

Creative DSLs in Kotlin

Design Strategies for Real-World Projects

Daniel Gronau

Apress®

Daniel Gronau

Creative DSLs in Kotlin
Design Strategies for Real-World Projects

Apress®

Daniel Gronau
Sandersdorf, Sachsen-Anhalt, Germany

ISBN 979-8-8688-2820-1 e-ISBN 979-8-8688-2821-8
https://doi.org/10.1007/979-8-8688-2821-8

© Daniel Gronau 2026

This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Unless otherwise expressly agreed by the Publisher in writing, the Publisher does not authorize use or reproduction of all or any part of this work in any manner for the purpose of training, fine-tuning, evaluating, or developing artificial intelligence models, technologies, or systems. In accordance with Article 4(3) of the EU Directive 2019/790 on copyright and related rights in the Digital Single Market, the Publisher expressly reserves all rights with respect to this work under the text and data mining exception of the directive.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, New York, NY 10004. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.comwww.springeronline.com, or visit . Apress Media, LLC is a Delaware LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.This Apress imprint is published by the registered company APress Media, LLC, part of Springer Nature.
The registered company address is: 1 New York Plaza, New York, NY 10004, U.S.A.

To my parents Eva and Wolfgang Gronau

Introduction

> *A beginning is the time for taking the most delicate care that the balances are correct.*
> —Frank Herbert, *Dune*

The book title might have already given it away: this book is about writing domain-specific languages (DSLs) in Kotlin. It was born out of curiosity and frustration – the curiosity to explore the language's expressiveness while pushing its boundaries, and the frustration of finding few resources beyond common examples.

At some point, I realized that I gathered more material on my hard drive than I had ever seen in a single book on the topic. I don't mean to suggest that this book wrote itself, but, in a sense, I only had to arrange the parts and fill in some gaps. I found that writing a book can be very rewarding when you really care about the topic. If you share my curiosity – or perhaps my frustration – this book is for you, and I'm glad to have you on board.

Why Another Book About DSLs in Kotlin?

Kotlin is a powerful language that is well suited for writing domain-specific languages. There is already a lot of literature on the subject, but in my opinion, many articles and books fall short in a few key areas:

- They often focus on oversimplified examples that do not provide a clear understanding of how to write DSLs in practice.
- They present solutions in a vacuum, without explaining the design process, the requirements and constraints that apply, or the trade-offs that were made.
- They do not attempt to categorize the different types of DSLs or compare their relative strengths and weaknesses.
- They don't take into account that Kotlin and Java code often coexist, and ignore the interoperability issues that arise when a Kotlin DSL needs to be called from Java.

This book aims to fill these gaps by providing a comprehensive guide to writing DSLs in Kotlin, with a focus on practical examples and real-world considerations. One of the main goals of this book is to go beyond toy examples and provide real-world insights into the process of writing DSLs in practice. This includes discussion of the design process, the requirements and constraints that apply, the trade-offs that must be made, and the challenges that arise.

To achieve this, the book progresses from common examples to more challenging and complex cases, ending with a discussion of real-world DSLs that have been successful in practice. In addition to providing practical guidance, it attempts to establish a comprehensive terminology for classifying DSLs, similar to the way the design pattern movement has established a common language for coding solutions. Overall, the book aims to provide a comprehensive and practical guide to designing and implementing DSLs using Kotlin.

Who Is This Book Not (Yet) Intended For?

This book is not intended for readers who are new to Kotlin or Java. Although it contains a chapter on Kotlin language features and their potential application in DSLs, it is not a detailed introduction to the Kotlin language. If you are not already familiar with Kotlin or Java, it is recommended that you take the time to learn these languages before attempting to create DSLs. Writing DSLs can be challenging, and it is important to have a solid foundation in the host language before attempting to create a full-fledged DSL. This book will still be here when you are ready to take on the challenge of developing a DSL.

How This Book Is Structured

Chapters 1–3 give a basic overview of DSLs and the design process. Chapter 4 takes a closer look at the Kotlin language from the perspective of a DSL designer. Chapters 5–11 introduce different kinds of DSLs. Chapter 12 discusses the use of code generation. Chapter 13 focuses on issues with Java interoperability. Chapter 14 explores successful examples of existing DSLs.

Prerequisites

You should have installed Kotlin 2.2.0 or higher in order to run all code examples in this book. Note that some examples were written specifically for Kotlin JVM. For convenience, it is recommended that you use an IDE like IntelliJ IDEA or Eclipse, but you can also try out smaller examples online on pages like play.kotlinlang.org.

Downloading the Code

The source code for this book and more complete examples are available to readers at https://github.com/Apress/Creative-DSLs-in-Kotlin. You can also discover related titles from Apress here: https://link.springer.com/brands/apress.

Contacting the Author

Please do not hesitate to contact me if you find any errors or have any suggestions for improvement; your feedback is very valuable to me and will help me improve this book for future readers.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub. For more detailed information, please visit https://www.apress.com/gp/services/source-code.

Table of Contents

Part I: Writing DSLs

Chapter 1: Introduction

What Is a DSL?

Internal and External DSLs

Code Generation for Internal DSLs

Common Use Cases for DSLs

DSL Design Principles

Kotlin and DSLs

What's Next?

Chapter 2: Requirements Analysis

Defining the Problem Domain

Research

Specifying the Output of the DSL

The Syntactic Gap

Learning Experience

Safety of Use

Extensibility

Maintainability

Performance and Memory Requirements

Java Interoperability

Closed or Open Source

Ready, Steady, Go?

What's Next?

Chapter 3: Writing a DSL
Imagine the Ideal Syntax
Lowering the Cognitive Load
Prototyping
Formalization and Documentation
Implementation
DSL-Specific Challenges
Name Collisions
Coupling
Code Conventions
Disorganized Output
Testing
Documentation
Sample Project
What's Next?
Chapter 4: Relevant Language Features
Backtick Identifiers
Named Arguments and Default Values
Trailing Lambda Arguments
Varargs
Vararg Position and Trailing Lambda Syntax
Property Syntax
Extensions and Receivers
Type Narrowing

Loan Pattern
The @DslMarker Annotation
Extension Properties
Operator Overloading
Unary Operators
Binary Arithmetic Operators
Range and In Operators
Index Access and Invoke Operators
Equality and Comparison Operators
Overload Responsibly
Infix Notation for Functions
Functional Interfaces
Generics
Resolving Type Erasure Conflicts
Reified Generics
Value Classes
Type Aliases
Annotations
Reflection
Context Parameters
Experimental Features
Collection Literals
Contracts
Summary

Part II: DSL Categories

Chapter 5: Algebraic DSLs

Case Study: A DSL for Complex Numbers

Case Study: Modular Arithmetic – Using a Context

Java Interoperability

No Surprises, Please!

Summary

Common Applications

Pros

Cons

What's Next?

Chapter 6: Builder Pattern DSLs and Method Chaining

The Classical Builder

Named Arguments Instead of Builders

Nesting Builders

Flattening Instead of Nesting

Nesting with Varargs

The Type-Safe Builder Pattern

Counting Builder

Builders with Multiple Stages

Summary

Common Applications

Pros

Cons

What's Next?
Chapter 7: Loan Pattern DSLs
Case Study: HttpRequest DSL
Case Study: HttpRequest with AutoDSL
Builder Type Inference
Summary
Common Applications
Pros
Cons
What's Next?
Chapter 8: Modeling State Transitions in DSLs
Case Study: DSL for SQL Queries Using the Builder Pattern
The "Separate Classes" Approach
The "Chameleon Class" Approach
The "Phantom Type" Approach
Case Study: DSL for SQL Queries Using the Loan Pattern
The "Separate Classes" Approach
The "Chameleon Class" Approach
The "Phantom Type" Approach
Summary
Common Applications
Pros
Cons
What's Next?

Chapter 9: String-Parsing DSLs
Case Study: Forsyth–Edwards Notation
Case Study: Chemical Equations As Strings
Writing a Parser for Chemical Equations
Using a Parser Library
Summary
Common Applications
Pros
Cons
What's Next?
Chapter 10: Annotation-Based DSLs
Case Study: Mapper DSL
Synergy with String-Based DSLs
Summary
Common Applications
Pros
Cons
What's Next?
Chapter 11: Hybrid DSLs
Quasi-Lingual DSLs
Case Study: Chemical Equations
Case Study: Pattern Matching
Summary
Common Applications

Pros

Cons

What's Next?

Part III: Supplemental Topics

Chapter 12: Code Generation for DSLs

Case Study: Physical Quantities

Writing an Annotation Processor Using KSP

Designing the DSL and Writing the Annotations Module

Writing the Annotation Processor

Case Study: Generating Data Class Patterns

Summary

Common Applications

Pros

Cons

What's Next?

Chapter 13: Java Interoperability

Renaming Identifiers

Value Classes and Mangling

Package-Level Definitions of Functions and Variables

Generate Overloaded Methods

Accessing Fields

Generics

Calling Functions with Reified Type Parameters

Checked Exceptions

Prevent Java Access
Summary
What's Next?
Chapter 14: Real-World DSL Examples
KotlinPoet
Gradle
Kotest
MockK
Better-Parse
Konform-kt
Arrow
Summary
What's Next?
Index

About the Author

Daniel Gronau
is a seasoned software developer with nearly 25 years of industry experience and a passion for programming languages. After starting with Java in 2001, he pursued modern alternatives such as Ceylon, Scala, and Kotlin. Based in Germany, his experience spans companies like Spread Group (sprd.net AG) and Guardian Industries, and he currently works for a division of Aurubis AG.



About the Technical Reviewer

Charles Bushakra
is a retired software engineer with almost 40 years of experience in programming and technical documentation. He earned a master's degree in Computer Science from DePaul University in 2016. Charles has worked in a variety of fields over the years, from small business accounting, embedded systems for scientific instruments and scheduling software, to writing documentation at Borland Software and Juniper Networks. He finished his career as a software engineer at Siemens DISW, working on two large-scale documentation processing systems built around DITA and custom XML processing engines written in Java.

Part I
Writing DSLs

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
D. Gronau, *Creative DSLs in Kotlin*
https://doi.org/10.1007/979-8-8688-2821-8_1

1. Introduction

Daniel Gronau[1] ✉
(1) Sandersdorf, Sachsen-Anhalt, Germany

> *There are two fatal errors that keep great projects from coming to life:*
>
> 1. *Not finishing*
> 2. *Not starting*
>
> —Attributed to Buddha Gautama

In this chapter, we will get an overview of domain-specific languages and their intended purpose, as well as the differences between internal and external domain-specific languages (DSLs). We will also explore the relationship between DSLs and our chosen host language, Kotlin.

What Is a DSL?

> *A Domain-Specific Language (DSL) is a computer language that's targeted to a particular kind of problem, rather than a general-purpose language that's aimed at any kind of software problem.*

—Martin Fowler, Domain-Specific Languages Guide

The intent of writing a DSL is to make a particular domain more accessible and easier to read and write, avoid errors, and sometimes follow established standards or conventions (e.g., SQL for database access, or mathematical notation). The target audience for DSLs may be the authors themselves, library users, or people who have domain knowledge but don't usually write code. Sometimes it is sufficient for experts to be able to read and understand DSL code, but not write it, in order to check correctness and provide feedback.

DSLs are often used to model business logic in specific domains, such as transactions for a financial company. They can also be useful for communicating at the boundaries of a system, including tasks such as database access, serialization, web connectivity, and UI design. In addition, DSLs can be used to perform calculations or simulations, for testing and logging, code generation, and even for writing other DSLs. In all of these cases, DSLs provide a specialized language tailored to the needs of a particular domain, allowing developers to more easily express concepts and perform tasks in a natural and intuitive way.

Internal and External DSLs

This book discusses DSLs that are implemented within Kotlin and are therefore limited to the language's existing functionality. These are *internal DSLs* (or *embedded DSLs*). A major advantage is that these DSLs don't need any special treatment, no extra steps like reading and parsing files, so they fit seamlessly with the rest of the code.

A disadvantage of internal DSLs is that they are limited by the syntax inherited from the host language. Kotlin offers great freedom in DSL design, especially compared to Java. However, it is still possible that the language is not expressive enough to design the DSL you need. In such cases, *external DSLs* are an option: they have their own rules and syntax, and require lexers, parsers, etc. This results in a higher overhead compared to internal DSLs. However, writing external DSLs

has become much easier with new libraries, frameworks, and improved tools.

An edge case is internal DSLs that are realized entirely inside strings, the way, for example, regular expressions work: although they are technically internal DSLs, they feel and behave more like external DSLs because the embedding in the language is very shallow.

Code Generation for Internal DSLs

One challenge in designing a DSL is the risk of combinatorial explosion, where the number of possible combinations of elements or operations in the DSL becomes too large to manage effectively. For example, in a DSL representing physical quantities, there may be many possible combinations for multiplying or dividing quantities. In order to avoid illegal conversions and provide a pleasing syntax, it may be necessary to write a significant amount of boilerplate code to handle all valid operations. In such cases, code generation can be a useful tool to automate the creation of this boilerplate code, making it easier to manage the complexity of the DSL while keeping the DSL expressive and maintainable.

There are also libraries that create DSLs for you. If you need a well-known style of DSL, all you have to do is describe what you need (e.g., by annotating your business classes accordingly), and the library will generate the DSL code for you. An example of this approach is AutoDSL for Kotlin.

Common Use Cases for DSLs

While it is difficult to clearly distinguish between the different types of use cases, a general classification is still helpful. In particular, each DSL category has use cases that it can model better than others. Some common types of use cases for DSLs include

- **Implementation Support**

    - **Code Generation:** DSLs can be used to generate code in a specific programming language or format, allowing repetitive tasks to be automated and complex systems to be built.
    - **Library Customization:** DSLs can help integrate external code by customizing it to follow the paradigms, conventions, and patterns of the internal code base. A good example is Kotlin's adaptation of the core Java libraries.
    - **Testing:** DSLs can be used to define and execute tests in a domain-specific manner, helping to validate the behavior of a system.
- **System Management**
    - **Configuration Management:** DSLs can be used to configure and manage systems, applications, or infrastructure in a declarative manner.
    - **Workflow Orchestration:** Workflow Orchestration DSLs can be used to define and manage complex workflows or business processes.
- **Runtime Behavior**
    - **Data Creation and Initialization:** DSLs can be used to define and construct data structures in a domain-specific manner, helping to represent and manipulate complex data.
    - **Data Transformation:** DSLs can provide concise and expressive ways to perform transformations on data, such as filtering, aggregating, or mapping.
    - **Data Validation:** DSLs can be used to define validation rules and constraints specific to a particular domain.
    - **Defining Operations:** DSLs can be used to specify complex operations in a domain-specific way, helping to understand and reason about the behavior of the system.
    - **Execute Actions:** DSLs can provide a natural way to specify and execute actions, such as triggering events or initiating processes.
- **Peripheral Systems**

    - **Logging:** DSLs can provide a specialized language for logging messages and events, giving the log messages more structure and making them easier to process.
    - **Monitoring:** DSLs can be used to monitor the system status and to measure parameters such as system performance or memory usage.
    - **Reporting and Analytics:** DSLs can be used to define queries, aggregations, and transformations for reporting and analysis purposes.
- **Specific Applications**
    - **Natural Language Processing:** DSLs can be used to create language models, define linguistic rules, or implement specific language processing tasks.
    - **Simulation and Modeling:** DSLs can be used to build simulation models or mathematical models for various fields such as physics, engineering, finance, or biology.

Each of these use cases has its own unique challenges and requirements, and different categories of DSLs may be better suited to modeling certain types of use cases than others.

DSL Design Principles

There are a few general principles an internal DSL should follow:

- **Conciseness:** The DSL syntax should be succinct and expressive.
- **Consistency:** The DSL syntax should adhere to a certain style; similar tasks should require a similar syntax. The behavior of the DSL should be logical and predictable.
- **Coverage:** The DSL must cover the problem domain; there should be no gaps, as well as no overreach into other areas.
- **Usability:** The DSL should be easy, safe, and intuitive to use. It's not enough to make a DSL concise; it should also take user expectations into account and follow the *Principle of Least Surprise*. Error handling

should be comprehensive, providing clear and informative error messages that help users identify and resolve problems quickly.
- **Modularity:** If it makes sense to use a part of the DSL on its own, it should be easy to do so.
- **Extensibility:** A DSL should be designed to be easily extended and customized. Users should be able to add new functionality or modify existing behavior without significant effort or disruption to the overall design.
- **Interoperability:** DSLs often need to interact with existing systems or integrate with other DSLs. Designing a DSL with interoperability in mind allows seamless integration with external components and simplifies data exchange. Sometimes it may even be necessary to provide a way to bypass DSL functionality, in order to allow access from other languages such as Java, or for automated tools.
- **Maintainability:** The code of the DSL implementation should be easy to read and to maintain.

In many DSL tutorials and related literature, there is a tendency to focus only on the "sexy" principles of DSL design, such as conciseness and usability. In practice, however, a DSL project can fail if the other principles are overlooked, or if a good compromise between conflicting requirements can't be found. Ultimately, successful DSL design requires a holistic approach that considers all relevant factors.

Kotlin and DSLs

At this point, it's worth considering the characteristics of Kotlin that make it well suited for building DSLs. Kotlin is a programming language developed by JetBrains, the company behind popular IDEs like IntelliJ IDEA, WebStorm, and PyCharm. From the beginning, Kotlin was designed with a focus on readability, practicality, security, and interoperability.

Compared to Java, Kotlin has a more concise and expressive syntax, making it easier to write and read code. It also has a number of language features that are particularly useful for building DSLs.

Together, these features allow developers to create DSLs with a fluid and intuitive API that is easy to use and understand, and lends itself naturally to this style of coding. We take a closer look at the most important features in Chapter 4.

In Kotlin, it is often easy to add "miniature DSLs" to existing code on the fly. This means that the boundary between everyday code and DSLs is fluid, which seems to be a deliberate design choice. This flexibility allows developers to gradually adapt and improve existing code in an organic way, without the need for major refactoring. In my opinion, this kind of language design plays a significant role in the success of Kotlin as a language.

What's Next?

Now that we know what a DSL is, the next question is what information we need to gather to design one. As with any other project, it is crucial to have a clear picture of the intended scope beforehand. Not all DSL requirements are obvious, and sometimes they conflict with each other; therefore, it is helpful to approach the task systematically.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
D. Gronau, *Creative DSLs in Kotlin*
https://doi.org/10.1007/979-8-8688-2821-8_2

2. Requirements Analysis

Daniel Gronau[1] ✉
(1) Sandersdorf, Sachsen-Anhalt, Germany

> *If you think good architecture is expensive, try bad architecture.*
> —Brian Foote and Joseph Yoder[1]

As a software engineer, it's easy to get caught up in the excitement of building a DSL and lose sight of the bigger picture. However, it's important to take a step back and carefully consider the goals and requirements of your DSL before diving into implementation. This will help ensure that your DSL is complete, well-behaved, maintainable, and extensible.

It is important that everyone involved has a clear understanding of the scope and purpose of your DSL. This includes identifying the target audience and how the DSL will be used. Getting everyone on the same page minimizes the risk of overlooking important details. However, this shouldn't be a long and boring process. To simplify this task, this chapter is essentially a checklist of decisions to make before you begin implementation.

Defining the Problem Domain

The first question to answer is, of course, which domain to tackle. For small domains and domains with existing standards, this question is usually easy to answer. For larger domains, it can be tempting to

overgeneralize. Try to be specific. Sometimes a conscious decision to leave out some of the less-used and harder-to-implement parts of the problem domain can make the DSL more flexible and leaner. As in other areas of software development, it can be helpful to explicitly specify non-goals. Also, don't forget to specify the target platform, language level, etc.

In general, it's a good practice to organize your DSL in layers so that the lower layers don't depend on the higher ones and can therefore be used independently. For example, if you are designing a DSL for linear algebra, the part of the DSL that works with matrices might be independent of the rest of the system, allowing that part to be used in other contexts as well.

If the application already uses other DSLs, you should consider whether using the new DSL in parallel would be confusing or might produce unexpected results. In some cases, it may be wise to mimic the syntax of an existing DSL in order to simplify the experience that users already have. A real-world example of this approach is the KotlinPoet library, which mimics its sister project JavaPoet.

Research

After you have identified the problem domain, you should look at existing DSLs. You may be surprised at the number of existing solutions, and of course, there is little point in reinventing the wheel. Even if the results don't fit your use case perfectly, they might serve as a base implementation or a source of inspiration.

> **Note** There is an ongoing debate about how much a project should rely on libraries, since over-reliance can lead to problems such as *dependency hell*. However, you shouldn't go to the other extreme and try to build everything from scratch – this is known as the "Not Invented Here" syndrome. Instead, it is important to strike a balance. The decision to use or avoid a library should always be conscious and documented.

If there is no existing DSL for your problem, you might also consider using a DSL generator. Using such libraries can make designing a DSL a breeze. This book discusses AutoDSL as an example of this approach.

Specifying the Output of the DSL

In most cases, the expected output of the DSL is fairly obvious. Sometimes, you can choose to perform an action (such as a query) directly or create objects that can perform that action instead. Creating objects is often the better option. This approach is usually easier to test and debug, and it can be convenient to be able to create these instances in other ways.

Another question is whether to create the target entities of an external API directly or follow the adapter pattern or similar approaches. The decision depends on the intended architecture of your application, but in general, you should try to use external APIs only at the boundaries of the application to reduce dependencies. This guideline should also be followed for DSLs.

Most systems produce not only direct output but also secondary data such as logs or metrics. Users should have easy and flexible access to this type of data produced by a DSL. In the case of logging, for example, this may mean not forcing a specific logging framework on the user, but rather using a logging façade such as SLF4J or Kotlin-Logging in your DSL. You should also, of course, specify the expected secondary output of your DSL.

The Syntactic Gap

Usually, an internal DSL can only be an approximation of the syntax you really want; there is a *syntactic gap*. It is tempting to subordinate everything else in order to get as close to the ideal syntax as possible. But of course, beauty comes at a price; pushing the boundaries of the host language's syntax can be problematic, and neglecting the other requirements can render a beautiful DSL useless.

That's why it's important to have a rough idea of how close the internal DSL needs to get to the ideal syntax, and when to start compromising. A relevant question in this context is how "tech-savvy" the users of the DSL are. Generally speaking, people without programming experience require more streamlined DSLs than software engineers, who are often more tolerant of imperfections in the syntax.

Learning Experience

Having a beautiful and concise syntax for a DSL doesn't necessarily mean that it's easy to master. Once a DSL reaches a certain size and complexity, it is naive to assume that people can use it without assistance. The user's learning experience is often an afterthought, but I think it is important to consider it in the analysis process.

The DSL itself can be designed to have a shallow learning curve, for example, by mimicking known systems or by emphasizing consistency over conciseness. An example of a "too concise" DSL might be regular expressions, as many users – myself included – have to look up the syntax over and over again.

Training and documentation are, of course, essential to a good learning experience. However, the quality of error and warning messages is equally important and can significantly impact how easily users can learn and understand a new DSL. To address this, it is crucial to understand your audience and their skill level.

In the end, it's not the DSL itself that creates value, but the people who use it effectively, and we should always keep that in mind. Planning for the necessary support in advance can help improve the DSL and empower users to get the most out of it.

Safety of Use

An often-overlooked requirement is how and how well the user must be protected against pitfalls and misuse of the DSL. This includes questions like whether misuse should cause compile-time or runtime

errors or whether it's acceptable for the DSL to express some states that shouldn't be allowed.

Reasons for unsafe behavior may include overly flexible syntax, lack of sanity checks, or errors due to operator precedence. In some cases, the requirement for safety of use conflicts with the pursuit of ideal syntax, and a trade-off between beauty and safety must be made. A certain amount of unsafe behavior may be acceptable for technical or experienced users, or in areas with lower safety requirements, such as test data generation.

Note that we are only discussing *accidental* misuse here. Kotlin is a programming language that supports features such as casting, reflection, and dynamic proxies. As such, Kotlin DSLs are generally vulnerable to deliberate manipulation. As with any other data from external sources, the output from DSLs cannot necessarily be trusted and may need to be verified.

Extensibility

Inadequate extensibility is a common pitfall in the design of internal DSLs, which usually work with rather specialized techniques and language features. Sometimes new DSL requirements can't be properly implemented with the chosen feature set, which can render the whole DSL unusable. Therefore, it is useful to have a general understanding of which extensions may be requested after the initial release of the DSL. This information will help avoid using techniques that are not flexible enough to handle future requirements when implementing the DSL.

Another type of extensibility that is often overlooked is user customization. For example, consider a DSL for working with physical quantities: there are too many units to cover them all, but users may have a particular set of units that they need in their daily work. Therefore, it would be beneficial if the DSL allowed users to add custom units. Such considerations can greatly increase the usefulness and success of a DSL. If possible, it is recommended that you open your DSL to user extensions and provide documentation on how to do so.

However, it may be advisable to limit extensibility if you can predict that extending the DSL may lead to unwanted or confusing results.

Maintainability

A DSL project requires not only initial implementation but also ongoing maintenance. It is essential to estimate the resources required to maintain and update the code, which influences decisions such as acceptable external dependencies or the potential need for code generation.

An often-underestimated aspect of maintenance is establishing effective feedback channels for DSL users. A bug tracker should be considered the bare minimum, but larger projects may benefit from additional features, such as discussion forums and dedicated user support. Good communication with users can play a major role in the overall success of a DSL, so it is beneficial to make it an integral part of the DSL strategy.

Performance and Memory Requirements

Performance considerations often don't receive much attention. However, DSLs typically call for extra operations, instantiate additional classes, and may trigger garbage collections. If you are working with big data or have a DSL design that isn't mindful of its resource usage, you may experience performance and memory issues. While these issues can often be resolved through optimization later on, the solution might not be compatible with the chosen DSL syntax. That's why it's important to estimate performance requirements up front and use load testing and metrics accordingly.

Java Interoperability

This is a Kotlin-specific consideration: since many projects use a mix of Java and Kotlin, it may be necessary to use a Kotlin DSL from Java code. This direction is usually more challenging than using Java from Kotlin

code. Depending on the language features, a Kotlin DSL might be practically unusable from Java. However, in many cases, some "glue code" can help bridge the gap, and the Kotlin language includes features designed to address Java interoperability issues.

If Java interoperability is required, this should be considered during the design stage. Chapter 13 discusses the challenges and possible solutions in more detail.

Closed or Open Source

One important consideration that should be decided up front is whether to open source the DSL project. Doing so can have several benefits, including community contributions, increased exposure, and potential collaborations. However, it also means giving up some control over the direction of the project and potentially exposing any bugs or vulnerabilities to the public. In addition, open-source projects require ongoing maintenance and support, which can be time-consuming and resource-intensive. Ultimately, the decision to open source a DSL project should be carefully weighed against the potential benefits and drawbacks.

Ready, Steady, Go?

Once you have identified the requirements for your DSL project, it is important to carefully consider its scope, complexity, and benefits before moving forward. While building DSLs can be very beneficial, it is important to ensure that they have a clear purpose and provide tangible value to their users and that the scope of the project is manageable for your organization.

If you find that the project does not meet these criteria, it may be best to cancel it. However, if you believe that the project is both feasible and useful, you can proceed with implementation. Keep the overall goals and purpose of the DSL in mind as you work, and be prepared to adapt and refine your approach as needed. Remember that building a

DSL is a means to an end, not an end in itself. It should ultimately serve the needs of its users and provide value to your organization or the public. Therefore, careful consideration of the project's feasibility, purpose, and value is critical before beginning implementation.

What's Next?

Once clear requirements have been established, the next step is to design and implement the DSL. This phase often involves more trial and error and requires more creativity than typical projects. The next chapter presents a lightweight approach that considers these specifics.

Footnotes

1 https://www.laputan.org/mud/mud.html#BigBallOfMud

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
D. Gronau, *Creative DSLs in Kotlin*
https://doi.org/10.1007/979-8-8688-2821-8_3

3. Writing a DSL

Daniel Gronau[1] ✉
(1) Sandersdorf, Sachsen-Anhalt, Germany

> *Prototype, then polish. Get it working before you optimize it.*
> —Eric S. Raymond, *The Art of Unix Programming*[1]

Now that you've collected the requirements for your DSL and are ready to move into the design and implementation phase, it's advisable to take a structured approach. While it's only natural to be excited to get started, adding a little structure to your workflow can help you avoid common pitfalls and ensure the success of your DSL.

The following four-step workflow can be a helpful guide as you begin to design and implement your DSL:

- **Imagine the Ideal Syntax:** Begin by considering the concepts and ideas you want to express in your DSL, and brainstorm possible syntaxes that could be used to represent them. Consider readability, expressiveness, and usability as you explore different options.
- **Prototype:** Once you have a rough idea of the syntax you want to use, create a prototype of your DSL using Kotlin's language features. This will allow you to experiment with different approaches and see how they work in practice.
- **Formalize and Document:** Once you are satisfied with the prototype of your DSL, formalize the syntax and semantics of your DSL.

- **Implementation:** With the syntax and semantics of your DSL formally defined, you can begin implementing your DSL in Kotlin. Use the features and techniques you have learned to create a complete and functional DSL that is ready for use.

However, designing a DSL often involves trial and error. Things might not work out as planned and have to be reconsidered. A practical approach must take this into account. As Figure 3-1 shows, the suggested workflow isn't strictly linear; it allows for iterations.

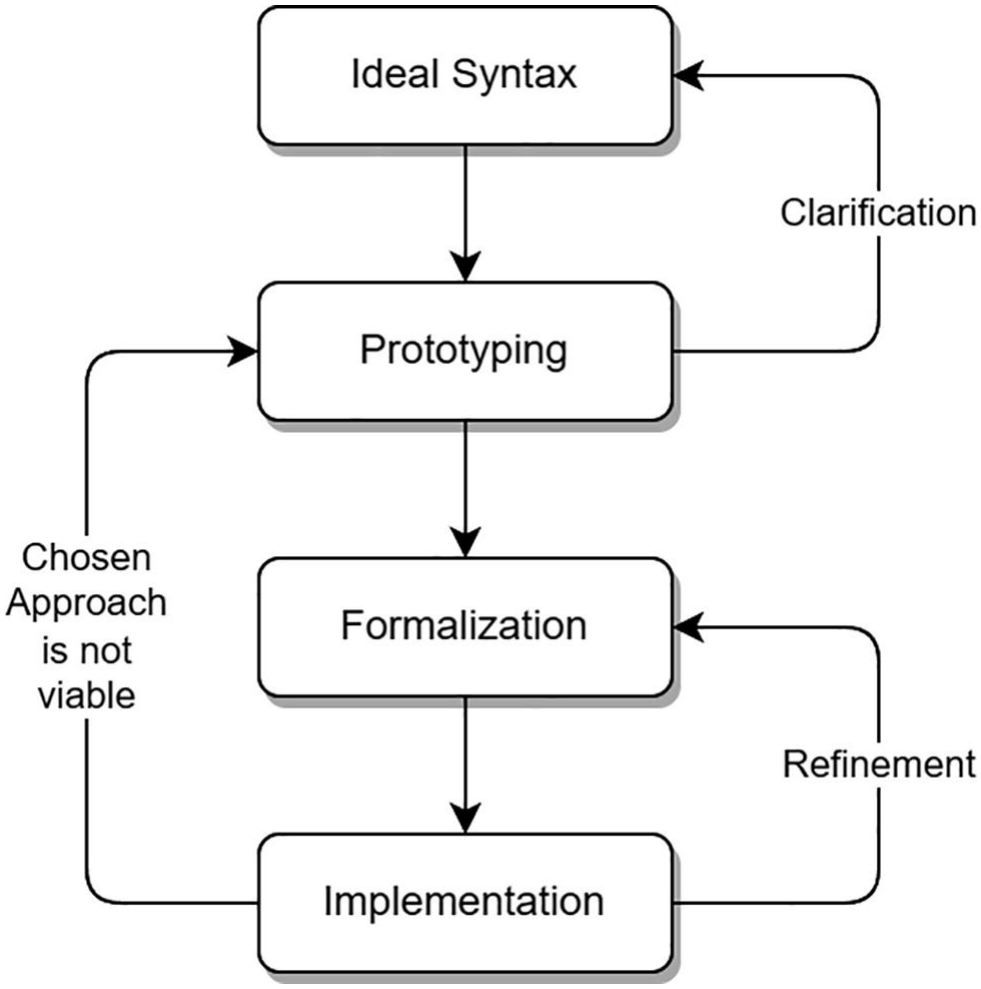


Figure 3-1 DSL Development Workflow

Following this workflow will ensure that your DSL project has a well-organized and comprehensive structure while maintaining the flexibility to address problems as they arise.

Imagine the Ideal Syntax

It may seem strange to begin the process of designing and implementing a DSL by imagining an ideal syntax, especially if you are more familiar with languages like Java, which tend to have limited options for building DSLs. However, this step can be extremely valuable, especially if you have some creative freedom in designing your DSL. Even if the "ideal" syntax is largely predetermined by factors such as industry standards or existing conventions, it can be helpful to take some time to explore different options and consider how different syntaxes might affect the readability, expressiveness, and usability of your DSL.

As a software developer, it's natural to think in terms of language features and implementation details. In particular, experienced programmers have a strong understanding of the capabilities and limitations of a programming language, and they tend to think in terms of what is and isn't possible based on these factors. However, this kind of thinking can sometimes limit your creativity and lead to tunnel vision. When designing a DSL, it's important to remember that the goal is not simply to write a working application, but to create a unique and consistent language.

Rather than focusing too narrowly on language features and implementation details, it can be helpful to step back and consider the broader goals and purpose of your DSL. Get into a brainstorming mode, and be open to exploring creative and unconventional approaches. The goal is to sketch out what an ideal DSL syntax might look like, regardless of how it is implemented. If your team can't agree on a single syntax, it might be okay to have two candidates, but probably not more.

In this step, it's generally not necessary to specify a formal grammar. Instead, it may be more effective to write lots of examples of how you envision the DSL being used, and include comments and explanations to help clarify your intentions. Be sure to note any inconsistencies or gaps in the examples, and specify what the output for each example

should look like. This is important information that will help ensure that your DSL is clear and intuitive to use.

If possible, it can also be helpful to get feedback from domain experts or potential users of your DSL. This can help you identify important details that you may have overlooked and can help you create a DSL that meets the needs and expectations of your target audience. Few things are more frustrating than presenting a new implementation only to discover that you have forgotten a crucial detail, so it pays to be thorough and to seek feedback whenever possible.

Lowering the Cognitive Load

Cognitive Load Theory claims that there are two types of difficulties in learning to use a system: intrinsic cognitive load, which is caused by the inherent complexity of the system itself, and extraneous cognitive load, which is caused by the way the information is presented. A good analogy would be a book on a very challenging topic (intrinsic load) versus a book written in a small, hard-to-read font or in a language less familiar to the reader (extrinsic load).

When working on the ideal syntax, thinking about ways to reduce extraneous cognitive load can be a very useful strategy. Of course, consistent and intuitive syntax is an important factor, but there are many other ways to reduce cognitive load, such as

- **Meaningful Naming:** Descriptive names make it easier for users to understand the purpose and functionality without additional cognitive load.
- **Natural Language Elements:** These can make the DSL read more like plain language, reducing the cognitive gap between the DSL and the user's everyday vocabulary.
- **Defaults and Smart Defaults:** Minimize the need for users to explicitly specify redundant information. The DSL should intelligently infer context whenever possible.
- **Gradual Complexity:** Introduce complexity gradually. Or as Alan Key put it: "Simple things should be simple, complex things should be

possible."
- **Reuse Existing Features:** Don't introduce features that can already be expressed in Kotlin or in widely used libraries.

If you focus too much on syntax alone, you may end up with a DSL that is very pretty, but hard to use. The concept of cognitive load can help broaden your perspective and lead to more user-friendly solutions.

Prototyping

Now that you have defined an ideal syntax, it's time to start coding. Because it's often necessary to experiment and iterate to get the syntax and structure of your DSL just right, it can be very helpful to start with a prototype.

A prototype is a simplified version of your DSL that allows you to try different approaches and see how they work in practice. You can choose to implement only selected features of your DSL ("spike implementations"), or you can create an empty shell that simply verifies that your chosen syntax is viable. With a prototype, you can quickly test different ideas and identify potential problems, save time and effort by avoiding extensive testing and sanity checks, and focus on experimenting with different approaches to see what works best. Prototyping can also help minimize the risk of spending time on solutions that ultimately prove to be unworkable.

As you work on prototyping your DSL, it's important to periodically check that the syntax you've chosen is acceptable and meets your goals and requirements. Feedback from domain experts or potential users of your DSL can be invaluable in this regard, helping you to identify any problems or areas for improvement. An advantage of the prototyping approach is that it allows you to easily present different versions of your DSL and gather feedback on each one. This can help you refine your syntax and structure until you have a design that is both effective and intuitive.

