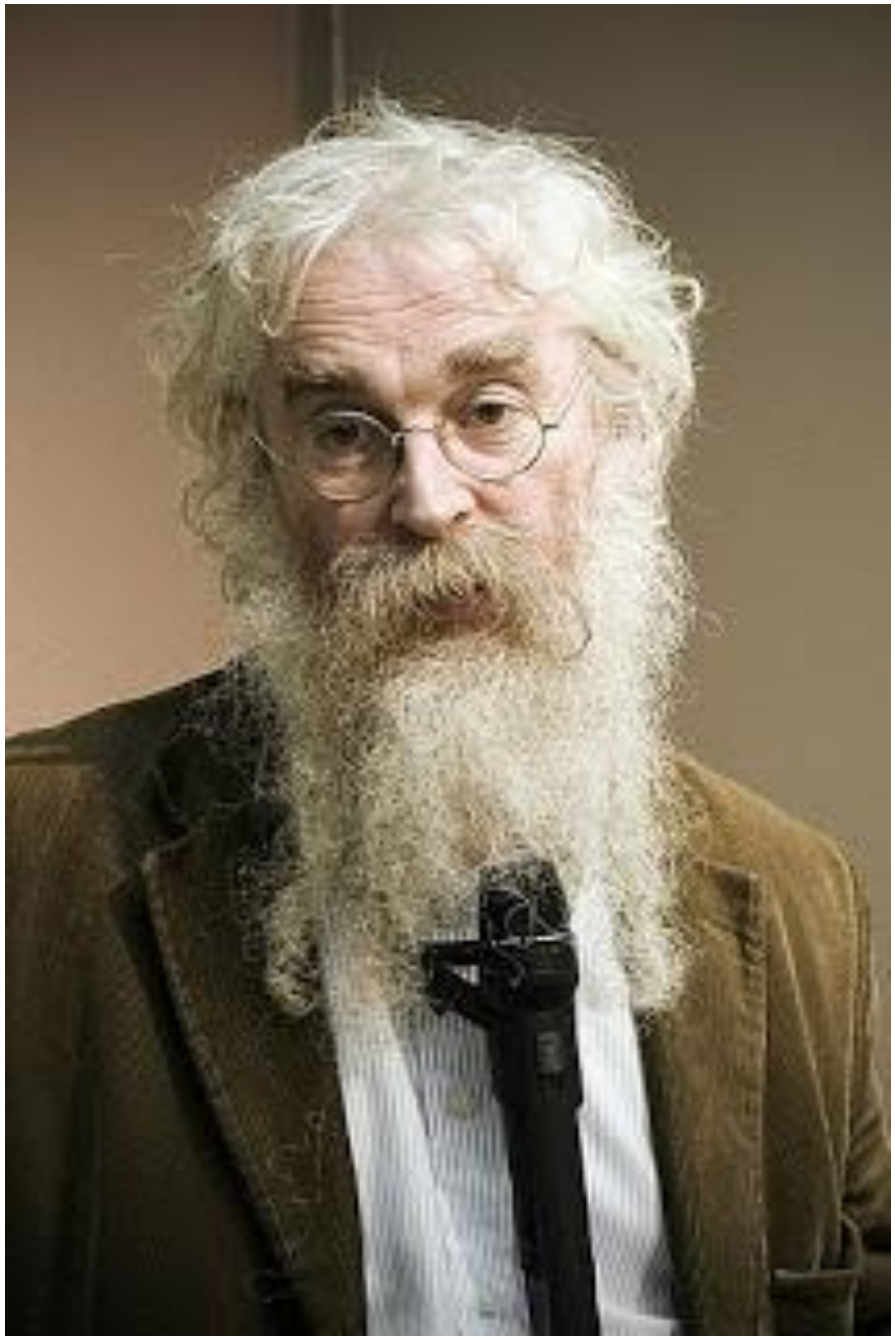
Pragmatic Functional Refactoring with Java 8

Raoul-Gabriel Urma (@raoulUK)
Richard Warburton (@RichardWarburto)




I
LEGO

First-class Functions

Currying

Immutability

Optional Data Types

Conclusions

Step 1: filtering invoices from Oracle

```
public List<Invoice> findInvoicesFromOracle(List<Invoice> invoices) {
    List<Invoice> result = new ArrayList<>();
    for(Invoice invoice: invoices) {
        if(invoice.getCustomer() == Customer.ORACLE) {
            result.add(invoice);
        }
    }
    return result;
}
```

Step 2a: abstracting the customer

```
public List<Invoice> findInvoicesFromCustomer(List<Invoice> invoices,
                                              Customer customer) {
    List<Invoice> result = new ArrayList<>();
    for(Invoice invoice: invoices) {
        if(invoice.getCustomer() == customer) {
            result.add(invoice);
        }
    }
    return result;
}
```

Step 2b: abstracting the name

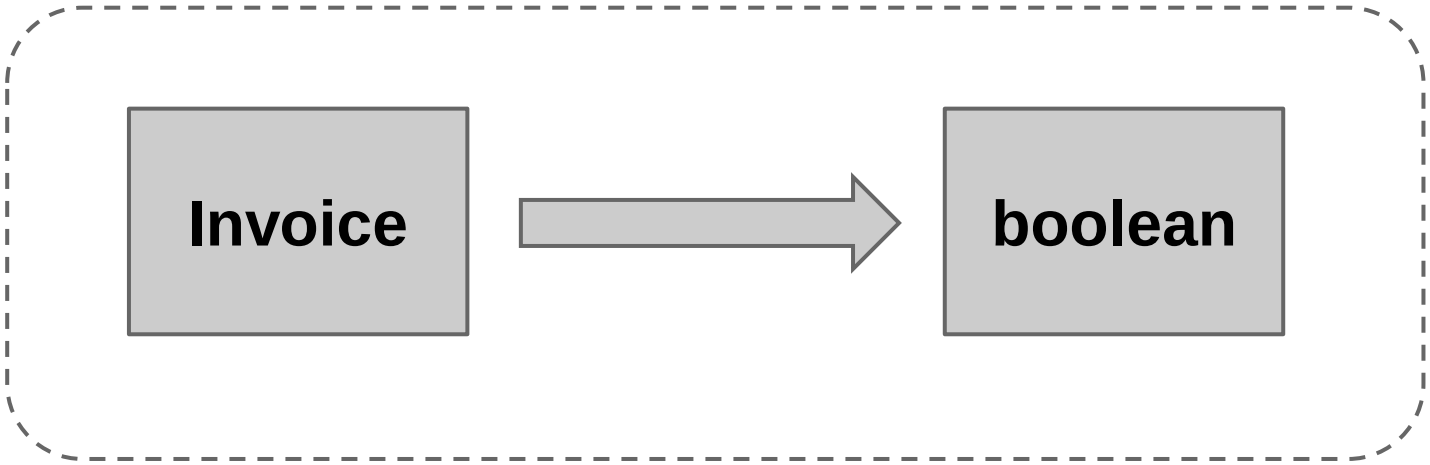
```
public List<Invoice> findInvoicesEndingWith(List<Invoice> invoices,
                                            String suffix) {
    List<Invoice> result = new ArrayList<>();
    for(Invoice invoice: invoices) {
        if(invoice.getName().endsWith(suffix)) {
            result.add(invoice);
        }
    }
    return result;
}
```

