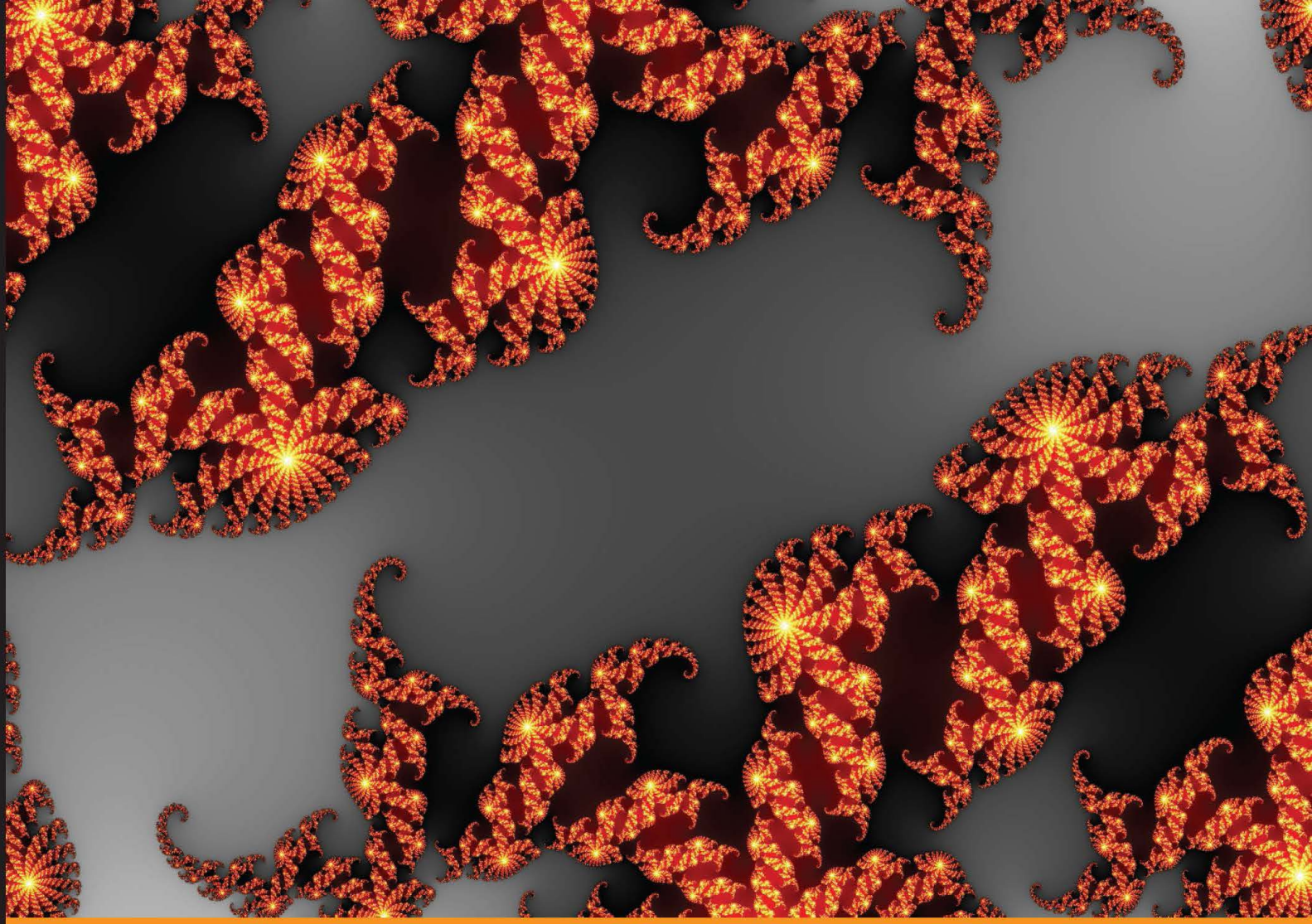

Community Experience Distilled

Clojure for Domain-specific Languages

Learn how to use Clojure language with examples and develop domain-specific languages on the go

Ryan D. Kelker

[PACKT] open source* community experience distilled PUBLISHING

Clojure for Domain-specific Languages

Learn how to use Clojure language with examples and develop domain-specific languages on the go

Ryan D. Kelker

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Clojure for Domain-specific Languages

First published: December 2013

Production Reference: 1111213

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK..

