

C o m m u n i t y E x p e r i e n c e D i s t i l l e d

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*
PUBLISHING 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



BIRMINGHAM - MUMBAI

Clojure for Domain-specific Languages

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

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

Project Coordinator

Kranti Berde

Reviewers

Hussein Baghdadi

Paulo Suzart

Jon Vlachogiannis

Arthur Ulfeldt

Proofreader

Stephen Copestake

Indexer

Rekha Nair

Acquisition Editors

Kunal Parikh

Meeta Rajani

Graphics

Abhinash Sahu

Lead Technical Editor

Arun Nadar

Production Coordinator

Kirtee Shingan

Technical Editors

Gauri Dasgupta

Kapil Hemnani

Mrunmayee Patil

Cover Work

Kirtee Shingan

Copy Editors

Alisha Aranha

Dipti Kapadia

Sayanee Mukherjee

Karuna Narayanan

Alfida Paiva

Kirti Pai

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	1
Chapter 1: An Overview of Domain-specific Languages with Clojure	7
Domain-specific languages (DSL)	8
Limited scope	8
Syntax	8
Using a DSL	9
Popular DSLs	9
A contract between language and domain	9
The language of trust	10
Internal versus External DSLs	10
External DSLs	10
Internal DSLs	11
Clojure libraries	12
The characteristics of a Clojure library	13
The current state of Clojure libraries	13
Database domains	14
The HTML domain	17
Formative	18
Hiccup	19
Mustache	20
Clostache	20
The ECMA/JavaScript domain	22
ClojureScript	22
Comparing ClojureScript and JavaScript	23
The Audio domain	26
Music-as-data	26
Overtone	27

Image domains	29
Summary	31
Chapter 2: Design Concepts with Clojure	33
Every function is a little program	33
A pure function	34
Floor-to-roof development	34
Each function only does one thing	34
Patterns for success	34
DRY	35
KISS	35
YAGNI	35
Writing Clojure	35
Spacing and alignment	36
Syntax	40
Name conventions	46
Collection types	47
Summary	48
Chapter 3: Clojure Editing and Project Creation	49
The origin of Emacs and its usage	49
Installing and setting up Emacs	50
Setting up Emacs	51
Creating and editing CLJ files in Emacs	53
Running a Clojure REPL inside Emacs	56
The nrepl.el Emacs extension	56
Leiningen and project management	61
Installing Leiningen and starting a project	61
Including Clojure or Java libraries in your project	64
Compiling your project to a Java JAR	65
Leiningen	66
Summary	67
Chapter 4: Features, Functions, and Macros	69
Namespaces	69
Java inside Clojure	72
Immutability	75
Dynamic objects	76
Metadata	77
Lazy sequences	79
Destructuring	80
Functions and arity	82
Anonymous functions	85

Macros	86
Summary	88
Chapter 5: Collections and Sequencing	89
Collections	89
Collections by example	90
Vectors	94
Vectors by example	94
Lists	96
Lists by example	96
Maps	97
Maps by example	97
Sets	103
Sets by example	103
Sequences	104
Sequences by example	104
:let, :while, and :when	116
Summary	117
Chapter 6: Assignment and Concurrency	119
Variables	119
Transients	125
Atoms	126
Agents	128
Refs	134
Futures	135
Promises	137
Summary	138
Chapter 7: Flow Control, Error Handling, and Math	139
Flow control	139
Object comparison	145
Casting	145
Error handling	147
Arithmetic	148
Addition and subtraction	148
Multiplication	148
Division	149
Remainder and modulus	149
Increment and decrement	150
Greatest and least values	150
Equality	151
Summary	152

Chapter 8: Methods for Abstraction	153
Creating and constructing classes	153
Creating interfaces and implementing them with deftype	153
Using records, protocols, and type extensions	154
Overriding methods with reify and proxy	158
Working with reify	158
Implementing interface methods with proxy	160
Custom symbol definitions with macros	161
Definitions using records	161
Making definitions using proxy	163
Making definitions using deftype	164
Multimethod polymorphism	164
Creating the Bottle and Customer classes	166
Testing the customer-drink methods	167
Relationships with hierarchies	169
Resolving parent relationship conflicts	172
Assertion testing with metadata	174
Input constraints with :pre	175
Output constraints with :post	175
Summary	176
Chapter 9: An Example Twitter DSL	177
Creating Java-based abstractions	177
Making Java objects easier to manipulate	178
Retrieving values in a better way	179
Examples of our Twitter DSL	180
The Retweet bot	180
Creating an event notifier	181
Reading the OAuth configuration	181
Twitter account registration and application keys	181
Adding required dependencies	182
Creating the project and API configuration	183
Reading the Twitter configuration	183
Making our most important macro	186
Building the deftwitter macro	188
Building the twitter-> macro	191
Handling search queries	193
Adding the tdsl.search namespace	193
Search macros and functions	193
Handling tweets	198
Adding the tdsl.tweet namespace	198

Tweet macros and functions	198
Adding user-related features	201
Adding the tdsl.user namespace	202
User macros and functions	202
User details and multimethods	203
Adding logging features	206
Summary	209
Chapter 10: Unit Testing	211
Exploring the clojure.test framework	211
Testing tdsl.core	212
Using the is macro	213
Using the are macro	214
Developing the final test	215
The expectations framework	217
Using the expect macro	217
Search testing	218
The midje framework	221
Using the fact macro	222
The specj framework	224
Using the describe, it, should, and should= macros	225
Using the should-contain macro	226
Summary	228
Chapter 11: Clojure DSLs inside Java	229
Making a Java-callable Clojure class	229
Class naming	229
Data hiding	230
AOT – the ahead-of-time compilation	231
Java-wrapping your Clojure	232
Summary	234
Appendix	235
Chapter 1: An Overview of Domain-specific Languages with Clojure	235
Chapter 2: Design Concepts with Clojure	237
Chapter 3: Clojure Editing and Project Creation	238
Chapter 4: Features, Functions, and Macros	239
Chapter 5: Collections and Sequencing	239
Chapter 6: Assignment and Concurrency	240
Chapter 7: Flow Control, Error Handling, and Math	240
Chapter 8: Methods for Abstraction	240
Chapter 9: An Example Twitter DSL	241

Table of Contents

Chapter 10: Unit Testing	241
Chapter 11: Clojure DSLs inside Java	241
Index	243

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 (IDEs)** 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.

The characteristics of a Clojure library

As with most programming languages, Clojure has multiple library solutions for certain classes of problem domains. Some libraries may be a Clojure wrapping of an existing Java library or domain-specific language. It's perfectly acceptable to build a Clojure library or DSL based completely on an existing Java library, but there are some drawbacks that must be kept in mind when using a library of this type. Depending on how the library returns objects, generally speaking, the objects have the characteristics of a Java object and not of a Clojure object.

Pure Clojure library objects are immutable by default, as opposed to Java's default objects being mutable. This means that Java objects welcome mutation or changes without instantiating a new object reflecting the side effects of the applied mutations. This can cause a problem if you're unaware that you're dealing with a mutable object, and you try to treat it as a normal Clojure object. Generally speaking, you'll know when you're dealing with a pure Java object because you'll probably have to use Java interoperation notation to make method calls specifically related to the object type.



Clojure libraries, in comparison to other languages, are significantly smaller in terms of line count. There are many reasons for this, but some of the key reasons are that Clojure's hyper-powerful functions, macros, and sequence handling features can greatly reduce tasks that usually span over several lines in most other languages.

The current state of Clojure libraries

Clojure is still a young language, so Clojure's best development practices are still evolving. There's currently a project that aims to provide a collection of well-documented and feature-complete libraries that are compatible with all versions of Clojure at and beyond Version 1.3.0. The project is called **Clojurewerkz** and can be found at <http://clojurewerkz.org/>.

Clojure's standard library is quite large and is growing increasingly with every new version. You can view the latest and other versions of the Clojure standard library documentation at <http://clojure.github.com/clojure/>. There's also a community-driven website dedicated to example-based documentation for Clojure's standard and contributed libraries at <http://clojuredocs.org/>. You can also view the source code of the Clojure language at the following GitHub page: <https://github.com/clojure/clojure/>.

Database domains

You could argue that SQL libraries are probably the most popular and widely used form of an Internal DSL. Because of this, we're going to look at the database problem domain with a Clojure Internal DSL library called **Korma**. If this book doesn't cover Korma as much as you would have liked, you can visit the official website for more details at <http://sqlkorma.com/>.

To fully appreciate a feature-rich DSL, we have to understand the complexities under the hood. For example, this is what a raw SQL query may look like:

```
SELECT name
FROM customer
WHERE id = 10
```

Now let's look at this same SQL query with a Clojure-based Internal SQL DSL:

```
(select customer
  (fields :name)
  (where {:id 10}))
```

Looking at the two DSL code examples side by side in the following table, what conclusions can we draw from this?

DSL for the database domain	Clojure DSL for the database domain
SELECT name	(select customer
FROM customer	(fields :name)
WHERE id = 10	(where {:id 10}))

Here are four points that I think are very important to keep in mind when structuring a language:

- Some DSLs are based on other DSLs such as SQL
- The syntax of a DSL doesn't always make the process shorter or longer in terms of line count
- The DSL can't (or really shouldn't) remove the nomenclature of the problem domain
- Depending on the type of operations being performed, a DSL may be more verbose in some very specific circumstances

Now let's look at another example in the following table where the difference is mostly the line count:

DSL for the database domain	Clojure DSL for the database domain
<pre>INSERT INTO customer (name, age) VALUES ('Hickey', 200)</pre>	<pre>(insert customer (values {:name "Hickey" :age 200}))</pre>

Comparing SQL's `SELECT` statement with Korma's `SELECT` statement, the two weren't all that different. The same could even be said about the `INSERT` statements of SQL and Korma. So why would we use such a DSL if the language isn't drastically different?

Because it's easier than writing a database connection from scratch, then having to write raw SQL for every database request, and then properly casting the database data types so that the host language can manipulate the results properly. Remember, a DSL doesn't remove the problem domain; it separates much of the complexity involved with the problem domain.

Let's take a look at a different Internal database DSL by the name of `clojure.java.jdbc`. This DSL is actually used to build the Korma SQL DSL and relies heavily on Java libraries such as `java.sql` and `javax.sql`. We'll briefly compare the two Clojure libraries to better understand how DSLs of the same problem domain can have very different syntax.

Korma uses `clojure.java.jdbc`, but `clojure.java.jdbc` actually requires you to write SQL in certain circumstances. Korma has the ability to run SQL as well, but it tries very hard to create higher-level abstractions in comparison to the `clojure.java.jdbc` library. Let us compare raw SQL to the Korma and `clojure.java.jdbc` libraries, and see what observations we can make about their differences.

Here are the conclusions we come to from the following table:

Raw SQL
<pre>SELECT name FROM customer WHERE id = 10</pre>
Korma Internal DSL
<pre>(select customer (fields :name) (where {:id 10}))</pre>
clojure.java.jdbc
<pre>(sql/with-connection mysql-db-connection (sql/with-query-results rows ["SELECT name FROM customer WHERE id = ?" 10] rows))</pre>

Here are some key observations that you may find interesting:

- The `SELECT` operations of `Clojure.java.jdbc`, act more as a language wrapper than a proper DSL.
- The Solution of `Clojure.java.jdbc` in comparison to Korma's solution, requires you to directly write the language of the problem domain.
- Both the raw SQL and Korma solutions are easier to read in this case because their solutions are smaller than the `clojure.java.jdbc` solution. Keep in mind, though, that the smallest solution doesn't always mean the best or easiest solution to work with. Let's take a better look at `clojure.java.jdbc` by comparing its `INSERT` operations in the following table:

Raw SQL
<pre>INSERT INTO customer (name,age) VALUES ('Hickey',200)</pre>
Korma Internal DSL
<pre>(insert customer (values {:name "Hickey" :age 200}))</pre>

```
clojure.java.jdbc
(sql/with-connection mysql-db-connection
 (sql/insert-record :customer
  {:name "Hickey" :age 200}))
```

We can already see that the `INSERT` solution of `clojure.java.jdbc` feels more like a DSL than its `SELECT` solution. For one, you don't have to write any SQL, which is one of the main reasons why you would use an Internal SQL DSL. Also, notice how, in comparison to Korma's solution, the line count is the same but the character count is much more.

This portion of the book was a just a small glimpse at what these two Clojure libraries can do. I would encourage you, the reader, to explore them a little bit more for a better understanding of their full capabilities. If you wish to know more about the Korma SQL project, you can visit the official website at <http://sqlkorma.com/>. If you wish to know about the `clojure.java.jdbc` project, you can visit the official GitHub repository at <https://github.com/clojure/java.jdbc>.

If you're not interested in using a SQL database with Clojure, you can try any of the following NoSQL Clojure solutions:



- **Redis:** It is available at <https://github.com/ptaoussanis/carmine>
- **Apache CouchDB:** It is available at <https://github.com/clojure-clutch/clutch>
- **Apache Cassandra:** It is available at <https://github.com/pingles/clj-hector>

There are too many NoSQL solutions to list them all, but most Clojure libraries can be found at <http://github.com>.

You can also find a list of different NoSQL database solutions at <http://nosql.findthebest.com>.

The HTML domain

The HTML problem domain has quite a unique collection of Clojure libraries that can make working with HTML feel like a different language altogether. Many Clojure HTML DSLs and libraries often just generate code based on native Clojure data types; in the process, it reminds us all how writing HTML directly can give us a headache. We'll compare some Clojure HTML libraries and see how these tools can help reduce development time.

Formative

The following is a basic HTML form that accepts four inputs. These four inputs are username, e-mail, a checkbox, and a submit button. This form will make a HTTP POST method to the location `/guestbook`. The submit button in this form also displays `Submit Form` instead of the default `Submit` value.

```
<form method="POST" action="/guestbook">
  Username
<br />
<input type="input" name="username" value=""></input>
<br />
  Email
<br />
<input type="email" name="email" value=""></input>
<br />
<input type="checkbox" value="true" name="agree"></input> Agree
<br />
<input type="submit" value="Submit Form" />
</form>
```

As you can see, XML syntax-based languages such as HTML can cause a developer to write a whole lot of brackets for even the simplest of tasks. Also notice how most tags require you to write the name of the tag both at the beginning and at the end of the element definition. Writing HTML without a library or a DSL will often take more time than you might have anticipated because of the tedious nature of writing XML syntax.

The following is the same form written in a Clojure Internal DSL called **Formative**. Formative is a Clojure library that tries to solve the problem of making HTML forms in Clojure. If you wish to know more about Formative, you can view the official GitHub project page at <https://github.com/jkk/formative>.

```
(f/render-form
  {:method "POST"
   :action "/guestbook"
   :fields [{:name :username}
             {:name :email :type :email}
             {:name :agree :type :checkbox}]
   :submit-label "Submit Form"})
```

Formative is the clear winner in comparison to raw HTML. We can see that the three of the four inputs named `username`, `email`, and `checkbox` are defined within the form fields with the form key named `:fields`. Also notice how a submit button doesn't have to be defined in this form. That's because the submit button code is automatically generated with our form when rendered to HTML. You might have also noticed that the form `POST` location and method are set with the form keys `:method` and `:action`.

Hiccup

Formative makes HTML form building somewhat exciting, but let's see how we can develop this same form in another DSL and Clojure library called **Hiccup**. Hiccup is like Formative in the sense that it generates HTML after converting Clojure data, but Hiccup isn't restricted to only forms. Hiccup provides two methods for us to make HTML forms. The following code is the same form as the previous HTML and Formative examples, but this time it's written in the Hiccup DSL:

```
(form-to [:post "/guestbook"]
  (label "username-label" "Username") [:br]
  (text-field "username") [:br]
  (label "email-label" "Email") [:br]
  (email-field "email") [:br]
  (check-box "agree")
  (label "agree-label" "Agree") [:br]
  (submit-button "Submit Form"))
```

This may look a little crowded in comparison to Formative's solution, but this is still clearly better than writing HTML directly. We don't have to write the tag name of elements more than once when defining an element. We also don't have to write anything that feels as if it's outside the Clojure language, so there's almost no learning curve when developing with Hiccup.

Now that we've seen a Hiccup form using Hiccup helper methods, let's take a look at a Hiccup form without any helper methods. The following is the same form as Formative's and the last Hiccup example:

```
[:form {:method "POST"
        :action "/guestbook"}
  [:label {:name "username-label"} "Username"]
  [:br]
  [:input {:type "text"
           :name "username"
           :value ""}]
  [:br]]
```

```
[[:label {:name "email-label"} "Email"]  
[:br]  
[:input {:type "email"  
         :name "email"  
         :value ""}]  
[:br]  
[:input {:type "checkbox"  
         :name "agree"  
         :value "true"}] " Agree"  
[:br]  
[:input {:type "submit"  
         :value "Submit Form"}]]
```

This is obviously the more verbose way to build a HTML form without writing HTML directly. Also, building forms this way is almost as time-consuming as writing HTML directly and somewhat defeats the purpose of using a DSL. If you wish to know more about Hiccup and its capabilities, you can visit the GitHub project homepage at <https://github.com/weavejester/hiccup>.

Mustache

Let's take a look at one more Clojure library and DSL before moving on. The library that we're about to look at is a parser for an External DSL language by the name of **Mustache**. Mustache is an External logicless templating DSL that is mainly used for templating websites, but its uses aren't restricted to strictly webpage templating. If you wish to know more about the Mustache templating language, you can visit the project's GitHub homepage at <http://mustache.github.com/>.

Mustache's templating syntax ignores everything that's not a Mustache tag. The tags are then replaced with data supplied to the parser and then the parser returns the data within the template. What's unique about Mustache's templating language is that it doesn't have common flow control tags such as `else` and `if`.

Clostache

Because Mustache is an External DSL, we'll have to use a Clojure library to parse Mustache templates. There are a few good Clojure libraries that can parse Mustache, but we're only going to focus on one in this example. **Clostache** is a Mustache parsing library that supports multiple versions of Clojure and is compliant with the Mustache specification.

We're going to compare the Hiccup library and the Mustache library Clostache when programmatically generating the following sample HTML code:

```
<div>
  <a href="#"> Link Red </a>
  <br/>
  <a href="#"> Link Black </a>
  <br/>
  <a href="#"> Link Yellow </a>
  <br/>
  <a href="#"> Link White </a>
  <br/>
</div>
```

First, let's look at the Hiccup solution, and then compare it to the Mustache solution. The following is one way we could generate the sample HTML with the Hiccup library:

```
[:div
 (interleave
  (for [c ['Red 'Black 'Yellow 'White]
        :let [link [:a {:href "#"} (str "Link " c)]]]
    link)
  (repeat 4 [:br]))]
```

This example of the preceding code probably looks insane to you if you're a Clojure beginner, but this code will produce the same code as the previous HTML example. This isn't the only way to do it, but it's one of the nicer ways of generating the sample HTML. Let's try this again with Mustache and the Clostache Clojure library, as follows:

```
(def template "
<div>
  {{#colors}}
  <a href=\"#\">>Link {{color}}</a>
  <br />
  {{/colors}}
</div>")
(clostache.parser/render
 template
 {:colors
  [[:color 'Red]
   [:color 'Black]
   [:color 'Yellow]
   [:color 'White]]})
```

Looking at this example may make more sense than the previous Hiccup solution that we just looked at, but there are some things that we need to understand about Mustache to fully understand this solution. The `{{#colors}}` tag tells Mustache to loop through our Clojure collection of colors. Inside this loop, there's the `{{color}}` tag. The `{{color}}` tag belongs to the `{{#colors}}` collection and Mustache automatically assumes that this is the color inside the `{{#colors}}` collection loop.

As we can see, there are many ways to use different libraries to achieve the same result in this problem domain. Some methods and libraries are obviously better in their own respects, but choosing a library and an Internal DSL is a combination of preference and project requirements. Clojure's ability to use either Java or Clojure libraries will give any developer plenty of solutions to choose from.

The ECMA/JavaScript domain

There are many DSLs that ultimately compile into the JavaScript language. There's CoffeeScript for Node.js, Script# for C#.Net, and Haxe that has native compile to JS features. Clojure, on the other hand, has an alternative compiler just to handle JavaScript.

ClojureScript

The alternative compiler doesn't come with the **Java Virtual Machine (JVM)** implementation of the Clojure language, so you'll have to download the compiler from a GitHub project called **ClojureScript**. The project page can be found at <https://github.com/clojure/clojurescript>. Because it might be a while before you want to go through all the trouble to set up the compiler, you can run ClojureScript directly in your web browser using the ClojureScript website <https://himera.herokuapp.com/index.html>.

ClojureScript's target language is JavaScript, so using any Java interoperation functions on objects will no longer work. We'll have to use JavaScript interoperation instead, so the knowledge of JavaScript language is required to some extent to get the best out of ClojureScript's capabilities. We'll cover some of the differences between ClojureScript and the JavaScript language, but you can view a full comparison chart at <https://himera.herokuapp.com/synonym.html>.

Comparing ClojureScript and JavaScript

The JavaScript variable definitions are similar to the C language's style of defining variables. The following is a JavaScript variable named `car` that holds the value `Red`:

```
var car = 'Red';
```

The following is the same JavaScript variable written in ClojureScript. Notice that this syntax is also how you would define an object named `car` in the Clojure JVM implementation:

```
(def car "Red")
```

Let's look at a more interesting example where the JavaScript actually does something, and let's see what the same code would look like in ClojureScript. The following is a JavaScript example of destructuring parts of an object. The parts of the object are then placed within a string and alerted in a browser window as follows:

```
var example = {type: "car", color: "red"};
var vehicle = example.type,
    color = example.color;

alert('The '+vehicle+' is '+color);
```

Here's how we would produce the same results with ClojureScript. The following example displays local variables being assigned values using `let`. The first local variable `example` is assigned to what looks like a JSON JavaScript object, but what is actually a key value set. The next two local variables are `type` and `color`, and are assigned by matching the key values in the local variable `example`. What you'll see after the local variable definitions is JavaScript interoperability with the call to `(js/alert)`. This will make an alert message window within the web browser and display the local variables within a string, as follows:

```
(let [example {:type "car" :color "red"}
      {:keys [type color]} example]
  (js/alert
   (str "The " type " is " color)))
```

Let us take a look at using dynamic bindings in JavaScript and then rewrite the same code in ClojureScript:

```
var x = 1, y = 1, doubleIt = { x: x*2, y: y*2};

with(doubleIt) {
  alert(x*y);
}
```

The preceding JavaScript example will define two variables with the same value of 1. Those values are `x` and `y`, and they will have their values changed when `with` uses the `doubleIt` object to locally bind new values to `x` and `y` within the scope of the `with` curly braces. If you were to run this JavaScript in your browser, you would get an alert message window stating that `x` multiplied by `y` is 4.

Now let's see how the same thing can be done in ClojureScript, as follows:

```
(def ^:dynamic x 1)
(def ^:dynamic y 1)

(binding [x 2 y 2]
  (js/alert
    (* x y)))
```

The preceding example is the same as the JavaScript concept. Both variables have a local value of 2 within the braces of `binding`. Then the multiples of both values are alerted in a browser message window. One thing that's very different in this example as opposed to the JavaScript example is that the variables have metadata tags in front of the variable names. This metadata setting is `^:dynamic` and is needed to rebind the values, because by default, Clojure doesn't allow the rebinding of defined objects. The metadata allows Clojure to know that this object should be able to change and allows us to bind other values to the object definition. Unlike JavaScript, the mutation of a Clojure object can't be done unless the object type is explicitly mutable and mutated explicitly (think of the `swap!` function). Keep this in mind when working with Java objects because you might treat it like a Clojure object and be very surprised when you call a function that changes the value of your object.

For loops in JavaScript remind us of our days programming C, but Clojure and ClojureScript take a different route and we're about to see just how. The following example is a JavaScript `for` loop that counts down from 3 and does nothing when reaching 0. Before the loop reaches zero, the web browser will alert three messages that display the current number that the countdown is currently at.

```
for(i = 3; i >= 1; i--) { alert(i); }
```

The preceding example was a simple one liner, but let's take a look at the ClojureScript way of doing it:

```
(loop [i 3]
  (if-not (zero? i)
    (do (js/alert i)
        (recur (dec i)))))
```

Loop defines a local variable `i` with the value of 3. Within the loop body, you'll see that action is taken if the variable `i` isn't equal to zero. If the variable isn't equal to zero, the web browser will send an alert message displaying the current number of countdown. After the message is displayed, `recur` changes the local variable bindings by decrementing `i` with the `dec` function before the loop restarts.

Let's look at one last ClojureScript example before moving on to other domains. The following example is defining a JavaScript object type that has a function named `fullTitle`. When this function is called, the web browser makes an alert message window displaying a formatted version of the author and title of the book object, as follows:

```
function Book(author,title) {
  this.author = author;
  this.title = title;
}

Book.prototype.fullTitle = function() {
  msg = "Author: "+this.author+"\r\nTitle: "+this.title;
  alert(msg);
}

var book = new Book("B. Writer","Clojure All Day");
book.fullTitle();
```

The following is the ClojureScript solution for the same result. We define the type, `Book`, as a child to the JavaScript object, `Object`, and implement a function named `full-title`. One difference you might have already noticed is that the function name is in front of the defined variable and is prefixed with a decimal. This is because we're calling a method function that belongs from within the defined JavaScript object as opposed to calling an externally defined function.

```
(deftype Book [author title]
  Object
  (full-title [_]
    (js/alert
      (str "Author: " author "\r\nTitle: " title))))

(def book (Book. "B. Writer" "Clojure All Day"))

(.full-title book)
```


The Audio domain

Although you can use Java audio libraries and DSLs within Clojure, there are only about two really solid DSLs for synthetic audio manipulation from within the Clojure language. This is not only because Clojure is a very young language, but also because these libraries are very good at what they do. Because some of the many DSLs in this section can get very complex, we'll look at only some of the more basic uses of them.

Music-as-data

The first Clojure library and Internal DSL we'll look at is named **Music-as-data**. This is a project aimed at easy note, pattern, and sample playback. The notes are synthetics and the samples are bound to the WAV files on the project's classpath.

Although the documentation seems to have fallen behind on the project, there are some examples and snippets that still work and are worth covering. If you wish to know more about the project, you can view the project's GitHub homepage at <https://github.com/jonromero/music-as-data>.

You might wonder why we would use examples from a project with out-of-date documentation? The documentation might not be up-to-date, but the principles of a DSL are very strong in this particular library. For example, you can create instruments, samples, and chords without any knowledge outside the Clojure language with the exception of knowing the names of the musical notes you wish to play.

It's okay if you're not a music theory major. No knowledge of music theory is required to exploit this library to the fullest extent. Luckily for us, there's a function that helps us play musical notes. We can use the `(p)` function to playback a sequence of musical notes. We can also use `(pattern)` and `(chord)` to modify how the note sequences are played.

If we wish to play each note at the same speed, we will simply use the `(p)` function and a sequence of musical note symbols. If we want the speed of the playback to progress for each musical note, we will wrap the sequence of notes in the `(pattern)` function. The `(pattern)` function not only accepts a sequence of musical notes, but also accepts a duration integer, which will play each note for the stated duration. The following are two examples of playing the musical pattern of the notes A4 and B4. The second of the two examples has a duration of 5 seconds of play for each note within the musical sequence, as follows:

```
;; Example 1
(p (pattern [A4 B4]))
;; Example 2
(p (pattern [A4 B4] 5))
```

Let's say we want to make a little melody but we don't really know anything about audio programming. First, we should define the initial part of our melody and call it `pattern-1`. The following is an example of the `pattern-1` definition to start our series of musical notes:

```
(def pattern-1 (pattern [A4 B4 C5 D5] 3))
```

Let's also define the second pattern of our melody, and since we know nothing about music theory, let's reverse `pattern-1` in a definition of `pattern-2` and call it a day:

```
(def pattern-2 (pattern [D5 C5 B4 A4] 3))
```

Now that we've defined two parts of our melody, let's combine the patterns and play the melody to hear what we've created. Let's not forget that the `(p)` function allows us to play a collection of musical notes and patterns, as follows:

```
(def melody (flatten (merge pattern-1 pattern-2)))
```

```
(p melody)
```

If you don't already know, we have to merge the two pattern sequences with the `(merge)` function, and then flatten the collection of note patterns into a single collection of note patterns with the `(flatten)` function.

Overtone

If you know the mathematics of music theory, the Music-as-data library and Internal DSL probably don't meet your musical needs. Thankfully, there's a library that allows a musical theorist to express even more complex sounds; this library is known as Overtone.

Overtone is by far the most complete and well-documented Clojure sound library and the project's GitHub homepage can be located at <https://github.com/overtone/overtone>. Overtone is the obvious leader in Clojure synthetic sound-generation and sound-pattern composition. Unlike the Music-as-data library, the understanding of music at its core is needed to take full advantage of this library and Internal DSL.

We'll try to play the same melody as we did in the Music-as-data example, but we'll use the most basic features of the Overtone library to jazz it up a bit. Any serious musically-inclined software developer should study the Overtone documentation to better understand how to exploit this DSL. Before we start working on our melody, we'll have to define some playback helpers and which instruments they'll use.

Overtone has a collection of instruments already built-in, so we can get started right away as we did with the Music-as-data library. In the following example code, we're telling Overtone to use the built-in piano:

```
(ns test.core
  (:use [overtone.live]
        [overtone.inst.piano]))
```

Next, we'll need to define a function to play our melodies. In the following example code, we're going to specify a definition named `p`. It will accept a collection of musical notes named `chord` and a real number (not negative) named `delay`. `doseq` will then take each note out of the chord sequence and play the piano note before forcing the program's current thread to wait for the amount of delay time specified multiplied by 100 milliseconds, as follows:

```
(defn p
  [chord delay]
  (doseq [f chord]
    (piano f)
    (Thread/sleep
     (* delay 100)))))
```

Now that we have a function that can play a musical sequence with a piano, we'll have to define a pattern that will be the foundation for the rest of the patterns. We'll also need a function that can translate our pattern into piano-friendly notes. We can call this function `builder`. The `builder` function takes a note sequence named `chord-pattern` and a setting of either major or minor with the required variable named `m-or-m`. This function will take every note out of our chord pattern and apply the `chord` function to each note. After each note has been turned into another pattern of notes, the `builder` function uses the `(flatten)` function on the newly-generated patterns to make sure all of the note sequences become one sequence for our instrument to play. Once all of this is done, the `builder` function will return an instrument-playable format. The following is an example of this function:

```
(def pattern
  [:A4 :B4 :C5 :D5])
(defn builder
  [chord-pattern m-or-m]
  (->> chord-pattern
    (map
     #(chord % m-or-m))
    flatten))
```

Since we have our components for making a basic melody, let's define our first two patterns. The first pattern is pretty simple and uses the `builder` function on our already defined pattern named `pattern`. This pattern is called `pattern-1` and it will play our pattern in minor chords. Our second pattern named `pattern-2` is simply reversing the first pattern we made. The following code is an example of this:

```
(def pattern-1
  (builder pattern :minor))

(def pattern-2
  (reverse pattern-1))
```

Now that we have two musical patterns, we can define a melody named `melody`. This definition will repeat our patterns twice with the `repeat` function and then flatten the musical sequences into one musical sequence. The following is an example of this very definition:

```
(def melody
  (flatten
    (repeat 2 [pattern-1
               pattern-2])))
```

Now that we have a melody and a function that can play the piano for us, let's play each note in the melody with the `play` delay of 3. The following code is an example of this:

```
(p melody 3)
```

Image domains

Because Clojure is such a young language, only a few image libraries and DSLs are written in Clojure. **Quil** is a library and DSL that can handle both the creation and animation of images within the Clojure language. Quil is based on a programming project that specializes in image and animation creation; this project is called **Processing**. Because image generation is as difficult as, or even more difficult than, audio generation, we'll only cover some basic image generation examples. You can find out more information on both these projects by visiting the documentation section of Quil's GitHub project homepage at <https://github.com/quil/quil>.

Quil is very easy to use but a knowledge of graphing coordinates and geometry is required to exploit this library to the fullest extent. Don't feel afraid if you're not a mathematician. This section of the chapter only covers basic line drawing and animations.

Quil provides a Clojure macro named `defsketch` and allows us to send our drawing information to the main application. This macro accepts an application name and a few named options for rendering the final results. We'll define a still-frame image within `defsketch` by using what's called an anonymous function (a function without a name).

This is an explanation of the following example of how to draw vertical lines with Quil. The anonymous function for the `setup` option declares that the image should use an anti-aliasing filter with the function named `smooth`. The anonymous function for the `draw` option declares the width of the line with a function named `stroke-weight` and then states that a line should be drawn. The `line` function accepts four arguments in the form of `x1`, `y1`, `x2`, and `y2`. The starting point of the line is obviously `x1` and `y1` with the ending point of the line being obviously `x2` and `y2`.

```
(defsketch drawing
  :title "Book Example"
  :setup (fn [] (smooth))
  :draw (fn []
          (stroke-weight 9)
          (line 100 50 100 100))
  :size [200 200])
```

The difference in the `y` axis coordinates is what draws the line this way, if you didn't know. To make this line horizontal and grow longer for every passing second, we will have to make a few interesting changes to our anonymous function for the `setup` and `draw` options. First the `frame-rate` function will have to be called for the `setup` option. This allows Quil to know that this image is indeed an animation. Secondly, we will define an atom named `right` to hold the `x2` value for our function as the line animates closer to the right of the window. Last but not least, the anonymous function for `draw` will increment and return the value of the `right` atom when the function is called. The following code is an example of these changes:

```
(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])
```

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.

Summary

A DSL is a language dedicated to solving only one type of problem. The language can't make the problem domain less complex, but it can separate the complexities of the problem for the person using the language. Generally speaking, a DSL only simplifies the nomenclature of the problem domain but never completely removes it. The same terms used to solve the problem need to persist within the language, so that the expressions can be written and understood by those familiar with that particular domain.

An Internal DSL is built on top of or within another language and uses that language as its core foundation. Clojure's power and flexibility allows for highly expressive Internal DSLs to be built quickly and cleanly. Although this book doesn't cover External DSLs, Clojure has many External DSL-friendly features such as code generation, the ability to have nonalphabetical definitions, and metadata, to name a few.

As we can see, there are usually competing languages in each problem domain. Some of these languages can actually increase the complexity or the time needed to solve our problems if certain aspects of the languages we use aren't taken into consideration. We should also ask ourselves if we even need a DSL when trying to solve our problems, because sometimes they add more layers of complexity than the original problem.

In the next chapter, you'll learn about editing Clojure programs with the Emacs and Leiningen applications.

For additional information and documentation, please refer to the chapter information sources in the next section.

2

Design Concepts with Clojure

This chapter 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 whichever works for you.

This chapter will cover:

- Pure functions
- General programming concepts
- Clojure writing styles

By the end of this chapter, you should be able to articulate the concepts of DRY, KISS, YAGNI, and bottom-up development. In addition to this, you should have a better understanding of how to write nicer Clojure as the anti-patterns section displays many examples of both what to do, and what not to do.

Every function is a little program

When I first started getting deep into Clojure development, my friend Tom Marble taught me a very good lesson with a single sentence. I'm not sure if he's the originator of this idea, but he told me to think of writing functions as though "every function is a small program". I'm not really sure what I thought about functions before I heard this, but it all made sense the very moment he told me this.

Why write a function as if it were its own program? Because both a function and a program are created to handle a specific set of problems, and this method of thinking allows us to break down our problems into a simpler group of problems. Each set of problems might only need a very limited collection of functions to solve them, so to make a function that fits only a single problem isn't really any different from writing a small program to get the very same result. Some might even call this the Unix philosophy, in the sense that you're trying to build small, extendable, simple, and modular code.

A pure function

What are the benefits of a program-like function? There are many benefits to this approach of development, but the two clear advantages are that the debugging process can be simplified with the decoupling of task, and this approach can make our code more modular. This approach also allows us to better build **pure functions**. A pure function isn't dependent on any variable outside the function. Anything other than the arguments passed to the function can't be realized by a pure function. Because our program will cause side effects as a result of execution, not all of our functions can be truly pure. This doesn't mean we should forget about trying to develop program-like functions. Our code inherently becomes more modular because pure functions can survive on their own. This is key when needing to build flexible, extendable, and reusable code components.

Floor-to-roof development

It is also known as bottom-up development and is the concept of building basic low-level pieces of a program and then combining them to build the whole program. This approach leads to more reusable code that can be more easily tested because each part of the program acts as an individual building block and doesn't require a large portion of the program to be completed to run a test.

Each function only does one thing

When a function is written to perform a specific task, that function shouldn't do anything unrelated to the original problem it's needed to solve. For example, if you were to write a function named `parse-xml`, the function should be able to act as a program that can only parse XML data. If the example function does anything else other than parse lines of XML input, it is probably badly designed and will cause confusion when trying to debug errors in our programs. This practice will help us keep our functions to a more reasonable size and can also help simplify the debugging process.

Patterns for success

If you're reading this book, you're probably already familiar with some of the programming principles that'll be covered in this chapter. Don't feel discouraged if you don't and please review these principles if you do. This section will cover three important programming principles that can be used together to write better code. These three principles are **DRY**, **KISS**, and **YAGNI**. As you'll see, each principle has an inherently minimalistic nature to its core concepts and works nicely when applied with others.

DRY

Don't repeat yourself: The concept of DRY is fairly simple. This programming concept has become one of the most fundamental concepts in modern programming. The idea is to produce cleaner code by writing abstractions that eliminate repetition.

Reasonably so; duplication causes unnecessary bloating and requires modifications to be applied in multiple locations for the same change. Because duplicate code is scattered across multiple locations, when a change is made in one location and not in others, bugs are often the consequence of not using the DRY principle. The bigger picture of the DRY principle isn't necessarily to eliminate all forms of duplication, but to eliminate multiple ways of expressing the same thing. You have to ask yourself, what's easier to modify and maintain? One representation of one thing or three completely separate representations of the same thing? Using the DRY principle helps to avoid building and maintaining multiple representations, and ultimately leads to easier maintenance and cleaner code.

KISS

Keep it simple, stupid: This is the principle of breaking things down into subtasks and keeping your solutions as clear and as simple as possible. Even though the complexity of software is increased when adding functionality, this doesn't mean that the method of implementation can't be simple. When using the KISS principle, you should try to solve the problem before you begin writing your solution. This will help you break up the implementation of your solution into smaller, simpler steps.

YAGNI

You aren't going to need it: This principle works very well with the KISS principle because you try to keep things minimal and simple by only adding functionality when needed. This helps to prevent unnecessary bloat and allows to focus more on the functionality that's needed immediately.

Writing Clojure

Since Clojure is a young language, the common anti-patterns of Clojure development are still evolving. If you're coming from another Lisp-based language, then you can probably transplant many of the best practices when developing with the Clojure language. It's very important, for those new to Lisp-based languages and Clojure in general, to review the evolving anti-patterns to write more maintainable and better code. With that said, style guides aren't law and should be broken when a more pragmatic approach can be taken.

This section will go over the proper way of writing Clojure code and then display the improper way of writing the same code. Sometimes showing what not to do is better than showing what to do, so this section tries to show both, with the hope that we can better understand the reasoning behind the method. Again, style guides aren't law, but they encourage best practices for real-world Clojure development.

Spacing and alignment

As with most programming languages, spacing and alignment are important in writing maintainable Clojure. Unlike Python, whitespace characters are ignored in most circumstances. This section will specifically look at how beautiful Clojure can be, and how ugly someone can make it.