Step 3: messy code-reuse!

```
private List<Invoice> findInvoices(List<Invoice> invoices,
                                   Customer customer,
                                   String suffix,
                                   boolean flag) {
    List<Invoice> result = new ArrayList<>();
    for(Invoice invoice: invoices) {
        if((flag && invoice.getCustomer() == customer)
           || (!flag && invoice.getName().endsWith(suffix))) {
            result.add(invoice);
        }
    }
    return result;
}
```

Step 4a: modeling the filtering criterion

```
public interface Predicate<T> {
    boolean test(T t);
}
```



Predicate<Invoice>

Step 4b: using different criterion with objects

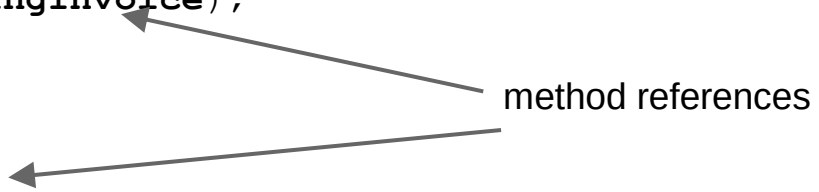
```
private List<Invoice> findInvoices(List<Invoice> invoices,
                                   Predicate<Invoice> p) {
    List<Invoice> result = new ArrayList<>();
    for(Invoice invoice: invoices) {
        if(p.test(invoice)) {
            result.add(invoice);
        }
    }
    return result;
}
```

Step 4b: using different criterion with objects

```
List<Invoice> specificInvoices =
    findInvoices(invoices, new FacebookTraining());

class FacebookTraining implements Predicate<Invoice> {
    @Override
    public boolean test(Invoice invoice) {
        return invoice.getCustomer() == Customer.FACEBOOK
               && invoice.getName().endsWith("Training");
    }
}
```

Step 5: method references

```
public boolean isOracleInvoice(Invoice invoice) {
    return invoice.getCustomer() == Customer.ORACLE;
}

public boolean isTrainingInvoice(Invoice invoice) {
    return invoice.getName().endsWith("Training");
}

List<Invoice> oracleInvoices =
    findInvoices(invoices, this::isOracleInvoice);


List<Invoice> trainingInvoices =
    findInvoices(invoices, this::isTrainingInvoice);
```

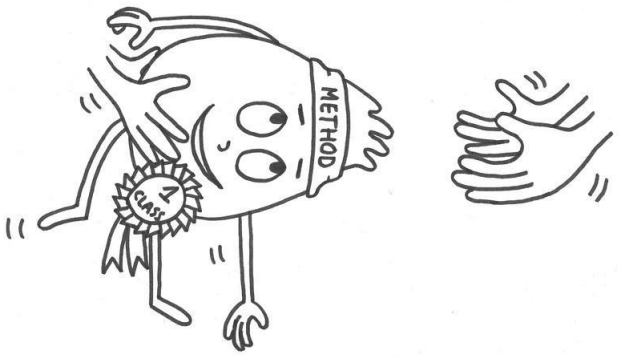


Step 6: lambdas

```
List<Invoice> oracleInvoices =
    findInvoices(invoices,
                 invoice -> invoice.getCustomer() == Customer.ORACLE);

List<Invoice> trainingInvoices =
    findInvoices(invoices,
                 invoice -> invoice.getName().endsWith("Training"));
```

First-class functions

- Common Object Oriented concept: Function Object
- All it means is the ability to use a function just like a regular value
  - Pass it as argument to a method
  - Store it in a variable
- Helps cope with requirement changes



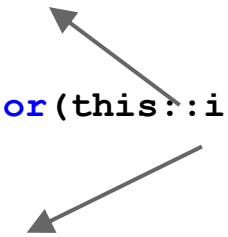
Composing functions

Functions

Composing functions

Composing functions: example

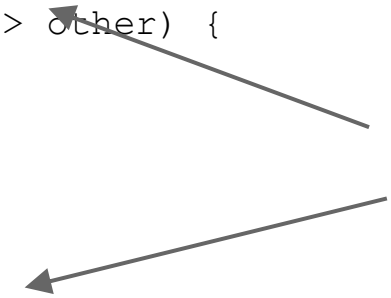
```
import java.util.function.Predicate;
Predicate<Invoice> isFacebookInvoice = this::isFacebookInvoice;

List<Invoice> facebookAndTraining =
    invoices.stream()
            .filter(isFacebookInvoice.and(this::isTrainingInvoice))
            .collect(toList());

List<Invoice> facebookOrGoogle =
    invoices.stream()
            .filter(isFacebookInvoice.or(this::isGoogleInvoice))
            .collect(toList());
```



creating more complex functions from building blocks

Composing functions: why does it work?

```
public interface Predicate<T> {

    boolean test(T t);

    default Predicate<T> and(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) && other.test(t);
    }

    default Predicate<T> or(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) || other.test(t);
    }
}
```