ISBN 978-1-78216-650-4

www.packtpub.com

Cover Image by Sheetal Aute (sheetala@packtpub.com)

Credits

Author
Ryan D. Kelker

Reviewers
Hussein Baghdadi
Paulo Suzart
Jon Vlachogiannis
Arthur Ulfeldt

Acquisition Editors
Kunal Parikh
Meeta Rajani

Lead Technical Editor
Arun Nadar

Technical Editors
Gauri Dasgupta
Kapil Hemnani
Mrunmayee Patil

Copy Editors
Alisha Aranha
Dipti Kapadia
Sayanee Mukherjee
Karuna Narayanan
Alfida Paiva
Kirti Pai

Project Coordinator
Kranti Berde

Proofreader
Stephen Copestake

Indexer
Rekha Nair

Graphics
Abhinash Sahu

Production Coordinator
Kirtee Shingan

Cover Work
Kirtee Shingan

About the Author

Ryan D. Kelker is a Clojure enthusiast and works as a freelance—he is willing to take on any project that sounds interesting. He started exploring computers and the Internet at a very early age and he eventually ended up building both machines and software. Starting with MS DOS, batch files, and QBasic, he eventually floated towards Arch Linux and the Clojure language.

He has four certifications from both CompTIA and Cisco, and has decided not to pursue any additional certifications. These days, he spend most of his time reading about software development, cyber security, and news surrounding up-and-coming computer languages. While away from the computer, he is usually reading a book or going out to eat with the people he loves the most.

I would like to thank Packt publishing for giving me the opportunity to write something great, and `Chatsubolabs.com` for giving me my first Clojure job. I would also like to thank Tom Marble, Edward Raison, and Kevin Raison for teaching me countless lessons.

About the Reviewers

Hussein Baghdadi is a programming-language junkie. Switching between programming paradigms is his favorite trampoline. He worked in various domains from Telecommunications to e-commerce, passing through Big Data systems and independent software providers. His current areas of interest are machine learning, natural language processing, and game development. His favorite language is Clojure. He works as a moderator at `JavaRanch.com`. When he isn't coding, reading, or talking to himself, he plays the Spanish guitar.

First, I would like to thank Pack Publishing for the great opportunity they gave to me to be a small part in this unexpected journey.

I want to thank all the employers who showed faith in me and allowed me to push code into their SCMs.

Thank you Rich Hickey for creating such a beautiful language. Thank you Sun Microsystems for conjuring the JVM.

Thank you my real friends. You have always helped me and enriched my life. Your support is making the plane fly.

Special respect and gratefulness for my virtuoso guitar teacher who showed to me what a real master (and a real human) is made of.

My parents gave me unconditional love and taught me honor and dignity. My unconditional love goes to them.

Finally, thank you Syria, the Netherlands, and Deutschland.

Paulo Suzart worked in different companies in the last ten years—from e-commerce to insurance—as a Java programmer and lately as an SOA specialist. Currently, he runs a digital media start-up as CTO.

He truly believes that start-ups are open fields for new technologies and functional-programming languages such as Scala and Clojure.

Jon Vlachogiannis is a Senior Executive with a track record of more than 15 years of successfully managing complex, high-risk, high-value projects—from conceptualization through implementation, launch, and product development.

When not giving tech presentations around the world, Jon uses C and Erlang to design and build databases for Big Data analysis, specializing in processing enormous amounts of data in real time.

