Part 4 Everyday Java

11 Using Optional as a better alternative to null

This chapter covers

- What's wrong with null references and why you should avoid them
- From null to Optional: rewriting your domain model in a null-safe way
- Putting optionals to work: removing null checks from your code
- Different ways to read the value possibly contained in an optional
- Rethinking programming given potentially missing values

11.1 How do you model the absence of a value?

- `NullPointerException`s are a pain for any Java developer, novice, or expert.
- British computer scientist Tony Hoare introduced `null` references back in 1965 while designing ALGOL W programming language. After many years, he regretted this decision, calling it
  "my billion-dollar mistake."

Imagine that you have the following nested object structure for a person who owns a car and has car insurance in the following listing.

Listing 11.1 The `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; }
}
```

What's problematic with the following code?

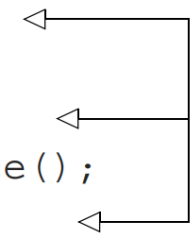
```
public String getCarInsuranceName(Person person) {
    return person.getCar().getInsurance().getName();
}
```

Reducing NullPointerExceptions with defensive checking

A first attempt to write a method preventing a `NullPointerException` is shown in the following listing.

Listing 11.2 `Null`-safe attempt 1: deep doubts

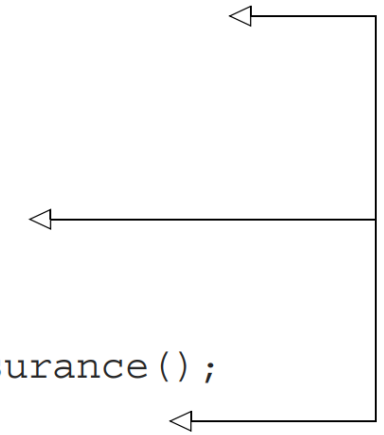
```
public String getCarInsuranceName(Person person) {
    if (person != null) {
        Car car = person.getCar();
        if (car != null) {
            Insurance insurance = car.getInsurance();
            if (insurance != null) {
                return insurance.getName();
            }
        }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



Each null check increases the nesting level of the remaining part of the invocation chain.

Listing 11.3 `Null`-safe attempt 2: too many exits

```
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();
}
```



Each null check adds a further exit point.

Problems with null

- To recap our discussion so far, the use of `null` references in Java causes both theoretical and practical problems:
  - *It's a source of error*.
  - *It bloats your code*.
  - *It's meaningless*.
  - It *breaks Java philosophy*.
  - *It creates a hole in the type system*.

What are the alternatives to null in other languages?

- Groovy
  - a *safe navigation operator*, represented by `?.`, to safely navigate potentially `null` values.
    - It returns a null in the event that any value in the chain is a null.
- Haskell
  - It includes a `Maybe` type, which essentially encapsulates an optional value. A value of type `Maybe` can contain a value of a given type or nothing.
- Scala
  - It has a similar construct called `Option[T]` to encapsulate the presence or absence of a value of type `T`.
- Java 8 takes inspiration from this idea of an optional value by introducing a new class called `java.util.Optional<T>`!

11.2 Introducing the Optional class

- Java 8 introduces a new class called `java.util.Optional<T>` that's inspired by Haskell and Scala.
  - When a value is present, the `Optional` class wraps it. Conversely, the absence of a value is modeled with an empty optional returned by the method `Optional.empty`.
  - Trying to dereference a `null` invariably causes a `NullPointerException`, whereas `Optional.empty()` is a valid, workable object of type `Optional` that can be invoked in useful ways.