returns a new function that is the result of composing two functions

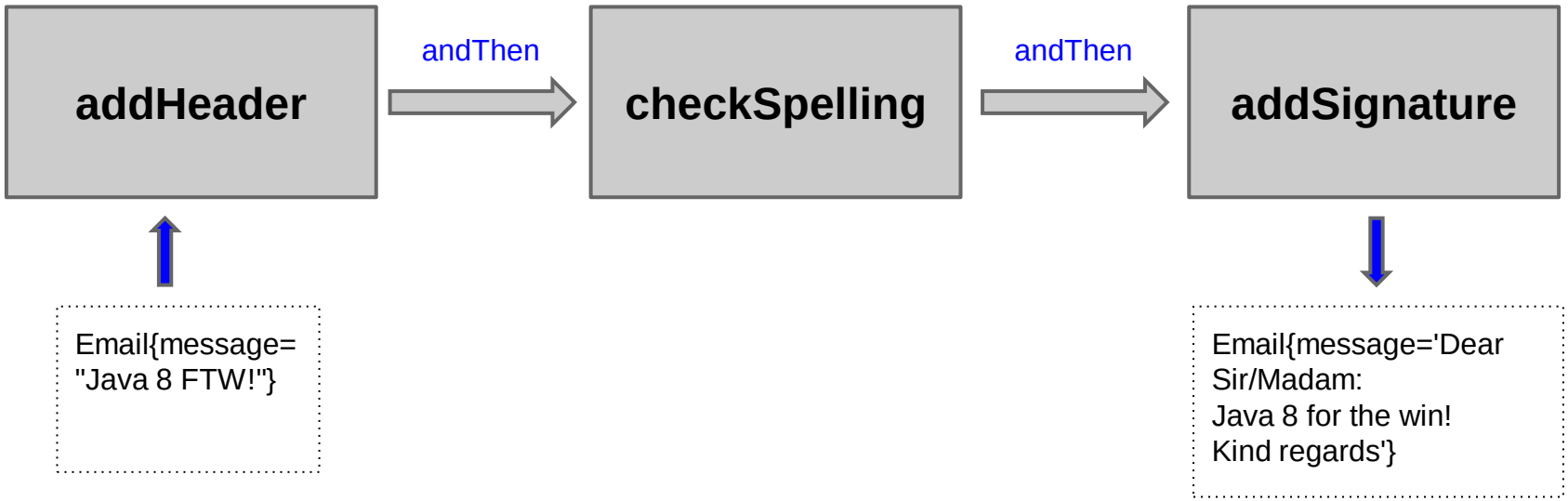
Creating function pipelines (1)

```
public class Email {
    private final String message;

    public Email(String message) {
        this.message = message;
    }

    public Email addHeader() {
        return new Email("Dear Sir/Madam:\n" + message);
    }

    public Email checkSpelling() {
        return new Email(message.replaceAll("FTW", "for the win"));
    }

    public Email addSignature() {
        return new Email(message + "\nKind regards");
    }
}
```

Creating function pipelines (2)

```
import java.util.function.Function;
Function<Email, Email> addHeader = Email::addHeader;

Function<Email, Email> processingPipeline =
    addHeader.andThen(Email::checkSpelling)
             .andThen(Email::addSignature);
```

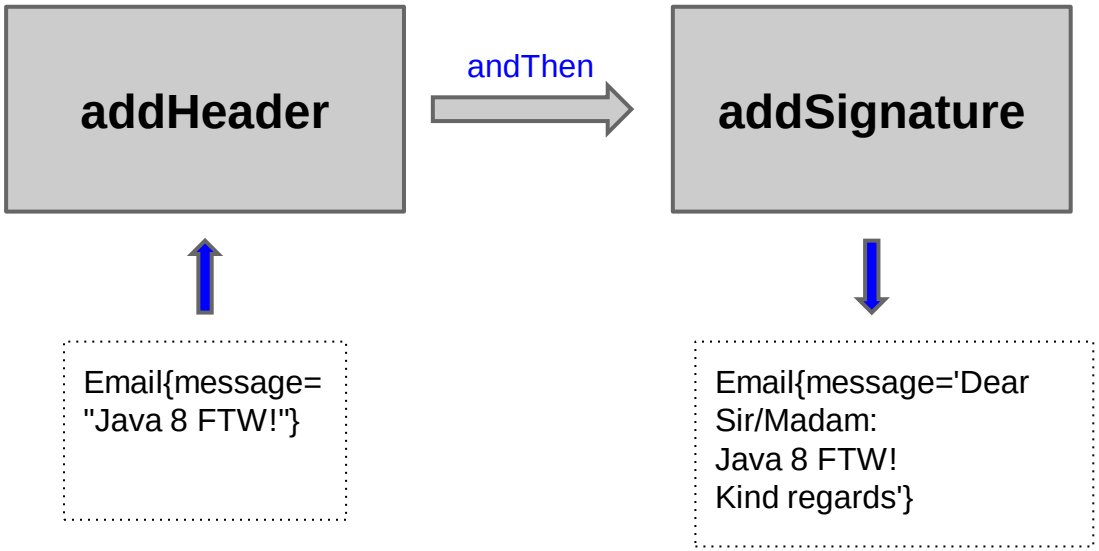
composing functions



Creating function pipelines (3)

```
import java.util.function.Function;
Function<Email, Email> addHeader = Email::addHeader;

Function<Email, Email> processingPipeline =
    addHeader.andThen(Email::addSignature);
```

composing functions



First-class Functions

Currying

Immutability

Optional Data Types

Conclusions

Example: A Conversion Function

```
double convert(double amount, double factor, double base)
{
    return amount * factor + base;
}

// Usage
double result = convert(10, 1.8, 32);
assertEquals(result, 50, 0.0);
```

Using the conversion function

```
// Temperatures
double cInF = convert(10, 1.8, 32);
double higherTemp = convert(30, 1.8, 32);

// Currencies
double dollarsInPounds = convert(10, 0.6, 0);

// Distance
double kilometresInMiles = convert(27, 0.62137, 0);
```

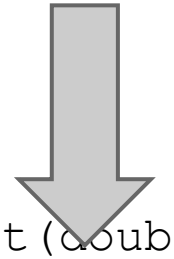
So what’s the problem?