When indenting in Clojure, you want to use two spaces rather than four spaces or tab characters, as shown in the following code snippet:

```
;; Yes
(defn example
  [is-true? is-false?]
  (when (true? is-true?)
    (when (false? is-false?)
      (println "two space indent"))))

;; No
(defn example
  [is-true? is-false?]
  (when (true? is-true?)
    (when (false? is-false?)
      (println "two space indent"))))
```

The exception to this rule applies when you're passing arguments to a function. There's no limit as to how many spaces can be used because the goal is to vertically align the arguments of the function call, as shown in the following code snippet:

```
;; Yes
(example ["M.Fogus"
         "R.Hickey"
         "C.Emerick"]
         ["J.Rogan"
         "D.Stanhope"
         "B.Burr"])

;; No
(example ["M.Fogus"
         "R.Hickey"
         "C.Emerick"]
         ["J.Rogan"
         "D.Stanhope"
         "B.Burr"])
```

This exception also applies to the `let` special form and Clojure collection types, as shown in the following code snippet:

```
;; Yes
(let [x 1
      y 2
      z (+ x y)]
  (apply + [x y z]))
```

```
;; No
(let [x 1
      y 2
      z (+ x y)]
  (apply + [x y z]))
```

```
;; Yes
{:first "Brian"
 :middle ""
 :last "Redban"}
```

```
;; No
{:first "Brian"
 :middle ""
 :last "Redban"}
```

```
;; Yes
["Apples"
 "Oranges"
 "Grapes"]
```

```
;; No
["Apples"
 "Oranges"
 "Grapes"]
```

```
;; Yes
{"key" "value"
 "map" "literal"}
```

```
;; No
{"key" "value"
 "map" "literal"}
```

```
;; Yes
```

```
#{"hash-set"
  "tes-hsah"
  "odd count"}
```

```
;; No
#{"hash-set"
  "tes-hsah"
  "odd count"}
```

Defining a function in Clojure is pretty clear, but there is a wrong way to do so, as shown in the following code snippet:

```
I

;; Yes
(defn x [y] y)

;; Also okay
(defn a
  [b]
  b)

;; No
(defn l
  [m] m)

;; No
(defn
  o [p] p)

;; Yes
(defn x
  ([] (x 1))
  ([y] (+ 1 y)))

;; No
(defn y ([] (y 1)) ([z] (+ 1 z)))
```

Not enough whitespace leads to clutter and non-clarifying whitespace leads to longer lines of code, as shown in the following code snippet:

```
;; Yes
(conj [1 2] 3)
;; No
(conj [1,2] 3)

;; Yes
```

```
(do (x y))
;; No
(do ( x y ) )
```

As commas aren't required, don't use commas when separating elements of Clojure's collection types, as shown in the following code snippet:

```
;; Yes
[\a \b \c \d \e]

;; No
[\a, \b, \c, \d, \e]

;; Yes
{:os "archlinux"
 :machine "acer"}

;; Yes
{:os "archlinux" :machine "acer"}

;; No
{:os "archlinux",
 :machine "acer"}
```

If you're coming from a language such as ECMAScript/JavaScript, you might be tempted to close nested braces in the same way that you would do in JavaScript. You'll want to avoid this practice in Clojure because it can potentially cause confusion and your code will take more page room than necessary, as shown in the following code snippet:

```
;; Yes
(defn xyz
  [i]
  (let [example? true]
    (fn [x]
      (if example?
        (* x i)
        (* 2 x))))))

;; No
(defn xyz
  [i]
  (let [example? true]
    (fn [x]
      (if example?
```

```
        (* x i)
        (* 2 x)
      )
    )
  )
)
```

Organizing and grouping top-level Clojure forms make your code easier to read and manage, as shown in the following code snippet:

```
;; Yes
(def _name "Ryan")

(defn user [user-name]
  (str "User " user-name))

(def tool :can-opener)
(def can {:can-opener "beans"})

(defn open-can [can]
  (tool can))

;; No
(def _name "Ryan")
(defn user [user-name]
  (str "User " user-name))
(def tool :can-opener)
(defn open-can [can]
  (tool can))
(def can {:can-opener "beans"})
```

Syntax

Clojure's `:pre` and `:post` assertion test features are a great way for testing that the input and output of a function are indeed valid. Because of this wonderful feature, using `:pre` and `:post` should be preferred over performing checks that throw errors, as shown in the following code snippet:

```
;; Yes
(defn example [x y]
  {:pre [(pos? x)
         (neg? y)]
    :post [(< 5 %)]}
  (+ x y))

;; No
```

```
(defn example [x y]
  (let [value (+ x y)]
    (when (and (pos? x)
               (neg? y)
               (< 5 value))
      value)))
```

Defining variables within a function can be a big mistake. You should write a macro to create a custom definition form, or use the `let` form to define local variables. In Clojure, all variables defined outside the `let`, `if-let`, and `when-let` forms are global in scope, as shown in the following code snippet:

```
;; Yes
(defn example [& x]
  (let [str-version (apply str x)
        _ "Another local variable"]
    (-> str-version
        (clojure.string/reverse)
        first)))

;; No
(defn example [& x]
  (def str-version (apply str x))
  (-> str-version
      (clojure.string/reverse)
      first))
```

The names of function arguments need to be chosen wisely. If you're not careful, you'll clobber a globally-accessible function or variable. This may lead to unexpected results and will eventually lead to having to use the fully-qualified name of the variable or function you wish to use within your function, as shown in the following code snippet:

```
;; Yes
(defn example [-map]
  (map
   #(let [[k v] %]
       (str "Key: " k " Value: " v))
   -map))

;; No
(defn example [map]
  (map
   #(let [[k v] %]
       (str "Key: " k " Value: " v))
   map))
```


When testing a Boolean value, if the test is true, and there's nothing to be done when false, prefer `when` over the combination of `if` and `do`, as shown in the following code snippet:

```
;; Yes
(when working?
  (a b c)
  (x y)
  (z))

;; No
(if working?
  (do (a b c)
      (x y)
      (z)))
```

When needing to define and test a local variable, prefer the use of the `if-let` form over the combined use of `let` and `if`, as shown in the following code snippet:

```
;; Yes
(if-let [apple (:apple fruits)]
  (println apple))

;; No
(let [apple (:apple fruits)]
  (if apple
    (println apple)))
```

This should also be the case when needing to use the `when` form, as shown in the following code snippet:

```
;; Yes
(when-let [apple (:apple fruits)]
  (println apple))

;; No
(let [apple (:apple fruits)]
  (when apple
    (println apple)))
```

The use of `if-not` should be preferred over the combined use of `if` and `not`, as shown in the following code snippet:

```
;; Yes
(if-not (xyz)
  (println "false"))

;; No
(if (not
    (xyz))
  (println "false"))
```

Prefer the use of the `when-not` form instead of the combined use of the `if`, `not`, and `do` forms, as shown in the following code snippet:

```
;; Yes
(when-not xyz
  (println false))

;; No
(if (not xyz)
  (do
    (println false)))
```

Using `when-not` should also be preferred over the combined use of the `when` and `not` forms, as shown in the following code snippet:

```
;; Yes
(when-not xyz
  (println false))

;; No
(when (not xyz)
  (println false))
```

Prefer the use of the `not=` form over the combined use of the `not` and `=` forms, as shown in the following code snippet:

```
;; Yes
(true? (not= a b))

;; No
(true?
  (not
    (= a b)))
```

The use of anonymous functions to make a single function call shouldn't be wrapped when calling forms such as `flatten` and `map`, as shown in the following code snippet:

```
(def nums [1 2 3])
;; Yes
(map inc nums)
;; No
(map #(inc %) nums)
```

The use of `complement` should be preferred over the use of anonymous functions when needing the inverse of a function that returns a Boolean value, as shown in the following code snippet:

```
;; Yes
(filter
  (complement nil?)
  [true false nil 123 nil])

;; No
(filter
  #(not (nil? %))
  [true false nil 123 nil])
```

To increase readability and prevent deep nesting, prefer the use of `comp` over the use of anonymous functions that wrap multiple function calls, as shown in the following code snippet:

```
;; Yes
((comp clojure.string/reverse
      str)
 123456)

;; No
(#(clojure.string/reverse
  (str %))
 123456)
```

As an alternative, use the `thread last` and `thread first` macros to increase readability and to prevent deep nesting of function parentheses, as shown in the following code snippet:

```
;; Yes
(->> [1 2 3 4 5]
  (map str)
  first
  example)

;; No
(example
  (first
    (map str [1 2 3 4 5])))

;; Yes
(-> [1 2 3 4 5]
  reverse)
```

```
((partial map str))
first)

;; No
(first
 (map str
  (reverse [1 2 3 4 5])))
```

When calling multiple methods using Java interoperation, prefer single or double decimal interop notation over the use of thread-first macros, as shown in the following code snippet:

```
;; Okay
(-> (java.util.UUID/randomUUID) .version)

;; Better
(.version (java.util.UUID/randomUUID))

;; Best
(.. java.util.UUID randomUUID version)
```

When testing a variable or an expression, prefer the `case` form over both the `condp` and `cond` forms as shown in the following code snippet:

```
;; Okay
(cond
  (= x 5) :is-y
  (= x 4) :is-z
  :else :no-match)

;; Better
(condp = x
  5 :is-y
  4 :is-z
  :no-match)

;; Best
(case x
  5 :is-y
  4 :is-z
  :no-match)
```

Name conventions

Unlike many of the popular languages in use today, the use of underscores and camel case for either functions or variables names is frowned upon in Clojure, as shown in the following code snippet:

```
;; Yes
(def clojure-global-variable 100)
;; No
(def clojureGlobalVariable 100)
;; No
(def clojure_global_variable 100)

;; Yes
(defn clojure-global-fn [x] x)
;; No
(defn clojureGlobalFn [x] x)
;; No
(defn clojure_global_fn [x] x)
```

When writing a function that returns a Boolean value, add a question mark at the end of the function names to indicate that it returns either true or false, as shown in the following code snippet:

```
;; Yes
(defn red? [x]
  (... (str x)
    toLowerCase
    (contains "red")))
;; No
(defn is-red [x]
  (... (str x)
    toLowerCase
    (contains "red")))
;; No
(defn red-p [x]
  (... (str x)
    toLowerCase
    (contains "red")))
```

Dynamic definitions must attach the `^:dynamic` metadata before the definition symbol, and the symbol must contain **stars** around the symbol to indicate that it is indeed dynamic and rebindable, as shown in the following code snippet:

```
;; Yes
(def ^:dynamic *abc* 0)
;; No
(def ^:dynamic abc 0)
```

When handling values that you aren't going to employ, use the underscore character for their assignment, as shown in the following code snippet:

```
;; Yes
(def example
  ["red" "blue" "green"])

(defn use-example
  [_ blue green]
  (println blue green))

;; No
(def example
  ["red" "blue" "green"])

(defn use-example
  [[red blue green]]
  (println blue green))
```

Collection types

When defining a key/value collection type, prefer the use of the Clojure keyword data type as a key over the use of string keys, as shown in the following code snippet:

```
;; Yes
(hash-map :abc 123)
(zipmap [:abc :def]
        [123 456])
{:abc 123}

;; No
(hash-map "abc" 123)
(zipmap ["abc" "def"]
        [123 456])
{"abc" 123}
```

When getting values from a key/value collection type, use the keyword data type to get the value from the collection, as shown in the following code snippet:

```
(def comedian
  {:name :Doug-Stanhope
   :like :Bill-Burr})

;; Yes
(:name comedian)

;; Yes
```

```
(get comedian :name :not-found)
;; No
(comedian :name)
;; No
(get comedian :name)
;; No
(if-not (:name comedian)
  :not-found
  (:name comedian))
```

Summary

Every function is a little program. When writing a function, you should apply the Unix philosophy to create a more simple, flexible, and extensible function. The Unix philosophy, in terms of programming, is to build small, extendable, simple, and modular code. This will help simplify debugging and help prevent unwanted side effects that can be commonly attributed to a function written to perform more than one task.

Building an application from the bottom-up requires a strong set of low-level functions to better support the abstraction process when building higher-level functions. This approach encourages writing more modular and reusable code because each function acts as an individual building block. These building block functions will more likely be used in another project with little or no modifications. Limiting a function's capabilities to a single task helps prevent unwanted side effects and leads to creating better building blocks.

In the next chapter, you'll learn about editing Clojure files and project management. If you're already comfortable with your current set of tools, you can skip the next chapter.

3

Clojure Editing and Project Creation

This chapter will help get you 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 very basics to get you started.

Emacs is actually more than a text editor and this chapter will guide you through the installation and setup of the Emacs editor. Once Emacs is running and configured for Clojure development, you'll be exposed to working with a Clojure REPL session from within the Emacs editor. Emacs might take a few days to get used to if you're not already familiar with the software, but it's important to get comfortable using Emacs because it's currently the best Clojure IDE available.

Lein is a project management utility that makes creating and managing Clojure-based projects simpler. This chapter will explain as to why you should use Lein when starting a project and then guide you through the most basic project operations. By the end of this chapter you should feel confident in using Lein to create and manage your Clojure projects.

The origin of Emacs and its usage

Emacs was originally developed in the 1970s, and leverages the power of keyboard shortcuts. Although Emacs currently has a graphical user interface, keyboard manipulation is still the fastest and best way to edit within the Emacs editor. If you're a Vim user or coming from an editor that was originally designed for graphical systems, Emacs may seem overwhelming at first. Even though Vim and GUI-based editors have shortcuts, Emacs can only be used effectively by learning the shortcuts of the most common editing operations.

Emacs is a text editor that can be programmatically extended with a language called Emacs Lisp. Emacs Lisp is interpreted by the Emacs editor to make custom hot-keys, run applications from within the text-editor, and much more. Using the Emacs package manager, you can download and install extensions to add new features, such as syntax-highlighting support for your favorite language, and non-text editing-related features, such as e-mail and web browsing support from within the editor window. Emacs has many directory structure- and managerial-related extensions as well. For example, if you like to use Git for your projects, you could install one of the many Git repository managers for executing Git-related operations from within the Emacs editor.

Emacs has varying levels of support and advanced features depending on what language you're working with. Each Emacs buffer window's editing behavior is dictated by what's called a major mode. If you write in Ruby, you'll want to install and invoke the ruby mode. If you write in CoffeeScript, you'll want to install and invoke the coffee mode. Most feature-complete major modes handle syntax-highlighting, proper indention levels, and file-type detection; however, because major modes are developed by different individuals and communities, each major mode differs in both quality and features.

Emacs is a great tool and editor for many popular languages, but Lisp-based languages have some of the best support by far. There are many extensions that help Lisp-based languages with parenthesis and bracket alignment, coloring, and match highlighting. Although Emacs isn't technically an IDE, it acts as one for many popular languages, such as Common Lisp, Emacs Lisp, and Clojure. Unlike other language IDEs, Emacs can be controlled entirely by the keyboard, which allows for increased productivity because you don't have to move a hand to execute an action as you would with a mouse in most other editors. There's also a small window called the mini buffer and it allows for commands and operations that don't have keyboard shortcuts to be executed. Because you don't need a mouse to control the editor, you can also run Emacs in a command-line interface – if you prefer.

Installing and setting up Emacs²⁴

If you're an Arch Linux user, you can install Emacs with the following command-line command:

```
$ sudo pacman -Sy emacs
```

Users of Fedora and Fedora-based distributions can install Emacs with the following command-line command:

```
$ sudo yum install emacs
```

Ubuntu and other Debian-based distributions can install Emacs with the following command-line command:

```
$ sudo apt-get install emacs24
```

If you're using Mac OS X, you can install Emacs with the universal binary from <http://emacsformacosx.com>. If you wish to use MacPorts (<https://www.macports.org/>), you can use the following command-line command:

```
$ sudo port install emacs
```

Windows users can install Emacs with the Chocolatey Package Manager (<http://chocolatey.org/>) and the following command-line command:

```
C:\> cinst Emacs
```

If for whatever reason you've installed Emacs 23 instead of 24, be sure to install the `package.el` extension before continuing this chapter. You can get the `package.el` extension at <https://github.com/technomancy/package.el>. Further instructions on how to manually install extensions can be found at http://ergoemacs.org/emacs/emacs_installing_packages.html.

Setting up Emacs

Now that Emacs is installed, start Emacs with the command-line command `Emacs` and make sure that the editor starts properly. If your operating system starts a graphical version of Emacs and you wish to run a console version, you can run the command-line command `emacs -nw`. The operator `-nw` argument tells Emacs no window system.

Because you won't learn how to use Emacs until the next section, you'll have to edit the Emacs configuration files with your favorite text editor. If you're a user of Windows XP and above, you can find the Emacs configuration file and global configuration directory at `C:\Documents and Settings\Application Data\<username>\` or `c:\Users\<username>\AppData\Roaming\`. For all other operating systems, you can find your Emacs configuration file and global configuration directory at `~/.emacs` and `~/.emacs.d`.

After you have located the proper Emacs configuration directory for your operating system, you'll want to use your favorite text editor to open the `emacs.d/init.el` file. This file is the global configuration for Emacs instances that run under your current user profile. `init.el` is short for initialize Emacs Lisp, and is loaded before the `~/.emacs` configuration file.

Once you've opened `.emacs.d/init.el` in your favorite text editor, you'll want to install the Emacs major mode `clojure-mode` by pasting and saving the following code:

```
(require 'package)
(add-to-list 'package-archives
  '("marmalade" . "http://marmalade-repo.org/packages/"))
(package-initialize)
```

Once you have saved the `.emacs.d/init.el` file, you'll want to open the Emacs editor by typing the Emacs command into the console command line. Once the editor is open, you'll want to hold the *Alt* key and press the *x* key. If done correctly, you should see a small status window at the bottom of the editor displaying the text **M-x**.

Next you'll want to type the command `package-list-packages` to list the available Emacs extension packages. If done correctly, you should see what's called a package list. This list will display the package name, version, and a short description as to what the package does. If you don't see this screen, please try again.

If you do see this screen, you'll want to search for the term **clojure-mode** by holding the *Ctrl* key and then pressing the *s* key. This should bring up a search message input screen at the bottom of the Emacs editor. Once you see this screen, just type the word `clojure-mode` and Emacs should focus and highlight the found package. When you have located the package, hold the *Ctrl* key and then press the *a* key to move your cursor to the beginning of the line. If done correctly, your cursor should be in front of the `clojure-mode` package, and you can safely press the *i* key to mark the package for installation. When done correctly the letter **I** will appear to the furthest left of the package Name.

You can now press the *x* key to execute the install process of the `clojure-mode` package. Once the installation of the package is complete, Emacs will display a status message of warnings and then finally the outcome of the installation process. If the installation process has failed for any reason, please read the project documentation and issues thread on the project's Github homepage for possible solutions. This web page can be found at <https://github.com/jochu/clojure-mode>.

Now that the `clojure-mode` is installed, when you open files ending with the `.clj` extension you'll get some Clojure-specific editing features. This mode will make editing Clojure files easier by adding syntax-highlighting, proper indentation, alignment, spacing, and evaluation capabilities. To fully understand how helpful the `clojure-mode` is, try writing some code in a text file, and then try writing that same code in a `.clj` file.

Creating and editing CLJ files in Emacs

Once Emacs is set up, test that you don't get any errors when running the Emacs command from your operating system command line. Before moving on to setting up Emacs for Clojure editing, we'll have to cover some basic commands for operating Emacs. These commands will mostly cover buffer or frame manipulation, saving files, opening files, common edit operations, and then closing the Emacs editor.

Emacs has a special notation for expressing keyboard shortcuts. This notation makes it easier to read and write keyboard combinations. The combinations don't always, but do often, display a small message of what was performed by pressing the accepted combination. If the keyboard combination isn't recognized, the mini buffer will display a message stating that the combination hasn't been defined. Although messages are displayed within the mini buffer, the mini buffer can accept keyboard input when some commands require extra parameters. This following chart is an explanation of Emacs keyboard key notations:

C-	Press and hold the <i>Ctrl</i> key
M-	Press and hold the <i>Alt</i> key
S-	Press and hold the <i>Shift</i> key
DEL	The backspace key, but not literally the <i>Del</i> key
RET	<i>Enter/return</i> key
SPC	The space bar key
ESC	The escape key
TAB	<i>Tab</i> key

Now that you have Emacs and some understanding of key sequence notation, let's go over some basic buffer window manipulation. Please re-read the preceding chart if you're having any trouble understanding the following Emacs operations. Also, if there's a space in the notation sequence, you no longer have to hold down any of the keys before the space.

Before moving on to some exercises, take a quick moment to look over the following Emacs commands. These commands will help you with the exercises in this chapter.

Buffer window manipulation commands

<i>C-x 3</i>	Split a buffer window vertically to create a new buffer window. This is great when needing to look at multiple files side by side.
<i>C-x 2</i>	Split a buffer window horizontally to create a new buffer window.
<i>C-x 1</i>	Make the current buffer-in-context the only buffer visible. This is a faster way to remove multi-split windows and just focus on one. Although the other buffers are no longer visible, the buffers are still there and can still be accessed.
<i>C-x 0</i>	Close the buffer frame but not the buffer. This is good if you need to move a frame out of the way, but you still wish to work on it later.
<i>C-x k</i>	Close the buffer but not the frame. This is good when you want to close the buffer you're editing, but you don't wish to alter the frame layout.
<i>C-x o</i>	Change the buffer in a multi-split window frame.
<i>C-x C-c</i>	Close the Emacs editor.
<i>C-x left</i>	Go to the previous buffer window.
<i>C-x right</i>	Go to the next buffer window.

As an exercise, let's make a 2-by-2 editing square. In a new Emacs editor window, let's split the buffer vertically with the keyboard sequence *C-x 3*. If you did this correctly, you should see two buffers side by side. Next, we'll split one of these frames horizontally with the keyboard sequence *C-x 2*. If you did this correctly, you should have a total of three buffer windows. Next, we're going to switch our cursor focus on the yet-to-be-split frame with the keyboard sequence *C-x o*. If you did this correctly, your cursor should be visible in the tallest frame. Now for the final stage; in making a 2-by-2 editing square, split the tallest frame with the keyboard sequence *C-x 2*. You should see a complete square of four buffer windows if you did everything correctly. If not, please start over and try again.

If you've comfortably finished the previous exercise, the next command list and exercise set will help you open and save files.

File manipulation commands

<i>C-x C-w</i>	Write to file/save as
<i>C-x C-s</i>	Save file
<i>C-x C-f</i>	Open a file from location

As an exercise, let's open a new Emacs window and create, alter, and save a new file. Once you open a new Emacs window, let's create a new buffer that will write to a file called `testing.txt` by using the keyboard sequence *C-x C-f*. If you did this correctly, the mini buffer should prompt you for a file location. Type in the location as to where you want to save your file. For example, `/tmp/testing.txt` might be a good spot for a Unix-based machine, but Windows users might want to try `C:\testing.txt`. Once you type in the file location and hit the *Enter* key, you should get a new blank buffer window.

Now you can type `Hello World` and save your new file with the keyboard sequence *C-x C-s*. Now let's save the same file under another name using the keyboard sequence *C-x C-w*. If done correctly, the mini buffer will prompt you for the location you wish to save the file under. Again, `/tmp/testing-2.txt` might be a good spot for a Unix-based machine, but Windows users might want to try `C:\testing-2.txt`. Now that you have saved two separate files, close the Emacs editor with the keyboard sequence *C-x C-c*. If done correctly, Emacs should have exited and two files named `testing.txt` and `testing-2.txt` should contain the text `Hello World`.

Now that you can manipulate window buffers, open, and edit files, let's look at some common editing operations. Emacs has an insane amount of pre-defined shortcuts, so it's really important to narrow down and review only the operations that would be of use to you in an everyday setting. It's also important to try out unfamiliar commands to really understand how you may benefit from using them.

Common Editing Commands

<i>C-a</i>	Move cursor to the beginning of a line.
<i>C-e</i>	Move cursor to the end of a line.
<i>C-k</i>	Remove line in front of cursor.
<i>C-o</i>	Insert blank line after cursor.
<i>C-SPC</i>	Start and stop highlighting.
<i>M-w</i>	Copy highlighted text.
<i>C-w</i>	Cut highlighted text.
<i>C-y</i>	Paste previously copied text.

The commands in the preceding table should help get you started with basic Emacs editing. To fully understand how some of these key sequences work, create a new file and try to complete some basic editing tasks. Move on to the next section once you feel confident in manipulating buffer windows, opening and closing files, and making basic edits.

Running a Clojure REPL inside Emacs

What is a Clojure REPL? Read, Evaluate, Print, and Loop terminals help with development by giving developers a quick way to run and test code without having to write files or compile files already written. There are a few REPLs for Clojure, but the REPL that this section covers is REPL-y and nrepl.el. Clojure-mode allows for Emacs to run a Clojure REPL instance within Emacs by connecting to a REPL local server. Emacs will start the server for you or you can run your own REPL instance and tell Emacs to connect to it. Looking at the following output, REPL-y displays all of the helper commands that are available to you at startup session. This can be invoked inside of a clojure-mode buffer frame with the Emacs command *C-c C-z*.

```
nREPL server started on port 46865
REPL-y 0.1.10
Clojure 1.5.1
  Exit: Control+D or (exit) or (quit)
Commands: (user/help)
  Docs: (doc function-name-here)
        (find-doc "part-of-name-here")
  Source: (source function-name-here)
         (user/sourcery function-name-here)
Javadoc: (javadoc java-object-or-class-here)
Examples from clojuredocs.org: [clojuredocs or cdoc]
        (user/clojuredocs name-here)
        (user/clojuredocs "ns-here" "name-here")

user=>
```

The nrepl.el Emacs extension

Now that you have an idea as to what a Clojure REPL is, let's go back to Emacs and install nrepl.el. We're installing nrepl.el because it adds shortcut keys for manipulating both Clojure and REPL buffers that work well with clojure-mode. Once you learn what Emacs commands help you the most, a REPL experience outside Emacs might feel frustrating and less productive. Because nrepl.el and clojure-mode are hosted in the same extension repository (marmalade-repo.org), you don't have to add or change anything in your Emacs configuration. If you skipped the Emacs setup section, please go back and read about how to set up clojure-mode for Emacs. If you don't wish to use nrepl.el and you prefer to use the REPL that comes with clojure-mode (REPL-y), you can skip the rest of this section.

Just as when installing the clojure-mode, you want to open up the Emacs editor and type the Emacs command `M-x package-list-packages`. Once the package list is fully loaded, search for the nrepl package with the Emacs command `C-s nrepl`. Once you have found the nrepl package, you can mark the package for installation by pressing the `i` key. If done correctly, you should see an **I** to the far left of the package name. This means that the package is marked for installation and you can install the package by pressing the `x` key. Once the installation is complete, Emacs will give you an overall status of the installation outcome. If the installation didn't complete for any reason, please refer to the official nrepl.el project page for further information. The nrepl.el Github project homepage can be located at <https://github.com/technomancy/nrepl.el>.

Testing the setup

Once the installation of nrepl.el is complete, open a new temporary file named `testing.clj`. If you're using a Unix-based system, I would recommend the path `/tmp/testing.clj`. If you're using a Windows-based system, I would recommend the path `C:\testing.clj`. If you don't remember how to do this operation in Emacs, the command is `C-x C-f /tmp/testing.clj`. Put the following code into your new file and save it with the command `C-x C-s`.

```
(ns testing)
(defn book-example
  [x y z]
  (println "x" x "y" y "z" z))
```

After saving the file, split the buffer window vertically using the Emacs command `C-x 3`. You should now see two frames of the same buffer window. Now you can start your Clojure REPL with the Emacs command `M-x nrepl-jack-in`. If done correctly, one of your frame buffers will have the name `*nrepl*`, and the other will have the name `testing.clj`. You can now switch between your file buffer and the Clojure REPL frame using the Emacs command `C-x o`. Before moving on, let's take a look at a few helpful Emacs commands. Both the Clojure and the REPL buffers have their own set of Emacs commands. This section will go over a few very basic commands, but you can view the full command list at the nrepl Github project page at <https://github.com/kingtim/nrepl.el>.

Clojure Buffer Commands

<i>C-c C-l</i>	Loads a Clojure file. You use this command and type in the location of the file you wish to load, or you can press <i>Enter</i> to load the file of the current buffer. You'll need to do this for the REPL to recognize any files that you're working with.
<i>C-c C-e</i>	Evaluates the first form behind the cursor and displays the output in the REPL buffer frame. This comes in handy when testing a function you just wrote.
<i>C-c C-r</i>	Evaluates the highlighted region and displays the output in the REPL buffer frame. This is good for testing sections of a larger file without having to place the region in another file for testing.
<i>C-c C-b</i>	When the REPL is evaluating some code and it starts to hang, or run out of control, or you just wish to stop any further evaluation, you would use this command.
<i>C-c M-n</i>	To switch the REPL buffer to the current namespace of your working Clojure file, you would use this command. This is a lot easier than having to go to the REPL window and then typing <code>ns my-name-space</code> .
<i>C-c C-k</i>	Loads the current buffer into the REPL window.
<i>C-c C-l</i>	Loads a file into the REPL window.
<i>C-c C-d</i>	Displays the documentation string of a symbol at the point of the cursor.
<i>C-c C-j</i>	This command will open your default web browser to a Java documentation page for the symbol at the point of your cursor.

REPL Buffer Commands

<i>C-RET</i>	This command will close any open parenthesis and evaluate the REPL input. For example, you would use this when typing <code>(prn (reverse [1 2])</code> .
<i>C-j</i>	Because the <i>Enter/Return</i> key tells the REPL buffer to evaluate the input, you'll have to use this command to make a new line and maintain the proper indentation level.
<i>C-c C-u</i>	You would use this command to remove any text before the current cursor position.

REPL Buffer Commands

<i>C-c C-b</i>	Just as in the Clojure buffer, you would use this command to stop any further evaluation as the code is being evaluated.
<i>C-up or C-down</i>	Look through the REPL input history.
<i>TAB</i>	Complete the rest of the symbol at the cursor position. For example, you can type <code>pr</code> and hit <i>Tab</i> twice to get a list of possible completion symbols.
<i>C-c C-d</i>	Displays the documentation string of a symbol at the point of the cursor.
<i>C-c C-j</i>	This command will open your default web browser to a Java documentation page for the symbol at the point of your cursor.

To better understand the benefits of both `clojure-mode` and `nrepl.el`, let's try out some commands on `testing.clj`. If you haven't already, you can switch to the proper buffer by using the Emacs command `C-x o`. If done correctly, your cursor should be focused in the `testing.clj` file buffer. To use your files in the current REPL session, you'll have to load the current buffer or load a file from a specified location.

Because `testing.clj` is already open, all we have to do is load the current buffer with the Emacs command `C-c C-k`. If done correctly, the mini buffer should display the status `"#'testing/book-example"`. Now the REPL session is aware of the function `testing/book-example`. The same result can be achieved by using the Emacs command `C-c C-l`, but this requires a location to be specified. The default location is of the current focused buffer.

Having to type fully-qualified names can be a pain. Luckily for us, `nrepl.el` has a nice command that allows us to switch namespaces so we don't have to type the full names of the functions we wish to call. While still in the `testing.clj` buffer frame, use the Emacs command `C-c M-n` to tell the REPL session to switch to the current namespace of the file buffer. If done correctly, the REPL session input line should start with `"testing>"` and above that line should be `"#<Namespace testing> "`.

Now we can switch to the REPL buffer to call our functions by using the Emacs command `C-c C-z`. Once the cursor takes focus within the REPL frame, type `"(book-example 1 2 3)"` and hit the *Enter* key. If done correctly, you should see the output `"x 1 y 2 z 3"`. You should now switch back to the file buffer with the Emacs command `C-x o`.

Without saving the changes to `testing.clj`, place `"(def abc 123)"` at the bottom of the file. Because using commands such as `C-c C-k` and `C-c C-l` require the file to be saved before it can be loaded into the REPL session, we'll have to use another command to load our new variable. Placing the cursor after the definition of the variable `abc`, use the Emacs command `C-c C-e` to evaluate our definition form. If done correctly, the mini buffer should display the status message `"#'testing/abc"`.

Switch to the REPL session by using the Emacs command `C-c C-z`. Put `"abc"` in the input and hit the *Enter* key. If done correctly, you should see the output of the value `123`. You should now switch back to the file buffer with the Emacs command `C-x o`. Save the changes of `testing.clj` with the Emacs command `C-x C-s`.

Go to the bottom of the file where the definition of the variable `abc` is located. Make a new line and type `"(println abc)"`. Then, place your cursor over the letter *p* in the `println` symbol and use the Emacs command `C-c C-s`. If done correctly, your REPL session is no longer visible and you should see the source code of the `print` line function. If you were to repeat the previous steps, but use the Emacs command `C-c C-d` instead of `C-c C-s`, you would get the documentation of the function instead of the source code.

You can now restore your REPL session frame by switching to the `testing.clj` file buffer by using the Emacs command `C-c C-z`. If done correctly, the file buffer and the REPL session should be side by side with your cursor focus in the REPL session. When accidentally evaluating code that hangs or code that runs for a long period of time, sending a break signal will stop the process and return you to the REPL input line. Type `"(Thread/sleep 90000)"` in the REPL session and press *Enter*. Break the sleep process by using the Emacs command `C-c C-b`. This same command can also be issued from the file buffer frame when evaluating code with the Emacs command `C-c C-e`.

Because the REPL session evaluates input when pressing the *Enter* key, we have to use an Emacs command to make a new line and to keep proper alignment. The Emacs command to do this is `C-j`. To test this out, type `"(println "` and then the command `C-j`. Type `"(reverse "` and then the command `C-j`. Type `"[1 2 3])" ,` and then hit the *Enter* key to evaluate the input. If done correctly, your input should look like the following code:

```
testing> (println (reverse [1 2 3]))
```

You can now close the Emacs editor with the Emacs command `C-x C-c`. Hopefully the exercises in this section helped you get a good beginners grip on both Emacs and the Clojure REPL. You can get more information about Emacs at <https://www.gnu.org/software/emacs/>, and more information about `nrepl.el` at <https://github.com/kingtim/nrepl.el>.

Leiningen and project management

Leiningen is both one of the easiest ways to get started with Clojure and a Clojure project automation utility. Before Leiningen came around, the state of Clojure documentation wasn't nearly as good as it is today, and there was no simple way to start a new Clojure project. After the release of Leiningen, it has since become the de facto standard for starting and building a Clojure application.

One of the best things about Leiningen is that it makes dependency management very easy. If there's a missing dependency, Leiningen will attempt to fetch and install the missing dependency from a remote location. All project dependencies are downloaded from an Apache Maven repository if the dependency doesn't already exist on the system. Alternative and additional repositories can also be defined within the project configuration. Leiningen is also extendable with plugins and project templates. This adds another layer of convenience because there are tons of plugins and templates that probably already solve a problem that you have. You can view a list of just a few plugins on the official Leiningen Github repository wiki page at <https://github.com/technomancy/leiningen/wiki/Plugins>. Another great thing about Leiningen is that it can compile and package your project into many formats. Some build formats are only available by leveraging the Leiningen plugin system, but by default there are two build formats. You can build your project into a JAR file without dependencies included, or you can build your project into a JAR with all dependencies included.

Installing Leiningen and starting a project

Leiningen requires Version 6 or higher of the Java Development Kit. You can update your JDK installation by downloading the latest version of the Java platform from the official website at <http://www.oracle.com/technetwork/java/javase/downloads/index.html>. Once the installation is complete, verify the version by using the command-line command `Java version`. If you're a Windows user, you can skip this paragraph. The first step is to make the directory `~/bin` if it doesn't exist. You can do this by opening the command line and entering the command `mkdir ~/bin`. Now edit the file `~/bashrc` and add the command `export PATH=$PATH:~/bin`. Now run the command-line command `source ~/bashrc`. This allows for the programs placed in the `~/bin` directory to be accessible from any terminal session under your user profile. The next step is to download the Leiningen script from <https://raw.githubusercontent.com/technomancy/leiningen/stable/bin/lein>. If you have the `wget` program installed, you can use the command `wget {URL}`. If you have the `curl` program installed, you can use the command `curl {URL} -O lein`. Once the download is complete, move the downloaded file to the `~/bin` directory with the command-line command `mv lein ~/bin; chmod 755 ~/bin/lein`.

If done correctly, the `lein` command-line command should be available to you from any terminal path. Now you can run the command-line command `lein self-install`. If you're using a Mac or another Unix-based operating system, you can skip this paragraph. Windows operating systems can download the lein batch file from <https://raw.githubusercontent.com/technomancy/leiningen/stable/bin/lein.bat>. If you're running a Cygwin Linux environment application (<http://cygwin.com>), you can attempt the instructions in the previous paragraph.

Once the proper files have been downloaded, you can create a new Clojure project using the command-line command `lein new test-project`. Windows users will have to use `lein.bat` instead of `lein` when entering the command. If done correctly, you should have a new directory named `test-project` and it should have a similar tree structure as the following example:

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

6 directories, 5 files

To better explain the directories, the `doc` folder contains the project documentation, the `src` folder contains the code for your project, the `resources` folder contains file resources for the project, and the `test` folder contains assertion tests for the project. The files that end with the suffix `md` are markdown text-files, and you can review the markdown syntax at <http://daringfireball.net/projects/markdown/syntax>. The files that end with the suffix `clj` are obviously Clojure source-code files. The most important file in a lein project is `project.clj`. Leiningen will get all of its information about the project from this file. Because not all Clojure projects are targeted to be a standalone application, the default `project.clj` file won't get your project running. You'll have to edit this file to look like the following code:

```
(defproject test-project "0.1.0-SNAPSHOT"
  :description "FIXME: write description"
  :url "http://example.com/FIXME"
  :license {:name "Eclipse Public License"
            :url "http://www.eclipse.org/legal/epl-v10.html"}
  :main test-project.core
  :dependencies [[org.clojure/clojure "1.5.1"]])
```

The attribute with the key `:main` tells the lein application where to start the program. The value of the attribute is set to `test-project.core` because the namespace and file location of our main function will be located at `src/test_project/core.clj`. Now that lein knows where to start the application, you'll have to edit the file `src/test_project/core.clj` to reflect the following code:

```
(ns test-project.core
  (:gen-class))

(defn -main
  [& args]
  (println "Arguments"
           (count args))
  (doseq [a args]
    (println "Argument: " a)))
```

By defining the function `-main`, lein is able to call the function when the command-line command `lein run` is executed. Try running the command-line command `lein run 1 2 3`. If done correctly, you should see the following output:

```
Arguments 3
Argument:  1
Argument:  2
Argument:  3
```

When running a Clojure REPL from Leiningen, there are, by default, two ways to start a REPL session with the lein application. You can use the command-line command `lein repl` to both start and connect to a locally-hosted REPL server. If you wish to use another REPL client such as `nrepl.el`, you can start a REPL server with the command-line command `lein repl :headless`. If done correctly, lein will display a message similar to `nREPL` server started on port 37174.

If you choose to run just the REPL server, you can connect to the server with a `lein` command or an Emacs command. If you don't wish to use Emacs, you can connect with the command-line command `lein repl :connect localhost:0000`. The four zeros at the end of the command should actually be the port number that the REPL server is listening on. If you're using Emacs, use the Emacs command `M-x nrepl`. This command will ask you for the location of the REPL server and the port number of the listening server. Because the server is running on your local machine, the server location can either be the IP address `127.0.0.1` or you can use the fully-qualified domain name of your machine. If you don't know the name of your machine, you can use `localhost` for the server location. If you don't know or remember the port that the REPL server is listening on, you can always look at the command-line output of the REPL server. The output always reports what the listening port number of the REPL server is.

Including Clojure or Java libraries in your project

Leiningen fetches and installs libraries from remote Maven repositories if the library doesn't exist on the local machine. When needing to find a Java library, you can search the central Maven repository to find the proper library and version. The central repository search page is located at <http://search.maven.org>. Clojure has a similar repository for Clojure libraries and the search page is located at <http://clojars.org>.

