Creative DSLs in Kotlin

Daniel Gronau

Creative DSLs in Kotlin

Daniel Gronau

Bibliographic Information of the German National Library:
The Deutsche Nationalbibliothek records this publication in the German National Bibliography;
Detailed bibliographical data are available on the Internet at http://dnb.dnb.de.

Feedback: creativeDsls@proton.me

Production and publisher:
BoD - Books on Demand, Norderstedt

ISBN: 9783759759863

Table of Contents

Dedication

— To my parents Eva and Wolfgang Gronau —

Preface

> A novice was trying to fix a broken Lisp machine by turning the power off and on.
> Knight, seeing what the student was doing, spoke sternly: "You cannot fix a machine by just power-cycling it with no understanding of what is going wrong."
> Knight turned the machine off and on.
> The machine worked.
>
> — an AI Koan

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 I find that many of these 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 tradeoffs 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 resulting 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 tradeoffs 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 those who are new to Kotlin or Java. While this book includes a chapter discussing Kotlin language features and their potential use in DSLs, it is not meant to be a comprehensive guide to learning 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 diving into 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 DSL development.

What's in this Book

Chapter 1, Introduction, defines what a DSL is, discusses the differences between internal and external DSLs, and provides a general overview of DSL usage and underlying design principles.

Chapter 2, Requirements Analysis, helps you define the requirements and expectations for a DSL design before implementing it. Jumping in without defining your goals can lead to a nasty surprise, as there are more things to consider than you might think.

Chapter 3, Writing a DSL, suggests a simple methodology for implementing a DSL. Its steps help you avoid tunnel vision, get out when you hit a wall, and iterate your way to success.

Chapter 4, Relevant Language Features, takes a closer look at the Kotlin language from the perspective of a DSL designer. Even if you are very familiar with Kotlin, you may have never used some of the more obscure language features, such as extension properties.

Chapter 5, Algebraic DSLs, introduces the first category of DSLs, which support number-like behavior using operator overloading and infix functions. This is a fairly straightforward type of DSL, but mastering it will be useful in many other contexts.

Chapter 6, Builder Pattern DSLs, covers the well-known builder pattern that you probably already know from Java, and shows some variations that attempt to make builders safer to use.

Chapter 7, Loan Pattern DSLs, is about using extension functions and the Loan Pattern to solve similar problems as the Builder Pattern DSL, but in a more idiomatic and convenient way.

Chapter 8, Modeling State Transitions in DSLs, presents various methods for modeling the stages or states of elements in your DSL. This is particularly useful when constructing objects in stages or modeling a finite state machine. The chapter explores alternatives to the simplistic "one state, one class" approach.

Chapter 9, String-Parsing DSLs, gives an overview of DSLs embedded in strings, which can be considered an intermediate case between internal and external DSLs. The chapter, for educational purposes, presents a manually written parser as well as demonstrating how the same parser can be written using a parser-combinator library.

Chapter 10, Annotation-based DSLs, explores the application of annotations for a DSL, which adds new meaning or behavior to existing code. The provided example code also illustrates how to use reflection to bring annotations to life.

Chapter 11, Hybrid DSLs, explains how to address scenarios where one of the previously discussed DSL categories is insufficient on its own and must be combined. Combining language features presents its own set of obstacles, and this chapter will equip you to overcome them.

Chapter 12, Code Generation for DSLs, discusses the benefits of code generation when manual DSL writing involves excessive boilerplate. At first, this task may seem daunting, but the example demonstrates that it is not something to be afraid of.

Chapter 13, Java interoperability, provides an overview of potential issues when utilizing Kotlin DSLs within Java code, and offers practical solutions.

Chapter 14, Real-World DSL Examples, explores established and tested DSLs for valuable insight. In my view, small-scale examples do not provide sufficient guidance for quality DSL design. It is crucial to observe how DSLs tackle real-world problems, make compromises, and still provide a superior user experience.

Prerequisites

To try out the code examples in this book or in the accompanying project at github.com/creativeDsls[1], you will need an integrated development environment (IDE) that can run Kotlin. I recommend using JetBrains' IntelliJ IDEA, which has a free community edition that is sufficient for this purpose. Alternatively, you can also use the online Kotlin sandbox play.kotlinlang.org for smaller examples. This book uses the Kotlin version 2.0.0 and assumes Java language level 11 unless otherwise noted.

It is also recommended that you familiarize yourself with the Kotlin Documentation[2], which is a valuable resource for learning about the language and its features. This book will refer to the Kotlin documentation for specific or non-DSL-related topics, rather than repeating information that can already be found there.

Typographical Conventions

This book uses a few typographical conventions to structure its content.

Quotes look like this:

> To be, or not to be: that is the question
>
> — Shakespeare, Hamlet, Act 3 Scene 1

Inlined code snippets are shown like this: `val answer = 42`

Tips, warnings etc. are presented as follows:

When the white frost comes, do not eat the yellow snow.

Jabberwocky

Beware the Jabberwock, my son! The jaws that bite, the claws that catch! Beware the Jubjub bird, and shun the frumious Bandersnatch!

Example code is shown like this, and may include a file name or other indication of origin before the code block:

src/main/kotlin/personDemo/Person.kt

```
import java.time.ZonedDateTime
import java.time.temporal.ChronoUnit.YEARS

data class Person(val firstName: String, val lastName: String, val age: Int) {
   constructor(firstName: String, lastName: String, dateOfBirth: ZonedDateTime):
       this(firstName, lastName, YEARS.between(dateOfBirth,
ZonedDateTime.now()).toInt())
}
```

Definitions or additional information may be presented as follows:

The name "Kotlin"

"Kotlin" is a small Russian island in the Baltic Sea. Naming languages or projects after islands has been a long tradition in the Java ecosystem. Besides Java itself, there are projects like Lombok, the Komodo IDE and the Ceylon language. The Jakarta project is named after the capital of Indonesia, which is located on the island of Java.

Tools used for writing this book

The book is written in the AsciiDoc[3] format. For PDF and eBook generation, I used the AsciidocFX[4] editor. The main writing and programming tool was IntelliJ IDEA[5] by

JetBrains, using the Asciidoctor plugin[6]. The diagrams were prototyped using the ditaa[7] library and then finalized in drawio[8].

I used DeepL Write[9] by DeepL SE and ChatGPT[10] by OpenAI as a writing assistants. As a non-native speaker, it can be challenging to avoid grammatical errors and achieve a natural writing style. Therefore, I'm thankful to DeepL and OpenAI for providing public access to their remarkable language processing technologies.

Feedback

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. Thank you in advance for taking the time to share your thoughts with me.

To give feedback, e-mail me at creativeDSLs@proton.me

[1] creativeDsls: https://github.com/creativeDsls

[2] Kotlin Documentation: https://kotlinlang.org/docs/home.html

[3] AsciiDoc: https://asciidoc.org

[4] AsciidocFx: https://asciidocfx.com

[5] IntelliJ IDEA: https://www.jetbrains.com/idea

[6] Asciidoctor Plugin: https://plugins.jetbrains.com/plugin/7391-asciidoc

[7] ditaa: https://ditaa.sourceforge.net

[8] drawio: https://www.drawio.com/

[9] DeepL Write: https://www.deepl.com/write

[10] ChatGPT: https://openai.com/blog/chatgpt/

Part I - Writing DSLs

Chapter 1. Introduction

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

1.1. 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 for writing a DSL is to make a particular domain more accessible, easier to read and write, to avoid errors, and sometimes to 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.

Domain-specific languages are often used to model business logic in specific domains, such as financial 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. They are also often used for testing and logging, code generation, and even for writing DSLs themselves. 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.

1.2. Internal and External DSLs

This book discusses DSLs that are embedded within Kotlin, and are therefore limited to the language's existing expressions. 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 allows 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 recently become much easier with new libraries, frameworks, and improved tools.

An edge case are internal DSLs that are realized entirely inside strings, the way e.g. 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.

1.3. 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 for representing physical quantities, there may be a large number of possible results when multiplying or dividing different 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 possible combinations. 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[1].

1.4. 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:** 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.

1.5. 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, but also 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 DSL code 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 and strikes a balance that meets the needs of the domain and the users.

1.6. 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.

[1] AutoDSL: https://github.com/F43nd1r/autodsl

Chapter 2. Requirements Analysis

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

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.

2.1. 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 lean. 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.

2.2. 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.

There is an ongoing debate about how much software should depend on libraries, and you should be aware of related issues such as the dreaded "dependency hell". On the other hand, don't fall into the "Not Invented Here" trap, where an organization tries to build everything from scratch. As in most cases, it is important to find a good balance. The decision to use or avoid a library should always be a conscious one, and the decision process should be 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 for Kotlin[1] to give you an idea of this approach.

2.3. Specifying the Output of the DSL

In most cases, it is fairly obvious what kind of output is expected. Sometimes you have a choice between performing some action (like a query) directly, or creating objects that can perform that action instead. Most of the time, creating objects should be preferred. This approach is usually easier to test and debug, and sometimes it is convenient to be able to create these instances in other ways.

Another potential question is whether to create the target entities of an external API directly, or to follow the Adapter Pattern or similar approaches. In this case, the decision depends on the intended architecture of your application (such as Hexagonal or Onion Architecture), but in general you should try to use non-essential external APIs only at the boundaries of the application, and DSLs should follow this guideline as well.

Most systems produce not only direct output, but also secondary data such as logs, metrics, reports, and so on. Users should have easy and flexible access to this type of data produced by a DSL. For logging, this may mean not forcing a specific logging framework on the user, but rather using a logging facade such as SLF4J[2] in your DSL. Of course, you should also specify the expected secondary output of your DSL.

2.4. 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. In general, people without programming experience need more streamlined DSLs than software engineers, who will often tolerate a few "warts" in the DSL syntax.

2.5. 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, e.g. 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.

Of course, training and documentation are essential to a good learning experience. However, an often overlooked issue is the quality of error and warning messages, which can have a significant impact on how easily users can learn and understand a new DSL. To get these factors right, you need to understand your audience and their skills.

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.

2.6. 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.

In rare cases, the DSL needs to be protected not only against accidental misuse, but also against deliberate attacks. Preventing such attacks is quite difficult, even more so in the case of DSLs, and requires a great deal of technical expertise. Keep in mind that the

techniques shown in this book are designed to protect against accidents, but can often be rendered ineffective by the use of casts or reflection.

2.7. Ensuring 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 rough idea of what extensions might be requested after the first version of the DSL. Later, in the implementation phase, this information will help avoid using techniques that are not flexible enough to handle future requirements.

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 unexpected or insecure results.

2.8. Maintainability

The DSL 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. While a basic bug tracker is essential, larger projects may benefit from additional features such as discussion forums or 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.

2.9. Performance and Memory Requirements

Often, performance considerations don't get much attention. However, in most cases, DSLs call extra operations, instantiate extra classes, and may trigger garbage collections. If you are working with big data or have a wasteful DSL design, you may run into performance and memory problems. That's why it's important to estimate performance requirements up front and use load testing and metrics accordingly.

2.10. Java Interoperability

This is a Kotlin specific question: There are many environments that use a mix of Java and Kotlin, so it may be necessary to use a DSL written in Kotlin from Java code. Usually this direction is more challenging than using Java from Kotlin code, and 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 itself includes some features to increase interoperability with Java.

If Java interoperability is required, it should be considered at the design stage. The challenges and possible solutions are discussed in more detail in Chapter 13.

2.11. 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 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 from the original developers, 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.

2.12. 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.

[1] AutoDSL: https://github.com/F43nd1r/autodsl

[2] SLF4J: https://www.slf4j.org

Chapter 3. Writing a DSL

> Prototype, then polish. Get it working before you optimize it.
>
> — Eric S. Raymond

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

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:** 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.

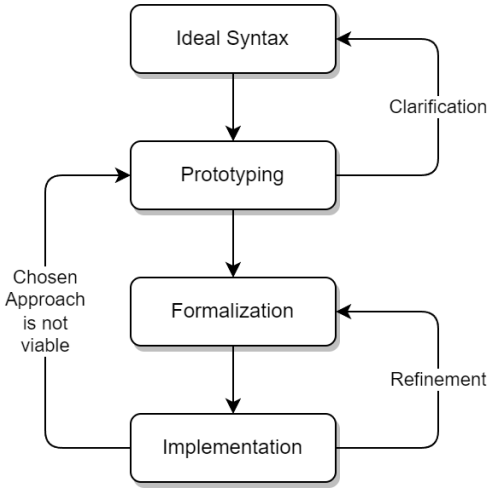

Figure 1. DSL-Development Workflow

By following this straightforward workflow, you will ensure a well-organized and comprehensive structure for your DSL project.

3.1. 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 more 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 create a working program, 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.

3.1.1. Lowering the Cognitive Load

Cognitive Load Theory[1] 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 information about the system 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.

3.2. 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 (also known as "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 the 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.

3.3. 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 "allowed" configurations, and that no rare or unusual cases are overlooked.

A completeness problem in DSL design 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 for 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.

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.

3.4. 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 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.

3.5. DSL-Specific Challenges

Implementing a DSL is often different from the usual programming tasks, and therefore comes with its own challenges and pitfalls. The following are some of the issues that deserve special attention.

3.5.1. 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.

3.5.2. 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 excusable because there is often no other way to write a convenient DSL. But Kotlin is much more expressive, e.g. 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 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 bloated and inflexible. This makes it difficult to evolve the DSL or the business logic independently.

- **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 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.

3.5.3. 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 important 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, documenting that this was a conscious decision not to follow code conventions at this point.

3.5.4. 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 guaranties:** 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.

3.5.5. Documentation

Many software developers don't like to write documentation, but it's important. 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.

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 an example project for a DSL:

- **Make it clear and concise:** Keep the example project focused and to the point, and avoid unnecessary detail, complexity, and external dependencies.
- **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.

