Sample Chapter

Kotlin IN ACTION

Dmitry Jemerov
Svetlana Isakova

Foreword by Andrey Breslav

MANNING

Kotlin in Action

by Dmitry Jemerov and Svetlana Isakova

Sample Chapter 11

brief contents

11 DSL construction

This chapter covers

- Building domain-specific languages
- Using lambdas with receivers
- Applying the `invoke` convention
- Examples of existing Kotlin DSLs

In this chapter, we'll discuss how you can design expressive and idiomatic APIs for your Kotlin classes through the use of *domain-specific languages* (DSLs). We'll explore the differences between traditional and DSL-style APIs, and you'll see how DSL-style APIs can be applied to a wide variety of practical problems in areas as diverse as database access, HTML generation, testing, writing build scripts, defining Android UI layouts, and many others.

Kotlin DSL design relies on many language features, two of which we haven't yet fully explored. One of them you saw briefly in chapter 5: lambdas with receivers, which let you create a DSL structure by changing the name-resolution rules in code blocks. The other is new: the `invoke` convention, which enables more flexibility in combining lambdas and property assignments in DSL code. We'll study those features in detail in this chapter.

11.1 From APIs to DSLs

Before we dive into the discussion of DSLs, let's get a better understanding of the problem we're trying to solve. Ultimately, the goal is to achieve the best possible code readability and maintainability. To reach that goal, it's not enough to focus on individual classes. Most of the code in a class interacts with other classes, so we need to look at the interfaces through which these interactions happen—in other words, the APIs of the classes.

It's important to remember that the challenge of building good APIs isn't reserved to library authors; rather, it's something every developer has to do. Just as a library provides a programming interface for using it, every class in an application provides possibilities for other classes to interact with it. Ensuring that those interactions are easy to understand and can be expressed clearly is essential for keeping a project maintainable.

Over the course of this book, you've seen many examples of Kotlin features that allow you to build *clean APIs* for classes. What do we mean when we say an API is clean? Two things:

- It needs to be clear to readers what's going on in the code. This can be achieved with a good choice of names and concepts, which is important in any language.
- The code needs to look clean, with minimal ceremony and no unnecessary syntax. Achieving this is the main focus of this chapter. A clean API can even be indistinguishable from a built-in feature of a language.

Examples of Kotlin features that enable you to build clean APIs include extension functions, infix calls, lambda syntax shortcuts, and operator overloading. Table 11.1 shows how these features help reduce the amount of syntactic noise in the code.

Table 11.1 Kotlin support for clean syntax

Regular syntax	Clean syntax	Feature in use
`StringUtil.capitalize(s)`	`s.capitalize()`	Extension function
`1.to("one")`	`1 to "one"`	Infix call
`set.add(2)`	`set += 2`	Operator overloading
`map.get("key")`	`map["key"]`	Convention for the `get` method
`file.use({ f -> f.read() } )`	`file.use { it.read() }`	Lambda outside of parentheses

`sb.append("yes")`
`sb.append("no")`
	`with (sb) {`
	`  append("yes")`
	`  append("no")`
	`}`
		Lambda with a receiver

In this chapter, we'll take a step beyond clean APIs and look at Kotlin's support for constructing DSLs. Kotlin's DSLs build on the clean-syntax features and extend them with the ability to create *structure* out of multiple method calls. As a result, DSLs can be even more expressive and pleasant to work with than APIs constructed out of individual method calls.

Just like other features of the language, Kotlin DSLs are *fully statically typed*. This means all the advantages of static typing, such as compile-time error detection and better IDE support, remain in effect when you use DSL patterns for your APIs.

As a quick taste, here are a couple of examples that show what Kotlin DSLs can do. This expression goes back in time and returns the previous day (all right, just the previous date):

```
val yesterday = 1.days.ago
```

and this function generates an HTML table:

```
fun createSimpleTable() = createHTML().
    table {
        tr {
            td { +"cell" }
        }
    }
```

Over the course of the chapter, you'll learn how these examples are constructed. But before we begin a detailed discussion, let's look at what DSLs are.

11.1.1 The concept of domain-specific languages

The general idea of a DSL has existed for almost as long as the idea of a programming language. We make a distinction between a *general-purpose programming language*, with a set of capabilities complete enough to solve essentially any problem that can be solved with a computer; and a *domain-specific language*, which focuses on a specific task, or *domain*, and forgoes the functionality that's irrelevant for that domain.

The most common DSLs that you're no doubt familiar with are SQL and regular expressions. They're great for solving the specific tasks of manipulating databases and text strings, respectively, but you can't use them to develop an entire application. (At least, we hope you don't. The idea of an entire application built in the regular-expression language makes us shudder.)

Note how these languages can effectively accomplish their goal by reducing the set of functionality they offer. When you need to execute an SQL statement, you don't start by declaring a class or a function. Instead, the first keyword in every SQL statement indicates the type of operation you need to perform, and each type of operation has its own distinct syntax and set of keywords specific to the task at hand. With the regular-expression language, there's even less syntax: the program directly describes the text to be matched, using compact punctuation syntax to specify how the text can vary. Through such a compact syntax, a DSL can express a domain-specific operation much more concisely than an equivalent piece of code in a general-purpose language.

Another important point is that DSLs tend to be declarative, as opposed to general-purpose languages, most of which are imperative. Whereas an *imperative language* describes the exact sequence of steps required to perform an operation, a *declarative language* describes the desired result and leaves the execution details to the engine that interprets it. This often makes the execution more efficient, because the necessary optimizations are implemented only once in the execution engine; on the other hand, an imperative approach requires every implementation of the operation to be optimized independently.

As a counterweight to all of those benefits, DSLs of this type have one disadvantage: it can be difficult to combine them with a host application in a general-purpose language. They have their own syntax that can't be directly embedded into programs in a different language. Therefore, to invoke a program written in a DSL, you need to either store it in a separate file or embed it in a string literal. That makes it non-trivial to validate the correct interaction of the DSL with the host language at compile time, to debug the DSL program, and to provide IDE code assistance when writing it. Also, the separate syntax requires separate learning and often makes code harder to read.

To solve that issue while preserving most of the other benefits of DSLs, the concept of *internal DSLs* has recently gained popularity. Let's see what this is about.

11.1.2 Internal DSLs

As opposed to *external DSLs*, which have their own independent syntax, *internal DSLs* are part of programs written in a general-purpose language, using exactly the same syntax. In effect, an internal DSL isn't a fully separate language, but rather a particular way of using the main language while retaining the key advantages of DSLs with an independent syntax.

To compare the two approaches, let's see how the same task can be accomplished with an external and an internal DSL. Imagine that you have two database tables, `Customer` and `Country`, and each `Customer` entry has a reference to the country the customer lives in. The task is to query the database and find the country where the majority of customers live. The external DSL you're going to use is SQL; the internal one is provided by the Exposed framework (https://github.com/JetBrains/Exposed), which is a Kotlin framework for database access. Here's how you do this with SQL:

```
SELECT Country.name, COUNT(Customer.id)
    FROM Country
    JOIN Customer
      ON Country.id = Customer.country_id
GROUP BY Country.name
ORDER BY COUNT(Customer.id) DESC
   LIMIT 1
```

Writing the code in SQL directly may not be convenient: you have to provide a means for interaction between your main application language (Kotlin in this case) and the query language. Usually, the best you can do is put the SQL into a string literal and hope that your IDE will help you write and verify it.

