JAVA 8

LAMBDAS, MONADS && JAVA COLLECTIONS

Grzegorz Piwowarek

GRZEGORZ PIWOWAREK

@PIVOVARIT

GRZEGORZ PIWOWAREK

@PIVOVARIT

visionsoftondal.com

Plan:

-lambda expressions
-java.util.function
-monad
-Optional
-Stream

lambda expressions

(...) -> statement

- Anonymous function

lambda expressions

```
x -> x + 1
```

lambda expressions

x -> x + 1

```
Function<Integer, Integer> foo1 = x -> x + 1;

Function<String, String>   foo2 = x -> x + 1;


List<Integer> foo3 = new ArrayList<>();
```

- No type information

lambda expressions

```
x -> x + 1
() -> 42
() -> {return 42;}
(x, y) -> {}
() -> {}
```

method references

```
Function<Integer, Integer> foo1 = x -> InferenceExample.fooFunction1(x);

Function<Integer, Integer> foo2 = this::fooFunction2;

Function<Integer, Integer> foo3 = InferenceExample::fooFunction1;

public static Integer fooFunction1(Integer i) {
    return i + 1;
}

public Integer fooFunction2(Integer i) {
    return i + 1;
}
```

java.util.function

```java
@FunctionalInterface
public interface Function<T, R> {

    /**
     * Applies this function to the given argument.
     *
     * @param t the function argument
     * @return the function result
     */
    R apply(T t);
```

java.util.function

```
@Test
public void shouldComposeFunctions() throws Exception {
    // given
    Function<Integer, Integer> addOne = i -> i + 1;
    Function<Integer, Integer> timesTwo = i -> 2 * i;

    // when
    addOne.
    //
}
```

```
apply(Integer t)                                                    Integer
andThen(Function<? super Integer, ? extends V> after)               Function<Integer, V>
compose(Function<? super V, ? extends Integer> before)              Function<V, Integer>
```

Function<T, R>, BiFunction<T,U,R>

Consumer<T> extends Function<T,Void>
Supplier<T> extends Function<Void, T>
Predicate<T> extends Function<T, Boolean>
UnaryOperator<T> extends Function<T, T>
BinaryOperator<T> extends BiFunction<T,T,T>

Function<T, R>, BiFunction<T,U,R>

Consumer<T> extends Function<T,Void>
Supplier<T> extends Function<Void, T>
Predicate<T> extends Function<T, Boolean>
UnaryOperator<T> extends Function<T, T>
BinaryOperator<T> extends BiFunction<T,T,T>

Function<T, R>, BiFunction<T,U,R>

Consumer<T> extends Function<T,Void>
Supplier<T> extends Function<Void, T>
Predicate<T> extends Function<T, Boolean>
UnaryOperator<T> extends Function<T, T>
BinaryOperator<T> extends BiFunction<T,T,T>