If you find that you've missed some details when specifying the ideal syntax for your DSL, it's important to fix its specification, rather than just "muddling through" with an incomplete or inadequate design. The ideal syntax description is intended to serve as a guide for both the prototyping and implementation phases, and as a benchmark for the quality of the final syntax of your DSL. As such, it's important to take the time to get it right.

Formalization and Documentation

Once a prototype has demonstrated what a realistic syntax might look like, it's time to formalize the syntax. This process involves defining the structure and rules of your DSL in a more precise and detailed way to ensure that it is clear and consistent. The extent of this process will depend on the size and complexity of your project, and the level of precision and formality required.

In some cases, formalizing the syntax of your DSL may involve creating a simple documentation page or set of examples that demonstrate the structure and usage of your DSL. In other cases, it may be necessary to define a grammar that specifies the syntax of your DSL in a more precise and formal way.

Regardless of the approach you take, it's important to pay careful attention to the documentation of the mapping from your DSL to its output. This documentation should clearly explain how the various elements of your DSL are translated into the desired output, and it should be thorough and complete to ensure that your DSL is clear and easy to use.

Ensuring completeness is a crucial aspect of DSL design: it's important to make sure that the language is capable of expressing all required configurations and that no rare or unusual cases are overlooked.

A common problem leading to incompleteness is a lack of *orthogonality*, which refers to the idea that different elements of the language should be independent and should not overlap or interfere

with each other. For example, consider a DSL describing animals that doesn't allow you to select both "mammal" and "lays eggs" as characteristics, even though this combination actually exists (e.g., for the platypus). In this case, the DSL would lack orthogonality because it wouldn't allow you to describe certain animals completely and accurately.

Tip Don't skip the formalization step. While a prototype can be a useful tool for exploring different approaches and identifying potential problems, it is not a substitute for a formal specification of your DSL. The final implementation of your DSL will need precise specifications to ensure that it is clear and consistent, and future users of your DSL will also need detailed documentation to understand how to use it effectively. By formalizing the syntax of your DSL now, you can save time and effort later.

Implementation

The final step in the process of designing a DSL is implementation, which involves turning your DSL design into a working, functional language. While it may be possible to reuse some parts of your prototype in the final implementation, don't be afraid to start from scratch if necessary. The goal of the implementation phase is to create a high-quality DSL that is well-structured, flexible, and efficient, but often, prototype code doesn't meet these standards.

Be prepared for the possibility that your prototype is not thorough enough or does not cover all necessary cases and that you will hit a roadblock during the implementation phase. If this happens, it's important not to panic and to take a step back to assess the situation. One option you might consider in this situation is to go back to the prototype phase and explore other approaches or ideas. While it may be tempting to try to push through with your current approach, this can often be counterproductive, as it can limit your field of vision and make it harder to find a creative and effective solution.

If you find that you are writing a lot of boilerplate code during the implementation phase, you may want to consider using a source code generator to automate this process. This can save you time and effort and help you create a DSL that is easier to maintain and extend.

Finally, be sure to follow best practices when implementing your DSL. This includes writing tests and adding sanity checks to ensure that your DSL is reliable and behaves as expected, and following good coding practices to ensure that your DSL is well organized and easy to understand. By taking the time to do things right, you can create a DSL that is robust, reliable, and effective.

DSL-Specific Challenges

Implementing a DSL differs from typical programming tasks and comes with its own set of challenges and pitfalls. The following points highlight some issues that deserve special attention.

Name Collisions

A good DSL can be used extensively in a codebase, but this can increase the risk of naming conflicts, especially if the DSL adds extension methods to classes like `Int` or `String` that are used frequently. One way to mitigate this risk is to try to limit the scope of your DSL functions by putting them into DSL-specific objects or classes whenever possible. It's also a good idea to consider the potential for collisions when naming your functions, operators, etc., so that they are less likely to cause conflicts.

Coupling

When you write a DSL to create classes that are also under your control, you may be tempted to tightly integrate the DSL with those classes. In Java, this kind of tight coupling may be necessary because there is often no other way to write a convenient DSL. But Kotlin is much more expressive, for example, due to features like extension methods, so this excuse doesn't apply. It's generally good practice to avoid tightly

integrating a DSL with the classes it uses or creates, as this can lead to a number of problems, such as

- **Entangling DSL Code with Business Logic:** Such tight integration can make it difficult to separate the two and to make changes to one without affecting the other.
- **Making the DSL Part of the Business API:** When the DSL is part of the business API, the latter can become brittle, making it difficult to evolve the DSL independently from the business logic.
- **Limiting the Usefulness of Result Classes:** Tightly coupled result classes may not work on their own, or may be difficult to use with other tools and frameworks, or from other JVM languages such as Java. This can limit their usefulness in a variety of contexts, such as working with big data, testing, or code generation.
- **Complicating DSL Replacement:** It can be difficult to replace a tightly coupled DSL when the need arises. This can make it challenging to evolve your codebase over time and to take advantage of new technologies or approaches.

In general, it's a good idea to design your DSL in a way that minimizes coupling between the DSL and the classes it creates or operates on to avoid these kinds of problems. In most cases, Kotlin is expressive enough to create DSLs that are flexible and easy to use while still keeping the DSL and the classes it creates separate.

Code Conventions

It's generally a good practice to follow code conventions, as this can make your code more consistent and easier for other developers to understand. However, there may be cases where you need to compromise on certain conventions in order to create an expressive DSL. If you must compromise on code conventions, it's recommended to document your decision and the reasoning behind it, as this can make it easier for other developers to use and maintain your DSL.

You should also address any resulting warnings from the compiler or IDE. Often, you can use an annotation to override the warning,

thereby documenting that this was a conscious decision not to follow code conventions at this point.

Disorganized Output

The syntax of a domain-specific language tends to shape the structure of its output. This can introduce some inflexibility and produce results that do not quite meet the requirements. For example, the output may contain unnecessary elements or parts that could be combined.

Consider a DSL for building mathematical expressions. When assembling the different parts of an expression, it can be challenging for the DSL to identify cases that allow simplification, such as multiplying by one or adding zero. Additionally, the output may violate mathematical conventions. For example, factors are usually written before variables, not after.

If the output violates requirements like this, an optimization or normalization step may be necessary before returning the result. We will see examples of this throughout the book.

Testing

For some DSL categories, testing can be more difficult than for normal code because the code may be less rigid than usual, or – to use a mechanical analogy – may have more moving parts and degrees of freedom. This makes it more likely that edge cases or unwanted behavior will be missed. A particular challenge is compile-time guarantees: There's no convenient way to test that certain unwanted code structures won't compile. Overall, depending on the type of DSL, testing can be more challenging than for ordinary code and may require more attention and effort. Some common challenges in testing DSLs include

- **Complex Code Structures:** DSLs can have more complex code structures than ordinary code, such as classes that act as wrappers or intermediate builder classes.
- **Combinatorial Explosion:** DSLs may allow their elements to be combined as building blocks. This can make it difficult to test all

possible combinations and edge cases, and to ensure that the DSL behaves as expected.
- **Compile-Time Guarantees:** Some DSLs use type-level programming to introduce compile-time guarantees, but unfortunately, there is no convenient way to test that certain unwanted code structures don't compile.
- **Unusual Testing Scenarios:** Depending on the type of DSL, special testing scenarios may be required. For example, if your DSL is used to generate code, you may need to test the generated code in addition to the DSL itself.

Overall, it's important to be aware of the unique challenges of testing DSLs and the extra effort and attention that may be required to ensure that your DSL is reliable and bug-free.

Documentation

Many software developers don't like to write documentation, but it's necessary. When writing documentation for a DSL, keep in mind that it is essentially its own language, and users may not be familiar with all of its features and concepts. Therefore, it's important to provide clear, concise explanations of how the DSL works and how it should be used, as well as plenty of examples to illustrate key concepts. It's also a good idea to include visualizations or diagrams to help users understand complex concepts or interactions between different parts of the DSL.

Overall, the key is to be thorough and clear in your documentation, to provide enough information and examples to help users understand and use the DSL effectively, and to keep it up to date.

Sample Project

Creating a sample project can be a very effective way to help users understand and learn how to use the DSL. By providing a complete, working example that shows how the different elements of the DSL can be used and combined in a real-world context, you can give users a much better understanding of how to apply the DSL to their own

problem domain. There are a few key things to keep in mind when creating a sample project for a DSL:

- **Make It Clear and Concise:** Keep the sample project focused and to the point, and avoid unnecessary detail, complexity, and external dependencies.
- **Showcase the Recommended Style:** Often, a DSL can be applied in various ways, making it difficult to determine the best approach. An example project can promote an effective style by showcasing best practices or particular idioms.
- **Use Meaningful Examples:** Select examples that are relevant to the problem domain and that demonstrate the key features and capabilities of the DSL.
- **Provide Clear Explanations:** Along with the example code, provide clear explanations of what the code does and how it uses the DSL.
- **Make It Engaging:** This can be difficult if the subject matter itself is pretty dry, but if you have the chance, try to make the sample project fun, interactive and memorable.

What's Next?

By now, you should have a good understanding of how to approach a DSL project. The next chapter provides an overview of Kotlin features useful for DSL design. You may already be familiar with many of these features, but they are often used differently to build DSLs. Understanding them well will help you bring your ideas to life.

Footnotes

1 Eric S. Raymond, *The Art of Unix Programming*, O'Reilly, 2003.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
D. Gronau, *Creative DSLs in Kotlin*
https://doi.org/10.1007/979-8-8688-2821-8_4

4. Relevant Language Features

Daniel Gronau[1] ✉
(1) Sandersdorf, Sachsen-Anhalt, Germany

> *I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail.*
> —Abraham Maslow, *The Psychology of Science*[1]

Kotlin is a modern and beautiful programming language that offers several key features not found in Java. Here are some of the most notable features that set Kotlin apart:

- **Null-Safe Type System:** Kotlin's null-safety feature is a major improvement over Java, where null values can cause null pointer exceptions. This feature ensures that a variable cannot be null unless explicitly declared as such, reducing the risk of runtime errors.
- **Concise Syntax:** Kotlin's syntax is designed to be concise, reducing the amount of boilerplate code that developers need to write. Optional semicolons, operator overloading, and expression body functions are just a few examples of the syntactic sugar that not only makes Kotlin code easier to read and maintain but also helps you write expressive DSLs.
- **Object-Oriented Design:** In some regards, Kotlin takes a more object-oriented approach than Java, replacing static members with object declarations that can be used in a more flexible and modular way.

- **Functional Programming Features:** Kotlin supports functional programming by providing first-class and higher-order functions. The language encourages this style of programming by offering top-level and local functions, as well as trailing lambda syntax.
- **Multiplatform Support:** Kotlin can be compiled to run on multiple platforms, including the JVM and JavaScript, and as a native application.

Additionally, Kotlin offers excellent interoperability with Java, enabling easy calling of Java code from Kotlin and vice versa. Kotlin includes several annotations that support calling Kotlin code from Java, ensuring the two languages can work together seamlessly.

As mentioned in the preface, this book assumes that the reader has basic Kotlin knowledge. However, DSLs often use little-known language features, or they use some features differently than ordinary code. Even seasoned programmers are sometimes surprised by how certain parts of a DSL work. This chapter provides a brief overview of language features that may be relevant to writing DSLs. After reading it, you should have a better understanding of how some common DSL “tricks” work.

Backtick Identifiers

Kotlin allows for nearly any identifier as long as it is enclosed in backticks. This feature was introduced, in part, for Java interoperability. For instance, you can call a Java method with the same name as a Kotlin keyword. They can also be helpful for supporting code generation. We can use backticks for DSLs when we need descriptive identifiers that don’t follow the usual syntactic rules.

Backtick identifiers start and end with a backtick (`), and they can contain almost any character. Note that the backticks are just delimiters and are not part of the identifier. One common use of backticks is to make test names more descriptive, which offers more flexibility than Java's underscores and camel case.

```
fun `check that the slithy toves gyre and gimble
in the wabe`() {
    ...
}
```

Listing 4-1 Backtick Syntax Example

For Kotlin on the JVM, all characters except `\r \n , . ; : \ | / [ ] ? < >` and backticks themselves are allowed, and you will get a compiler error if you try to use an identifier that contains one of these characters. However, if you need to use some of them anyway, you can rename the method on the JVM with a valid identifier and suppress the compiler check.

```
@Suppress("INVALID_CHARACTERS")
@JvmName("diamond")
fun `<>`(carat: Double) {
    println("A $carat carat diamond!")
}
...
`<>`(5.5) // A 5.5 carat diamond!
```

Listing 4-2 Getting Forbidden Characters in Backticks Work

This is especially useful when simulating an operator with an infix function; see the section “Infix Notation for Functions” for details. I would still advise exercising caution when using this hack because relying on code that suppresses compilation errors is generally risky.

Named Arguments and Default Values

Java relies on the order of the parameters when calling a method or constructor, which can quickly become confusing. In contrast, Kotlin allows you to refer to arguments by name, which is much more readable and doesn’t require you to remember the order of the parameters.

```
fun makeColor(red: Int, green: Int, blue: Int,
alpha: Int)
    = Color(red, green, blue, alpha)

// call by argument order
val color1 = makeColor(220, 200, 100, 128)

// call by named argument
val color2 = makeColor(
        alpha = 128,
        red = 220,
        green = 200,
        blue = 100
    )
```

Listing 4-3 Named Arguments Example

In Java, a similar naming behavior can be mimicked using the builder pattern, but in many cases, Kotlin's built-in solution is more convenient.

In Kotlin, you can also assign default values to function and constructor arguments, which greatly simplifies the code compared to Java, where you have to write multiple methods or constructors with different combinations of default values to achieve the same effect. Therefore, Kotlin's default value approach reduces boilerplate code and improves readability. In the example above, it would make sense to set the alpha value to 255, since it is more common to define an opaque color than a translucent one.

```
fun makeColorWithDefault(
        red: Int,
        green: Int,
        blue: Int,
        alpha: Int = 255 // setting a default
value
    ) = Color(red, green, blue, alpha)
```

```
// setting all parameters
val color3 = makeColorWithDefault(220, 200, 100,
128)

// using the default value 255 for alpha
val color4 = makeColorWithDefault(220, 200, 100)
```

Listing 4-4 Default Argument Example

Both language features are useful on their own, but they complement each other very well. A nice example is the auto-generated `copy()` method in data classes.

```
data class Person(
    val firstName: String,
    val lastName: String,
    val age: Int
) {
    // the method is auto-generated, but would
look roughly like this:
    fun copy(firstName: String = this.firstName,
             lastName: String = this.lastName,
             age: Int = this.age
        ) = Person(firstName, lastName, age)
}

val person = Person("John", "Doe", 23)

val personOnBirthday = person.copy(age =
person.age + 1)
```

Listing 4-5 Copy Method in Data Classes

In the `copy()` method, the values of the current object are set as default values of the new instance, and thanks to the named argument feature, you can pick just the arguments which should be changed, and leave all others untouched.

Trailing Lambda Arguments

If a method expects an argument of a function type, you can use the usual lambda syntax with curly brackets when calling it. For example, you can merge a list of strings using the `fold()` method like this.

```
listOf("one", "two", "three").fold("", { s, t -> s
+ t })
```

Listing 4-6 Lambda Argument

However, if such an argument comes last, you can pull it out of the argument list and append it after the closing parenthesis.

```
listOf("one", "two", "three").fold("") {
   s, t -> s + t
}
```

Listing 4-7 Lambda As Trailing Argument

In case the function type is the only argument, you don't have to write the empty parentheses. The `map()` method is an example of a method with a single lambda argument.

```
listOf("one", "two", "three").map {
    s -> s.length
}
```

Listing 4-8 Single Lambda Argument

While this syntactic sugar might not look very impressive at first glance, it allows us to write very natural-looking DSLs for nested structures. Listing 4-9 shows an example from the Kotlin documentation.[2]

```
html {
    head {
        title {+"Page Title"}
```

```
    }
    body {
        // page content ...
    }
}
```

Listing 4-9 Trailing Lambdas in a DSL

Varargs

Varargs, short for "variable arguments," are a useful feature in both Java and Kotlin. They allow methods to take a variable number of arguments. However, Kotlin has made several improvements that make varargs safer and more convenient to use. For instance, you can specify default values for them.

Additionally, the syntax for varargs is more intuitive in Kotlin. In Java, it can sometimes be difficult to determine if an array is intended as a single value for a vararg or if its elements should be used as individual values. Kotlin addresses this problem by introducing the unary *spread operator* *, which indicates that the elements of an array should be used as values for a vararg rather than the array itself. Kotlin also allows you to use a combination of individual values and spread array elements.

```
val someArray = arrayOf(4, 6, 8)
val list = listOf(2, 0, *someArray, 5) // contains
2, 0, 4, 6, 8, 5
```

Listing 4-10 Mixing Single Arguments and Elements of Spread Arrays

Vararg Position and Trailing Lambda Syntax

Unlike Java, where a vararg must always be the last argument, Kotlin allows you to put the vararg anywhere, although you may need to use named arguments to avoid ambiguity.

```
fun notLast(vararg numbers: Int, someString:
```

```
String): String =
        "${numbers.toList()} - $someString"

val s = notLast(1, 2, 3, someString = "Hi!") //
[1, 2, 3] - Hi!
```

Listing 4-11 Varargs Not in Last Position

Note that vararg elements can't be assigned individually when referenced by a named argument, but must be bundled into an array instead.

```
val namedVararg = notLast(
    someString = "Hi!",
    numbers = intArrayOf(1, 2, 3)
) // [1, 2, 3] - Hi!
```

Listing 4-12 Varargs Addressed As Named Argument

At first glance, having the ability to put varargs wherever you want doesn't seem very useful. But there is one particular use case that is very interesting from a DSL design perspective: You can place a vararg as the second-to-last argument before a trailing lambda argument, and no named arguments are required.

```
fun varargAndLambda(
    someString: String,
    vararg numbers: Int,
    block: (Int) -> Unit
) { ... }

varargAndLambda("Hi!", 1, 2, 3) {
    ...
}
```

Listing 4-13 Varags Before a Trailing Lambda

Property Syntax

Kotlin allows you to control how properties are read and written. This makes it easy to hide DSL functionality. An easy example is checking preconditions before setting a value.

```
class TemperatureSensor {
    var celsius: Double = 0.0
        set(value) {
            require(value >= -273.15) {
                "Temperature is under absolute
zero."
            }
            field = value
        }
}
```

Listing 4-14 Property with Precondition

In the same way, you can perform additional actions when you read a value, or even change the return value itself.

```
class SensitiveData {
    val logger =
Logger.getLogger(this::class.java.name)

    var secretValue: Int = 42
        get() {
            logger.info("Access to secret value
$field ")
            return field
        }
}
```

Listing 4-15 Logging when Reading Properties

There are many other things you can do with properties, such as caching, lazy evaluation, delegation to other properties, or input sanitization. In the section "Extensions and Receivers," we'll discuss another use of the property syntax.

Extensions and Receivers

Extension functions, lambdas, and properties are some of Kotlin's most important features for DSLs. They allow you to add functionality to existing classes (even final ones) or objects without modifying them. These extensions are stand-alone constructs that operate on a *receiver*, which is the target class or object they are extending. The function body is in the receiver's scope, so you can access its public fields, methods, and so on. You can also refer to the receiver itself with this. Listing 4-16 shows an example of an extension function.

```
fun Int.digits(base: Int = 10): List<Int> =
    generateSequence(this.absoluteValue) { value -
>
        (value / base).takeIf { it > 0 }
    }
    .map { it % base }
    .toList()
    .reversed()

val zero = 0.digits()          // [0]
val taxiCab = 1729.digits()    // [1, 7, 2, 9]
val taxiBin = 1729.digits(2)   // [1, 1, 0, 1, 1,
0, 0, 0, 0, 0, 1]
val taxiHex = 1729.digits(16) // [6, 12, 1]
```

Listing 4-16 List of Digits As Extension Function

From the user's perspective, a call with a receiver appears exactly like a normal method call or property access on an instance of the receiver. This makes extension functions a great tool for adding DSL

functionality to classes over which you have no control. An excellent example is the so-called *scope functions* `apply()`, `run()`, `let()`, and `also()` in the Kotlin API. These functions make it easier to use expression body syntax for functions or concise variable assignments.

On the JVM, extension functions are implemented using static methods, with the receiver instance as the first argument. This means that extension methods can also be called from Java, although it looks less elegant.

```
// Java code
List<Integer> digits = IntUtilsKt.digits(1729,
10);  // [1, 7, 2, 9]
```

Listing 4-17 Calling an Extension Function from Java

Note Extension methods were first introduced in C# in 2007 (although there were predecessors, such as Objective-C categories). They made their way into Kotlin as "extension functions," specifically to seamlessly integrate Java classes into the Kotlin ecosystem. This approach was chosen over Scala's approach, because, although it is more powerful, it is also more complex to understand and manage. The benefits of extension methods were so apparent that they were introduced in Scala as well. Java does not natively support extension methods, but there are Java libraries that do, such as Manifold,[3] Project Lombok,[4] and Fluent.[5]

It is safe to say that extension methods are an important and successful advancement in modern object-oriented programming.

Type Narrowing

Interestingly, generic extension functions have a feature that normal instance methods lack: they can bind generic parameters to a particular type, thus limiting the range of possible recipients. The following example shows an extension function that calculates the product of numbers in a list.

```
fun List<Double>.product() = fold(1.0,
Double::times)

val p = listOf(1.0, 2.0, 3.0).product()  // 6.0
```

Listing 4-18 Type Narrowing in Extension Methods

The call to `Double::times` is possible only because the receiver is not just any list, but specifically a `List<Double>`, and this additional type information is also passed to the function body. This feature of extension methods can also be used in DSLs to perform compile-time checks.

Loan Pattern

The *loan pattern* is beneficial in DSLs because it controls the life cycle of the receiver class and hides the steps necessary to initialize and finalize instance creation or operation. Since lambdas can have receivers, it is easy to implement the loan pattern using the trailing lambda syntax.

Note In object-oriented programming, the loan pattern is a design pattern that involves encapsulating the use of a resource, such as a database connection or file handle, within a limited scope or block of code. This ensures the resource is properly acquired, used, and released, preventing resource leakage or conflicts with other code accessing the same resource.

Essentially, the loan pattern involves creating a resource object or acquiring a resource handle at the beginning of a code block, using the resource as needed within the block, and releasing or disposing of the resource at the end of the block. This ensures the resource is used only for the block's duration and is properly cleaned up when finished, even if an error occurs.

The loan pattern is useful when resources are limited or expensive to acquire. It can also improve code maintainability and robustness by making it easier to understand how resources are used and ensuring they are properly managed.

Consider the well-known `java.util.StringBuilder` class. It allows you to perform complex string operations, but to use it, you need to construct it and call its `toString()` method at the end. When using the loan pattern, these steps can be hidden, and the code looks cleaner.

```
val theUsualWay: String = StringBuilder()
    .append("World")
    .insert(0, "Hello ")
    .append('!')
    .toString()

// using a lambda argument with receiver
fun sb(block: StringBuilder.() -> Unit): String =
    StringBuilder()
        .apply { block(this) }
        .toString()

val usingTheLoanPattern: String = sb {
    append("World")
    insert(0, "Hello ")
    append('!')
}
```

Listing 4-19 Using the Loan Pattern for StringBuilder

Building DSLs using this pattern is very common, as it has several advantages over the classic builder pattern.

The @DslMarker Annotation

When you nest multiple extension functions with trailing lambdas, overlapping scopes can be a problem: things visible in the outer code blocks are also visible in the inner ones. For example, in a DSL for HTML generation, you might write

```
val page = html {
    head {...}
```

```
    body {
        head {} // ouch, head() is also visible
here
    }
}
```

Listing 4-20 Scope Issues in Nested Extension Functions

To avoid this problem, Kotlin provides a scope control mechanism:

- Define a custom annotation.
- Annotate that annotation itself with `@DslMarker`.
- Mark all involved receiver classes (or a common superclass) with your annotation.
- Now you can't access outer scope elements directly. You can still reference them indirectly, for example, using the syntax `this@html.head{...}`.

In our example, such an annotation might look like this.

```
@DslMarker
annotation class HtmlMarker
```

Listing 4-21 Custom Annotation for Scope Control

If the receiver classes of the lambda arguments of the `head()` and `body()` functions are annotated with `@HtmlMarker`, the above example would no longer compile.

Extension Properties

You can define not only extension functions and lambdas but also extension properties. Generally, they aren't used nearly as much as extension functions, but they can help to beautify DSLs, as they don't require you to write empty parentheses. Listing 4-22 shows how to create a custom `Amount` class by adding extension properties for the different currencies to `Double`.

```
data class Amount(val value: BigDecimal, val
currency: String)

val Double.USD
    get() = Amount(this.toBigDecimal(), "USD")

val Double.EUR
    get() = Amount(this.toBigDecimal(), "EUR")

val usdAmount = 22.46.USD // Amount(value=22.46,
currency=USD)

val eurAmount = 17.11.EUR // Amount(value=17.11,
currency=EUR)
```

Listing 4-22 Extension Property Example

The best syntax we could achieve with an extension function is `22.46.USD()`. However, the parentheses are no longer needed when using extension properties.

Extension properties are also useful when the result of a computation is a function. Consider currying, a technique that turns functions with multiple arguments into a chain of functions with single arguments. When written as an extension property, currying can be implemented as follows (for functions with two arguments).

```
val <A, B, R> ((A, B) -> R).curry: (A) -> (B) -> R
    get() = { a -> { b -> this@curry(a, b) } }

fun repeatString(i: Int, s: String): String =
s.repeat(i)

val rs3 = ::repeatString.curry(3)

val abc3 = rs3("Abc") // AbcAbcAbc
val x3 = rs3("x")     // xxx
```

Listing 4-23 Implementing Currying with Extension Properties

If we had defined `curry` as a function, we would have to write `::repeatString.curry()(3)` instead, which looks quite confusing.

Operator Overloading

Kotlin allows operator overloading, but is conservative in the sense that it allows only a fixed set of operators.

> **Note** The Boolean operators `&&` and `||`, the access operators `.`, `?.`, and `!!`, the unary spread operator `*`, the Elvis operator `?:`, and the property delegation operators `provideDelegate`, `getValue`, and `setValue` cannot be overloaded.

Some overloading functions require specific return types. The type `R` is used in the following tables to indicate that there are no such restrictions.

Unary Operators

Table 4-1 Unary Operators

Binary Arithmetic Operators

Table 4-2 Binary Arithmetic Operators

If these operators are defined, `a` is mutable, and the left and right sides have matching types (`B` is a subtype of `A`), they can also be used in the assignments `+=`, `-=`, `*=`, `/=`, and `%=`. If you don't want the normal binary form, but only the assignment, you can define it explicitly.

Table 4-3 Arithmetic Assignment Operators

Again, `a` must be mutable, and `B` must be a subtype of `A`. Also, the return type of the function must be `Unit`. Having both the binary and assignment versions of an operator in scope results in an ambiguity error.

Range and In Operators

Table 4-4 Range and In Operators

The `..<` operator was introduced as a replacement for the `until` infix function.

Index Access and Invoke Operators

Table 4-5 Index Access and Index Operators

Note that the index access operator `[]` requires at least one element, while the invoke operator `()` can also be used without arguments.

Equality and Comparison Operators

Table 4-6 Equality and Comparison Operators

Overload Responsibly

While overloaded operators can be a powerful tool in DSL design, it is important to use them judiciously and carefully. There are many potential applications for overloaded operators. However, it is important to ensure a clear association or analogy between the operation being performed and the chosen operator.

For example, using the `/` operator to concatenate file paths makes sense because it is a common path separator. Similarly, using the unary `+` operator to "add" a single value within a trailing lambda block has

become a standard convention. And using `..` instead of `:` may be acceptable because of its visual similarity.

At some point, however, overloading operators can become confusing or even counterproductive. For example, using the `!` operator to invert a matrix may be a stretch since it has no clear association with matrix inversion. In general, it is important to avoid being too clever when designing a DSL, as users may not have the same associations or understanding of certain symbols or operators.

One solution is to use meaningful infix functions with expressive names, rather than relying solely on overloaded operators. While this may be less concise, it can make code easier to understand and less prone to confusion. Ultimately, the goal should be to create a DSL that is intuitive and easy to use without sacrificing clarity or consistency.

Infix Notation for Functions

Infix notation allows function names to be used like binary operators. Well-known examples in the Kotlin API include `to` for creating pairs and `until` and `downTo` for creating ranges.

The corresponding function must be an extension function with one argument; the receiver becomes the left side, and the argument becomes the right side of the operator. Note that you can still use the normal function call syntax. Here is an example of checking preconditions.

```
infix fun <T> T.shouldBe(expected: T) {
    require(this == expected)
}

fun testIfExpected(s: String) {
    s.shouldBe("expected") // normal syntax
    s shouldBe "expected"  // infix syntax
}
```

Listing 4-24 Infix Function for Checking Preconditions

One weakness of infix notation is that you can't explicitly specify generics. In this case, you can fall back on the normal function call syntax – but users of the DSL may not know this.

The combination of infix and backtick notation allows you to define new "operators," at least visually.

```
infix fun Double.`^`(exponent: Int) =
    this.pow(exponent)

val result = 1.5 `^` 3 // 3.375
```

Listing 4-25 Infix Function Mimicking an Operator

Infix functions can also help make the trailing lambda syntax more readable. For example, we can write a function that takes a pair of lists and combines them using a given lambda.

```
infix fun <P, Q, R> Pair<List<P>,
List<Q>>.zipWith(
    body: (P, Q) -> R
): List<R> = first.zip(second).map { (p, q) ->
body(p, q) }

val numbers = listOf(1, 2, 3, 4)
val strings = listOf("x", "y", "z")
val zippedList = numbers to strings zipWith { i, c
-> "$i -> $c" }
    // [1 -> x, 2 -> y, 3 -> z]
```

Listing 4-26 Infix Functions and Trailing Lambda Syntax

Without using `infix`, we would have to write `(numbers to strings).zipWith { i, c -> "$i -> $c" }` instead.

Functional Interfaces

Imagine you have an interface for checking if a string meets some requirement, with a single abstract method, and you need an anonymous implementation.

```
interface StringCheck {
    fun check(s: String): Boolean
}

val stringCheck = object : StringCheck {
    override fun check(s: String) = s.length < 10
}
```

Listing 4-27 Interface with a Single Abstract Method

Such code is pretty ugly, and far too verbose to expect a DSL user to implement your interface that way. But since the interface has only a single abstract method (abbreviated as "SAM"), it can be written as a *functional interface*, which allows a simplified syntax.

```
fun interface StringCheck {
    fun check(s: String): Boolean
}

val stringCheck = StringCheck { s -> s.length < 10
}
```

Listing 4-28 Functional Interface

Note the `fun` keyword in the interface definition. I think you will agree that this syntax looks much better, making it useful for DSLs. The lambda-like syntax is just syntactic sugar: under the hood, the compiler translates it into an implementation equivalent to the first example.

Generics

Generics are a useful abstraction over concrete types in all sorts of contexts, including DSL design. A specific use case is the

implementation of compile-time checks. Here is a simple example that models currencies, similar to the code shown for extension properties.

```
interface USD
interface Euro

data class Currency<T>(val value: BigDecimal)

val Double.USD
    get() = Currency<USD>(toBigDecimal())
val Double.EUR
    get() = Currency<Euro>(toBigDecimal())

operator fun <T> Currency<T>.plus(that:
Currency<T>) =
   copy(value = this.value + that.value)

val works = 3.1.USD + 4.5.USD    //
Currency(value=7.6)
val worksToo = 2.1.EUR + 4.2.EUR //
Currency(value=6.3)

// this code doesn't compile:
// val oops = 3.1.EUR + 4.5.USD
```

Listing 4-29 Using Phantom Types for Implementing Currencies

You can't add amounts of different currencies together, because the definition of + ensures that both amounts belong to the same currency. The generic type parameter T is called a phantom type, and this code is a very simple example of type-level programming.

Type-level programming is a programming paradigm in which types themselves are used as values that can be manipulated and computed at compile time, rather than being used only to check the correctness of program syntax and logic. In other words, type-level programming uses types to encode computations that are evaluated by the compiler at

compile time rather than at runtime. Type-level programming can be used to achieve a variety of goals, such as improving program performance and enforcing stronger type constraints.

Phantom types are a type-level programming technique in which a type is used to encode additional information about the data it represents without actually storing any data at runtime. Phantom types are types that have no values; they are used only for their type-level information. They can be used to enforce stronger type constraints, such as ensuring that only certain operations are performed on certain types of data. This can help reduce runtime errors and improve the safety of the application.

Resolving Type Erasure Conflicts

Because of type erasure, two different generic functions can end up with the same signature in the JVM. Consider the following example.

```
// does not compile

fun average(list: List<Int>): Double =
    list.sum().toDouble() / list.size

fun average(list: List<Double>): Double =
    list.sum() / list.size
```

Listing 4-30 Conflict due to Type Erasure

The compiler complains that the declarations have the same JVM signature. In Java, the only solution is to rename the functions. In Kotlin, we can keep the same names in the code and just rename them in the JVM.

```
@JvmName("averageInt")
fun average(list: List<Int>): Double =
    list.sum().toDouble() / list.size

@JvmName("averageDouble")
```

```
fun average(list: List<Double>): Double =
    list.sum() / list.size

val intAverage = average(listOf(1, 2, 3)) // 2.0
val doubleAverage = average(listOf(7.0, 8.0, 9.0))
// 8.0
```

Listing 4-31 Fixing the Type Erasure Conflict

Reified Generics

Kotlin offers an interesting feature called *reified generics* that helps to overcome Java's type erasure for generics on the JVM in some situations. Type erasure is a JVM technique that allows Java to check generics at compile time, while discarding type information at runtime. In contrast, reified generics in Kotlin allow type information to be preserved at runtime. This means that developers can perform type-safe operations at runtime without having to resort to workarounds or unsafe casts.

For example, in the currency implementation shown in Listing 4-29, using the `toString()` method produced results such as `Currency(value = 11.2)` without indicating the currency used. However, with reified generics, we can retrieve the type information and use it to improve the output.

```
inline fun <reified T> Currency<T>.display() :
String =
       "${T::class.simpleName!!} $value"

 val someDollars = 42.0.USD.display() // USD 42.0
```

Listing 4-32 Example for Reified Generics

Note that the expression `T::class` is used, which "shouldn't work," because type erasure removes all generic type information at runtime. In the example, however, the function is defined as an `inline` function, and the generic parameter `T` is marked as `reified`. The

details are beyond the scope of this book, but essentially, the inlining allows the compiler to gather the generic type information from the scope where the inlining happens, and make it appear as if no type erasure occurred. Note that inline functions are subject to some restrictions and behave slightly differently than normal functions.

> **Caution** Functions with reified type parameters cannot be called from Java because it has no mechanism for inlining functions, and only inlining allows Kotlin to capture the generic type. This issue will be discussed in more detail in Chapter 13.

Value Classes

Value classes are a feature introduced in Kotlin 1.5 that allows developers to create lightweight, efficient classes that represent simple values. Value classes are designed to be used for values that are commonly used and require little to no additional functionality beyond what is already provided by the underlying data type.

In Kotlin, a value class is defined using the `value` modifier and must have a single primary constructor with exactly one parameter. The parameter must be of a non-nullable type, such as `Int`, `Long`, or `String`. Value classes cannot extend other classes, and they cannot be extended by other classes.

Value classes are optimized for performance, as they are designed to avoid the overhead of creating a full object instance whenever possible, very similar to the handling of primitive types such as `Int`, which are generally used in their unboxed state in the JVM, unless a boxed instance is really required.

One of the main advantages of value classes is that they can be used to create more expressive and type-safe APIs. For example, a value class representing a particular unit of measurement can enforce at compile time that only valid unit conversions are performed, while avoiding the performance hit of using a full-fledged “normal” class.

Note Kotlin's JVM backend requires a `@JvmInline` annotation, which may become obsolete in the future. Also, the single constructor argument restriction may be dropped. This depends on the introduction of the OpenJDK Project Valhalla,[6] which aims to introduce value class functionality to Java.

```
@JvmInline
value class Kilometers(val value: Double)

@JvmInline
value class Miles(val value: Double)

fun Kilometers.toMiles() : Miles =
    Miles(this.value * 0.6214)

val marathonInMiles =
    Kilometers(42.195).toMiles() //
Miles(value=26.219973)
```

Listing 4-33 Physical Units Implemented with Value Classes

Type Aliases

DSL types can become quite complex, especially when dealing with generics. Sometimes, it is useful to give a specific type a shorter name. For example, a typical parser implementation takes a collection of tokens and the current position as input, and returns a parsing result indicating success or failure.

```
sealed class ParseResult<out T> {
    data class Success<T>(val value: T, val pos:
Int)
        : ParseResult<T>()
    data class Failure(val failure: String, val
pos: Int)
```

```
        : ParseResult<Nothing>()
    }

    sealed class Token {
        data class IdToken(val id: String) :
Token()
        //...
    }

    typealias Parser<T> = (List<Token>, Int) ->
ParseResult<T>

    val parseId: Parser<String> = { tokens, pos ->
        when (val token = tokens.getOrNull(pos)) {
            is IdToken -> Success(token.id, pos +
1)
            null -> Failure("EOF", pos)
            else -> Failure("Not an ID", pos)
        }
    }
```

Listing 4-34 Shortening Types with Type Aliases

Using a type alias can enhance the readability of your code and help DSL users to understand what certain DSL elements mean. It can particularly guide their implementation when your DSL allows for custom extensions.

