Optional

Raoul-Gabriel Urma
Richard Warburton

ITERATR LEARNING

Outline of module

1. The problem with null
2. How Optional improves upon null
3. How to use Optional in your code
4. Data Modelling Approaches
5. Practical: refactoring to remove nulls

The problem with null

NullPointerException

Raise your hand if you’ve come across this before:

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

Person/Car/Insurance data model

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

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

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

Car Insurance Name

What's possibly problematic with the following code?

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

Defensive checking (1)

```
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 "Unknown";
}
```

Defensive checking (2)

```
public String getCarInsuranceName(Person person) {
    if (person == null) {
        return "Unknown";
    }
    Car car = person.getCar();
    if (car == null) {
        return "Unknown";
    }
    Insurance insurance = car.getInsurance();
    if (insurance == null) {
        return "Unknown";
    }
    return insurance.getName();
}
```

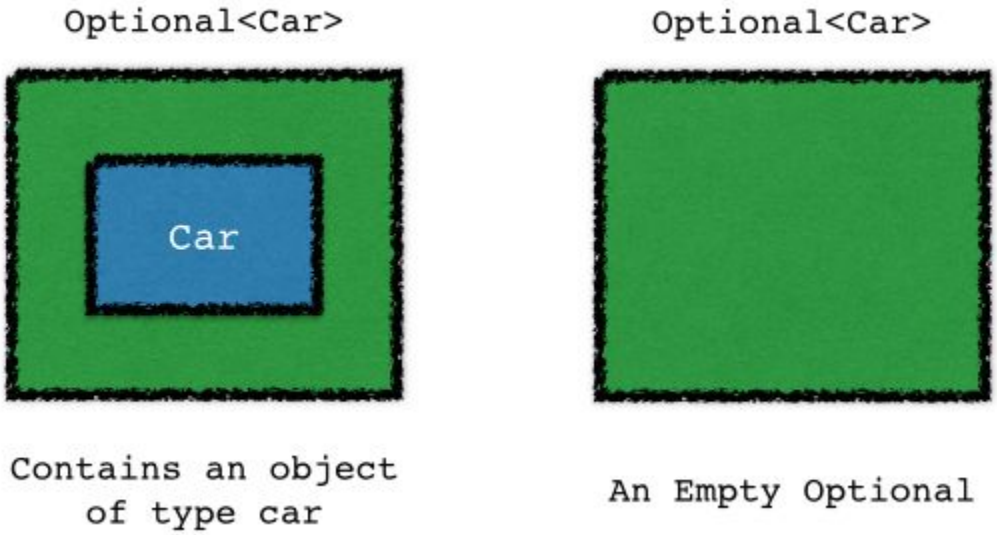
Problems with null

1. Error-prone checking
2. Verbose checking
3. No useful semantic meaning

How Optional improves upon null

Optional in a nutshell

- Java 8 introduces a new class **java.util.Optional<T>**
- Optional encapsulates an optional value
- You can view Optional as a single-value container that either contains a value or doesn't



Updating our 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; }
}
```

Benefits

- More comprehensible model where it’s immediately understandable whether to expect an optional value
  - better maintainability

- You need to actively unwrap an Optional to deal with the absence of a value
  - fewer errors

How to use Optional in your code

Creating Optional objects

```
Optional<Car> optCar = Optional.empty();

Optional<Car> optCar = Optional.of(car);

Optional<Car> optCar = Optional.ofNullable(car);
```

Do something if a value is present (1)

Before

```
if(insurance != null) {
  System.out.println(insurance.getName());
}
```

After

```
Optional<Insurance> optInsurance = car.getInsurance();

optInsurance.ifPresent(insurance ->
      System.out.println(insurance.getName()));
```

Do something if a value is present (2)

```
if(optInsurance.isPresent()) {
  System.out.println(optInsurance.get());
}
```

- **get** throws a NoSuchElementException if no value contained in the Optional object (null doesn't propagate)

- Combining **isPresent** and **get** is not recommended
  - nested checks
  - have to work with exceptions to handle default values/actions

- We explore more idiomatic alternatives using map and flatMap soon

Default values or actions