He designed LDB that powers his mobile data analysis company—BugSense—and processes real-time data from more than 520 million mobile devices around the world. In his spare time, he creates programs that create programs and music in Clojure such as music-as-data, and runs algorithms on FOREX (mostly HFT).

Jon started programming at the age of eight. He has worked for and with leading companies and agencies in the public and private sectors including P&G and NATO. His contributions range from photorealism algorithms to digital signage to geocoding applications to highly-scalable distributed systems.

Jon can also be found teaching Python to the United Nations in order to identify nuclear explosions, running an algorithmic fund for start-ups, or skating around the globe.

Arthur Ulfeldt has been an enthusiastic member of the Clojure community since shortly after the language was released, and he works with Clojure full time on both web development and infrastructure projects. He is a frequent speaker at various Clojure User Groups and devotes a great part of his time to helping people learn Clojure.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Clojure for Domain-specific Languages is an example-oriented guide to building custom languages. Many of the core components of Clojure are covered to help you better understand your options when making a domain-specific language. By the end of this book, you should be able to make an internal DSL.

What this book covers

Chapter 1, *An Overview of Domain-specific Languages with Clojure*, will help you learn specifically what a domain-specific language (DSL) is and why you may use one. This will include a comparison of many existing DSLs, both in Clojure and other languages.

Chapter 2, *Design Concepts with Clojure*, will go over some basic concepts that apply to software development in any programming language. Each section will explain what the concept is and why the concept should be applied to your projects. As with all sources of information, choose what works for you.

Chapter 3, *Clojure Editing and Project Creation*, will help you get started with the Emacs text editor and the Lein project utility. Because entire books can be written on either of the software discussed in this chapter, each section will only go over the basics to help you get started.

Chapter 4, *Features, Functions, and Macros*, briefly goes over some of Clojure's key components. More specifically, Clojure namespaces, Java classes, immutability, metadata, lazy sequences, collection destructuring, functions, relationships, and macros.

Chapter 5, *Collections and Sequencing*, specifically focuses on Clojure's collection data structures and ways to construct, use, and manipulate them. Each section will focus on a certain data structure, and the end of the chapter will focus more on operations that can be used on sequences.

Chapter 6, *Assignment and Concurrency*, starts off by getting more hands-on with variables and how to manipulate them. The variable section explains the most common variable definition methods and functions for handling Clojure variables. It's also okay to skip sections if you feel that you already know the material well enough.

Chapter 7, *Flow Control, Error Handling, and Math*, starts off by explaining and displaying examples of common flow control methods, then it moves on to object comparison and casting. We will also be covering error handling and, finally, we will move on to the *Arithmetic* section that contains a lot of surprises for those who aren't familiar with Clojure. After reading all of the sections, remember to try some of the examples in a REPL session while reviewing the chapter.

Chapter 8, *Methods for Abstraction*, starts off with an explanation of the classes we need for the multimethod polymorphism tutorial. By the end of this chapter, you should be familiar with making classes, polymorphic functions, and your own data types.

Chapter 9, *An Example Twitter DSL*, is focused on building a Twitter DSL but it starts off by covering some of the key concepts of building layers of Clojure code on top of Java libraries. Some concepts you may already be familiar with, but you're encouraged to read the initial part of this chapter to better understand the concepts presented in this chapter.

Chapter 10, *Unit Testing*, will briefly cover the `clojure.test`, `expectations`, `midje`, and `speclj` unit testing frameworks. Each framework will be used to make tests for the Twitter mini-DSL created in the last chapter.

Chapter 11, *Clojure DSLs inside Java*, will help you learn about generating Java classes and making them available in your Java project. The first half of this chapter will cover how to build a Clojure class that can be called from both Java and Clojure. The second part will cover the importing and use of Clojure source files from within Java.