Looking at the file `project.clj`, there's a collection value for the dependencies setting. This collection can hold multiple vectors that contain two values. The first value is the name of the Maven artifact, and the second value is the version number. The following example of the `project.clj` file describes which version of Clojure to use with the dependencies setting. The first and only item in the dependencies list contains the artifact id of `org.clojure/clojure` with the version number of `1.5.1`. After adding additional dependencies, you can fetch the resources using the command-line command `lein deps`.

```
(defproject test-project "0.1.0-SNAPSHOT"
  :description "FIXME: write description"
  :url "http://example.com/FIXME"
  :license {:name "Eclipse Public License"
            :url "http://www.eclipse.org/legal/epl-v10.html"}
  :main test-project.core
  :dependencies [[org.clojure/clojure "1.5.1"]])
```

Compiling your project to a Java JAR

Leiningen has two very convenient commands to compile and package a project into a portable Java archive. Because of some licensing restrictions of the dependencies used in some projects, you might have to package your JAR file without the dependencies included. You can achieve this with the command-line command `lein jar`. If you need to package your project and all of the required dependencies, you can run the command-line command `lein uberjar`. Using the same test project from the previous examples, run the command-line command `lein jar`. Once the process is complete, the output file should be located at `test-project/target/test-project-0.1.0-SNAPSHOT.jar`. We can run this file with the command-line command `java -jar {FILE-NAME}`. Because the `lein` command didn't include the dependencies, the Clojure language wasn't included in the JAR bundle. Trying to run this file will result in an error complaining that the Clojure classes weren't found. The following is an example of this type of error:

```
$ java -jar target/test-project-0.1.0-SNAPSHOT.jar 1 2 3
Exception in thread "main" java.lang.NoClassDefFoundError: clojure/lang/IFn
-----
    at java.lang.Class.getDeclaredMethods0 (Native Method)
    at java.lang.Class.privateGetDeclaredMethods (Class.java:2451)
    at java.lang.Class.getMethod0 (Class.java:2694)
-- cut short --
```

Now let's make a JAR file with all the dependencies included. In the same project, run the command-line command `lein uberjar`. The file output should be located at `test-project/target/test-project-0.1.0-SNAPSHOT-standalone.jar`. Try running this file with the same command-line arguments as the previous JAR file. The output should match the output of running the command-line command `lein run 1 2 3`. The following is an example of this:

```
$ java -jar target/test-project-0.1.0-SNAPSHOT-standalone.jar 1 2 3
Arguments 3
Argument: 1
Argument: 2
Argument: 3

$ lein run 1 2 3
Arguments 3
Argument: 1
Argument: 2
Argument: 3
```


Leiningen

Leiningen is a Clojure utility for managing projects and running tasks. Leiningen is sometimes referred to as simply `lein`, and has become the de facto standard for starting and building a Clojure application. `lein` is also one of the quickest ways to start developing with Clojure.

When you start a new `lein` project, the dependencies, class paths, and directory structure are already created and configured for you. Leiningen can also use project templates to generate a project that's different from the default `lein` setup. These templates will already have a default set of dependencies configured for your Clojure project. Leiningen downloads either Clojure or Java dependencies from a remote Maven repository if the dependency doesn't already exist on the local machine. The dependencies configuration is stored inside `project.clj`. Two main places to find Java and Clojure dependencies are <http://search.maven.org> and <http://clojars.org>.

Plugins can be used to extend `lein`'s capabilities. By default, `lein` only compiles Clojure source, but there are plugins for other languages such as ClojureScript. Plugins can be stored in a `lein` profile setting, but project-specific plugins are defined inside `project.clj`.

When bundling your project into a portable Java Archive or JAR, you need to define the main class in `project.clj` with the setting key of `:main` and a namespace as the value. The namespace value has to be a namespace that generates a class for the JAR to be executable. If the JAR is to be executable, the class must contain a function with the name `-main`. When making a standalone executable, use the command-line command `lein uberjar` to include the project dependencies.

Leiningen Commands

<code>lein deps</code>	Gets check and fetches missing project dependencies.
<code>lein deps :tree</code>	Displays a project dependency tree.
<code>lein compile</code>	Compiles the current project.
<code>lein jar</code>	Builds a JAR file that doesn't include all of the project dependencies.
<code>lein uberjar</code>	Builds a JAR file that includes all of the project dependencies.
<code>lein install</code>	Installs the current project to a local Maven repository.

Summary

The Emacs editor is a keyboard shortcut-centric document editor developed in the 1970s. The editor is still in use and developing today, but Emacs is drastically different from many common editors. Many popular document editors have keyboard shortcut capabilities, but Emacs can only be used effectively by knowing the keyboard combinations to perform common operations needed on a daily basis. Another major difference that sets Emacs apart from other popular editors is that Emacs can be extended with a programming language called Emacs Lisp. Emacs Lisp doesn't have to live inside the Emacs editor, but it's mainly used to add or modify the editor's capabilities. Because Emacs Lisp is a very complete language, many Emacs extensions are built for interacting with other services and applications such as Twitter, Reddit, and e-mail.

Emacs Lisp is also used to add programming-language-specific features. These features often operate in what's called a major mode. A language-specific major mode commonly handles syntax-highlighting, proper indentation levels, and file-type detection. Because each major mode is developed and maintained by a different community of developers, the quality of a mode will vary depending on the developer's goals. The Clojure major mode for Emacs is called `clojure-mode` and it can be installed from an Emacs Lisp extension repository. Although Emacs is great for general purpose editing, the Clojure major mode makes the Emacs editor act more like an IDE.

The following chart is an explanation of Emacs keyboard key notations:

C-	Press and hold the <i>Ctrl</i> key
M-	Press and hold the <i>Alt</i> key
S-	Press and hold the <i>Shift</i> key
DEL	The backspace key, but not literally the <i>Del</i> key
RET	<i>Enter/return</i> key
SPC	The space bar key
ESC	The escape key
TAB	<i>Tab</i> key

4

Features, Functions, and Macros

This chapter briefly goes over some of the Clojure key components. More specifically, Clojure namespaces, Java classes, immutability, metadata, lazy sequences, collection de-structuring, functions, relationships, and macros.

Here are the major topics that this chapter will cover:

- Handling Java packages from within in a Clojure application.
- The importance of Clojure data types being immutable by default, explicitly dynamic, and contextually mutable (loop, binding, and so on). Attaching data to data without modifying the value of the host data (metadata).
- Creating and using collection types of an infinite continuation without consuming all available memory (lazy sequencing).
- Building and binding locally-scoped attributes of a collection type.
- Functions with multiple arities and collection-like attributes.
- Anonymous functions and the benefit of the syntax involved.
- Understanding basic macro syntax with an example.

Namespaces

Looking at the following code snippets, a Clojure namespace is an object that represents a group of Clojure symbols. Similar to private Java class methods and fields, a symbol in a Clojure namespace can only be accessed from another namespace if the symbol isn't explicitly private. Each symbol is a name assignment for a binding, macro, or a function within the parent namespace.

The `*ns*` variable represents the current namespace and is always bound to a `clojure.lang.Namespace` namespace. The following code snippet prints the current namespace object:

```
$ clj
Clojure 1.5.1
user=> (println "Current Namespace" *ns*)
Current Namespace #<Namespace user>
nil
```

When defining a symbol, the fully qualified name of the symbol will be the symbol name prefixed with the namespace name and a slash. Other namespaces within the application will be able to access the symbol with the fully qualified name. The following example shows the fully qualified name of a symbol being accessed from another namespace:

```
user=> (def a-symbol 123)
#'user/a-symbol
user=> (ns example-ns)
nil
example-ns=> user/a-symbol
123
```

The `require` form will have to be used to allow a namespace to be aware of another namespace. You can then use the fully qualified name of the symbols within the current namespace.

```
example-ns=> (require 'clojure.string)
nil
example-ns=> (clojure.string/split "a b c" #"\\s")
["a" "b" "c"]
```

Looking at the following examples, if you require a namespace with a long segmented name or if you just prefer an alternative name for the namespace, you can rename the required namespace using an alias. When using an alias, try not to share a name with an existing symbol. An alias won't override an existing symbol within the current namespace but it may cause confusion.

```
$ clj
Clojure 1.5.1
user=> (require '[clojure.string :as str])
nil
user=> (str/join ["clojure.string" "=> alias is str"])
"clojure.string=> alias is str"
user=> (str "To String function & not namespace alias")
"To String function & not namespace alias"
user> (require '[clojure.string :as xyz])
nil
user=> (xyz/join ["clojure.string" "=> alias is xyz"])
"clojure.string=> alias is xyz"
```

Using an entire namespace within another namespace can sometimes cause unexpected collisions that cause symbols to be overridden:

```
$ clj
Clojure 1.5.1
user> (ns example)
nil
example> (defn rest [& _] "example")
WARNING: rest already refers to: #'clojure.core/rest in namespace:
example, being replaced by: #'example/rest
#'example/rest
example> (ns user)
nil
user> (use 'example)
WARNING: rest already refers to: #'clojure.core/rest in namespace:
user, being replaced by: #'example/rest
nil
user> (rest [1 2 3])
"example"
```

To help overcome this issue, you can use the `:only` keyword to limit what symbols are available from the current namespace:

```
user> (ns xyz)
nil
xyz> (def abc 123) (def xyz 321)
#'xyz/abc
#'xyz/xyz
xyz> (ns user)
nil
user> (use '[xyz :only [abc]])
nil
user> abc
123
user> xyz
CompilerException java.lang.RuntimeException: Unable to resolve
symbol: xyz in this context, compiling:(NO_SOURCE_PATH:0:0)
```

Another way to achieve a similar result is to require a namespace and refer to the needed symbols with the `:refer` keyword:

```
user> (require '[clojure.string :refer [split]])
nil
user> (doc split)
-----
clojure.string/split
```

```
([s re] [s re limit])  
Splits string on a regular expression. Optional argument limit is  
the maximum number of splits. Not lazy. Returns vector of the splits.  
nil
```

Java inside Clojure

The `use` and `require` forms can only be used on Clojure namespaces. When a namespace requires resources from a Java class, the `import` form is to be used:

```
user> (import java.util.Date)  
java.util.Date  
user> (import [java.util Date Calendar])  
java.util.Calendar  
user> (java.util.Calendar/getInstance)  
#inst "2013-04-11T15:48:05.966+09:00"
```

After importing a Java class into your Clojure namespace, you can access the class without the fully qualified name:

```
user> (import '[java.util Date])  
java.util.Date  
user> (Date.)  
#inst "2013-04-11T06:47:39.332-00:00"
```

If the Java class is in the proper class path, but not imported, the class can only be reached by the fully qualified name. Because Clojure symbols don't normally contain decimals, with the exception of namespaces and Java-specific calls, using a fully qualified name is required to access a symbol containing a decimal/dot:

```
user> (java.util.Calendar/getInstance)  
#inst "2013-04-11T15:48:05.966+09:00"  
user> (def bad.symbol 123)  
#'user/bad.symbol  
user> user/bad.symbol  
123  
user> bad.symbol  
CompilerException java.lang.ClassNotFoundException: bad.symbol,  
compiling: (NO_SOURCE_PATH:0:0)
```

Clojure automatically imports all the Java classes found in the `java.lang` Java package. This means that the fully qualified name isn't required to access these specific Java classes:

```
user> (Boolean. true)  
true  
user> (+ (Integer. "1") (Integer. -1))
```

```

0
user> (Character. \a)
\a

```

When adding a decimal at the end of a Java class, Clojure assumes you are calling a Java class constructor. There are a few ways to call Java class methods but you have to take special care to recognize when you're calling a method that causes mutation to the class instance:

```

user> (def cookie
      (doto(java.net.HttpCookie. "cookie" "value")
        (.setDomain "http://xyz.com")
        (.setComment "Example Cookie")))
#'user/cookie
user> (.getDomain cookie) (.getComment cookie)
"xyz.com"
"Example Cookie"
user> (.. cookie (setDomain "easy on the eyes"))
nil
user> (.setDomain cookie "not easy on the eyes")
nil
user> (.toString cookie)
"cookie=\"value\"; $Domain=\"not easy on the eyes\""

```

Some Java classes actually don't have any public constructors and only contain static fields and methods such as the `java.lang.Math` class. When a static field or method is required, constructing a new instance of the parent class isn't necessary:

```

user> java.util.Calendar/APRIL
3
user> java.util.Calendar/MAY
4
user> Integer/MAX_VALUE
2147483647
user> (. Integer MAX_VALUE)
2147483647
user> (Integer/parseInt "123")
123
user> (.. Integer (parseInt "123") doubleValue)
123.0

```


When working with Java interfaces, you can implement the methods of the interface with the `reify` form. Keep in mind that the method return type can't be changed. The following is an example of implementing the Java interface `javax.tools.Tool`:

```
user> (def string-fn
      (reify javax.tools.Tool
        (toString [this] "Override")
        (getSourceVersions [this] #{"Example"})))
#'user/string-fn
user> (.toString string-fn)
"Override"
user> (.getSourceVersions string-fn)
#{ "Example" }
```

Using the proxy macro form, extending a Java interface or class is slightly different. The first argument of the proxy form is a vector of a class or classes and the second contains the parameters for calling the class constructor. Because the proxy form generates a new class without a name or identifier, the class is anonymous. In the following example, the anonymous class is bound to a Clojure symbol so that the class instance can be manipulated by other functions after calling the instance constructor:

```
user> (defn proxy-example [port]
      (proxy [java.net.ServerSocket] [port]
        (toString [] (str "Bound to port " port))
        (getLocalPort [] (println "Listening on" port) port)))
#'user/proxy-example
user> (def bound-socket (proxy-example 8888))
#'user/bound-socket
user> (.getLocalPort bound-socket)
Listening on 8888
8888
user> (str bound-socket)
"Bound to port 8888"
```

The proxy and reify forms can only implement methods already defined within the class. Because of this limit, extending a Java class with new functions can be achieved using a Clojure protocol and the `extend-type` form. In the following example, the protocol named `Example` acts in a similar way to a Java interface. Any Java class that is extended by the protocol will have a set of functions that can only be used on other classes extended by the same protocol.

```
user> (defprotocol Example (size [date]))
Example
user> (extend-type java.util.Date
      Example)
```

```

        (size [date] (count (str date))))
nil
user> (def example-date (java.util.Date.))
#'user/example-date
user> (str example-date)
"Sat Apr 13 11:59:02 JST 2013"
user> (size example-date)
28
user> (size "example string")
IllegalArgumentException No implementation of method: :size of
protocol: #'user/Example found for class: java.lang.String  clojure.
core/-cache-protocol-fn (core_deftype.clj:541)
user> (extend-type java.lang.String
      Example
      (size [s] (count s)))
nil
user> (size "example string")

```

Immutability

Clojure objects can't be changed by default. Because Clojure objects aren't mutable, a Clojure function will return a new modified copy of the original object it operates on. With structural sharing, the new object will share pieces of data with the original object and only create new data that doesn't exist in the original object instance. This might not make a lot of sense if you're coming from a language such as PHP or Python, but you'll soon be grateful for the stability that immutable objects provide.

In the following example, notice how the original string doesn't change after calling `clojure.string/replace`:

```

user> (def s "This is an example")
#'user/s
user> (clojure.string/replace s "an example" "a string")
"This is a string"
user> s
"This is an example"
user> (def i 10)
#'user/i
user> (inc i)
11
user> i
10

```

An immutable object is an assurance that your data won't unknowingly change. In the following example, notice how the `computer` variable doesn't change:

```
user> (def computer {:os "Arch Linux" :memory :4096-MB})
#'user/computer
user> (def downgrade (assoc computer :memory :1024-MB))
#'user/downgrade
user> downgrade
{:os "Arch Linux", :memory :1024-MB}
user> (def change-os (assoc downgrade :os "Haiku OS"))
#'user/change-os
user> change-os
{:os "Haiku OS", :memory :1024-MB}
user> (println "Computer didn't change" computer)
Computer didn't change {:os Arch Linux, :memory :4096-MB}
nil
```

Although objects aren't mutable, mutation can occur when using the `recur` form on the `loop` form bindings.

```
user> (loop [v []
             i 1]
      (if-not (= i 4)
        (recur (conj v i) (inc i))
        (do (println "Now immutable") v)))
Now immutable
[1 2 3 4]
```

Dynamic objects

Clojure syntax is very explicit when defining and manipulating dynamic Clojure objects. The compiler will throw a nasty fit if you try to improperly change or define an object:

```
user> (def a)
#'user/a
user> (binding [a 123] (println a))
IllegalStateException Can't dynamically bind non-dynamic var: user/a
clojure.lang.Var.pushThreadBindings (Var.java:353)
```

All dynamic objects must tell the compiler that they are truly dynamic, or the compiler will refuse to treat them:

```
user> (def *b*)
Warning: *b* not declared dynamic and thus is not dynamically re-
bindable, but its name suggests otherwise. Please either indicate
```

```

^:dynamic *b* or change the name. (NO_SOURCE_PATH:1)
#'user/*b*

user> (def ^:dynamic *b* :127.0.0.1)
#'user/*b*
user> (binding [*b* 123]
      (println *b*)
      (binding [*b* 456]
        (println *b*)))
123
456
nil
user> (println *b*)
:127.0.0.1
nil

```

Metadata

The Clojure language has the ability to attach data to data without altering the value. Looking at the following example, the optional metadata key value set is placed before the symbol name:

```

user> (defn ^{:doc "I'm a function"
              :mp3 false
              :private true
              :another-fn (comp #(apply str %) rest str)}
      function [x] x)
#'user/function
user> (function 123)
123

```

Notice how the documentation is stored under the `:doc` key. Using the `doc` form, you can retrieve the documentation string of any Clojure symbol with a `:doc` metadata key:

```

user> (clojure.pprint/pprint (meta #'function))
{:arglists ([x]),
 :ns #<Namespace user>,
 :name function,
 :column 1,
 :private true,
 :another-fn #<core$comp$fn__4156 clojure.
core$comp$fn__4156@1bf2724a>,

```

```
:doc "I'm a function",
:line 1,
:file "NO_SOURCE_PATH",
:mp3 false}
nil
user> (doc function)
-----
user/function
([x])
  I'm a function
nil
```

To view the metadata, if any, use the `meta` form on an object. When you want only the metadata of the Clojure symbol and not the data bound to the symbol, prefix the symbol with `#'`:

```
user> (clojure.pprint/pprint (meta #'function))
{:arglists ([x]),
 :ns #<Namespace user>,
 :name function,
 :column 1,
 :private true,
 :another-fn #<core$comp$fn__4156 clojure.core$comp$fn__4156@194c8bc6>,
 :doc "I'm a function",
 :line 1,
 :file "NO_SOURCE_PATH",
 :mp3 false}
nil
```

Because metadata doesn't alter the value of the data, it's possible to attach functions to functions:

```
user> (def meta-fn (:another-fn (meta #'function)))
#'user/meta-fn
user> (meta-fn "Drop first")
"rop first"
```

Using the `with-meta` form, you can add metadata to a return object instead of the bound symbol:

```
user> (def data-with-meta
      (with-meta ["Clojure" "Language"]
        {:type :programming :artificial true}))
#'user/data-with-meta
user> (clojure.pprint/pprint (meta #'data-with-meta))
{:ns #<Namespace user>,
```

```

      :name data-with-meta,
      :column 1,
      :line 1,
      :file "NO_SOURCE_PATH"}
nil
user> (meta data-with-meta)
{:artificial true, :type :programming}
nil
user> data-with-meta
["Clojure" "Language"]

```

Because metadata is a key value map, you can access the values by the key name:

```

user> (defn artificial? [x]
      (true?
        (:artificial (meta x))))
#'user/artificial?
user> (artificial? "abc")
false
user> (meta data-with-meta)
{:artificial true, :type :programming}
nil
user> (artificial? data-with-meta)
true

```

Lazy sequences

Values in a Clojure lazy sequence aren't evaluated until the value is needed. For example, the following function won't work because it tries to evaluate all the values in an infinite recursive call to itself:

```

user> (defn nums []
      (cons (rand-int 10)
            (nums)))
#'user/nums
user> (take 10 (nums))
StackOverflowError  java.util.Random.nextDouble (Random.java:444)

```

By wrapping a sequence in the `lazy-seq` form, the body of the `lazy-seq` form will only be evaluated when needed and the results of the evaluation will be cached. The following is an example of the previous function, but this time it returns a lazy sequence:

```

user> (defn lazy-nums []
      (lazy-seq
        (cons (rand-int 10)
              (lazy-nums))))

```

```
                (lazy-nums)))
#'user/lazy-nums
user> (take 5 (lazy-nums))
(3 4 4 3 6)
user> (take 20 (lazy-nums))
(2 3 0 8 4 8 2 3 7 0 3 9 8 8 1 3 4 0 5 4)
```

Destructuring

Clojure collections can be taken apart with new bindings. The bindings hold a value within the collection and can be bound in multiple ways for multiple collection types. The following is the example collection that will be used for the rest of the destructuring examples:

```
user> (def abc {:a 1 :b 2 :c 3})
#'user/abc
user> (keys abc) (vals abc)
(:a :c :b)
(1 3 2)
user> (:a abc) (:c abc)
1
3
```

Using the `let` form, one way of binding a key value collection is to provide: a hash-map that describes the new binding and where the value should come from.

```
user> (let [{value-a :a
             value-b :b
             value-c :c} abc]
      (str value-c
            value-b
            value-a))
"321"
```

Non-key value collections can be bound by their position in the collection:

```
user> (sort (vals abc))
(1 2 3)
user> (let [[a b c] (sort (vals abc))
          ab [a b]
          bc [b c]
          ca [c a]]
      (println ab bc ca))
[1 2] [2 3] [3 1]
Nil
```

If the value is a collection, another collection can be provided to describe the bindings by position.

```
user> (let [xyz (sort (vals abc))
           {[_ y z] :xyz} {:xyz xyz}]
      (format "Y = %s Z = %s" y z))
"Y = 2 Z = 3"
```

When de-structuring by position, the ampersand character is used to bind the rest of the collection after some initial bindings. Values bound to the underscore character indicate that the value is not needed or won't be used.

```
user> (let [abc [:A :B :C]
           [a & bc] abc
           [_ _ c] abc]
      (format "A = %s BC = %s C = %s" a bc c))
"A = :A BC = (:B :C) C = :C"
```

The simplest way to bind single or multiple values by their key is to use as hash-set within the `let` form:

```
user> abc
{:a 1, :c 3, :b 2}
user> (let [{:keys [a]} abc] a)
1
```

When needing all the key bindings under a single symbol, provide a symbol value to the `:as` key within the hash-map:

```
user> (let [{:keys [a b c] :as d} abc]
      (println "abc = " d))
abc = {:a 1, :c 3, :b 2}
nil
```

Providing a map collection to the `:or` key will bind missing values to a default value. The `:as` key binding symbol will only represent non-missing values:

```
user> (let [{:keys [a b c d e]
           :or {d :d-notfound
                e :e-notfound}
           :as params} abc]
      (println "Params = " params
              "\nAll Values "
              a b c d e))

Params = {:a 1, :c 3, :b 2}
All Values 1 2 3 :d-notfound :e-notfound
nil
```


Functions and arity

A function always returns a value and the function can be used as an argument for other functions. Clojure has a few ways to define and compose functions, but the simplest and easiest way to define a new function is with the `defn` form:

```
user> (def func (fn [x y] (+ x y)))
#'user/func
user> (func 1 2)
3
user> (defn func [x y] (+ x y))
#'user/func
user> (func 1 2)
3
user> (def func #(+ %1 %2))
#'user/func
user> (func 1 2)
3
```

Arguments of a Clojure function, like collections or sequences, can be de-structured in much of the same way:

```
user> (defn func [[_ _ c]] c)
#'user/func
user> (func [1 2])
nil
user> (func [1 2 3])
3
user> (def abc {:a 1 :b 2 :c 3})
#'user/abc
user> (defn func
      [{a :a b :b c :c}]
      [a b c])
#'user/func
user> (func abc)
[1 2 3]
user> (defn func [{:keys [c]}] c)
#'user/func
user> (func abc)
3
user> (defn func [{:keys [a b c] :as d}] d)
#'user/func
user> (func abc)
{:a 1, :c 3, :b 2}
user> (dissoc abc :a)
```

```

{:c 3, :b 2}
user> (func (dissoc abc :a))
{:c 3, :b 2}
user> (defn func
  [{:keys [a b c]
    :or {a :not-found
        b :not-found
        c :not-found}
    :as d}]
  (println "D = " d "\nAll values" a b c))
#'user/func
user> (func {:aa 1 :ab 2 :ac 3})
D =  {:ab 2, :aa 1, :ac 3}
All values :not-found :not-found :not-found
nil

```

To assert that arguments in a function are optional, you can place the optional symbols after an ampersand character. If there's no symbol before the ampersand or if there's more than one symbol after the ampersand, the optional symbols must be in a collection.

```

user> (defn func [& x])
#'user/func
user> (defn func [& x y])
CompilerException java.lang.RuntimeException: Unexpected parameter,
compiling: (NO_SOURCE_PATH:1:1)
user> (defn func [& [x y]])
#'user/func
user> (defn func [x & y])
#'user/func
user> (defn func [& {:keys [a b c]}])
      (println a b c)
#'user/func
user> (func :a 1 :b 2 :c 3)
1 2 3
nil
user> (func :a 1 :b 2)
1 2 nil
nil
user> (func :a 1)
1 nil nil
nil
user> (func)
nil nil nil
nil

```

```
user>
user> (defn func [& [a b c]]
      (println a b c))
#'user/func
user> (func 1 2 3)
1 2 3
nil
user> (func 1 2)
1 2 nil
nil
user> (func 1)
1 nil nil
nil
user> (func)
nil nil nil
nil
```

Functions with required arguments must place the argument symbols before the ampersand if any:

```
user> (defn func [a & [b c]]
      (println "required" a)
      (println "optional" b c))
#'user/func
user> (func 1 2 3)
required 1
optional 2 3
nil
user> (defn func [{:keys [a]} & [b c]]
      (println "required" a)
      (println "optional" b c))
#'user/func
user> (func {:a 1} 2 3)
required 1
optional 2 3
nil
user> (func {:b 1} 2 3)
required nil
optional 2 3
user> (func 2 3)
required nil
optional 3 nil
nil
user> (func 2)
required nil
```

```

optional nil nil
nil
user> (func {:a 1})
required 1
optional nil nil

```

Another way to define optional arguments is to define different bodies based on the number of arguments provided to the function:

```

user> (defn func
      "This is a documentation string"
      ([a] (prn "Only a " a))
      ([a b] (prn "Only a b" a b))
      ([a b c] (prn "Only a b c" a b c)))
#'user/func
user> (doc func)
-----
user/func
([a] [a b] [a b c])
This is a documentation string
nil
user> (func 1)
"Only a " 1
nil
user> (func 1 2)
"Only a b" 1 2
nil
user> (func 1 2 3)
"Only a b c" 1 2 3
nil

```

Anonymous functions

An anonymous function is a function without a callable name binding. Sometimes you might only need a function in one place and one place only, so to create a function with a name can be a burden. Clojure currently has two ways of creating an anonymous function. There's the reader macro pound sign, and then there's the `fn` form:

```

user> (defn not-anonymous [x y] (+ x y))
#'user/not-anonymous
user> (not-anonymous 1 2)
3
user> (#(+ %1 %2) 1 2)
3

```

```
user> ((fn [x y] (+ x y)) 1 2)
3
user> ((fn name-but-not-callable [x y]
        (+ x y)) 1 2)
3
```

To get a better understanding of why you would do this, look at the following examples and decide which one looks easier to write:

```
user> (map #(+ 2 %) [1 2 3])
(3 4 5)
user> (defn add-two [i] (+ 2 i))
#'user/add-two
user> (map add-two [1 2 3])
(3 4 5)
```

Macros

Creating a macro is similar to writing a function with the exception that macros are written to generate code that gets evaluated later. You can think of a macro as a code template that gets placed around existing code.

```
user> (macroexpand '(when true "smile"))
(if true (do "smile"))
user> (eval (macroexpand '(when true "smile")))
"smile"
```

Writing macros is useful for writing code that can't be written with functions. For example, macros are often used to wrap dynamic bindings over other bodies of code. Looking at the following example, there are two symbol definitions. The first definition is a macro named `example` and the second is a function named `example-fn`. Both bodies of the definitions look similar, but the macro body has a few special characters. The ``` character before the beginning of the `let` body indicates that the body is a macro template. The trailing pound sign in the `let` definition means that the compiler should generate a unique name for the binding so that the binding can be accessed during evaluation. If the `let` bindings didn't have a trailing pound sign, the compiler would complain about the qualified name of the binding. The `~` character before a symbol indicates that the symbol should be accessed from outside the current template body. The `~@` characters before a symbol indicates that the symbol is a collection and that every item within it needs to be accessed in the same order:

```
user> (defmacro example [run? & code]
      `(let [msg# "Running Example: "]
        (when ~run?
          (println msg#)))
```

```

~@code)))
#'user/example
user> (defn example-fn [run? & code]
      (let [msg "Running Example: "]
        (when run?
          (println msg)
          code)))
#'user/example-fn

```

First, let's see what the macro does, and then let's see how it's different from the function:

```

user> (example false
          (println "abc")
          (map inc [1 2 3]))
nil
user> (example true
          (println "abc")
          (map inc [1 2 3]))
Running Example:
abc
(2 3 4)

```

Notice how the arguments weren't evaluated until the `run` argument was set to `true`. Let's see what the function version does with the same arguments in the following example. Pay close attention the `run` argument and the output:

```

user> (example-fn false
          (println "abc")
          (map inc [1 2 3]))
abc
nil
user> (example-fn true
          (println "abc")
          (map inc [1 2 3]))
abc
Running Example:
(nil (2 3 4))

```

With the function version, the arguments were evaluated regardless of the `run` argument. Notice that, when the `run` argument was set to `true`, the argument evaluated before the body of the function. This is why `abc` appears before `Running Example` whereas this doesn't happen in the macro version. Also, the return value of `(nil (2 3 4))` is very different from the macro's return value of `(2 3 4)`.

The syntax within a macro is dependent on non-alphanumeric characters to declare how a symbol or a body of code is to be evaluated. The following three examples briefly show the most common and basic macro syntax:

```
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
```

Summary

This chapter lightly brushed over some of the main features and capabilities of the Clojure language. If you didn't understand most of the examples presented in this chapter, please review the chapter until you have a confident understanding of each section. The chapters that follow after this will expand on each section in greater detail with a lot more examples, so it's important that you don't feel lost.

5

Collections and Sequencing

There will be a lot of REPL examples in this chapter and in the chapters to come. This chapter focuses on the Clojure 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.

After a brief explanation on what a Clojure collection and sequence are, this chapter will start by covering some basic operations that can be used on any collection. The sections after that will be data-specific until you reach the sequencing section. With the amount of code covered in this chapter, a short Clojure comment above the REPL session example may be the only explanation for some examples. By the end of this chapter, you should feel confident in manipulating Clojure collections and sequences.

In this chapter, you'll be learning about the following data types:

- Vectors
- Maps
- Sets
- Sequences

Collections

A Clojure collection is a data structure that can hold multiple objects. Because objects are immutable by default, you cannot make changes to a collection. When adding an object to a collection, without modifying the original collection, the added object is represented along with the original collection and is not a copy of it. When removing an object from a collection, without modifying the original collection, the remaining objects of the collection are represented and are not a copy of it. This is called structure sharing and this helps with performance and the Clojure concurrency model.

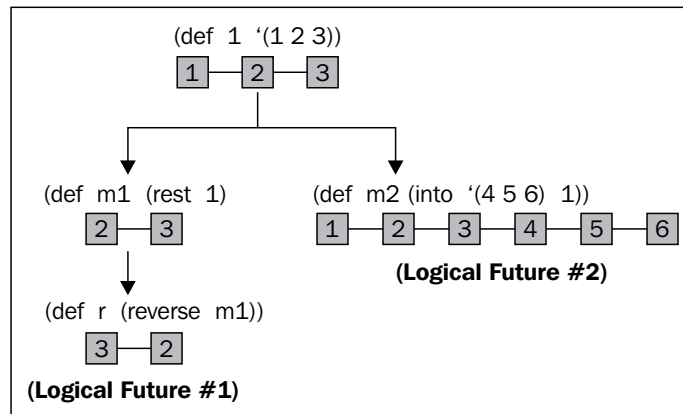


Image source: <http://debasishg.blogspot.jp/2010/05/grokking-functional-data-structures.html>

The Clojure collection data structures are represented as a persistent and immutable logical list that implements the Java Iterable interface. The following REPL session displays not all but many of the operations that can be performed on a collection. All examples that need further explaining will have a Clojure comment before the example to help clarify what's being presented.

Collections by example

Here we'll define a variable holding a collection of three numbers. This collection will be counted with the `count` form and the output message will be formatted with `format`. This is explained in the following code:

```
user> (def coll-example [1 2 3])
#'user/coll-example
user> (format "%d objects in the collection"
              count coll-example)
"3 objects in the collection"
```

The `empty` form will take a collection type and return an empty version of the same type:

```
user> (empty coll-example)
[]
```

The `not-empty` form will return `nil` if the collection is empty or else the collection is returned:

```
user> (not-empty coll-example)
[1 2 3]
user> (not-empty [])
nil
```

One way of creating a new collection based on existing collection data is to use the `into` and `conj` forms.

```
user> (into coll-example [4 5])
[1 2 3 4 5]
user> (conj coll-example 4 5)
[1 2 3 4 5]
```

Because `coll-example` is a Clojure vector, the `contains?` form will return a true Boolean value if the number provided is within the collection index range. Think of `array[0]` `array[1]` and so on.

The `partial` form in this example is used to create a function with predefined arguments. In this case, the `partial` form makes a function that calls the `contains?` form with `coll-example` as the first argument. The output of each `contains?` test will be returned in the same order that they were tested in:

```
user> (dec (count coll-example))
2
user> ;; The range here is 0 to 2 for coll-example
user> (let [contains? (partial contains? coll-example)]
      [(contains? 0)
       (contains? 1)
       (contains? 2)
       (contains? 3)])

[true true true false]
```

When using `contains?` on hash-maps you're searching for the presence of a key:

```
user> (contains? (hash-map :a "b") :a)
true
user> (contains? (hash-map :a "b") "b")
false
```

The `distinct?` form can check if there are any duplicate values within a collection:

```
user> (distinct? coll-example)
true
user> (conj coll-example 3)
[1 2 3 3]
user> (distinct? (conj coll-example 3))
false
```

The `empty?` form will return a Boolean value after testing whether a collection is empty or not:

```
user> (empty? coll-example)
false
```

When needing to test each value within a collection, you can use the `every?` and `not-every?` forms. These forms take a test function as the first argument and a collection for the second:

```
user> (every? integer? coll-example)
true
user> (every? string? coll-example)
false
user> (not-every? integer? coll-example)
false
```

The `some` form will return `nil` if the test function fails on every value within a collection, but it will also return the value of the function on the first test to pass:

```
user> (some string? coll-example)
nil
user> (some string? (conj coll-example "string"))
true
user> (some #(if (= 1 %) 1) [1 2 3])
1
user> (some #(if (= 3 %) :three) [1 2 3])
:three
user> (some #(if (= 4 %) :four) [1 2 3])
nil
```

The `not-any?` form is similar to the `some` form but it will only return a Boolean value:

```
user> (not-any? string? coll-example)
true
```

Use the `sequential?` form to test if a collection implements the sequential interface:

```
user> (sequential? coll-example)
true
```

Use the `associative?` form to see if the collection can accept new associations. You'll want to test for this when possibly working with Java arrays:

```
user> (associative? coll-example)
true
```

With vector collections, we can associate new values by using the `assoc` form to assign a value to a position:

```
user> coll-example
[1 2 3]
user> (assoc coll-example 0 2 1 3 2 4)
[2 3 4]
user> ;; When using a key-value based collection, you can
user> ;; replace or add key-values using the assoc form.
user> (def hmap-example (hash-map :a 1 :b 2))
#'user/hmap-example
user> (assoc hmap-example :c 3)
{:a 1, :c 3, :b 2}
```

Check to see if the collection type is a sorted map or set with the `sorted?` form:

```
user> (sorted? coll-example)
false
user> (sorted? (apply sorted-set coll-example))
true
```

Because Clojure can also be working with Java Objects, you want to use the `coll?` form to check that you're working with a `IPersistentCollection` and not a Java Array or something like it:

```
user> (coll? (to-array coll-example))
false
user> (coll? coll-example)
true
user> (seq? coll-example)
false
user> (seq? (seq coll-example))
true
user> ;; Checking the collection type
user> (vector? coll-example)
true
```

```
user> (list? coll-example)
false
user> (map? coll-example)
false
user> (set? coll-example)
false
```

Vectors

A Clojure vector is a collection that maintains object order and is presented as a list with square brackets. If object order is important, this is the collection type that you'll probably end up using. It's important to note that vectors holding keyword values might be mistaken for a `hash-map` key-value pair. So be sure to check the collection type with the `vector?` form to test that it isn't indeed a `hash-map`.

Vectors by example

```
user> (def coll-example [1 2 3])
#'user/coll-example
user> (type coll-example)
clojure.lang.PersistentVector
```

You can use the `vec` form to create a new vector from an existing collection in the following way:

```
user> (type #{1 2 3})
clojure.lang.PersistentHashSet
user> (type (vec #{1 2 3}))
clojure.lang.PersistentVector
```

You can also create a new vector by simply placing the values between two square brackets or by passing values to the vector form as arguments in the following way:

```
user> [1 2 3]
[1 2 3]
user> (vector 1 2 3)
[1 2 3]
```

To force the values to be interpreted as a specific primitive data type, you can use the `vector-of` form. What's presented in the following code is ASCII characters:

```
user> (vector-of :char 65 98 67 100)
[\A \b \C \d]
```

When using the `conj` form, you have to remember that vector types will place the new data at the end of the collection and in the front of the collection if it's a list type:

```
user> coll-example
[1 2 3]
user> (conj coll-example 4 5)
[1 2 3 4 5]
user> (apply list coll-example)
(1 2 3)
user> (conj (apply list coll-example) 4 5)
(5 4 1 2 3)
```

The `peek` form is similar to the `last` form but faster:

```
user> (time (last coll-example))
"Elapsed time: 0.080287 msecs"
3
user> (time (peek coll-example))
"Elapsed time: 0.04492 msecs"
3
```

`last`

A similar situation when wanting everything but the last value in a vector is shown in the following code:

```
user> (time (pop coll-example))
"Elapsed time: 0.078137 msecs"
[1 2]
user> (time (butlast coll-example))
"Elapsed time: 0.116885 msecs"
(1 2)
```

Because vectors are not a key-value collection, when using the `get` form you're requesting a position from within the vector:

```
user> (= (first coll-example) (get coll-example 0))
true
user> (get coll-example 2)
3
user> ;; Same thing with the assoc form.
user> (assoc coll-example 0 "one")
["one" 2 3]
```

To get a range of values from a vector, use the `subvec` form. The rest of the vector will be returned if only one number is provided for the range:

```
user> (def coll-example (vec (range 0 10)))
#'user/coll-example
[0 1 2 3 4 5 6 7 8 9]
user> (subvec coll-example 5)
[5 6 7 8 9]
user> (subvec coll-example 5 7)
[5 6]
```

Lists

Lists in Clojure are an easy way to group multiple elements. Elements in a list may be of any data type, but there are no key-value lists in Clojure. Other data types are responsible for key-value collections:

Lists by example

There are a few ways to make a list in Clojure:

```
user> (println "List =>" '(1 2 3))
List => (1 2 3)
user> (def -list (list :a :b :c 1 2 3))
#'user/-list
user> [(first -list) (last -list)]
[:a 3]
```

When you have a collection that you would like in a list, place the collection as the last argument in the `list*` form:

```
user> (list* 1 2 3 (range 4 7))
(1 2 3 4 5 6)
user> ;; Cons will add an element to the front of a list
user> (cons 4 '(1 2 3))
(4 1 2 3)
```

Be careful with the `conj` form. When working with a list, the element gets added to the front but, if it's a vector, the element gets added to the end:

```
user> (conj '(1 2 3) 4)
(4 1 2 3)
user> (conj [1 2 3] 4)
[1 2 3 4]
```

Be careful when using `peek`. When you're working with a persistent list, it's equal to the first but, when working with a vector, it's equal to the last:

```
user> (def -list (apply list (range 0 100)))
#'user/-list
user> (time (first -list))
"Elapsed time: 0.056577 msecs"
0
user> (time (peek -list))
"Elapsed time: 0.060406 msecs"
0
user> (type -list)
clojure.lang.PersistentList
```

Same care needs to be taken when using `pop`. It removes the first item in a list, but the last item in a vector:

```
user> (time (pop '(1 2 3)))
"Elapsed time: 0.060339 msecs"
(2 3)
user> (time (rest '(1 2 3)))
"Elapsed time: 0.099527 msecs"
(2 3)
```

Maps

Clojure has two types of maps. The `PersistentArrayMap` and the `PersistentHashMap` data types are responsible for key-value collections. The only exception is the `PersistentHashSet` data type, which looks like `#{:key-here :key-here}`. The difference between a `Map` and a `Set` is that the key is the value in a set, where as a key holds a separate value in a `Map`.