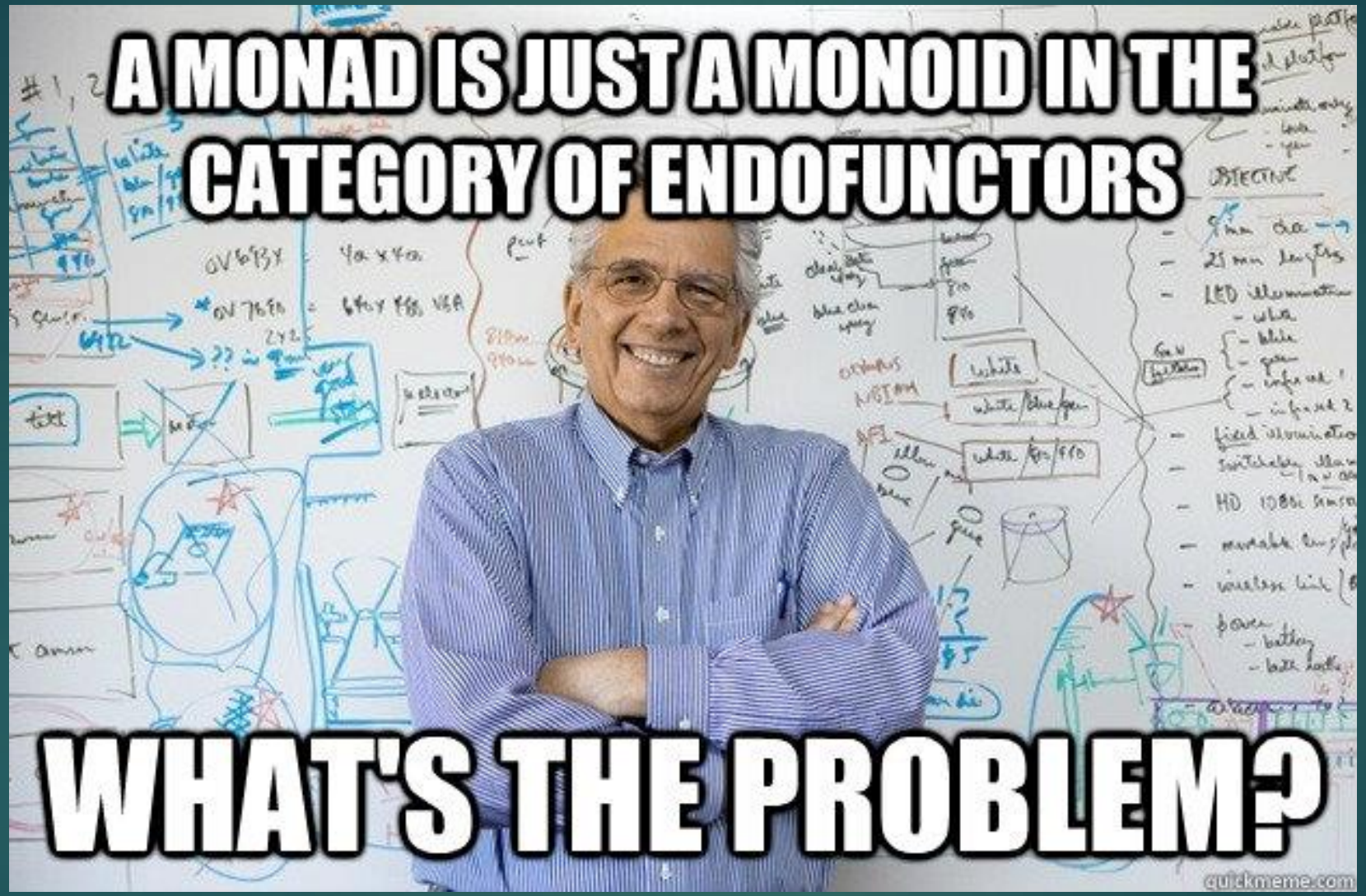
Function<T, R>, BiFunction<T,U,R>

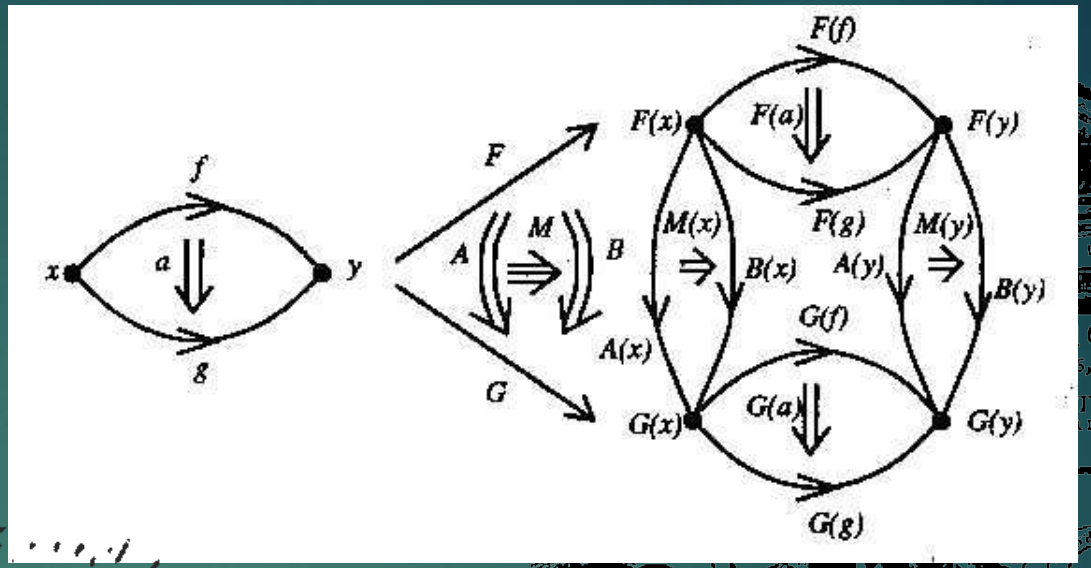
Consumer<T> extends Function<T,Void>
Supplier<T> extends Function<Void, T>
Predicate<T> extends Function<T, Boolean>
UnaryOperator<T> extends Function<T, T>
BinaryOperator<T> extends BiFunction<T,T,T>