As a comparison, here's the same query built with Kotlin and Exposed:

```
(Country join Customer)
    .slice(Country.name, Count(Customer.id))
    .selectAll()
    .groupBy(Country.name)
    .orderBy(Count(Customer.id), isAsc = false)
    .limit(1)
```

You can see the similarity between the two versions. In fact, executing the second version generates and runs exactly the same SQL query as the one written manually. But the second version is regular Kotlin code, and `selectAll`, `groupBy`, `orderBy`, and others are regular Kotlin methods. Moreover, you don't need to spend any effort on converting data from SQL query result sets to Kotlin objects—the query-execution results are delivered directly as native Kotlin objects. Thus we call this an internal DSL: the code intended to accomplish a specific task (building SQL queries) is implemented as a library in a general-purpose language (Kotlin).

11.1.3 Structure of DSLs

Generally speaking, there's no well-defined boundary between a DSL and a regular API; often the criterion is as subjective as "I know it's a DSL when I see it." DSLs often rely on language features that are broadly used in other contexts too, such as infix calls and operator overloading. But one trait comes up often in DSLs and usually doesn't exist in other APIs: *structure*, or *grammar*.

A typical library consists of many methods, and the client uses the library by calling the methods one by one. There's no inherent structure in the sequence of calls, and no context is maintained between one call and the next. Such an API is sometimes called a *command-query API*. As a contrast, the method calls in a DSL exist in a larger structure, defined by the *grammar* of the DSL. In a Kotlin DSL, structure is most commonly created through the nesting of lambdas or through chained method calls. You can clearly see this in the previous SQL example: executing a query requires a combination of method calls describing the different aspects of the required result set, and the combined query is much easier to read than a single method call taking all the arguments you're passing to the query.

This grammar is what allows us to call an internal DSL a *language*. In a natural language such as English, sentences are constructed out of words, and the rules of grammar govern how those words can be combined with one another. Similarly, in a DSL, a single operation can be composed out of multiple function calls, and the type checker ensures that the calls are combined in a meaningful way. In effect, the function names usually act as verbs (`groupBy`, `orderBy`), and their arguments fulfill the role of nouns (`Country.name`).

One benefit of the DSL structure is that it allows you to reuse the same context between multiple function calls, rather than repeat it in every call. This is illustrated by the following example, showing the Kotlin DSL for describing dependencies in Gradle build scripts (https://github.com/gradle/gradle-script-kotlin):

```
dependencies {
    compile("junit:junit:4.11")
    compile("com.google.inject:guice:4.1.0")
}
```

Structure through lambda nesting

In contrast, here's the same operation performed through a regular command-query API. Note that there's much more repetition in the code:

```
project.dependencies.add("compile", "junit:junit:4.11")
project.dependencies.add("compile", "com.google.inject:guice:4.1.0")
```

Chained method calls are another way to create structure in DSLs. For example, they're commonly used in test frameworks to split an assertion into multiple method calls. Such assertions can be much easier to read, especially if you can apply the infix call syntax. The following example comes from kotlintest (https://github.com/kotlintest/kotlintest), a third-party test framework for Kotlin that we'll discuss in more detail in section 11.4.1:

Structure through chained method calls

```
str should startWith("kot")
```

Note how the same example expressed through regular JUnit APIs is noisier and not as readable:

```
assertTrue(str.startsWith("kot"))
```

Now let's look at an example of an internal DSL in more detail.

11.1.4 Building HTML with an internal DSL

One of the teasers at the beginning of this chapter was a DSL for building HTML pages. In this section, we'll discuss it in more detail. The API used here comes from the kotlinx.html library (https://github.com/Kotlin/kotlinx.html). Here's a small snippet that creates a table with a single cell:

```
fun createSimpleTable() = createHTML().
    table {
        tr {
            td { +"cell" }
        }
    }
```

It's clear what HTML corresponds to the previous structure:

```
<table>
  <tr>
    <td>cell</td>
  </tr>
</table>
```

The `createSimpleTable` function returns a string containing this HTML fragment.

Why would you want to build this HTML with Kotlin code, rather than write it as text? First, the Kotlin version is type-safe: you can use the `td` tag only in `tr`; otherwise, this code won't compile. What's more important is that it's regular code, and you can use any language construct in it. That means you can generate table cells dynamically (for instance, corresponding to elements in a map) in the same place when you define a table:

```
fun createAnotherTable() = createHTML().table {
    val numbers = mapOf(1 to "one", 2 to "two")
    for ((num, string) in numbers) {
        tr {
            td { +"$num" }
            td { +string }
        }
    }
}
```

The generated HTML contains the desired data:

```
<table>
  <tr>
    <td>1</td>
    <td>one</td>
  </tr>
  <tr>
    <td>2</td>
    <td>two</td>
  </tr>
</table>
```

HTML is a canonical example of a markup language, which makes it perfect for illustrating the concept; but you can use the same approach for any languages with a similar structure, such as XML. Shortly we'll discuss how such code works in Kotlin.

Now that you know what a DSL is and why you might want to build one, let's see how Kotlin helps you do that. First we'll take a more in-depth look at *lambdas with receivers*: the key feature that helps establish the grammar of DSLs.

11.2 Building structured APIs: lambdas with receivers in DSLs

Lambdas with receivers are a powerful Kotlin feature that allows you to build APIs with a structure. As we already discussed, having structure is one of the key traits distinguishing DSLs from regular APIs. Let's examine this feature in detail and look at some DSLs that use it.

11.2.1 Lambdas with receivers and extension function types

You had a brief encounter with the idea of lambdas with receivers in section 5.5, where we introduced the `buildString`, `with`, and `apply` standard library functions. Now let's look at how they're implemented, using the `buildString` function as an example. This function allows you to construct a string from several pieces of content added to an intermediate `StringBuilder`.

To begin the discussion, let's define the `buildString` function so that it takes a regular lambda as an argument. You saw how to do this in chapter 8, so this should be familiar material.

Listing 11.1 Defining `buildString()` that takes a lambda as an argument

```
fun buildString(
        builderAction: (StringBuilder) -> Unit      <-- Declares a parameter of a function type
): String {
    val sb = StringBuilder()
    builderAction(sb)                               <-- Passes a StringBuilder as an argument to the lambda
    return sb.toString()
}

>>> val s = buildString {
...     it.append("Hello, ")                        <-- Uses "it" to refer to the StringBuilder instance
...     it.append("World!")
... }
>>> println(s)
Hello, World!
```

This code is easy to understand, but it looks less easy to use than we'd prefer. Note that you have to use `it` in the body of the lambda to refer to the `StringBuilder` instance (you could define your own parameter name instead of `it`, but it still has to be explicit). The main purpose of the lambda is to fill the `StringBuilder` with text, so you want to get rid of the repeated `it.` prefixes and invoke the `StringBuilder` methods directly, replacing `it.append` with `append`.

To do so, you need to convert the lambda into a *lambda with a receiver*. In effect, you can give one of the parameters of the lambda the special status of a *receiver*, letting you refer to its members directly without any qualifier. The following listing shows how you do that.