Maps by example

The hash-map form can be used to make a key-value collection:

```
user> (def -hmap (hash-map :key1 "value" :key2 "value2"))
#'user/-hmap
user> (vals -hmap)
("value2" "value")
user> (keys -hmap)
(:key2 :key1)
user> (:key2 -hmap)
"value2"
```


The array-map form can do the same:

```
user> (def -amap (array-map "a" 1 "b" 2))
#'user/-amap
user> (keys -amap)
("a" "b")
```

Remember to use the `get` form when needing to set default values for missing keys:

```
user> (get -amap "a" :not-found)
1
user> (get -amap "c" :not-found)
:not-found
user> (-hmap :key2)
"value2"
user> (-hmap :key3)
nil
user> (-hmap :key3 :not-found)
:not-found
```

Because the keyword here is not a keyword type, you'll have to put the key in front of the map to get its value:

```
user> (-amap "a")
1
```

Using the `zipmap` form, you can interleave a collection of keys and that of values to create a map:

```
user> (def -map
      (zipmap [:k1 :k2 :k3] (range 100 104)))
#'user/-map
user> -map
{:k3 102, :k2 101, :k1 100}
user> [(keys -map) (vals -map)]
[(:k3 :k2 :k1) (102 101 100)]
```

The `sorted-map` form sorts by keys from least to greatest:

```
user> (def -smap (sorted-map :k3 102 :k1 100 :k2 101))
#'user/-smap
user> -smap
{:k1 100, :k2 101, :k3 102}
```

We'll use `sorted-map-by` to sort from the greatest value to the least value. Switch the `%1` and `%2` arguments around for the least to greatest:

```
user> (into
      (sorted-map-by
       #(compare (%2 -smap) (%1 -smap)))
      -smap)
{:k3 102, :k2 101, :k1 100}
```

Clojure has a very cool form that returns a map of a Java object's JavaBean properties:

```
user> (def calendar-map (bean (java.util.Calendar/getInstance)))
#'user/calendar-map
user> ;; The keywords are CaemelCase instead of lisp-case.
user> (:weekYear calendar-map)
2013
user> (:timeInMillis calendar-map)
1367469433849
```

You still have to call `bean` on the other Java objects returned from the map:

```
user> (clojure.pprint/pprint
      (bean (:timeZone calendar-map)))
{:rawOffset 32400000,
 :lastRuleInstance nil,
 :displayName "Japan Standard Time",
 :dirty false,
 :class sun.util.calendar.ZoneInfo,
 :ID "Asia/Tokyo",
 :DSTSavings 0}
```

There's another cool form that returns a map of how many times an element has occurred in a collection:

```
user> (frequencies [2 2 2 2 1 1 1 1 1 3])
{2 4, 1 5, 3 1}
user> (defn n4-occurred [& coll]
      (let [_ (frequencies coll)]
        (format "4 occurred %d times"
                 (get _ 4 0))))
#'user/n4-occurred
user> (n4-occurred 1 1 2 3 4 4 4 5)
"4 occurred 3 times"
user> (n4-occurred 1 1 2 3 5)
"4 occurred 0 times"
```

You can use `assoc` to add or override values of a map:

```
user> (def -map {:a 1 :b 2 :c 3})
#'user/-map
user> (assoc -map :a 9)
{:a 9, :c 3, :b 2}
user> (assoc -map :d 4 :e 5)
{:a 1, :c 3, :b 2, :d 4, :e 5}
user> (sort (assoc -map :d 4 :e 5))
[{:a 1} {:b 2} {:c 3} {:d 4} {:e 5}]
```

When dealing with nested values, use `assoc-in` to add or override the value of the position/key:

```
user> (def -map {:leaders [:joe-rogan :brian-redban]
                 :orgs [{:main "deathsquad-comedy"}]})
#'user/-map
user> ;; Add a value to :leaders and use clojure.pprint/pprint to
user> (clojure.pprint/pprint
      (let [position (count (:leaders -map))]
        (assoc-in -map [:leaders position] :bert-kreischer)))

{:leaders [:joe-rogan :brian-redban :bert-kreischer],
 :orgs [{:main "deathsquad-comedy"}]}
user> ;; Replace a value in :leaders based on position

user> (clojure.pprint/pprint
      (let [replace (-> (:leaders -map) count dec)]
        (assoc-in -map [:leaders replace] :joey-diaz)))

{:leaders [:joe-rogan :joey-diaz],
 :orgs [{:main "deathsquad-comedy"}]}
user> ;; Replace a nested position value
user> (clojure.pprint/pprint
      (assoc-in -map [:orgs 0 :main] "Icehouse Chronicles")))

{:leaders [:joe-rogan :brian-redban],
 :orgs [{:main "Icehouse Chronicles"}]}
```

When needing to remove a value, use the `dissoc` form:

```
user> (dissoc -map :leaders)
{:orgs [{:main "deathsquad-comedy"}]}
user> (dissoc -map :orgs)
{:leaders [:joe-rogan :brian-redban]}
```

You can use `find` if you don't need to set a default value because you can do with the `get` form. The `find` form also returns the map entry instead of the key's value:

```
user> (find -map :leaders)
[:leaders [:joe-rogan :brian-redban]]
user> (find -map :leaders123)
nil
```

To get a key or a value of a map element, use the `key` and `val` forms:

```
user> (def -map {:k1 1 :k2 2 :k3 3})
#'user/-map
user> [(key (first -map)) (val (first -map))]
[:k3 3]
user> (keys -map)
(:k3 :k1 :k2)
user> (vals -map)
(3 1 2)
```

To get the values of a nested map, use the `get-in` form:

```
(def -map {:book "The War of Art"
           :details {:author "S.Pressfield"
                     :rating 10
                     :notes {:other "Good book"}}})
#'user/-map
user> (get-in -map [:details :notes :other])
"Good book"
```

When accessing a collection value from within a nested map, the position of the array must be given:

```
user> (def -map
      {:users
       {:logged-in
        [{:username "abc" :email "abc"}
         {:username "123" :email "123"}
         {:username "qaz" :email "qaz"}]}})
#'user/-map
user> (get-in -map [:users :logged-in 2])
{:username "qaz", :email "qaz"}
```

To update a value with a form call, use `update-in`:

```
user> (clojure.pprint/pprint
      (update-in
       -map
       [:users :logged-in 2 :username]
       #(.. % toUpperCase)))
{:users
 {:logged-in
  [{:username "abc", :email "abc"}
   {:username "123", :email "123"}
   {:username "QAZ", :email "qaz"}]}}
```

Sometimes you'll only need a few values from a map. You can use `select-keys` to specify the needed ones:

```
user> (def -map {:k1 1 :k2 2 :k3 3})
#'user/-map
user> (select-keys -map [:k3 :k1])
{:k1 1, :k3 3}
```

The merge form will allow you to combine the maps:

```
user> (sort (merge {:a 1 :b 2} {:c 4 :d 5}))
([:a 1] [:b 2] [:c 4] [:d 5])
user> ;; Be careful not to override values.
user> (sort (merge {:a 1 :b 2} {:b 3 :c 4}))
([:a 1] [:b 3] [:c 4])
```

The merge-with form allows you to specify how the values of a map get merged:

```
user> (merge-with
      #(if (= %1 %2) %1 (str %1 "-" %2))
      {:fruit "apple"
       :type "red/green"
       :size "small food"}
      {:fruit "orange"
       :type "orange"
       :size "small food"})

{:fruit "apple-orange", :size "small food", :type "red/green-orange"}
user> (merge-with + {:abc 123 :def 456} {:xyz 999 :abc 123})

{:xyz 999, :abc 246, :def 456}
```

Sets

You might be wondering why you would use a set when you could use a map? You use a set when you need the key to be the value. This is easier and shorter than making a map that looks like `{:value1 :value1, :value :value}`. Also, a set's values will be unique since the key is the value.

Sets by example

There are three ways to make a hash-set:

```
user> (= (hash-set :a :b) #{:a :b})
true
```

One way is to pass a collection to the set form:

```
user> (def -vector [1 2 3])
#'user/-vector
user> (set -vector)
#{1 2 3}
user> (type (set -vector))
clojure.lang.PersistentHashSet
user> ;; Order isn't maintained
user> (println (set "abc") "=" (set "cba"))
#{a b c} = #{a b c}
nil
```

Another is by passing values to the sorted-set form:

```
user> ;; Duplicate keys are ignored
user> (sorted-set 3 2 1 3 2 1)
#{1 2 3}
user> ;; The sorted-set-by form takes a comparator
user> (sorted-set-by > 1 2 3)
#{3 2 1}
user> ;; Remember that duplicate keys are ignored
user> (conj #{1 2 3 4 5} 3)
#{1 2 3 4 5}
user> (conj #{1 2 3} 4)
#{1 2 3 4}
```

Similar to `dissoc` for maps, you can remove values with the `disj` form:

```
user> (def -set #{:a :b :c :d :e})
#'user/-set
user> (disj -set :c :d :e)
#{:a :b}
```

To test if a key exists, just place the keyword before the set. Use the `get` form if the key isn't a keyword type:

```
user> (:a -set)
:a
user> (get -set :a)
:a
user> (get -set :z)
nil
user> (get -set :z :not-found)
:not-found
```

Sequences

Clojure sequences implement the `ISeq` interface to make a thread-safe logical list. Sequences produced by functions in Clojure are usually lazy. This means that a value within the list isn't computed until the value is needed.

Learning to manipulate sequences will save you both time and many lines of code. Some sequence manipulation forms do the same thing and only differ by execution speed. Please be sure to pay extra attention when execution times are displayed in the examples.

Sequences by example

To test if an object implements the `ISeq` interface, use the `seq?` form:

```
user> (seq? [1])
false
user> (seq? (seq [1]))
true
user> (seq [])
nil
```

When `nil` isn't an acceptable value, use the `sequence` form to return an empty sequence instead of `nil` or `false` as shown in the following code:

```
user> (sequence [])
()
user> (seq? (sequence []))
true
```

One of the many ways to construct a sequence is to use the `repeat` form for an infinite seq:

```
user> (def forever-21 (repeat 21))
#'user/forever-21
user> (first forever-21)
21
user> (second forever-21)
21
user> (nth forever-21 3)
21
user> ;; Repeat takes a limiting number too
user> (clojure.pprint/pprint
      (repeat 3 "1 sequence = 3 times this"))
("1 sequence = 3 times this"
 "1 sequence = 3 times this"
 "1 sequence = 3 times this")
```

To generate a sequence of numbers, use the `range` form:

```
user> (range 0 8 2)
(0 2 4 6)
user> (range 0 8)
(0 1 2 3 4 5 6 7)
user> (range 8)
(0 1 2 3 4 5 6 7)
user> (take 8 (range))
(0 1 2 3 4 5 6 7)
```

When anything is needed repeatedly, use the `repeatedly` form to get an infinite seq. Unlike the `repeat` form, `repeatedly` will call the provided function for each value provided in the sequence:

```
user> (def -seq
      (repeatedly
       #(rand-nth [65 97 66 98])))
#'user/-seq
user> (take 10 -seq)
(98 65 98 97 65 98 65 97 65 97)
user> (apply
      (partial vector-of :char)
      (take 10 -seq))
[\b \A \b \a \A \b \A \a \A \a]
```


When needing an infinite sequence of performing a certain task on a single object, use `iterate` in the following way:

```
user> (def not-forever-21 (iterate inc 21))
#'user/not-forever-21
user> (take 10 not-forever-21)
(21 22 23 24 25 26 27 28 29 30)
```

Use `lazy-seq` to make an infinite sequence without running out of memory:

```
user> (defn two-steps
      [i]
      (lazy-seq
       (cons i (two-steps (+ 2 i)))))
#'user/two-steps
user> (take 10 (two-steps 0))
(0 2 4 6 8 10 12 14 16 18)
```

For infinite recursion `lazy-cat` can also be used:

```
user> (def fib
      (lazy-cat [0 1]
                 (map + (rest fib) fib)))
#'user/fib
user> (take 10 fib)
(0 1 1 2 3 5 8 13 21 34)
```

Using `cycle` is similar to `repeatedly`, but it repeats the contents of a sequence:

```
user> (take 16 (cycle (range 1 4)))
(1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1)
```

The `interleave` form is a good way to merge multiple collections while maintaining the element order:

```
user> (let [-type ['Old 'New 'Used]
          colors ['Red 'Blue 'Green]
          cars ['Lexus 'Toyota 'Nissan]
          details (partition 3 (interleave -type colors cars))]
      (doseq [car details]
        (println car)))
(Old Red Lexus)
(New Blue Toyota)
(Used Green Nissan)
```

Use `interpose` to place an element between all elements of a sequence:

```
user> (apply str (interpose "," ["csv" "format" "line"]))
"csv,format,line"
```

The `tree-seq` form's first argument is used to identify which part of a sequence is a tree node. The second argument is a function that returns a sequence of the branches' children:

```
user> (clojure.pprint/pprint
      (tree-seq
        vector?
        identity
        [:books
         [:reading [:one]
          :read [:many]]]))

([[:books [:reading [:one] :read [:many]]]
  :books
  [:reading [:one] :read [:many]]
  :reading
  [:one]
  :one
  :read
  [:many]
  :many)
nil
```

To handle an entire XML sequence, you'll have to use the `parse` form from the `clojure.xml` namespace:

```
user> (use '[clojure.xml :only [parse]])
nil
user> (-> "<head><title>Clojure</title></head>"
        .getBytes
        java.io.ByteArrayInputStream.
        parse
        clojure.pprint/pprint)
{:tag :head,
 :attrs nil,
 :content [{:tag :title, :attrs nil, :content ["Clojure"]}]}
nil
```

Note in the previous example that only one of the two XML tags was fully parsed. To parse all tags, use `xml-seq`:

```
user> (-> "<head><title>Clojure</title></head>"
      .getBytes
      java.io.ByteArrayInputStream.
      parse
      xml-seq
      clojure.pprint/pprint)

({:tag :head,
  :attrs nil,
  :content [{:tag :title, :attrs nil, :content ["Clojure"]}]}
 {:tag :title, :attrs nil, :content ["Clojure"]}
 "Clojure")
nil
```

The `java.util.StringTokenizer` objects can't have sequence forms applied to them. Use the `enumeration-seq` form to make this possible:

```
user> (def str-tokenizer
      (java.util.StringTokenizer. "abc def ghi"))
#'user/str-tokenizer
user> (first str-tokenizer)
IllegalArgumentException Don't know how to create ISeq
from: java.util.StringTokenizer
user> (first (enumeration-seq str-tokenizer))
"abc"
```

Some Java objects will need to be wrapped with the `iterator-seq` form to use sequence forms on them:

```
user> (def needs-iter
      (java.util.ArrayList. ["a" "b" "c" "d"]))
#'user/needs-iter
user> (conj needs-iter "e")
ClassCastException java.util.ArrayList cannot be cast to
clojure.lang.IPersistentCollection
user> (conj (iterator-seq
          (.iterator needs-iter)) "e")
("e" "a" "b" "c" "d")
```

The `file-seq` form can be used to create a recursive sequence of a path and its files:

```
user> ;; mkdir -p '/tmp/clojure/example' '/tmp/clojure/example2'
user> (doseq [file (file-seq
```

```

(clojure.java.io/file "/tmp/clojure/"))
:let [parent (.getParent file)
      name (.getName file)
      display (str parent "/" name)]]
(println display))

/tmp/clojure
/tmp/clojure/example2
/tmp/clojure/example

```

There are many different ways of accessing a seq form's value by position:

```

user> (first [1 2 3])
1
user> (second [1 2 3])
2
user> (last [1 2 3])
3
user> (rest [1 2 3])
(2 3)
user> (next [1 2 3])
(2 3)
user> (next (rest [1 2 3]))
(3)

```

When manipulating a sequence, calling multiple position-related functions isn't uncommon. There is a set of functions applying common patterns that usually take two functions to perform the same task. The following is an example of these functions and the patterns they're supposed to replace:

```

user> (ffirst [[1 2 3] 4 5])
1
user> (first (first [[1 2 3] 4 5]))
1
user> (nfirst [[1 2 3] 4 5])
(2 3)
user> (next (first [[1 2 3] 4 5]))
(2 3)
user> (fnext [[1 2 3] 4 5])
4
user> (first (next [[1 2 3] 4 5]))
4
user> (nnext [1 2 3 4 5])
(3 4 5)
user> (next (next [1 2 3 4 5]))
(3 4 5)

```

Use the `nth` form to give default values when a position within a sequence doesn't exist:

```
user> (let [nums [5 4 3 2 1]
           missing-msg (str "Max size is " (count nums))]
      (nth nums 6 missing-msg))

"Max size is 5"
```

`nthnext` is good for skipping values before returning the rest of a sequence:

```
user> (nthnext [5 4 3 2 1] 2)
(3 2 1)
user> (nthnext "Four Five" 5)
(\F \i \v \e)
```

`rand-nth` will return a random value from a sequence:

```
user> (rand-nth [:apple :orange :cherry])
:orange
user> (rand-nth "abc")
\b
user> (rand-nth "abc")
\c
```

Sequence functions seem to do the same thing but some are faster than others:

```
user> (time (butlast [:a :b :c]))
"Elapsed time: 0.055317 msecs"
(:a :b)
user> (time (drop-last [:a :b :c]))
"Elapsed time: 0.086947 msecs"
(:a :b)
```

There are many ways to take values from a sequence:

```
user> (take 10
      (map char (iterate inc 65)))
(\A \B \C \D \E \F \G \H \I \J)
user> (take-last 5 (range 1 11))
(6 7 8 9 10)
user> (take-nth 10 (range 110))
(0 10 20 30 40 50 60 70 80 90 100)
```

You can use `take-while` to test if values should continue to be placed in a new sequence. If the test fails, all values after the failing test will be ignored. All passing values will be returned in a new sequence:

```
user> (take-while string? ["a" "b" 1 "c"])
("a" "b")
```

Dropping values is very similar to taking values:

```
user> (drop 15 (range 17))
(15 16)
user> (drop 5 (range 11))
(5 6 7 8 9 10)
user> (drop-last (range 1 5))
(1 2 3)
user> (drop-while integer? [1 2 3 "string" 4 5 6])
("string" 4 5 6)
```

The `keep` form is similar to the `map` form. The difference is that the value returned has to be true or false but not nil. Nil doesn't show up in the returned sequence:

```
user> (keep #(when (pos? %1) %1) [1 2 -3 -4 5 6])
(1 2 5 6)
user> (map #(when (pos? %1) %1) [1 2 -3 -4 5 6])
(1 2 nil nil 5 6)
```

When dealing with a sequence that's position-sensitive (array, vec, and so on), use `keep-indexed` to decide which part of the sequence to keep. The following example drops the first two elements:

```
user> (keep-indexed #(when (>= %1 2) %2) [:a 1 :b 2 :c 3])
(:b 2 :c 3)
```

`concat` is similar to `merge`, but `concat` maintains the order of the sequence from left to right and elements get merged if it's one level deep.

```
user> (concat [:a :b] [:c :d] [:e :f] [:g [:h :i]])
(:a :b :c :d :e :f :g [:h :i])
user> (merge [:a :b] [:c :d] [:e :f] [:g [:h :i]])
[:a :b [:c :d] [:e :f] [:g [:h :i]]]
```

If you're not using a set collection, you can use the `distinct` form to assure unique values:

```
user> (take 10 (cycle [1 2 3]))
(1 2 3 1 2 3 1 2 3 1)
user> (distinct (take 10 (cycle [1 2 3])))
(1 2 3)
```

Grouping values of a sequence is as easy as passing a value test function and a collection as an argument to the `group-by` form:

```
user> (group-by string? ["a" "b" :k 1])
{true ["a" "b"], false [:k 1]}
user> (type (group-by string? ["a" 1]))
```

```
clojure.lang.PersistentArrayMap
user> (get (group-by string? ["a" 1]) true)
["a"]
```

You can also group by the value of a key:

```
user> (def pay-levels
      [{:pay-level 0.25 :name "pawn"}
       {:pay-level 1 :name "wizard"}
       {:pay-level 1 :name "monk"}
       {:pay-level 10 :name "ruler"}])
#'user/pay-levels
user> ;; Notice the group with the pay-level of 1
user> (clojure.pprint/pprint
      (group-by :pay-level pay-levels))
{0.25 [{:name "pawn", :pay-level 0.25}],
 1 [{:name "wizard", :pay-level 1} {:name "monk", :pay-level 1}],
 10 [{:name "ruler", :pay-level 10}]}
```

You can also group a sequence by position. The partition form needs a grouping number and a collection to group:

```
user> (partition 3 (range 6))
((0 1 2) (3 4 5))
```

Supplying a step number tells partition where to start a group after creating one:

```
user> (partition 3 1 (range 6))
((0 1 2) (1 2 3) (2 3 4) (3 4 5))
```

Because partition was only putting out sections of three, another collection is provided to take the place of the missing values:

```
user> (partition 3 1 ["RANOUT" "PAD" "ING"] (range 6))
((0 1 2) (1 2 3) (2 3 4) (3 4 5) (4 5 "RANOUT"))
user> (partition 3 (range 13))
((0 1 2) (3 4 5) (6 7 8) (9 10 11))
```

If defining missing values isn't what you're in for, you can use the partition-all form:

```
user> (partition-all 3 1 (range 6))
((0 1 2) (1 2 3) (2 3 4) (3 4 5) (4 5) (5))
```

Using partition-by allows you to group a collection based on the outcome of a test function:

```
user> (partition-by keyword? [:a :b :c 1 2 3])
((:a :b :c) (1 2 3))
```

Similar to `partition-by`, `split-at` makes a single partition after a specified number of elements:

```
user> (split-at 3 [:a :b :c :d :e :f])
[[:a :b :c] (:d :e :f)]
user> (split-at 3 (range 10))
[(0 1 2) (3 4 5 6 7 8 9)]
user> ;; Similar results can be made with split-with
user> (split-with (partial >= 2) (range 10))
[(0 1 2) (3 4 5 6 7 8 9)]
```

```
user> (split-with #(%1 #{:a :b :c}) [:a :b :c :d :e :f])
[[:a :b :c] (:d :e :f)]
```

The `filter` form filters a sequence returning a sequence of all the elements that passed the filter test:

```
user> (filter #(> %1 5) (range 11))
(6 7 8 9 10)

user> (filter #{:e :f} [:a :b :c :d :e :f])
(:e :f)
```

To filter out anything that does pass the test, use the `remove` form instead of `filter`. Note in the following example how there's nothing over 5:

```
user> (remove #(> %1 5) (range -5 10))
(-5 -4 -3 -2 -1 0 1 2 3 4 5)
```

`Replace` can be used to replace other elements of a sequence/collection:

```
user> (replace {"apple" "orange"} ["apple"])
["orange"]
```

For example, let's say we wanted to convert the hex values `a`, `b`, `c`, `d`, `e`, and `f`:

```
user> (def lton
      (hash-map
       :a 10 :b 11 :c 12 :d 13 :e 14 :f 15))
#'user/lton
user> (defn l-to-n [& ls] (replace lton ls))
#'user/l-to-n
user> (l-to-n :a :b :c)
(10 11 12)
```


Use `shuffle` to get a different order of the same sequence:

```
user> (shuffle (range 5))
[0 3 4 1 2]
user> (shuffle (range 5))
[4 2 0 3 1]
```

Use the `for` form when needing the return values of a sequence in a sequence:

```
user> (for [x (range 5)] (inc x))
(1 2 3 4 5)
```

Use the `doseq` form when needing to make side effects. Note that the return value is `Nil`:

```
user> (doseq [x (range 5)] (println x))
0
1
2
3
4
Nil
```

The `map` form is a good way to call a single function on one or more collections and return a sequence of the results:

```
user> (map + [1 2] [1 2] [1 2])
(3 6)
user> (map + [1 2] [1 2])
(2 4)
user> (map + [1 2])
(1 2)
user> (map #(+ 5 %1) [1 2])
(6 7)

user> (map #(if (> %1 5) %1 0) (range 10))
(0 0 0 0 0 6 7 8 9)
```

Similar to `keep-indexed`, we're going by sequence element position and not value:

```
user> (map-indexed
      #(when (= %1 2) %2)
      (hash-map :snake 10 :dog 2 :wolf 20))
(nil nil [:wolf 20])
```

The `mapcat` form is a combination of `map` and `concat`. The `map` form is applied first and then the result of `map` is merged with `concat`.

```
user> (mapcat drop-last [[1 2 3] [2 3 4]])
(1 2 2 3)
user> (mapcat rest [[1 2 3] [2 3 4]])
(2 3 3 4)
```

The `reductions` form returns a sequence of all the values produced during a `reduce` call:

```
user> (reductions + 3 (repeat 3 3))
(3 6 9 12)
user> ;; Notice how the value here is the last value
user> ;; in the previous example
user> (reduce + 3 (repeat 3 3))
12
```

The `max-key` form gets the maximum value by sorting the return value of the first argument:

```
user> (apply max-key val (hash-map :a 1 :b 2 :c 3))
[:c 3]
user> ;; min-key does the same to look for the smallest
user> (apply min-key val (hash-map :a 1 :b 2 :c 3))
[:a 1]
user> (max-key count [1 2] [1 2 3] (range 5))
(0 1 2 3 4)
user> (min-key count [1 2] [1 2 3] (range 5))
[1 2]
```

For some reason, you might need to evaluate all the elements of a `lazy-seq` form. The `doall` form can help you.

```
user> (let [xyz (map println (range 3))])
nil

user> (let [xyz (doall (map println (range 3)))]))
0
1
2
nil
```

If you just want the side effects of computing a lazy value, you can use the `dorun` form:

```
user> (dorun 0 (repeatedly
               #(println
                  (java.util.Calendar/getInstance))))
#inst "2013-05-03T15:35:51.625+09:00"
nil
user> ;; Notice the numbers on the far right
user> (dorun 5 (repeatedly
               #(println
                  (java.util.Calendar/getInstance))))
#inst "2013-05-03T15:35:58.617+09:00"
#inst "2013-05-03T15:35:58.618+09:00"
#inst "2013-05-03T15:35:58.619+09:00"
#inst "2013-05-03T15:35:58.620+09:00"
#inst "2013-05-03T15:35:58.621+09:00"
#inst "2013-05-03T15:35:58.621+09:00"
nil
user> (dorun
      (take 5
         (repeatedly
          #(println
             (java.util.Calendar/getInstance))))))
#inst "2013-05-03T15:37:21.092+09:00"
#inst "2013-05-03T15:37:21.092+09:00"
#inst "2013-05-03T15:37:21.093+09:00"
#inst "2013-05-03T15:37:21.093+09:00"
#inst "2013-05-03T15:37:21.094+09:00"
```

:let, :while, and :when

In the previous section, you learned how to access the values of a sequence, but there's a little more to it. There'll be situations where you wish there was an easier way to apply restraints, filters, and bindings when dealing with a sequence. Luckily for us, Clojure has got us covered. The `doseq` and `for` forms also accept bindings with `:let`, filters with `:when`, and restraints with `:when`.

```
user> (for [i (range 10)
           :when (even? i)
           :while (>= 6 i)
           :let [i2 (+ i 2)]]
      i2)
(2 4 6 8)
user> (doseq [i (range 10)]
```

```
      :when (even? i)
      :while (>= 6 i)
      :let [i2 (+ i 2)]]
    (println i i2))
0 2
2 4
4 6
6 8
```

Summary

You can't go without collections and sequences in Clojure. Each collection type has its own benefits and limitations, but all collections are a group of elements. Be careful when using the same form on many different collection types, because some forms act differently depending on the collection type.

Most sequences in Clojure are lazy but don't forget to wrap a recursive function's body with `lazy-seq` if it doesn't have a termination point. Using the `doall` and `dorun` forms should be with caution when dealing with infinitely recursive results. If you find yourself wrapping the `cat` form with the `lazy-seq` form, remember to use the `lazy-cat` form.

If you ever find yourself binding values before placing them in the `for` or `doseq` form, check to see if adding `:let` bindings to the body will help with your situation. The same can be said about the `:when` and `:while` forms. Where possible, prefer this when working with entire sequences.

In the next chapter, you'll learn about data types that help enable Clojure's concurrency model.

6

Assignment and Concurrency

This chapter starts off by getting more hands-on with variables and how to manipulate them. The *Variables* 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.

Once you're comfortable with moving on, the following sections will focus on some of the Clojure concurrency-related data types. Not using a REPL session to try some of the provided examples can possibly hinder your understanding of the examples presented. Trying some of the examples in a REPL session will better help you understand this chapter.

Variables

Clojure variables differ a lot in comparison to variables in other languages. For example, to change the value of a variable, you have to explicitly state that the variable can be changed when the variable is defined. Variables can also be seen by all threads in a multithreaded program; however, when a variable is bound with a new value, the new binding can only be seen on the thread that changed the value.

The most common global variable definition is made with the `def` form, as shown in the following code snippet:

```
user> (def def1 123)
#'user/def1
```

Documentation strings are placed after the symbol, as shown in the following code snippet:

```
user> (def def1 "one two three" 123)
#'user/def1
user> (doc def1)
-----
```

```
user> def1
one two three
nil
user>
```

Define a truly dynamic variable by placing the dynamic meta tag in front of the symbol. Also, notice how the definition has earmuffs or stars on both sides of the symbol to indicate that it's dynamic, as shown in the following code snippet:

```
user> (def ^:dynamic *def2* 123)

#'user/*def2*
user> (str *def2*
      (binding [*def2* 456] *def2*))
"123456"
```

If you don't explicitly state that a variable is dynamic and the symbol has earmuffs, Clojure will ignore the earmuffs and treat it as a nondynamic variable, as shown in the following code snippet:

```
user> (def *def2* 123)
Warning: *def2* not declared dynamic and thus is not dynamically
rebindable, but its name suggests otherwise. Please either indicate
^:dynamic *def2* or change the name. (NO_SOURCE_PATH:1)
#'user/*def2*
```

The binding form will only work on dynamic variables, as shown in the following code snippet:

```
user> (def def1 123)
#'user/def1
user> (binding [def1 456] def1)
IllegalStateException Can't dynamically bind non-dynamic var: user/
def1  clojure.lang.Var.pushThreadBindings (Var.java:353)
```

The defonce form will define a variable if it doesn't already exist, and vice versa:

```
user> (def abc 123)
#'user/abc
user> abc
123
user> (defonce abc 456)
nil
user> abc
123
```

There'll be some cases where you need a symbol that just doesn't exist yet. Use the `declare` form if you need a valid symbol that will be set in the future, as shown in the following code snippet:

```
user> (declare abc2)
#'user/abc2
user> abc2
#<Unbound Unbound: #'user/abc2>
user> (declare ^:dynamic *green*)
#'user/*green*
```

When working within the `bind` form, you might have to change the value of a variable within the code body. It's not recommended, but using the `set!` form on these bound variables will allow you to override the set binding, as shown in the following code snippet:

```
user> (declare ^:dynamic *green*)
#'user/*green*
user> (let [print-green #(println "Green =>" *green*)]
      (print-green)
      (binding [*green* "abc"]
        (print-green)
        (set! *green* 123)
        (print-green))
      (print-green))

Green => #<Unbound Unbound: #'user/*green*>
Green => abc
Green => 123
Green => #<Unbound Unbound: #'user/*green*>
nil
```

Global dynamic variables can also have their binding set with the `set!` form, as shown in the following code snippet:

```
user> (let [current *ns*]
      (set! *ns*
            (or (find-ns 'example)
                (create-ns 'example)))
      (println "Hi from" *ns*)
      (set! *ns* current))
Hi from #<Namespace example>
#<Namespace user>
```


The `alter-var-root` form changes the root value of a variable by providing a function to handle its current value. Whenever the variable is accessed, the output of the function provided to handle the original value will always be called:

```
user> (defn match? [n n2] (= n n2))
#'user/match?
user> (match? 2 2)
true
user> (match? 2 4)
false
user> (alter-var-root
      (var match?)
      (fn [parent-var]
        (fn [n n2]
          (println
            (format "(= %s %s) = %s" n n2 (parent-var n n2)))))))

user> ;; the value returned was removed for brevity
user> (match? 2 2)
(= 2 2) = true
nil
user> (match? 2 4)
(= 2 4) = false
nil
```

There may be a situation where you need to dynamically generate some variable bindings. When this is the case, use the `with-bindings` form to apply these bindings to the body form, as shown in the following code snippet:

```
user> (declare ^:dynamic *green*)
#'user/*green*
user> (def -bindings (hash-map #'*green* "blue"))
#'user/-bindings
user> (with-bindings -bindings (println *green*))
blue
nil
user> (with-bindings -bindings
      (println *green*)
      (with-bindings
        (assoc -bindings #'*green* "orange")
        (println *green*)))

blue
orange
nil
```

The `with-bindings*` form is the same as the `with-bindings` form with a few exceptions. The third argument is a function, and the rest are arguments to be passed to the function, as shown in the following code snippet:

```
user> (with-bindings* {'*green* "red"}
      #(println "*green* is" *green* %)
      "!!!!!!!!!!")

*green* is red !!!!!!!!!!
nil
```

Manipulating the `loop` form bindings might be difficult to understand for the imperative programmer. The `with-local-vars` form acts as an imperative alternative to the `loop` form, as shown in the following code snippet:

```
user> (with-local-vars [colors [:red :blue]
                      return []]
      (while (seq @colors)
        (var-set return
          (conj @return
            (last @colors))))
      (var-set colors
        (drop-last @colors)))
      @return)
[:blue :red]
```

Local variables in a `let` form can be bound to an anonymous function, but sometimes you might want a known function locally. You can use the `letfn` form to achieve this. This is great for making recursive functions, as shown in the following code snippet:

```
user> (letfn [(my-self [n]
              (when (<= n 3)
                (println "n = " n)
                (my-self (inc n)))))
      (my-self 0))

n = 0
n = 1
n = 2
n = 3
nil
```

There'll be situations when you might need a random symbol for your variables to prevent clobbering an existing variable name. Using the `gensym` form, you can set a prefix for the random symbol name, as shown in the following code snippet:

```
user> (gensym "unique-name-number")
unique-name-number24989
user> (gensym "unique-name-number")
unique-name-number24992
user> (gensym "no-collision-name")
no-collision-name24997
```

If no prefix is provided, the generated symbol is prefixed with `G__`, as shown in the following code snippet:

```
user> (gensym)
G__25359
user> (gensym)
G__25362
```

A symbol of a variable is just a symbol until you use the `var` form on it, as shown in the following code snippet:

```
user> (var *green*)
#'user/*green*
```

Prefixing a variable symbol with `#'` is a short-hand version of using `var`, as shown in the following code snippet:

```
user> (= (var *green*) #'*green*)
true
user> #'*green*
#'user/*green*
```

If a variable doesn't exist when you try to access it, you'll get an exception error. Use the `find-var` form to help prevent these errors, as shown in the following code snippet:

```
user> (find-var "user/orange")
nil
user> (find-var "user/*green*")
#'user/*green*
```

Another way of checking if a variable exists is by using the `var?` form for a Boolean test value, as shown in the following code snippet:

```
user> (var? "string")
false
```

```
user> (def abc 123)
#'user/abc
user> (var? abc)
false
user> (var? #'abc)
true
```

Transients

A **Transient** is the only mutable data structure in the Clojure language and it can be used for collections, maps, and sets. Transients behave the same way as other Clojure data structures after they have been made persistent, but Transients are considered mutable until they are explicitly made persistent.

Just as with any other Clojure variable, you'll define a variable with the `def` form but the value is wrapped with the `transient` form, as shown in the following code snippet:

```
user> (def t-map (transient {}))
#'user/t-map
```

Things get interesting here, because unlike other collection types, we have to use a special set of functions to alter the collection. You'll notice that the functions called to alter the collection end with exclamation marks. These marks indicate that a change is being made and there will be side effects, as shown in the following code snippet:

```
user> (assoc! t-map :a 1 :b 2)
#<TransientArrayMap clojure.lang.PersistentArrayMap$TransientArrayMap
@3213c05b>

user> (dissoc! t-map :b)
#<TransientArrayMap clojure.lang.PersistentArrayMap$TransientArrayMap
@3213c05b>
```

The `persistent!` form is required to ensure that no other changes can be made to the object. After you call `persistent!` on the object, the collection object will behave like other Clojure collections, as shown in the following code snippet:

```
user> (persistent! t-map)
{:a 1}
user> t-map
#<TransientArrayMap clojure.lang.PersistentArrayMap$TransientArrayMap
@3213c05b>
```

One thing you might notice about some of the functions used to modify Transients is that they differ somewhat from their persistent counterparts. For example, in the following example, we have to use the `doseq` form to add multiple values to the Transient collection, because the `conj!` form can only take a single value, whereas the `conj` form can take multiple values:

```
user> (def t-coll (transient []))
#'user/t-coll
user> (doseq [x [:a :b]] (conj! t-coll x))
nil
user> (count t-coll)
2
user> (pop! t-coll)
#<TransientVector clojure.lang.PersistentVector$TransientVector@3d5a748>
user> (persistent! t-coll)
[:a]
```

It's also important to remember that, once a mutable object has been converted to a persistent one, you can't call the `persistent!` form on the same object without raising an exception error, as shown in the following code snippet:

```
user> (def t-set (transient #{:a :b :c}))
#'user/t-set
user> (disj! t-set :c)
#<TransientHashSet clojure.lang.PersistentHashSet$TransientHashSet@1f3ee5>
user> (persistent! t-set)
#{:a :b}>
user> (persistent! t-set)
IllegalAccessError Transient used after persistent! call clojure.lang.PersistentHashMap$TransientHashMap.ensureEditable (PersistentHashMap.java:321)
```

Atoms

Clojure allows for controlled mutations when using an **Atom**. An Atom provides a safe way to access and possibly replace a value bound to it. Since changes to an Atom are synchronous, if any error occurs while altering the Atom, an exception will be thrown in the current thread trying to make the changes. In the next section about **Agents**, you'll see why this might matter.

To create an Atom, pass a value to the `atom` form, as shown in the following code snippet:

```
user> (def -a (atom []))
#'user/-a
```

The preceding code doesn't provide the value of the Atom; it provides the object that is the Atom:

```
user> -a
#<Atom@24d4e62c: []>
```

This is the value of the Atom and it's being retrieved using the reader macro `@`. This is shorthand for the `deref` form, as shown in the following code snippet:

```
user> @-a
[]
user> (deref -a)
[]
```

When needing to access the value of an Atom, you'll have to use `@` or the `deref` form, as shown in the following code snippet:

```
user> (conj @-a 1 2)
[1 2]
user> @-a
[]
```

Using the `swap!` form is a way to change the value of an Atom by using the Atom's current value and passing it as the first argument to the provided function, as shown in the following code snippet:

```
user> (swap! -a conj 1 2)
[1 2]
user> @-a
[1 2]
user> (swap! -a #(cons 0 %))
(0 1 2)
user> @-a
(0 1 2)
```

You can remove the value of an Atom and set a new one with the `reset!` form, as shown in the following code snippet:

```
user> (reset! -a "apples")
"apples"
user> @-a
"apples"
```

There may be a situation when you want to change the value if the current value matches a test value. You can use the `compare-and-set!` form to achieve this, as shown in the following code snippet:

```
user> (compare-and-set! -a "oranges" "grapes")
false
user> @-a
"apples"
```

Notice how nothing happened in the previous example. This time the test value is set to "apples", as shown in the following code snippet:

```
user> (compare-and-set! -a "apples" "grapes")
true
user> @-a
"grapes"
```

Agents

Agents are similar to Atoms, except for the fact that changes are made asynchronously. Unlike an Atom, when attempted changes to an Agent fail, you won't know until you check. This is because changes to an Agent happen in a different thread, so the thread trying to make the change won't be aware of anything going on in the Agent.

