function<T, V>>> higherCompose() {
  return (Function<U, V> f) -> (Function<T, U> g) -> (T x) -> f.apply(g.apply(x));
}
```

```
final Function<
    Function<Integer, Integer>,
    Function<
        Function<Integer, Integer>,
        Function<Integer, Integer>>> higherCompose = higherCompose();
```

Functions - Java8

@SvenRuppert

```
final Function<
        Function<Integer, Integer>,
        Function<
                Function<Integer, Integer>,
                Function<Integer, Integer>>> higherCompose = higherCompose();
```

```
final Integer apply = <Integer,Integer,Integer>higherCompose() // won´t compile
```

```
final Integer apply = Main.<Integer, Integer, Integer>higherCompose()
        .apply(x -> 2 * x)  // second
        .apply(y -> y + 1)  // first
        .apply( t: 2);
```

Functions - Java8


```
Function<Integer, Integer> func = (x) -> x + 2;
int result = func
    .compose((Function<Integer, Integer>) x -> x + 10)
    .compose((Function<Integer, Integer>) x -> x + 5)
      .compose(x -> x + 10)
      .compose(x -> x + 5)
    .apply( t: 2);
```

```
  final Integer apply = <Integer,Integer,Integer>higherCompose() // won´t compile
final Integer apply = Main.<Integer, Integer, Integer>higherCompose()
    .apply(x -> 2 * x)  // second
    .apply(y -> y + 1)  // first
    .apply( t: 2);
```

Compose is not so nice.. but

Functions - Java8

@SvenRuppert

```
Function<Integer, Integer> func = (x) -> x + 2;
int result = func
    .compose((Function<Integer, Integer>) x -> x + 10)
    .compose((Function<Integer, Integer>) x -> x + 5)
      .compose(x -> x + 10)
      .compose(x -> x + 5)
    .apply( t: 2);
```

```
default <V> Function<T, V> andThen(Function<? super R, ? extends V> after) {
    Objects.requireNonNull(after);
    return (T t) -> after.apply(apply(t));
}
```

```
Function<Integer, Integer> func = x -> x + 2;
func
    .andThen(x -> x + 10)
    .andThen(x -> x + 5)
    .apply( t: 2);
```

Attention : make a semantic equal function !!

Functions - Java8


```
Function<Integer, Integer> funcA = (x) -> x + 2;
Function<Integer, Integer> funcB = (x) -> x + 10;
Function<Integer, Integer> funcC = (x) -> x + 5;
```

```
funcA
    .compose(funcB)
    .compose(funcC)
    .apply( t: 2);
```

```
funcA
    .andThen(funcB)
    .andThen(funcC)
    .apply( t: 2);
```

How to get an instance of a function?

Functions - Java8

@SvenRuppert

```
public static Function<Integer, Integer> funcA(){
    return (x) -> x + 2;
}

public static Function<Integer, Integer> funcB(){
    return (x) -> x + 10;
}

public static Function<Integer, Integer> funcC(){
    return (x) -> x + 5;
}
```

```
funcA()
    .compose(funcB())
    .compose(funcC())
    .apply( t: 2);
```

```
funcA()
    .andThen(funcB())
    .andThen(funcC())
    .apply( t: 2);
```

Functions - Java8


```
Function<Integer, Function<Integer, Integer>> adder
    = (x) -> {return (y) -> {return x + y;};};

Function<Integer, Integer> adder10 = adder.apply( t: 10);
Function<Integer, Integer> adder05 = adder.apply( t: 5);
Function<Integer, Integer> adder02 = adder.apply( t: 5);

adder02
    .andThen(adder05)
    .andThen(adder10)
    .apply( t: 2);
```

```
adder02
    .compose(adder05)
    .compose(adder10)
    .apply( t: 2);
```

```
adder
    .apply( t: 2)
    .andThen(adder.apply( t: 10))
    .andThen(adder.apply( t: 5))
    .apply( t: 2);
```

Functions - Java8


```
public static Function<Integer, Function<Integer, Integer>> adder(){
  return (x) -> (y) -> x + y ;
}

public static void main(String[] args) {
  adder()
      .apply( t: 2)
      .andThen(adder().apply( t: 10))
      .andThen(adder().apply( t: 5))
      .apply( t: 2);
}
```

Functions - Java8 - extract the logic !


```
final List<String> names = Arrays.asList(
        "Hugo",
        "Willy",
        "Simon",
        "Erwin",
        "Sigfried"
);
```

```
names
        .stream()
        .filter(s -> s.contains("i"))
        .forEach(out::println);
```

```
names
        .stream()
        .filter(s -> s.contains("g"))
        .forEach(out::println);
```

Functions - Java8


```
final List<String> names = Arrays.asList(
    "Hugo",
    "Willy",
    "Simon",
    "Erwin",
    "Sigfried"
);
```

```
final Consumer<String> println = out::println;
```

```
final String v = "i";
```

```
final Predicate<String> predicateI = s -> s.contains(v);
```

```
final Predicate<String> predicateG = s -> s.contains("g");
```

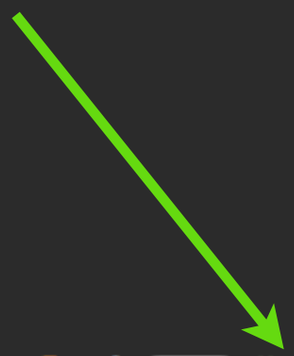
```
names
    .stream()
    .filter(predicateI)
    .forEach(println);
```

```
names
    .stream()
    .filter(predicateG)
    .forEach(println);
```

Functions - Java8


```
final Function<String, Predicate<String>> funPredicate
    = str -> (String s) -> s.contains(str);
```

```
names
    .stream()
    .filter(funPredicate.apply( t: "i"))
    .forEach(println);
```



```
names
    .stream()
    .filter(funPredicate.apply( t: "g"))
    .forEach(println);
```

EXERCISES

Functions - Basics

HOW TO USE FUNCTIONS…

@SvenRuppert

Currying

GENERIC BACK AND FORWARD

@SvenRuppert

Currying

@SvenRuppert

```
BiFunction<A, B, R>
```

```
                                Function<A, Function<B, R>>
```

```
Function<BiFunction<A, B, R>, Function<A, Function<B, R>>>
```

```
Function<
    BiFunction<A, B, R>,
    Function<A, Function<B, R>>> transform
  = new Function<BiFunction<A, B, R>, Function<A, Function<B, R>>>() {
@Override
public Function<A, Function<B, R>> apply(BiFunction<A, B, R> fIn) {
  return new Function<A, Function<B, R>>() {
    @Override
    public Function<B, R> apply(A a) {
      return (b) -> fIn.apply(a, b);
    }
  };
}
};
```

Currying

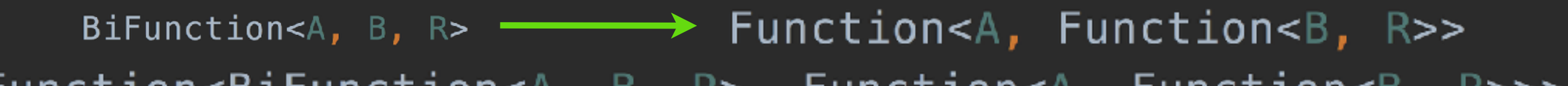
@SvenRuppert

```
BiFunction<A, B, R>  ⟶  Function<A, Function<B, R>>

Function<BiFunction<A, B, R>, Function<A, Function<B, R>>>
```

```
Function<
        BiFunction<A, B, R>,
        Function<A, Function<B, R>>> transform = {{
        fIn -> a -> (b) -> fIn.apply(a, b);
    }
};
return transform;
        }
    };
  }
};
```

Currying

@SvenRuppert




```
static <A, B, R> Function<BiFunction<A, B, R>, Function<A, Function<B, R>>> curryBiFunction() {
  return (func) -> a -> b -> func.apply(a, b);
}
```

```
static <A, B, R> Function<Function<A, Function<B, R>>, BiFunction<A, B, R>> unCurryBifunction() {
  return (func) -> (a, b) -> func.apply(a).apply(b);
}
```

Currying

@SvenRuppert

```
@FunctionalInterface
public interface TriFunction<T1, T2, T3, R> {
  R apply(T1 t1, T2 t2, T3 t3);

  default <V> TriFunction<T1, T2, T3, V> andThen(Function<? super R, ? extends V> after) {
    Objects.requireNonNull(after);
    return (T1 t1, T2 t2, T3 t3) -> after.apply(apply(t1, t2, t3));
  }
}
```

```
  static <A, B, C, R> Function<
      TriFunction<A, B, C, R>,
      Function<A, Function<B, Function<C, R>>>> curryTriFunction() {
    return (func) -> a -> b -> c -> func.apply(a, b, c);
  }

  static <A, B, C, R> Function<
      Function<A, Function<B, Function<C, R>>>,
      TriFunction<A, B, C, R>> unCurryTrifunction() {
    return (func) -> (a, b, c) -> func.apply(a).apply(b).apply(c);
  }
```

EXERCISES

Currying

GENERIC BACK AND FORWARD

@SvenRuppert

Memoizing

PURE FUNCTIONAL BEHAVIOR

@SvenRuppert

Memoizing

@SvenRuppert