Listing 11.2 Redefining `buildString()` to take a lambda with a receiver

```
fun buildString(
        builderAction: StringBuilder.() -> Unit     <-- Declares a parameter of a function type with a receiver
) : String {
    val sb = StringBuilder()
    sb.builderAction()                              <-- Passes a StringBuilder as a receiver to the lambda
    return sb.toString()
}

>>> val s = buildString {
...     this.append("Hello, ")                      <-- The "this" keyword refers to the StringBuilder instance.
...     append("World!")                            <-- Alternatively, you can omit "this" and refer to StringBuilder implicitly.
... }
>>> println(s)
Hello, World!
```

Pay attention to the differences between listing 11.1 and listing 11.2. First, consider how the way you use `buildString` has improved. Now you pass a lambda with a receiver as an argument, so you can get rid of `it` in the body of the lambda. You replace the calls to `it.append()` with `append()`. The full form is `this.append()`, but as with regular members of a class, an explicit `this` is normally used only for disambiguation.

Next, let's discuss how the declaration of the `buildString` function has changed. You use an *extension function type* instead of a regular function type to declare the parameter type. When you declare an extension function type, you effectively pull one of the function type parameters out of the parentheses and put it in front, separated from the rest of the types with a dot. In listing 11.2, you replace `(StringBuilder) -> Unit` with `StringBuilder.() -> Unit`. This special type is called the *receiver type*, and the value of that type passed to the lambda becomes the *receiver object*. Figure 11.1 shows a more complex extension function type declaration.

Receiver type **Parameter types** **Return type**

```
String.(Int, Int) -> Unit
```

Figure 11.1 An extension function type with receiver type `String` and two parameters of type `Int`, returning `Unit`

Why an *extension* function type? The idea of accessing members of an external type without an explicit qualifier may remind you of extension functions, which allow you to define your own methods for classes defined elsewhere in the code. Both extension functions and lambdas with receivers have a *receiver object*, which has to be provided when the function is called and is available in its body. In effect, an extension function type describes a block of code that can be called as an extension function.

The way you invoke the variable also changes when you convert it from a regular function type to an extension function type. Instead of passing the object as an argument, you invoke the lambda variable as if it were an extension function. When you have a regular lambda, you pass a `StringBuilder` instance as an argument to it using the following syntax: `builderAction(sb)`. When you change it to a lambda with a receiver, the code becomes `sb.builderAction()`. To reiterate, `builderAction` here isn't a method declared on the `StringBuilder` class; it's a parameter of a function type that you call using the same syntax you use to call extension functions.

Figure 11.2 shows the correspondence between an argument and a parameter of the `buildString` function. It also illustrates the receiver on which the lambda body will be called.

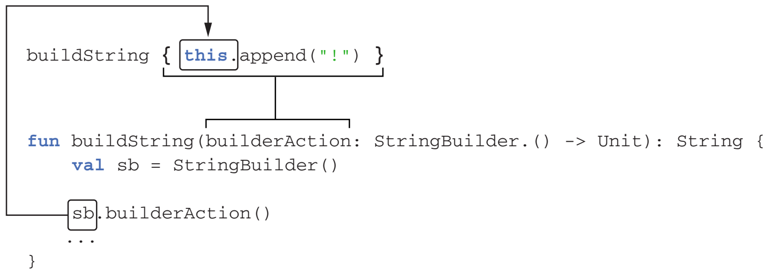

Figure 11.2 The argument of the `buildString` function (lambda with a receiver) corresponds to the parameter of the extension function type (`builderAction`). The receiver (`sb`) becomes an implicit receiver (`this`) when the lambda body is invoked.

You can also declare a variable of an extension function type, as shown in the following listing. Once you do that, you can either invoke it as an extension function or pass it as an argument to a function that expects a lambda with a receiver.

Listing 11.3 Storing a lambda with a receiver in a variable

```
val appendExcl : StringBuilder.() -> Unit =
        { this.append("!") }

>>> val stringBuilder = StringBuilder("Hi")
>>> stringBuilder.appendExcl()
>>> println(stringBuilder)
Hi!

>>> println(buildString(appendExcl))
!
```

appendExcl is a value of an extension function type.

You can call appendExcl as an extension function.

You can also pass appendExcl as an argument.

Note that a lambda with a receiver looks exactly the same as a regular lambda in the source code. To see whether a lambda has a receiver, you need to look at the function to which the lambda is passed: its signature will tell you whether the lambda has a receiver and, if it does, what its type is. For example, you can look at the declaration of `buildString` or look up its documentation in your IDE, see that it takes a lambda of type `StringBuilder.() -> Unit`, and conclude from this that in the body of the lambda, you can invoke `StringBuilder` methods without a qualifier.

The implementation of `buildString` in the standard library is shorter than in listing 11.2. Instead of calling `builderAction` explicitly, it is passed as an argument to the `apply` function (which you saw in section 5.5). This allows you to collapse the function into a single line:

```
fun buildString(builderAction: StringBuilder.() -> Unit): String =
        StringBuilder().apply(builderAction).toString()
```

The `apply` function effectively takes the object on which it was called (in this case, a new `StringBuilder` instance) and uses it as an implicit receiver to call the function or lambda specified as argument (`builderAction` in the example). You've also seen another useful library function previously: `with`. Let's study their implementations:

```
inline fun <T> T.apply(block: T.() -> Unit): T {
    block()
    return this
}
```

Equivalent to this.block(); invokes the lambda with the receiver of "apply" as the receiver object

Returns the receiver

```
inline fun <T, R> with(receiver: T, block: T.() -> R): R =
    receiver.block()
```

Returns the result of calling the lambda

Basically, all `apply` and `with` do is invoke the argument of an extension function type on the provided receiver. The `apply` function is declared as an extension to that receiver, whereas `with` takes it as a first argument. Also, `apply` returns the receiver itself, but `with` returns the result of calling the lambda.

If you don't care about the result, these functions are interchangeable:

```
>>> val map = mutableMapOf(1 to "one")
>>> map.apply { this[2] = "two"}
>>> with (map) { this[3] = "three" }
>>> println(map)
{1=one, 2=two, 3=three}
```

The `with` and `apply` functions are used frequently in Kotlin, and we hope you've already appreciated their conciseness in your own code.

We've reviewed lambdas with receivers and talked about extension function types. Now it's time to see how these concepts are used in the DSL context.

11.2.2 Using lambdas with receivers in HTML builders

A Kotlin DSL for HTML is usually called an *HTML builder*, and it represents a more general concept of *type-safe builders*. Initially, the concept of builders gained popularity in the Groovy community (www.groovy-lang.org/dsls.html#_builders). Builders provide a way to create an object hierarchy in a declarative way, which is convenient for generating XML or laying out UI components.

Kotlin uses the same idea, but in Kotlin builders are type-safe. That makes them more convenient to use, safe, and in a sense more attractive than Groovy's dynamic builders. Let's look in detail at how HTML builders work in Kotlin.

Listing 11.4 Producing a simple HTML table with a Kotlin HTML builder

```
fun createSimpleTable() = createHTML().
    table {
        tr {
            td { +"cell" }
        }
    }
```

This is regular Kotlin code, not a special template language or anything like that: `table`, `tr`, and `td` are just functions. Each of them is a higher-order function, taking a lambda with a receiver as an argument.

The remarkable thing here is that those lambdas *change the name-resolution rules*. In the lambda passed to the `table` function, you can use the `tr` function to create the `<tr>` HTML tag. Outside of that lambda, the `tr` function would be unresolved. In the same way, the `td` function is only accessible in `tr`. (Note how the design of the API forces you to follow the grammar of the HTML language.)