You create an Agent the same way you would create an Atom, except that, in this case, you'll use the `agent` form, as shown in the following code snippet:

```
user> (def -ag (agent []))
#'user/-ag
```

We need to create a function that'll show us the current state of the `-ag` agent, as shown in the following code snippet:

```
user> (defn show-agent []
      (dotimes [_ 6]
        (Thread/sleep 1000)
        (println "Slept for" _ "second(s).\n"
                  "-ag = " @-ag)))
#'user/show-agent
```

In the following example, a thread blocking action is sent to `-ag` using the `send-off` form and the `show-agent` function that we created in the previous example. This function will tell us what the current value of `-ag` is over a five-second period. Since the new value of `-ag` is being computed in another thread, the old value is returned until the new value is finally available. The old value is then replaced with the new value once the new value is available:

```
user> (do (send-off
          -ag
          #(do (Thread/sleep 5000)
                (conj % 1 2)))
        (show-agent))
Slept for 0 second(s).
-ag = []
Slept for 1 second(s).
-ag = []
Slept for 2 second(s).
-ag = []
Slept for 3 second(s).
-ag = []
Slept for 4 second(s).
-ag = [1 2]
Slept for 5 second(s).
-ag = [1 2]
nil
```

Notice the elapsed time in the following example. The Agent is returned directly after the `send-off` form, as shown in the following code snippet:

```
user> (time
      (send-off
        -ag
        #(do (Thread/sleep 5000)
              (conj % 1 2))))
"Elapsed time: 0.247131 msecs"
#<Agent@7ed8f082: [1 2]>
user> @-ag
[1 2]
```

After waiting for five seconds, the following is the result of the same Agent:

```
user> @-ag
[1 2 1 2]
```


When needing to wait for the Agent to finish the actions sent to it, you can use the `await` form, as shown in the following code snippet:

```
user> (time
      (do (send-off
            -ag
            #(do (Thread/sleep 5000)
                  (conj % 1 2)))
          (await -ag)))
"Elapsed time: 5001.544295 msecs"
nil
user> @-ag
[1 2 1 2 1 2]
```

When a timeout limit is needed, use the `await-for` form. The return value is `false` if the Agent action wasn't completed before the timeout limit, as shown in the following code snippet:

```
user> (time
      (do (def -ag (agent []))
          (send-off
            -ag
            #(do (Thread/sleep 5000)
                  (conj % 1 2)))
          (await-for 2000 -ag)))
"Elapsed time: 2001.554332 msecs"
false
```

If finished before the timeout limit, the return value is `true`, as shown in the following code snippet:

```
user> (time
      (do (def -ag (agent []))
          (send-off
            -ag
            #(do (Thread/sleep 5000)
                  (conj % 1 2)))
          (await-for 6000 -ag)))
"Elapsed time: 5001.522776 msecs"
true
```

By default, an Agent's value won't change during an error. For example, when trying to do `(+ 1 ["abc"])`, as shown in the following code snippet:

```
user> (def -ag (agent ["abc"]))
#'user/-ag
```

```
user> (send -ag + 1)
#<Agent@74f6c562: ["abc"]>
user> @-ag
["abc"]
```

Let's use the `agent-error` form to test whether an error occurred during the send, as shown in the following code snippet:

```
user> (let [error (agent-error -ag)]
      (if-not error
        @-ag
        (println error)))
#<ClassCastException java.lang.ClassCastException:
clojure.lang.PersistentVector cannot be cast to java.lang.Number>
nil
```

Let's give this Agent a new value, as shown in the following code snippet:

```
user> (restart-agent -ag 2)
2
user> @-ag
2
```

Let's try `(+ 1 agent)` one more time, as shown in the following code snippet:

```
user> (send -ag inc)
3
user> @-ag
3
```

If you want to reject further Agent sendings, use the `shutdown-agents` form to complete pending sends and prevent the occurrence of further sends, as shown in the following code snippet:

```
user> (shutdown-agents)
nil
```

The value doesn't change after calling `shutdown-agents`, as shown in the following code snippet:

```
user> (send -ag inc)
3
user> @-ag
3
```

After restarting your REPL session, you can define Agents again:

```
user> (def -ag (agent 4))
#'user/-ag
```

When sending actions to an Agent, the functions inside the Agent send can access the current Agent with the `*agent*` symbol, as shown in the following code snippet:

```
user> (defn looping [n i]
      (if (<= n 99999)
          (send-off *agent* looping (inc i)))
      i)
#'user/looping
user> (do (send-off -ag looping 0)
      (println "-ag is " @-ag)
      (Thread/sleep 500)
      (println "-ag is " @-ag)
      (Thread/sleep 1000)
      (println "-ag is " @-ag))
-ag is 4
-ag is 12492
-ag is 33728
nil
user> @-ag
100001
```

When an error occurs inside an Agent action, you wouldn't know until you tested for an error. Using `:error-handler` can give an Agent a method to handle the error immediately, as shown in the following code snippet:

```
user> (defn handler [a e]
      (println
        (str (java.util.Date.) " >> \n" e))
      #'user/handler)
user> (def -ag (agent 4 :error-handler handler))
#'user/-ag
user> (send -ag + "abc")
#<Agent@401898e5: 4>
Fri May 10 11:24:18 JST 2013 >>
#<ClassCastException java.lang.ClassCastException:
java.lang.String cannot be cast to java.lang.Number>
```

The `error-handler` form will allow you to get and call `fn` bound to `:error-handler`, as shown in the following code snippet:

```
user> ((error-handler -ag) "a" "e")
Fri May 10 11:25:23 JST 2013 >>
e
nil
```

Use the `set-error-handler!` form to add or change the error handler for an Agent, as shown in the following code snippet:

```
user> (set-error-handler!
      -ag
      (fn [a e] (println "!!!!" e)))
nil
user> (send -ag + "abc")
#<Agent@401898e5: 4>
!!!! #<ClassCastException java.lang.ClassCastException:
java.lang.String cannot be cast to java.lang.Number>
```

By default, the Agent maintains its value when an error occurs, but `:continue` is the name of this error handling mode, as shown in the following code snippet:

```
user> (def -ag (agent 4 :error-mode :continue))
#'user/-ag
user> (send -ag + "abc")
#<Agent@2c43a738: 4>
```

Use `set-error-mode!` to set the mode to either `:fail` or `:continue`, as shown in the following code snippet:

```
user> (set-error-mode! -ag :fail)
nil
user> (send -ag + "abc")
#<Agent@2c43a738 FAILED: 4>
```

Refs

Refs are similar to **Atoms** because they're both used for making synchronous changes. Unlike **Atoms**, you can read and modify multiple **Refs** simultaneously using the `dosync` form to perform a transaction.

Using the `ref` form, we can make a **Ref**, as shown in the following code snippet:

```
user> (def -r (ref []))
#'user/-r
```

Just as with **Atoms** and **Agents**, you can use `deref` or the `@` reader macro to get the value of a **Ref**, as shown in the following code snippet:

```
user> ;; This is not the value of the ref
user> -r
#<Ref@356f95be: []>
user> ;; This is the value of the ref
user> @-r
[]
```

Just as with **Atoms** and **Agents**, the `:validator` option is also available. In the following example, we'll add a `:validator` option so that the changes to the **Ref** will fail if the value is empty. In the following example, you'll notice that the transactions are made with the `dosync` form:

```
user> (def -r (ref [1] :validator not-empty))
#'user/-r
user> ;; Create a transaction to change the ref
user> (dosync (ref-set -r [2]))
[2]
user> @-r
[2]
```

Make sure our `not-empty` validator is working, as shown in the following code snippet:

```
user> (dosync (ref-set -r []))
IllegalStateException Invalid reference state
clojure.lang.ARef.validate (ARef.java:33)
```

Use `alter` when the order of the operands matters, as shown in the following code snippet:

```
user> (dosync (alter -r conj 4 5))
[3 4 5]
user> @-r
[3 4 5]
```

When the operands are commutative (order doesn't matter), you should use `commute` for performance, as shown in the following code snippet:

```
user> (dosync (commute -r #(apply + %)))
12
```

Using the `ensure` form on a `Ref` at the beginning of a transaction can prevent other transactions from modifying the value of the `Ref` in the current transaction, as shown in the following code snippet:

```
user> (def -r2 (ref 1))
#'user/-r2
user> (dosync (ensure -r) (commute -r2 + @-r))
13
user> @-r2
13
```

Some functions should never be called in a transaction. Use the `io!` form to assert this, as shown in the following code snippet:

```
user> (defn never-run-in-transaction []
      (io! (println "Safety")))
#'user/never-run-in-transaction
user> (dosync (never-run-in-transaction)
      (ref-set -r 10))
IllegalStateException I/O in transaction
user/never-run-in-transaction (NO_SOURCE_FILE:2)
```

Notice how `never-run-in-transaction` prevented the `ref-set` value from changing to `-r`:

```
user> @-r
18
```

Futures

Futures are similar to **Agents** because the body of the form is also executed asynchronously. A **Future** will cache the return result of the body and will return the cached value without re-evaluating the body after the initial evaluation is complete. This is one of the key differences between a **Future** and an **Agent**.

A **Future** will run the contents of the body in another thread, as shown in the following code snippet:

```
user> (def f (future
      (Thread/sleep 3000)
      (println "start work")))
```

```
(Thread/sleep 3000)
(println "end work")
"value"))
#'user/f
```

After the Future is complete, and/or when the Future is accessed for the first time, the returned value is cached and the body isn't executed again, as shown in the following code snippet:

```
user> @f
start work
end work

"value"
```

There's no output this time, as shown in the following code snippet:

```
user> @f
"value"
```

To check if a Future is complete without blocking the current thread, you can use the `future-done?` form, as shown in the following code snippet:

```
user> (def f (future
              (Thread/sleep 10000)
              "value"))
#'user/f
user> (future-done? f)
false
user> (future-done? f)
true
```

When you wish to stop the Future prematurely, use the `future-cancel` form, as shown in the following code snippet:

```
user> (def f (future
              (Thread/sleep 10000)
              "value"))
#'user/f
user> (future-cancel f)
true
```

The return value of `false` for an already cancelled Future is as follows:

```
user> (future-cancel f)
false
```

The `future-cancelled?` form can tell you if the Future has been cancelled without throwing an error, as shown in the following code snippet:

```
user> (future-cancelled? f)
true
```

If it is cancelled and you try to access the Future, you'll get an error, as follows:

```
user> @f
CancellationException
java.util.concurrent.FutureTask$Sync.innerGet
```

Trying to access a Future that's not a Future will cause an error, so use the `future?` form to be sure, as shown in the following code snippet:

```
user> (future? f)
true
user> (def not-future [])
#'user/not-future
user> @not-future
ClassCastException clojure.lang.PersistentVector
cannot be cast to java.util.concurrent.Future
```

Promises

A **Promise** acts similarly to a Future, but a Promise in Clojure requires the value to be delivered with the `deliver` form. In the following example, the Promise receives its value from another thread:

```
user> (time
      (let [p (promise)
            t (Thread.
                (fn []
                  (Thread/sleep 5000)
                  (deliver p 1)))]
          (println "Starting deliver thread.")
          (.start t)
          (println "Hanging until p is delivered")
          (println "The value of p is " @p)))

Starting deliver thread.
Hanging until p is delivered
The value of p is 1
"Elapsed time: 5002.641549 msecs"
nil
```


Summary

Variables can be both immutable and dynamic, because a dynamic variable's value can change although the previous value of the variable hasn't been changed and is still accessible. When working with truly dynamic variables, the value of the variable can be dictated by the `binding` form. The root value of the dynamic variable is used if the variable isn't rebound using the `binding` form.

Atoms provide a safe way to access and replace a value bound to them. Atoms are synchronous; thus, if an error occurs while altering the Atom, an exception will be thrown in the current thread trying to make the changes. Agents are immune to this.

Changes to an Agent are made asynchronously. Unlike an Atom, you'll have to check whether an error occurred because an error exception won't be thrown in the current thread. This is because changes to an Agent happen in a different thread, so the thread trying to make the change won't be aware of anything going on in the Agent.

When needing to read and make synchronous changes simultaneously, use Refs. Unlike Atoms, you can read and modify multiple Refs simultaneously using the `dosync` form to perform a transaction.

7

Flow Control, Error Handling, and Math

This chapter starts by explaining and displaying examples of common flow control methods. Similar to the `for` loop flow control structures in imperative languages, Clojure has many flow control structures that bind locally scoped variables but don't actually loop (`if-let`, `when-let`, and so on).

Object comparison and casting are covered once you complete the *Flow control* section. Purposely so; the *Object comparison* section doesn't cover numerical comparisons. Numerical comparisons are in the *Arithmetic* section, so don't panic if you don't know how to compare numbers by the end of the *Object comparison* section.

Error handling will be easy for you to pick up if you're coming from a Java language background. You shouldn't worry if you've never had to get a handle on error handling in Java. Clojure keeps error handling very simple.

Arithmetic is the final section and contains many surprises for those who aren't familiar with Clojure. After reading all the sections, try some of the examples in a REPL session while reviewing the chapter.

Flow control

Flow control is important for controlling when and how your code should be executed. Unlike many popular languages, some flow control mechanisms in Clojure (such as, `if-let`, `when-first`) can simultaneously test and define local variables before executing a body of code. This section will cover many but not all forms of flow control.

The `if` form will only take a maximum of three arguments, with the first argument being the object tested for its Boolean value. The second required argument will be evaluated if the Boolean test is `true`, and the third optional argument will be evaluated if the Boolean test is `false`:

```
user> (if true "!" "!!")
"!"
user> (if false "!" "!!")
"!!"
user> (if false "!" )
nil
```

Because the `if` form only takes a maximum of three arguments, the second and third arguments must be wrapped in the `do` form when you need to evaluate more than a single form:

```
user> (if false
      (do (println "Do is needed for multiple")
          (println "forms to be accepted"))
      (println "This will print because false"))
This will print because false
nil
```

The `if-not` form operates in the same way, but the tested argument has the `not` form being applied to it:

```
user> (if-not false "True!" "false :(")
"True!"
user> (if-not true "True!" "false :(")
"false :("
```

The `if-let` form will make a local binding only if the value of the binding isn't `nil` or `false`:

```
user> (if-let [abc (seq [])]
      (println "abc isn't nil!"))

nil
user> (if-let [abc (seq [1 2 3])]
      (println "abc isn't nil!"))

abc isn't nil!
nil
user> (if-let [x false] (println "x is true") (println "false"))
false
nil
```

Because the required second argument of the `if` form requires the `do` form when using multiple code bodies, the `when` form is more appropriate to use when needing to run a body of code based on the outcome of a Boolean value:

```
user> (when false
      (println "You'll see me if not false")
      (println "Me too!!"))
nil
user> (when true
      (println "You'll see me if not false")
      (println "Me too!!"))
You'll see me if not false
Me too!!
nil
```

The `when-not true` form acts in the same manner as the `when` form but the tested value is wrapped in the `not` form:

```
user> (when-not true
      (println "You'll see me if not true")
      (println "Me too!!"))
nil
user> (when-not false
      (println "You'll see me if not true")
      (println "Me too!!"))
You'll see me if not true
Me too!!
nil
```

The working of the `when-let` form is similar to the `if-let` form but the entire body is evaluated if the tested value is `true`. Nothing will be evaluated if the test returns a false value:

```
user> (when-let [abc 123] (println abc))
123
nil
user> (when-let [abc nil] (println abc))
nil
```

We can use the `when-first` form as a shortcut, so you don't have to use the `when-let` form and then bind the first value of a collection:

```
user> (when-first [abc (range 1 10)] (println abc))
1
nil
user> (when-first [abc []] (println abc))
nil
```

The `cond` form will evaluate both a series of tests and the return body of the first test to return `true`. The `:else` keyword and body is placed after the final test and is evaluated only if all previous tests have failed:

```
user> (let [x 1 y 2]
      (cond
        (= x 2) (println "x = 2")
        (= y 3) (println "y = 3")
        :else (println "No match. x & y =" x "," y)))
```

No match. x & y = 1 , 2

nil

user> ;; Notice the missing :else body

```
user> (let [x 1 y 2]
      (cond
        (= x 2) (println "x = 2")
        (= y 3) (println "y = 3")))
```

nil

The `condp` form is similar to the `cond` form with a few exceptions. Instead of passing conditions to `condp`, you pass result values and their return bodies. The first argument is a function that will produce the result value:

```
user> (condp = 1
      2 "It's 2"
      1 "It's 1")
"It's 1"
```

Instead of adding `:else` as you would when using the `cond` form, you simply place the failing operation at the end of the `condp` body:

```
user> (let [[x y z] (range 4)]
      (condp = (rand-int 4)
        x (println "Random = x = " x)
        y (println "Random = y = " y)
        z (println "Random = z = " z)
        (println "No match!")))
```

Random = y = 1

nil

```
user> (let [[x y z] (range 4)]
      (condp = (rand-int 4)
        x (println "Random = x = " x)
        y (println "Random = y = " y)
        z (println "Random = z = " z)
```

```

        (println "No match!"))))
Random = z = 2
nil
user> (case (rand-nth [:a :b :c :d])
      :a "It's a"
      :b "B match"
      :c "Matched C."
      "No match!!!!")

"B match"
user> (case (rand-nth [:a :b :c :d])
      :a "It's a"
      :b "B match"
      :c "Matched C."
      "No match!!!!")

"Matched C."

```

Loops in Clojure take a single value or many values and then bind them to a locally scoped variable. When a value is modified in the `recur` form of the loop body, the changes are applied to the locally-scoped variable:

```

user> (loop [i 0]
      (if (>= i 5)
        (println i)
        (recur
         (inc i))))

5
nil
user> (loop [example []]
      (if (= (count example) 4)
        example
        (recur
         (conj example (rand-int 5)))))

[1 4 4 2]
user> (loop [example [1 2 3 4]]
      (if-not (seq example)
        (println "Done!")
        (do (println (first example))
            (recur
             (rest example)))))

1
2
3
4
Done!
nil

```

Although the `recur` form is primarily used in loops, the `recur` form can also be used to create a recursive function. Keep in mind that a function that repeatedly calls itself should do so using the `lazy-seq` form to prevent a stack overflow:

```
user> (defn example [n]
      (if (< n 3)
          (do (println "Increment" n)
              (recur (inc n)))
          (println "Finished at" n)))
#'user/example
user> (example 0)
Increment 0
Increment 1
Increment 2
Finished at 3
nil
```

An alternative to `recur` is to have your function return an anonymous function that calls your named function. You can then call your function with `trampoline`. The `trampoline` form will execute the returned anonymous function. Notice how the function will continue to call itself as long as it keeps returning a function:

```
user> (defn example [n]
      (if (> n 0)
          (fn []
              (println "n is" n)
              (example (dec n))))
      #'user/example)
user> (trampoline example 5)
n is 5
n is 4
n is 3
n is 2
n is 1
nil
```

The `while` form in Clojure acts the same way as it does in any other language. The body of the form will execute as long as the test evaluates to `true`:

```
user> (let [_ (atom 3)]
      (while (not= 0 @_ )
          (println @_ )
          (swap! _ dec)))

3
2
1
nil
```

Object comparison

Currently Clojure has four ways of comparing two objects. The first way is to use the `compare` form, which will return the number zero if the match is perfect:

```
user> (defn compare-to [x y] (zero? (compare x y)))
#'user/compare-to
user> (compare-to "abc" "abc")
true
```

The second way is to use the `equal` form:

```
user> (= "abc" "abc")
true
```

The third, fourth, and nastiest way is to use Java interoperation methods of the object:

```
user> (.. "abc" (equals "abc"))
true
user> (.. "abc" (compareTo "abc"))
0
```

Casting

To force a value to represent a certain data type, you'll have to cast the values as it's done in most modern languages. It's pretty straightforward like everything else in Clojure:

```
user> (format "%s = %d" (type \A) (int \A))
"class java.lang.Character = 65"
user> (char 0x41)
\A
user> (= 0x41 (byte 0x41) (int \A))
true
```

Sometimes, casting values can come at the price of affecting the performance. Notice the time elapsed on casting the value `1.0`:

```
user> (time (short 1.0))
"Elapsed time: 0.073201 msecs"
1
user> (time (int 1.0))
"Elapsed time: 0.07077 msecs"
1
user> (time (long 1.0))
"Elapsed time: 0.070731 msecs"
1
```


Some languages cast string values to integers, but Clojure isn't one of them. Java interoperation methods are required to convert a string to an integer:

```
user> (int "123")
ClassCastException java.lang.String cannot be cast to
java.lang.Character
user> (time (Integer/parseInt "123"))
"Elapsed time: 0.145911 msecs"
123
user> (time (Integer. "123"))
"Elapsed time: 0.199703 msecs"
123
```

The rest of the casting forms are self-explanatory:

```
user> (float 100)
100.0
user> (float (/ 1 2))
0.5
user> (double 100)
100.0
user> (double (/ 1 2))
0.5
user> (bigint 100)
100N
user> (bigint (/ 1 2))
0N
user> (bigdec 10)
10M
user> (bigdec 10.0)
10.0M
user> (rationalize 0.75)
3/4
user> (rationalize 1.0)
1N
user> (bytes (.getBytes "abc"))
#<byte[] [B@20b1fda8>
user> (String. (bytes (.getBytes "abc")))
"abc"
```

Error handling

If you have any Java experience, this section will be one of the easiest aspects to understand in Clojure. Clojure's error handling is exactly the same as Java's. The `try`, `catch`, `finally`, and `throw` forms all work in the same way. If you don't have any Java experience, here's some information on the forms. The `try` form attempts to run the code body, the `catch` form will catch any errors, and the `finally` form will always run regardless of errors. You use the `throw` form to assert that an error has occurred:

```
user> (try (+ "a" "b")
          (catch Exception e
            (println (.getMessage e))))
java.lang.String cannot be cast to java.lang.Number
nil
user> (defn x [y z]
      (try (+ y z)
        (catch ClassCastException e
          (println "Requires 2 numbers"))
        (catch Exception e
          (println "You broke it!"))
        (finally
          (println "Always prints")))))

#'user/xuser> (x 1 "a")
Requires 2 numbers
Always prints
nil
user> (x 1 2)
Always prints
3
user> (defn x [y z]
      (when (or (zero? y) (zero? z))
        (throw (Exception. "No zeros please"))))
      (+ y z))

#'user/x
user> (x 1 2)
3
user> (x 0 2)
Exception No zeros please user/x (NO_SOURCE_FILE:2)
```

Arithmetic

Clojure's Lisp syntax makes arithmetic operations easier to read and write. In languages such as C or PHP, arithmetic operations involving more than two numbers result in having to type the operator before each additional number. Looking at the following example, which would you rather write?

- **Other languages:** `1 + 2 + 3 + 4`
- **Clojure language:** `(+ 1 2 3 4)`

Addition and subtraction

The `-` and `+` symbols can be used to add and subtract values:

```
user> (+ 1 2 3)
6
user> (+ -1 -2 -3)
-6
user> (+ 1 2 -3)
0
user> (apply + (range 10))
45
user> (- 1 2 3)
-4
user> (- -1 -2 -3)
4
user> (- 1 2 -3)
2
user> (apply - (range 10))
-45
```

Multiplication

The star symbol (`*`) can be used to multiply numbers:

```
user> (* 1 2 3)
6
user> (* -1 -2 -3)
-6
user> (* 1 2 -3)
-6
user> (apply * (range 10))
0
```

Division

The result of dividing two numbers is called the quotient; thus, division in Clojure is performed with the `quot` form:

```
user> (quot 4 2)
2
user> (quot 4 (float 2))
2.0
user> (quot 4 (double 2))
2.0
user> (quot 4 (bigint 2))
2N
user> (quot 4 (bigdec 2))
2M
user> (quot 4 0)
ArithmeticException / by zero
```

Because `quot` can only accept a numerator and denominator, you'll have to use the `/` operator to continually divide with multiple denominators:

```
user> ;; (4/2) / 2 = 1
user> (/ 4 2 2)
1
user> (/ 4 2)
2
user> (/ 4)
1/4
user> (/ 1 2)
1/2
```

Remainder and modulus

The `rem` and `mod` forms can be used to get the remainder or modulus value of two numbers:

```
user> (rem 100 99)
1
user> (rem 100 98)
2
user> (rem 100 97)
3
user> (mod 100 99)
1
user> (mod 100 98)
2
user> (mod 100 97)
3
```

Increment and decrement

The `inc` and `dec` forms can be used for incrementing and decrementing numbers:

```
user> (inc 0)
1
user> (inc -1)
0
user> ;; This means (+ 1/2 2/2) because 2/2 represents the 1
user> (inc 1/2)
3/2
user> (dec 0)
-1
user> (dec -1)
-2
user> ;; This means (- 3/4 4/4) because 4/4 represents the 1
user> (dec 3/4)
-1/4
```

Greatest and least values

With the `min` or `max` forms, you can find the least or greatest value within a series of numbers:

```
user> (max 2)
2
user> (max 1 2)
2
user> (max 0 1 2)
2
user> (apply max (range 3))
2
user> (min 2)
2
user> (min 1 2)
1
user> (min 0 1 2)
0
user> (apply min (range 3))
0
```

Equality

Equality in Clojure also checks the type, so you'll have to use `==` to ignore the value types:

```
user> (= 1 (int 1.0) (int 1/1))
true
user> (= 1 1.0 1/1)
false
user> (== 1 1.0 1/1)
true
user> (not= 1 1)
false
user> (not= 1 2)
true

user> (<= -1 0)
true
user> ;; -1 <= 0 and -1 <= 1
user> (<= -1 0 1)
true
user> (<= -1 0 1 2 3)
true
user> (>= -1 0)
false
user> ;; -1 >= 0 and -1 >= 1
user> (>= -1 0 1)
false
user> (>= -1 0 1 2 3)
false

user> (< -1 0)
true
user> (< -1 0 1)
true
true
user> (> -1 0)
false
user> (> -1 0 1)
false

user> (nil? 0)
false
user> (zero? 0)
true
user> (even? 0)
true
```

```
user> (odd? 0)
false
user> ;; pos? = positive?
user> (pos? 0)
false
user> (pos? 1)
true
user> ;; neg? = negative?
user> (neg? 0)
false
user> (neg? -1)
true
```

Summary

Clojure gives you many flow control options, and some of these options can also bind locally-scoped variables. The locally-scoped variables in the `loop` form should be changed in the `recur` form body placed at the end of a loop. If you forget to test whether `recur` has to be called again, the loop will continue forever. Also, because the `if` form can only take two and optionally three arguments, you can use the `do` form to pass entire code bodies as a single argument.

Error handling in Clojure is very much like error handling in Java. By simply wrapping a body of code with the `try` form, you can catch specific or all exception types with the `catch` form. Remember to use the `finally` form when you need to run bodies of code regardless of the occurrence of an error.

Arithmetic in Clojure is unique because many operations can take an unlimited amount of arguments. Also, because of the Lisp syntax, it's a lot easier to only having to write the operation symbol once instead of having to place it between every operand.

8

Methods for Abstraction

This chapter starts off with an explanation of the classes we need to make 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.

This chapter will cover:

- Creating and constructing classes
- Overriding methods
- Multimethod polymorphism
- Custom data types
- Parent and child relationships

Creating and constructing classes

This section focuses on implementing Clojure-defined interfaces and creating classes. After covering interfaces, you'll learn how to make classes with and without class-specific methods.

Creating interfaces and implementing them with deftype

Interfaces in Clojure can be created using `definterface`. Just as with a Java interface, `definterface` creates a class that is designed to have its methods belong to another class, as follows:

```
user> (definterface ICan
      (^String canSize [])
      (^String openCan [])
      (^String getLabel []))
user.ICan
```


ICan is now a Java interface and can be used as such. In the following example, the `deftype` form is used to create a class named `Can`, and it will implement the `ICan` interface. The word `this` is normally seen in object-oriented languages, but here the word has been replaced with an underscore symbol to represent the current instance of the class, as shown in the following code snippet:

```
user> (deftype Can
      [contents can-size]
      ICan
      (canSize [_] (name can-size))
      (openCan [_] (name contents))
      (getLabel [_] (format "A %s can of %s"
                           (name can-size)
                           (name contents))))

user.Can
```

The constructor of the `Can` class can be accessed by appending a decimal mark to the class name followed by the constructor arguments. In this case, there are only two arguments required, as shown in the following code snippet:

```
user> (def c (Can. :Apples :small))
#'user/c
user> (type c)
user.Can
```

Invoking the interface methods on the `c` symbol will require the Java interoperation syntax, because the `Can` class implements a Java interface and not a Clojure protocol, as shown in the following code snippet:

```
user> (.canSize c)
"small"
user> (.openCan c)
"Apples"
user> (.getLabel c)
"A small can of Apples"
```

Using records, protocols, and type extensions

The `defrecord` form will define a class that is named constructor arguments. This class can have its constructor arguments accessed the same way you would access values of a `HashSet` or `HashMap`, as shown in the following code snippet:

```
user> (defrecord Not-A-Can [x y])
user.Not-A-Can
user> (def nac (Not-A-Can. 1 2))
```

```

#'user/nac
;; same as above but not so nice
user> (def nac (->Not-A-Can 1 2))
#'user/nac
user> (type nac)
user.Not-A-Can
user> nac
#user.Not-A-Can{:x 1, :y 2}
user> [(:x nac) (:y nac)]
[1 2]

```

Although records can implement their own methods, records without a defined set of methods can be extended with a protocol. A protocol is similar to a Java interface, but the methods aren't called with the Java interoperability syntax, as shown in the following code snippet:

```

user> (defprotocol PCan
      (can-size [_])
      (open-can [_])
      (label [_]))
PCan

```

A protocol also has values that can be accessed in the same way you would access values of a `HashSet` or `HashMap`, as shown in the following code snippet:

```

user> (keys PCan)
(:on-interface :on :sigs :var :method-map :method-builders)
user> (clojure.pprint/pprint (:sigs PCan))
{:label {:doc nil, :arglists ([_]), :name label},
 :open-can {:doc nil, :arglists ([_]), :name open-can},
 :can-size {:doc nil, :arglists ([_]), :name can-size}}
nil
user> (keys (:sigs PCan))
(:label :open-can :can-size)

```

Because the `Not-A-Can` record class didn't implement any methods, the `extend-type` form will extend the class and provide the protocol method overrides. In the following example, `Not-A-Can` is being extended by the `PCan` protocol so that it can inherit the methods described by the `extend-type` form. Remember, instead of using the word `this`, the underscore symbol is used to represent the current instance, as shown in the following code snippet:

```

user> (extend-type Not-A-Can
      PCan
      (can-size [_] (:x _))
      (open-can [_] "Not really a can. Sorry.")
      (label [_] (str "Example: " (:x _) (:y _))))

```

Once the `Not-A-Can` record class has been extended, you may call the class-specific methods without having to use Java interoperation syntax. Trying to make these specific calls on a non `PCan` class object or `extended-type` will always result in an error exception, as shown in the following code snippet:

```
user> (can-size nac)
1
user> (open-can nac)
"Not really a can. Sorry."
user> (label nac)
"Example: 12"
user> (type nac)
user.Not-A-Can
user> (extends? PCan Not-A-Can)
true
```

Because `extend-type` only extends a single class, `extend-protocol` is required to extend multiple classes with a single protocol. The `extend-protocol` form only takes two types of arguments after the first protocol. The first of the two types is the class. The second of the two is the method overrides of the protocol. The overrides must come directly after a class.

In the following example, the `String`, `Integer`, and `Float` classes are being extended with the `PCan` protocol in the same code body. Visually speaking, when looking at the code body, the multiple method overrides are delimited by the class that precedes them, as shown in the following code snippet:

```
user> (extend-protocol PCan
      String
      (can-size [_] "small")
      (open-can [_] _)
      (label [_] (str "Can of java.lang.String")))
Integer
      (can-size [_] "mini")
      (open-can [_] _)
      (label [_] (str "Can of Integer(s)")))
Float
      (can-size [_] "mini")
      (open-can [_] _)
      (label [_] (str "This can holds Float(s)")))
nil
user> (extends? PCan String)
true
user> (extends? PCan Float)
true
```

```

user> (can-size (int 1))
"mini"
user> (open-can (float 1.0))
1.0
user> (label (float 1.0))
"This can holds Float(s)"
user> (label "ABC")
"Can of java.lang.String"

```

The second way of implementing protocol methods is by placing the protocol class after the record arguments followed by the method overrides, as shown in the following code snippet:

```

user> (defrecord Can [contents size]
      PCan
      (can-size [_] (name (:size _)))
      (open-can [_] (name (:contents _)))
      (label [_] (format "A %s can of %s"
                        (can-size _)
                        (open-can _))))
      User.Can

```

The Can record class still behaves and looks like a regular record but now the methods of the PCan protocol can be called on the record, as shown in the following code snippet:

```

user> (def apple-can (Can. :Apples :small))
#'user/apple-can
user> apple-can
#user.Can{:contents :Apples, :size :small}
user> (can-size apple-can)
"small"
user> (open-can apple-can)
"Apples"
user> (label apple-can)
"A small can of Apples"

```

Looking at the following example, you'll notice that the Can record isn't in the list of classes that are extended by PCan. That's because `extend-type` or `extend-protocol` wasn't used to give the Can record class its methods. The `satisfies?` form confirms whether an object is extended by a protocol, as shown in the following code snippet:

```

user> (doseq [_ (extenders PCan)] (println _))
java.lang.String
java.lang.Integer
java.lang.Float

```

```
user.Not-A-Can
nil
user> (satisfies? PCan (int 1))
true
user> (satisfies? PCan "abc")
true
user> (satisfies? PCan (float 1))
true
```

Overriding methods with reify and proxy

Overriding methods of a protocol can be implemented with the `reify` form, while overrides for interfaces and classes can be implemented with the `proxy` form. These two forms are meant to provide a way to make anonymous instances of protocols and interfaces, but these forms double as a simple way to write method overrides.

Working with reify

We'll use the `PCan` protocol from the last section in the following examples. The following is the `PCan` protocol, in case you don't have it from the last section:

```
user> (defprotocol PCan
  (can-size [_])
  (open-can [_])
  (label [_]))

PCan
```

In the following example, `reify` is used to make a locally-scoped instance of the `PCan` class. Because the returned class can be requested by its symbol name, it's technically no longer anonymous, as shown in the following code snippet:

```
user> (let [local-can (reify PCan
  (open-can [_] "tuna")
  (can-size [_] "tiny")
  (label [_] "Tuna in water"))
  open-msg (format "Opening a %s can of %s"
    (can-size local-can)
    (open-can local-can))
  discover-msg (format "I found a can of %s"
    (.toLowerCase (label local-can)))]
  (println
    (format "%s\n%s"
      discover-msg
      open-msg)))
```

```
I found a can of tuna in water
Opening a tiny can of tuna
nil
```

Another way to use `reify` is to modify overrides based on previous method calls. For this example, we'll need to define a class named `Can` and construct an instance within an atom named `orange-can`. Before this can happen, the `Can` record class has to be created to follow the upcoming examples:

```
user> (defrecord Can [contents size]
      PCan
      (can-size [_] (name (:size _)))
      (open-can [_] (name (:contents _)))
      (label [_] (format "A %s can of %s"
                        (can-size _)
                        (open-can _))))

User.Can
```

The `orange-can` symbol will be an atom that contains the `Can` class instance. This is because updates can be applied to the `Can` class using the `swap!` form, as shown in the following code snippet:

```
user> (def orange-can (atom (Can. :oranges :large)))
#'user/orange-can
```

In the following example, `reify` is used to change the contents and label of a `Can` class. The `eat-contents!` function will change the contents of the can to `empty`, and then modify the label of the can to be prefixed with `(Empty)`. Remember that a symbol ending with an exclamation point is stating that there will be side effects.

```
user> (defn eat-contents! [can-atom]
      (let [{:keys [size contents]} @can-atom
            size (name size)
            contents (name contents)]
        (reset! can-atom
                 (reify PCan
                     (open-can [_] "empty")
                     (can-size [_] size)
                     (label [_]
                      (format "(Empty) A %s can of %s"
                              size
                              contents))))))

#'user/eat-contents!
```

The `eat-canned-food!` function is then defined so that we can see a status message when the contents of a can have been eaten, as shown in the following code snippet:

```
user> (defn eat-canned-food! [can-atom]
      (println
        (format "Eating a can of %s"
                  (open-can @can-atom)))
        (eat-contents! can-atom)
        :ate)
      #'user/eat-canned-food!)
```

Before calling `eat-canned-food!`, let's look at the output of the `PCan` methods, as follows:

```
user> (open-can @orange-can)
"oranges"
user> (label @orange-can)
"A large can of oranges"
```

Now, let's see the difference in the output of the `PCan` methods after calling `eat-canned-food!` on the `orange-can` atom, as shown in the following code snippet:

```
user> (eat-canned-food! orange-can)
Eating a can of oranges
:ate
user> (open-can @orange-can)
"empty"
user> (can-size @orange-can)
"large"
user> (label @orange-can)
"(Empty) A large can of oranges"
```

Implementing interface methods with proxy

The `proxy` form is somewhat similar to the `reify` form, but `proxy` is for Java interfaces and classes. The following example interface will be the focus of the `proxy` form examples in this section:

```
user> (definterface IBox
      (^Float length [])
      (^Float width [])
      (^Float height [])
      (^Float volume []))
user.IBox
user> (type IBox)
java.lang.Class
```

The idea here is to construct a new `IBox` by taking the arguments from a function and placing them inside the proxy method overrides, as shown in the following code snippet:

```
user> (defn box [l w h]
      (proxy [IBox] []
        ;; the empty vec after [IBox] means that no arguments
        ;; are being passed to the class constructor
        (length [] (float l))
        (width [] (float w))
        (height [] (float h))
        (volume [] (apply * (map float [l w h])))))
#'user/box
```

Since `IBox` is an interface, Java interoperation syntax is required to call the methods. The following is an example of constructing a new box and calling the `IBox` methods:

```
user> (let [b (box 1 2 3)]
      (println "Length:" (.length b)
               "Width:" (.width b)
               "Height:" (.height b)
               "Volume:" (.volume b)))
Length: 1.0 Width: 2.0 Height: 3.0 Volume: 6.0
nil
```

Custom symbol definitions with macros

Binding class instances to a symbol can sometimes be tedious. Using macros can help ease the tedious pattern of using the combination of `def` and the class constructors.

Definitions using records

We want `(def user2 (User. "Real Name" mod? admin?))`, to become `(def user2 "Real Name" mod? admin?)`.

A protocol named `PUser` will be required for this example. This protocol will return the `real-name`, `mod?`, and `admin?` values of a record, as shown in the following code snippet:

```
user> (defprotocol PUser
      (real-name [_])
      (mod? [_])
      (admin? [_]))
PUser
```


Next, a record class named `User` will implement the `PUser` protocol, as shown in the following code snippet:

```
user> (defrecord User [name mod? admin?]
      PUser
      (real-name [_] (:name _))
      (mod? [_] (:mod? _))
      (admin? [_] (:admin? _)))
user.User
```

Now that we have the `User` record class, we need to make a macro that'll bind a symbol to an instance of the `User` record class. The reason why we need a macro in this case is because a macro generates a code template that will be evaluated when returned. This macro will tell the `def` form that the `-symbol` variable isn't a template form, but rather a value that needs to be embedded within the template. Also, the `User` instance will be created before being passed as a value, as shown in the following code snippet:

```
User> ;; Macro templates are prefixed with the '`' character.
user> (defmacro defuser [-symbol name & [mod? admin?]]
      `(def ~-symbol
         ~(User. name
                (boolean mod?)
                (boolean admin?))))
#'user/defuser
```

Using the `macroexpand-1` form, we can see whether the macro has created the proper code we wish to execute, as shown in the following code snippet:

```
user> (clojure.pprint/pprint
      (macroexpand-1
        '(defuser user2 "Kevin Pereira" true false)))
(def user2 {:name "Kevin Pereira", :mod? true, :admin? false})
nil
```

We can now safely use the macro knowing the code generated is the desired result, as shown in the following code snippet:

```
user> (defuser user2 "Kevin Pereira" true false)
#'user/user2
user> user2
#user.User{:name "Kevin Pereira", :mod? true, :admin? false}
user> (type user2)
user.User
```

Now that `defuser` has given us a new `User` instance, we can test that all the methods work properly, as shown in the following code snippet:

```
user> (real-name user2)
"Kevin Pereira"
user> (mod? user2)
true
user> (admin? user2)
false
```

Making definitions using proxy

Making custom definition forms with `proxy` isn't that different from making definition forms with records. One of the key differences here is that you have to use Java interoperability syntax to access the methods of the class instance.