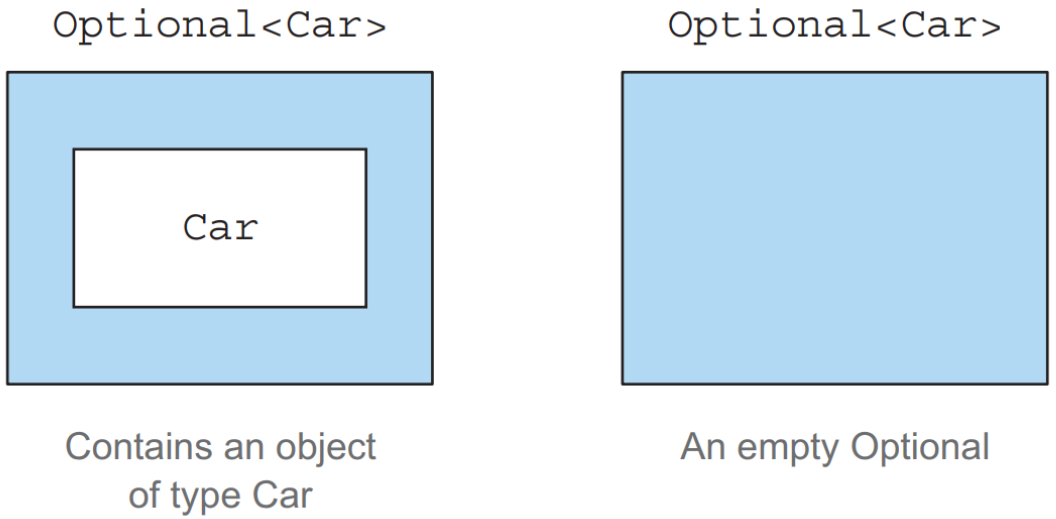


Figure 11.1 An optional `Car`

- An important, practical semantic difference in using `Optionals` instead of `nulls` is that in the first case, declaring a variable of type `Optional<Car>` instead of `Car` clearly signals that a missing value is permitted there.

With this act in mind, you can rework the original model from listing 11.1, using the `Optional` class as shown in the following listing.

Listing 11.4 Redefining the `Person/Car/Insurance` data model by 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; }
}
```

A person may not own a car, so you declare this field Optional.

A car may not be insured, so you declare this field Optional.

An insurance company must have a name.

- Consistently using `Optional` values creates a clear distinction between a missing value that's planned for and a value that's absent only because of a bug in your algorithm or a problem in your data.
- It's important to note that the intention of the `Optional` class isn't to replace every single `null` reference. Instead, its purpose is to help you design more-comprehensible APIs so that by reading the signature of a method, you can tell whether to expect an optional value.

11.3 Patterns for adopting Optionals

- Creating Optional objects
  - Empty optional
    - `Optional.empty`:
      ```
      Optional<Car> optCar = Optional.empty();
      ```
  - Optional from a non-null value
    - `Optional.of`:
      ```
      Optional<Car> optCar = Optional.of(car);
      ```
    - If car were null, a `NullPointerException` would be thrown immediately.
  - Optional from null
    - `Optional.ofNullable`:
      ```
      Optional<Car> optCar = Optional.ofNullable(car);
      ```
    - If `car` were `null`, the resulting `Optional` object would be empty.

Extracting and transforming values from Optionals with map