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.

[1] Wikipedia - Cognitive Load: https://en.wikipedia.org/wiki/Cognitive_load

Chapter 4. Relevant Language Features

> 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, Toward a Psychology of Being

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 as concise as possible, 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:** 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 also supports functional programming, including 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, JavaScript, and as a native application

In addition, Kotlin has excellent interoperability with Java, making it easy to call Java code from Kotlin and vice versa. Kotlin also includes several annotations that make it easier to call Kotlin code from Java, ensuring that the two languages can work together seamlessly.

As mentioned in the preface, this book assumes that the reader has a basic knowledge of Kotlin. However, DSLs often use little-known language features, or they use some features in a different way than ordinary code. Sometimes even the users of the DSL may not be aware of the kind of "machinery" that makes it work.

This chapter gives a brief overview of language features that may be relevant to writing DSLs. After reading this chapter, you should have a better understanding of how some common DSL "tricks" work.

4.1. Backtick Identifiers

Kotlin allows almost arbitrary identifiers as long as they are enclosed in backticks. The main reason for introducing this feature was to allow the use of identifiers that are keywords in Kotlin but not in Java, such as `fun` or `when`. For DSLs, we can use them when we need descriptive identifiers that don't follow the usual syntactic rules.

Backtick identifiers start and end with a backtick ` and can contain almost any character. Note that the backticks themselves are not part of the identifier, just delimiters. A typical use case for backticks are DSLs for test libraries. While Java limits you to underscores and camel case, Kotlin allows very descriptive names for test functions by using the backtick syntax:

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

For Kotlin on the JVM, all characters except `\r \n , . ; : \ | / [ ? ] < > ` ` 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 `<>`() {
   println("this works!")
}
```

This is especially useful when simulating an operator with an infix function, see the section Infix Notation for Functions for details.

4.2. 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
    )
```

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

// setting all parameters
val color1 =
    makeColor(220, 200, 100, 128)

// using the default value 255 for alpha
val color2 =
    makeColor(220, 200, 100)
```

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

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

```
    // the method is autogenerated, 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 agedPerson = person.copy(age = person.age + 1)
```

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.

While the `copy()` method is convenient when working with immutable data classes, it doesn't scale well for nested data classes: Imagine you need a copy of a company, with the email of the address of the CEO changed. That means you need nested `copy()` calls as well, going down level by level. This is lengthy, hard to read and error-prone.

In functional programming, this problem is often solved with an "optics" package, which contains lenses and similar abstractions that allow to easily compose copy operations for different nesting levels. If you want to learn more about this topic, I suggest you have a look at Arrow Optics[1].

4.3. Trailing Lambda Arguments

If a method expects an argument of a function type, you can use the usual curly bracketed lambda syntax 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 })
```

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
```

```
}
```

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

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

While this syntactic sugar might not look very impressive at first glance, it allows to write very natural looking DSLs for nested structures. Here is an example from the Kotlin documentation:

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

```
html {
    head {
        title {+"XML encoding with Kotlin"}
    }
    // ...
}
```

4.4. Varargs

Varargs (from "variable arguments") are a useful feature in both Java and Kotlin, allowing methods to take a variable number of arguments. However, Kotlin has made several improvements to varargs that make them safer and more convenient to use.

One of the most important improvements in Kotlin is that the syntax for varargs is now unambiguous. In Java, it was sometimes difficult to tell whether an array was intended to be a single argument to a vararg, or whether its elements should be used as individual arguments. Kotlin addresses this problem by introducing the unary "spread operator" *, which indicates that the elements of an array (rather than the array itself) should be used as arguments to a vararg.

In addition, Kotlin allows more flexible use of varargs. You can freely combine single-value arguments with elements of spread arrays:

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

4.4.1. 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 varargMethod(vararg numbers: Int, someString: String) { ... }

varargMethod(1, 2, 3, someString = "Hi!")
```

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

```
varargMethod(
    someString = "Hi!",
    numbers = intArrayOf(1, 2, 3)
)
```

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 put a vararg as the second-to-last argument before a trailing lambda argument.

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

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

As the code snippet shows, no named arguments are required in this case.

4.5. 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
        }
}
```

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 at
${LocalDateTime.now()}")
            return field
        }
}
```

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

4.6. Extensions and Receivers

One of Kotlin's most important features for DSL design are extension functions, lambdas, and properties, which allow you to add functionality to existing classes - even final ones - without touching them. These extensions are standalone constructs that operate on a *receiver*, which is the target class they are extending. The function body is placed in the scope of the receiver, so you can access its public fields, methods, etc., and you can also refer to the receiver itself with `this`. Here is what an extension function looks like:

```
fun Int.digits(base: Int = 10): List<Int> =
    generateSequence(this.absoluteValue) {
        (it / 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]
```

From the user's point of view, the call with receiver looks exactly like a normal method call (or property access) on a receiver instance. This makes extension functions a great tool for adding DSL functionality to classes over which you have no control. A great example is the so-called "scope functions" `apply()`, `run()`, `let()`, and `also()` in the Kotlin API, which make it easier to use e.g. expression body syntax for functions or concise variable assignment.

On the JVM where Kotlin is typically compiled, 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

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

Extension Functions in Different Languages

Extension methods, originally introduced in C#, found their way into Kotlin as "extension functions", specifically with the goal of seamlessly integrating Java classes into the Kotlin ecosystem. This approach was chosen over Scala's *implicit conversion* approach, which, while more powerful, was considered more complex to understand and manage. The benefits of extension methods became so apparent that they were also introduced in Scala 3. Plain Java doesn't provide support for extension methods, but there are Java libraries that do, such as Manifold[2], Project Lombok[3], or Fluent[4]. It is safe to say that extension methods are an important and successful advancement in modern object-oriented programming.

4.6.1. Type Narrowing

Interestingly, generic extension functions have a feature that normal instance methods don't have: They can bind generic parameters to a particular type, thus limiting the range of possible recipients. Here is an example for calculating the product of numbers as an extension function for a list:

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

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

The call to `Double::times` is only possible 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 be used in DSLs to perform compile-time checks.

4.6.2. Loan Pattern

What is the Loan Pattern?

The Loan Pattern is a design pattern in object-oriented programming that involves encapsulating the use of a resource (such as a database connection or file handle) within a limited scope or block of code. The pattern is designed to ensure that the resource is properly acquired, used, and released without the risk of resource leakage or conflicts with other code that may be accessing the same resource.

In essence, the Loan Pattern involves creating a resource object or acquiring a resource handle at the beginning of a block of code, using the resource as needed within the block, and then releasing or disposing of the resource at the end of the block. This ensures that the resource is only used for the duration of the block, and that it is properly cleaned up when the block is finished, even if an error or exception occurs during the block.

The Loan Pattern is particularly useful when resources are limited or expensive to acquire. It can also help improve the maintainability and robustness of code by making it easier to reason about the use of resources and to ensure that they are properly managed throughout the program.

Lambdas can also have receivers, which is useful when using the Loan pattern. Using this pattern can be beneficial in DSLs, as it helps to control the lifecycle of the receiver class and hide the steps necessary to initialize and finalize the instance creation or operation.

For example, 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()
```

```
// the extension function
fun sb(block: StringBuilder.() -> Unit): String =
    StringBuilder()
        .apply { block(this) }
        .toString()

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

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

4.6.3. The @DslMarker annotation

When you nest multiple extension functions, 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:

```
html {
    head {...}
    body {
        head {} // ouch, head() is defined in html's scope, but also visible here
    }
}
```

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

- Define a custom annotation
- Annotate that annotation 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, e.g. using the syntax `this@html.head{⋯}`

In our example, such an annotation might look like this

```
@DslMarker
annotation class HtmlMarker
```

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.

4.6.4. Extension properties

You can not only define 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 to write empty parentheses. In the following example, we want 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: Amount = 22.46.USD

val eurAmount: Amount = 17.11.EUR
```

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

Extension properties are especially useful when the result of the computation is a function. Consider currying, which is a technique for turning functions that take multiple arguments into a chain of functions that take single arguments. Written as an extension property, it could be implemented like this (for functions with two arguments):

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

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

val sf3 = ::someFunction.curry(3)
println(sf3("Abc")) // AbcAbcAbc
println(sf3("x")) // xxx
```

If we had written `curry` as a function, we would have to write `val sf3 = ::someFunction.curry()(3)` instead, which looks very confusing.

4.7. Operator Overloading

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

Overloading Restrictions

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.

4.7.1. 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` and 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` and returns the original value

4.7.2. 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, but is now deprecated.

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:

Operator	Overwriting Function	Remarks
`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`	

Again, `a` must be mutable, `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.

4.7.3. 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): R`	Defines also `!in`.

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

4.7.4. Index Access and Invoke 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(): R`	
`a(b)`	`fun A.invoke(b: B): R`	
`a(b, c)`	`fun A.invoke(b: B, c: C): R`	Or more arguments

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

4.7.5. 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`

4.7.6. Overload Responsibly

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

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.

4.8. 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
}
```

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.

As mentioned above, 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 = 1.2 `^` 3
```

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

Without `infix`, we would have to write `(numbers to strings).zipWith { i, c → "$i → $c" }` instead, which looks quite cluttered.

4.9. Functional Interfaces

Imagine you have an interface for checking strings, with a single abstract function, 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
}
```

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 to implement it anonymously:

```
// note the "fun" keyword
fun interface StringCheck {
    fun check(s: String): Boolean
}

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

I think you will agree that this syntax looks much better, making it useful for DSLs. However, the lambda-like syntax is just syntactic sugar: Under the hood, the compiler translates it into an implementation equivalent to the first example.

4.10. Generics

Generics are a useful abstraction over concrete types in all sorts of contexts, including DLS 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):

```
import java.math.BigDecimal

interface Euro
interface BritishPound
```

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

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

val Double.GBP
    get() = Currency<BritishPound>(this.toBigDecimal())

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

val works = 3.1.EUR + 4.5.EUR // 7.6 €
val worksToo = 2.1.GBP + 4.2.GBP // 6.3 £

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

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 and Phantom Types

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 involves using types to encode complex computations and algorithms 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, but 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 program.

4.10.1. 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
```

The compiler complains `The following declarations have the same JVM signature (average(Ljava/util/List;)D)`. 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

println(average(listOf(1, 2, 3))) // 2.0
println(average(listOf(7.0, 8.0, 9.0))) // 8.0
```

4.10.2. 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.

```
inline fun <reified T> List<T>.combine(): Unit = when(T::class) {
    Int::class -> (this as List<Int>).sum()
    String::class -> (this as List<String>).joinToString()
    else -> this.toString()
}.let { println(it) }
```

```
fun main() {
    listOf<Int>().combine() // 0
    listOf(1,2,3).combine()  // 6
    listOf("x","y","z").combine() // xyz
    listOf(true, false).combine() // [true, false]
}
```

Note the expression `T::class`, which shouldn't work, considering that type erasure removes all generic type information at runtime. 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 basically the inlining allows the compiler to get the generic type information from where the inlining occurs, and make it look like there was no type erasure. It should be noted that inline functions are subject to some restrictions and behave slightly differently than normal functions, e.g. in handling returns.

Java Interoperability

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 Chapter 13.5.1.

4.11. 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 on 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 help ensure that only valid unit conversions are performed, and can help catch errors at compile time rather than at runtime.

@JvmInline

The JVM backend requires a `@JvmInline` annotation, which may be obsolete in the future. Also, the single constructor argument restriction may be dropped. This depends on the introduction of Project Valhalla[5], 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)
```

4.12. Anonymous Objects

While anonymous objects have no name, they still have their own type, which is also unnamed. Here is a somewhat silly example to illustrate this point:

```
object {
    fun sayHi() = "Hello!"
}.sayHi()
```

The anonymous object is not of type `Any`, otherwise we couldn't call `sayHi()` on it. It defines its own (unnamed) type that exposes the variables and functions defined within it. For this reason, anonymous objects can be used in DSLs as a setup or environment for later computations.

A DSL with a method that measures the duration of several calls and returns the average time taken could be written as follows:

```
fun <T:Any> T.measureTime(block: T.() -> Unit): Double {
   val start = System.nanoTime()
   repeat(1000) { block() }
   val end = System.nanoTime()
   return (end - start) / 1000.0
}
```

```
val env = object { val x = complicatedStuff() }

val nsSomeCall = env.measureTime { someCall(x) }
val nsOtherCall = env.measureTime { otherCall(x) }
```

The necessary setup information for the measurement phase is stored inside an anonymous object in the `env` variable.

4.13. Annotations

You can write entire DSLs using annotations, but more often annotations can support DSLs by describing how certain fields or classes should be handled. They are especially powerful if your DSL has a certain default behavior, but needs to take some edge cases or exceptions into account, such as "do not persist this property".

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

Annotation Processors

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[6], which is no longer actively developed, and the recommended Kotlin Symbol Processing API[7] (KSP), which will be discussed in Chapter 12.

Finally, Kotlin includes annotations to support Java access. DSL code is often more difficult to call from Java, so we'll discuss this topic in Chapter 13.

4.14. Reflection

Sometimes you need to inspect or deconstruct classes, call unknown methods, react to annotations, etc., which can be done using reflection. If you need more than the most basic reflection in Kotlin, you need to import a separate dependency:

Gradle (.kts)

```
dependencies {
    implementation(kotlin("reflect"))
```

```
}
```

Maven

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

Depending on your use case, you might also consider alternatives like kotlinx.reflect.lite.

4.15. Experimental Features

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

4.15.1. Context Parameters

Context Parameters[8]] - formerly known as "Context Receivers" - are an experimental feature introduced in Kotlin 1.6.2, and at the writing of this book, its beta phase is planned for Kotlin 2.2.

The basic idea of Context Parameters is to get a class in scope that provides a certain service:

```
interface EnvironmentContext {
    fun getProperty(name: String): String
}

context(EnvironmentContext)
fun methodWithContext() {
    val userName = getProperty("userName")
    ...
}
```

In this scenario, `methodWithContext()` gets access to members of the specified `EnvironmentContext` class, similar to an extension function. However, there is not only a syntactic difference, but also a semantic one: The function does not extend a receiver

class, but is executed in the scope of an unrelated class that provides additional functionality. This design also allows you to have multiple contexts in scope, or to provide a context for an extension function.

To call `methodWithContext()`, you must provide an implementation of `EnvironmentContext`. If the calling function isn't already in the same context, you can use the `with()` function.

