Creating Interfaces and Abstract Classes

We'll cover the following

Object-oriented programming is programming with hierarchies of abstractions—complex applications typically have multiple interfaces, abstract classes, and implementation classes. We heavily rely on interfaces to specify behavior of abstractions and use abstract classes to reuse implementations. So learning about interfaces and abstract classes is a good starting point for learning about building object-oriented hierarchies.

In recent years interfaces have dramatically evolved in Java. In the past, interfaces were permitted to carry only method declarations and no implementation: they could tell a lot about what's possible, but never did anything—kinda like my boss. In recent versions of Java, interfaces can have default methods, static methods, and even private methods. With such improvements in Java, it's fair to ask how interfaces in Kotlin measure up to interfaces in modern Java. We'll start with that question and explore how Kotlin interfaces fare in comparison. Then we'll explore how abstract classes differ in Kotlin when compared to Java.

Creating interfaces

Semantically, Kotlin interfaces are a lot like Java interfaces, but they differ widely in syntax. You can honor the original design-by-contract intent of interfaces by writing abstract methods in them. In addition, you may also implement methods within interfaces, but without the `default` keyword that Java requires. Much like the way `static` methods in Kotlin classes are placed in companion objects, interfaces can also have `static` methods, but only via companion objects written within the interfaces.

To get a good feel for interfaces in Kotlin, we'll create an interface with a few abstract methods and a method with implementation. Then we'll inherit from that interface to see how the implementation gets carried over to the derived class. Finally, we'll look at the Kotlin way to define `static` methods for interfaces.

Let's start with a `Remote` interface that represents a remote control device.

```
interface Remote {
  fun up()
  fun down()

  fun doubleUp() {
    up()
    up()
  }
}
```

remote.kts

Abstract methods, like `up()`, are merely declared within the interface. Implementing a method, like `doubleUp()`, in an interface is much like implementing a method in a class—no extra syntax or ceremony. Any class implementing the interface will have to override the abstract methods. However, they get the implementation in the `doubleUp()` method for free and may optionally override the implementation as well.

The methods implemented within interfaces—that is, Kotlin-type default methods—work even in Java 1.6, even though `default` methods were introduced only in Java 8. To make the Kotlin `default` methods visible as default methods in the bytecode, you can use the `@JvmDefault` annotation, which we'll cover in More Annotations.

To see how the interface interplays with an implementor, let's implement the `Remote` interface in a `TVRemote` class that may be used to control the volume of a TV:

```
class TV {
  var volume = 0
}

class TVRemote(val tv: TV) : Remote {
  override fun up() { tv.volume++ }
  override fun down() { tv.volume-- }
}
```

remote.kts

To convey that the class `TVRemote` is implementing the `Remote` interface, we place `Remote` following the colon `:` after the primary constructor's parameter list. Kotlin follows the Ruby and C# style for inheritance syntax. The `TVRemote` is required to implement the abstract methods of `Remote`, and the `override` keyword is required for any method that is overriding a method of a base class or an interface. The `TVRemote` class isn't overriding the implementation of `doubleUp()` in Remote.

To see how these classes and the interface work together, let's exercise the methods of `Remote` through an instance of `TVRemote`.

```
val tv = TV()
val remote: Remote = TVRemote(tv)

println("Volume: ${tv.volume}") //Volume: 0
remote.up()
println("After increasing: ${tv.volume}") //After increasing: 1
remote.doubleUp()
println("After doubleUp: ${tv.volume}") //After doubleUp: 3
```

remote.kts

The reference remote is of type `Remote` but refers to an instance of `TVRemote` at runtime. The call to `up()` was handled by the `TVRemote` instance, but the call to `doubleUp()` landed in the implementation within `Remote`.

In Java, interfaces may have `static` methods too, but in Kotlin we can't place `static` methods directly in interfaces, just like we can't place them directly in classes. Use companion objects to create `static` methods in interfaces. For example, let's create a `combine()` method that will bind two remotes so that the operations are performed on both remotes, one after the other:

```
companion object {
    fun combine(first: Remote, second: Remote): Remote = object: Remote {
        override fun up() {
          first.up()
          second.up()
        }

        override fun down() {
          first.down()
          second.down()
        }
    }
  }
```

remote.kts

This companion object goes directly into the `Remote` interface. To access the method in the companion object, use the `Remote` interface, like so:

```
val tv = TV()
val remote: Remote = TVRemote(tv)
remote.up()
remote.doubleUp()

val anotherTV = TV()
val combinedRemote = Remote.combine(remote, TVRemote(anotherTV))

combinedRemote.up()
println(tv.volume) //4
println(anotherTV.volume) //1
```

remote.kts

When implementing an interface, you must implement all the abstract methods. When implementing multiple interfaces, any methods that collide—that is, have the same name and signature—must be implemented in the class as well.

Creating abstract classes

Kotlin also supports abstract classes. The classes have to be marked `abstract` to be considered abstract, and abstract methods have to be marked `abstract` in abstract classes.

Here's an example of an abstract base class and a class that extends from it.

```
abstract class Musician(val name: String, val activeFrom: Int) {
  abstract fun instrumentType(): String
}

class Cellist(name: String, activeFrom: Int) : Musician(name, activeFrom) {
  override fun instrumentType() = "String"
}

val ma = Cellist("Yo-Yo Ma", 1961)

println(ma.name)
```

abc.kts

Since the `instrumentType()` method isn't implemented in the base class, it has to be marked as `abstract`. When overriding it in the derived class, the `override` keyword is required. The derived class `Cellist`'s primary constructor receives the two parameters and passes them to the base class. It may receive additional parameters not needed by the base, and those additional parameters may be declared using `val` or `var` to become properties in the derived class.

The main differences between an abstract class and an interface are these:

- The properties defined within interfaces don't have backing fields; they have to rely on abstract methods to get properties from implementing classes. On the other hand, properties within abstract classes can use backing fields.

- You may implement multiple interfaces but can extend from at most one class, abstract or not.

Interface or abstract class?

Should you create an interface or an abstract class instead? Interfaces can't contain fields, but a class may implement multiple interfaces. On the other hand, abstract base classes can have fields, but a class may only extend from at most one abstract class. So each has pros and cons. Let's discuss how to choose between them.

If you want to reuse state between multiple classes, then an abstract class is a good choice. You can implement the common state in the abstract class and have implementing classes override the methods, while reusing the state provided by the abstract class.

If you want multiple classes to abide by one or more contracts or specifications, but you want those classes to choose their own implementations, then interfaces are the better choice. If you choose this route, you may also move some common methods into the interfaces, while still letting the implementing classes choose how they implement the state.

In both modern Java and Kotlin, interfaces have a slight advantage over abstract classes. Interfaces have the ability to carry method implementations, but without state. A class may implement multiple interfaces. Where possible, prefer interfaces over abstract or base classes, as this offers more flexibility.

In the example for interfaces, the `TVRemote` relied on a public property `volume` of `TV` to implement the methods of the `Remote` interface.

QUIZ

1 Which of the of following options correctly builds an abstract class?

2 A class can be extended from how many abstract classes?

3

How can multiple inheritance be used in Kotlin?

Retake Quiz

In the next lesson, we’ll explore an alternative to this approach.