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

Raise your hand if you ever got a `NullPointerException` during your life as a Java developer. Keep it up if this `Exception` is the one you encounter most frequently. Unfortunately, we can't see you at this moment, but we believe that there's a high probability that your hand is raised now. We also guess that you may be thinking something like "Yes, I agree. `NullPointerExceptions` are a pain for any Java developer, novice, or expert. But there's not much we can do about them, because this is the price we pay to use such a convenient, and maybe unavoidable, construct as

`null` references." This feeling is common in the (imperative) programming world; nevertheless, it may not be the whole truth and is more likely a bias with solid historical roots.

British computer scientist Tony Hoare introduced `null` references back in 1965 while designing ALGOL W, one of the first typed programming languages with heap-allocated records, later saying that he did so "simply because it was so easy to implement." Despite his goal "to ensure that all use of references could be absolutely safe, with checking performed automatically by the compiler," he decided to make an exception for `null` references because he thought that they were the most convenient way to model *the absence of a value*. After many years, he regretted this decision, calling it "my billion-dollar mistake." We've all seen the effect. We examine a field of an object, perhaps to determine whether its value is one of two expected forms, only to find that we're examining not an object but a `null` pointer that promptly raises that annoying `NullPointerException`.

In reality, Hoare's statement could underestimate the costs incurred by millions of developers fixing bugs caused by `null` references in the past 50 years. Indeed, the vast majority of the languages[1] created in recent decades, including Java, have been built with the same design decision, maybe for reasons of compatibility with older languages or (more probably), as Hoare states, "simply because it was so easy to implement." We start by showing you a simple example of the problems with `null`.

11.1 How do you model the absence of a value?

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; }
}
```

[1] Notable exceptions include most typed functional languages, such as Haskell and ML. These languages include *algebraic data types* that allow data types to be expressed succinctly, including explicit specification of whether special values such as `null` are to be included on a type-by-type basis.