Delegating to a Parameter

We'll cover the following

The problem

In the previous example, we wrote `Worker` by `JavaProgrammer()`, which says that the `Manager` instance is delegating to an implicitly created instance of the `JavaProgrammer`, but that poses two issues. First, the instances of `Manager` class can only route to instances of `JavaProgrammer`, not to instances of any other `Worker` implementors. Second, an instance of `Manager` doesn't have access to the delegate; that is, if we were to write a method in the `Manager` class, we can't access the delegate from that method.

The solution

It's easy to fix those limitations by tying the delegate to the parameter passed to the constructor instead of to an implicitly created instance.

```
// version5/project.kt
class Manager(val staff: Worker) : Worker by staff {
  fun meeting() =
    println("organizing meeting with ${staff.javaClass.simpleName}")
}
```

The constructor of the `Manager` class receives a parameter named `staff` which also serves as a property, due to `val` in the declaration. If the `val` is removed, `staff` will still be a parameter, but not a property of the class. Irrespective of whether or not `val` is used, the class can delegate to the parameter `staff`.

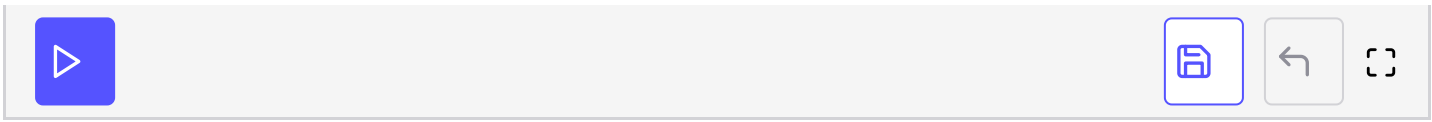
In the `meeting()` method of the `Manager` class, we're able to access `staff` since it's a property of the object. Calls to methods like `work()` will go to staff due to delegation. Let's confirm this behavior by creating a couple of instances of this

delegation. Let's confirm this behavior by creating a couple of instances of this version of `Manager`.

```
val doe = Manager(CSharpProgrammer())
val roe = Manager(JavaProgrammer())

doe.work() //...write C#...
doe.meeting()//organizing meeting with CSharpProgrammer

roe.work() //...write Java...
roe.meeting()//organizing meeting with JavaProgrammer
```



version5/project.kt

The first instance of `Manager` receives an instance of `CSharpProgrammer`, and the second instance receives an instance of `JavaProgrammer`. That shows that the delegation is flexible; it's not tied to one class, `JavaProgrammer`, and may use different implementations of `Worker`. When `work()` is called on the two instances of `Manager`, Kotlin routes the calls automatically to the respective delegates. Also, when `meeting()` is called on the `Manager` instances, those calls send an invitation to the respective staff properties that are part of the `Manager` instances.

In the next lesson, we'll learn to deal with method collisions inside delegate classes.