> **Note** Originally, type aliases had to be declared at the top level. However, Kotlin 2.2 introduced experimental support for nested type aliases, which became a stable feature in Kotlin 2.3. Nested type aliases have some limitations: you cannot capture the generics of the enclosing class, nor can you expose `private` types via nested type aliases. However, they can be marked as `internal` or `private` themselves.

Annotations

Although you can write entire DSLs using annotations, they more often play a supporting role, such as describing how certain fields or classes should be handled. Annotations are especially useful when your DSL has a default behavior but needs to account for edge cases or exceptions, such as "do not persist this property".

Another useful application for annotations is code generation. For example, the AutoDSL[7] library uses the information provided via annotations to construct the DSL classes for you.

> **Note** Annotation processors allow custom processor code to be executed during the build process according to the annotations present in the application code. Kotlin provides two annotation processors: the older kapt, which is no longer actively developed, and the recommended Kotlin Symbol Processing API (KSP),[8] which will be discussed in Chapter 12.

Kotlin also includes annotations to support Java interoperability. Since DSL code is often more difficult to call from Java, we'll address this topic in Chapter 13.

Reflection

Sometimes, you need to inspect or deconstruct classes, call unknown methods, or react to annotations. This can be done using reflection. However, if you need more than basic reflection in Kotlin, you must import a separate dependency.

```
// in Gradle
dependencies {
    ...
    implementation(kotlin("reflect"))
}
```

```
// in Maven
<dependencies>
  <dependency>
      <groupId>org.jetbrains.kotlin</groupId>
      <artifactId>kotlin-reflect</artifactId>
  </dependency>
</dependencies>
```

Listing 4-35 Dependency for Full Reflection Support

Context Parameters

Context parameters[9] were introduced in Kotlin 2.2 and became stable in 2.4. The main idea is to provide a function with data from dependencies that are not central to the logic, but rather play a supporting role.

> **Caution** Be careful to avoid potential confusion: context parameters are replacing an older proposal called *context receivers*. The two features appear quite similar and use the same keyword `context`. The main difference between the two is that context parameters are named, whereas context receivers are not. Keep this distinction in mind when searching for online resources, for example.

In principle, you could use normal function arguments (or an extension receiver) instead of context parameters to pass the data. However, the difference is often more of a semantic kind. For example, imagine that you need a logger, a data source, or environment properties. These arguments don't seem as integral to the function's logic as regular arguments. In DSLs, an additional concern is that operators and infix functions can't take more arguments. Consider this attempt to write a DSL for currency amounts.

```
enum class Currency {USD, EUR, GBP}
```

```
data class Amount(val value: Double, val currency:
Currency) {

    infix fun into(newCurrency: Currency): Amount
{
        val factor = TODO()
        return Amount(value * factor, newCurrency)
    }
}

operator fun Double.times(currency: Currency) =
Amount(this, currency)

val dollars = 10.0 * USD
val euros = dollars into EUR
```

Listing 4-36 Incomplete DSL for Currencies

The DSL syntax isn't bad, but the problem is implementing the `into()` infix method, which requires a currency converter – potentially one that needs live updates from a stock exchange. Adding another argument to the method would defeat the purpose of the DSL. Adding an argument to the `Amount` class would complicate its construction and require us to provide a converter even if we don't use the `into()` method in our code.

Context parameters come to the rescue at this point. They enable us to use infix syntax and decouple our Amount class from the necessary converter, which can be supplied externally. For instance, we could use the `with()` function. Listing 4-37 shows the complete example that uses a test converter with a fixed conversion rate.

```
import somePackage.Currency.*

fun interface CurrencyConverter {
    fun factor(from: Currency, to: Currency):
Double
```

```
}

enum class Currency { USD, EUR, GBP }

data class Amount(val value: Double, val currency:
Currency) {

    context(converter: CurrencyConverter)
    infix fun into(newCurrency: Currency): Amount
{
        val factor = converter.factor(currency,
newCurrency)
        return Amount(value * factor, newCurrency)
    }
}

operator fun Double.times(currency: Currency) =
Amount(this, currency)

fun main() {
    val testConverter = CurrencyConverter{ _, _ ->
0.9}
    with(testConverter) {
        val dollars = 10 * USD
        val euros = dollars into EUR
        require(euros.value == 9.0) { "Euros
should be 9.0" }
    }
    val pound = 20.0 * GBP  // the context is only
needed for "into"
}
```

Listing 4-37 Currency DSL Example with Context Parameters

I hope this example has convinced you that context parameters are a useful addition to the Kotlin language. Of course, some details couldn't

be shown here; for instance, you can have multiple context parameters. One convenient feature is that context parameters are automatically passed down the call hierarchy if the next function needs the same parameter. If you don't access a context parameter by its name, you can use an underscore, similar to unused lambda arguments.

If you use a Kotlin version below 2.4, you can enable context parameters by specifying the compiler argument `-Xcontext-parameters` in your preferred build system.

Experimental Features

Kotlin allows you to use experimental features via a compiler argument or the `@OptIn` annotation. Some of these features are quite stable and useful for DSLs, so it may be worth taking the risk of using them even before they become an official part of the language.

Collection Literals

Since Kotlin 2.4, you can write collections as literals.[10] For example, you can write `val myList = ["foo", "bar", "baz"]`. The default type is `List`, but you can specify other types such as `MutableList` or `Set`. This feature is useful for DSL design because it's not just "special syntax," but it uses the new `of` operator, which can be overwritten to create instances of other classes. So, if your DSL defines something "collection-like" such as a vector, you can achieve a very elegant syntax.

To use this feature, set the compiler argument `-Xcollection-literals`.

Contracts

Contracts are an experimental feature introduced in Kotlin 1.3 and expanded upon in Kotlin 2.2.20. They provide the compiler with additional information about the behavior of a function. This enables the compiler to perform actions such as smart casts and completeness checks that it could not achieve with static code analysis alone.

For instance, in the following example, the variable `number` is known to not be `null` after the `isNullOrZero()` check failed, which allows a smart cast.

```
@OptIn(ExperimentalContracts::class)
fun Int?.isNullOrZero(): Boolean {
    contract {
        returns(false) implies (this@isNullOrZero
!= null)
    }
    return this == null || this == 0
}

fun oneHundredDividedBy(number: Int?) {
    if (!number.isNullOrZero()) {
        println("100 / $number = ${100/number}")
// smart cast
    }
}
```

Listing 4-38 A Contract Enabling a Smart Cast

You can also narrow down type information when performing a Boolean check.

```
sealed class ContactData {
    @OptIn(ExperimentalContracts::class)
    fun isEmail(): Boolean {
        contract {
            returns(true) implies
(this@ContactData is Email)
            returns(false) implies
(this@ContactData is Phone)
        }
        return this is Email
    }
```

```
}

data class Email(val address: String):
ContactData() {
    fun send(msg: String) = println("sending
'$msg' to $address")
}

data class Phone(val number: String):
ContactData() {
    fun call(msg: String) = println("calling
$number telling '$msg'")
}

fun sendMessage(cd: ContactData, msg: String) =
    if (cd.isEmail()) cd.send(msg) else
cd.call(msg) // smart casts
```

Listing 4-39 A Contract Narrowing a Type

The given examples run already in Kotlin 1. Kotlin 2.2.0 extended contracts to work on property accessors and certain operators, to support assertions about generic return types and lambda arguments, and added the `returnNotNull()` function. You can enable contracts using the annotation `@OptIn(ExperimentalContracts::class)`, or the compiler argument `-Xopt-in=kotlin.contracts.ExperimentalContracts`. For some of the new additions, you also need the annotation `@OptIn(ExperimentalExtendedContracts::class)` or additional compiler arguments. Contracts can help your DSL to avoid boilerplate checks, casts, and the need to specify generics explicitly.

Summary

This chapter has provided a concise, high-level overview of many of the features of the Kotlin language that are relevant from a DSL perspective. While further details can be explored in the extensive Kotlin documentation, this brief tour should have given you a solid understanding of the extensive toolkit available for building DSLs. Kotlin offers remarkable creative freedom in this area, allowing you to use your imagination to create powerful and intuitive DSLs.

In the next chapters, we will systematically look at different types of DSLs, starting with Algebraic DSLs, which are based on operator overloading.

Footnotes

1 Maslow, Abraham Harold (1966). *The Psychology of Science: A Reconnaissance*. Harper & Row. p. 15–16. ISBN 978-0-8092-6130-7.

2 https://kotlinlang.org/docs/type-safe-builders.html#how-it-works

3 https://github.com/manifold-systems/manifold/tree/master/manifold-deps-parent/manifold-ext

4 https://projectlombok.org/features/experimental/ExtensionMethod

5 https://github.com/rogerkeays/fluent

6 https://openjdk.org/projects/valhalla

7 https://github.com/F43nd1r/autodsl

8 https://kotlinlang.org/docs/ksp-overview.html

9 https://kotlinlang.org/docs/context-parameters.html

10 https://kotlinlang.org/docs/whatsnew24.html#support-for-collection-literals

Part II
DSL Categories

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
D. Gronau, *Creative DSLs in Kotlin*
https://doi.org/10.1007/979-8-8688-2821-8_5

5. Algebraic DSLs

Daniel Gronau[1] ✉
(1) Sandersdorf, Sachsen-Anhalt, Germany

> *We could, of course, use any notation we want; do not laugh at notations; invent them, they are powerful. In fact, mathematics is, to a large extent, invention of better notations.*
> —Richard Feynman, *The Feynman Lectures on Physics*, Vol. 1

A common class of problems deals with objects that exhibit number-like behavior. These "algebraic" structures include complex numbers, quaternions, vectors, matrices, physical and monetary quantities, and many more. With Kotlin, it is relatively easy to write DSLs for these problems, especially if the required operators match the existing ones.

Note As with many other categories of DSLs, there seems to be no established name for this type of DSL in the literature. I think calling them "algebraic DSLs" covers their behavior quite well, but you may find the same concept under different names elsewhere.

The Kotlin API itself contains some nice examples of small algebraic DSLs, such as `BigInteger` and `BigDecimal`, which can be used pretty much like "normal" numbers.

Even if your problem domain does not have an inherent algebraic structure, the concepts and ideas presented in this chapter may still be

partially applicable. For example, consider chemical equations: while their components may not be inherently number-like, they do involve operations such as “addition” of molecules and scalar multiplication by coefficients. This demonstrates the versatility and flexibility of algebraic DSLs, as they can be adapted and used in various domains beyond traditional numerical applications.

Case Study: A DSL for Complex Numbers

To keep things realistic, we will use an existing implementation that you might find in a production environment. We will use the Apache Commons Numbers library, which is written in Java. The implementation behaves pretty much as expected. Listing 5-1 showcases how to use the library.

```
val a = Complex.ofCartesian(3.0, 4.0)
val b = Complex.ofCartesian(5.0, 6.0)
val c = a.add(b)       //  8 + 10i
val d = a.times(b)     // -9 + 38i
val e = a.pow(b)       // -1.860 + 11.837i
```

Listing 5-1 Complex Numbers in Apache Commons Numbers

There are static factory methods for creating `Complex` objects, such as `Complex.ofCartesian()` and `Complex.ofPolar()`. You can then use methods like `add()` and `times()` on these objects.

If we follow the steps from Chapter 3, we should now come up with an ideal syntax. This is easy, since in mathematics, we would simply write `3.0 + 4.0i` instead of `Complex.ofCartesian(3.0, 4.0)`. I think it is acceptable to have no special syntax for other ways of initializing complex numbers, such as the polar definition. If we consider that `4.0i` is actually a multiplication, we are already there: a syntax like `3.0 + 4.0*i` can be implemented in Kotlin.

First, we need to define the imaginary unit `i` with `val i = Complex.I`. Next, the arithmetic operations can be written using

operator overloading. We need to define operations not only between two `Complex` instances but also between `Complex` and `Double` and vice versa. Listing 5-2 gives an example of the addition and subtraction operations.

```
operator fun Complex.plus(that: Complex): Complex
    = this.add(that)
operator fun Complex.plus(that: Double): Complex
    = this.add(that)
operator fun Double.plus(that: Complex): Complex
    = that.add(this)

operator fun Complex.minus(that: Complex): Complex
    = subtract(that)
operator fun Complex.minus(that: Double): Complex
    = subtract(that)
operator fun Double.minus(that: Complex): Complex
    = Complex.ofCartesian(this,
0.0).subtract(that)
```

Listing 5-2 Additions and Subtractions for Complex Numbers

The other operations can be implemented the same way. Note that the functions specify explicit return types, which avoids nullability issues when using results from Java APIs. A small improvement would be a helper method for converting double to complex numbers, instead of relying on the clumsy `Complex.ofCartesian()` factory.

Next, we need to support the negation of complex numbers, and the unary plus as well, for symmetry reasons. This looks a lot like the code above.

```
operator fun Complex.unaryPlus(): Complex = this
operator fun Complex.unaryMinus(): Complex =
this.negate()
```

Listing 5-3 Identity and Negation of Complex Numbers

If you can't find a suitable operator to overload, you can use infix functions instead. In our example, we could define `pow` as an exponentiation operation, so we could write, for example, `3.0 + 4.0*i pow 3.0`. The infix operations have a lower priority than the overloaded operators, so the example calculation would be interpreted as `(3.0 + 4.0*i) pow 3.0`.

> **Tip** Note that the infix extension function can have the same name and arguments as an existing function – here `Complex.pow()` – but in this case, it will only be called when using infix notation; otherwise, the original method will be executed.

```
infix fun Complex.pow(that: Complex): Complex =
this.pow(that)
infix fun Complex.pow(that: Double): Complex =
this.pow(that)
```

Listing 5-4 Exponentiation of Complex Numbers

Alternatively, we could have used `` `^` `` in backtick notation, which may be more readable than `pow`, but is harder to type.

And that's already it; we have a basic DSL for complex numbers. Here is the complete code.

```
import org.apache.commons.numbers.complex.Complex

val i = Complex.I

operator fun Complex.unaryPlus(): Complex
   = this
operator fun Complex.unaryMinus(): Complex
    = this.negate()

operator fun Complex.plus(that: Complex): Complex
    = this.add(that)
```

```
operator fun Complex.plus(that: Double): Complex
    = this.add(that)
operator fun Double.plus(that: Complex): Complex
    = that.add(this)

operator fun Complex.minus(that: Complex): Complex
    = this.subtract(that)
operator fun Complex.minus(that: Double): Complex
    = this.subtract(that)
operator fun Double.minus(that: Complex): Complex
    = fromDouble(this).subtract(that)

operator fun Complex.times(that: Complex): Complex
    = this.multiply(that)
operator fun Complex.times(that: Double): Complex
    = this.multiply(that)
operator fun Double.times(that: Complex): Complex
    = that.multiply(this)

operator fun Complex.div(that: Complex): Complex
    = this.divide(that)
operator fun Complex.div(that: Double): Complex
    = this.divide(that)
operator fun Double.div(that: Complex): Complex
    = fromDouble(this).divide(that)

infix fun Complex.pow(that: Complex): Complex
    = this.pow(that)
infix fun Complex.pow(that: Double): Complex
    = this.pow(that)

private fun fromDouble(d: Double): Complex
    = Complex.ofCartesian(d, 0.0)
```

Listing 5-5 DSL for Complex Numbers

Now our initial usage example can be written as

```
val a = 3.0 + 4.0*i
val b = 5.0 + 6.0*i
val c = a + b    //  8 + 10i
val d = a * b    // -9 + 38i
val e = a pow b  // -1.860 + 11.837i
```

Listing 5-6 Complex Number DSL – Usage Example

Using complex numbers now feels as intuitive as using one of the built-in number types.

Case Study: Modular Arithmetic – Using a Context

Imagine you're faced with large modular arithmetic calculations. Often, simply calculating the modulo after an operation isn't enough. Certain operations, such as division, differ from standard arithmetic. In addition, neglecting to optimize operations such as multiplication and exponentiation can result in slower performance.

To solve this problem, you decide to create a DSL tailored to modular arithmetic. However, a problem arises: where do you specify the modulus for these calculations? A global variable would be a precarious and unsafe solution. Embedding the modulus in each operand proves tedious and raises the question of how to handle operands with different moduli in a single operation.

A common solution would be to define a class that holds the modulus, and make the parts of the DSL members of that class, so that they all have access to the modulus information.

```
import java.math.BigInteger
import java.math.BigInteger.ONE
import java.math.BigInteger.ZERO

data class Modulus(val modulus: BigInteger) {
```

```
    init { require(modulus > ONE) }
    constructor(modulus: Long):
this(BigInteger.valueOf(modulus))

    @JvmInline
    value class Residue(val n: BigInteger)

    val Long.m
        get() = Residue(remainder(toBigInteger()))

    val Int.m
        get() = Residue(remainder(toBigInteger()))

    operator fun Residue.plus(that: Residue) =
        Residue(remainder(this.n + that.n))

    operator fun Residue.minus(that: Residue) =
        Residue(remainder(this.n - that.n))

    operator fun Residue.times(that: Residue) =
        Residue(remainder(this.n * that.n))

    operator fun Residue.div(that: Residue) =
        Residue(remainder(this.n *
that.n.modInverse(modulus)))

    private fun remainder(n: BigInteger) = when {
        n < ZERO -> (n % modulus) + modulus
        else -> n % modulus
    }
}
```

Listing 5-7 Modular Calculations Using a Surrounding Class

The example is straightforward. Residues can be created from integer numbers using an extension property `m`. Then, the required operations on the residues are defined as usual. Note that the division

operation will fail when the divisor is not invertible. In order to use this DSL, we must bring the `Modulus` class into scope using the `with()` function.

```
val x = with(Modulus(7)) {
    val a = 3.m + 5.m  // Residue(n=1)
    val b = 3.m - 5.m  // Residue(n=5)
    val c = 3.m * 5.m  // Residue(n=1)
    val d = 3.m / 5.m  // Residue(n=2)
    a + b + c + d
}
println(x)             // Residue(n=2)

with(Modulus(10)) {
    println(3.m + 5.m) // Residue(n=8)
    println(3.m - 5.m) // Residue(n=8)
    println(3.m * 5.m) // Residue(n=5)
    println(3.m / 7.m) // Residue(n=9)
    println(3.m / 5.m) // ArithmeticException:
BigInteger not invertible
}
```

Listing 5-8 Modular Calculation Example

The syntax can be further improved by introducing a helper function to provide the scope.

```
fun <R> modulus(m: Long, body: Modulus.() -> R) =
     with(Modulus(m)) {
        body()
    }

...

val x = modulus(7) {
    val a = 3.m + 5.m // Residue(n=1)
```

```
    val b = 3.m - 5.m // Residue(n=5)
    val c = 3.m * 5.m // Residue(n=1)
    val d = 3.m / 5.m // Residue(n=2)
    a + b + c + d
}
println(x)              // Residue(n=2)
```

Listing 5-9 Helper Function for Modular Calculations

While this solution works, its scalability is limited because the entire DSL is contained within a class. Although you could alleviate this by converting some functions to extension functions to streamline the class, the operators must remain within the class. This limitation is due to the fact that the operators are already extension functions and can only have one receiver, namely their first operand. Another challenge arises when the DSL requires several unrelated sources of information, forcing them to be combined into a single class. Essentially, these problems occur because the DSL is tightly coupled to its enclosing class.

The Kotlin language feature called *context parameters* was specifically designed to address scenarios like this. They provide a flexible and controlled way to bring contextual data into scope. For more information, see Chapter 4. Here is a version of the previous DSL using context parameters.

```
import java.math.BigInteger
import java.math.BigInteger.ONE
import java.math.BigInteger.ZERO

data class Modulus(val modulus: BigInteger) {
    init { require(modulus > ONE) }
    constructor(modulus: Long) :
this(BigInteger.valueOf(modulus))
}

@JvmInline
value class Residue(val n: BigInteger)
```

```
context(_: Modulus)
val Long.m
    get() =
Residue(remainder(this.toBigInteger()))

context(_: Modulus)
val Int.m
    get() =
Residue(remainder(this.toBigInteger()))

context(_: Modulus)
operator fun Residue.plus(that: Residue) =
    Residue(remainder(this.n + that.n))

context(_: Modulus)
operator fun Residue.minus(that: Residue) =
    Residue(remainder(this.n - that.n))

context(_: Modulus)
operator fun Residue.times(that: Residue) =
    Residue(remainder(this.n * that.n))

context(m: Modulus)
operator fun Residue.div(that: Residue) =
    Residue(remainder(this.n *
that.n.modInverse(m.modulus)))

context(m: Modulus)
private fun remainder(n: BigInteger) = when {
    n < ZERO -> (n % m.modulus) + m.modulus
    else -> n % m.modulus
}

fun <R> modulus(m: Long, body: context(Modulus) ()
-> R) =
```

```
    with(Modulus(m)) {
        body(this)
    }
```

Listing 5-10 Modular Calculations Using Context Parameters

The usage pattern hasn't changed; you can use `with()` in exactly the same way to provide the `Modulus` instance, or you can use the `modulus()` helper function. You can easily write new functions that use the context, and within those functions, all existing operations will work as expected, because the required context parameter is passed down to them.

```
context(_: Modulus)
fun square(n: Residue) = n * n

...

val z = modulus(7) {
    square(3.m + 5.m) + square(3.m - 5.m)
}
println(z) // Residue(n=5)
```

Listing 5-11 Custom Extension of the Context Parameter-Based DSL

In my opinion, context receivers provide an elegant solution in situations where algebraic DSLs need additional information from the environment.

Java Interoperability

Java doesn't allow operator overloading, and extension methods become normal static methods with the receiver as the first argument. This means that the DSLs will definitely look less elegant in Java.

In case of the first case study, since the underlying Apache Commons Numbers library itself is written in Java, we are probably better off using its methods. However, our DSL still works and is quite

easy to use if you know how to translate the operators into method names: instead of `val a = 3.0 + 4.0*i`, you would have to write `Complex a = plus(3.0, times(4.0, getI()));` inside a Java class.

No Surprises, Please!

Algebraic DSLs are usually easy to implement, and often elegant. Therefore, you might be tempted to use them in situations where they are not a good fit.

Using algebraic notation when the domain lacks an underlying algebraic structure can be surprising and confusing for users of the DSL. Even when there is an underlying algebraic structure, it can be confusing when a common notation already exists, as with mathematical sets. This problem is exacerbated by the fact that the range of operators and their precedence are fixed in Kotlin. Additionally, if an operation deviates from its "normal" behavior, it increases the likelihood of usage errors. One example of this are domains with non-commutative multiplication, such as quaternions and matrices, which may catch less experienced users off guard.

Just because you think the use of certain operators is intuitive and obvious doesn't mean your users necessarily agree. Therefore, it is crucial to exercise good judgment, listen to feedback, and adhere to the *Principle of Least Surprise* when designing algebraic DSLs, rather than blindly adopting them because of their ease of implementation.

Summary

The case studies presented in this chapter serve to illustrate that creating algebraic DSLs is usually not difficult and can greatly enhance usability. They can transform an unreadable mess of nested function calls into a clearly structured, easy-to-understand formula. Algebraic DSLs work well in conjunction with other types of DSLs, and the introduction of context parameters in Kotlin has opened up additional

applications. This makes algebraic DSLs indispensable for DSL designers.

Common Applications

- Defining operations
- Data transformation

Pros

- Easy to write
- Intuitive to use
- Can use infix functions as operator replacement

Cons

- Limited selection of operators
- Operator precedence can’t be changed
- Possible name collisions with other DSLs
- Difficult to use from Java client code

What’s Next?

The next two chapters cover object initialization, beginning with the Builder Pattern, which is widely used in Java. Although there are good alternatives in Kotlin, this pattern remains relevant and allows for useful enhancements.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
D. Gronau, *Creative DSLs in Kotlin*
https://doi.org/10.1007/979-8-8688-2821-8_6

6. Builder Pattern DSLs and Method Chaining

Daniel Gronau[1] ✉
(1) Sandersdorf, Sachsen-Anhalt, Germany

> *When we mean to build, We first survey the plot, then draw the model; And when we see the figure of the house, Then must we rate the cost of the erection.*
> —William Shakespeare, *Henry IV*, Part 2, Act 1, Scene 3

A common task is to initialize a complex, sometimes nested object. The classic solution is the *builder pattern*: the builder class is mutable, uses method chaining (also called "fluent interfaces") to simplify value assignment, and contains a terminal build method that constructs the domain object. Note that while builders are the most prominent use case, method chaining can be used in other contexts, and the points made for builders can usually be applied to such cases as well.

This chapter covers the classic builder pattern as it is often used in Java. Next, we will show how named arguments can often replace the builder pattern in Kotlin, especially when constructing smaller classes. We will also briefly discuss nested builders. Next, we will address the issue of mandatory fields in builders by introducing the *type-safe builder pattern*, and we will also cover the related *counting builder*. Finally, we will discuss why builders are not as widely used in Kotlin as they are in Java.

The Classical Builder

As an example of a classical builder with realistic complexity, we can use `java.net.http.HttpRequest`, which is written in Java. A typical builder call could look like this.

```
val request = HttpRequest
    .newBuilder(URI.create("https://acme.com:9876/products"))
    .GET()
    .header("Content-Type",
        "application/x-www-form-urlencoded; charset=UTF-8")
```

```
    .header("Accept-Encoding", "gzip, deflate")
    .timeout(Duration.ofSeconds(5L))
    .build()
```

Listing 6-1 Initializing an HttpRequest

As this example shows, a builder has three parts:

- **Builder Initialization**: The builder needs a starting point, which can be provided by a constructor or factory method. This initializes the builder instance and prepares it for data collection.
- **Data Collection**: Through method chaining, the builder instance collects the necessary data and configuration options. Each method call adds a specific attribute or behavior to the final object being constructed.
- **Target Object Construction**: To complete the construction process, a terminal method, often called `build()`, is called. This method performs any necessary validation on the collected data and proceeds to instantiate the target object with the provided configuration.

By following this pattern, builders provide a structured approach to constructing objects with customizable parameters. This provides a convenient and readable way to initialize complex objects in a flexible manner.

If you want to write a builder in Kotlin, you can take advantage of the `apply()` or `also()` method. Consider this example.

```
data class Person(
    val firstName: String,
    val lastName: String,
    val age: Int?
)

class PersonBuilder {
    private var firstName: String? = null
    private var lastName: String? = null
    private var age: Int? = null

    fun setFirstName(firstName: String): PersonBuilder {
        this.firstName = firstName
        return this
    }

    fun setLastName(lastName: String): PersonBuilder {
        this.lastName = lastName
```

```
        return this
    }

    fun setAge(age: Int): PersonBuilder {
        this.age = age
        return this
    }

    fun build() = Person(firstName!!, lastName!!, age)
}
```

Listing 6-2 Simple Builder Implementation

This looks pretty much like you would write the builder in Java, but using `apply()` is shorter and more idiomatic.

```
class PersonBuilder {
    private var firstName: String? = null
    private var lastName: String? = null
    private var age: Int? = null

    fun setFirstName(firstName: String) = apply {
        this.firstName = firstName
    }

    fun setLastName(lastName: String) = apply {
        this.lastName = lastName
    }

    fun setAge(age: Int) = apply {
        this.age = age
    }

    fun build() = Person(firstName!!, lastName!!, age)
}
```

Listing 6-3 Builder Implementation Using apply()

Named Arguments Instead of Builders

In Kotlin, you may not need a builder because the language has features like named and default arguments that can provide similar functionality. Consider this implementation of a color in RGBA space.

```
data class Color(
    val red: Int,
    val green: Int,
    val blue: Int,
    val alpha: Int
)
val c1 = Color(0, 100, 130, 200)
```

Listing 6-4 Instantiating a Class

Written like this, the four `Int` values may be a little confusing to read, especially when you expect only three values, as is usual for RGB colors. However, instead of relying on a builder, you can clarify the meaning by rewriting the last line as follows.

```
val c2 = Color(
    red = 0,
    green = 100,
    blue = 130,
    alpha = 200
)
```

Listing 6-5 Instantiating a Class with Named Arguments

The syntax is different, but the functionality of this code is very similar to a builder. Of course, as with builders, the arguments can be listed in any order. One notable difference is that for named arguments, each value can only be set once, while a builder allows you to overwrite already set values, which can be confusing. Unlike builders, omitting mandatory values for named arguments will result in compile-time errors.

The `Color` class is immutable, but if you want to get modified copies, you can use the `copy()` method.

```
val c3 = Color(0, 100, 130, 200)
val c4 = c3.copy(red = 120)
```

Listing 6-6 Using the copy() Method

The `copy()` method is auto-generated for all data classes. In Java, you would have to write a method equivalent to `fun withRed(r: Int): Color`, sometimes called a *wither* in analogy to "getter" and "setter". Such methods may also be necessary in Kotlin when you can't use data classes, but they are less common than in Java.

Nesting Builders

When an object has complex components, it makes sense to have not only a top-level builder but also builders for those components, their own subcomponents, and so on. A typical example of such nested builders is the DSL of the KotlinPoet[1] library.

```
val file = FileSpec.builder("pkg", "HelloWorld")
  .addType(
    TypeSpec.classBuilder("Greeter")
      .primaryConstructor(
        FunSpec.constructorBuilder()
          .addParameter("name", String::class)
          .build()
      )
      .addProperty(
        PropertySpec.builder("name", String::class)
          .initializer("name")
          .build()
      )
      .addFunction(
        FunSpec.builder("greet")
          .addStatement("println(%P)", "Hello, \$name")
          .build()
      )
      .build()
  )
  .addFunction(
    FunSpec.builder("main")
      .addParameter("args", String::class, VARARG)
      .addStatement("%T(args[0]).greet()",
          ClassName("pkg","Greeter"))
      .build()
  )
  .build()
```

Listing 6-7 Nested Builders in KotlinPoet

As this example shows, the need to call `build()` at the end of each nested builder leads to a lot of visual clutter. To avoid this problem, some DSLs make builder nesting more convenient by having two versions of each nested method: one version that takes the constructed object as an argument as usual, and another version that

takes a builder of the object instead. This way, the user doesn't have to repeatedly call `build()` methods for the nested builders.

Flattening Instead of Nesting

An alternative to nesting builders is to handle everything in the top-level builder, by putting the nested content between a start and end method. In KotlinPoet, control flows are implemented this way.

```
val funSpec = FunSpec.constructorBuilder()
    .addParameter("value", String::class)
    .beginControlFlow("require(value.isNotEmpty())")
    .addStatement("%S", "value cannot be empty")
    .endControlFlow()
    .build()
```

Listing 6-8 Flattened Builder Call in KotlinPoet

This approach can make the DSL code more readable, but it requires more discipline on the part of the user to ensure that the start and end methods are placed properly. To give an example implementation, consider a `Person` class that contains a name, a phone number, and a list of contacts that also have a name and optionally a phone.

```
data class Contact(
    val name: String,
    val phone: String?)

data class Person(
    val name: String,
    val phone: String,
    val contacts: List<Contact>)

class PersonBuilder {
    private var name: String? = null
    private var phone: String? = null
    private var addingContact = false
    private var contactName: String? = null
    private var contactPhone: String? = null
    private val contacts: MutableList<Contact> =
mutableListOf()

    fun beginContact() = apply {
```

```
        require(!addingContact)
        addingContact = true
    }

    fun endContact() = apply {
        require(addingContact)
        contacts.add(Contact(contactName!!, contactPhone))
        contactName = null
        contactPhone = null
        addingContact = false
    }

    fun setName(name: String) = apply {
        if (addingContact) this.contactName = name
        else this.name = name
    }

    fun setPhone(phone: String) = apply {
        if (addingContact) this.contactPhone = phone
        else this.phone = phone
    }

    fun build(): Person {
        require(!addingContact)
        return Person(name!!, phone!!, contacts)
    }
}
```

Listing 6-9 Implementing a Flattened Builder

And this is how the DSL could be used.

```
val superman = PersonBuilder()
    .setName("Superman")
    .beginContact()
    .setName("Wonder Woman")
    .endContact()
    .setPhone("555-3213-125")
    .beginContact()
    .setName("Lois Lane")
    .setPhone("555-4112-423")
    .endContact()
    .build()
```

Listing 6-10 Using the Flattened Builder

The process flow of a flattened builder can also be thought of as a very simple state transition, namely from the outer layer to the inner layer and back. Chapter 8 demonstrates techniques to safely implement such state transitions, ensuring that the code won't compile if the start and end methods are placed incorrectly.

Although there are certainly valid use cases for flattening builders, the nesting approach is simpler, both conceptually and in terms of implementation, and usually scales better. Therefore, nested builders should be preferred in most cases.

Nesting with Varargs

Method chaining using varargs can lead to quite elegant DSLs when building a tree-like structure where each node has only one type of child. For example, consider mathematical expressions that need to be converted into *Reverse Polish Notation* (RPN). Unlike in infix notation, in RPN, operators come last, operate on the numbers previously placed on the stack, and put the result back on the stack. For instance, the infix expression `(3 + 4) * 6` would be written as `3 4 + 6 *` in RPN. The `+` would add `3` and `4` together and place the result `7` on the stack. The stack now contains `7 6 *`, and after the multiplication, we obtain the expected result of `42`. This notation is used in stack-based languages, compilers, and older pocket calculators, and has the advantage that no parentheses are needed.

First, we need the tokens for the RPN stack. To simplify the code, we will only consider sums and products.

```
sealed interface Token {
    data object PLUS : Token
    data object TIMES : Token
    data class VAL(val value: Double) : Token
}
```

Listing 6-11 Tokens for RPN Stack

We want to write a builder that allows sums and products of arbitrary arity to be nested, and the `build` method should convert the expression tree into an RPN stack.

```
import myPackage.Token.VAL

sealed interface Expression {

    class Sum(vararg val ops: Expression) : Expression {
        override fun build(): List<Token> = when {
                  ops.isEmpty() -> listOf(VAL(0.0))
            else -> ops.first().build() +
```

```
                    ops.drop(1).flatMap { it.build() +
Token.PLUS }
        }
    }

    class Product(vararg val ops: Expression) : Expression {
        override fun build(): List<Token> = when {
                    ops.isEmpty() -> listOf(VAL(1.0))
            else ->ops.first().build() +
                    ops.drop(1).flatMap { it.build() +
Token.TIMES }
        }
    }

    class Val(val value: Double) : Expression {
                override fun build(): List<Token> =
listOf(VAL(value))
    }

    fun build(): List<Token>
}
```

Listing 6-12 A DSL for Mathematical Expressions

Note that `Sum` and `Product` take other `Expression`s as varargs. The conversion in their `build` methods is rather crude, for the sake of brevity. A real-world implementation involving more operations would typically use Dijkstra's Shunting Yard Algorithm[2] to convert from infix notation to RPN.

The calculation of `6+3*7+((4+5)*9)` would be expressed in the DSL as follows.

```
val rpn = Sum(
    Val(6.0),
    Product(
        Val(3.0),
        Val(7.0)
    ),
    Product(
        Sum(
            Val(4.0),
            Val(5.0)
        ),
        Val(9.0)
```

```
    )
).build()

// [VAL(value=6.0), VAL(value=3.0), VAL(value=7.0), TIMES,
PLUS,
//  VAL(value=4.0), VAL(value=5.0), PLUS, VAL(value=9.0),
TIMES, PLUS]
```

Listing 6-13 Using Vararg Nesting

Therefore, the RPN stack is `6 3 7 * + 4 5 + 9 * +`, and you can easily check that this yields the expected result of `207`. As mentioned above, the vararg style isn't always feasible, but when it can be used, it often improves readability.

The Type-Safe Builder Pattern

A common problem with builders is the inability to enforce the setting of mandatory fields. While it's possible to check for these conditions in the build method, it would be better if the compiler could prevent incomplete objects from being built. To achieve this, we can use *type-level programming*, although it requires some boilerplate code.