```
Stream<Player> players = Stream.of(ronaldo, messi,
    rooney);
Optional<Player> optFirstPenalty =
     players.filter(p -> p.getConfidence() > 90)
            .findAny();
```

Default value

```
Player p = optFirstPenalty.orElse(terry);
```

Default action

```
Player p = optFirstPenalty.orElseThrow(
              SurrenderGameException::new);
```

Extracting values from Optionals with map

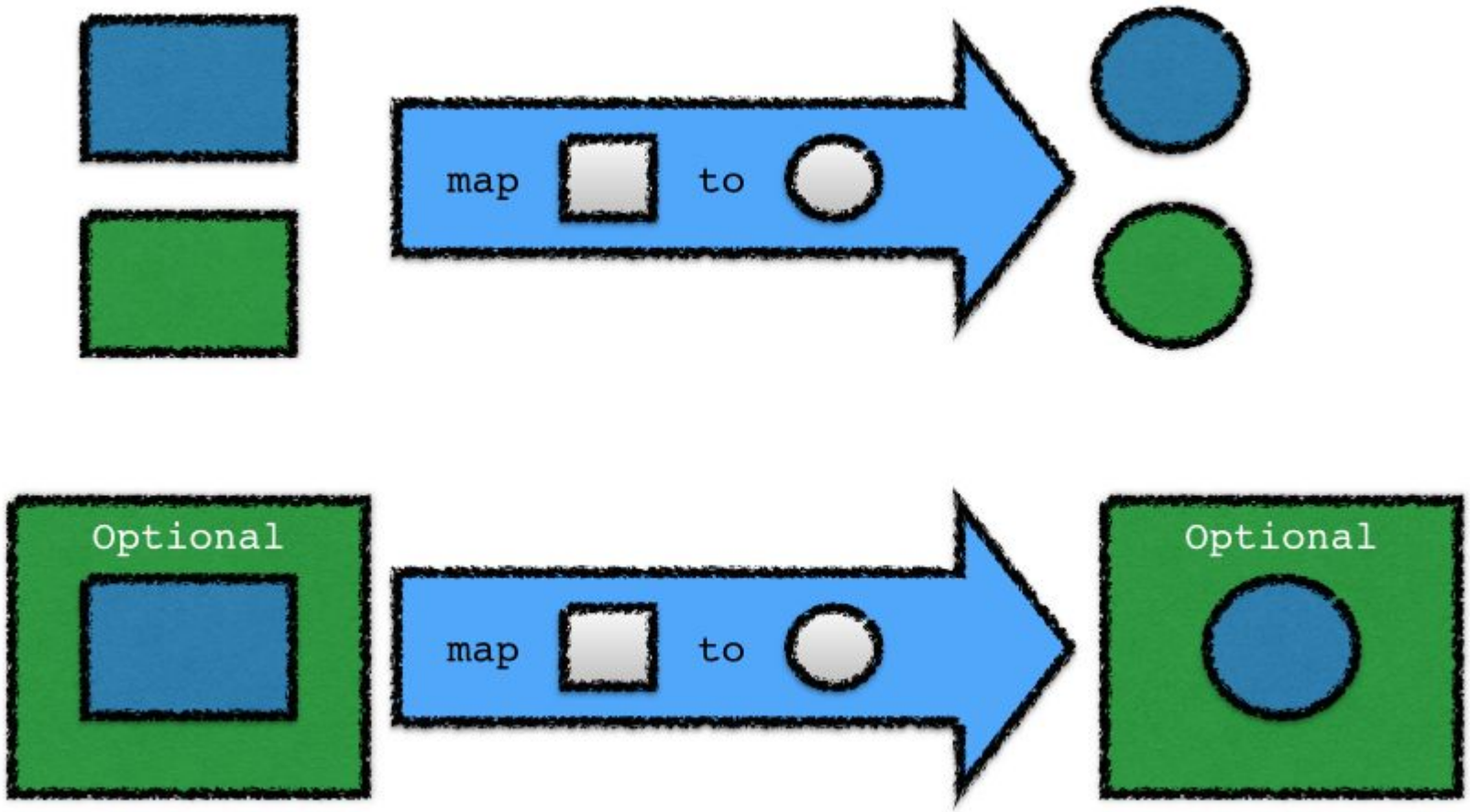
Before:

```
String name = null;
if(insurance != null) {
    name = insurance.getName();
}
```

After:

```
Optional<Insurance> optInsurance =
          Optional.ofNullable(insurance);
Optional<String> name =
          optInsurance.map(Insurance::getName);
```

Understanding map



Chaining methods

How can we rewrite the following in a safe way?

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

First try

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

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!

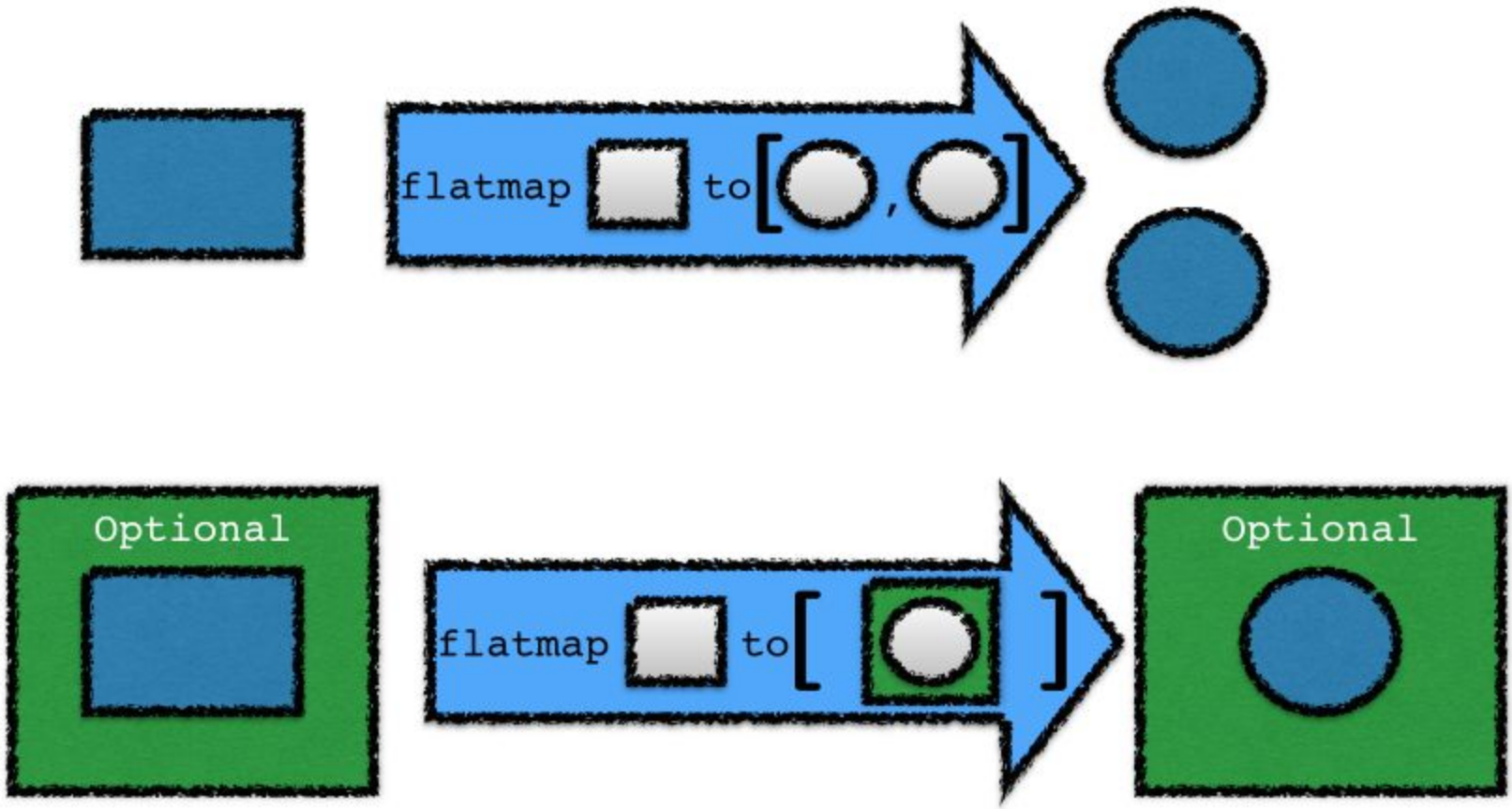
```
public class Person {
