

# Dealing with Method Collisions

## We'll cover the following ^

- Why collisions occur
- How to deal with collisions
- Collisions in two interfaces

## Why collisions occur #

The Kotlin compiler creates a wrapper in the delegating class for each method that's in the delegate. What if there's a method in the delegating class with the same name and signature as in the delegate? Kotlin resolves this conflict in favor of the delegating class. As a consequence, you can be selective and don't have to delegate every single method of the delegate class—let's explore this further.

In the previous example, the `Worker` has a `takeVacation()` method and the `Manager` is delegating calls to that method to the `Worker` delegate. Although that's the default behavior for delegation in Kotlin, it's unlikely that any `Manager` would settle for that; while it makes perfect sense for a `Manager` to delegate `work()`, one would expect the `takeVacation()` to be executed on the instance of `Manager` and not be delegated.

## How to deal with collisions #

Kotlin requires the delegating class to implement a delegate interface, but without actually implementing each of the methods of the interface. We saw this in the `Manager`—it implements `Worker`, but didn't provide any implementations for the `work()` or `takeVacation()` methods. For every method of the delegate interface, the Kotlin compiler creates a wrapper. But that's true only if the delegating class doesn't already have an implementation of a method. If the delegating class has an implementation for a method that's in the interface, then it must be marked as `override`, that implementation takes precedence, and a wrapper method isn't created.

To illustrate this behavior, let's implement the `takeVacation()` method in the `Manager`:

```
// version6/project.kts
class Manager(val staff: Worker) : Worker by staff {
    override fun takeVacation() = println("of course")
}
```

With the `override` keyword, you're making it very clear to the reader of the code that you're implementing a method from the interface, not some arbitrary function that happens to accidentally have the same name as a method in the delegate. Seeing this, the Kotlin compiler won't generate a wrapper for `takeVacation()`, but it will generate a wrapper for the `work()` method.

Let's invoke the methods of the interface on an instance of this version of the `Manager` class.

```
val doe = Manager(CSharpProgrammer())
doe.work()           //...write C#...
doe.takeVacation()  //of course
```



version6/project.kt

The call to the `work()` method on the instance of `Manager` is routed to the delegate, but the call to the `takeVacation()` method is handled by the `Manager`—no delegation there. Kotlin nicely takes care of resolving method collisions and also makes it easy to override select methods of the delegate interface in the delegating class.

## Collisions in two interfaces #

In the examples so far, we've looked at delegating the methods of a single interface to an implementing class. A class may delegate to multiple interface implementors as well. If there are any method collisions between the interfaces, then the candidate class should override the conflicting methods. The next example will illustrate this behavior.

Let's modify the `Worker` interface to add a new method:

```
// version7/project.kts
interface Worker {
    fun work()
    fun takeVacation()
    fun fileTimeSheet() = println("Why? Really?")
}
```

The new method `fileTimesheet()` is implemented within the interface. The classes that implement the `Worker` interface can readily reuse it but may override that method if they dare.

Let's now look at a new interface, `Assistant`, with a few methods:

```
// version7/project.kts
interface Assistant {
    fun doChores()
    fun fileTimeSheet() = println("No escape from that")
}
```

This interface has two methods, one of which is implemented in it. Here's a class that implements the `Assistant` interface:

```
// version7/project.kts
class DepartmentAssistant : Assistant {
    override fun doChores() = println("routine stuff")
}
```

Let's see the effect of delegating to both interfaces.

```
class Manager(val staff: Worker, val assistant: Assistant) :
    Worker by staff, Assistant by assistant {

    override fun takeVacation() = println("of course")

    override fun fileTimeSheet() {
        print("manually forwarding this...")
        assistant.fileTimeSheet()
    }
}
```

version7/project.kt

The `Manager` class now has two properties defined as the primary constructor

parameters. It also delegates to both of those objects, the first for the `Worker` interface and the second for the `Assistant` interface. If we don't implement the `fileTimeSheet()` method in the `Manager` class, we'll get a compilation error due to the collision of the `fileTimeSheet()` methods in the two interfaces. To resolve this conflict, we have implemented that method in the `Manager` class.

Let's use the latest version of the `Manager` class to see the delegation to two objects in action:

```
val doe = Manager(CSharpProgrammer(), DepartmentAssistant())
doe.work()           //...write C#...
doe.takeVacation()   //of course
doe.doChores()        //routine stuff
doe.fileTimeSheet()  //manually forwarding this...No escape from that
```



version7/project.kt

The call to the `work()` method was delegated to the implementation of the `Worker` interface. The call to `takeVacation()` wasn't delegated; it was executed on the `Manager` instance. The call to `doChores()` method was delegated to the implementation of the `Assistant` interface. Finally, the call to `fileTimeSheet()` method was executed on the `Manager` instance. Thus, the `Manager` instance has intercepted the call to `fileTimeSheet()`, avoiding any conflict or ambiguity to a call on `Worker` or on an `Assistant`. The `Manager` instance can decide to manually route the call to either the `Worker` implementation, or the `Assistant` implementation, or to both. In this example, the `Manager` intercepted the call to `fileTimeSheet()` and then delegated manually to the implementor of the `Assistant` interface via the `assistant` property.

## QUIZ



Why do collisions occur in a delegating class?



How do you deal with a collision in the delegating class?

[Retake Quiz](#)

---

Kotlin's facility for delegation is a breath of fresh air when you consider what's lacking in Java. But we have to be careful about a few things when using delegation. Let's go over those in the next lesson to avoid any surprises later.