The following interface will be the class of the macro return type. This interface describes a phone number, the ability to accept incoming calls, and whether the phone is connected to the network.

```
user> (definterface IPhone
      (^Number number [])
      (^Boolean acceptIncoming [])
      (^Boolean connected []))
user.IPhone
```

The `defphone` macro works a little differently than the previous macro example from the records section. This macro makes local bindings outside of the code generation template, and then evaluates the local binding within a code template, as shown in the following code snippet:

```
user> (defmacro defphone
      [-symbol
       -number
       accept-incoming?
       connected?]
      ;; the empty vector after [IPhone] states that no arguments
      ;; are being passed to the class constructor
      (let [phone (proxy [IPhone] [])
            (number [] -number)
            (acceptIncoming [] (boolean accept-incoming?))
            (connected [] (boolean connected?))])
        `(def ~-symbol ~phone)))
#'user/defphone
```

After creating a new `IPhone` instance with the `defphone` macro, the methods can be called with Java interoperation syntax, as shown in the following code snippet:

```
user> (defphone phone1 5551115555 true true)
#'user/phone1
user> (.number phone1)
5551115555
user> (.acceptIncoming phone1)
true
user> (.connected phone1)
true
```

Making definitions using `deftype`

Using `proxy` to implement an interface can lead to some lengthy code. As you'll see in the following example, using `deftype` makes the macro a lot cleaner:

```
user> (deftype PhoneType [phone-num]
      IPhone
      (number [_] phone-num)
      (acceptIncoming [_] true)
      (connected [_] true))
user.PhoneType
user> (defmacro defphone-type [-symbol -number]
      `(def ~-symbol (PhoneType. ~-number)))
#'user/defphone-type
user> (defphone-type phone2 2222)
#'user/phone2
user> (type phone2)
user.PhoneType
user> (type phone1)
user.proxy$java.lang.Object$IPhone$8d520228
user> (.number phone2)
2222
user> (.number phone1)
5551115555
```

Multimethod polymorphism

The `defmulti` form will create a multifunction that returns a value, which is then used to determine which body of code to evaluate in the `defmethod` form. The first argument of the `defmulti` form is the symbol that holds the multimethod. The second argument is a function, and the number of arguments this function has will also be required by the `defmethod` forms.

The following example will make a decision on the return vector that's based on the customer and bottle arguments. The vector will look similar to `[:sweet :carbon true true]`.

```
user> (defmulti customer-drinks
      (fn [customer bottle]
        [(if (:sweet? bottle) :sweet :not-sweet)
         (if (:carbonated? bottle) :carbon :no-carbon)
         (:drinks-sweet? customer)
         (:drinks-carbonated? customer)])))

#'user/customer-drinks
```

To make this multimethod work properly, multimethod handlers have to be defined with `defmethod`. The first argument will have to be a multimethod, of course, and the second value will have to match the return value of the multimethod. The second argument can be a class or a value, but in this case it's going to be similar to the vector discussed at the beginning of this section. The third argument is the same number of arguments as defined by the `defmulti` binding.

The following `defmethod` form is for customers who can drink sweet and noncarbonated drinks:

```
user> (defmethod customer-drinks
      [:sweet :no-carbon true false]
      [customer bottle]
      (println "Yum! This is sweet and non-carbonated."))

#<MultiFn clojure.lang.MultiFn@64350a8>
```

The following `defmethod` form is for customers who can drink nonsweet and noncarbonated drinks:

```
user> (defmethod customer-drinks
      [:not-sweet :no-carbon false false]
      [customer bottle]
      (println "This is non-sweet and non-carbonated."))

#<MultiFn clojure.lang.MultiFn@64350a8>
```

The following `defmethod` form is for customers who can drink nonsweet and carbonated drinks:

```
user> (defmethod customer-drinks
      [:not-sweet :carbon false true]
      [customer bottle]
```

```
(println "I love unsweet and carbonated drinks."))
```

```
#<MultiFn clojure.lang.MultiFn@64350a8>
```

The final `defmethod` form is for customers who can drink sweet and carbonated drinks, as follows:

```
user> (defmethod customer-drinks
      [:sweet :carbon true true]
      [customer bottle]
      (println "This is bad for me, but I'll drink it."))
```

```
#<MultiFn clojure.lang.MultiFn@64350a8>
```

Creating the Bottle and Customer classes

Before moving on, some classes need to be defined for the upcoming examples. The first one is the `Bottle` class. This class will hold the characteristics of a customer's drink, as shown in the following code snippet:

```
user> (defrecord Bottle [type sweet? carbonated?])
user.Bottle
user> (defmacro defdrink [s type sweet? carbonated?]
      `(def ~s
         ~(Bottle. type
                  sweet?
                  carbonated?)))

#'user/defdrink
```

The second class that needs to be defined is the `Customer` class. This class will hold the details about what they prefer to drink, as shown in the following code snippet:

```
user> (defrecord Customer
      [drinks-sweet?
       drinks-carbonated?])

user.Customer
user> (defmacro defcustomer
      [s
       drinks-sweet?
       drinks-carbonated?]
      `(def ~s
         ~(Customer. drinks-sweet?
                     drinks-carbonated?)))

#'user/defcustomer
```

Using the `:default` keyword as the second argument in the `defmethod` form will state that this `defmethod` form should be used when others fail to match the `defmulti` return value. You'll be needing this so that you can properly handle the outcome of invalid argument types. The following example specifically checks for the `Customer` and `Bottle` record classes:

```
user> (defmethod customer-drinks
      :default
      [customer _]
      (if-not (= Customer (type customer))
        (println "This isn't a customer")
        (if (not= Bottle (type _))
          (println "I don't know if I should drink this.")
          (let [{:keys [drinks-sweet? drinks-carbonated?]}
                customer
                sweet-msg (if drinks-sweet?
                              "sweet"
                              "non-sweet")
                carb-msg (if drinks-carbonated?
                              "carbonated"
                              "non-carbonated")
                msg (format "I can only have a %s %s drink."
                           sweet-msg
                           carb-msg)]
            (println msg))))))
#<MultiFn clojure.lang.MultiFn@64350a8>
```

Testing the customer-drink methods

Now that we have two new classes and some multimethods, we need to define some `Customer` and `Bottle` variables. Since we have a multimethod that handles four types of drinks, we'll define one of each with the `defdrink` macro, as follows:

```
user> (defdrink orange-juice :orange-juice true false)
#'user/orange-juice
user> (defdrink generic-soda :generic-soda true true)
#'user/generic-soda
user> (defdrink carbon-water :carbon-water false true)
#'user/carbon-water
user> (defdrink cold-coffee :cold-coffee false false)
#'user/cold-coffee
```

The `customer-drinks` multimethod also recognizes four types of customer preferences, so we'll define one of each with the `defcustomer` macro, as follows:

```
user> (defcustomer customer1 true false)
#'user/customer1
user> (defcustomer customer2 true true)
#'user/customer2
user> (defcustomer customer3 false true)
#'user/customer3
user> (defcustomer customer4 false false)
#'user/customer4
```

Looking at the following example, you can see that orange-juice is sweet and isn't carbonated; you'll also see that `customer1` would prefer this kind of drink:

```
user> orange-juice
#user.Bottle{:type :orange-juice, :sweet? true, :carbonated? false}
user> customer1
#user.Customer{:drinks-sweet? true, :drinks-carbonated? false}
```

Let's call the `customer-drinks` multimethod with this customer-and-drink combination, as shown in the following code snippet:

```
user> (customer-drinks customer1 orange-juice)
Yum! This is sweet and non-carbonated.
nil
```

So far so good, but what about a drink that's not to the customer's liking?

```
user> cold-coffee
#user.Bottle{:type :cold-coffee, :sweet? false, :carbonated? false}
user> (customer-drinks customer1 cold-coffee)
I can only have a sweet non-carbonated drink.
nil
```

Seems like the `customer-drinks` multimethod is working just fine with customer-and-bottle combinations, so let's see what happens when a customer is offered something that's not drinkable, as shown in the following code snippet:

```
user> (customer-drinks customer1 "not a bottle")
I don't know if I should drink this.
nil
```

The customer wasn't able to drink the string of characters because there wasn't a `defmethod` form for this type; so the `:default defmethod` kicked in as a last resort. The same thing happens when you tell `number` to drink orange-juice in the following example:

```
user> (customer-drinks 1 orange-juice)
This isn't a customer
nil
user> (customer-drinks 1 2)
This isn't a customer
nil
```

Let's see what one other customer says about a couple of drinks, as shown in the following code snippet:

```
user> customer2
#user.Customer{:drinks-sweet? true, :drinks-carbonated? true}
user> generic-soda
#user.Bottle{:type :generic-soda, :sweet? true, :carbonated? true}
user> (customer-drinks customer2 generic-soda)
This is bad for me, but I'll drink it.
nil
user> (customer-drinks customer2 orange-juice)
I can only have a sweet carbonated drink.
nil
user> (customer-drinks customer2 "not a bottle")
I don't know if I should drink this.
nil
```

Relationships with hierarchies

Hierarchies in Clojure are used to make child-parent relationships between classes, keywords, and symbols. These relationships can be used to determine which code body to evaluate for a `defmulti` function. This chapter will go over the capabilities of hierarchies, and then introduce examples of how to use hierarchies with `defmulti` polymorphism.

Hierarchies are global by default but, for the following example, the `local-h` symbol will hold the relationships. Because the `make-hierarchy` form returns a persistent array-map, the hierarchy is wrapped inside an atom so that the `swap!` form can apply and maintain changes to the hierarchy:

```
user> (def local-h (atom (make-hierarchy)))
#'user/local-h
```


In the following example, the `derive` form is stating that `::phone` is the child of `::device`:

```
user> (let [output (swap! local-h derive ::phone ::device)]
      (clojure.pprint/pprint output))
{:parents {:user/phone #{:user/device}},
 :ancestors {:user/phone #{:user/device}},
 :descendants {:user/device #{:user/phone}}}
nil
```

The `isa?` form will allow you to test the child-parent relationship between symbols, classes, and keywords. The global hierarchy of the current namespace is assumed by the `isa?` form, unless a hierarchy is passed as the first argument. Because the `local-h` atom holds our local hierarchy, we'll have to use dereferencing syntax (the `@` sign prefixing `local-h`) to access the relationships, as shown in the following code snippet:

```
user> (isa? @local-h ::phone ::device)
true
user> (isa? @local-h ::device ::phone)
false
user> (parents @local-h ::phone)
#{:user/device}
```

Because `::device` is the parent of `::phone`, any children parented by `::phone` will also be a child of `::device`. In the upcoming examples, `::phone` also becomes a child of `::pocket-device` and `::pocket-device` becomes a child of `::device`:

```
user> (clojure.pprint/pprint
      (swap! local-h derive ::phone ::pocket-device))
{:parents {:user/phone #{:user/device :user/pocket-device}},
 :ancestors {:user/phone #{:user/device :user/pocket-device}},
 :descendants
  {:user/pocket-device #{:user/phone}, :user/device #{:user/phone}}}
nil
```

In the following code snippet, you'll notice that `::phone` has two separate parents:

```
user> (isa? @local-h ::phone ::device)
true
user> (isa? @local-h ::phone ::pocket-device)
true
user> (let [output (swap! local-h derive ::pocket-device ::device)]
      nil)
user> (isa? @local-h ::pocket-device ::device)
true
user> (isa? @local-h ::phone ::pocket-device)
true
```

Using the `ancestors` form on `::phone`, `::pocket-device`, and `::device` will display all related relationships that came before them, as shown in the following code snippet:

```
user> (ancestors @local-h ::phone)
#{:user/device :user/pocket-device}
user> (ancestors @local-h ::pocket-device)
#{:user/device}
user> (ancestors @local-h ::device)
nil
```

In the following example, the `underive` form is used to remove `::phone` and `::pocket-device` as children of `::device`. Pay close attention to the output of the `parents` form:

```
user> (let [_ (swap! local-h underive ::phone ::device)
            _ (swap! local-h underive ::pocket-device ::device)]
      (parents _ ::phone))
#{:user/pocket-device}
user> (isa? @local-h ::pocket-device ::device)
false
user> (isa? @local-h ::phone ::pocket-device)
true
user> (ancestors @local-h ::phone)
#{:user/pocket-device}
```

Hierarchies affect how a multifunction reacts depending on the parent-child relationship. In the following example, there's the `customer-uses` multifunction and the expected return classes are `::can-eat` and `::device`:

```
user> (defmulti customer-uses
      (fn [customer _] (class _)))
#'user/customer-uses
user> (defmethod customer-uses ::can-eat [customer _]
      (println customer "eats" (:contents _)))

#<MultiFn clojure.lang.MultiFn@5c161337>
user> (defmethod customer-uses ::device [customer _]
      (println customer "is calling from" (:number _)))

#<MultiFn clojure.lang.MultiFn@5c161337>
```

A few classes need to be made before the `customer-uses` multifunction can be used. The first class is `Phone` and it will be a child of `::device`. The second class is `Food` and the parent will be `::can-eat`, as shown in the following code snippet:

```
user> (defrecord Phone [number])
user.Phone
user> (derive Phone ::device)
nil
user> (isa? Phone ::device)
true
user> (defrecord Food [contents])
user.Food
user> (derive Food ::can-eat)
nil
user> (isa? Food ::can-eat)
true
```

Now, the `customer-uses` multifunction can be tested out on the two relationships created between the classes, as shown in the following code snippet:

```
user> (let [apple (Food. "apple")]
      (customer-uses "Ryan" apple))
Ryan eats apple
nil
user> (let [phone (Phone. 1115558888)]
      (customer-uses "Ryan" phone))
Ryan is calling from 1115558888
nil
```

Resolving parent relationship conflicts

The previous examples were simple, but sometimes relationships can get a little tricky when there's more than one parent. Before moving on to solve this type of issue, some more classes will need to be made for this example, as shown in the following code snippet:

```
user> (defrecord HTML5 [url])
user.HTML5
user> (defrecord Flash [url])
user.Flash
user> (derive HTML5 ::html)
nil
user> (derive HTML5 ::video-tech)
nil
user> (derive Flash ::video-tech)
nil
```

The `Flash` class is a child of `::video-tech`, but the `HTML5` class is a child of both `::html` and `::video-tech`. The following example will explore what kinds of issues this will cause for the multifunction named `technology`:

```
user> (defmulti technology class)
#'user/technology
user> (defmethod technology ::video-tech [_]
      (println "This is a video capable technology"))
#<MultiFn clojure.lang.MultiFn@6cd08990>
user> (defmethod technology ::html [_]
      (println "This is a markup language"))
#<MultiFn clojure.lang.MultiFn@6cd08990>
```

The `Flash` class has only one parent, as shown in the following code snippet, so obviously there are no surprises here:

```
user> (let [video (Flash. "x.com/x.avi")]
      (technology video))
This is a video capable technology
nil
```

Because the `HTML5` class has more than one parent, an exception error is raised when trying to call the multifunction, as shown in the following code snippet:

```
user> (let [html-video (HTML5. "x.com/x.avi")]
      (try
        (technology html-video)
        (catch Exception e (.getMessage e))))

"Multiple methods in multimethod 'technology' match dispatch
value: class user.HTML5 -> :user/video-tech and :user/html,
and neither is preferred"
```

The `technology` multifunction needs to know which parent relationship wins when having to decide between the two relationships. This can be done with the `prefer-method` form. In the following example, the `technology` multifunction is being told how to handle the relationship conflict between `::html` and `::video-tech`:

```
user> (prefer-method technology ::html ::video-tech)
#<MultiFn clojure.lang.MultiFn@6cd08990>
```

The following are the results after calling `prefer-method` on the `technology` multifunction:

```
user> (let [html-video (HTML5. "x.com/x.avi")]
      (technology html-video))
This is a markup language
nil
```

```
user> (let [video (Flash. "x.com/x.avi")]
      (technology video))
This is a video capable technology
nil
```

Assertion testing with metadata

Clojure uses the `test` form to evaluate assertions placed in the metadata of functions. Because metadata doesn't affect the input or output of its parent function, these tests shouldn't be confused with the `:pre` and `:post` input and output constraints, respectively.

The following example function is incorrect on purpose so that you can see what it looks like when a test fails. Also, notice how the `:test` metadata value is actually an anonymous function that makes assertions about the return value of the example function:

```
user> (defn example-fn
      {:test #(do
                (assert (= (example-fn 1) 2))
                (assert (= (example-fn -1) -2)))}
      [i] i)
#'user/example-fn
```

Now, it's time to fail some tests:

```
user> (example-fn 1)
1
user> (test #'example-fn)
AssertionError Assert failed: (= (example-fn 1) 2)
```

That was pretty easy, no? The following is the same function written properly so that it can pass its tests. If everything goes right, the keyword value of `:ok` will be returned by the `test` form.

```
user> (defn example-fn
      {:test #(do
                (assert (= (example-fn 1) 2))
                (assert (= (example-fn -1) -2)))}
      [i] (if (pos? i) (inc i) (dec i)))
#'user/example-fn
user> (test #'example-fn)
:ok
```

Input constraints with :pre

Clojure functions can apply input constraints by placing value tests within an `array-map` as the value to the `:pre` key. This `array-map` comes right after the function parameters and right before the body of the function. The tests within the `array-map` are treated as assertions to be evaluated before the body of the function can be evaluated, as shown in the following code snippet:

```
user> (defn x [y] {:pre [(even? y)]} (inc y))
#'user/x
user> (x 1)
AssertionError Assert failed: (even? y)  user/x (NO_SOURCE_FILE:1)
user> (x 2)
3
```

A collection of tests won't be treated as a single assertion. Each test within the collection is treated as its own assertion test. Notice how each assertion throws its own exception error in the following REPL example:

```
user> (defn x [y]
      {:pre [(even? y)
              (<= y 4)
              (not= y 2)]}
      (inc y))
#'user/x
user> (x 1)
AssertionError Assert failed: (even? y)  user/x (NO_SOURCE_FILE:1)
user> (x 6)
AssertionError Assert failed: (<= y 4)  user/x (NO_SOURCE_FILE:1)
user> (x 2)
AssertionError Assert failed: (not= y 2)  user/x (NO_SOURCE_FILE:1)
user> (x 4)
5
```

Output constraints with :post

Clojure functions can put constraints on return values as easily as it was to constrain the input in the previous section. There are some differences you should be aware of, though. For example, the syntax that represents the return value is a percent sign and is similar to the Clojure anonymous function syntax, as shown in the following code snippet:

```
user> (defn x [y] {:post [(even? %)]} (inc y))
#'user/x
user> (x 1)
```

```
2
user> (x 2)
AssertionError Assert failed: (even? %) user/x (NO_SOURCE_FILE:1)
user> (defn x [y z]
      {:post [(< (first %) 10)
              (even? (apply + %))]}
      [y z])
#'user/x
user> (x 2 2)
[2 2]
user> (x 10 2)
AssertionError Assert failed: (< (first %) 10) user/x (NO_SOURCE_
FILE:1)
user> (x 8 2)
[8 2]
user> (x 9 2)
AssertionError Assert failed: (even? (apply + %)) user/x (NO_SOURCE_
FILE:1)
```

When needing to place constraints on both the input and output, you can safely use an array-map containing both `:pre` and `:post`, as shown in the following code snippet:

```
user> (defn x [y z]
      {:pre [(odd? z)]
       :post [(even? %)]}
      (+ y z))
#'user/x
user> (x 1 2)
AssertionError Assert failed: (odd? z) user/x (NO_SOURCE_FILE:1)
user> (x 1 3)
4
user> (x 2 3)
AssertionError Assert failed: (even? %) user/x (NO_SOURCE_FILE:1)
user> (x 3 3)
6
```

Summary

This chapter covered input and output constraints of functions, custom data types, and multiple forms of polymorphism.

In the next chapter, you'll start building a Twitter DSL using most of what you've learned in the previous chapters.

9

An Example Twitter DSL

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

This chapter will cover:

- Clojure-wrapping Java objects
- Twitter OAuth configuration
- Twitter DSL example uses
- Building a Twitter DSL syntax

Creating Java-based abstractions

Before moving on to making an example DSL, there are some concepts that need to be covered regarding Java interoperation. Because the Twitter DSL example will be based on an existing Java library, this section covers many (but not all) the possible ways to make Java objects, fields, and methods more Clojure-friendly. This is important for keeping the level of code verbosity to a minimum and helping to hide the Java interoperability syntax.

The first and, probably, the most obvious layer of Clojure you can build off a Java class is to make a function that returns a new instance of the Java class. For this and the remaining examples of this section, the `java.util.Calendar` class will be used to explain how to build a nice layer of Clojure on top of Java:

```
user> (defn date [] (java.util.Calendar/getInstance))
#'user/date
```


The `date` function will now return a new instance of `java.util.Calendar`. The next step would be to make a `defdate` macro so that a new `Calendar` instance can easily be bound to a symbol. This macro will also make sure that the binding is dynamic since the Java object's value can possibly change:

```
user> (defmacro defdate [-symbol]
      `(def ~(with-meta -symbol {:dynamic true}))
        (date)))
#'user/defdate
```

After expanding the macro to check whether it's producing the correct code, the `defdate` macro can be used to bind a symbol:

```
user> (macroexpand-1 '(defdate *today*))
(def *today* (user/date))
user> (defdate *today*)
#'user/*today*
user> (:dynamic (meta #'*today*))
true
user> *today*
#inst "2013-06-11T16:25:03.185+09:00"
```

Making Java objects easier to manipulate

So far, everything looks good but, in the following example, adding three more days to a copy of the date instance seems like a very verbose hassle.

```
user> (let [today (... *today* clone)
          days java.util.Calendar/DAY_OF_WEEK]
      (... today (add days 3))
      today)
#inst "2013-06-14T16:25:03.185+09:00"
```

Since `java.util.Calendar/DAY_OF_WEEK` is very verbose, the `week-day` symbol should represent this value for the remainder of the examples:

```
user> (def week-day java.util.Calendar/DAY_OF_WEEK)
#'user/week-day
```

The `days-later` function can also replace the verbose `let` form that adds three more days to a copy of the date instance.

```
user> (defn days-later [date n]
      (let [d (... date clone)
            action (... d (add week-day n))]
        d))
#'user/days-later
```

Now that all the verbose code has been simplified to a single function, adding additional days to any date should be as easy as using pure Clojure objects. Notice how the value of `*today*` hasn't changed after adding three additional days with the `days-later` function:

```
user> (days-later *today* 3)
#inst "2013-06-14T16:25:03.185+09:00"
user> *today*
#inst "2013-06-11T16:25:03.185+09:00"
```

If you were to pass a negative number to the `days-later` function, it would actually give you the days before that date. Another and, arguably, a nicer way to do this is to make a function called `days-before`. This function can change a positive value for the number of days into a negative value before calling `days-later`:

```
user> (defn days-before [date n]
      (days-later date (* -1 n)))
#'user/days-before
user> (days-before *today* 3)
#inst "2013-06-08T16:25:03.185+09:00"
user> *today*
#inst "2013-06-11T16:25:03.185+09:00"
```

Now that the `days-before` and `days-later` functions exist, we can get the modified dates without modifying the value of `*today*`:

```
user> (do (println *today*)
      (binding [*today* (days-before *today* 1)]
        (println *today*))
      (binding [*today* (days-later *today* 2)]
        (println *today*)))
#inst "2013-06-11T16:25:03.185+09:00"
#inst "2013-06-10T16:25:03.185+09:00"
#inst "2013-06-12T16:25:03.185+09:00"
```

Retrieving values in a better way

The last two examples of this section are about making object method calls simpler than they would be with Java interoperation. As you can see in the following code snippet, you wouldn't want to write this type of code every time you wanted to know the month of a given `Calendar` object:

```
user> (.. *today* (get java.util.Calendar/MONTH))
5
user> java.util.Calendar/JUNE
5
```

There needs to be a function that makes getting either the numeric or string value of the month simpler than what's currently available with just the Java interoperation syntax. The following `month-of` function will make getting these values much easier:

```
user> (defn month-of [date & [str?]]
      (let [month java.util.Calendar/MONTH]
        (if-not str?
          (inc (.. date (get month)))
          (.. date
              (getDisplayName
               month
               java.util.Calendar/LONG
               java.util.Locale/US))))))
#'user/month-of
user> (month-of *today*)
6
user> (month-of *today* true)
"June"
```

As you can see, simple functions go a long way when dealing with Java classes. You'll learn more about building on top of Java classes as you move on to making a small Twitter DSL.

Examples of our Twitter DSL

This section is a small collection of little programs that use the example DSL built in this chapter. These programs don't apply any type of error checks and aren't designed to be used in a real-world setting.

The Retweet bot

On Twitter, you'll commonly find an account that only retweets what other people say about a specific topic. This DSL allows us to make a retweet bot that retweets mentions of the Clojure language in just a few lines:

```
(use 'tdsl.core
    'tdsl.tweet
    'tdsl.search)

(deftwitter *account*)

(twitter-> *account*
  (query-> "Clojure"
    (doseq [t tweets]
      (tweet-> t (re-tweet))))
  :done))
```

Creating an event notifier

Twitter has proven that it's a great platform for finding out about events in almost real time. This little program sends a Tweet message when the application fails:

```
(use 'tdsl.core
      'tdsl.tweet)

(deftwitter *account*)

(defn notify [msg]
  (let [d (java.util.Date.)
        msg (format "%s %s" d msg)]
    (twitter-> *account* (send-tweet msg))))

(try
  (+ 1 \A)
  (catch Exception e
    (notify "Program Failed")))
```

Reading the OAuth configuration

Before being able to read an OAuth configuration file, we'll have to set up a Twitter account and get a valid set of API keys.

Twitter account registration and application keys

The steps to register an account on Twitter are as follows:

1. Go to <https://twitter.com/signup>.
2. Register and confirm your Twitter account.
3. Go to <https://dev.twitter.com>.
4. Sign in with your new Twitter account.
5. Go to <https://dev.twitter.com/apps/new>.
6. Complete the required form fields and click on **Create your Twitter application**.
7. Once the form has been completed, you should be redirected to your application's API page. Click on the **Settings** tab of your application's API page.
8. Scroll towards the bottom and set the application's access settings to **Read, Write, and Access direct messages**.
9. Click on **Update this Twitter application's settings** at the bottom of the settings page.

10. 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.
11. The information on your application's API page will allow your Clojure application to access your Twitter account, so don't give away this information. Specifically, don't give away the consumer key, consumer secret, access token, and access token secret.

Adding required dependencies

If you don't use the Leiningen Clojure project utility, you'll have to adapt the following steps to its equivalent in your development environment.

One of the easiest ways of accessing the latest Twitter API is to use an existing Java library. This project will depend on the `twitter4j` Java library, so we'll have to add the latest version to the project dependency list. The dependency name is `org.twitter4j/twitter4j-core` and the Version is `3.0.3`. After adding this to your project, your `project.clj` file should look similar to the following file:

File: `tdsl/project.clj`

```
(defproject tdsl "0.1.0-SNAPSHOT"
  :description "FIXME: write description"
  :url "http://example.com/FIXME"
  :license {:name "Eclipse Public License"
            :url "http://www.eclipse.org/legal/epl-v10.html"}
  :dependencies [[org.clojure/clojure "1.5.1"]
                [org.twitter4j/twitter4j-core "3.0.3"]])
```

After adding and saving the changes to the `project.clj` file, you can run the `lein deps` command-line interface command. The output of this command should resemble something similar to the following output if your local Maven repository doesn't already contain the dependency:

```
$ lein deps
Retrieving org/twitter4j/twitter4j-core/3.0.3/twitter4j-core-3.0.3.pom
from central
Retrieving org/twitter4j/twitter4j-core/3.0.3/twitter4j-core-3.0.3.jar
from central
```

Creating the project and API configuration

Using your system's command-line interface, create a new Clojure project named `tdsl`. If you don't use the Leiningen Clojure project utility, you'll have to adapt the following steps to what would be the equivalent in your development environment:

```
$ lein new tdsl
```

Generating a project called `tdsl` based on the 'default' template.

To see other templates (app, lein plugin, etc), try ``lein help new``.

In your project's classpath, with your favorite text editor, create a file named `api-config`. This file will hold the sensitive Twitter API information for the DSL:

```
$ emacs -nw api-config
```

The configuration file should be a line- and space-delimited configuration. Each line should hold only one configuration setting, and the settings key should be separated by one space from the settings value. The example output, shown as follows, should resemble your `api-config` file:

```
$ cat api-config
consumer-key xxx
consumer-secret xxx
access-token xxx
access-token-secret xxx
```

The information needed for this configuration file is located on the details page of your Twitter application API page. You can go to <https://dev.twitter.com/apps> to access the details page of your application.

Reading the Twitter configuration

In the `tdsl.core` namespace, we need to create a function called `load-config`. The following instructions and examples will be an incremental build of the `load-config` function. The `tdsl.core` namespace is located in the `tdsl/src/tdsl/core.clj` file

if you're using Leiningen to manage your Clojure project.

The first step for making our configuration's reading function is to define a multi-arity function named `load-config`. If no arguments are passed to the `load-config` function, the function will call itself with `api-config` as the configuration path argument:

```
tdsl.core> (defn load-config
            ([ ] (load-config "api-config"))
```

```
      ([config-path] config-path))
#'user/load-config
tdsl.core> (load-config)
"api-config"
tdsl.core> (load-config "abc")
"abc"
```

The next step is to read the lines of the configuration file; only one line can be read at a time. It's important to use the `with-open` form to read the configuration file because, once the file has been opened, the `with-open` form will close the file even if an error occurs:

```
tdsl.core> (defn load-config
             ([ ] (load-config "api-config"))
             ([config-path]
              (with-open [rdr (clojure.java.io/reader
                               (str config-path))]
                (doseq [ _ (line-seq rdr)] (println _)))))
#'user/load-config
tdsl.core> (load-config)
consumer-key xxx
consumer-secret xxx
access-token xxx
access-token-secret xxx
nil
```

Now we need to add a loop body that'll loop through each line of the configuration file and then append the settings to a map. Because each line has a set space delimited key value, each line must be split so that the first value of the split can be the key and the second value can be the value of the key. The key is then converted to a keyword data type to make the retrieval of the values easier. If the key is retained as a string, you will have to use the `get` form to retrieve the values:

```
tdsl.core> (defn load-config
             ([ ] (load-config "api-config"))
             ([config-path]
              (with-open [rdr (clojure.java.io/reader
                               (str config-path))]
                (loop [config {}]
                  (lines (line-seq rdr)]
                    (let [line (last lines)]
                      (if-not line
                        config
                        (let [[k v] (.. line (split " "))]]
                          (recur
```

```

                                (when (and k v)
                                  (assoc config (keyword k) v))
                                (drop-last lines)))))))))

#'user/load-config
tdsl.core> (clojure.pprint/pprint (load-config))
{:consumer-key "xxx",
 :consumer-secret "xxx",
 :access-token "xxx",
 :access-token-secret "xxx"}
nil

```

Everything seems to be working, so we need to add some form of error handling. In the example presented here, the error is caught and printed to the standard output in the final version of the `load-config` function, but it's up to you how the error should be handled:

```

tdsl.core> (defn load-config
  ([] (load-config "api-config"))
  ([config-path]
   (try
    (with-open [rdr (clojure.java.io/reader
                     (str config-path))]
      (loop [config {}]
        (lines (line-seq rdr))
        (let [line (last lines)]
          (if-not line
            config
            (let [[k v] (.. line (split " "))]]
              (recur
               (when (and k v)
                 (assoc config (keyword k) v))
               (drop-last lines)))))))
    (catch Exception ex
      (println
       (format "Error> load-config : [File %s] : %s"
               config-path
               (.getMessage ex))))))

#'user/load-config
tdsl.core> (load-config "abc")
Error> load-config : [File abc] : abc (No such file or directory)
nil

```


The `core.clj` file should look like the following after the final version of the `load-config` function has been written:

```
(ns tdsl.core)

(defn load-config
  ([] (load-config "api-config"))
  ([config-path]
   (try
    (with-open [rdr (clojure.java.io/reader
                     (str config-path))]
      (loop [config {}]
        (lines (line-seq rdr)]
          (let [line (last lines)]
            (if-not line
              config
              (let [[k v] (.. line (split " "))]]
                (recur
                 (when (and k v)
                   (assoc config (keyword k) v))
                 (drop-last lines)))))))
    (catch Exception ex
      (println
       (format "Error> load-config : [File %s] : %s"
               config-path
               (.getMessage ex)))))))
```

Making our most important macro

There are some macros that need to be made to help hide some of the complexities of our DSL. If it's not already open, open the `tdsl/src/tdsl/core.clj` file in your favorite Clojure code editor. The following code and explanations will be additions to the `core.clj` file.

The `deftwitter` macro will parse a Twitter configuration file, create a new Twitter instance with this configuration, and then bind the instance to a Clojure symbol. Before making this macro, let's see what the Java interoperation pattern would look like without this macro.

If you're using an already opened REPL session, depending on your development environment you'll have to close the session and open a new one so that the session can see the new dependency within `classpath`. In the REPL, you'll have to import two dependencies for this example. The following example will have to be evaluated in your REPL if you wish to follow it. If not, you can skip the REPL examples and move on to making the `deftwitter` macro within the `core.clj` file:

```
tdsl.core> (import '[twitter4j.conf ConfigurationBuilder]
               '[twitter4j TwitterFactory])
twitter4j.TwitterFactory
```

The `ConfigurationBuilder` class allows us to create the OAuth settings for the `TwitterFactory` class based on our API configuration file. The `TwitterFactory` class is required to create a new Twitter client instance. For more information on the two classes, refer to the Javadoc website of the `twitter4j` library located at <http://twitter4j.org/oldjavadocs/3.0.3/index.html>.

The configuration settings need to be configured before the `TwitterFactory` class can use it. The `twitter-auth` symbol represents the `ConfigurationBuilder` class that holds the configuration returned by the `load-config` function. Since we have neither a function nor a macro to do this for us, we will have to write the following code every time we need to configure the settings for a Twitter instance:

```
tdsl.core> (def twitter-auth
            (let [config (load-config)
                  {:keys [consumer-key
                          consumer-secret
                          access-token
                          access-token-secret]} config]
              (... (ConfigurationBuilder.)
                   (setPrettyDebugEnabled true)
                   (setUseSSL true)
                   (setOAuthConsumerKey consumer-key)
                   (setOAuthConsumerSecret consumer-secret)
                   (setOAuthAccessToken access-token)
                   (setOAuthAccessTokenSecret access-token-secret)
                   build)))

#'tdsl.core/twitter-auth
```

The next step would be to bind an authorized Twitter instance to a symbol with the `twitter-auth` credentials:

```
tdsl.core> (def my-account
             (... (TwitterFactory. twitter-auth)
                  getInstance))

#'tdsl.core/my-account
tdsl.core> (... my-account getScreenName)
"cljds1"
tdsl.core> (... my-account getId)
1389920761
```

Having to write this much code every time you wanted to bind a symbol to a Twitter instance would obviously be too tedious. This is where the `deftwitter` macro comes into play. This macro should come directly after the `load-config` function body.

Building the `deftwitter` macro

The first step is to modify the namespace by importing the required Twitter classes inside the `tdsl.core` namespace. The beginning of the `tdsl.core` file should look like the example code shown as follows:

```
(ns tdsl.core
  (:import [twitter4j TwitterFactory]
           [twitter4j.conf ConfigurationBuilder]))
```

The second step is to define a macro that can accept a symbol and an optional API configuration path:

```
tdsl.core> (defmacro deftwitter [-symbol & [config-path]]
  ;; body goes here
)
#'tdsl.core/deftwitter
```

This macro needs to build the Twitter configuration settings with or without a configuration path:

```
tdsl.core> (defmacro deftwitter [-symbol & [config-path]]
  (let [config (if-not config-path
                  (load-config)
                  (load-config config-path))]
    {:keys [consumer-key
            consumer-secret
            access-token]
```

```

        access-token-secret]} config
    auth (... (ConfigurationBuilder.)
             (setPrettyDebugEnabled true)
             (setUseSSL true)
             (setOAuthConsumerKey consumer-key)
             (setOAuthConsumerSecret consumer-secret)
             (setOAuthAccessToken access-token)
             (setOAuthAccessTokenSecret access-token-secret)
             build)]
    ;; body here
  ))
  #'tdsl.core/deftwitter

```

The last step in building this macro is to make an instance with the API configuration using the macro and binding the instance to a symbol:

```

tdsl.core> (defmacro deftwitter [-symbol & [config-path]]
  (let [config (if-not config-path
                  (load-config)
                  (load-config config-path))
        {:keys [consumer-key
                 consumer-secret
                 access-token
                 access-token-secret]} config]
    auth (... (ConfigurationBuilder.)
              (setPrettyDebugEnabled true)
              (setUseSSL true)
              (setOAuthConsumerKey consumer-key)
              (setOAuthConsumerSecret consumer-secret)
              (setOAuthAccessToken access-token)
              (setOAuthAccessTokenSecret access-token-secret)
              build)]
      `(def ~(with-meta -symbol {:dynamic true})
         (... (TwitterFactory. ~auth) getInstance))))
  #'tdsl.core/deftwitter

```

After the macro is completed, your `core.clj` file should look like the following code:

File: `tdsl/src/tdsl.core`

```

(ns tdsl.core
  (:import [twitter4j TwitterFactory]
            [twitter4j.conf ConfigurationBuilder]))

```

```
(defn load-config
  ([] (load-config "api-config"))
  ([config-path]
    (try
      (with-open [rdr (clojure.java.io/reader
                      (str config-path))]
        (loop [config {}]
          (lines (line-seq rdr))
          (let [line (last lines)]
            (if-not line
              config
              (let [[k v] (.. line (split " "))]
                (recur
                 (when (and k v)
                   (assoc config (keyword k) v))
                 (drop-last lines)))))))
      (catch Exception ex
        (println
         (format "Error> load-config : [File %s] : %s"
                  config-path
                  (.getMessage ex)))))))

(defmacro deftwitter [-symbol & [config-path]]
  (let [config (if-not config-path
                  (load-config)
                  (load-config config-path))
        {:keys [consumer-key
                  consumer-secret
                  access-token
                  access-token-secret]} config]
    `auth (.. (ConfigurationBuilder.)
               (setPrettyDebugEnabled true)
               (setUseSSL true)
               (setOAuthConsumerKey consumer-key)
               (setOAuthConsumerSecret consumer-secret)
               (setOAuthAccessToken access-token)
               (setOAuthAccessTokenSecret access-token-secret)
               build)
      `(def ~(with-meta -symbol {:dynamic true})
        (.. (TwitterFactory. ~auth) getInstance))))
```

If done correctly, you should be able to run the following commands in your REPL without an error:

```
tdsl.core> (deftwitter *instance*)
#'tdsl.core/*instance*
tdsl.core> (:dynamic (meta #'*instance*))
true
tdsl.core> (.. *instance* getScreenName)
"cljds1"
tdsl.core> (.. *instance* getId)
1389920761
```

Building the twitter-> macro

The `twitter->` macro will accept the first argument as the Twitter instance and bind the instance to the `tdsl.core/*twitter*` global variable. The `*twitter*` variable will be used by the additional arguments passed to the `twitter->` macro. This is done so that the functions that use the `*twitter*` variable can call the Java methods on a `deftwitter` instance regardless of the symbol name.