```
public class Memoizer<T, U> {
  private final Map<T, U> memoizationCache = new ConcurrentHashMap<>();

  private Function<T, U> doMemoize(final Function<T, U> function) {
    return input -> memoizationCache.computeIfAbsent(input, function);
  }

  public static <T, U> Function<T, U> memoize(final Function<T, U> function) {
    return new Memoizer<T, U>().doMemoize(function);
  }
```

Memoizing

@SvenRuppert

```
private static final BiFunction<Integer,Integer,Integer> biFunction = (x, y) -> x * y;
private static final Function<Integer, Function<Integer, Integer>> function = x -> y -> x * y;

public static final Function<Integer, Function<Integer,Integer>> memoizationFunction
    = Memoizer.memoize(x -> Memoizer.memoize(y -> x * y));
```

```
public static void main(String[] args) {
  System.out.println("memoizationFunction = " + memoizationFunction.apply( t: 2).apply( t: 3));
  System.out.println("memoizationFunction = " + memoizationFunction.apply( t: 2).apply( t: 3));
}
```

Memoizing

@SvenRuppert

```
public static BiFunction<Integer, Integer, Integer> mul = (x, y) -> x * y;

public static final Function<Integer, Function<Integer, Integer>> memoizationFunction
    = memoize(x -> memoize(y -> mul.apply(x, y)));
```

```
public static void main(String[] args) {
  System.out.println("memoizationFunction = " + memoizationFunction.apply( t: 2).apply( t: 3));
  System.out.println("memoizationFunction = " + memoizationFunction.apply( t: 2).apply( t: 3));
}
```

Memoizing

@SvenRuppert

```
public static <T> Function<T, Function<T, T>> create(BiFunction<T, T, T> biFunction) {
  return memoize(x -> memoize(y -> biFunction.apply(x, y)));
}

public static void main(String[] args) {
  final BiFunction<Integer, Integer, Integer> biFunction = (x, y) -> x * y;
  final Function<Integer, Function<Integer, Integer>> function = create(biFunction);

  System.out.println("memoizationFunction = " + function.apply( t: 2).apply( t: 3));
  System.out.println("memoizationFunction = " + function.apply( t: 2).apply( t: 3));
}
```

Memoizing

@SvenRuppert

```
private static <T1, T2, R> Function<T1, Function<T2, R>> create(BiFunction<T1, T2, R> biFunction) {
  return memoize(x -> memoize(y -> biFunction.apply(x, y)));
}

public static void main(String[] args) {
  final Function<Integer, Function<Integer, Integer>> function = create((x, y) -> x * y);
  System.out.println("memoizationFunction = " + function.apply( t: 2).apply( t: 3));
  System.out.println("memoizationFunction = " + function.apply( t: 2).apply( t: 3));
}
```

Memoizing

@SvenRuppert

```
public static <T1, T2, R> BiFunction<T1, T2, R> backToBiFunction(final Function<T1, Function<T2, R>> function) {
  return (x, y) -> function.apply(x).apply(y);
}

private static <T1, T2, R> Function<T1, Function<T2, R>> create(BiFunction<T1, T2, R> biFunction) {
  return memoize(x -> memoize(y -> biFunction.apply(x, y)));
}

public static void main(String[] args) {
  final Function<Integer, Function<Integer, Integer>> function = create((x, y) -> x * y);

  System.out.println("memoizationFunction = " + backToBiFunction(function).apply( t: 2, u: 3));
  System.out.println("memoizationFunction = " + backToBiFunction(function).apply( t: 2, u: 3));
}
```

Memoizing

@SvenRuppert

```
public static <T1, T2, R> BiFunction<T1, T2, R> memoize(final BiFunction<T1, T2, R> biFunc) {
  return backToBiFunction(create(biFunc));
}

public static <T1, T2, R> BiFunction<T1, T2, R> backToBiFunction(final Function<T1, Function<T2, R>> function) {
  return (x, y) -> function.apply(x).apply(y);
}

private static <T1, T2, R> Function<T1, Function<T2, R>> create(BiFunction<T1, T2, R> biFunction) {
  return Memoizer.memoize(x -> Memoizer.memoize(y -> biFunction.apply(x, y)));
}

public static void main(String[] args) {
  final BiFunction<Integer, Integer, Integer> function = memoize((x, y) -> x * y);

  System.out.println("memoizationFunction = " + function.apply( t: 2, u: 3));
  System.out.println("memoizationFunction = " + function.apply( t: 2, u: 3));
}
```

Memoizing

@SvenRuppert

```
public static <T1, T2, R> BiFunction<T1, T2, R> memoize(final BiFunction<T1, T2, R> biFunc) {
  final Function<T1, Function<T2, R>> transformed
      = Memoizer.memoize(
          x -> Memoizer.memoize(
              y -> biFunc.apply(x, y)));
  return (x, y) -> transformed.apply(x).apply(y);
}
```

```
public static void main(String[] args) {
  final BiFunction<Integer, Integer, Integer> function = memoize((x, y) -> {
    System.out.println("execute x/y = " + x + " / " + y);
    return x * y;
  });

  System.out.println("memoizationFunction = " + function.apply( t: 2, u: 3));
  System.out.println("memoizationFunction = " + function.apply( t: 2, u: 3));
}
```

Memoizing

@SvenRuppert

```
@FunctionalInterface
public interface TriFunction<T1, T2,T3, R> {
  R apply(T1 t1, T2 t2, T3 t3);

  default <V> TriFunction<T1, T2,T3, V> andThen(Function<? super R, ? extends V> after) {
    Objects.requireNonNull(after);
    return (T1 t1, T2 t2, T3 t3) -> after.apply(apply(t1, t2, t3));
  }
}

public static <T1, T2,T3, R> TriFunction<T1, T2,T3, R> memoize(final TriFunction<T1, T2,T3, R> threeFunc) {
  final Function<T1, Function<T2, Function<T3, R>>> transformed
      = Memoizer.memoize(
          x -> Memoizer.memoize(
              y -> Memoizer.memoize(
                  z -> threeFunc.apply(x, y,z))));
  return (x, y, z) -> transformed.apply(x).apply(y).apply(z);
}

public static void main(String[] args) {
  final TriFunction<Integer, Integer, Integer, Integer> function = memoize((x, y, z) -> x * y * z);

  System.out.println("memoizationFunction = " + function.apply( t1: 2, t2: 3, t3: -1));
  System.out.println("memoizationFunction = " + function.apply( t1: 2, t2: 3, t3: -1));
}
```

Memoizing

@SvenRuppert

```
// legacy code - changes are not allowed
public static class MyLegacyClass {
  public String doWorkA(int a, int b) { return "hello " + a + b; }
}

public static <T1,T2,R> void doNewStuff(BiFunction<T1,T2,R> func, T1 valueA, T2 valueB) {
  //some usefull generic thing here
  final R result  = func.apply(valueA, valueB);
  System.out.println("result = " + result);
}
```

```
public static void main(String[] args) {
  final MyLegacyClass myLegacyClass = new MyLegacyClass();

  //String doWorkA(int a, int b) -> BiFunction ;-)
  final BiFunction<Integer, Integer, String> doWork = myLegacyClass::doWorkA;

  doNewStuff(doWork, valueA: 1, valueB: 2);
}
```

Memoizing

@SvenRuppert

```
final MyLegacyClass myLegacyClass = new MyLegacyClass( );
final BiFunction<Integer, Integer, Value> doWork = myLegacyClass::doWorkA;
final BiFunction<Integer, Integer, Value> memoize = Memoizer.memoize(doWork);
```

EXERCISES

Memoizing

PURE FUNCTIONAL BEHAVIOR

@SvenRuppert

Case

HOW TO DECIDE?

@SvenRuppert

Case


```
switch (value){
    case "A": break;
    case "B": break;
    case "C": break;
    default: break;
}
```

```
if (value.equals("A")){

} else if(value.equals("B")){

} else if(value.equals("C")) {

} else {
    //default
}
```

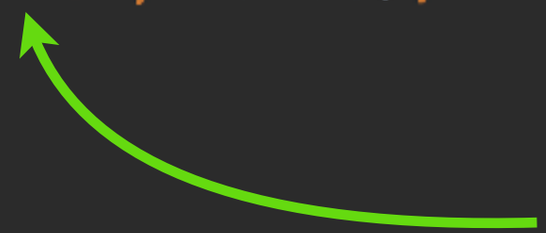
```
String r =  value.equals("A")
        ? "x" : value.equals("B")
                ? "x" : value.equals("C")
                        ? "x" : "default";
```

Case

@SvenRuppert

```
public class Case<T> extends Pair<Supplier<Boolean>, Supplier<Result<T>>>

                    private boolean isMatching() { return getT1().get(); }
                    public Result<T> result() { return getT2().get(); }
```

```
public static <T> Case<T> matchCase(Supplier<Boolean> condition ,
                                    Supplier<Result<T>> value) {
  return new Case<>(condition , value);
}

public static <T> DefaultCase<T> matchCase(Supplier<Result<T>> value) {
  return new DefaultCase<>(() -> true , value);
}
```



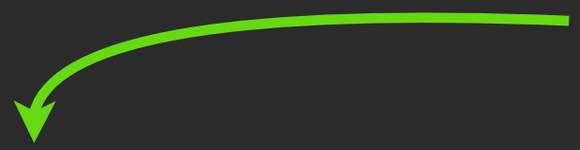
Case

@SvenRuppert

```
public class Case<T> extends Pair<Supplier<Boolean>, Supplier<Result<T>>>

                    private boolean isMatching() { return getT1().get(); }
                    public Result<T> result() { return getT2().get(); }