-
```
String name = null;
if(insurance != null){
name = insurance.getName();
}
```
- `map` method:

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

If the `Optional` is empty, nothing happens.

- You could also think of an `Optional` object as being a particular collection of data, containing at most a single element.

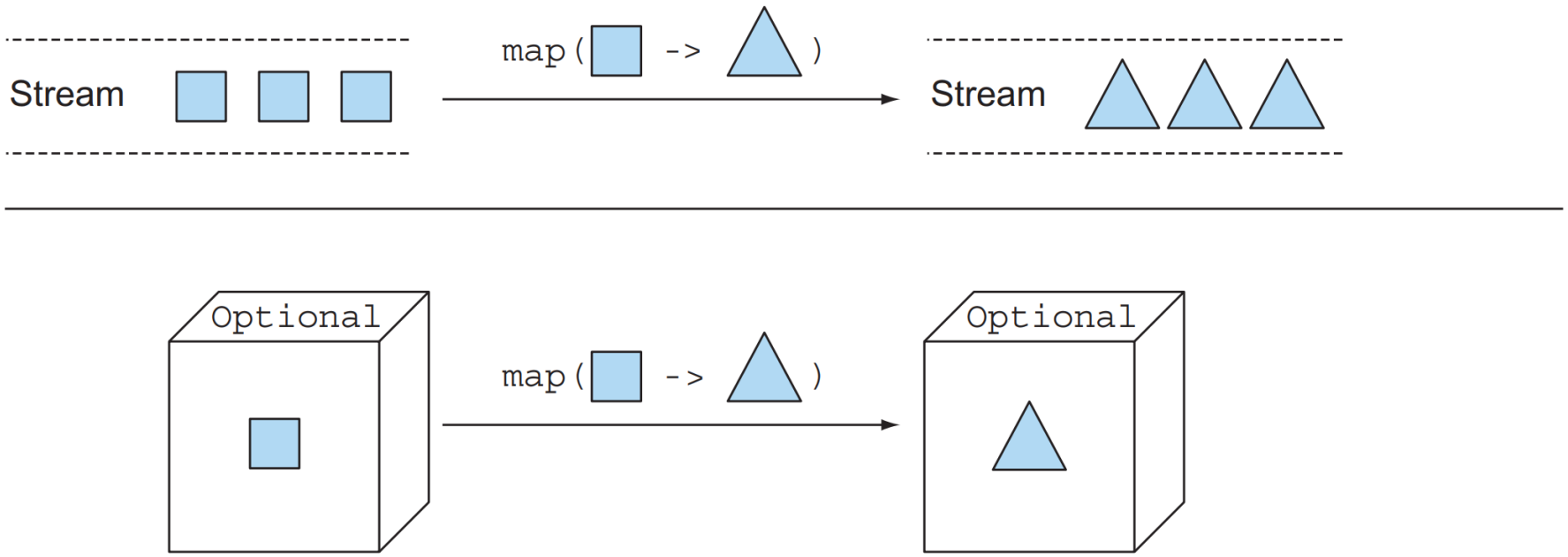


Figure 11.2 Comparing the `map` methods of `Streams` and `Optionals`

This idea looks useful, but how can you use it to rewrite the code in listing 11.1,

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

Chaining Optional objects with flatMap

Because you've learned how to use `map`, your first reaction may be to use `map` to rewrite the code as follows:

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