This macro should go above the `load-config` function in the `core.clj` file. The macro is actually very short and will evaluate all the arguments after the first argument as a body of code:

```
(def ^:dynamic *twitter*)
(def ^:dynamic *tweets*)

(defmacro twitter-> [deftwitter-instance & body]
  `(binding [*twitter* ~deftwitter-instance] ~@body))
```

If done correctly, you should be able to run the following commands in your REPL without an error:

```
tdsl.core> (deftwitter *instance*)
#'tdsl.core/*instance*
tdsl.core> (twitter-> *instance*
                    (... *twitter* getScreenName))
"cljds1"
```

Now the `core.clj` file should look like the following code:

```
(ns tdsl.core
  (:import [twitter4j TwitterFactory]
           [twitter4j.conf ConfigurationBuilder]))
```

```
(def ^:dynamic *twitter*)
(def ^:dynamic *tweets*)

(defmacro twitter-> [deftwitter-instance & body]
  `(binding [*twitter* ~deftwitter-instance] ~@body))

(defn load-config
  ([] (load-config "api-config"))
  ([config-path]
   (try
    (with-open [rdr (clojure.java.io/reader
                     (str config-path))]
      (loop [config {}
             lines (line-seq rdr)]
        (let [line (last lines)]
          (if-not line
              config
              (let [[k v] (.. line (split " "))]
                (recur
                 (when (and k v)
                   (assoc config (keyword k) v))
                 (drop-last lines)))))))
    (catch Exception ex
      (println
       (format "Error> load-config : [File %s] : %s"
               config-path
               (.getMessage ex)))))))

(defmacro deftwitter [-symbol & [config-path]]
  (let [config (if-not config-path
                  (load-config)
                  (load-config config-path))
        {:keys [consumer-key
                 consumer-secret
                 access-token
                 access-token-secret]} config]
    auth (.. (ConfigurationBuilder.)
              (setPrettyDebugEnabled true)
              (setUseSSL true)
              (setOAuthConsumerKey consumer-key)
              (setOAuthConsumerSecret consumer-secret)
              (setOAuthAccessToken access-token)
              (setOAuthAccessTokenSecret access-token-secret)
              build)])
  `(def ~(with-meta -symbol {:dynamic true})
     (.. (TwitterFactory. ~auth) getInstance))))
```

Handling search queries

For brevity, this section will only cover retrieving the username, screen name, date, and text of the search results. Please refer to the documentation of the `twitter4j` library for more information. The documentation page for the `QueryResult` class is located at <http://twitter4j.org/oldjavadocs/3.0.3/twitter4j/QueryResult.html>.

Adding the `tdsl.search` namespace

The search capabilities should go in their own namespace, so you'll need to create a new file at `tdsl/src/tdsl/search.clj`. The namespace of this file should use two dynamic variables from the `tdsl.core` namespace, namely `*twitter*` and `*tweets*`. The namespace also needs to import the `Query` class from the `twitter4j` Java library. If done correctly, the beginning of the file should be similar to the REPL session shown as follows:

```
tdsl.core> (ns tdsl.search
            (:use [tdsl.core
                  :only [*twitter*
                        *tweets*]])
            (:import [twitter4j Query]))

nil
tdsl.search>
```

Search macros and functions

Because a DSL is used to hide but not remove the complexity of a problem, we need to define a macro that makes the most commonly accessed values the easiest to access. This means we need to define a macro that makes it easy to search and get details of the search results. This macro should be called `query->`, and it should provide the locally scoped variables, `query`, `query-count`, `next-page?`, and `tweets`.

The `query->` macro should only accept a `String` as the first argument and then bind the results of the search call to the `*tweets*` dynamic variable. The remaining arguments should be treated as a code body that needs to access the `*tweets*` dynamic variable. The `query->` macro should look like the following REPL session:

```
tdsl.search> (defmacro query->
              "Local bindings: query, query-count, next-page?, tweets"
              [^String term & body]
              {:pre (string? term)})
  `(let [_# (. *twitter* (search (Query. ~term)))
        ~'query _#
        ~'query-count (. _# getCount)
```



```
~'next-page? (.. _# hasNext)
~'tweets (.. _# getTweets)]
(or (binding [*tweets* ~'tweets] ~@body)
~'query)))

#'tdsl.search/query->
```

If done correctly, you should be able to run the following commands in a REPL session without an error:

```
tdsl.search> (use 'tdsl.core)
nil
tdsl.search> (deftwitter *my-account*)
#'tdsl.search/*my-account*
tdsl.search> (twitter->
  *my-account*
  (query-> "Clojure"
    (format "There are %d Tweets and%sanother page"
      query-count
      (if next-page? " " " not "))))
```

The output will look similar to the following:

```
"There are 15 Tweets and another page"
```

If everything works fine, we will make another macro and some functions that will retrieve data from every single search result that is returned. This macro is very simple and will make it easier for other functions to retrieve the search data:

```
tdsl.search> (defmacro from-tweet [memfn-call]
  `(map (memfn ~memfn-call) *tweets*))
#'tdsl.search/from-tweet
```

The following functions will successfully call the `from-tweet` macro as long as it's called within the body of the `query->` macro. The `text` function is the first of these functions to be called:

```
tdsl.search> (defn text [] (from-tweet getText))
#'tdsl.search/text
```

The `text` function obviously retrieves the message of every Tweet but the next function isn't as straightforward. The `user` function, which is shown as follows, returns `PersistentArrayMap` containing a username and screen name for each of the returned Tweets. Vectors are used to maintain the order of the items in the collection:

```
tdsl.search> (defn user []
  (let [_ (from-tweet getUser)]
    (mapv (partial zipmap [:user :screen-name])
```

```

        (mapv #(vec [(getName %)
                     (getScreenName %)])
              _)))

#'tdsl.search/user

```

The last function we need is the date function. Instead of returning a collection of `java.util.Date` objects, this function will return a collection of `Strings`:

```

tdsl.search> (defn date [] (map str (from-tweet getCreatedAt)))
#'tdsl.search/date

```

If done correctly, you should be able to run the following commands in a REPL session without an error:

```

tdsl.search> (use 'tdsl.core)
nil
tdsl.search> (deftwitter *my-account*)
#'tdsl.search/*my-account*
tdsl.search> (clojure.pprint/pprint
              (twitter-> *my-account*
                        (query-> "functional clojure"
                                (let [d (first (date))
                                      u (-> (user) first :user)
                                      m (first (text))
                                      n (subs m 0 30)]
                                  (format "<%s> %s - %s" u d m))))))

"<Overtone> Sun Jun 16 21:02:44 JST 2013 - RT @rickerbh: Learning
Clojure"
nil

```

Although the previously mentioned functions make things a bit easier, it's still not easy enough. Another macro will be needed to make this process much easier. This macro is called `from-query`.

The `from-query` macro will accept symbols since they are also functions. These symbols will be cast to a `String` datatype, so they can be made into keywords in a map. The keywords will be used as keys for the Tweet data. The value of each key will be the function's returned object with the same name as the key. For example, `{:text "xyz"}` is the result of `{:text (text tweet)}`:

```

tdsl.search> (defmacro from-query
              "Types: text, user, date"
              [& types]
              `(loop [_# [_# (mapv keyword ~(mapv str types))
                          return# (hash-map)]

```

```
(let [t# (last _#)]
  (if-not t#
    (let [k# (keys return#)
          values# (for [-k# k#]
                     (map #(hash-map -k# %)
                           (-k# return#)))]
      [size# (count k#)
        (if (= size# 1)
          (first values#)
          (map #(apply merge %)
                (partition size#
                           (apply interleave values#))))])
    (let [type-to-fn# (-> t# name str symbol list)]
      (recur
        (drop-last _#)
        (assoc return# t# (eval type-to-fn#)))))))
```

If done correctly, you should be able to run the following commands in a REPL session without an error:

```
tdsl.search> (use 'tdsl.core)
nil
tdsl.search> (deftwitter *my-account*)
#'tdsl.search/*my-account*
tdsl.search> (clojure.pprint/pprint
  (twitter->
    *my-account*
    (query->
      "Clojure is"
      (for [tweet (take 2 (from-query text date))
            :let [{:keys [text date]} tweet]]
        (format "%s> %s" date (subs text 0 30))))))
("Sun Jun 16 22:49:05 JST 2013> @pcalcado Seeing Node and Cloj"
 "Sun Jun 16 19:32:59 JST 2013> From a clojure view scheme loo")
nil
```

The search.clj file should now look like the following code:

```
(ns tdsl.search
  (:use [tdsl.core
        :only [*twitter*
               *tweets*]])
  (:import [twitter4j Query]))

(defmacro query->
```

```

"Local bindings: query, query-count, next-page?, tweets"
[^String term & body]
{:pre (string? term)}
`(let [_# (... *twitter* (search (Query. ~term)))
      ~'query _#
      ~'query-count (... _# getCount)
      ~'next-page? (... _# hasNext)
      ~'tweets (... _# getTweets)]
  (or (binding [*tweets* ~'tweets] ~@body)
      ~'query)))

(defmacro from-tweet [memfn-call]
  `(map (memfn ~memfn-call) *tweets*))

(defn text [] (from-tweet getText))

(defn user []
  (let [_ (from-tweet getUser)]
    (mapv (partial zipmap [:user :screen-name])
          (mapv #(vec [(getName %)
                      (getScreenName %)])
                _))))

(defn date [] (map str (from-tweet getCreatedAt)))

(defmacro from-query
  "Types: text, user, date"
  [& types]
  `(loop [_# (mapv keyword ~(mapv str types))
        return# (hash-map)]
    (let [t# (last _#)]
      (if-not t#
        (let [k# (keys return#)
              values# (for [-k# k#]
                        (map #(hash-map -k# %)
                             (-k# return#)))
              size# (count k#)]
          (if (= size# 1)
            (first values#)
            (map #(apply merge %)
                  (partition size#
                             (apply interleave values#))))))
        (let [type-to-fn# (-> t# name str symbol list)]
          (recur
            (drop-last _#)
            (assoc return# t# (eval type-to-fn#)))))))

```

Handling tweets

For brevity, this section will only cover sending, deleting, unfavoriting/favoriting, and retweeting messages. Please refer to the documentation of the `twitter4j` library for more information. The documentation pages for the `Status`, `TweetsResources`, and `FavoritesResources` classes are located at <http://twitter4j.org/oldjavadocs/3.0.3/twitter4j/Status.html>, <http://twitter4j.org/oldjavadocs/3.0.3/twitter4j/api/TweetsResources.html>, and <http://twitter4j.org/oldjavadocs/3.0.3/twitter4j/api/FavoritesResources.html>.

Adding the `tdsl.tweet` namespace

The Tweet handling capabilities should go in their own namespace, so you'll need to create a new file at `tdsl/src/tdsl/tweet.clj`. The namespace of this file should use the `*twitter*` variable from the `tdsl.core` namespace. The namespace also needs to import the `Status` class from the `twitter4j` Java library. If done correctly, the beginning of the file should be similar to the REPL session as follows:

```
user> (ns tdsl.tweet
      (:use [tdsl.core
             :only [*twitter*]))
      (:import [twitter4j Status]))
nil
```

Tweet macros and functions

The first function for this namespace will allow a Twitter instance to send a Tweet. The `send-tweet` function will take `String` and send the message if the API keys of the Twitter instance are authorized with write permissions:

```
tdsl.tweet> (defn send-tweet [^String msg]
              (... *twitter* (updateStatus msg)))
#'tdsl.tweet/send-tweet
```

The first of the two macros in this namespace is called `tweet->`. The `tweet->` macro takes a Twitter `Status` object and binds the dynamic variable `*tweet*` with the `Status` object. All arguments after the first should be a body of code, which need access to the `*tweet*` binding:

```
tdsl.tweet> (def ^:dynamic *tweet*)

(defmacro tweet-> [^Status tweet & body]
  `(binding [*tweet* ~tweet] ~@body))

#'tdsl.tweet/*tweet*
#'tdsl.tweet/tweet->
```

The second and final macro in this namespace is called `status-do`. This macro takes a Java class method name and tells the current Twitter instance to call the method with the message ID of `*tweet*` as the first and only argument. The message ID of `*tweet*` comes from the `tweet-id` function:

```
tdsl.tweet> (defn tweet-id [] (.. *tweet* getId))

(defmacro status-do [call]
  `(.. *twitter*
      (~call (tweet-id))))

#'tdsl.tweet/tweet-id
#'tdsl.tweet/status-do
```

The `status-do` macro now allows us to easily make functions that apply to the `*tweet*` binding. The first of the many functions is the `delete` function. This function simply passes the `destroyStatus` method call to the `status-do` macro to remove a Tweet message owned by the `*twitter*` instance:

```
tdsl.tweet> (defn delete []
              (status-do destroyStatus))
#'tdsl.tweet/delete
```

The `retweet` is just as easy to build. The body of the function simply passes the `retweetStatus` method call to the `status-do` macro.

```
tdsl.tweet> (defn re-tweet []
              (status-do retweetStatus))
#'tdsl.tweet/re-tweet
```

The `favorite?` function doesn't use the `status-do` macro because the method required to test whether the Tweet message is a favorite doesn't accept any arguments:

```
tdsl.tweet> (defn favorite? []
              (.. *tweet* isFavorited))
#'tdsl.tweet/favorite?
```

The `toggle-favorite` function will call the `favorite?` function to test what action it should take next. If the `Status` object is already a favorite of the Twitter instance, the `destroyFavorite` method call is passed to the `status-do` macro. If not, the `createFavorite` method call is passed to the `status-do` macro:

```
(defn toggle-favorite []
  (if-not (favorite?)
    (status-do createFavorite)
    (status-do destroyFavorite)))
#'tdsl.tweet/toggle-favorite
```

If done correctly, you should be able to run the following commands in a REPL session without an error:

```
tdsl.tweet> (use 'tdsl.core 'tdsl.search)
nil
tdsl.tweet> (deftwitter *my-account*)
#'tdsl.tweet/*my-account*
tdsl.tweet> (twitter->
  *my-account*
  (query->
    "Clojure"
    (let [tweet-obj (send-tweet "Testing")]
      (tweet-> tweet-obj
        (println "Fav?" (favorite?))
        (tweet-> (toggle-favorite)
          (println "Fav?" (favorite?)))
        (delete)
        (println "Removed")
        (tweet-id))))))

Fav? false
Fav? true
Removed
347292424776859648
tdsl.tweet> (twitter->
  *my-account*
  (query->
    "Clojure"
    (let [msg-coll (take 2 (from-query text))
          msg-max (count msg-coll)]
      (doseq [_ (range msg-max)
        :let [x (:text (nth msg-coll _))
              msg (apply str (take 30 x))
              tweet-obj (nth tweets _)]
          (tweet-> tweet-obj (println (tweet-id) msg))
        :done))))

347280477473886208 Macro protocols for Clojure wr
347280446658342912 @mzp Clojureならそんなの不要です
:done
```

The twitter.clj file should now look like the following code:

```
(ns tdsl.tweet
  (:use [tdsl.core
        :only [*twitter*]])
  (:import [twitter4j Status]))
```

```
(defn send-tweet [^String msg]
  (... *twitter* (updateStatus msg)))

(def ^:dynamic *tweet*)

(defmacro tweet-> [^Status tweet & body]
  `(binding [*tweet* ~tweet] ~@body))

(defn tweet-id [] (... *tweet* getId))

(defmacro status-do [call]
  `(... *twitter*
    (~call (tweet-id))))

(defn delete []
  (status-do destroyStatus))

(defn re-tweet []
  (status-do retweetStatus))

(defn favorite? []
  (... *tweet* isFavorited))

(defn toggle-favorite []
  (if-not (favorite?)
    (status-do createFavorite)
    (status-do destroyFavorite)))
```

Adding user-related features

For brevity, this section will only cover how to retrieve some of the publicly available information of a Twitter user. Please refer to the documentation of the `twitter4j` library for more information. The documentation pages for the `Status`, `User`, and `UsersResources` classes are located at <http://twitter4j.org/oldjavadocs/3.0.3/twitter4j/Status.html>, <http://twitter4j.org/oldjavadocs/3.0.3/twitter4j/User.html>, and twitter4j.org/oldjavadocs/3.0.3/twitter4j/api/UsersResources.html.

Adding the `tdsl.user` namespace

You'll need to create a new file at `tdsl/src/tdsl/user.clj`. The namespace of this file should use the `*twitter*` variable from the `tdsl.core` namespace. The namespace also needs to import the `Status` class from the `twitter4j` Java library. If done correctly, the beginning of the file should be similar to the REPL session shown as follows:

```
tdsl.user> (ns tdsl.user
             (:use [tdsl.core
                    :only [*twitter*]])
             (:import [twitter4j Status]))

nil
```

User macros and functions

The first function for this namespace is called `user-obj`. This function is used to return the `User` class object based on a Twitter ID. This function is needed for the first macro for this namespace. This macro is called `user-attr` and is used to call a Java method with no arguments on the provided user ID:

```
tdsl.user> (defn user-obj [id]
             (... *twitter* (showUser id)))

(defmacro user-attr [id method]
  `(... (user-obj ~id) ~method))
#'tdsl.user/user-obj
#'tdsl.user/user-attr
```

The `user-attr` macro is used by the `id->user` and `user->id` functions to easily retrieve the username and ID of a Twitter user. The `user->id` function will convert a Twitter String ID to a Twitter Long ID and the `id->user` function will do the opposite:

```
tdsl.user> (defn id->user [^Long id]
             (user-attr id getScreenName))

(defn user->id [^String name]
  (user-attr name getId))
#'tdsl.user/id->user
#'tdsl.user/user->id
```

If done correctly, you should be able to run the following commands in the REPL session without an error:

```
tdsl.user> (use 'tdsl.core)
nil
tdsl.user> (deftwitter *account*)
#'tdsl.user/*account*
tdsl.user> (twitter-> *account*
              (let [uid (user->id "cljds1")
                    uname (id->user uid)]
                (format "%s = %s" uname uid)))

"cljds1 = 1389920761"
```

User details and multimethods

The first and only multimethod in this namespace is called `user-details`. The `user-details` multimethod will be able to retrieve Twitter user information from three different data types. These types are `Long`, `String`, and the `Status` classes from the `twitter4j` library:

```
tdsl.user> (defmulti user-details class)
#'tdsl.user/user-details
```

The first `defmethod` for `user-details` will be for the `Long` class. The first argument is the Twitter user id. This `defmethod` will use the `user-attr` macro to make method calls on the provided user id, and the username of the Twitter account will be retrieved with the `id->user` function.

The second `defmethod` for `user-details` is for the `String` class and it behaves the same as the `Long` method handler; however, the `user->id` function is used to get the username:

```
tdsl.user> (defmethod user-details Long [user-id]
              (let [uname (id->user user-id)
                    _ user-id]
                {:id _
                 :user uname
                 :created-at (user-attr _ getCreatedAt)
                 :language (user-attr _ getLang)
                 :friends-count (user-attr _ getFriendsCount)
                 :followers-count (user-attr _ getFollowersCount)})))

(defmethod user-details String [username]
  (let [id (user->id username)
```

```
      _ id]
    {:id _
     :user username
     :created-at (user-attr _ getCreatedAt)
     :language (user-attr _ getLang)
     :friends-count (user-attr _ getFriendsCount)
     :followers-count (user-attr _ getFollowersCount)}})

#<MultiFn clojure.lang.MultiFn@3de1ad91>
#<MultiFn clojure.lang.MultiFn@3de1ad91>
```

If done correctly, you should be able to run the following commands in the REPL session without an error:

```
tdsl.user> (use 'tdsl.core)
nil
tdsl.user> (deftwitter *account*)
#'tdsl.user/*account*
tdsl.user> (twitter-> *account*
                     (let [{:keys [id]} (user-details
                                           (user->id "cljdsl"))]
                       (println "Looking up details with id" id)
                       (clojure.pprint/pprint
                        (user-details (id->user id))))))
```

Looking up details with the ID 1389920761:

```
{:id 1389920761,
 :user "cljdsl",
 :created-at #inst "2013-04-29T16:41:19.000-00:00",
 :language "en",
 :friends-count 21,
 :followers-count 1}
nil
```

The last defmethod for user-details doesn't actually accept a user ID, but it accepts a Tweet Status message. This comes in handy when you need user information based on a message you found in a search query:

```
tdsl.user> (defmethod user-details Status [tweet]
            (let [id (... tweet getUser getId)
                  _ id]
              {:id _
               :user (id->user _)
               :created-at (user-attr _ getCreatedAt)}
```

```

:language (user-attr _ getLang)
:friends-count (user-attr _ getFriendsCount)
:followers-count (user-attr _ getFollowersCount)))))

```

```
#<MultiFn clojure.lang.MultiFn@3de1ad91>
```

If done correctly, you should be able to run the following commands in the REPL session without an error:

```

tdsl.user> (use 'tdsl.core 'tdsl.search)
nil
tdsl.user> (deftwitter *account*)
#'tdsl.user/*account*
tdsl.user> (clojure.pprint/pprint
            (twitter-> *account*
                      (query-> "Clojure"
                              (user-details (first tweets)))))

{:id 29231624,
 :user "stofke72",
 :created-at #inst "2009-04-06T16:15:02.000-00:00",
 :language "en",
 :friends-count 102,
 :followers-count 65}
nil

```

The `user.clj` file should now look like the following code:

```

(ns tdsl.user
  (:use [tdsl.core
        :only [*twitter*]])
  (:import [twitter4j Status]))

(defn user-obj [id]
  (.. *twitter* (showUser id)))

(defmacro user-attr [id method]
  `(.. (user-obj ~id) ~method))

(defn id->user [^Long id]
  (user-attr id getScreenName))

(defn user->id [^String name]
  (user-attr name getId))

(defmulti user-details class)

```

```
(defmethod user-details Long [user-id]
  (let [uname (id->user user-id)
        _ user-id]
    {:id _
     :user uname
     :created-at (user-attr _ getCreatedAt)
     :language (user-attr _ getLang)
     :friends-count (user-attr _ getFriendsCount)
     :followers-count (user-attr _ getFollowersCount)}))

(defmethod user-details String [username]
  (let [id (user->id username)
        _ id]
    {:id _
     :user username
     :created-at (user-attr _ getCreatedAt)
     :language (user-attr _ getLang)
     :friends-count (user-attr _ getFriendsCount)
     :followers-count (user-attr _ getFollowersCount)}))

(defmethod user-details Status [tweet]
  (let [id (.. tweet getUser getId)
        _ id]
    {:id _
     :user (id->user _)
     :created-at (user-attr _ getCreatedAt)
     :language (user-attr _ getLang)
     :friends-count (user-attr _ getFriendsCount)
     :followers-count (user-attr _ getFollowersCount)}))
```

Adding logging features

The logging features can be found in the `tdsl.log` namespace. This new file should be located at `tdsl/src/tdsl/log.clj` and the namespace of this file should use the `*tweets*` variable from the `tdsl.core` namespace:

```
tdsl.log> (ns tdsl.log
            (:use [tdsl.core
                  :only [*tweets*]]))

nil
```

For simple logging capabilities, you'll actually only need one macro to log all the Twitter messages bound to the `*tweets*` variable. This macro is called `log->` and it requires a minimum of two arguments. The first is the path and the second is the function that decides what data to log. All arguments after the first two are treated as a single body of code that needs access to the `*tweets*` variable:

```
tdsl.log> (defmacro log->
  "Can be used after *tweets* has been bound.
  log-fn returns write data and accepts 1 arg."
  [path log-fn & body]
  `(do (doseq [_# *tweets*]
    (spit ~path
      (format "%s\n"
        (~log-fn _#))
      :append true))
    ~@body))

#'tdsl.log/log->
```

If done correctly, you should be able to run the following commands in the REPL session without an error:

```
tdsl.log> (use 'tdsl.core
  'tdsl.search
  'tdsl.user)

nil
tdsl.log> (deftwitter *myuser*)
#'tdsl.log/*myuser*
tdsl.log> (twitter-> *myuser*
  (query-> "Clojure"
    (log-> "/tmp/out.txt"
      (fn [tweet] (user-details tweet))
      (println "Logged all found."))
    :done))
```

The output should look like the following:

```
Logged all found.
:done
```

If no errors have occurred, you can read the data with the `split-lines` and `load-string` functions:

```
tdsl.log> (def logged-data
           (clojure.string/split-lines
            (slurp "/tmp/out.txt")))
#'tdsl.log/logged-data
tdsl.log> (doseq [line (map load-string
                           (take 2 logged-data))]
          (clojure.pprint/pprint line))
{:language "en",
 :friends-count 102,
 :followers-count 65,
 :created-at #inst "2009-04-06T16:15:02.000-00:00",
 :user "stofke72",
 :id 29231624}
{:language "en",
 :friends-count 319,
 :followers-count 204,
 :created-at #inst "2009-02-04T20:41:09.000-00:00",
 :user "pyrtsa",
 :id 20090803}
nil
```

The `log.clj` file should now look like the following code:

```
(ns tdsl.log
  (:use [tdsl.core
        :only [*tweets*]]))

(defmacro log->
  "Can be used after *tweets* has been bound.
  log-fn returns write data and accepts 1 arg."
  [path log-fn & body]
  `(do (doseq [_# *tweets*]
        (spit ~path
              (format "%s\n"
                      (~log-fn _#))
              :append true))
    ~@body))
```

Summary

Hopefully, this chapter has helped you improve your understanding of the power of Clojure and how flexible it can be. Although the most powerful aspects of the Clojure language weren't used in the example DSL, much of the simplicity and expressiveness of Clojure was displayed.

As time moves on, there will be more pure Clojure implementations of libraries and DSLs. Clojure is still a young language; so, many libraries and DSLs will have to rely on Java libraries as long as time is the key factor in the development process.

The next chapter explores several unit-testing libraries by writing unit tests for our Twitter DSL.

10

Unit Testing

Unit testing has become a popular method of determining the correctness of a portion of code. These tests are built during the development process and help isolate the correct and incorrect code behaviors of a program. Specifically, in test-driven development, the tests are written first, and then the code is written to pass the tests.

Although the Clojure language is still very young, there are many unit testing frameworks that help segregate the tests from the development source code. This chapter will briefly cover the `clojure.test`, `expectations`, `midje`, and `speclj` unit testing frameworks. Each framework will be used to perform tests on the Twitter DSL created in the last chapter.

This chapter covers some of the basics of how to use the following unit testing frameworks:

- `clojure.test`
- `expectations`
- `midje`
- `speclj`

Exploring the `clojure.test` framework

The `clojure.test` framework is the default Clojure unit testing framework that comes with the Clojure standard library. This section will focus on writing tests for some of the existing functions of the Twitter DSL. The full API documentation for the framework is located at <http://clojure.github.io/clojure/clojure.test-api.html>.

Testing tdsl.core

If you used the Leiningen project utility to create the `tdsl` project in the last chapter, a core testing file already exists within `tdsl/test/tdsl/` directory:

```
$ cd tdsl; ls
doc/  resources/  src/  target/  test/  api-config  api-config~  project.clj  README.md
```

```
$ tree test/
test/
├── tdsl
│   └── core_test.clj
```

1 directory, 1 file

As you can see, the default core test file requires all the public symbols from the `tdsl.core` and `clojure.test` namespaces using `require`. If you're not using Leiningen, you can create the `core_test.clj` file at `tdsl/test/tdsl/core_test.clj`, and paste the following code:

```
(ns tdsl.core-test
  (:require [clojure.test :refer :all]
            [tdsl.core :refer :all]))

(deftest a-test
  (testing "FIXME, I fail."
    (is (= 0 1))))
```

If you're not using the Leiningen utility, you can skip this instruction. If you are, you can run this test with the `lein test` command, as shown in the following terminal output:

```
$ lein test
lein test tdsl.core-test
lein test :only tdsl.core-test/a-test
FAIL in (a-test) (core_test.clj:7)
  FIXME, I fail.
expected: (= 0 1)
actual: (not (= 0 1))

Ran 1 tests containing 1 assertions
1 failures, 0 errors.
Tests failed.
```

If you're not using Leiningen, you can enter the testing namespace with a REPL session, and then run `(clojure.test/test-ns 'tdsl.core-test)`:

```
tdsl.core-test> (clojure.test/test-ns 'tdsl.core-test)

Testing tdsl.core-test

FAIL in (a-test) (core_test.clj:7)
FIXME, I fail.
expected: (= 0 1)
actual: (not (= 0 1))
{:pass 0, :test 1, :error 0, :fail 1}
```

Using the `is` macro

The `clojure.test` framework uses the `is` macro to assert facts. If an optional message is supplied to the `is` macro as the second argument, this message will appear if and when the test fails.

The first test to write for `tdsl.core` will test if the `twitter->` macro successfully binds the `*twitter*` dynamic variable to the first argument. We can test this by passing the first argument to the `twitter->` macro and then passing the `*twitter*` dynamic variable as the code body, as shown in the following code snippet:

```
(deftest testing-twitter->
  (testing "twitter-> Macro working?"
    (is (= 1 (twitter-> 1 *twitter*))
        "Can't bind *twitter*"))))
```

If everything was done correctly, your test results should look similar to the following output:

```
$ lein test

lein test tdsl.core-test

Ran 1 tests containing 1 assertions
0 failures, 0 errors
```

Using the are macro

The `are` macro acts as a test template to run multiple values against the same test. For example, instead of writing multiple tests with the `is` macro, you can replace repetitive patterns with the `are` macro. If you look at the examples, the same tests are written using the `is` and `are` macros. Which one looks better for you to write?

```
(deftest xyz-is
  (testing "is macro"
    (is (= 1 1))
    (is (= 2 2))
    (is (= 3 4))))
```

```
(deftest xyz-are
  (testing "are macro"
    (are [x y] (= x y)
      1 1
      2 2
      3 4)))
```

Using the `are` macro, we can write a test for the `load-config` function of the `tdsl.core` namespace. This test will check to see if all the configuration keys are available. If any part of this test fails, it means that the configuration file isn't properly formatted.

```
(deftest testing-load-config-keys
  (let [cfg (load-config)
        cfg-ks (keys cfg)]
    (testing "load-config Has all keys?"
      (are [k] (true? (some #(= % k) cfg-ks))
        :consumer-key
        :consumer-secret
        :access-token
        :access-token-secret)))))
```

Having the proper keys doesn't mean you have the proper values, so we'll have to add another test to ensure that the values are properly formatted. Using the `are` macro again, we need to check that the values match a regular expression. The `:access-token` keyword has a dash in the value, so the regular expression will be different when the `k` binding matches the `:access-token` key:

```
(deftest testing-load-config-keys
  (let [cfg (load-config)
        cfg-ks (keys cfg)]
    (testing "load-config Has all keys?"
      (are [k] (true? (some #(= % k) cfg-ks))
```

```

      :consumer-key
      :consumer-secret
      :access-token
      :access-token-secret))
(testing "load-config Has valid values?"
  (are [k] (if (= k :access-token)
    (string? (re-find #"^\d+[-]\w+$" (k cfg)))
    (string? (re-find #"^\w+$" (k cfg))))
    :consumer-key
    :consumer-secret
    :access-token
    :access-token-secret))))

```

If everything was done correctly, your test results should look similar to the following output:

```
$ lein test
```

```
lein test tdsl.core-test
```

```
Ran 2 tests containing 9 assertions.
```

```
0 failures, 0 errors.
```

Developing the final test

As you can see, the `clojure.test` framework is very straightforward to use, but there's one last test to write to completely test the `tdsl.core` namespace. This test should test the `deftwitter` macro by making sure the symbol defined is an instance of the `twitter4j.TwitterImpl` class, as shown in the following code snippet:

```

(deftwitter *account*)

(deftest testing-deftwitter
  (is (= (class *account*)
    twitter4j.TwitterImpl)))

```

If everything was done correctly, your `tdsl/test/tdsl/core_test.clj` file should look similar to the following code:

```

(ns tdsl.core-test
  (:require [clojure.test :refer :all]
    [tdsl.core :refer :all]))

(deftest testing-twitter->

```

```
(testing "twitter-> Macro working?"
  (is (= 1 (twitter-> 1 *twitter*))
    "Can't bind *twitter*"))

(deftest testing-load-config-keys
  (let [cfg (load-config)
        cfg-ks (keys cfg)]
    (testing "load-config Has all keys?"
      (are [k] (true? (some #(= % k) cfg-ks))
        :consumer-key
        :consumer-secret
        :access-token
        :access-token-secret))
    (testing "load-config Has valid values?"
      (are [k] (if (= k :access-token)
        (string? (re-find #"^\\d+[-]\\w+$" (k cfg)))
        (string? (re-find #"^\\w+$" (k cfg))))
        :consumer-key
        :consumer-secret
        :access-token
        :access-token-secret))))

(deftwitter *account*)

(deftest testing-deftwitter
  (is (= (class *account*)
    twitter4j.TwitterImpl)))
```

After saving the final changes, your test results should look similar to the following output:

```
$ lein test
```

```
lein test tdsl.core-test
```

```
Ran 3 tests containing 10 assertions.
```

```
0 failures, 0 errors
```

The expectations framework

The expectations framework is an alternative to the `clojure.test` framework and is self-described as a minimalist unit testing framework. You can get the full documentation at <http://jayfields.com/expectations/>.

If you're using the Leiningen utility, you can add the following to the `:dependencies` configuration in your `project.clj` file and run the `lein deps` command:

```
[expectations "1.4.49"]
```

If done correctly, your `project.clj` file should look like the following:

```
(defproject tdsl "0.1.0-SNAPSHOT"
  :description "FIXME: write description"
  :url "http://example.com/FIXME"
  :license {:name "Eclipse Public License"
            :url "http://www.eclipse.org/legal/epl-v10.html"}
  :dependencies [[org.clojure/clojure "1.5.1"]
                [org.twitter4j/twitter4j-core "3.0.3"]
                [expectations "1.4.49"]])
```

If you're not using the Leiningen utility, you can get the JAR file and dependency information from the Clojars repository located at <https://clojars.org/expectations>.

Using the expect macro

Before moving on, you'll have to create a new test file at `tdsl/test/tdsl/query_test.clj`. The header of this file should match the following code's header. Notice how the `clojure.test` namespace isn't included in the code:

```
(ns tdsl.search-test
  (:require [tdsl.search :refer :all])
  (:use tdsl.core
        expectations))

;; Assumes deftwitter is correctly operating

(deftwitter *account*)
```

Although the expectations framework is very Leiningen-friendly, not everyone runs the `lein` command, so we'll have to start a REPL to run tests in this namespace. The `expectations/run-all-tests` function will run all the expectation tests within the current namespace and should be placed at the end of your testing file.

The expectations framework, in my opinion, is less descriptive as to what's being tested, but uses a lot less code to run these tests. The `expect` macro is very powerful and has capabilities that go beyond what's explained in this book, so please refer for more information to the documentation located at <http://jayfields.com/expectations/>.

The `expect` macro is similar to the `clojure.test/is` macro, with the exception that the `expect` macro assumes the two arguments passed are `true`, depending on the first argument. For example, if the first argument is the name of a class, it expects the second argument to be of or derived from that class. If the first argument is a regular expression, the second argument is assumed to be tested for the match.

The very brief and flexible `expect` macro will greatly reduce the amount of code needed to perform tests, in comparison to the `clojure.test` framework.

Search testing

The first thing to test in the `tdsl.search` namespace would be the `query->` macro. This macro allows for all the other functions in the namespace to function properly. There are four local bindings created by this macro, so there should be four tests created using the `expect` macro, as shown in the following code snippet:

```
tdsl.search-test> ;; expect query bindings to work properly

(tdsl-> *account*
  (query-> "twitter"
    (expect twitter4j.internal.json.QueryResultJSONImpl
      query)
    (expect 15 query-count)
    (expect true next-page?)
    (expect java.util.ArrayList tweets)))

#'tdsl.search-test/G__2345
tdsl.search-test> (clojure.pprint/pprint
  (expectations/run-all-tests))

Ran 4 tests containing 4 assertions in 102 msecs
?[32m0 failures, 0 errors?[0m.
{:pass 4,
 :test 4,
 :error 0,
 :type :summary,
 :fail 0,
 :ignored-expectations 0,
 :run-time 102}
```

Secondly, the `text`, `user`, and `date` functions will have to be tested. You'll notice the values to be tested here are in a `let` form and evaluated before being sent to the `expect` macro as the second argument. That's because when passing functions that depend on the binding of the `*tweets*` variable, an error is thrown stating that the variable isn't bound. This means that the test values need to be evaluated before being passed to the `expect` macro, as shown in the following code snippet:

```
tdsl.search-test> ;; expect query results macro from-tweet to work
properly

(twitter-> *account*
  (query-> "twitter"
    (let [text-list? (every? string? (text))
          user-list? (every? #(and (:user %)
                                   (:screen-name %))
                              (user))
          date-list? (every? string? (date))]
      (expect true text-list?)
      (expect true user-list?)
      (expect true date-list?))))

#'tdsl.search-test/G__3003
tdsl.search-test> (clojure.pprint/pprint
  (expectations/run-all-tests))

Ran 7 tests containing 7 assertions in 180 msecs
?[32m0 failures, 0 errors?[0m.
{:pass 7,
 :test 7,
 :error 0,
 :type :summary,
 :fail 0,
 :ignored-expectations 0,
 :run-time 180}
nil
```

The last test for this namespace with this framework is for the `from-query` macro. This test is very similar to the last test, but the values come from the `from-query` macro local bindings. If all works well, you should have all the tests passing:

```
tdsl.search-test> ;; expect from-query bindings to work properly

(twitter-> *account*
  (query-> "twitter"
    (let [data (from-query text user date)]
```

```
data-test #(every? string? (map % data))
user-exist? (every? #(and (:user %)
                          (:screen-name %))
                  (map :user data))
text-exist? (data-test :text)
date-exist? (data-test :date)]
(expect true user-exist?)
(expect true text-exist?)
(expect true date-exist?)))))
#'tdsl.search-test/G__3201
tdsl.search-test> (clojure.pprint/pprint
                  (expectations/run-all-tests))

Ran 10 tests containing 10 assertions in 265 msecs
?[32m0 failures, 0 errors?[0m.
{:pass 10,
 :test 10,
 :error 0,
 :type :summary,
 :fail 0,
 :ignored-expectations 0,
 :run-time 265}
nil
```

The final test file should look like the following:

```
(ns tdsl.search-test
  (:require [tdsl.search :refer :all])
  (:use tdsl.core
        expectations))

;; Assumes deftwitter is correctly operating

(deftwitter *account*)

;; expect query bindings to work properly

(twitter-> *account*
  (query-> "twitter"
    (expect twitter4j.internal.json.QueryResultJSONImpl
              query)
    (expect 15 query-count)
    (expect true next-page?)
    (expect java.util.ArrayList tweets))))
```

```
;; expect query results macro from-tweet to work properly

/twitter-> *account*
  (query-> "twitter"
    (let [text-list? (every? string? (text))
          user-list? (every? #(and (:user %)
                                   (:screen-name %))
                              (user))
          date-list? (every? string? (date))]
      (expect true text-list?)
      (expect true user-list?)
      (expect true date-list?))))

;; expect from-query bindings to work properly

/twitter-> *account*
  (query-> "twitter"
    (let [data (from-query text user date)
          data-test #(every? string? (map % data))
          user-exist? (every? #(and (:user %)
                                     (:screen-name %))
                               (map :user data))
          text-exist? (data-test :text)
          date-exist? (data-test :date)]
      (expect true user-exist?)
      (expect true text-exist?)
      (expect true date-exist?))))

(expectations/run-tests ['tdsl.search-test])
```

The midje framework

The `midje` unit testing framework is one of the most flexible, feature-rich, and well-documented Clojure frameworks out there. Because of all the possible types of tests that can be performed with the `midje` framework, only the most basic of tests will be explained here. More specifically, the testing and grouping of tests. You can view the full `midje` framework documentation at <https://github.com/marick/Midje/wiki>.

If you're using the Leiningen utility, you can add the following to the `:dependencies` configuration in your `project.clj` file and run the `lein deps` command:

```
[midje "1.5.1"]
```

If done correctly, your `project.clj` file should look like the following:

```
(defproject tdsl "0.1.0-SNAPSHOT"
  :description "FIXME: write description"
  :url "http://example.com/FIXME"
  :license {:name "Eclipse Public License"
            :url "http://www.eclipse.org/legal/epl-v10.html"}
  :dependencies [[org.clojure/clojure "1.5.1"]
                 [org.twitter4j/twitter4j-core "3.0.3"]
                 [expectations "1.4.49"]
                 [midje "1.5.1"]])
```

If you're not using the Leiningen utility, you can get the JAR file and dependency information from the Clojars repository located at <https://clojars.org/midje>.