What you need for this book

You will need the following software for this book:

- **Clojure**: Clojure 1.5.x
- **Leiningen**: Leiningen 2.x (Optional, but heavily used in the book. You can still use the examples with the project manager of your choice.)
- **Emacs**: Emacs 24.x (Optional, but required for *Chapter 3*, *Clojure Editing and Project Creation*.)

Who this book is for

If you've already developed a few Clojure applications and wish to expand your knowledge on Clojure or domain-specific languages in general, this book is for you. If you're an absolute Clojure beginner, you may only find the detailed examples of the core Clojure components of value. If you've developed DSLs in other languages, this Lisp- and Java-based book might surprise you with the power of Clojure.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The anonymous function for the `setup` option declares that the image should use an anti-aliasing filter with the function named `smooth`."

A block of code is set as follows:

```
(def right (atom 55))

(defsketch drawing
  :title "Book Example"
  :setup (fn []
           (smooth)
           (frame-rate 3))
  :draw (fn []
          (let [x2 (do (swap! right inc)
                       @right)]
            (stroke-weight 9)
            (line 50 100 x2 100)))
  :size [200 200])
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
user> (defmacro example4 [& args]
        `'(println ~args))
#'user/example4
user> (example4 1 2 3)
(clojure.core/println (1 2 3))
```

```
user> (defmacro example5 [& args]
       `'(println ~@args))
#'user/example5
user> (example5 1 2 3)
(clojure.core/println 1 2 3)

user> (defmacro example6 [s]
       `(let [s-name# ~s]
          (println 's-name# "Binding holds" s-name#)))
#'user/example6
user> (example6 "String")
s-name__28785__auto__ Binding holds String
nil
```

