# Creative DSLs in Kotlin

Daniel Gronau

# Creative DSLs in Kotlin

Daniel Gronau

Version 0.1

# Table of Contents

# Creative DSLs in Kotlin

Published by:

Daniel Gronau
Ramsiner Str. 35
06792 Sandersdorf-Brehna
Germany

Email: creativeDSLs@protonmail.com

Distribution: epubli – a service of the neopubli GmbH, Berlin, Germany
eBook ISBN: XXX-X-XXX-XXXXX-X
Paperback ISBN: XXX-X-XXX-XXXXX-X

# Dedication

— To my parents Eva and Wolfgang Gronau —

# Acknowledgements

Writing a book isn't a walk in the park, but is being passionate about it makes it much easier. That's why I am grateful to all the people that helped me to develop and grow this passion: the excellent teachers throughout my life, both formal instructors at university and informal mentors among my colleagues and friends. They nurtured my curiosity, helped me to understand new concepts, and encouraged me to develop new skills. Without their guidance and support, this book would not have been possible. …

And last but not least, I want to thank all the heroes who provide free knowledge and awesome open source tools - like the Kotlin language itself - who make the world a better place.

# Preface

## 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 available on this topic, 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 trade-offs that were made
- They do not attempt a categorization of the different types of DSLs, or a comparison of their relative strengths and weaknesses
- They don't take into account that Kotlin and Java code often coexists, and ignore the resulting interoperability issues occurring when a Kotlin DSL must 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 discussing the design process, the requirements and constraints that apply, the trade-offs that must be made, and the challenges that arise.

To achieve this, the book progresses from common examples to more challenging and complex cases, and finishing with a discussion of real-world DSLs that have been successful in practice. In addition to providing practical guidance, it is attempted to find a comprehensive terminology for classifying DSLs, similar to the way the design patterns movement established a common language for coding solutions.

Overall, the goal of the book is 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 very suitable for those who are new to Kotlin or Java. While this book does contain 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 yet familiar with Kotlin or Java, it is recommended to 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-scale 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 gives a general overview about the usage of DSL and underlying design principles.

*Chapter 2, Requirements Analysis*, helps to define requirements and expectations of a DSL design before implementing it. Jumping into action without defining your goals may end in a nasty surprise, as there are more things to consider than you may think.

*Chapter 3, Writing a DSL*, suggests a simple methodology for implementing a DSL. The given steps help to avoid tunnel vision, to get unstuck when hitting a wall, and to 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 might have never used one of the more obscure language features, like extension properties.

*Chapter 5, Algebraic DSLs*, introduces the first category of DSLs, which supports number-like behavior by employing operator overloading and infix functions. This is a quite straightforward DSL type, but mastering it will come handy in many other contexts.

*Chapter 6, Builder Pattern DSLs*, covers the well known Builder Pattern you probably already know from Java, and shows some variations attempting to make builders safer to use.

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

*Chapter 8, Modeling State Transitions in DSLs*, explores different strategies to model states or stages of elements in your DSL, e.g. when you want to construct an object in stages, or model a finite state machine. There are interesting alternatives to the naive "one state, one class" approach.

*Chapter 9, String-Parsing DSLs*, gives you an overview about DSLs embedded in strings, which can be seen as an edge case between internal and external DSLs. The chapter contains - for educational purposes - a manually written parser, but also shows how the same parser can be written using a parser-combinator library.

*Chapter 10, Annotation-based DSLs*, discusses the use of annotations for a DSL, which allows to add new meaning or additional behavior to existing code. The example code shows also how to use reflection behind the scenes in order to bring the annotations to life.

*Chapter 11, Hybrid DSLs*, shows how to deal with situations where one of the already covered DSL categories alone just doesn’t cut it, and you have to combine them. Mixing language features comes with its own set of challenges, and this chapter will prepare you to overcome them.

*Chapter 12, Code Generation for DSLs*, discusses how code generation can help you when writing the DSL manually is just too much boilerplate. At first, this task may sound complicated, but the example shows that it isn’t really something to be afraid of.

*Chapter 13, Java interoperability*, gives an overview of the issues that may arise when Kotlin DSLs are used from Java code, and gives some advice on how to solve these issues.

*Chapter 14, Real-World DSL Examples*, discusses some well-known and battle-tested DSLs, in order to learn from it. In my opinion, toy-examples are not enough to learn good DSL design. It is important to see how DSLs deal with real-world issues, how they have to compromise and still manage to deliver a great user experience.

## Prerequisites

To try out the code samples in this book or from the associated project at github.com/creativeDsls, 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 Kotlin version 1.8 and assumes Java language level 11 unless otherwise specified.

It is also recommended that you familiarize yourself with the Kotlin Documentation, 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.

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.

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-standing tradition in the Java ecosystem. Beside 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 format. For PDF and eBook generation, I used the AsciidocFX editor. The main writing and programming tool was IntelliJ IDEA by JetBrains, using the Asciidoctor plugin. The diagrams were made using the ditaa library.

I used ChatGPT by OpenAI as a writing assistant. As a non-native speaker, it is difficult to avoid grammar mistakes and to find to a natural writing style, and I'm thankful that OpenAI granted public access for testing this incredible technology.

## Feedback

Please do not hesitate to contact me if you find any errors or have suggestions for improvement. Your feedback is very valuable to me and will help to improve this book for future readers. Thank you in advance for taking the time to let me know your thoughts.

To give feedback, e-mail me at creativeDsls@protonmail.com.

# Part I - Writing DSLs

# Chapter 1. Introduction

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

Domain-specific languages are often used to model business logic in specific domains, such as financial trades 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. Additionally, DSLs can be used for performing calculations or simulations. They are also commonly used in testing and logging, as well as for code generation and even for writing DSLs themselves. In all of these cases, DSLs provide a specialized language that is tailored to the needs of a particular domain, allowing developers to more easily express concepts and perform tasks in a way that is natural and intuitive.

## 1.2. Internal and External DSLs

This books discusses DSLs which are embedded into Kotlin, and are therefore limited to the existing expressions of the language. These are *internal DSLs* (or "embedded DSLs"). One major advantage is that these DSL don't need special treatment, there are no extra steps needed like reading and parsing files, so they fit seamless with the rest of the code.

One disadvantage of internal DSLs is the limitation of the syntax inherited by the host language. Kotlin allows for great freedom in DSL design, especially compared to Java. Nevertheless, 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 leads to a larger overhead compared to internal DSLs. However, writing external DSLs has become much easier recently by new libraries and frameworks, and improved tooling.

An edge case are internal DSLs realized entirely inside strings, the way e.g. regular expressions work: While 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 when 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. To avoid illegal conversions and provide a pleasing syntax, it might be necessary to write a significant amount of boilerplate code to handle all the possible combinations. In such cases, code generation can be a useful tool to help automate the creation of this boilerplate code and make it easier to manage the complexity of the DSL, while keeping the DSL expressive and maintainable.

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

## 1.4. Common use cases for DSLs

While it is difficult to clearly distinguish between the various types of use cases, a general classification is still helpful. In particular, every DSL category has use cases 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 to automate repetitive tasks and to build complex systems.
  - **Testing:** DSLs can be used to define and execute tests in a domain-specific way, 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 way.
  - **Workflow Orchestration:** DSLs can be employed to define and manage complex workflows or business processes.
- **Runtime Behaviour**
  - **Data Creation and Initialization:** DSLs can be used to define and construct data structures in a domain-specific way, 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 the behavior of the system, and to reason about it.
  - **Executing 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 therefore simplifying further processing.
    - **Monitoring:** DSLs can be used to observe the system status and to measure parameters like system performance or memory usage.
    - **Reporting and Analytics:** DSLs can be used to define queries, aggregations, and transformations for reporting and analytics purposes.
- **Specific Applications**
    - **Natural Language Processing:** DSLs can be utilized 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 challenges and requirements, and different DSL categories may be better suited to modeling certain types of use cases over 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 stick to a certain style, similar tasks should require a similar syntax. The behavior of the DSL should be logical and predictable.
- **Coverage:** The DSL needs to 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 isn't enough to make a DSL concise, it should also take user expectations into account, and follow the *Principle of Least Surprise*. The error handling should be comprehensive and provide clear and informative error messages that help users identify and fix issues 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 in a way that allows easy extension and customization. Users should be able to add new functionality or modify existing behavior without significant effort or disrupting 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 might be even necessary to provide a way to *bypass* DSL functionality, in order to allow access from other languages like 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. However, in practice, a DSL project can fail if you overlook the other principles or are unable to reach a good compromise between the sometimes conflicting requirements. Ultimately, a successful DSL design requires a holistic approach that takes into account 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 such as IntelliJ IDEA. From its inception, Kotlin was designed with a focus on readability, practicality, safety, and interoperability.

In comparison 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 fluent and intuitive API that is easy to use and understand, and lend themselves naturally to this coding style. We will take a closer look at the most important features in chapter 4.

In Kotlin, it is often easy to add "miniature DSLs" on the fly in existing code. That means the boundary between "everyday code" and DSLs isn’t clear-cut, which seems to be a conscious 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.

# Chapter 2. Requirements Analysis

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 extendable.

It is important to establish a clear understanding of the scope and purpose of your DSL among all stakeholders. This includes determining the target audience, and how the DSL will be used. Having everyone on the same page minimizes the risk that important details have been overlooked. However, this shouldn’t be a long-winded and boring process. To simplify this task, this chapter is basically a check-list of decisions to make before you start implementing.

## 2.1. Defining the Problem Domain

The first question to answer is obviously which domain to tackle. For small domains and areas with existing standards, this question is usually easy to answer. For larger domains it can be tempting to over-generalize. Try to be specific. Sometimes, the deliberate decision to drop some lesser 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 specify non-goals explicitly. Also, don’t forget to specify the target platform, the language level etc.

Generally, it is a good practice to organize your DSL in layers, so that the lower layers don’t rely on the higher ones, and can therefore be used independently. E.g. if you design a DSL for linear algebra, the DSL part for working with matrices could be independent of the rest of the system, which allows to use this part in other contexts as well.

When the application already employs other DSLs, it should be considered whether a parallel use of the new DSL would be confusing or if it could yield unexpected results. In some cases it might be smart to imitate the syntax of an existing DSL in order to facilitate the experience the users already have. A real-world example for this approach is the KotlinPoet library mimicking its JavaPoet sister project.

## 2.2. Research

After specifying the problem domain, you should check out existing DSLs. You might be surprised by the plethora of existing solutions, and of course it makes little sense to reinvent the wheel. Even if the results don’t fit your use case perfectly, they might serve as a base implementation, or as a source of inspiration.

There is an ongoing debate of how much software should depend on libraries, and you should be aware of related problems like the dreaded "dependency hell". On the other hand, you shouldn’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. A decision whether to use a library should be always deliberate, and the decision process should be documented.

If there is no existing DSL for your problem, you might also investigate whether using a DSL generator could be an option for you. Using such libraries can make designing a DSL a breeze. This book covers AutoDSL for Kotlin to give you an impression of this approach.

## 2.3. Specifying the Output of the DSL

In most cases it is rather obvious which kind of output is expected. Sometimes you have a choice of executing some action (like a query) directly, or to create objects being able to perform that action instead. Most of the time, creating objects should be preferred. Usually this approach is easier to test and to debug, and sometimes it is convenient to be able to create these instances by other means.

Another potential question is whether to generate 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 (like Hexagonal Architecture, Onion Architecture etc.), but generally you should try to use non-essential external APIs only at the boundaries of the application, and DSLs should follow this guideline as well.

## 2.4. Syntactical Gap

Usually, an internal DSL can only be an approximation of the syntax you really wish for, there is a "syntactical gap". It is tempting to subordinate everything else in order to come as close as possible to the intended ideal syntax. But of course beauty has its price, pushing the limits of the host language's syntax can be problematic, and neglecting the other requirements can render a beautiful DSL useless.

That's why it is important to have a rough idea how close the internal DSL needs to come to the ideal syntax, and when to start compromising. One relevant question in this context is how "tech-savvy" users of the DSL are. Generally, people without coding 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 is easy to master. Once a DSL reaches a certain size and complexity, it is naive to assume that people can use it without any assistance. The learning experience of the users is often just an afterthought, but I think it is important to account for it already in the analysis process.

The DSL itself can be designed in a way to have a shallow learning curve, e.g. by mimicking known systems, or by emphasizing consistency over conciseness. An example for a "too concise" DSL might be regular expressions, as many users - including me - have to look up the syntax over and over again.

Of course, training and documentation are essential factors for a good learning experience. However, an often overlooked topic 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. In order to get those factors right, you have to know your target audience and their skill-set.