The name-resolution context in each block is defined by the receiver type of each lambda. The lambda passed to `table` has a receiver of a special type `TABLE`, which defines the `tr` method. Similarly, the `tr` function expects an extension lambda to `TR`. The following listing is a greatly simplified view of the declarations of these classes and methods.

Listing 11.5 Declaring tag classes for the HTML builder

```
open class Tag

class TABLE : Tag {
    fun tr(init : TR.() -> Unit)
}
class TR : Tag {
    fun td(init : TD.() -> Unit)
}
class TD : Tag
```

The tr function expects a lambda with a receiver of type TR.

The td function expects a lambda with a receiver of type TD.

`TABLE`, `TR`, and `TD` are utility classes that shouldn't appear explicitly in the code, and that's why they're named in capital letters. They all extend the `Tag` superclass. Each class defines methods for creating tags allowed in it: the `TABLE` class defines the `tr` method, among others, whereas the `TR` class defines the `td` method.

Note the types of the `init` parameters of the `tr` and `td` functions: they're extension function types `TR.() -> Unit` and `TD.() -> Unit`. They determine the types of receivers in the argument lambdas: `TR` and `TD`, respectively.

To make it clearer what happens here, you can rewrite listing 11.4, making all receivers explicit. As a reminder, you can access the receiver of the lambda that's the argument of the `foo` function as `this@foo`.

Listing 11.6 Making receivers of HTML builder calls explicit

```
fun createSimpleTable() = createHTML().
    table {
        (this@table).tr {
            (this@tr).td {
                +"cell"
            }
        }
    }
```

this@table has type TABLE.

this@tr has type TR.

The implicit receiver this@td of type TD is available here.

If you tried to use regular lambdas instead of lambdas with receivers for builders, the syntax would become as unreadable as in this example: you'd have to use the `it` reference to invoke the tag-creation methods or assign a new parameter name for every lambda. Being able to make the receiver implicit and hide the `this` reference makes the syntax of builders nice and similar to the original HTML.

Note that if one lambda with a receiver is placed in the other one, as in listing 11.6, the receiver defined in the outer lambda remains available in the nested lambda. For instance, in the lambda that's the argument of the `td` function, all three receivers (`this@table`, `this@tr`, `this@td`) are available. But starting from Kotlin 1.1, you'll be able to use the `@DslMarker` annotation to constrain the availability of outer receivers in lambdas.

We've explained how the syntax of HTML builders is based on the concept of lambdas with receivers. Next, let's discuss how the desired HTML is generated.

Listing 11.6 uses functions defined in the kotlinx.html library. Now you'll implement a much simpler version of an HTML builder library: you'll extend the declarations of the `TABLE`, `TR`, and `TD` tags and add support for generating the resulting HTML. As the entry point for this simplified version, a top-level `table` function creates a fragment of HTML with `<table>` as a top tag.

Listing 11.7 Generating HTML to a string

```
fun createTable() =
    table {
        tr {
            td {
            }
        }
    }

>>> println(createTable())
<table><tr><td></td></tr></table>
```

The `table` function creates a new instance of the `TABLE` tag, initializes it (calls the function passed as the `init` parameter on it), and returns it:

```
fun table(init: TABLE.() -> Unit) = TABLE().apply(init)
```

In `createTable`, the lambda passed as an argument to the `table` function contains the invocation of the `tr` function. The call can be rewritten to make everything as explicit as possible: `table(init = { this.tr { … } })`. The `tr` function will be called on the created `TABLE` instance, as if you'd written `TABLE().tr { … }`.

In this toy example, `<table>` is a top-level tag, and other tags are nested into it. Each tag stores a list of references to its children. Therefore, the `tr` function should not only initialize the new instance of the `TR` tag but also add it to the list of children of the outer tag.

Listing 11.8 Defining a tag builder function

```
fun tr(init: TR.() -> Unit) {
    val tr = TR()
    tr.init()
    children.add(tr)
}
```

This logic of initializing a given tag and adding it to the children of the outer tag is common for all tags, so you can extract it as a `doInit` member of the `Tag` superclass. The `doInit` function is responsible for two things: storing the reference to the child tag and calling the lambda passed as an argument. The different tags then just call it: for instance, the `tr` function creates a new instance of the `TR` class and then passes it to the `doInit` function along with the `init` lambda argument: `doInit(TR(), init)`. The following listing is the full example that shows how the desired HTML is generated.

Listing 11.9 A full implementation of a simple HTML builder

```
open class Tag(val name: String) {
    private val children = mutableListOf<Tag>()

    protected fun <T : Tag> doInit(child: T, init: T.() -> Unit) {
        child.init()
        children.add(child)
    }

    override fun toString() =
        "<$name>${children.joinToString("")}</$name>"
}

fun table(init: TABLE.() -> Unit) = TABLE().apply(init)

class TABLE : Tag("table") {
    fun tr(init: TR.() -> Unit) = doInit(TR(), init)
}
class TR : Tag("tr") {
    fun td(init: TD.() -> Unit) = doInit(TD(), init)
}
class TD : Tag("td")

fun createTable() =
    table {
        tr {
            td {
            }
        }
    }
>>> println(createTable())
<table><tr><td></td></tr></table>
```

Stores all nested tags

Initializes the child tag

Stores a reference to the child tag

Returns the resulting HTML as String

Creates, initializes, and adds to the children of TABLE a new instance of the TR tag

Adds a new instance of the TD tag to the children of TR

Every tag stores a list of nested tags and renders itself accordingly: it renders its name and all the nested tags recursively. Text inside tags and tag attributes aren't supported here; for the full implementation, you can browse the aforementioned kotlinx.html library.

Note that tag-creation functions add the corresponding tag to the parent's list of children on their own. That lets you generate tags dynamically.

Listing 11.10 Generating tags dynamically with an HTML builder

```
fun createAnotherTable() = table {
    for (i in 1..2) {
        tr {                          // Each call to "tr" creates a
            td {                      // new TR tag and adds it to
            }                         // the children of TABLE.
        }
    }
}
>>> println(createAnotherTable())
<table><tr><td></td></tr><tr><td></td></tr></table>
```

As you've seen, lambdas with receivers are a great tool for building DSLs. Because you can change the name-resolution context in a code block, they let you create *structure* in your API, which is one of the key traits that distinguishes DSLs from flat sequences of method calls. Now let's discuss the benefits of integrating this DSL into a statically typed programming language.

11.2.3 Kotlin builders: enabling abstraction and reuse

When you write regular code in a program, you have a lot of tools to avoid duplication and to make the code look nicer. Among other things, you can extract repetitive code into new functions and give them self-explanatory names. That may not be as easy or even possible with SQL or HTML. But using internal DSLs in Kotlin to accomplish the same tasks gives you a way to abstract repeated chunks of code into new functions and reuse them.

Let's look at an example from the Bootstrap library (http://getbootstrap.com), a popular HTML, CSS, and JS framework for developing responsive, mobile-first projects on the web. We'll consider a specific example: adding drop-down lists to an application. To add such a list directly to an HTML page, you can copy the necessary snippet and paste it in the required place, under the button or other element that shows the list. You only need to add the necessary references and their titles for the drop-down menu. The initial HTML code (a bit simplified to avoid too many style attributes) looks like this.