Function<T, R>, BiFunction<T,U,R>

Consumer<T> extends Function<T,Void>
Supplier<T> extends Function<Void, T>
Predicate<T> extends Function<T, Boolean>
UnaryOperator<T> extends Function<T, T>
BinaryOperator<T> extends BiFunction<T,T,T>

MONAD


A MONAD IS JUST A MONOID IN THE CATEGORY OF ENDOFUNCTORS
WHAT'S THE PROBLEM?
quickmeme.com



GOOGLE IMAGES…

MONAD

http://got-steam.com/

Design pattern

Why bother? :

Boilerplate -1
Readability +1
Complexity -1
Responsibility -1

MONAD

type: M<T>

"unit": T -> M<T>

"bind": M<T>.bind(T -> M<U>) = M<U>

"bind": M<T>.bind(T -> U)= M<U>

MONAD

type: M<T>

"unit": T -> M<T>

"bind": M<T> bind(T -> M<U>) = M<U>

"bind": M<T>.bind(T -> U)= M<U>

"bind": M<T> bind(T -> U)= M<U>

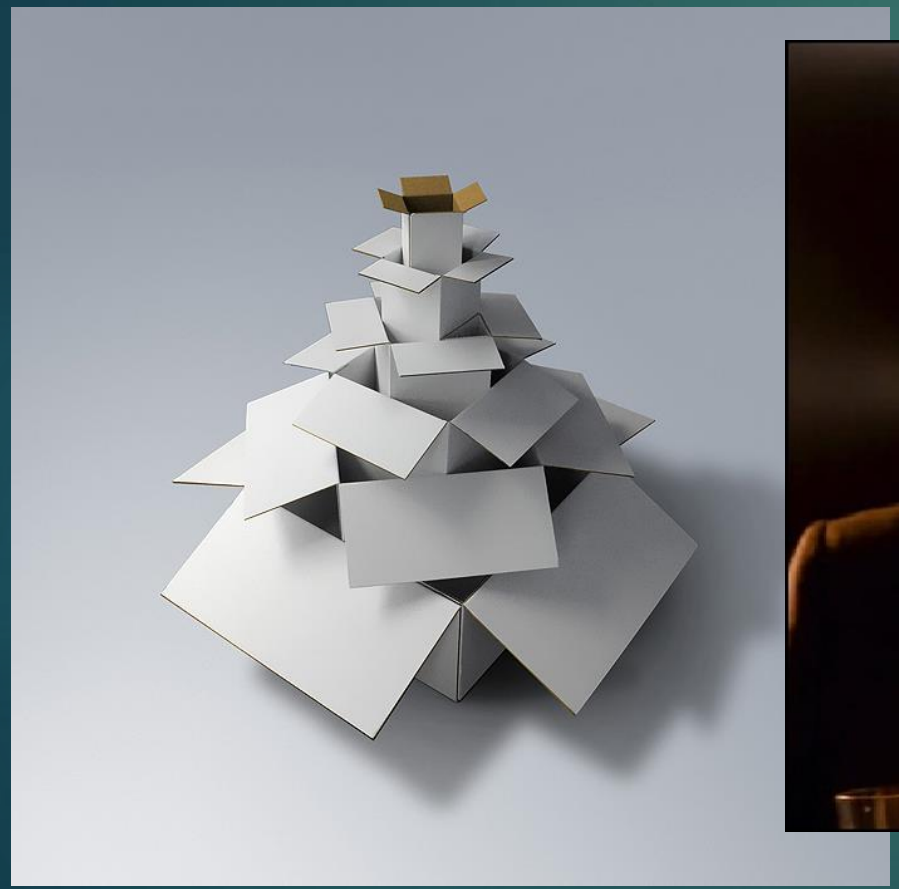
What if U: M<?> ?