In the end it's not the DSL itself which generates value, but the people using it effectively, and this is something we should never forget. Planning for the required assistance upfront can help to improve the DSL, and empower the users to get the most out of it.

## 2.6. Safety of Use

An often overlooked requirement is how and how well the user has to be protected against pitfalls and misuses of the DSL. This includes questions like whether misuses should cause compile time or only runtime errors, or whether it is acceptable that the DSL can express some states that shouldn't be allowed.

Reasons for unsafe behavior can be e.g. an overly flexible syntax, missing sanity checks or mistakes stemming from operator precedences. In some cases the safety-of-use requirement is conflicting with the pursuit of the ideal syntax, and then you need to find a compromise between a beauty and safety.

Some level of unsafe behaviour might be acceptable for tech-savvy or experienced users, or in areas with lower safety requirements, like test data generation.

## 2.7. Ensuring Extensibility

Insufficient extensibility is a common pitfall when designing internal DSLs, which usually operate with quite specialized techniques and language features. Sometimes new DSL requirements can't be properly implemented with the chosen feature set, which may render the whole DSL useless.

Therefore, it makes sense to get upfront a rough idea which extensions could be requested after the first version of the DSL. Later, in the implementation phase, this information helps to avoid using techniques that are not flexible enough to handle future requirements.

Another type of extensibility that is often overlooked is customization by users. For instance, consider a DSL for working with physical quantities: there are too many units to cover all of them, but users might have a particular set of units that they require in their daily work. Therefore, it would be beneficial if the DSL allows users to add custom units. Such considerations can significantly increase the usefulness and success of a DSL. If possible, it is recommended to open your DSL for extension by users, and also provide documentation on how to do it. However, it may be advisable to restrict extensibility when you can predict that extending the DSL may lead to unexpected or unsafe outcomes.

## 2.8. Maintainability

The DSL has to be not only implemented, but must be maintained as well. It should be estimated how many resources are necessary to keep the code running and to update it. Knowing these expectations can help to decide e.g. which dependencies are acceptable, or if code generation is required.

## 2.9. Performance and Memory Requirements

Often performance considerations don't get much attention. However, in most cases DSLs invoke additional operations, instantiate extra classes and may trigger garbage collections. When working with big data, or having a wasteful DSL design, you might run in performance and memory problems. That's why it is necessary to estimate performance requirements upfront, and to employ load tests and metrics accordingly.

## 2.10. Java Interoperability

This is a Kotlin-specific question: There are plenty of environments using a mix of Java and Kotlin, so it might be required 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 to bridge the gap, and the Kotlin language itself contains some features to increase the interoperability with Java.

If Java interoperability is required, it should be already considered in the design phase. The respective challenges and possible solution are discussed in chapter 13 in more detail.

## 2.11. Closed or Open Source

One important consideration that should be decided up-front is whether to make the DSL project open source. 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, as well as potentially exposing any flaws or security vulnerabilities to the public. Additionally, open source projects require ongoing maintenance and support from the original developers, which can be time-consuming and resource-intensive. Ultimately, the decision to make a DSL project open source should be weighed carefully against the potential benefits and drawbacks.

## 2.12. Ready, Steady, Go?

After identifying the requirements for your DSL project, it is important to carefully consider its scope, complexity, and benefits before moving forward. While DSLs can be highly beneficial to build, it is important to ensure that they have a clear purpose and add tangible value for 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 just a means to an end, and not an end in itself. It should ultimately serve the needs of its users and add value to your organization. As such, careful consideration of the project's feasibility, purpose, and value is critical before beginning implementation.

# Chapter 3. Writing a DSL

Now that you have defined the requirements for your DSL and are ready to move on to 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 when you begin designing and implementing your DSL:

- **Imagine the ideal syntax:** Begin by considering the concepts and ideas that 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.
- **Prototyping:** 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.
- **Formalization:** 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.

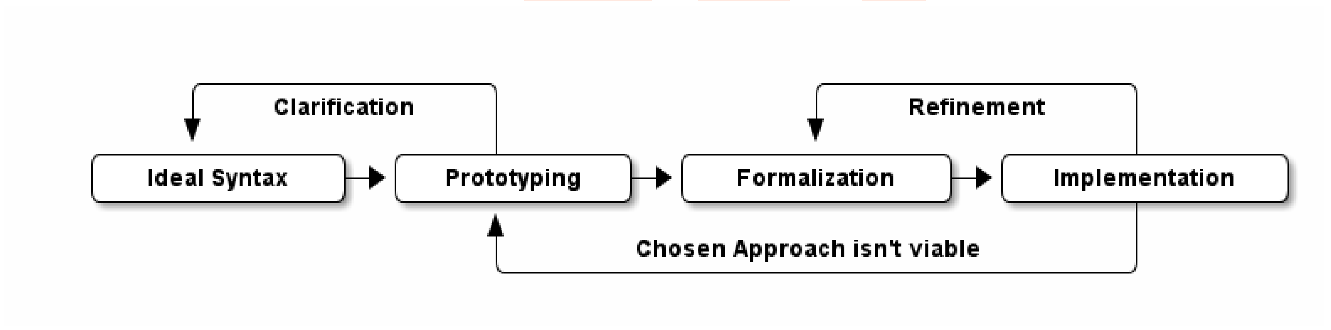

*Figure 1. DSL-Development Workflow*

By following this simple workflow, you can ensure that your DSL project is well-structured and successful.

## 3.1. Imagine the Ideal Syntax

It may seem strange at first 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 the design of 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. Especially experienced programmers have a strong understanding of the capabilities and limitations of a programming language, and 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 keep in mind that the goal is not simply to create a working program, but a unique and consistent language.

Instead of focusing too narrowly on language features and implementation details, it can be helpful to take a step back and consider the broader goals and purpose of your DSL. Get into brainstorming mood, be open to exploring creative and unconventional approaches. The goal is to sketch out how an ideal DSL syntax could look like, regardless of how to implement it. If your team can't agree on one 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 can be more effective to write lots of examples of how you envision the DSL being used, and to include comments and explanations to help clarify your intentions. Make sure to pay attention to any inconsistencies or gaps in the examples, and to specify how the output for each example should look. This is crucial information that will help ensure that your DSL is clear and intuitive to use.

If possible, it can also be helpful to seek feedback from domain experts or potential users of your DSL. This can help you identify any 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.

## 3.2. Prototyping

Now that you have defined an ideal syntax, it's finally time to begin 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 out different approaches and see how they work in practice. You might choose to implement only selected features of your DSL (a.k.a. "spike implementations"), or to create an empty shell that simply verifies that your chosen syntax is viable. With a prototype, you can quickly test out different ideas and identify potential issues, you can save time and effort by avoiding extensive testing and sanity checks, and you can 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 regularly check that the syntax you have 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, as they can help you identify any issues or areas for improvement. One 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 have 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 guidance 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 how a realistic syntax could look, 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 scope and complexity of your project, and on the level of precision and formality required.

In some cases, formalizing the syntax of your DSL may involve creating a simple documentation page or a 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 should be thorough and complete to ensure that your DSL is clear and easy to use.

Ensuring completeness is a crucial aspect of designing a DSL: It's essential to make sure that the language is capable of expressing all "allowed" configurations, and that no rare or unusual cases are overlooked.

One completeness issue in DSL design is a lack of orthogonality, which refers to the idea that different elements of the language should be independent and not overlap or interfere with each other. For example, consider a DSL for describing animals that doesn't allow you to choose both "mammal" and "lays eggs" as characteristics, even though this combination actually exists (e.g. for the platypus). In this case, the DSL would be lacking in orthogonality, as it doesn't allow you to fully and accurately describe certain animals.

The formalization step shouldn't be skipped. While a prototype can be a useful tool for exploring different approaches and identifying potential issues, 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 a detailed documentation to understand how to use it effectively. By formalizing the syntax of your DSL now, you can save time and effort later on.

## 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 match these standards.

Be prepared for the possibility that your prototype may not be thorough enough, or may not cover all the necessary cases, and that you may 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 return to the prototype phase and explore other approaches or ideas. While it may be tempting to try to power through with your current approach, this can often be counterproductive, as it can limit your field of view 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 might want to consider using a source code generator to automate this process. This can help you save time and effort, and can help you create a DSL that is easier to maintain and extend.

Finally, be sure to follow best practices when implementing your DSL. This may include 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.

The implementation of a DSL is quite often different from the usual programming tasks, therefore it comes with its own challenges and pitfalls. Some points that deserve special attention are listed below.

### 3.4.1. Name Clashes

A good DSL can be used extensively in a code base, but this can increase the risk of name clashes, 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 pulling them into DSL-specific objects or classes whenever possible. It's also a good idea to consider the potential for clashes already when naming your functions, operators etc., so that they are less likely to cause conflicts.

### 3.4.2. Coupling

When writing a DSL for creating classes that are also under your control, you might be tempted to integrate the DSL tightly into these classes. This can backfire, for various reasons:

- DSL code gets entangled with business logic
- the DSL becomes part of the business API, making it bloated and inflexible
- In many cases it is important that the result classes can work on its own. This can be the case when code generation or analysis tools are involved, when working with big data, or for testing
- at some point in time, the DSL may get obsolete

In Java, this kind of tight coupling might 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 count.

It's generally a good practice to avoid tightly integrating a DSL into the classes it is creating, as this can lead to a number of problems. Some potential issues include:

- **Entangling DSL code with business logic:** Such tight integration can complicate to separate the two and to make changes to either without impacting the other.

- **Making the DSL part of the business API:** The DSL may become part of the business API, which can make it bloated and inflexible. This complicates evolving the DSL or the business logic independently of one another.
- **Limiting the usefulness of the result classes:** Tightly coupled result classes may not be able to work on their own or may be hard to use with other tools and frameworks, or from other JVM languages like Java. This can limit their usefulness in a variety of contexts, such as when working with big data, testing or code generation.
- **Making the DSL harder to replace:** It can be difficult to replace a tightly coupled DSL if the need arises. This can make it harder 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, in order to avoid these kinds of issues. In Kotlin, you can use features like extension methods to create DSLs that are flexible and easy to use, while still keeping the DSL and the classes it creates separate.

### 3.4.3. Code Conventions

It’s generally a good practice to follow code conventions, as this can make your code more consistent and easier to understand for other developers. However, there may be cases where you need to compromise on certain conventions in order to create an expressive DSL. If you do need to make compromises 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.

### 3.4.4. Testing

For some DSL categories, testing can be more difficult than for normal code, as the code might be less rigid than usual, or - to use a mechanical analogy - it can have more moving parts and degrees of freedom. This makes it more likely to overlook edge-cases or unwanted behavior. A particular challenge are compile-time guarantees: There is no convenient way to test that certain unwanted code structures don’t compile. Overall, depending on the type of DSL, testing can be more challenging than for ordinary code, and might require more attention and effort.

Some common challenges in testing DSLs include:

- **More complex code structures:** DSLs can have more complex code structures than ordinary code, e.g. classes acting as wrappers, or intermediate builder classes.
- **Combinatorial explosion:** DSLs may allow to combine its elements as building blocks. This can make it more difficult to test all possible combinations and edge cases, and to ensure that the DSL is behaving as expected.
- **Compile-time guaranties:** Some DSLs use type-level programming to introduce compile-time guaranties, 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 might be required. E.g., if your DSL is used for code generation, you may need to test the generated code

in addition to the DSL itself.

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

### 3.4.5. Documentation

Many software developers don't like to write documentations, but it is important. When writing documentation for a DSL, keep in mind that it is essentially its own language, and that users may not be familiar with all of its features and concepts. Therefore, it's crucial 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 an example 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 including unnecessary details, complexity and external dependencies.
- **Use meaningful examples:** Choose 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 is doing and how it is using 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.

# Chapter 4. Relevant Language Features

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 syntactical sugar that makes Kotlin code not only more readable and easier to maintain, it also helps to 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, notably 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 basic Kotlin knowledge. However, DSLs often use little-known language features, or they use some features in other ways than common code. Sometimes, even the users of the DSL might not be aware of the kind of "machinery" used to make it work.

This chapter gives a short overview of language features that might be relevant for 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 usage of identifiers that are keywords in Kotlin but not in Java, like `fun` or `when`. For DSLs, we can use them when we need descriptive identifiers which don't follow the usual syntactic rules.

Backtick identifiers start and end with a backtick `` ` `` - these are not part of the identifier, but just delimiters - and can contain almost all characters. E.g. for Kotlin on the JVM, all characters except ``\r\n,.;:\|/[]<>` `` are allowed.