Listing 11.11 Building a drop-down menu in HTML using Bootstrap

```
<div class="dropdown">
  <button class="btn dropdown-toggle">
    Dropdown
```

```
    <span class="caret"></span>
  </button>
  <ul class="dropdown-menu">
    <li><a href="#">Action</a></li>
    <li><a href="#">Another action</a></li>
    <li role="separator" class="divider"></li>
    <li class="dropdown-header">Header</li>
    <li><a href="#">Separated link</a></li>
  </ul>
</div>
```

In Kotlin with kotlinx.html, you can use the functions `div`, `button`, `ul`, `li`, and so on to replicate the same structure.

Listing 11.12 Building a drop-down menu using a Kotlin HTML builder

```
fun buildDropdown() = createHTML().div(classes = "dropdown") {
    button(classes = "btn dropdown-toggle") {
        +"Dropdown"
        span(classes = "caret")
    }
    ul(classes = "dropdown-menu") {
        li { a("#") { +"Action" } }
        li { a("#") { +"Another action" } }
        li { role = "separator"; classes = setOf("divider") }
        li { classes = setOf("dropdown-header"); +"Header" }
        li { a("#") { +"Separated link" } }
    }
}
```

But you can do better. Because `div`, `button`, and so on are regular functions, you can extract the repetitive logic into separate functions, improving the readability of the code. The result may look as follows.

Listing 11.13 Building a drop-down menu with helper functions

```
fun dropdownExample() = createHTML().dropdown {
    dropdownButton { +"Dropdown" }
    dropdownMenu {
        item("#", "Action")
        item("#", "Another action")
        divider()
        dropdownHeader("Header")
        item("#", "Separated link")
    }
}
```

Now the unnecessary details are hidden, and the code looks much nicer. Let's discuss how this trick is implemented, starting with the `item` function. This function has two parameters: the reference and the name of the corresponding menu item. The function code should add a new list item: `li { a(href) { +name } }`. The only question that remains is, how can you call `li` in the body of the function? Should it be an extension? You can indeed make it an extension to the `UL` class, because the `li` function is itself an extension to `UL`. In listing 11.13, `item` is called on an implicit `this` of type `UL`:

```
fun UL.item(href: String, name: String) = li { a(href) { +name } }
```

After you define the `item` function, you can call it in any `UL` tag, and it will add an instance of a `LI` tag. Having extracted `item`, you can change the original version to the following without changing the generated HTML code.

Listing 11.14 Using the `item` function for drop-down menu construction

```
ul {
    classes = setOf("dropdown-menu")
    item("#", "Action")                    You can use the "item"
    item("#", "Another action")            function instead of "li" here.
    li { role = "separator"; classes = setOf("divider") }
    li { classes = setOf("dropdown-header"); +"Header" }
    item("#", "Separated link")
}
```

The other extension functions defined on `UL` are added in a similar way, allowing you to replace the remaining `li` tags.

```
fun UL.divider() = li { role = "separator"; classes = setOf("divider") }

fun UL.dropdownHeader(text: String) =
    li { classes = setOf("dropdown-header"); +text }
```

Now let's see how the `dropdownMenu` function is implemented. It creates a `ul` tag with the specified `dropdown-menu` class and takes a lambda with a receiver as an argument that's used to fill the tag with content.

```
dropdownMenu {
    item("#", "Action")
    ...
}
```

You replace the `ul { ... }` block with the invocation of `dropdownMenu { ... }`, so the receiver in the lambda can stay the same. The `dropdownMenu` function can take an extension lambda to `UL` as an argument, which allows you to call functions such as `UL.item` as you did before. Here's how the function is declared:

```
fun DIV.dropdownMenu(block: UL.() -> Unit) = ul("dropdown-menu", block)
```

The `dropdownButton` function is implemented in a similar way. We omit it here, but you can find the full implementation in the samples for the kotlinx.html library.

Last, let's look at the `dropdown` function. This one is less trivial, because it can be called on any tag: the drop-down menu can be put anywhere in the code.

Listing 11.15 The top-level function for building a drop-down menu

```
fun StringBuilder.dropdown(
        block: DIV.() -> Unit
): String = div("dropdown", block)
```

This is a simplified version that you can use if you want to print your HTML to a string. The full implementation in kotlinx.html uses an abstract `TagConsumer` class as the receiver and thus supports different destinations for the resulting HTML.

This example illustrates how the means of abstraction and reuse can help improve your code and make it easier to understand. Now let's look at one more tool that can help you support more flexible structures in your DSLs: the `invoke` convention.

11.3 More flexible block nesting with the "invoke" convention

The `invoke` convention allows you to call objects of custom types as functions. You've already seen that objects of function types can be called as functions; with the `invoke` convention, you can define your own objects that support the same syntax.

Note that this isn't a feature for everyday use, because it can be used to write hard-to-understand code, such as `1()`. But it's sometimes very useful in DSLs. We'll show you why, but first let's discuss the convention itself.

11.3.1 The "invoke" convention: objects callable as functions

In chapter 7, we discussed in detail Kotlin's concept of *conventions*: specially named functions that are called not through the regular method-call syntax but using different, more concise notations. As a reminder, one of the conventions we discussed was `get`, which allows you to access an object using the index operator. For a variable `foo` of type `Foo`, a call to `foo[bar]` is translated into `foo.get(bar)`, provided the corresponding `get` function is defined as a member in the `Foo` class or as an extension function to `Foo`.

In effect, the `invoke` convention does the same thing, except that the brackets are replaced with parentheses. A class for which the `invoke` method with an `operator` modifier is defined can be called as a function. Here's an example of how this works.

Listing 11.16 Defining an `invoke` method in a class

```
class Greeter(val greeting: String) {
    operator fun invoke(name: String) {        // Defines the "invoke" method on Greeter
        println("$greeting, $name!")
    }
}

>>> val bavarianGreeter = Greeter("Servus")
>>> bavarianGreeter("Dmitry")                   // Calls the Greeter instance as a function
Servus, Dmitry!
```

This code defines the `invoke` method in `Greeter`, which allows you to call instances of `Greeter` as if they were functions. Under the hood, the expression `bavarianGreeter("Dmitry")` is compiled to the method call `bavarianGreeter.invoke("Dmitry")`. There's no mystery here. It works like a regular convention: it provides a way to replace a verbose expression with a more concise, clearer one.

The `invoke` method isn't restricted to any specific signature. You can define it with any number of parameters and with any return type, or even define multiple overloads of `invoke` with different parameter types. When you call the instance of the class as a function, you can use all of those signatures for the call. Let's look at the practical situations where this convention is used, first in a regular programming context and then in a DSL.

11.3.2 The "invoke" convention and functional types

You may remember seeing `invoke` earlier in the book. In section 8.1.2 we discussed that you can call a variable of a nullable function type as `lambda?.invoke()`, using the safe-call syntax with the `invoke` method name.

Now that you know about the `invoke` convention, it should be clear that the way you normally invoke a lambda (by putting parentheses after it: `lambda()`) is nothing but an application of this convention. Lambdas, unless inlined, are compiled into classes that implement functional interfaces (`Function1` and so on), and those interfaces define the `invoke` method with the corresponding number of parameters:

```
interface Function2<in P1, in P2, out R> {
    operator fun invoke(p1: P1, p2: P2): R
}
```

This interface denotes a function that takes exactly two arguments.

When you invoke a lambda as a function, the operation is translated into a call of the `invoke` method, thanks to the convention. Why might that be useful to know? It gives you a way to split the code of a complex lambda into multiple methods while still allowing you to use it together with functions that take parameters of a function type. To do so, you can define a class that implements a function type interface. You can specify the base interface either as an explicit `FunctionN` type or, as shown in the following listing, using the shorthand syntax: `(P1, P2) -> R`. This example uses such a class to filter a list of issues by a complex condition.

