Types and Characteristics of DSLs

We'll cover the following

- External vs. internal DSLs
- Context-driven and fluent

We'll first look at the types of DSLs and then discuss their main characteristics. Knowing these will help you make a right choice, both for the type of DSL and for the language to program them.

External vs. internal DSLs

When designing, you have to choose between an external or an internal DSL.

You enjoy greater freedom if you opt to create an external DSL, but you're responsible for creating the parser to parse and process the DSL. That can be a lot of effort, and you have to balance that against the flexibility you get.

You design an internal or embedded DSL within the confines of a host language. The language's compilers and tools serve as the parser. The good news is you don't expend any effort to parse. However, to achieve fluency and implement certain features, you have to be very creative and apply some tricks to make that possible in the host language.

Some examples of external DSLs are CSS, the ANT build file, the good old Make build file, and so on. On the other hand, the Rake build file and the Gradle build file are good examples of internal DSLs.

In this chapter, we focus on building internal DSLs with Kotlin as the host language.

Context-driven and fluent

DSLs are context-driven and highly fluent. Context reduces noise in communication, it makes it succinct, concise, and expressive. It's like a smile and

eye contact between friends to acknowledge hearing a favorite song. No words are spoken, but yet there's communication because they share a common context. Fluency also reduces noise and at the same time makes it easy to express ideas.

Context can also reduce the possibility of errors. The same things may have different meanings in different contexts. With an implicit context, the semantics of the vocabulary falls in place, within reasonable bounds.

The fluency in Kotlin favors concise and expressive syntax. Context may be managed through parameters and receivers in Kotlin. And Kotlin also has facilities to narrow the context down further to minimize errors, as we'll see soon.
