Lecture 20

Optional

Using `Optional` as a better alternative to `null`

SOEN 6441, Summer 2018

René Witte
Department of Computer Science
and Software Engineering
Concordia University

Outline

`java.lang.NullPointerException`

null

History

Introduced by Tony Hoare in 1965 for the ALGOL W programming language.

Hoare (2009)

> *"I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years."*

Modeling the absence of a value

A 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; }
}
```

Using the model classes

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

Reducing `NullPointerExceptions` with defensive checking

```java
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";
}
```

Null-safe Attempt #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`

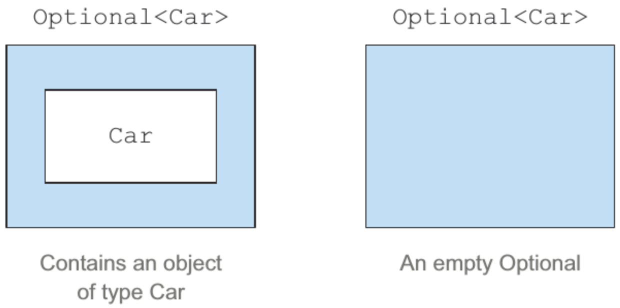
Theoretical and Practical Problems

- It's a source of error.
  `NullPointerException` is by far the most common exception in Java.
- It bloats your code.
  It worsens readability by making it necessary to fill your code with often deeply nested `null` checks.
- It's meaningless.
  It doesn't have any semantic meaning, and in particular it represents the wrong way to model the absence of a value in a statically typed language.
- It breaks Java philosophy.
  Java always hides pointers from developers except in one case: the `null` pointer.
- It creates a hole in the type system.
  `null` carries no type or other information, meaning it can be assigned to any reference type. This is a problem because, when it's propagated to another part of the system, you have no idea what that `null` was initially supposed to be.

The Optional class

`java.util.Optional<T>`

- Inspired by Haskell (`Maybe`) and Scala (`Option[t]`)
- Encapsulates an optional value
- Must explicitly check for presence or absence using the methods provided by `Optional`
- Checking enforced by the type system – you can't forget to do it!
- APIs can now make it explicit if methods can accept or return missing values



Designing with `Optional`

Redefining the `Person/Car/Insurance` data model using `Optional`

```
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; }
}
```

Note

A `Person` might or might not have a car

- Thus, we no longer declare a field `Car` and simply set it `null` when missing
- Rather, we explicity model it as `Optional<Car>`

Outline

René Witte

Université Concordia University

Creating `Optional` objects

Empty `Optional`

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

`Optional` from a non-`null` value

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

`Optional` from (potential) `null`

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

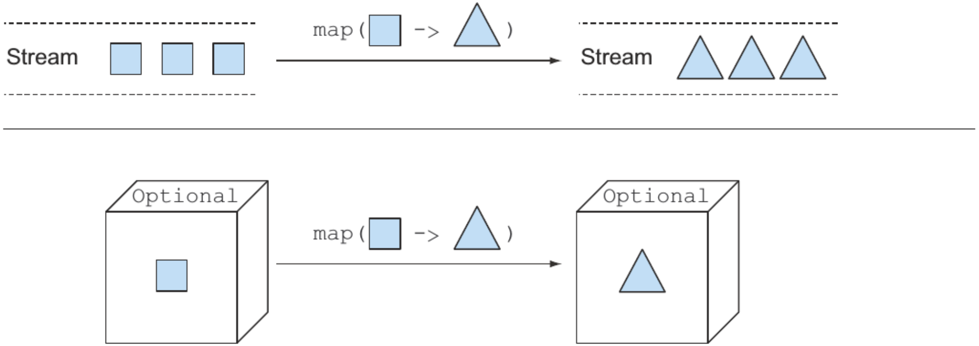
Extracting and transforming values from optionals with map

Extracting values from an object

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

Optional with map

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



René Witte
Concordia University

Chaining `Optional` objects

Goal: Safe chaining of calls like:

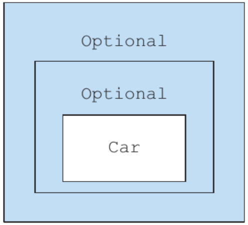
```
return person.getCar().getInsurance().getName();
```

Using `map`?

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

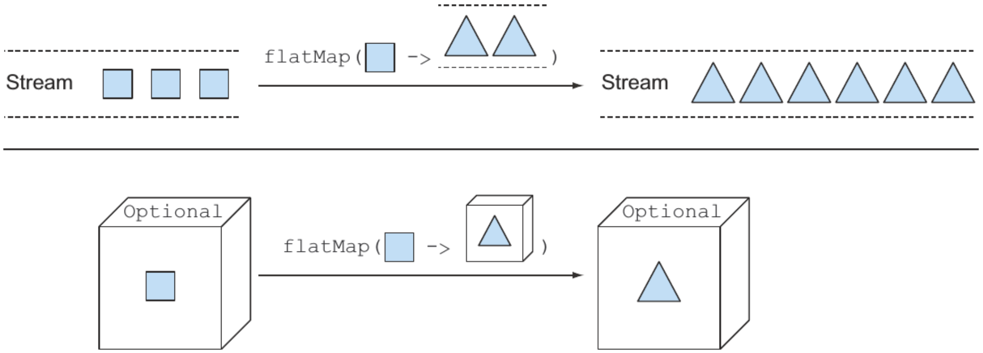
Does not compile

`getCar` returns `Optional<Car>`, so after `map` we have an `Optional<Optional<Car>>`



René Witte
Concordia University

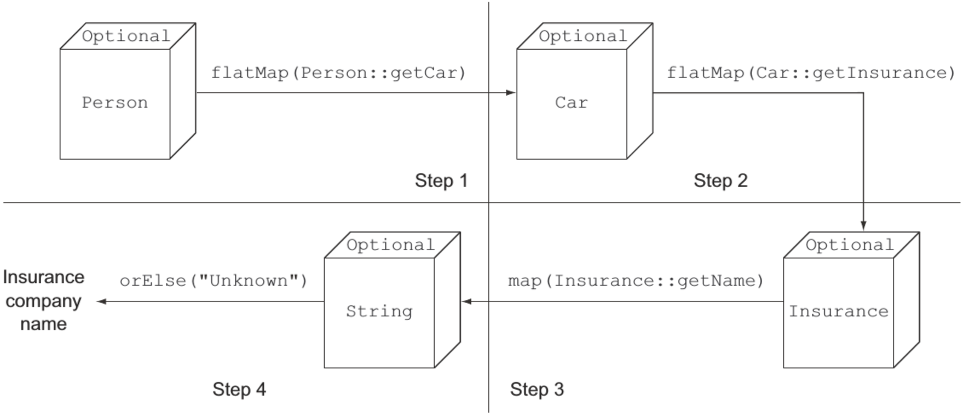
Using `flatMap`



Finding a car's insurance company name with `Optionals`

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

The `Person/Car/Insurance` dereferencing chain using optionals



API Design with `Optional`

René Witte

Concordia University

Note

`Optional` fields are not serializable

Consequences

- Cannot use `Optional` fields for classes that have to be serializable
- Design for optional-return

Example

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

Default actions and unwrapping an optional

Reading the value from an `Optional`

- `get()`
  returns wrapped value if present, otherwise throws a NoSuchElementException
- `orElse(T other)`
  provides the value if present, otherwise a default value
- `orElseGet(Supplier<? extends T> other)`
  lazy counterpart to `orElse`
- `orElseThrow(Supplier<? extends X> exceptionSupplier)`
  similar to `get`, but with custom exception
- `ifPresent(Consumer<? super T> consumer)`
  execute action if value is present (no action otherwise)

Combining two optionals

Dealing with two optionals in one method