Listing 11.17 Extending a function type and overriding `invoke()`

```
data class Issue(
    val id: String, val project: String, val type: String,
    val priority: String, val description: String
)

class ImportantIssuesPredicate(val project: String)
        : (Issue) -> Boolean {

    override fun invoke(issue: Issue): Boolean {
        return issue.project == project && issue.isImportant()
    }
```

Uses the function type as a base class

Implements the "invoke" method

```
    private fun Issue.isImportant(): Boolean {
        return type == "Bug" &&
                (priority == "Major" || priority == "Critical")
    }
}

>>> val i1 = Issue("IDEA-154446", "IDEA", "Bug", "Major",
...                "Save settings failed")
>>> val i2 = Issue("KT-12183", "Kotlin", "Feature", "Normal",
... "Intention: convert several calls on the same receiver to with/apply")
>>> val predicate = ImportantIssuesPredicate("IDEA")
>>> for (issue in listOf(i1, i2).filter(predicate)) {     <-- Passes the predicate
...     println(issue.id)                                      to filter()
... }
IDEA-154446
```

Here the logic of the predicate is too complicated to put into a single lambda, so you split it into several methods to make the meaning of each check clear. Converting a lambda into a class that implements a function type interface and overriding the `invoke` method is one way to perform such a refactoring. The advantage of this approach is that the scope of methods you extract from the lambda body is as narrow as possible; they're only visible from the predicate class. This is valuable when there's a lot of logic both in the predicate class and in the surrounding code and it's worthwhile to separate the different concerns cleanly.

Now let's see how the `invoke` convention can help you create a more flexible structure for your DSLs.

11.3.3 The "invoke" convention in DSLs: declaring dependencies in Gradle

Let's go back to the example of the Gradle DSL for configuring the dependencies of a module. Here's the code we showed you earlier:

```
dependencies {
    compile("junit:junit:4.11")
}
```

You often want to be able to support both a nested block structure, as shown here, and a flat call structure in the same API. In other words, you want to allow both of the following:

```
dependencies.compile("junit:junit:4.11")

dependencies {
    compile("junit:junit:4.11")
}
```

With such a design, users of the DSL can use the nested block structure when there are multiple items to configure and the flat call structure to keep the code more concise when there's only one thing to configure.

The first case calls the `compile` method on the `dependencies` variable. You can express the second notation by defining the `invoke` method on `dependencies` so that it takes a lambda as an argument. The full syntax of this call is `dependencies.invoke({…})`.

The `dependencies` object is an instance of the `DependencyHandler` class, which defines both `compile` and `invoke` methods. The `invoke` method takes a lambda with a receiver as an argument, and the type of the receiver of this method is again `DependencyHandler`. What happens in the body of the lambda is already familiar: you have a `DependencyHandler` as a receiver and can call methods such as `compile` directly on it. The following minimal example shows how that part of `DependencyHandler` is implemented.

Listing 11.18 Using `invoke` to support flexible DSL syntax

```
class DependencyHandler {
    fun compile(coordinate: String) {                   // Defines a regular
        println("Added dependency on $coordinate")      // command API
    }

    operator fun invoke(                                // Defines "invoke" to
            body: DependencyHandler.() -> Unit) {       // support the DSL API
        body()                                          // "this" becomes a receiver of
    }                                                   // the body function: this.body()
}

>>> val dependencies = DependencyHandler()

>>> dependencies.compile("org.jetbrains.kotlin:kotlin-stdlib:1.0.0")
Added dependency on org.jetbrains.kotlin:kotlin-stdlib:1.0.0

>>> dependencies {
...     compile("org.jetbrains.kotlin:kotlin-reflect:1.0.0")
>>> }
Added dependency on org.jetbrains.kotlin:kotlin-reflect:1.0.0
```

When you add the first dependency, you call the `compile` method directly. The second call is effectively translated to the following:

```
dependencies.invoke({
    this.compile("org.jetbrains.kotlin:kotlin-reflect:1.0.0")
})
```

In other words, you're invoking `dependencies` as a function and passing a lambda as an argument. The type of the lambda's parameter is a function type with a receiver, and the receiver type is the same `DependencyHandler` type. The `invoke` method calls the lambda. Because it's a method of the `DependencyHandler` class, an instance of that class is available as an implicit receiver, so you don't need to specify it explicitly when you call `body()`.

One fairly small piece of code, the redefined `invoke` method, has significantly increased the flexibility of the DSL API. This pattern is generic, and you can reuse it in your own DSLs with minimal modifications.

You're now familiar with two new features of Kotlin that can help you build DSLs: lambdas with receivers and the `invoke` convention. Let's look at how previously discussed Kotlin features come in play in the DSL context.

11.4 Kotlin DSLs in practice

By now, you're familiar with all the Kotlin features used when building DSLs. Some of them, such as extensions and infix calls, should be your old friends by now. Others, such as lambdas with receivers, were first discussed in detail in this chapter. Let's put all of this knowledge to use and investigate a series of practical DSL construction examples. We'll cover fairly diverse topics: testing, rich date literals, database queries, and Android UI construction.

11.4.1 Chaining infix calls: "should" in test frameworks

As we mentioned previously, clean syntax is one of the key traits of an internal DSL, and it can be achieved by reducing the amount of punctuation in the code. Most internal DSLs boil down to sequences of method calls, so any features that let you reduce syntactic noise in method calls find a lot of use there. In Kotlin, these features include the shorthand syntax for invoking lambdas, which we've discussed in detail, as well as *infix function calls*. We discussed infix calls in section 3.4.3; here we'll focus on their use in DSLs.

Let's look at an example that uses the DSL of kotlintest (https://github.com/kotlintest/kotlintest, the testing library inspired by Scalatest), which you saw earlier in this chapter.

Listing 11.19 Expressing an assertion with the kotlintest DSL

```
s should startWith("kot")
```

This call will fail with an assertion if the value of the `s` variable doesn't start with "kot". The code reads almost like English: "The `s` string should start with this constant." To accomplish this, you declare the `should` function with the `infix` modifier.

Listing 11.20 Implementing the `should` function

```
infix fun <T> T.should(matcher: Matcher<T>) = matcher.test(this)
```

The `should` function expects an instance of `Matcher`, a generic interface for performing assertions on values. `startWith` implements `Matcher` and checks whether a string starts with the given substring.

Listing 11.21 Defining a matcher for the kotlintest DSL

```
interface Matcher<T> {
    fun test(value: T)
}
```

```
class startWith(val prefix: String) : Matcher<String> {
    override fun test(value: String) {
        if (!value.startsWith(prefix))
            throw AssertionError("String $value does not start with $prefix")
    }
}
```

Note that in regular code, you'd capitalize the name of the `startWith` class, but DSLs often require you to deviate from standard naming conventions. Listing 11.21 shows that applying infix calls in the DSL context is simple and can reduce the amount of noise in your code. With a bit more cunning, you can reduce the noise even further. The kotlintest DSL supports that.

Listing 11.22 Chaining calls in the kotlintest DSL

```
"kotlin" should start with "kot"
```

At first glance, this doesn't look like Kotlin. To understand how it works, let's convert the infix calls to regular ones.

```
"kotlin".should(start).with("kot")
```

This shows that listing 11.22 was a sequence of two infix calls, and `start` was the argument of the first one. In fact, `start` refers to an object declaration, whereas `should` and `with` are functions called using the infix call notation.