  @SafeVarargs
  public static <T> Result<T> match(DefaultCase<T> defaultCase , Case<T>... matchers) {

    return Stream
        .of(matchers)
        .filter(Case::isMatching)
        .map(Case::result)
        .findFirst()
        .orElseGet(defaultCase::result);
  }
```

Case

@SvenRuppert



```
Result<String> result = Case
    .match(
        Case.matchCase(() -> Result.success("default value")) ,
        Case.matchCase(() -> value.equals("A") , () -> Result.success("x")) ,
        Case.matchCase(() -> value.equals("B") , () -> Result.success("x")) ,
        Case.matchCase(() -> value.equals("C") , () -> Result.success("x")) ,
        Case.matchCase(() -> value.equals("X") , () -> Result.failure("error message"))
    );

result.ifPresentOrElseAsync(
    success -> { /* something usefull */ } ,
    failed -> { /* something usefull */}
);
```

Case


```
match(
    matchCase(() -> success("default value")) ,
    matchCase(() -> value.equals("A") , () -> success("x")) ,
    matchCase(() -> value.equals("B") , () -> success("x")) ,
    matchCase(() -> value.equals("C") , () -> success("x")) ,
    matchCase(() -> value.equals("X") , () -> failure("error message")))
    .ifPresentOrElseAsync(
        success -> { /* something usefull */ } ,
        failed -> { /* something usefull */}
    );
```

EXERCISES

Case

HOW TO DECIDE?

@SvenRuppert

Streams

HOW TO USE, HOW TO BUILD

@SvenRuppert

Streams

@SvenRuppert

Streams

A Stream is active after creation

want to define a Stream without starting it

maybe creating it interactively

we need a function !

Function<T, Stream<R>>

EXERCISES

Streams

HOW TO USE, HOW TO BUILD

@SvenRuppert

Refactoring example

based on Java8

@SvenRuppert

Java8 - Functional Interfaces - Example

@SvenRuppert

typical legacy implementation

```
public void writeUser(final User user){
  final int update = update(user, connectionPool());
  // check result
}
```

```
private int update(User user, final JDBCConnectionPool connectionPool) {
  final DataSource dataSource = connectionPool.getDataSource();
  try {
    final Connection connection = dataSource.getConnection();
    final int count;
    try (final Statement statement = connection.createStatement()) {
      final String sql = createUpdteSQL(user);
      count = statement.executeUpdate(sql);
      statement.close();
    }
    dataSource.evictConnection(connection);
    return count;
  } catch (final SQLException e) {
    e.printStackTrace();
  }
  return -1;
}
```

Java8 - Functional Interfaces - Example

typical legacy implementation

```
private String createUpdteSQL(final User user) {
  return ""; // vendor SQL , MySQL/Oracle...
}


public static class User { }

public abstract JDBCConnectionPool connectionPool();

public interface JDBCConnectionPool {...}

public interface DataSource {...}
```

Java8 - Functional Interfaces - Example

@SvenRuppert

```java
@FunctionalInterface
public interface BasicOperation {
  @NotNull
  String createSQL();
}
```

```java
@FunctionalInterface
public interface Update extends BasicOperation {

  default int update(final JDBCConnectionPool connectionPool) {
    final HikariDataSource dataSource = connectionPool.getDataSource();
    try {
      final Connection connection = dataSource.getConnection();
      final int count;
      try (final Statement statement = connection.createStatement()) {
        final String sql = createSQL();
        count = statement.executeUpdate(sql);
        statement.close();
      }
      dataSource.evictConnection(connection);
      return count;
    } catch (final SQLException e) {
      e.printStackTrace();
    }
    return -1;
  }

}
```

Java8 - Functional Interfaces - Example


```java
@FunctionalInterface
public interface DAO {
  JDBCConnectionPool connectionPool();
}
```

```java
public abstract class GenericDAO implements DAO {

  private JDBCConnectionPool connectionPool;

  public DAO workOnPool(final JDBCConnectionPool connectionPool) {
    this.connectionPool = connectionPool;
    return this;
  }

  @Override
  public JDBCConnectionPool connectionPool() { return connectionPool; }
}
```

Java8 - Functional Interfaces - Example

@SvenRuppert

```
public interface UserDAO extends DAO {

  default void writeUser(final User user) {
    ((Update) () -> createInsert(user)).update(connectionPool());
  }
```

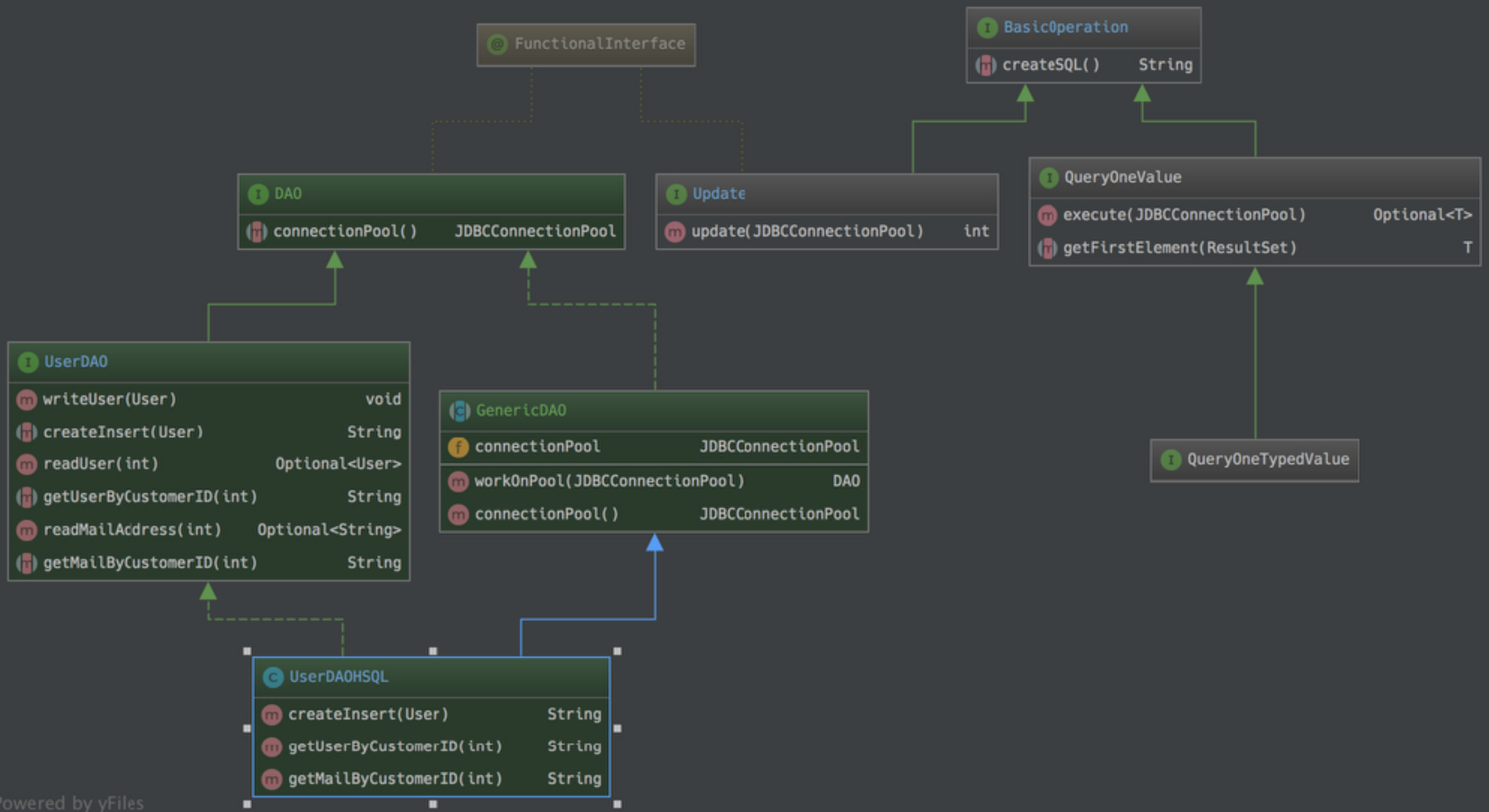
```
String createInsert(User user);
```

Java8 - Functional Interfaces - Example

```
public class UserDAOHSQL extends GenericDAO implements UserDAO {

  public String createInsert(final User user) {
    final String sql = "INSERT INTO CUSTOMER " +
        "( CUSTOMER_ID, FIRSTNAME, LASTNAME, EMAIL ) " +
        " VALUES " +
        "(  " +
        user.getCustomerID() + ", " +
        "'" + user.getFirstname() + "', " +
        "'" + user.getLastname() + "', " +
        "'" + user.getEmail() + "' " +
        ")";
    return sql;
  }
```

Java8 - Functional Interfaces - Example

@SvenRuppert



Java8 - Functional Interfaces - Example

Sourcecode is on github

svenruppert / JDBC-DAO

Code | Issues 1 | Pull requests 0 | Projects 1

minimalistic JDBC based DAO

connection-pool | dao | design-pattern | functional | java | java8

Pattern in Java 8/9

how pattern will change…

@SvenRuppert

Virtual Proxy

Virtual Proxy


```
public class ServiceProxy implements Service {
  private Service service = new ServiceImpl();
  public String work(String txt) {  return service.work(txt); }
}
```

What could we change now ?

Virtual Proxy:

create the Delegator later