A typical use case for backticks are DSLs for test libraries. While Java limits you to underscores and camelCase, 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`() {
   ...
}
```

Another common use case is to simulate an operator with an infix function, e.g. you could define an exponentiation operation using a syntax like `5.0 `^` 3` (see section Infix Notation for Functions).

## 4.2. Named Arguments and Default Values

Java relies on the order of the parameters when calling a method or constructor, which can get confusing very quickly. In contrast, Kotlin allows you to address arguments by name, which is much more readable, and doesn't require you to remember the order of 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 imitated by 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 need 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 enhances readability. In the example above, it would be sensible to set the alpha value to 255, as 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, containing "lenses" and similar abstractions, which allow to easily compose copy operations for different nesting levels. If you want to learn more about this topic, I would suggest to check out Arrow Optics.

## 4.3. Trailing Lambda Arguments

If a method expects an argument of a function type, you can use the usual lambda syntax with curly braces when you call it. E.g. you can merge a list of strings using the `fold()` method as follows:

```
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 inside its curly braces:

```
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 accept a variable number of arguments. However, Kotlin has made several improvements to varargs, making them safer and more convenient to use.

One of the main 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 for a vararg, or if its elements should be used as individual arguments. Kotlin addressed this problem by introducing the unary "spread operator" `*`, which indicates that the elements of an array (and not the array itself) should be used as arguments for a vararg.

Furthermore, Kotlin allows for a more flexible use of varargs. You can freely combine single-value arguments with elements from spread arrays, which looks like this:

```
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

In contrast to Java, where a vararg must always occur as last argument, Kotlin allows to put the vararg anywhere, even though you might need to use named arguments in order to avoid ambiguity:

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

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

Note that varargs can't be assigned one by one when referred by a named arguments, but have to be bundled up in an array instead:

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

At first glance, having the choice to put varargs wherever you want doesn't seem to be terribly useful. But there is one particular use case which is very interesting from a DSL design perspective: You can put a vararg as 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, in this case there are no named arguments required.

## 4.5. Property-Syntax

Kotlin allows to control how properties are read and written. This makes it easy to hide DSL functionality in plain sight. A straightforward 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
        }
}
```

Similarly, you can perform additional actions when reading 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 more things you can do with properties, like caching, lazy evaluation, delegation to other properties or input sanitation. Later in the Extensions and Receivers section we'll discuss another use of the property syntax.

## 4.6. Extensions and Receivers

One of the most important Kotlin features for DSL design are extension functions, lambdas and properties, which allow to add functionality to existing classes - even final ones - without touching them. These extensions are stand-alone constructs operating on a so-called receiver, which is the target class they are extending. The function body is put in the scope of the receiver, so you can access its public fields, methods etc., and you can also refer to the receiver itself using `this`. Here is how 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 how a normal method call (or property access) on a receiver instance would look like. This makes extension functions a great tool for adding DSL features to classes that you have no control over. A great example are the `apply()`, `run()`, `let()` and `also()` functions in the Kotlin API, which make it easier to use e.g. expression body syntax for functions, or succinct variable assignments.

**Extension Method History**

Extension methods, initially introduced in C#, found their way into Kotlin as "extension functions", especially with the seamless integration of Java classes into the Kotlin ecosystem in mind. This approach was chosen over Scala's implicit conversion approach, which, although more powerful, was deemed more complex to comprehend and manage. The advantages of extension methods became so evident that they were incorporated in Scala 3 as well. It is safe to say that extension methods are an important and successful advancement in contemporary object-oriented programming.

### 4.6.1. Type Narrowing

Interestingly, generic extension functions have a capability that normal instance methods don't have: They can fixate generic parameters on a certain type, narrowing down the range of possible receivers. 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 carried over to the function body. This feature of extension methods can be used in DSLs for performing 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 usage 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 leaks 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 completes, 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 to improve the maintainability and robustness of code, by making it easier to reason about the usage of resources and ensuring that they are properly managed throughout the program.

Lambdas can have receivers too, which is practical when applying the Loan Pattern. Using this pattern can be beneficial in DSLs, as it helps to control the life-cycle of the receiver class, and to hide the steps necessary for initializing and finalizing the instance creation or operation.

Take for instance the well known `java.util.StringBuilder` class. It allows to do perform complex String operations, but in order to use it, you need to construct it, and to call its `toString()` method at the end. When applying 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 method
fun sb(block: StringBuilder.() -> Unit): String =
    StringBuilder()
        .apply { block(this) }
        .toString()

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

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

### 4.6.3. The @DslMarker annotation

When you nest several extension functions, the overlapping scopes can pose a problem: Things visible in the outer code blocks are also visible in the inner ones. E.g. in a DSL for HTML generation, one could write:

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

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

- Define a custom annotation
- Annotate this annotation with `@DslMarker`
- Mark all involved receiver classes (or a common super class) with your annotation
- Now, you can't directly access elements from the outer scope. You still can refer them indirectly, e.g. using the syntax `this@html.head{…}`

In our example, such an annotation could look like this:

```
@DslMarker
annotation class HtmlMarker
```

When the receiver classes of the lambda arguments of the `head()` and `body()` functions are annotated with `@HtmlMarker`, the example above wouldn't compile any longer.

### 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.

### 4.6.5. Context Receivers

Context receivers are still an experimental feature in Kotlin, so some details could change in the future. We won't use them in this book, but they are an interesting concept, and might turn out very useful for writing DSLs. The basic idea is to get a class providing a certain service into scope:

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

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

Here, `methodWithContext()` can access members of the given `EnvironmentContext` class, similar as in an extension function. The difference is that you don't call the method on an instance of the context, it is just available. This also allows to have multiple contexts in scope.

To call `methodWithContext()`, an `EnvironmentContext` implementation must be provided:

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

Context receivers in Kotlin share similarities with extension functions but lean more towards the concept of dependency injection. They come into play when there's a need to incorporate global information within a specific scope while maintaining flexibility to accommodate different versions. By using a context receiver to offer DSL functionality, you gain control over the scope, can influence the general behavior of the DSL and can prevent potential name conflicts.

## 4.7. Operator Overloading

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

The boolean operators `&&` and `||`, the access operators `.`, `?.` and `!!`, the (unary) spread operator `*` and the Elvis operator `?:` 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. |

When these operators are defined, `a` is mutable, and left hand side and right hand side have matching types (`B` is a subtype of `A`), they can be also used in the assignments `+=`, `-=`, `*=`, `/=` and `%=`.

When 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 for the function must be `Unit`. Having both the binary and the assignment version of an operator in scope leads to 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 is new, and is thought 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 designing DSLs, 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 some clear association or analogy between the operation being performed and the chosen operator.

For example, using the / operator to concatenate file paths makes sense, as it is a common path separator. Similarly, using the unary + operator to "add" a single value inside a trailing lambda block has become a standard convention. And using .. instead of : may be acceptable due to its visual similarity.

However, at some point overloading operators can become confusing or even counterproductive. For example, using the ! operator to invert a matrix may be a stretch, as it does not have a 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 understandings of certain symbols or operators.

One solution is to use meaningful infix functions with expressive names instead of 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

The infix notation allows names of functions 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 respective function must be an extension function with one argument.The receiver becomes the left-hand side and the argument becomes the right-hand side of the operator.Note that you can still use the normal function call syntax. Here is an example for 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
}
```

A weakness of the infix notation is that you can’t explicitly specify generics. In this case, you can fall back to the normal function call syntax - but users of the DSL might not know this.

As already mentioned, combining infix and backtick notation allows to define at least visually new "operators":

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

val result = 1.2 `^` 3
```

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

Such code is quite ugly, and way too verbose to expect a DSL user to implement your interface this way. But as the interface has only a single abstract method (abbreviated as "SAM"), it can be written as a functional interface, which allows to use 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 }
```

The lambda will be automatically translated back to an implementation as shown above (this process is called a "SAM conversion"). I think you agree that this syntax looks much better, making it useful for DSLs.

## 4.10. Generics

Generics are a useful abstraction over concrete types in all kinds of contexts, including DLS design. A specific use case is the implementation of compile time checks. Here is a simple example modelling 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
```

Adding amounts of different currencies together isn't possible, 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 for type-level programming.

### Type-level Programming and Phantom Types

**Type-level programming** is a programming paradigm where types themselves are used as values that can be manipulated and computed upon at compile-time, rather than just being used to check the correctness of program syntax and logic. In other words, type-level programming involves using types to encode complex computations and algorithms, which are evaluated by the compiler at compile-time instead of runtime. Type-level programming can be used to achieve a wide range of goals, such as improving program performance, reducing runtime errors, and enforcing stronger type constraints.

**Phantom types** are a type-level programming technique where a type is used to encode additional information about the data that it represents, without actually storing any data at runtime. Phantom types are types that have no values, but are used purely 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 to reduce runtime errors and improve the safety of the program.

### 4.10.1. Reified Generics

Kotlin offers a interesting feature called "reified generics", which 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 make it possible to retain type information 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>).fold("", String::plus)
    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 eliminates any 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 obtain the generic type information from the place where the inlining is happening, and make it look like as there were no type erasure. It should be noted that inline functions are subjected to some restrictions and vary slightly from normal functions, e.g. regarding their return behavior.

## 4.11. Value Classes

Value classes are a feature introduced in Kotlin 1.5 that allow developers to create lightweight, efficient classes that represent simple values. Value classes are designed to be used for values that are frequently 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 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. Instead, the value of a value class is typically represented directly in memory or as a primitive type, depending on the underlying data type.

One of the main benefits of value classes is that they can be used to create more expressive and type-safe APIs. For example, a value class representing a specific measurement unit can help to ensure that only valid unit conversions are performed, and can help to catch errors at compile-time rather than runtime.

For the JVM backend, a `@JvmInline` annotation is required, which may be no longer needed in the future. Further, the single constructor argument restriction might be dropped as well. This depends on the introduction of Project Valhalla, 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 - also unnamed - type. Here is a somewhat silly example to illustrate this point:

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

The anonymous object hasn't the type `Any`, else we couldn't call `sayHi()` on it. It defines its own type, which exposes variables and functions defined inside it. For this reason anonymous objects can be used in DSLs as a setup-stage or environment for later calculations.

A DSL with a method that measures the durations 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 required setup for the measuring runs is stored inside an anonymous object in the `env` variable.

## 4.13. Annotations

You can write whole DSLs using annotations, but more often annotations can support DSLs, e.g. by describing out how certain fields or classes should be handled. They are especially powerful when your DSL shows a certain default behavior, but needs to consider some edge cases or exceptions, like "don't persist this property".

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

**Annotation Processors**

Annotation processors allow to execute custom processor code during the build process, according to the annotations present in the application code. Kotlin features two annotation processors, the older kapt, which won't be developed further, and the recommended Kotlin Symbol Processing API (KSP), which will be covered in chapter 12.

## 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 have to import a separate dependency:

*Gradle (.kts)*

```
dependencies {
    implementation("org.jetbrains.kotlin:kotlin-reflect:1.8.10")
}
```

*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. Conclusion

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

# Part II - DSL Categories

# Chapter 5. Algebraic DSLs

A common class of problems deals with objects that exhibit numeric-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 when the required operators match with the existing ones.

As for many other categories of DSLs, there seems to be no established name for this kind of DSL in the literature. I think calling them *Algebraic DSLs* covers their behavior quite well, but you might find the same concept under different names elsewhere.

The Kotlin API itself contains some nice examples for small Algebraic DSLs, e.g. for `BigInteger` and `BigDecimal`, which can be used pretty much like "normal" numbers.

Even if your problem domain does not possess an inherent algebraic structure, the concepts and ideas presented in this chapter can still be partially applicable. Consider chemical equations as an example: 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 utilized in various domains beyond traditional numeric applications.

## 5.1. Case Study: A DSL for Complex Numbers

In order to keep things realistic, we will use an existing implementation, that you could find in a production environment. We will use the Apache Commons Numbers: Complex library, which is written in Java. The implementation behaves pretty much as expected:

```kotlin
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, as 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 to initialize complex numbers, like the polar definition.

When 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 clashes, so it would be also an option to import `Complex.I` instead.

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

```kotlin
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 =
    Complex.ofCartesian(this, 0.0).add(that)
```

The other operations look basically the same. We could also support interoperability with `Int` in addition to `Double`, but it was omitted for the sake of brevity. A little improvement would be a helper method for the conversion from double to complex, instead of relying on the awkward `Complex.ofCartesian` factory.