Unfortunately, this code doesn't compile. Why? The variable `optPerson` is of type `Optional<Person>`, so it's perfectly fine to call the `map` method. But `getCar` returns an object of type `Optional<Car>` (as presented in listing 11.4), which means that the result of the `map` operation is an object of type `Optional<Optional<Car>>`. As a result, the call to `getInsurance` is invalid because the outermost optional contains as its value another optional, which of course doesn't support the `getInsurance` method. Figure 11.3 illustrates the nested optional structure you'd get.

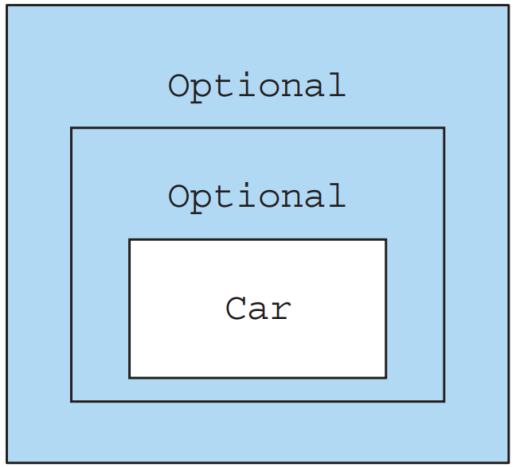


Figure 11.3 A two-level optional

- `flatMap` method:
  - `flatMap` method flattens this two-level optional into a single optional containing a triangle.

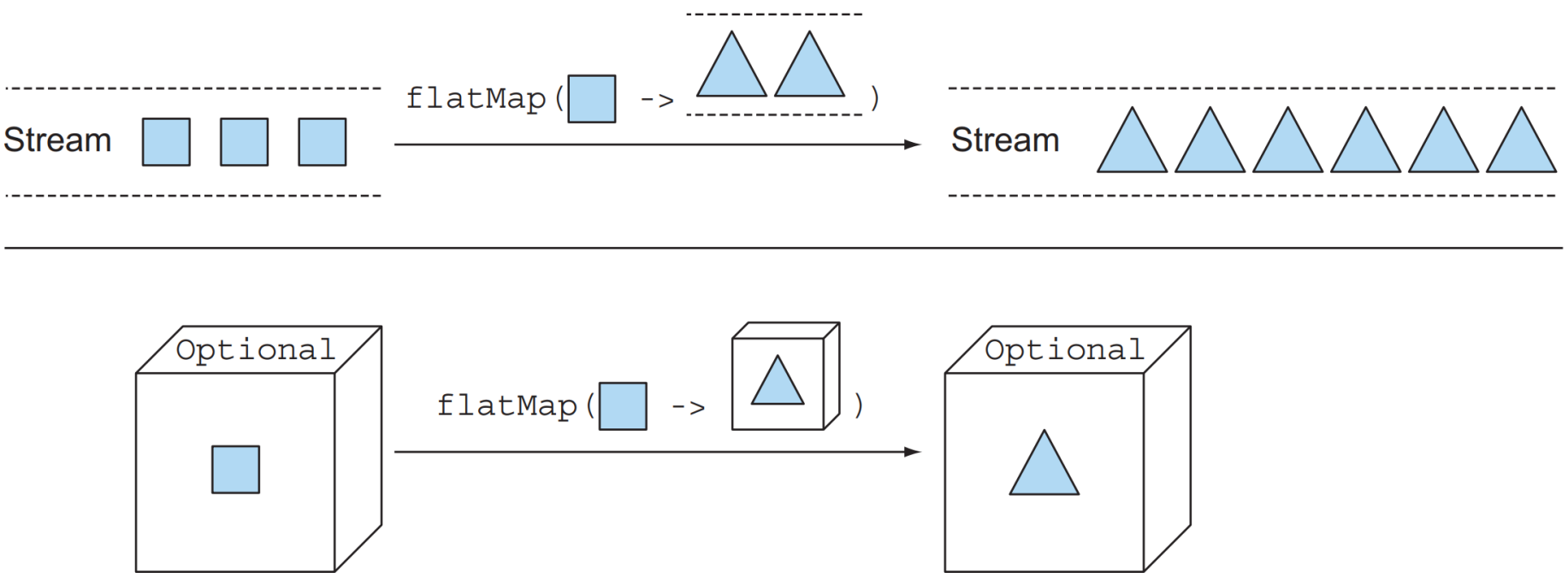


Figure 11.4 Comparing the `flatMap` methods of `Stream` and `Optional`

FINDING A CAR'S INSURANCE COMPANY NAME WITH OPTIONALS

Listing 11.5 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");
}
```