By using generics to track the state of mandatory fields, the build method can be adapted to accept only builders with all mandatory values set. For example, consider the following class for a product that requires a product ID, name, and price, while the other attributes are optional.

```
data class Product(
    val id: UUID,
    val name: String,
    val price: BigDecimal,
    val description: String?,
    val images: List<URI>
)
```

Listing 6-14 Product Item Class

The first requirement for our builder is three classes to represent the state of the mandatory fields. They are similar to `Optional`, except that the empty and full states are represented by different subclasses. The type parameter `T: Any` was used because it prevents `T` from being inhabited by a nullable type.

```
sealed class Val<T: Any>
```

```
class Without<T: Any> : Val<T>()

class With<T: Any>(val value: T): Val<T>()
```

Listing 6-15 Helper Classes for the Type-Safe Builder

With the help of these classes, we can start to write the type-safe builder for `Product`.

```
data class ProductBuilder<
      ID: Val<UUID>,
      NAME: Val<String>,
      PRICE: Val<BigDecimal>> private constructor(
    val id: ID,
    val name: NAME,
    val price: PRICE,
    val description: String?,
    val images: List<URI>) {
    ...
}
```

Listing 6-16 Signature of the Type-Safe Builder

The generic signature looks complicated, but the idea behind it is simple: Each mandatory field has its own generic type parameter that keeps track of whether it is already set or not. The constructor was made private because we want to make sure we start with an empty builder. As a replacement, we implement a companion object that simulates a constructor using the `invoke()` operator.

```
data class ProductBuilder<
      ID: Val<UUID>,
      NAME: Val<String>,
      PRICE: Val<BigDecimal>> private constructor(
    val id: ID,
    val name: NAME,
    val price: PRICE,
    val description: String?,
    val images: List<URI>) {

    companion object {
        operator fun invoke() = ProductBuilder(
            id = Without(),
            name = Without(),
```

```
            price = Without(),
            description = null,
            images = listOf()
        )
    }

    fun id(uuid: UUID) =
        ProductBuilder(With(uuid), name, price, description,
images)

    fun name(name: String) =
        ProductBuilder(id, With(name), price, description,
images)

    fun price(price: BigDecimal) =
        ProductBuilder(id, name, With(price), description,
images)

    fun description(desc: String) =
       copy(description = desc)

   fun addImage(image: URI) =
        copy(images = images + image)
}
```

Listing 6-17 Type-Safe Builder for Product

The inferred return type of this `invoke()` operation is `ProductBuilder<Without<UUID>, Without<String>, Without<BigDecimal>>`, which fortunately we don't have to write out. When an optional field is set, these type parameters don't change, but when a mandatory field is set, the signature changes from `Without` to `With` for that particular field. Since the mandatory field setters return a builder with a changed signature, we can't use the `copy()` method in these cases if we don't want to use casts.

Of course, a crucial part of our DSL is still missing: the `build()` function. But we can't write it as part of the builder class, because it needs to check the generic signature. It *must* be an extension method, because only there can you fix the type parameters to concrete types, which is known as *type narrowing*.

```
fun ProductBuilder<With<UUID>, With<String>,
With<BigDecimal>>.build() =
```

```
    Product(id.value, name.value, price.value, description,
images)
```

Listing 6-18 The build() Function for ProductBuilder

Note that you can access the `value` fields of the `With` instances, because the type inference matches on the narrowed type. Now we have a builder with a `build()` method that can only be called if all mandatory fields are set.

```
val product = ProductBuilder()
    .id(UUID.randomUUID())
    .name("Comb")
    .description("Green Plastic Comb")
    .price(12.34.toBigDecimal())
    .build()
```

Listing 6-19 ProductBuilder Usage Example

You can check that the code no longer compiles after removing one of the mandatory fields.

> **Note** The type-safe builder pattern was pioneered by Rafael Ferreira[3] in Scala, using ideas from Haskell. The code shown here is based on the implementation by Daniel Sobral.[4]

Counting Builder

I have to admit that this is one of the more exotic builder variations, but I decided to include it because it is an interesting technique, and because this type of construction might be useful in other contexts. Consider the following `Polygon` class, which could be part of a graphics library.

```
import java.awt.geom.Point2D

data class Polygon(val points: List<Point2D>)
```

Listing 6-20 Polygon Class

However, a problem arises when we want to ensure that polygons are constructed with at least three points. To solve this problem, we could create a builder that counts the number of points added and only allows the construction of polygons with three or more points.

While the straightforward solution is to check the number of points at runtime, we can be more secure by preventing an invalid builder from being created at

compile time, which again makes use of *type-level programming*. Although this may seem unusual at first, the implementation is quite simple.

```
sealed interface Nat
interface Z : Nat
interface S<N : Nat> : Nat

class PolygonBuilder<N : Nat> private constructor() {

    companion object {
        operator fun invoke() =
            PolygonBuilder<Z>()
    }

    val points: MutableList<Point2D> =
        mutableListOf()

    @Suppress("UNCHECKED_CAST")
    fun add(point: Point2D) =
        (this as PolygonBuilder<S<N>>)
            .also { points += point }
}

fun <N : Nat> PolygonBuilder<S<S<S<N>>>>.build() =
Polygon(points)
```

Listing 6-21 A Counting Builder

First, we create a sealed interface `Nat` to represent the natural numbers, and two sub-interfaces: `Z` representing zero, and `S<N>` representing the successor of a natural number `N`. For example, the number 3 would be written as `S<S<S<Z>>>`. This is called the *Peano Representation* of the natural numbers. Note that even if we don't know the innermost part of `S<S<S<...>>>`, we can still deduce that the given number is greater than or equal to 3, which is exactly what we need for our validation. These recursively constructed numbers are used by the builder class as a generic "counter" parameter, holding the number of items in the list.

Note When asked to count, the usual answer is "one, two, three...," not "zero, successor of zero, successor of successor of zero...," so you might wonder where the strange *Peano Representation* comes from. The theory behind it is actually a fundamental part of mathematics. In 1889, Giuseppe Peano published his famous nine axioms[5] to define natural numbers and their properties in a formal way, and

the Peano Representation follows directly from these axioms. In his paper, the first axiom states the existence of zero; the next four axioms cover basic properties of equality (it is reflexive, symmetric, transitive, and closed), but the following four axioms rely crucially on the use of the successor function:

6) For every natural number, its successor is also a natural number.

7) If the successors of two natural numbers are equal, then the numbers themselves are equal.

8) Zero is not a successor of a natural number.

9) Any natural number can be reached from zero by repeatedly applying the successor function (this is also known as "induction").

That's why, from a mathematical point of view, the Peano Representation can be seen as a fundamental way of writing natural numbers, while our usual number systems (decimal, binary, hexadecimal...) could be regarded as "convenient abbreviations."

Similar to the type-safe builder pattern, the builder class must hide its constructor, because a call like `PolygonBuilder<S<S<Z>>>()` would initialize the builder with the wrong counter. That's why we simulate a constructor using the `invoke()` operator in the companion object, which only returns builders with a counter correctly initialized to 0. The `add()` method appends a point to the list, as well as casts the instance to one with an incremented counter. This is safe because the counter is a phantom type. Alternatively, we could have constructed a new builder object on each `add()` call.

The last ingredient is the `build()` method, which must be an extension function for the same reasons as in the type-safe builder example. The function "counts" the points by checking the type signature of the builder. This is how our builder might be used.

```
val polygon = PolygonBuilder()
    .add(Point2D.Double(1.0, 2.3))
    .add(Point2D.Double(2.1, 4.5))
    .add(Point2D.Double(2.4, 5.0))
    .build()
```

Listing 6-22 Usage Example for the Counting Builder

If one of the `add()` calls is removed, the code will no longer compile because the type of the `PolygonBuilder` no longer matches the signature of the `build()` extension function. Of course, you can use this pattern to count more than one thing, and you can also combine it with the type-safe builder pattern.

Builders with Multiple Stages

It is possible to build objects in stages. However, since there are several ways to implement this use case, and since these techniques are not only applicable to builders, Chapter 8 covers this topic in detail.

Summary

The builder pattern is quite popular in Java – there are even libraries like Project Lombok[6] that generate builders for you. The downside is that builders are quite inflexible and may not be very safe to use, although variations like the type-safe builder pattern can help. In Kotlin, using named and default parameters can already provide functionality similar to a builder. The next chapter introduces another common approach in Kotlin that has some advantages over the classic builder pattern.

Common Applications

- Data creation and initialization
- Configuration management
- Workflow orchestration
- Code generation
- Testing
- Logging

Pros

- Easy to write
- Applicable to a wide range of design tasks
- Variations of the pattern can fix some of its shortcomings
- Can be auto-generated (e.g., using Project Lombok)
- Easy to use from Java client code

Cons

- Often not the most natural syntax for the problem at hand
- Nested builders don't look nice
- Inflexible structure
- Boilerplate code: need for a `build()` method
- Assignments are disguised as method calls

What's Next?

The builder pattern is not the most idiomatic way to construct, initialize, or configure data in Kotlin. In the next chapter, we will see how the loan pattern can accomplish the same tasks, while offering more freedom, better support for nesting, and greater readability than builders.

Footnotes

1 https://square.github.io/kotlinpoet

2 Dijkstra, Edsger (1961). *Algol 60 translation: An Algol 60 translator for the X1 and making a translator for Algol 60*. *Stichting Mathematisch Centrum. Rekenafdeling*. Stichting Mathematisch Centrum.

3 Rafael Ferreira, Type-safe Builder Pattern in Scala, http://blog.rafaelferreira.net/2008/07/type-safe-builder-pattern-in-scala.html

4 Daniel Sobral, Type Safe Builder Pattern, http://dcsobral.blogspot.de/2009/09/type-safe-builder-pattern.html

5 Guiseppe Peano, *Arithmetices principia, nova methodo exposita*.

6 https://projectlombok.org

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
D. Gronau, *Creative DSLs in Kotlin*
https://doi.org/10.1007/979-8-8688-2821-8_7

7. Loan Pattern DSLs

Daniel Gronau[1] ✉
(1) Sandersdorf, Sachsen-Anhalt, Germany

Neither a borrower nor a lender be,

For loan oft loses both itself and friend,

And borrowing dulls the edge of husbandry.
—William Shakespeare, *Hamlet*, Act 1, Scene 3

This pattern is known by several different names, one of which is "builder pattern." This can be particularly confusing because it is similar to, and often is used instead of, the classic builder pattern discussed in the previous chapter. Another commonly used term is "lambdas with receivers," but I would argue that this describes an implementation detail rather than the underlying idea. To avoid confusion, and to provide a more accurate description of the underlying mechanism, this book will refer to it as the "loan pattern DSL."

The loan pattern DSL uses a mutable builder class that exposes its members through the loan pattern, which was introduced in Chapter 4, in the section "Extensions and Receivers." This approach leverages the power of extension methods and trailing lambda syntax in Kotlin. It is particularly useful for creating deeply nested structures. In such cases, the code can quickly become cluttered and difficult to read when using the classic builder pattern. This approach also makes it easier to enforce constraints and ensure type safety than using builders. The greater flexibility of loan pattern DSLs sometimes allows other DSL techniques to be incorporated, whereas the builder approach is often too rigid for such improvements. This is why it is often used as a scaffolding for hybrid DSLs, which will be discussed in Chapter 11.

In this chapter, we will first write a more convenient replacement for the `HttpRequest.Builder` that was introduced in the previous chapter. If you have control over the business classes that you want to construct, you can use a library such as AutoDSL[1] to generate a DSL for you. We will explore this in the second part of the chapter.

Case Study: HttpRequest DSL

As discussed in the last chapter, `HttpRequest` already comes with a builder which looks like this.

```
val builderRequest = HttpRequest
    .newBuilder(URI.create("https://acme.com:9876/products"))
    .GET()
    .header(
        "Content-Type",
        "application/x-www-form-urlencoded; charset=UTF-8")
    .header("Accept-Encoding", "gzip, deflate")
    .timeout(Duration.ofSeconds(5L))
    .build()
```

Listing 7-1 HttpRequest Builder

This doesn't look too bad, but there is some noise in the form of the `newBuilder()` and `build()` calls, and the other method calls are actually assignments in disguise. The question is how to improve this in Kotlin. Here is a suggestion for an improved syntax.

```
val loanPatternRequest =
httpRequest("https://acme.com:9876/products") {
    method = GET
    headers {
      "Content-Type" .. "application/x-www-form-urlencoded;
charset=UTF-8"
      "Accept-Encoding" .. "gzip, deflate"
    }
    timeout = 5 * SECONDS
}
```

Listing 7-2 An Improved HttpBuilder DSL

There are no `newBuilder()` and `build()` calls anymore; the assignments are true assignments, and the headers have their own subsection. You can specify the web address as either a `String` or a URI; the example shows the `String` version. As you may have guessed, the `httpRequest` methods use the loan pattern: they initialize a builder class, expose it as a receiver, and handle the final `build()` call. The methods also take care of initializing a mandatory field – the URI – by requiring it as an explicit argument.

```
fun httpRequest(uri: URI, block: HttpRequestBuilder.() -> Unit)
=
    HttpRequestBuilder(uri).apply(block).build()

fun httpRequest(address: String, block: HttpRequestBuilder.() ->
Unit) = block: HttpRequestBuilder.() -> Unit) =
```

```
    HttpRequestBuilder(URI.create(address)).apply(block).build()
```

Listing 7-3 The HttpRequest Methods

The builder contains some mutable fields and a `build()` method to construct the `HttpRequest` instance.

```
typealias HttpMethod = Pair<String, BodyPublisher?>

class HttpRequestBuilder(var uri: URI) {

    var method: HttpMethod? = null
    var timeout: Duration? = null
    var expectContinue: Boolean? = null
    var version: HttpClient.Version? = null
    private val headers = mutableMapOf<String, String>()

    ...

    internal fun build(): HttpRequest =
        with(HttpRequest.newBuilder(uri)) {
            // set the values
            // ...
            this.build()
        }

    ...
}
```

Listing 7-4 The Outline of HttpRequestBuilder

The `headers` field cannot be set directly, because we want to use a nested structure here. The `headers()` method works in the same way as the `httpRequest()` method, and exposes the inner `Headers` class, which in turn allows the `headers` field to be populated. The `..` range operator was chosen to collect the key-value pairs because it resembles a sideways colon. Since we don't want to expose all of the `HttpRequestBuilder` fields in its inner `Headers` class, we use the `@DslMarker` mechanism discussed in Chapter 4 to limit the scope within `Headers`.

```
@DslMarker
annotation class HttpRequestDsl

@HttpRequestDsl
class HttpRequestBuilder(var uri: URI) {
    ...
    private val headers = mutableMapOf<String, String>()
```

```
    ...
    fun headers(block: Headers.() -> Unit) {
        Headers().apply(block)
    }
    ...
    @HttpRequestDsl
    inner class Headers {
        operator fun String.rangeTo(value: String) {
            this@HttpRequestBuilder.headers[this@rangeTo] =
value
        }
    }
   ...
}
```

Listing 7-5 DslMarker and Headers

Next, some fields and methods are required to assist in setting the HTTP method. However, you can still define your own, for example, `method = "OPTION" to someBodyPublisher`.

```
typealias HttpMethod = Pair<String, BodyPublisher?>

class HttpRequestBuilder(var uri: URI) {

    var method: HttpMethod? = null
    ...
    val GET: HttpMethod = "GET" to null
    val DELETE: HttpMethod = "DELETE" to null
    fun PUT(bp: BodyPublisher): HttpMethod = "PUT" to bp
    fun POST(bp: BodyPublisher): HttpMethod = "POST" to bp
    ...
}
```

Listing 7-6 Predefined HTTP Methods

Some operator overloading functions help to specify a `Duration` in a more natural way. Note that these are only visible within `HttpRequestBuilder`, so naming conflicts can be avoided.

```
class HttpRequestBuilder(var uri: URI) {
    ...
    operator fun Long.times(unit: TemporalUnit): Duration =
        Duration.of(this, unit)
```

```
    operator fun Int.times(unit: TemporalUnit): Duration =
        Duration.of(this.toLong(), unit)
}
```

Listing 7-7 Duration Helper Methods

And that's it; we covered everything except for a few details of the `build()` method. Listing 7-8 shows the complete code.

```
fun httpRequest(uri: URI, block: HttpRequestBuilder.() -> Unit)
=
    HttpRequestBuilder(uri).apply(block).build()

fun httpRequest(address: String, block: HttpRequestBuilder.() ->
Unit) =, block: HttpRequestBuilder.() -> Unit) =
    HttpRequestBuilder(URI.create(address)).apply(block).build()

typealias HttpMethod = Pair<String, BodyPublisher?>

@DslMarker
annotation class HttpRequestDsl

@HttpRequestDsl
class HttpRequestBuilder(var uri: URI) {

    var method: HttpMethod? = null
    var timeout: Duration? = null
    var expectContinue: Boolean? = null
    var version: HttpClient.Version? = null
    private val headers = mutableMapOf<String, String>()

    val GET: HttpMethod = "GET" to null
    val DELETE: HttpMethod = "DELETE" to null
    fun PUT(bp: BodyPublisher): HttpMethod = "PUT" to bp
    fun POST(bp: BodyPublisher): HttpMethod = "POST" to bp

    fun headers(block: Headers.() -> Unit) {
        Headers().apply(block)
    }

    internal fun build(): HttpRequest =
        with(HttpRequest.newBuilder(uri)) {
            headers.forEach { (key, value) -> header(key, value)
}
            timeout?.let { timeout(it) }
```

```
            expectContinue?.let { expectContinue(it) }
            version?.let { version(it) }
            method?.let {
                when (method) {
                    GET -> GET()
                    DELETE -> DELETE()
                    else -> method(method!!.first,
method!!.second)
                }
            }
            this.build()
        }

    @HttpRequestDsl
    inner class Headers {
        operator fun String.rangeTo(value: String) {
            this@HttpRequestBuilder.headers[this@rangeTo] =
value
        }
    }

    operator fun Long.times(unit: TemporalUnit): Duration =
        Duration.of(this, unit)

    operator fun Int.times(unit: TemporalUnit): Duration =
        Duration.of(this.toLong(), unit)
}
```

Listing 7-8 Complete HttpRequestBuilder DSL

Retrofitting `HttpRequestBuilder` with a loan pattern DSL proved to be a relatively simple task, but the resulting DSL is convenient and idiomatic. Customizing existing libraries in this way, particularly those written in Java, makes it easier to meet user needs and integrate them more seamlessly into the Kotlin ecosystem.

Case Study: HttpRequest with AutoDSL

This type of DSL is very common, and its structure is predictable. Therefore, it is not surprising that there are libraries for automatically deriving such DSLs. One such library is AutoDSL, which we want to discuss now. It must be set up as an annotation processor (via either kapt or KSP). If you would like to try it out, please follow the instructions on the GitHub project page.

Caution Please make sure that you are using the correct project. There is an older library called "AutoDsl" on GitHub that inspired this project. Unfortunately, it is no longer maintained and is incompatible with Kotlin 1.4 and later versions.

In the previous section, we had to put in some effort to write the `HttpRequestBuilder` DSL. Now, let's see what we can get "for free" instead! Note that we can't annotate the `HttpRequest` class itself. Therefore, we have to use an intermediate class instead, and call the `build()` method at the end. Typically, we wouldn't do this for classes under our control, but we would annotate them directly instead.

```
typealias HttpMethod = Pair<String, HttpRequest.BodyPublisher?>

val GET: HttpMethod = "GET" to null
val DELETE: HttpMethod = "DELETE" to null
fun PUT(bp: HttpRequest.BodyPublisher): HttpMethod = "PUT" to
bp
fun POST(bp: HttpRequest.BodyPublisher): HttpMethod = "POST" to
bp

@AutoDsl
data class Header(val key: String, val value: String)

@AutoDsl
data class HttpRequestBuilder(
    val uri: URI,
    val method: HttpMethod = GET,
    val timeout: Duration? = null,
    val expectContinue: Boolean? = null,
    val version: HttpClient.Version? = null,
    @AutoDslSingular("header")
    val headers: List<Header> = listOf()
) {

    fun build(): HttpRequest =
        with(HttpRequest.newBuilder(uri)) {
            headers.forEach { (key, value) -> header(key,
value) }
            timeout?.let { timeout(it) }
            expectContinue?.let { expectContinue(it) }
            version?.let { version(it) }
            method.let {
                when (method) {
```

```
                    GET -> GET()
                    DELETE -> DELETE()
                    else -> method(method.first, method.second)
                }
            }
            this.build()
        }
}
```

Listing 7-9 A Generated DSL for HttpRequest

It doesn't get much easier than that: all classes that should be included in the DSL are marked with the `@AutoDsl` annotation. If there are lists that should be specified element-wise rather than as a whole, you add an `@AutoDslSingular` annotation containing the name of the helper method.

When you compile the project using Gradle, the generated classes `HeaderDsl` and `HttpRequestBuilderDsl` should usually be located in the folder `build/generated`. This depends on how you have integrated the AutoDSL processor and set up your project, of course. The sample call from the previous section would now look like this.

```
val request = httpRequestBuilder {
    uri = URI.create("https://acme.com:9876/products")
    method = GET
    header {
        key = "Content-Type"
        value = "application/x-www-form-urlencoded;
charset=UTF-8"
    }
    header {
        key = "Accept-Encoding"
        value = "gzip, deflate"
    }
    timeout = Duration.ofSeconds(5)
}.build()
```

Listing 7-10 Calling the Generated DSL

Admittedly, the code isn't quite as convenient or concise as the manually written DSL, but it's close. It's definitely more readable and intuitive than a traditional builder. AutoDSL also keeps track of mandatory fields, such as `uri`, and throws an `IllegalStateException` if they are not set.

Builder Type Inference

In some cases, the compiler can improve its type inference by inspecting the method calls inside the trailing lambda block. As of Kotlin 1.7.0, this feature is enabled by default, but in older versions, you can turn it on using the `-Xenable-builder-inference` compiler option. There are no real drawbacks to using this feature, but if you want to look into the details, you can check out the section "Using Builders with Builder Type Inference" of the Kotlin documentation.[2]

Summary

The loan pattern DSL has several advantages over the classic builder pattern style, and is very common in Kotlin. It really shines when dealing with nested structures, and allows other DSL techniques to be integrated more easily. The Kotlin language provides several features to improve the user experience, such as the `@DslMarker` mechanism and builder type inference.

Common Applications

- Data creation and initialization
- Configuration management
- Workflow orchestration
- Code generation
- Testing
- Logging
- Data validation
- Reporting and analytics

Pros

- Easy to read, especially for nested structures
- Very flexible and intuitive
- Can be auto-generated using libraries like AutoDSL
- Allows you to limit the visibility, for instance, for overloaded operators

Cons

- Behavior is harder to control than with the builder pattern
- Enforcing safe usage can be challenging
- Difficult to use from Java client code

What's Next?

So far, we discussed building instances that didn't require different phases or stages for their construction. The next chapter will look into the topic of building objects that actually need such state transitions.

Footnotes

1 https://github.com/F43nd1r/autodsl

2 https://kotlinlang.org/docs/using-builders-with-builder-inference.html

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
D. Gronau, *Creative DSLs in Kotlin*
https://doi.org/10.1007/979-8-8688-2821-8_8

8. Modeling State Transitions in DSLs

Daniel Gronau[1] ✉
(1) Sandersdorf, Sachsen-Anhalt, Germany

> *All things change in a dynamic environment. Your effort to remain what you are is what limits you.*
> —Puppet Master, *Ghost in the Shell*[1]

DSLs often need to model complex state transitions that occur during program execution, to model workflows, or when objects need to be constructed in stages. In this chapter, we will explore three different techniques for modeling such use cases, how to choose the right one, and how to use these techniques in conjunction with other patterns such as the builder and the loan pattern. The three approaches are

- **Separate Classes:** The most intuitive implementation, where each state is represented by an independent class.
- **Chameleon Class:** The different states are represented by interfaces with their respective methods. A single class implements all interfaces, but the return type of its methods is always one of the interfaces, never the class itself.
- **Phantom Type:** A single class has a generic type parameter representing the different states, and extension methods can only be called for a particular state by specifying the appropriate type parameter for their receiver.

Case Study: DSL for SQL Queries Using the Builder Pattern

To allow a good comparison between the implementations, we will write a DSL for building SQL queries using the different approaches. Since the SQL language is very complex, we have to limit our example to queries using only `SELECT`, `FROM`, `JOIN`, `WHERE`, and `GROUP BY` clauses and simple string arguments.[2] The diagram in Figure 8-1 gives an overview of the allowed transitions.

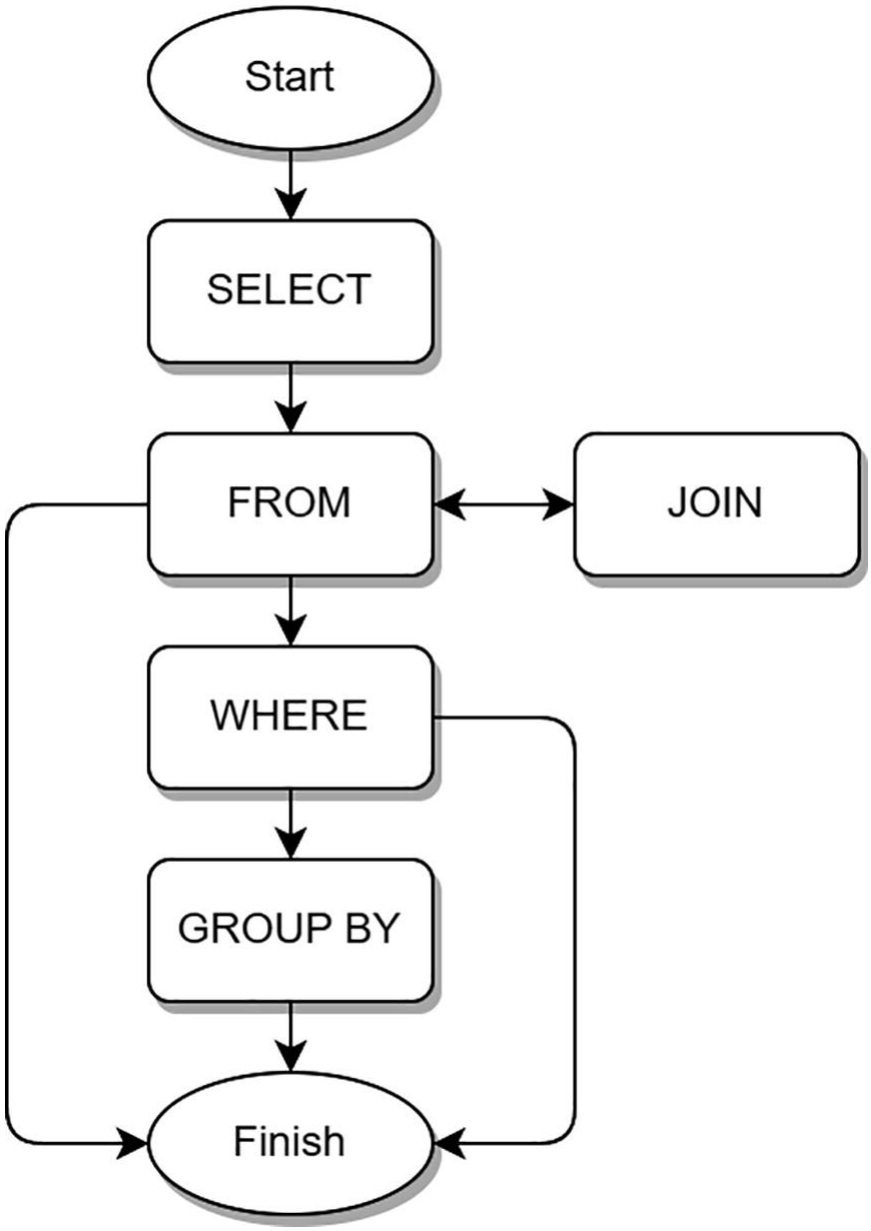


Figure 8-1 Simplified SELECT SQL Query Transitions

The final DSL will allow us to write basic SQL queries like this.

```
val queryAll = SELECT("p.firstName", "p.lastName",
"p.income")
    .FROM("Person", "p")
    .build()

val queryJoin = SELECT("p.firstName",
"p.lastName", "p.income")
    .FROM("Person", "p")
    .JOIN("Address", "a").ON("p.addressId","a.id")
    .WHERE("p.age > 20")
    .AND("p.age <= 40")
    .AND("a.city = 'London'")
    .build()

val queryGroupBy = SELECT("p.age",
"min(p.income)", "max(p.income)")
    .FROM("Person", "p")
    .WHERE("p.age > 20")
    .GROUP_BY("p.age")
    .build()
```

Listing 8-1 SQL DSL Usage Examples

While this example code uses a builder pattern syntax, we will also discuss what a loan pattern implementation of the same functionality might look like.

The "Separate Classes" Approach

The first approach is straightforward: each state is represented by a separate class that is independent of all other classes. While this is conceptually very simple, the main drawback is that all collected data must be transferred between the classes representing the states. Often,

it is more convenient and readable to pass a dedicated *Data Transfer Object* (DTO) instead of using multiple parameters.

> **Note** A Data Transfer Object (DTO) is a container object that is used to hold and transfer data across layers, without any business logic or behavior. DTOs are often used to simplify the interface between different application subsystems and to reduce the amount of data that needs to be exchanged between them. A DTO's primary purpose is to provide a simple, standardized way of representing data and to allow for easy serialization and deserialization.

In preparation for writing the SQL DSL, we need two helper classes and the DTO.

```
data class NameWithAlias(
    val name: String,
    val alias: String? = null
) {
    override fun toString(): String = when (alias)
{
        null -> name
        else -> "$name AS $alias"
    }
}

data class TableJoin(
    val nameWithAlias: NameWithAlias,
    val column1: String,
    val column2: String
)

data class QueryDTO(
    val columns: List<String>,
    val tableName: NameWithAlias,
```

```
    val joinClauses: List<TableJoin> =
emptyList(),
    val whereConditions: List<String> =
emptyList(),
    val groupByColumns: List<String> = emptyList()
)
```

Listing 8-2 DTO and Helper Classes

The `NameWithAlias` class simplifies working with names that can be aliased in SQL. The `TableJoin` class contains all the information necessary for joining two tables. Note that our simple model does not contain left or right joins. We will reuse these two helper classes throughout the whole chapter.

After this preparation, we can write the `SELECT` part.

```
fun SELECT(vararg columns: String) =
SelectClause(*columns)

class SelectClause(vararg val columns: String) {
    fun FROM(tableName: String, alias: String? =
null) =
        FromClause(
            QueryDTO(
                columns = columns.asList(),
                tableName =
NameWithAlias(tableName, alias)
            )
        )
}
```

Listing 8-3 The SELECT Clause

There is no `build()` method; the only way forward is going into the `FromClause`, which is a bit more involved, as there might be multiple tables joined together.

```
data class FromClause(val queryDTO: QueryDTO) {

    fun JOIN(tableName: String, alias: String? =
null) =
        JoinClause(queryDTO,
NameWithAlias(tableName, alias))

    fun WHERE(condition: String) =
        WhereClause(queryDTO.copy(
            whereConditions = listOf(condition)
        ))

    fun GROUP_BY(vararg groupByColumns: String) =
        GroupByClause(queryDTO.copy(
            groupByColumns =
groupByColumns.toList()
        ))

    fun build() = build(queryDTO)
}
```

Listing 8-4 The FROM Clause

From here, you can go to a `JoinClause`, which mimics SQL syntax by allowing you to write something like `JOIN("Address","a").ON("p.addressId", "a.id")`. The other exit points lead to a `WhereClause` or a `GroupByClause`. Additionally, the `FromClause` has a `build()` method, because the `WHERE` and `GROUP BY` parts are optional. The `JoinClause` only provides an `ON()` method, which leads back to the `FromClause`.

```
data class JoinClause(
    val queryDTO: QueryDTO,
    val tableName: NameWithAlias
) {
```

```
    fun ON(firstColumn: String, secondColumn:
String) =
        FromClause(queryDTO.copy(
            joinClauses = queryDTO.joinClauses +
                TableJoin(tableName, firstColumn,
secondColumn)
        ))
}
```

Listing 8-5 The JOIN Clause

The `WhereClause` is conceptually quite simple. However, using a string to represent the different conditions is insecure and should be avoided in production code. Our SQL subset only allows us to proceed to the `GroupByClause`, whereas the full SQL syntax would also allow other clauses such as `HAVING` or `ORDER BY`. Alternatively, we can end the query by calling the `build()` method.

```
data class WhereClause(val queryDTO: QueryDTO) {
    fun AND(condition: String) =
        copy(queryDTO = queryDTO.copy(
            whereConditions =
queryDTO.whereConditions +
                condition
        ))

    fun GROUP_BY(vararg groupByColumns: String) =
        GroupByClause(queryDTO.copy(
            groupByColumns =
groupByColumns.toList()
        ))

    fun build() = build(queryDTO)
}
```

Listing 8-6 The WHERE Clause

The `GroupByClause` only allows you to call the `build()` method.

```
data class GroupByClause(val queryDTO: QueryDTO) {
    fun build() = build(queryDTO)
}
```

Listing 8-7 The GROUP BY Clause

The only thing missing is the common `build(queryDTO)` method used by `FromClause`, `WhereClause`, and `GroupByClause`.

```
private fun build(queryDTO: QueryDTO): String =
with(StringBuilder()) {

    val (columns, tableName, joinClauses,
whereConditions, groupByColumns)
        = queryDTO

    append("SELECT ${columns.joinToString(", ")}")
    append("\nFROM $tableName")

    joinClauses.forEach { (n, c1, c2) ->
        append("\n  JOIN $n ON $c1 = $c2")
    }

    if (whereConditions.isNotEmpty())
        append("\nWHERE
${whereConditions.joinToString("\n  AND ")}")

    if (groupByColumns.isNotEmpty())
        append("\nGROUP BY
${groupByColumns.joinToString(", ")}")

    append(';')

}.toString()
```

Listing 8-8 The build() Method

Bundling all data into a DTO instance, as shown here, can significantly reduce the overhead of moving all data around, especially by taking advantage of the power of the `copy()` method. In the next section, we will explore an alternative implementation of the same DSL.

The "Chameleon Class" Approach

While using a separate DTO class makes the separate class approach more convenient, it would be nicer if we didn't have to copy data in the first place. But what about all the guarantees the first solution provides, for instance, that you can't call `build()` or `JOIN()` in a `SELECT` clause? One way to do this is to use a technique I call the *chameleon class*. The basic idea is to adapt the type of this class to the current state it represents, and change it accordingly when the state changes. A chameleon class

- Implements multiple interfaces
- Never exposes its own type, but always acts as one of those interfaces
- Has a private constructor to avoid exposing its own type
- Holds common data

First, we must translate our former state classes into interfaces.

```
interface SelectClause {
    fun FROM(table: String, alias: String? =
null): FromClause
}

interface FromClause {
    fun JOIN(tableName: String, alias: String? =
null): JoinClause
    fun WHERE(condition: String): WhereClause
    fun GROUP_BY(vararg groupByColumns: String):
GroupByClause
    fun build(): String
}
```

```
interface JoinClause {
    fun ON(firstColumn: String, secondColumn:
String): FromClause
}

interface WhereClause {
    fun AND(condition: String): WhereClause
    fun GROUP_BY(vararg groupByColumns: String):
GroupByClause
    fun build(): String
}

interface GroupByClause {
    fun build(): String
}
```

Listing 8-9 Interfaces of the Chameleon Class

All that remains is to implement these interfaces in a single chameleon class and keep track of the data. It's important to make the constructor private, because the initial type should not be the type of the class itself, but `SelectClause`. That's why the `SELECT()` method in the companion object is used as a starting point for the DSL.

```
class QueryBuilder private constructor(val
columns: List<String>):
    SelectClause, FromClause, JoinClause,
WhereClause, GroupByClause {
    var tableName = NameWithAlias("", null)
    var joinTableName = NameWithAlias("", null)
    val joinClauses = mutableListOf<TableJoin>()
    val whereConditions = mutableListOf<String>()
    val groupByColumns = mutableListOf<String>()

    companion object {
```

```
        fun SELECT(vararg columns: String):
SelectClause =
            QueryBuilder(columns.asList())
    }

    // SelectClause
    override fun FROM(table: String, alias:
String?): FromClause =
        this.apply { tableName =
NameWithAlias(table, alias) }

    // FromClause
    override fun JOIN(tableName: String, alias:
String?): JoinClause =
        this.apply { joinTableName =
NameWithAlias(tableName, alias) }

    override fun WHERE(condition: String):
WhereClause =
        this.apply { whereConditions += condition
}

    // JoinClause
    override fun ON(
        firstColumn: String,
        secondColumn: String
    ): FromClause = this.apply { joinClauses +=
            TableJoin(joinTableName, firstColumn,
secondColumn)
    }

    // WhereClause
    override fun AND(condition: String):
WhereClause =
```

```
        this.apply { whereConditions += condition
}

    // FromClause and WhereClause
    override fun GROUP_BY(vararg groupByColumns:
String): GroupByClause =
        this.apply { this.groupByColumns +=
groupByColumns.toList() }

    // FromClause, WhereClause and GroupByClause
    override fun build(): String =
with(StringBuilder()) {

        append("SELECT ${columns.joinToString(",
") { it }}")
        append("\nFROM $tableName")

        joinClauses.forEach { (n, c1, c2) ->
            append("\n  JOIN $n ON $c1 = $c2")
        }

        if (whereConditions.isNotEmpty())
            append("\nWHERE
${whereConditions.joinToString("\n  AND ")}")

        if (groupByColumns.isNotEmpty())
            append("\nGROUP BY
${groupByColumns.joinToString(", ")}")

        append(';')

    }.toString()
}
```

Listing 8-10 The Chameleon Class

It doesn't matter to the compiler that you return the same object over and over again at runtime, because only the static type decides which methods can be called, and that static type is never `QueryBuilder` itself, but one of the interfaces for the SQL clauses. Using the DSL looks the same as before, and you still can't call methods out of order.