"bind": M<T> bind(T -> U)= M<U>

What if U: M<?> ?

"bind": M<T> bind(T -> U)= M<U>

What if U: M<?> ?



http://www.ivanaborovnjak.com/project/box-in-a-box/

Monads in Java 8

Optional

Stream

CompletableFuture

Monads in Java 8

Optional

Stream

CompletableFuture

Optional

Encapsulation of operations on optional values

Optional

type: M<T>

"unit": T -> M<T>

"bind": M<T> bind(T -> M<U>) = M<U>

Optional

type: Optional<T>

"unit": T -> M<T>

"bind": M<T> bind(T -> M<U>) = M<U>

Optional

type: Optional<T>

"unit": Optional.ofNullable(), Optional.of()

"bind": M<T> bind(T -> M<U>) = M<U>

Optional

type: Optional<T>

"unit": Optional.ofNullable(), Optional.of()

"bind": Optional.flatMap()

Filtering an Optional

.filter(Predicate<T>)

Unwrapping an Optional

.get()

.orElse(T default)

.orElseGet(Supplier<T>)

.orElseThrow(Supplier<Ex>)

.ifPresent(Consumer<T>)

Java 7 style

```java
public static String fooJava7(Map<String, Person> people) {
    final Person kowalski = people.get("Kowalski");

    if (kowalski != null) {
        final Address address = kowalski.getAddress();

        if (address != null) {
            final City city = address.getCity();

            if (city != null) {
                final String cityName = city.getCityName();

                if (!cityName.isEmpty()) {
                    return cityName;
                }
            }
        }
    }
    return "UNKNOWN";
}
```

Java 8 style

```
public static String fooJava8(Map<String, Person> people) {
    return Optional.ofNullable(people.get("Kowalski"))
            .map(Person::getAddress)
            .map(Address::getCity)
            .map(City::getCityName)
            .filter(name -> !name.isEmpty())
            .orElse("UNKNOWN");
}
```

Java 8 style - flatMap

```
public static String fooJava8FlatMap(Map<String, Person> people) {
    return Optional.ofNullable(people.get("Kowalski"))
            .map(Person::getAddress)
            .flatMap(Address::getOptionalCity) // because we do not want Optional<Optional<City>>
            .map(City::getCityName)
            .filter(name -> !name.isEmpty())
            .orElse("UNKNOWN");
}
```

Java 7,5 style ;)

```java
public static String fooJava75(Map<String, Person> people) {
    final Optional<Person> kowalski = Optional.ofNullable(people.get("Kowalski"));

    if (kowalski.isPresent()) {
        final Optional<City> city = kowalski.get().getAddress().getOptionalCity();
        if (city.isPresent()) {
            return city.get().getCityName();
        }
    }

    return "UNKNOWN";
}
```

Java 7,5 style ;)

```java
public static String fooJava75(Map<String, Person> people) {
    final Optional<Person> kowalski = Optional.ofNullable(people.get("Kowalski"));

    if (kowalski.isPresent()) {
        final Optional<City> city = kowalski.get().getAddress().
        if (city.isPresent()) {
            return city.get().getCityName();
        }
    }

    return "UNKNOWN";
}
```

Stream

Encapsulation of operations on multiple items

Stream

type: Stream<T>

"unit": Stream.of(), Arrays.stream(), Collection.stream()

"bind": Stream.flatMap()

Stream

```
Stream.of("a", "b", "c")
        .forEach(e -> System.out.println(e));

//a
//b
//c
```

Stream

```
Stream.of("a", "b", "c")
        .map(s -> s.toUpperCase())
        .forEach(e -> System.out.println(e));

//A
//B
//C
```

Stream

```
Stream.of("a", "b", "c")
        .map(s -> s.toUpperCase())
        .map(s -> s + "_postfix")
        .forEach(e -> System.out.println(e));

//A_postfix
//B_postfix
//C_postfix
```