```
public class VirtualService implements Service {
  private Service service = null;
  public String work(String txt) {
    if(service == null) { service = new ServiceImpl(); }
    return service.work(txt);
  }
}
```

```
public class VirtualService implements Service {
  private Service service = null;
  public String work(String txt) {
    if(service == null) { service = new ServiceImpl(); }
    return service.work(txt);
  }
}
```

This is NOT ThreadSafe

fixed decision for an implementation

how to combine it with a FactoryPattern ?

Virtual Proxy


```
if(service == null) { service = new ServiceImpl(); }
```

```
public interface ServiceFactory {
  Service createInstance();
}
```

```
public interface ServiceStrategyFactory {
   Service realSubject(ServiceFactory factory);
}
```

Virtual Proxy - Not - ThreadSafe


```
if(service == null) { service = new ServiceImpl(); }
```

```
private ServiceFactory serviceFactory = ServiceImpl::new;
```

```
public class ServiceStrategyFactoryImpl implements ServiceStrategyFactory {
    Service realSubject;
    public Service realSubject(final ServiceFactory factory) {
        if (realSubject == null) {
            realSubject = factory.createInstance();
        }
        return realSubject;
    }
}
```

Virtual Proxy - Not - ThreadSafe


```
if(service == null) { service = new ServiceImpl(); }
```

```
public class ServiceProxy implements Service {
  private ServiceFactory serviceFactory = ServiceImpl::new;
  private ServiceStrategyFactory strategyFactory = new ServiceStrategyFactoryImpl();

  public String work(String txt) {
    return strategyFactory.realSubject(serviceFactory).work(txt);
  }
}
```

Virtual Proxy - Show some code….


```java
public interface Service {
  String doWork(String value);
}
```

```java
public static class DefaultService implements Service {
  @Override
  public String doWork(String value) {
    return Optional.ofNullable(value)
        .orElse( other: "")
        .toUpperCase();
  }
}
```

```java
public static class VirtualNotThreadSafeProxyService implements Service {

  private Service delegator;

  @Override
  public String doWork(final String value) {
    if(delegator == null) delegator = new DefaultService();
    return delegator.doWork(value);
  }
}
```

Virtual Proxy - Show some code….


```
public interface Service {...}

public static class DefaultService implements Service {...}
```

```
@FunctionalInterface
public interface Factory<T> {
  T createInstance();
}
```

```
public static class ServiceFactory implements Factory<Service> {
  @Override
  public Service createInstance() { return new DefaultService(); }
}
```

Virtual Proxy - Show some code….


```
public interface Service {...}

public static class DefaultService implements Service {...}
```

```
@FunctionalInterface
public interface Factory<T> {
  T createInstance();
}
```

```
@FunctionalInterface
public interface Strategy<T> {
  T realSubject(Factory<T> factory);
}
```

```
public static class StrategyNotThreadSafe<T> implements Strategy<T>{
  private T delegator;
  @Override
  public T realSubject(final Factory<T> factory) {
    System.out.println("factory = " + factory.getClass().getSimpleName());
    if(delegator == null) {
      System.out.println("StrategyNotThreadSafe - create Delegator ");
      delegator = factory.createInstance();
    }
    return delegator;
  }
}
```

Virtual Proxy - Show some code….

@SvenRuppert

```java
@FunctionalInterface
public interface Factory<T> {
  T createInstance();
}
```

```java
@FunctionalInterface
public interface Strategy<T> {
  T realSubject(Factory<T> factory);
}
```

```java
public interface Service {...}

public static class DefaultService implements Service {...}
```

```java
public static class VirtualProxy implements Service {

  private Factory<Service> factory;
  private Strategy<Service> strategy;

  public VirtualProxy(final Factory<Service> factory,
                      final Strategy<Service> strategy) {
    this.factory = factory;
    this.strategy = strategy;
  }

  @Override
  public String doWork(final String value) {
    return strategy.realSubject(factory).doWork(value);
  }
}
```

Virtual Proxy - Show some code….

@SvenRuppert

```
public interface Service {...}

public static class DefaultService implements Service {...}
```

```
Supplier<T>
```

```
@FunctionalInterface
public interface Factory<T> {
  T createInstance();
}
```

```
@FunctionalInterface
public interface Strategy<T> {
  T realSubject(Factory<T> factory);
}
```

```
Function<Supplier<T>,T>
```

Virtual Proxy - Show some code….

@SvenRuppert

```
Supplier<T>
```

```
Function<Supplier<T>,T>
```

```
public interface Service {...}

public static class DefaultService implements Service {...}
```

```
public static class StrategyNotThreadSafe<T> implements Function<Supplier<T>,T> {
  private T delegator;
  @Override
  public T apply(final Supplier<T> supplier) {
    System.out.println("factory = " + supplier.getClass().getSimpleName());
    if(delegator == null) {
      System.out.println("StrategyNotThreadSafe - create Delegator ");
      delegator = supplier.get();
    }
    return delegator;
  }
}
```

Virtual Proxy - Show some code….

@SvenRuppert

```
public interface Service {...}

public static class DefaultService implements Service {...}
```

```
public static class VirtualProxy implements Service {

  private Supplier<Service> factory;
  private Function<Supplier<Service>,Service> strategy;

  public VirtualProxy(final Supplier<Service> supplier ,
                      final Function<Supplier<Service>,Service> function) {
    this.factory = supplier;
    this.strategy = function;
  }

  @Override
  public String doWork(final String value) {
    return strategy.apply(factory).doWork(value);
  }
}
```

Virtual Proxy - Show some code….


```
public interface Service {...}

public static class DefaultService implements Service {...}
```

```
public static abstract class ProxyBuilder<T> {

  protected Supplier<T> serviceSupplier;
  protected Function<Supplier<T>, T> serviceStrategyFunction;

  public ProxyBuilder<T> withSupplier(final Supplier<T> serviceSupplier) {
    this.serviceSupplier = serviceSupplier;
    return this;
  }

  public ProxyBuilder<T> withStrategyFunction(final Function<Supplier<T>,
    this.serviceStrategyFunction = serviceStrategyFunction;
    return this;
  }

  public abstract T build();
```

Virtual Proxy - Show some code….

@SvenRuppert

```
public interface Service {...}

public static class DefaultService implements Service {...}
```

```
public static class ServiceProxyBuilder extends ProxyBuilder<Service> {
  @Override
  public Service build() {
    return new Service() {
      @Override
      public String doWork(final String value) {
        return serviceStrategyFunction.apply(serviceSupplier).doWork(value);
      }
    };
  }
}
```

Virtual Proxy - Show some code….

@SvenRuppert

```
public static class ServiceProxyBuilder extends ProxyBuilder<Service> {
  @Override
  public Service build() {
    return new Service() {
      @Override
      public String doWork(final String value) {
        return serviceStrategyFunction.apply(serviceSupplier).doWork(value);
      }
    };
  }
}
```

```
public interface Service {...}

public static class DefaultService implements Service {...}
```

```
public static class ServiceProxyBuilder extends ProxyBuilder<Service> {
  @Override
  public Service build() {
    return value -> serviceStrategyFunction.apply(serviceSupplier).doWork(value);
  }
}
```

Decorator - pre Java8

Decorator - pre Java8


```java
public interface CalculatorService {
  double calculate(double value);
}

// Default we want to Decorate
public static class CalculatorDefaultService implements CalculatorService {
  @Override
  public double calculate(final double value) {
    final double result = value + 100;
    System.out.println(this.getClass().getSimpleName() + " - in / out = " + value + " - " + result);
    return result;
  }
}
```

```java
//not nice...
public static abstract class AbstractCalculatorService implements CalculatorService {
  private final CalculatorService delegator;

  public AbstractCalculatorService(final CalculatorService delegator) { this.delegator = delegator; }

  protected abstract double calculateImpl(final double value);

  public final double calculate(final double value) { return calculateImpl(delegator.calculate(value)); }
}
```

Decorator - pre Java8


```java
public interface CalculatorService {
  double calculate(double value);
}

// Default we want to Decorate
public static class CalculatorDefaultService implements CalculatorService {
  @Override
  public double calculate(final double value) {
    final double result = value + 100;
    System.out.println(this.getClass().getSimpleName() + " - in / out = " + value + " - " + result);
    return result;
  }
}
```

```java
//not nice...
public static abstract class AbstractCalculatorService implements CalculatorService {
  private final CalculatorService delegator;

  public AbstractCalculatorService(final CalculatorService delegator) { this.delegator = delegator; }

  protected abstract double calculateImpl(final double value);

  public final double calculate(final double value) { return calculateImpl(delegator.calculate(value)); }
}
```

```java
public static class RoundUpService extends AbstractCalculatorService {
  public RoundUpService(final CalculatorService delegator) { super(delegator); }

  protected double calculateImpl(double value) {
    final double result = Math.ceil(value);
    System.out.println(this.getClass().getSimpleName() + " - in / out = " + value + " - " + result);
    return result;
  }
}
```

Decorator - pre Java8

@SvenRuppert