Next, we need also to support negation of complex numbers (and maybe the unary plus, too, for symmetry reasons). This looks very similar to the code above:

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

When you can't find an appropriate operator to overload, you can use infix functions instead. In our example, we could define `pow` as exponentiation operation, so we can write e.g. `3.0 + 4.0*i pow 3.0`. The infix operations have 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 be only called when using infix notation, else the original method will be called.

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

```kotlin
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) = add(that)
operator fun Complex.plus(that: Double) = add(that)
operator fun Double.plus(that: Complex) = fromDouble(this).add(that)
operator fun Complex.minus(that: Complex) = subtract(that)
operator fun Complex.minus(that: Double) = subtract(that)
operator fun Double.minus(that: Complex) = fromDouble(this).subtract(that)

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

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

infix fun Complex.pow(that: Complex) = pow(that)
infix fun Complex.pow(that: Double) = 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 feels now as intuitive as using one of the built-in numeric types.

## 5.2. Java Interoperability

Java doesn't allow operator overloading, and extension methods become just normal static methods with the receiver as the first argument. That means that our DSL definitely looks no longer elegant in Java. Given that in our case the underlying Apache Commons Numbers library itself is written in Java, we are probably better off using their methods.

However, our DSL is still working, and is quite straightforward to use, when you know how the operators translate to 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.

## 5.3. Conclusion

The case study presented in this chapter serves to illustrate that creating algebraic DSLs is usually not that difficult. However, it is important to consider certain factors when deciding whether to employ an algebraic DSL. While algebraic notation can be powerful and expressive, it may not always be suitable for every use case. For instance, using algebraic notation for sum and product types, such as `Either` and tuples like `Pair` and `Triple`, might be unconventional and potentially confusing to some users, despite the underlying algebraic structure. Additionally, certain behaviors, like non-commutative multiplication found in quaternions and matrices, can introduce unexpected complexities 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 simply adopting them because of their ease of implementation.

### 5.3.1. Preferable Use Cases

- Define operations

### 5.3.2. Rating

- ☀☀☀☀☀ - for Simplicity of DSL design
- ☀☀☀☀☀ - for Elegance
- ☀☀☀☀☀ - for Usability
- ☀☀☼☼☼ - for possible Applications

### 5.3.3. Pros & Cons

| Pros | Cons |
|---|---|
| • easy to write<br>• intuitive to use<br>• can use infix functions when no operator fits | • possible name clashes with other DSLs<br>• operator precedence can’t be changed<br>• difficult to use from Java client code |

# Chapter 6. Builder Pattern DSLs

A common task is the initialization of a complex, sometimes nested object. The classical solution for this is the Builder Pattern. The builder class is mutable, uses method chaining (a.k.a. "Fluent Interfaces") to simplify value assignments, and contains a terminal build method, which constructs the domain object.

This chapter covers the classical builder pattern as often used in Java. Sometimes this is overkill is Kotlin, as named arguments can be used for smaller classes. We will also discuss nesting builders. Next, we will address the problem of mandatory fields in builders by presenting the *Typesafe Builder Pattern*. Finally, we will cover the more exotic *Counting Builder*. In the conclusion section we discuss why in Kotlin builders are not as widely used as 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:

*See 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 illustrates, a builder has three parts:

- **Builder Initialization:** The builder requires a starting point, which can be achieved through a constructor or a factory method. This initializes the builder instance and prepares it for data collection.
- **Data Collection:** Through method chaining, the builder instance accumulates 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 finalize the construction process, a terminal method, often named `build()`, is invoked. 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 offer a structured approach to construct 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 might not need a builder, because the language contains features like named and default arguments, which can provide a similar functionality. Take this implementation of an RGBA color:

```kotlin
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 might be a little confusing to read, especially when you expect only three values as for RGB colors. However, instead on relying on a builder, you can simply clarify the meaning by rewriting the last line as follows:

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

The syntax differs, but functionality-wise this code resembles closely a builder. Of course, as for a builder, the arguments can be listed in any order. A noteworthy difference of named arguments is that every argument can be set only once, while a builder allows to set and - depending on the implementation - possibly overwrite an argument value multiple times.

`Color` is immutable, but if you want to get modified copies of it, you can use the `copy()` method, like this:

```kotlin
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 might 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 these components, their own subcomponents, etc. A typical example for such nested builders is the DSL of the KotlinPoet 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 demonstrates, the necessity to call `build()` at the end of every nested builder leads to a lot of visual clutter. To avoid this issue, some DSLs make builder nesting more convenient by having two versions of every "nested" method: One version that takes as usual the constructed object as argument, and another version that accepts a builder of the object instead. This way, the user doesn't need to call repeatedly `build()` methods for the nested builders.

## 6.4. 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()
```

Using this approach can make the DSL code more readable, but it requires more discipline from the user, who has to ensure that the start and end methods are placed properly.

To give an example implementation, consider a person class containing a name, a phone number and a list of contacts, which in turn 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 be also regarded as a very simple state transition, namely from the outer level to the inner level and back. Chapter 8 demonstrates techniques to implement such state transitions in a safe way, so the code wouldn't compile when 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 both conceptually and implementation-wise simpler, it also scales better, and should be therefore preferred.

## 6.5. The Typesafe Builder Pattern

A common issue 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 already prevent to build incomplete objects. 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 only accept builders with all mandatory values set. As an example, consider the following class for a product, which requires a product id, the name and the 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 prerequisite for our builder are three classes for representing the state of the mandatory fields. They are similar to `Optional`, with the difference 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 underlying idea is simple: Each
mandatory field has its own generic type parameter that tracks whether it is already
set or not. The constructor is private in order to prevent that the builder is . Now
we can 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 {
                inline 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 we thankfully 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 will change from `Without` to `With` for this particular field. As the setters for the mandatory fields 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, one crucial part is missing: The `build()` method. However, we can't write it as part of the builder class, as it needs to inspect the generic signature. It *has* to be an extension method, because only there you can "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, as the type inference matches on the "narrowed down" type. Now we have a builder with a `build()` method that can be only 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.

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

## 6.6. 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 kind 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 issue, 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 achieve better safety by preventing the creation of an invalid builder at compile time. This can be achieved by using a recursive type parameter to keep track of the number of points, once again employing type level programming. Though 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 instance, 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 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 points in the list.

### The Peano Axioms

When asked to count, the usual response is "zero, 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 in order to define natural numbers and their properties in a formal way, and the Peano Representation follows directly from these axioms.

The first axiom covers the existence of zero, the following 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 any natural number, its successor is a natural number as well
- If the successors of two natural numbers are equal, then the numbers themselves are equal, too
- Zero is not the successor of a natural number
- Every 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 is the most basic way to write natural numbers, and our usual number systems (decimal, binary, hexadecimal…) could be regarded as convenient abbreviations.

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 returns only 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, as the counter is a phantom type. Alternatively, we also could have constructed a new builder object on every `add()` call.

The last ingredient is the `build()` method, which has to be an extension function, for the same reasons as in the typesafe builder example. The function is "counting" the points by inspecting the type signature of the builder. This is how a usage of our builder could look like:

```kotlin
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, as the type of the `PolygonBuilder` does no longer comply with the signature of the `build()` extension function.

Of course, you can use this pattern to count more than one thing, and you can combine it with the Typesafe Builder Pattern as well.

## 6.7. Builders with multiple stages

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

## 6.8. Conclusion

The Builder Pattern is quite popular in Java - there are libraries like Project Lombok which generate builders for you. The downside is that builders are quite inflexible and might be not very safe to use (although variations like the Typesafe Builder Pattern can help). In Kotlin, using named and default parameters can already provide a functionality similar to a builder. The next chapter will present another common approach in Kotlin, which has some advantages over the Builder Pattern.

### 6.8.1. Preferable Use Cases

- Creating data
- Generating code
- Configuring systems
- Testing
- Logging

### 6.8.2. Rating

- ☀☀☀☀☼ - for Simplicity of DSL design
- ☀☀☼☼☼ - for Elegance
- ☀☀☀☼☼ - for Usability
- ☀☀☀☀☼ - for possible Applications

### 6.8.3. Pros & Cons

| Pros | Cons |
|---|---|
| • easy to understand<br>• applicable for a wide range of construction tasks<br>• variations of the pattern can fix some of its shortcomings<br>• can be autogenerated (e.g. using Project Lombok)<br>• easy to use from Java client code | • often not the most natural syntax for the problem<br>• nested builders don't look nice<br>• inflexible structure<br>• boilerplate code (e.g. need for a `build()` method)<br>• assignments are disguised as method calls |

# Chapter 7. Loan Pattern DSLs

This pattern is known by various names, and it can indeed be confusing. One of these names is the "builder pattern," which can lead to ambiguity since it shares similar use-cases with the classical builder pattern. However, in Kotlin, this pattern is often used as a replacement for the classical builder pattern. To provide a more accurate description of the underlying mechanism and avoid confusion, it will be referred here as the "Loan Pattern DSL".

In contrast to the classic builder pattern, the Loan Pattern DSL takes a different approach by using a mutable builder class that exposes its members through the Loan Pattern. This approach leverages the power of extension methods and the 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 classical builder pattern. Further, it can 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 to incorporate other DSL techniques like operator overloading, while the builder approach is often too rigid for such improvements.

In this chapter, we will first write a more convenient replacement for the `HttpRequest.Builder` presented 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

As discussed in the last chapter, `HttpRequest` comes already with a builder, which 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 form of the `newBuilder()` and `build()` calls, and the other method calls are actually assignments in disguise. The question is how we can 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 actually assignments, headers have their own subsection, and durations can be calculated. For convenience, you have the choice whether to specify the web-address as String or URI (the example shows the String version).

As you might have guessed, httpRequest is a method employing the Loan Pattern: It initializes a builder class, exposes it as a receiver, and takes care of the final build() call. Also note how it takes care of a mandatory field (in our case the address) by requiring it as an explicit argument. We have two versions of the function, depending on how the address is specified:

```kotlin
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:

```kotlin
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>()

    ...

    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 enables to fill the headers field. The range operator .. was chosen to collect the key-value-pairs because it looks somewhat like :. As discussed in the fourth chapter, we don't want to expose all the HttpRequestBuilder fields from its inner class Headers, and that's why we use te @DslMarker mechanism to limit the scope inside of Headers.