- Abstraction
  - How do you write something that operates on different conversion functions?

- Reuse
  - How do you use the conversion function in different places?

Currying the conversion function

```
double convert(double amount, double factor, double base) {
    return amount * factor + base;
}
```



```
DoubleUnaryOperator convert(double factor, double base) {
    return amount -> amount * factor + base;
}
```

Using the conversion function

```
DoubleUnaryOperator convert(double factor, double base) {
    return amount -> amount * factor + base;
}

DoubleUnaryOperator convertCtoF = convert(1.8, 32);

double result = convertCtoF.applyAsDouble(10);
assertEquals(result, 50, 0.0);
```

Partial Application Examples

```
// Temperatures
DoubleUnaryOperator convertCtoF = convert(1.8, 32);

// Currencies
DoubleUnaryOperator convert$to£ = convert(0.6, 0);

// Distance
DoubleUnaryOperator convertKmToMi = convert(0.62137, 0);
```

Definition

```
// int -> (int -> int)
IntFunction<IntUnaryOperator>
curry(IntBinaryOperator biFunction) {
    return f -> (s -> biFunction.applyAsInt(f, s));
}
```

Usage

```
IntFunction<IntUnaryOperator> add
    = curry((f, s) -> f + s);

int result = add.apply(1)
                .applyAsInt(2);
assertEquals(3, result);
```

Example currying use cases

- Large factory methods
  - Partially apply some of the arguments and then pass this factory object around.

- Combinators libraries
  - Parsers
  - HTML templating

Summary

- Currying is about splitting up the arguments of a function.

- Partial Application is a function “eating” some of its arguments and returning a new function

First-class Functions

Currying

Immutability

Optional Data Types

Conclusions

Mutable objects

```
public class TrainJourney {

    private int price;
    private TrainJourney onward;

    public TrainJourney(int price, TrainJourney onward) {
        this.price = price;
        this.onward = onward;
    }

    public int getPrice() {
        return price;
    }

    public TrainJourney getOnward() {
        return onward;
    }

    public void setPrice(int price) {
        this.price = price;
    }

    public void setOnward(TrainJourney onward) {
        this.onward = onward;
    }
}
```

Immutable objects

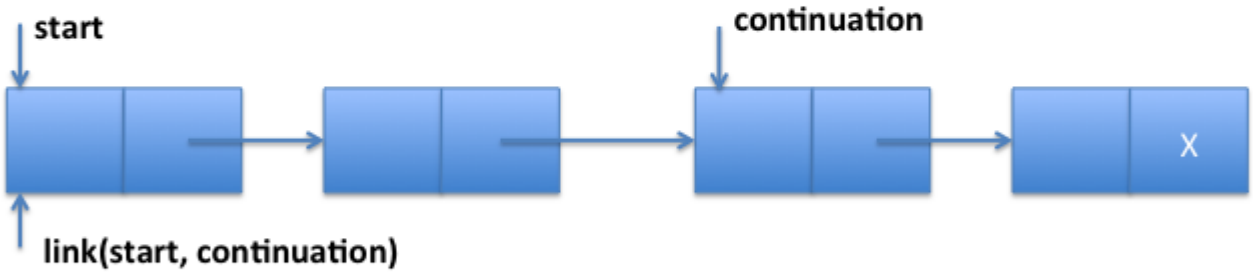
```
public final class TrainJourneyImmutable {

    private final int price;
    private final TrainJourneyImmutable onward;

    public TrainJourneyImmutable(int price, TrainJourney onward) {
        this.price = price;
        this.onward = onward;
    }

    public int getPrice() {
        return price;
    }

    public TrainJourneyImmutable getOnward() {
        return onward;
    }

    public TrainJourneyImmutable withPrice(int price) {
        return new TrainJourneyImmutable(price, getOnward());
    }

    public TrainJourneyImmutable withOnward(TrainJourneyImmutable onward) {
        return new TrainJourneyImmutable(getPrice(), onward);
    }
}
```

Scenario: be able to link together train journeys to form longer journeys.

Destructive link

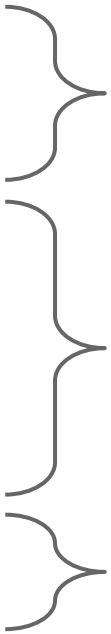
before

after



Mutable approach

```
public TrainJourney link(TrainJourney start, TrainJourney continuation) {
        if (start == null) {
        return continuation;
    }

    TrainJourney t = start;
    while (t.getOnward() != null) {
        t = t.getOnward();
    }

    t.setOnward(continuation);
    return start;
}
```



