

Eduardo Diaz, Shantanu Kumar,  
Akhil Wali

# Clojure: High Performance JVM Programming

## Learning Path

Explore asynchronous channels, logic, reactive programming, and much more



Packt

# Clojure: High Performance JVM Programming

---

# Table of Contents

[Clojure: High Performance JVM Programming](#)

[Credits](#)

[Preface](#)

[What this learning path covers](#)

[What you need for this learning path](#)

[Who this learning path is for](#)

[Reader feedback](#)

[Customer support](#)

[Downloading the example code](#)

[Errata](#)

[Piracy](#)

[Questions](#)

[1. Module 1](#)

[1. Getting Started with Clojure](#)

[Getting to know Clojure](#)

[Installing Leiningen](#)

[Using a REPL](#)

[The nREPL protocol](#)

[Hello world](#)

[REPL utilities and conventions](#)

[Creating a new project](#)

[Project structure](#)

[Creating a standalone app](#)

[Using Cursive Clojure](#)

[Installing Cursive Clojure](#)

[Getting started with Clojure code and data](#)

[Lists in Clojure](#)

[Operations in Clojure](#)

[Functions in Clojure](#)

[Clojure's data types](#)

[Scalars](#)

[Collection data types](#)

[Summary](#)

## [2. Namespaces, Packages, and Tests](#)

[Namespaces in Clojure](#)

[Packages in Clojure](#)

[The classpath and the classloader](#)

[Back to Clojure namespaces](#)

[Playing with namespaces](#)

[Creating a new namespace](#)

[Working with namespaces on the REPL](#)

[Testing in Clojure](#)

[Testing from the command line](#)

[Testing in IntelliJ](#)

[Summary](#)

## [3. Interacting with Java](#)

[Using Maven dependencies](#)

[Clojure interop syntax](#)

[Creating an object](#)

[Calling an instance method](#)

[Calling a static method or function](#)

[Accessing inner classes](#)

[Writing a simple image namespace](#)

[Writing the tests](#)

[The let statement](#)

[Destructuring in Clojure](#)

[Sequential destructuring](#)

[Associative destructuring](#)

[Exposing your code to Java](#)

[Testing from Groovy](#)

[Proxy and reify](#)

[Summary](#)

## [4. Collections and Functional Programming](#)

[Basics of functional programming](#)

[Persistent collections](#)

[Types of collections in Clojure](#)

[The sequence abstraction](#)

[Specific collection types in Clojure](#)

[Vectors](#)

[Lists](#)

[Maps](#)

[Sorted maps and hash maps](#)

[Common properties](#)

[Sets](#)

[Sorted sets and hash sets](#)

[Common properties](#)

[Union, difference, and intersection](#)

[Applying functional programming to collections](#)

[The imperative programming model](#)

[The functional paradigm](#)

[Functional programming and immutability](#)

[Laziness](#)

[Summary](#)

## [5. Multimethods and Protocols](#)

[Polymorphism in Java](#)

[Multimethods in Clojure](#)

[Keyword hierarchies](#)

[isa?](#)

[parents](#)

[descendants](#)

[underive](#)

[A la carte dispatch functions](#)

[Protocols in Clojure](#)

[Records in Clojure](#)

[Summary](#)

## [6. Concurrency](#)

[Using your Java knowledge](#)

[The Clojure model of state and identity](#)

[Promises](#)

[Pulsar and lightweight threads](#)

[Futures](#)

[Software transactional memory and refs](#)

[Atoms](#)

[Agents](#)

[Validators](#)

[Watchers](#)

[core.async](#)

[Why lightweight threads?](#)

[Gblocks](#)

[Channels](#)

[Transducers](#)

[Summary](#)

## [7. Macros in Clojure](#)

[Lisp's foundational ideas](#)

[Macros as code modification tools](#)

[Modifying code in Java](#)

[Modifying code in Groovy](#)

[The @ToString annotation](#)

[The @TupleConstructor annotation](#)

[The @Slf4j annotation](#)

[Writing your first macro](#)

[Debugging your first macro](#)

[Quote, syntax quote, and unquoting](#)

[Unquote splicing](#)

[gensym](#)

[Macros in the real world](#)

[References](#)

[Summary](#)

## [2. Module 2](#)

### [1. Performance by Design](#)

[Use case classification](#)

[The user-facing software](#)

## Computational and data-processing tasks

A CPU bound computation

A memory bound task

A cache bound task

An input/output bound task

Online transaction processing

Online analytical processing

Batch processing

A structured approach to the performance

The performance vocabulary

Latency

Throughput

Bandwidth

Baseline and benchmark

Profiling

Performance optimization

Concurrency and parallelism

Resource utilization

Workload

The latency numbers that every programmer should know

Summary

## 2. Clojure Abstractions

Non-numeric scalars and interning

Identity, value, and epochal time model

Variables and mutation

Collection types

Persistent data structures

Constructing lesser-used data structures

Complexity guarantee

$O(<7)$  implies near constant time

The concatenation of persistent data structures

Sequences and laziness

Laziness

Laziness in data structure operations

[Constructing lazy sequences](#)

[Custom chunking](#)

[Macros and closures](#)

[Transducers](#)

[Performance characteristics](#)

[Transients](#)

[Fast repetition](#)

[Performance miscellanea](#)

[Disabling assertions in production](#)

[Destructuring](#)

[Recursion and tail-call optimization \(TCO\)](#)

[Premature end of iteration](#)

[Multimethods versus protocols](#)

[Inlining](#)

[Summary](#)

[3. Leaning on Java](#)

[Inspecting the equivalent Java source for Clojure code](#)

[Creating a new project](#)

[Compiling the Clojure sources into Java bytecode](#)

[Decompiling the .class files into Java source](#)

[Compiling the Clojure source without locals clearing](#)

[Numerics, boxing, and primitives](#)

[Arrays](#)

[Reflection and type hints](#)

[An array of primitives](#)

[Primitives](#)

[Macros and metadata](#)

[String concatenation](#)

[Miscellaneous](#)

[Using array/numeric libraries for efficiency](#)

[HipHip](#)

[primitive-math](#)

[Detecting boxed math](#)

[Resorting to Java and native code](#)

## Proteus – mutable locals in Clojure

### Summary

## 4. Host Performance

### The hardware

#### Processors

Branch prediction

Instruction scheduling

Threads and cores

#### Memory systems

Cache

Interconnect

#### Storage and networking

### The Java Virtual Machine

The just-in-time compiler

Memory organization

HotSpot heap and garbage collection

Measuring memory (heap/stack) usage

Determining program workload type

Tackling memory inefficiency

#### Measuring latency with Criterium

Criterium and Leiningen

### Summary

## 5. Concurrency

### Low-level concurrency

Hardware memory barrier (fence) instructions

Java support and the Clojure equivalent

### Atomic updates and state

Atomic updates in Java

Clojure's support for atomic updates

Faster writes with atom striping

### Asynchronous agents and state

Asynchrony, queueing, and error handling

Why you should use agents

Nesting

## Coordinated transactional ref and state

Ref characteristics

Ref history and in-transaction deref operations

Transaction retries and barging

Upping transaction consistency with ensure

Lesser transaction retries with commutative operations

Agents can participate in transactions

Nested transactions

Performance considerations

Dynamic var binding and state

Validating and watching the reference types

Java concurrent data structures

Concurrent maps

Concurrent queues

Clojure support for concurrent queues

Concurrency with threads

JVM support for threads

Thread pools in the JVM

Clojure concurrency support

Future

Promise

Clojure parallelization and the JVM

Moore's law

Amdahl's law

Universal Scalability Law

Clojure support for parallelization

pmap

pcalls

pvalues

Java 7's fork/join framework

Parallelism with reducers

Reducible, reducer function, reduction transformation

Realizing reducible collections

Foldable collections and parallelism

Summary

## 6. Measuring Performance

Performance measurement and statistics

A tiny statistics terminology primer

Median, first quartile, third quartile

Percentile

Variance and standard deviation

Understanding Criterium output

Guided performance objectives

Performance testing

The test environment

What to test

Measuring latency

Comparative latency measurement

Latency measurement under concurrency

Measuring throughput

Average throughput test

The load, stress, and endurance tests

Performance monitoring

Monitoring through logs

Ring (web) monitoring

Introspection

JVM instrumentation via JMX

Profiling

OS and CPU/cache-level profiling

I/O profiling

Summary

## 7. Performance Optimization

Project setup

Software versions

Leiningen project.clj configuration

Enable reflection warning

Enable optimized JVM options when benchmarking

Distinguish between initialization and runtime

## Identifying performance bottlenecks

Latency bottlenecks in Clojure code

Measure only when it is hot

Garbage collection bottlenecks

Threads waiting at GC safepoint

Using jstat to probe GC details

Inspecting generated bytecode for Clojure source

Throughput bottlenecks

## Profiling code with VisualVM

### The Monitor tab

The Threads tab

The Sampler tab

Setting the thread name

The Profiler tab

The Visual GC tab

The Alternate profilers

## Performance tuning

Tuning Clojure code

CPU/cache bound

Memory bound

Multi-threaded

I/O bound

JVM tuning

Back pressure

## Summary

# 8. Application Performance

## Choosing libraries

Making a choice via benchmarks

Web servers

Web routing libraries

Data serialization

JSON serialization

JDBC

## Logging

## Why SLF4J/LogBack?

### The setup

Dependencies

The logback configuration file

Optimization

### Data sizing

Reduced serialization

Chunking to reduce memory pressure

Sizing for file/network operations

Sizing for JDBC query results

### Resource pooling

JDBC resource pooling

### I/O batching and throttling

JDBC batch operations

Batch support at API level

Throttling requests to services

### Precomputing and caching

### Concurrent pipelines

Distributed pipelines

### Applying back pressure

Thread pool queues

Servlet containers such as Tomcat and Jetty

HTTP Kit

Aleph

### Performance and queueing theory

Little's law

Performance tuning with respect to Little's law

### Summary

## 3. Module 3

### 1. Working with Sequences and Patterns

Defining recursive functions

Thinking in sequences

Using the seq library

Creating sequences

- [Transforming sequences](#)
- [Filtering sequences](#)
- [Lazy sequences](#)
- [Using zippers](#)
- [Working with pattern matching](#)
- [Summary](#)
- 2. Orchestrating Concurrency and Parallelism**
  - [Managing concurrent tasks](#)
    - [Using delays](#)
    - [Using futures and promises](#)
  - [Managing state](#)
    - [Using vars](#)
    - [Using refs](#)
    - [Using atoms](#)
    - [Using agents](#)
  - [Executing tasks in parallel](#)
    - [Controlling parallelism with thread pools](#)
  - [Summary](#)
- 3. Parallelization Using Reducers**
  - [Using reduce to transform collections](#)
    - [What's wrong with sequences?](#)
    - [Introducing reducers](#)
  - [Using fold to parallelize collections](#)
  - [Processing data with reducers](#)
  - [Summary](#)
- 4. Metaprogramming with Macros**
  - [Understanding the reader](#)
  - [Reading and evaluating code](#)
  - [Quoting and unquoting code](#)
  - [Transforming code](#)
    - [Expanding macros](#)
    - [Creating macros](#)
    - [Encapsulating patterns in macros](#)
    - [Using reader conditionals](#)

[Avoiding macros](#)

[Summary](#)

## [5. Composing Transducers](#)

[Understanding transducers](#)

[Producing results from transducers](#)

[Comparing transducers and reducers](#)

[Transducers in action](#)

[Managing volatile references](#)

[Creating transducers](#)

[Summary](#)

## [6. Exploring Category Theory](#)

[Demystifying category theory](#)

[Using monoids](#)

[Using functors](#)

[Using applicative functors](#)

[Using monads](#)

[Summary](#)

## [7. Programming with Logic](#)

[Diving into logic programming](#)

[Solving logical relations](#)

[Combining logical relations](#)

[Thinking in logical relations](#)

[Solving the n-queens problem](#)

[Solving a Sudoku puzzle](#)

[Summary](#)

## [8. Leveraging Asynchronous Tasks](#)

[Using channels](#)

[Customizing channels](#)

[Connecting channels](#)

[Revisiting the dining philosophers problem](#)

[Using actors](#)

[Creating actors](#)

[Passing messages between actors](#)

[Handling errors with actors](#)

[Managing state with actors](#)  
[Comparing processes and actors](#)

[Summary](#)

## [9. Reactive Programming](#)

[Reactive programming with fibers and dataflow variables](#)

[Using Reactive Extensions](#)

[Using functional reactive programming](#)

[Building reactive user interfaces](#)

[Introducing Om](#)

[Summary](#)

## [10. Testing Your Code](#)

[Writing tests](#)

[Defining unit tests](#)

[Using top-down testing](#)

[Testing with specs](#)

[Generative testing](#)

[Testing with types](#)

[Summary](#)

## [11. Troubleshooting and Best Practices](#)

[Debugging your code](#)

[Using tracing](#)

[Using Spyscope](#)

[Logging errors in your application](#)

[Thinking in Clojure](#)

[Summary](#)

## [A. References](#)

[Bibliography](#)

[Index](#)

# Clojure: High Performance JVM Programming

---

# Clojure: High Performance JVM Programming

Copyright © 2017 Packt Publishing

All rights reserved. No part of this course 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 course to ensure the accuracy of the information presented. However, the information contained in this course is sold without warranty, either express or implied. Neither the authors, 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 course.

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

Published on: January 2017

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78712-959-7

[www.packtpub.com](http://www.packtpub.com)

# Credits

## Authors

Eduardo Díaz

Shantanu Kumar

Akhil Wali

## Reviewers

Ning Sun

Nate West

Eduard Bondarenko

Matjaz Gregoric

Matt Revelle

## Content Development Editor

Rohit Kumar Singh

## Graphics

Jason Monteiro

## Production Coordinator

Shraddha Falebhai

# Preface

In the last few years, we have seen a widespread tendency to create new languages for JVM. There are all sorts of new languages with different paradigms and different ways of working. Clojure is one of those languages, one that we believe is worth learning.

The Java Virtual Machine plays a huge role in the performance of the Clojure programs. Clojure brings functional programming to the Java Virtual Machine (JVM), and also to web browsers through ClojureScript.

With the recent rise of parallel data processing and multicore architectures, functional programming languages have become more popular for creating software that is both provable and performant. Like other functional programming languages, Clojure focuses on the use of functions and immutable data structures for writing programs. Clojure also adds a hint of Lisp through the use of symbolic expressions and a dynamic type system.

## What this learning path covers

[Module 1](#), *Clojure for Java Developers*, introduces you to Clojure and how opinionated it is. You will learn why immutable objects are not only possible, but it is a good idea to use them. You will also learn about functional programming and see how it fits the concept of immutable programs. You will understand the very powerful idea of representing your code as a data structure of the same language. Further, we will find similarities and differences from the Java language that you already know, so you can understand how the Clojure world works.

[Module 2](#), *Clojure High Performance Programming, Second Edition*, increases the focus on the JVM tools for performance management, and it explores how to use those. This module will equip the you with performance measurement and profiling tools and with the know-how of

analyzing and tuning the performance characteristics of Clojure code.

[Module 3](#), , *Mastering Clojure*, walks you through the interesting features of the Clojure language. We will also discuss some of the more advanced and lesser known programming constructs in Clojure. Several libraries from the Clojure ecosystem that we can put to practical use in our own programs will also be described. You won't need to be convinced any more about the elegance and power of the Clojure language by the time you've finished this module.

# What you need for this learning path

You need the Java SDK. You should be able to run samples on any OS; our samples should be easier to follow in environments where there is a shell available. (We focus mainly on Mac OS X.)

You should acquire Java Development Kit version 8 or higher (which you can obtain from

<http://www.oracle.com/technetwork/java/javase/downloads/>) for your operating system to work through all the examples. This book discusses the Oracle HotSpot JVM, so you may want to get Oracle JDK or OpenJDK (or Zulu) if possible. You should also get the latest Leiningen version from <http://leinigen.org/>, and JD-GUI from <http://jd.benow.ca/>.

You'll also need a text editor or an integrated development environment (IDE). If you already have a text editor that you prefer, you can probably use it. Navigate to <http://dev.clojure.org/display/doc/Getting+Started> for a list of environment-specific plugins to write code in Clojure. If you don't have a preference, it is suggested that you use Eclipse with Counterclockwise (<http://doc.ccw-ide.org/>) or Light Table (<http://lighttable.com/>). Some examples in this book will also require a web browser, such as Chrome (42 or above), Firefox (38 or above), or Microsoft Internet Explorer (9 or above).

# Who this learning path is for

This learning path is targeted at Java developers who are comfortable with building applications and now want to leverage the power of Clojure.

# Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this course—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail <[feedback@packtpub.com](mailto:feedback@packtpub.com)>, and mention the course's title in 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 at [www.packtpub.com/authors](http://www.packtpub.com/authors).

# Customer support

Now that you are the proud owner of a Packt course, 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 this course from your account at <http://www.packtpub.com>. If you purchased this course elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the course in the **Search** box.
5. Select the course for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this course from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the course's webpage at the Packt Publishing website. This page can be accessed by entering the course's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac

- 7-Zip / PeaZip for Linux

The code bundle for the course is also hosted on GitHub at <https://github.com/PacktPublishing/Clojure-High-Performance-JVM-Programming>. We also have other code bundles from our rich catalog of books, videos, and courses available at <https://github.com/PacktPublishing/>. Check them out!

## 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 courses—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this course. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your course, 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 to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the course in the search field. The required information will appear under the **Errata** section.

## Piracy

Piracy of copyrighted 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](mailto: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

If you have a problem with any aspect of this course, you can contact us at [<questions@packtpub.com>](mailto:questions@packtpub.com), and we will do our best to address the problem.

# Part 1. Module 1

*Clojure for Java Developers*

*Transition smoothly from Java to the most widely used functional  
JVM-based language – Clojure*

# Chapter 1. Getting Started with Clojure

Welcome to the world of Clojure! If you are here, you probably know a little about Lisp or Clojure, but you don't really have an idea of how things work in this world.

We will get to know Clojure by comparing each feature to what you already know from Java. You will see that there are lists, maps and sets just like in Java, but they are immutable. To work with these kinds of collections, you need a different approach; a different paradigm.

This is what we will try to accomplish in this book, to give you a different way to approach problems. We hope you end up using Clojure in your every day life, but if you don't, we hope you use a new approach toward problem solving.

In this chapter, we will cover the following topics:

- Getting to know Clojure
- Installing Leiningen
- Using a **Read Eval Print Loop (REPL)**
- Installing and using Cursive Clojure
- Clojure's simple syntax
- Clojure's data types and their relationship to the JVM's data types
- Special syntax for functions

## Getting to know Clojure

Before getting started with Clojure, you should know some of its features and what it shares with Java.

Clojure is a programming language that inherits a lot of characteristics

from Lisp. You might think of Lisp as that weird programming language with all the parentheses. You need to keep in mind that Clojure chooses to embrace functional programming. This makes it very different from current mainstream programming languages. You will get to know about immutable data structures and how to write programs without changing variable values.

You will also find that Clojure is a dynamic programming language, which makes it a little easier and faster to write programs than using statically typed languages. There is also the concept of using a REPL, a tool that allows you to connect to a program running environment and change code dynamically. It is a very powerful tool.

At last, you will find out that you can convert Clojure to anything you like. You can create or use a statically typed system and bend the language to become what you like. A good example of this is the `core.typed` library, which allows you to specify the type information without adding support to the compiler.

# Installing Leiningen

We are used to having certain tools to help us build our code, such as Ant, Maven, and Gradle.

In the Clojure ecosystem, the de facto standard for dependency and build management is Leiningen (affectionately named after the short story "Leiningen versus the Ants", which I recommend reading at [http://en.wikipedia.org/wiki/Leiningen\\_Versus\\_the\\_Ants](http://en.wikipedia.org/wiki/Leiningen_Versus_the_Ants)); Leiningen strives to be a familiar to Java developers, it gets the best ideas from Maven, like: convention over configuration. It also gets ideas from Ant like custom scripting and plugins.

Installing it is very simple, let's check how to do it on Mac OS X (installing on Linux should be the same) using bash as your default shell.

You should also have Java 7 or 8 already installed and configured in your path.

You can check the detailed instructions on the Leiningen project page <http://leinigen.org/>. If you want to get a Leiningen installation up and running, this is what you would have to do:

```
curl -O
https://raw.githubusercontent.com/technomancy/lein/stable/bin/lein
# The next step just set up the lein script in your path, you can
do it any way you wish
mv lein ~/bin
echo "export PATH=$PATH:~/bin/">> ~/.bashrc
source ~/.bashrc
# Everything should be running now, let's test it
lein help
```

The first time you run the `lein` command, it downloads everything needed from the internet. This makes it very easy to distribute your code, you can

even include the `lein` script with your own projects and make it easier for other developers to get up and running, the only real requirement is the JDK.

# Using a REPL

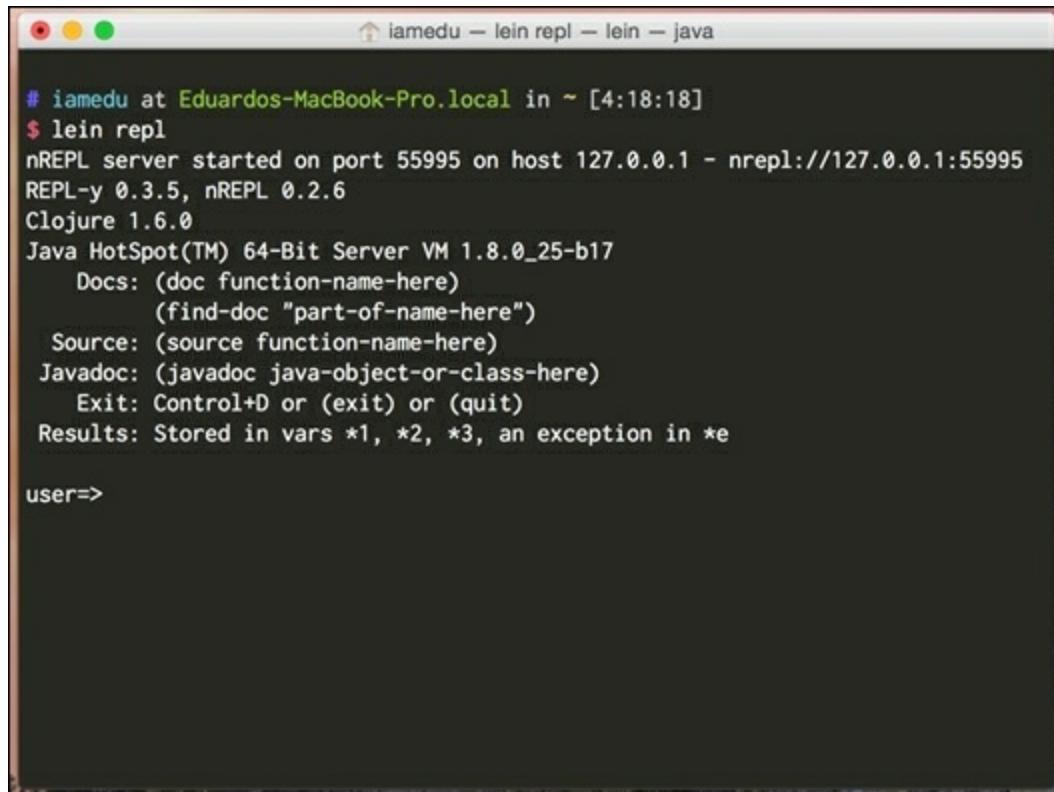
One of the main advantages of Clojure (and Lisp) is interactive development, the REPL is the base of what can be achieved with interactive programming, it allows you to connect to a running VM running Clojure and execute or modify code on the fly.

There is a story about how NASA was able to debug and correct a bug on a \$100 million piece of hardware that was 100 million miles away (<http://www.flownet.com/gat/jpl-lisp.html>).

We have that same power with Clojure and Leiningen and invoking it is very simple, you just need a single command:

```
lein repl
```

This is what you'll get after running the preceding command:



The screenshot shows a terminal window titled "iamedu — lein repl — lein — java". The window contains the following text:

```
# iamedu at Eduardos-MacBook-Pro.local in ~ [4:18:18]
$ lein repl
nREPL server started on port 55995 on host 127.0.0.1 - nrepl://127.0.0.1:55995
REPL-y 0.3.5, nREPL 0.2.6
Clojure 1.6.0
Java HotSpot(TM) 64-Bit Server VM 1.8.0_25-b17
  Docs: (doc function-name-here)
        (find-doc "part-of-name-here")
  Source: (source function-name-here)
Javadoc: (javadoc java-object-or-class-here)
  Exit: Control+D or (exit) or (quit)
Results: Stored in vars *1, *2, *3, an exception in *e

user=>
```

Let's go into a bit more detail, as we can see we are running with the following programs:

- Java 8
- Clojure 1.6.0

We can also get some nice suggestions on how to see documentation, source, Javadoc, and previous errors.

## The nREPL protocol

One particular thing that is important to note is the nREPL protocol; Someday it might grant us the power to go into a machine running 100 million miles away.

When you fire up your REPL, the first thing you see is:

```
nREPL server started on port 55995 on host 127.0.0.1 -  
nrepl://127.0.0.1:55995  
REPL-y 0.3.5, nREPL 0.2.6
```

What it is saying is that there's a Clojure process running an nREPL server on port 55995. We have connected to it using a very simple client that allows us to interact with the Clojure process.

The really interesting bit is that you can connect to a remote host just as easily; let's try attaching an REPL to the same process by simply typing the following command:

```
lein repl :connect localhost:55995
```

Most IDEs have a good integration with Clojure and most of them use this exact mechanism, as clients that work a little more intelligently.

## Hello world

Now that we are inside the REPL, (any of the two) let's try writing our

first expression, go on and type:

```
"Hello world"
```

You should get back a value from the REPL saying Hello world, this is not really a program, and it is the Hello world value printed back by the print phase of the REPL.

Let's now try to write our first Lisp form:

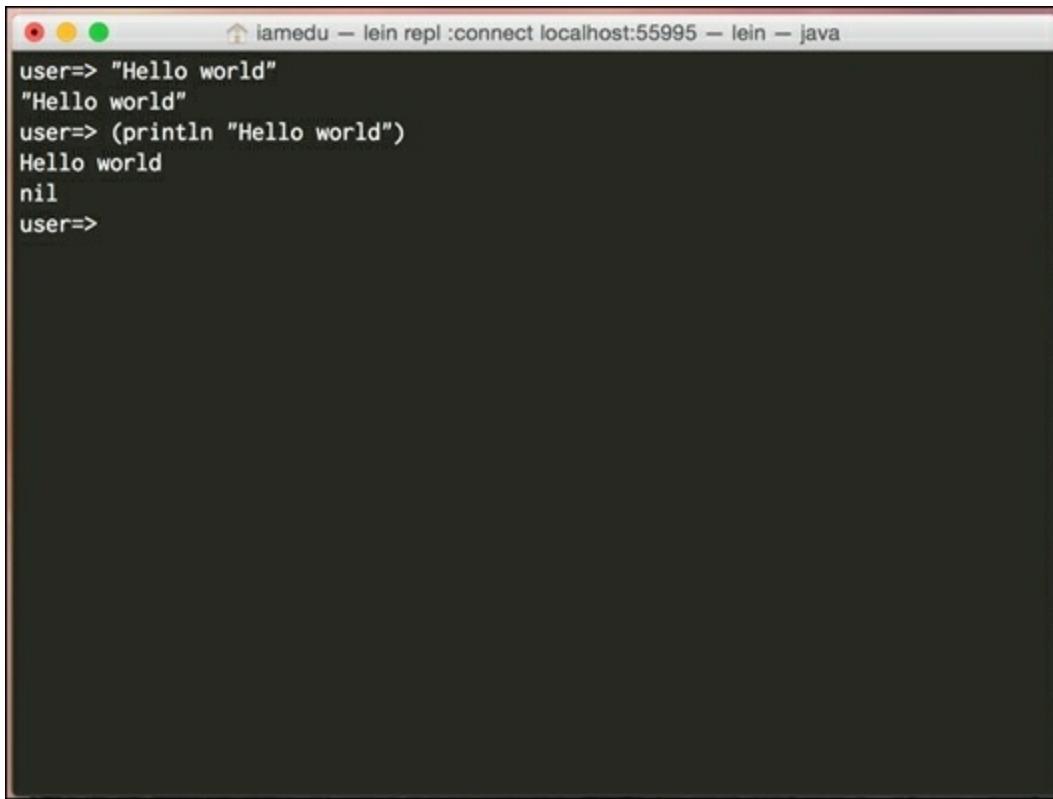
```
(println "Hello world")
```

This first expression looks different from what we are used to, it is called an S-expression and it is the standard Lisp way.