```
fun test() {
    val environmentContext = EnvironmentContextImpl()
    with(environmentContext) {
        methodWithContext()
    }
}
```

Using multiple context parameters is a great way to decouple unrelated aspects of the environment while maintaining a high degree of flexibility. Unfortunately, with multiple receivers, the `with()` calls must be nested, although this may change in the future. For now, if you find this impractical, you can use this function presented in the Kotlin Discussions forum:

https://discuss.kotlinlang.org/t/using-with-function-with-multiple-receivers/2062/6

```
@OptIn(ExperimentalContracts::class)
@Suppress("SUBTYPING_BETWEEN_CONTEXT_RECEIVERS")
inline fun <A, B, R> with(a: A, b: B, block: context(A, B) () ->
R): R {
    contract {
        callsInPlace(block, InvocationKind.EXACTLY_ONCE)
    }
    return block(a, b)
}
```

Context parameters come into play when there's a need to include global information within a specific scope while maintaining the flexibility to accommodate different versions. By using a context parameter to provide DSL functionality, you gain control over the scope, you can influence the overall behavior of the DSL, and you can avoid potential naming conflicts.

You can opt in to context parameters by specifying the compiler argument `-Xcontext-receivers`.

4.15.2. Contracts

Kotlin Contracts[9] are a way to give the compiler additional information about the result of functions, allowing it to perform smart casts it couldn't do with just static code analysis.

Here is an example taken from the documentation:

https://kotlinlang.org/docs/whatsnew13.html#custom-contracts

```
fun String?.isNullOrEmpty(): Boolean {
    contract {
        returns(false) implies (this@isNullOrEmpty != null)
    }
    return this == null || isEmpty()
}

...

fun bar(x: String?) {
    if (!x.isNullOrEmpty()) {
        println("length of '$x' is ${x.length}") // smart cast
    }
}
```

You can also narrow the type when you perform 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
```

At the moment, contracts are still quite limited, but in some situations you can use them to protect the DSL user from boilerplate checks, casts, or explicit specification of generics.

You can opt in to experimental contracts using

- the compiler argument `-Xopt-in=kotlin.contracts.ExperimentalContracts` or
- the annotation `@OptIn(ExperimentalContracts::class)`.

4.16. Conclusion

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.

[1] Arrow Optics: https://arrow-kt.io/learn/immutable-data/intro

[2] Manifold Extension Methods: http://manifold.systems/docs.html#the-extension-manifold

[3] Project Lombok Extension Methods: https://projectlombok.org/features/experimental/ExtensionMethod

[4] Fluent: https://github.com/rogerkeays/fluent

[5] OpenJDK Project Valhalla: https://openjdk.org/projects/valhalla

[6] kapt: https://kotlinlang.org/docs/kapt.html

[7] KSP: https://kotlinlang.org/docs/ksp-overview.html

[8] Context Parameters: https://github.com/Kotlin/KEEP/blob/context-parameters/proposals/context-parameters.md

[9] Kotlin Contracts: https://github.com/Kotlin/KEEP/blob/master/proposals/kotlin-contracts.md

Part II - DSL Categories

Chapter 5. Algebraic DSLs

> 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.

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.

5.1. 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: Complex[1] library, which is written in Java. The implementation behaves pretty much as expected:

```
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
```

There are some static factory methods to create `Complex` objects, like `Complex.ofCartesian()` or `Complex.ofPolar()`, and then you can use some 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`. This is a bit risky because of possible name conflicts, so it would also be an option to import `Complex.I` instead.

Next, the arithmetic operations can be written using operator overloading. We not only need operations between two `Complex` instances, but also between `Complex` and `Double`. Here is an example for addition:

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

The other operations are basically the same. We could also support interoperability with `Int` in addition to `Double`, but for the sake of brevity it was omitted. A small improvement would be a helper method for converting double to complex, instead of relying on the clumsy `Complex.ofCartesian()` factory.

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

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

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`.

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)
```

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() = this

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

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

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

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

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

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

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

Now our usage example can be written as:

```
val a = 3.0 + 4.0*i
val b = 5.0 + 6.0*i
val c = a + b
val d = a * b
val e = a pow b
```

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

5.2. Case Study: Modular Arithmetic - Dealing with 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 significantly from standard arithmetic. In addition, neglecting to optimize operations such as multiplication and exponentiation can result not only in slower performance, but also in arithmetic overflows.

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:

```
data class Modulus(val modulus: Long) {
    init {
        require(modulus > 1)
    }

    @JvmInline
    value class Modular(val n: Long)

    val Long.m
        get() = Modular(remainder(this))

    val Int.m
        get() = Modular(remainder(this.toLong()))
```

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

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

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

    operator fun Modular.div(that: Modular) =
        Modular(remainder(this.n * inverse(that.n)))

    private fun remainder(n: Long) = when {
        n < 0 -> (n % modulus) + modulus
        else -> n % modulus
    }

    private data class GcdResult(val gcd: Long, val x: Long, val y: Long)

    private fun inverse(a: Long): Long =
        extendedGCD(a, modulus)
            .run {
                when (gcd) {
                    1L -> remainder(x)
                    else -> throw ArithmeticException(
                        "Can't divide by $a (mod $modulus)"
                    )
                }
            }

    private fun extendedGCD(a: Long, b: Long): GcdResult =
        when (b) {
            0L -> GcdResult(a, 1, 0)
            else -> {
                val result = extendedGCD(b, a % b)
                val x = result.y
                val y = result.x - (a / b) * result.y
                GcdResult(result.gcd, x, y)
            }
        }
}
```

As mentioned above, the division operation requires some arithmetic effort, but otherwise the example is straightforward. One way to use this DSL is to bring the `Modulus`

class into scope using the `with()` function:

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

with(Modulus(10)) {
    println(3.m + 5.m) // Modular(n=8)
    println(3.m - 5.m) // Modular(n=8)
    println(3.m * 5.m) // Modular(n=5)
    println(3.m / 7.m) // Modular(n=9)
    println(3.m / 5.m) // throws exception "Can't divide by 5 (mod 10)"
}
```

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

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.

For those daring enough to delve into an experimental language feature, Context Parameters were specifically designed to address scenarios like these by enabling a more flexible separation between the context scope and its consumers. Here is a rewritten version of the DSL above:

```
data class Modulus(val modulus: Long) {
    init {
        require(modulus > 1)
    }
}

@JvmInline
value class Modular(val n: Long)

context(Modulus)
val Long.m
    get() = Modular(remainder(this))

context(Modulus)
val Int.m
    get() = Modular(remainder(this.toLong()))

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

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

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

context(Modulus)
operator fun Modular.div(that: Modular) =
    Modular(remainder(this.n * inverse(that.n)))

context (Modulus)
```

```
private fun remainder(n: Long) = when {
    n < 0 -> (n % modulus) + modulus
    else -> n % modulus
}

private data class GcdResult(val gcd: Long, val x: Long, val y: Long)

context(Modulus)
private fun inverse(a: Long): Long =
    extendedGCD(a, modulus)
        .run {
            when (gcd) {
                1L -> remainder(x)
                else -> throw ArithmeticException(
                    "Can't divide by $a (mod $modulus)"
                )
            }
        }

private fun extendedGCD(a: Long, b: Long): GcdResult =
    when (b) {
        0L -> GcdResult(a, 1, 0)
        else -> {
            val result = extendedGCD(b, a % b)
            val x = result.y
            val y = result.x - (a / b) * result.y
            GcdResult(result.gcd, x, y)
        }
    }

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

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:

```
context(Modulus)
fun square(n: Modular) = n * n
```

```
...

val x = modulus(7) {
    square(3.m + 5.m) + square(3.m - 5.m)
}
println(x) // Modular(n=2)
```

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

5.3. 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 the 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()));` in a Java class.

5.4. Conclusion

The case studies presented in this chapter serve to illustrate that creating algebraic DSLs is usually not that difficult. However, it is important to consider certain factors when deciding whether to use an algebraic DSL. Although algebraic notation can be powerful and expressive, it is not always appropriate for every use case.

For example, the use of algebraic notation in a type-constructing DSL to denote sum and product types may be unconventional and potentially confusing to some users, despite the underlying algebraic structure. In addition, certain behaviors, such as non-commutative multiplication found in quaternions and matrices, can introduce unexpected complexity and increase the likelihood of usage errors.

Therefore, it is crucial to exercise good judgment and adhere to the *Principle of Least Surprise* when designing algebraic DSLs, rather than blindly adopting them because of their ease of implementation.

Common Applications

- Defining operations
- Data transformation

Pros

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

Cons

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

[1] Apache Common Numbers: https://github.com/apache/commons-numbers/tree/master/commons-numbers-complex

Chapter 6. Builder Pattern DSLs and Method Chaining

> Builders are the poets of object creation, crafting instances with eloquent syntax and profound semantics.
>
> — ChapGPT 3.5

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 *Typesafe Builder Pattern*, and we will also cover the related *Counting Builder*. Finally, we discuss why builders are not as widely used in Kotlin as they are in Java.

6.1. Classical Builder

As an example for 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:

https://docs.oracle.com/en/java/javase/18/docs/api/java.net.http/java/net/http/HttpRequest.Builder.html

```
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()
```

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, providing 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:

```
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)
}
```

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)
}
```

6.2. 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 an RGBA color:

```
data class Color(
    val red: Int,
    val green: Int,
    val blue: Int,
    val alpha: Int = 255
)

val c = Color(0, 100, 130, 200)
```

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 simply clarify the meaning by rewriting the last line as follows:

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

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 c = Color(0, 100, 130, 200)
val c1 = c.copy(red = 120)
```

The `copy()` method is autogenerated for all `data` classes. In Java, you would have to write a method equivalent to `fun withRed(r: Int): Color` (sometimes called *wither* similar 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.

6.3. 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:

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

```
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()
```

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.

6.3.1. 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()
```

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)
    }
}
```

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()
```

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 implement such state transitions in a safe way, so that the code won't compile if the start and end methods are placed incorrectly.

While there are certainly valid use cases for a flattening builder, the usual approach based on nesting is not only simpler conceptually and implementation-wise, it also scales better, and should therefore be preferred.

6.3.2. Nesting with Varargs

If you are building a tree-like structure where each node has only one type of child, method chaining using varargs can lead to quite elegant DSLs. A good example is data validation, where you check if a value matches your specification, and if it does not, you get a list of problems. You can have validators that just check a simple condition, like a `String` not being blank, but you can also have validators that combine the results of sub-validators.

Here is our validation result type:

```
sealed interface Validation {
    data object Success : Validation
    data class Failure(val reasons: List<String>) : Validation

    operator fun Validation.plus(that: Validation): Validation = when {
        this is Failure && that is Failure ->
            this.copy(reasons = this.reasons + that.reasons)
        this is Success -> that
        else -> this
```

```
    }
}
```

Now we need the `Validator` interface and a validate extension function that can be called on any object. When nesting validators, we would not only see our own `validate()` method, but also the one from our caller, etc., so we use the `@DslMarker` mechanism to hide the latter. We also include a helper function to construct a validator based on a condition.

```
@DslMarker
annotation class ValidationDsl

@ValidationDsl
fun interface Validator<T> {
    fun validate(t: T): Validation
}

fun <T> T.validate(
    vararg validators: Validator<T>
): Validation = validators
    .fold<_, Validation>(Validation.Success) { result, validator ->
        result + validator.validate(this)
    }

private infix fun Boolean.then(reason: String) = when {
    this -> Validation.Failure(listOf(reason))
    else -> Validation.Success
}
```

Writing simple validators is straightforward, especially since `Validator` is a functional interface, so I'll only show a few examples for string validators:

```
fun notBlank() = Validator<String> {
    it.isBlank() then "String can't be blank"
}

fun minLength(min: Int) = Validator<String> {
    (it.length < min) then "String '$it' must have at least $min characters"
}

fun maxLength(max: Int) = Validator<String> {
    (it.length > max) then "String '$it' must have at most $max characters"
```

```
}
```

Using them is very simple, a typical call could be `myString.validate(notBlank(), maxLength(10))`.

Now comes the interesting part, namely validators that take other validators as arguments, or more specifically, as varargs:

```
fun <T> forAll(vararg validators: Validator<T>) = Validator<List<T>> {
    it.fold<_, Validation>(Validation.Success) { result, element ->
        result + element.validate(*validators)
    }
}

fun <T, S> KProperty1<T, S>.validate(
    vararg validators: Validator<S>
) = Validator<T> {
    validators.fold<_, Validation>(Validation.Success) { result, validator ->
        result + validator.validate(this.call(it))
    }
}

fun <T, S> KFunction1<T, S>.validate(
    vararg validators: Validator<S>
) = Validator<T> {
    validators.fold<_, Validation>(Validation.Success) { result, validator ->
        result + validator.validate(this.call(it))
    }
}
```

The `forAll()` function applies the given validators to all elements of a list. The next two functions allow you to validate properties and getter-like functions of the given value. If you look at the sample calls, you can see how the nested varargs give the DSL a tree-like shape, without much syntactic clutter:

```
data class Child(val name: String, val friends: List<String>)

val child = Child("Charlie", listOf("Snoopy", "Lucy", "Linus"))

val result = child.validate(
    Child::name.validate(
        minLength(2),
```

```
        maxLength(3)
    ),
    Child::friends.validate(
        forAll(
            notBlank(),
            maxLength(5)
        )
    )
)

// Failure(reasons=[
//   String 'Charlie' must have at most 3 characters,
//   String 'Snoopy' must have at most 5 characters
// ])
```

As mentioned above, this style isn't always feasible, but when it can be used, it often improves readability.

6.4. The Typesafe 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>)
```

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>()
```

With the help of these classes, we can write the builder:

```
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>) {
    ...
}
```

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)
}
```

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 (at least if we don't want to use casts).