   private Optional<Car> car;
   public Optional<Car> getCar() { return car; }
}
```

Chaining methods with flatMap

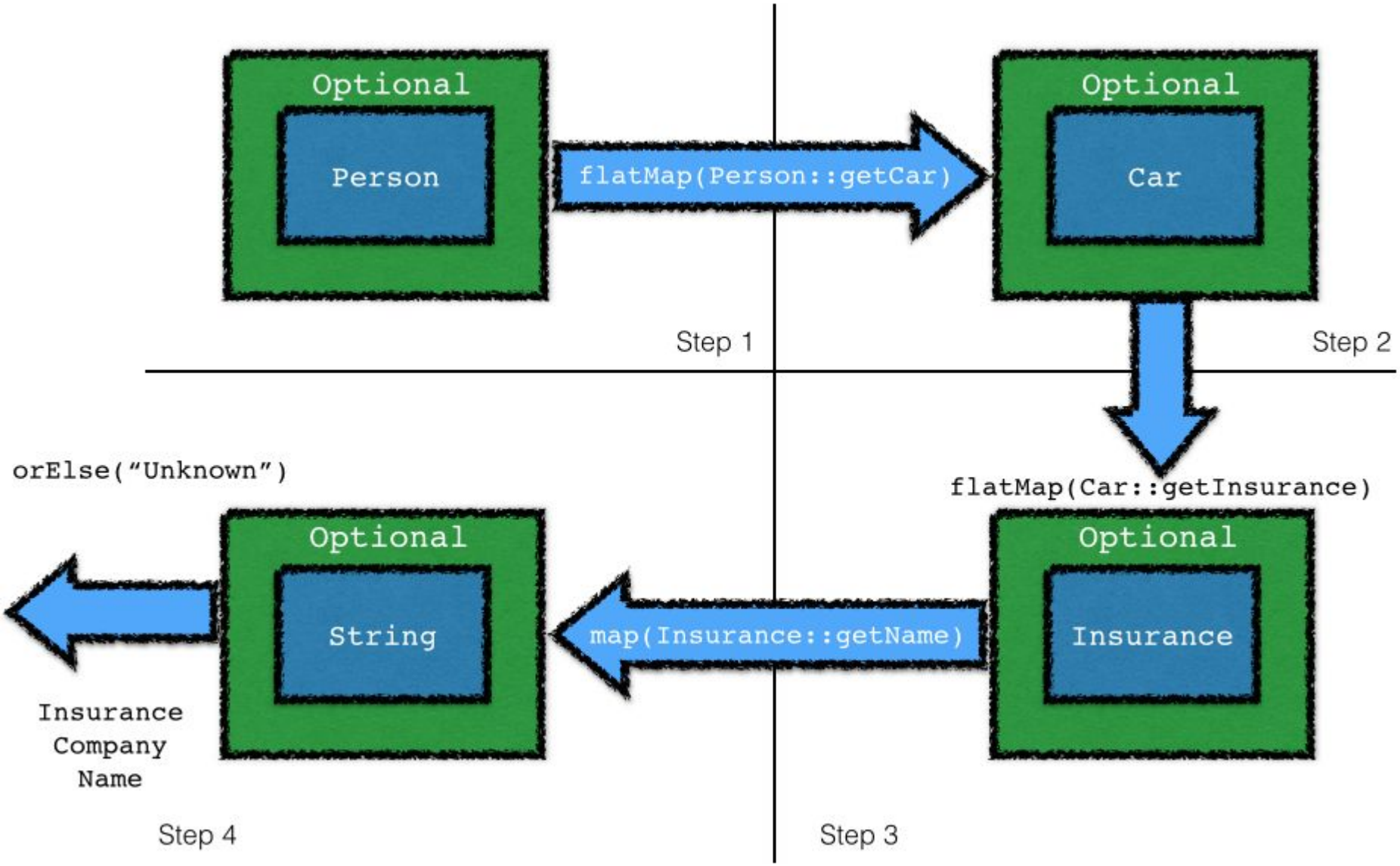
```
public String getCarInsuranceName(Person person) {
    return Optional.ofNullable(person)
                   .flatMap(Person::getCar)
                   .flatMap(Car::getInsurance)
                   .map(Insurance::getName)
                   .orElse("Unknown");
}
```

default value if the resulting Optional is empty

Understanding flatMap (1)



Understanding flatMap (2)



Rejecting values with `filter`

Before