There are a couple of things to remember with S-expressions:

- They are lists (hence, the name, Lisp)
- The first element of the list is the action that we want to execute, the rest are the parameters of that action (one two three).

So we are asking for the string Hello world to be printed, but if we look a bit closer at the output, as shown in the following screenshot, there is a nil that we weren't expecting:



A screenshot of a terminal window titled "iamedu — lein repl :connect localhost:55995 — lein — java". The window contains the following text:

```
iamedu — lein repl :connect localhost:55995 — lein — java
user=> "Hello world"
"Hello world"
user=> (println "Hello world")
Hello world
nil
user=>
```

The reason for this is that the `println` function returns the value `nil` (Clojure's equivalent for null) after printing `Hello world`.

## Note

In Clojure, everything has a value and the REPL will always print it back for you.

# REPL utilities and conventions

As we saw, the Leiningen nREPL client prints help text; but how does that work? Let's explore some of the other utilities that we have.

Try each of them to get a feeling of what it does with the help of the following table:

Function	Description	Sample

doc	Prints out a function's docstring	(doc println)
source	Prints a function's source code, it must be written in Clojure	(source println)
javadoc	Open the javadoc for a class in the browser	(javadoc java.lang.Integer)

Let's check how these functions work:

```

user=> (javadoc java.util.List)
;; Should open the javadoc for java.util.List

user=> (doc doc)
-----
clojure.repl/doc
([name])
Macro
  Prints documentation for a var or special form given its name
nil

user=> (source doc)
(defmacro doc
"Prints documentation for a var or special form given its name"
{:added "1.0"}
[name]
(if-let [special-name ('{& fn catch try finally try} name)]
  (#'print-doc (#'special-doc special-name))
  (cond
    (special-doc-map name) `(#'print-doc (#'special-doc
`~name))
    (find-ns name) `(#'print-doc (#'namespace-doc (find-ns
`~name)))
    (resolve name) `(#'print-doc (meta (var ~name))))))
nil

```

What you are seeing here is metadata pertaining to the `doc` function; Clojure has the ability to store metadata about every function or `var` you use. Most of the Clojure core functions include a doc string and the source of the function and this is something that will become very handy in your day to day work.

Besides these functions, we also get easy access to the latest three values and the latest exceptions that happened in the REPL, let's check this out:

```
user=> 2
2
user=> 3
3
user=> 4
4
user=> (* *1 *2 *3) ;; We are multiplying over here the last
three values
24 ;;We get 24!
user=> (/ 1 0) ;; Let's try dividing by zero
ArithmaticException Divide by zero clojure.lang.Numbers.divide
(Numbers.java:156)
user=> *e
#<ArithmaticException java.lang.ArithmaticException: Divide by
zero>

user=> (.getMessage *e)
"Divide by zero"
```

## Note

`*e` gives you access to the actual plain old Java exception object, so you can analyze and introspect it at runtime.

You can imagine the possibilities of being able to execute and introspect code with this, but what about the tools that we are already used to? How can we use this with an IDE?

Let's check now how to create a new Clojure project, we'll use Leiningen from the command line to understand what is happening.

# Creating a new project

Leiningen can help us create a new project using templates, there is a wide variety of templates available and you can build and distribute your own in Maven.

Some of the most common types of templates are:

- Creating a `.jar` library (the default template)
- Creating a command-line app
- Creating a Clojure web app

Let's create a new Clojure command-line app and run it:

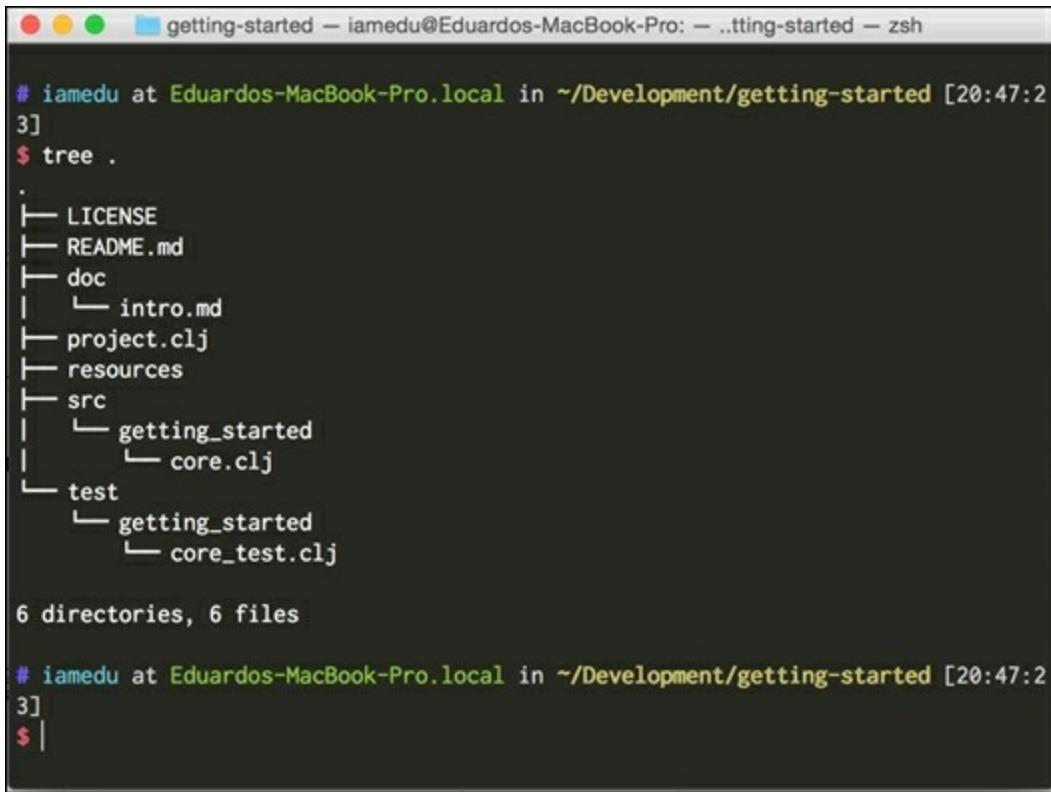
```
lein new app getting-started
cd getting-started
lein run
# Hello, world!
```

## Project structure

Leiningen is similar to other Java development tools; it uses a similar convention and allows for heavy customizations in the `project.clj` file.

If you are familiar with Maven or Gradle, you can think of it as `pom.xml` or `build.gradle` respectively.

The following screenshot is the project structure:



```
# iamedu at Eduardos-MacBook-Pro.local in ~/Development/getting-started [20:47:2
3]
$ tree .
.
├── LICENSE
├── README.md
├── doc
│   └── intro.md
├── project.clj
└── resources
├── src
│   └── getting_started
│       └── core.clj
└── test
    └── getting_started
        └── core_test.clj

6 directories, 6 files

# iamedu at Eduardos-MacBook-Pro.local in ~/Development/getting-started [20:47:2
3]
$ |
```

As you can see in the preceding screenshot, there are four main folders:

- `resources`: It holds everything that should be in the class path, such as files, images, configuration files, properties files, and other resources needed at runtime.
- `src`: Your Clojure source files; they are ordered in a very similar fashion to the `classpath`.
- `dev-resources`: Everything that should be in the `classpath` in development (when you are running Leiningen). You can override your "production" files here and add files that are needed for tests to run.
- `test`: Your tests; this code doesn't get packaged but it is run every time you execute the Leiningen test.

## Creating a standalone app

Once your project is created, you can build and run a Java standalone command-line app quite easily, let's try it now:

```
lein uberjar
java -jar target/uberjar/getting-started-0.1.0-SNAPSHOT-
standalone.jar
# Hello, World!
```

As you can see, it is quite easy to create a standalone app and it is very similar to using Maven or Gradle.

# Using Cursive Clojure

Java already has some great tools to help us be more productive and write higher quality code and we don't need to forget about those tools. There are several plugins for Clojure depending on what your IDE is. Have a look at them from the following table:

IDE	Plugins
IntelliJ	Cursive Clojure, La Clojure
NetBeans	NetBeans Clojure (works with NetBeans 7.4)
Eclipse	CounterClockwise
Emacs	Cider
VIM	vim-fireplace, vim-leiningen

A lot of people writing real Clojure code use Emacs and I actually like using vim as my main development tool, but don't worry, our main IDE will be IntelliJ + Cursive Clojure throughout the book.

## Installing Cursive Clojure

You can check the full documentation for Cursive at their website (<https://cursiveclojure.com/>), it is still under development but it is quite stable and a great aid when writing Clojure code.

We are going to use the latest IntelliJ Community Edition release, which at the time of this writing is version 14.

You can download IntelliJ from here  
<https://www.jetbrains.com/idea/download/>.

Installing Cursive Clojure is very simple, you need to add a repository for IntelliJ. You'll find the instructions to your specific IntelliJ version here: <https://cursiveclojure.com/userguide/>.

After you have installed Cursive Clojure, we are ready to go.

Now, we are ready to import our getting started project into Cursive Clojure.

## Note

Cursive Clojure doesn't currently have support to create Leiningen projects from within the IDE; however, support is great in order to import them.

Here is how you will do it:

1. Click on **File**.
2. Import project.
3. Look for your project.
4. Open the folder or the `project.clj` file.
5. Follow the **Next** steps in the IDE.

Now, we are ready to go, you can use the Cursive Clojure as your main development tool. There are a few more things to do with your IDE but I recommend you to look for them; they are important and will come in handy:

- To know how to execute the project
- To know how to execute the tests
- To open an REPL connected to some project.
- The key binding to execute some given piece of code (run form before cursor in REPL)
- The key binding to execute a given file (load file in REPL)

One important part of Clojure programming is that it can modify and

reevaluate code in runtime. Check the manual of your current version of Clojure and check for the structural editing section (<https://cursiveclojure.com/userguide/paredit.html>). It is one of the most useful functionalities of Clojure IDEs and a direct consequence of the Clojure syntax.

I recommend you to check other functionalities from the manual. I really recommend checking the Cursive Clojure manual, it includes animations of how each functionality works.

You will use the last two key bindings quite a lot, so it is important to set them up correctly. There is more information about keybindings at <https://cursiveclojure.com/userguide/keybindings.html>.

# Getting started with Clojure code and data

Let's take a deep dive into Clojure's syntax now, it is pretty different from other languages but it is actually much simpler. Lisps have a very regular syntax, with few special rules. As we said earlier, Clojure code is made of S-expressions and S-expressions are just lists. Let's look at some examples of lists to become familiar with lists in Lisp.

```
(1 2 3 4)
(println "Hello world")
(one two three)
("one" two three)
```

All of the above are lists, but not all of them are valid code. Remember, only lists where the first element is a function can be considered valid expressions. So, here only the following could be valid expressions:

```
(println "Hello world")
(one two three)
```

If `println` and `one` are defined as functions.

Let's see a piece of Clojure code, to finally explain how everything works.

```
(defn some-function [times parameter]
  "Prints a string certain number of times"
  (dotimes [x times]
    (println parameter)))
```

## Lists in Clojure

Clojure is based around "forms" or lists. In Clojure, same as every Lisp, the way to denote a list is with parentheses, so here are some examples of lists in the last code:

```
(println parameter)
(dotimes [x times] (println parameter))
(defn some-function [times parameter] (dotimes [x times] (println
parameter)))
```

Lists are one data type in Clojure and they are also the way to express code; you will learn later about all the benefits of expressing code as data. The first one is that it is really simple, anything you can do must be expressed as a list! Let's look at some other examples of executable code:

```
(* 1 2 3)
(+ 5 9 7)
(/ 4 5)
(- 2 3 4)
(map inc [1 2 3 4 5 6])
```

I encourage you to write everything into the REPL, so you get a good notion of what's happening.

## Operations in Clojure

In Clojure, MOST of the executable forms have this structure:

```
(op parameter-1 parameter-2 ...)
```

**op** is the operation to be executed followed by all the parameters it needs, let's analyze each of our previous forms in this new light:

```
(+ 1 2 3)
```

We are asking to execute the `+` (addition) operation with the parameters `1`, `2`, and `3`. The expected result is `6`.

Let's analyze something a bit more complicated:

```
(map inc [1 2 3 4 5 6])
```

In this, we are asking to execute the `clojure.core/map` function with two parameters:

- `inc` is a function name, it takes a number and increments it
- `[1 2 3 4 5 6]` is a collection of numbers

Map applies the `inc` function to each member of the passed collection and returns a new collection, what we expect is a collection containing `[2 3 4 5 6 7]`.

## Functions in Clojure

Now let's check how a function definition is essentially the same as the previous two forms:

```
(defn some-function [times parameter]
  "Prints a string certain number of times"
  (dotimes [x times]
    (println parameter)))
```

The `defn` is the operation that we are asking for. It has several parameters, such as:

- `some-function` is the name of the function that we are defining
- `[times parameter]` is a collection of parameters
- `"Prints a string certain number of times"` is the docstring, it is actually an optional parameter
- `(dotimes [x times] (println parameter))` is the body of the function that gets executed when you call `some-function`

The `defn` calls a function into existence. After this piece of code is executed, `some-function` exists in the current namespace and you can use it with the defined parameters.

The `defn` is actually written in Clojure and supports a few nice things. Let's now define a multi-arity function:

```
(defn hello
  ([] (hello "Clojure"))
  ([name] (str "Hello " name)))
```

Over here we are defining a function with two bodies, one of them has no arguments and the other one has one argument. It is actually pretty simple to understand what's happening.

Try changing the source in your project's `core.clj` file similar to the following example:

```
(ns getting-started.core
  (:gen-class))

(defn hello
  ([] (hello "Clojure"))
  ([name] (str "Hello " name)))

(defn -main
  "I don't do a whole lot ... yet."
  [& args]
  (println "Hello, World!")
  (println (hello))
  (println (hello "Edu")))
```

Now run it, you'll get three different Hello outputs.

As you can see, Clojure has a very regular syntax and even if it's a little strange for newcomers, it is actually quite simple.

Here, we have used a few data types that we haven't properly introduced; in the next section we'll take a look at them.

# Clojure's data types

Now is when everything you know about Java pays off; even the list forms that you saw earlier implement the `java.util.List` interface. Clojure was designed to be embeddable and to have a great integration with the host platform, so it's only natural that you can use everything you already know about Java types and objects.

There are two data types in Clojure: scalars and collections.

## Scalars

In every language you need primitive types; you use them in everyday life as they represent numbers, strings, and Booleans. These primitive types are called scalars in the Clojure world.

Clojure has a couple of very interesting types like ratios and keywords. In the following table, you get to know the different types of scalars, how they compare to Java and a simple example of how to use each of them.

Clojure data type	Java data type	Sample	Description
String	String	"This is a string" "This is a multiline string"	A string of characters; in Clojure you can use multiline strings without a problem
Boolean	Boolean	true false	Literal Boolean values
Character	Character	\c \u0045 ;; Unicode char	Character values, they are <code>java.lang.Character</code> instances, you can

		45 E	define Unicode characters
Keywords	Doesn't exist in java	:key :sample :some-keyword	They evaluate themselves and they are often used as keys. They are also functions that look for themselves in a map.
Number	Numbers are automatically handled as <code>BigDecimal</code> , <code>BigInteger</code> or lower precision depending on what's necessary	<code>42N ;;BigInteger</code> <code>42 ;;long</code> <code>0.1M ;;BigDecimal</code>	It is important to remember the trade-offs of Java numbers, if precision is important, you should always use big decimals and <code>bignumerics</code> .
Ratio	Doesn't exist	<code>22/7</code>	Clojure provides great numerical precision; if necessary it can retain the ration and execute exact operation. The tradeoff when using ratios is speed.
Symbol	Doesn't exist	some-name	Symbols are identifiers in Clojure, very similar to a variable name in Java.
nil	null	nil	The null value
Regular expressions	<code>java.util.regex.Pattern</code>	<code>#"\d"</code>	Regular expressions, in Clojure you get free syntax to define regular expressions, but in the end it is a plain old Java reggae Pattern

## Collection data types

In Clojure there are two types of collections: sequential and associative collections. Sequential are things you can iterate, such as lists. Associative collections are maps, sets, and things you can access by a certain index. Clojure's collections are fully compatible with Java and it can even implement the `java.util` interfaces, such as `java.util.List` and `java.util.Map`.

One of the main characteristics of collections in Clojure is that they are immutable; it has a lot of benefits that we'll see later.

Let's have a look at the characteristics of each collection data type available in Clojure and compare them with Java with the help of a sample (in Clojure) and its description.

Clojure data type	Java data type	Sample	Description
List	List	(1 2 3 4 5)	A simple list, notice the quote character before the list, if you don't specify it Clojure will try to evaluate the form as an instruction
Vector	Array	[1 2 3 4 5]	It is the main workhorse in Clojure, it is similar to an array because you can access elements in a random order
Set	HashSet	#{}{1 2 3 4}	A normal Java hash set
Map	HashMap	{:key 5 :key-2 "red"}	A Clojure map

# Summary

As you can see, Clojure has a mature development environment that is always evolving. You can set up command-line tools and your IDE in a very similar fashion to the way you will do in a normal Java development.

We also learned a little about Clojure's regular syntax, its data types and how they relate to Java's own data types.

Overall, you should now be comfortable with:

- Lisp syntax
- Creating a Leiningen project from scratch
- Running and packaging your code
- Importing a Leiningen project into IntelliJ
- Using the REPL
- Knowing the relationship between Clojure types and Java types

In the next chapter, we will get an idea of how to organize our code and how that organization takes advantage of Java packages.

# Chapter 2. Namespaces, Packages, and Tests

We now have a working installation of Clojure and IntelliJ.

As a Java developer, you are used to working with classes as the minimal unit of organization. Clojure has a very different sense and gives you different tools to organize your code.

For starters, you should keep in mind that code and data are separate; you don't have a minimal unit with attributes and functions that work over those attributes. Your functions can work on any data structure that you wish, as long as you follow the rules of how the function works.

In this chapter, we will start writing some simple functions to illustrate how separation of functions and data works and we will have a look at the tools Clojure gives us to make the separation.

In this chapter, we will cover the following topic:

- How namespaces work compared to the classpath and Java packages
- Unit tests
- More Clojure examples and syntax

## Namespaces in Clojure

Clojure namespaces might be familiar to you, as a Java developer, and for a very good reason, they have a very deep relationship with Java's packages and the classpath.

First of all, let's review what we already know from Java.

## Packages in Clojure

The Java code is organized in packages, a package in Java is a namespace that allows you to group a set of similar classes and interfaces.

You can think of a package as something very similar to a folder in your computer.

The following are some common packages that you use a lot when programming in Java:

- `java.lang`: Everything that's native to Java, including basic types (integer, long, byte, boolean, character, string, number, short, float, void, and class), the basic threading primitives (Runnable, Thread), the basic primitives for exceptions (Throwable, Error, Exception), the basic exceptions and errors (`NoSuchMethodError`, `OutOfMemoryError`, `StackOverflowError`, and so on) and runtime access classes like `Runtime` and `System`.
- `java.io`: This package includes the primitives for input and output, such as console, file, readers, input streams, and writers.
- `java.util`: This is one of the most heavily used packages besides `java.lang`. This includes the classic data structures (map, set, list) along with the most common implementations of such data structures. This package also includes utilities like properties tools, scanner for reading from various input resources, `ServiceLoader` to load custom services from the `ClassLoader`, `UUID` generator, timers, and so on.
- `java.util.logging`: The logging utilities, you normally use them to give you different levels of alert, from a debug to serious conditions.
- `java.text`: These are utilities to manage text, dates, and numbers in a language independent way.
- `javax.servlet`: This includes the primitives to create web apps and deployment in standard web containers.

Each one of these packages groups several related functionalities, the `java.lang` package is particularly important, since it has every Java core type, such as string, long, and integer. Everything inside the `java.lang` package is available automatically everywhere.

The `java.lang` package provides a bit more than just code organization, it also provides access security. If you remember about Java, there are three security access levels:

- private
- public
- protected

In the case of packages, we are concerned with the protected level of access. The classes in the same package allow every other class in the same package to access its protected attributes and methods.

There are also ways to analyze a package in runtime but they are involved and allow for very little to be done.

Packages are implemented at the top of Java's classpath and the classloader.

# The classpath and the classloader

Java was designed to be modular and for that it needs some way to load your code easily. The answer to this was the classloader, the classloader allows you to read resources from every entry of the classpath; you can look at resources in the classpath as a hierarchical structure similar to the file system.

The classloader is just a list of entries; each entry can be a directory in the filesystem or a JAR file. At this point, you should also know that JAR files are just zip files.

The classloader will treat each entry as a directory (JAR files are just zipped directories) and it will look for files in each directory.

There are a lot of concepts here to remember, let's try to summarize them:

- JAR files are ZIP files; they might contain several classes, properties, files, and so on.
- The classpath is a list of entries; each entry is a JAR file or a system directory.
- The classloader looks for resources in each entry of the classpath, so you can think of classpath resources as a combination of all the directories in the classpath (repeated resources are not overwritten)

If you are not already familiar with how classloaders look for resources in classpath entries, this is the general process; let's imagine that you want to load a class: `test.Test`, what happens next?

1. You tell the JVM that you want to load `test.Test`.
2. The JVM knows to look for the `test/Test.class` file.
3. It starts looking for it in each entry of the classpath.
4. If the resource is a ZIP file, it "unzips" the directory.
5. It looks for the resource in the directory which represents the entry.

If you were to see the default classpath resources, you will probably see something, such as:

```
java:  
  lang:  
    String.class  
    ...  
  io:  
    IOException.class  
    ...  
  util:  
    List.class
```

It is important to note that each entry in the classpath doesn't just store class files, it can actually store any type of resource, It is a commonplace to store configuration files, such as .properties or .xml.

Nothing forbids you from storing anything else in the classpath resources, such as images, mp3 or even code! You can read and access anything from the classpath's resource like you can from the filesystem at runtime. The one thing that you can't do is modify the classpath's resource contents (at least not without some esoteric magic).

## Back to Clojure namespaces

Now that we have had our little review of how packages and the classpaths work in Java, it's time to go back to Clojure. You should understand that Clojure attempts to make the hosting platform transparent; this means a couple of very important things:

- Anything that you can do with the classpath from Java, you can also do with Clojure (you can read configuration files, images, etc).
- Namespaces use the classpath just as Java does with packages, which makes them easy to understand. Nevertheless, don't underestimate them, Clojure namespace declarations can be more involved.

Let's get practical and play a little with namespaces.

# Playing with namespaces

Lets create a new Playground, in order to create it use the following command:

```
lein new app ns-playground
```

You can open this project with IntelliJ, as we did in [Chapter 1, Getting Started with Clojure](#).

Let's look in detail at what was created for us:

```
.  
├─ CHANGELOG.md  
├─ LICENSE  
├─ README.md  
├─ doc  
│ └─ intro.md  
├─ project.clj  
├─ resources  
└─ src  
    └─ ns_playground  
        └─ core.clj  
└─ test  
    └─ ns_playground  
        └─ core_test.clj  
  
6 directories, 7 files
```

This project structure looks similar to Java projects, we have:

- **resources**: These are the non-source files that get added to the classpath
- **src**: Our source code
- **test**: Our testing code

The code inside `src` and `test` is already structured into namespaces: by having a quick look, we could say that the name of the namespace is `ns_playground`. Let's check the source code:

```
(ns ns-playground.core
  (:gen-class))

(defn -main
  "I don't do a whole lot ... yet."
  [& args]
  (println "Hello, World!"))
;; Code for src/ns_playground/core.clj
```

## Tip

`:gen-class` was added here in order to create a Java class and allow the Java interpreter to start a static `main` method. It is not needed if you don't intend to create a standalone program.

We can see that the `(ns ns-playground.core)` form has been used at the top, as you might have guessed, this is how we declare a namespace in Clojure.

If you are observant, you will notice something odd; the namespace has a dash instead of an underscore like the folder.

There are some reasons that lead to this:

- Clojure like most lisp variable names can have dashes in it (it is actually the preferred style to name the variables, as opposed to camel case in Java).
- Every namespace in Clojure is represented as a package containing several Java classes. The namespace is used as a name of the Java package and as you know, the dash is not acceptable in class or package names; so every filename and folder name must have low dashes.

## Tip

Due to the nature of Lisp, you can use dashes in variable names (they will get converted to underscores at compile time). In fact, this is the recommended way to name your variables. In Clojure, (and most Lisps) `some-variable-name` is a more idiomatic style than `someVariableName`.

## Creating a new namespace

Let's create a new namespace; in Cursive Clojure it is easy to do so, just right-click on the `ns_playground` package and go to **New | Clojure Namespace**, it asks for a name and we can call it `hello`.

This creates a `hello.clj` file with the following contents:

```
(ns ns-playground.hello)
```

As you can see, namespace creation is quite easy; you can do it by hand with two simple steps:

1. Create a new file; it doesn't have to follow the package naming specification, but it helps to maintain your code order and it is a de facto practice.
2. Add your namespace declaration.

That's it! It is true, even though a namespace definition can become quite complex, as it is the place where you define the Java packages that you wish to import, namespaces or functions from those namespaces that you intend to use. But you will normally just use a subset of those capabilities.

### Tip

Keep in mind that a namespace in Clojure is normally represented by a single file.

For your initial namespaces, I will advice you to have two of those capabilities in mind:

:import	Allows you to import the Java classes from a package that you wish to use
:require	Allows you to bring in whatever Clojure namespace that you wish to use

The syntax of both `require` and the `import` is simple, let's look at a couple of examples before we actually use it.

Let's start with the `import` option:

```
(:import java.util.List)
```

You'll notice that this is similar to what you can do in Java, we are importing the `List` interface here.

The good thing with Clojure is that it allows you to do some more specific things. Let's check how to import two classes at once:

```
(:import [java.util ArrayList HashMap])
```

You can extend this to the number of classes you want to use.

The `require` option uses a similar syntax and then builds some more on it. Let's check requiring a single function from a namespace:

```
(:require [some.package :refer [a-function another-function]])
```

As you can see, it is familiar and the interesting part is when you start importing everything:

```
(:require [some.package :refer [:all]])
```

You can also use a custom name for everything inside your package:

```
(:require [some.package :as s])
```

`;;` And then use everything in the package like this:

```
(s/a-function 5)
```

Or you could even combine different keywords:

```
(:require [some.package :as s :refer [a-function]]))
```

Let's try a bit of what we just learned, using the following code:

```
(ns ns-playground.hello
  (:import [java.util Date]))  
  
(def addition +)  
  
(defn current-date []
  "Returns the current date"
  (new Date))  
  
(defn <3 [love & loved-ones]
  "Creates a sequence of all the {loved-ones} {loved} loves"
  (for [loved-one loved-ones]
    (str love " loves " loved-one)))  
  
(defn sum-something [something & nums]
  "Adds something to all the remaining parameters"
  (apply addition something nums))  
  
(def sum-one (partial sum-something 1))
```

## Note

You must have noticed the `&` operator in the arguments of the `<3` and `sum-something` functions; this allows those functions to receive any number of arguments and we can call them, as shown: `(sum-something 1 2 3 4 5 6 7 8)` or `(sum-something)`. They are called **variadic** functions. In Java you will call this feature **varargs**.

Everything looks great, but we haven't yet seen how to require and use these functions from some other package. Let's write a test to see how this will be done.

## Working with namespaces on the REPL

A great way of playing with namespaces is by using the REPL and we'll also get the benefit of getting to know it better.

Since we are going to play with namespace, we need to know of a few functions that will help us move between namespaces and require other namespaces. The functions are listed as follows:

Function	Description	Sample usage
in-ns	Sets <code>*ns*</code> to the namespace named by the symbol, creating it if needed.	<code>(in-ns 'ns-playground.core)</code>
require	Loads <code>libs</code> , skipping any that are already loaded.	<code>(require '[clojure.java.io :as io])</code>
import	For each name in <code>class-name-symbols</code> , adds a mapping from name to the class named by <code>package.name</code> to the current namespace.	<code>(import java.util.Date)</code>
refer	refers to all public <code>vars</code> of <code>ns</code> , subject to filters.	<code>(refer 'clojure.string :only '[capitalize trim])</code>

Let's go into the REPL window of our IntelliJ. We can check what namespace we are in with the `*ns*` instruction. Let's try now:

```
*ns*
=> #<Namespace ns-playground.core>
```

Imagine that we need to execute a code and test the code from within the `ns-playground.hello` namespace, we can do that with the `in-ns` function:

```
(in-ns 'ns-playground.hello)
=> #<Namespace ns-playground.hello>
```

We want to know what `str` does, it seems to receive three strings:

```
(str "Hello" "world")
```

```
=>"Hello world"
```

Let's try the `for` form now:

```
(for [el ["element1""element2""element3"]] el)
=> ("element1""element2""element3")

(for [el ["element1""element2""element3"]]
  (str "Hello " el))
=> ("Hello element1""Hello element2""Hello element3")
```

The `for` macro takes a collection of items and returns a new lazy sequence applying the body of the `for` to each element.