```kotlin
@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, there are some fields and methods required in order to simplify setting the HTTP method. However, you can still define your own ones, 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
    ...
}
```

In order to simplify the definition of a `Duration`, there are some operator overloading functions defined. Note that these are only visible inside of `HttpRequestBuilder`, so name clashes 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)
    }

    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 adapting existing libraries in this way, especially those written in Java, it becomes possible to better meet the needs of users and to integrate them more seamlessly into the Kotlin ecosystem. The end result is often a more natural and intuitive experience for developers.

## 7.2. Case Study: HttpRequest with AutoDSL

As this kind of DSL is very common, and its structure is quite predictable, it shouldn't come as a surprise that there exist libraries for deriving such DSLs automatically. At this point we will cover the AutoDSL library, which needs to be set up as an annotation processor (either via kapt or KSP). To do this, please follow the description on the GitHub project page.

Please make sure to use the right GitHub project. There is an older library called "AutoDsl", which was the inspiration for the project covered here. Unfortunately, it is no longer maintained, and doesn't work for Kotlin 1.4 or newer.

Remember the work 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 are auto-generating an intermediate class instead, and therefore we have to call the `build()` method at the end. Usually, for classes under our control we wouldn't do this, but instead 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 can't get much simpler than that: All classes which should be included in the DSL are marked with the `@AutoDsl` annotation, and when there are lists that should be specified element-wise and not 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 usually find the generated classes `HeaderDsl` and `HttpRequestBuilderDsl` in a `generated-sources/…` folder, but of course this depends on how you integrated the AutoDSL processor, and how you set up your project.

The example call from the previous section would now 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 comfortable and concise as for the manually written DSL, but it comes close, and looks definitely nicer and more intuitive than a traditional builder. AutoDSL also keeps track of mandatory fields like `uri`, and throws an `IllegalStateException` if they were 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. Since Kotlin 1.7.0, this feature is enabled by default, but in older versions you can turn in on using the `-Xenable-builder-inference` compiler option. There is no real drawback using this feature, but if you want to look into the details, you can check out the Kotlin Documentation - Using builders with builder type inference.

## 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 to integrate other DSL techniques more easily. The Kotlin language provides several features to improve the user experience, like the `@DslMarker` mechanism and builder type inference.

### 7.4.1. Preferable Use Cases

- Creating data
- Transforming data
- Execute actions
- Configuring systems
- Generating code
- Testing

### 7.4.2. Rating

- ☀☀☀☀☼ - for Simplicity of DSL design
- ☀☀☀☀☼ - for Elegance
- ☀☀☀☀☼ - for Usability
- ☀☀☀☀☀ - for Application Scope

### 7.4.3. Pros & Cons

| Pros | Cons |
|---|---|
| • easy to read, especially for nested constructions<br>• very flexible and intuitive<br>• can be autogenerated (e.g. using AutoDSL) | • behavior is harder to control than for the Builder Pattern<br>• safe usage can't be always guaranteed<br>• might be more difficult to use from Java client code |

# Chapter 8. Modeling State Transitions in DSLs

DSLs often need to model complex state transitions that occur during the execution of the program, to model workflows, or are needed when objects must 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 Loan pattern. The three approaches are:

- **Separate Classes**: The most intuitive implementation. Every 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 its methods return always one of the interfaces, never the class itself.
- **Phantom Type Approach**: A single class has a generic type parameter representing the different states, and extension methods can be only called for the designated state by specifying the corresponding type parameter for their receiver.

## 8.1. Case Study: DSL for SQL queries (using the Builder Pattern)

In order to allow a good comparison between the implementations, we will write a DSL for building SQL queries using the different approaches. As the SQL language is very complex, we have to limit our example to queries using just `SELECT`, `FROM`, `WHERE` and `GROUP BY` clauses, and use simple `String` arguments (for a serious SQL-DSL implementation check out jOOQ). The following state diagram gives an overview of the allowed transitions:

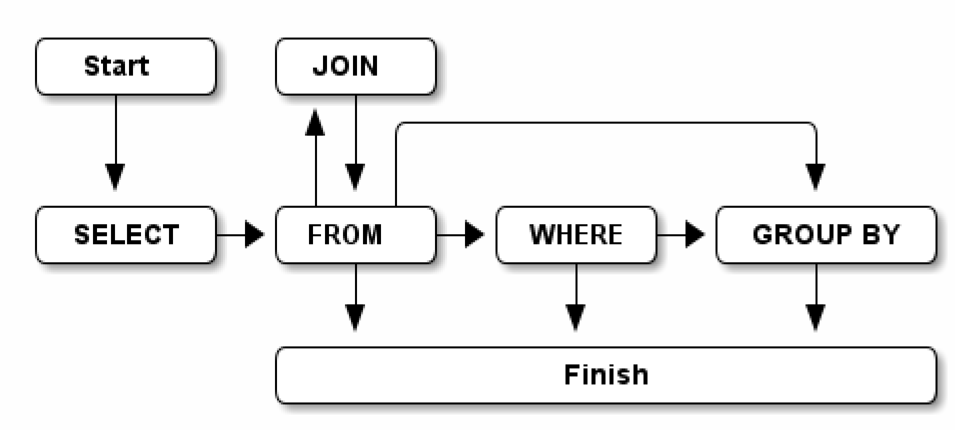

*Figure 2. Simplified Select SQL Query*

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 is using a builder pattern syntax, we will also discuss how a loan pattern implementation of the same functionality could look like.

### 8.1.1. The 'Separate Classes' Approach (Builder Pattern)

The first approach is straight-forward: Every state is represented by a separate class, which is independent of all other classes. While this is conceptually very simple, the main drawback is that all the collected data have to be transferred between the classes representing the states. Often it is more convenient and readable to pass a dedicated data transfer object (DTO)around, instead of using several 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.

As 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 the SQL syntax by permitting 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` offers just 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 safe and should be avoided in production code. Our SQL subset allows to progress to the `GroupByClause` (while the full syntax would also permit `HAVING`, `ORDER BY` etc). Alternatively, we can finish 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` allows just a call to the `build()` method:

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

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

The only missing part 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 in a DTO instance as shown here can reduce the overhead of moving all the data around substantially, especially by leveraging 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 (Builder Pattern)

While having a separate DTO class makes the separate class approach more convenient, it would be nicer if we wouldn't need to copy data around in the first place. But what is with all the guarantees a chained builder provides, e.g. that you can't call `build()` or `JOIN()` in a `SELECT` clause? One way to achieve this is using a technique I dubbed "chameleon class". The basic idea is to adapt the type of this class to the state it currently represents, and change it accordingly when the state changes.

**The Chameleon Class**

A chameleon class

- implements different interfaces
- never exposes its own type, but always acts as one of these interfaces
- has a private constructor in order to avoid leaking its own type
- holds common data

First need to translate our former state classes into interfaces:

```kotlin
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
}
```

Now all left to do is to implement these interfaces in a single chameleon class, and to keep track of the data. It is important to make the constructor private, as the initial type shouldn't be the type of the class itself, but `SelectClause`. That's why the `SELECT()` method in the companion object is used as starting point for the DSL:

```kotlin
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()
}
```

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

The chameleon class concept might look somewhat strange at first, but results usually in compact and readable code. However, be aware that this approach is susceptible to name clashes, when two interfaces contain methods with the same name and parameters, but different return types.

### 8.1.3. The Phantom Type Approach (Builder Pattern)

The third approach uses phantom types. The implementation is based on a DTO class with a generic parameter. This type parameter isn't used as type for any data inside the class - this is why it is called a "phantom type". Instead, this parameter is used by extension functions, which require that their receiver has 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 switch easily between states. As the generic parameter doesn't refer to any real data, the cast itself is safe. Of course, the `cast()` function must be private in order to avoid abuse:

```
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 straight-forward:

```
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>.groupBy(
        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 e.g. 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(…)` in order to be callable from a DTO with a sub-interface, which looks quite cryptic.

Chameleon classes and the phantom type implementation are conceptually similar, and it depends on the problem at hand whether a class with all the methods, or a DTO with extension methods is preferable. In case the DSL has to be called from Java, it should be considered 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, just extension functions operating on them, which means that new functionality can be added more easily than for the other techniques.

## 8.2. Case Study: DSL for SQL queries (using the Loan Pattern)

So far, all examples used a builder pattern syntax. This doesn't have to be the case. A DSL using the loan pattern could 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 might be better suited in cases which need deeper nesting, or which require more complex operations in the trailing lambda bodies.

One difference to 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, e.g. infix functions like AS. Also, we still need build() methods, as it is not clear when we are done with constructing the query. In cases with only one exit state, the construction can be performed behind the scenes, as usual in loan pattern implementations.

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

### 8.2.1. The 'Separate Classes' Approach (Loan Pattern)

Here is how an implementation using separate classes could look like. We start out 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 form of a SELECT function. It executes the given body (where the columns can be added) and hands the results over to the SelectClause class, which in turn has a method for proceeding 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 schema 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 challenging to read and write, but allows for a more flexible syntax inside 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 (Loan Pattern)

To use a chameleon class, first we have to turn the clause data classes in 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 missing part 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 (Loan Pattern)

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

```
interface CanGroupBy
interface CanBuild

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

```kotlin
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. If unsure, I would recommend to start with the separate classes approach, especially when prototyping. When using a DTO as recommended, the code can be transformed easily into the chameleon or phantom type style later.

### 8.3.1. Preferable Use Cases

- Creating data
- Configuring systems
- Testing

### 8.3.2. Rating

- ☀☀☀☀☀ - for Simplicity of DSL design
- ☀☀☀☼☼ - for Elegance
- ☀☀☀☼☼ - for Usability
- ☀☀☼☼☼ - for Application Scope

### 8.3.3. Pros & Cons

| Pros | Cons |
|---|---|
| • enforces the correct state transitions<br>• natural way to write code that creates data in stages<br>• natural way to write DSLs for Finite State Machines | • hard to read code<br>• boilerplate code<br>• the "separate classes" approach requires to copy data over<br>• the "phantom type" approach is hard to use from Java client code |

# Chapter 9. String-Parsing DSLs

In some cases, the host language doesn't allow to create the DSL syntax you would like. A string-parsing DSL gives you the freedom to write basically anything. One topic where traditional DSL categories are particularly bad is giving the user some leeway, e.g. concerning upper and lower case syntax, order of operations or the use of special characters like symbols. String-based DSLs can easily incorporate such leniency.

The downside is that these DSLs are hidden from the scrutiny of the compiler: There are no compile time checks, no hints for auto-completion etc., which means the user has to rely on the documentation. Another issue is writing the parser: Dissecting a string is usually more involved than collecting the data in other DSL types, although it is also no rocket science.

Kotlin supports - like many languages - regular expressions, which can be seen as a kind of mini-parser. For small, isolated use cases they can be very convenient. However, employing them for larger problems can lead to code which 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 an easy example, where we need only a few string functions. If you don't play chess, you probably never heard of the Forsyth–Edwards Notation (FEN):

**Forsyth–Edwards Notation**

The Forsyth–Edwards Notation is a standard notation for describing the current position of pieces on a chessboard. It consists of a string of characters that represent the placement of the pieces, which are organized by rank and file. In FEN, uppercase letters are used to represent white pieces, and lowercase letters are used to represent black pieces. The digits 1-8 are used to represent empty squares, and a forward slash is used to separate ranks. FEN also includes information about:

- which player has the next move
- 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 (like "Q" for a queen), plus "P" for pawn. To keep the string short, empty squares are written with a kind of run-length encoding, e.g. 3 empty fields are represented by a 3. The castling rights are given by "k" for king-side and "q" for queen-side, again upper-case for white and lower-case for black. If nobody can castle, a `-` is written instead. The en-passant field contains either a square like `b3` or again a `-`.

A FEN looks like this: `rnbqkbnr/pp1ppppp/8/2p5/4P3/5N2/PPPP1PPP/RNBQKB1R b KQkq - 1 2`. If you put the rows one below the other, and expand the numbers to empty fields, 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
```

```kotlin
        }

    private fun castlingFen() = when {
        castling.isEmpty() -> "-"
        else -> castling.joinToString("") { it.symbol }
    }

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

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

### Round-trip Tests

When you have code that transforms forth and back between different formats, it is convenient to write round-trip tests: You’re providing test data for the "easier" format, transform it to the other one(s) and back, and compare it with the original. Comparing data in the same format is simpler and safer than comparing different ones - often just a string comparison is sufficient.

Now we can write the DSL function for parsing a FEN:

```kotlin
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()
    .mapIndexed { rowIndex, row ->
        row.mapIndexedNotNull { colIndex, ch ->
            values().find { it.symbol == ch.toString() }
                ?.let { "${'a' + colIndex}${rowIndex + 1}" to it }
        }
    }
    .flatten()
    .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 were omitted for better readability. In easy cases like this, it is probably overkill to write a parser or to use a parser library. The hardest part was to read the piece positions correctly, and this took just a few lines.

## 9.2. Case Study: Chemical Equations as Strings

Writing a DSL for chemical equations is challenging using Kotlin syntax, because the concise notation doesn't naturally lend itself to be easily expressed using language features. That's why using the string parsing approach seems like a good fit. If you are curious how a hybrid DSL for chemicals could 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 scope 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) : Part {
    override fun toString() = when (subscript) {
        1 -> symbol
        else -> symbol + subscript
    }
}

data class Group(val parts: List<Part>, val subscript: Int) : Part {
    override fun toString() = when (subscript) {
        1 -> parts.joinToString("", "(", ")")
        else -> parts.joinToString("", "(", ")") + subscript
    }
}

data class Molecule(val coefficient: Int, val parts: List<Part>) {
    override fun toString() = when (coefficient) {
        1 -> parts.joinToString("")
        else -> "$coefficient${parts.joinToString("")}"
    }
}

data class Equation(val leftSide: List<Molecule>, val rightSide: List<Molecule>, val
reversible: Boolean = false) {
    override fun toString() = leftSide.joinToString(" + ") +
            (if (reversible) " <-> " else " -> ") +
            rightSide.joinToString(" + ")
}
```

An Element contains a chemical symbol, like "H" (hydrogen) or "Ba" (barium), and optionally a subscript, which is counting 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 contain not only 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. 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 either of a single molecule or a "sum" of molecules.

The code and 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 chart summarizes the structure of our model classes:

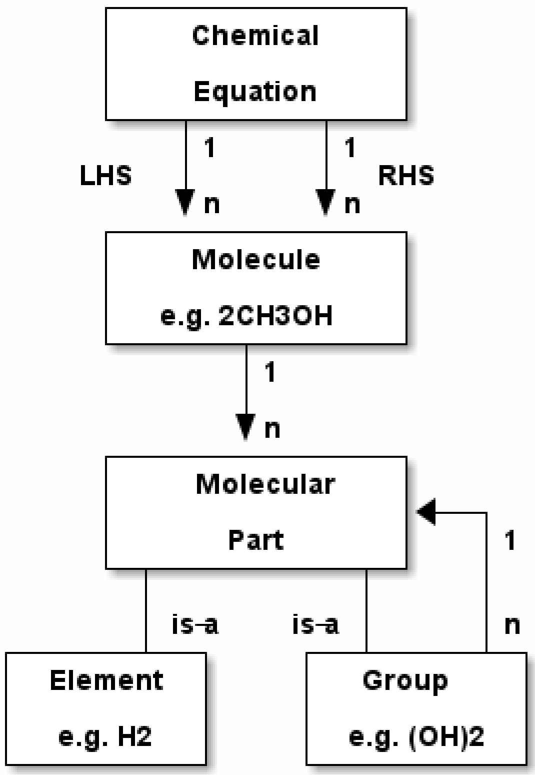

*Figure 3. Model for Chemical Equations*

The equation mentioned above for making barium phosphate could be written like this:

```
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),
    listOf(water, bariumPhosphate),
    false)

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

### 9.2.1. Writing a parser for Chemical Equations

If you never worked with parsers, it can be a little confusing. Writing them by yourself is not really difficult, but boring and tedious, so most of the time using a library will be the better choice. Nevertheless, I think it is instructive to see how a simple parser works, so a naive manual implementation is presented first, before utilizing a parser combinator library.