Any command-line input or output is written as follows:

```
├── doc
│   └── intro.md
├── project.clj
├── README.md
├── resources
├── src
│   └── test_project
│       └── core.clj
└── test
    └── test_project
        └── core_test.clj
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Click on the **Details** tab of your application's API page and then click on **Create my access token** at the bottom of the page."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
An Overview of Domain-specific Languages with Clojure

In this chapter, you'll learn specifically what a domain-specific language (DSL) is and why you should use one. This will include the comparison of many existing DSLs both in Clojure and other languages.

This chapter will cover examples for the following domains:

- Clojure libraries
- Database domain
- HTML domain
- ECMA/JavaScript domain
- Audio domain
- Image domain

This chapter will also go over a few internal and external DSLs, but this book is aimed at the development of an internal DSL. That doesn't mean the information presented in this chapter or book can't be applied to the development of an external DSL. After learning the pros and cons of external and internal DSLs, examples of both will be presented and explained.

After learning the difference between a Clojure library and a Clojure DSL, this chapter will cover several problem domains. Each domain has several examples surrounding different DSLs and compares the solutions of each example. This includes comparing non-Clojure solutions to Clojure-based solutions.

By the end of this chapter, you should have an understanding as to when and where to use a DSL.

Domain-specific languages (DSL)

A **domain-specific language** (**DSL**) is the opposite of a general-purpose language in the sense that it's designed from beginning to end to solve a very specific set of problems. A DSL can only help solve a single set of problems and is limited by design. The limits of the language are not bound to the expressiveness of the language but to the scope of the problems it can handle. Now we will look at this in more detail.

Limited scope

The limited scope of problems that a DSL can handle is one of the key differences between a general-purpose language and a domain-specific language. It's important for the DSL to remain within its problem domain. For example, a language specifically designed for image manipulation shouldn't contain the expressive ability to manipulate audio, and vice versa.

Syntax

The way you would think about solving a problem in a specific problem domain is the same way you would express the solution in the DSL. Some DSLs can fall short in some way, but a solid DSL usually covers most, if not all, aspects of a problem domain and is developed gradually from the ground up. In general, the concept is to make the complex simpler in terms of literal expression.

Making the complex simpler doesn't necessarily mean to simplify in terms of a DSL. The reason to use a DSL is to separate the user from the complexity of the problem domain. Using a DSL can simplify the solution-building process, but the DSL doesn't actually change the problem in the problem domain.

All well-written DSLs should be recognizable by a person familiar with the problem domain. If you were to write a language specifically designed to handle the process of ordering food, assuming the user knows how to make an order, a nontechnical user should be able to understand the syntax of the language without any knowledge of the language's complexities. Nevertheless, depending on the DSL's requirements and the intended user audience, some knowledge of the parent language may be required for the end user to fully understand the expressions of the DSL.

Syntax might be the first thing you see in any language, but that's not the real difference between a good language and a not-so-good language. A good DSL makes expressing a specific solution easier in comparison to expressing the same solution in a general-purpose language.

Using a DSL

For instance, if we want to build a website, we'll probably need a web language for a much simpler development process. If we want to express something in the web browser, we'll probably write a web page in HTML. If we want to add or modify the styles of our web page, we'll use CSS. Then, if we wish to add event-related functionality, we would use ECMAScript (also known as JavaScript) to do so.

Popular DSLs

DSL	Domain
Structured Query Language (SQL)	Databases
HyperText Markup Language (HTML)	Websites
Postscript	Publishing

Websites are some of the best examples of DSLs working together in concert. The dynamic results of a page are often rendered by languages and frameworks such as PHP, Node.js, Ruby on Rails, or Django. The content of the page is often retrieved from a database with SQL, SQL **object-relational mapping** (**ORM**), or a solution-specific NoSQL DSL. When the dynamic page is requested, the language or framework decides what to display and what not to display. If the page requires content from a database, the language or framework will use a database DSL to get the requested content. Then, the retrieved content is formatted with the DSL HTML. The HTML itself may contain other DSLs such as ECMAScript and/or CSS. Many web languages often have their own templating system that can also be its very own DSL.

A contract between language and domain

We can also think of DSL as an agreement or a protocol. The user agrees to be ignorant of its inner workings and not of the problem, and the DSL agrees to ignore everything not related to the problem. This agreement allows for the DSL to act as a loyal intermediary between the solution and the problem.

A better way to understand this protocol or agreement is to think about your everyday activities. For example, if you were to pick up the phone and order a pizza, you would be speaking a DSL. The problem is that you need a pizza, and the solution can only be expressed in a contextual manner for that very specific problem. This is the agreement of that DSL. Now, if you were to go into a video game store and try to order a pizza, the people there may understand the language, but you wouldn't get a pizza, because that domain is outside the agreement of that type of language.

The language of trust

The user must be able to trust that the DSL is doing what it says it's doing in the expressions. The actions of the DSL, either hidden or verbose, must be in direct relation to what's explicitly being expressed to avoid unwanted side effects or mutations. For example, you wouldn't want to use a DSL if a statement such as *roll the ball* actually did more than just rolling the ball.

Internal versus External DSLs