Knowing this, understanding the `<3` function is easy, let's try it:

```
(<3 "They""tea")
=> ("They love tea")

(clojure.repl/doc <3)
ns-playground.hello/<3
([& loved-ones])
  Creates a sequence of all the {loved-ones} {loved} loves
```

We've used the REPL to test some simple functions, but let's now try to test something else like reading a properties file from the classpath.

We can add a `test.properties` file to the resources folder with the following contents:

```
user=user
test=password
sample=5
```

Remember to restart the REPL, as the changes to the contents that some piece of the classpath points to are not visible to a running REPL.

Let's try reading our properties file as an input stream, we can use the `clojure.java.io` namespace to do it, and we can check it as shown:

```
(require '[clojure.java.io :as io])
(io/resource "test.properties")
=> #<URL file:/Users/iamedu/Clojure4Java/ns-
playground/resources/test.properties>
(io/input-stream (io/resource "test.properties"))
=> #<BufferedInputStream java.io.BufferedInputStream@2f584e71>
;; Let's now load it into a properties object
(import [java.util Properties])
=> java.util.Properties
(def props (Properties.)) ;; Don't worry about the weird syntax,
we will look it soon.
=> #'ns-playground.core/props
(.load props (io/input-stream (io/resource "test.properties")))
props
=> {"user""user", "sample""5", "test""password"}
```

We can now define our function for reading properties, we can input this into the REPL:

```
(defn read-properties [path]
  (let [resource (io/resource path)
        is (io/input-stream resource)
        props (Properties.)]
    (.load props is)
    (.close is)
    props))
=> #'ns-playground.core/read-properties
(read-properties "test.properties")
=> {"user""user", "sample""5", "test""password"}
```

## Note

The `let` form lets us create local 'variables', instead of using the `(io/resource path)` directly in the code. We can create a reference once and use it through the code. It allows us to use simpler code and to have a single reference to an object.

In the end, we can redefine the `hello` namespace to include everything we've checked, such as this:

```
(ns ns-playground.hello
```

```
(:require [clojure.java.io :as io])
(:import [java.util Date Properties]))  
  
(def addition +)  
  
(defn current-date []
"Returns the current date"
  (new Date))  
  
(defn <3 [love & loved-ones]
"Creates a sequence of all the {loved-ones} {loved} loves"
  (for [loved-one loved-ones]
    (str love " loves " loved-one)))  
  
(defn sum-something [something & nums]
"Adds something to all the remaining parameters"
  (apply addition something nums))  
  
(defn read-properties [path]
  (let [resource (io/resource path)
        is (io/input-stream resource)
        props (Properties.)]
    (.load props is)
    props))  
  
(def sum-one (partial sum-something 1))
```

**Remember to include the `Properties` class in the `import` and to define the `:require` keyword for `clojure.java.io`.**

# Testing in Clojure

Clojure already comes with a unit testing support built-in, as a matter of fact Leiningen has already created a test for us; let's take a look at it right now.

Open the `test/ns_playground/core_test.clj` file, you should be able to see this code:

```
(ns ns-playground.core-test
  (:require [clojure.test :refer :all]
            [ns-playground.core :refer :all]))
(deftest a-test
  (testing "FIXME, I fail."
  (is (= 0 1))))
```

Again, as you can see, we are using `:require` to include functions from the `clojure.test` and the `ns-playground.core` packages.

## Note

Remember, the `:refer :all` works similar to how `char import static clojure.test.*` will work in Java.

## Testing from the command line

Let's first learn how to run these tests. From the command line, you can run:

```
lein test
```

You should get the following output:

```
lein test ns-playground.core-test
lein test :only ns-playground.core-test/a-test
FAIL in (a-test) (core_test.clj:7)
```

```
FIXME, I fail.  
expected: (= 0 1)  
actual: (not (= 0 1))
```

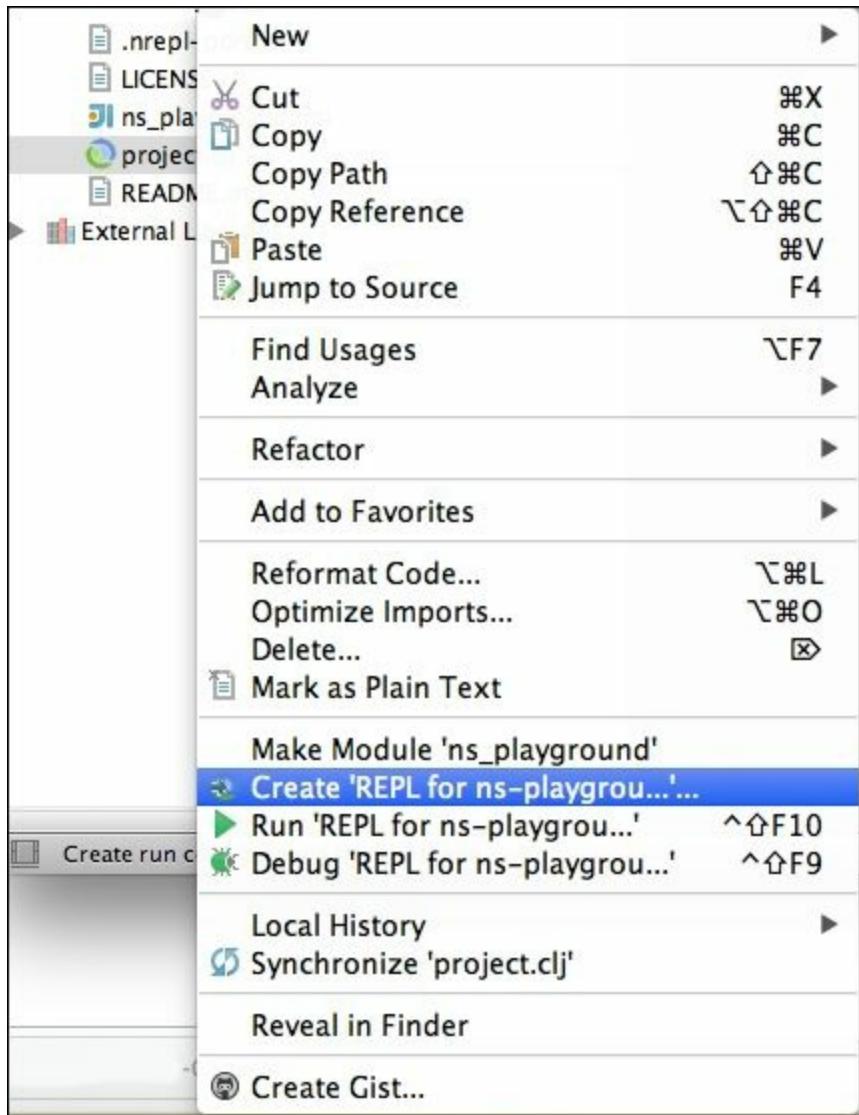
```
Ran 1 tests containing 1 assertions.  
1 failures, 0 errors.  
Tests failed.
```

We see that there is one test failing, we will go back to this in a bit; for now, let's see how to test in IntelliJ.

## Testing in IntelliJ

First of all, we need a new REPL configuration. You can do it as you learned in the previous chapter. You just need to follow the following steps:

1. Right click on the `project.clj` file and select **Create REPL for ns-playground**, as shown in the following screenshot:



2. Then click on **OK** in the next dialog.
3. After that, you should run the REPL again by right clicking the `project.clj` file and selecting **Run REPL for ns-playground**.
4. After that you can run any tests, just open your test file and go to **Tools | Run Tests** in the current NS in REPL. You should see something similar to the following screenshot:

```

(ns ns-playground.core-test
  (:require [clojure.test :refer :all]
            [ns-playground.core :refer :all]))

(deftest a-test
  (testing "FIXME, I fail."
    (is (= 0 1)))

```

5. As you can see, it signals that your test is currently failing. Let's fix it and run our test again. Change the `(is (= 0 1))` line to `(is (= 1 1))`.
6. Now, let's try some real tests for our previously defined functions; don't worry if you can't understand all the code for now, you are not supposed to:

```

(ns ns-playground.hello-test
  (:import [java.util Date])
  (:require [clojure.test :refer :all]
            [ns-playground.hello :as hello :refer [<3]]
            [ns-playground.core :refer :all]))


(defn- lazy-contains? [col element]
  (not (empty? (filter #(= element %) col))))


(deftest a-test
  (testing "DONT FIXME, I don't fail."
    (is (= 42 42))))


(deftest current-date-is-date
  (testing "Test that the current date is a date"
    (is (instance? Date (hello/current-date)))))


(deftest check-loving-collection
  (testing "Check that I love clojure and you"
    (let [loving-seq (<3
                     "I""Clojure""you""doggies""chocolate")]
      (is (not (lazy-contains? loving-seq "I love Vogons"))))
      (is (lazy-contains? loving-seq "I love Clojure")))
      (is (lazy-contains? loving-seq "I love doggies")))
      (is (lazy-contains? loving-seq "I love chocolate")))))

```

```
(is (lazy-contains? loving-seq "I love you"))))
```

## Note

We can't use the Clojure contents function here because it has a different function. It looks for keys in a map.

Run the tests and you'll see that everything passes correctly but there's a lot going on over here, let's go over it little by little:

```
(ns ns-playground.core-test
  (:import [java.util Date])
  (:require [clojure.test :refer :all]
            [ns-playground.hello :as hello :refer [<3]]
            [ns-playground.core :refer :all]))
```

This is the namespace declaration, let's list everything it does:

- It declares the `ns-playground.core-test` package.
- It imports the `java.util.Date` class.
- It makes everything in the `clojure.test` namespace available in the current namespace, if we were in Java we might have used `import static clojure.test.*` to get a similar effect. We can achieve this with the `:refer :all` keywords.
- It makes everything in the `ns-playground.hello` namespace available with the `hello` shortcut but we need to prefix every function or value defined in `ns-playground.hello` with `hello` and it also makes the `<3` function available without a prefix. To generate an alias and make everything available with the `hello` alias, we use the `:as` keyword and then pass a vector to `:refer` to include certain elements.
- It makes everything in the `ns-playground.core` namespace available in the current namespace. We achieve this with the `:refer :all` keywords.

```
(defn- lazy-contains? [col element]
  (not (empty? (filter #(= element %) col))))
```

This is the declaration of a function called `lazy-contains?`, it is a boolean function and it is customary in Clojure to call it a predicate.

## Note

The name of the function including the question mark might be something that looks awkward to you. In Clojure and Lisp, you can use question marks in the names of functions and it is common to do it for functions that return Booleans.

It receives two parameters: `col` and `element`.

The actual body of the function looks a bit complicated but it is actually very simple. Whenever you encounter a function that looks similar to the one mentioned in the preceding section, try to read it from the inside out. The innermost part is, as follows:

```
# (= element %)
```

This is a shorter way of writing an anonymous function which has a single parameter. If we want to write another function that compares its argument against the `element`, without the syntactic sugar, we can do it in the following method:

```
(fn [e1]
  (= element e1))
```

This is an anonymous function or in other words it is a function that has no name, but it works as every other function; we will read more about anonymous functions when we get back to functional programming.

Our anonymous function is a parameter to the following form:

```
(filter # (= element %) col)
```

This new form filters the collection `col` and returns a new collection with only the elements that pass the test. Let's see an example where we have

used the predefined Clojure function `even?`:

```
;; This returns only the even numbers in the collection  
(filter even? [1 2 3 4])  
;;=> (2 4)
```

Our filter function now returns every element in the collection that passes the `#(= element %)` test. So we get every element that is equal to the element passed to `lazy-contains?`.

We then ask if none of the elements equal to `element` in `col` with the following form:

```
(empty? (filter #(= element %) col))
```

But we want to know if there is some element equal to `element`, so at last we negate the previous form:

```
(not (empty? (filter #(= element %) col)))
```

Imagine that if you had to write this in Java (and I asked to add every element that matches the element to a list), you will have something similar to this:

```
List<T> filteredElements = new ArrayList<T>();  
for(T e1 : col) {  
    if(e1 == element) {  
        filteredElements.add(e1);  
    }  
}  
return !filteredElements.isEmpty();
```

There is a big difference, it is more verbose and to understand it we need to "run" the algorithm in our heads. This is called imperative programming, Clojure allows us to do imperative programming as well as functional programming, which is a type of declarative programming. When you get used to it, you'll see that it's easier to reason about than loops.

## Note

Interactive programming means, telling every step of how something should be done to a computer. Declarative programming just asks for a result and doesn't give details of how to achieve it.

The actual tests are simple to understand:

```
(deftest current-date-is-date
  (testing "Test that the current date is a date"
    (is (instance? Date (hello/current-date)))))
```

This test checks the current date returns an instance of `java.util.Date`, the `is` form works as the Java assert instruction:

```
(deftest check-loving-collection
  (testing "Check that I love clojure and you"
    (let [loving-seq (<3 "I""Clojure""you""doggies""chocolate")]
      (is (not (lazy-contains? loving-seq "I love Vogons")))
          (is (lazy-contains? loving-seq "I love Clojure"))
          (is (lazy-contains? loving-seq "I love doggies"))
          (is (lazy-contains? loving-seq "I love chocolate"))
          (is (lazy-contains? loving-seq "I love you")))))
```

This test checks the `<3` function, it checks that the returned collection contains I love Clojure, I love doggies, I love chocolate and I love you and it should not contain I love Vogons.

This test is simple to understand. What might be not so simple to understand is the `<3` function, we'll look into it with the REPL.

# Summary

In this chapter, we got to know some utilities that we can use for better management of our code and we have some more examples of everyday Clojure code. In particular:

- Working of namespace in Clojure and their relation to Java packages
- Writing out-of-the-box unit tests and executing them with Leiningen and Cursive Clojure
- Delving into the Clojure Interactive development workflow and a bit of the Clojure mindset
- Writing very simple functions and testing them

In the next chapter, we will learn about Java interop, so we can start using the familiar classes and libraries we already know in our Clojure code.

We will also learn how to use Clojure from Java, so you can start using it in your everyday Java projects.

# Chapter 3. Interacting with Java

We know a bit about how to organize our code and how that relates to packages in Java. Now, you surely need to use your old Java code and all the libraries you already know; Clojure encourages a new way to think about programming and it also allows you to use all the dependencies and code that you've already generated.

Clojure is a **Java Virtual Machine (JVM)** language and as such it is compatible with most Java dependencies and libraries out there; you should be able to use all the tools out there. You should also be able to use your Clojure programs with Java-only programs, this requires a bit of custom coding but in the end you can use Clojure in the right places of your project.

To be able to do this, we'll have to learn:

- Using Maven dependencies
- Using plain old Java classes from your Clojure code base
- A bit more about the Clojure language, in particular the `let` statements and destructuring
- Creating a Java interface for your Clojure code
- Using the Java interface from other Java projects

## Using Maven dependencies

Let's say that we want to write an image manipulation program; it is a very simple program that should be able to create thumbnails. Most of our codebase is in Clojure, so we want to write this in Clojure too.

There are a bunch of Java libraries meant to manipulate images, we decide to use `imgscalr`, which is very simple to use and it looks like it is available in Maven Central (<http://search.maven.org/>).

Let's create a new Leiningen project, as shown:

```
lein new thumbnails
```

Now, we need to edit the `project.clj` file in the `thumbnails` project:

```
(defproject thumbnails "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 "1.6.0"]])
```

You can add the `imgscalr` dependency similar to the following code:

```
(defproject thumbnails "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 "1.6.0"]
                [org.imgscalr/imgscalr-lib "4.2"]])
```

As you can see, you just need to add a dependency to the `:dependencies` vector, the dependencies are automatically resolved from:

- Maven Local
- Maven Central
- Clojars

## Note

The Maven Local points to your local maven repository that is in the `~/.m2` folder. If you wish, you can change it with Leiningen's `:local-repo` key.

You can add your own repositories, let's say you need to add **jcenter** (Bintray's Java repository) you can do so, as shown:

```
(defproject thumbnails "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 "1.6.0"]
              [org.imgscalr/imgscalr-lib "4.2"]]
:repositories [["jcenter" "http://jcenter.bintray.com/"]])
```

## Note

Leiningen supports a wide array of options to configure your project, for more information you can check the sample at Leiningen's official repository:

<https://github.com/technomancy/leiningen/blob/master/sample.project.clj>.

In order to download the dependencies, you have to execute the following code:

```
lein deps
```

## Tip

You don't need to execute `lein deps` every time you want to download dependencies, you can do it to force a download but Leiningen will automatically download them when it needs to.

You can check the current dependencies by running:

```
lein deps :tree
```

You will get something similar to this:

```
[clojure-complete "0.2.3" :scope "test" :exclusions
[[org.clojure/clojure]]]
[org.clojure/clojure "1.6.0"]
[org.clojure/tools.nrepl "0.2.6" :scope "test" :exclusions
[[org.clojure/clojure]]]
[org.imgscalr/imgscalr-lib "4.2"]]
```

This lists your current dependency tree.

# Clojure interop syntax

Clojure was designed to be a Hosted Language, which means that it can run in different environments or runtimes. One important philosophy aspect is that Clojure does not attempt to get in the way of your original host; this allows you to use your knowledge of the underlying platform to your advantage.

In this case, we are using the Java platform. Let's look at the basic interrupt syntax that we need to know.

## Creating an object

There are two ways to create an object in Clojure; for example, let's have a look at how to create an instance of `java.util.ArrayList`.

```
(def a (new java.util.ArrayList 20))
```

Here, we are using the `new` special form, as you can see it receives a symbol (the name of the class `java.util.ArrayList`) and in this case it is an integer.

The symbol `java.util.ArrayList` represents the classname and any Java class name will do here.

Next, you can actually pass any number of parameters (including 0 parameters). The next parameters are the parameters of the constructor.

Lets have a look at the other special syntax that is available to create objects:

```
(def a (ArrayList.))
```

The difference here is that we have a trailing dot; we prefer to see this syntax since it is shorter.

# Calling an instance method

Once we have created our object we can call instance methods. This is done similar to how we call Clojure functions, using the special dot form.

If we want to add an element to our newly created list, we will have to do it, as shown:

```
(. add a 5)
```

This syntax might look a little strange; here is how this syntax is formed:

```
(. instance method-name args*)
```

Similar to the two different options that we had when creating an object, we have another way to do this:

```
(.method-name instance args*)
```

You might think that this is more familiar, since the method name starting with a dot resembles how we write the Java method calls.

# Calling a static method or function

Being able to call methods and create objects gives us a great deal of power, with this simple construct we have gained a lot of power; we can now use most of the Java standard libraries and also the custom ones.

However, we still need a few more things; one of the most important ones is calling static methods. The static methods have a feel similar to Clojure functions, there is no `this` instance, you can simply call them as normal Clojure functions.

For instance, if we want an `emptyMap` from the `Collections` class, we can do it as shown:

```
(java.util.Collections/emptyMap)
```

You can think of static methods as functions and the class as a namespace. It is not exactly right but the mental model will help you understand it easily.

## Accessing inner classes

Another common doubt when using Java – Clojure interop is how to access inner classes.

Imagine you want to represent a single entry from a map with the `java.util.AbstractMap.SimpleEntry` class.

You might think that we have to do something similar to this:

```
(java.util.AbstractMap.SimpleEntry. "key" "value")
```

That's what you will normally do when writing Java, but in Clojure you might need to do something such as this:

```
(java.util.AbstractMap$SimpleEntry. "key" "value")
```

What we are seeing here is actually an exposed implementation detail; if you look at the classes in the JAR files or in your classpath, you will see the precise file name `AbstractMap$SimpleEntry`, as shown in the following screenshot:

```
# iamedu at Eduardos-MacBook-Pro.local in /Library/Java/JavaVirtualMachines/jdk1.8.0_66.jdk/Contents/Home [5:53:37]
$ jar -tf ./jre/lib/rt.jar | grep AbstractMap
java/beans/MetaData$java_util_AbstractMap_PersistenceDelegate.class
java/util/AbstractMap$1$1.class
java/util/AbstractMap$1.class
java/util/AbstractMap$2$1.class
java/util/AbstractMap$2.class
java/util/AbstractMap$SimpleEntry.class
java/util/AbstractMap$SimpleImmutableEntry.class
java/util/AbstractMap.class
```

This is what you need to keep in mind, always prefix the inner classes with

the parent (or more correctly containing) class (in this case `java.util.AbstractMap`) and the dollar sign.

# Writing a simple image namespace

Let's now write some Clojure code and create a file in `src/thumbnails/image.clj`.

Let's try to do this the Clojure way. First of all, write the namespace declaration and evaluate it:

```
(ns thumbnails.image
  (:require [clojure.java.io :as io])
  (:import [javax.imageio ImageIO]
           [java.awt.image BufferedImageOp]
           [org.imgscalr Scalr Scalr$Mode]))
```

Now open up a REPL and write the following code:

```
(def image-stream (io/input-stream
"http://imgs.xkcd.com/comics/angular_momentum.jpg"))
(def image
  (ImageIO/read image-stream))
image
(.getWidth image)
```

We now have an image instance and you can call all of the Java methods in the REPL. This is one of Clojure's core concepts, you can play with the REPL and check your code before really writing it and you can do it in an interactive way, as shown:

```
user=> (def image-stream (io/input-stream "http://imgs.xkcd.com/comics/angular_momentum.jpg"))
#'user/image-stream
user=> (def image (ImageIO/read image-stream))
#'user/image
user=> image
#object[java.awt.image.BufferedImage 0x69a568f0 "BufferedImage@69a568f0: type = 10 ColorModel: #pixelBits = 8 numComponents = 1 color space = java.awt.color.ICC_ColorSpace@6e4f49da transparency = 1 has alpha = false isAlphaPre = false ByteInterleavedRaster: width = 600 height = 386 #numDataElements 1 dataOff[0] = 0"]
user=> (.getHeight image)
386
user=> |
```

In the end, we want to stick with the following contents:

```

(ns thumbnails.image
  (:require [clojure.java.io :as io])
  (:import [javax.imageio ImageIO]
           [java.awt.image BufferedImageOp]
           [org.imgscalr Scalr Scalr$Mode]))


(defn load-image [image-stream]
  (ImageIO/read image-stream))

(defn save-image [image path]
  (ImageIO/write image "PNG" (io/output-stream path)))

(defn image-size [image]
  [(.getWidth image) (.getHeight image)])

(defn generate-thumbnail [image size]
  (Scalr/resize image Scalr$Mode/FIT_TO_WIDTH size (into-array
  BufferedImageOp [])))

(defn get-image-width [image-path]
  (let [image (load-image image-path)
        [w _] (image-size image)]
    w))

```

## Tip

You can see that in this code we use the inner class syntax, with `Scalr$Mode`. Mode is not actually a class but an `enum`, you can use the same syntax for all other inner types.

The code is pretty simple, it is very similar to what you've already seen; we'll go through the differences either way.

You can import the following classes:

- `javax.imageio.ImageIO`
- `java.awt.image(BufferedImageOp`
- `org.imgscalr.Scalr`
- `org.imgscalr.Scalr.Mode`

You have to be careful with the `Mode` class, since it is an inner class (it is inside another class) Clojure uses the special name `Scalr$Mode`.

## Tip

When importing inner classes, you have to be careful with the naming process, in Java you will use the name: `org.imgscalr.Scalr.Mode`; in Clojure you use the name: `org.imgscalr.Scalr$Mode`. The `load-image`, `save-image`, and `image-size` functions are self explanatory and the `generate-thumbnail` function is pretty simple as well; however, it has a special detail, it calls the following as the last argument:

```
(into-array BufferedImageOp [])
```

If you look at the `ImageScalr` javadoc, (<http://javadox.com/org.imgscalr/imgscalr-lib/4.2/org/imgscalr/Scalr.Mode.html>) you can see that the `resize` method has several overloaded implementations; most of them have a `varargs` argument as their last argument. In Clojure, you have to declare these `varargs` arguments as an array.

# Writing the tests

Now that you have written your image processing code, it is a good time to write the tests.

Let's just check if we can generate a thumbnail. Create a new `thumbnails.thumbnail-test` namespace, in the tests.

Remember, if you create the file, it must be named `test/thumbnails/thumbnaile_test.clj`.

Add the following contents to it:

```
(ns thumbnails.thumbnail-test
  (:require [clojure.test :refer :all]
            [clojure.java.io :as io]
            [thumbnails.image :refer :all]))


(deftest test-image-width
  (testing "We should be able to get the image with"
    (let [image-stream (io/input-stream
"http://imgs.xkcd.com/comics/angular_momentum.jpg")
          image (load-image image-stream)]
      (save-image image "xkcd-width.png")
      (is (= 600 (get-image-width (io/input-stream "xkcd-
width.png")))))))

(deftest test-load-image
  (testing "We should be able to generate thumbnails"
    (let [image-stream (io/input-stream
"http://imgs.xkcd.com/comics/angular_momentum.jpg")
          image (load-image image-stream)
          thumbnail-image (generate-thumbnail image 50)]
      (save-image thumbnail-image "xkcd.png")
      (is (= 50 (get-image-width (io/input-stream
"xkcd.png"))))))
```

Here we are using some unknown features, such as the `let` form and destructuring. We will see this in more detail in the next section.

# The let statement

Clojure gives us a `let` statement to name things; it allows us to do something very similar to variable declaration in other languages.

Keep in mind that we are not actually creating a variable in the same sense, as in Java, whenever we declare a variable. We state that we want to reserve a certain amount of memory to store something in the later stages; it can be a value for primitives or a memory location for objects. What we do here is simply name a value. This is a local scope that is useful to write cleaner and easier to understand code. Lets have a look at how it works:

```
(let [x 42] x)
```

This is the simplest `let` statement that we could write and it is exactly the same as just writing `42`. However, we can write something a little more complex, such as this:

```
(let [x 42
      y (* x x)]
  (println "x is " x " and y " y))
```

It looks self explanatory; to value `42` and `y`, we are assigning the value of multiplying `42` by `42`. In the end, we print `x` is `42` and `y` `1764`. It is important to note two things here:

- We can use a previously defined value in the `let` statement; for example, we use `x` when defining `y`.
- The `let` statement creates a scope, we can't use `x` or `y` outside of our `let` statement.

The `let` statement can even be nested, we could do something similar to the following example:

```
(let [x 42]
  (let [y (* x x)]
```

```
(println "x is " x " and y " y)))
```

It is a bit more complicated, since we are opening an unneeded set of parentheses and also writing more code; however, it allows us to see how lexical scope works.

Lets have a look at another interesting example:

```
(let [x 42]
  (let [y (* x x)]
    (let [x 41]
      (println "x is " x " and y " y))))
```

In here, we are masking the value of `x` with `41` and again these are not variables. We are not changing a memory region, we are merely creating a new scope with a new `X` value.

Going back to our test, the `let` statement begins with the following code:

```
image (load-image image-path)
```

It is pretty clear to understand, but the next line might prove a bit more difficult:

```
[w_] (image-size image)
```

It looks pretty strange; we are assigning the value of `(image-size image)` to `[w_]` but `[w_]` is not a symbol name!

What is happening here is that we are using a mechanism called destructuring to take the result of `(image-size image)` apart and just use the piece of information that we are interested in, which in this case is the width of the image.

Destructuring is one of the key features of Clojure, it can be used almost everywhere where symbol binding happens, such as:

- Let expressions

- Function parameter lists

Destructuring helps write more concise code but it might strike you as strange when you are not used to it. Let's talk about it in depth in the next section.

# Destructuring in Clojure

Destructuring is a feature in Clojure that is not common in other lisps; the idea is to allow you to write more concise code in scenarios where code doesn't really add value (for example, getting the first element from a list or the second parameter from a function) and concentrating only on what is important to you.

In order to understand this better, let's see an example of why destructuring can help you:

```
(let [v [1 2 3]] [(first v) (nth v 2)]) ;; [1 3]
```

What's wrong with the previous code? Nothing really, but you need to start thinking about what is `v`, what the first value of `v` is, what the `nth` function does, and at what index `v` starts.

Instead we can do this:

```
(let [[f s t] [1 2 3]] [f t]) ;; [1 3]
```

Once you are used to destructuring, you will see that you don't need to think about how to get the elements you need. In this case, we directly access the first, second, and third elements from our vector and use the first and third out of the three elements. With good naming it can become even easier.

Lets now take a deep dive into what destructuring is.