return the continuation journey if there is no start journey

find the last stop in the first journey

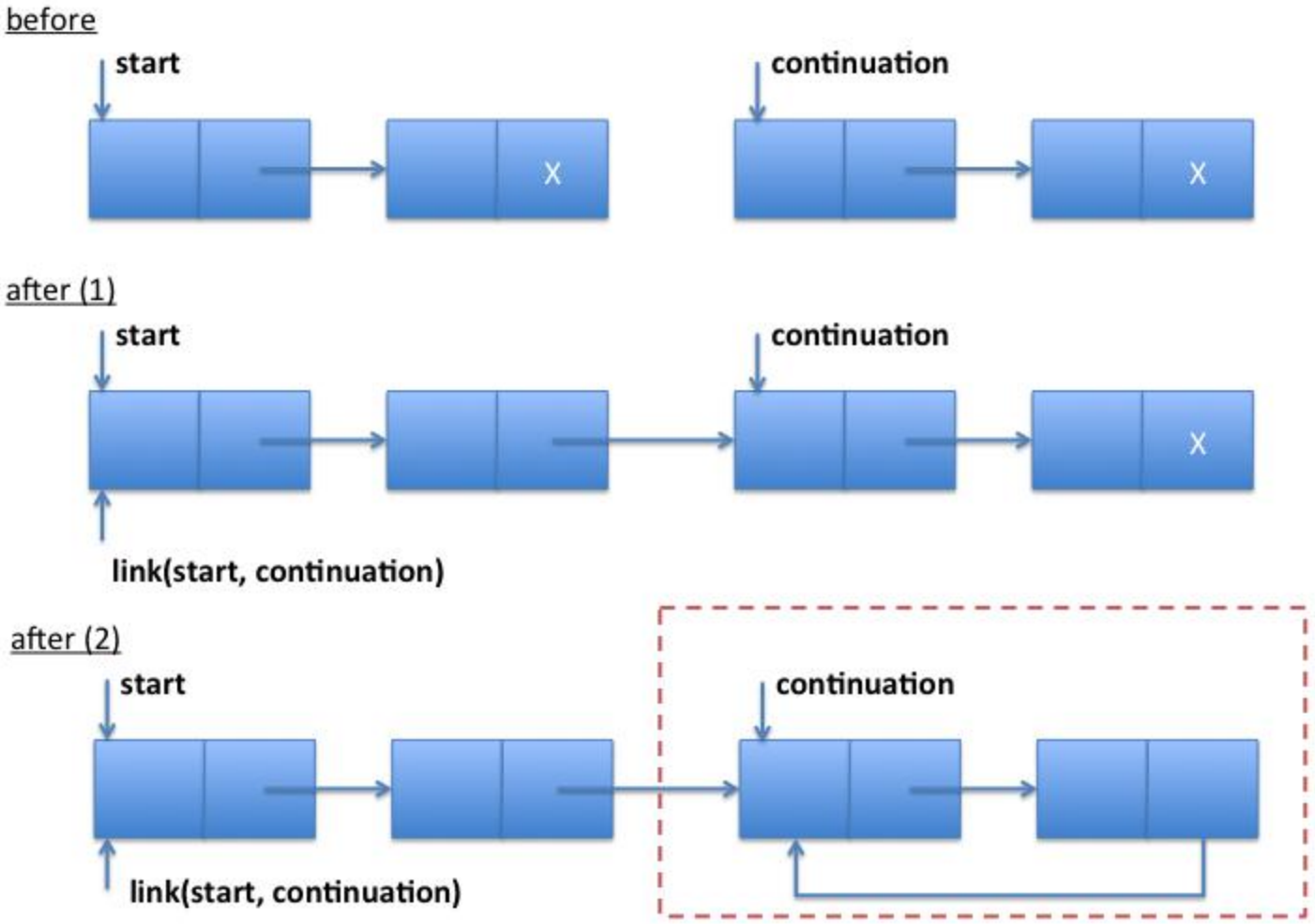
modify it to link to the continuation journey

Mutable approach: problem

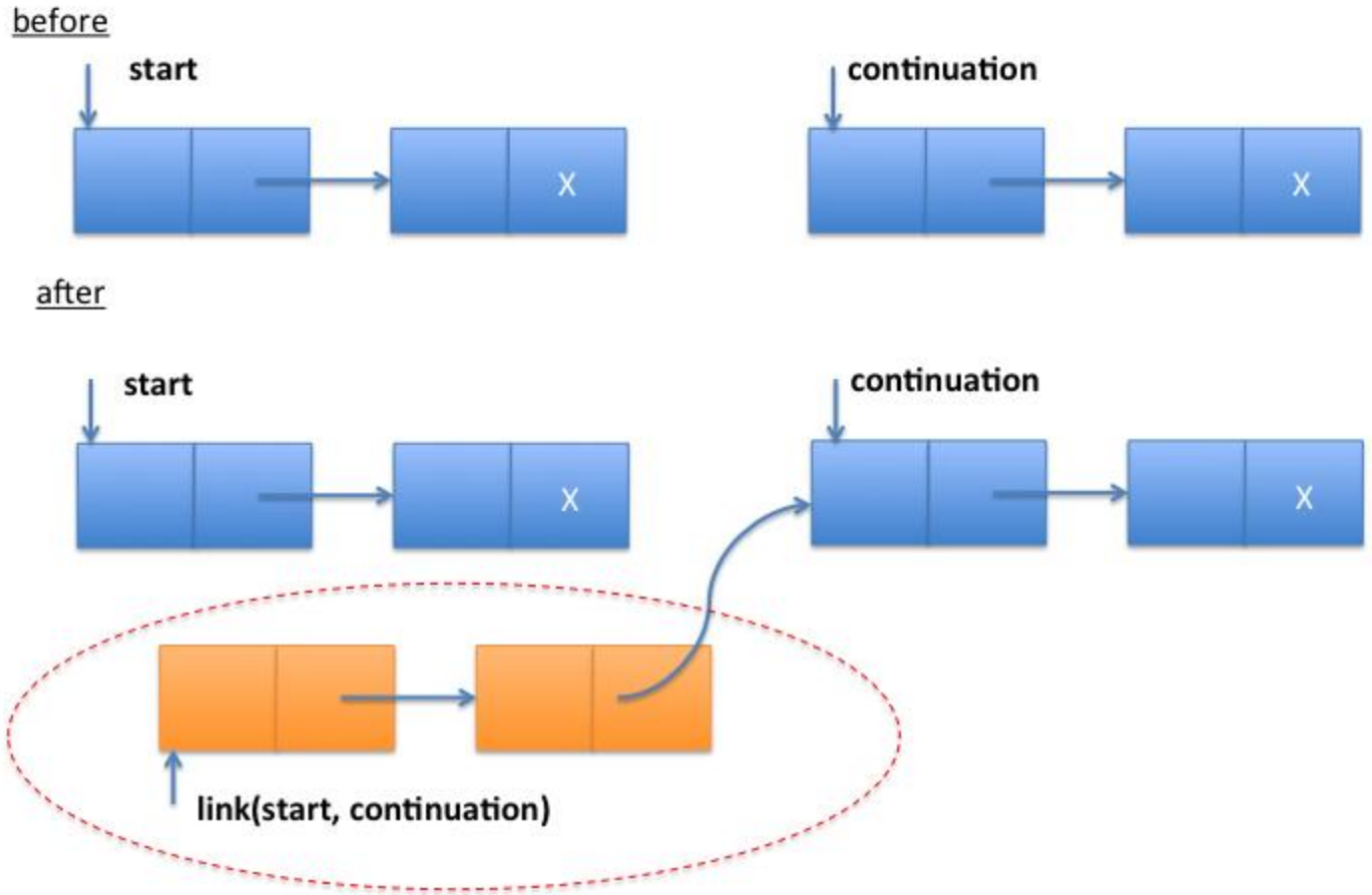
```
TrainJourney firstJourney = link(start, continuation);
TrainJourney secondJourney = link(start, continuation);

visit(secondJourney,
      tj -> { System.out.print(tj.price + " - "); });
```

```
static void visit(TrainJourney journey, Consumer<TrainJourney> c) {
    if (journey != null) {
        c.accept(journey);
        visit(journey.onward, c);
    }
}
```

java.lang.StackOverflowError



Functional-style append



Immutable approach

```
public TrainJourneyImmutable link(TrainJourneyImmutable start,
                                  TrainJourneyImmutable continuation) {
    return start == null ? continuation
         : new TrainJourneyImmutable(start.getPrice(),
                                     link(start.getOnward(), continuation));
}
```