Of course, a crucial part is missing: The `build()` method. 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)
```

Note that you can access the `value` fields of the `With` classes, 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:

```
ProductBuilder()
    .id(UUID.randomUUID())
```

```
    .name("Comb")
    .description("Green plastic comb")
    .price(12.34.toBigDecimal())
    .build()
```

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

Origins

The Typesafe Builder Pattern was pioneered by Rafael Ferreira[2] in Scala, using ideas from Haskell. The code shown here is based on the implementation by Daniel Sobral[3].

6.5. 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>)
```

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 obvious 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. This can be achieved by using a recursive type parameter to keep track of the number of points, again using Type-Level Programming. While this may seem odd 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)
```

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 to check our condition. These recursively constructed numbers are used by the builder class as a generic "counter" parameter, holding the number of items in the list.

The Peano Axioms

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. In 1889 Giuseppe Peano published his famous nine axioms to define natural numbers and their properties in a formal way, and the Peano Representation follows directly from these axioms.

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 next four axioms rely crucially on the use of the successor function:

- For every natural number, its successor is also a natural number.
- If the successors of two natural numbers are equal, then the numbers themselves are equal.
- Zero is not a successor of a natural number.
- 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 more fundamental way of writing natural numbers, and our usual number systems (decimal, binary, hexadecimal…) can be regarded as convenient abbreviations.

Similar to the Typesafe Builder Pattern, the builder class must hide its constructor, because a call like `PolygonBuilder<S<S<Z>>>()` would initialize the builder with a 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, but also 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 Typesafe 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()
```

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 Typesafe Builder Pattern.

6.6. 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.

6.7. Conclusion

The Builder Pattern is quite popular in Java - there are even libraries like Project Lombok[4] 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 Typesafe 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 autogenerated (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 (e.g. need for a `build()` method)
- Assignments are disguised as method calls

[1] KotlinPoet: https://square.github.io/kotlinpoet

[2] Rafael Ferreira, Typesafe Builder Pattern in Scala: http://blog.rafaelferreira.net/2008/07/type-safe-builder-pattern-in-scala.html

[3] Daniel Sobral, Typesafe Builder Pattern: http://dcsobral.blogspot.de/2009/09/type-safe-builder-pattern.html

[4] Project Lombok: https://projectlombok.org

Chapter 7. Loan Pattern DSLs

> Neither a borrower nor a lender be,
> For loan oft loses both itself and friend,
> And borrowing dulls the edge of husbandry.
>
> — 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 used instead of, the classic builder pattern discussed in the last 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, it will be referred to here as the "Loan Pattern DSL".

The Loan Pattern DSL uses a mutable builder class that exposes its members through the Loan Pattern introduced in Chapter 4.6.2. 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 hard to read when using the classic builder pattern. It can also be easier to enforce constraints and ensure type safety with this approach compared to using builders. Sometimes the greater flexibility of Loan Pattern DSLs allows other DSL techniques to be incorporated, while the builder approach is often too rigid for such improvements. That's why it's often used as a scaffolding for hybrid DSLs, as will be discussed in Chapter 11.

In this chapter, we will first write a more convenient replacement for the `HttpRequest.Builder` introduced in the last chapter. If you have control over the business classes you want to construct, you can use libraries like AutoDSL to generate a DSL for you. We will explore this in the second part of this chapter.

7.1. Case Study: HttpRequest DSL

As discussed in the last chapter, `HttpRequest` already comes with a builder that looks 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()
```

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 request = 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
}
```

There are no more `newBuilder()` and `build()` calls, the assignments are really assignments, headers have their own subsection, and durations can be calculated. For convenience, you can choose to specify the web address as either a string or a URI (the example shows the `String` version).

As you may have guessed, `httpRequest` is a method that uses the Loan Pattern: It initializes a builder class, exposes it as a receiver, and takes care of the final `build()` call. It also takes care of initializing a mandatory field (the URI) by requiring it as an explicit argument. We have two versions of this function, depending on how the URI is specified:

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

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

The builder contains some mutable fields, and a `build()`-Method to construct the `HttpRequest`:

```
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()
        }

    ...
}
```

The `headers` field can't be set directly, because we want to use a nested structure here. The `headers()` method works analogous to 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 looks a bit like `:`. We don't want to expose all of the `HttpRequestBuilder` fields in its inner `Headers` class, so we use the `@DslMarker` mechanism that we had discussed in Chapter 4.6.3 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
        }
    }
   ...
}
```

Next, some fields and methods are required to assist in setting the HTTP method. However, you can still define your own, e.g. `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
    ...
}
```

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)
}
```

And that's almost it, we covered everything except some details of the `build()` method. Here is the complete code:

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

fun httpRequest(uri: String, block: HttpRequestBuilder.() -> Unit) =
    HttpRequestBuilder(URI.create(uri)).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)
}
```

Retrofitting HttpRequestBuilder with a Loan Pattern DSL proved to be a relatively simple task, but the resulting DSL is convenient and idiomatic. By customizing existing libraries in this way, especially those written in Java, it becomes easier to meet user needs and integrate them more seamlessly into the Kotlin ecosystem.

7.2. Case Study: HttpRequest with AutoDSL

Since this type of DSL is very common, and its structure is quite predictable, it shouldn't be surprising that there are libraries for automatically deriving such DSLs. At this point we will discuss the AutoDSL[1] library, which has to be set up as an annotation processor (either via kapt or KSP). Please follow the description on the GitHub project page.

Please make sure you are using the correct GitHub project. There is an older library called "AutoDsl" which was the inspiration for this project. Unfortunately, it is no longer maintained and does not work with Kotlin 1.4 or newer.

Remember the work we put into `HttpRequestBuilder` in the last section? Let's see what we can get "for free" instead. Note that we can't annotate the `HttpRequest` class itself, so we have to use an intermediate class instead, and therefore we have to call the `build()` method at the end. Normally, we wouldn't do this for classes under our control, but would annotate them directly.

```
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()
        }
}
```

It doesn't get much easier than that: All classes that should be included in the DSL are marked with the `@AutoDsl` annotation, and 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.

If you compile the project using IntelliJ IDEA, you should normally find the generated classes `HeaderDsl` and `HttpRequestBuilderDsl` in a `generated-sources/⋯` folder, but of course this depends on how you have integrated the AutoDSL processor and how you have set up your project.

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()
```

Granted, the code isn't quite as convenient and concise as the manually written DSL, but it comes close, and definitely looks nicer and more intuitive than a traditional builder. AutoDSL also keeps track of mandatory fields like `uri` and throws an `IllegalStateException` if they are not set.

7.3. 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 Kotlin Documentation - Using builders with builder type inference[2].

7.4. Conclusion

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 autogenerated (e.g. using AutoDSL)

- Allows to limit the visibility, e.g. 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

[1] AutoDSL: https://github.com/F43nd1r/autodsl

[2] Kotlin Documentation, Using builders with builder type inference: https://kotlinlang.org/docs/using-builders-with-builder-inference.html

Chapter 8. Modeling State Transitions in DSLs

> All things change in a dynamic environment. Your effort to remain what you are is what limits you.
>
> — Ghost in the Shell

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.

8.1. 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 (for a serious SQL DSL implementation, see jOOQ[1]). The following state diagram gives an overview of the allowed transitions:

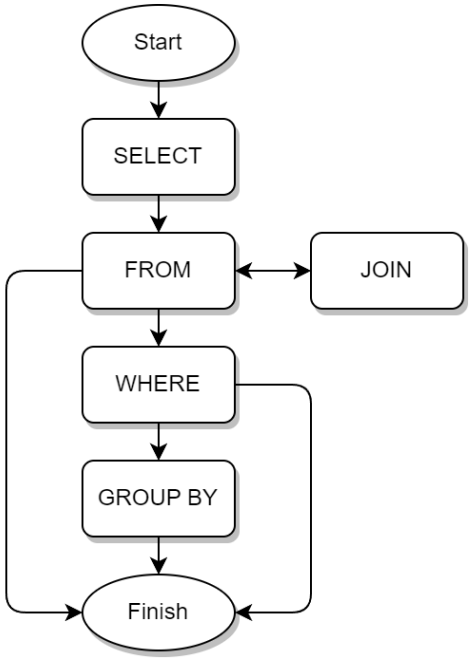

Figure 2. Simplified Select SQL Query

The final DSL will allow 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()
```

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.

8.1.1. 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.

Data Transfer Objects

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 subsystems of an application, and to reduce the amount of data that needs to be exchanged between them. The primary purpose of a DTO is to provide a simple, standardized way of representing data that can be easily serialized and deserialized.

In preparation for writing the SQL query 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()
)
```

Then we can write the `SELECT` part, which is straightforward:

```
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)
            )
        )
}
```

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)
}
```

From here you can go to a `JoinClause`, which mimics SQL syntax by allowing you to write something like `fromClause.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)
        ))
}
```

The `WhereClause` is quite simple, but of course using `String` to represent the different conditions is not very secure and should be avoided in production code. Our SQL subset allows us to proceed to the `GroupByClause` (while the full syntax would also allow `HAVING`, `ORDER BY`, etc.). 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)
}
```

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

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

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

```
}
```

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()
```

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.

8.1.2. 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, e.g. 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.

The Chameleon Class

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
}
```

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()
}
```

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.

8.1.3. 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 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>
```

The extension functions for the state transitions are straightforward:

```
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()
```

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

Declaration-Site vs Use-Site Variance

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 (`out`), contravariant (`in`), or invariant (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.

8.2. 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"
}.GROUB_BY{
    +"p.age"
}.build()
```

This looks quite different from the Builder pattern syntax, and it is 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`. Also, we still need `build()` methods, as it is not clear when we are done constructing the query. In cases with only one exit state, the construction can be done behind the scenes, as usual in Loan Pattern implementations.

Note that for a serious implementation, the @DslMarker mechanism should be used, since the join clause is nested, but for the sake of brevity it won't be used in the following use cases.

8.2.1. 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()
)
```

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)
            )
        }
}
```

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)
}

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)
        ))
}
```

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

```
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)
}

data class GroupByClause(val queryDTO: QueryDTO) {

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

The `build(queryDTO)` function 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.

8.2.2. 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
}
```

All the ⋯Body classes remain unchanged, so we will skip them. 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()
}
```

8.2.3. 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, and are omitted.

```
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()
```

8.3. Conclusion

In this chapter we discussed how state transitions can be expressed using different techniques. The DSLs can use either an underlying builder pattern or a loan pattern syntax, and there are different approaches to implementing them. When in doubt, I would recommend starting with the separate classes approach, especially when 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 code 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

[1] https://www.jooq.org

Chapter 9. String-Parsing DSLs

> Words are, of course, the most powerful drug used by mankind.
>
> — Rudyard Kipling

In some cases, the host language doesn't allow you to create the DSL syntax you want. A string parsing DSL gives you the freedom to write basically anything. One area where other DSL categories are particularly bad is in giving the user some leeway, for example with respect to uppercase and lowercase syntax, the order of operations, or the use of special characters such as symbols. String-based DSLs can easily incorporate such leniency. In addition, Kotlin's support for multiline (or "raw") strings makes it convenient to write larger string-based DSLs, and allows for "graphical" DSLs, such as 2D representations of game levels.

One drawback is that these DSLs are hidden from the scrutiny of the compiler: There are no compile-time checks, no auto-completion hints, etc., so the user has to rely on the documentation. Another issue is writing the parser, which is usually more involved than collecting the data in other DSL types, although I hope this chapter convinces you that it is nothing to be afraid of either.

Regular Expressions

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.

9.1. Case Study: Forsyth–Edwards Notation

Before we dive into complicated problems requiring parsers, we will start with a simple example where we only need a few string functions. If you don't play chess, you've probably never heard of the Forsyth–Edwards Notation[1] (FEN):

Forsyth–Edwards Notation

The Forsyth-Edwards Notation is a standard notation for describing the current position of pieces on a chessboard (as opposed to Portable Game Notation, which is

used to document entire games). It consists of a string of characters representing the placement of pieces, organized by rank. In FEN, uppercase letters are used to represent white pieces and lowercase letters are used to represent black pieces. The numbers 1 through 8 are used to represent successive empty squares, and a slash is used to separate ranks. FEN also contains information about:

- which player moves next
- castling availability
- en passant capture availability
- number of half-moves without check or pawn moves (for the 50- or 75-move rule)
- number of moves played so far

Rows are separated by `/` and 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, e.g. three consecutive empty squares are represented by a `3`. Castling rights are indicated by `k` for king's side and `q` for queen's side, again in upper case for white and lower case for black. If no one can castle, a `-` is written instead. The en-passant entry contains either a square like `b3` or another `-`.

A FEN looks like this: `rnbqkbnr/pp1ppppp/8/2p5/4P3/5N2/PPPP1PPP/RNBQKB1R b KQkq - 1 2`. 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
```

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 fiftyMoves: 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 FEN() = "${boardFen()} ${toMove.symbol} " +
            "${castlingFen()} $enPassant $fiftyMoves $move"
}
```

The class already contains a function to generate a FEN - this is not required, but it makes testing much easier.

Round-trip Tests

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(s) 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:

```
fun readFEN(fenString: String): Position = fenString
    .split(" ")
    .let { part ->
        Position(
            pieces = getPieces(part[0]),
            toMove = getToMove(part[1]),
            castling = getCastling(part[2]),
            enPassant = part[3],
            fiftyMoves = 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 ->
            values().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
        }
    }
```

The `readFEN()` function calls some helper functions for the different parts, and assembles the `Position` class. Most of the sanity checks have been omitted for better readability. In simple cases like this, it is probably overkill to write a parser or use a parser library. The hardest part was to read the piece positions correctly, and that took only a few lines.

9.2. 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 - Hybrid DSLs.

For our case study, we won't cover the full notation, e.g. we won't support writing ions or bonds. An example of a simple chemical equation in standard notation would be $3Ba(OH)_2 + 2H_3PO_4 \rightarrow 6H_2O + Ba_3(PO_4)_2$. 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>
) {
    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()
```

An `Element` contains a chemical symbol, like `"H"` (hydrogen) or `"Ba"` (barium), and optionally a subscript that counts the number of atoms. A feature of the chemical notation is that you can also define groups like "$(OH)_2$" in a molecule, which is why we need the `Group` class as well. A group can not only contain elements, but also other groups.