The chameleon class concept may look a bit strange at first, but usually results in compact and readable code. Be aware, however, that this approach is prone to name conflicts when two interfaces contain methods with the same name and parameters, but different return types.

The "Phantom Type" Approach

The third approach uses phantom types. The implementation is based on a DTO class with a generic parameter. This type parameter is not used as a type for any data within the class – that's why it's called a "phantom type." Instead, this parameter is used by extension functions that require their receiver to have the correct state.

For the SQL query DSL, we need a type hierarchy containing the different clauses. As a slight complication, we also need two additional interfaces for methods that are present in multiple clauses. Then, we need the DTO class itself. The `cast()` extension function allows us to easily switch between states. Since the generic parameter doesn't refer to any real data, the cast itself is safe. Of course, the `cast()` function must be private in order to prevent misuse.

```
interface CanGroupBy
interface CanBuild

sealed interface State
interface SelectClause : State
interface FromClause : State, CanGroupBy, CanBuild
interface JoinClause : State
```

```
interface WhereClause : State, CanGroupBy,
CanBuild
interface GroupByClause : State, CanBuild

data class QueryDTO<out State>(
    val columns: List<String>,
    val tableName: NameWithAlias =
NameWithAlias(""),
    val joinTableName: NameWithAlias =
NameWithAlias(""),
    val joinClauses: List<TableJoin> =
emptyList(),
    val whereConditions: List<String> =
emptyList(),
    val groupByColumns: List<String> = emptyList()
)

@Suppress("UNCHECKED_CAST")
private fun <S : State> QueryDTO<*>.cast():
QueryDTO<S> =
    this as QueryDTO<S>
```

Listing 8-11 DTO with Phantom Type

Implementing the `SELECT` function, which provides a starting point, and the subsequent extension functions for the state transitions is straightforward.

```
fun SELECT(vararg columns: String) =
   QueryDTO<SelectClause>(columns =
columns.toList())

fun QueryDTO<SelectClause>.FROM(
    table: String,
    alias: String?
): QueryDTO<FromClause> =
```

```
    copy(tableName = NameWithAlias(table,
alias)).cast()

fun QueryDTO<FromClause>.JOIN(
    tableName: String,
    alias: String?
): QueryDTO<JoinClause> =
    copy(joinTableName = NameWithAlias(tableName,
alias)).cast()

fun QueryDTO<FromClause>.WHERE(
    condition: String
): QueryDTO<WhereClause> =
    copy(whereConditions = whereConditions +
condition).cast()

fun QueryDTO<JoinClause>.ON(
    firstColumn: String,
    secondColumn: String
): QueryDTO<FromClause> =
    copy(joinClauses = joinClauses +
        TableJoin(joinTableName, firstColumn,
secondColumn)
    ).cast()

fun QueryDTO<WhereClause>.AND(
    condition: String
): QueryDTO<WhereClause> =
    copy(whereConditions = whereConditions +
condition)

fun QueryDTO<CanGroupBy>.GROUP_BY(
    vararg groupByColumns: String
): QueryDTO<GroupByClause> =
```

```
    copy(groupByColumns =
groupByColumns.toList()).cast()

fun QueryDTO<CanBuild>.build(): String =
with(StringBuilder()) {

    append("SELECT ${columns.joinToString(", ")}")
    append("\nFROM $tableName")

    joinClauses.forEach { (n, c1, c2) ->
        append("\n  JOIN $n ON $c1 = $c2")
    }

    if (whereConditions.isNotEmpty())
        append("\nWHERE
${whereConditions.joinToString("\n  AND ")}")

    if (groupByColumns.isNotEmpty())
        append("\nGROUP BY
${groupByColumns.joinToString(", ")}")

    append(';')

}.toString()
```

Listing 8-12 The SELECT Function and Extension Functions of the DTO

Note that `GROUP_BY()` can be called, for example, on `QueryDTO<FromClause>` even though it is defined as `fun QueryDTO<CanGroupBy>.groupBy(...)`. This is allowed because the phantom type in `QueryDTO` was defined as contravariant using the `out` keyword. Without this, we would have to use a signature like `fun <S: CanGroupBy> QueryBuilder<S>.groupBy(...)` in order to be callable from a DTO with a sub-interface, which looks rather cryptic.

Note Declaration-site variance is a way of specifying the variance of a generic type when the type is defined or declared. In the declaration of a generic type or interface, you use variance annotations to specify whether the type parameter is covariant (using `out`), contravariant (using `in`), or invariant (using neither). In contrast, use-site variance allows you to specify the variance of a generic type when you use it in a particular context (at the use site) using type bounds or wildcards. While Kotlin supports both styles, Java only allows use-site variance.

Chameleon classes and the phantom type implementation are conceptually similar, and it depends on the problem at hand whether a class with all methods or a DTO with extension methods is preferable. If the DSL needs to be called from Java, keep in mind that only the chameleon approach preserves the DSL syntax. On the other hand, the phantom type approach doesn't have fixed APIs for the different states, only extension functions that operate on them, which means that new functionality can be added more easily than with the other techniques.

Case Study: DSL for SQL Queries Using the Loan Pattern

So far, all examples have used a builder pattern syntax. This doesn't have to be the case. A DSL that uses the loan pattern style might look like this.

```
val queryAll = SELECT{
    +"p.firstName"
    +"p.lastName"
    +"p.income"
}.FROM{
    "Person" AS "p"
}.build()
```

```
val queryJoin = SELECT{
    +"p.firstName"
    +"p.lastName"
    +"p.income"
}.FROM{
    "Person" AS "p"
    JOIN{
        "Address" AS "a"
        ON("p.addressId","a.id")
    }
}.WHERE {
    +"p.age > 20"
    +"p.age <= 40"
    +"a.city = 'London'"
}.build()

val queryGroupBy = SELECT{
    +"p.age"
    +"min(p.income)"
    +"max(p.income)"
}.FROM{
    "Person" AS "p"
}.WHERE {
    +"p.age > 20"
}.GROUP_BY{
    +"p.age"
}.build()
```

Listing 8-13 SQL DSL Usage Examples

This looks quite different from the builder pattern syntax, and it's debatable whether this style looks better for this particular use case. It may be better suited for cases that require deeper nesting or more complex operations in the trailing lambda bodies.

One difference from the builder example is that `JOIN` is now nested, which seems more natural here. The lambda bodies give more freedom to use other DSL techniques, such as infix functions like `AS`. Note that we still need to use `build()` methods because it isn't clear in SQL when constructing the query is finished. In cases with only one exit state, the construction can be done behind the scenes, as is typical in loan pattern implementations.

Note that the `@DslMarker` mechanism should be used here since the `JOIN` clause is nested. For the sake of brevity, however, it won't be applied in the following implementations.

The "Separate Classes" Approach

Here is what an implementation using separate classes might look like. We start as usual with the DTO, using the same helper classes `NameWithAlias` and `TableJoin` as before.

```
data class QueryDTO(
    val columns: List<String>,
    val tableName: NameWithAlias =
NameWithAlias(""),
    val joinClauses: List<TableJoin> =
emptyList(),
    val whereConditions: List<String> =
emptyList(),
    val groupByColumns: List<String> = emptyList()
)
```

Listing 8-14 The DTO

Now we need a starting point, in the form of a `SELECT` function. It executes the given body (where the columns can be added) and passes the results to the `SelectClause` class, which in turn has a method to proceed to the `FromClause`.

```
fun SELECT(body: SelectBody.() -> Unit) =
```

```
    SelectClause(QueryDTO(columns =
SelectBody().apply(body).columns))

class SelectBody {
    val columns = mutableListOf<String>()
    operator fun String.unaryPlus() { columns +=
this }
}

class SelectClause(val queryDTO: QueryDTO) {
    fun FROM(body: FromBody.() -> Unit) =
        FromBody().apply(body).let{
            FromClause(queryDTO.copy(
                tableName = it.tableName,
                joinClauses = it.joinClauses)
            )
        }
}
```

Listing 8-15 The SELECT Clause

The `FromBody` is a little more complex, as it contains the nested `JOIN` clause.

```
class FromBody {

    var tableName = NameWithAlias("")
    val joinClauses  = mutableListOf<TableJoin>()

    operator fun String.unaryPlus() {
        tableName = NameWithAlias(this)
    }

    infix fun String.AS(alias: String) {
        tableName = NameWithAlias(this, alias)
    }
```

```
    fun JOIN(body: JoinBody.() -> Unit) {
        JoinBody().apply(body).also { joinClauses
+=
            TableJoin(it.tableName,
it.firstColumn, it.secondColumn)
        }
    }
}

data class FromClause(val queryDTO: QueryDTO) {

    fun WHERE(body: WhereBody.() -> Unit) =
        WhereClause(queryDTO.copy(whereConditions
=
            WhereBody().apply(body).conditions))

    fun GROUP_BY(body: GroupByBody.() -> Unit) =
        GroupByClause(queryDTO.copy(groupByColumns
=
            GroupByBody().apply(body).columns))

    fun build() = build(queryDTO)
}

class JoinBody {
    var tableName = NameWithAlias("")
    var firstColumn = ""
    var secondColumn = ""

    operator fun String.unaryPlus() {
        tableName = NameWithAlias(this)
    }

    infix fun String.AS(alias: String) {
        tableName = NameWithAlias(this, alias)
```

```
    }

    fun ON(firstColumn: String, secondColumn:
String) {
        this.firstColumn = firstColumn
        this.secondColumn = secondColumn
    }
}

data class JoinClause(
    val queryDTO: QueryDTO,
    val tableName: NameWithAlias
) {
    fun ON(firstColumn: String, secondColumn:
String) =
        FromClause(queryDTO.copy(
            joinClauses = queryDTO.joinClauses +
                TableJoin(tableName, firstColumn,
secondColumn)
        ))
}
```

Listing 8-16 The FROM Clause

This scheme continues in the same style for the other clauses.

```
class WhereBody {
    val conditions = mutableListOf<String>()
    operator fun String.unaryPlus() { conditions
+= this }
}

data class WhereClause(val queryDTO: QueryDTO) {

    fun AND(condition: String) =
```

```
        copy(queryDTO =
queryDTO.copy(whereConditions =
            queryDTO.whereConditions + condition))

    fun GROUP_BY(body: GroupByBody.() -> Unit) =
        GroupByClause(queryDTO.copy(groupByColumns
=
            GroupByBody().apply(body).columns))

    fun build() = build(queryDTO)
}

class GroupByBody {
    val columns = mutableListOf<String>()
    operator fun String.unaryPlus() { columns +=
this }
}

data class GroupByClause(val queryDTO: QueryDTO) {

    fun build() = build(queryDTO)
}
```

Listing 8-17 The WHERE and GROUP_BY Clauses

The `build(queryDTO)` function is skipped here, as it is identical to the builder-style version of the code.

Admittedly, the code is more difficult to read and write, but it allows for more flexible syntax within the trailing lambda blocks, which feels more natural and structured compared to the builder pattern syntax for a wide range of problems. Using the same techniques as before, we can improve the code.

The "Chameleon Class" Approach

To use a chameleon class, we must first turn the clause data classes into interfaces.

```
interface SelectClause {
    fun FROM(body: FromBody.() -> Unit):
FromClause
}

interface FromClause {
    fun WHERE(body: WhereBody.() -> Unit):
WhereClause
    fun GROUP_BY(body: GroupByBody.() -> Unit):
GroupByClause
    fun build(): String
}

interface WhereClause {
    fun GROUP_BY(body: GroupByBody.() -> Unit):
GroupByClause
    fun build(): String
}

interface GroupByClause {
    fun build(): String
}
```

Listing 8-18 Interfaces of the Chameleon Class

All the `...Body` classes remain unchanged, and are omitted. The only thing missing is the chameleon class itself.

```
data class QueryBuilder private constructor(val
columns: List<String>) :
    SelectClause, FromClause, WhereClause,
GroupByClause {

    var tableName = NameWithAlias("")
    val joinClauses = mutableListOf<TableJoin>()
    val whereConditions = mutableListOf<String>()
```

```
    val groupByColumns = mutableListOf<String>()

    companion object {
        fun SELECT(body: SelectBody.() -> Unit):
SelectClause =
            QueryBuilder(columns =
SelectBody().apply(body).columns)
    }

    override fun FROM(body: FromBody.() -> Unit):
FromClause =
        this.apply {
            val fromBody = FromBody().apply(body)
            tableName = fromBody.tableName
            joinClauses += fromBody.joinClauses
        }

    override fun WHERE(body: WhereBody.() ->
Unit): WhereClause =
        this.apply {
            whereConditions +=
WhereBody().apply(body).conditions
        }

    override fun GROUP_BY(body: GroupByBody.() ->
Unit): GroupByClause =
        this.apply {
            groupByColumns +=
GroupByBody().apply(body).columns
        }

    override fun build(): String =
with(StringBuilder()) {
```

```
        append("SELECT ${columns.joinToString(",
")}")
        append("\nFROM $tableName")

        joinClauses.forEach { (n, c1, c2) ->
            append("\n  JOIN $n ON $c1 = $c2")
        }

        if (whereConditions.isNotEmpty())
            append("\nWHERE
${whereConditions.joinToString("\n  AND ")}")

        if (groupByColumns.isNotEmpty())
            append("\nGROUP BY
${groupByColumns.joinToString(", ")}")

        append(';')

    }.toString()
}
```

Listing 8-19 The Chameleon Class

The "Phantom Type" Approach

Implementing the DSL using phantom types is very similar to the corresponding builder pattern code. Again, the `...Body` classes are unchanged.

```
interface CanGroupBy
interface CanBuild

sealed interface State
interface SelectClause : State
interface FromClause : State, CanGroupBy, CanBuild
interface WhereClause : State, CanGroupBy,
CanBuild
```

```
interface GroupByClause : State, CanBuild

data class QueryDTO<out State>(
    val columns: List<String>,
    val tableName: NameWithAlias =
NameWithAlias(""),
    val joinTableName: NameWithAlias =
NameWithAlias(""),
    val joinClauses: List<TableJoin> =
emptyList(),
    val whereConditions: List<String> =
emptyList(),
    val groupByColumns: List<String> = emptyList()
)

@Suppress("UNCHECKED_CAST")
private fun <S: State> QueryDTO<*>.cast():
QueryDTO<S> =
    this as QueryDTO<S>

fun SELECT(body: SelectBody.() -> Unit):
QueryDTO<SelectClause> =
    QueryDTO(columns =
SelectBody().apply(body).columns)

fun QueryDTO<SelectClause>.FROM(
    body: FromBody.() -> Unit
): QueryDTO<FromClause> =
    FromBody().apply(body).let {
        this@FROM.copy(
            tableName = it.tableName,
            joinClauses = it.joinClauses
        )
    }.cast()
```

```
fun QueryDTO<FromClause>.WHERE(
    body: WhereBody.() -> Unit
): QueryDTO<WhereClause> =
    copy(whereConditions =
WhereBody().apply(body).conditions).cast()

fun QueryDTO<CanGroupBy>.GROUP_BY(
    body: GroupByBody.() -> Unit
): QueryDTO<GroupByClause> =
    copy(groupByColumns =
GroupByBody().apply(body).columns).cast()

private fun QueryDTO<CanBuild>.build(): String =
with(StringBuilder()) {

    append("SELECT ${columns.joinToString(", ")}")
    append("\nFROM $tableName")

    joinClauses.forEach { (n, c1, c2) ->
        append("\n  JOIN $n ON $c1 = $c2")
    }

    if (whereConditions.isNotEmpty())
        append("\nWHERE
${whereConditions.joinToString("\n  AND ")}")

    if (groupByColumns.isNotEmpty())
        append("\nGROUP BY
${groupByColumns.joinToString(", ")}")

    append(';')

}.toString()
```

Listing 8-20 Phantom Type Implementation

Summary

In this chapter, we discussed how state transitions can be expressed using different techniques. The DSLs can use either an underlying builder or loan pattern, and there are different approaches to implementing them. When in doubt, I would recommend starting with the separate classes approach, especially while prototyping. If you use a DTO (as recommended), the code can easily be converted to the chameleon or phantom type style later.

Common Applications

- Data creation and initialization
- Data transformation
- Testing
- Workflow orchestration
- Simulation and modeling

Pros

- Enforces correct state transitions
- Intuitive way to write code that generates data incrementally
- Ideal for modeling finite state machines

Cons

- DSL implementation can be difficult to read
- Can lead to boilerplate code
- The “separate classes” approach requires data to be copied over
- The “phantom type” approach is difficult to use from Java client code

---

What’s Next?

The DSL categories we’ve discussed thus far offer a high degree of flexibility. However, some applications require more freedom than Kotlin’s syntax allows. Fortunately, there is a simple solution: enclose the entire DSL in double quotation marks. Extracting the necessary

information from this string can be more challenging, but the next chapter will teach you how to do so.

Footnotes

1 Ghost in the Shell, directed by Mamoru Oshii, Production I.G., 1995.

2 A good example for a real-world SQL DSL library is jOOQ, https://www.jooq.org

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
D. Gronau, *Creative DSLs in Kotlin*
https://doi.org/10.1007/979-8-8688-2821-8_9

9. String-Parsing DSLs

Daniel Gronau[1] ✉
(1) Sandersdorf, Sachsen-Anhalt, Germany

> *Words are, of course, the most powerful drug used by mankind.*
> —Rudyard Kipling[1]

Sometimes, the host language doesn't allow you to create the desired DSL syntax. A string parsing DSL provides the flexibility to write almost anything. Other DSL categories tend to be particularly bad at giving users leeway. For example, they often don't allow for differences in uppercase and lowercase syntax, the order of operations, or the use of special characters, such as symbols. Additionally, Kotlin's support for multiline strings makes it convenient to write larger string-based DSLs and allows for "graphical" DSLs, such as two-dimensional representations of game levels.

One drawback is that these DSLs are hidden from the compiler's scrutiny: there are no compile-time checks or auto-completion hints, so users have to rely on documentation. Another issue is writing the parser, which is usually more involved than collecting data with other DSL types. However, I hope this chapter convinces you that it is not as daunting as it seems.

Caution Kotlin, like many languages, supports *Regular Expressions*, which can be thought of as a kind of mini-parser. For small, isolated use cases, they can be very convenient. However, using them for larger problems can result in code that is very hard to maintain because this approach doesn't scale well. In my opinion, using regular expressions as full-blown "parsers" may be acceptable for prototypes and test environments, but not in production code.

Case Study: Forsyth–Edwards Notation

Before we dive into problems that require parsers, we will start with a simple example that relies only on a few string functions. Unless you play chess, you've

probably never heard of *Forsyth–Edwards Notation*[2] (FEN), but it's quite simple.

FEN is a standard way to describe the current position of the pieces on a chessboard. It differs from the more widely known Portable Game Notation (PGN), which documents entire games. FEN consists of a string of characters representing the placement of pieces, organized by rank. In FEN, uppercase letters represent white pieces and lowercase letters represent black pieces. Numbers from 1 to 8 encode successive empty squares, and a slash (/) separates ranks. After the board position, additional information is provided:

- Which player moves next?
- Castling availability
- En passant capture availability
- The number of half-moves without check or pawn moves
- The number of moves played so far

A representation of a typical chess position looks like this when written FEN: `rnbqkbnr/pp1ppppp/8/2p5/4P3/5N2/PPPP1PPP/RNBQKB1R b KQkq - 1 2`.

Pieces have the usual names used in chess notation (such as `Q` for a queen), plus `P` for a pawn. To keep the string short, empty squares are written with a kind of run-length encoding; for example, three consecutive empty squares are represented by a `3`. If you put the rows one under the other and expand the numbers to empty spaces, you can "see" the board position.

```
rnbqkbnr
pp.ppppp
........
..p.....
....P...
.....N..
PPPP.PPP
RNBQKB.R
```

The rest of the string is also not difficult: a `w` or `b` indicates which side is to move. Castling rights are indicated by `k` for the king's side and `q` for the queen's side. Again, uppercase is used for white and lowercase for black. If no side can castle, a dash (–) is written instead. The en passant entry contains either a square, such as `b3`, or a dash (–). Next come the number of half-moves without check and pawn moves,[3] followed by the current move count.

This is the `Position` class we want to create using a FEN string.

```
enum class Piece(val symbol: String) {
    WhitePawn("P"),   WhiteRook("R"),  WhiteKnight("N"),
    WhiteBishop("B"), WhiteQueen("Q"), WhiteKing("K"),
    BlackPawn("p"),   BlackRook("r"),  BlackKnight("n"),
    BlackBishop("b"), BlackQueen("q"), BlackKing("k"),
}

enum class Color(val symbol: String) {
    Black("b"),
    White("w")
}

data class Position(
    val pieces: Map<String, Piece>,
    val toMove: Color,
    val castling: List<Piece>,
    val enPassant: String,
    val halfMoveClock: Int,
    val move: Int
) {

    private fun boardFen() =
        (8 downTo 1).joinToString("/") { row ->
            ('a'..'h').joinToString("") { col ->
                pieces["$col$row"]?.symbol ?: "1"
            }
        }.fold("") { acc, ch ->
            if (acc.isNotEmpty() && acc.last().isDigit() &&
ch == '1')
                acc.dropLast(1) + (acc.last() + 1)
            else acc + ch
        }

    private fun castlingFen() = when {
        castling.isEmpty() -> "-"
        else -> castling.joinToString("") { it.symbol }
    }

    fun asFEN() = "${boardFen()} ${toMove.symbol} " +
            "${castlingFen()} $enPassant $halfMoveClock
$move"
```

```
}
```

Listing 9-1 Representation for a Board Position

The class already contains a function to generate a FEN string – this is not required, but it makes testing much easier.

> **Note** If you have code that transforms back and forth between different formats, it is convenient to write *round-trip tests*: you provide test data for the "easier" format, transform it to the other and back, and compare it to the original. Comparing data in the same format is easier and safer than comparing data in different formats – often a simple string comparison is sufficient.

Now we can write the DSL function for parsing a FEN string.

```
import myPackage.Piece.*

fun String.toFEN(): Position = split(" ")
    .let { part ->
        Position(
            pieces = getPieces(part[0]),
            toMove = getToMove(part[1]),
            castling = getCastling(part[2]),
            enPassant = part[3],
            halfMoveClock = part[4].toInt(),
            move = part[5].toInt()
        )
    }

private fun getPieces(piecesStr: String) = piecesStr
    .fold("") { acc, ch ->
        acc + if (ch.isDigit())
".".repeat(ch.toString().toInt()) else ch
    }
    .split("/")
    .reversed()
    .flatMapIndexed { rowIndex, row ->
        row.mapIndexedNotNull { colIndex, ch ->
            Piece.entries.find { it.symbol == ch.toString()
}
                ?.let { "${'a' + colIndex}${rowIndex + 1}"
to it }
```

```
        }
    }
    .toMap()

private fun getToMove(toMoveStr: String) = when (toMoveStr)
{
    "w" -> Color.White
    "b" -> Color.Black
    else -> error("Unknown color symbol '$toMoveStr'")
}

private fun getCastling(castlingStr: String) = castlingStr
    .mapNotNull { ch ->
        when (ch) {
            'K' -> WhiteKing
            'k' -> BlackKing
            'Q' -> WhiteQueen
            'q' -> BlackQueen
            else -> null
        }
    }
```

Listing 9-2 A DSL for Parsing a FEN String

The `toFEN()` function uses helper functions for the more difficult parts and assembles the `Position` class. Most of the sanity checks have been omitted to improve readability. For simple cases like this one, writing a parser or using a parser library is probably excessive. The most challenging aspect was correctly interpreting the piece positions, which only required a few lines of code. However, this approach has its limits. As soon as readability declines or correctness becomes difficult to guarantee, a more systematic approach should be chosen.

Case Study: Chemical Equations As Strings

Writing a DSL for chemical equations is challenging because the concise notation isn't easily portable into the more involved syntax of a host language, even one as flexible as Kotlin. That's why the string parsing approach seems like a good fit. If you are curious about what a hybrid DSL for chemical equations might look like, you can skip ahead to Chapter 11.

For our case study, we won't cover the full notation; for instance, we won't support writing ions or bonds. An example of a simple chemical equation in standard notation would be `3Ba(OH)2 + 2H3PO4 → 6H2O + Ba3(PO4)2`. Of course, in the

context of a DSL, subscripts and special symbols are not very practical, so the target syntax would look more like `3Ba(OH)2 + 2H3PO4 -> 6H2O + Ba3(PO4)2`. To express such an equation, we use the following code.

```
sealed interface Part

data class Element(
    val symbol: String,
    val subscript: Int = 1
) : Part {
    override fun toString() = symbol +
subscript.oneAsEmpty()
}

data class Group(
    val parts: List<Part>,
    val subscript: Int = 1
) : Part {
    override fun toString() =
        parts.joinToString("", "(", ")") +
                subscript.oneAsEmpty()
}

data class Molecule(
    val coefficient: Int,
    val parts: List<Part>
) {
    constructor(coefficient: Int, vararg parts: Part) :
        this (coefficient, parts.toList())
    constructor(vararg parts: Part) :
        this (1, parts.toList())
    override fun toString() = coefficient.oneAsEmpty() +
            parts.joinToString("")
}

enum class Arrow(val symbol: String) {
    IRREVERSIBLE("->"),
    REVERSIBLE("<->")
}

data class Equation(
    val leftSide: List<Molecule>,
```

```
    val arrow: Arrow,
    val rightSide: List<Molecule>
) {
    override fun toString() = listOf(
        leftSide.joinToString(" + "),
        arrow.symbol,
        rightSide.joinToString(" + ")
    ).joinToString(" ")
}

private fun Int.oneAsEmpty(): String =
    takeIf { this > 1 }?.toString().orEmpty()
```

Listing 9-3 Expressing Chemical Equations

An `Element` contains a chemical symbol, such as `"H"` for hydrogen or `"Ba"` for barium, and an optional subscript indicating the number of atoms. A feature of the chemical notation is that you can also define groups such as $(OH)_2$ within a molecule. This is why we need the `Group` class. Note that a group can contain not only elements but also other groups.

A `Molecule` is a collection of elements or groups, which we subsume under the `Part` interface. It can also have a coefficient in front of it. An equation consists of two sides and either an arrow (`->`) or, in the case of reversible reactions, a double arrow (`<->`) in the middle. Both sides consist of either a single molecule or a "sum" of molecules.

The `toString()` methods provide output in chemical notation, which is useful for testing purposes. Note that secondary constructors are necessary when using varargs because they are not permitted in primary constructors of data classes.

This was a lot of information to digest. For a better overview, see Figure 9-1, which summarizes the structure of our model classes.

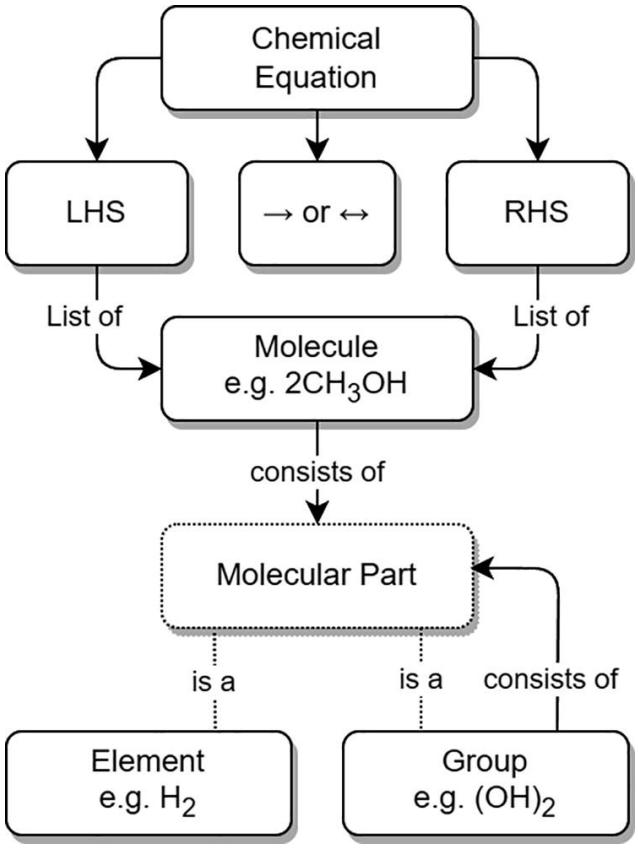


Figure 9-1 Model for Chemical Equations

Using our model classes, our example equation for making barium phosphate could be written as follows.

```
val Ba  = Element("Ba")
val Ba3 = Element("Ba", 3)
val O   = Element("O")
val O4  = Element("O", 4)
val H   = Element("H")
val H2  = Element("H", 2)
val H3  = Element("H", 3)
val P   = Element("P")

val bariumHydroxide = Molecule(3, Ba, Group(listOf(O, H),
2))
val phosphoricAcid = Molecule(2, H3, P, O4)
val water = Molecule(6, H2, O)
val bariumPhosphate = Molecule(Ba3, Group(listOf(P, O4), 2))

val equation = Equation(
```

```
        listOf(bariumHydroxide, phosphoricAcid),
        Arrow.IRREVERSIBLE,
        listOf(water, bariumPhosphate)
    )
// 3Ba(OH)2 + 2H3PO4 -> 6H2O + Ba3(PO4)2
```

Listing 9-4 Using the Model Classes

Writing a Parser for Chemical Equations

If you have never worked with parsers before, it can be a bit confusing. Writing them yourself is not that difficult, but boring and tedious, so using a library is usually the better choice. Nevertheless, I think it is helpful to understand how a parser works. First, I will present a naive manual implementation, and then I will show how to use a parser library.

First, we need some general code for a rudimentary parser. We start with a common interface, `ParseResult`, because we also need to account for parsing errors. Real-world implementations would include useful information in this `Failure` class, but for our purposes, a simple object will suffice. Next, we need a `Success` class to hold the current successful parsing result and the current location we are working on. For the location, we simply use the remaining string – more performance-oriented implementations typically just use the index of the input string. Overall, these classes have a lot in common with Java's `Optional` class.

```
sealed interface ParseResult<out T> {
    fun <U> map(body: (T) -> U): ParseResult<U> =
        when (this) {
            is Success -> Success(body(value), remaining)
            is Failure -> Failure
        }

    fun <U> flatMap(body: (T, String) -> ParseResult<U>):
ParseResult<U> =
        when (this) {
            is Success -> body(value, remaining)
            is Failure -> Failure
        }

    fun filter(cond: (T) -> Boolean): ParseResult<T> =
        when {
            this is Success && cond(value) -> this
            else -> Failure
        }
```

```
}

data class Success<T>(
    val value: T,
    val remaining: String
) : ParseResult<T>

data object Failure: ParseResult<Nothing>
```

Listing 9-5 A Rudimentary Parser Implementation

Then we need some helper functions for reading and combining parse results, and for generating lists of individual results.

```
infix fun <T> ParseResult<T>.or(
    that: () -> ParseResult<T>
): ParseResult<T> =
    when (this) {
        is Success -> this
        is Failure -> that()
    }

fun <T> givenThat(
    cond: Boolean,
    body: () -> Success<T>
): ParseResult<T> =
    when {
        cond -> body()
        else -> Failure
    }

fun <T> ParseResult<T>.orNull(): Success<T>? = this as?
Success<T>

fun <T> sequence(
  start: ParseResult<T>,
  step: (String) -> ParseResult<T>
): ParseResult<List<T>> =
    Success(
        value = generateSequence(start.orNull()) { last ->
            step(last.remaining).orNull()
        }.toList(),
        remaining = ""
```

```
    ).filter {
        it.isNotEmpty()
    }.flatMap { list, _ ->
        Success(list.map { it.value },
list.last().remaining)
    }
```

Listing 9-6 Helper Functions for the Parser

Now that we have minimal parsing support in place, we can start working on parsing equations. Note that we are assuming there are no spaces in the formula because dealing with them everywhere would be tedious, and they can easily be filtered out at the top level. First, we need to identify all the element symbols and functions necessary for recognizing string patterns and natural numbers.

```
private val elements = setOf(
    "H", "He", "Li", "Be", "B", "C", "N", "O", "F", "Ne",
    "Na", "Mg", "Al", "Si", "P", "S", "Cl", "Ar",
    "K", "Ca", "Sc", "Ti", "V", "Cr", "Mn", "Fe", "Co",
    "Ni", "Cu", "Zn", "Ga", "Ge", "As", "Se", "Br", "Kr",
    "Rb", "Sr", "Y", "Zr", "Nb", "Mo", "Tc", "Ru", "Rh",
    "Pd", "Ag", "Cd", "In", "Sn", "Sb", "Te", "I", "Xe",
    "Cs", "Ba", "La", "Ce", "Pr", "Nd", "Pm", "Sm",
"Eu",  // etc.
)

fun parsePattern(string: String, pattern: String):
ParseResult<String> =
    givenThat(string.startsWith(pattern)) {
        Success(pattern, string.drop(pattern.length))
    }

fun parseNum(string: String): ParseResult<Int> =
    string.takeWhile { it.isDigit() }.length.let {
digitCount ->
        givenThat(digitCount > 0) {
            Success(string.take(digitCount).toInt(),
                    string.drop(digitCount))
        }
    }
```

Listing 9-7 Low-Level Parts of the Parser

The simplest function is `parsePattern()`, which tries to find a given prefix in the string. The function `parseNum()` is a bit more complicated, as it needs to determine the number of digits first. With one exception (the `findElement()` function), all the other functions don't read the string directly, but use these two low-level functions and combine the results in some way – that's why this approach is called "combinatory parsing."

The first example of this "assembling" is the function for reading the equation arrow, which can be either `->` or `<->`.

```
fun parseArrow(string: String): ParseResult<Arrow> =
    parsePattern(string, "<->").map { Arrow.REVERSIBLE } or
        { parsePattern(string, "->").map {
Arrow.IRREVERSIBLE } }
```

Listing 9-8 Parsing the Arrow

Reading an element is not difficult; the only pitfall is that two-letter symbols must be checked before single-letter symbols; otherwise the function would find just `H` (hydrogen) in a string starting with `He` (helium).