```
public Insurance findCheapestInsurance(Person person, Car car) {
  // queries services provided by the different insurance companies
  // compare all those data
  return cheapestCompany;
}
```

First attempt

```
public Optional<Insurance> nullSafeFindCheapestInsurance(
  Optional<Person> person, Optional<Car> car) {
    if (person.isPresent() && car.isPresent()) {
      return Optional.of(findCheapestInsurance(person.get(),car.get()));
    } else {
      return Optional.empty();
    }
}
```

Better solution

```
return person.flatMap(p -> car.map(c -> findCheapestInsurance(p, c)));
```

Rejecting certain values with filter

Task: check value of an object

```
Insurance insurance = ...;
if(insurance != null && "CI␣Inc".equals(insurance.getName())){
  System.out.println("ok");
}
```

Using filter

```
Optional<Insurance> optInsurance = ...;
optInsurance.filter(insurance -> "CI␣Inc".equals(insurance.getName()))
            .ifPresent(x -> System.out.println("ok"));
```

Outline

Wrapping a potentially `null` value in an optional

Dealing with existing Java APIs

E.g., given a `Map<String, Object>`, calling

```
Object value = map.get("key");
```

returns `null` if key is not in map.

Wrapping in `Optional`

```
Optional<Object> value = Optional.ofNullable(map.get("key"));
```

Exception vs. Optional

Dealing with exceptions from existing APIs

E.g., converting `String` to `int` using `Integer.parseInt(String)` can throw a `NumberFormatException`.

Converting a `String` into an `Integer` returning an `Optional`

```
public static Optional<Integer> stringToInt(String s) {
  try {
    return Optional.of(Integer.parseInt(s));
  } catch (NumberFormatException e) {
    return Optional.empty();
  }
}
```

Summary

Optionals

- `null` references have been historically introduced in programming languages to generally signal the absence of a value.
- Java 8 introduces the class `java.util.Optional<T>` to model the presence or absence of a value.
- Create `Optional` objects with the static factory methods `Optional.empty`, `Optional.of`, and `Optional.ofNullable`.
- The `Optional` class supports many methods such as `map`, `flatMap`, and `filter`, similar to the methods of a stream.
- Using `Optional` forces you to actively unwrap an optional to deal with the absence of a value; as a result, you protect your code against unintended `null` pointer exceptions.
- Using `Optional` can help you design better APIs in which, just by reading the signature of a method, users can tell whether to expect an optional value.

Outline

Reading Material

Required

- [UFM14, Chapter 10] (Using Optional as a better alternative to null)

References

[UFM14] Raoul-Gabriel Urma, Mario Fusco, and Alan Mycroft.
Java 8 in Action: Lambdas, streams, and functional-style programming.
Manning Publications, 2014.
https://www.manning.com/books/java-8-in-action.