Although many DSLs strive to do the same thing, not all DSLs are built the same. Many try to hide as much of the host language as they possibly can get away with, and some are completely outside the host language. Then there's the in-between, where the host language is used for leverage and becomes a power tool for the user instead of a burden.

The use of a DSL is restricted to the way it's implemented. A DSL that requires itself to be embedded within the host language can be called an **Internal DSL**. A DSL with its own syntax and that isn't required to be embedded within another language can be called an **External DSL**.

Regardless of the type of DSL we may end up building, the concept remains the same. We need to separate the user from the complexities of the problem without separating the user from the problem itself. We'll look at the major differences between an Internal and an External DSL, although this book will mostly focus on Internal DSL development with Clojure. That doesn't mean that the information expressed in this book can't be applied to External DSL development or to the development of other languages as well.

External DSLs

An External DSL is like making a completely new language from beginning to end. The language itself is separate from your development language and requires many programming concepts found in the development of a general-purpose high-level language. Some of the main programming concepts of an External DSL encompass other major concepts such as code generation, compilation, symbol parsing, and language interpretation.

There are many reasons to build and use an External DSL. You might need a language with a simpler or more expressive syntax than what the host language is able to provide. You might also need a language that is completely separated from the host language.

> *Martin Fowler, in a 2005 blog article about language tools, once stated that some XML configuration files can be thought of as an External DSL that uses XML to help simplify the parsing process.*

His point rings true in the sense that an XML configuration file uses a special grammar that's independent from the host language parsing the configuration file.

One of the many positive things about an External DSL is that you can choose the environment in which the code can be executed. Another advantage is that you essentially have an unlimited amount of expressiveness that can be built into the language because you're not restricted by the host language used to build the parser. This type of freedom allows for a custom syntax that the host language wouldn't be able to provide.

Although the custom syntax or grammar of your External DSL can make life easier in some ways, there are a few pitfalls that must be taken into consideration if you wish to develop an External DSL. Developing a new language that is separate from the host language means that the features of your language are restricted to the capabilities of your parser. Another major concern might be the lack of utilities that support the use of your language. For example, existing **Integrated Development Environments** (**IDE**s) won't be able to give you many common features such as syntax highlighting and support for enhanced error exception handling. You'll also have to figure out a practical way to handle, warn about, and report errors whenever they might occur.

Clojure has many language-building friendly features that can transform the structure of the Clojure language itself to fit most of your expressive needs. This brings us back to the concept of an Internal DSL and how we can use Clojure as just another building block.

Internal DSLs

An Internal DSL is essentially a language that is either built on top of an existing language or extends another language, and is most commonly packaged as a language library. As with an External DSL, an Internal DSL is written in a very specific grammar, but unlike the External DSL, the grammar is restricted to the legal syntax of the host language. This means that your DSL will instantly adopt all the nice and not-so-nice syntactical features.

Adopting the host language's syntactical features means that you'll have to design your language in a way that also makes sense in the host language. This also means that all of the features and problems of the host language will have great influence on how your DSL will be implemented. An Internal DSL instantly gives us a great advantage over an External DSL in the sense that we have tools that can already understand our language, because we are using a language that has to follow the same syntax rules of the host language. Another great advantage is that our language can easily be integrated with existing projects of the host language. Error handling and reporting are also simplified in many ways, because the Internal DSL can just hitch a ride on the host language's error-handling methods. We can also see how an Internal DSL might greatly decrease the learning curve for a group of developers who already know the host language of the DSL.

Using an Internal DSL is a nicer way of interacting with a specific problem domain from within a general-purpose language. This allows for the host language to act as an intermediary between other Internal DSLs that might be required for a completely different set of problems within the project. For example, you can query results from a SQL database with a database DSL and use the host language to format the query results, so that another DSL can handle the query results properly.

If it's not already obvious, you generally wouldn't use an Internal DSL when the problem you're trying to solve can be reasonably expressed without much effort in your general-purpose language. For example, fetching data from a URL probably wouldn't require a DSL, but being able to safely handle and manipulate a PDF file type may very well require a DSL. Every general-purpose language has its strong and weak points, like all languages, so where you might need an Internal DSL will mostly depend on how well the host language can generically handle the problem.

Some Internal DSLs are actually built on top of an existing set of Internal DSLs and libraries to bring forth the overall functionality of the language. Usually, language libraries are often bundles of well-established libraries and DSLs working together in concert to handle the actions of the parent library's methods. Not every DSL or library will depend on another DSL or library, but development time can be greatly decreased by using existing code instead of starting from nothing.

Clojure libraries

Many libraries in Clojure could be classified as an Internal DSL. Clojure's LISP-based syntax, and the unique ability to define variables and methods with nonalphabetical characters, makes many Clojure libraries feel like their own language. The difference between a Clojure library and an Internal DSL isn't all that much.