> **Caution** It is a common problem that two parsers may match for the same input. Usually, the parser that reads the longer prefix is the one you want to run, so make sure to evaluate it first.

```
fun parseElement(string: String): ParseResult<Element> =
    findElement(string, 2).or {
        findElement(string, 1)
    }.flatMap { symbol, s ->
        parseNum(s).flatMap { subscript, s1 ->
            Success(Element(symbol, subscript), s1)
        } or {
            Success(Element(symbol, 1), s)
        }
    }

fun findElement(string: String, charCount: Int):
ParseResult<String> =
    givenThat(elements.contains("$string##".take(charCount)))
{
        Success(string.take(charCount),
string.drop(charCount))
```

```
    }
```

Listing 9-9 Parsing Elements

First, the `findElement()` function tries to find elements, first with two characters, then – if that fails – with one character. Artificially extending the string with some characters that definitely won't match (here #) avoids a possible `IndexOutOfBoundException`. The `flatMap` block in `parseElement()` tries to find a trailing number. If the number is found, it is used to construct the element; otherwise the default subscript of one is used. Now we are ready to take care of the groups.

```
fun parsePart(string: String): ParseResult<Part> =
    parseElement(string) or { parseGroup(string) }

fun parseGroup(string: String): ParseResult<Group> =
    parsePattern(string, "(").flatMap { _, s1 ->
        sequence(parsePart(s1)) { remaining ->
            parsePart(remaining)
        }
    }.flatMap { parts, remaining ->
        parsePattern(remaining, ")")
            .flatMap { _, s3 -> Success(parts, s3) }
    }.flatMap { parts, s ->
        parseNum(s).flatMap { subscript, s1 ->
            Success(Group(parts, subscript), s1)
        } or {
            Success(Group(parts, 1), s)
        }
    }
```

Listing 9-10 Parsing Groups and Parts

The `parsePart()` method reads either an element symbol or a group. The `parseGroup()` first looks for an opening parenthesis. Then it tries to read as many parts as possible, but at least one. After that, it looks for a closing parenthesis. The final `flatMap()` call handles an optional subscript for the whole group, similar to `parseElement()`. With this preparation, everything is in place to assemble a molecule.

```
fun parseMolecule(string: String): ParseResult<Molecule> =
    (parseNum(string) or { Success(1, string) })
        .flatMap { coefficient, s ->
```

```
            sequence(parsePart(s)) { remaining ->
                parsePart(remaining)
            }.flatMap { parts, remaining ->
                Success(Molecule(coefficient, parts),
remaining)
            }
        }
```

Listing 9-11 Parsing Molecules

First, the function looks for a possible coefficient in front; otherwise, it uses one by default. Then it tries to read as many element or group parts as possible. If some parts are found, the molecule is built; otherwise, the parser fails. This is the parser for the left and right sides of the equation.

```
fun parseSide(string: String): ParseResult<List<Molecule>> =
    sequence(parseMolecule(string)) { remaining ->
        parsePattern(remaining, "+")
            .flatMap { _, s2 -> parseMolecule(s2) }
    }
```

Listing 9-12 Parsing the Sides of the Formula

The function generates a list of molecules, while requiring that there is always a + in between. Now the parser for the whole equation can be written as follows.

```
fun parseEquation(string: String): ParseResult<Equation> =
    parseSide(string).flatMap { lhs, s1 ->
        parseArrow(s1).flatMap { arrow, s2 ->
            parseSide(s2).flatMap { rhs, s3 ->
                Success(Equation(lhs, arrow, rhs), s3)
            }
        }
    }
```

Listing 9-13 Parsing Formulas

It simply reads the left side, the arrow symbol, the right side, and combines them. Now all we need is an `equation()` function, which is the entry point of our DSL.

```
fun equation(string: String): Equation? =
    parseEquation(string.replace(" ", ""))
        .orNull()
        ?.let { result ->
```

```
            result.value.takeIf { result.remaining.isEmpty()
}
        }
```

Listing 9-14 Parsing Equations

This function removes all spaces from the input string, calls the parser, checks that no unparsed string part is left, and returns the result, or if a parse failure occurred, `null`. Now we can write, for instance, which is probably as concise as it gets for an internal DSL.

As mentioned in the previous section, a real chemical equation looks closer to $3Ba(OH)_2 + 2H_3PO_4 \rightarrow 6H_2O + Ba_3(PO_4)_2$, and with some simple modifications, we could allow this syntax as well. In general, it is relatively easy to make the syntax of a string-based DSL more lenient, while other DSL categories often struggle with this kind of flexibility.

Using a Parser Library

As already mentioned, it's not difficult to write such a parser by hand. However, using a library has many advantages: it improves readability and maintainability, the code is easier to debug, you get more information if the parsing fails, and the library is usually better tested than our manual code.

To give you an idea of what using a parser library looks like, I rewrote the example code using the better-parse project,[4] which is an example of the parser-combinator approach.

```
private val elements = setOf(
    "H", "He", "Li", "Be", "B", "C", "N", "O" // etc.
)

val equationGrammar = object : Grammar<Equation>() {

    val ws by regexToken("\\s+", ignore = true)

    val irreversible by literalToken("->")

    val reversible by literalToken("<->")

    val plus by literalToken("+")

    val leftPar by literalToken("(")

    val rightPar by literalToken(")")
```

```
    val num by regexToken("\\d+")

    val symbol by token { cs, from ->
        when {
            elements.contains("$cs##".substring(from, from +
2)) -> 2
            elements.contains("$cs##".substring(from, from +
1)) -> 1
            else -> 0
        }
    }

    val arrow: Parser<Arrow> by (irreversible asJust
Arrow.IRREVERSIBLE) or
            (reversible asJust Arrow.REVERSIBLE)

    val number: Parser<Int> by (num use { text.toInt() })

    val element: Parser<Element> by (symbol and
optional(number))
        .map { (s, n) -> Element(s.text, n ?: 1) }

    val group: Parser<Group> by (skip(leftPar) and
            oneOrMore(parser(this::part)) and
            skip(rightPar) and
            optional(number))
        .map { (parts, n) -> Group(parts, n ?: 1) }

    val part: Parser<Part> = element or group

    val molecule: Parser<Molecule> = (optional(number) and
oneOrMore(part))
        .map { (n, parts) -> Molecule(n ?: 1, parts) }

    val side: Parser<List<Molecule>> = separated(molecule,
plus)
        .map { it.terms }

    override val rootParser: Parser<Equation> by (side and
arrow and side)
        .map { (lhs, a, rhs) -> Equation(lhs, a, rhs) }
}
```

```
// calling an example string
val eq = equationGrammar
    parseToEnd("3Ba(OH)2 + 2H3PO4 -> 6H2O + Ba3(PO4)2")
```

Listing 9-15 Using a Parser Library

Going into the details of this particular library is beyond the scope of this book; the important point is how much using a parser-combinator library can improve readability and simplify the implementation. However, you can still see the same pieces of grammar, assembled in a similar way to our original code.

Summary

String-based DSLs allow for very idiomatic syntax, and can also give the user some leeway by being more forgiving than other DSL types. The drawbacks are a lack of compile-time checks, less tooling support (e.g., autocomplete features), and, on the implementation side, the complexity and overhead of parsing, and the difficulty of extending them later.

Common Applications

- Data creation and initialization
- Data transformation
- Defining operations
- Execute actions
- Code generation
- Testing
- Natural language processing
- Configuration management

Pros

- Allows almost any syntax, very flexible and extensible
- It is easy to allow for some leniency
- Parser libraries help to write readable parser code

Cons

- No compile-time checks
- No tooling support like code suggestions or autocomplete when using the DSL
- Learning curve for using parser libraries
- Dependence on a parser library
- Can be difficult to extend at a later point in time

What's Next?

The next chapter covers a ubiquitous – and sometimes overused – type of DSL in Kotlin and Java: Annotations. Incidentally, due to their limitations, annotations often rely on string-parsing DSLs, such as SQL queries in JPA annotations.

Footnotes

1 Rudyard Kipling in his speech "The Power of Words," Royal College of Surgeons, London, 1923.

2 For an illustrated explanation, see https://www.chess.com/terms/fen-chess.

3 This "half-move clock" is necessary because of the 50-move rule, which prevents the players from shuffling pieces endlessly without making any progress.

4 https://github.com/h0tk3y/better-parse

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
D. Gronau, *Creative DSLs in Kotlin*
https://doi.org/10.1007/979-8-8688-2821-8_10

10. Annotation-Based DSLs

Daniel Gronau[1] ✉
(1) Sandersdorf, Sachsen-Anhalt, Germany

> *I have discovered a truly marvelous proof of this, which however the margin is not large enough to contain.*
> —Pierre de Fermat[1]

Annotations can be used to associate data, actions, or behaviors with classes, methods, fields, and so on. Examples include dependency injection frameworks, such as Spring or Dagger; serializers for formats like XML or JSON, such as Gson, Jackson, or JAXB; and authorization and access control, such as Spring Security. In Java, Project Lombok[2] aims to reduce the boilerplate code in various ways, and uses annotations to do so.

There are two possible approaches for using annotations. One option is to have an entry point that is aware of the annotations. For instance, a serializer can follow instructions on a class to generate the desired format. The second option is to use annotation processors to execute actions, such as code generation, during the build process.

Case Study: Mapper DSL

Mapping between similar classes is a common and often tedious task. Fortunately, there are already many solutions to help with this, such as the MapStruct library,[3] which is a code generator for mapper classes.

In this case study, we will write a very simple mapper DSL to transform data classes at runtime. For simplicity, we will only allow one-to-one relations between the properties. The actual transformation code is not for the faint of heart; it relies on reflection and is very insecure to keep it short. Explaining the details is beyond the scope of this book; the focus of the case study is on how to use annotations to express your intentions.

Let's start with an example. Suppose we have a `User` class and a `Person` class, and we want to convert users to persons.

```
data class User(
    val id: UUID,
    val firstName: String,
    val familyName: String,
    val birthDay: ZonedDateTime
)

data class Person(
    val firstName: String,
    val lastName: String,
    val age: Int
)
```

Listing 10-1 Example Classes for Mapping

We have to consider the following cases:

1. If the source and target parameters have the same name and type, like `firstName`, the mapper should just pass the values.
2. If the source and target parameters have the same type but different names, like `familyName` and `lastName`, we need to specify a mapping between source and target.
3. If the source and target parameters have different types, like `birthday` and `age`, we need to specify some kind of transformer in addition to the source and target.

This is what our DSL might look like.

```
@Mapping("familyName", "lastName")
@Mapping("birthDay", "age", AgeTransformer::class)
object UserToPerson : Mapper<User, Person>()

val user = User(
    UUID.randomUUID(), "Andrew", "Wiles",
    ZonedDateTime.parse("1953-04-11T00:00:00Z")
)
val person = UserToPerson.map(user)
```

Listing 10-2 DSL Usage Example

The definition of the `@Mapping` annotation is straightforward: it must be present at runtime, it should only be valid on classes, and it must be possible to use multiple `@Mapping` annotations on one target (which requires the `@Repeatable` annotation). Since the annotation fields cannot be null, we need to use a dummy default class for the `transformer` field.

```
@Repeatable
@Target(AnnotationTarget.CLASS)
@Retention(AnnotationRetention.RUNTIME)
annotation class Mapping(
    val source: String,
    val target: String,
    val transformer: KClass<*> = Nothing::class
)
```

Listing 10-3 Definition of the Mapping Annotation

The `AgeTransformer` must be a subclass of a function from `ZonedDateTime` to `Int`. If the annotation specifies a class that isn't a function (like the default, which is `Nothing`), we will simply ignore it.

Since the transformer is immutable and reusable, we can implement it as an `object`.

```
object AgeTransformer : (ZonedDateTime) -> Int {
    override fun invoke(z: ZonedDateTime) =
        ChronoUnit.YEARS.between(z,
ZonedDateTime.now()).toInt()
}
```

Listing 10-4 A Transformer Class

Alternatively, we could implement a helper class `Transformer`, which allows a slightly more concise definition of such transformer objects.

```
open class Transformer<in A, out B>(
    val lambda: (A) -> B
) : (A) -> B {
    override fun invoke(a: A): B = lambda(a)
}

object AgeTransformerViaHelper :
Transformer<ZonedDateTime, Int>({ z ->
    ChronoUnit.YEARS.between(z,
ZonedDateTime.now()).toInt()
})
```

Listing 10-5 Using a Helper Class for Transformers

Now the only thing missing is the `Mapper` class, which has to figure out which fields to map and which transformers to use. As mentioned before, this code is neither pretty nor particularly failsafe, but here it is.

```
abstract class Mapper<S : Any, T : Any> {
    private val mappingAnnotations: Map<String,
Mapping> =
```

```
        this::class.findAnnotations<Mapping>
().associateBy { it.target }
    private val sourceClass: KClass<S> =
findMapperClassGeneric(0)
    private val targetClass: KClass<T> =
findMapperClassGeneric(1)
    private val sourceProperties: Map<String,
KProperty1<S, *>> =
        sourceClass.memberProperties.associateBy {
it.name }
    private val targetConstructor: KFunction<T> =
        targetClass.primaryConstructor?: error(
            "${targetClass.simpleName} has no
primary constructor")

    @Suppress("UNCHECKED_CAST")
    private fun <A : Any>
findMapperClassGeneric(index: Int): KClass<A> =
        this::class.allSupertypes.first {
it.classifier == Mapper::class }
            .arguments[index].type!!.classifier as
KClass<A>

    private fun applyTransformer(mapping:
Mapping?, value: Any?): Any? {
        val transformerClass =
mapping?.transformer
            ?.takeIf {
it.isSubclassOf(Function1::class) }
            ?: return value
        val transformer =
transformerClass.objectInstance
            ?:
transformerClass.primaryConstructor?.call()
```

```
            ?: error("Cannot instantiate
transformer")
        @Suppress("UNCHECKED_CAST")
        return (transformer as Function1<Any?,
Any?>)(value)
    }

    fun map(source: S): T =
        targetConstructor.parameters.associateWith
{ parameter ->
            val targetName = parameter.name
                ?: error("Unnamed constructor
parameter")
            val mapping =
mappingAnnotations[targetName]
            val sourceName = mapping?.source ?:
targetName
            val value =
sourceProperties[sourceName]?.get(source)
                ?: error("No source property
'$sourceName'")
            applyTransformer(mapping, value)
        }.let { args ->
targetConstructor.callBy(args) }
}
```

Listing 10-6 The Mapper Class

Since the code uses reflection, you need to include a dependency on the `Kotlin-reflect` library, as described in Chapter 4. Of course, the example could be improved in many ways; for example, sometimes you need multiple source fields to calculate a target field, and converting fields with generic types doesn't work in the current version.

Tip One design choice in our example deserves attention: if the names and types of the source and target classes match, no `@Mapping` annotation is required. This implicit mapping behavior may seem obvious and trivial, but it highlights the principle of *Convention over Code*, which suggests that common use cases should work seamlessly without requiring explicit instructions or extra configuration. It applies to language design in general, but seems to apply more often to annotation-based DSLs than to other DSL categories. Adherence to this principle can improve the user experience and greatly enhance readability.

For serious applications, I would suggest checking out the already mentioned MapStruct project. It is a Java library, but has Kotlin support, and provides much more functionality than our example DSL. A major difference is that MapStruct generates source code, which avoids the performance hit of using reflection, and makes debugging much more convenient.

Synergy with String-Based DSLs

Syntactically, annotation-based DSLs are quite limited: The structure of an annotation is fixed, and only a few data types are allowed as fields. Fortunately, one of these data types is `string`, and the last chapter showed how expressive string-based DSLs can be. It is therefore natural to overcome the limitations of the annotation-based approach by embedding string-parsing DSLs in annotations. We wouldn't gain new insight from implementing such a DSL, but Spring Data JPA can serve as an example.

```
@Repository
interface UserRepository :
JpaRepository<UserEntity, Long> {
    @Query("SELECT u FROM UserEntity u WHERE
u.lastLame = :lastName")
```

```
    fun findAllByLastName(@Param("lastName")
familyName: String):
       List<UserEntity>
}
```

Listing 10-7 Using String-Parsing DSLs Inside Annotations

The `@Query` annotation has no fields for the `FROM` and `WHERE` clauses; it allows the entire query to be specified as a string (which is itself a DSL). In my opinion, this is clearly the better approach for this use case.

Summary

In some cases, it feels very natural to integrate a DSL into the existing user code and use it to influence how certain structures are processed or translated. In these cases, annotation-based DSLs are a good choice. While these DSLs are often easy to use, the implementation overhead can be significant. Another problem can be the overuse of annotations to the point of unreadability, and the mixing of annotations from different frameworks on the same class, method, or property, which can be very confusing.

Common Applications

- Data creation and initialization
- Data transformation
- Data validation
- Execute actions
- Code generation
- Configuring systems
- Testing
- Logging
- Monitoring
- Reporting and analytics

Pros

- Can feel very natural and intuitive to use
- Uses a common, dedicated syntax
- Can support "convention over code" by marking only the special cases

Cons

- May pollute host code
- Can't be used for external code
- May conflict with other annotation-based DSLs
- Often relies heavily on reflection
- Can be hard to debug

What's Next?

In practice, you often won't find different DSL types neatly separated. While mixing different types can yield great synergies, it also comes with its own set of challenges. The next chapter discusses such hybrid DSLs. And, as promised in Chapter 9, the chemical equations make a comeback.

Footnotes

1 Handwritten note by Pierre de Fermat in his copy of Diophantos's *Arithmetica*, referring to his famous "Last Theorem." It is very likely that he only had a proof for simple cases and assumed that these results could be generalized. The theorem was proven 358 years later, in 1994, by Andrew Wiles.

2 https://projectlombok.org

3 https://mapstruct.org

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
D. Gronau, *Creative DSLs in Kotlin*
https://doi.org/10.1007/979-8-8688-2821-8_11

11. Hybrid DSLs

Daniel Gronau[1] ✉
(1) Sandersdorf, Sachsen-Anhalt, Germany

> *In diversity, there is beauty and strength.*
> —Dr. Maya Angelou[1]

In many cases, real-world DSLs are difficult to categorize because they use a mix of different techniques. For example, some parts might resemble a builder, others might look like an algebraic DSL, and a particular feature uses annotations, too. I call that a *hybrid DSL*.

If a DSL can mostly be written in one style, it's usually best to stick to that style to ensure consistency. For example, I wouldn't recommend mixing builder and loan pattern styles. However, there's nothing wrong with designing a DSL that combines different approaches if it gets the job done and provides a coherent language.

Quasi-Lingual DSLs

One common reason for combining features from different DSL categories is to mimic natural language to some extent. I would call this important subset of hybrid DSLs *quasi-lingual*. Although their syntax is rigid and contains artifacts such as braces, parentheses, and commas, these DSLs are still "readable" in the literal sense. The following snippet from the MockK library[2] should give you a good idea.

```
every {
    car.door(DoorType.FRONT_LEFT).windowState()
} returns WindowState.UP
```

Listing 11-1 MockK As an Example for a Quasi-Lingual DSL

Technically, writing a quasi-lingual DSL is not very different from writing other hybrid DSLs, although it may be more difficult to model grammatical structures correctly. Naming conflicts with keywords such as `is`, `as`, `when`, or `for` are also more problematic. A good rule of thumb is to keep the DSL grammar simple and well-structured, and to compromise when necessary to reduce complexity. For instance, rather than modeling words such as `should` and `be` separately, it might be preferable to use `shouldBe` if doing so helps keep the grammar simple.

Case Study: Chemical Equations

Writing a DSL for chemical equations is not an easy task because the notation is very concise and cannot easily be mimicked using Kotlin syntax. For our case study, we will attempt to recreate the string-based DSL from Chapter 9 using a hybrid approach. We can use the same underlying model classes.

```
sealed interface Part

data class Element(
    val symbol: String,
    val subscript: Int = 1
): Part {
    override fun toString() = symbol +
subscript.oneAsEmpty()
}

data class Group(
    val parts: List<Part>,
```

```
    val subscript: Int = 1
): Part {
    override fun toString() =
        parts.joinToString("", "(", ")") +
                subscript.oneAsEmpty()
}

data class Molecule(
    val coefficient: Int,
    val parts: List<Part>
) {
    constructor(coefficient: Int, vararg parts:
Part) :
            this (coefficient, parts.toList())
    constructor(vararg parts: Part) :
            this (1, parts.toList())
    override fun toString() =
coefficient.oneAsEmpty() +
            parts.joinToString("")
}

enum class Arrow(val symbol: String) {
    IRREVERSIBLE("->"),
    REVERSIBLE("<->")
}

data class Equation(
    val leftSide: List<Molecule>,
    val arrow: Arrow,
    val rightSide: List<Molecule>
) {
    override fun toString() = listOf(
        leftSide.joinToString(" + "),
        arrow.symbol,
        rightSide.joinToString(" + ")
```

```
    ).joinToString(" ")
}

private fun Int.oneAsEmpty(): String =
    takeIf { this > 1 }?.toString().orEmpty()
```

Listing 11-2 Expressing Chemical Equations

As a refresher, Figure 11-1 shows the corresponding structure diagram.

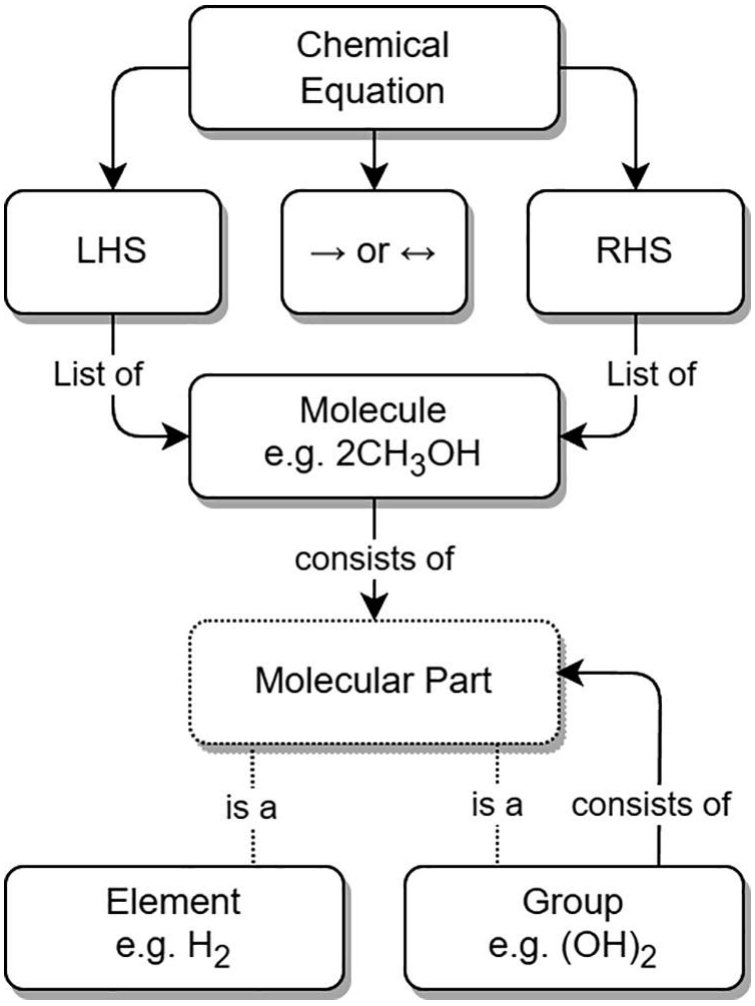


Figure 11-1 Model for Chemical Equations

As a first step toward a DSL, we define all the elements.

```
val H = Element("H")
```

```
val He = Element("He")
val Li = Element("Li")
// etc.
```

Listing 11-3 Element Definitions

Next, we need a convenient way to add a subscript to an element or group. We could use the invoke operator `()` or the index access operator `[]`. Since parentheses will be needed elsewhere, the index access operator appears to be the better choice. We can now write `H[2]` to denote H_2.

```
operator fun Element.get(subscript: Int) =
    apply { require(this.subscript == 1 &&
subscript > 1) }
        .copy(subscript = subscript)

operator fun Group.get(subscript: Int) =
    apply { require(this.subscript == 1 &&
subscript > 1) }
        .copy(subscript = subscript)
```

Listing 11-4 Extension Functions for Subscripts

Both functions include a sanity check that prevents nonsensical calls such as `H[-5]` or `H[2][7]`.

The next task is to assemble molecules from either elements or groups. We can use the minus operator − to represent a chemical bond. For example, a water molecule could be written as `H[2]-O`.

```
operator fun Part.minus(that: Part) =
    Molecule(1, this, that)

operator fun Molecule.minus(that: Part) =
    copy(parts = parts + that)

operator fun Element.minus(that: Part) =
```

```
    Group(listOf(this, that), 1)

operator fun Group.minus(that: Part) =
    copy(parts = parts + that)
```

Listing 11-5 Extension Functions for Chemical Bonds

Note that, in general, groups and sometimes molecules must be placed in parentheses due to precedence rules.

A molecule may have an optional coefficient in front of it. Also, the same multiplication operation should "promote" an element or group to a molecule. For example, we should be able to write `3*O[2]` to represent an oxygen molecule with coefficient three. We introduce the extension function `Part.toMolecule()` for that kind of promotion. Again, we need sanity checks for invalid inputs such as `-2*H[2]` or `3*(2*O[2])`.

```
operator fun Int.times(that: Molecule) =
    that.apply { require(coefficient == 1 &&
this@times > 1) }
        .copy(coefficient = this)

operator fun Int.times(that: Part) =
that.toMolecule(this)
    .apply { require(coefficient > 1) }

fun Part.toMolecule(coefficient: Int = 1):
Molecule = when(this) {
    is Group if subscript == 1 ->
        Molecule(coefficient, this.parts)
    is Part ->
        Molecule(coefficient, this)
}
```

Listing 11-6 Extension Functions for Coefficients

Next, we need a way to group the left and right sides of an equation into a list of molecules. The obvious choice for this is the plus operator +. As before, we "promote" molecule parts to full molecules when necessary. This time, the precedence rules for * and + align with the intended use, so parentheses aren't necessary at this level.

```
operator fun Molecule.plus(that: Molecule)
:List<Molecule> =
    listOf(this, that)

operator fun Molecule.plus(that: Part)
:List<Molecule> =
    listOf(this, that.toMolecule())

operator fun Part.plus(that: Molecule)
:List<Molecule> =
    listOf(this.toMolecule(), that)

operator fun List<Molecule>.plus(that: Part)
:List<Molecule> =
    this + that.toMolecule()
```

Listing 11-7 Extension Functions for Grouping Molecules

In case you're wondering why there's no `List<Molecule>.plus(that: Molecule)` function: it would just be a special case of adding elements to a list, which is already defined in the standard library.

The final step in the DSL involves collecting everything into an equation, This part is a bit tedious, because we may have not only lists of molecules but also single molecules or parts of molecules on both sides of the equation. To simplify this task, we introduce the `side` helper functions. We must also distinguish between reversible and irreversible equations.

We can use the backtick notation for the arrows. Since the characters in `` `->` `` and `` `<->` `` are not valid on the JVM, we must

rename the functions there using the `@JvmName` annotation. Unfortunately, we also have to suppress the corresponding compiler warnings.

```
private fun side(list: List<Molecule>):
List<Molecule> = list
private fun side(molecule: Molecule) =
listOf(molecule)
private fun side(part: Part) =
listOf(part.toMolecule())

@JvmName("reactsTo")
@Suppress("INVALID_CHARACTERS")
infix fun List<Molecule>.`->`(that:
List<Molecule>) =
    Equation(side(this), Arrow.IRREVERSIBLE,
side(that))

@JvmName("reactsTo")
@Suppress("INVALID_CHARACTERS")
infix fun Molecule.`->`(that: List<Molecule>) =
    Equation(side(this), Arrow.IRREVERSIBLE,
side(that))

@JvmName("reactsTo")
@Suppress("INVALID_CHARACTERS")
infix fun List<Molecule>.`->`(that: Molecule) =
    Equation(side(this), Arrow.IRREVERSIBLE,
side(that))

@JvmName("reactsTo")
@Suppress("INVALID_CHARACTERS")
infix fun Molecule.`->`(that: Molecule) =
    Equation(side(this), Arrow.IRREVERSIBLE,
side(that))
```

```
@JvmName("reactsTo")
@Suppress("INVALID_CHARACTERS")
infix fun Part.`->`(that: List<Molecule>) =
    Equation(side(this), Arrow.IRREVERSIBLE,
side(that))

@JvmName("reactsTo")
@Suppress("INVALID_CHARACTERS")
infix fun List<Molecule>.`->`(that: Part) =
    Equation(side(this), Arrow.IRREVERSIBLE,
side(that))

@JvmName("reactsTo")
@Suppress("INVALID_CHARACTERS")
infix fun Part.`->`(that: Part) =
    Equation(side(this), Arrow.IRREVERSIBLE,
side(that))

@JvmName("reactsTo")
@Suppress("INVALID_CHARACTERS")
infix fun Part.`->`(that: Molecule) =
    Equation(side(this), Arrow.IRREVERSIBLE,
side(that))

@JvmName("reactsTo")
@Suppress("INVALID_CHARACTERS")
infix fun Molecule.`->`(that: Part) =
    Equation(side(this), Arrow.IRREVERSIBLE,
side(that))

@JvmName("reactsReversableTo")
@Suppress("INVALID_CHARACTERS")
infix fun List<Molecule>.`<->`(that:
List<Molecule>) =
```

```
    Equation(side(this), Arrow.REVERSIBLE,
side(that))

@JvmName("reactsReversableTo")
@Suppress("INVALID_CHARACTERS")
infix fun Molecule.`<->`(that: List<Molecule>) =
    Equation(side(this), Arrow.REVERSIBLE,
side(that))

@JvmName("reactsReversableTo")
@Suppress("INVALID_CHARACTERS")
infix fun List<Molecule>.`<->`(that: Molecule) =
    Equation(side(this), Arrow.REVERSIBLE,
side(that))

@JvmName("reactsReversableTo")
@Suppress("INVALID_CHARACTERS")
infix fun Molecule.`<->`(that: Molecule) =
    Equation(side(this), Arrow.REVERSIBLE,
side(that))

@JvmName("reactsReversableTo")
@Suppress("INVALID_CHARACTERS")
infix fun Part.`<->`(that: List<Molecule>) =
    Equation(side(this), Arrow.REVERSIBLE,
side(that))

@JvmName("reactsReversableTo")
@Suppress("INVALID_CHARACTERS")
infix fun List<Molecule>.`<->`(that: Part) =
    Equation(side(this), Arrow.REVERSIBLE,
side(that))

@JvmName("reactsReversableTo")
@Suppress("INVALID_CHARACTERS")
```

```
infix fun Part.`<->`(that: Part) =
    Equation(side(this), Arrow.REVERSIBLE,
side(that))

@JvmName("reactsReversableTo")
@Suppress("INVALID_CHARACTERS")
infix fun Part.`<->`(that: Molecule) =
    Equation(side(this), Arrow.REVERSIBLE,
side(that))

@JvmName("reactsReversableTo")
@Suppress("INVALID_CHARACTERS")
infix fun Molecule.`<->`(that: Part) =
    Equation(side(this), Arrow.REVERSIBLE,
side(that))
```

Listing 11-8 Assembling the Equation

So, what does our DSL look like in action? Here are a few examples.

```
//2H2 + O2 -> 2H2O
val makingWater =
    2*H[2] + O[2] `->` 2*(H[2]-O)

//3Ba(HO)2 + 2H3PO4 -> 6H2O + Ba3(PO4)2
val makingBariumPhosphate =
    3*(Ba-(O-H)[2]) + 2*(H[3]-P-O[4]) `->`
        6*(H[2]-O) + (Ba[3]-(P-O[4])[2])

//H2SO4 + 8HI <-> H2S + 4I2 + 4H2O
val sulfuricAcidAndHydrogenIodide =
    H[2]-S-O[4] + 8*(H-I) `<->`
        (H[2]-S) + 4*I[2] + 4*(H[2]-O)

//CuSO4 + 4H2O -> [Cu(H2O)4]SO4
val copperSulfateComplex =
```

```
    Cu-S-O[4] + 4*(H[2]-O) `->`
        (Cu-(H[2]-O)[4])-S-O[4]
```

Listing 11-9 Usage Examples

We can also include an optional improvement, which is a matter of taste: we could add some extension properties for low subscript numbers of elements and groups. For example, we could write `N._2` instead of `N[2]`.

```
val Element._2
    get() = this.apply { require(subscript == 1)
}.copy(subscript = 2)
val Element._3
    get() = this.apply { require(subscript == 1)
}.copy(subscript = 3)
val Element._4
    get() = this.apply { require(subscript == 1)
}.copy(subscript = 4)
// etc.

val Group._2
    get() = this.apply { require(subscript == 1)
}.copy(subscript = 2)
val Group._3
    get() = this.apply { require(subscript == 1)
}.copy(subscript = 3)
val Group._4
    get() = this.apply { require(subscript == 1)
}.copy(subscript = 4)
// etc.

// alternative syntax

val makingBariumPhosphateSub = 3*(Ba-(O-H)._2) +
2*(H._3-P-O._4) `->`
```

```
6*(H._2-O) + (Ba._3-(P-O._4)._2)
```

Listing 11-10 Alternative Implementation for Subscripts

Please decide for yourself which version you prefer. I personally find the syntax with the index operator `[]` more readable. However, there is nothing wrong with providing users with various options as long as they don't interfere with each other and the DSL remains readable.

Simulating dense chemical notation is difficult. While using operator overloading and infix notation made our example substantially shorter, it still contains a lot of clutter. Of course, one would get used to the DSL over time, but there is clearly a learning curve involved. As you've seen, the same problem can be tackled with a string-based DSL, which seems to be the more elegant approach in this case. Nevertheless, it's impressive how far you can push the syntax toward such specific notation in Kotlin.

Case Study: Pattern Matching

While Kotlin's `when` expression is more versatile than Java's `switch`, languages like Scala and Haskell take it a step further by providing *pattern matching*. This feature allows you to deconstruct and match values against specific patterns. Pattern matching provides a concise and powerful way to perform conditional branching and data extraction based on the structure and content of the input.

With pattern matching, you define a set of patterns describing the possible forms or values an input can have. These patterns can include literals, variables, data constructors, or more complex patterns, such as lists or tuples. The language matches the input against these patterns and executes the expression associated with the first matching pattern.

In this case study, our goal is to provide similar functionality in Kotlin, though it won't be as elegant as its built-in counterparts in other languages. An ideal syntax might look like this.

```
data class Person(
```

```
    val firstName: String,
    val lastName: String,
    val age: Int
)

val p = Person("Andy", "Smith", 43)

// this is not valid Kotlin, but a suggestion for
the ideal syntax
val result = match(p) {

    Person("Andy", "Miller", _) ->
        "It's Andy Miller!"

    Person("Andy", lastName != "Miller", val age)
->
        "Some other Andy of age $age."

    else -> "Some unknown person."
}
```

Listing 11-11 Our Ideal Syntax for Pattern Matching

However, we have to make some compromises to make it work in Kotlin:

- We can't use `Person` in the match cases, so we use a helper function called `person`.
- It is difficult to support a mix of literal values and patterns, so we need to wrap values such as `"Andy"` in a pattern. For this purpose, we can use the unary plus, and write `+"Andy"` instead. If a value shouldn't match a certain value, it makes sense to use the unary minus, so `-"Andy"` would match any string that isn't `"Andy"`.
- For numbers, the unary operators are already defined, so we fall back on a syntax like `eq(42)` and `!eq(42)`.
- Arrow notation is not possible, so we use `then` instead.

- Comparisons as well as `and` and `or` can only be infix functions, not operators.
- The right sides should be evaluated only when needed, so we have to provide a lambda for lazy evaluation.
- For capturing variables on the left side and using them on the right side, we must define a `val` for the capture pattern beforehand.
- `else` is a keyword, so `otherwise` is used instead. Since it is not possible to determine at compile time whether the given conditions are exhaustive, the `otherwise` branch is mandatory.
- The underscore _ isn't a valid identifier, so we use `any` instead.
- In some cases, we need to provide additional generic type information.

That's a pretty long list, so let's see what our example looks like now, using an achievable syntax.

```
val result = match(p) {

    person(+"Andy", +"Miller", any()) then {
        "It's Andy Miller!"
    }

    val ageCapture = capture<Int>()
    person(+"Andy", -"Miller", ageCapture) then {
", ageCapture) then {
        "Some other Andy of age
${ageCapture.value}."
    }

    otherwise { "Some unknown person." }
}
```

Listing 11-12 A Realistic Syntax for Pattern Matching

That's not too bad, even though the + prefix looks a bit odd at first. The problem is that Kotlin's selection of overridable operators is quite

limited. That's why the unary plus has become the standard for such use cases, and is also used that way in the Kotlin documentation when introducing type-safe builders.[3]

The core of the DSL is quite small. First, there is the pattern type, which is just a Boolean test function. Therefore, we can use a type alias instead of introducing a new interface. The `MatchResult` is just an interface that wraps a given value. The `Matcher` class provides a context for keeping track of the result and defines the `then` and `otherwise` methods. It also introduces the unary plus as an alias for the `eq` pattern (which will be defined later). Finally, the `match()` function ties everything together and acts as the entry point for the DSL.

```
typealias Pattern<P> = (P) -> Boolean

interface MatchResult<T : Any> {
    val value: T
}

class Matcher<P, T : Any>(private val obj: P) {

    private var result: T? = null

    operator fun Any.unaryPlus() = eq(this)

    infix fun Pattern<P>.then(value: () -> T) {
        if (result == null && this(obj)) {
            result = value()
        }
    }

    fun otherwise(default: () -> T) = object :
MatchResult<T> {
        override val value = result ?: default()
    }
```

```
}

fun <P, T : Any> match(
    obj: P,
    body: Matcher<P, T>.() -> MatchResult<T>
): T = Matcher<P, T>(obj).run(body).value
```

Listing 11-13 Pattern Matching Core Functionality

Note that the `body` parameter of the `match()` method requires a `MatchResult` as a return value, and immediately extracts its content. Then why doesn't the block just return a `T` value directly? Requiring a special type is a trick to "convince" users to call the `otherwise()` method at the end of the block, since it's the only obvious way to construct such an instance.

The unary plus as a synonym for the `eq` pattern is defined directly in `Matcher` to prevent name conflicts and unexpected behavior outside of `match` blocks. It's good practice to keep the scope of potentially dangerous or confusing DSL elements as small as possible.

