Extracted from:

# Programming DSLs in Kotlin

Design Expressive and Robust Special Purpose Code



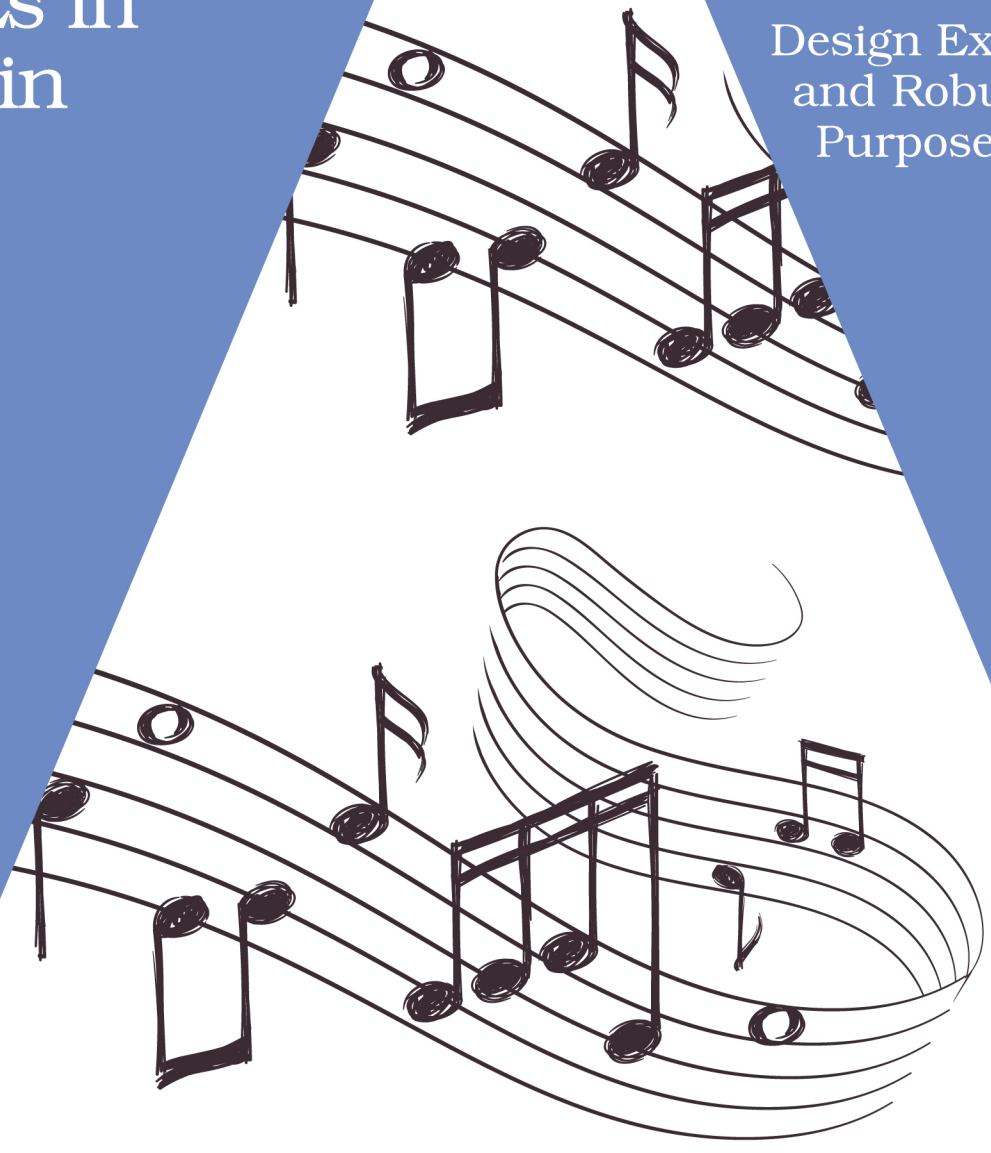
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

# Programming DSLs in Kotlin

## Design Expressive and Robust Special Purpose Code



Venkat Subramaniam

*edited by Jacquelyn Carter*

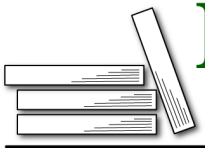
# Programming DSLs in Kotlin

Design Expressive and Robust Special Purpose Code

Venkat Subramaniam

The Pragmatic Bookshelf

Raleigh, North Carolina





For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Jacquelyn Carter
Copy Editor: L. Sakhi MacMillan
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.