There are two types of destructuring:

- **Sequential destructuring:** It allows us to take sequential data structures apart and bind the values that you are interested in directly to symbols
- **Associative destructuring:** It allows us to take maps apart and bind

only the key reference values that you are interested in directly to symbols

## Sequential destructuring

Sequential destructuring should be easy to understand with some examples; lets have a look:

```
(let [[f s] [1 2]] f) ;; 1
(let [[f s t] [1 2 3]] [f t]) ;; [1 3]
(let [[f] [1 2]] f);; 1
(let [[f s t] [1 2]] t);; nil
(let [[f & t [1 2]] t);; (2)
(let [[f & t [1 2 3]] t);; (2 3)
(let [[f & t [1 2 3]] t);; (2 3)
(let [[f & [_ t]] [1 2 3]] [f t])
```

In these examples, as convention, we use `f` for first, `s` for second, `t` for third, and `a` for all the others.

The same destructuring idea and syntax can be used with function parameters, as shown in the next example:

```
(defn func [[f _ t]]
  (+ f t))
(func [1 2 3]) ;; 4
```

### Note

Here we use the symbol `_`, there is a convention in Clojure to use the `_` symbol whenever you are not interested in some value and you don't need to use it in the future. In the previous example, we aren't interested in the second parameter of the `func` function.

As you can see, it lets us write a much more concise code and focus only on what's important, which is the algorithm or business.

## Associative destructuring

We've already seen sequential destructuring that allows getting certain elements of a sequence by index. In Clojure, there is also associative destructuring, which allows you to take just the keys of the map in which you are interested.

Again, an example is worth more than a thousand words:

```
(let [{a-value a} {:a-value 5}] a-value) ;;= 5
(let [{a-value :a c-value :c} {:a 5 :b 6 :c 7}] c-value) ;;= 7
(let [{:keys [a c]} {:a 5 :b 6 :c 7}] c) ;;= 7
(let [{:syms [a c]} {'a 5 :b 6 'c 7}] c) ;;= 7
(let [{:strs [a c]} {:a 5 :b 6 :c 7 "a" 9}] [a c]) ;;= [9 nil]
(let [{:strs [a c] :or {c 42}} {:a 5 :b 6 :c 7 "a" 9}] [a c]) ;;= [9 42]
```

## Tip

Thinking of symbols as keys to a map can feel strange, nonetheless it is important to remember this feature; it could come in handy at some point.

As you can see, it's pretty simple too, but we have a few more options:

- We can reference some keys and assigning them a name, as shown in the first and second example
- We can reference keyword keys, as in the third example
- We can reference string keys, as in the fourth example
- We can define default values with the `:or` keyword!

Destructuring is one of the most used features of Clojure and it allows you to write very concise code.

Going back to our test code, it should now be pretty easy to understand the `get-image-width` function:

```
(defn get-image-width [image-path]
  (let [image (load-image image-path)
        [w _] (image-size image)]
    w))
```

As you can see, it sets the `image` value as the loaded image and then it calculates the width, gets the width only and returns that value.

We can now understand the `test-load-image` test:

```
(deftest test-load-image
  (testing "We should be able to generate thumbnails"
    (let [image-stream      (io/input-stream
"http://imgs.xkcd.com/comics/angular_momentum.jpg")
          image           (load-image image-stream)
          thumbnail-image (generate-thumbnail image 50)]
      (save-image thumbnail-image "xkcd.png")
      (is (= 50 (get-image-width (io/input-stream
"xkcd.png"))))))))
```

It just initializes an `image-stream` value, it then loads an image from that stream and generates a thumbnail. It finally loads the generated thumbnail and checks that the image width is 50px.

Now that we've written our tests and we are sure that everything works, we can use our little library from the Clojure projects, but what happens if we want to use it from a pure Java (or groovy, or scala) project?

# Exposing your code to Java

If you want to be able to use Clojure code from other JVM languages, in Clojure, there are a couple of ways in which you can do it:

- You can generate new Java classes and use them as you normally would; it can implement some interface or extend from some other class
- You can generate a proxy on the fly, this way you can implement a contract (in the form of a class or an interface) that some framework requires with little code and effort
- You can use the `clojure.java.api` package to call Clojure functions directly from Java

## Note

You can find more information on how this works at the following location: [http://www.falkoriemenschneider.de/a\\_2014-03-22\\_Add-Awesomeness-to-your-Legacy-Java.html](http://www.falkoriemenschneider.de/a_2014-03-22_Add-Awesomeness-to-your-Legacy-Java.html).

Let's have a look at how we can define a Java class.

Create a new namespace called `thumbnails.image-java` and write the following code:

```
(ns thumbnails.image-java
  (:require [thumbnails.image :as img])
  (:gen-class
    :methods [[loadImage [java.io.InputStream]
               java.awt.image.BufferedImage]
              [saveImage [java.awt.image.BufferedImage String]
               void]
              [generateThumbnail [java.awt.image.BufferedImage
               int] java.awt.image.BufferedImage]]
    :main false
    :name thumbnails.ImageProcessor))

(defn -loadImage [this image-stream]
```

```

(img/load-image image-stream)

(defn -saveImage [this image path]
  (img/save-image image path))

(defn -generateThumbnail [this image size]
  (img/generate-thumbnail image size))

```

This code is very similar to the Clojure code that we have already seen, except for the `gen-class` directive and the function names starting with a dash.

Let's review the `gen-class` in better detail:

```

(:gen-class
  :methods [[loadImage [java.io.InputStream]
  java.awt.image.BufferedImage]
            [saveImage [java.awt.image.BufferedImage String]
  void]
            [generateThumbnail [java.awt.image.BufferedImage
  int] java.awt.image.BufferedImage]]
  :main false
  :name thumbnails.ImageProcessor)

```

When the Clojure compiler sees this, it generates the byte code of a class but it needs a little help from the keywords to know how to generate the class.

- The name key defines the name of the class, it is a symbol
- The main key defines whether this class should have a main method or not
- The method key defines all the methods and their signatures, it is a vector with three parts: `[methodName [parameterTypes] returnType]`

The methods are then implemented as functions starting with the `(-)` character, the prefix can be changed with the prefix key.

You also need to tell Clojure to compile this class in advance, in Leiningen it can be achieved with `:aot`, go to your `project.clj` file and add an `:aot`

key with the namespace or namespaces to be compiled in a vector; if you want everything to be compiled in advance, you could use the special :all value.

In the end, you should have something similar to this:

```
1 (defproject thumbnails "0.1.0-SNAPSHOT"
2   :description "FIXME: write description"
3   :url "http://example.com/FIXME"
4   :license {:name "Eclipse Public License"
5             :url "http://www.eclipse.org/legal/epl-v10.html"}
6   :dependencies [[org.clojure/clojure "1.6.0"]
7                 [org.imgscalr/imgscalr-lib "4.2"]])
8   :aot [thumbnails.image-java]
9   :repositories [["jcenter" "http://jcenter.bintray.com/"]])
```

## Tip

If you want all of your code to be compiled in advance, you can use :aot :all in your project.clj.

Now, we can install our library to our Maven local repository. Go to the command line and run:

```
$ lein install
```

You'll get an output similar to the following screenshot:

```
# iamedu at Eduardos-MacBook-Pro.local in ~/Development/clj/source/chapter03/initial/thumbnails [6:25:18]
$ lein install
Created /Users/iamedu/Development/clj/source/chapter03/initial/thumbnails/target/thumbnails-0.1.0-SNAPSHOT.jar
Wrote /Users/iamedu/Development/clj/source/chapter03/initial/thumbnails/pom.xml
Installed jar and pom into local repo.
```

Now, you are good to go; you should have a thumbnails:thumbnails:0.1.0-SNAPSHOT dependency in your Maven local repository.

## Testing from Groovy

In order to see how this works with several JVM languages, we will use Groovy and Gradle to test. We can use Java and Maven just as easily. Remember that you can get the source from the code bundle so that you don't need to know everything that's happening here.

There are two files here; in the build.gradle file, we specify that we want to use our local Maven repository and we specify our dependency, as:

```
apply plugin: 'java'  
apply plugin: 'groovy'  
  
repositories {  
    jcenter()  
    mavenLocal()  
}  
  
dependencies {  
    compile "thumbnails:thumbnails:0.1.0-SNAPSHOT"  
    testCompile "org.spockframework:spock-core:0.7-groovy-2.0"  
}
```

Then we can write our test, as the following code:

```
package imaging.java  
  
import thumbnails.ImageProcessor  
import spock.lang.*  
  
class ImageSpec extends Specification {  
    def "Test we can use imaging tools"() {  
        setup:  
            def processor = new ImageProcessor()  
            def imageStream =
```

```
getClass().getResourceAsStream("/test.png")

when:
def image = processor.loadImage(imageStream)
def thumbnail = processor.generateThumbnail(image, 100)

then:
thumbnail.getWidth() == 100
}

}
```

You can then run the tests:

```
gradle test
```

As you can see, it is very easy to run your code from Java, Groovy, or even Scala. There are other ways to use Clojure with Java, particularly, if you want to implement an interface or generate a class dynamically.

# Proxy and reify

There are situations when you are interacting with Java libraries, where you must send an instance of a specific Java class to some method; writing a class isn't the best option, you should rather create an instance that conforms to a contract expected by some framework on the fly. We have two options to do this:

- **Proxy**: It allows you to implement a Java interface or extend from some super class. In reality, it creates a new object that calls your Clojure functions when needed
- **Reify**: Reify allows you to implement interfaces and Clojure protocols (we will see them later). It is not capable of extending classes. It is a better performant than the proxy and should be used whenever possible.

Let's look at a minimal example:

```
(import '(javax.swing JFrame JLabel JTextField JButton)
         '(java.awt.event ActionListener)
         '(java.awt GridLayout))

(defn sample []
  (let [frame (JFrame. "Simple Java Integration")
        sample-button (JButton. "Hello")]
    (.addActionListener
     sample-button
     (reify ActionListener
       (actionPerformed
        [_ evt]
        (println "Hello world")))))
  (doto frame
    (.add sample-button)
    (.setSize 100 40)
    (.setVisible true)))
(sample)
```

## Tip

`doto` is a macro that allows us to call several methods on an instance; you can think of it as a way to call all of the methods separately. It works great with Java Beans!

Open up an REPL and write the code; it should show a window with a button that prints `Hello world` (in the terminal) when clicked:

```
user=> (sample)
#<JFrame javax.swing.JFrame[frame
e=Simple Java Integration,resizab
ootPane(javax.swing.JRootPane[,0,
ut,alignmentX=0.0,alignmentY=0.0,
=,preferredSize=],rootPaneCheckingEnabled=true]>
user=> Hello world
Hello world
Hello world
Hello world
[]
```



If you are familiar with swing, then you know that the `addActionListener` of `JButton` needs a callback which is an instance of `ActionListener` and we are creating said instance with the `reify` function.

In Java code, you might normally do something similar to the following code:

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Hello world")
    }
})
```

We call this an anonymous class and it is essentially the same as a closure in functional languages. In the previous example, the code was replaced by a `reify`:

```
(reify ActionListener
```

```
(actionPerformed
  [_ evt]
  (println "Hello world"))
```

The `reify` statement receives the interface that you are implementing and all the methods that you are implementing as you list. In this case, we just implement `actionPerformed` to receive the action event.

This is the structure:

```
(reify InterfaceOrProtocol
  (method [self parameter-list]
    method-body)
  (method2 [self parameter-list]
    method-body))
```

This creates an instance of `ActionListener`, you can do the same with servlets, threads, collections, lists, or any other Java interface defined by anyone.

One particular thing that you need to remember here is that you need to always add `self` as the first parameter to your method implementations; it takes the place of the `this` keyword that works in Java.

# Summary

In this chapter, you have gained a lot of power from Clojure with a few new primitives.

As you can see, there are plenty of ways to interact with your current codebase; specifically, you can now:

- Use Java code from Clojure
- Use Clojure code from Java
- Reuse Java frameworks by creating objects that adhere to their contracts

With all of our new tools in mind, we are ready to tackle more concepts and a little bit more complexity with collections and data structures.

# Chapter 4. Collections and Functional Programming

We are now comfortable with using Java code from our Clojure programs, and we also know how to expose our Clojure programs with a Java API. However, we need to take a deeper look at Clojure and its true nature, which is functional programming.

In this chapter, we will cover the following topics:

- Basics of functional programming
- Persistent collections
- Sequential and associative collections
- The sequence abstraction
- Collection types
- Applying functional programming to collections

## Basics of functional programming

This is a topic that you can read about in lots of different places, and it seems that everyone has their own opinion of what functional programming is. There is however, some common ground that you will find in almost every definition, which relates to the benefits you gain from functional programming, such as:

- Easier code reuse
- Functions are easier to test
- Functions are easier to reason about

In order to get these benefits, you need to take into account the following things:

- You should think of functions as first class citizens
- Functions should minimize side effects (they shouldn't change any

- state)
- Functions should only depend on their parameters (this is called referential transparency)

Lets take a look at two examples of functions (or methods) in Java to illustrate how, even in Java, you can get benefits from writing functions without side effects and context dependency.

```
public void payRent(BigDecimal amount) {  
    User user = getCurrentUser();  
    if(user.payAmount != amount) {  
        System.out.println("Cannot pay");  
    } else {  
        user.money -= amount;  
    }  
}
```

Imagine you had to test the preceding function; you might have a number of problems:

1. You need to know how to get the current user; you might need to mock a database, or session storage. Or in the worst case scenario, you might need a real session storage service.
2. How can you know if something was paid for or not?

Now, look at this other example:

```
public boolean payRent(User user, BigDecimal amount,  
ValidateStrategy strategy) {  
    if(strategy.validatePayment(user, amount)) {  
        user.money -= amount;  
        return true;  
    } else {  
        return false;  
    }  
}
```

The preceding code is easier to test; you can create a user instance any way you want and with the ValidateStrategy class (or interface) you

could do whatever you need.

In the end, instead of a side-effect you get a return value stating if the action was possible or not. This way you don't need to mock and you can reuse it in different contexts.

Now that we have seen some common ground for functional programming, let's take a look at Clojure's value proposition for functional programming:

- Functions are first class citizens or values. The same as with integers or strings, you can create them in runtime, pass them around, and receive them in other functions.
- The same way that functions are values, the data structures are also values; they can't be modified in the sense that they can be modified in Java but they are a fixed value, just as an integer is a fixed value.
- Immutable data structures are very important, they allow for safe and simple multithreaded code.
- Laziness (of data structures) allows deferring evaluation until needed, to execute just what you must.

# Persistent collections

One of the most important features in Clojure is that collections are persistent. That does not mean that they are persistent to disk, it means that you can have several historical versions of a collection with the guarantee that updating or looking for something in any of those versions is going to have the same effort (complexity). You get all this with very little extra memory.

How? It is actually pretty simple. Clojure shares a common structure between several different data structures. If you add a single element to a data structure, Clojure shares the common part between the two structures and keeps track of the differences.

Let's see what we mean with an example:

```
(def sample-coll [:one :two :three])
(def second-sample-coll (conj sample-coll :four))
(def third-sample-coll (replace {:one 1} sample-coll))

sample-coll ;; [:one :two :three]
second-sample-coll ;; [:one :two :three :four]
third-sample-coll ;; [1 :two :three :four]
```

As you can see, when you `conj` a new item into a collection, or even when you replace some elements from it, you aren't changing the original collection, you are just generating a new version of it.

## Note

In Clojure, you can use `conj` (`conjoin`) as a verb. It means adding new elements into a collection in an efficient manner.

This new version doesn't modify the previous collections you had in any way.

This is a big difference from how common imperative languages work and at the first glance it might seem like a bad idea, but Clojure uses efficient algorithms that give us a couple of advantages, specifically:

- Different versions of the same collection share common parts, allowing us to use little memory
- When some part of the collection is not visible it gets garbage collected

What you get out of this is similar memory usage to what you would have with a mutable collection. Remember that there is a cost in space and time but it is negligible for most use cases.

Why would you want to have an immutable data collection? The main advantage is that it is simple to reason about them; passing them around to functions does not change them and when you are writing concurrent code, there is no chance that some other thread has modified your collection and you don't need to worry about explicitly handling locks.

# Types of collections in Clojure

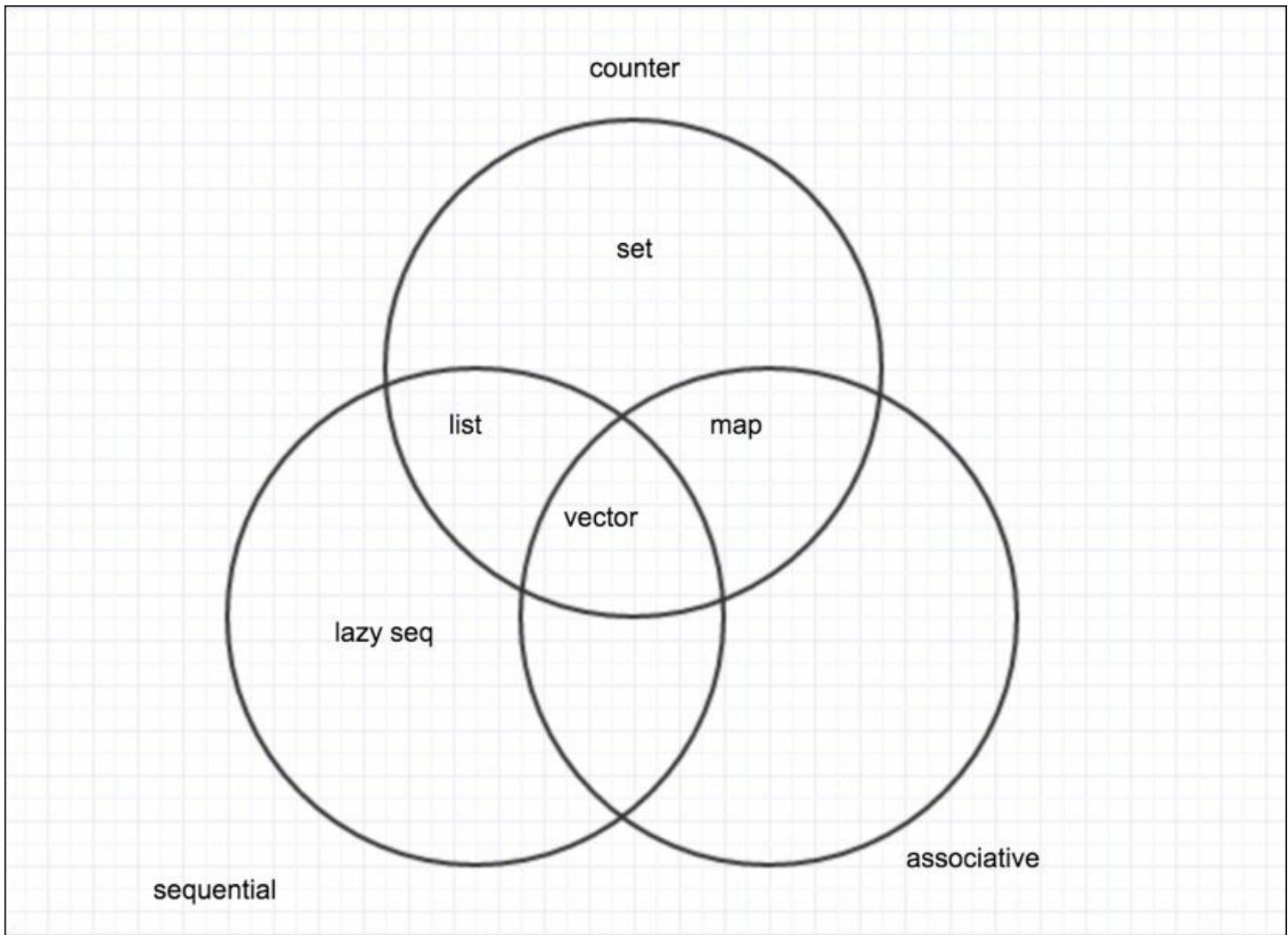
There are three types of collections in Clojure: counted, sequential, and associative. They are not mutually exclusive, meaning one collection might be any.

Let's look at each type:

- **Counted collection:** A counted collection is a collection which knows its size in constant time. It doesn't need to traverse its elements to get a count.
- **Sequential collection:** A sequential collection can be traversed sequentially; it's the most common approach that you would use for a list. The easiest way to think about this is similar to Java's list, which you can traverse with a for-loop or an iterator. In Clojure vectors, lists and lazy sequences are sequential collections.
- **Associative collections:** Associative collections can be accessed by keys; maps are the natural choice here. We said that one collection can be of any type; Clojure's vectors can also be used as associative collections, and each element index can be used as a key. You can think of it as a map where the keys are 0, 1, 2, 3, and so on.

Clojure has some functions that tell us if a given collection is of each type, sequential or associative. As you can guess, vectors return true for both. The following are those functions:

Function name	List	Vector	Map	Lazy sequence	Set
counted?	true	true	true	false	true
sequential?	true	true	false	true	false
associative?	false	true	true	false	false



In the previous table and diagram, you can see that we take **Set** into account and as you can see, it's neither sequential nor associative.

We should look at another property; whether a collection is counted or not. It means that a collection knows how many elements it has. Lists, vectors, maps, and sets are all counted; lazy sequences are not counted, since they are generated on the fly and they could even be infinite sequences.

We will learn more about all of these sequences in the later sections of this chapter.

# The sequence abstraction

Clojure has some unique features that make it different from other Lisps; one of them is the sequence abstraction. You can think of it as an interface that collections comply with. Clojure has a standard API of functions that you can use with sequences. Here are some examples of those functions:

- The `distinct` function: This function returns a sequence that includes each element of the original sequence just once:

```
(def c [1 1 2 2 3 3 4 4 1 1])
(distinct c) ;; (1 2 3 4)
```

- The `take` function: This function takes a number of elements from the original sequence:

```
(take 5 c) ;; (1 1 2 2 3)
```

- The `map` function: This function applies a function to each element of a sequence and creates a new sequence with these elements:

```
(map #(+ % 1) c) ;; (2 2 3 3 4 4 5 5 2 2)
```

The interesting part here is that these functions receive and return sequences and you can compose them together. It can be seen in the following code:

```
(->> c
  (distinct)
  (take 5)
  (reverse)) ;; (4 3 2 1)
```

`;;` This is known as a threading macro, it applies `distinct`, then `take 5` then `reverse` to the  
`;;` collection `c` so this is the same as writing:  
`;;` `(reverse (take 5 (distinct c)))` but much more readable

These are just some functions that accept and return sequences, but there are a lot more that you can use out of the box. The only assumption is that

your sequence argument can respond to three functions:

- `first`: This function returns the first of a sequence
- `rest`: This function returns another sequence, containing everything but the first element
- `cons`: This function receives two parameters, an item and another `seq` and returns a new `seq` containing the item followed by all the items in the second parameter

## Note

One of the functions that you'll find yourself using more is the `seq` function, it can convert any collection to a `seq`, even Java native arrays and objects that implement the `java.util.Iterable` interface. One of its main uses is to test a collection for emptiness.

# Specific collection types in Clojure

Now that you know about Clojure's general collection properties and the sequence abstraction, it is a good time to get to know about Clojure's specific collection implementations.

## Vectors

Vectors are Clojure's workhorse; together with map, it is the most used collection. Don't be afraid of them; they have nothing to do with Java's `java.util.Vector`. They are just a series of ordered values, such as a list or an array.

They have the following properties:

- They are immutable
- They can be accessed sequentially
- They are associative (they are maps of their indices, meaning that their keys are 0, 1, 2, and so on)
- They are counted, meaning they have a finite size
- They have random access, so you can access any element with almost constant time (with the `nth` function)
- The `conj` function appends a given element to them

## Tip

The `nth` function allows us to get the `nth` element of any `seq`, but you shouldn't use it without care. It has no problem handling vectors and it returns in constant time, but it takes linear time when used with a list, since it has to traverse all the collections looking for the element you asked. Try to use it just with vectors.

They have a literal syntax; you can define a vector with square brackets, as shown:

```
[42 4 2 3 4 4 5 5]
```

Besides the literal syntax, there's another function that you can use to build a vector. The `vec` function can build a vector out of any sequence passed to it:

```
(vec (range 4)) ;;= [0 1 2 3]
```

Another important benefit of vectors is that they are used for function arguments for declarations and for `let` bindings.

Take a look at the following example:

```
(def f [some-param & some-other-params] ...)  
(let [one 1 two (f p p-2 p-3) ] ...)
```

As you can see, the parameters in the function are defined as a vector, same as the `let` binding.

One of the main complaints about Lisps is that they use too many parentheses, Clojure's decision to use vectors instead in these structures is welcomed and makes the code much easier to read.

There are several ways to access a certain element of a vector:

- **Using the vector as a function:** Vectors can be used as functions of their keys; we haven't discussed maps yet but you will see that this is because they are associative:

```
(def v [42 24 13 2 11 "a"] )  
(v 0) ;;= 42  
(v 99) ;;= java.lang.IndexOutOfBoundsException
```

- **The nth function:** The `nth` function can receive an extra parameter for signaling when an index is not found and can be used, as shown:

```
(nth v 0) ;;= 42  
(nth v 99 :not-found) ;;= :not-found
```

```
(nth v 99) ;; java.lang.IndexOutOfBoundsException
```

- **The get function:** The `get` function can receive an extra parameter for signaling when an index is not found, it is used as shown. An important thing to keep in mind is that unlike `nth`, `get` cannot be used in sequences:

```
(get v 0) ;; 42
(get v 99 :not-found) ;; :not-found
(get v 99) ;; nil
```

You should use vectors almost always; in particular, if you want to do any of the following there is no other way to go:

- You need random access to a collection (either modifying or accessing it)
- You need to add elements at the tail of the collection

## Lists

Lists are the most important collection type in other Lisps. In Clojure, they are used to represent code, but their functionality is almost limited to that.

Lists in Clojure are single linked lists; as you can imagine, this means that they are not good for random access (you need to iterate the list until you get to the wanted index). That said, you can still use lists as sequences with every function of the API.

Let's list their properties:

- They are immutable
- They can be accessed sequentially
- They are not associative
- They are counted, meaning they have a finite size
- They shouldn't be accessed in random order. If you want the 99th element, then Clojure will have to visit all the first 98 elements to get the 99th.
- The `conj` function prepends a given element to it

You can use destructuring with lists, as seen in the previous chapter. You shouldn't be afraid to use the first function (or even nth with a small index).

## Tip

Lists have their use cases and as you learn more you'll probably be comfortable using them in some places (such as macros), but as a rule of thumb, try to use vectors instead.

# Maps

Maps are probably the most important collection type across all languages. They are also very important in Clojure.

Maps are collections of key value pairs, which mean that you can access or store an element by a key. We have been calling this type of collection an associative collection. Keys can be of any type of value in Clojure, even functions, lists, sets, vectors, or other maps.

## Sorted maps and hash maps

There are two types of maps in Clojure, each one of them with its own advantages.

- **Hash maps:** They are the most used form of map in Clojure; the literal syntax of maps creates this type of maps. They have a nearly constant lookup time, which makes them extremely fast and usable in most scenarios. Their down-side is that you can't access them in an ordered fashion. You can create them, as shown:

```
{:one 1 :two 2}  
(hash-map :one 1 :two 2)
```

- **Sorted maps:** If you need to be able to access a map's key-value pairs in a certain order, then you have to use a sorted map. The downside of sorted maps is that the lookup time is *not* constant, which means that

they are a little slower to access by key. However, when you need to traverse a map in the order of the keys, then this is the only way to go. A strong constraint here is that the keys must be comparable between them. Sorted maps can be created, as shown:

```
(sorted-map :sample 5 :map 6) ;; {:sample 5, :map 6}
(sorted-map-by > 1 :one 5 :five 3 :three) ;; {5 :five, 3
:three, 1 :one}
```

## Tip

Comparable objects are the ones that implement the `compareTo` interface.

## Common properties

Associative objects, including maps have the following properties:

- They are functions of their keys:

```
(def m #{:one 1 :two 2 :three 3})
(m :one) ;; 1
(m 1) ;; nil
```

- They can be used with associative destructuring:

```
(let [{:keys [a b c d]} #{:a 5}]
  [a b])
; [:a nil]
```

- They can be accessed with the `get` function:

```
(get m :one) ;; 1
(get m 1) ;; nil
(get m 1 :not-found) ;; :not-found
```

You can convert a map to a `seq` with the `seq` function; you will get a sequence where each element is a vector representing a key-value pair in the map:

```
(seq {:one 1 42 :forty-two :6 6}) ;; ([{:one 1} [:6 6] [42 :forty-
two]])
(doseq [[k v] (seq {:one 1 42 :forty-two :6 6})]
```

```
(println k v)
;; :one 1
;; :6 6
;; 42 :forty-two
```

`Doseq` is similar to Java's for-each loop. It executes the body for each element in a sequence.

It works as shown: `(doseq [x sequence] ;;)`. This works the same way as the let statement, you can use destructuring if needed:

```
(body-that-uses x))
```

## Sets

Clojure sets are a collection of unique elements. You can think of them as mathematical sets and as such, Clojure has operations, such as union, intersection and difference.

Let's look at the properties of sets:

- They are immutable
- They are associative (their keys are their elements)
- They are counted, meaning they have a finite size
- Their elements are unique (contained at most once)

## Sorted sets and hash sets

There are two kinds of sets: hash-sets and sorted-sets.

- **Hash-set:** Besides the properties that we already saw, hash-sets are unordered. They are implemented using a hash map as a backing implementation.
- **Sorted-set:** Besides the properties that we already saw, sorted-sets are sorted. They can be used as a parameter to all the functions that expect a sorted seq. They can be accessed sequentially in sorted order:

```
(doseq [x (-> (sorted-set :b :c :d)
                  (map name))])
```

```
(println x))  
;; b  
;; c  
;; d
```

You can also reverse them without problems, filter them, or map them similarly to a vector or list.

## Common properties

Sets are associative, which gives them some properties of maps:

- They are functions of their elements:

```
(#{:a :b :c :d} :a);; :a  
(#{:a :b :c :d} :e);; nil
```

- They can be used with map destructuring:

```
(let [{:keys [b]} #{:b}] b);; :b  
(let [{:keys [c]} #{:b}] b);; nil  
(let [{:keys [c]} (sorted-set :b)] c);; nil  
(let [{:keys [b]} (sorted-set :b)] b);; :b
```

- The `get` function can be used to access their elements:

```
(get #{:a :b :c :d} :e :not-found);; :not-found  
(get #{:a :b :c :d} :a);; :a  
(get #{:a :b :c :d} :e);; nil
```

## Union, difference, and intersection

If you remember mathematical sets, you'll know that the three main operations you can execute on them are the following:

- **Union** (`union a b`): The union includes all of the elements both in `a` and `b`
- **Difference** (`difference a b`): The difference is all the elements that are in `a` except for the elements that are also in `b`
- **Intersection** (`intersection a b`): It includes only the elements that are both in `a` and `b`

Here are some examples:

```
(def a #{:a :b :c :d :e})  
(def b #{:a :d :h :i :j :k})  
  
(require '[clojure.set :as s])  
  
(s/union a b) ;; #{:e :k :c :j :h :b :d :i :a}  
(s/difference a b) ;; #{:e :c :b}  
(s/intersection a b) ;; #{:d :a}
```

## Applying functional programming to collections

Now that we have a better understanding of how collections work, we have a better foundation to understand functional programming and how to make the most out of it.

This requires a different way of thinking about how to solve problems and you should keep your mind open.

Something that you might have found really strange about all of the collections is this feature: *They are immutable*.

This is indeed something quite strange; if you are used to Java, how can you possibly write programs without adding or removing elements from a list or set?

How is that even possible? In Java, we are used to writing `for` and `while` loops. We are used to mutating variables every step of the way.

How can we cope with immutable data structures? Let's find out in the subsequent sections.

## The imperative programming model

The software industry has been using a single software paradigm for a long time; this paradigm is an imperative programming model.

In the imperative paradigm, you have to tell the computer what to do at every single step. You are responsible for how the memory works, for whether it is running in a single core or multi core and, if you want to use multi core, you need to make sure that you change the program state correctly and avoid concurrency problems.

Let's see how you would calculate the factorial in an imperative style:

```
int factorial(int n) {  
    int result = 1;  
    for (int i = 1; i <= n; i++) {  
        result *= i;  
    }  
    return result;  
}
```

You are creating a variable `result` and a variable `i`. You change the variable `i` each time by assigning it the value `i + 1`. You can change the result by multiplying by `i`. The computer just executes your orders, comparing, adding, and multiplying. This is what we call the imperative programming model, because you need to tell the computer the exact commands it needs to execute.

This has worked fine in the past for various reasons, such as:

- The tight constraints of memory size forced programmers to make use of the memory as efficient as possible
- It was easier to think about a single thread of execution and how the computer executes it step-by-step

Of course, there were some drawbacks. A code can get complicated easily and the world has changed; the constraints that existed many years ago are gone. In addition, most of today's computers have more than one CPU. Multi-threading with shared mutable states is burdensome.

This makes thinking about this complicated. We get in trouble even in single threaded programs; just think, what would be the outcome of the

following code?