First, we need some general code for a rudimentary parser. We start from a common interface `ParseResult`, as we also have 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 this class empty. We then need a class `Success` to hold the current successful parsing result, together with current location we are working on. For the location, we simply use the remaining string - more performance-oriented implementations typically use just the index of the input string. Overall, these classes have a lot in common with Java's `Optional` class:

```
sealed interface ParseResult<out T> {

    infix fun or(that: () -> ParseResult<@UnsafeVariance T>): ParseResult<T>

    fun <U> flatMap(body: (T, String) -> ParseResult<U>): ParseResult<U>

    fun filter(cond: (T) -> Boolean): ParseResult<T>
}

class Failure<T> : ParseResult<T> {

    override fun or(that: () -> ParseResult<T>): ParseResult<T> = that()

    override fun <U> flatMap(body: (T, String) -> ParseResult<U>) = Failure<U>()

    override fun filter(cond: (T) -> Boolean): ParseResult<T> = this

    override fun toString() = "Failure"
}

data class Success<T>(val value: T, val remaining: String) : ParseResult<T> {

    override fun or(that: () -> ParseResult<T>): ParseResult<T> = this

    override fun <U> flatMap(body: (T, String) -> ParseResult<U>): ParseResult<U> =
        body(value, remaining)

    override fun filter(cond: (T) -> Boolean): ParseResult<T> = when {
        cond(value) -> this
        else -> Failure()
    }
}
```

Then we have some helper functions for reading parse results, and for generating lists from individual results:

```
fun <T> successWhen(cond: Boolean, body: () -> Pair<T, String>): ParseResult<T> =
    when {
        cond -> body().run { Success(first, second) }
        else -> Failure()
    }

fun <T> ParseResult<T>.successOrNull(): Success<T>? = when (this) {
    is Success -> this
    else -> null
}

fun <T> sequence(start: ParseResult<T>, step: (String) -> ParseResult<T>):
ParseResult<List<T>> =
    Success(
        generateSequence(start.successOrNull()) { last ->
            step(last.remaining).successOrNull()
        }.toList(), ""
    ).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 with working on the equation parsing. Note that we assume there are no whitespaces in the formula, because handling them everywhere is painful, and we can easily filter them out at top level.

First, all the element names must be known:

```
private val elements = setOf(
    "H", "He", "Li", "Be", "B", "C", "N", "O", // etc.
)
```

Next, we need functions who can recognize given patterns and natural numbers:

```
fun parsePattern(string: String, pattern: String): ParseResult<String> = when {
    string.startsWith(pattern) ->
        Optional.of(pattern to string.substring(pattern.length))
    else -> Optional.empty()
}

fun parseNum(string: String): ParseResult<Int> =
    Optional.of(
        string.takeWhile { it.isDigit() }.length
    ).filter { digitCount ->
        digitCount > 0
    }.map { digitCount ->
        string.substring(0, digitCount).toInt() to string.substring(digitCount)
    }
```

The easiest function is parsePattern(), which tries to find a given prefix in the string. parseNum() is slightly more involved, as it needs to determine the number of digits first. With one exception (the function findElement()), all functions don't read directly from the string, but use these two low level functions and combine the results in certain ways - that's why this approach is known as "parser combinator".

The first example for this "assembling" is the function for reading the equation arrow, which can be either -> or <->:

```kotlin
fun parseArrow(string: String): ParseResult<String> =
    parsePattern(string, "<->")
        .or { parsePattern(string, "->") }
```

Reading an element is not difficult, the only pitfall is that two-letter symbols must be checked before the one-letter symbols, else the function would just find H in a string starting with He.

It is a common problem that two parsers could match for the same input. Usually the parser reading the longer prefix is the one you want to execute, so you have to make sure to evaluate it first.

```kotlin
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> =
    successWhen(elements.contains("$string!!".take(charCount))) {
        "$string!!".take(charCount) to string.drop(charCount)
    }
```

First, the findElement() function tries to find elements, first with two, and then - if this was unsuccessful - with one character. Artificially prolonging the string with some characters that definitely won't match (here !) avoids a possible IndexOutOfBoundException. The map block in parseElement() attempts to find a trailing number. If the number is found, it is used to construct the element, else the default subscript of 1 is used.

Now the groups can be tackled:

```
fun parsePart(string: String): ParseResult<Part> =
    Failure<Part>() or { 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 or a group. The chain starts with an Failure<Part>, which is a trick to avoid casts for the more specialized return types of parseElement() and parseGroup(). The parseGroup() looks first for an opening parenthesis. Then it tries to read as many parts as possible, but at least one. After this, it looks for a closing parenthesis. The final map() 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, else it uses 1 as default. Then it tries to read as many element or group parts as possible. If some parts were found, the molecule is build, else the parser fails.

This is the parser for gathering the left- and right-hand 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, rhs, arrow == "<->"), s3)
            }
        }
    }
```

It just reads the left-hand side, the arrow symbol, the right-hand side, and combines them. Now the only missing part is a `equation()` function, which is the only part of our DSL which will be exposed to the user:

```
fun equation(string: String): Equation? =
    parseEquation(string.replace(" ", ""))
        .successOrNull()
        ?.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 at it can get 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 a few simple modifications, we could allow this syntax as well. Generally speaking, allowing the syntax of a String-based DSL to be more lenient is relatively easy, while other DSL categories often struggle with this kind of flexibility.

As already stated, writing such a parser manually isn't difficult. However, using a library has many advantages: It improves readability and maintainability, the code is easier to debug, you get more information when the parsing failed, and the library is usually better tested than our manual code.

To give you an impression how using a parser library looks like, I rewrote the example code using the better-parse project, which is an example for the parser-combinator approach:

```
val equationGrammar = object : Grammar<Equation>() {
    val ws by regexToken("\\s+", ignore = true)
    val reactsTo by literalToken("->")
    val reversibleTo 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<Boolean> by (reactsTo asJust false) or
            (reversibleTo asJust true)
    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, rhs, a) }
}

// calling an example string
val eq = equationGrammar.parseToEnd("3Ba(OH)2 + 2H3PO4 -> 6H2O + Ba3(PO4)2")
```

Going into the details of this specific 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 recognize the same pieces of grammar, which are assembled in a similar way as in our original code.

## 9.3. Conclusion

String-based DSLs allow to provide a very idiomatic syntax, and can also give the user some leeway by being more lenient than other DSL types. The price to pay is a lack of compile-time checks, less tooling support (e.g. autocomplete features), and from the implementation side the complexity and overhead involved with parsing, and the difficulty to extend them later.

### 9.3.1. Preferable Use Cases

- Creating data
- Transforming data

- Define operations
- Execute actions
- Generating code
- Testing

### 9.3.2. Rating

- ☀☀☼☼☼ - for Simplicity of DSL design
- ☀☀☀☀☀ - for Elegance
- ☀☀☀☀☼ - for Usability
- ☀☀☀☀☀ - for possible Applications

### 9.3.3. Pros & Cons

| Pros | Cons |
|---|---|
| • allows almost any syntax<br>• it is easy to allow for some leniency<br>• very flexible and extendable<br>• parser libraries help to write readable parser code | • no compile time checks<br>• no tooling support like code suggestions or autocomplete when using the DSL<br>• writing parsers must be learned<br>• having a dependency on a parser library<br>• difficult to extend at a later point in time<br>• hard to combine with other DSL types |

# Chapter 10. Annotation-based DSLs

Annotations can be used to tie specify actions or behavior to classes, methods, fields etc. Typical examples include dependency injection frameworks (like Spring or Dagger), transformers to formats like XML or JSON (like Gson, Jackson or JAXB), or authorization (like Spring Security). In Java, Project Lombok aims to reduce the boilerplate of the language in many ways, and uses annotations as well.

As mentioned in 4.10 Annotations, you might want to look into annotation processors, when you want to integrate your DSL somehow in the build process.

## 10.1. Case Study: Mapper DSL

Mapping between similar classes is a common and often tedious task. There exist already many solutions, e.g. the MapStruct 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. The actual transformation code is not for the faint of heart, it relies on reflection and is very unsafe in order to keep it short. Explaining the details is beyond the scope of this book, the focus of the case study lies on how to employ annotations for expressing your intentions.

Let's start with an example. Assume we have a `User` and a `Person` class, and that we want to transform users into 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 three cases:

1. the source and target parameters have the same name and type, like `firstName`
2. the source and target parameters have the same type, but different names, like `familyName` and `lastName`
3. the source and target parameters have different types, like `birthday` and `age`

In the first case, our mapper should just transfer the values. In the second case, we must specify how the names should be mapped. In the third case, we also need some kind of transformer. As you might have guessed, we will specify the required information using annotations. This is how our DSL could 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 need to be present at runtime, they should be only valid on classes, and it must be possible to use several of them (which requires the @Repeatable annotation). As the annotation fields cannot be null, we have 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 could be just a function from ZonedDateTime to Int. When a class is specified in the annotation which isn't a function (like the default value, which is Nothing), we will simply ignore it. As 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()
}
```

Now the only thing missing is the Mapper class, which needs to figure out which fields need to be mapped, and which transformers to use. As mentioned, it is neither pretty nor safe, but here it 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 args = 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)
            val hasTransformer = ann?.transformer?.isSubclassOf(Function1::class) ?:
false
            when {
                hasTransformer -> {
                    val transformer = ann!!.transformer.objectInstance
                        ?: ann.transformer.primaryConstructor!!.call()
                    transformer::class.memberFunctions
                        .find { it.name == "invoke" }!!
                        .call(transformer, sourceValue)
                }
                else -> sourceValue
            }
        }.toTypedArray()
        println(args.toList())
        return targetConstructor.call(*args) as T
    }
}
```

As it uses reflection, you need to include a dependency to the `kotlin-reflect` library, as described in chapter 4.11 Reflection. Of course, the example could be improved in many ways, e.g. sometimes you would need multiple source fields to calculate a target field.

**Convention over Code**

One design choice in our example deserves attention: When the names and types of the source and target classes match, no `@Mapping` annotation is needed. 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 be more often applicable to annotation based DSLs than to other DSL categories.

It is important to remember that not everything needs to be explicitly stated, and that common use-cases should seamlessly work "out of the box" whenever possible. By following the *Convention over Code* approach and providing a well-defined default behavior, which requires no additional effort, decision-making, or actions from the user, you can improve the user experience and enhance usability considerably.

For serious applications, I would suggest to give MapStruct a try. It is a Java library, but seems to work well with Kotlin, and has much more functionality than our example DSL. One main difference is that MapStruct generates sourcecode, avoiding the performance hit of using reflection, and making 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. Thankfully, one of these data types is `String`, and the last chapter showed how expressive string-based DSLs can be. Therefore, it is only natural to overcome the limitations of the annotation-based approach by embedding string-based DSLs in annotations.

Implementing such a DSL wouldn’t give much new insights, however, Spring Data JPA can serve as an example:

```
@Repository
interface UserRepository : JpaRepository<UserEntity, Long> {
    @Query("SELECT u FROM UserEntity u WHERE u.last_name = :lastName")
    fun findAllByLastName(@Param("lastName") lastName: String):
       List<UserEntity>
}
```

The `@Query` annotation doesn't have fields for the `FROM` and `WHERE` clause, it allows to specify the whole query as a string (which is a DSL itself). In my opinion, this is clearly the better approach for this use-case.