A `Molecule` is a collection of elements or groups (which we subsume under a `Part` interface), and can also have a coefficient in front of it. An equation consists of two sides and either an arrow `->` or `-` in case of reversible reactions - a double arrow `<->` in the middle. Both sides consist of either a single molecule or a "sum" of molecules.

The code overwrites the `toString()` methods in order to give the output in chemical notation. Note that lists were used instead of varargs, because data classes don't allow varargs in their primary constructor.

This diagram summarizes the structure of our model classes:

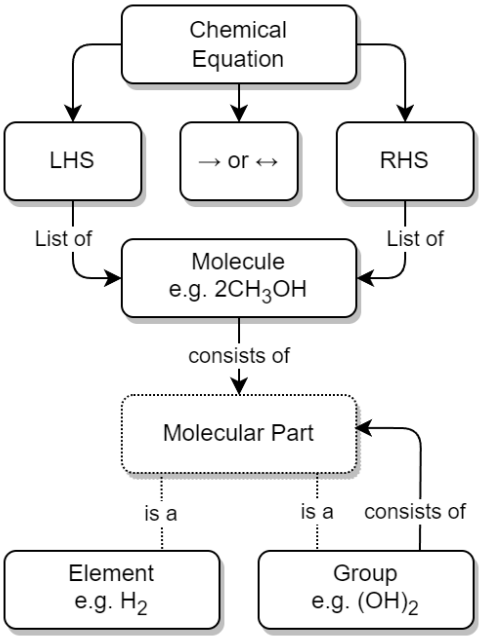

Figure 3. Model for Chemical Equations

The above equation for making barium phosphate could be written as follows:

```
val Ba = Element("Ba")
val Ba3 = Element("Ba", 3)
val O = Element("O")
val O2 = Element("O", 2)
val O4 = Element("O", 4)
val H2 = Element("H", 2)
val H3 = Element("H", 3)
val P = Element("P")

val bariumHydroxide = Molecule(3, listOf(Ba, Group(listOf(O, H), 2)))
val phosphoricAcid = Molecule(2, listOf(H3, P, O4))
val water = Molecule(6, listOf(H2, O))
val bariumPhosphate = Molecule(1, listOf(Ba3, Group(listOf(P, O4), 2)))

val equation = Equation(
    listOf(bariumHydroxide, phosphoricAcid),
    Arrow.IRREVERSIBLE,
    listOf(water, bariumPhosphate))
```

```
println(equation) // 3Ba(HO)2 + 2H3PO4 -> 6H2O + Ba3(PO4)2
```

9.2.1. 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 really difficult, but boring and tedious, so using a library will be the better choice most of the time. Nevertheless, I think it is instructive to see how a simple parser works, so a naive manual implementation will be presented first, before using a parser combinator library.

First, we need some general code for a rudimentary parser. We start with a common interface `ParseResult`, as we also need to cover the case when parsing a certain element fails. Real-world implementations would include useful information in this `Failure` class, but for our use case we will leave it empty. We then need a `Success` class to hold the current successful parsing result, along with 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. All in all, 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>
```

Then we have 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)
    }
```

Now that we have some minimal parsing support in place, we can start working on equation parsing. Note that we assume that there are no whitespaces in the formula, because dealing with them everywhere is tedious, and we can easily filter them out at the top level.

First, we need to know all the element symbols:

```
private val elements = setOf(
```

```
    "H", "He", "Li", "Be", "B", "C", "N", "O", // etc.
)
```

Next, we need functions recognizing given patterns and natural numbers:

```
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))
        }
    }
```

The simplest function is `parsePattern()`, which tries to find a given prefix in the string. `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 "parser combinator".

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 } }
```

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 just find `H` in a string starting with `He`.

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 you 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))
    }
```

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 1 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)
        }
    }
```

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()`.

Now 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)
            }
        }
```

First, the function looks for a possible coefficient in front, otherwise it uses 1 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 side of the equation:

```
fun parseSide(string: String): ParseResult<List<Molecule>> =
    sequence(parseMolecule(string)) { remaining ->
        parsePattern(remaining, "+")
            .flatMap { _, s2 -> parseMolecule(s2) }
    }
```

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)
            }
        }
    }
```

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 only part of our DSL that will be exposed to the user:

```
fun equation(string: String): Equation? =
    parseEquation(string.replace(" ", ""))
        .orNull()
        ?.let { result ->
            result.value.takeIf { result.remaining.isEmpty() }
        }
```

This function removes all spaces from the input string, calls the parser, checks that no "unparsed" string is left, and returns the result or null. Now we can write e.g. `equation("3Ba(OH)2 + 2H3PO4 -> 6H2O + Ba3(PO4)2")`, which is as concise as possible for an internal DSL.

As mentioned in the last chapter, a "real" chemical equation looks more like $3Ba(OH)_2 + 2H_3PO_4 \rightarrow 6H_2O + Ba_3(PO_4)_2$, 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.

9.2.2. 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[2] project, 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")
```

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. However, you can still see the same pieces of grammar, assembled in a similar way to our

original code.

9.3. Conclusion

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
- It is easy to allow for some leniency
- Very flexible and extensible
- 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

[1] Wikipedia, FEN: https://en.wikipedia.org/wiki/Forsyth%E2%80%93Edwards_Notation
[2] better-parse: https://github.com/h0tk3y/better-parse

Chapter 10. Annotation-based DSLs

> I have discovered a truly marvelous proof of this, which however the margin is not large enough to contain.
>
> — Pierre de Fermat

Annotations can be used to bind specified actions or behaviors to classes, methods, fields, etc. Typical examples include dependency injection frameworks (such as Spring or Dagger), serializers to formats such as XML or JSON (such as Gson, Jackson, or JAXB), or authorization and access-control (such as Spring Security). In Java, Project Lombok[1] aims to reduce the boilerplate of the language in many ways, and uses annotations to do so.

As mentioned in Chapter 4.13, you may want to look into annotation processors if you need to integrate your DSL into the build process in some way.

10.1. Case Study: Mapper DSL

Mapping between similar classes is a common and often tedious task. There are already many solutions, such as the MapStruct[2] library, which is a code generator for mapper classes.

In this case study, we will write a very simple mapper DSL to transform e.g. 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)
```

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 person = UserToPerson.map(getSomeUser())
```

The definition of the `@Mapping` annotations is straightforward. They must be present at runtime, they should only be valid on classes, and it must be possible to use multiple of them (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
)
```

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()
}
```

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 AgeTransformer : Transformer<ZonedDateTime, Int>({ z ->
    ChronoUnit.YEARS.between(z, ZonedDateTime.now()).toInt()
})
```

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, this is neither pretty nor secure, but here goes:

```
abstract class Mapper<S : Any, T : Any> {
    fun map(s: S): T {
        val annotations = this::class.findAnnotations(Mapping::class)

        val targetType = this::class.supertypes[0]
            .arguments[1]
            .type!!
            .classifier as KClass<*>

        val targetConstructor = targetType.primaryConstructor!!

        val targetArgs = targetConstructor.parameters.map { targetParam ->

            val ann = annotations.find { it.target == targetParam.name }

            val sourceParam = ann?.source ?: targetParam.name

            val sourceValue = s::class.memberProperties.find {
                it.name == sourceParam
            }!!.getter.call(s)

            ann?.transformer?.isSubclassOf(Function1::class)
                .takeIf { it == true }
                ?.let {
                    val transformer = ann!!.transformer.objectInstance
                        ?: ann.transformer.primaryConstructor!!.call()
                    transformer::class.memberFunctions
                        .find { it.name == "invoke" }!!
```

```
                        .call(transformer, sourceValue)
                } ?: sourceValue
        }.toTypedArray()

        @Suppress("UNCHECKED_CAST")
        return targetConstructor.call(*targetArgs) as T
    }
}
```

Since it uses reflection, you need to include a dependency on the `kotlin-reflect` library, as described in Chapter 4.14. 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 won't work.

Convention over Code

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 usability.

For serious applications, I would suggest checking out MapStruct[2]. It is a Java library, but seems to work well with Kotlin, and has 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.

10.2. 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-based DSLs in annotations.

Implementing such a DSL wouldn't provide much new insight, but the 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>
}
```

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.

10.3. Conclusion

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
- Hard to debug

[1] Project Lombok: https://projectlombok.org
[2] MapStruct: https://mapstruct.org

Chapter 11. Hybrid DSLs

> In diversity, there is beauty and strength.
>
> — Maya Angelou

In many cases, it is difficult to categorize a DSL because it uses a mix of different techniques: Some parts might resemble a builder, others might look like an algebraic DSL, and one place uses annotations. That's what I call a "hybrid DSL".

If most of a DSL can be written in a certain style, it is usually a good idea to stick to that style to ensure a consistent look and feel. So I wouldn't advocate mixing Builder and Loan Pattern styles, but sticking to one. But there is nothing wrong with designing a DSL that combines different approaches, if it gets the job done and manages to provide a coherent language.

11.1. Quasi-Lingual DSLs

A common reason for mixing 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 the syntax is usually rigid and contains artifacts such as braces, parentheses, or commas, it is still "readable" in a very literal sense. This snippet from the mocking library MockK[1] should give you a good idea:

```
every { car.door(DoorType.FRONT_LEFT).windowState() } returns WindowState.UP
```

Technically, writing a quasi-lingual DSL is not very different from writing other hybrid DSLs, although it may be harder to model grammatical structures correctly. Also, naming conflicts with keywords like `is`, `as`, etc. are more of a problem. A good rule of thumb is to keep the DSL grammar simple and well-structured, and to compromise when it helps to reduce complexity. For example, instead of trying to model words like `should` and `be` separately, it might be advisable to use `shouldBe` instead if it helps to keep the grammar simple.

11.2. Case Study: Chemical Equations

Writing a DSL for chemical equations is not an easy task because the notation is very concise and cannot be easily mimicked using Kotlin syntax. For our case study, we will try 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
) : Part {
    override fun toString() = symbol + subscript.oneAsEmpty()
}

data class Group(
    val parts: List<Part>,
    val subscript: Int
) : 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())

    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()
```

As a refresher, here is the corresponding structure diagram:

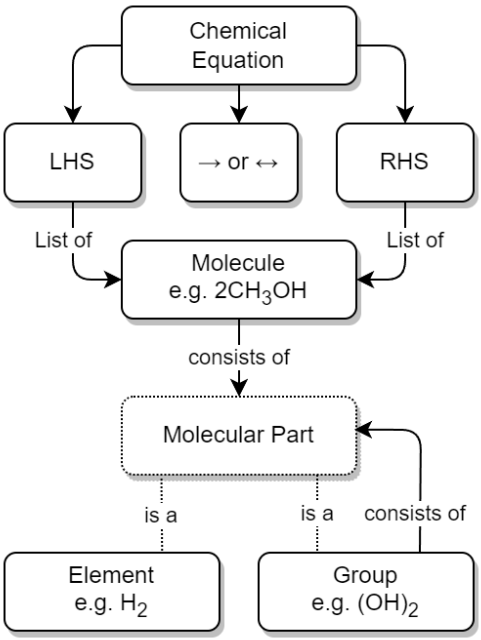

Figure 4. Model for Chemical Equations

As a first step, we define the elements as `val`s:

```
val H = Element("H", 1)
val He = Element("He", 1)
val Li = Element("Li", 1)
// etc.
```

Next, we need a convenient way to add the subscript to an element or group. We could use the invoke operator `()` or the index access operator `[]`. Since we will need parentheses in other places, the index access operator seems to be the better choice for this job. 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)
```

The code for both functions includes a sanity check that won't allow nonsensical calls like `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, e.g. a water molecule could be written as `H[2]-O`:

```
operator fun Part.minus(that: Part) =
    Molecule(1,listOf(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)
```

In general, we have to put groups and sometimes molecules in parentheses because of the precedence rules. A molecule can have an optional coefficient in front of it. Also, the same multiplication operation should "promote" an element or group to a molecule, allowing for example to write `2*O[2]` for an oxygen molecule with coefficient two. For the promotion we introduce the extension function `Part.toMolecule()`. Again, we need sanity checks to prevent calls like `-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 && this.subscript == 1 -> Molecule(coefficient, this.parts)
    else -> Molecule(coefficient, this)
}
```

Next, we need a way to group the left and right sides of an equation into a list of molecules, and the obvious choice for an operator is +. As before, we "promote" molecule parts to full molecules when necessary. This time, the precedence rules for * and + play nicely with the intended use, so we don't need parentheses 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()
```

In case you wonder why there is no `List<Molecule>.plus(that: Molecule)` function: This would be just a special case of adding elements to a list, which is already defined in the standard library.