The `should` function has a special overload that uses the `start` object as a parameter type and returns the intermediate wrapper on which you can then call the `with` method.

Listing 11.23 Defining the API to support chained infix calls

```
object start

infix fun String.should(x: start): StartWrapper = StartWrapper(this)

class StartWrapper(val value: String) {
    infix fun with(prefix: String) =
        if (!value.startsWith(prefix))
            throw AssertionError(
                "String does not start with $prefix: $value")
}
```

Note that, outside of the DSL context, using an `object` as a parameter type rarely makes sense, because it has only a single instance, and you can access that instance rather than pass it as an argument. Here, it does make sense: the `object` is used not to pass any data to the function, but as part of the grammar of the DSL. By passing `start` as an argument, you can choose the right overload of `should` and obtain a `StartWrapper` instance as the result. The `StartWrapper` class has the `with` member, taking as an argument the actual value that you need to perform the assertion.

The library supports other matchers as well, and they all read as English:

```
"kotlin" should end with "in"
"kotlin" should have substring "otl"
```

To support this, the `should` function has more overloads that take object instances like `end` and `have` and return `EndWrapper` and `HaveWrapper` instances, respectively.

This was a relatively tricky example of DSL construction, but the result is so nice that it's worth figuring out how this pattern works. The combination of infix calls and `object` instances lets you construct fairly complex grammars for your DSLs and use those DSLs with a clean syntax. And of course, the DSL remains fully statically typed. An incorrect combination of functions and objects won't compile.

11.4.2 Defining extensions on primitive types: handling dates

Now let's take a look at the remaining teaser from the beginning of this chapter:

```
val yesterday = 1.days.ago
val tomorrow = 1.days.fromNow
```

To implement this DSL using the Java 8 `java.time` API and Kotlin, you need just a few lines of code. Here's the relevant part of the implementation.

Listing 11.24 Defining a date manipulation DSL

```
val Int.days: Period
    get() = Period.ofDays(this)              <-- "this" refers to the value
                                                 of the numeric constant.
val Period.ago: LocalDate
    get() = LocalDate.now() - this              <-- Invokes LocalDate.minus
                                                    using operator syntax
val Period.fromNow: LocalDate
    get() = LocalDate.now() + this              <-- Invokes LocalDate.plus
                                                    using operator syntax
>>> println(1.days.ago)
2016-08-16
>>> println(1.days.fromNow)
2016-08-18
```

Here, `days` is an extension property on the `Int` type. Kotlin has no restrictions on the types that can be used as receivers for extension functions: you can easily define extensions on primitive types and invoke them on constants. The `days` property returns a value of type `Period`, which is the JDK 8 type representing an interval between two dates.

To complete the sentence and support the `ago` word, you need to define another extension property, this time on the `Period` class. The type of that property is a `LocalDate`, representing a date. Note that the use of the `-` (minus) operator in the `ago` property implementation doesn't rely on any Kotlin-defined extensions. The `LocalDate` JDK class defines a method named `minus` with a single parameter that matches the Kotlin convention for the `-` operator, so Kotlin maps the operator to that method automatically. You can find the full implementation of the library, supporting all time units and not just days, in the kxdate library on GitHub (https://github.com/yole/kxdate).

Now that you understand how this simple DSL works, let's move on to something more challenging: the implementation of the database query DSL.

11.4.3 Member extension functions: internal DSL for SQL

You've seen the significant role played by extension functions in DSL design. In this section, we'll study a further trick that we've mentioned previously: declaring extension functions and extension properties in a class. Such a function or property is both a member of its containing class and an extension to some other type at the same time. We call such functions and properties *member extensions*.

Let's look at a couple of examples that use member extensions. They come from the internal DSL for SQL, the Exposed framework, mentioned earlier. Before we get to that, though, we need to discuss how Exposed allows you to define the database structure.

In order to work with SQL tables, the Exposed framework requires you to declare them as objects extending the `Table` class. Here's a declaration of a simple `Country` table with two columns.

Listing 11.25 Declaring a table in Exposed

```
object Country : Table() {
    val id = integer("id").autoIncrement().primaryKey()
    val name = varchar("name", 50)
}
```

This declaration corresponds to a table in the database. To create this table, you call the `SchemaUtils.create(Country)` method, and it generates the necessary SQL statement based on the declared table structure:

```
CREATE TABLE IF NOT EXISTS Country (
    id INT AUTO_INCREMENT NOT NULL,
    name VARCHAR(50) NOT NULL,
    CONSTRAINT pk_Country PRIMARY KEY (id)
)
```

As with generating HTML, you can see how declarations in the original Kotlin code become parts of the generated SQL statement.

If you examine the types of the properties in the `Country` object, you'll see that they have the `Column` type with the necessary type argument: `id` has the type `Column<Int>`, and `name` has the type `Column<String>`.

The `Table` class in the Exposed framework defines all types of columns that you can declare for your table, including the ones just used:

```
class Table {
    fun integer(name: String): Column<Int>
    fun varchar(name: String, length: Int): Column<String>
```

```
    // ...
}
```

The `integer` and `varchar` methods create new columns for storing integers and strings, respectively.

Now let's see how to specify properties for the columns. This is when member extensions come into play:

```
val id = integer("id").autoIncrement().primaryKey()
```

Methods like `autoIncrement` and `primaryKey` are used to specify the properties of each column. Each method can be called on `Column` and returns the instance it was called on, allowing you to chain the methods. Here are the simplified declarations of these functions:

```
class Table {
    fun <T> Column<T>.primaryKey(): Column<T>
    fun Column<Int>.autoIncrement(): Column<Int>
    // ...
}
```

Sets this column as a primary key in the table

Only integer values can be auto-incremented.

These functions are members of the `Table` class, which means you can't use them outside of the scope of this class. Now you know why it makes sense to declare methods as member extensions: you constrain their applicability scope. You can't specify the properties of a column outside the context of a table: the necessary methods won't resolve.

Another great feature of extension functions that you use here is the ability to restrict the receiver type. Although any column in a table can be its primary key, only numeric columns can be auto-incremented. You can express this in the API by declaring the `autoIncrement` method as an extension on `Column<Int>`. An attempt to mark a column of a different type as auto-incremented will fail to compile.

What's more, when you mark a column as `primaryKey`, this information is stored in the table containing the column. Having this function declared as a member of `Table` allows you to store the information in the table instance directly.

Member extensions are still members

Member extensions have a downside, as well: the lack of extensibility. They belong to the class, so you can't define new member extensions on the side.

For example, imagine that you wanted to add support for a new database to Exposed and that the database supported some new column attributes. To achieve this goal, you'd have to modify the definition of the `Table` class and add the member extension functions for new attributes there. You wouldn't be able to add the necessary declarations without touching the original class, as you can do with regular (nonmember) extensions, because the extensions wouldn't have access to the `Table` instance where they could store the definitions.

Let's look at another member extension function that can be found in a simple `SELECT` query. Imagine that you've declared two tables, `Customer` and `Country`, and each `Customer` entry stores a reference to the country the customer is from. The following code prints the names of all customers living in the USA.

Listing 11.26 Joining two tables in Exposed

```
val result = (Country join Customer)
    .select { Country.name eq "USA" }
result.forEach { println(it[Customer.name]) }
```

Corresponds to this SQL code: WHERE Country.name = "USA"

The `select` method can be called on `Table` or on a join of two tables. Its argument is a lambda that specifies the condition for selecting the necessary data.