Related topics

- Domain Driven Design
  - Value Classes are Immutable
- Core Java Improvements
  - New date & time library in Java 8 has many Immutable Objects
  - Current Value Types proposal is immutable
- Tooling
  - final keyword only bans reassignment
  - JSR 308 - improved annotation opportunities
  - Mutability Detector
  - Findbugs

Downsides to Immutability

- Increased memory allocation pressure
- Some problems harder to model with immutability
  - Deep object graphs
  - Example: Car simulation
- Library interactions
  - ORMs especially
  - Serialisation usually ok these days
- Alternative: small blobs of simple and localised state

Immutable objects reduce the scope for bugs.

First-class Functions

Currying

Immutability

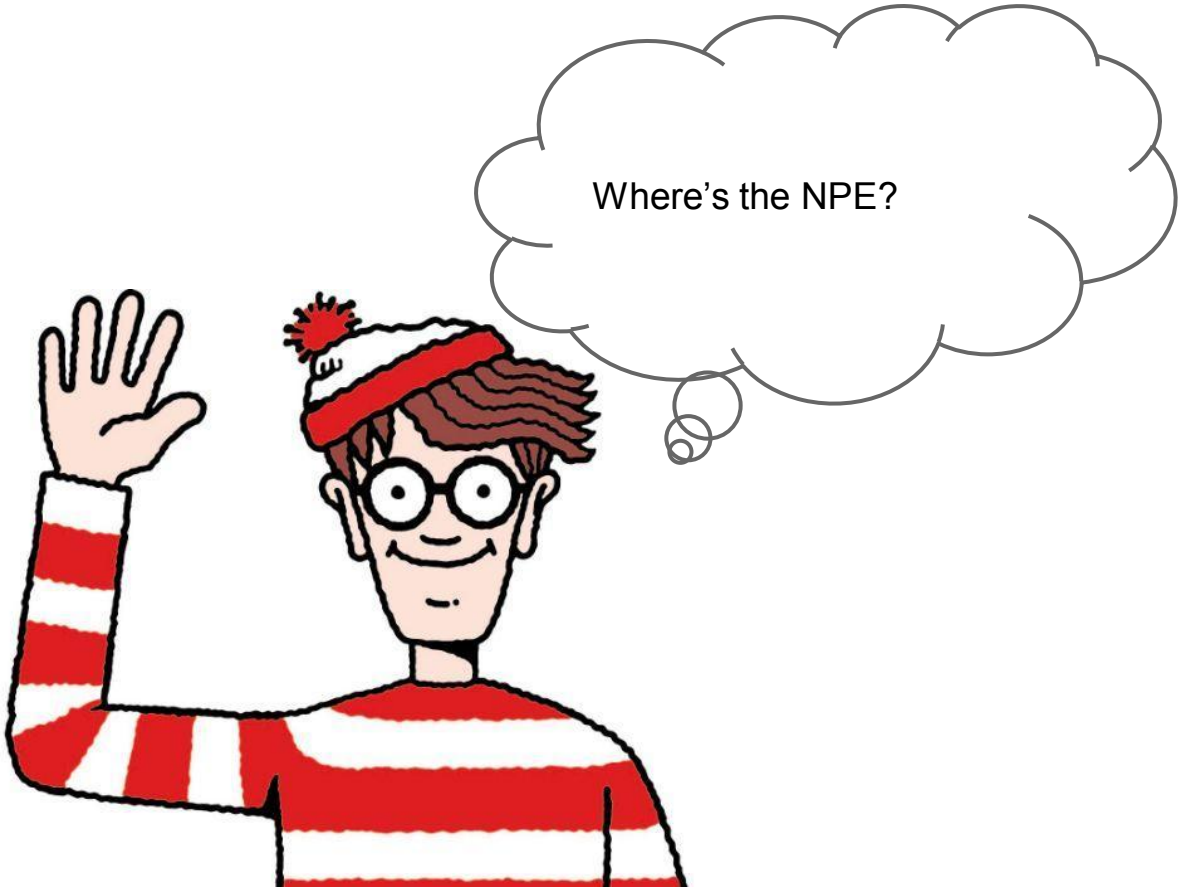
Optional Data Types

Conclusions

Don’t we all love it?

```
Exception in thread "main" java.lang.NullPointerException
```

```
public String getCarInsuranceName(Person person) {
      return person.getCar().getInsurance().getName();
}
```



Defensive checking

```
public String getCarInsuranceName(Person person) {
    if (person != null) {
        Car car = person.getCar();
        if (car != null) {
            Insurance insurance = car.getInsurance();
            if (insurance != null) {
                return insurance.getName();
            }
        }
    }
    return "No Insurance";
}
```

Optional

- Java 8 introduces a new class `java.util.Optional<T>`
  - a single-value container

- Explicit modelling
  - Immediately clear that its an optional value
  - better maintainability

- You need to actively unwrap an Optional
  - force users to think about the absence case
  - fewer errors

Updating model

```
public class Person {
    private Optional<Car> car;
    public Optional<Car> getCar() { return car; }
}

public class Car {
    private Optional<Insurance> insurance;
    public Optional<Insurance> getInsurance() { return insurance; }
}

public class Insurance {
    private String name;
    public String getName() { return name; }
}
```