The last part of the DSL is collecting everything into an equation. This is not complicated, but 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, we use some helper functions. We also need to consider the two different types of equations, reversible and irreversible.

For the arrows we can use the backtick notation. Since `` `->` `` and `` `<->` `` contain characters that are not valid on the JVM, we have to rename the functions there using `@JvmName`, and unfortunately we also have to suppress the corresponding compiler warnings.

```
@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))

// same functions with `<->`, for equations with Arrow.REVERSIBLE
```

So, how does our DSL look 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]
```

There is an optional improvement, which is more a matter of taste: We could add some extension properties for low subscript numbers of elements and groups, which would allow to write e.g. `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)
// 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)
// etc.

// new syntax
val eq = 3*(Ba-(O-H)._2) + 2*(H._3-P-O._4) `->`
    6*(H._2-O) + (Ba._3-(P-O._4)._2)
```

Please decide for yourself which version you prefer. Personally, I find the syntax with the index operator `[]` more readable.

Simulating the dense chemical notation is hard, and while using operator overloading and infix notation made our example substantially shorter, it still contains a lot of clutter. Of course, after some time one would get used to the DSL, but there is clearly a learning curve involved. You have already seen how the same problem can be tackled with a string-based DSL, which seems to be the more elegant approach in this specific case. Nevertheless, it is still impressive how far you can push the syntax towards such a specific notation in Kotlin.

11.3. Case Study: Pattern Matching

Kotlin's `when` is certainly more versatile than Java's `switch`, but languages like Scala or

Haskell go a step further and allow pattern matching. This feature allows you to deconstruct and match values against specific patterns. It provides a concise and powerful way to perform conditional branching and data extraction based on the structure and content of the input.

In pattern matching, you define a set of patterns that describe the possible forms or values that an input can take. These patterns can include literals, variables, data constructors, or even more complex patterns such as lists or tuples. The language then matches the input against these patterns and executes the corresponding expression associated with the first matching pattern.

In this case study, we want to provide similar functionality in Kotlin, although 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
)

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", age) ->
        "Some other Andy of age $age."

    else -> "Some unknown person."
}
```

However, we have to make some compromises to make it work in Kotlin:

- We can't use `Person` in the match cases, but we need to write a helper function (which we will call `person`).
- It is difficult to support a mix of literal values and patterns, so we need to wrap values like `"Andy"` in a pattern, e.g. using unary plus, like `+"Andy"`.
- For numbers, unary plus can't be used, so we fall back on a syntax like `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 need lambda braces for lazy evaluation.
- Capturing variables on the left and using them on the right requires using a `val` to define a capture pattern.
- `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.
- _ isn't a valid identifier, so we use `any()`
- In some cases, we need to provide 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
        { "Some other Andy of age ${ageCapture.value}." }

    otherwise { "Some unknown person." }
}
```

That's not too bad, even if the `+' prefix looks a bit odd at first. The problem is that the choice of overridable operators in Kotlin is quite limited. That's why the unary plus has become something of a standard for such use cases, and is also used that way in the Kotlin documentation[2].

The core of the DSL is quite small. First, we have the pattern type, which is just a test function, so 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, defines the `then` and `otherwise` methods, and 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 an 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
```

Note that the `body` parameter of the `match()` method requires a `MatchResult` as a return value, and immediately extracts its content. So 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 this is 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 avoid 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, there are still patterns missing for the left sides 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 }
```

Comparing values requires some type checking to ensure that the value is comparable. That's why we need reified generics in this case:

```
// greater than
inline fun <reified C : Comparable<C>> gt(value: C): Pattern<C> =
    { it > value }
```

```
// greater or equal
inline fun <reified C : Comparable<C>> ge(value: C): Pattern<C> =
    { it >= value }

// less than
inline fun <reified C : Comparable<C>> lt(value: C): Pattern<C> =
    { it < value }

// less or equal
inline fun <reified C : Comparable<C>> le(value: C): Pattern<C> =
    { it <= value }
```

For the predicates `all()`, `any()` and `none()` on `Iterable`s, 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) }
```

To capture values, we need a class that implements `Pattern<T>`, which 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>()
```

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, but also stores the value. On the right side you can read the value from the predefined `val`. The first syntax example demonstrates the usage:

```
val result = match(p) {
    ...
    val ageCapture = capture<Int>()
    person(+"Andy", !+"Miller", ageCapture) then
        { "Some other Andy of age ${ageCapture.value}." }
    ...
}
```

Now the only pattern missing is the one for decomposing a data class, but unfortunately it is not possible to write code to handle all data classes at once in a typesafe manner. So we are forced to write a pattern for each data class we want to use in a pattern, but this is easy to do:

```
fun person(
    firstName: Pattern<String> = any(),
    lastName: Pattern<String> = any(),
    age: Pattern<Int> = any()
): Pattern<Person?> = {
    when (it) {
        null -> false
        else -> firstName(it.firstName) &&
                lastName(it.lastName) &&
                age(it.age)
    }
}
```

Especially for data classes with many arguments, defining `any()` as the default pattern for all arguments is very useful, as it allows you to invoke the pattern for the data class with named arguments, and ignore the arguments 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.

Of course, you can write many more patterns, but the DSL is already functional as it is. Despite the complexity of the topic, it wasn't too difficult to come up with a fairly usable DSL that demonstrates the power and expressiveness of Kotlin.

11.4. Conclusion

Writing good hybrid DSLs can be a complex task, requiring careful consideration and integration of different language features, as well as careful testing. Sometimes all the language features you need won't work together seamlessly, and in such cases it may be better to stick to a single DSL type instead, which may feel a bit boring but ensures that the DSL works reliably and predictably.

However, a well-designed hybrid DSL can combine different language features in a way that feels intuitive and organic. By exploiting the strengths of various DSL types and carefully designing their integration, you can create very powerful and expressive DSLs. Because there are so many degrees of freedom, it is hard to get this right, so hybrid DSLs usually require extensive testing to cover 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 steeper learning curve
- It can be difficult to control edge cases
- Higher maintenance effort required

- Java interoperability can be challenging

[1] MockK: https://mockk.io/#dsl-examples
[2] Kotlin Documentation, Type Safe Builders: https://kotlinlang.org/docs/type-safe-builders.html

Part III - Supplemental Topics

Chapter 12. Code Generation for DSLs

> 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 you are prototyping a DSL, and you find a nice, expressive syntax, but it turns out that it would take a lot of boilerplate code to make it work. A common reason is combinatorial explosion, or you need many classes that follow the same pattern (e.g., tuple classes or fixed-length vectors). In such cases, you might consider using code generation.

A popular library for generating Kotlin code is KotlinPoet[1]. If you are also working with Java, you may also want to check out its sister project JavaPoet[2]. There is even some interoperability between them.

12.1. 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 {
    val amount: Double
}

// base units
data class Second(override val amount: Double) : Quantity
data class Meter(override val amount: Double) : Quantity
data class Kilogram(override val amount: Double) : Quantity

// derived units
data class SquareMeter(override val amount: Double) : Quantity
data class CubicMeter(override val amount: Double) : Quantity
data class MeterPerSecond(override val amount: Double) : Quantity
data class MeterPerSecondSquared(override val amount: Double) : Quantity
data class Newton(override val amount: Double) : Quantity
data class Joule(override val amount: Double) : Quantity
data class Watt(override val amount: Double) : Quantity
data class Pascal(override val amount: Double) : Quantity
```

Representing Quantities and Currency Amounts

> You should always consider whether the precision provided by `Double` is sufficient for the kind of calculations required by your application, and switch to e.g. `BigDecimal` if not.

At this point, we can't even add two quantities together. It would be nice if we could write this function only once in `Quantity`, but of course only quantities of the same unit can be added. There are techniques to make this safe, but they would require the use of generics in Kotlin.

Instead, we will use KotlinPoet[1]. You will need to add a dependency to your project, e.g:

Gradle (.kts)

```
implementation("com.squareup:kotlinpoet:1.18.0")
```

First, we will generate some extension functions for the missing addition:

```
private fun makeAddition(kClass: KClass<out Quantity>) =
    FunSpec.builder("plus")
        .addModifiers(KModifier.OPERATOR)
        .receiver(kClass)
        .returns(kClass)
        .addParameter("that", kClass)
        .addStatement("return copy(amount = this.amount + that.amount)")
        .build()

fun makeAdditions() =
    Quantity::class.sealedSubclasses.map { makeAddition(it) }
```

As you can see, KotlinPoet relies heavily on the builder pattern. In the `makeAdditions()` method, we take advantage of the fact that a sealed class - here `Quantity` - knows about its subclasses. As a result, we get a list of `FunSpec` instances representing functions, constructors, getters or setters. You can just print the contents to the console to see what the code would look like. Later we will add these instances to a `FileSpec` that can be written to the file system. The generated functions will look like this:

```
public operator fun Second.plus(that: Second) =
   copy(amount = this.amount + that.amount)
```

The next operations follow pretty much the same pattern. They define subtraction, negation, and scalar multiplication:

```
private fun makeSubtraction(kClass: KClass<out Quantity>) =
    FunSpec.builder("minus")
        .addModifiers(KModifier.OPERATOR)
        .receiver(kClass)
        .returns(kClass)
        .addParameter("that", kClass)
        .addStatement("return copy(amount = this.amount - that.amount)")
        .build()

private fun makeNegation(kClass: KClass<out Quantity>) =
    FunSpec.builder("unaryMinus")
        .addModifiers(KModifier.OPERATOR)
        .receiver(kClass)
        .returns(kClass)
        .addStatement("return copy(amount = -this.amount)")
        .build()

private fun makeScalarMultiplication(kClass: KClass<out Quantity>) =
    FunSpec.builder("times")
        .addModifiers(KModifier.OPERATOR)
        .receiver(Double::class)
        .returns(kClass)
        .addParameter("that", kClass)
        .addStatement("return that.copy(amount = this * that.amount)")
        .build()

fun makeSubtractions() =
    Quantity::class.sealedSubclasses.map { makeSubtraction(it) }

fun makeNegations() =
    Quantity::class.sealedSubclasses.map { makeNegation(it) }

fun makeScalarMultiplications() =
    Quantity::class.sealedSubclasses.map { makeScalarMultiplication(it) }
```

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, e.g. we can define not only `5.0.s` to get `Second(5.0)`, but also `1.0.min` to get `Seconds(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. It is obvious that we need additional information to write the conversions for a certain unit: We need 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.
)
```

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 look 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) }
```

In case you were wondering about the `(u, k, f)` part: This is the destructuring syntax, which works e.g. for `Pair`, `Triple` and data classes. Here is an example of a generated pair of transformations:

```
public val Double.kJ: Joule
  get() = Joule(this * 1000.0)

public val Joule.kJ: Double
  get() = this.amount / 1000.0
```

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 the order of N^2 (we won't implement all possible combinations, but 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),
```

```
    // etc.
)
```

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))
        }
    }
```

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)

// etc.
```

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. For convenience, I have added two extension functions to `FileSpec` that allow you to add multiple properties or functions at once:

```
fun main() {
    FileSpec.builder("creativeDSLs.chapter_12.units", "generated")
        .addProperties(makeQuantityToAmounts())
        .addProperties(makeAmountToQuantities())
        .addFunctions(makeAdditions())
        .addFunctions(makeSubtractions())
        .addFunctions(makeNegations())
        .addFunctions(makeScalarMultiplications())
        .addFunctions(makeMultiplications())
        .addFunctions(makeDivisions())
        .build()
        .writeTo(Path.of("./src/main/kotlin/"))
```

```
}

fun FileSpec.Builder.addProperties(properties: List<PropertySpec>) =
    this.also { properties.forEach { this.addProperty(it) } }

fun FileSpec.Builder.addFunctions(functions: List<FunSpec>) =
    this.also { functions.forEach { this.addFunction(it) } }
```

As you can see, 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 (e.g., 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, e.g:

```
val acceleration = 30.0.m_s / 1.0.s
val force = acceleration * 64.0.kg
val energy = force * 5.0.m
println("${energy.kJ} kiloJoule")
```

The example code is written in such a way that you manually generate the code via the `main()` 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.

12.2. 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[3].

Before deciding to write an annotation processor, it's important to understand how it works and what its limitations are. 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, reporting, or to provide tooling support. Often, a third module is added for testing purposes, because not only do you want to have unit tests for the processor classes, but you also need to verify that the processor works as intended when building

client code.

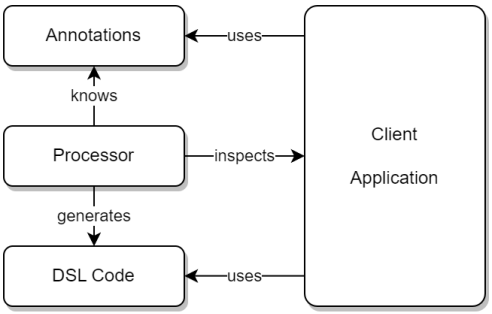

Figure 5. Generating DSL Code with KSP

KSP and Reflection

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 think about making the process of gathering information as easy as possible for the processor, e.g. by using annotations.

12.2.1. Designing the DSL and writing the annotations module

What would an annotation-based DSL for defining the relationship between physical quantities look like? We don't want to hardcode the annotation processor to use our specific quantity hierarchy, but to be more flexible. Therefore, we will assume that the there is a sealed interface that all quantities implement, and that the quantities are data classes with a single `Double` argument.

We have three types of operations:

- All classes implementing the top-level interface should support the basic operations like +, -, negation and scalar multiplication. Since a sealed interface knows its implementing classes, it is sufficient to mark only this interface with an annotation, which we will call `@QuantityOperations`.
- The conversions from and to `Double` can be expressed by `@Conversion` annotations on the corresponding class.
- The multiplication involves three classes, but it seems most convenient to annotate the resulting product class and refer to the factor classes in the annotation.