Where does the `eq` method come from? We can say now that it's an infix function taking `"USA"` as an argument, and you may correctly guess that it's another member extension.

Here you again come across an extension function on `Column` that's also a member and thus can be used only in the appropriate context: for instance, when specifying the condition of the `select` method. The simplified declarations of the `select` and `eq` methods are as follows:

```
fun Table.select(where: SqlExpressionBuilder.() -> Op<Boolean>) : Query

object SqlExpressionBuilder {
    infix fun<T> Column<T>.eq(t: T) : Op<Boolean>
    // ...
}
```

The `SqlExpressionBuilder` object defines many ways to express conditions: compare values, check for being not `null`, perform arithmetic operations, and so on. You'll never refer to it explicitly in the code, but you'll regularly call its methods when it's an implicit receiver. The `select` function takes a lambda with a receiver as an argument, and the `SqlExpressionBuilder` object is an implicit receiver in this lambda. That allows you to use in the body of the lambda all the possible extension functions defined in this object, such as `eq`.

You've seen two types of extensions on columns: those that should be used to declare a `Table`, and those used to compare the values in a condition. Without member extensions, you'd have to declare all of these functions as extensions or members of `Column`, which would let you use them in any context. The approach with member extensions gives you a way to control that.

> **NOTE** In section 7.5.6, we looked at some code that worked with Exposed while talking about using delegated properties in frameworks. Delegated properties often come up in DSLs, and the Exposed framework illustrates that well. We won't repeat the discussion of delegated properties here, because

we've covered them in detail. But if you're eager to create a DSL for your own needs or improve your API and make it cleaner, keep this feature in mind.

11.4.4 Anko: creating Android UIs dynamically

While talking about lambdas with receivers, we mentioned that they're great for laying out UI components. Let's look at how the Anko library (https://github.com/Kotlin/anko) can help you build a UI for Android applications.

First let's see how Anko wraps familiar Android APIs into a DSL-like structure. The following listing defines an alert dialog that shows a somewhat bothersome message and two options (to proceed further or to stop the operation).

Listing 11.27 Using Anko to show an Android alert dialog

```
fun Activity.showAreYouSureAlert(process: () -> Unit) {
    alert(title = "Are you sure?",
          message = "Are you really sure?") {
        positiveButton("Yes") { process() }
        negativeButton("No") { cancel() }
    }
}
```

Can you spot the three lambdas in this code? The first is the third argument of the `alert` function. The other two are passed as arguments to `positiveButton` and `negativeButton`. The receiver of the first (outer) lambda has the type `AlertDialogBuilder`. The same pattern comes up again: the name of the `AlertDialogBuilder` class won't appear in the code directly, but you can access its members to add elements to the alert dialog. The declarations of the members used in listing 11.27 are as follows.

Listing 11.28 Declarations of the `alert` API

```
fun Context.alert(
        message: String,
        title: String,
        init: AlertDialogBuilder.() -> Unit
)

class AlertDialogBuilder {
    fun positiveButton(text: String, callback: DialogInterface.() -> Unit)
    fun negativeButton(text: String, callback: DialogInterface.() -> Unit)
    // ...
}
```

You add two buttons to the alert dialog. If the user clicks the Yes button, the `process` action will be called. If the user isn't sure, the operation will be canceled. The `cancel` method is a member of the `DialogInterface` interface, so it's called on an implicit receiver of this lambda.

Now let's look at a more complex example where the Anko DSL acts as a complete replacement for a layout definition in XML. The next listing declares a simple form with two editable fields: one for entering an email address and another for putting in a password. At the end, you add a button with a click handler.

Listing 11.29 Using Anko to define a simple activity

```
verticalLayout {
    val email = editText {
        hint = "Email"
    }
    val password = editText {
        hint = "Password"
        transformationMethod =
            PasswordTransformationMethod.getInstance()
    }
    button("Log In") {
        onClick {
            logIn(email.text, password.text)
        }
    }
}
```

Declares an EditText view element, and stores a reference to it

An implicit receiver in this lambda is a regular class from Android API: android.widget.EditText.

A short way to call EditText.setHint("Password")

Calls EditText.setTransformationMethod(...)

Declares a new button...

...and defines what should be done when the button is clicked.

References declared UI elements to access their data

Lambdas with receivers are a great tool, providing a concise way to declare structured UI elements. Declaring them in code (compared to XML files) lets you extract repetitive logic and reuse it, as you saw in section 11.2.3. You can separate UI and business logic into different components, but everything will still be Kotlin code.

11.5 Summary

- Internal DSLs are an API design pattern you can use to build more expressive APIs with structures composed of multiple method calls.
- Lambdas with receivers employ a nesting structure to redefine how methods are resolved in the lambda body.
- The type of a parameter taking a lambda with a receiver is an extension function type, and the calling function provides a receiver instance when invoking the lambda.
- The benefit of using Kotlin internal DSLs rather than external template or markup languages is the ability to reuse code and create abstractions.
- Using specially named objects as parameters of infix calls allows you to create DSLs that read exactly like English, with no extra punctuation.
- Defining extensions on primitive types lets you create a readable syntax for various kinds of literals, such as dates.
- Using the `invoke` convention, you can call arbitrary objects as if they were functions.

- The kotlinx.html library provides an internal DSL for building HTML pages, which can be easily extended to support various front-end development frameworks.
- The kotlintest library provides an internal DSL that supports readable assertions in unit tests.
- The Exposed library provides an internal DSL for working with databases.
- The Anko library provides various tools for Android development, including an internal DSL for defining UI layouts.

JAVA/PROGRAMMING LANGUAGES

Kotlin IN ACTION

Jemerov • Isakova

Developers want to get work done—and the less hassle, the better. Coding with Kotlin means less hassle. The Kotlin programming language offers an expressive syntax, a strong intuitive type system, and great tooling support along with seamless interoperability with existing Java code, libraries, and frameworks. Kotlin can be compiled to Java bytecode, so you can use it everywhere Java is used, including Android. And with an efficient compiler and a small standard library, Kotlin imposes virtually no runtime overhead.

Kotlin in Action teaches you to use the Kotlin language for production-quality applications. Written for experienced Java developers, this example-rich book goes further than most language books, covering interesting topics like building DSLs with natural language syntax. The authors are core Kotlin developers, so you can trust that even the gnarly details are dead accurate.

What's Inside

- Functional programming on the JVM
- Writing clean and idiomatic code
- Combining Kotlin and Java
- Domain-specific languages

This book is for experienced Java developers.

Dmitry Jemerov and **Svetlana Isakova** are core Kotlin developers at JetBrains.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit www.manning.com/books/kotlin-in-action

Free eBook

SEE INSERT

"Explains high-level concepts and provides all the necessary details as well."

—From the Foreword by Andrey Breslav Lead Designer of Kotlin

"Like all the other great *in Action* titles from Manning, this book gives you everything you need to become productive quickly."

—Kevin Orr, Sumus Solutions

"Kotlin is fun and easy to learn when you have this book to guide you!"

—Filip Pravica, Info.nl

"Thorough, well written, and easily accessible."

—Jason Lee, NetSuite

ISBN-13: 978-1-61729-329-0
ISBN-10: 1-61729-329-6

MANNING $44.99 / Can $51.99 [INCLUDING eBOOK]