Of course, we still need patterns that can be used on the left side of the `then` infix functions. Most of these are fairly easy to write.

```
// matches everything
fun <P> any(): Pattern<P> = { true }

// matches nothing
fun <P> none(): Pattern<P> = { false }

// matches null values
fun <P> isNull(): Pattern<P> = { it == null }

// negates a pattern
operator fun <P> Pattern<P>.not(): Pattern<P> = {
!this@not(it) }

// conjunction of patterns
```

```
infix fun <P> Pattern<P>.and(that: Pattern<P>):
Pattern<P> =
    { this@and(it) && that(it) }

// disjunction of patterns
infix fun <P> Pattern<P>.or(that: Pattern<P>):
Pattern<P> =
    { this@or(it) || that(it) }

// equality to a value
fun <P> eq(value: P): Pattern<P> = { it == value }

// equality to one of the values
fun <P> oneOf(vararg values: P): Pattern<P> = { it
in values }

// type check
fun <P> isA(kClass: KClass<*>): Pattern<P> =
    { kClass.isInstance(it) }

// instance equality
fun <P> isSame(value: P): Pattern<P> = { it ===
value }
```

Listing 11-14 Simple Patterns for the DSL

Comparing values requires some type checking to ensure that the value is actually comparable.

```
// greater than
fun <C : Comparable<C>> gt(value: C): Pattern<C> =
{ it > value }

// greater or equal
fun <C : Comparable<C>> ge(value: C): Pattern<C> =
{ it >= value }
```

```
// less than
fun <C : Comparable<C>> lt(value: C): Pattern<C> =
{ it < value }

// less or equal
fun <C : Comparable<C>> le(value: C): Pattern<C> =
{ it <= value }
```

Listing 11-15 Comparison Patterns for the DSL

For the predicates `all()`, `any()`, and `none()` on `Iterable` items, we can define corresponding patterns.

```
// checks that all elements match the given
pattern
fun <P> all(p: Pattern<P>) : Pattern<Iterable<P>>
=
    { it.all(p) }

// checks that at least one element matches the
given pattern
fun <P> any(p: Pattern<P>) : Pattern<Iterable<P>>
=
    { it.any(p) }

// checks that no element matches the given
pattern
fun <P> none(p: Pattern<P>) : Pattern<Iterable<P>>
=
    { it.none(p) }
```

Listing 11-16 Iterable Patterns for the DSL

To capture values, we need a class that implements `Pattern<T>`, but can also hold a value.

```
class Capture<P : Any> : Pattern<P> {
```

```
    lateinit var value: P
        private set

    override fun invoke(obj: P) = true.also {
value = obj }
}

inline fun <reified P : Any> capture() =
Capture<P>()
```

Listing 11-17 Capture Patterns for the DSL

To capture values, you first define a value with the `capture<T>()` method. Then you can use it as a pattern on the left side of `then`, which always succeeds, as well as stores the value. On the right side, you can read the value from the predefined `val`. The syntax example in Listing 11-12 already demonstrated the usage.

```
val result = match(p) {
    person(+"Andy", +"Miller", gt(42)) then {
       "It's Andy Miller!"
    }
    val ageCapture = capture<Int>()
    person(+"Andy", -"Miller", ageCapture) then {
        "Some other Andy of age
${ageCapture.value}."
    }
    otherwise { "Some unknown person." }
}
```

Listing 11-18 Using Capture Patterns

We now need a pattern for decomposing a data class. Unfortunately, it is not possible to write code that can handle all data classes at once in a type-safe manner. Therefore, we must write a pattern for each data class that we want to use, but this is easy to do.

```
fun person(
    firstName: Pattern<String> = any(),
    lastName: Pattern<String> = any(),
    age: Pattern<Int> = any()
): Pattern<Person?> = {
    it != null &&
        firstName(it.firstName) &&
        lastName(it.lastName) &&
        age(it.age)
}
```

Listing 11-19 Data Class Patterns for the DSL

For data classes with many arguments, defining `any()` as the default pattern for all arguments is useful, because it allows you to invoke the pattern for the data class with named arguments, and ignore the ones you don't care about. While writing such pattern classes is not difficult, it can quickly become tedious. The next chapter discusses how to generate such boilerplate code.

Despite the complexity of the topic, we were able to develop a fairly usable hybrid DSL by combining language features that complement each other.

Summary

Creating effective hybrid DSLs requires careful consideration and integration of different language features, as well as thorough testing. Sometimes, the necessary language features won't work together seamlessly. In such cases, it may be better to use a single DSL type, which may seem boring but ensures the DSL works reliably and predictably.

However, a well-designed hybrid DSL can combine different language features in an intuitive and organic way. By exploiting the strengths of various DSL types and carefully designing their integration, you can create very powerful and expressive DSLs. Due to the many

degrees of freedom, achieving this can be challenging, so hybrid DSLs typically require extensive testing to address all edge cases.

Common Applications

- Data creation and initialization
- Data transformation
- Defining operations
- Execute actions
- Code generation
- Configuration management
- Testing
- Logging
- Monitoring
- Reporting and analytics

Pros

- Can support a wide range of problems
- Allows you to get creative with different techniques
- Can be very concise due to many implementation options

Cons

- Can look messy
- Higher perceptual complexity can lead to a steeper learning curve
- It can be difficult to control edge cases
- Higher maintenance effort required
- Java interoperability can be challenging

What's Next?

This is the last chapter dedicated to a DSL type. The final three chapters cover supplemental topics. We will start with a subject that was hinted at throughout the book: code generation. It is one of the "big guns" and comes with complexity and a steep learning curve. However, once the code works, it can almost feel like magic.

Footnotes

1 Dr. Maya Angelou, Wouldn't Take Nothing for My Journey Now, Bantam, 1993.

2 https://mockk.io/#dsl-examples

3 https://kotlinlang.org/docs/type-safe-builders.html

Part III
Supplemental Topics

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
D. Gronau, *Creative DSLs in Kotlin*
https://doi.org/10.1007/979-8-8688-2821-8_12

12. Code Generation for DSLs

Daniel Gronau[1] ✉
(1) Sandersdorf, Sachsen-Anhalt, Germany

Do not cite the Deep Magic to me, witch! I was there when it was written.
—C. S. Lewis, *The Lion, the Witch and the Wardrobe*

Sometimes, when prototyping a DSL, you find a nice, expressive syntax, but it turns out that a lot of boilerplate code is needed to make it work. One common reason for this is combinatorial explosion. Or you might need many classes that follow the same pattern, such as tuple classes or fixed-length vectors. In such cases, you might consider using code generation.

One popular library for generating Kotlin code is KotlinPoet,[1] which we will use throughout this chapter.

Case Study: Physical Quantities

Let's say you've already defined quantities for physical units such as seconds, square meters, or Watts.

```
sealed interface Quantity<Q: Quantity<Q>> {
    val amount: Double
}

// base units
data class Second(override val amount: Double) :
    Quantity<Second>
data class Meter(override val amount: Double) :
    Quantity<Meter>
data class Kilogram(override val amount: Double) :
     Quantity<Kilogram>

// derived units
data class SquareMeter(override val amount: Double) :
    Quantity<SquareMeter>
data class CubicMeter(override val amount: Double) :
```

```
    Quantity<CubicMeter>
data class MeterPerSecond(override val amount: Double) :
    Quantity<MeterPerSecond>
data class MeterPerSecondSquared(override val amount: Double) :
    Quantity<MeterPerSecondSquared>
data class Newton(override val amount: Double) :
    Quantity<Newton>
data class Joule(override val amount: Double) :
    Quantity<Joule>
data class Watt(override val amount: Double) :
    Quantity<Watt>
data class Pascal(override val amount: Double) :
    Quantity<Pascal>
```

Listing 12-1 Physical Units

> **Caution** Representing quantities and currency amounts is not trivial. You should always consider whether the precision provided by `Double` is sufficient for the kind of calculations required by your application, and switch, for instance, to `BigDecimal` if not. However, we will stick with `Double` for our example implementation, which is sufficient for demonstration purposes.

You might wonder about the strange type signature `Quantity<Q: Quantity<Q>>`. It allows us to write generic, type-safe functions for the subclasses, in our case addition, subtraction, negation, and scalar multiplication. This technique is known as the *Curiously Recurring Template Pattern* (CRTP) and was popularized in C++, but works in Kotlin and Java as well. Without it, you could still write the same functions, but there would be no way to stop the compiler from accepting, for instance, the addition of `Meter` and `Second`, because it would infer `Quantity` (their common super-type) as the function's generic parameter. Of course, we could still perform runtime checks, but you would clearly prefer to catch this kind of error at compile time.

As the `Quantity` interface is sealed and we therefore know its subclasses and the signature of their constructors, it is safe to use reflection for an implementation:[2]

```
import kotlin.reflect.full.primaryConstructor

operator fun <Q: Quantity<Q>> Q.plus(that: Q) : Q =
    this::class.primaryConstructor!!.call(this.amount +
that.amount)

operator fun <Q: Quantity<Q>> Q.minus(that: Q) : Q =
    this::class.primaryConstructor!!.call(this.amount -
that.amount)
```

```
operator fun <Q: Quantity<Q>> Q.unaryMinus() : Q =
    this::class.primaryConstructor!!.call(-this.amount)

operator fun <Q: Quantity<Q>> Double.times(that: Q) : Q =
    that::class.primaryConstructor!!.call(this * that.amount)
```

Listing 12-2 Addition, Subtraction, Negation, and Scalar Multiplication of Physical Units

The next part is more interesting: we need conversions from and back to `Double`. Note that we can define these conversions for more units than we have classes defined for; for example, we can define not only `5.0.s` to get `Second(5.0)` but also `1.0.min` to get `Second(60.0)`. And vice versa, we want to be able to get not only seconds from a `Second` instance but also minutes and so on. We could write all of this manually, but it would be very tedious, so we will use code generation instead. As a first step, we collect the additional information needed to write the conversions for a certain unit: its name, the quantity subclass, and a factor to apply.

```
private val fromToDouble = listOf(
    Triple("s", Second::class, 1.0),
    Triple("min", Second::class, 60.0),
    Triple("h", Second::class, 3600.0),
    Triple("yr", Second::class, 31_556_925.216),

    Triple("mm", Meter::class, 0.001),
    Triple("cm", Meter::class, 0.01),
    Triple("in", Meter::class, 0.0254),
    Triple("ft", Meter::class, 0.3048),
    Triple("yd", Meter::class, 0.9144),
    Triple("m", Meter::class, 1.0),
    Triple("km", Meter::class, 1000.0),
    Triple("mi", Meter::class, 1609.344),

    // etc.
)
```

Listing 12-3 List of Conversion Factors

To make the DSL a bit more readable, we will generate extension properties instead of extension functions, so we don't have to use brackets. The generating functions can be written in KotlinPoet[3] like this.

```
private fun makeDoubleToQuantity(
    unit: String,
    kClass: KClass<out Quantity<*>>,
    factor: Double
) = PropertySpec.builder(unit, kClass)
```

```
        .receiver(Double::class)
        .getter(
            FunSpec.getterBuilder()
                .addStatement("return %T(this * %L)", kClass,
factor)
                .build()
        )
        .build()

private fun makeQuantityToDouble(
    unit: String,
    kClass: KClass<out Quantity<*>>,
    factor: Double
) = PropertySpec.builder(unit, Double::class)
        .receiver(kClass)
        .getter(
            FunSpec.getterBuilder()
                .addStatement("return this.amount / %L", factor)
                .build()
        )
        .build()

fun makeDoubleToQuantities() =
    fromToDouble.map { (u, k, f) -> makeDoubleToQuantity(u, k,
f) }

fun makeQuantityToDoubles() =
    fromToDouble.map { (u, k, f) -> makeQuantityToDouble(u, k,
f) }
```

Listing 12-4 Code Generator for Unit Conversions

The KotlinPoet library itself uses the builder pattern. As a result, we get lists of `FunSpec` instances, which can represent functions, constructors, getters, or setters. Later, we will add these extension properties to a `FileSpec` that can be written to the file system. The generated functions will look like this.

```
public val Double.kJ: Joule
  get() = Joule(this * 1000.0)

public val Joule.kJ: Double
  get() = this.amount / 1000.0
```

Listing 12-5 Example for Generated Conversions Functions

So far, we can generate a lot of boilerplate code, but for the next task – multiplying and dividing quantities – it would be extremely tedious to write the necessary code by hand, even for our modest example. If we have N physical units, the number of possible multiplications and divisions is of order N^2, and although we won't actually implement all possible combinations, it's still a lot. When we have such polynomial or even exponential growth, we are dealing with a combinatorial explosion.

To tackle this problem, we first need all valid multiplication equations. This could look like this, where the first two values of a triple are the types of the factors, and the third is the type of the product.

```
val multiply = listOf(
    Triple(Meter::class, Meter::class, SquareMeter::class),
    Triple(Meter::class, SquareMeter::class, CubicMeter::class),
    Triple(MeterPerSecond::class, Second::class, Meter::class),
    Triple(MeterPerSecondSquared::class, Second::class,
        MeterPerSecond::class),
    Triple(MeterPerSecondSquared::class, Kilogram::class,
Newton::class),
    Triple(Pascal::class, SquareMeter::class, Newton::class),
    Triple(Newton::class, Meter::class, Joule::class),
    Triple(Watt::class, Second::class, Joule::class),
)
```

Listing 12-6 List of Supported Multiplication of Physical Units

Now we evaluate these equations for both multiplication and division. A slight complication is that we also want to add functions with swapped operands, but only if they are of different types.

```
private fun makeMultiplication(
    in1: KClass<out Quantity<*>>,
    in2: KClass<out Quantity<*>>,
    out: KClass<out Quantity<*>>
) = FunSpec.builder("times")
        .addModifiers(KModifier.OPERATOR)
        .receiver(in1)
        .returns(out)
        .addParameter("that", in2)
        .addStatement("return %T(this.amount * that.amount)",
out)
        .build()

private fun makeDivision(
    in1: KClass<out Quantity<*>>,
    in2: KClass<out Quantity<*>>,
```

```
    out: KClass<out Quantity<*>>
) = FunSpec.builder("div")
        .addModifiers(KModifier.OPERATOR)
        .receiver(in1)
        .returns(out)
        .addParameter("that", in2)
        .addStatement("return %T(this.amount / that.amount)",
out)
        .build()

fun makeMultiplications() =
    multiply.flatMap { (in1, in2, out) ->
        when {
            in1 == in2 -> listOf(makeMultiplication(in1, in2,
out))
            else -> listOf(
                makeMultiplication(in1, in2, out),
                makeMultiplication(in2, in1, out))
        }
    }

fun makeDivisions() =
    multiply.flatMap { (in1, in2, out) ->
        when {
            in1 == in2 -> listOf(makeDivision(out, in1, in2))
            else -> listOf(
                makeDivision(out, in1, in2),
                makeDivision(out, in2, in1))
        }
    }
```

Listing 12-7 Code Generator for Unit Multiplication and Division

This is the result of the function generation process.

```
public operator fun Newton.times(that: Meter) =
    Joule(this.amount * that.amount)

public operator fun Meter.times(that: Newton) =
    Joule(this.amount * that.amount)

public operator fun Joule.div(that: Meter) =
    Newton(this.amount / that.amount)

public operator fun Joule.div(that: Newton) =
```

```
    Meter(this.amount / that.amount)
```

Listing 12-8 Example for Generated Unit Multiplication and Division Functions

To finish the DSL, we need to write the generated code to a file. For simplicity, we will write it right next to the generating file, but it is common to have separate directories for generated code.

```
fun writeGeneratedFile() {
    FileSpec.builder(Quantity::class.java.packageName,
"generated")
        .addProperties(makeQuantityToDoubles())
        .addProperties(makeDoubleToQuantities())
        .addFunctions(makeMultiplications())
        .addFunctions(makeDivisions())
        .build()
        .writeTo(Path.of("./src/main/kotlin/"))
}
```

Listing 12-9 Writing the File Containing the Generated Code

Working with KotlinPoet is pretty straightforward: you use the various spec classes to assemble your code, and the `FileSpec` and `ClassSpec` classes allow you to write the file or class to the file system. Behind the scenes, KotlinPoet does a lot of work for you, such as managing imports or simplifying your code (for instance, converting function bodies with braces to expression syntax where possible).

With our generated DSL in place, we can now calculate physical quantities in a safe and convenient way. For example, after defining the necessary conversions (like `m_s` and `kJ`), we can write.

```
val acceleration = 30.0.m_s / 1.0.s
val force = acceleration * 64.0.kg
val energy = force * 5.0.m
val result = "${energy.kJ} kiloJoule"
// 9.6 kiloJoule
```

Listing 12-10 Usage Example for the Physical Units DSL

The example code is written in such a way that you manually generate the code via the `writeGeneratedFile()` method when the DSL has changed. This is a simple approach if you know that code changes won't happen very often, but it can quickly become cumbersome if changes become more frequent, and is not feasible if you don't know in advance what code needs to be generated. In the next section, we will discuss using an annotation processor instead.

Writing an Annotation Processor Using KSP

There are two APIs for annotation processors in Kotlin. The older one is called `kapt`, which is no longer actively developed, but is still used for many projects. The more modern API is `KSP`, which stands for *Kotlin Symbol Processing*[4].

Before deciding to write an annotation processor, it's important to understand how it works and what its limitations are. In addition to the client code, you will need at least two modules: one module containing annotations, related interfaces, etc., that you can use in your client code to specify your requirements to the annotation processor, and one module containing the annotation processor itself that is integrated into the build process to do things like code generation and reporting, or to provide tooling support.

Figure 12-1 illustrates the dependencies between the KSP modules and client code.

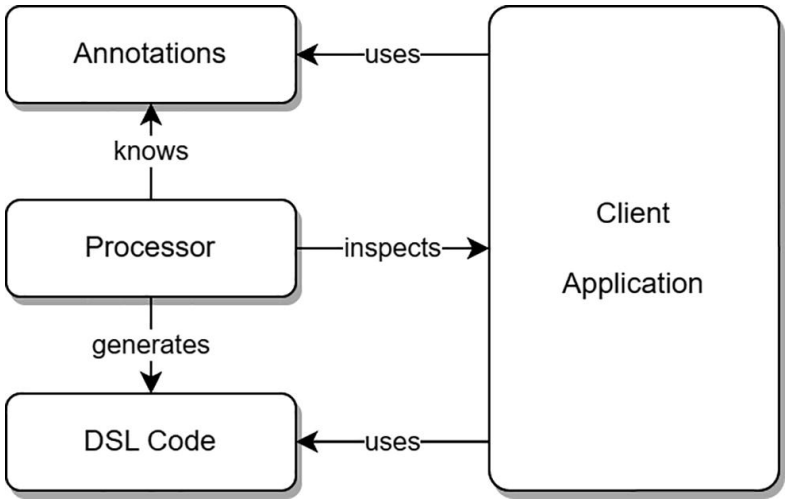


Figure 12-1 Interactions Between KSP Modules

> **Caution** At the time you call KSP, the client code is not yet built, which means **you can't use regular reflection** and no `KClass` instances of client classes. The KSP API gives you syntactic information about the code, but working with this API isn't as convenient and comprehensive as using reflection.

The lack of reflection support means that KSP may not be the right tool if you need to rely heavily on code inspection, and that you should make the process of gathering information as easy as possible for the processor.

Designing the DSL and Writing the Annotations Module

What would an annotation-based DSL for defining the relationship between physical quantities look like? As shown before, the four operations of addition, subtraction, negation, and scalar multiplication can be directly implemented using generic functions. There are two types of operations that require code generation:

- The conversions from and to `Double` can be expressed by `@Conversion` annotations on the corresponding class.

- The multiplication operation involves three classes. The most natural implementation seems to be annotating the resulting product class and referencing the factor classes in the annotation. We call this annotation `@MultiplicationResult`. We can also implement divisions using the same information.

The resulting annotation-based DSL might look like this.

```
sealed interface Quantity<Q: Quantity<Q>>{
    val amount: Double
}

@Conversion("mm2", 0.000_001)
@Conversion("m2", 1.0)
@Conversion("km2", 1_000_000.0)
@MultiplicationResult(Meter::class, Meter::class)
data class SquareMeter(override val amount: Double) :
    Quantity<SquareMeter>
```

Listing 12-11 Usage Example for an Annotation-Based DSL

Here are the definitions of the required annotations.

```
@Repeatable
@Target(AnnotationTarget.CLASS)
@Retention(AnnotationRetention.RUNTIME)
annotation class Conversion(
    val derivedUnit: String,
    val factor: Double
)

@Repeatable
@Target(AnnotationTarget.CLASS)
@Retention(AnnotationRetention.RUNTIME)
annotation class MultiplicationResult(
    val factor1: KClass<*>,
    val factor2: KClass<*>
)
```

Listing 12-12 Annotation Definitions for the Physical Quantity DSL

Since the annotation processor needs to know about these annotations, they must be placed in a separate module. Placing them directly in the client code would create circular dependencies between the client module and the annotation processor, which would obviously not work.

Writing the Annotation Processor

This book can only provide a high-level overview of KSP. It explains the general structure of the KSP module, but not the specifics of the KSP API, which may change. For a more detailed discussion, please refer to the KSP documentation.

> **Caution** KSP runs in *incremental mode*[5] by default, which means that it tries to avoid unnecessary reprocessing of the sources. This is implemented by restricting the results of methods like `Resolver.getAllFiles()` and `Resolver.getSymbolsWithAnnotation()` to only the files that have changed.
>
> This setting is based on the assumption that the KSP code itself usually doesn't change during subsequent runs, which in turn means that it won't work correctly during the implementation of the KSP module itself. To disable incremental mode in the modules that use your KSP during development, set the Gradle property `ksp.incremental=false`, for example, in their `gradle.settings` files.
>
> There are other settings that affect which files are considered "dirty" (changed), such as `ksp.intermodule.change` and `aggregating`; see the documentation for more information.

A KSP implementation consists of three main parts:

- **A `SymbolProcessorProvider`:** This class allows the KSP library to use Java's Service Provider Interface[6] mechanism to discover new processors.
- **A `SymbolProcessor`:** This class is the starting point of the annotation processor. Typically, it determines what tasks need to be performed, for instance, by inspecting annotations.
- **`KSVisitor` Classes:** After determining *what* to do, the processor usually delegates the work to one or more visitor classes that know *how* to do a particular task. You don't have to follow this pattern, but it helps to clarify responsibilities and is the preferred approach according to the KSP documentation.

The provider class contains a `create()` method that can instantiate a particular processor.

```
class UnitsProcessorProvider : SymbolProcessorProvider {
    override fun create(
        environment: SymbolProcessorEnvironment
    ): SymbolProcessor = UnitsSymbolProcessor(
        codeGenerator = environment.codeGenerator
    )
}
```

Listing 12-13 The Symbol Processor Provider

In order to use the SPI mechanism mentioned above, the provider class must be registered in a text file called `com.google.devtools.ksp.processing.SymbolProcessorProvider` located

in the `resources/META-INF/services` folder. In this file, you simply add a single line with the qualified name of the processor-provider class.

The processor will find all classes annotated with `@Conversion` and `@MultiplicationResult`, and delegate code generation to the appropriate visitors.

```
class UnitsSymbolProcessor(
    private val codeGenerator: CodeGenerator,
) : SymbolProcessor {

    override fun process(resolver: Resolver): List<KSAnnotated>
{

        val conversionDeclarations = invokeVisitor(
            resolver,
            Conversion::class,
            ConversionVisitor(codeGenerator)
        )

        val multiplicationDeclarations = invokeVisitor(
            resolver,
            MultiplicationResult::class,
            MultiplicationVisitor(codeGenerator)
        )

        return (conversionDeclarations +
multiplicationDeclarations)
            .distinct()
    }

    private fun invokeVisitor(
        resolver: Resolver,
        annotation: KClass<*>,
        visitor: KSVisitorVoid
    ): List<KSClassDeclaration> = resolver
        .getSymbolsWithAnnotation(annotation.qualifiedName!!)
        .filterIsInstance<KSClassDeclaration>()
        .distinct()
        .partition { it.validate() }
        .let { (valid, invalid) ->
            valid.forEach { it.accept(visitor, Unit) }
            invalid
        }
}
```

Listing 12-14 The Symbol Processor Class

The first visitor provides conversion functions from and to `Double`.

```
class ConversionVisitor(
    private val codeGenerator: CodeGenerator,
) : KSVisitorVoid() {

    @OptIn(KspExperimental::class)
    override fun visitClassDeclaration(
        classDeclaration: KSClassDeclaration,
        data: Unit
    ) {
        val shortName =
classDeclaration.simpleName.getShortName()
        val typeName =
classDeclaration.asStarProjectedType().toTypeName()

        val annotations: List<Conversion> = classDeclaration
            .getAnnotationsByType(Conversion::class).toList()

        val fileSpec =
            FileSpec.builder(
                packageName =
classDeclaration.packageName.asString(),
                fileName = shortName.lowercase() + "Conversions"
            ).run {
                annotations.forEach { conversion ->
                    addConversions(
                        conversion.derivedUnit, typeName,
conversion.factor
                    )
                }
                build()
            }

        fileSpec.writeTo(
            codeGenerator,
            Dependencies(
                aggregating = false,
                classDeclaration.containingFile!!
            )
        )
    }

    private fun FileSpec.Builder.addConversions(
```

```
        unit: String,
        className: TypeName,
        factor: Double
    ) {
        addProperty(makeDoubleToQuantity(unit, className,
factor))
        addProperty(makeQuantityToDouble(unit, className,
factor))
    }

    private fun makeDoubleToQuantity(
        unit: String,
        className: TypeName,
        factor: Double
    ) = PropertySpec.builder(unit, className)
        .receiver(Double::class)
        .getter(
            FunSpec.getterBuilder()
                .addStatement(
                    "return %T(this * %L)",
                    className, factor
                )
                .build()
        )
        .build()

    private fun makeQuantityToDouble(
        unit: String,
        className: TypeName,
        factor: Double
    ) = PropertySpec.builder(unit, Double::class)
        .receiver(className)
        .getter(
            FunSpec.getterBuilder()
                .addStatement("return this.amount / %L", factor)
                .build()
        )
        .build()
}
```

Listing 12-15 The Visitor for Generating Conversions

The resulting methods are written in a separate file; for instance, the conversions for the quantity class `Meter` are located in `meterConversions.kt`. The next visitor generates the code to multiply and divide quantities.

```
class MultiplicationVisitor(
    private val codeGenerator: CodeGenerator
) : KSVisitorVoid() {

    override fun visitClassDeclaration(
        classDeclaration: KSClassDeclaration,
        data: Unit
    ) {
        val shortName =
classDeclaration.simpleName.getShortName()

        val factorPairs: List<Pair<ClassName, ClassName>> =
            classDeclaration
                .getAnnotations(MultiplicationResult::class)
                .map(KSAnnotation::arguments)
                .map { args ->
                    val factor1 = (args.first { arg ->
                        arg.name?.getShortName() == "factor1"
                    }.value as KSType).toClassName()
                    val factor2 = (args.first { arg ->
                        arg.name?.getShortName() == "factor2"
                    }.value as KSType).toClassName()
                    factor1 to factor2
                }

        val fileSpec =
            FileSpec.builder(
                packageName =
classDeclaration.packageName.asString(),
                fileName = shortName.lowercase() +
"Multiplications"
            ).run {
                factorPairs.forEach { (factor1, factor2) ->
                    addFunctions(
                        factor1 = factor1,
                        factor2 = factor2,
                        result = classDeclaration.toClassName()
                    )
                    if (factor1 != factor2) {
                        addFunctions(
                            factor1 = factor2,
                            factor2 = factor1,
                            result =
classDeclaration.toClassName()
```

```
                    )
                }
            }
            build()
        }

    fileSpec.writeTo(
        codeGenerator,
        Dependencies(
            aggregating = false,
            classDeclaration.containingFile!!
        )
    )
}

private fun FileSpec.Builder.addFunctions(
    factor1: ClassName,
    factor2: ClassName,
    result: ClassName
) {
    addFunction(makeMultiplication(factor1, factor2, result))
    addFunction(makeDivision(result, factor1, factor2))
}

private fun makeMultiplication(
    in1: ClassName,
    in2: ClassName,
    out: ClassName
) = FunSpec.builder("times")
    .addModifiers(KModifier.OPERATOR)
    .receiver(in1)
    .returns(out)
    .addParameter("that", in2)
    .addStatement(
        "return %T(this.amount * that.amount)", out
    )
    .build()

private fun makeDivision(
    in1: ClassName,
    in2: ClassName,
    out: ClassName
) = FunSpec.builder("div")
    .addModifiers(KModifier.OPERATOR)
```

```
            .receiver(in1)
            .returns(out)
            .addParameter("that", in2)
            .addStatement(
                "return %T(this.amount / that.amount)", out
            )
            .build()

    private fun KSClassDeclaration.getAnnotations(
        annotationClass: KClass<*>
    ): List<KSAnnotation> =
        annotations.filter {
            it.annotationType.resolve().declaration.qualifiedName
                ?.asString() == annotationClass.qualifiedName
        }.toList()
}
```

Listing 12-16 The Visitor for Generating Multiplicative Functions

As a small difficulty, we have to make sure that the operations with switched factors are added only if the factor classes are different.

If you read the code carefully, you will notice the rather complicated annotation handling code. Why can't we just get the annotation and read its arguments like we did in `ConversionVisitor`? This is because accessing arguments of type `KClass` or `Class` will result in a `ClassNotFoundException` if the corresponding classes are not yet known.

This is all we need to generate the necessary functions. To call our KSP module in client code, we need to add the KSP plugin and a dependency on our module to our Gradle build file.

```
plugins {
    ...
    id("com.google.devtools.ksp") version "2.3.6"
    ...
}
...
dependencies {
    ...
    ksp(/* reference to the KSP module */)
    ...
}
```

Listing 12-17 Referencing the KSP Module from Client Code

If the KSP module is in a Maven repository, it can be referenced with group, artifact, and version just like any other dependency. If it is part of the same multi-module project, it can

be referenced as `ksp(project(":moduleName"))` instead. After this change, the processor will be invoked for each build and generate the necessary files.

Case Study: Generating Data Class Patterns

In the last chapter, we developed a small pattern matching DSL, but it lacked the ability to create pattern functions for data classes. This is a perfect job for code generation. In this section, we will only discuss the KSP visitor class, as the remaining parts are very similar to the previous example.

```
@Target(AnnotationTarget.CLASS)
@Retention(AnnotationRetention.RUNTIME)
annotation class DataClassPattern

...

private const val patternPackage =
    "creativeDSLs.chapter_11.listing_11_to_19"

class PatternVisitor(
    private val codeGenerator: CodeGenerator,
    private val logger: KSPLogger
) : KSVisitorVoid() {

    private val patternClassName = ClassName(patternPackage,
"Pattern")

    override fun visitClassDeclaration(
        classDeclaration: KSClassDeclaration,
        data: Unit
    ) {
        val shortName =
classDeclaration.simpleName.getShortName()

        if (Modifier.DATA !in classDeclaration.modifiers) {
            logger.error(
                "Can't generate pattern, " +
                        "<$shortName> isn't a data class",
                classDeclaration
            )
            return
        }

        val parameters =
classDeclaration.primaryConstructor!!.parameters
```

```
        val funSpec = patternFunction(
            shortName, parameters,
            classDeclaration
        )

        val fileSpec = FileSpec.builder(
            packageName =
classDeclaration.packageName.asString(),
            fileName = shortName.decap() + "Pattern"
        ).addFunction(funSpec).addImport(patternPackage,
"any").build()

        fileSpec.writeTo(
            codeGenerator, Dependencies(
                aggregating = false,
                classDeclaration.containingFile!!
            )
        )
    }

    private fun patternFunction(
        shortName: String,
        parameters: List<KSValueParameter>,
        classDeclaration: KSClassDeclaration
    ) = FunSpec.builder(functionName(shortName))
        .addParameters(parameters.map { param ->
            ParameterSpec.builder(
                name = param.name!!.getShortName(),
                type = patternClassName
                    .parameterizedBy(param.type.toTypeName())
            )
                .defaultValue("any()")
                .build()
        })
        .returns(
            patternClassName.parameterizedBy(
                classDeclaration.toClassName().copy(nullable =
true)
            )
        )
        .beginControlFlow("return")
        .beginControlFlow("when(it)")
        .addCode("null -> false\n")
        .addCode(
```

```
            "else -> %L", parameters
                .joinToString(" &&\n        ", "", "\n") { param
->
                    val shortName = param.name!!.getShortName()
                    "$shortName(it.$shortName)"
                })
        .endControlFlow()
        .endControlFlow()
        .build()

    private fun String.decap(): String =
        this.replaceFirstChar { it.lowercase(Locale.ROOT) }

    private fun functionName(shortName: String) =
shortName.decap()
        .let { decap ->
            if (decap == shortName) "${decap}Pattern" else decap
        }
}
```

Listing 12-18 Annotation and Visitor for Data Classes in the Pattern Matching DSL

The `@DataClassPattern` annotation can be very simple; it's just a marker for the KSP module. The symbol processor collects all classes with this annotation and calls the visitor. The visitor first makes sure that the input is indeed a data class, then generates the pattern function, and finally writes this function to a file in the same package. Here is the sample output for a given data class.

```
enum class Continent {
    Europe, Africa, Asia, NorthAmerica,
    SouthAmerica, Australia, Antarctica
}

// given data class
@DataClassPattern
data class Country(
    val name: String,
    val capital: String,
    val continent: Continent,
    val millionPeople: Double
)

// function generated by PatternVisitor
public fun country(
  name: Pattern<String> = any(),
```

```
  capital: Pattern<String> = any(),
  continent: Pattern<Continent> = any(),
  millionPeople: Pattern<Double> = any(),
): Pattern<Country?> = {
  when(it) {
    null -> false
    else -> name(it.name) &&
            capital(it.capital) &&
            continent(it.continent) &&
            millionPeople(it.millionPeople)
  }
}
```

Listing 12-19 Example Output of the KSP Module

This function can be used in a `match` block as described in the last chapter. It should be mentioned that our annotation processor has room for improvements; for instance, it can't handle data classes with type parameters.

Summary

Using code generation requires careful consideration due to the effort involved and the limitations of annotation processors. However, KSP makes it possible to generate flexible, powerful, and well-integrated DSLs, which would otherwise be difficult or impossible to write. Libraries like KotlinPoet make generating the desired code quite intuitive.

Common Applications

- Data creation and initialization
- Data transformation
- Data validation
- Defining operations
- Execute actions
- Testing
- Reporting and analytics
- Simulation and modeling

Pros

- Automates the process of writing boilerplate code
- Very flexible and customizable
- Often the only practical way to handle a combinatorial explosion
- Intuitive libraries such as KotlinPoet are available

Cons

- Requires some up-front effort and setup

- KSP requires additional modules
- The KSP API is somewhat convoluted
- May cause longer build times
- Bug fixing can be challenging

What's Next?

One of the big selling points of Kotlin is its compatibility with Java. However, while calling Java from Kotlin usually "just works," the other way around can be more problematic at times. As DSLs often use Kotlin features in creative ways, calling a Kotlin DSL from Java can be even more challenging compared to "normal" code. Fortunately, there are ways to make Kotlin DSLs at least accessible from Java.

Footnotes

1 https://square.github.io/kotlinpoet

2 Using reflection requires adding `org.jetbrains.kotlin:kotlin-reflect` as a project dependency.

3 To use the library, you need to add a project dependency to `com.squareup:kotlinpoet`.

4 For a detailed overview, please refer to https://kotlinlang.org/docs/ksp-quickstart.html.

5 https://kotlinlang.org/docs/ksp-incremental.html

6 https://docs.oracle.com/javase/tutorial/sound/SPI-intro.html

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
D. Gronau, *Creative DSLs in Kotlin*
https://doi.org/10.1007/979-8-8688-2821-8_13

13. Java Interoperability

Daniel Gronau[1] ✉
(1) Sandersdorf, Sachsen-Anhalt, Germany

> *Translation is not the art of failure but the art of the possible.*
> —Benjamin Paloff[1]

Using a Kotlin DSL from Java can be challenging. While some Kotlin features, such as operator overloading, are simply unavailable, there are ways to make other features more accessible from Java and create a useful and convenient DSL in both languages. Fortunately, Kotlin has several built-in features that improve its interoperability with Java. These features are described in the section "Calling Kotlin from Java"[2] of the Kotlin documentation.

This chapter discusses Java interoperability from a DSL design perspective and presents relevant built-in features and useful techniques for solving DSL-specific problems. While most interoperability problems are easy to fix, the hard part is often figuring out what's going wrong and knowing which language features are available to you in these situations.

Note that retrofitting an existing DSL for Java interoperability is often more costly than designing it with interoperability in mind from the beginning. Writing tests in both Kotlin and Java can help you avoid problems later on.

