Nested and Inner Classes

We'll cover the following

- Pros and cons of interfaces
- The inner keyword
- Anonymous inner class

In the previous example, you may wonder why TV didn't directly implement the `Remote` interface instead of having a separate class `TVRemote` that implements the interface. Having a separate class like `TVRemote` instead of directly implementing the interface has a few pros and cons. Let's discuss the pros first, then the cons, and arrive at a solution to give us the best of both options.

Pros and cons of interfaces

The first benefit of having a `TVRemote` is that we may have multiple instances of `TVRemote` for a single instance of `TV`, much like how cars and garage doors have multiple remotes. This design capability can save relationships where each person can amiably control a `TV` instance without bothering someone else near a single remote to do it. Second, the instances of `TVRemote` may carry their internal state separate from any state contained in a `TV` instance. For example, a `TVRemote` instance used by one person may have a dim light on the remote turned on to help operate in the dark.

Implementing an interface in a separate class has drawbacks—the methods of `TVRemote` that implement the Remote interface have to use the public methods of the `TV`. If the `TV` implements the interface, then we don't have to rely on any public methods, and also the implementation can efficiently use internals visible only within the class. And, instead of passing an instance of `TV` to the constructor of `TVRemote`, if we implement the interface directly in `TV`, we don't need the extra references kept within instances of `TVRemote`.

The `inner` keyword

An alternative design option can help us keep the pros and at the same time avoid the cons. Using inner classes we can get the benefits offered by having the separate class, but without compromising efficiency. While the solution that follows may appear esoteric at first sight, this technique is used extensively in C#, Java, and Kotlin to implement iterators on collections.

In Kotlin a class may be nested—placed inside—another class. Unlike in Java, Kotlin nested classes can't access the private members of the nesting outer class. But if you mark the nested class with the `inner` keyword, then they turn into inner classes and the restriction goes away.

Let's move the `TVRemote` class from the previous example to be an inner class of `TV`:

```
class TV {
  private var volume = 0

  val remote: Remote
    get() = TVRemote()

  override fun toString(): String = "Volume: ${volume}"

  inner class TVRemote : Remote {
    override fun up() { volume++ }
    override fun down() { volume-- }

    override fun toString() = "Remote: ${this@TV.toString()}"
  }
}
```

nestedremote.kts

Unlike the `TV` class in the previous version, in this example the `volume` property of TV is `private` and can't be directly accessed from outside the instance of TV. `TVRemote`, though, is defined as an inner class, and its methods `up()` and `down()` access volume as if it were a property of `TVRemote`. The inner class has direct access to the members of the outer class, including `private members`. The TV class now provides a `public` property `remote` that returns an instance of `TVRemote`.

Let's use the modified version of TVRemote:

```
val tv = TV()
val remote = tv.remote

println("$tv") //Volume: 0
println("$tv") //Volume: 0
remote.up()
println("After increasing: $tv") //After increasing: Volume: 1

remote.doubleUp()
println("After doubleUp: $tv") //After doubleUp: Volume: 3
```

nestedremote.kts

A user of a TV instance can obtain a reference to a `Remote` for the TV instance using the `remote` property. The getter of this property has been designed so that each call to `remote` will result in a different instance, much like each call to the `iterate()` method on a collection will return a new iterator instance.

If a property or method in the inner class shadows a corresponding member in the outer class, you can access the member of the outer class from within a method of the inner using a special `this` expression. You can see this in the `toString()` method of `TVRemote`, shown here again for convenience:

```
override fun toString() = "Remote: ${this@TV.toString()}"
```

Read the syntax `this@TV` within the string template as "this of TV"—that is, `this` will refer to the `TVRemote` instance but `this@TV` refers to the instance of the outer class `TV`. Let's verify that the access to the outer instance from the inner instance worked:

```
// nestedremote.kts
println(remote) //Remote: Volume: 3
```

What if you want to access the `toString()` of Any—that is, the base class of TV from a method of `TVRemote`? Instead of asking for `this` of TV ask for `super` of TV—the base class of the outer class:

```
override fun toString() = "Remote: ${super@TV.toString()}"
```

Use `super@Outer` syntax sparingly; it's a design smell to bypass a class to get to its base class, defeating the intent of polymorphism and method overriding.

Anonymous inner class

If we need a special state within the nested or inner classes, we may place

If we need a special state within the nested or inner classes, we may place properties in them much like how we keep state in outer classes. Also, instead of

creating an inner class within a class, we may create an anonymous inner class within a method as well. Let’s turn the inner class from the previous example to an anonymous inner class within the `remote` property’s getter:

```
class TV {
    private var volume = 0
    val remote: Remote get() = object: Remote {
        override fun up() { volume++ }
        override fun down() { volume-- }
        override fun toString() = "Remote: ${this@TV.toString()}"
    }
    override fun toString(): String = "Volume: ${volume}"
}
```

The `inner` keyword isn’t used for anonymous inner classes. The anonymous instance implements the Remote interface and, of course, has no name. Other than those differences, the class isn’t any different from the `TVRemote` inner class it replaced.

We’ve looked at ways to implement interfaces and to create nested and inner classes.

In the next lesson, let’s look at extending classes.