Optional.map

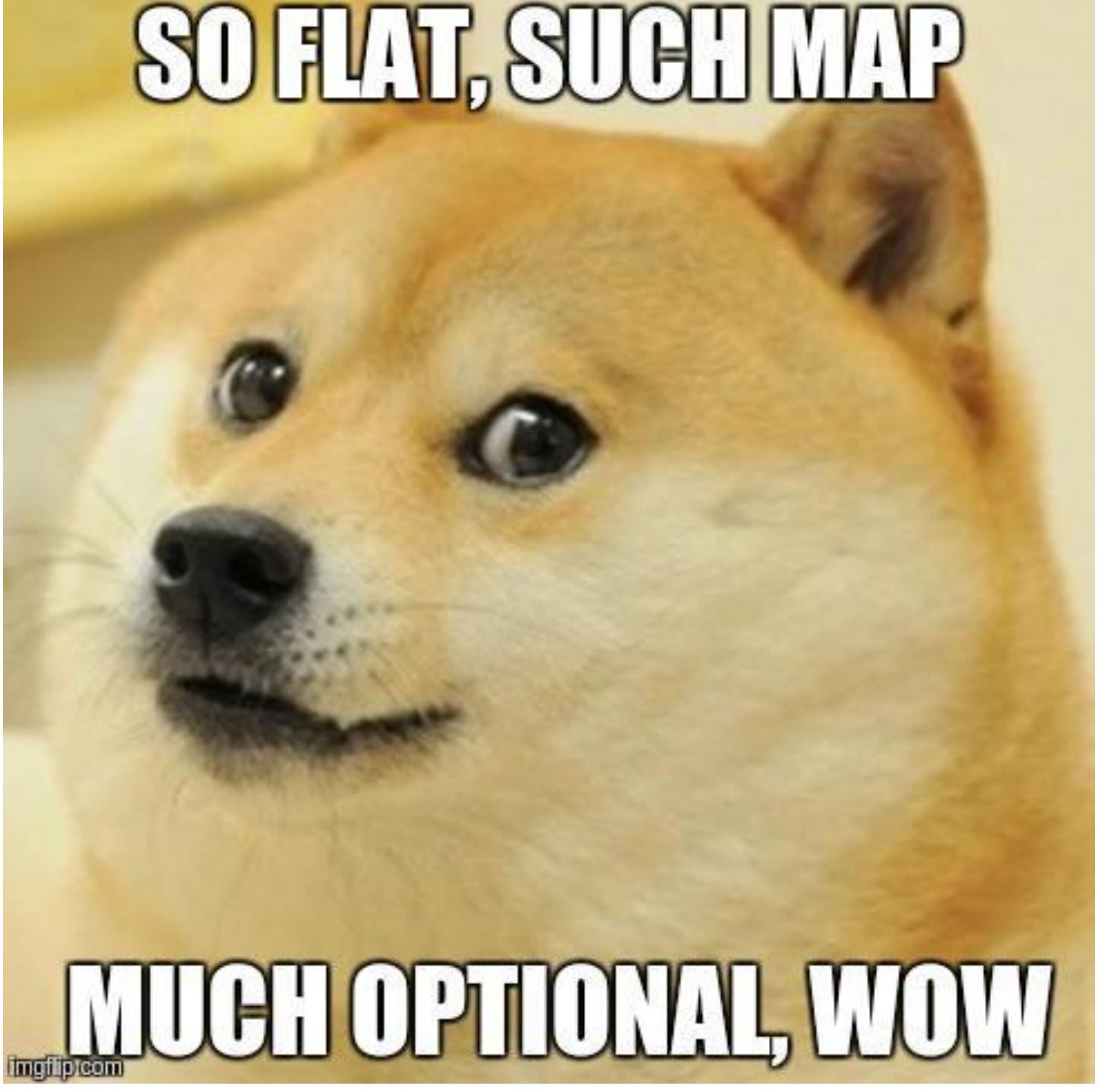
Why it doesn’t work?

```
Optional<People> optPerson = Optional.ofNullable(person);
Optional<String> name =
    optPeople.map(Person::getCar)
                .map(Car::getInsurance)
                .map(Insurance::getName);
```