```java
public interface CalculatorService {
  double calculate(double value);
}

// Default we want to Decorate
public static class CalculatorDefaultService implements CalculatorService {
  @Override
  public double calculate(final double value) {
    final double result = value + 100;
    System.out.println(this.getClass().getSimpleName() + " - in / out = " + value + " - " + result);
    return result;
  }
}
```

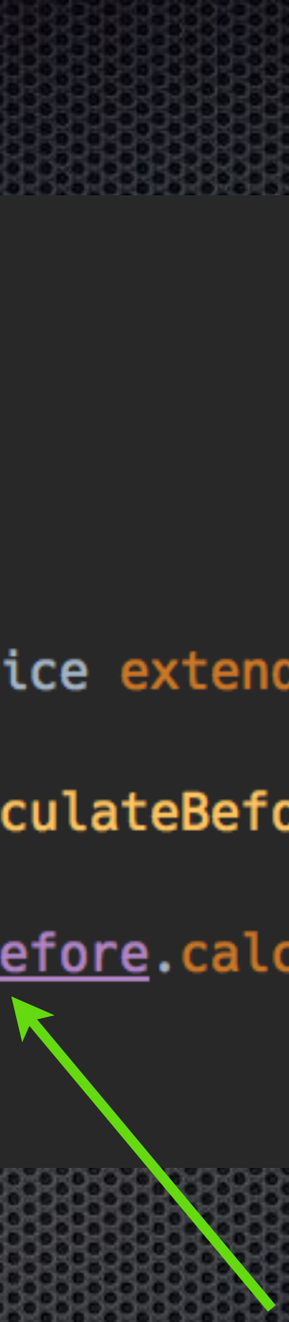
```java
//not nice...
public static abstract class AbstractCalculatorService implements CalculatorService {
  private final CalculatorService delegator;

  public AbstractCalculatorService(final CalculatorService delegator) { this.delegator = delegator; }

  protected abstract double calculateImpl(final double value);

  public final double calculate(final double value) { return calculateImpl(delegator.calculate(value)); }
}
```

```java
public static class RoundUpService extends AbstractCalculatorService {
  public RoundUpService(final CalculatorService delegator) { super(delegator); }

  protected double calculateImpl(double value) {
    final double result = Math.ceil(value);
    System.out.println(this.getClass().getSimpleName() + " - in / out = " + value + " - " + result);
    return result;
  }
}
```

```java
public static void main(String[] args) {
  final CalculatorService calculatorService =
      new RoundUpService(
          new AddHalfService(
              new SubOneService(
                  new CalculatorDefaultService())));

  final double calculate = calculatorService.calculate( value: 10.3d);
  System.out.println("calculate = " + calculate);
```

Decorator - Java8


```java
@FunctionalInterface
public interface CalculatorService {
  double calculate(double value);
}

@FunctionalInterface
public interface CalculatorDecoratorService extends CalculatorService {

  default CalculatorDecoratorService calculateBefore(CalculatorService before) {
    Objects.requireNonNull(before);
    return (double value) -> calculate(before.calculate(value));
  }
}
```



```java
public static final CalculatorService CalculatorDefaultService = (value) -> {
  final double result = value + 100;
  System.out.println("CalculatorDefaultService - in / out = " + value + " - " + result);
  return result;
};
```

Decorator - Java8

@SvenRuppert

```java
@FunctionalInterface
public interface CalculatorService {
  double calculate(double value);
}

@FunctionalInterface
public interface CalculatorDecoratorService extends CalculatorService {

  default CalculatorDecoratorService calculateBefore(CalculatorService before) {
    Objects.requireNonNull(before);
    return (double value) -> calculate(before.calculate(value));
  }
}
```

```java
public static final CalculatorService CalculatorDefaultService = (value) -> {
  final double result = value + 100;
  System.out.println("CalculatorDefaultService - in / out = " + value + " - " + result);
  return result;
};
```

```java
public static final CalculatorDecoratorService RoundUpService = (value) -> {
  final double result = Math.ceil(value);
  System.out.println("RoundUpService - in / out = " + value + " - " + result);
  return result;
};
```

Decorator - Java8


```java
@FunctionalInterface
public interface CalculatorService {
  double calculate(double value);
}

@FunctionalInterface
public interface CalculatorDecoratorService extends CalculatorService {

  default CalculatorDecoratorService calculateBefore(CalculatorService before) {
    Objects.requireNonNull(before);
    return (double value) -> calculate(before.calculate(value));
  }
}
```

```java
final double calculate = RoundUpService
    .calculateBefore(AddHalfService)
    .calculateBefore(SubOneService)
    .calculateBefore(CalculatorDefaultService)
    .calculate( value: 10.3d);
```

```java
final double calculateShort = RoundUpService
    .calculateBefore(value -> value + 0.5d)
    .calculateBefore(value -> value - 1)
    .calculateBefore(CalculatorDefaultService)
    .calculate( value: 10.3d);
```

Decorator - Java8 - modify orig Interface

@SvenRuppert

```java
@FunctionalInterface
public interface CalculatorService {
  double calculate(double value);
}

@FunctionalInterface
public interface CalculatorDecoratorService extends CalculatorService {

  default CalculatorDecoratorService calculateBefore(CalculatorService before) {
    Objects.requireNonNull(before);
    return (double value) -> calculate(before.calculate(value));
  }
}
```

```java
@FunctionalInterface
public interface CalculatorService {

  double calculate(double value);

  default CalculatorService andThen(CalculatorService after) {
    Objects.requireNonNull(after);
    return (double value) -> after.calculate(calculate(value));
  }
}
```

Decorator - Java8 - DoubleUnaryOperator

@SvenRuppert

```
//  public interface CalculatorService extends DoubleUnaryOperator {
//    double applyAsDouble(double value);
//  }
```

```
public static final DoubleUnaryOperator CalculatorDefaultService = (value) -> {
  final double result = value + 100;
  System.out.println("CalculatorDefaultService - in / out = " + value + " - " + result);
  return result;
};
```

```
final double calculate = CalculatorDefaultService
    .andThen(SubOneService)
    .andThen(AddHalfService)
    .andThen(RoundUpService)
    .applyAsDouble( operand: 10.3d);
```

```
final double calculateShort = CalculatorDefaultService
    .andThen(value -> value - 1)
    .andThen(value -> value + 0.5d)
    .andThen(Math::ceil)
    .applyAsDouble( operand: 10.3d);
```

Interpreter - pre Java8

Interpreter - pre Java8


```
interface Expression {
  int interpret();
}

public static abstract class Operator implements Expression {
  protected final Expression leftExpression;
  protected final Expression rightExpression;

  public Operator(final Expression leftExpression, final Expression rightExpression) {
    this.leftExpression = leftExpression;
    this.rightExpression = rightExpression;
  }
}

public static class Add extends Operator {
  public Add(final Expression leftExpression, final Expression rightExpression) {
    super(leftExpression, rightExpression);
  }

  @Override
  public int interpret() {
    return leftExpression.interpret() + rightExpression.interpret();
  }
}
```

Interpreter - pre Java8


```
public static class Number implements Expression {
  private final int n;

  public Number(int n) { this.n = n; }

  @Override
  public int interpret() { return n; }
}
```

```
public static Expression getOperator(String s, Expression left, Expression right) {
  return
      (s.equals("+")) ? new Add(left, right) :
      (s.equals("-")) ? new Subtract(left, right) :
      (s.equals("*")) ? new Product(left, right) :
                        null; // not nice ;-)
}
```

Interpreter - pre Java8


```
public static boolean isOperator(String s) {
  return s.equals("+") || s.equals("-") || s.equals("*");
}
```

```
public static int evaluate(String expression) {
  final Stack<Expression> stack = new Stack<>();
  for (final String s : expression.split( regex: " ")) {
    if (isOperator(s)) {
      Expression right = stack.pop();
      Expression left = stack.pop();
      stack.push(getOperator(s, left, right));
    } else {
      Expression i = new Number(Integer.parseInt(s));
      stack.push(i);
    }
  }
  return stack.pop().interpret();
}
```

```
public static void main(String[] args) {
  String expression = "7 3 - 2 1 + *";
  System.out.println(evaluate(expression));
}
```

Interpreter - Java8 - Java9


```
private static final Map<String, IntBinaryOperator> OPERATOR_MAP = new HashMap<>();

static {
  OPERATOR_MAP.put("+", (a, b) -> a + b);
  OPERATOR_MAP.put("*", (a, b) -> a * b);
  OPERATOR_MAP.put("-", (a, b) -> a - b);
}
```

```
public static int evaluate(String expression) {
  Stack<Integer> stack = new Stack<>();
  for (final String s : expression.split( regex: " ")) {
    IntBinaryOperator op = OPERATOR_MAP.get( s );
    if (op != null) {
      int right = stack.pop();
      int left = stack.pop();
      stack.push(op.applyAsInt( left, right ));
    } else {
      stack.push(Integer.parseInt(s));
    }
  }
  return stack.pop();
}
```

Interpreter - Java8 - Java9

Main Idea - DataStructure to Function

```
private static final Map<String, IntBinaryOperator> OPERATOR_MAP = new HashMap<>();

static {
  OPERATOR_MAP.put("+", (a, b) -> a + b);
  OPERATOR_MAP.put("*", (a, b) -> a * b);
  OPERATOR_MAP.put("-", (a, b) -> a - b);
}
```

```
private static final Function<String, Optional<IntBinaryOperator>> str2OpFNC = (str) ->
    (isNull(str))      ? empty() :
    (str.equals("+")) ? of((a, b) -> a + b) :
    (str.equals("*")) ? of((a, b) -> a * b) :
    (str.equals("-")) ? of((a, b) -> a - b) :
                        empty();
```

Interpreter - Java8 - Java9