Using the fact macro

Before moving on, you'll have to create a new test file at `tdsl/test/tdsl/tweet_test.clj`. The header of this file should match the following code's header. Notice how the `clojure.test` namespace isn't included in the code:

```
(ns tdsl.tweet-test
  (:require [tdsl.tweet :refer :all])
  (:use tdsl.core
        midje.repl))

;; Assumes deftwitter is correctly operating

(deftwitter *account*)
```

The first thing to test in the `tdsl.tweet` namespace would be the `tweet->` macro. This macro allows for all the other functions in the namespace to function properly. To do this, we need to use the `fact` macro from the `midje.repl` namespace. The `fact` macro takes a single argument on both sides of the `'=>'` symbol. This macro states that the left side is to be expected on the right side of the symbol. Unlike the `expectations` framework, the first argument can be a description of what's being tested, as shown in the following code snippet:

```
tdsl.tweet-test>
(tweet-> *account*
  (let [-tweet (send-tweet "API example request")]
    (fact "*tweet* is bound"
      (tweet-> -tweet *tweet*) => -tweet)
    (fact "Tweeting a new message"
      (class -tweet) =>
```

```

        twitter4j.internal.json.StatusJSONImpl)
    (fact "remove tweet"
      (tweet-> -tweet (class (delete))) =>
        twitter4j.internal.json.StatusJSONImpl)))

true

```

To group tests, you can place multiple fact forms within the `facts` body. The first argument of the `facts` body describes the group of facts that are being tested, as shown in the following code snippet:

```

tdsl.tweet-test>
(twitter->
  *account*
  (let [-tweet (send-tweet "Clojure tweets")]
    (facts "Can tweet, favorite, unfavorite, and remove tweet"
      (tweet-> -tweet
        (tweet-> (toggle-favorite)
          (fact "favoriting tweet status"
            (favorite?) => true)
          (fact "removing favorite from tweet
status"
              (tweet-> (toggle-favorite)
                (favorite?) => false))))))

    (fact "remove tweet"
      (class (tweet-> -tweet (delete)))
      => twitter4j.internal.json.StatusJSONImpl)))

true

```

The `midje.repl` namespace comes with a function to test an entire namespace. The `load-facts` function will accept a namespace name and print the test results of each `midje` fact in the namespace, as shown in the following code snippet:

```

tdsl.tweet-test> (load-facts 'tdsl.tweet-test)
All checks (6) succeeded.
{:failures 0}

```

The final test file should look like the following:

```

(ns tdsl.tweet-test
  (:require [tdsl.tweet :refer :all])
  (:use tdsl.core
        midje.repl))

;; Assumes deftwitter is correctly operating

```

```
(deftwitter *account*)

(twitter->
  *account*
  (let [-tweet (send-tweet "API example request")]
    (fact "*tweet* is bound"
      (tweet-> -tweet *tweet*) => -tweet)
    (fact "Tweeting a new message"
      (class -tweet) =>
        twitter4j.internal.json.StatusJSONImpl)
    (fact "remove tweet"
      (tweet-> -tweet (class (delete))) =>
        twitter4j.internal.json.StatusJSONImpl)))

(twitter->
  *account*
  (let [-tweet (send-tweet "Clojure tweets")]
    (facts "Can tweet, favorite, unfavorite, and remove tweet"
      (tweet-> -tweet
        (tweet-> (toggle-favorite)
          (fact "favoriting tweet status"
            (favorite?) => true)
          (fact "removing favorite from tweet
status"
              (tweet-> (toggle-favorite)
                (favorite?) => false))))))
    (fact "remove tweet"
      (class (tweet-> -tweet (delete)))
      => twitter4j.internal.json.StatusJSONImpl)))

;; Run tests

(load-facts 'tdsl.tweet-test)
```

The specjlj framework

The specjlj unit testing framework is a great alternative to the other unit testing frameworks mentioned in this chapter. It's a very well-documented Clojure framework and has many types of tests that can be performed easily. You can view the full Specjlj framework documentation at <http://www.specjlj.com/docs>.

If you're using the Leiningen utility, you can add the following to the `:dependencies` configuration in your `project.clj` file and run the `lein deps` command:

```
[specjlj "2.5.0"]
```

If done correctly, your `project.clj` file should look like the following:

```
(defproject tdsl "0.1.0-SNAPSHOT"
  :description "FIXME: write description"
  :url "http://example.com/FIXME"
  :license {:name "Eclipse Public License"
            :url "http://www.eclipse.org/legal/epl-v10.html"}
  :dependencies [[org.clojure/clojure "1.5.1"]
                 [org.twitter4j/twitter4j-core "3.0.3"]
                 [expectations "1.4.49"]
                 [midje "1.5.1"]
                 [speclj "2.5.0"]])
```

If you're not using the Leiningen utility, you can get the JAR file and dependency information from the Clojars repository located at <https://clojars.org/speclj>.

Using the `describe`, `it`, `should`, and `should=` macros

Before moving on, you'll have to create a new test file at `tdsl/test/tdsl/user_test.clj`. The header of this file should match the following code's header. Notice how the `clojure.test` namespace isn't included in the code:

```
(ns tdsl.user-test
  (:require [tdsl.user :refer :all])
  (:use tdsl.core
        speclj.core
        speclj.run.standard))

;; Assumes deftwitter is correctly operating

(deftwitter *account*)
```

The `describe` macro groups a series of tests together. The first argument is the description of what the test group is; the remaining arguments are treated as a code body containing tests. The `it`, `should`, and `should=` macros are some of the more basic forms that the `speclj` framework provides. The `it` macro takes a test name `String` and an assertion body from the `speclj` framework. The `should` macro is an assertion that can be placed within the `it` macro body and assumes that the returned value will make the test `true`. The `should=` macro assumes that the second argument should match the first argument with the first argument being what's expected as a result:

```
tdsl.user-test>
(describe
  "Testing Twitter user components"
```



```
(before
  (println "tdsl.user testing"))
(after
  (println "completed tdsl.user tests"))
(twitter-> *account*
  (let [user (user-obj "cljds1")
        id->user (id->user 1389920761)
        user->id (user->id "cljds1")]
    (it "User name to user object resolution"
      (should= twitter4j.internal.json.UserJSONImpl
        (class user)))
    (it "User screenname to user id"
      (should (integer? user->id)))
    (it "User id to user screenname"
      (should (string? id->user))))))

#<Description Description: "Testing Twitter user components">
```

Using the should-contain macro

The `should-contain` macro can handle many types of test, but for the type of test needed, we can use the `should-contain` macro to check if a collection contains a key. Because the `it` macro can accept more than one assertion, we'll also check if the map values are correct with the `should` macro, as shown in the following code snippet:

```
tdsl.user-test>
(describe
  "Testing user details"
  (before
    (println "tdsl.user details testing"))
  (after
    (println "completed tdsl.user details tests"))
  (twitter-> *account*
    (let [details (user-details "cljds1")]
      (it "ID set"
        (should-contain :id details)
        (should (integer? (:id details))))
      (it "Username set"
        (should-contain :user details)
        (should (string? (:user details))))
      (it "Language set"
        (should-contain :language details)
        (should (string? (:language details))))
      (it "Friend count set"
```

```

        (should-contain :friends-count details)
        (should (integer? (:friends-count details))))
      (it "Follower count set"
        (should-contain :followers-count details)
        (should (integer? (:followers-count details))))))
    #<Description Description: "Testing user details">

```

After the tests have been evaluated, you can get a report on the results by calling the `run-specs` function, as shown in the following code snippet:

```

tdsl.user-test> (run-specs)
tdsl.user testing
completed tdsl.user tests
.
tdsl.user details testing
completed tdsl.user details tests
.
Finished in 0.00171 seconds
2 examples, 0 failures
The final test file should look like the following.
(ns tdsl.user-test
  (:require [tdsl.user :refer :all])
  (:use tdsl.core
        speclj.core
        speclj.run.standard))

;; Assumes deftwitter is correctly operating

(deftwitter *account*)

(describe
  "Testing Twitter user components"
  (before
    (println "tdsl.user testing"))
  (after
    (println "completed tdsl.user tests"))
  (twitter-> *account*
    (let [user (user-obj "cljds1")
          id->user (id->user 1389920761)
          user->id (user->id "cljds1")]
      (it "User name to user object resolution"
        (should= twitter4j.internal.json.UserJSONImpl
                  (class user)))
      (it "User screenname to user id"
        (should (integer? user->id)))

```

```
(it "User id to user screenname"
  (should (string? id->user))))))

(describe
  "Testing user details"
  (before
    (println "tdsl.user details testing"))
  (after
    (println "completed tdsl.user details tests"))
  (twitter-> *account*
    (let [details (user-details "cljdsl")]
      (it "ID set"
        (should-contain :id details)
        (should (integer? (:id details))))
      (it "Username set"
        (should-contain :user details)
        (should (string? (:user details))))
      (it "Language set"
        (should-contain :language details)
        (should (string? (:language details))))
      (it "Friend count set"
        (should-contain :friends-count details)
        (should (integer? (:friends-count details))))
      (it "Follower count set"
        (should-contain :followers-count details)
        (should (integer? (:followers-count details)))))))

(run-specs)
```

Summary

As you can see, there are many unit testing options for such a young language. Some solutions are more verbose than others and some offer more testing utilities, but much the same results can be achieved with each framework.

Test-first or test-driven development can help ensure that the changes in your DSL are behaving as expected. Although unit testing can increase project stability, it doesn't ensure a bug-free environment. Your tests will not only have to be written properly to be effective, but they'll also have to be kept up-to-date as the language grows.

In the next chapter, you'll be learning how to make Java-callable Clojure classes.

11

Clojure DSLs inside Java

In this chapter you'll 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 of this chapter will cover importing and using Clojure source files from within Java.

This chapter covers the following topics:

- Generating Java language classes
- Class data hiding
- Ahead-of-time compilation
- Making Java wrappers for Clojure functions

Making a Java-callable Clojure class

The Clojure language comes with a macro that makes Java class generation very simple and straightforward. The name of the macro is `gen-class`, and this section will cover some of its basic uses on the `tdsl.core` namespace. This file is located at `tdsl/src/tdsl/core.clj` if you've followed the example DSL tutorial.

Class naming

Arguments passed to the `gen-class` macro describe how to build a named class for the Java language. Unlike the Java language, the `gen-class` macro optionally allows for the name of the class to be different from the actual filepath. For example, if your Clojure source file is located in `/org/example/hello.clj`, the class doesn't have to be `org.example.hello` if you define a new class name. The name of a class can be set by passing two additional arguments to the `gen-class` macro. The first of the two is the `:name` keyword, and the second can be a string or a symbol:

```
;; name the class
(gen-class :name "tdsl.core")
```

```
;; same as above
(gen-class :name tdsl.core)
```

Data hiding

The `gen-class` macro has a way of limiting what methods a class has by requiring all methods that are to be included to have a prefix in the name of the symbols. The default prefix is the `-` character. For example, a function named `my-fn` will be ignored while a function named `-my-fn` will be included as a class method. To change the default prefix, you can pass the `:prefix` keyword followed by a string to the `gen-class` macro:

```
(ns tdsl.core
  (:gen-class :name tdsl.core
              :prefix "java-")
  (:import [twitter4j TwitterFactory]
           [twitter4j.conf ConfigurationBuilder]))
```

Because we don't have any functions yet that can be found by `gen-class`, we'll have to make one by creating a new function called `java-loadConfig`. This should go right after the `load-config` function and accept one path argument:

```
(defn java-loadConfig [^String path]
  (load-config path))
```

Now, we need to tell the `gen-class` the name of the method and the method's argument type and return type. We do this by passing the `:methods` keyword and a collection of method description vectors. We also need to tell the `gen-class` that this class doesn't have a main method by setting the `:main` keyword parameter to `false`:

```
(ns tdsl.core
  (:gen-class :name tdsl.core
              :prefix "java-"
              :methods [{#{:static true}
                        [loadConfig
                         [String]
                         clojure.lang.PersistentArrayMap]]
              :main false)
  (:import [twitter4j TwitterFactory]
           [twitter4j.conf ConfigurationBuilder]))
```

AOT – the ahead-of-time compilation

Ahead-of-time compilation comes with many benefits but, for our purposes, we'll be using it only to compile the named classes so that we can use them in Java. To enable ahead-of-time compilation for the `tdsl.core` namespace, we'll have to add the AOT configuration values to your `project.clj` file. If you're not using Leiningen, you'll have to do the equivalent process in whatever tools you choose to use.

```
(defproject tdsl "0.1.0-SNAPSHOT"
  :description "FIXME: write description"
  :url "http://example.com/FIXME"
  :license {:name "Eclipse Public License"
            :url "http://www.eclipse.org/legal/epl-v10.html"}
  :aot [tdsl.core]
  :dependencies [[org.clojure/clojure "1.5.1"]
                 [org.twitter4j/twitter4j-core "3.0.3"]
                 [expectations "1.4.49"]
                 [midje "1.5.1"]
                 [speclj "2.5.0"]])
```

Now that the class has been described and the AOT compilation is enabled for the namespace, we can compile our project to produce a class. If you're still in the source directory `/tdsl/src/tdsl`, change to the `/tdsl/` directory before running the following commands:

```
$ lein compile
Compiling tdsl.core
$ cd target/classes/
$ ls -l tdsl/*.class
tdsl/core.class
tdsl/core$deftwitter.class
tdsl/core$fn__14.class
tdsl/core__init.class
tdsl/core$java_loadConfig.class
tdsl/core$load_config.class
tdsl/core$loading__4910__auto__.class
tdsl/core$twitter__GT__.class
```

Back to working in the `/tdsl/` project directory, we'll have to make a Java JAR file so that we can have a portable archive that will allow us to import the `tdsl.core` namespace in our Java projects:

```
$ lein uberjar
Created /tdsl/target/uberjar+provided/tdsl-0.1.0-SNAPSHOT.jar
Including tdsl-0.1.0-SNAPSHOT.jar
-- removed output for brevity --
Created /tdsl/target/tdsl-0.1.0-SNAPSHOT-standalone.jar
```

Now, we need to make a Java file to call in our class. This file should be named `DslJava.java` and it should be located in the root directory of the Clojure project. The source code should look similar to the following code:

```
import tdsl.core;

class DslJava {
    public static void main (String[] args) {
        System.out.println(core.loadConfig("api-config"));
    }
}
```

If everything is done correctly, we should be able to compile `DslJava` without any errors:

```
$ javac -cp '.:target/tdsl-0.1.0-SNAPSHOT-standalone.jar' DslJava.java
```

With a similar command, the program can now be executed:

```
$ java -cp '.:target/tdsl-0.1.0-SNAPSHOT-standalone.jar' DslJava
{:consumer-key "x", :consumer-secret "x", :access-token "x", :access-token-secret "x"}
```

Java-wrapping your Clojure

There's another way of getting Java to use Clojure. You can import from the Clojure language JAR file and either load a Clojure file or evaluate a string of code. For this section, you'll need to create a temporary directory called `test-dir`. For all further instructions, we will assume that you're working from the temporary directory.

To be able to use Clojure from within Java, you'll have to get the Clojure language JAR file from <http://search.maven.org>. The file can be located by searching for `org.clojure` and the Clojure artifact ID. If you're using the Leiningen utility, you can get the JAR file from your local Maven repository:

```
$ cp ~/.m2/repository/org/clojure/clojure/1.5.1/clojure-1.5.1.jar ./
```

Now we need to create a file called `ExampleClass.java`. This file needs to import Clojure's runtime (`clojure.lang.RT`) and variable (`clojure.lang.Var`) classes:

```
import clojure.lang.RT;
import clojure.lang.Var;
```

The second to the last step is to make a main method for the `ExampleClass` class and have it load a Clojure file named `example.clj`. The `RT.loadResourceScript` method will evaluate a Clojure source code file and make the information accessible from Java. The `RT.var` method will help take Clojure variables and turn them into a Java object.

```
class ExampleClass {
    public static void main (String[] args) {
        try{
            RT.loadResourceScript("example.clj");
        }catch(Exception ex) { /* error */ }

        Var mapOp = RT.var("example", "map-op");

        System.out.println( mapOp.invoke(1, 2, 3) );
    }
}
```

The final step is to make the `example.clj` file and compile our `ExampleClass` program. The `example.clj` file should look like the one shown next. There's only one method called `map-op` in the `example` namespace and it takes a series of numbers as the `x` argument:

```
(ns example)
(defn map-op [& x] (apply str (map inc x)))
```

We can now compile and run the `ExampleClass` program:

```
$ javac -cp '.:clojure-1.5.1.jar' ExampleClass.java
$ java -cp '.:clojure-1.5.1.jar' ExampleClass
234
```


Summary

This was a short but interesting chapter. As you can see, Clojure does a very good job at keeping Java programs in mind. The ability to go from Clojure to Java and back is what really makes Clojure appealing. This also allows for Clojure to possibly be used with other JVM languages in the same way as Java uses Clojure. Hopefully, this book has helped you get a good grip on Clojure and all of its DSL development-friendly capabilities. Thanks for reading.

References

Chapter 1: An Overview of Domain-specific Languages with Clojure

- **Clostache Project:**
<https://github.com/fhd/clostache>
- **Quil Project:**
<https://github.com/quil/quil>
- **Processing Project:**
<http://processing.org/>
- **Overtone Project:**
<https://github.com/overtone>
- **Music-as-data Project:**
<https://github.com/jonromero/music-as-data>
- **Clojure Script Project:**
<https://himera.herokuapp.com/synonym.html>
- **Mustache Project:**
https://en.wikipedia.org/wiki/Mustache_%28template_system%29
- **Mustache Project:**
mustache.github.com/

- **Clojure.java.jdbc, February 20, 2013:**
<https://github.com/clojure/java.jdbc/>
- **W3 Schools SQL, February 18, 2013:**
http://www.w3schools.com/sql/sql_insert.asp
- **Chris Granger, 2011:**
<http://sqlkorma.com>
- **Clojure Contrib Libraries, February 13, 2013:**
<http://dev.clojure.org/display/doc/Clojure+Contrib+Libraries>
- **Dmitry Kandalov, November 8, 2009, slide(s) 35:**
<http://www.slideshare.net/dkandalov/dsl-explained-2450024>
- **Martin Fowler, June 12, 2005:**
<http://martinfowler.com/articles/languageWorkbench.html#XmlConfigurationFiles>
- **Tiziano Perrucci, February 1, 2012:**
http://prezi.com/fuxyebx_nazm/external-dsl-with-parser-combinators/
- **Nestor Arocha, June 8, 2012:**
<http://pydsl.blogspot.jp/2012/06/internal-vs-external-dsls.html>
- **Weerasak, May 14, 2008, slide(s) 27, 30, 44:**
<http://www.slideshare.net/weerasak/domain-specific-languages>
- **Martin Fowler, May 15, 2008:**
<http://www.martinfowler.com/bliki/DomainSpecificLanguage.html>
- **Martin Fowler / Infoq.com, October 31, 2006:**
<http://www.infoq.com/presentations/domain-specific-languages>
- **Wikipedia:**
https://en.wikipedia.org/wiki/Domain-specific_language
https://en.wikipedia.org/wiki/Interpreted_language
https://en.wikipedia.org/wiki/Interpreter_%28computing%29
https://en.wikipedia.org/wiki/Library_%28computing%29
- *DSLs in Actions* by Debasish Ghosh, December 2010, page 17 to 19

Chapter 2: Design Concepts with Clojure

- **GitHub/bbatsov:**
<https://github.com/bbatsov/clojure-style-guide>
- **Wikipedia YAGNI:**
<https://en.wikipedia.org/wiki/YAGNI>
- **Artima / Christopher Diggins, July 24, 2011:**
<http://www.artima.com/weblogs/viewpost.jsp?thread=331531>
- **Apache:**
<https://people.apache.org/~fhanik/kiss.html>
- **Artima / Dave Thomas, 2013:**
<http://www.artima.com/intv/dry.html>
- **O'Reilly, November 24, 2009:**
http://programmer.97things.oreilly.com/wiki/index.php/Don%27t_Repeat_Yourself
- **Vasja Volin, December 13, 2010:**
<http://superprofundo.com/2010/12/13/top-down-and-bottom-up-pros-and-cons/>
- **Paul Graham, 1993:**
<http://www.paulgraham.com/progbot.html>
- **Wikipedia:**
https://en.wikipedia.org/wiki/Domain-specific_language
https://en.wikipedia.org/wiki/Interpreted_language
https://en.wikipedia.org/wiki/Interpreter_%28computing%29
https://en.wikipedia.org/wiki/Library_%28computing%29
- *DSLs in Actions* by Debasish Ghosh, December 2010, page 17 to 19

Chapter 3: Clojure Editing and Project Creation

- **Leiningen:**
<http://leiningen.org/>
<https://github.com/technomancy/leiningen>
- **nrepl.el:**
<https://github.com/technomancy/nrepl.el>
- **REPL-y:**
<https://github.com/trptcolin/reply>
- **Emacs Wiki:**
<http://www.emacswiki.org/emacs/EmacsKeyNotation>
<http://emacswiki.org/emacs/DotEmacsDotD>
- **Official Ubuntu Help:**
<https://help.ubuntu.com/community/EmacsHowto>
- **StackExchange SuperUser:**
<http://superuser.com/questions/242587/install-gnu-emacs-gui-in-fedora>
- **Emacs MacPorts:**
http://wikemacs.org/wiki/Installing_Emacs_on_OS_X
- **GNUvince:**
<https://gnuvince.wordpress.com/2012/02/19/why-i-still-use-emacs/>
- **StackOverflow:**
<http://stackoverflow.com/questions/232486/best-common-lisp-ide>
- **GNU is Not Unix:**
https://www.gnu.org/software/emacs/manual/html_node/emacs/Major-Modes.html
<https://www.gnu.org/software/emacs/emacs-faq.html#Origin-of-the-term-Emacs>
<https://www.gnu.org/software/emacs/>

Chapter 4: Features, Functions, and Macros

- **Clojuredocs.org:**
http://clojuredocs.org/clojure_core/clojure.core/lazy-seq
- **Clojure.org:**
<http://clojure.org/documentation>
http://clojure.org/special_forms
<http://clojure.org/cheatsheet>
http://clojure.org/java_interop
- **R. Mark Volkmann:**
<http://java.ociweb.com/mark/clojure/article.html#Namespaces>
<http://java.ociweb.com/mark/clojure/article.html#JavaInterop>
- **Clojure Documentation Project:**
<http://clojure-doc.org/articles/language/namespaces.html>
<http://clojure-doc.org/articles/language/interop.html>

Chapter 5: Collections and Sequencing

- **clojure.org:**
http://clojure.org/data_structures
<http://clojure.org/sequences>
- **Clojuredocs:**
<http://clojuredocs.org/quickref/Clojure%20Core>
- **Stackoverflow;**
<http://stackoverflow.com/questions/10275321/how-to-pipe-a-clojure-iterator-seq-using-the-thread-macro>
- **Structural sharing image source:**
<http://debasishg.blogspot.jp/2010/05/grokking-functional-data-structures.html>

Chapter 6: Assignment and Concurrency

- **Clojure.org:**
<http://clojure.org/atoms>
<http://clojure.org/refs>
<http://clojure.org/agents>
- **Webatu:**
<http://faustus.webatu.com/clj-quick-ref.html>
- **Clojuredocs:**
<http://clojuredocs.org/quickref/Clojure%20Core>
- **Clojure-Doc:**
http://clojure-doc.org/articles/language/concurrency_and_parallelism.html

Chapter 7: Flow Control, Error Handling, and Math

- **Clojuredocs.org:**
<http://clojuredocs.org/quickref/Clojure%20Core>

Chapter 8: Methods for Abstraction

- **Clojure.org:**
http://clojure.org/runtime_polymorphism
<http://clojure.org/multimethods>
http://clojure.org/java_interop
- **Clojure-Doc.org:**
<http://clojure-doc.org/articles/language/polymorphism.html>

Chapter 9: An Example Twitter DSL

- **twitter4j:**
<http://twitter4j.org/>
- **Clojure:**
http://clojure.org/java_interop
- **Twitter:**
<http://dev.twitter.com>

Chapter 10: Unit Testing

- <http://www.guru99.com/unit-testing.html>
- <https://github.com/jaycfields/expectations>
- <http://jayfields.com/expectations/>
- <https://github.com/marick/Midje>
- <https://github.com/marick/Midje/wiki>
- <https://github.com/slagyr/specclj>
- <http://www.specclj.com/docs>

Chapter 11: Clojure DSLs inside Java

- **Clojure.org:**
<http://clojure.org/compilation>
- **Wikibooks.org:**
https://en.wikibooks.org/wiki/Clojure_Programming/Tutorials_and_Tips#Invoking_Clojure_from_Java
- **walkwithoutrhythm.net:**
<http://walkwithoutrhythm.net/blog/2012/03/26/how-to-call-clojure-1-dot-3-functions-from-java/>

Index

Symbols

`^`:dynamic metadata 46
`==` 151
`:access-token` key 214
`:action` form key 19
`:as` key 81
`~/bin` directory 61
`-` character 230
`~` character 86
`~@` characters 86
`:doc` key 77
`(flatten)` function 28
`-main` function 63
`:method` form key 19
`:methods` keyword 230
`-my-fn` function 230
`*ns*`variable 70
`:only` keyword 71
`(pattern)` function 26
`(p)` function 26
`:post`
 using 40
`:pre`
 using 40
`-` symbol 148
`+` symbol 148
`*tweets*`variable 219
`:validator` option 134

A

agents 128-133
ahead-of-time compilation. *See* AOT
alignment 36-40
alter-var-root form 122

anonymous function 85
AOT 231, 232
Apache Cassandra 17
Apache CouchDB
 URL 17
are macro 214
arithmetic operations
 about 148
 addition 148
 dec form 150
 divivision 149
 equality 151
 inc form 150
 max form 150
 min form 150
 mod form 149
 multiplication 148
 rem form 149
 substraction 148
array-map form 98
assoc form 93
associative? form 93
atom 126, 128
audio domain
 about 26
 Music-as-data 26, 27
 overtone 27-29

B

bottom-up development. *See* pure functions
Buffer window manipulation commands
 C-x 0 54
 C-x 1 54
 C-x 2 54
 C-x 3 54

- C-x C-c 54
- C-x k 54
- C-x left 54
- C-x o 54
- C-x right 54
- builder function** 29

C

- C-a command** 55
- case form** 45
- casting** 145, 146
- C-c C-b command** 58, 59
- C-c C-d command** 58, 59
- C-c C-e command** 58
- C-c C-j command** 58, 59
- C-c C-k command** 58
- C-c C-l command** 58
- C-c C-n command** 58
- C-c C-r command** 58
- C-c C-u command** 58
- C-e command** 55
- C-j command** 58
- C-k command** 55
- C- key** 53
- CLJ files**
 - creating, in Emacs 53-56
- Clojure**
 - agents 128-132
 - atom 126, 127
 - Futures 135, 136
 - Java wrapping 232, 233
 - Lists 96
 - name conventions 46
 - PersistentArrayMap 97
 - PersistentHashMap 97
 - Promise 137
 - Refs 134
 - transient 125
 - transient 125
 - variables 119
 - writing 35, 36
- Clojure Buffer commands**
 - C-c C-b 58
 - C-c C-d 58
 - C-c C-e 58
 - C-c C-j 58

- C-c C-k 58
- C-c C-l 58
- C-c C-r 58
- C-c M-n 58
- Clojure collection**
 - about 89, 90
 - concurrency model 89, 90
 - creating 90-93
 - destructuring 80, 81
- Clojure function** 82-84
- clojure.java.jdbc** 17
- Clojure library**
 - about 12
 - characteristic 13
 - current state 13
- Clojure namespace** 69-71
- Clojure objects**
 - immutability 75, 76
- Clojure REPL, running in Emacs**
 - nrepl.el Emacs extension 56
- ClojureScript**
 - about 22
 - comparing, with JavaScript 23-25
- Clojure sequences**
 - about 104
 - doall form 115
 - dorun form 116
 - doseq form, using 114
 - filter form 113
 - for form, using 114
 - interleave form 106
 - iterator-seq form 108
 - manipulating 109-112
 - parse form 107, 108
 - partition-all form 112
 - range form, using 105, 106
 - reductions form, using 115
 - seq? form, using 104
- clojure.test framework**
 - about 213
 - are macro, using 214, 215
 - exploring 211
 - final test, developing 215, 216
 - is macro, using 213
 - tdsl.core, testing 212
- Clojure vector**
 - about 94

- creating 94-97
- Clojurewerkz** 13
- Clostrache** 20, 22
- Cloujure**
 - object, comparing 145
- C-o command** 55
- common editing commands**
 - C-a 55
 - C-e 55
 - C-k 55
 - C-o 55
 - C-SPC 55
 - C-w 55
 - C-y 55
 - M-w 55
- compare-and-set! form** 128
- complement** 44
- cond form** 45
- condp form** 45
- config function** 186
- ConfigurationBuilder class** 187
- conj form** 96
- C-RET command** 58
- C-SPC command** 55
- C-up or C-down command** 59
- C-w command** 55
- C-x 0 command** 54
- C-x 1 command** 54
- C-x 2 command** 54
- C-x 3 command** 54
- C-x C-c command** 54
- C-x C-f command** 55
- C-x C-s command** 55
- C-x C-w command** 55
- C-x C-z command** 60
- C-x k command** 54
- C-x left command** 54
- C-x o command** 54
- C-x right command** 54
- C-Y command** 55

D

- database domain**
 - about 14
 - language, structuring 14-17
- days-later function** 178

- dec form** 150
- declare form** 121
- defn form** 82
- deftwitter macro**
 - building 188-191
- deliver form** 137
- DEL key** 53
- describe macro** 225
- destroyStatus method** 199
- dissoc form** 100
- distinct? form** 92
- doall form** 115
- domain-specific language.** *See* DSL
- Don't repeat yourself (DRY)** 35
- dorun form** 116
- doseq form** 114, 126

DSL

- about 8
- external DSL 10
- internal DSL 10
- language and domain 9
- limited scope 8
- syntax 8
- trust 10
- using 9
- dynamic Clojure objects** 76

E

- eat-contents! function** 159
- ECMA / JavaScript domain**
 - ClojureScript 22
 - ClojureScript, comparing with JavaScript 23-25

Emacs

- about 49
- CLJ files, creating in 53-56
- CLJ files, editing in 53-56
- Clojure REPL, running inside 56
- origin 49, 50
- setting up 51, 52

Emacs24

- installing 50, 51
- setting up 50, 51

- empty form** 91
- empty? form** 92
- ensure form** 135

- enumeration-seq form 108
- equality 151
- error handling
 - about 147
- ESC key 53
- every? form 92
- ExampleClass class 233
- expectations framework
 - about 217, 218
 - expect macro, using 217
 - search testing 218-220
- expect macro 218
- external DSL 10, 11
- extended-protocol form 156
- extended-type form 156

F

- favorite? function 199
- File manipulation commands
 - C-x C-f 55
 - C-x C-s 55
 - C-x C-w 55
- file-seq form 108
- filter form 113
- Flash class
- flow control 139-144
- for form 114
- formative 18
- frame-rate function 30
- from-query macro 219
- function
 - tasks 34
- future-cancelled? form 137
- Futures 135, 136

G

- gen-class macro 230
- get form 98
- get-in form 101
- group-by form 111

H

- hash-map form 97
- hierarchies relationships
 - about 169-172

- parent relationship conflicts,
 - resolving 172,173

Hiccup 19

HTML domain

- about 17
- Clostrache 20, 22
- formative 18, 19
- Hiccup 19, 20
- Mustache 20

I

- if-let form 41, 42
- is-? form 170
- import form 72
- inc form 150
- Integrated Development Environments (IDEs) 11
- interfaces
 - creating, in Clojure 153, 154
 - implimenting, with deftype 153, 154
- interleave form 106
- internal DSL 11, 12
- is macro
 - using 213

J

- Java-based abstractions
 - creating 177
 - Java objects, creating for
 - manipulation 178, 179
 - values, retrieving 179, 180
- Java callable Clojure class
 - class naming 229
 - creating 229
- Java class 72, 73, 74
- Java JAR
 - project, compiling to 65

K

- keep form 111
- Keep it simple, stupid. *See* KISS
- key notations, Emacs
 - C- 53
 - DEL 53
 - ESC 53

- M- 53
- RET 53
- S- 53
- SPC 53
- TAB 53
- key/value collection type**
 - defining 47
- KISS 35**
- Korma 14**
- Korma Internal DSL 16**

L

- lazy sequence 79**
- Lein 49**
- lein compile command 66**
- lein deps command 221, 224**
- lein deps:tree command 66**
- Leiningen**
 - about 61, 66
 - commands 66
 - Github repository wiki page 61
 - installing 61-64
 - project, starting with 61-64
- Leiningen commands**
 - lein compile 66
 - lein deps 66
 - lein deps:tree 66
 - lein install 66
 - lein jar 66
 - lein uberjar 66
- lein install command 66**
- lein jar command 66**
- lein test command 212**
- lein uberjar command 66**
- let form 41, 80, 219**
- line function 30**
- list*form 96**
- lists, Clojure**
 - about 96
 - creating 96
- load-config function 185-230**
- load-facts function 223**
- load-string function 208**
- Long method 203**

M

- macroexpand-1 form 162**
- Macro**
 - building 186, 187
 - creating 86, 87
 - deftwitter macro, building 188-191
 - twitter-> macro, building 191
- mapcat form 115**
- map form 114**
- maps, Clojure**
 - about 97
 - hash-map form, creating 97-102
- max form 150**
- max-key form 115**
- merge form 102**
- merge-with form 102**
- metadata**
 - viewing 78
- midje framework**
 - about 221
 - fact macro, using 222, 223
- min form 150**
- M- key 53**
- mod form 149**
- music-as-data 26**
- Mustache 20**
- M-w command 55**
- M-x nrepl command 64**

N

- not-any? form 92**
- Not-A-Can record class 155, 156**
- not-empty form 91**
- not-every? form 92**
- not= form 43**
- nrepl.el Emacs extension**
 - about 56
 - setup, testing 57-60
- nth form 110**

O

- OAuth configuration**
 - API configuration, creating 183

- project, creating 183
- reading 181
- required dependencies, adding 182
- Twitter account, registering 181
- Twitter configuration, reading 183-185

object

- comparing 145

object-relational mapping (ORM) 9

output constraints

- :post key, using 175, 176

overtone 27

P

parse form 107

partial form 91

peek form 95

persistent! form 126

processing 29

project

- Clojure, including in 64
- compiling, to Java JAR 65
- Java libraries, including in 64

Promise 137

protocol, using 155

proxy form

- used, for creating custome symbol
definations 163, 164
- used, for interface method
implementation 160, 161
- working with 159, 160

pure functions 34

Q

query-> macro 193

Quil

- about 29, 30
- GitHub project homepage 29

quot form 149

R

Raw SQL 16

Redis

- URL 17

reductions form 115

Refs 134, 135

reify forms

- about 74
- working 158-160

rem form 149

repeatedly form 105

repeat form 105

repeat function 29

REPL Buffer Commands

- C-c C-b 59
- C-c C-d 59
- C-c C-j 59
- C-c C-u 58
- C-j 58
- C-RET 58
- C-up or C-down 59
- TAB 59

RET key 53

retweet bot

- making 180

RT.var method 233

run-specs function 227

S

satisfies? form 157

search queries

- functions, searching 193-196
- handling 193
- macros, searching 193-196
- tdsl.search namespace, adding 193

send-off form 129

send-tweet function 198

seq? form 104

sequence form 104

sequential? form 93

set form 103

set! form 121

sets, Clojure

- hash-set, creating 103

setup option 30

should-contain macro

- using 226

should macro 225

should= macro 225

show-agent function 129

- shutdown-agents form 131
- S- key 53
- some form 92
- sorted? form 93
- sorted-map form 98
- sorted-set form 103
- spacing 36-40
- SPC key 53
- specjlj framework
 - about 224
 - describe macro 225
 - should-contain macro, using 226
 - should macro 225
 - should= macros 225
- split-lines function 208
- star symbol (*) 148
- status-do macro 199
- subvec form 96
- swap! form 127

T

- TAB command 59
- TAB key 53
- tdsl.core
 - testing 212
- tdsl.tweet namespace
 - adding 198
- test-dir 232
- test form 174
- toggle-favorite function 199
- transient 125
- tree-seq form 107
- tweet-> macro 222
- Tweets
 - handling 198-200
- twitter4j.TwitterImpl class 215
- Twitter DSL
 - building 177
 - event notifier 181
 - retweet bot 180
- TwitterFactory class 187
- twitter-> macro
 - about 213
 - building 191

U

- underive form 171
- unit testing 211
- user-related features
 - adding 201
 - logging features, adding 207, 208
 - tdsl.user namespace, adding 202
 - user-attr macro 202
 - user-details 203, 204
 - user-details multimethod 203, 204
 - user macro 202

V

- variables 119-124
- vec form 93, 94

W

- when form 42
- when-let form 41
- when-not form 43
- with-bindings form 123
- with-meta form 78

X

- x key 57

Y

- You aren't going to need it (YAGNI) 35

Z

- zipmap form 98



Thank you for buying **Clojure for Domain-specific Languages**

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

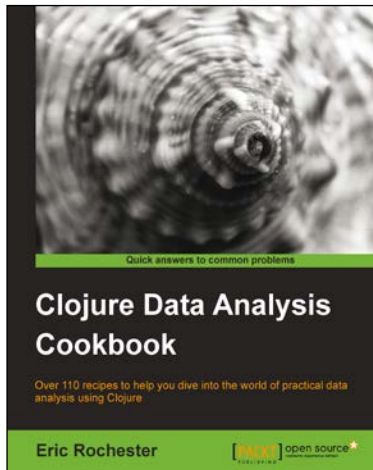
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



Clojure Data Analysis Cookbook

ISBN: 978-1-78216-264-3

Paperback: 342 pages

Over 100 recipes to help you dive into the world of practical data analysis using Clojure

1. Get a handle on the torrent of data the modern Internet has created
2. Recipes for every stage from collection to analysis
3. A practical approach to analyzing data to help you make informed decisions



JavaScript Unit Testing

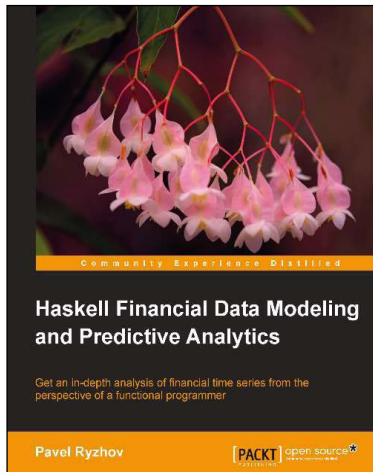
ISBN: 978-1-78216-062-5

Paperback: 190 pages

Your comprehensive and practical guide to efficiently performing and automatic JavaScript unit testing

1. Learn and understand, using practical examples, synchronous and asynchronous JavaScript unit testing
2. Cover the most popular JavaScript Unit Testing Frameworks including Jasmine, YUITest, QUnit, and JsTestDriver
3. Automate and integrate your JavaScript Unit Testing for ease and efficiency

Please check www.PacktPub.com for information on our titles

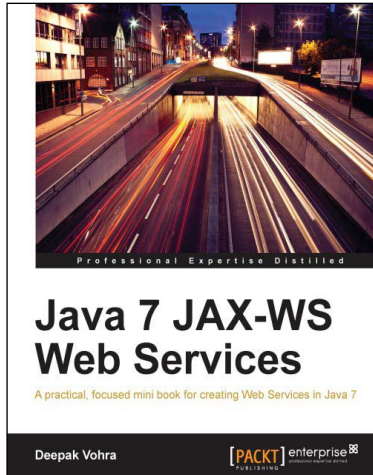


Haskell Financial Data Modeling and Predictive Analytics

ISBN: 978-1-78216-943-7 Paperback: 112 pages

Get an in-depth analysis of financial time series from the perspective of a functional programmer

1. Understand the foundations of financial stochastic processes
2. Build robust models quickly and efficiently
3. Tackle the complexity of parallel programming



Java 7 JAX-WS Web Services

ISBN: 978-1-84968-720-1 Paperback: 64 pages

A practical, focused mini book for creating Web Services in Java 7

1. Develop Java 7 JAX-WS web services using the NetBeans IDE and Oracle GlassFish server
2. End-to-end application which makes use of the new clientjar option in JAX-WS wsimport tool
3. Packed with ample screenshots and practical instructions

Please check www.PacktPub.com for information on our titles