A default value if the resulting Optional is empty

PERSON/CAR/INSURANCE DEREFERENCING CHAIN USING OPTIONALS

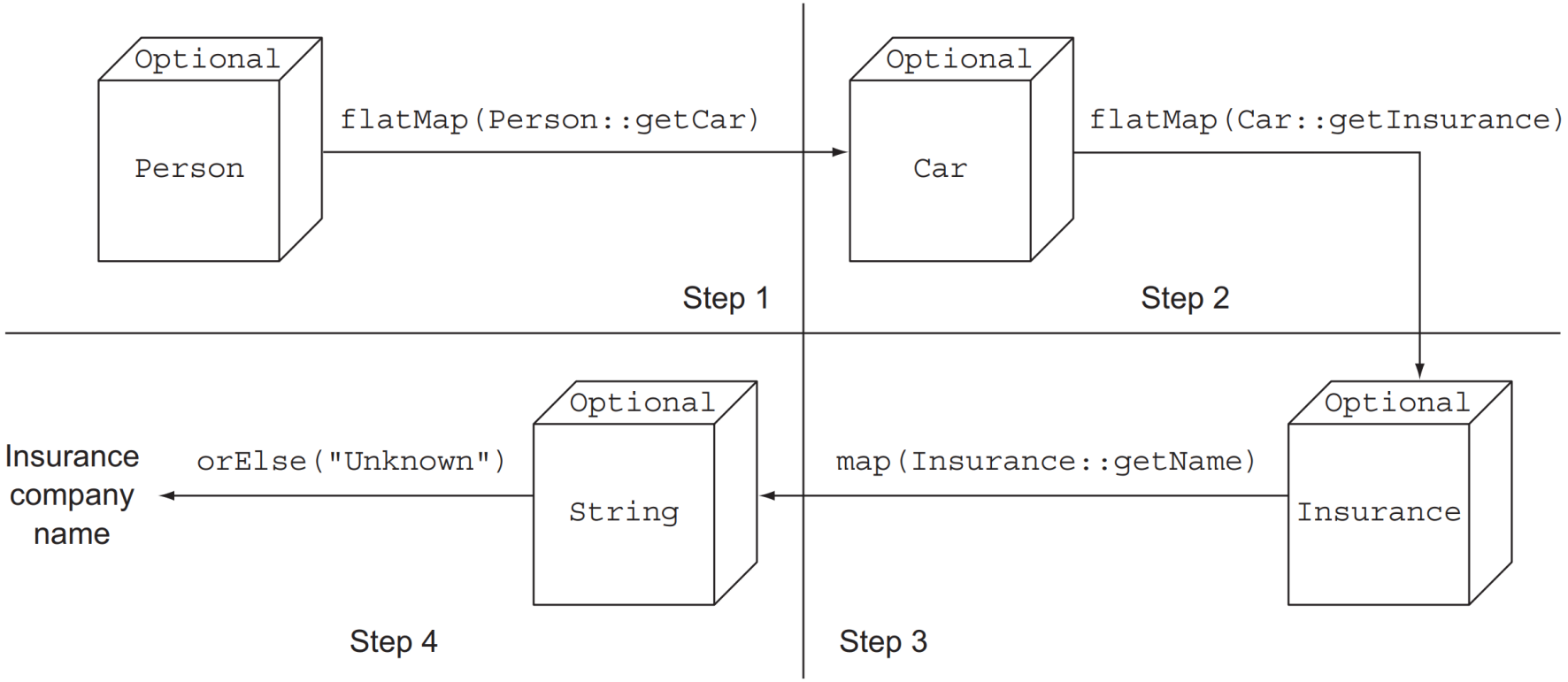


Figure 11.5 The `Person/Car/Insurance` dereferencing chain using optionals

Manipulating a stream of optionals

- The `Optional`'s `stream()` method, introduced in Java 9, allows you to convert an `Optional` with a value to a `Stream` containing only that value or an empty `Optional` to an equally empty `Stream`.

Listing 11.6 Finding distinct insurance company names used by a list of `persons`

Convert the list of persons into a Stream of Optional<Car> with the cars eventually owned by them.

FlatMap each Optional<Car> into the corresponding Optional<Insurance>.

```
public Set<String> getCarInsuranceNames(List<Person> persons) {
    return persons.stream()
                  .map(Person::getCar)
                  .map(optCar -> optCar.flatMap(Car::getInsurance))
                  .map(optIns -> optIns.map(Insurance::getName))
                  .flatMap(Optional::stream)
                  .collect(toSet());
}
```

Map each Optional<Insurance> into the Optional<String> containing the corresponding name.

Transform the Stream<Optional<String>> into a Stream<String> containing only the present names.

Collect the result Strings into a Set to obtain only the distinct values.

- You could have obtained this result with a `filter` followed by a `map`, of course, as follows:

```
Stream<Optional<String>> stream = ...
Set<String> result = stream.filter(Optional::isPresent)
                           .map(Optional::get)
                           .collect(toSet());
```

Default actions and unwrapping an Optional

- The `Optional` class provides several instance methods to read the value contained by an `Optional` instance:
  - `get()` is the simplest but also the least safe of these methods. It returns the wrapped value if one is present and throws a `NoSuchElementException` otherwise.
  - `orElse(T other)` allows you to provide a default value when the optional doesn't contain a value.
  - `orElseGet(Supplier<? extends T> other)` is the lazy counterpart of the `orElse` method, because the supplier is invoked only if the optional contains no value.
  - `orElseGet(Supplier<? extends T> other)` is the lazy counterpart of the `orElse` method, because the supplier is invoked only if the optional contains no value.

Default actions and unwrapping an Optional (Cont.)

- `orElseThrow(Supplier<? extends X> exceptionSupplier)` is similar to the `get` method in that it throws an exception when the `optional` is empty, but it allows you to choose the type of exception that you want to throw.
- `ifPresent(Consumer<? super T> consumer)` lets you execute the action given as argument if a value is present; otherwise, no action is taken.
- `ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction)`. This differs from `ifPresent` by taking a `Runnable` that gives an empty-based action to be executed when the `Optional` is empty.

