Narrowing Access with Scope Control

We'll cover the following

- Narrowing access
- Removing automatic routing

You've seen how to create internal DSLs in Kotlin. Unlike external DSLs, internal DSLs have the benefit of riding on a host language, and we don't have to implement parsers. That removes a lot of effort on our part as DSL designers. However, anything that's possible in the host language, like property access or calls to arbitrary functions and methods in scope are all permitted within the DSL code. That freedom may be too much at times. Kotlin makes an effort to narrow that access a little with a scope control annotation.

Narrowing access

In spite of the facility Kotlin offers, there's no way to tell the compiler to reject calls to a top-level function or an access to a variable that's in lexical scope. But, with a special `@DSLMarker` annotation, you can tell the compiler to disallow implicit access to the members within the parent receiver of a nested lambda.

We discussed in Multiple Scopes with Receivers, how nested lambdas have two receivers, one that's the `this` context object of the executing lambda and the other that's the receiver of the parent lambda. When creating a DSL, we may want to limit a lambda to accessing only the immediate implicit receiver and disallow automatic access to the members of the parent receiver. Let's take a look at an example where placing that limitation will be useful.

Let's modify the code we created previously to create the XML output for languages and authors. Previously we nested a call to `element()` within another `element()` to create a child element. But what if we add a call to `root()` inside the lambda passed to `element()`, like so:

```
val xmlString = xml {
```

```
  root("languages") {
    langsAndAuthors.forEach { name, author ->
      element("language", "name" to name) {
        element("author") { text(author) }
        root("oops") {} //This makes no sense, but we get no errors
      }
    }
  }
}
println(xmlString)
```

The Kotlin compiler won't give any errors here because the second call to `root()` is legal. Whereas the second call to `element()` is on the `Node` instance, which is the implicit receiver `this`, the call to root is on the parent receiver; that is, the call to the second call `root()` is routed to the same instance that handled the first call to `root()`.

Of course, that doesn't make sense, and you can ask Kotlin to reject that call. To do that, you have to annotate either the classes or the base classes that partake in building the DSL with a custom DSL Marker annotation. Such a custom annotation itself is annotated with a `@DslMarker` annotation.

Let's modify the code to reject the second call to `root()` from within the lambda passed to `element()`.

First, create a DSL Marker annotation:

```
@DslMarker
annotation class XMLMarker
```

The annotation class `XMLMarker` is considered a DSL Marker annotation since it's annotated with `@DslMarker`. Any class annotated with a DSL Marker annotation—`XMLMarker` in this example—will signal to the Kotlin compiler to restrict calls without an object reference, such as calls without a `someobj`. prefix, to only the immediate receiver and not the parent receiver. In other words, without a DSL Marker annotation, a call like `foo()` will go to the parent receiver if it's not handled by the immediate receiver. However, with the marker annotation on the processing class, a call to `foo()` will fail compilation if it's not handled by the immediate receiver. Thus we can get early feedback from the compiler, to verify if the DSL conforms to the intended syntax.

Removing automatic routing

Let's now make use of the DSL Marker annotation `XMLMarker` by annotating both the `XMLBuilder` and Node classes:

```
@XMLMarker
class XMLBuilder {
    fun root(rootElementName: String, block: Node.() -> Unit): Node =
        Node(rootElementName).apply(block)
}

@XMLMarker
class Node(val name: String) {
    var attributes: Map<String, String> = mutableMapOf()
//...
```

After creating the `XMLMarker` annotation and annotating the classes `XMLBuilder` and `Node` like above, the following code will fail compilation:

```
val xmlString = xml {
  root("languages") {
    langsAndAuthors.forEach { name, author ->
      element("language", "name" to name) {
        element("author") { text(author) }
        root("oops") {} //ERROR: can't call root from here
      }
    }
  }
}
println(xmlString)
```

You see that the automatic routing of calls to the parent receiver has been removed. You may still make an explicit call to members of the parent receiver if you like, for example, with this syntax:

```
this@xml.root("oops") {}
```

When designing DSLs, most certainly use DSL Marker annotations. While the annotation won't help reject calls to top-level methods or access to variables and members in lexical scope, it will help reject automatic calls to methods that are part of the parent receivers. This is a useful feature to control scope a little within DSLs.

The next lesson concludes the discussion for this chapter.