Kotlin for Internal DSLs

We'll cover the following

Kotlin is a wonderful language for creating internal DSLs. Static typing generally poses some strong limitations on a language's ability to serve as a host for internal DSLs. Surprisingly, though, some of Kotlin's unique capabilities not found in other statically typed languages push the limits to make creating internal DSLs not only possible but a pleasant experience as well. Let's quickly review some of the capabilities of Kotlin that will serve as your allies when designing DSLs.

Optional semicolons

Kotlin doesn't insist on semicolons—see Semicolon Is Optional. That's a clear benefit for fluency. A semicolon disrupts flow and is noise in a lot of situations. Not having to use it is especially critical for creating expressive DSL syntax.

We can compare two expressions:

```
starts.at(14.30)
ends.by(15.20)
```

The first is less noisy than this one:

```
starts.at(14.30);
ends.by(15.20);
```

That `;` adds little value other than giving false comfort. Letting go of semicolons is

That ; adds little value other than giving false comfort. Letting go of semicolons is a good first step in creating DSLs. Is it really that big a deal? one may wonder. It's a small step, but when we mix this feature with other capabilities, the difference is huge. We don't have to look far; the Kotlin feature we'll see next illustrates that sentiment.

No dot and parenthesis with `infix`

Kotlins' support for infix notation using the `infix` keyword—Function Fluency with infix—is another welcome feature for DSLs. Have a look at the following code:

```
starts.at(14.30)
ends.by(15.20)
```

Using `infix`, we can write it as this:

```
starts at 14.30
ends by 15.20
```

The lack of dot and parenthesis, along with the absence of semicolon, makes the code almost read like English. If you're eager to see how to design for code like this, we'll cover it later in the chapter.

Get domain specific using extension functions

Even though Kotlin is statically typed, it permits the ability to perform compile-time function injection—see Injecting Using Extension Functions and Properties. As a result, you may design the ability for users of your library to code like the following, by injecting functions like `days()` into `Int`. Even though `Int` doesn't have a `days()` function built-in, once we inject that function into `Int`, we can write:

```
2.days(ago)
```

Additionally, extension functions may also be annotated with the `infix` keyword and then, instead of our previous example, one may write:

```
2 days ago
```

The fluency we can get for little effort in Kotlin is amazing. It's perfectly OK to take a minute to wipe away tears of joy before continuing on.

No parenthesis required to pass a lambda

If the type of the last parameter of a function is a lambda expression, then you may place the lambda expression argument outside the parenthesis—see Use Lambda as the Last Parameter. As a bonus, if a function takes only one lambda expression as its argument, then there's no need for parenthesis in the call. If the function is associated with a class, then the dot and parenthesis may be removed by using the `infix` keyword.

Thus, we could have written this:

```
"Release Planning".meeting({
  starts.at(14.30);
  ends.by(15.20);
})
```

But we can achieve the following fluency with Kotlin's capabilities:

```
"Release Planning" meeting {
  starts at 14.30
  ends by 15.20
}
```

That fluency lifts a heavy weight from our shoulders.

Implicit receiver rock DSL creation

One of the most significant feature of Kotlin for designing DSLs is the ability to pass implicit receivers to lambda expressions—see Implicit Receivers. This feature singlehandedly sets Kotlin apart from other statically typed languages for the ability to attain fluency in DSLs. And the receivers are a great way to pass a context object between layers of code in DSL.

Here's a little DSL to place an order for some office supplies:

```
placeOrder {
  an item "Pencil"
  an item "Eraser"
  complete {
    this with creditcard number "1234-5678-1234-5678"
  }

}
```

In this code, there's a context of an order and a context of a payment, but the code may need both to perform the payment transaction. That's not an issue thanks to implicit receivers.

In the execution of each lambda expression, there's an implicit receiver that carries the context—a thread of conversation—forward. This makes it easy to carry the state forward for processing between the layers of code without the need to pass parameters or maintain global state. That's a breath of fresh air.

A few more features to aid DSLs

We discussed some of the major features of Kotlin that influence the design of internal DSLs. When designing DSLs, you may occasionally find useful a few other minor capabilities.

The methods in the Any class that promote fluency, `also()`, `apply()`, `let()`, and `run()`—in Behavior of the Four Methods—will serve well to invoke lambdas and to set implicit receivers. These methods can reduce the code we need to implement the DSL.

Kotlin uses `this` to refer to the current object and `it` to refer to the single parameter of a lambda expression. If you're struggling to make the syntax fluent, sometimes, you may reach for these keywords and put them to work in your favor, like in this example:

```
please add it to this cart now
```

You may also reluctantly use operator overloading in some situations, but only where it's highly intuitive and makes the DSL very fluent—don't use it otherwise; see the caution expressed in Overloading Operators.

The next lesson explores some challenges faced while building fluency into the code.