Designing with Delegates

We'll cover the following

- A design problem
- The inheritance misroute
- Delegation, the hard way
- Delegation using Kotlin's by

In the previous chapter, you saw that Kotlin offers a different and safer way to use inheritance than Java. Likewise, Kotlin has much better support than Java for delegation. Before we dive into the syntax for delegation, we'll take a small problem and design it using inheritance, to better understand the reasons to use delegation over inheritance. Soon we'll run into issues where inheritance begins to become a hindrance, and you'll see how delegation may be a better choice for the problem. We'll then look at how Kotlin is able to readily support our new design using delegation, without flinching.

A design problem

Imagine an application that simulates the execution of software projects by corporate teams (don't worry, we'll limit the scope so our program, unlike some enterprise projects, actually completes). We need workers to get real stuff done—let's start with a Worker interface:

```
// version1/project.kts
interface Worker {
  fun work()
  fun takeVacation()
}
```

A worker may perform two tasks: does work and occasionally takes vacation. Let's implement two classes, `JavaProgrammer` and `CSharpProgrammer`, that specialize in two different languages—the idea of being polyglot has not caught this company's attention yet.

attention yet.

```
class JavaProgrammer : Worker {
  override fun work() = println("...write Java...")
  override fun takeVacation() = println("...code at the beach...")
}

class CSharpProgrammer : Worker {
  override fun work() = println("...write C#...")
  override fun takeVacation() = println("...branch at the ranch...")
}
```

version1/project.kts

The programmers perform their `work()` based on their language specialty and enjoy their vacations according to the established industry standards.

The company wants to have a software development manager for their teams—let's write that class:

```
// version1/project.kts
class Manager
```

The `Manager` class is very small, agile, and highly efficient, and as you'd expect, does nothing. A call to `work()` on an instance of this manager won't lead anywhere. We need to design some logic into the `Manager`, but that's not easy.

The inheritance misroute

Now that we have the interface Worker and the two implementations, `JavaProgrammer` and `CSharpProgrammer`, let's focus on using them with the `Manager` class. The company will want to rely on the `Manager` to execute and deliver the project. The `Manager`, in turn, will need a programmer to get the actual work done. In the most simple form, a call to `work()` on the `Manager` will have to be executed on an implementation of `Worker`. One way to achieve this is to use inheritance, which is a common approach we use in Java. By inheriting `Manager` from `JavaProgrammer`, we won't have to write the `work()` method in the `Manager` class, and that often serves as a temptation to use inheritance. Let's take this approach and see where it leads.

As a first step, we have to annotate the JavaProgrammer class as open:

```
// version2/project.kts
```

```
open class JavaProgrammer : Worker {
```

Then, we can enhance the `Manager` class to inherit from the `JavaProgrammer` class:

```
// version2/project.kts
class Manager : JavaProgrammer()
```

Now we can call `work()` on an instance of Manager:

```
// version2/project.kts
val doe = Manager()
doe.work() //...write Java...
```

That worked, but this design has a drawback. This `Manager` class is stuck to the `JavaProgrammer` class and can't use what's offered by a `CSharpProgrammer` class (or any other classes that implement `Worker` in the future). That's not fair, but that's the consequence of inheritance. And let's examine another unintended consequence—substitutability.

We didn't mean `Manager` is a `JavaProgrammer` or a kind-of `JavaProgrammer`, but sadly that was implied from inheritance. As a result, the following code will compile:

```
// version2/project.kts
val coder: JavaProgrammer = doe
```

Even though this wasn't intended by the design, we can't stop it. Our real intention here is that `Manager` should rely upon or use a `JavaProgrammer` or any worker that can get the tasks done. But that's delegation, not inheritance. We want to be able to delegate work from a `Manager` instance to the instance of any `Worker`. Let's move toward that design and see how it solves the problem at hand, but without bringing along the aforementioned unintended behaviors.

Delegation, the hard way

Although in languages like Java we have a syntax for inheritance, there's nothing to specify delegation. You may use a reference to refer to another object, but the language leaves you with the full burden to implement the design.

Let's look at an example of delegation. Even though the code here is in Kotlin, for a few minutes we'll use only facilities that are available in Java. Here's Kotlin code

with the Java approach to delegate `Manager` to a `Worker`.

```
// version3/project.kts
class Manager(val worker: Worker) {
 fun work() = worker.work()

 fun takeVacation() = worker.work() //yeah, right, like that's gonna happen
}
```

Before we discuss the quality of this design, let's use the modified `Manager` class:

```
// version3/project.kts
val doe = Manager(JavaProgrammer())
doe.work() //...write Java...
```

We created an instance of `Manager` and passed an instance of `JavaProgrammer` as an argument to the constructor. The benefit this design has over inheritance is that the `Manager` isn't tightly coupled to the `JavaProgrammer` class. Thus, we may instead pass to the constructor an instance of `CSharpProgrammer` class, or just about any class that implements the `Worker` interface. In other words, an instance of `Manager` may delegate to an instance of any class that implements `Worker`—`JavaProgrammer`, `CSharpProgrammer`, and so on.