```
private static final Function<String, Optional<IntBinaryOperator>> str2OpFNC = (str) ->
    (isNull(str))      ? empty() :
    (str.equals("+")) ? of((a, b) -> a + b) :
    (str.equals("*")) ? of((a, b) -> a * b) :
    (str.equals("-")) ? of((a, b) -> a - b) :
                        empty();
```

```
private static final Function<String, Integer> evaluate = (expression) -> {
  final Stack<Integer> stack = new Stack<>();
  for (final String s : expression.split( regex: " ")) {
    str2OpFNC.apply(s)
        .ifPresentOrElse(op -> {
              int right = stack.pop(); // not nice - state outside !?!
              int left = stack.pop();
              stack.push(op.applyAsInt(left, right));
            },
            () -> stack.push(Integer.parseInt(s))
        );
  }
  return stack.pop();
};
```

Observer

HOW REACTIVE WILL START

@SvenRuppert

Observer

@SvenRuppert

```
public static class Observable<KEY, VALUE> {

  private final Map<KEY, Consumer<VALUE>> listeners = new ConcurrentHashMap<>();

  public void register(KEY key, Consumer<VALUE> listener) { listeners.put(key, listener); }

  public void unregister(KEY key) { listeners.remove(key); }

  public void sendEvent(VALUE event) { listeners.values().forEach(listener -> listener.accept(event)); }
}
```

```
public static void main(String[] args) {
  final Observable<String, String> observable = new Observable<>();
  observable.register( key: "key1", System.out::println);
  observable.register( key: "key2", System.out::println);

  observable.sendEvent( event: "Hello World!");

  observable.unregister( key: "key1");
  observable.sendEvent( event: "Hello World!");
}
```

Observer


```
public static class Registry {
  private static final Observable<String, String> observable = new Observable<>();

  public static void register(String key, Consumer<String> consumer){
    observable.register(key, consumer);
  }

  public static void unregister(String key){
    observable.unregister(key);
  }

  public static void sendEvent(String input){
    observable.sendEvent(input);
  }
}
```

```
//Usage of the Registry
Registry.register( key: "key1" , System.out::println);
Registry.register( key: "key2" , System.out::println);

Registry.sendEvent( input: "Hello World");

Registry.unregister( key: "key1");
Registry.sendEvent( input: "Hello World again");
```

Observer


```
public interface Registration {
  public void remove();
}
```

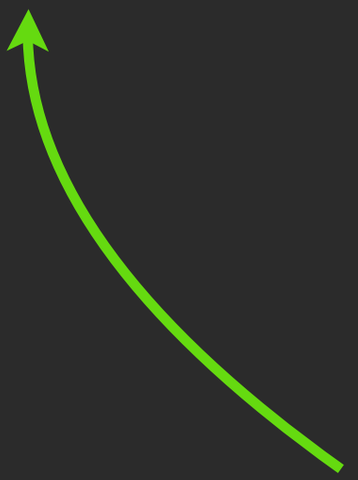
```
public class Observable<KEY, VALUE> {

  private final Map<KEY, Consumer<VALUE>> listeners = new ConcurrentHashMap<>();

  public Registration register(KEY key, Consumer<VALUE> listener) {
    listeners.put(key, listener);

    return () -> listeners.remove(key);
  }

  public void sendEvent(VALUE event) { listeners.values().forEach(c -> c.accept(event)); }
}
```

```
public class Registry {
  private static final Observable<String, String> observable = new Observable<>();

  public static Registration register(String key, Consumer<String> consumer) {
    return observable.register(key, consumer);
  }

  public static void sendEvent(String input) { observable.sendEvent(input); }
}
```

Observer

@SvenRuppert

```
public static void main(String[] args) {
  final Registration registerA = Registry.register( key: "key1" , System.out::println);
  final Registration registerB = Registry.register( key: "key2" , System.out::println);
  Registry.sendEvent( input: "Hello World");

  //done by life cycle
  registerA.remove();

  Registry.sendEvent( input: "Hello World again");
}
```



key not needed anymore

Observer


```
public class Observable<VALUE> {

  private final Set<Consumer<VALUE>> listeners = newKeySet();

  public Registration register(Consumer<VALUE> listener) {
    listeners.add(listener);
    return () -> listeners.remove(listener);
  }

  public void sendEvent(VALUE event) {
    listeners.forEach(c -> c.accept(event));
  }
}
```

HOW TO CHAIN OBSERVER?

EXERCISES

Observer

HOW REACTIVE WILL START

@SvenRuppert

CompletableFuture

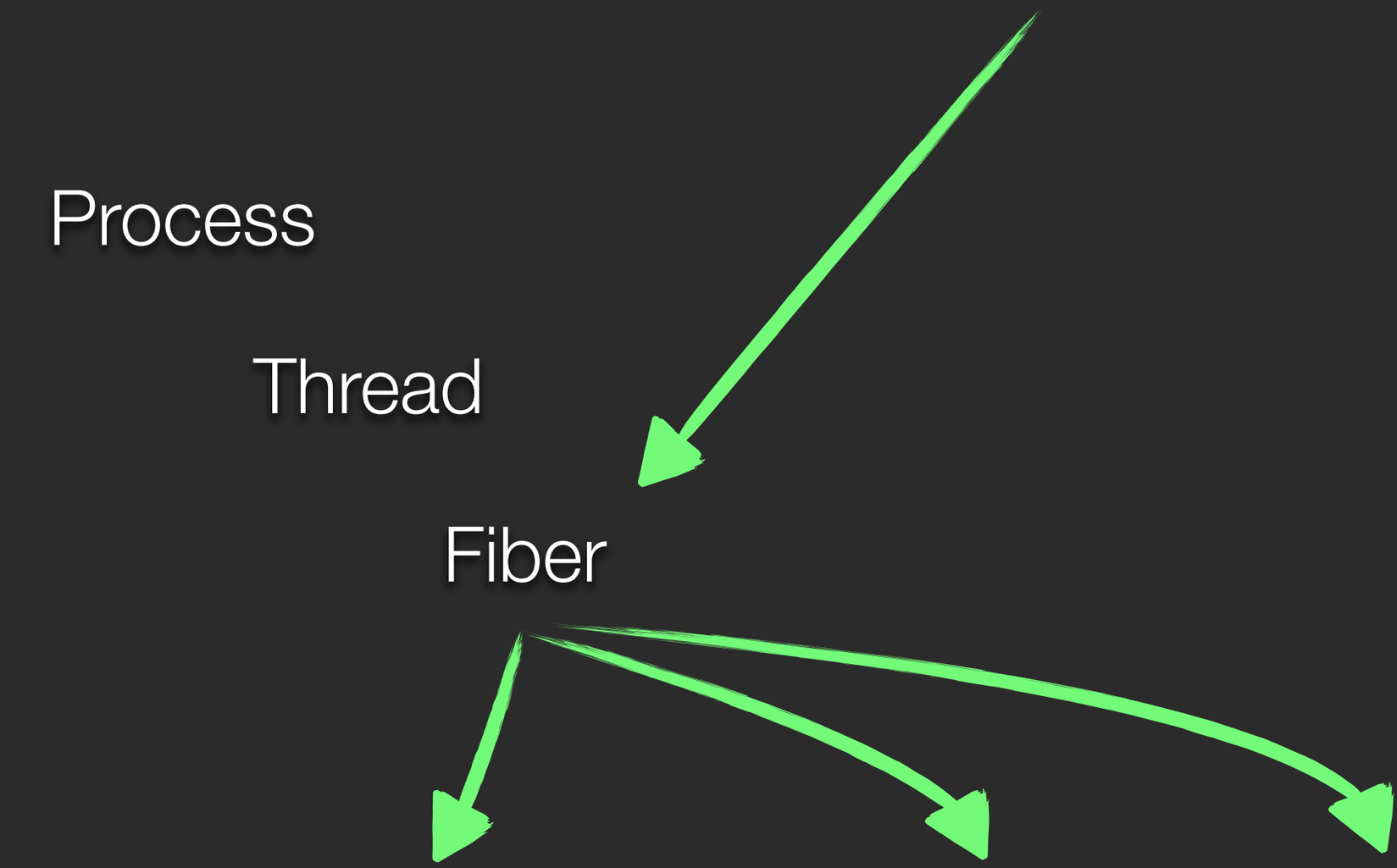
HOW TO USE FUNCTIONS… ASYNC…

@SvenRuppert

Process

Thread

Fiber

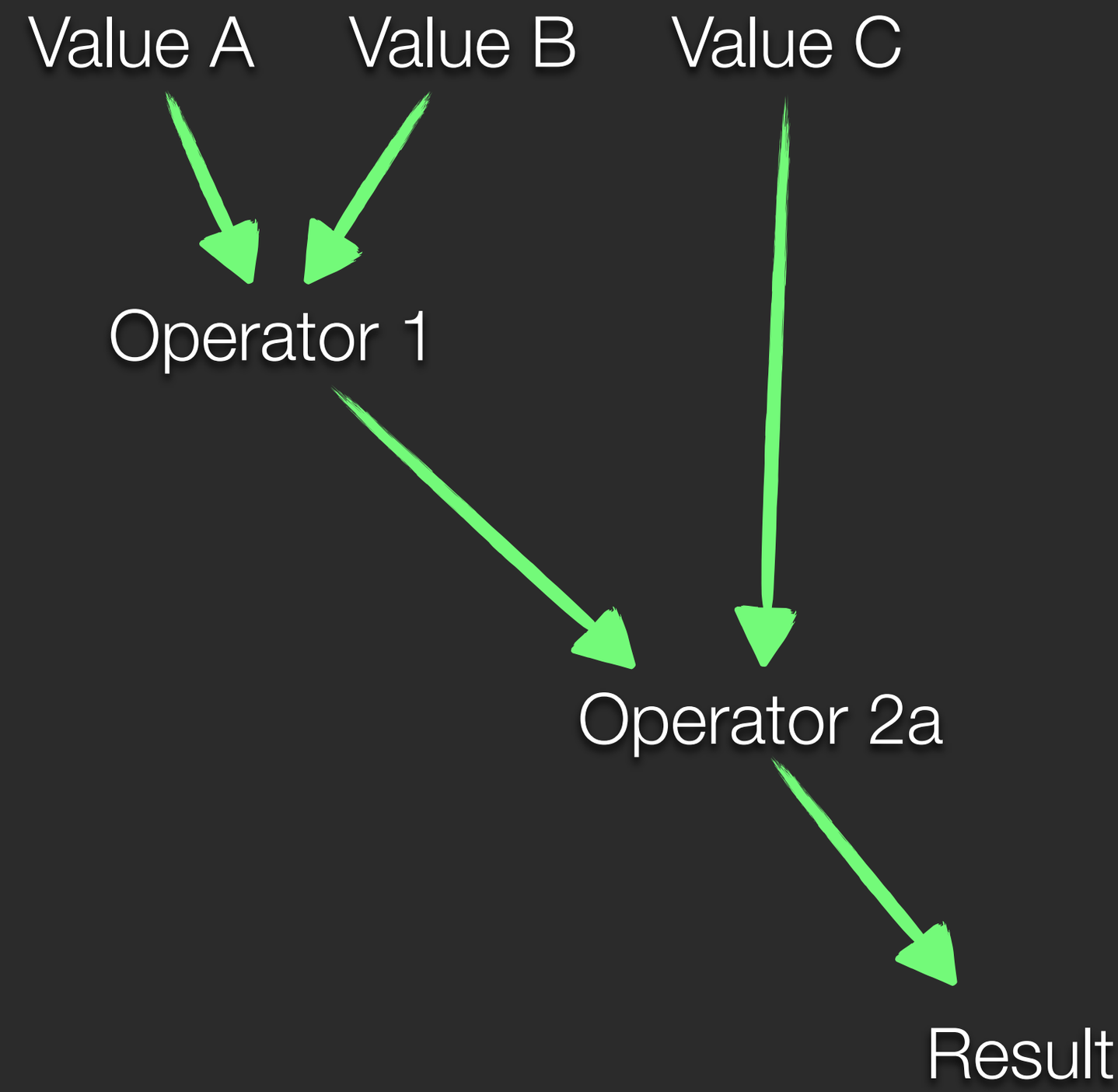


CompletionStage CompletionStage CompletionStage

Java8 - CompletableFuture