```
Insurance ins = car.getInsurance();
if(ins != null && "Sport_Insurance".equals(ins.getName())){
  System.out.println("This is an expensive insurance!");
}
```

After

```
optInsurance.filter(ins ->
        "Sport_Insurance".equals(ins.getName()))
            .ifPresent(ins ->
    System.out.println("This is an expensive insurance!"))
```

Java 9 - `Optional::stream`

```
Optional<Setting> lookupSettingByName(final String name)
```

Before

```
String homeDir = Stream.of("appHome", "user.home")
          .map(Setting::lookupSettingByName)
          .filter(Optional::isPresent)
          .map(Optional::get)
          .findFirst().get();
```

After

```
String homeDir = Stream.of("appHome", "user.home")
          .map(Setting::lookupSettingByName)
          .flatMap(Optional::stream)
          .findFirst().get();
```

Java 9 - `Optional::ifPresentOrElse`

```
Optional<Booking> lookupBooking(final String reference)
```

Before

```
Optional<Booking> booking = lookupBooking(reference);
if (booking.isPresent()) {
    Ui.displayCheckIn(booking.get());
} else {
    Ui.displayMissingBookingPage();
}
```

After

```
lookupBooking(reference)
    .ifPresentOrElse(
      Ui::displayCheckIn,
      Ui::displayMissingBookingPage);
```

Java 9 - `Optional::or`

```
Optional<Client> findClient(String id)
Optional<Client> lookupCompanyDetails(String id)
```

Before

```
Optional<Client> client = findClient(clientId);
if (!client.isPresent()) {
    client = lookupCompanyDetails(clientId);
}
```

After

```
Optional<Client> client
    = findClient(clientId)
        .or(() -> lookupCompanyDetails(clientId));
```

Unit testing with Optional

```
assertEquals(Optional.of(ferrari488), raoul.getCar());

assertEquals(Optional.empty(), richard.getCar());
```

Also: https://github.com/npathai/hamcrest-optional

```
assertThat(optional, hasValue(startsWith("CAMB")));
```

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

Exercise

How would you rewrite the following code using an Optional object?

`com.java_8_training.problems.optional.RefactorToOptional1`

```
public int readDuration(Properties props, String name) {
   String value = props.getProperty(name);
   if (value != null) {
       int i = Integer.parseInt(value);
       if (i > 0) {
          return i;
       }
   }
   return 0;
}
```

make sure a property exists with the required name

try to convert the String property to a number

check the number is positive

return 0 if conversion failed

The End

Data Modelling Approaches

Updating our 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; }
}
```

Left to right

- Pros
  - Conceptually simple
  - How to find insurance given an owner?
- Cons
  - Coupling of caller to a method call chain
  - How to find owner given a car?
  - How to find car given the insurance?