An additional benefit in this solution is that the `JavaProgrammer` class no longer has to be marked as `open` since we're not inheriting from it.

But an undesirable aspect to this design is that the code is verbose and violates a few fundamental software design principles. Let's discuss the issues one at a time.

Within the `Manager` class, we implement the `work()` method that merely routes the call to the instance of `Worker` that's referenced by the `Manager` instance. Likewise, in the `takeVacation()` method, we're merely routing the call to the `worker` reference. Imagine having more methods in `Worker`—that's even more routing code in `Manager`. All the routing code looks the same, except for the method routed to. That's a violation of the Don't Repeat Yourself or ***DRY*** principle presented in The Pragmatic Programmer From Journeyman to Master.

Besides not being DRY, the code also fails the Open-Closed Principle (OCP) coined by Bertrand Meyer and discussed in Agile Software Development, Principles, Patterns, and Practices. The principle says that a software module—classes, functions, and so on—should be open for extension but closed from modification.

functions, and so on—should be open for extension but closed from modification. In other words, to extend a class we shouldn't have to change it. Sadly, though, in

the current implementation of the design, suppose we add a method `deploy()` to `Worker`, and the `Manager` wants to delegate calls to that method, then we'll have to change the `Manager` class to add the routing method—OCP violation.

These issues give us a sense of why Java programs use inheritance more than delegation—delegation has issues due to lack of language support, and inheritance is so easy to reach for. But `Manager` isn't a `JavaProgrammer`, and modeling it using inheritance will lead to violation of LSP. The programmers are left with a "doomed if you do, doomed if you don't" kind of solution.

Kotlin solves this problem by supporting delegation at the language level.

Delegation using Kotlin's `by`

In the previous example we implemented delegation by hand, routing method calls from `Manager` to the `Worker` delegate. The body of the `Manager` is smelly with all these duplicated method calls and the DRY and OCP violations. That's the only option if we were programming in Java. But in Kotlin, instead, we can ask the compiler to generate the crufty routing code for us, and then the `Manager` can route like a boss without any fuss.

Let's take a small step to explore the simplest use of delegation in Kotlin for this problem at hand.

```
// version4/project.kts
class Manager() : Worker by JavaProgrammer()
```

This version of `Manager` doesn't have any methods of its own, at least not at code-writing time. It implements the `Worker` interface by way of or via the `JavaProgrammer`. Upon seeing the `by` keyword, the Kotlin compiler will implement, at the bytecode level, the methods in the `Manager` class that belong to Worker, and route the call to the `JavaProgrammer` instance supplied after the `by` keyword. In other words, the `by` keyword in this example does at compile time what we painstakingly did manually in the previous example where we implemented delegation by hand.

Kotlin requires the left side of the by to be an interface. The right side is an implementor of that interface.

Let's exercise this version of the `Manager` class by creating an instance:

```
// version4/project.kts
val doe = Manager()
doe.work() //...write Java...
```

It was easy to create an instance of `Manager` and call the `work()` method on it.

At first sight, this solution almost looks like the inheritance solution, but there are some key differences. First, the class `Manager` isn't inheriting from `JavaProgrammer`. Recall that with the inheriting solution we were able to assign an instance of `Manager` to a reference of type `JavaProgrammer`. Thankfully, that's no longer possible, and the following will result in an error:

```
// version4/project.kts
val coder: JavaProgrammer = doe //ERROR: type mismatch
```

Second, in the inheritance solution, calls to methods like `work()` weren't implemented in `Manager`; instead they were sent to the base class. In the case of Kotlin delegation, the compiler internally creates methods within the `Manager` class and does the routing. In effect, when we call `doe.work()` we are calling the invisible method `work()` within the `Manager` class. This method, synthesized by the Kotlin compiler, routes the call to the delegate, the instance of `JavaProgrammer` given in the class declaration.

QUIZ

1 When should delegation be used?

2

Which classes are placed around the `by` operator to implemet delegation?

Retake Quiz

The above solution is the simplest form of delegation but has some limitations. We'll identify those and resolve them in the next lesson.