Stream && Optional

```
public String getCityNameFor(String person) {
    return people.stream()
            .filter(Predicate.isEqual("Kowalski"))
            .map(Person::getAddress)
            .map(Address::getCity)
            .map(City::getCityName)
            .findAny()
            .orElseThrow(RuntimeException::new);

}
```

Stream

lazy-initialized
nonreusable

intermediate operations

.map()
.flatMap()
.filter()
.peek()

intermediate operations

```
public static void main(String[] args) {
    final Stream<Integer> stream = Stream.of(1, 2, 3)
            .map(i -> {
                System.out.println(i.toString());
                return i;
            });
}
```

.map()
.flatMap()
.filter()
.peek()

Stream not consumed:
does not print anything

Java7

```
public List<ComponentDto> getComponentsJ7(ResourceId id) {
    final ArrayList<ComponentDto> result = new ArrayList<>();

    for (Map.Entry<ResourceMappingConfiguration, RegistrationData> entry : registrations.entrySet()) {
        if (entry.getKey().getAssignedResource().equals(id)) {
            final Optional<ServiceRegistration<?>> resourceRegistration = entry.getValue().getResourceRegistration();

            if (resourceRegistration.isPresent()) {

                for (Bundle bundle : resourceRegistration.get().getReference().getUsingBundles()) {

                    if (ComponentUtils.isESComponent(bundle)) {
                        final ComponentDto dto = ComponentDto.from(bundle);

                        if (!result.contains(dto)) {
                            result.add(dto);
                        }
                    }
                }
            }

        }
    }

    Collections.sort(result);
    return result;
}
```

Java8

```
public List<ComponentDto> getComponentsJ8(ResourceId id) {
    return registrations.entrySet().stream()
            .filter(e -> e.getKey().getAssignedResource().equals(id))
            .map(Map.Entry::getValue)
            .map(RegistrationData::getResourceRegistration)
            .filter(Optional::isPresent)
            .map(r -> r.get().getReference())
            .flatMap(ref -> stream(ref.getUsingBundles()))
            .filter(ComponentUtils::isESComponent)
            .map(ComponentDto::from)
            .distinct().sorted()
            .collect(toList());
}
```

Consuming Stream

.forEach(Consumer<T>)
.collect()
.reduce(BinaryOperator<T>)
.allMatch(), anyMatch(), noneMatch()
.findFirst(), findAny()
.count()
.toArray()

Stream.reduce()

```
public Integer getLongestCityNameLength (Collection<Person> people) {
    return people.stream()
            .map(p -> p.getAddress().getCity().getCityName())
            .reduce((a, b) -> a.length() > b.length() ? a : b)
            .map(String::length)
            .orElse(0);

}
```

Collectors

.toList(), toMap(), toSet(), toCollection()

.minBy(), maxBy()

.joining()

.partitioningBy()

...and many others

Collectors.toList()

```
public List<City> getCitiesAsList(Collection<Person> people) {
    return people.stream()
            .map(p -> p.getAddress().getOptionalCity())
            .filter(Optional::isPresent)
            .map(Optional::get)
            .distinct()
            .collect(toList()); // with static import
}
```

Collectors.toMap()

```
public Map<Person, String> getPersonCityMap(Collection<Person> people) {
    return people.stream()
            .collect(Collectors.toMap(p -> p, p -> p.getAddress().getCity().getCityName()));

}
```

Collectors.joining()

```
public String getCitiesAsList(Collection<Person> people) {
    return people.stream()
            .map(p -> p.getAddress().getOptionalCity())
            .filter(Optional::isPresent)
            .map(Optional::get)
            .map(City::getCityName)
            .distinct()
            .collect(Collectors.joining(","));
}
```

Debuggability?

IntelliJ IDEA:
- v14.0 - partial support
- v15.0 - full support

Stream in APIs

BufferedReader.lines()

Files.newDirectoryStream()

Random.ints()

…

you want more?

https://github.com/
jasongoodwin/better-java-monads

Thank You!

REFERENCES:

-"MONADIC JAVA" BY MARIO FUSCO
-"WHAT'S WRONG WITH JAVA 8" BY PIERRE-YVES SAUMONT
-WWW.ORACLE.COM
-"A FISTFUL OF MONADS" - LEARN YOU A HASKELL FOR GREAT GOOD