The resulting annotation-based DSL might look like this:

```
@QuantityOperations
sealed interface Quantity {
    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
```

Here are the definitions of the required annotations:

```
@Target(AnnotationTarget.CLASS)
@Retention(AnnotationRetention.RUNTIME)
annotation class QuantityOperations

@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<*>
)
```

Since the annotation processor needs to know about these annotations as well, it would be tied to your specific client code if you put the annotations there. That's why it's almost always a good idea to put the annotations in a separate module that allows the annotation processor to be used in other environments.

12.2.2. Writing the Annotation Processor

This book can only give a very high level overview of KSP. This means that the general structure of the KSP module is explained, but not the specifics of the KSP API. Please refer to KSP Documentation[4] for a more detailed discussion.

In your gradle build file, you will need dependencies to the KSP API (`com.google.devtools.ksp:symbol-processing-api`), to the KSP extensions of KotlinPoet (`com.squareup:kotlinpoet-ksp`), and to the module with your custom annotations.

Incremental Mode

KSP runs in incremental mode[5] by default, which means that it tries to avoid unnecessary re-processing 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`, e.g. 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, e.g. 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:

```
import com.google.devtools.ksp.processing.SymbolProcessor
import com.google.devtools.ksp.processing.SymbolProcessorEnvironment
import com.google.devtools.ksp.processing.SymbolProcessorProvider
```

```
class UnitsProcessorProvider : SymbolProcessorProvider {
    override fun create(
        environment: SymbolProcessorEnvironment
    ): SymbolProcessor = UnitsSymbolProcessor(
        codeGenerator = environment.codeGenerator,
        logger = environment.logger,
        options = environment.options
    )
}
```

This provider class must be registered in a text file called `SymbolProcessorProvider` located in the `resources/META-INF/services` folder. In this file you simply add a single line with the qualified name of the provider class.

The processor will find all classes annotated with `@Conversion`, `@QuantityOperations` and `@MultiplicationResult` and delegate code generation to the appropriate visitors:

```
class UnitsSymbolProcessor(
    private val codeGenerator: CodeGenerator,
    private val logger: KSPLogger,
    private val options: Map<String, String>
) : SymbolProcessor {

    override fun process(resolver: Resolver): List<KSAnnotated> {

        val conversionDeclarations = invokeVisitor(
            resolver,
            Conversion::class,
            ConversionVisitor(codeGenerator, logger)
        )

        val operationsDeclarations = invokeVisitor(
            resolver,
            QuantityOperations::class,
            OperationsVisitor(codeGenerator, logger)
        )

        val multiplicationDeclarations = invokeVisitor(
            resolver,
            MultiplicationResult::class,
            MultiplicationVisitor(codeGenerator, logger)
        )
```

```
        return listOf(
            conversionDeclarations,
            operationsDeclarations,
            multiplicationDeclarations
        ).flatten().distinct().filterNot { it.validate() }
    }

    private fun invokeVisitor(
        resolver: Resolver,
        annotation: KClass<*>,
        visitor: KSVisitorVoid
    ): List<KSClassDeclaration> =
        resolver.getSymbolsWithAnnotation(annotation.qualifiedName!!)
            .distinct()
            .filterIsInstance<KSClassDeclaration>()
            .toList()
            .onEach { it.accept(visitor, Unit) }
}
```

The first visitor provides conversion functions from and to `Double`:

```
class ConversionVisitor(
    private val codeGenerator: CodeGenerator,
    private val logger: KSPLogger
) : KSVisitorVoid() {

    @OptIn(KspExperimental::class)
    override fun visitClassDeclaration(
        classDeclaration: KSClassDeclaration,
        data: Unit
    ) {
        val shortName = classDeclaration.simpleName.getShortName()

        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 ->
```

```
                addProperty(
                    makeDoubleToQuantity(
                        conversion.derivedUnit,
                        classDeclaration.toClassName(),
                        conversion.factor
                    )
                )
                addProperty(
                    makeQuantityToDouble(
                        conversion.derivedUnit,
                        classDeclaration.toClassName(),
                        conversion.factor
                    )
                )
            }
            build()
        }

    fileSpec.writeTo(codeGenerator, false)
}

fun makeDoubleToQuantity(
    unit: String,
    className: ClassName,
    factor: Double
) = PropertySpec.builder(unit, className)
        .receiver(Double::class)
        .getter(
            FunSpec.getterBuilder()
                .addStatement("return %T(this * %L)", className, factor)
                .build()
        )
        .build()

fun makeQuantityToDouble(
    unit: String,
    className: ClassName,
    factor: Double
) = PropertySpec.builder(unit, Double::class)
        .receiver(className)
        .getter(
            FunSpec.getterBuilder()
                .addStatement("return this.component1() / %L", factor)
                .build()
```

```
            )
            .build()
}
```

The resulting methods are written in a separate file, e.g. for the quantity class `Meter`, they are located in `meterConversions.kt`.

Note that we don't rely on a specific argument name like `amount` in the quantity class, but rather access the value using the `component1()` method. This allows us to rename the argument in the client code without breaking the annotation processor.

The next visitor deals with basic operations such as adding and subtracting quantities:

```
class OperationsVisitor(
    private val codeGenerator: CodeGenerator,
    private val logger: KSPLogger
) : KSVisitorVoid() {

    override fun visitClassDeclaration(
        classDeclaration: KSClassDeclaration,
        data: Unit) {

        val shortName = classDeclaration.simpleName.getShortName()

        if (Modifier.SEALED !in classDeclaration.modifiers) {
            logger.error("Can't generator operations, " +
                "<$shortName> is not a sealed class.")
        }

        val subclasses: Sequence<ClassName> =
            classDeclaration.getSealedSubclasses().map { it.toClassName() }

        val fileSpec =
            FileSpec.builder(
                packageName = classDeclaration.packageName.asString(),
                fileName = shortName.lowercase() + "Operations"
            ).run {
                subclasses.forEach { subclass ->
                    addFunction(makeAddition(subclass))
                    addFunction(makeSubtraction(subclass))
                    addFunction(makeNegation(subclass))
                    addFunction(makeScalarMultiplication(subclass))
                }
                build()
```

```
            }

        fileSpec.writeTo(codeGenerator, false)
    }

    private fun makeAddition(className: ClassName) =
        FunSpec.builder("plus")
            .addModifiers(KModifier.OPERATOR)
            .receiver(className)
            .returns(className)
            .addParameter("that", className)
            .addStatement("return copy(this.component1() + that.component1())")
            .build()

    private fun makeSubtraction(className: ClassName) =
        FunSpec.builder("minus")
            .addModifiers(KModifier.OPERATOR)
            .receiver(className)
            .returns(className)
            .addParameter("that", className)
            .addStatement("return copy(this.component1() - that.component1())")
            .build()

    private fun makeNegation(className: ClassName) =
        FunSpec.builder("unaryMinus")
            .addModifiers(KModifier.OPERATOR)
            .receiver(className)
            .returns(className)
            .addStatement("return copy(-this.component1())")
            .build()

    private fun makeScalarMultiplication(className: ClassName) =
        FunSpec.builder("times")
            .addModifiers(KModifier.OPERATOR)
            .receiver(Double::class)
            .returns(className)
            .addParameter("that", className)
            .addStatement("return that.copy(this * that.component1())")
            .build()
}
```

In this case, we don't extract any information from the annotation, but rather from the class itself, which is assumed to be `sealed`, and thus allows us to enumerate its child classes.

The last visitor generates the code to multiply and divide quantities:

```
class MultiplicationVisitor(
    private val codeGenerator: CodeGenerator,
    private val logger: KSPLogger
) : 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
                val factor2 = args.first { arg ->
                    arg.name?.getShortName() == "factor2"
                }.value as KSType
                factor1.toClassName() to factor2.toClassName()
            }

        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.toString() != factor2.toString()) {
                        addFunctions(
                            factor1 = factor2,
                            factor2 = factor1,
                            result = classDeclaration.toClassName()
                        )
                    }
```

```
            }
            build()
        }

    fileSpec.writeTo(codeGenerator, false)
}

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.component1() * that.component1())", 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.component1() / that.component1())", out)
    .build()

private fun KSClassDeclaration.getAnnotations(
    annotationClass: KClass<*>
): List<KSAnnotation> =
    annotations.filter { it.shortName.getShortName() ==
```

```
annotationClass.simpleName }.toList()
}
```

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:

build.gradle.kts

```
plugins {
    ...
    id("com.google.devtools.ksp") version "2.0.0-1.0.21"
}
...
dependencies {
    ...
    ksp(/* reference to the KSP module */)
    ...
}
```

If the module is in a 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.

12.3. 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. In this case study, we will only discuss the annotation and the KSP visitor class, as the remaining parts are very similar to the previous one.

```
@Target(AnnotationTarget.CLASS)
@Retention(AnnotationRetention.RUNTIME)
annotation class DataClassPattern
```

```
...

private const val patternPackage = "creativeDSLs.chapter_12.patterns"

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()

        logger.warn("found $shortName")

        if (Modifier.DATA !in classDeclaration.modifiers) {
            logger.error("Can't generate pattern, " +
                "<$shortName> isn't a data class")
        }

        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, false)
    }

    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 ->
                "${param.name!!.getShortName()}" +
                "(it.${param.name!!.getShortName()})"
        })
        .endControlFlow()
        .endControlFlow()
        .build()

    private fun String.decap(): String =
        this.replaceFirstChar { it.lowercase(Locale.getDefault()) }

    private fun functionName(shortName: String) = shortName.decap()
        .let { decap ->
            if (decap == shortName) "${decap}Pattern" else decap
        }
}
```

The `@DataClassPattern` annotation can be very simple, it's just a marker for the KSP. 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)
  }
}
```

This function can be used in a `match` block as described in the last chapter.

It should be mentioned that our annotation processor is not perfect, e.g. it can't handle data classes with generic fields.

12.4. Conclusion

The decision to use code generation requires careful consideration because of the effort required. However, this technique allows you to implement DSLs that would just be too much overhead without it. And with libraries like KotlinPoet[1], it is quite intuitive to generate the code you want. Kotlin-Poet is itself a nice example of a real-world DSL, and will be explored as such in the final chapter.

Using code generation in conjunction with annotation processors like KSP can produce flexible, powerful, and well-integrated DSLs that wouldn't otherwise be possible.

12.4.1. Preferable Use Cases

- 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 combinatorial explosion
- Intuitive libraries such as Kotlin-Poet are available

Cons

- Requires some up-front effort and setup
- Strong dependency on the library used
- Longer build times if generation is done for every build
- Code can get out of sync if built on demand only
- Bug fixing can be challenging

[1] KotlinPoet: https://square.github.io/kotlinpoet
[2] JavaPoet: https://github.com/square/javapoet
[3] Kotlin Symbol Processing: https://kotlinlang.org/docs/ksp-overview.html
[4] KSP - Documentation: https://kotlinlang.org/docs/ksp-quickstart.html
[5] KSP - Incremental: https://kotlinlang.org/docs/ksp-incremental.html
[6] Service Provider Interface: https://docs.oracle.com/javase/tutorial/sound/SPI-intro.html

Chapter 13. Java Interoperability

> Translation is the art of failure.
>
> — Umberto Eco

Using a Kotlin DSL from Java can be challenging. While some Kotlin features, such as operator overloading, simply aren't available, there are ways to make other features more accessible from Java, and to create a DSL that is useful and convenient in both languages. Fortunately, Kotlin has a number of built-in features to improve its interoperability with Java, which are described in Kotlin Documentation: Calling Kotlin from Java[1].

This chapter discusses Java interoperability from a purely DSL design perspective, presenting the relevant built-in features and useful techniques for solving DSL-specific problems. Note that retrofitting an existing DSL for Java interoperability is more expensive than designing for it from the beginning.

13.1. 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. Of course, sometimes additional annotations may be needed, such as `@JvmStatic` 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")
}
```

Calling these members is now trivial:

Java code

```
public class CallStrangeNames {
```

```
    public static void main(String[] args) {
        NamingTest.checkThisOut();
        NamingTest.instanceOf();
    }
}
```

13.1.1. 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, such as when there are multiple methods with 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 methods using a hashing algorithm that takes into account the package name, class name, and method name. This creates a unique name for each method, which prevents naming conflicts in the bytecode.