Updating our model

```
public class Insurance {
  private String name;
  private Car car;
  public String getName() { return name; }
  public Car getCar() { return car; }
}
public class Car {
  private Person owner;
  public Person getOwner() {
    return owner;
  }
}
public class Person {

}
```

Inverse

- Pros
  - Conceptually simple
  - No optional fields
  - How to find owner given a car?
  - How to find car given an insurance?
- Cons
  - How to find insurance given a car?
  - How to find insurance given an owner?

Updating our model

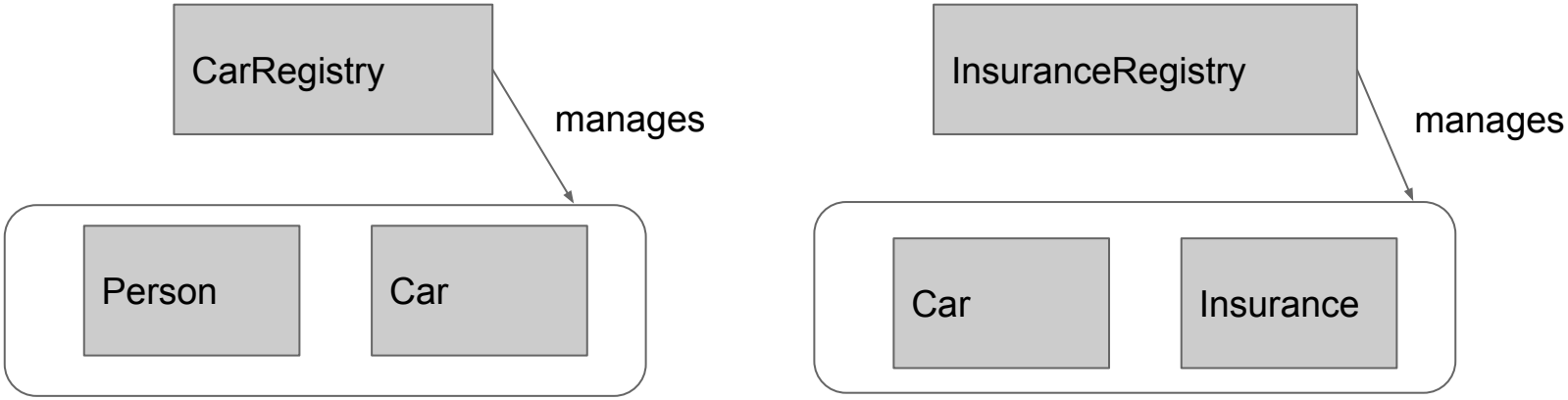
```
public class Car {
  private Person owner;
  private Optional<Insurance> insurance;
  public Person getOwner() { return owner; }
  public Optional<Insurance> getInsurance() {
  return insurance;
  }
}
public class Person { }

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

Split

- Pros
  - Conceptually simple
  - How to find owner given a car?
  - How to find insurance given a car?
- Cons
  - How to find insurance given an owner?
  - How to find car given an insurance?
  - How to find owner given an insurance?

Relationship managers



- Pros
  - Decouples Person, Car, Insurance
  - Extensible to many to many relationships
- Cons
  - More set up

Updating our model

```
public class CarRegistry {

    private Map<Person, Car> registry = new HashMap<>();

    public void addCarToRegistry(Person owner, Car car) {
        registry.put(owner, car);
    }

    public Optional<Car> findCarForPerson(Person owner) {
        return Optional.ofNullable(registry.get(owner));
    }
}
```

Updating our model

```
public String getCarInsuranceName(Person person) {
   return carRegistry.findCarForPerson(person)
                     .flatMap(insuranceRegistry::fetchInsuranceForCar)
                     .map(Insurance::getName)
                     .orElse("Unknown");
}
```

vs (previous)

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