## 10.3. Conclusion

In some cases, it can feel very natural to integrate a DSL in 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 substantial. Another problem can be overusing annotations to the point of unreadability, and using annotations from different frameworks on the same class, method or property, which can be very confusing.

### 10.3.1. Preferable Use Cases

- Creating data
- Transforming data
- Execute actions
- Generating code
- Configuring systems
- Testing
- Logging

### 10.3.2. Rating

- ☀☀☼☼☼ - for Simplicity of DSL design
- ☀☀☼☼☼ - for Elegance
- ☀☀☀☀☼ - for Usability
- ☀☀☀☼☼ - for possible Applications

### 10.3.3. Pros & Cons

| Pros | Cons |
|---|---|
| • usage can feel very natural and intuitive<br>• uses a dedicated syntax<br>• can be a good way to mark exceptions (e.g. "don’t serialize this field") | • pollutes the host code<br>• can’t be used for external code<br>• can clash with other annotation-based DSLs<br>• relies often heavily on reflection<br>• hard to debug in case of problems |

# Chapter 11. Hybrid DSLs

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

When most of a DSL can be written in a certain style, it is usually a good idea to stick with it in order to ensure a coherent look and feel. So I wouldn't advocate to mix builder and loan pattern styles together, but to decide on one. But there is nothing wrong with designing a DSL which combines different approaches, when 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 imitate - to a degree - natural language. I would call this important subgroup of Hybrid DSLs "quasi-lingual". Even though the syntax is usually rigid and contains artifacts like braces, parenthesis or commas, it is still "readable" in a very literal sense. This snippet from the mocking library MockK should give you a good impression:

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

Technically, writing a quasi-lingual DSL is not very different from other Hybrid DSLs, although it might be harder to model grammatical structures correctly. Also, name clashes 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. E.g. instead of trying to model words like `should be` separated, it might be advisable to use `shouldBe` instead, when it helps to keep the grammar simple.

## 11.2. Case Study: Chemical Equations

Writing a DSL for chemical equations is no easy task, because the notation is very concise and can't be imitated easily using Kotlin syntax. For our case study, we will attempt to recreate the string-based DSL from chapter 9 using a hybrid approach. We will use the same underlying model classes:

```
sealed interface Part

data class Element(val symbol: String, val subscript: Int) : Part {
    override fun toString() = when (subscript) {
        1 -> symbol
        else -> symbol + subscript
    }
}

data class Group(val parts: List<Part>, val subscript: Int) : Part {
    override fun toString() = when (subscript) {
        1 -> parts.joinToString("", "(", ")")
        else -> parts.joinToString("", "(", ")") + subscript
    }
}

data class Molecule(val coefficient: Int, val parts: List<Part>) {
    override fun toString() = when (coefficient) {
        1 -> parts.joinToString("")
        else -> "$coefficient${parts.joinToString("")}"
    }
}

data class Equation(val leftSide: List<Molecule>, val rightSide: List<Molecule>, val
reversible: Boolean = false) {
    override fun toString() = leftSide.joinToString(" + ") +
            (if (reversible) " <-> " else " -> ") +
            rightSide.joinToString(" + ")
}
```

As a refresher, here is the related structure chart:

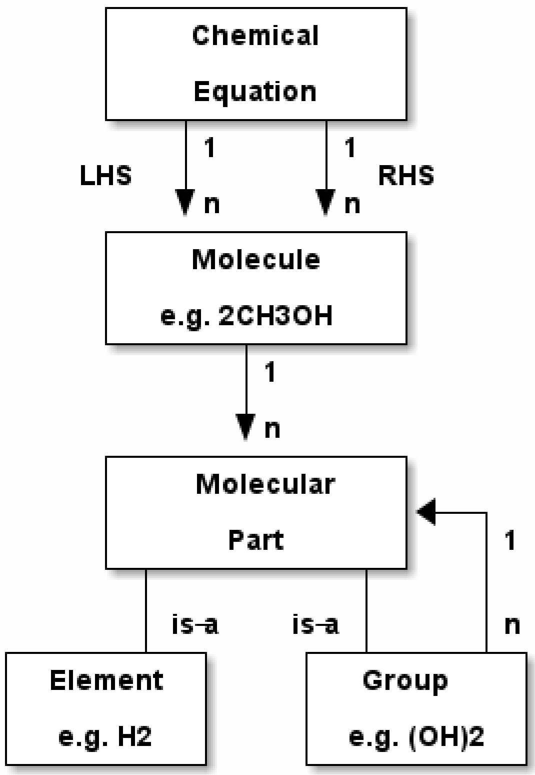

*Figure 4. Model for Chemical Equations*

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 a group. For this, we could use the invoke-operator `()` or the index access operator `[]`. As we will need parentheses in other places as well, 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 contains a sanity check, which 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`. We also need to assemble groups, and we have to distinguish them from molecules, so we need to use a different operator for the bonds inside groups. From the few remaining choices the range operator `..` seems like a good fit, so we can write e.g. a carboxyl group as `(C..O..O..H)`:

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

operator fun Molecule.minus(that: Part) =
    copy(parts = parts + that)

operator fun Element.rangeTo(that: Part) =
    Group(listOf(this, that),1)

operator fun Group.rangeTo(that: Part) =
    copy(parts = parts + that)
```

Generally, we have to put groups and sometimes molecules in parentheses because of the precedence rules.

A molecule can have an optional coefficient in front. Also, the same multiplication operation should "promote" an element or group to a molecule, allowing e.g. to write `2*O[2]` resulting in an oxygen molecule with a coefficient of two. Again, we need sanity checks, rendering calls like `-2*H[2]` or `3*(2*O[2])` invalid:

```
operator fun Int.times(that: Molecule) =
    that.apply { require(coefficient == 1 && this@times > 1) }
```

```
        .copy(factor = this)

operator fun Int.times(that: Part) =
    Molecule(this, listOf(that))
        .apply { require(coefficient > 1) }
```

Next, we need a way to group the left and right side of an equation to 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 along with the intended use, so we won't need parentheses on this level.

```
operator fun Molecule.plus(that: Molecule) =
    listOf(this, that)

operator fun Molecule.plus(that: Part) =
    listOf(this, Molecule(1,listOf(that)))

operator fun Part.plus(that: Molecule) =
    listOf(Molecule(1,listOf(this)), that)

operator fun List<Molecule>.plus(that: Part) =
    this + Molecule(1, listOf( that))
```

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 in an equation. This is not complicated, but lengthy, because we might encounter not only lists of molecules, but single molecules or molecule parts on both sides of the equation. Further, we have to account for the two different equation types:

```
infix fun List<Molecule>.reactsTo(that: List<Molecule>) =
    Equation(this, that, false)

infix fun Molecule.reactsTo(that: List<Molecule>) =
    Equation(listOf(this), that, false)

infix fun List<Molecule>.reactsTo(that: Molecule) =
    Equation(this, listOf(that), false)

infix fun Molecule.reactsTo(that: Molecule) =
    Equation(listOf(this), listOf(that), false)

infix fun Part.reactsTo(that: List<Molecule>) =
    Equation(listOf(Molecule(1,listOf(this))), that, false)

infix fun List<Molecule>.reactsTo(that: Part) =
    Equation(this, listOf(Molecule(1, listOf(that))), false)
```

```
infix fun Part.reactsTo(that: Part) =
    Equation(listOf(Molecule(1,listOf(this))), listOf(Molecule(1,listOf(that))),
false)

infix fun Part.reactsTo(that: Molecule) =
    Equation(listOf(Molecule(1,listOf(this))), listOf(that), false)

infix fun Molecule.reactsTo(that: Part) =
    Equation(listOf(this), listOf(Molecule(1,listOf(that))), false)

// same functions for reversibleTo, just with
// an equation having reversible == true
```

Unfortunately, we have to resort to infix functions, as there seems to be no suitable operator available. A common trick is to use the backtick syntax to mimic an operator, but `` `->` `` and `` `<->` `` won't work: `<` and `>` are two of the very few characters that are not allowed in backtick syntax on the JVM.

So, how does our DSL look in action? Here are a few examples:

```
//2H2 + O2 <-> 2H2O
val makingWater =
    2*H[2] + O[2] reversibleTo 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]) reactsTo
        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) reversibleTo (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) reactsTo (Cu..(H[2]..O)[4])-S-O[4]
```

There is one optional improvement, which is more a matter of taste: We could add some extension properties for low subscripts 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) reactsTo
            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.

## 11.3. Case Study: Pattern Matching

Kotlin's `when` is certainly more versatile than Java's `switch`, but languages like Scala or Haskell go one 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 contents of the input.

In pattern matching, you define a set of patterns that describe the possible shapes or values that an input can take. These patterns can include literals, variables, data constructors, or even more complex patterns like lists or tuples. The language then matches the input against these patterns and executes the corresponding code block or 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 could look like this:

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

val p = Person("Andy", "Smith", 43)

// this is not Kotlin, but 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 allow for 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 the unary plus, like `+"Andy"`
- for numbers, unary plus can't be used, so we fall back to a syntax like `eq(42)`
- The arrow notation is not possible, we will use `then` instead
- Comparisons as well as `and` and `or` can be only infix functions, not operators
- The right sides should be only evaluated if needed, so we need lambda braces for lazy evaluation.
- Capturing variables on the left and using them on the right requires using a `val` for defining a "capture" pattern.
- `else` is a keyword, so `otherwise` is used instead. As it is not possible to determine at compile time whether the given conditions are exhaustive, the `otherwise` branch is mandatory
- In some cases, we need to provide generic type information

That's a rather long list, let's see how our example looks now using an attainable 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 doesn't look too bad, even if the `+` prefixes take some getting used to. The problem is that the choices for overridable operators in Kotlin are quite limited. That's why the unary plus became something like a standard for such use cases, and is used this way in the Kotlin documentation as well.

The core of the DSL is quite small. It consists of a matcher providing a context for keeping track of the result and defining the `then` and `otherwise` methods. Then 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 wrapping a given value. And 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 return value, and immediately extracts its content. So why doesn't the block simply 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, because this is the only obvious way to construct such a instance.

The unary plus as synonym for the `eq` pattern is defined directly in `Matcher`, in order to avoid name clashes 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-hand sides of the `then` infix functions. Most of them are quite 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 }
```

For comparing values, some type checks are needed in order 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 `all()`, `any()` and `none()` predicates 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) }
```

For capturing values we need a class implementing `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>()
```

For capturing values, you first define a val using the `capture<T>()` method. Then you can use it on the left-hand side of `then` as a pattern, which always succeeds, but also stores the value. On the right-hand side the value can be read from the val. The initial example for the syntax 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 missing pattern 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 every data class we want to use in a pattern, but it is straightforward to do so:

```
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 default pattern for all arguments is very useful, as it allows to call the pattern for the data class using named arguments, and to ignore the arguments you don't care about.

Even though writing such pattern classes is not hard, it becomes quickly annoying. 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. Regardless of the complexity of the topic, it wasn't that difficult to come up with a quite usable DSL, which demonstrates the power and expressiveness of Kotlin.

## 11.4. Conclusion

Writing high-quality hybrid DSLs can be a complex task, as it requires careful consideration and integration of different language features. It's important to recognize that not all language features may work seamlessly together, and in such cases, it is often better to stick with a consistent style throughout the DSL.

However, when done well, a well-designed hybrid DSL can effectively combine various techniques in a way that feels intuitive and organic. By leveraging the strengths of different language features and carefully designing their integration, you can create very powerful and expressive DSLs.

### 11.4.1. Preferable Use Cases

- Creating data
- Transforming data
- Define operations
- Execute actions
- Generating code
- Configuring systems
- Testing
- Logging

### 11.4.2. Rating

- ★★★☆☆ - for Simplicity of DSL design
- ★★★★☆ - for Elegance
- ★★★★☆ - for Usability
- ★★★★★ - for possible Applications

### 11.4.3. Pros & Cons

| Pros | Cons |
|---|---|
| • can support a wide range of problems<br>• allows to get creative with different techniques<br>• can get very concise by having many implementation options | • might look incoherent<br>• high perceptual complexity → steeper learning curve<br>• difficult to control and predict the outcome<br>• higher maintenance effort needed<br>• Java interoperability can be very challenging |

# Part III - Supplemental Topics

# Chapter 12. Code Generation for DSLs

Sometimes you are prototyping a DSL and find a nice syntax, but it turns out it would need a lot of boilerplate code to make it work. One common reason is a combinatorial explosion, or there are many classes needed which 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. If you work with Java, too, you might also check out its sister project JavaPoet. There is even some interoperability between the libraries.

## 12.1. Case Study: Physical Quantities

Let's assume you have already defined quantities for physical units, like seconds, square meters, or Watt:

```
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
```

You should always consider whether the precision provided by `Double` is sufficient for the kind of calculations required by your application, and switch e.g. to `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 are allowed to be added. There are techniques to make this safe, but would require the use of generics in Kotlin.

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

*Gradle (.kts)*

```
implementation("com.squareup:kotlinpoet:1.12.0")
```

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

```
fun makeAddition(kClass: KClass<out Quantity>) =
    FunSpec.builder("plus")
        .addModifiers(KModifier.OPERATOR)
        .receiver(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 use the fact that a sealed class - here `Quantity` - "knows" about its subclasses. As a result, we get a list of `FunSpec` instances, which represent functions, constructors, getters or setters. You can simply print the content in the console to see what the code would look like. Later we will add these instances to a `FileSpec`, which 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:

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

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

fun makeScalarMultiplication(kClass: KClass<out Quantity>) =
    FunSpec.builder("times")
        .addModifiers(KModifier.OPERATOR)
        .receiver(Double::class)
        .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 following 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 not only define `5.0.s` to give us `Second(5.0)`, but also `1.0.min` to give us `Seconds(60.0)`. And conversely, we want to be able to get from a `Second` instance not only the seconds, but also minutes etc. It is obvious that we need additional information to write the conversions for a certain unit: We need its name, the quantity class, 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.
)
```

In order to make the DSL slightly more readable, we will generate extension properties instead of extension functions, so we don't have parenthesis. The generating functions look like this:

```
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()
```

```
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 are 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 for a generated pair of conversions:

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

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

So far, we can already generate a lot of boilerplate code, but for the next task - the multiplication and division of quantities - it would be extremely tedious to write the necessary code manually, even for our modest example. When we have N physical units, the number of possible multiplications and divisions is of order $N^2$ (we won't implement all possible combinations, but it is 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 product type:

```
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)
)
```

Now we evaluate these equations both for multiplications and divisions. A slight complication is that we also want to add functions with the operands switched, but only if they have different types:

```kotlin
fun makeMultiplication(
    in1: KClass<out Quantity>,
    in2: KClass<out Quantity>,
    out: KClass<out Quantity>) = FunSpec
        .builder("times")
        .addModifiers(KModifier.OPERATOR)
        .receiver(in1)
        .addParameter("that", in2)
        .addStatement("return %T(this.amount * that.amount)", out)
        .build()

fun makeDivision(
    in1: KClass<out Quantity>,
    in2: KClass<out Quantity>,
    out: KClass<out Quantity>) = FunSpec
        .builder("div")
        .addModifiers(KModifier.OPERATOR)
        .receiver(in1)
        .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 how the generated functions look like:

```kotlin
public operator fun Newton.times(that: Meter) =
    Joule(this.amount * that.amount)
```

```kotlin
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)
```

In order to finish the DSL, we need to write the generated code in a file. For simplicity, will write it directly next to the generating file, but it is common to have separate directories for generated code. For convenience, I added two extension functions for `FileSpec`, which allow to add multiple properties or functions at once:

```kotlin
fun main() {
    FileSpec.builder("creativeDSLs.chapter_11", "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 quite straightforward. You use the different spec classes to assemble your code, and the `FileSpec` and `ClassSpec` classes allow you to write the file or class to the filesystem. Behind the scenes, KotlinPoet does a lot of work for you, e.g. managing imports or simplifying your code (e.g. turning function bodies with curly braces into expression syntax if possible).

With our generated DSL in place, we can now calculate physical quantities in a safe and convenient way, e.g.:

```kotlin
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 a way where you generate the code manually via the `main()` method when the DSL has changed. This is a simple approach when you know that code changes don't happen very often, but it can become cumbersome quickly when changes become more frequent. In the next section, we will discuss the use of an annotation processor instead.

## 12.2. Writing an annotation processor using KSP

There are two APIs for Kotlin annotation processors. 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.

Before deciding to write an annotation processor, it is important to understand how it works and what it's limitations are. You need at least two modules: One module containing annotations, related interfaces etc. you can use in your client code to invoke the annotation processor, and one module containing the annotation processor itself, which will be integrated in the build process to do things like code generation, reporting or tom provide tooling support. Often, a third module is added for testing purposes, as you not only want to have unit tests for the processor classes, but you also need to check whether the processor works as intended when building client code.

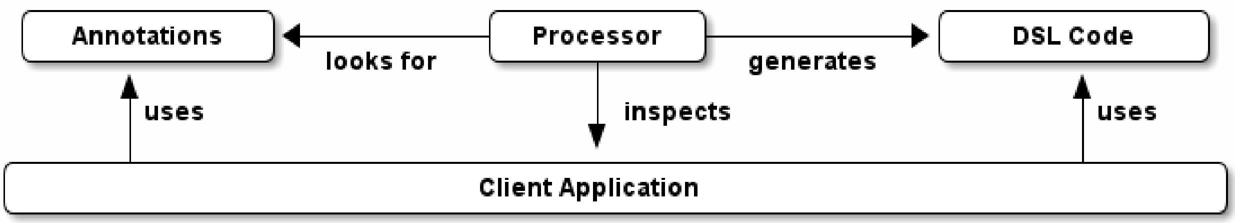

*Figure 5. Annotation processor using KSP*

At the time KSP is invoked, the client code is not yet built, which means **you can't use regular reflection** for client classes. The KSP API provides you with syntactical information about the code, but working with this API isn't as convenient as using reflection.

The missing reflection support means that KSP might be not the right tool if you need to rely heavily on code inspection, and that you should think about making the process of information gathering as easy as possible for the processor, e.g. by using annotations.

### 12.2.1. Case Study: Designing the DSL and writing the annotations module

How could an annotation based DSL for defining the relation between physical quantities look like? Now, we could annotate the `Quantity` interface with all the necessary information we had in the `fromToDouble` and `multiply` lists before:

```
@Conversion("s", Second::class, 1.0)
```

```
@Conversion("min", Second::class, 60.0)
@Conversion("h", Second::class, 3600.0)
...
@Multiply(Meter::class, Meter::class, SquareMeter::class)
@Multiply(Meter::class, SquareMeter::class, CubicMeter::class)
@Multiply(MeterPerSecond::class, Second::class, Meter::class)
...
sealed interface Quantity {
    val amount: Double
}
```

So we need two annotations in the annotation module, so that the client application can ust them, and the processor can react to them:

```
import kotlin.reflect.KClass

@Repeatable
@Target(AnnotationTarget.CLASS)
@Retention(AnnotationRetention.RUNTIME)
annotation class Conversion(
    val derivedUnit: String,
    val baseUnit: KClass<*>,
    val factor: Double
)

@Repeatable
@Target(AnnotationTarget.CLASS)
@Retention(AnnotationRetention.RUNTIME)
annotation class Multiply(
    val factor1: KClass<*>,
    val factor2: KClass<*>,
    val result: KClass<*>
)
```

### 12.2.2. Case Study: Writing the Annotation Processor

This book can give just a very high-level overview of KSP. Please consult the KSP Documentation for a more comprehensive discussion.

KSP uses Java's Service Provider Interface mechanism to discover new processors. That's why we need to write a provider first:

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

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

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

Now we need to write the processor. The basic structure looks like this:

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

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

To get the annotated elements, we can ask the resolver to provide them:

```
val conversions = resolver
    .getSymbolsWithAnnotation("creativeDsl.chapter12.ksp.annotations.Conversion")
    .filterIsInstance<KSClassDeclaration>()
val multiplications = resolver
    .getSymbolsWithAnnotation("creativeDsl.chapter12.ksp.annotations.Multiply")
    .filterIsInstance<KSClassDeclaration>()
```

## 12.3. Case Study: Generating Data Class Patterns

## 12.4. Conclusion

Using code generation means bringing out the big guns, it certainly requires some planning and setup. However, this technique allows you to implement DSLs that would be just too much overhead without. And with libraries like KotlinPoet, it is quite intuitive to generate the code you want. Kotlin-Poet is itself a nice example for a real-world DSL, and will be explored as such in the next chapter.

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

### 12.4.1. Preferable Use Cases

- Creating data
- Transforming data
- Define operations
- Testing

### 12.4.2. Rating

- ☀☀☼☼☼ - for Simplicity of DSL design
- ☀☀☀☀☀ - for Elegance
- ☀☀☀☀☼ - for Usability
- ☀☀☀☀☀ - for possible Applications

### 12.4.3. Pros & Cons

| Pros | Cons |
|---|---|
| • automatize writing of boilerplate code<br>• very flexible and adaptable<br>• intuitive libraries like Kotlin-Poet are available<br>• if the generator function is correct, so are all the outputs, e.g. no typos or copy-paste errors | • requires some up-front effort and setup<br>• strong dependency on the used library<br>• longer build times when generation is done for every build<br>• code can get out of sync when generation is done only on request |

# Chapter 13. Java Interoperability

Using a Kotlin DSL from Java can be challenging. While some Kotlin features like operator overloading simply can't be used, there are means to make other features more accessible from Java, and to create a DSL which is useful and convenient in both languages. Fortunately, Kotlin has some built-in features to improve Java interoperability, which can be found in the Kotlin Documentation: Calling Kotlin from Java. In this chapter, Java interoperability will be discussed purely from a DSL design perspective, by presenting the relevant built-in features and useful techniques for solving DSL-specific issues.

## 13.1. Top level Functions and Variable Definitions

Java doesn't allow 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 file name, including the `kt` ending, and following 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*

```
package com.acme

@file:JvmName("Utils")

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.

## 13.2. Value Classes and Mangling

In Kotlin, value classes are represented by their underlying type in the JVM bytecode. However, this approach can lead to naming conflicts, e.g. when there are multiple methods with the same name, and value class parameters with the same underlying type. To avoid this issue, 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 name the method explicitly 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);`.

# Chapter 14. Real-World DSL Examples

In this chapter we will have a look of some widely used and successful DSLs, characterize them, look at interesting details, and maybe even suggest some improvements.

## 14.1. KotlinPoet

KotlinPoet is a source code generator for Kotlin, and the sister-project to JavaPoet. We used it in chapter 11 to generate sources for operations with physical quantities.

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

*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 stand-alone library, I would rather use a Loan Pattern DSL instead:

```
val greeterClass = ClassName("", "Greeter")
val file = fileSpec("", "HelloWorld") {
    +classSpec("Greeter") {
        primaryConstructor = constructorSpec {
            +parameter("name", String::class)
        }
        +propertySpec("name", String::class) {
            initializer("name")
        }
        +functionSpec("greet") {
            +statement("println(%P)", "Hello, \$name")
        }
    }
    +functionSpec("main") {
        +parameter("args", String::class, VARARG)
        +statement("%T(args[0]).greet()", greeterClass)
    }
}
```

## 14.2. Gradle .kts

Gradle is an amazing build system. In contrast to descriptive approaches like the XML-based Apache Maven, 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 to write an expressive DSL in the Java ecosystem. Alas, 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 like autocompletion, content assistance, source navigation, quick documentation and refactoring support are limited.

To overcome these problems, 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, in order to make the transition as smooth as possible. I think, in this regard, the Kotlin DSL was a resounding success. Here is a comparison, taken from the Gradle Userguide: Migrating build logic from Groovy to Kotlin:

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

The Kotlin Script DSL itself uses mainly the Loan Pattern. For the definition of dependencies, there are two notations: The example shows the "string notation", which is a String-Parsing DSL. Alternatively, the "map notation" can be used, which looks like this: `implementation(group = "com.example", name = "lib", version = "1.1")`

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

## 14.3. kotest

## 14.4. better-parse

# Epilog

Congratulations on completing "Creative DSLs in Kotlin"! Over the course of 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 provided you with 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 certainly many skills and insights that can't be taught by a book, but only be learned through experience and dedication. Creativity, personal style, and the ability to make good design choices are all crucial 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 enjoyable.

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 wish that you will experience the same sense of joy and wonder on your own way. Thank you for joining me on this journey!

# Bibliography

The bibliography list is a style of AsciiDoc bulleted list.

*Books*

- [taoup] Eric Steven Raymond. 'The Art of Unix Programming'. Addison-Wesley. ISBN 0-13-142901-9.
- [walsh-muellner] Norman Walsh & Leonard Muellner. 'DocBook - The Definitive Guide'. O'Reilly & Associates. 1999. ISBN 1-56592-580-7.

*Articles*

- [prokopov] Nikita Prokopov: 'Designing good DSL', https://tonsky.me/blog/dsl/

# Index