Caveats of Kotlin Delegation

We'll cover the following

Implementation caveat

In the example we've created so far, the `Manager` may delegate calls to an instance of a `JavaProgrammer`, but a reference to a `Manager` may not be assigned to a reference of a `JavaProgrammer`—that is, a `Manager` may use a `JavaProgrammer`, but a `Manager` may not be used as a `JavaProgrammer`. In other words, a `Manager` has a `JavaProgrammer` but is not a kind of `JavaProgrammer`. Thus, delegation offers reuse without accidentally leading to substitutability as inheritance does.

However, there's one small consequence of how Kotlin implements delegation. The delegating class implements the delegating interface, so a reference to the delegating class may be assigned to a reference of the delegating interface. Likewise, a reference to a delegating class may be passed to methods that expect a delegate interface. In other words, for example, the following isn't valid:

```
val coder: JavaProgrammer = doe //ERROR: type mismatch
```

But the following is possible in Kotlin:

```
val employee: Worker = doe
```

This means a `Manager` is a, or kind of a, `Worker`. The true intention of delegation is for a `Manager` to use a `Worker`, but Kotlin's delegate implementation introduces a side effect that a `Manager` may be treated as a `Worker`.

Delegating to a property caveat

Also, use caution when delegating to a property. We passed a parameter of type

`Worker` to the constructor of `Manager`, using `val`. In Prefer val over var, we discussed why we should prefer `val` over `var`. That recommendation is useful here too. If we decide to change the property that's used as a delegate from `val` to `var`, a few consequences arise. Let's see what those are with an example.

Here's the full listing we'll use for this example:

```
interface Worker {
  fun work()
  fun takeVacation()
  fun fileTimeSheet() = println("Why? Really?")
}

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...")
}

class Manager(var staff: Worker) : Worker by staff

val doe = Manager(JavaProgrammer())

println("Staff is ${doe.staff.javaClass.simpleName}")
doe.work()

println("changing staff")
doe.staff = CSharpProgrammer()
println("Staff is ${doe.staff.javaClass.simpleName}")
doe.work()
```

version8/project.kt

The `Manager`'s primary constructor defines a mutable property named `staff` and delegates to that same object. We created the instance doe by passing an instance of `JavaProgrammer`. Then we get the property `staff` from the instance `doe` and ask for the type. When we call `work()` on doe we expect the work method of the `JavaProgrammer` attached to the `staff` property to be called.

Then we change the `staff` property to an instance of the `CSharpProgrammer` class and check the type of the object referenced by the `staff` property, to confirm it is `CSharpProgrammer` now and not `JavaProgrammer`. Finally, we invoke the `work()` method. Which work method does this call land in, `CSharpProgrammer` or

`JavaProgrammer`?

Let's look carefully at the declaration once again:

```
class Manager(var staff: Worker) : Worker by staff
```

It's easy to mistake what the delegate is. The delegate at the far right is the parameter and not the property. The declaration is really taking a parameter named `staff` and assigning it to a member named `staff`, like so: `this.staff = staff`. So there are two references to the given object: one held inside the class as a backing field and one held for the purpose of delegation. When we change the property to an instance of `CSharpProgrammer`, though, we only modified the field and not the reference to the delegate. Hence the following output:

```
Staff is JavaProgrammer
...write Java...
changing staff
Staff is CSharpProgrammer
...write Java...
```

Kotlin doesn't delegate to a property of an object but to the parameter passed to the primary constructor—this behavior may change in the future.

Another problem arises with the code we just used. When we replace `staff` with an instance of `CSharpProgrammer`, the originally attached instance of `JavaProgrammer` is no longer attached to the object as its property. But it can't get garbage collected since the delegate holds on to it. Thus, the lifetime of the delegate is the same as the object's lifetime, though the properties may come and go along the way.

Knowing these caveats will help you safely benefit from the delegates feature. But there's more to delegates than applying on an entire class. Let's dig in further in the next lesson.