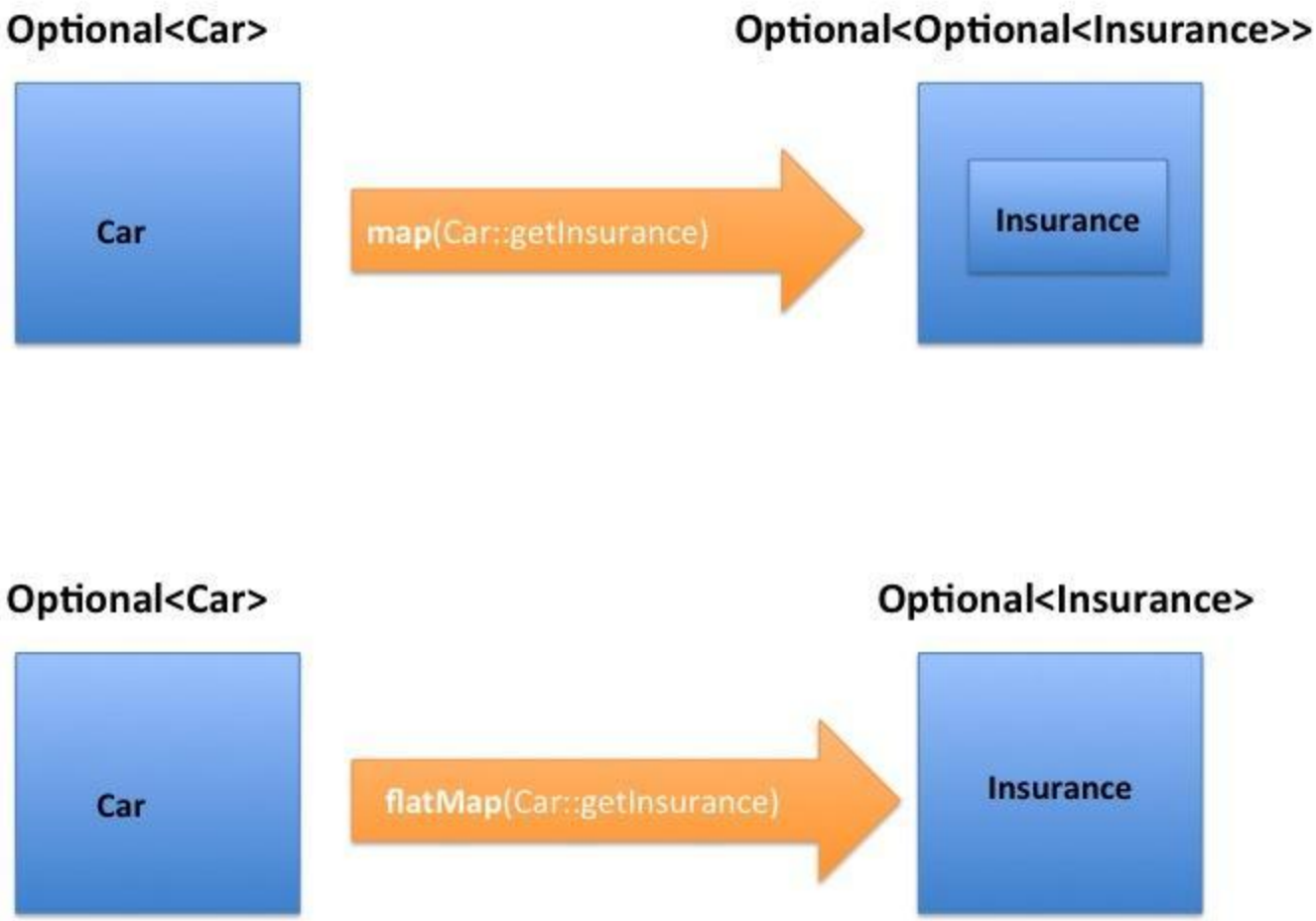
returns Optional<Optional<Car>>

Invalid, the inner Optional object doesn’t support the method getInsurance!

```
public class Person {
    private Optional<Car> car;
    public Optional<Car> getCar() { return car; }
}
```


SO FLAT, SUCH MAP
MUCH OPTIONAL, WOW
imgflip.com

Optional.flatMap



Chaining methods with flatMap

```
public String getCarInsuranceName(Person person) {
    return Optional.ofNullable(person)
                   .flatMap(Person::getCar)
                   .flatMap(Car::getInsurance)
                   .map(Insurance::getName)
                   .orElse("Unknown");
}
```

default value if the resulting Optional is empty

A comment on naming

```
public String getCarInsuranceName(Person person) {
    Set<Person> personSet = person == null
                          ? emptySet()
                          : singleton(person);

    return personSet.stream()
                    .flatMap(Person::getCar)
                    .flatMap(Car::getInsurance)
                    .map(Insurance::getName)
                    .orElse("Unknown");
}
```

Optional in Fields

- Should it be used for fields, or just public methods?

- Pros
  - Explicit modelling
  - Null-safe access
  - Simple getters

- Cons
  - More indirection and GC overhead in Java 8
  - Not every library understands Optional yet
  - Some libraries require Serializable fields

Consistent use of Optional replaces the use of null

First-class Functions

Currying

Immutability

Optional Data Types

Conclusions

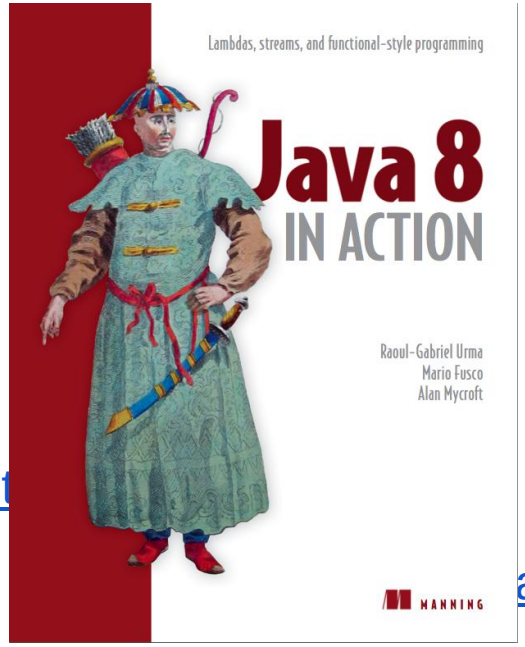
Summary of benefits

- **First-class functions** let you cope for requirement changes
- **Currying** lets you split the parameters of a function to re-use code logic
- **Immutability** reduces the scope for bugs
- **Optional data types** lets you reduce null checking boilerplate and prevent bugs

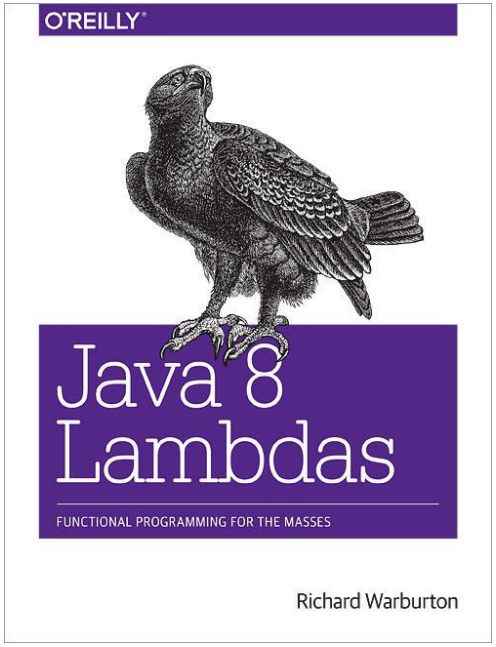
http://java8training.com

20th/21st April in Stockholm

http://www.jfokus.se/jfokus/training.jsp#java8



ht a8lambdas



Any Questions?

Richard Warburton
@richardwarburto

Raoul-Gabriel Urma
@raoulUK

Generalised curry function

```
// F -> (S -> R)
<F, S, R> Function<F, Function<S, R>>
curry(BiFunction<F, S, R> biFunction) {
    return f -> s -> biFunction.apply(f, s);
}
```