```
final ExecutorService pool = Executors
    .newFixedThreadPool(Runtime
                        .getRuntime()
                        .availableProcessors());

final Supplier<String> task = () -> "Hello World";


final CompletableFuture<String> async = CompletableFuture.supplyAsync(task, pool);

final CompletableFuture<Void> result = async.thenAcceptAsync(System.out::println);

result.join();

pool.shutdown();
```

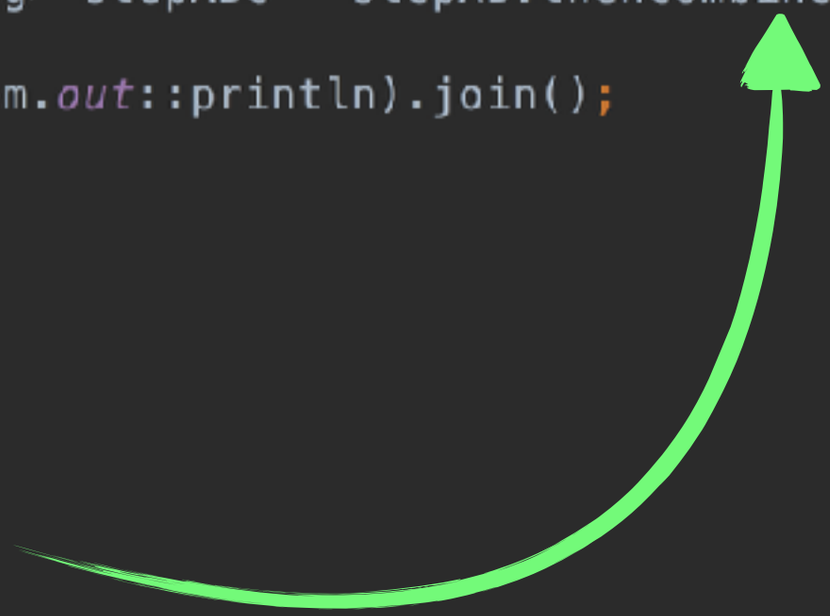
```
final ExecutorService pool = newFixedThreadPool(getRuntime()
                                .availableProcessors());

supplyAsync(() -> "Hello World", pool)
    .thenAcceptAsync(System.out::println)
    .join();

pool.shutdown();
```


Value A
Value B
Value C
Operator 1
Operator 2a
Result

Java8 - CompletableFuture

@SvenRuppert

```
public static void main(String[] args) {

  final ExecutorService pool = newFixedThreadPool(getRuntime().availableProcessors());

  final CompletableFuture<String> stepA = supplyAsync(() -> "A", pool);
  final CompletableFuture<String> stepB = supplyAsync(() -> "B", pool);
  final CompletableFuture<String> stepC = supplyAsync(() -> "C", pool);

  final CompletableFuture<String> stepAB = stepA.thenCombineAsync(stepB, (a, b) -> a + b);
  final CompletableFuture<String> stepABC = stepAB.thenCombineAsync(stepC, (ab, c) -> ab + c);

  stepABC.thenAcceptAsync(System.out::println).join();

  pool.shutdown();