Kotlin users don't need to be aware of this behavior, as calls to the mangled methods are automatically translated correctly. However, if Java users want to call a mangled method, they would have to use the strange mangled name. To avoid this problem, you can explicitly name the method on the JVM using 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") }
```

From Java, you can call the example methods by their JVM names and with `double` arguments, e.g. `displayKm(23.0);` and `displayMiles(42.3);`.

13.2. Package Level Definitions of Functions and Variables

Java does not allow you to define functions and variables outside of classes, so Kotlin puts them as static functions and variables in an artificial class. The default name of this class

is derived from the filename, including the `kt` ending, and follows the upper-camel-case naming convention for classes. Consider the following example:

utils.kt

```
package com.acme

fun someFunction() {
    ...
}
```

The function could be called from Java as `com.acme.UtilsKt.someFunction()`. Often, you would prefer to use another class name for your DSL, e.g. `Utils` instead of `UtilsKt`. This can be easily achieved by including a `JvmName` annotation:

utils.kt

```
@file:JvmName("Utils")
package com.acme

fun someFunction() {
    ...
}
```

It is even possible to map the contents of multiple Kotlin files to the same Java class, but then an additional `@file:JvmMultifileClass` annotation is needed in every file.

13.3. Generate Overloaded Methods

Java doesn't have default arguments, they are "simulated" by having several overloaded methods. The Kotlin compiler can generate such overloaded methods when requested via 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")
}
```

In a Java class, you can see now four methods of this name. If 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"
```

As you can see, you can either call the method with all arguments, or you can leave the right-most arguments with default values off. Of course, you have always to specify the argument `d`, which has no default.

You can also annotate constructors with `@JvmOverloads`.

13.4. Accessing Fields

In Kotlin, fields are exposed as properties, and even if it looks like you are accessing a field with a call like `somePerson.name`, behind the scenes you are accessing a getter or setter of this property. Of course, you can use these getters and setters from Java as well, e.g. by using `somePerson.getName()`, but if you want to allow direct field access, you need to use the `@JvmField` annotation, e.g. like this:

```
data class Person(@JvmField val name: String, @JvmField val age: Int)
```

Now you can call the fields directly, as in Kotlin.

13.5. Generics

A common problem is that due to its declaration-side variance, Kotlin often generates generic signatures with wildcards like `List<? extends String>` on the JVM. Such types can be awkward to use from the Java side, can prevent the use of certain Java libraries (such as Dagger[2]), or even lead to cryptic compile-time errors. The solution is to annotate the offending type with `@JvmSuppressWildcards`, so that you get the type signature like `List<String>` on the JVM.

In some cases, you may have the opposite problem, where it would be more convenient to have wildcards on the JVM when the Kotlin compiler doesn't produce them. In this case, you can use the `@JvmWildcards` annotation instead.

13.5.1. Calling Functions with Reified Type Parameters

I am afraid I have some bad news for you: Java has no inlining mechanism, and without inlining, the resolution of reified type parameters simply doesn't work. As a consequence, you can't call such functions from Java, not even via reflection.

A workaround is to write a version of the function with an explicit class parameter:

```
inline fun <reified T> tellType(list: List<T>) {
    println(T::class.qualifiedName)
}

// for Java calls
fun <T: Any> tellTypeJava(list: List<T>, clazz: Class<T>) {
    println(clazz.kotlin.qualifiedName)
}
```

You can call the second function as usual from Java, e.g. `tellTypeJava(List.of(1,2,3), Integer.class);`.

This approach will work for many use cases, but it should be noted that a reified type contains information about its own type parameters, while a class parameter just denotes a raw type. If this type information is needed, our simplistic approach won't work. It is difficult to give a general solution for the more complicated cases, but replacing the class parameter with e.g. `TypeToken` (from either Guava[3] or Gson[4]) might help.

13.6. Checked Exceptions

Kotlin doesn't have the concept of "checked exceptions", but if a function that might throw such an exception is called from Java, the Java compiler expects that the exception is declared in the function signature. In order to avoid problems in such cases, you can give the Kotlin compiler a hint to add a checked exception to the function signature in the byte-code by annotating the function with `@Throws(SomeCheckedException::class)`.

13.7. 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 knowledge of the underlying architectural concepts, such as continuations. It's usually a better solution

to provide dedicated functions for Java access, e.g. by wrapping the coroutine in a `CompletableFuture`, which is much easier to handle in Java:

```
// hidden from Java
@JvmSynthetic
suspend fun getStuff(): String {
    ...
}

private val scope = CoroutineScope(EmptyCoroutineContext)

// dedicated Java API
fun getStuffForJava(): CompletableFuture<String> =
    scope.future { getStuff() }
```

13.8. Conclusion

Writing DSLs often requires the use of advanced language features, so it's no surprise that calling this code from Java can be challenging, and that the Kotlin compiler may need some pointers for good Java interoperability. If calling your DSL from Java is a requirement, you should consider it in your design from the beginning. In particular, writing tests not only in Kotlin but also in Java can help avoid problems down the road. Most interoperability problems are easy to fix, often the hard part is figuring out what's going wrong and knowing what language features are available to you in these situations.

[1] Kotlin Documentation, Calling Kotlin from Java: https://kotlinlang.org/docs/java-to-kotlin-interop.html

[2] Dagger: https://dagger.dev

[3] Guava: https://github.com/google/guava

[4] Gson: https://github.com/google/gson

Chapter 14. Real-World DSL Examples

> Few things are harder to put up with than the annoyance of a good example.
>
> — Mark Twain

In this chapter, we will look at some interesting and successful DSLs, characterize them, look at interesting details, and maybe even suggest some improvements. I encourage you to take a closer look at these projects and learn how they build coherent DSLs with sometimes seemingly "impossible" features.

14.1. KotlinPoet

KotlinPoet[1] is a source code generator for Kotlin, and the sister project to JavaPoet. We used it in Chapter 12 to generate sources for operations on physical quantities, for example.

The library closely follows the syntax of JavaPoet, which is written in Java, and I think this was the right decision. The builder pattern works quite well for this purpose. This is the introductory example on the KotlinPoet homepage:

https://square.github.io/kotlinpoet Hello World Example

```
class Greeter(val name: String) {
  fun greet() {
    println("""Hello, $name""")
  }
}

fun main(vararg args: String) {
  Greeter(args[0]).greet()
}
```

https://square.github.io/kotlinpoet Code Generation of the Example

```
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)
```

If KotlinPoet were a standalone library, I would rather use a Loan Pattern DSL instead:

```
val greeterClass = ClassName("", "Greeter")
val exampleFile = file("", "HelloWorld") {
    clazz("Greeter") {
        primaryConstructor = constructor {
            parameter("name", String::class)
        }
        property("name", String::class) {
            initializer("name")
        }
        function("greet") {
            statement("println(%P)", "Hello, \$name")
        }
    }
    function("main") {
        parameter("args", String::class, VARARG)
        statement("%T(args[0]).greet()", greeterClass)
    }
```

```
}
```

I think the Loan Pattern style is easier to read because it shows the nested structures more clearly. As mentioned above, there were good reasons to stick with the builder pattern approach, but it would be nice to have the option to use something closer to my suggestion.

14.2. Gradle .kts

Gradle[2] is an amazing build system. Unlike descriptive approaches like the XML-based Apache Maven[3], the build process is "programmable", which gives the user a lot of flexibility.

However, the original Gradle implementation was written in Groovy - at the time probably the best choice for writing an expressive DSL in the Java ecosystem. Unfortunately, the Groovy language comes with its own set of problems. It is a scripted language with a weak type system, and 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 decided to provide an alternative DSL based on Kotlin script (.kts). Obviously, the new DSL should look very similar to the old Groovy style to make the transition as smooth as possible. I think the Kotlin DSL was a resounding success in this regard. Here is a comparison, taken from the Gradle User Guide: Migrating build logic from Groovy to Kotlin[4]:

Groovy

```
plugins {
    id 'java-library'
}
dependencies {
    implementation 'com.example:lib:1.1'
    runtimeOnly 'com.example:runtime:1.0'
    testImplementation('com.example:test-support:1.3') {
        exclude(module: 'junit')
    }
    testRuntimeOnly 'com.example:test-junit-jupiter-runtime:1.3'
}
```

Kotlin Script

```
plugins {
    `java-library`
```

```
}
dependencies {
    implementation("com.example:lib:1.1")
    runtimeOnly("com.example:runtime:1.0")
    testImplementation("com.example:test-support:1.3") {
        exclude(module = "junit")
    }
    testRuntimeOnly("com.example:test-junit-jupiter-runtime:1.3")
}
```

At first glance, it is hard to tell which is which. There are parts of the DSLs that deviate more, but then it looks more like a conscious decision to clean up and standardize the syntax than a limitation of the language.

The Kotlin script DSL itself mainly uses the Loan pattern. There are two notations for defining dependencies: The example shows the "string notation", which is a string parsing DSL. Alternatively, you can use the "map notation", which looks like this: `implementation(group = "com.example", name = "lib", version = "1.1")`.

In my opinion, Gradle is a good example of how Kotlin DSLs can help modernize an already established and successful solution without causing major disruptions.

14.3. Kotest

Kotest[5] is a widely used testing framework for Kotlin, and it is interesting to compare it with the most popular Java testing framework, which is JUnit[6].

First of all, Kotest provides a plethora of different testing styles[7] (called "specs"), and is much less opinionated in this regard than JUnit. For the sake of brevity, I'll use `StringSpec` for the following examples:

https://kotest.io/docs/framework/testing-styles.html#string-spec

```
class MyTests : StringSpec({
    "strings.length should return size of string" {
        "hello".length shouldBe 5
    }
})
```

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:

```
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'
```

Of course, you can do something similar in JUnit by using the `@ParameterizedTest` annotation instead of `@Test`, and if you need even more flexibility, there is a `DynamicTest.dynamicTest()` method that takes a lambda argument. The difference, however, is that Kotest doesn't require special constructs in such cases, but allows the user to take advantage of existing language features. Obviously, doing more with less helps to reduce the cognitive load while learning to use the framework.

Kotest can be categorized as quasi-lingual DSL, where many parts mimic natural language:

https://kotest.io/docs/framework/exceptions.html

```
val exception = shouldThrow<IllegalAccessException> {
   // code in here that you expect to throw an IllegalAccessException
}

exception.message should startWith("Something went wrong")
```

14.4. MockK

Mocking, spying, stubbing, and argument capture are essential for writing concise and expressive unit tests. In Java, libraries such as Mockito[8] are very popular, but face some challenges when dealing with Kotlin language features such as objects, top-level functions, and coroutines.

MockK is an amazing mocking framework for Kotlin, and features a very intuitive hybrid DSL. Here is a very simple example from the documentation:

https://mockk.io/

```
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)
```

The library has already been mentioned in Chapter 12 as an example of a quasi-lingual DSL, and blends well with quasi-lingual testing frameworks such as Kotest.

14.5. better-parse

Better-parse is the parser-combinator library we used in Chapter 9.2.2. It features a succinct hybrid DSL, which gives you a lot of flexibility. Here is one of the example parsers of the project:

https://github.com/h0tk3y/better-parse/blob/master/demo/demo-jvm/src/main/kotlin/com/example/BooleanExpression.kt

```
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))
}
```

The grammar is defined within an object as a sequence of token definitions and parsers that build upon each other. An 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, making it much easier for the library to work with them.

An interesting problem is how to handle recursive parser definitions. In the example in Chapter 9, a "molecular part" could be either an "element" or a "group", but a group itself consists of parts. The library solves this in an elegant way by allowing to refer to parsers 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
    ...
}
```

Calling `parser(this::part)` allows you to "break the cycle". I suspect that this is one reason - besides performance considerations - why the DSL uses an object as its context, rather than the more common trailing lambda syntax. While there are certainly ways to make a

recursive definition work in Loan Pattern DSLs, this is clearly a more straightforward solution. This is a good example of why creativity and out-of-the-box thinking are so important when writing DSLs.

14.6. Konform-kt

Konform-kt[9] is a library for data validation. The resulting validation object contains a list of issues of the given data structure, allowing each problem to be precisely located.

The DSL relies heavily on the Loan Pattern, allows different types of nesting to "drill down" into the data structure, and provides many predefined constraints. Here is an example from the documentation:

from https://github.com/konform-kt/konform

```
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 {
            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)
        }
    }
}
```

Even without knowing the library, you can probably tell what conditions will be checked and what output you can 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, but you don't have to, as there are already sensible default messages available. Even though Konform-kt doesn't use any features we haven't already covered, there is a lot to learn from how the DSL presents itself to the user and what makes it pleasant to use.

14.7. Arrow

Arrow[10] is a huge collection of features that promote functional programming in Kotlin, such as working with coroutines, error handling, immutable types, and much more. There is far too much content to cover here, but to give you a very basic example, here is how the library improves working with nullable types. Very often you see Kotlin code like this, which is hard to read due to various null checks:

https://arrow-kt.io/learn/typed-errors/nullable-and-option/

```
fun sendEmail(params: QueryParameters): SendResult? =
  params.userId()?.let { userId ->
    findUserById(userId)?.email?.let { email ->
```

```
      sendEmail(email)
    }
  }
```

Arrow provides the `nullable` DSL, which avoids nesting, and simplifies null checking:

https://arrow-kt.io/learn/typed-errors/nullable-and-option/

```
fun sendEmail(params: QueryParameters): SendResult? = nullable {
  val userId = ensureNotNull(params.userId())
  val user = findUserById(userId).bind()
  val email = user.email.bind()
  sendEmail(email)
}
```

I think you will agree that this code is much easier to read. But this is just an appetizer, there is much more to explore, both for your daily coding and as inspiration for writing idiomatic DSLs.

14.8. Conclusion

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 lack an engaging and enjoyable user experience. Such DSLs often rely heavily on a limited set of language features and miss the opportunity for a more exploratory approach, resulting in a dull and uninspired feel. The examples presented in this chapter should inspire you to strive for more, for DSLs that are creative, intuitive, and fun to use.

[1] KotlinPoet: https://square.github.io/kotlinpoet
[2] Gradle: https://gradle.org
[3] Apache Maven: https://maven.apache.org
[4] Gradle Migration: https://docs.gradle.org/current/userguide/migrating_from_groovy_to_kotlin_dsl.html
[5] Kotest: https://kotest.io
[6] JUnit: https://junit.org/junit5
[7] Kotest - Testing Styles: https://kotest.io/docs/framework/testing-styles.html
[8] Mockito: https://site.mockito.org/
[9] Konform-kt: https://github.com/konform-kt/konform
[10] Arrow: https://arrow-kt.io

Back Matter

Epilogue

> The covers of this book are too far apart.
>
> — Ambrose Bierce

Congratulations on completing Creative DSLs in Kotlin! Throughout this book, you have explored the many ways in which Kotlin's language features and design make 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 that this book has given you the knowledge and inspiration you need to create your own DSLs in Kotlin, and that you will continue to explore the many possibilities offered by this powerful and 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, and mastering these skills requires persistence and a willingness to learn and grow. But mastering these skills is very rewarding, and makes writing software so much more fun.

I sincerely hope that this book has been able to convey at least a little of the excitement I had writing DSLs and researching the topic myself, and I wish that you will experience the same sense of joy and wonder along your own path. Thank you for joining me on this journey!

Index