Combining two Optionals

Now suppose that you have a method that, given a `Person` and a `Car`, queries some external services and implements some complex business logic to find the insurance company that offers the cheapest policy for that combination:

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

Also suppose that you want to develop a null-safe version of this method, taking two optionals as arguments and returning an `Optional<Insurance>` that will be empty if at least one of the values passed in to it is also empty. The `Optional` class also provides an `isPresent` method that returns `true` if the optional contains a value, so your first attempt could be to implement this method as follows:

```
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();
    }
}
```

Quiz 11.1: Combining two optionals without unwrapping them

Using a combination of the `map` and `flatMap` methods you learned in this section, rewrite the implementation of the former `nullSafeFindCheapestInsurance()` method in a single statement.

Answer:

You can implement that method in a single statement and without using any conditional constructs like the ternary operator as follows:

```
public Optional<Insurance> nullSafeFindCheapestInsurance(
                                Optional<Person> person, Optional<Car> car) {
    return person.flatMap(p -> car.map(c -> findCheapestInsurance(p, c)));
}
```

Here, you invoke a `flatMap` on the first optional, so if this optional is empty, the lambda expression passed to it won't be executed, and this invocation will return an empty optional. Conversely, if the person is present, `flatMap` uses it as the input to a `Function` returning an `Optional<Insurance>` as required by the `flatMap` method. The body of this function invokes a `map` on the second optional, so if it doesn't contain any `Car`, the `Function` returns an empty optional, and so does the whole `nullSafeFindCheapestInsurance` method. Finally, if both the `Person` and the `Car` are present, the lambda expression passed as an argument to the `map` method can safely invoke the original `findCheapestInsurance` method with them.

Rejecting certain values with filter

Often, you need to call a method on an object to check some property. You may need to check whether the insurance's name is equal to `CambridgeInsurance`, for example. To do so in a safe way, first check whether the reference that points to an `Insurance` object is `null` and then call the `getName` method, as follows:

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

You can rewrite this pattern by using the `filter` method on an `Optional` object, as follows:

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

The `filter` method takes a predicate as an argument. If a value is present in the `Optional` object, and that value matches the predicate, the `filter` method returns that value; otherwise, it returns an empty `Optional` object. If you remember that you

Quiz 11.2: Filtering an optional

Supposing that the `Person` class of your `Person/Car/Insurance` model also has a method `getAge` to access the age of the person, modify the `getCarInsuranceName` method in listing 11.5 by using the signature

```
public String getCarInsuranceName(Optional<Person> person, int minAge)
```

so that the insurance company name is returned *only* if the person has an age greater than or equal to the `minAge` argument.

Answer:

You can filter the `Optional<Person>`, to remove any contained person whose age fails to be at least the `minAge` argument, by encoding this condition in a predicate passed to the `filter` method as follows:

```
public String getCarInsuranceName(Optional<Person> person, int minAge) {
    return person.filter(p -> p.getAge() >= minAge)
                 .flatMap(Person::getCar)
                 .flatMap(Car::getInsurance)
                 .map(Insurance::getName)
                 .orElse("Unknown");
}
```

Table 11.1 summarizes the methods of the `Optional` class.

Table 11.1 The methods of the `Optional` class

Method	Description
`empty`	Returns an empty `Optional` instance
`filter`	If the value is present and matches the given predicate, returns this `Optional`; otherwise, returns the empty one
`flatMap`	If a value is present, returns the `Optional` resulting from the application of the provided mapping function to it; otherwise, returns the empty `Optional`
`get`	Returns the value wrapped by this `Optional` if present; otherwise, throws a `NoSuchElementException`

Table 11.1 The methods of the `Optional` class *(continued)*

Method	Description
`ifPresent`	If a value is present, invokes the specified consumer with the value; otherwise, does nothing
`ifPresentOrElse`	If a value is present, performs an action with the value as input; otherwise, performs a different action with no input
`isPresent`	Returns `true` if a value is present; otherwise, returns `false`
`map`	If a value is present, applies the provided mapping function to it
`of`	Returns an `Optional` wrapping the given value or throws a `NullPointerException` if this value is `null`
`ofNullable`	Returns an `Optional` wrapping the given value or the empty `Optional` if this value is `null`

Table 11.1 The methods of the `Optional` class *(continued)*

Method	Description
`or`	If the value is present, returns the same `Optional`; otherwise, returns another `Optional` produced by the supplying function
`orElse`	Returns the value if present; otherwise, returns the given default value
`orElseGet`	Returns the value if present; otherwise, returns the one provided by the given `Supplier`
`orElseThrow`	Returns the value if present; otherwise, throws the exception created by the given `Supplier`
`stream`	If a value is present, returns a `Stream` containing only it; otherwise, returns an empty `Stream`

11.4 Practical examples of using Optional

- For backward-compatibility reasons, old Java APIs can't be changed to make proper use of optionals, but all is not lost. You can fix, or at least work around, this issue by adding to your code small utility methods that allow you to benefit from the power of optionals.