}
```



Java8 - CompletableFuture

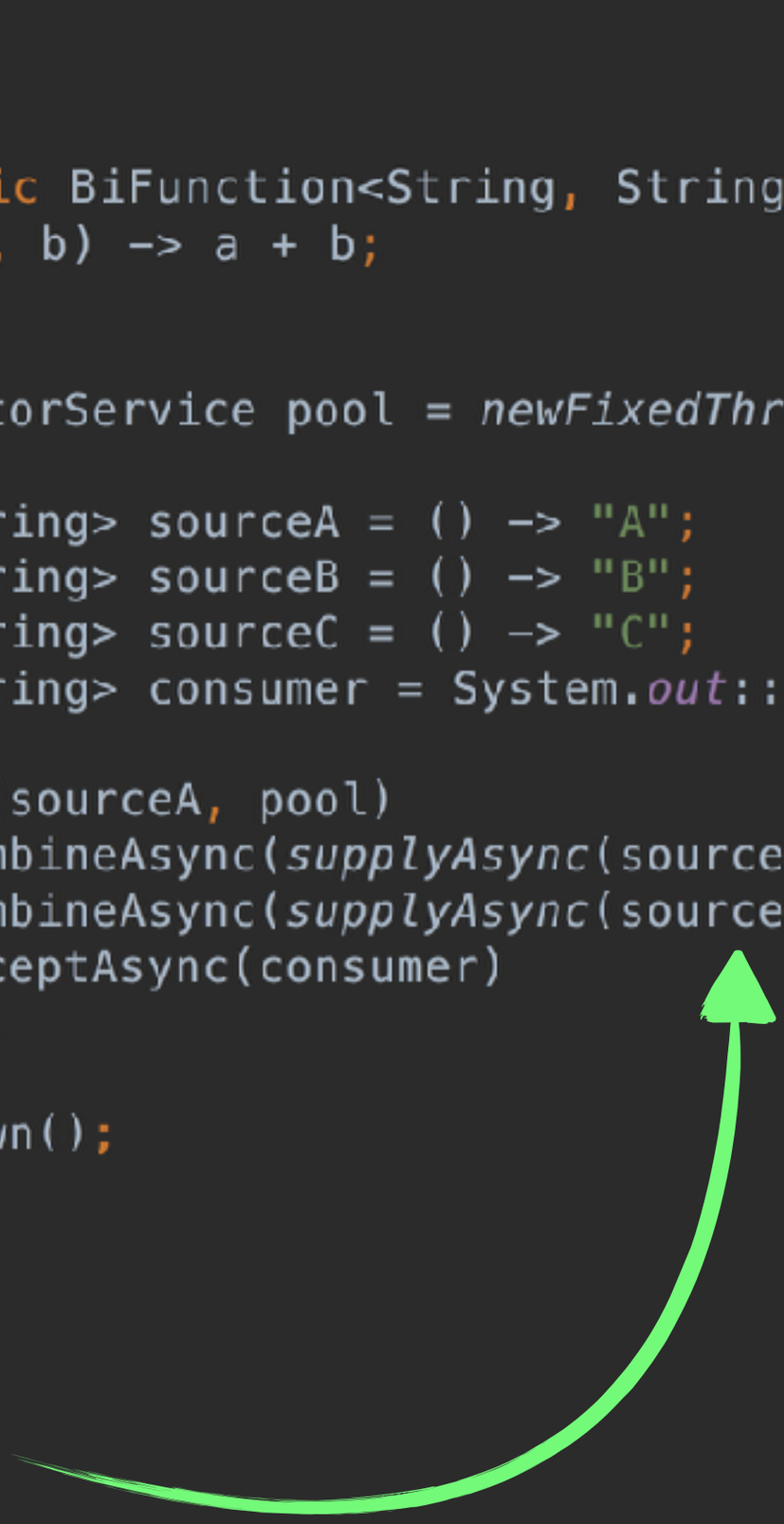
@SvenRuppert

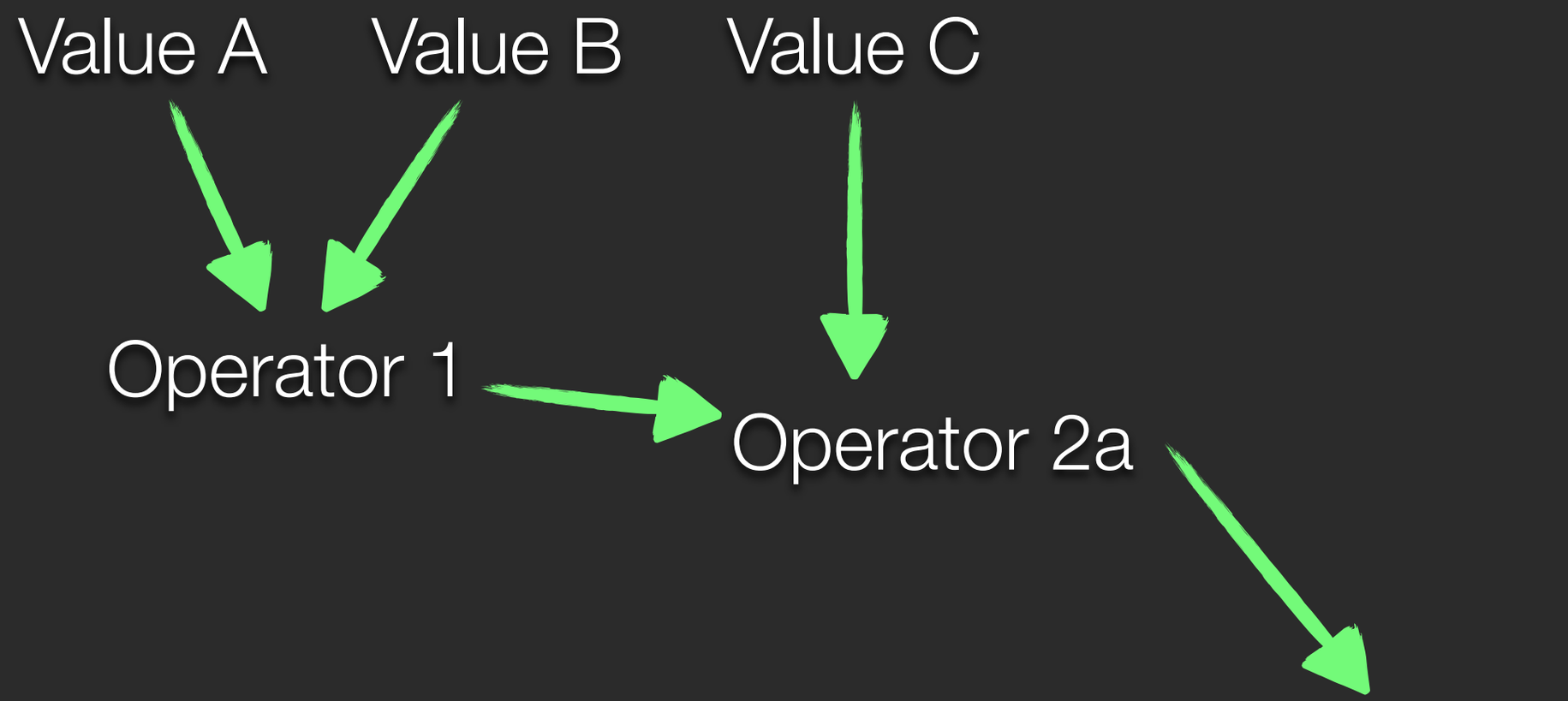
```
public static void main(String[] args) {

    final ExecutorService pool = newFixedThreadPool(getRuntime().availableProcessors())

    supplyAsync(() -> "A", pool)
        .thenCombineAsync(supplyAsync(() -> "B", pool), (a, b) -> a + b)
        .thenCombineAsync(supplyAsync(() -> "C", pool), (ab, c) -> ab + c)
        .thenAcceptAsync(System.out::println)
        .join();

    pool.shutdown();
}
```

Java8 - CompletableFuture

@SvenRuppert

```
public static BiFunction<String, String, String> fktConcat(){
  return (a, b) -> a + b;
}

final ExecutorService pool = newFixedThreadPool(getRuntime().availableProcessors());

Supplier<String> sourceA = () -> "A";
Supplier<String> sourceB = () -> "B";
Supplier<String> sourceC = () -> "C";
Consumer<String> consumer = System.out::println;

supplyAsync(sourceA, pool)
    .thenCombineAsync(supplyAsync(sourceB, pool), fktConcat())
    .thenCombineAsync(supplyAsync(sourceC, pool), fktConcat())
    .thenAcceptAsync(consumer)
    .join();

pool.shutdown();
```






```
supplyAsync(sourceA, pool)
    .thenCombineAsync(supplyAsync(sourceB, pool), fktConcat())
    .thenCombineAsync(supplyAsync(sourceC, pool), fktConcat())
    .thenAcceptAsync(consumer)
    .join();
```

EXERCISES

CompletableFuture

HOW TO USE FUNCTIONS… ASYNC…

@SvenRuppert

CompletableFutureQueue

HOW TO CREATE A GRAPH EASY..

@SvenRuppert

CompletableFutureQueue

CompletableFuture is active after creation

want to define a CompletableFuture without starting it

maybe creating it interactively

we need a function !

Function<T, CompletableFuture<R>>

CompletableFutureQueue

@SvenRuppert

```
private static final Function<String, String> step1 = (input) -> input.toUpperCase();
private static final Function<String, String> step2 = (input) -> input + " next A";
private static final Function<String, String> step3 = (input) -> input + " next B";
```

```
//manual
Function<String, CompletableFuture<String>> f = (value) ->
    CompletableFuture
        .completedFuture(value)
        .thenComposeAsync(v -> supplyAsync(() -> step1.apply(v)))
        .thenComposeAsync(v -> supplyAsync(() -> step2.apply(v)))
        .thenComposeAsync(v -> supplyAsync(() -> step3.apply(v)));
```

```
final Function<String, CompletableFuture<String>> combinedFkt
    = CFQ
    .define(step1)
    .thenCombineAsync(step2)
    .thenCombineAsync(step3)
    .resultFunction();

final CompletableFuture<String> hello2 = combinedFkt.apply( t: "Hello");
```

CompletableFutureQueue

@SvenRuppert

```
public class CompletableFutureQueue<T, R> {

  private Function<T, CompletableFuture<R>> resultFunction;

  private CompletableFutureQueue(Function<T, CompletableFuture<R>> resultFunction) {
    this.resultFunction = resultFunction;
  }

  /**...*/
  public static <T, R> CompletableFutureQueue<T, R> define(Function<T, R> transformation) {
    return new CompletableFutureQueue<>(t -> CompletableFuture.completedFuture(transformation.apply(t)));
  }

  /**...*/
  public <N> CompletableFutureQueue<T, N> thenCombineAsync(Function<R, N> nextTransformation) {
    final Function<T, CompletableFuture<N>> f = this.resultFunction
        .andThen(before -> before.thenComposeAsync(v -> supplyAsync(() -> nextTransformation.apply(v))));
    return new CompletableFutureQueue<>(f);
  }

  /**...*/
  public Function<T, CompletableFuture<R>> resultFunction() { return this.resultFunction; }
}
```

```
final Function<String, CompletableFuture<String>> combinedFkt
    = CFQ
    .define(step1)
    .thenCombineAsync(step2)
    .thenCombineAsync(step3)
    .resultFunction();

final CompletableFuture<String> hello2 = combinedFkt.apply( t: "Hello");
```

EXERCISES

CompletableFutureQueue

HOW TO CREATE A GRAPH EASY..

@SvenRuppert

Java 9

some nice news…

@SvenRuppert

Java9 - Optional


```
final Function<Integer, Optional<Integer>> function
    = (value) -> (value <= 4) ? Optional.of(value) : Optional.empty();

final List<Integer> collectionJava8 = Stream
    .of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    .map(function) // only to wrap
    .filter(Optional::isPresent)
    .map(Optional::get)
    .collect(Collectors.toList());

final List<Integer> collectionJava9 = Stream
    .of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    .map(function) // only to wrap
    .flatMap(Optional::stream)
    .collect(Collectors.toList());
```

we got Publisher / Subscriber

```
public class JEP266v001 {
  public static void main(String[] args) throws InterruptedException, ExecutionException {
    CompletableFuture consume = null;
    try (SubmissionPublisher pup = new SubmissionPublisher<String>()) {
      consume = pup.consume(System.out::println);
      IntStream.range(1, 10).forEach(pup::submit);
    } finally {
      consume.get();
    }

  }
}
```

Var

INTERFACE OR IMPLEMENTATION ?

@SvenRuppert

EXERCISES

Var

INTERFACE OR IMPLEMENTATION ?

@SvenRuppert

Summary

places to read more about it

www.functional-reactive.org

https://github.com/functional-reactive

Summary

please, follow me ;-)

@SvenRuppert