Renaming Identifiers

When writing DSLs, it is common to use unusual names that may not be allowed in Java, either because they use forbidden characters or because they match a Java keyword. In this case, the `@JvmName` annotation can help. Sometimes, however, additional annotations may be needed. For example, `@JvmStatic` is used to expose object members as static class members in Java.

```
object NamingTest {
    @JvmStatic
    @JvmName("checkThisOut")
    fun `check this out`() = println("backtick
notation")

    @JvmStatic
    @JvmName("instanceOf")
    fun instanceof() = println("instanceof")
}
```

Listing 13-1 Renaming Identifiers on the JVM

Calling these members is now trivial.

```
// Java code
public class CallStrangeNames {
    public static void main(String[] args) {
        NamingTest.checkThisOut();
        NamingTest.instanceOf();
    }
}
```

Listing 13-2 Calling Renamed Kotlin Identifiers from Java

Value Classes and Mangling

In Kotlin, value classes are represented by their underlying type in JVM bytecode. However, this approach can lead to naming conflicts when multiple functions have the same name and value class parameters with the same underlying type. To avoid this problem, Kotlin uses a technique called "name mangling."

During compilation, the compiler renames the affected functions using a hashing algorithm that takes into account the package name, class name, and function name. This creates a unique name for each function, preventing naming conflicts in the bytecode.

Kotlin users don't need to be aware of this behavior, because calls to the mangled functions are automatically translated correctly. However, Java users calling a mangled function would have to use the mangled name. To avoid this problem, you can rename the function on the JVM with the `@JvmName` annotation.

```
@JvmInline
value class Kilometers(val value: Double)

@JvmInline
value class Miles(val value: Double)

@JvmName("displayKm")
fun display(x: Kilometers) { println("${x.value}
km") }

@JvmName("displayMiles")
fun display(x: Miles) { println("${x.value}
miles") }
```

Listing 13-3 Renaming Functions with Value Class Arguments

In Java, you can call the example methods by their JVM names and with `double` arguments. For example, if the declarations are defined in a file named `util.kt`, the calls would be `UtilKt.displayKm(23.0);` and `UtilKt.displayMiles(42.3);` respectively.

Alternatively, if you would rather work with a class that resembles the original value class on the Java side, you can use the experimental `@JvmExposeBoxed` annotation instead.

Package-Level Definitions of Functions and Variables

Since Java does not allow you to define functions and variables outside of classes, Kotlin defines them as static functions and variables in an artificial class. This class's default name is derived from the file name, including the `kt` extension, and it follows the upper-camel-case naming convention for classes. Consider the following example.

```
// File utils.kt

package ...

fun someFunction() {
    println(42)
}
```

Listing 13-4 Package-Level Functions in Kotlin

The function could be called from Java as `UtilsKt.someFunction()`. Often, you would prefer to use another class name for your DSL, for example, `Utils` instead of `UtilsKt`. This can be easily achieved by annotating the file with `@JvmName`.

```
// File utils.kt

@file:JvmName("Utils")

package ...

fun someFunction() {
    println(42)
```

```
}
```

Listing 13-5 Renaming File Reference for Package-Level Functions

It is even possible to map the contents of multiple Kotlin files to the same Java class. In this case, each file requires an additional `@file:JvmMultifileClass` annotation.

Generate Overloaded Methods

Although Java doesn't support default arguments, they can be "simulated" by having several overloaded functions. The Kotlin compiler can generate such overloaded functions upon request with the `@JvmOverloads` annotation. Consider the following Kotlin function.

```
@JvmOverloads
fun withOverloading(
    s: String = "one",
    i: Int = 42,
    d: Double,
    b: Boolean = false
) {
    println("$s $i $d $b")
}
```

Listing 13-6 Using Overloading for Functions with Default Values

Because of the annotation, you can now see four functions of this name in Java. When you call the different implementations, you get the following results.

```
// Java Code
withOverloading("two", 12, 17.0, true);  // "two
12 17 true"
withOverloading("two", 12, 17.0);        // "two
12 17 false"
```

```
withOverloading("two", 17.0);              // "two
42 17 false"
withOverloading(17.0);                     // "one
42 17 false"
```

Listing 13-7 Using Overloaded Methods in Java

You can either call the method with all the arguments, or you can skip the right-most arguments with default values. Of course, you must always specify argument `d`, which has no default value.

Note that you can also annotate constructors with `@JvmOverloads`.

Accessing Fields

In Kotlin, fields are exposed as properties. Even if it looks like you are accessing a field with a call such as `somePerson.name`, you are actually accessing a getter or setter of this property behind the scenes. You can use these getters and setters in Java as well. For example, you can write `somePerson.getName()`. However, if you want to allow direct field access, you need to use the `@JvmField` annotation.

```
data class Person(@JvmField val name: String,
@JvmField val age: Int)
```

Listing 13-8 Allowing Field Access in Java

Now you can call the fields directly, as in Kotlin.

Generics

A common problem is that due to its declaration-side variance, Kotlin often generates generic signatures with wildcards, such as `List<? extends String>`, on the JVM. These types can be difficult to use from the Java side and may prevent the use of certain Java libraries (such as Dagger[3]). They can even lead to cryptic compile-time errors.

The solution is to annotate the problematic type with `@JvmSuppressWildcards` so that you get a type signature such as `List<String>` on the JVM.

Sometimes, the opposite problem occurs, and it would be more convenient to have wildcards on the JVM, but the Kotlin compiler doesn't produce them. In this case, use the `@JvmWildcards` annotation.

Calling Functions with Reified Type Parameters

Unfortunately, I have some bad news for you. Since Java lacks an inlining mechanism, resolving reified type parameters is flat-out impossible. Therefore, you cannot call such functions from Java, not even via reflection. One possible solution is to write a version of the function that has an explicit class argument.

```
// DSL code using a reified generic parameter
inline fun <reified T> tellType(list: List<T>) {
    println("$list is a ${T::class.qualifiedName}
list")
}

fun <T: Any> tellTypeJava(list: List<T>, clazz:
Class<T>) {
    println("$list is a
${clazz.kotlin.qualifiedName} list")
}
```

Listing 13-9 Bridge Function for Reified Generics

You can call the bridge function as usual from Java, for example, `tellTypeJava(List.of(1,2,3), Integer.class);`. This approach works for many use cases. However, it should be noted that a reified type contains information about its own type parameters, whereas a class argument merely denotes a raw type. If this type information is needed, our simplistic approach won't work. A general

solution for more complicated cases is difficult to provide, but replacing the class parameter with a `TypeToken` (from either Guava[4] or Gson[5]) might help.

Checked Exceptions

Kotlin doesn't have the concept of *checked exceptions.* However, if a function that might throw such an exception is called from Java, the Java compiler expects the exception to be declared in the function signature. To avoid problems in these cases, you can use the `@Throws(SomeCheckedException::class)` annotation to tell the Kotlin compiler to add a checked exception to the function signature in the JVM.

Prevent Java Access

Using some parts of your DSL from Java can be cumbersome, unintuitive, or even lead to unsafe behavior. If you find yourself in a situation where certain parts of your DSL should only be accessible from Kotlin, you can use the `@JvmSynthetic` annotation on files, functions, fields, and property getters and setters.

For example, handling coroutine calls from Java is possible, but requires an understanding of the underlying architectural concepts, such as continuations. Providing dedicated functions for Java access is usually a better solution. For example, you can wrap the coroutine in a `CompletableFuture`, which is much easier to handle in Java.

```
// hidden from Java
@JvmSynthetic
suspend fun getStuff(): String {
    return "stuff"
}
```

```
private val scope =
CoroutineScope(EmptyCoroutineContext)

// dedicated Java API
fun getStuffForJava(): CompletableFuture<String> =
    scope.future { getStuff() }
```

Listing 13-10 Providing a Bridge Function for a Coroutine

The call from Java is now trivial; for instance, you can wait for the result using `getStuffForJava().join();`. However, the complicated and potentially confusing `getStuff()` function is no longer visible in Java due to the annotation.

Summary

Writing DSLs often requires the use of advanced language features. Therefore, it's no surprise that calling this code from Java can be challenging and that the Kotlin compiler may require guidance to ensure good Java interoperability. If calling your DSL from Java is a requirement, consider this from the beginning of your design process and write tests and documentation for Java as well.

What's Next?

Throughout the book, I aimed to move beyond typical toy examples of DSLs and address the weaknesses and potential improvements of the provided code. However, a book can only contain so much information, and it can never replace studying well-written, real-world DSLs, which showcase clever tricks, sensible compromises, and a consistent style. That's why the final chapter discusses influential, interesting, and educational examples of real-world DSLs.

Footnotes

1 Benjamin Paloff: *Forensic Translation*, The Nation, April 7, 2015.

2 https://kotlinlang.org/docs/java-to-kotlin-interop.html

3 https://dagger.dev

4 https://github.com/google/guava

5 https://github.com/google/gson

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
D. Gronau, *Creative DSLs in Kotlin*
https://doi.org/10.1007/979-8-8688-2821-8_14

14. Real-World DSL Examples

Daniel Gronau[1] ✉
(1) Sandersdorf, Sachsen-Anhalt, Germany

> *Few things are harder to put up with than the annoyance of a good example.*
> —Mark Twain[1]

In this chapter, we will examine some interesting and successful DSLs, characterize them, explore interesting details, and suggest potential improvements. Take a closer look at these projects to learn how they build coherent DSLs with seemingly “impossible” features.

KotlinPoet

KotlinPoet[2] is a source code generator for Kotlin, and a sister project of JavaPoet, which is written in Java. In Chapter 12, we used KotlinPoet to generate source code for operations on physical quantities, as well as an extension of our pattern matcher. The KotlinPoet library closely follows the builder pattern syntax of JavaPoet. The following is the introductory example from the KotlinPoet homepage.

```
// The file to be generated
class Greeter(val name: String) {
  fun greet() {
    println("""Hello, $name""")
```

```
  }
}

fun main(vararg args: String) {
  Greeter(args[0]).greet()
}

// The generation code
val greeterClass = ClassName("", "Greeter")
val file = FileSpec.builder("", "HelloWorld")
  .addType(
    TypeSpec.classBuilder("Greeter")
      .primaryConstructor(
        FunSpec.constructorBuilder()
          .addParameter("name", String::class)
          .build()
      )
      .addProperty(
        PropertySpec.builder("name",
String::class)
          .initializer("name")
          .build()
      )
      .addFunction(
        FunSpec.builder("greet")
          .addStatement("println(%P)", "Hello,
\$name")
          .build()
      )
      .build()
  )
  .addFunction(
    FunSpec.builder("main")
      .addParameter("args", String::class, VARARG)
```

```
        .addStatement("%T(args[0]).greet()",
greeterClass)
        .build()
    )
    .build()

file.writeTo(System.out)
```

Listing 14-1 Hello World Example in the KotlinPoet Documentation

You might think that using the loan pattern here would be more "natural" and convenient, and you are correct when the general shape of the code is mostly fixed. However, when dealing with dynamic tasks that require more flexibility, when parts of the syntax tree are passed around, and code gets generated in loops or using recursion, the builder approach works better because its simplicity makes it easily decomposable, and therefore more "automation-friendly."

Gradle

Gradle[3] is an amazing build system. Unlike descriptive approaches such as the XML-based Apache Maven, Gradle's build process is "programmable," which gives the user a lot of flexibility.

However, Gradle was originally implemented in Groovy, which was probably the best choice at the time for writing an expressive DSL in the Java ecosystem. Unfortunately, the Groovy language has its own set of problems. It is a scripted language with a weak type system. As a result, IDE features such as autocompletion, content assistance, source navigation, quick documentation, and refactoring support are limited.

To address these issues, the Gradle team created an alternative DSL based on Kotlin Script (`.kts`). The new DSL was designed to closely resemble the old Groovy style to ensure a seamless transition. In this regard, I think the Kotlin DSL was a resounding success. The following comparison is taken from the Gradle User Guide.[4]

```
// Groovy
```

```
plugins {
    id 'java'
}

repositories {
    mavenCentral()
}

dependencies {
    testImplementation "junit:junit:4.13"
    implementation project(':shared')
}

// Kotlin Script

plugins {
    id("java")
}

repositories {
    mavenCentral()
}

dependencies {
    testImplementation("junit:junit:4.13")
    implementation(project(":shared"))
}
```

Listing 14-2 Comparing Groovy and Kotlin DSLs for Gradle

At first glance, it's difficult to distinguish one from the other. While there are parts of the DSLs that differ more, it appears to be a deliberate effort to clean up and standardize the syntax rather than a limitation of the language.

In my opinion, Gradle is a good example of how Kotlin DSLs can modernize an already successful and established solution without

causing major disruptions.

Kotest

Kotest[5] is a widely used testing framework for Kotlin. It is interesting to compare Kotest with JUnit,[6] the most popular Java testing framework.

First of all, Kotest provides many testing styles, called "specs," and is much less opinionated than JUnit in this regard. For brevity's sake, I'll use `StringSpec` for the following examples. The documentation[7] gives this introductory example.

```
class MyTests : StringSpec({
    "String.length" - {
        "should return the length of the string" {
            "sammy".length shouldBe 5
            "".length shouldBe 0
        }
    }
    "containers can be nested as deep as you want"
- {
        "and so we nest another container" - {
            "yet another container" - {
                "finally a real test" {
                    1 + 1 shouldBe 2
                }
            }
        }
    }
})
```

Listing 14-3 StringSpec Style in Kotest

Unlike JUnit, the individual tests are not functions in the body of the test class, but expressions within the lambda argument of the

respective `...Spec` superclass. This design makes Kotest very flexible and allows dynamic test creation, for instance, in a loop.

```
class LogicTest : StringSpec({

    val xorTable = listOf(
        Triple(true, true, false),
        Triple(true, false, true),
        Triple(false, true, true),
        Triple(false, false, false)
    )

    for((x, y, z) in xorTable) {
       "'$x' xor '$y' should be '$z'" {
           x xor y shouldBe z
       }
    }
})

// Runs 4 tests successfully:
// 'true' xor 'true' should be 'false'
// 'true' xor 'false' should be 'true'
// 'false' xor 'true' should be 'true'
// 'false' xor 'false' should be 'false'
```

Listing 14-4 Dynamic Test Creation in Kotest

Of course, you can achieve a similar result in JUnit by using the `@ParameterizedTest` annotation instead of `@Test`. If you need even more flexibility, there is a `DynamicTest.dynamicTest()` method that accepts a lambda argument. However, Kotest doesn't require special constructs in such cases; it allows users to take advantage of existing language features. Doing more with less reduces the cognitive load when learning to use the framework. Further, this approach avoids situations in which the framework behaves differently when using "special" features. In JUnit, for example, the rules for

fixtures in dynamic tests differ from those in normal tests. However, in Kotest, there is no such difference.

Kotest can be categorized as a quasi-lingual DSL, where many parts mimic natural language.

```
val exception =
shouldThrow<IllegalAccessException> {
   // code in here that you expect to throw an
IllegalAccessException
}
exception.message should startWith("Something went
wrong")
```

Listing 14-5 Kotest As Quasi-Lingual DSL[8]

While this is already fairly human-readable, I think the next library goes even further in this direction.

MockK

Mocking, spying, stubbing, and argument capture are essential for writing concise and expressive unit tests. While libraries like Mockito[9] are very popular in Java, they face some challenges when dealing with Kotlin language features such as objects, top-level functions, and coroutines.

MockK[10] is an amazing mocking framework for Kotlin, and features a very intuitive hybrid DSL. The following is a simple example from the documentation.

```
val car = mockk<Car>()

every { car.drive(Direction.NORTH) } returns
Outcome.OK

car.drive(Direction.NORTH) // returns OK
```

```
verify { car.drive(Direction.NORTH) }

confirmVerified(car)
```

Listing 14-6 MockK As Quasi-Lingual DSL

The library has already been mentioned in Chapter 11 as an example of a quasi-lingual DSL, and blends well with quasi-lingual testing frameworks such as Kotest. If you write DSLs that are intended for use alongside or within the context of other DSLs, it is always a good idea to find a matching style. MockK illustrates this principle very well.

Better-Parse

Better-parse is the parser-combinator library that we used in Chapter 9. It features a succinct hybrid DSL, which gives you a lot of flexibility. Below is the implementation of one of the example parsers.[11]

```
sealed class BooleanExpression

object TRUE : BooleanExpression()
object FALSE : BooleanExpression()
data class Variable(val name: String) :
BooleanExpression()
data class Not(val body: BooleanExpression) :
BooleanExpression()
data class And(val left: BooleanExpression, val
right: BooleanExpression) : BooleanExpression()
data class Or(val left: BooleanExpression, val
right: BooleanExpression) : BooleanExpression()
data class Impl(val left: BooleanExpression, val
right: BooleanExpression) : BooleanExpression()

object BooleanGrammar : Grammar<BooleanExpression>
() {
    val tru by literalToken("true")
```

```
    val fal by literalToken("false")
    val id by regexToken("\\w+")
    val lpar by literalToken("(")
    val rpar by literalToken(")")
    val not by literalToken("!")
    val and by literalToken("&")
    val or by literalToken("|")
    val impl by literalToken("->")
    val ws by regexToken("\\s+", ignore = true)

    val negation by -not * parser(this::term) map
{ Not(it) }
    val bracedExpression by -lpar *
parser(this::implChain) * -rpar

    val term: Parser<BooleanExpression> by
        (tru asJust TRUE) or
        (fal asJust FALSE) or
        (id map { Variable(it.text) }) or
        negation or
        bracedExpression

    val andChain by leftAssociative(term, and) {
a, _, b ->
        And(a, b)
    }
    val orChain by leftAssociative(andChain, or) {
a, _, b ->
        Or(a, b)
    }
    val implChain by rightAssociative(orChain,
impl) { a, _, b ->
        Impl(a, b)
    }
```

```
    override val rootParser by implChain
}

fun main(args: Array<String>) {
    val expr = "a & (b1 -> c1) | a1 & !b | !(a1 ->
a2) -> a"
    println(BooleanGrammar.parseToEnd(expr))
}
```

Listing 14-7 Better-Parse Example

The grammar is defined within an object as a sequence of token definitions and parsers that build upon each other. One interesting aspect of this DSL is that it relies heavily on property delegation. This allows the tokens and parsers to be stored in lists behind the scenes, which makes it much easier for the library to work with them.

One interesting problem is how to handle recursive parser definitions. In the "chemical equation" example in Chapter 9, a *molecular part* can be an *element* or a *group*. However, a group consists of parts itself. The library elegantly solves this problem by enabling parsers to be referred to via their property reference.

```
val equationGrammar = object : Grammar<Equation>()
{
    ...
    val element: Parser<Element> by (symbol and
optional(number))
        .map { (s, n) -> Element(s.text, n ?: 1) }

    val group: Parser<Group> by (skip(leftPar) and
            oneOrMore(parser(this::part)) and
            skip(rightPar) and
            optional(number))
        .map { (parts, n) -> Group(parts, n ?: 1)
}
```

```
    val part: Parser<Part> = element or group
    ...
}
```

Listing 14-8 Recursive Parser Definition in Better-Parse

Calling `parser(this::part)` allows you to "break the cycle." Besides performance considerations, I suspect that this is one reason why the DSL uses an object as its context, rather than the more common trailing lambda syntax. While there are ways to create a recursive definition in loan pattern DSLs, this solution is clearly more straightforward. It's a good example of why creativity and out-of-the-box thinking are important when writing DSLs.

Konform-kt

Konform-kt[12] is a data validation library. The resulting validation object contains a list of issues with the given data structure, allowing each problem to be precisely located. The DSL heavily relies on the loan pattern, allows for different types of nesting to "drill down" into the data structure, and provides many predefined constraints. The following example is taken from the documentation.

```
data class Person(val name: String, val email:
String?, val age: Int)

data class Event(
    val organizer: Person,
    val attendees: List<Person>,
    val ticketPrices: Map<String, Double?>
)

val validateEvent = Validation<Event> {
    Event::organizer {
        // even though the email is nullable
```

```
        // you can force it to be set in the
validation
        Person::email required {
            // Optionally set a hint, default hint
is "is required"
            hint = "Email address must be given"
            // Optionally set a user context,
default is `null`
            userContext = Severity.ERROR
            pattern(".+@bigcorp.com") hint
                "Organizers must have a BigCorp
email address"
        }
    }

    // validation on the attendees list
    Event::attendees {
        maxItems(100)
    }

    // validation on individual attendees
    Event::attendees onEach {
        Person::name {
            minLength(2)
        }
        Person::age {
            minimum(18) hint "Attendees must be 18
years or older"
        }
        // Email is optional but if it is set it
must be valid
        Person::email ifPresent {
            pattern(".+@.+\..+") hint
```

```
                "Please provide a valid email
address (optional)"
        }
    }

    // validation on the ticketPrices Map as a
whole
    Event::ticketPrices {
        minItems(1) hint "Provide at least one
ticket price"
    }

    // validations for the individual entries
    Event::ticketPrices onEach {
        // Tickets may be free in which case they
are null
        Entry<String, Double?>::value ifPresent {
            minimum(0.01)
        }
    }
}
```

Listing 14-9 Data Validation with Konform-kt

Even if you are not familiar with the library, you can probably tell which conditions will be checked and which output to expect. The DSL has a very consistent structure and descriptive naming conventions. If a condition is not met, you can provide your own using the `hint` infix function. However, you don't have to, because there are already sensible default messages available. Although Konform-kt doesn't use any fancy features that we haven't already covered, there is much to learn from how the DSL presents itself to the user and what makes user-friendly. I think the library is a great example of following the *Principle of Least Surprise*.

Arrow

Arrow[13] is a vast collection of features that promote functional programming in Kotlin. These features include working with coroutines, error handling, immutable types, and much more. There is far too much content to cover here, but here is a very basic example of how the library improves working with nullable types. Very often, you see Kotlin code like this, which is difficult to read due to numerous null checks.[14]

```
@JvmInline value class UserId(val value: Int)
data class User(val id: UserId, val email: Email?)

fun QueryParameters.userId(): UserId? =
get("userId")
    ?.toIntOrNull()
    ?.let { UserId(it) }
fun findUserById(id: UserId): User? = TODO()
fun sendEmail(email: Email): SendResult? = TODO()

fun sendEmail(params: QueryParameters):
SendResult? =
  params.userId()?.let { userId ->
    findUserById(userId)?.email?.let { email ->
      sendEmail(email)
    }
  }
```

Listing 14-10 Code with Nullability Handling

Arrow provides the `nullable` DSL, which avoids nesting, and simplifies null checking.

```
fun sendEmail(params: QueryParameters):
SendResult? = nullable {
  val userId = ensureNotNull(params.userId())
```

```
  val user = findUserById(userId).bind()
  val email = user.email.bind()
  sendEmail(email)
}
```

Listing 14-11 Nullability Handling with Arrow

I think you'll agree that this code is much easier to read. The concept of "flattening" monadic computations originates from Haskell, where it is known as *do notation*, and can also be found in Scala as *comprehensions*. It's great that Arrow brings a similar feature to Kotlin – not only for nullable types but also for `Either`, `Result`, and `Option`. But there's much more to explore in Arrow, both for your daily coding and for inspiration when writing idiomatic DSLs.

Summary

Exploring well-written DSLs can be a great source of inspiration. Frankly, a significant number of Kotlin DSLs currently available "get the job done," but they lack an engaging and enjoyable user experience. These DSLs often rely on a limited set of language features, missing the opportunity for a more exploratory approach. This results in a dull, uninspired feel. The examples in this chapter should inspire you to create DSLs that are creative, intuitive, and fun to use.

What's Next?

This is the last chapter of *Creative DSLs in Kotlin.* Now, it's up to you to write your own ones. You have explored the many ways in which Kotlin's design makes it an excellent choice for building DSLs. You have learned about the different types of DSLs and how Kotlin's language features can be used to create expressive and intuitive APIs. I hope this book has provided you with the knowledge and inspiration necessary to create your own DSLs in Kotlin. I also hope you continue to explore the many possibilities offered by this powerful, expressive language.

Of course, there are many skills and insights that can't be taught in a book, but can only be learned through experience and dedication. Creativity, personal style, and the ability to make good design decisions are critical to the success of any software project. Developing these skills requires persistence and a willingness to learn and grow. However, mastering these skills is very rewarding and makes writing software much more enjoyable.

I sincerely hope this book conveys some of the excitement I had while writing DSLs and researching the topic. I hope you experience the same sense of joy and wonder on your own path. Thank you for joining me on this journey!

Footnotes

1 Mark Twain. *The Tragedy of Pudd'nhead Wilson*. Charles L. Webster & Co., 1894.

2 https://square.github.io/kotlinpoet

3 https://gradle.org

4 https://docs.gradle.org/current/userguide/writing_build_scripts_intermediate.html#writing_build_scripts

5 https://kotest.io

6 https://junit.org

7 https://kotest.io/docs/framework/testing-styles.html#string-spec

8 https://kotest.io/docs/framework/exceptions.html

9 https://site.mockito.org

10 https://mockk.io

11 https://github.com/h0tk3y/better-parse/blob/master/demo/demo-jvm/src/main/kotlin/com/example/BooleanExpression.kt

12 https://github.com/konform-kt/konform

13 https://arrow-kt.io

14 https://arrow-kt.io/learn/typed-errors/wrappers/nullable-and-option/#option--nullable-dsl

Index

A

AgeTransformer 153
Algebraic DSLs
  BigInteger and BigDecimal 63
  complex numbers 64–67
  modular arithmetic calculations 67–72
  non-commutative multiplication 72
  operations 63
  structures 63
also() method 76
Annotations 54
  approaches 151
  code generation 151
  dependency injection frameworks 151
  mapper DSL 151, 152, 154–156
  processor 192–196, 198–200
  string-based DSLs 156, 157
apply() method 76
Arithmetic assignment operators 44
Arrow 226, 227
Assembling function 142

B

Backtick identifiers 30, 31
Better-parse 221–224
Binary arithmetic operators 43, 44
Boolean test function 171
Brainstorming mode 19
Builder pattern
  classical builder 75–78
  counting 89–92
  definition 75

multiple stages 92
named arguments 78, 79
nesting 79–85
SQL queries
chameleon classes approach 114–116
examples 109
overview 108
phantom type approach 117–120
separate classes approach 109–113
type inference 104
type-safe 85–88
build() method 97, 100, 112

C

cast() function 117
Chameleon classes 107
builder pattern 114–116
loan pattern 125–127
Checked exceptions 213
Chemical equations
assembling 165–167
challenging 136
chemical bonds 163
coefficients 163, 164
elements 162
examples 167
expressing 136, 137, 160, 161
extension functions for subscripts 162
grouping molecules 164
model classes 138, 139, 160
notation 136
operator overloading and infix notation 168
parser library 146–148
reversible and irreversible equations 165
structure diagram 162

subscripts 163, 167
writing parsers 139–146
Classical builder 75–78
ClassNotFoundException 199
Cognitive load theory 19, 20
Combinatory parsing 142
Complex numbers 64–67
Complex.ofCartesian() 64, 65
Complex.ofPolar() 64
Conciseness 7
Consistency 7
Concise syntax 29
Context parameters 55–57, 70
Context receivers 55
Contracts 58, 59
copy() method 32, 33, 79, 88
Counting builder 75, 89–92
Coverage 7
Curiously recurring template pattern (CRTP) 182

D

Data class patterns 200–204
Data transfer object (DTO) 109, 110
Declaration-site variance 119
Default values 31–33
Dependency injection 151
Dijkstra's Shunting Yard Algorithm 84
Documentation 21, 22
concepts 26
sample project 26, 27
Domain-specific languages (DSLs)
closed/open source 14
code generation 4
definition 3
design principles 7, 8

internal and external 4
Kotlin 8
performance and memory requirements 13, 14
problem domain 9, 10
requirements 15
research 10
safety usage 12
specifying output 11
syntactic gap 11
use cases
implementation support 5
peripheral systems 6
runtime behavior 6
special applications 6
system management 5
Do notation 227
@DslMarker annotation 40, 41
@DslMarker mechanism 98, 121
DSL-specific challenges
code conventions 24
coupling 23, 24
disorganized output 24
name collisions 23
testing 25, 26
DynamicTest.dynamicTest() method 220

E

Embedded DSLs 4
Equality and comparison operators 45
Experimental features
collection literals 58
contracts 58, 59
Extensibility 7, 13
External DSLs 4
Extraneous cognitive load 19

F

findElement() function 142, 143
Fluent interfaces 75
fold() method 33
Formalization 21, 22
Forsyth–Edwards notation (FEN) 132–136
FromBody 123
FromClause 111, 122
Functional interfaces 47, 48
Functional programming 30

G

Generics 48, 49, 212
getStuff() function 214
Getter method 79
Gradle 217, 218
Graphical DSLs 131
GroupByClause 112, 113

H

Hashing algorithm 209
headers() method 98
Helper function 69
HttpRequest 96–101
httpRequest() method 98
HttpRequest with AutoDSL 101–104
Hybrid DSLs
  chemical equations 160–168
  defined 159
  pattern matching 168–175
  quasi-lingual 159, 160

I

Ideal syntax 18, 19
IllegalStateException 104

Index access and index operators 45
IndexOutOfBoundException 143
Infix notation 46, 47
IntelliJ IDEA 8
Internal DSLs 4, 11, 13
Interoperability 7
into() method 56
Intrinsic cognitive load 19
invoke() operator 87

J

Java interoperability 14, 72
  accessing fields 211
  calling functions with reified type parameters 212, 213
  checked exceptions 213
  DSL design perspective 207
  generate overloaded methods 210, 211
  generics 212
  package-level definitions of functions and variables 209, 210
  preventing Java access 213, 214
  renaming identifiers 208
  value classes and mangling 208, 209
JoinClause 112
@JvmName annotation 165, 208, 209
@JvmOverloads annotation 210
@JvmSynthetic annotation 213
@JvmWildcards annotation 212

K

kapt 189
Konform-kt 224–226
Kotest 218–220
Kotlin 8, 12
  backtick identifiers 30, 31
  extensions and receivers

@DslMarker annotation 40, 41
loan pattern 39, 40
properties 41, 42
static methods 37
target class/object 37
type narrowing 38
user's perspective 37
features 29, 30
named arguments and default values 31–33
operator overloading 42–46
property syntax 36, 37
trailing lambda arguments 33, 34
varargs 34–36
KotlinPoet 215–217
KotlinPoet library 79, 80, 185
Kotlin symbol processing (KSP)
annotation processor 192–196, 198–200
annotation-based DSL 190, 191
definition 189
incremental mode 192
limitations 189
modules and client code 189, 190
Kotlin symbol processing API (KSP) 54
KSVisitor classes 193

L

Lambda arguments 33, 34
Lambdas with receivers 95
Learning experience 12
Literals 58
Loan pattern 39, 40
Loan pattern DSLs
definition 95
flexibility 95
HttpRequest 96–101

HttpRequest with AutoDSL 101–104
Loan pattern, SQL queries
chameleon classes approach 125–127
examples 120, 121
infix functions 121
phantom types approach 127, 129
separate classes approach 121–125

M

Maintainability 7, 13
map() method 33
Mapper DSL 151, 152, 154–156
MapStruct library 151
match() function 171
Miniature DSLs 8
MockK 221
MockK library 159
Modularity 7
Multiplatform support 30

N

Name mangling 208
Named arguments 31–33
Nesting builders
build() 80
components 79
flattening 80–83
varargs 83–85
“Not Invented Here” syndrome 10
Null-safe type system 29

O

Object-oriented design 29
Operator overloading 207
binary arithmetic operators 43, 44

equality and comparison operators 45
index access and index operators 45
overload responsibly 45, 46
range and in operators 44
return types 42
unary operators 43
@OptIn annotation 57
Orthogonality 22
Overloaded functions 210
Overload responsibly 45, 46

P

@ParameterizedTest annotation 220
parsePart() method 144
parsePattern() function 142
Parser-combinator approach 146
Pattern matching
achievable syntax 170
capture patterns 174
comparison patterns 173
compromises 169, 170
data class 175
definition 168
functionality 171
ideal syntax 169
infix functions 172
iterable patterns 173
simple patterns 172, 173
use cases 171
Peano representation 90, 91
Phantom types 49, 107
builder pattern 117–120
loan pattern 127, 129
Physical quantities
conversion factors 183, 184

defined 181
example 189
extension functions 184
function generation process 188
generated code 188
interface 183
KotlinPoet library 185
multiplication and division 186, 187
multiplication equations 186
polynomial/exponential growth 186
type-safe functions 182
unit conversions 184, 185
units 181, 182
Portable game notation (PGN) 132
Property syntax 36, 37
Prototyping 20, 21
PyCharm 8

Q

Quasi-lingual DSLs 159, 160

R

Range and in operators 44
Reflection 54
Regular expressions 131
Reified generics 50, 51
Residues 68
returnNotNull() function 59
Reverse polish notation (RPN) 83
Round-trip tests 134

S

Scope control mechanism 40
Scope functions 37
SelectClause 122

Separate classes 107
builder pattern 109–113
loan pattern 121–125
Setter method 79
Single abstract method (SAM) 48
Specs style 218
Spring Security 151
State transitions
approaches 107
builder pattern 108–120
loan pattern 120–129
String-based DSLs 156, 157
String-parsing DSLs
chemical equations 136–148
drawback 131
FEN 132–136
symbols 131
SymbolProcessor 192
SymbolProcessorProvider 192
Syntactic gap 11

T

toFEN() function 136
toString() method 39, 50, 138
Type aliases 52, 53
Type erasure conflicts 49, 50
Type-level programming 49, 85, 89
Type narrowing 88
Type-safe builder pattern 75
attributes 85
build() function 88
generics 85
helper classes 86
product 87
signature 86, 87

subclasses 86

U

Unary operators 43
Usability 7
User customization 13
UtilsKt.someFunction() 210

V

Value classes 51, 52, 208
Variable arguments (varargs)
defined 34
nesting 83–85
position and trailing lambda syntax 35, 36
spread operator * 34

W

WebStorm 8
WhereClause 112
Wither 79
with() function 56
writeGeneratedFile() method 189
Writing DSLs
cognitive load 19, 20
formalization and documentation 21, 22
ideal syntax 18, 19
implementation 17, 22, 23
prototyping 20, 21
trial and error 18

X, Y

-Xenable-builder-inference compiler 104

Z

ZonedDateTime 153

Table 4-2 Binary Arithmetic Operators

Operator	Overwriting Function	Remarks
a + b	`fun A.plus(b: B): R`	
a - b	`fun A.minus(b: B): R`	
a * b	`fun A.times(b: B): R`	
a / b	`fun A.div(b: B): R`	
a % b	`fun A.rem(b: B): R`	Until Kotlin 1.1, `mod` was used instead, which is now deprecated

Table 4-5 Index Access and Index Operators

Operator	Overwriting Function	Remarks
a[b]	`fun A.get(b: B): R`	
a[b, c]	`fun A.get(b: B, c: C): R`	Or more arguments
a[b] = x	`fun A.set(b: B, x: X): Unit`	
a[b, c] = x	`fun A.set(b: B, c: C, x: X): Unit`	Or more arguments
a()	`fun A.invoke(b: B): R`	
a(b)	`fun A.invoke(b: B): R`	
a(b, c)	`fun A.invoke(b: B, c: C): R`	Or more arguments

Table 4-3 Arithmetic Assignment Operators

Operator	Overwriting Function
a += b	`fun A.plusAssign(b: B): Unit`
a -= b	`fun A.minusAssign(b: B): Unit`
a *= b	`fun A.timesAssign(b: B): Unit`
a /= b	`fun A.divAssign(b: B): Unit`
a %= b	`fun A.remAssign(b: B): Unit`

Table 4-4 Range and In Operators

Operator	Overwriting Function	Remarks
`a .. b`	`fun A.rangeTo(b: B): R`	
`a ..< b`	`fun A.rangeUntil(b: B): R`	Introduced in Kotlin 1.8, experimental in 1.7.20
`a in b`	`fun B.contains(a: A): Boolean`	Also defines `!in`.

Table 4-1 Unary Operators

Operator	Overwriting Function	Remarks
+a	`fun A.unaryPlus(): R`	
-a	`fun A.unaryMinus(): R`	
!a	`fun A.not(): R`	
++a	`fun A.inc(): A`	Assigns the result to `a` and returns it
a++	`fun A.inc(): A`	Assigns the result to `a`, returns the original value
--a	`fun A.dec(): A`	Assigns the result to `a` and returns it
a--	`fun A.dec(): A`	Assigns the result to `a`, returns the original value

Table 4-6 Equality and Comparison Operators

Operator	Overwriting Function	Remarks
`a == b`	`fun equals(b: Any): Boolean`	Must be defined in `class A`. Also defines `!=`
`a < b`	`fun A.compareTo(b: B): Int`	Evaluates `a.compareTo(b) < 0`
`a <= b`	`fun A.compareTo(b: B): Int`	Evaluates `a.compareTo(b) <= 0`
`a > b`	`fun A.compareTo(b: B): Int`	Evaluates `a.compareTo(b) > 0`
`a >= b`	`fun A.compareTo(b: B): Int`	Evaluates `a.compareTo(b) >= 0`