Wrapping a potentially null value in an Optional

- An existing Java API almost always returns a `null` to signal that the required value is absent or that the computation to obtain it failed for some reason. The `get` method of a `Map` returns `null` as its value if it contains no mapping for the requested key, for example.
- Suppose that you have a `Map<String, Object>`, accessing the value indexed by `key` with

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

returns `null` if there's no value in the `map` associated with the `String` `"key"`.

- You can either add an ugly `if-then-else` that adds to code complexity, or you can use the method `Optional.ofNullable` that we discussed earlier:

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

Exceptions vs. Optional

- Throwing an exception is another common alternative in the Java API to returning a `null` when a value can't be provided. A typical example is the conversion of `String` into an `int` provided by the `Integer.parseInt(String)` static method. In this case, if the `String` doesn't contain a parseable integer, this method throws a `NumberFormatException`.
- You could also model the invalid value caused by nonconvertible `String`s with an empty optional, so you prefer that `parseInt` returns an optional.

Listing 11.7 Converting a `String` to 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();
    }
}
```

If the String can be converted to an Integer, return an optional containing it.

Otherwise, return an empty optional.

Our suggestion is to collect several similar methods in a utility class, which you can call `OptionalUtility`. From then on, you'll always be allowed to convert a `String` to an `Optional<Integer>` by using this `OptionalUtility.stringToInt` method. You can forget that you encapsulated the ugly `try/catch` logic in it.

Primitive optionals and why you shouldn't use them

- Note that like streams, optionals also have primitive counterparts—`OptionalInt`, `OptionalLong`, and `OptionalDouble`.
- We discourage using primitive optionals because they lack the `map`, `flatMap`, and `filter` methods, which are the most useful methods of the `Optional` class.

Putting it all together

- Suppose that you have some `Properties` that are passed as configuration arguments to your program. For the purpose of this example and to test the code you'll develop, create some sample `Properties` as follows:

```
Properties props = new Properties();
props.setProperty("a", "5");
props.setProperty("b", "true");
props.setProperty("c", "-3");
```

- Also suppose that your program needs to read a value from these `Properties` and interpret it as a duration in seconds. Because a duration has to be a positive (>0) number, you'll want a method with the signature

```
public int readDuration(Properties props, String name)
```

so that when the value of a given property is a `String` representing a positive integer, the method returns that integer, but it returns zero in all other cases.

- Try to implement the method that satisfies this requirement in imperative style, as shown in the next listing.

Listing 11.8 Reading duration from a property imperatively

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

Make sure that a property exists with the required name.

Try to convert the String property to a number.

Check whether the resulting number is positive.

Return 0 if any of the conditions fails.

Quiz 11.3: Reading duration from a property by using an Optional

Using the features of the `Optional` class and the utility method of listing 11.7, try to reimplement the imperative method of listing 11.8 with a single fluent statement.

Answer:

Because the value returned by the `Properties.getProperty(String)` method is a `null` when the required property doesn't exist, it's convenient to turn this value into an optional with the `ofNullable` factory method. Then you can convert the `Optional<String>` to an `Optional<Integer>`, passing to its `flatMap` method a reference to the `OptionalUtility.stringToInt` method developed in listing 11.7. Finally, you can easily filter away the negative number. In this way, if any of these operations returns an empty optional, the method returns the `0` that's passed as the default value to the `orElse` method; otherwise, it returns the positive integer contained in the optional. This description is implemented as follows:

```
public int readDuration(Properties props, String name) {
    return Optional.ofNullable(props.getProperty(name))
                   .flatMap(OptionalUtility::stringToInt)
                   .filter(i -> i > 0)
                   .orElse(0);
}
```