ISBN-13: 978-1-68050-793-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—March 2021

In natural languages, fluency refers to spoken or written words that effortlessly and smoothly flow to convey the essence of a thought, concept, or an idea. Like those "hums" in speech or noisy words like "basically" in writing, anything that disrupts the flow hinders fluency. Likewise, a syntax is fluent if it's smooth, flows well, and has as little noise or extraneous details as possible.

Fluent code isn't necessarily short or concise, but it doesn't contain anything superfluous. For example, this syntax is filled with noise:

```
fetch.balance(12345678);
```

Though we may be used to it, we should recognize that the dot, (), and ; are there because a language may insist on it and not because they are essential. In comparison to that noisy syntax, the following is more fluent:

```
fetch balance 12345678
```

Where possible, we should design code to facilitate that style.

When designing DSLs, aim for fluency, as it makes communication pleasant. It reduces the drudgery and enhances the effectiveness of what's being conveyed. Programmers used to C-like languages are sadly accustomed to the ceremonies and noisy syntax, whereas programmers used to languages like Ruby and Haskell often enjoy more fluency. Kotlin, being a multi-platform language, offers nice flexibilities, with as little ceremony and noise as possible. When designing our own DSLs, we can exploit that.

To design code with fluency, we have to aim for the fewest elements. The user of a DSL shouldn't be forced to write (), ;, dot, {}, classes, function declarations, or variable definitions using `val` or `var`.

If host languages give us options to avoid syntax like `new`, ;, (), ->, and the like, the specification feels a lot closer to a natural language than code. When designing our own internal DSLs, we have to exploit the flexibilities of the host languages to the fullest extent to achieve fluency.

As you'll see, DSLs can enjoy fluency when hosted on top of Kotlin. Kotlin is a language of low ceremony. For example, it doesn't insist on having ; at the end of each line. Functions that are marked `infix` can be called without a dot and parentheses. By using overloaded operators, where it makes sense, we can make the syntax of a DSL obvious.

In this chapter we'll use a hypothetical banking application to look at some ways to design syntax that are fluent, thanks to these capabilities of Kotlin.

## Remove Noisy Syntax

Users don't like ceremony. Noisy syntax makes for a very unpleasant experience. Furthermore, users may be technical—engineers, data scientists, people with advanced degrees—but unlike programmers, they're mainly interested in using the application and don't care where to place a ;, dot, or (). When designing a DSL, we should strive to remove as much noisy syntax as possible from the eyes and hands of the users.

Suppose we're designing fluent syntax for a DSL to be used in a banking application by a domain expert. To query for the balance of an account, the user may be asked to write something like this:

fluency/infix.kts

```
fetch.balance(12345678);
```

That may not bother a programmer who has endured the syntax of C-like languages. However, we'll receive no affection from our users if they have to key in that syntax. At the very least, they may be tempted to spank us with their little pinky that's forced to type that ending ;. As designers of DSLs we need to do better, to get rid of the ., (), and ;—all the noisy parts.

The DSL user should be able to type in just the essence, like so:

fluency/infix.kts

```
fetch balance 12345678
```

Achieving this level of fluency is incredibly easy and almost effortless in Kotlin.

First, Kotlin doesn't care for ;, so just don't tell the users about ; and we're good. Unless they are recovering programmers, users will never ask if they need to place a ;.

Syntax like `a + b` is called `infix` notation, as opposed to prefix notation (`+ a b`), which is the syntax of languages like Lisp and Clojure. Infix notation is intuitive and less noisy.

Dropping the . and () takes a little effort. Those two symbols can be dropped if a function is marked with the `infix` accessor. Kotlin permits `infix` only on functions that have a receiver (member functions or extension functions) and take a single argument.

We can easily support the fluent DSL snippet syntax `fetch balance...`, by writing a singleton `fetch` with a `balance` function marked with `infix`:

fluency/infix.kts

```
object fetch {
  infix fun balance(number: Int) = println("Fetch the balance for $number")
```

```
}
```

We defined a singleton with a lowercase `fetch` instead of the conventional Pascal-case `Fetch` in order to support the expected syntax. Instead of breaking away from the convention, we may also use a type alias, as we'll see later in this chapter.

When designing DSLs, make extensive use of `infix` functions. Pause when writing a function and ask if it should be marked with `infix`—in other words, do you want the user to express the call to this function with little noise? The answer is almost always yes.

## Design for Fluency

Internal DSLs enjoy a love-hate relationship with their host language. On one hand, the host language removes the burden of implementing a parser. On the other hand, the syntax that may be used for the internal DSLs is limited by the syntax permissible in the host language. This limitation can hinder our ability to introduce fluency, and we'll have to put extra effort into designing around the limitations. Let's look at some examples of such limitations we'll run into when we write DSLs in Kotlin, and devise some workarounds.