```
List l = new ArrayList();
doSomething(l);
System.out.println(l.size());
```

Is it 0? You can't possibly know because the `doSomething` method gets the list by reference and it can add or remove things without you knowing.

Now, imagine that you have a multithreaded program and a single `List` that can be modified by any of the threads. In the Java world, you have to know about `java.util.concurrent.CopyOnArrayList` and you need to know about its implementation details to know when it's a good idea to use it and when not to use it.

Even with these structures, it is difficult to think about multithreaded code. You still need to think about semaphores, locks, synchronizers, and so on.

The imperative world might be easy for the easy case, but it is not simple. The whole industry has realized this and there are many new languages and technologies that take ideas from other places. Java 8 has the streaming API and it includes lambda blocks, which are essentially functions. All these ideas are taken from the functional world.

## The functional paradigm

There are other ways of thinking about how to solve a problem; in particular, the functional paradigm has become important lately. It is nothing new; Lisp has supported this kind of programming since it was conceived in 1958. It has probably not been strong until recently, as it requires a more abstract way of thinking.

For you to get a better idea, let's see a couple of examples of how functional programming looks similar to the following code:

```
(map str [1 2 3 4 5 6]) ; ; ("1" "2" "3" "4" "5" "6")
```

```
(defn factorial [n]
  (reduce * (range 1 (inc n))))  
  
(factorial 5) ;;= 120
```

As you can see, it looks quite different; in the first case, we are passing the `str` function to another function called `map`.

In the second case, we are passing the `*` function to another function called `reduce`. In both cases, we are using functions as you would pass a list or a number, they are **first level citizens**.

One important difference in functional programming is that you don't need to tell the machine how to do things. In the first case, the `map` traverses the vector and applies the `str` function to each element, converting it to a `seq` of strings. You don't have to increment the index, you just need to tell the `map` what function you want to be applied to each element.

In the factorial case, there is a `reduce` function that receives the `*` and a `seq` from 1 to `n`.

It just works, you don't need to tell it how to do anything, just what you want done.

Both `map` and `reduce` are **higher order functions** because they accept functions such as parameters; they are also higher level abstractions.

## Note

Higher order functions are functions that either accept a function as an argument, return a function as result, or both.

You need to think on another level of abstraction and you don't care how things are really done, just that it gets the work done.

This comes with some benefit, if the implementation of a `map` someday

changes to become multithreaded, you would just need to update the versions and you would be ready to go!

## Functional programming and immutability

You may have also noticed that functional programming makes working with immutable structures necessary, because you can't mutate some or the other state in every step; you can just describe how you want to create a new collection based on some other collection and then get it. Clojure's efficient collections make it possible to share pieces of collections to keep memory usage at a minimum.

There are some other benefits to immutability:

- You can share your data structures with anyone you want because you are certain that nobody can change your copy.
- Debugging is simpler because you can test the program with some immutable value instead of some mutating state. When you get a value, you can find out which function returned the value that you got; there are not multiple places where a collection was mutated for you to check.
- Concurrent programming is simpler; again being certain that nobody can change your copy, even in other concurrently running threads, makes reasoning about your program simpler.

## Laziness

Clojure also supports lazy evaluation of transformations of sequences. Let's take a look at the `range` function:

```
(def r (range))
```

When running this function without parameters, you are creating an infinite sequence starting from 0.

It is an infinite sequence; so why does the Clojure REPL return automatically?

Clojure doesn't compute a collection value until needed, so in order to get a value you would have to do something, such as this:

```
(take 1 r);; (0)
(take 5 r);; (0 1 2 3 4)
```

## Tip

If you try to print an infinite sequence at the REPL, it will freeze.

Here, Clojure is resolving first one element and then five of the collection `r` because it needs to print them in the REPL.

## Tip

Lazy evaluation just works for collections and for sequence processing. Other operations (such as additions, method calls, and so on), are executed eagerly.

The interesting part is that you can define a new lazy collection by applying functions like filter and map to a certain collection.

For instance, let's get a new collection that contains all odd numbers:

```
(def odd-numbers (filter odd? r))
(take 1 odd-numbers) ;;; (1)
(take 2 odd-numbers) ;;; (1 3)
(take 3 odd-numbers) ;;; (1 3 5)
```

Now, `odd-numbers` is an infinite sequence of odd numbers and we have just asked for three of them. Whenever a number is already computed, it is not computed again. Let's change our collection a little bit in order to understand how this works:

```
(defn logging-odd? [number]
  (println number) ;;; This is terrible, it is a side effect and
  a source for problems
  ;;; Clojure encourages you to avoid side
  effects, but it is pragmatic
```

; ; and relies on you knowing what you are  
doing

(odd? number))

(def odd-numbers (filter logging-odd? r))

(take 1 odd-numbers)

; ; 0

; ; 1

; ; 2

; ; 3

; ; 4

; ; 5

; ; 6

; ; 7

; ; 8

; ; 9

; ; 10

; ; 11

; ; 12

; ; 13

; ; 14

; ; 15

; ; 16

; ; 17

; ; 18

; ; 19

; ; 20

; ; 21

; ; 22

; ; 23

; ; 24

; ; 25

; ; 26

; ; 27

; ; 28

; ; 29

; ; 30

; ; 31

; ; => (1)

(take 1 odd-numbers)

; ; => (1)

```
(take 2 odd-numbers)
;; => (1 3)

(take 3 odd-numbers)
;; => (1 3 5)

(take 4 odd-numbers)
;; => (1 3 5 7)

(take 10 odd-numbers)
;; => (1 3 5 7 9 11 13 15 17 19)
```

As you can see, some numbers get calculated first; you shouldn't expect or rely on a particular number of elements to be precomputed at a certain time.

Also, keep in mind that the computation isn't executed again when we ask for the same number of elements, since it has been already cached.

# Summary

Collections and functional programming in Clojure are extremely powerful tools that allow us to use a completely different paradigm of programming.

Here's what we have learned so far:

- The mechanics of immutable collections and what each collection type in Clojure is best for
- How sequence abstraction and how a lot of Clojure functions are available to work on collections, using this abstraction
- How functional programming enables us to write simpler programs that work better in parallel environments and help us save resources using laziness

In the subsequent chapters, we will learn about other new Clojure features that give us a new and much more powerful way to implement polymorphism than what Java offers.

# Chapter 5. Multimethods and Protocols

We now have a better understanding of how Clojure works; we understand how to perform simple operations with immutable data structures but we are missing some features that could make our lives much easier.

If you have been a Java programmer for a while, you are probably thinking about polymorphism and its particular flavor in Java.

Polymorphism is one of the concepts that enable us to reuse a code. It gives us the ability to interact with different objects with the same API.

Clojure has a powerful polymorphism paradigm that allows us to write simple code, create code that interacts with types that don't exist yet, and extend code in ways it wasn't devised for when a programmer wrote it.

To help us with polymorphism in Clojure, we have two important concepts that we will cover in this chapter:

- Multimethods
- Protocols

Each of them has its own use cases and things it is best at; we will look into them in the next section.

We will learn each of these different concepts by reviewing what we already know from Java and then we will learn similar concepts from Clojure that give us much more power.

## Polymorphism in Java

Java uses polymorphism heavily, its collection API is based on it. Probably

the best examples of polymorphism in Java are the following classes:

- `java.util.List`
- `java.util.Map`
- `java.util.Set`

We know that depending on our use case we should use a particular implementation of these data structures.

If we prefer to use an ordered Set, we might use a `TreeSet`.

If we need a Map in a concurrent environment, we will use a `java.util.concurrent.ConcurrentHashMap`.

The beautiful thing is that you can write your code using the `java.util.Map` and `java.util.Set` interfaces and if you need to change to another type of Set or Map, because the conditions have changed or someone has created a better version of the collection for you, you don't need to change any code!

Lets look at a very simple example of polymorphism in Java.

Imagine that you have a Shapes hierarchy; it might look similar to the following code:

```
package shapes;

public interface Shape {
    public double getArea();
}

public class Square implements Shape {
    private double side;
    public Square(double side) {
        this.side = side;
    }

    public double getArea() {
        return side * side;
    }
}
```

```

    }

}

public class Circle implements Shape {
    private double radius;
    public Circle(double radius) {
this.radius = radius;
    }

    public double getArea() {
        return Math.PI * radius * radius;
    }
}

```

You surely are aware of the power of this concept, you can now calculate the summation of all the areas of a collection of figures, such as this:

```

package shapes;

import java.util.List;
import java.util.Arrays;

public class Main {

    public static double totalArea(List<Shape> shapes) {

        double result = 0;
        for(Shape s : shapes) {
            result += s.getArea();
        }

        return result;
    }

    public static void main(String args[]) {
        System.out.println(totalArea(Arrays.asList(new Circle(5), new Square(3), new Circle(4.5))));
    }
}

```

The `totalArea` method doesn't care about the specific types of shapes that you pass to it and you can add new types of shapes, such as rectangles or trapezoids. Your same code will now work with new data types.

Now, with the same Java code base, imagine that you wanted to add a new function to your shape interface, something simple, such as a `getPerimeter` method.

This seems quite simple; you will have to modify each class that implements the Shape interface. I'm sure you've faced this problem a lot of times when you don't have access to the base source. The solution is to write a wrapper around your Shape objects but this introduces more classes and incidental complexity.

Clojure has its own idea of polymorphism, it is much simpler but also very powerful; you can in fact solve the perimeter problem with it in a very simple way.

One way to solve this is with multimethods; lets have a look at how they work.

# Multimethods in Clojure

Multimethods are similar to interfaces, they allow you to write a common contract and then a family of functions can fulfill that interface with a specific implementation.

They are extremely flexible, as you will see they grant you a very fine control over what function is going to get invoked for a specific data object.

Multimethods consist of three parts:

- A function (or method) declaration
- A dispatch function
- Each possible implementation of the function

One of the most interesting features of multimethods is that you can implement new functions for already existing types without having to write wrappers around your currently existing object.

The multimethod declaration works the same way as the interface; you define a common contract for the polymorphic function, as shown:

```
(defmulti name docstring? attr-map? dispatch-fn& options)
```

The `defmulti` macro defines the contract for your multimethod, it consists of:

- The multimethod's name
- An optional `docstring` (this is the documentation string)
- The attribute map
- The `dispatch-fn` function

## Note

The `dispatch` function gets called for every piece of content; it generates a

dispatch key that is later checked against its function implementation. When the dispatch key and the key in the function implementation match, the function is called.

The `dispatch` function receives the same parameters that the function you are calling receives and returns a dispatch key that is used to determine the function that should dispatch the request.

Each implementation function must define a dispatch key, if it matches with the `dispatch` function's result, then this function is executed.

An example should clarify:

```
(defmulti area :shape)

(defmethod area :square [{:keys [side]}] (* side side))

(area {:shape :square :side 5})
;;=> 25
```

Here, we are defining a multimethod called `area`; the `defmulti` statement has the following structure:

```
(defmultifunction-name dispatch-function)
```

In this case, the multimethod is called `area` and the `dispatch` function is the `:shape` keyword.

## Note

Remember, keywords can be used as functions that look up themselves in maps. So, for example, the result of `(:shape {:shape :square})` is `:square`.

Afterwards, we define a method, as shown:

```
(defmethodfunction-name dispatch-key [params] function-body)
```

Note that the `dispatch-key` is always the result of invoking the `dispatch-function` with `params` as parameters.

Finally, let's look at the invocation, `(area {:shape :square :side 5})` which is calling a multimethod. The first thing that happens is that we call the dispatch function `:shape`, as shown:

```
(:shape {:shape :square :side 5})  
;; :square
```

The `:square` function is now the dispatch key, we need to look for the method that has that dispatch key; in this case, the only method that we defined works. So, the function is executed and we get the result of 25.

It is pretty simple to add the area and perimeter for both square and circle, lets check the implementation:

```
(defmethod area :circle [{:keys [radius]}]  
  (* Math/PI radius radius))  
  
(defmulti perimeter :shape)  
  
(defmethod perimeter :square [{:keys [side]}] (* side 4))  
  
(defmethod perimeter :circle [{:keys [radius]}] (* 2 Math/PI  
  radius))
```

Now, we have defined how to calculate the perimeter and area of circles and squares with very little effort and without having to define a very strict object hierarchy. However, we are just starting to uncover the power of multimethods.

## Note

Keywords can be namespaced, it allows you to keep your code better organized. There are two ways to define a namespaced keyword, such as `:namespace/keyword` and `::keyword`. When using the `::` notation, the used namespace is the current namespace. So if you write `::test` in the REPL,

you will be defining :user/test.

Let's try another example, copy the following code into your REPL:

```
user=> (defmulti walk :type)
#'user/walk
user=>

user=> (defmethod walk ::animal [] "Just walk")
#object[clojure.lang.MultiFn 0x58c715bf "clojure.lang.MultiFn@58c715bf"]
user=> (defmethod walk ::primate [] "Primate walk")
#object[clojure.lang.MultiFn 0x58c715bf "clojure.lang.MultiFn@58c715bf"]
user=> (walk {:type ::animal})
"Just walk"
user=>

user=> (walk {:type ::primate})
"Primate walk"
user=>
```

As you can see, it just works as you might expect it to. However, let's see how you can create a keyword hierarchy to be a little bit more flexible than this.

## Keyword hierarchies

You can declare that a keyword derives from another keyword and then respond to other dispatch keys, for that you can use the `derive` function:

```
(derive ::hominid ::primate)
```

### Tip

When defining a keyword hierarchy, you have to use namespaced keywords.

Here, you are declaring that the `::hominid` key is derived from the `::animal` key and you can now use `::hominid` as `::animal`; let's see that now:

```
(walk {:type ::hominid})  
;; Primate Walk
```

We do have some problems when defining hierarchies, for instance what will happen if the same keyword were to be derived from two conflicting keywords? Let's give it a try:

```
(derive ::hominid ::animal)  
  
(walk {:type ::hominid})  
;; java.lang.IllegalArgumentException: Multiple methods in  
multimethod 'walk' match dispatch value: :boot.user/hominid ->  
:boot.user/animal and :boot.user/primate, and neither is  
preferred
```

We get an error that says, there are two methods that match the dispatch value. Since our hominid derives both from animal and primate, it doesn't know which to resolve.

We can work this out explicitly with:

```
(prefer-method walk ::hominid ::primate)  
(walk {:type ::hominid})  
; Primate walk
```

Now, everything works correctly. We know that we prefer to resolve to a primate when calling the walk multimethod with the hominid key.

You can also define a more specific method, just for the `hominid` key:

```
(defmethod walk ::hominid [_] "Walk in two legs")  
  
(walk {:type ::hominid})  
; Walk in two legs
```

The derivation hierarchy can get a little complex and we might need some functions to introspect relationships. Clojure has the following functions to work with type hierarchies.

- `isa?`
- `parents`
- `descendants`
- `underive`

## **isa?**

The `isa` function checks if a type derives from some other type, it works with Java classes as well as Clojure keywords.

It is simple to explain with examples:

```
(isa? java.util.ArrayList java.util.List)
;;=> true
```

```
(isa? ::hominid ::animal)
;;=> true
```

```
(isa? ::animal ::primate)
;;=> false
```

## **parents**

The `parent` function returns a set of a type's parents, they might be Java or Clojure keywords:

```
(parents java.util.ArrayList)
;;=> #
{java.io.Serializable java.util.List java.lang.Cloneable java.util.RandomAccess java.util.AbstractList}

(parents ::hominid)
#{:user/primate :user/animal}
```

## **descendants**

The `descendants` function, as you can imagine, returns a set of descendants of the passed keyword; it is important to keep in mind that in this case only Clojure keywords are allowed:

```
(descendants ::animal)
;;=> #{:boot.user/hominid}
```

## underive

The `underive` function breaks the relation between two types, as you can imagine it only works with the Clojure keywords:

```
(underive ::hominid ::animal)
;;=> (isa? ::hominid ::animal)
```

This function will normally be used at development time and they allow you to play around with your type hierarchy in a very simple and dynamic way.

## A la carte dispatch functions

Until now, we have used a keyword as a dispatch function but you can use any function you like with as many arguments as you want. Let's take a look at some examples:

```
(defn dispatch-func [arg1 arg2]
  [arg2 arg1])
```

This is a simple function, but it shows two important facts:

- The `dispatch` function can receive more than one argument
- The `dispatch` key can be anything, not just a keyword

Lets have a look at how we can use this `dispatch` function:

```
(defmulti sample-monomethod dispatch-func)
;; Here we are saying that we want to use dispatch-func to
calculate the dispatch-key
```

```
(defmethod sample-monomethod [:second :first] [first second]
  [:normal-params first second])
(defmethod sample-monomethod [:first :second] [first second]
  [:switch-params second first])

(sample-monomethod :first :second)
;;=> [:normal-params :first: second]

(sample-monomethod :second :first)
;; =>[:switch-params :first: second]
```

We are getting to know the `dispatch` function a little bit better; now that you know that you can implement any `dispatch` function, you have a very fine grained control over what function gets called and when.

Lets look at one more example, so we can finally grasp the complete idea:

```
user=> (defn avg [& coll]
  #_=> (/ (apply + coll) (count coll)))
#'user/avg
user=> (defn get-race [& ages]
  #_=> (if (> (apply avg ages) 120)
  #_=>      :timelord
  #_=>      :human))
#'user/get-race
user=> (defmulti travel get-race)
#'user/travel
user=>

user=> (defmethod travel :timelord [& ages]
  #_=> (str (count ages) " timelords travelling by tardis"))
#object[clojure.lang.MultiFn 0x6cc1d65c "clojure.lang.MultiFn@6cc1d65c"]
user=> (defmethod travel :human [& ages]
  #_=> (str (count ages) " humans travelling by car"))
#object[clojure.lang.MultiFn 0x6cc1d65c "clojure.lang.MultiFn@6cc1d65c"]
user=> (travel 2000 1000 100 200)
"4 timelords travelling by tardis"
user=> (travel 80 20 100 40)
"4 humans travelling by car"
user=> |
```

Now the true power of multimethods becomes apparent. You now have an adhoc way of defining polymorphic functions which has the possibility to define type hierarchies and even execute your own logic to determine the function that is going to be called finally.

# Protocols in Clojure

Multimethods are just one of the options for polymorphism you have in Clojure, there are other ways to implement polymorphic functions.

Protocols are a little easier to understand and they might feel more similar to Java interfaces.

Lets try to define our shape program using protocols:

```
(defprotocol Shape
  "This is a protocol for shapes"
  (perimeter [this] "Calculates the perimeter of this shape")
  (area [this] "Calculates the area of this shape"))
```

Here, we have defined a protocol and it is called shaped and everything that implements this protocol must implement the following two functions: `perimeter` and `area`.

There are a number of ways to implement a protocol; one interesting feature is that you can even extend Java classes to implement a protocol without an access to the Java source and without having to recompile anything.

Let's start by creating a record that implements the type.

# Records in Clojure

Records work exactly like maps, but they are much faster if you stick to the predefined keys. Defining a record is similar to defining a class, Clojure knows beforehand about the fields that the record will have, so it can generate byte code on the fly and the code that uses the records is much faster.

Lets define a `Square` record, as shown:

```
(defrecord Square [side]
  Shape
  (perimeter [{:keys [side]}] (* 4 side))
  (area [{:keys [side]}] (* side side)))
```

Here, we are defining the `Square` record and it has the following properties:

- It has only one field, `size`; this is going to work as a map with only the `side` key
- It implements the `Shape` protocol

Lets have a look at how a record is instantiated and how we can use it:

```
(Square. 5)
;;=> #user/Square{:size 5}

(def square (Square. 5))

(let [{side :side} square] side)
;;=> 5

(let [{:keys [side]} square] side)
;;=> 5

(doseq [[k v] (Square. 5)] (println k v))
;; :side 5
```

As you can see it works exactly like a map, you can even associate things to it:

```
(assoc (Square. 5) :hello :world)
```

The downside of doing this is that we no longer have the performance guarantees that we had when defining just the record fields; nonetheless, it is a great way of giving some structure to our code.

We still have to check how we can use our perimeter and area functions, it is pretty simple. Let's have a look:

```
(perimeter square)
;;=> 20
```

```
(area square)
;;=> 25
```

Just to continue with the example, let's define the `Circle` record:

```
(defrecord Circle [radius]
  Shape
  (perimeter [{:keys [radius]}] (* Math/PI 2 radius))
  (area [{:keys [radius]}] (* Math/PI radius radius)))

(def circle (Circle. 5))

(perimeter circle)
;;=> 31.41592653589793

(area circle)
;;=> 78.53981633974483
```

One of the promises was that we will be able to extend our existing records and types without having to touch the current code. Well, lets keep to that promise and check how to extend our records without having to touch existing code.

Imagine, we need to add a predicate telling us whether a shape has an area or not; we might then define the next protocol, as shown:

```
(defprotocol ShapeProperties
  (num-sides [this] "How many sides a shape has"))
```

Let's get directly to the `extend-type`, which is going to help us define this `num-sides` function for our old protocols. Note that with `extend-type` we can even define functions for existing Java types:

```
(extend-type Square
  ShapeProperties
  (num-sides [this] 4))
```

```
(extend-type Circle
ShapeProperties
  (num-sides [this] Double/POSITIVE_INFINITY))

(num-sides square)
;;=> 4

(num-sides circle)
;;=> Infinity
```

Protocols become much more interesting when you extend them for Java types. Lets create a protocol that includes some functions for list like structures:

```
(defprotocol ListOps
  (positive-values [list])
  (negative-values [list])
  (non-zero-values [list]))

(extend-type java.util.List
ListOps
  (positive-values [list]
    (filter #(> % 0) list))
  (negative-values [list]
    (filter #(< % 0) list))
  (non-zero-values [list]
    (filter #(not= % 0) list)))
```

And now you can use the positive values, negative values and non-zero-values with anything that extends from `java.util.List`, including Clojure's vectors:

```
(positive-values [-1 0 1])
;;=> (1)

(negative-values [-1 0 1])
;;=> (-1)

(no-zero-values [-1 0 1])
;;=> (-1 1)
```

It might not be very exciting to extend `java.util.List`, since you can define these three as functions and it works the same way but you can extend any custom Java type with this mechanism.

# Summary

Now we understand Clojure's way a little bit better and we have a better grasp of what to look for when we need polymorphism. We understand that when needing a polymorphic function we have several options:

- We can implement multimethods if we need a highly customized dispatching mechanism
- We can implement multimethods if we need to define a complex inheritance structure
- We can implement a protocol and define a custom type that implements that protocol
- We can define a protocol and extend existing Java or Clojure types with our custom functions for each type

Polymorphism in Clojure is very powerful. It allows you to extend the functionality of Clojure or Java types that already exist; it feels like adding methods to an interface. The best thing about it is that you don't need to redefine or recompile anything.

In the next chapter, we will talk about concurrency—one of the key features of Clojure. We will learn about the idea of what the identity and values are and how those key concepts make writing concurrent programs much easier.

# Chapter 6. Concurrency

Programming has changed, in the past we could just rely on computers getting faster year after year. This is proving to be more and more difficult; so, hardware manufacturers are taking a different approach. Now, they are embedding more processors into computers. Nowadays, it's not uncommon to see phones with just or four cores.

This calls for a different way of writing software, one in which we are able to execute some tasks in other processes, explicitly. The modern languages are trying to make this task feasible and easier for modern developers, and Clojure is no exception.

In this chapter, we will see how Clojure enables you to write simple concurrent programs by reviewing Clojure's core concepts and primitives; in particular, we need to understand the concept of identity and value that Clojure has embedded into the language. In this chapter, we will cover the following topics:

- Using your Java knowledge
- The Clojure model of state and identity
- Promises
- Futures
- Software transactional memory and refs
- Atoms
- Agents
- Validators
- Watchers

## Using your Java knowledge

Knowing Java and your way around Java's threading APIs gives you a great advantage, since Clojure relies on the tools that you already know.

Here, you'll see how to use threads and you can extend everything you see here to execute other services.

Before going any further, let's create a new project that we'll use as a sandbox for all of our tests.

Create it, as shown in the following screenshot:

```
# iamedu at Eduardos-MacBook-Pro.local in ~ [5:23:35]
$ lein new clojure-concurrency
Generating a project called clojure-concurrency based on the 'default' template.
The default template is intended for library projects, not applications.
To see other templates (app, plugin, etc), try 'lein help new'.
```

Modify the `clojure-concurrency.core` namespace, so that it looks similar to the following code snippet:

```
(ns clojure-concurrency.core)

(defn start-thread [func]
  (.start (Thread. func)))
```

It's easy to understand what's happening here. We are creating a thread with our function and then starting it; so that we can use it in the REPL, as follows:

```
--clojure-concurrency.core=> (require 'clojure-concurrency.core :reload-all)
WARNING: await already refers to: #'clojure.core/await in namespace: clojure-concurrency.core, being replaced by: #'co.paralleluniverse.pulsar.core/await
WARNING: promise already refers to: #'clojure.core/promise in namespace: clojure-concurrency.core, being replaced by: #'co.paralleluniverse.pulsar.core/promise
WARNING: test already refers to: #'clojure.core/test in namespace: clojure-concurrency.core, being replaced by: #'clojure-concurrency.core/test
nil
clojure-concurrency.core=> (in-ns 'clojure-concurrency.core)
#object[clojure.lang.Namespace 0x736fc8a0 "clojure-concurrency.core"]
clojure-concurrency.core=> (start-thread #(println "Hello threaded world"))
Hello threaded world
nil
```

## Tip

`java.lang.Thread` has a constructor, which receives an object implementing the runnable interface. You can just pass a Clojure function because every function in Clojure implements runnable and callable interfaces. This means that you can also use the executors transparently in Clojure!

We'll see a `nil` and `Hello` threaded world values in any random order. The `nil` value is what the start thread returns.

The `Hello` threaded world is a message from another thread. With this, we now have the basic tools to get to know and understand how threads work in Clojure.

# The Clojure model of state and identity

Clojure has very strong opinions about concurrency, in order to take it in a simpler way it redefines what state and identity mean. Let's explore the meaning of this concepts in Clojure.

When talking about state in Java, you probably think about the values of the attributes of your Java classes. The state in Clojure is similar to Java, it refers to the values of objects but there are very important differences that allow simpler concurrency.

In Clojure, identity is an entity that might have different values over time. Consider the following examples:

- I have an identity; I will be and continue being this particular individual, my opinions, ideas, and appearance might change over time but I am the same individual with the same identity.
- You have a bank account; it has a particular number and is run by a particular bank. The amount of money you have in it changes over time, but it is the same bank account.
- Consider a ticker symbol (such as GOOG), it identifies a stock in the stock market; the value associated with it changes over time, but not its identity.

**State** is a value that an identity took at some point in time. One of its important features is that it is immutable. State is a snapshot of an identity's value at some given time.

So, in the previous examples:

- Who you are right now, how you feel, how you look, and what you think is your current state
- The money you currently have in your bank account is its current state

- The value of the GOOG stock is its current state

All of these states are immutable; it doesn't matter who you are tomorrow or how much you win or spend. It is true and it will always be true that in this particular moment in time you have a certain state.

## Tip

Rich Hickey, Clojure's author, is a great talker and has several talks in which he explains the ideas and philosophy behind Clojure. In one of them, (Are We There Yet?) he explains this idea of state, identity, and time very extensively.

Let's now explain the two key concepts in Clojure:

- **Identity:** Throughout your whole life, you have a single identity; you never stop being you, even if you keep changing throughout your whole life.
- **State:** At any given moment in your life, you are a certain person with likes, dislikes, and some beliefs. We call this way of being at a moment of your life, the state. If you look at a particular moment in your life, you will see a fixed value. Nothing will change the fact that you were the way you were in that moment in time. That particular state is immutable; over time, you have different states or values, but each of those states is immutable.

We use this fact to write simpler concurrent programs. Whenever you want to interact with an identity, you look at it and you get its current value (a snapshot at the time), and then you operate with what you have.

In imperative programming, you normally have a guarantee that you have the latest value but it is very difficult to keep it in a consistent state. The reason for this is that you are relying on a shared mutable state.

A shared mutable state is the reason why you need to use a synchronized code, locks, and mutexes. It is also the reason for very complex programs,

with difficult bugs to track.

Nowadays, Java is learning the lessons from other programming languages and now it has primitives that allow simpler concurrent programming. These ideas are taken from other languages and newer ideas, so there is a good chance that someday you'll see similar concepts to what you'll study here in other mainstream programming languages.

There is no guarantee that you'll always have the latest value, but don't worry, you just have to think about things differently and use the concurrency primitives that Clojure gives you.

This is similar to how you work in real life, you don't know exactly what's happening with all of your friends or co-workers when you do something for them; you talk to them, get the current facts, and then go and get working. While you are at it, something needs to change; in this case we need a coordination mechanism.

Clojure has various such coordination mechanisms, let's have a look at them.

# Promises

If you are a full stack Java developer, there is a good chance that you have met promises in JavaScript.

Promises are simple abstractions that don't impose strict requirements on you; you can use them to calculate the result on some other thread, light process, or anything you like.

In Java, there are a couple of ways to achieve this; one of them is with futures (`java.util.concurrent.Future`) and if you want something more similar to JavaScript's promise there is a nice implementation called **jdeferred** (<https://github.com/jdeferred/jdeferred>), which you might have used before.

In essence a promise is just a promise that you can give to your caller, the caller can use it at any given time. There are two possibilities:

- If the promise has already been fulfilled, the call returns immediately
- If not, the caller will block until the promise is fulfilled

Let's take a look at an example; remember to use the start-thread function in the `clojure-concurrency.core` package:

```
clojure-concurrency.core=> (in-ns 'clojure-concurrency.core)
#object[clojure.lang.Namespace 0x736fc8a0 "clojure-concurrency.core"]
clojure-concurrency.core=> (def p (promise))
#'clojure-concurrency.core/p
clojure-concurrency.core=> (start-thread
    #_=>  #(do
        #_=>      (deref p)
        #_=>      (println "Hello world")))
nil
```

## Tip

Promises are only calculated once and then cached. So don't worry about using them as many times as you like once it has been calculated, there is

no runtime cost associated!

Let's stop here and analyze the code, we are creating a promise called `p` and then we start a thread that does two things.

It tries to get a value from `p` (the `deref` function tries to read the value from a promise) and then prints `Hello world`.

We won't see the `Hello world` message just yet; we will instead see a `nil` value. Why is that?

The start thread returns the `nil` value and what happens now is what we described in the first place; `p` is the promise and our new thread will block it until it gets a value.

In order to see the `Hello world` message, we need to deliver the promise. Let's do that now:

```
clojure-concurrency.core=> (deliver p 5)
Hello world
#object[co.paralleluniverse.pulsar.core$promise$reify__2871 0x6459f6f {:status :ready, :val 5}]
```

As you can see, we now get the `Hello world` message!

As I said, there is no need to use another thread. Let's now see another example in the REPL:

```
clojure-concurrency.core=> (def p (promise))
#'clojure-concurrency.core/p
clojure-concurrency.core=> (deliver p 5)
#object[co.paralleluniverse.pulsar.core$promise$reify__2871 0x68ead4cd {:status :ready, :val 5}]
clojure-concurrency.core=> @p
5
clojure-concurrency.core=> (println "Hello world")
Hello world
nil
```

## Tip

You can use `@p` instead of `(deref p)`, it works for every identity in this chapter too.

In this example we don't create separate threads; we create the promise, deliver it, and then use it in the same thread.

As you can see, promises are an extremely simple type of synchronization mechanism, you can decide whether to use threads, executor services (which are just thread pools), or some other mechanism, such as lightweight threads.

Let's have a look at Pulsar library for creating lightweight threads.

## Pulsar and lightweight threads

Creating a thread is an expensive operation and it consumes RAM memory. To know how much memory creating a thread consumes in Mac OS X or Linux, run the next command:

```
java -XX:+PrintFlagsFinal -version | grep ThreadStackSize
```

What you see here will depend on the OS and JVM version that you are using, with Java 1.8u45 on Mac OS X, we get the following output:

```
# iamedu at Eduardos-MacBook-Pro.local in ~/Development/clj/images/chapter06 [5:40:28]
$ java -XX:+PrintFlagsFinal -version | grep ThreadStackSize

    intx CompilerThreadStackSize          = 0                      {pd product}
    intx ThreadStackSize                 = 1024                  {pd product}
    intx VMThreadStackSize              = 1024                  {pd product}
java version "1.8.0_66"
Java(TM) SE Runtime Environment (build 1.8.0_66-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.66-b17, mixed mode)
```

I am getting a stack size of 1024 kilobytes per thread. What can we do to improve the numbers? Other languages, such as Erlang and Go create a few threads from the beginning and then execute their tasks using those threads. It becomes important to be able to suspend a particular task and run another in the same thread.

In Clojure there is a library called **Pulsar** (<https://github.com/puniverse/pulsar>), which is an interface for a Java API called **Quasar** (<https://github.com/puniverse/quasar>).

In order to support Pulsar, as of version 0.6.2, you need to do two things.

- Add the `[co.paralleluniverse/pulsar "0.6.2"]` dependency to your project
- Add an instrumentation agent to your JVM (adding `:java-agents [[co.paralleluniverse/quasar-core "0.6.2"]]` to your `project.clj`)

The instrumentation agent should be able to suspend functions in a thread and then change them for other functions. In the end, your `project.clj` file should look similar to:

```
(defproject clojure-concurrency "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 "1.6.0"]
                [co.paralleluniverse/pulsar "0.6.2"]]
  :java-agents [[co.paralleluniverse/quasar-core "0.6.2"]])
```

Let's write our last example with promises using Pulsar's lightweight threads called fibers.

Pulsar comes with its own promises in the `co.paralleluniverse.pulsar.core` package and they can be used as a drop-in replacement for promises in `clojure.core`:

```
(clojure.core/use 'co.paralleluniverse.pulsar.core)
(def p1 (promise))
(def p2 (promise))
(def p3 (promise))
(spawn-fiber #(clojure.core/deliver p2 (clojure.core/+ @p1 5)))
(spawn-fiber #(clojure.core/deliver p3 (clojure.core/+ @p1 @p2)))
(spawn-thread #(println @p3))
(clojure.core/deliver p1 99)
;; 203
```

This example is a bit more interesting, we use two of Pulsar's functions:

- `spawn-fiber`: This function creates a light thread, you can create thousands of fibers if you want in a single program. They are cheap to create and as long as you program them carefully, there shouldn't be many problems coming from this.
- `span-thread`: This is Pulsar's version of `start-thread`, it creates a real thread and runs it.

In this particular example, we calculate `p2` and `p3` in two fibers and then `p3` in a thread. At this point, everything is waiting for us to deliver the value of `p1`; we do it with the `deliver` function.

Pulsar has other very interesting features that allow for simpler parallel programming, have a look at the documentation if you are interested. In the last part of this chapter, we will have a look at `core.async`. Pulsar has an interface modelled after `core.async`, which you can use if you like.

# Futures

If you have been using Java for a while, you might know the `java.util.concurrent.Future` class, this is Clojure's implementation of futures and it is extremely similar to Java, only a bit simpler. Its interface and usage are almost identical to promises with a very important difference, when using futures everything will run in a different thread automatically.

Let's see a simple example using futures, in any REPL do the following:

```
(def f (future (Thread/sleep 20000) "Hello world"))
(prinln @f)
```

Your REPL should freeze for 20 seconds and then print `Hello world`.

## Tip

Futures are also cached, so you only need to pay once for the cost of calculation and then you can use them any number of times you want.

At first glance, futures look much easier than promises. You don't need to worry about creating threads or fibers, but there are downsides to this approach:

- You have less flexibility; you can only run futures in a predefined thread pool.
- You should be careful if your futures take too much time, they could end up NOT running because the implicit thread pool has a number of threads available. If they are all busy some of your tasks will end up queued and waiting.

Futures have their use cases, if you have very few processor intensive tasks, if you have IO bound tasks, maybe using promises with fibers is a good idea, since they allow you to keep your processor free to run more

tasks concurrently.

# Software transactional memory and refs

One of Clojure's most interesting features is **software transactional memory (STM)**. It uses **multiversion concurrency control (MVCC)**, in a very similar fashion to how databases work, implementing a form of optimistic concurrency control.

## Note

MVCC is what databases use for transactions; what this means is that every operation within a transaction has its own copy of variables. After executing its operations, it checks if any of the used variables changed during the transaction and if they did the transaction fails. This is called optimistic concurrency control because we are optimistic and we don't lock any variable; we let every thread do its work thinking that it's going to work correctly and then check if it was correct. In practice, this allows for greater concurrency.

Let's start with the most obvious example, a bank account.

Let's write some code now, enter into the REPL and write:

```
(def account (ref 20000))
(dosync (ref-set account 10))
(deref account)

(defn test []
  (dotimes [n 5]
    (println n @account)
    (Thread/sleep 2000))
  (ref-set account 90))

(future (dosync (test)))
(Thread/sleep 1000)
(dosync (ref-set account 5))
```

Try to write the future and the `dosync` functions at the same time so you have the same results.

We have just three lines of code here but there are quite a few things happening.

First of all we define a `ref` (`account`); refs are the managed variables in transactions. They are also the first implementation we see of Clojure's identity idea. Note that the account is an identity now and it might take multiple values throughout its life.

We now modify its value, we do this within a transaction since refs cannot be modified outside of transactions; thus, the `dosync` block.

In the end, we print the account and we can use (`deref account`) or `@account`, as we did for promises and futures.

Refs can be read from anywhere, there is no need for it to be within a transaction.

Let's look at something a little bit more interesting now, write or copy the next code into the REPL:

```
(def account (ref 20000))

(defn test []
  (println "Transaction started")
  (dotimes [n 5]
    (println n @account)
    (Thread/sleep 2000))
  (ref-set account 90))

(future (dosync (test)))
(future (dosync (Thread/sleep 4000) (ref-set account 5))))
```

If everything goes well, you should have an output similar to the following screenshot:

```
clojure-concurrency.core=> (future (dosync (test)))
Transaction started
0 20000
#object[clojure.core$future_call$reify__6736 0x490bc5f4 {:status :pending, :val nil}]
clojure-concurrency.core=> (future (dosync (Thread/sleep 4000) (ref-set account 5)))
#object[clojure.core$future_call$reify__6736 0x2d31b139 {:status :pending, :val nil}]
clojure-concurrency.core=> 1 20000
2 20000
Transaction started
0 5
1 5
2 5
3 5
4 5
```

This might seem a little strange, what is happening?

The first transaction starts its process using the current value for account, the other transaction then modifies the value of account before the first transaction is finished; Clojure realizes this and it restarts the first transaction.

## Tip

You shouldn't execute functions with side effects within transactions, as there is no guarantee that they will be executed only once. If you need to do something like that you should use an agent.

This is the first example of how a transaction works, but using `ref-set` is not a good idea in general.

Let's take a look at another example, the classic example of moving money from an account *A* to an account *B*:

```
(def account-a (ref 10000))
(def account-b (ref 2000))
(def started (clojure.core/promise))
```

```
(defn move [acc1 acc2 amount]
  (dosync
    (let [balance1 @acc1
          balance2 @acc2]
      (println "Transaction started")
      (clojure.core/deliver started true)
      (Thread/sleep 5000)
      (when (> balance1 amount)
        (alter acc1 - amount)
        (alter acc2 + amount)))
      (println "Transaction finished")))

(future (move account-a account-b 50))
@started
(dosync (ref-set account-a 20))
```

This is a better example of how transactions work; you will probably see something similar to the following screenshot:

```
" clojure.core=> (future (move account-a account-b 50))
#object[clojure.core$future_call$reify__6736 0x17f9691c {:status :pending, :val nil}]

Transaction started
clojure-concurrency.core=> @started
true
clojure-concurrency.core=> (dosync (ref-set account-a 20))
20
clojure-concurrency.core=>

clojure-concurrency.core=>
Transaction started

Transaction finished
```

First of all, you need to understand how the `alter` function works; it's simple and it receives:

- The ref that it has to modify

- The function that it has to apply
- The extra arguments

So this function:

```
(alter ref fun arg1 arg2)
```

Is translated to something like this:

```
(ref-set ref (fun @ref arg1 arg2))
```

This is the preferred way to modify the current value.

Let's see a step by step description of what's going on here:

1. We define two accounts with a balance of 10000 and 2000.
2. We try to move 500 units from the first account to the second but first we sleep for 5 seconds.
3. We announce (using the promise) that we have started the transaction. The current thread moves on, since it was waiting for the started value.
4. We set the balance of account-a to 20.
5. The first transaction realizes that something has changed and restarts.
6. The transaction goes on and is finished this time.
7. Nothing happens, since the new balance is not enough to move 50 units.

In the end, if you check the balance, like `[@account-a @account-b]`, you will see that the first account has 20 and the second account has 2000.

There is another use case that you should take into account; let's check the following code:

```
(def account (ref 1000))
(def secured (ref false))
(def started (promise))
```

```
(defn withdraw [account amount secured]
  (dosync
    (let [secured-value @secured]
      (deliver started true)
      (Thread/sleep 5000)
      (println :started)
      (when-not secured-value
        (alter account - amount)))
      (println :finished)))))

(future (withdraw account 500 secured))
@started
(dosync (ref-set secured true))
```

The idea is that if `secured` is set to true, you shouldn't be able to withdraw any money.

If you run it and then check for the value of `@account`, you'll see that even after changing the value of `secured` to true a withdrawal occurs. Why is that?

The reason is that transactions only check for values that you modify within a transaction or values that you read; here we are reading the `secured` value before the modification, so the transaction doesn't fail. We can tell the transaction to be a little bit more careful by using the following code:

```
(ensure secured)
;; instead of
@secured

(def account (ref 1000))
(def secured (ref false))
(def started (promise))

(defn withdraw [account amount secured]
  (dosync
    (let [secured-value (ensure secured)]
      (deliver started true)
```

```
(Thread/sleep 5000)
(println :started)
(when-not secured-value
  (alter account - amount))
(println :finished) )))  
  
(future (withdraw account 500 secured) )
@started
(dosync (ref-set secured true))
```

Here almost the same thing happened. What is the difference?

There is one subtle difference, the second transaction can't finish until the first transaction is done. If you look at it in detail, you will notice that you can't modify the secured value until after the other transaction runs.

This is similar to a lock; not the best idea but useful in some cases.

# Atoms

We have now seen how promises, futures, and transactions work in Clojure. We'll now see atoms.

Even though STM is very useful and powerful you'll see that in practice it is not very commonly used.

Atoms are one of Clojure's workhorses, when it comes to concurrent programming.

You can think of atoms as transactions that modify one single value. You might be thinking, what good is that? Imagine you had lots of events that you want to store in a single vector. If you are used to Java, you probably know that using a `java.util.ArrayList` package is bound to have problems; since, you are almost surely going to lose data.

In that case, you probably want to use a class from the `java.util.concurrent` package, how can you guarantee that you'll have no data loss in Clojure?

It's easy, atoms come to the rescue! Let's try this piece of code:

```
(clojure.core/use 'co.paralleluniverse.pulsar.core)
(def events (atom []))
(defn log-events [count event-id]
  (dotimes [_ count]
    (swap! events conj event-id)))
(dotimes [n 5]
  (spawn-fiber #(log-events 500 n)))
```

We are again using Pulsar and its lightweight threads. Here, we define an events atom and a `log-events` function.

The `log-events` execute a `swap!` a given number of times.

`Swap!` is similar to the `alter` function it receives:

- The atom that it should modify
- The function that it applies to the atom
- The extra arguments

In this case, it gives the atom the new value that comes from:

```
(conj events event-id)
```

We then spawn five fibers, each fiber adds 500 events to the events atom.

After running this, we can check for the number of events from each fiber like this:

```
(count (filter #(= 0 %) @events))  
;; 500  
(count (filter #(= 1 %) @events))  
;; 500  
(count (filter #(= 2 %) @events))  
;; 500  
(count (filter #(= 3 %) @events))  
;; 500  
(count (filter #(= 4 %) @events))  
;; 500
```

As you can see, we have 500 elements from each fiber with no data loss and using Clojure's default data structures. There is no need to use special data structures for each use case, locks, or mutexes. This allows for greater concurrency.

When you modify an atom, you need to wait for the operation to be complete meaning it is synchronous.

# Agents

What if you don't care about the result of some operation? You just need to fire something and then forget it. In that case agents are what you need.

Agents also run in separate thread pools, there are two functions that you can use to fire an agent:

- `send`: This executes your function in an implicit thread pool
- `send-off`: This tries to execute your function in a new thread but there's a change, it will reuse another thread

Agents are the way to go if you want to cause side effects in a transaction; since, they will only be executed after the transaction has completed successfully.

They work in a very simple manner, here is an example usage:

```
(def agt (agent 0))
(defn sum [& nums]
  (Thread/sleep 5000)
  (println :done)
  (apply + nums))
(send agt sum 10) ; You can replace send with send-off
                  ; if you want this to be executed in a
different thread
@agt
```

If you copy and paste the exact previous code you will see a `0` and then a `:done` message, if you check for the value of `@agt`, then you will see the value `10`.

Agents are a good way to execute a given task and modify some value in a different thread with simpler semantics than those of futures or manually modifying values in another thread.

# Validators

We have seen the primary concurrency primitives now, let's see some utilities that apply to all of them at once.

We can define a validator that checks if the new value of some function is desirable or not; you can use them for refs, atoms, agents, and even vars.

The `validator` function must receive a single value and return true if the new value is valid or false otherwise.

Let's create a simple `validator` that checks if the new value is lower than 5:

```
(def v (atom 0))
(set-validator! v #(< % 5))
(swap! v + 10)

;; IllegalStateException Invalid reference state
clojure.lang.ARef.validate (ARef.java:33)
```

We get an exception. The reason is that the new value (10) is not valid.

You can add 4 without a problem:

```
(swap! v + 4)
;; 4
```

Be careful with the validator and agents, since you will probably not know when an exception occurred:

```
(def v (agent 0))
(set-validator! v #(< % 5))
(swap! v + 10)
;; THERE IS NO EXCEPTION
```

## Watchers

Similar to validators, there are also watchers. Watchers are functions that are executed whenever Clojure's identities get a new value. An important

question is the thread in which watchers run. Watchers run in the same thread as the watched entity (if you add a watcher to an agent it will be run in the agent's thread), it will be run before the agent code executes, so you should be careful and use the old-value new-value instead of reading the value with `deref`:

```
(def v (atom 0))
(add-watch v :sample (fn [k i old-value new-value] (println (= i
v) k old-value new-value)))
(reset! v 5)
```

The `add-watch` function receives:

- The ref, atom, agent, or var that should be watched
- A key that will be passed to the watcher function
- A function with four parameters: the key, the reference itself, the old value, and the new value

After executing the previous code we get:

```
true :sample 0 5
```

# core.async

The `core.async` is yet another way of programming concurrently; it uses the idea of lightweight threads and channels to communicate between them.

## Why lightweight threads?

The lightweight threads are used in languages, such as go and Erlang. They excel in being able to run thousands of threads in a single process.

What is the difference between the lightweight threads and traditional threads?

The traditional threads need to reserve memory. This also takes some time. If you want to create a couple of thousand threads, you will be using a noticeable amount of memory for each thread; asking the kernel to do that also takes time.

What is the difference with lightweight threads? To have a couple of hundred lightweight threads, you only need to create a couple of threads. There is no need to reserve memory and lightweight threads are a mere software idea.

This can be achieved with most languages and Clojure is adding first class support (without changing the language this is part of the Lisp power) with using `core.async`! Let's have a look at how it works.

There are two concepts that you need to keep in mind:

- **Gblocks:** They are the lightweight threads.
- **Channels:** Channels are a way to communicate between gblocks, you can think of them as queues. Gblocks can publish a message to the channel and other gblocks can take a message from them. Just as

there are integration patterns for queues, there are integration patterns for channels and you will find concepts similar to broadcasting, filtering, and mapping.

Now, let's play a little with each of them so you can understand better how to use them for our program.

## Goblocks

You will find goblocks in the `clojure.core.async` namespace.

Goblocks are extremely easy to use, you need the `go` macro and you will do something similar to this:

```
(ns test
  (:require [clojure.core.async :refer [go]]))

(go
  (println "Running in a goblock!"))
```

They are similar to threads; you just need to remember that you can create goblocks freely. There can be thousands of running goblocks in a single JVM.

## Channels

You can actually use anything you like to communicate between goblocks, but it is recommended that you use channels.

Channels have two main operations: putting and getting. Let's check how to do it:

```
(ns test
  (:require [clojure.core.async :refer [go chan >! <!]]))

(let [c (chan)]
  (go (println (str "The data in the channel is" (<! c)))))
  (go (>! c 6)))
```

That's it!! It looks pretty simple, as you can see there are three main functions that we are using with channels:

- `chan`: This function creates a channel and the channels can store some messages in a buffer. If you want this functionality, you should just pass the size of the buffer to the `chan` function. If no size is specified, the channel can store only one message.
- `>!`: The `put` function must be used within a goblock; it receives a channel and the value you want to publish to it. This function blocks, if a channel's buffer is already full. It will block until something is consumed from the channel.
- `<!`: This takes function; this function must be used within a goblock. It receives the channel you are taking from. It is blocking and if you haven't published something in the channel it will park until there's data available.

There are lots of other functions that you can use with channels, for now let's add two related functions that you will probably use soon:

- `>!!`: The blocking `put`, works exactly the same as the `put` function; except it can be used from anywhere. Note that if a channel cannot take more data, this function will block the entire thread from where it runs.
- `<!!`: The blocking works exactly the same as the `take` function, except you can use this from anywhere and not just from inside goblocks. Just keep in mind that this blocks the thread where it runs until there's data available.

If you look into the `core.async` API docs (<http://clojure.github.io/core.async/>) you will find a fair amount of functions.

Some of them look similar to the functions that give you functionalities similar to queues, let's take a look at the `broadcast` function:

```
(ns test
```

```
(:require [clojure.core.async.lab :refer [broadcast]]
          [clojure.core.async :refer [chan <!>!! go-loop]]))

(let [c1 (chan 5)
      c2 (chan 5)
      bc (broadcast c1 c2)]
  (go-loop []
    (println "Getting from the first channel" (<! c1))
    (recur))
  (go-loop []
    (println "Getting from the second channel" (<! c2))
    (recur)))
  (>!! bc 5)
  (>!! bc 9))
```

With this you can publish it to several channels at the same time, this is helpful if you want to subscribe multiple processes to a single source of events with a great amount of separation of concern.

If you take a good look, you will also find familiar functions over there: `map`, `filter`, and `reduce`.

## Note

Depending on the version of `core.async`, some of these functions might not be there anymore.

Why are these functions there? Those functions are meant to modify collections of data, right?

The reason is that there has been a good amount of effort towards using channels as higher-level abstractions.

The idea is to see channels as collections of events, if you think of them that way it's easy to see that you can create a new channel by mapping every element of an old channel or you can create a new channel by filtering away some elements.

In recent versions of Clojure, the abstraction has become even more noticeable with transducers.

## Transducers

Transducers are a way to separate the computations from the input source. Simply, they are a way to apply a sequence of steps to a sequence or a channel.

Let's look at an example of a sequence:

```
(let [odd-counts (comp (map count)
                        (filter odd?))
      vs [[1 2 3 4 5 6]
          [:a :c :d :e]
          [:test]]]
  (sequence odd-counts vs))
```

The `comp` feels similar to the threading macros, it composes functions and stores the steps of the computation.

The interesting part is that we can use the same odd-counts transformation with a channel, such as:

```
(let [odd-counts (comp (map count)
                        (filter odd?))
      input (chan)
      output (chan 5 odd-counts)]
  (go-loop []
    (let [x (<! output)]
      (println x))
      (recur)))
  (>!! input [1 2 3 4 5 6])
  (>!! input [:a :c :d :e])
  (>!! input [:test]))
```

# Summary

We have checked the core Clojure mechanisms for concurrent programming, as you can see, they feel natural and they build on already existing paradigms, such as immutability. The most important idea is what an identity and value is; we now know that we can have the following values as identifiers:

- Refs
- Atoms
- Agents

We can also get the snapshot of their value with the defer function or the @ shortcut.

If we want to use something a little more primitive, we can use promises or futures.

We have also seen how to use threads, or Pulsar's fibers. Most of Clojure's primitives aren't specific to some concurrency mechanism, so we can use any parallel programming mechanism with any type of Clojure primitive.

# Chapter 7. Macros in Clojure

In this chapter, we will get to know one of Clojure's most complicated facilities: macros. We will learn what they are for, how to write them, and how to use them. It can be a little challenging, but there is good news too. You should be aware of some tools from your knowledge of the Java language that can help you understand macros better. We will progress little by little with comparisons to other JVM languages, and in the end, we will write some macros and understand that we have been using them for a while.

We will learn about the following topics:

- Understanding Lisp's foundational ideas
- Macros as code modification tools
- Modifying code in Groovy
- Writing your first macro
- Debugging your first macro
- Macros in the real world

## Lisp's foundational ideas

Lisp is a very different beast from what you used to know. According to Paul Graham, there are nine ideas that make Lisp different (these ideas have existed since the late 1950s), and they are:

1. Conditionals (remember, we are talking 1950s–1960s)
2. Functions as first-class citizens
3. Recursion
4. Dynamic typing
5. Garbage collection
6. Programs as sequences of expressions
7. The symbol type
8. Lisp's syntax

9. The whole language is there all the time: at compilation, runtime—always!

## Note

If you can, read Paul Graham's essay *Revenge of the Nerds* (<http://www.paulgraham.com/icad.html>), where he talks about Lisp, what makes it different, and why the language is important.

These ideas have thrived even after the Lisp age; most of them are common nowadays (can you imagine a language without conditionals?). But the last couple of ideas are what makes us Lisp lovers love the syntax (we will fully understand what they mean through this chapter).

Common languages are trying to achieve the very same things now with a slightly different approach, and you, as a Java developer, have probably seen this.

# Macros as code modification tools

One of the first and most common uses of macros is to be able to modify code; they work on the code level, as you will see. Why should we do that? Let's try to understand the problem with something that you are more familiar with—Java.

## Modifying code in Java

Have you ever used AspectJ or Spring AOP? Have you ever had problems with tools such as ASM or Javassist?

You have probably used code modification in Java. It is common in Java EE applications, just not explicit. (Have you ever thought about what the `@Transactional` annotation does in Java EE or Spring applications?)

As developers, we try to automate everything we can, so how could we leave out our own devtools?

We have tried to create ways to modify the bytecode at runtime so that we don't have to remember to open and close resources, or so that we can decouple dependencies and get dependency injection.

If you use Spring, you probably know about the following use cases:

- The `@Transactional` annotation modifies the annotated method to ensure that your code is wrapped in a database transaction
- The `@Autowired` annotation looks for the required bean and injects it into the annotated property or method
- The `@Value` annotation looks for a configuration value and then injects it

You could probably think of several other annotations that modify the way your classes work.

The important thing here is that you understand why we want to modify code, and you probably already know a few mechanisms for doing it, including AspectJ and Spring AOP.

Let's take a look at how it is done in the Java world; this is what an aspect in Java looks like:

```
package macros.java;

public aspect SampleJavaAspect {
    pointcut anyOperation() : execution(public * *.*(..));

    Object around() : anyOperation() {
        System.out.println("We are about to execute this " +
                           thisJoinPointStaticPart.getSignature());
        Object ret = proceed();
        return ret;
    }
}
```

Aspects have the advantage that you can modify any code you like without having to touch it. This also has its drawbacks since you could modify the code in ways the original author didn't expect and thus cause bugs.

Another drawback is that you have an extremely limited field of action; you can wrap your modifications around some code or execute something before or after.

The libraries that generate this code are extremely complex and they can either create a proxy around your objects or modify the bytecode, at runtime or compile time.

As you can imagine, there are lots of things that you must be aware of, and anything could go wrong. Hence, debugging could prove complicated.

## Modifying code in Groovy

Groovy has gone further down the road and it provides us with more solutions and more macro-like features.

Since Groovy 1.8, we have got a lot of AST transformations. What does AST stand for? It stands for **abstract syntax tree**—sounds complicated, right?

Before explaining it all, let's check what some of them do.

## The `@ToString` annotation

The `@ToString` annotation generates a simple `toString` method that includes information about the class of the object and the value of its properties.

## The `@TupleConstructor` annotation

The `@TupleConstructor` creates a constructor that is able to take all of the values of your class at once. Here is an example:

```
@TupleConstructor  
class SampleData {  
    int size  
    String color  
    boolean big  
}  
  
new SampleData(5, "red", false) // We didn't write this  
constructor
```

## The `@Slf4j` annotation

The `@Slf4j` annotation adds an instance of a logger, called `log` by default, to your class, so you can do this:

```
log.info('hello world')
```

This can be done without having to manually declare the `log` instance, the class name, and so on. There are lots of other things that you can do with

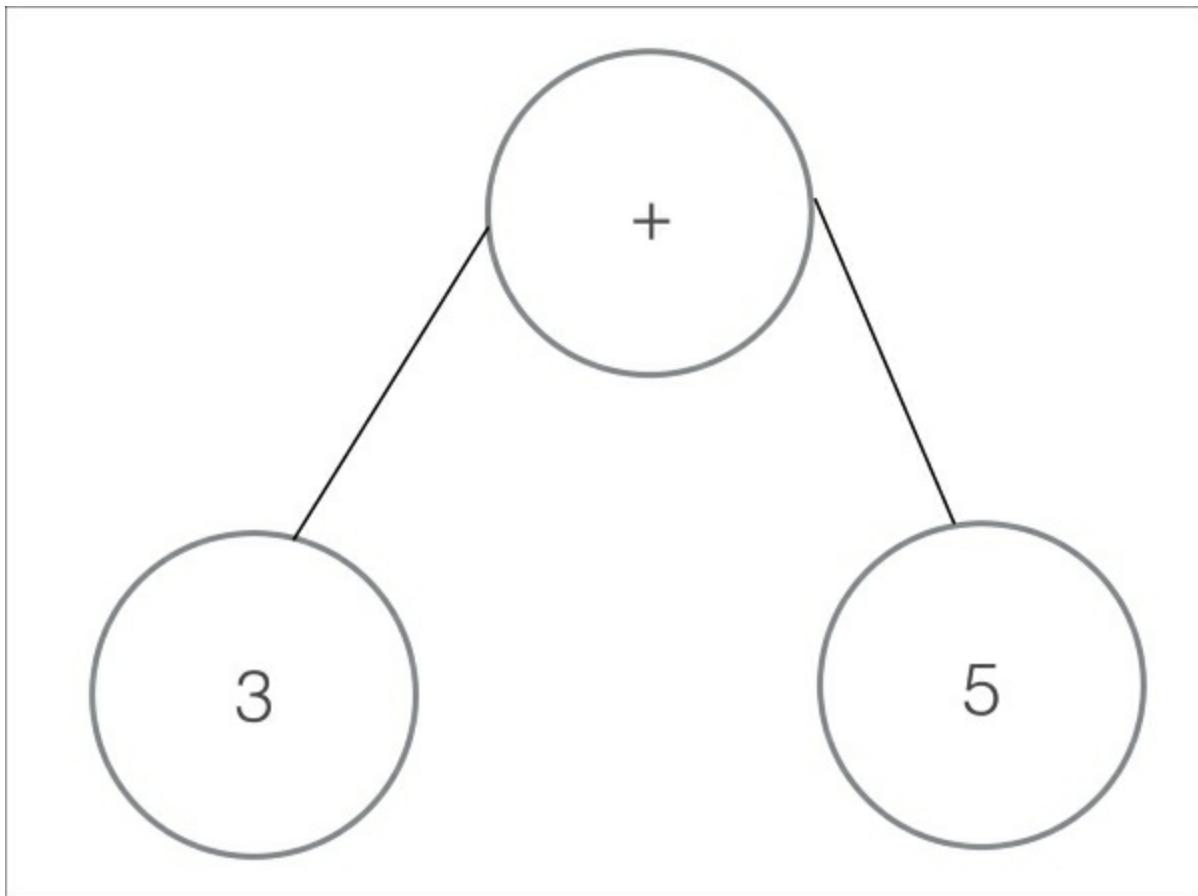
this type of annotation, but how do they work?

Now, what is AST and what does it have to do with Clojure macros? Come to think of it, it actually has a lot to do with them.

To answer that last question, you'll have to understand a little bit about how compilers work.

We all know that machines (your machine, the JVM, the Erlang BEAM machine) are not capable of understanding human code, so we need a process to convert whatever developers write into what machines understand.

One of the most important steps of the process is to create a syntax tree, something similar to the following figure:



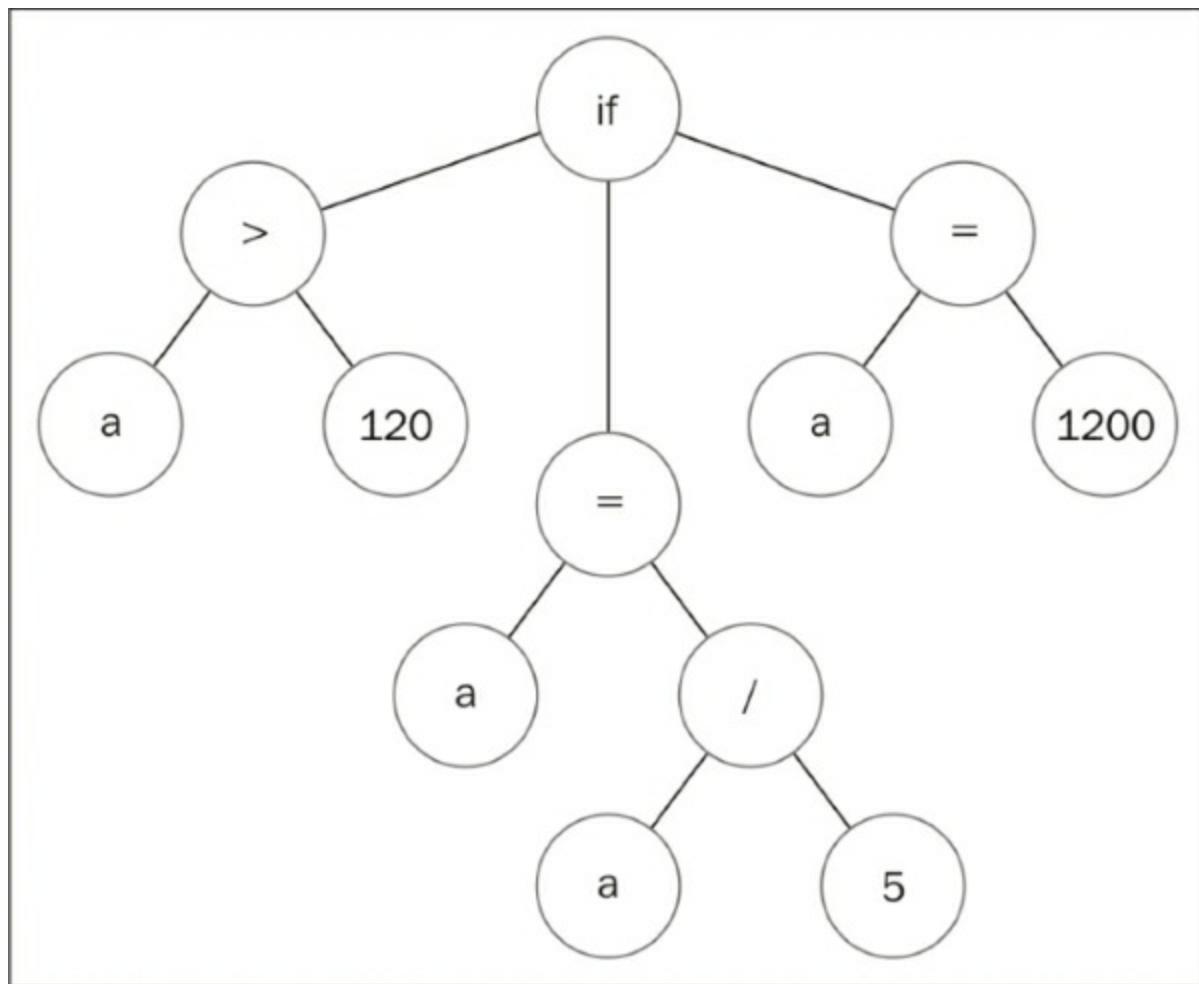
This is a very simple example of the following expression:

3 + 5

This tree is what we call the abstract syntax tree. Let's see the tree of something that's a bit more complicated, such as this piece of code:

```
if(a > 120) {  
    a = a / 5  
} else {  
    a = 1200  
}
```

Thus, the tree will look like the following figure:



As you can see, the figure is still pretty straightforward, and you can

probably understand how someone would execute code from a structure like this one.

Groovy's AST transformation is a way to meddle with such generated code.

As you can imagine, this is a much more powerful approach, but you are now messing with what the compiler generated; the probable downside to this is the complexity of the code.

Let's check, for instance, the code of the `@Slf4j` AST. It should be pretty simple, right? It just adds a log property:

```
private Expression  
transformMethodCallExpression(Expression exp) {  
    MethodCallExpression mce = (MethodCallExpression) exp;  
    if (!(mce.getObjectExpression()  
instanceof VariableExpression)) {  
        return exp;  
    }  
    VariableExpression variableExpression = (VariableExpression)  
mce.getObjectExpression();  
    if  
    (!variableExpression.getName().equals(logFieldName)  
        || !  
(variableExpression.getAccessedVariable()  
instanceof DynamicVariable)) {  
        return exp;  
    }  
    String methodName = mce.getMethodAsString();  
    if (methodName == null) return exp;  
    if (usesSimpleMethodArgumentsOnly(mce)) return  
exp;  
  
    variableExpression.setAccessedVariable(logNode);  
  
    if (!loggingStrategy.isLoggingMethod(methodName))  
return exp;  
  
    return  
loggingStrategy.wrapLoggingMethodCall(variableExpression,
```

```
methodName, exp);  
}
```

## Note

You can check the complete code at <https://github.com/groovy/groovy-core/blob/master/src/main/org/codehaus/groovy/transform/LogASTTransfo> and it's also included with the code bundle of this chapter.

This doesn't look simple at all. It is just a fragment and still looks very complicated. What happens here is that you have to deal with the Java bytecode format and with compiler complications.

Here, we should remember point number 8 that Paul Graham made about the syntax of Lisp.

Let's write our last code example in Clojure:

```
(if (> a 120)  
    (/ a 5)  
    1200)
```

There's something peculiar about this piece of code: it feels very similar to the AST! This is not a coincidence. Actually, in Clojure and Lisp, you are directly writing the AST. This is one of the features that make Lisp a very simple language; you directly write what the computer understands. This might help you understand a little more about why code is data and data is code.

Imagine if you could modify the AST the same way that you modify any other data structure in your programs. But you can, and that's what macros are for!

# Writing your first macro

Now that you have a clear understanding of how macros work and what they are for, let's start working with Clojure.

Let me present you with a challenge: write an `unless` function in Clojure, something that works like this:

```
(def a 150)

(my-if (> a 200)
       (println "Bigger than 200")
       (println "Smaller than 200"))
```

Let's give it a first try; maybe with something like the following syntax:

```
(defn my-if [cond positive negative]
  (if cond
    positive
    negative))
```

Do you know what would happen if you wrote this code and then ran it? If you test it, you will get the following result:

```
Bigger than 200
Smaller than 200
Nil
```

What's happening here? Let's modify it a bit so that we get a value and we can understand what's happening. Let's define it a bit differently, and let's return a value so that we see something different:

```
(def a 500)
(my-if (> a 200)
  (do
    (println "Bigger than 200")
    :bigger)
  (do
    (println "Smaller than 200")))
```

```
:smaller))
```

We will get the following output:

```
Bigger than 200
Smaller than 200
:bigger
```

What's going on here?

When you pass parameters to a function, everything is evaluated before the actual code of the function runs, so over here, before the body of your function runs, you execute both of the `println` methods. After that, the `if` runs correctly and you get `:bigger`, but we still got an output for the positive and negative cases of our `if`. It looks like our code is not working!

How can we fix this? With our current tools, we probably need to write closures and change the `my-if` code to accept functions as parameters:

```
(defn my-if [cond positive negative]
  (if cond
    (positive)
    (negative)))

(def a 500)
(my-if (> a 200)
  #(do
    (println "Bigger than 200")
    :bigger)
  #(do
    (println "Smaller than 200")
    :smaller))
```

This works, but there are several disadvantages:

- There are a lot of constraints now for the code (both clauses should now be functions)
- It doesn't work for every single case
- It is very complicated

In order to solve this problem, Clojure gives us macros. Let's have a look at how they work:

```
(defmacro my-if [test positive negative]
  (list 'if test positive negative))

(my-if (> a 200)
  (do
    (println "Bigger than 200")
    :bigger)
  (do
    (println "Smaller than 200")
    :smaller))
```

The output will be this:

```
;; Bigger than 200
;; :bigger
```

This is great! It works, but what just happened? Why did we just use a macro and why did it work?

## Note

Macros are not normal Clojure functions; they are supposed to generate code and should return a Clojure form. This means that they should return a list that we can use as normal Clojure code.

Macros return code that will be executed later. And here is where point number nine of Paul Graham's list comes into play: you have all of the language all the time.

In C++, you have a mechanism called a macro; when you use it, you have a very limited set of things that you can do compared to actual C++ code.

In Clojure, you can manipulate the Clojure code any way you want, and you can use the full language here too! Since Clojure code is data, manipulating the code is as easy as manipulating any other data structure.

## **Note**

Macros are run at compile time, which means that at the time of running the code, there is no trace of macros; every macro call is replaced with the generated code.

# Debugging your first macro

Now, as you can imagine, since things can get complicated when using macros, there should be some way to debug them. We have two functions to accomplish that:

- `macroexpand`
- `macroexpand-1`

The difference between them has to do with recursive macros. There is no rule telling you that you can't use a macro from a macro (the whole language is there all the time, remember?). If you wish to go all the way through any macro, you can use `macroexpand`; if you wish to go a single step forward, you can use `macroexpand-1`.

Both of them show you the code generated by a macro call; this is what happens when you compile your Clojure code.

Give this a try:

```
(macroexpand-1
' (my-if (> a 200)
  (do
    (println "Bigger than 200")
    :bigger)
  (do
    (println "Smaller than 200")
    :smaller)))
;; (if (> a 200) (do (println "Bigger than 200") :bigger) (do
  (println "Smaller than 200") :smaller))
```

There is not much more to macros than this; you now understand them to a good level of detail.

There are, however, many common problems that you will come across and tools for solving them that you should know about. Let's have a look.

# Quote, syntax quote, and unquoting

As you can see, the `my-if` macro uses a quote in it:

```
(defmacro my-if [test positive negative]
  (list 'if test positive negative))
```

This happens because you need the `if` symbol as the first element in the resulting form.

Quoting is very common in macros, since we need to build code instead of evaluating it on the fly.

There is another type of quoting very common in macros—syntax quoting—that makes it easier to write code similar to the final code you want to generate. Let's change the implementation of our macro to this:

```
(defmacro my-if [test positive negative]
  '(if test positive negative))

(macroexpand-1
 ' (my-if (> a 200)
   (do
     (println "Bigger than 200")
     :bigger)
   (do
     (println "Smaller than 200")
     :smaller)))
;; (if clojure.core/test user/positive user/negative)
```

Let's see what happens here. For one, `(if test positive negative)` looks much more beautiful than the `list` function we had before, but the code generated with `macroexpand-1` looks pretty strange. What happened?

We just used a different form of quoting that allows us to quote full

expressions. It does some interesting things. As you can see, it changes the parameters to fully qualified var names (`clojure.core/test`, `user/positive`, `user/negative`). This is something that you'll be grateful for in the future, but you don't need this for now.

What you need are the values of test, positive, and negative. How can you get them in this macro?

Using syntax quotes, you can ask for something to be evaluated inline with the unquote operator, like this:

```
(defmacro my-if [test positive negative]
  (if ~test ~positive ~negative))
```

Let's try our macro expansion again and see what we get:

```
user=> (defmacro my-if [test positive negative]
#_=>   `(if ~test ~positive ~negative))
#'user/my-if
user=> (macroexpand-1
#_=>   '(my-if (> a 200)
#_=>     (do
#_=>       (println "Bigger than 200")
#_=>       :bigger)
#_=>     (do
#_=>       (println "Smaller than 200")
#_=>       :smaller)))
(if (> a 200) (do (println "Bigger than 200") :bigger) (do (println "Smaller than 200") :smaller))
```

## Unquote splicing

There are some other cases that become common in macros. Let's imagine we want to reimplement the `>` function as a macro and retain the ability to compare several numbers; what would that look like?

Maybe a first attempt could be something like this:

```
(defmacro>-macro [&params]
  `(> ~params))
```

```
(macroexpand' (>-macro 5 4 3))
```

The output of the preceding code is as follows:

```
user=> (defmacro >-macro [& params]
#_=> `(> ~params))
#'user/>-macro
user=>

user=> (macroexpand'(>-macro 5 4 3))
(clojure.core/> (5 4 3))
user=> (>-macro 5 4 3)
ClassCastException java.lang.Long cannot be cast to clojure.lang.IFn  user/eval1219 (form-init7702615529919007918.clj:1)
```

Do you see the problem here?

The problem is that we are trying to pass a list of values to `clojure.core/>` instead of passing the values themselves.

This is easily solved with something called **unquote splicing**. Unquote splicing takes a vector or list of parameters and expands it as if you had used the `as` parameter on a function or macro.

It works like this:

```
(defmacro>-macro [&params]
'(> ~@params)) ;; In the end this works as if you had written
;; (> 5 4 3)

(macroexpand' (>-macro 5 4 3))
```

The output of the preceding code is as follows:

```
user=> (defmacro >-macro [& params]
  #'>-macro
user=> ;; (gt ~@params)) ; In the end this works as if you had written
user=> ;; (gt 5 4 3)

user=>

user=> (macroexpand '(>-macro 5 4 3))
(clojure.core/> 5 4 3)
user=> (>-macro 5 4 3)
true
```

You will use unquote splicing almost every time you have a variable number of arguments to a macro.

## gensym

Generating code can be troublesome, and we end up discovering common issues.

See if you can find the issue in the following code:

```
(def a-var "hello world")

(defmacro error-macro [&params]
  '(let [a-var "bye world"]
    (println a-var)))

;; (macroexpand-1 '(error-macro))
;; (clojure.core/let [user/a-var user/"bye user/world"]
(clojure.core/println user/a-var))
```

This is a common issue when generating code. You overwrite another value, Clojure doesn't even let you run this, and it displays something like the following screenshot:

```
user=> (def a-var "hello world")
#'user/a-var
user=> (defmacro error-macro [& params]
#_=> `(let [a-var "bye world"]
#_=> (println a-var)))
#'user/error-macro
user=> (macroexpand-1 '(error-macro))
(clojure.core/let [user/a-var "bye world"] (clojure.core/println user/a-var))
user=> (error-macro)

CompilerException java.lang.RuntimeException: Can't let qualified name: user/a-var, compiling:(/private/var/folders/4s/yxd1cnqn17dd30b_z4b27kyw0000gn/T/form-init7702615529919007918.clj:1:1)
```

But don't worry; there's another way in which you can make sure you are not messing with your environment, which is the `gensym` function:

```
(defmacro error-macro [&params]
(let [a-var-name (gensym'a-var)]
` (let [~a-var-name "bye world"]
(println ~a-var-name) )))
```

The `gensym` function creates a new `var-name` each time the macro is run, which guarantees that there is no other `var-name` that it obscures. If you try the macro expansion now, you will get this:

```
(clojure.core/let [a-var922"bye world"] (clojure.core/println a-var922))
```

The following screenshot is the result of the preceding code:

```
user=> (def a-var "hello world")
#'user/a-var
user=> (defmacro error-macro [& params]
  #_=>  (let [a-var-name (gensym 'a-var)]
  #_=>    `(let [~a-var-name "bye world"]
  #_=>      (println ~a-var-name)))
#'user/error-macro
user=> (macroexpand-1 '(error-macro))
(clojure.core/let [a-var1274 "bye world"] (clojure.core/println a-var1274))
user=> (error-
error-handler  error-macro      error-mode
user=> (error-
error-macro  error-mode
user=> (error-macro)
bye world
nil
```

# Macros in the real world

Do you want to know when it is that macros are used extensively? Think about `defn`; what's more, do this:

```
(macroexpand-1 '(defn sample [a] (println a)))  
;; (def sample (clojure.core/fn ([a] (println a))))
```

Did you know that `defn` is a macro in `clojure.core` that creates a function and binds it to a `var` in the current namespace?

Clojure is filled with macros; if you want some samples, you can look at Clojure core, but what else can you do with macros?

Let's have a look at some interesting libraries:

- `yesql`: The `yesql` library is a very interesting sample of code generation. It reads SQL code from a SQL file and generates the Clojure functions accordingly. Look for the `defquery` and `defqueries` macros in the `yesql` project on GitHub; it can be very enlightening.
- `core.async`: If you are familiar with the `go` language and `goroutines`, you would probably like to have that same functionality in the Clojure language. This isn't necessary since you could have provided them yourself! The `core.async` library is just `goroutines` for Clojure, and it is provided as a library (no obscure language change is needed). This shows a great example of the power of macros.
- `core.typed`: With macros, you can even change the dynamic nature of Lisp. The `core.typed` library is an effort that allows you to define type constraints for your Clojure code; macros are extensively used here to generate boilerplate code and checks. This is probably much more complex.

# References

If you need further references, you can look at the following list. There are entire books committed to the topic of macros. I recommend two in particular:

- Mastering Clojure Macros (<https://pragprog.com/book/cjclojure/>).
- Let over Lambda (<http://letoverlambda.com/>). It talks about common Lisp, but the knowledge is very valuable.

# Summary

You now understand the power of macros and have a very strong grasp of how they work, but we just touched the tip of the iceberg when it comes to macros.

In this chapter, we learned about the following:

- Fundamentals of how macros work
- Modifying your code in Groovy
- The relation of macros to other tools in the Java world
- Writing your own macros

I am sure you've enjoyed working with Clojure so far, and moving forward, I'd recommend you to keep reading and exploring this amazing language.

# Part 2. Module 2

*Clojure High Performance Programming, Second Edition*

*Become an expert at writing fast and high performant code in Clojure 1.7.0*

# Chapter 1. Performance by Design

Clojure is a safe, functional programming language that brings great power and simplicity to the user. Clojure is also dynamically and strongly typed, and has very good performance characteristics. Naturally, every activity performed on a computer has an associated cost. What constitutes acceptable performance varies from one use-case and workload to another. In today's world, performance is even the determining factor for several kinds of applications. We will discuss Clojure (which runs on the **JVM (Java Virtual Machine)**), and its runtime environment in the light of performance, which is the goal of this book.

The performance of Clojure applications depend on various factors. For a given application, understanding its use cases, design and implementation, algorithms, resource requirements and alignment with the hardware, and the underlying software capabilities is essential. In this chapter, we will study the basics of performance analysis, including the following:

- Classifying the performance anticipations by the use cases types
- Outlining the structured approach to analyze performance
- A glossary of terms, commonly used to discuss performance aspects
- The performance numbers that every programmer should know

## Use case classification

The performance requirements and priority vary across the different kinds of use cases. We need to determine what constitutes acceptable performance for the various kinds of use cases. Hence, we classify them to identify their performance model. When it comes to details, there is no sure shot performance recipe for any kind of use case, but it certainly helps to study their general nature. Note that in real life, the use cases listed in this section may overlap with each other.

## The user-facing software

The performance of user-facing applications is strongly linked to the user's anticipation. Having a difference of a good number of milliseconds may not be perceptible for the user but at the same time, a wait of more than a few seconds may not be taken kindly. One important element in normalizing anticipation is to engage the user by providing duration-based feedback. A good idea to deal with such a scenario would be to start the task asynchronously in the background, and poll it from the UI layer to generate a duration-based feedback for the user. Another way could be to incrementally render the results to the user to even out the anticipation.

Anticipation is not the only factor in user facing performance. Common techniques like staging or precomputation of data, and other general optimization techniques can go a long way to improve the user experience with respect to performance. Bear in mind that all kinds of user facing interfaces fall into this use case category—the Web, mobile web, GUI, command line, touch, voice-operated, gesture...you name it.

## Computational and data-processing tasks

Non-trivial compute intensive tasks demand a proportional amount of computational resources. All of the CPU, cache, memory, efficiency and the parallelizability of the computation algorithms would be involved in determining the performance. When the computation is combined with distribution over a network or reading from/staging to disk, I/O bound factors come into play. This class of workloads can be further subclassified into more specific use cases.

### A CPU bound computation

A CPU bound computation is limited by the CPU cycles spent on executing it. Arithmetic processing in a loop, small matrix multiplication, determining whether a number is a **Mersenne prime**, and so on, would be considered CPU bound jobs. If the algorithm complexity is linked to the number of iterations/operations  $N$ , such as  $O(N)$ ,  $O(N^2)$  and more, then the performance depends on how big  $N$  is, and how many CPU cycles each

step takes. For parallelizable algorithms, performance of such tasks may be enhanced by assigning multiple CPU cores to the task. On virtual hardware, the performance may be impacted if the CPU cycles are available in bursts.

## A memory bound task

A memory bound task is limited by the availability and bandwidth of the memory. Examples include large text processing, list processing, and more. For example, specifically in Clojure, the `(reduce f (pmap g coll))` operation would be memory bound if `coll` is a large sequence of big maps, even though we parallelize the operation using `pmap` here. Note that higher CPU resources cannot help when memory is the bottleneck, and vice versa. Lack of availability of memory may force you to process smaller chunks of data at a time, even if you have enough CPU resources at your disposal. If the maximum speed of your memory is  $X$  and your algorithm on single the core accesses the memory at speed  $X/3$ , the multicore performance of your algorithm cannot exceed three times the current performance, no matter how many CPU cores you assign to it. The memory architecture (for example, SMP and NUMA) contributes to the memory bandwidth in multicore computers. Performance with respect to memory is also subject to page faults.

## A cache bound task

A task is cache bound when its speed is constrained by the amount of cache available. When a task retrieves values from a small number of repeated memory locations, for example a small matrix multiplication, the values may be cached and fetched from there. Note that CPUs (typically) have multiple layers of cache, and the performance will be at its best when the processed data fits in the cache, but the processing will still happen, more slowly, when the data does not fit into the cache. It is possible to make the most of the cache using **cache-oblivious** algorithms. A higher number of concurrent cache/memory bound threads than CPU cores is likely to flush the instruction pipeline, as well as the cache at the time of

context switch, likely leading to a severely degraded performance.

## An input/output bound task

An **input/output (I/O)** bound task would go faster if the I/O subsystem, that it depends on, goes faster. Disk/storage and network are the most commonly used I/O subsystems in data processing, but it can be serial port, a USB-connected card reader, or any I/O device. An I/O bound task may consume very few CPU cycles. Depending on the speed of the device, connection pooling, data compression, asynchronous handling, application caching, and more, may help in performance. One notable aspect of I/O bound tasks is that performance is usually dependent on the time spent waiting for connection/seek, and the amount of serialization that we do, and hardly on the other resources.

In practice, many data processing workloads are usually a combination of CPU bound, memory bound, cache bound, and I/O bound tasks. The performance of such mixed workloads effectively depends on the even distribution of CPU, cache, memory, and I/O resources over the duration of the operation. A bottleneck situation arises only when one resource gets too busy to make way for another.

## Online transaction processing

**Online transaction processing (OLTP)** systems process the business transactions on demand. They can sit behind systems such as a user-facing ATM machine, point-of-sale terminal, a network-connected ticket counter, ERP systems, and more. The OLTP systems are characterized by low latency, availability, and data integrity. They run day-to-day business transactions. Any interruption or outage is likely to have a direct and immediate impact on sales or service. Such systems are expected to be designed for resiliency rather than delayed recovery from failures. When the performance objective is unspecified, you may like to consider graceful degradation as a strategy.

It is a common mistake to ask the OLTP systems to answer analytical queries, something that they are not optimized for. It is desirable for an informed programmer to know the capability of the system, and suggest design changes as per the requirements.

## Online analytical processing

**Online analytical processing (OLAP)** systems are designed to answer analytical queries in a short time. They typically get data from the OLTP operations, and their data model is optimized for querying. They basically provide for consolidation (roll-up), drill-down and slicing and dicing of data for analytical purposes. They often use specialized data stores that can optimize ad-hoc analytical queries on the fly. It is important for such databases to provide pivot-table like capability. Often, the OLAP cube is used to get fast access to the analytical data.

Feeding the OLTP data into the OLAP systems may entail workflows and multistage batch processing. The performance concern of such systems is to efficiently deal with large quantities of data while also dealing with inevitable failures and recovery.

## Batch processing

**Batch processing** is automated execution of predefined jobs. These are typically bulk jobs that are executed during off-peak hours. Batch processing may involve one or more stages of job processing. Often batch processing is clubbed with workflow automation, where some workflow steps are executed offline. Many of the batch processing jobs work on staging of data, and on preparing data for the next stage of processing to pick up.

Batch jobs are generally optimized for the best utilization of the computing resources. Since there is little to moderate the demand to lower the latencies of some particular subtasks, these systems tend to optimize for throughput. A lot of batch jobs involve largely I/O processing and are often

distributed over a cluster. Due to distribution, the data locality is preferred when processing the jobs; that is, the data and processing should be local in order to avoid network latency in reading/writing data.

# A structured approach to the performance

In practice, the performance of non-trivial applications is rarely a function of coincidence or prediction. For many projects, performance is not an option (it is rather a necessity), which is why this is even more important today. Capacity planning, determining performance objectives, performance modeling, measurement, and monitoring are key.

Tuning a poorly designed system to perform is significantly harder, if not practically impossible, than having a system well-designed from the start. In order to meet a performance goal, performance objectives should be known before the application is designed. The performance objectives are stated in terms of latency, throughput, resource utilization, and workload. These terms are discussed in the following section in this chapter.

The resource cost can be identified in terms of application scenarios, such as browsing of products, adding products to shopping cart, checkout, and more. Creating workload profiles that represent users performing various operations is usually helpful.

**Performance modeling** is a reality check for whether the application design will support the performance objectives. It includes performance objectives, application scenarios, constraints, measurements (benchmark results), workload objectives and if available, the performance baseline. It is not a replacement for measurement and load testing, rather, the model is validated using these. The performance model may include the performance test cases to assert the performance characteristics of the application scenarios.

Deploying an application to production almost always needs some form of **capacity planning**. It has to take into account the performance objectives

for today and for the foreseeable future. It requires an idea of the application architecture, and an understanding of how the external factors translate into the internal workload. It also requires informed expectations about the responsiveness and the level of service to be provided by the system. Often, capacity planning is done early in a project to mitigate the risk of provisioning delays.

# The performance vocabulary

There are several technical terms that are heavily used in performance engineering. It is important to understand these, as they form the cornerstone of the performance-related discussions. Collectively, these terms form a performance vocabulary. The performance is usually measured in terms of several parameters, where every parameter has roles to play—such parameters are a part of the vocabulary.

## Latency

**Latency** is the time taken by an individual unit of work to complete the task. It does not imply successful completion of a task. Latency is not collective, it is linked to a particular task. If two similar jobs— $j_1$  and  $j_2$  took 3 ms and 5 ms respectively, their latencies would be treated as such. If  $j_1$  and  $j_2$  were dissimilar tasks, it would have made no difference. In many cases the average latency of similar jobs is used in the performance objectives, measurement, and monitoring results.

Latency is an important indicator of the health of a system. A high performance system often thrives on low latency. Higher than normal latency can be caused due to load or bottleneck. It helps to measure the latency distribution during a load test. For example, if more than 25 percent of similar jobs, under a similar load, have significantly higher latency than others, then it may be an indicator of a bottleneck scenario that is worth investigating.

When a task called  $j_1$  consists of smaller tasks called  $j_2$ ,  $j_3$ , and  $j_4$ , the latency of  $j_1$  is not necessarily the sum of the latencies of each of  $j_2$ ,  $j_3$ , and  $j_4$ . If any of the subtasks of  $j_1$  are concurrent with another, the latency of  $j_1$  will turn out to be less than the sum of the latencies of  $j_2$ ,  $j_3$ , and  $j_4$ . The I/O bound tasks are generally more prone to higher latency. In network systems, latency is commonly based on the round-trip to another

host, including the latency from source to destination, and then back to source.

## Throughput

**Throughput** is the number of successful tasks or operations performed in a unit of time. The top-level operations performed in a unit of time are usually of a similar kind, but with a potentially different from latencies. So, what does throughput tell us about the system? It is the rate at which the system is performing. When you perform load testing, you can determine the maximum rate at which a particular system can perform. However, this is not a guarantee of the conclusive, overall, and maximum rate of performance.

Throughput is one of the factors that determine the scalability of a system. The throughput of a higher level task depends on the capacity to spawn multiple such tasks in parallel, and also on the average latency of those tasks. The throughput should be measured during load testing and performance monitoring to determine the peak-measured throughput, and the maximum-sustained throughput. These factors contribute to the scale and performance of a system.

## Bandwidth

**Bandwidth** is the raw data rate over a communication channel, measured in a certain number of bits per second. This includes not only the payload, but also all the overhead necessary to carry out the communication. Some examples are: Kbits/sec, Mbits/sec, and more. An uppercase B such as KB/sec denotes Bytes, as in kilobytes per second. Bandwidth is often compared to throughput. While bandwidth is the raw capacity, throughput for the same system is the successful task completion rate, which usually involves a round-trip. Note that throughput is for an operation that involves latency. To achieve maximum throughput for a given bandwidth, the communication/protocol overhead and operational latency should be

minimal.

For storage systems (such as hard disks, solid-state drives, and more) the predominant way to measure performance is **IOPS (Input-output per second)**, which is multiplied by the transfer size and represented as bytes per second, or further into MB/sec, GB/sec, and more. IOPS is usually derived for sequential and random workloads for read/write operations.

Mapping the throughput of a system to the bandwidth of another may lead to dealing with an impedance mismatch between the two. For example, an order processing system may perform the following tasks:

- Transact with the database on disk
- Post results over the network to an external system

Depending on the bandwidth of the disk sub-system, the bandwidth of the network, and the execution model of order processing, the throughput may depend not only on the bandwidth of the disk sub-system and network, but also on how loaded they currently are. Parallelism and pipelining are common ways to increase the throughput over a given bandwidth.

## Baseline and benchmark

The performance **baseline**, or simply baseline, is the reference point, including measurements of well-characterized and understood performance parameters for a known configuration. The baseline is used to collect performance measurements for the same parameters that we may benchmark later for another configuration. For example, collecting "throughput distribution over 10 minutes at a load of 50 concurrent threads" is one such performance parameter that we can use for baseline and benchmarking. A baseline is recorded together with the hardware, network, OS and JVM configuration.

The performance **benchmark**, or simply benchmark, is the recording of the performance parameter measurements under various test conditions. A

benchmark can be composed of a performance test suite. A benchmark may collect small to large amounts of data, and may take varying durations depending on the use-cases, scenarios, and environment characteristics.

A baseline is a result of the benchmark that was conducted at one point in time. However, a benchmark is independent of the baseline.

## Profiling

**Performance profiling**, or simply profiling, is the analysis of the execution of a program at its runtime. A program can perform poorly for a variety of reasons. A **profiler** can analyze and find out the execution time of various parts of the program. It is possible to put statements in a program manually to print the execution time of the blocks of code, but it gets very cumbersome as you try to refine the code iteratively.

A profiler is of great assistance to the developer. Going by how profilers work, there are three major kinds—instrumenting, sampling, and event-based.

- **Event-based profilers:** These profilers work only for selected language platforms, and provide a good balance between the overhead and results; Java supports event-based profiling via the JVMTI interface.
- **The instrumenting profilers:** These profilers modify code at either compile time, or runtime to inject performance counters. They are intrusive by nature and add significant performance overhead. However, you can profile the regions of code very selectively using the instrumenting profilers.
- **The sampling profilers:** These profilers pause the runtime and collect its state at "sampling intervals". By collecting enough samples, they get to know where the program is spending most of its time. For example, at a sampling interval of 1 millisecond, the profiler would have collected 1000 samples in a second. A sampling profiler also works for code that executes faster than the sampling interval (as in,

the code may perform several iterations of work between the two sampling events), as the frequency of pausing and sampling is proportional to the overall execution time of any code.

Profiling is not meant only for measuring execution time. Capable profilers can provide a view of memory analysis, garbage collection, threads, and more. A combination of such tools is helpful to find memory leaks, garbage collection issues, and so on.

## Performance optimization

Simply put, **optimization** is enhancing a program's resource consumption after a performance analysis. The symptoms of a poorly performing program are observed in terms of high latency, low throughput, unresponsiveness, instability, high memory consumption, high CPU consumption, and more. During the performance analysis, one may profile the program in order to identify the bottlenecks and tune the performance incrementally by observing the performance parameters.

Better and suitable algorithms are an all-around good way to optimize code. The CPU bound code can be optimized with computationally cheaper operations. The cache bound code can try using less memory lookups to keep a good hit ratio. The memory bound code can use an adaptive memory usage and conservative data representation to store in memory for optimization. The I/O bound code can attempt to serialize as little data as possible, and batching of operations will make the operation less chatty for better performance. Parallelism and distribution are other, overall good ways to increase performance.

## Concurrency and parallelism

Most of the computer hardware and operating systems that we use today provide concurrency. On the x86 architecture, hardware support for concurrency can be traced as far back as the 80286 chip. **Concurrency** is the simultaneous execution of more than one process on the same

computer. In older processors, concurrency was implemented using the context switch by the operating system kernel. When concurrent parts are executed in parallel by the hardware instead of merely the switching context, it is called **parallelism**. Parallelism is the property of the hardware, though the software stack must support it in order for you to leverage it in your programs. We must write your program in a concurrent way to exploit the parallelism features of the hardware.

While concurrency is a natural way to exploit hardware parallelism and speed up operations, it is worth bearing in mind that having significantly higher concurrency than the parallelism that your hardware can support is likely to schedule tasks to varying processor cores thereby, lowering the branch prediction and increasing cache misses.

At a low level, spawning the processes/threads, mutexes, semaphores, locking, shared memory, and interprocess communication are used for concurrency. The JVM has an excellent support for these concurrency primitives and interthread communication. Clojure has both—the low and higher level concurrency primitives that we will discuss in the concurrency chapter.

## Resource utilization

**Resource utilization** is the measure of the server, network, and storage resources that is consumed by an application. Resources include CPU, memory, disk I/O, network I/O, and more. The application can be analyzed in terms of CPU bound, memory bound, cache bound, and I/O bound tasks. Resource utilization can be derived by means of benchmarking, by measuring the utilization at a given throughput.

## Workload

**Workload** is the quantification of how much work is there in hand to be carried out by the application. It is measured in the total numbers of users,

the concurrent active users, the transaction volume, the data volume, and more. Processing a workload should take in to account the load conditions, such as how much data the database currently holds, how filled up the message queues are, the backlog of I/O tasks after which the new load will be processed, and more.

# The latency numbers that every programmer should know

Hardware and software have progressed over the years. Latencies for various operations put things in perspective. The latency numbers for the year 2015, reproduced with the permission of Aurojit Panda and Colin Scott of Berkeley University

([http://www.eecs.berkeley.edu/~rcs/research/interactive\\_latency.html](http://www.eecs.berkeley.edu/~rcs/research/interactive_latency.html)).

Latency numbers that every programmer should know are as shown in the following table:

Operation	Time taken as of 2015
L1 cache reference	1ns (nano second)
Branch mispredict	3 ns
L2 cache reference	4 ns
Mutex lock/unlock	17 ns
Compress 1KB with Zippy  (Zippy/Snappy: <a href="http://code.google.com/p/snappy/">http://code.google.com/p/snappy/</a> )	2 $\mu$ s (1000 ns = 1 $\mu$ s: micro second)
Send 2000 bytes over the commodity network	200ns (that is, 0.2 $\mu$ s)
SSD random read	16 $\mu$ s
Round-trip in the same datacenter	500 $\mu$ s
Read 1,000,000 bytes sequentially from SSD	200 $\mu$ s
Disk seek	4 ms (1000 $\mu$ s = 1 ms)

Read 1,000,000 bytes sequentially from disk	2 ms
Packet roundtrip CA to Netherlands	150 ms

The preceding table shows the operations in a computer vis-a-vis the latency incurred due to the operation. When a CPU core processes some data in a CPU register, it may take a few CPU cycles (for reference, a 3 GHz CPU runs 3000 cycles per nanosecond), but the moment it has to fall back on L1 or L2 cache, the latency becomes thousands of times slower. The preceding table does not show main memory access latency, which is roughly 100 ns (it varies, based on the access pattern)—about 25 times slower than the L2 cache.

# Summary

We learned about the basics of what it is like to think more deeply about performance. We saw the common performance vocabulary, and also the use cases by which performance aspects might vary. We concluded by looking at the performance numbers for the different hardware components, which is how performance benefits reach our applications. In the next chapter, we will dive into the performance aspects of the various Clojure abstractions.

# Chapter 2. Clojure Abstractions

Clojure has four founding ideas. Firstly, it was set up to be a functional language. It is not pure (as in purely functional), but emphasizes immutability. Secondly, it is a dialect of Lisp; Clojure is malleable enough that users can extend the language without waiting for the language implementers to add new features and constructs. Thirdly, it was built to leverage concurrency for the new generation challenges. Lastly, it was designed to be a hosted language. As of today, Clojure implementations exist for the JVM, CLR, JavaScript, Python, Ruby, and Scheme. Clojure blends seamlessly with its host language.

Clojure is rich in abstractions. Though the syntax itself is very minimal, the abstractions are finely grained, mostly composable, and designed to tackle a wide variety of concerns in the least complicated way. In this chapter, we will discuss the following topics:

- Performance characteristics of non-numeric scalars
- Immutability and epochal time model paving the way for performance by isolation
- Persistent data structures and their performance characteristics
- Laziness and its impact on performance
- Transients as a high-performance, short-term escape hatch
- Other abstractions, such as tail recursion, protocols/types, multimethods, and many more

## Non-numeric scalars and interning

Strings and characters in Clojure are the same as in Java. The string literals are implicitly interned. Interning is a way of storing only the unique values in the heap and sharing the reference everywhere it is required. Depending on the JVM vendor and the version of Java you use, the interned data may be stored in a string pool, Permgen, ordinary heap, or some special area in the heap marked for interned data. Interned data is subject to garbage

collection when not in use, just like ordinary objects. Take a look at the following code:

```
user=> (identical? "foo" "foo") ; literals are automatically interned
true
user=> (identical? (String. "foo") (String. "foo")) ; created string is not interned
false
user=> (identical? (.intern (String. "foo")) (.intern (String. "foo"))))
true
user=> (identical? (str "f" "oo") (str "f" "oo")) ; str creates string
false
user=> (identical? (str "foo") (str "foo")) ; str does not create string for 1 arg
true
user=> (identical? (read-string "\"foo\"") (read-string "\"foo\""))
; not interned
false
user=> (require '[clojure.edn :as edn]) ; introduced in Clojure 1.5
nil
user=> (identical? (edn/read-string "\"foo\"") (edn/read-string "\"foo\""))
false
```

Note that `identical?` in Clojure is the same as `==` in Java. The benefit of interning a string is that there is no memory allocation overhead for duplicate strings. Commonly, applications on the JVM spend quite some time on string processing. So, it makes sense to have them interned whenever there is a chance of duplicate strings being simultaneously processed. Most of the JVM implementations today have an extremely fast `intern` operation; however, you should measure the overhead for your JVM if you have an older version.

Another benefit of string interning is that when you know that two string tokens are interned, you can compare them faster for equality using `identical?` than non-interned string tokens. The equivalence function =

first checks for identical references before conducting a content check.

Symbols in Clojure always contain interned string references within them, so generating a symbol from a given string is nearly as fast as interning a string. However, two symbols created from the same string will not be identical:

```
user=> (identical? (.intern "foo") (.intern "foo"))
true
user=> (identical? (symbol "foo") (symbol "foo"))
false
user=> (identical? (symbol (.intern "foo")) (symbol (.intern
"foo"))))
false
```

Keywords are, on the basis of their implementation, built on top of symbols and are designed to work with the `identical?` function for equivalence. So, comparing keywords for equality using `identical?` would be faster, just as with interned string tokens.

Clojure is increasingly being used for large-volume data processing, which includes text and composite data structures. In many cases, the data is either stored as JSON or EDN (<http://edn-format.org>). When processing such data, you can save memory by interning strings or using symbols/keywords. Remember that string tokens read from such data would not be automatically interned, whereas the symbols and keywords read from EDN data would invariably be interned. You may come across such situations when dealing with relational or NoSQL databases, web services, CSV or XML files, log parsing, and so on.

Interning is linked to the JVM **Garbage Collection (GC)**, which, in turn, is closely linked to performance. When you do not intern the string data and let duplicates exist, they end up being allocated on the heap. More heap usage leads to GC overhead. Interning a string has a tiny but measurable and upfront performance overhead, whereas GC is often unpredictable and unclear. GC performance, in most JVM

implementations, has not increased in a similar proportion to the performance advances in hardware. So, often, effective performance depends on preventing GC from becoming the bottleneck, which in most cases means minimizing it.

# Identity, value, and epochal time model

One of the principal virtues of Clojure is its simple design that results in malleable, beautiful composability. Using symbols in place of pointers is a programming practice that has existed for several decades now. It has found widespread adoption in several imperative languages. Clojure dissects that notion in order to uncover the core concerns that need to be addressed. The following subsections illustrate this aspect of Clojure.

We program using logical entities to represent values. For example, a value of `30` means nothing unless it is associated with a logical entity, let's say `age`. The logical entity `age` is the identity here. Now, even though `age` represents a value, the value may change with time; this brings us to the notion of `state`, which represents the value of the identity at a certain time. Hence, `state` is a function of time and is causally related to what we do in the program. Clojure's power lies in binding an identity with its value that holds true at the time and the identity remains isolated from any new value it may represent later. We will discuss state management in [Chapter 5, Concurrency](#).

## Variables and mutation

If you have previously worked with an imperative language (C/C++, Java, and so on), you may be familiar with the concept of a variable. A **variable** is a reference to a block of memory. When we update its value, we essentially update the place in memory where the value is stored. The variable continues to point to the place where the older version of the value was stored. So, essentially, a variable is an alias for the place of storage of values.

A little analysis would reveal that variables are strongly linked to the

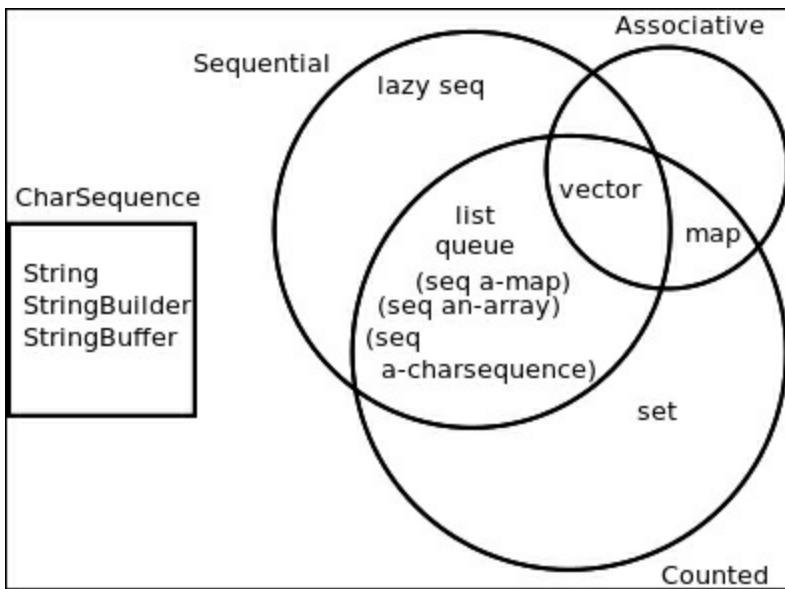
processes that read or mutate their values. Every mutation is a state transition. The processes that read/update the variable should be aware of the possible states of the variable to make sense of the state. Can you see a problem here? It conflates identity and state! It is impossible to refer to a value or a state in time when dealing with a variable—the value could change at any time unless you have complete control over the process accessing it. The mutability model does not accommodate the concept of time that causes its state transition.

The issues with mutability do not stop here. When you have a composite data structure containing mutable variables, the entire data structure becomes mutable. How can we mutate it without potentially undermining the other processes that might be observing it? How can we share this data structure with concurrent processes? How can we use this data structure as a key in a hash-map? This data structure does not convey anything. Its meaning could change with mutation! How do we send such a thing to another process without also compensating for the time, which can mutate it in different ways?

Immutability is an important tenet of functional programming. It not only simplifies the programming model, but also paves the way for safety and concurrency. Clojure supports immutability throughout the language. Clojure also supports fast, mutation-oriented data structures as well as thread-safe state management via concurrency primitives. We will discuss these topics in the forthcoming sections and chapters.

## Collection types

There are a few types of collections in Clojure, which are categorized based on their properties. The following Venn diagram depicts this categorization on the basis of whether the collections are counted (so that `counted?` returns `true`) or associative (so that `associative?` returns `true`) or sequential (so that `sequential?` returns `true`):



The previous diagram illustrates the characteristics that different kinds of data structures share. The sequential structures let us iterate over the items in the collection, the item count of counted structures can be found constant with respect to time, and associative structures can be looked at with keys for corresponding values. The **CharSequence** box shows the character sequence Java types that can be converted to a Clojure sequence using `(seq charseq)`.

# Persistent data structures

As we've noticed in the previous section, Clojure's data structures are not only immutable, but can produce new values without impacting the old version. Operations produce these new values in such a way that old values remain accessible; the new version is produced in compliance with the complexity guarantees of that data structure, and both the old and new versions continue to meet the complexity guarantees. The operations can be recursively applied and can still meet the complexity guarantees. Such immutable data structures as the ones provided by Clojure are called **persistent data structures**. They are "persistent", as in, when a new version is created, both the old and new versions "persist" in terms of both the value and complexity guarantee. They have nothing to do with storage or durability of data. Making changes to the old version doesn't impede working with the new version and vice versa. Both versions persist in a similar way.

Among the publications that have inspired the implementation of Clojure's persistent data structures, two of them are well known. Chris Okasaki's *Purely Functional Data Structures* has influenced the implementation of persistent data structures and lazy sequences/operations. Clojure's persistent queue implementation is adapted from Okasaki's *Batched Queues*. Phil Bagwell's *Ideal Hash Tries*, though meant for mutable and imperative data structures, was adapted to implement Clojure's persistent map/vector/set.

## Constructing lesser-used data structures

Clojure supports a well-known literal syntax for lists, vectors, sets, and maps. Shown in the following list are some less-used methods for creating other data structures:

- Map (`PersistentArrayMap` and `PersistentHashMap`):

```
{:a 10 :b 20} ; array-map up to 8 pairs
{:a 1 :b 2 :c 3 :d 4 :e 5 :f 6 :g 7 :h 8 :i 9} ; hash-map
for 9 or more pairs
```

- **Sorted map** (`PersistentTreeMap`):

```
(sorted-map :a 10 :b 20 :c 30) ; (keys ...) should return
sorted
```

- **Sorted set** (`PersistentTreeSet`):

```
(sorted-set :a :b :c)
```

- **Queue** (`PersistentQueue`):

```
(import 'clojure.lang.PersistentQueue)
(reduce conj PersistentQueue/EMPTY [:a :b :c :d]) ; add to
queue
(peek queue) ; read from queue
(pop queue) ; remove from queue
```

As you can see, abstractions such as `TreeMap` (sorted by key), `TreeSet` (sorted by element), and `Queue` should be instantiated by calling their respective APIs.

## Complexity guarantee

The following table gives a summary of the complexity guarantees (using the Big-O notation) of various kinds of persistent data structures in Clojure:

<b>Operation</b>	<b>PersistentList</b>	<b>PersistentHashMap</b>	<b>PersistentArrayMap</b>	<b>PersistentVector</b>	<b>PersistentQueue</b>
count	O(1)	O(1)	O(1)	O(1)	O(1)
conj	O(1)			O(1)	O(1)
first	O(1)			O(<7)	O(<7)
rest	O(1)			O(<7)	O(<7)

doseq	O(n)	O(n)	O(n)	O(n)	O(n)
nth	O(n)			O(<7)	O(<7)
last	O(n)			O(n)	O(n)
get		O(<7)	O(1)	O(<7)	O(<7)
assoc		O(<7)	O(1)	O(<7)	
dissoc		O(<7)	O(1)	O(<7)	
peek				O(1)	O(1)
pop				O(<7)	O(1)

A **list** is a sequential data structure. It provides constant time access for count and for anything regarding the first element only. For example, `conj` adds the element to the head and guarantees  $O(1)$  complexity. Similarly, `first` and `rest` provide  $O(1)$  guarantees too. Everything else provides an  $O(n)$  complexity guarantee.

Persistent hash-maps and vectors use the trie data structure with a branching factor of 32 under the hood. So, even though the complexity is  $O(\log_{32} n)$ , only  $2^{32}$  hash codes can fit into the trie nodes. Hence,  $\log_{32} 2^{32}$ , which turns out to be 6.4 and is less than 7, is the worst-case complexity and can be considered near-constant time. As the trie grows larger, the portion to copy gets proportionately tiny due to structure sharing. Persistent hash-set implementation is also based on hash-map; hence, the hash-sets share the characteristics of the hash-maps. In a persistent vector, the last incomplete node is placed at the tail, which is always directly accessible from the root. This makes using `conj` to the end a constant time operation.

Persistent tree-maps and tree-sets are basically sorted maps and sets respectively. Their implementation uses red-black trees and is generally more expensive than hash-maps and hash-sets. A persistent queue uses a persistent vector under the hood for adding new elements. Removing an element from a persistent queue takes the head off `seq`, which is created from the vector where new elements are added.

The complexity of an algorithm over a data structure is not an absolute measure of its performance. For example, working with hash-maps involves computing the hashCode, which is not included in the complexity guarantee. Our choice of data structures should be based on the actual use case. For example, when should we use a list instead of a vector? Probably when we need sequential or **last-in-first-out (LIFO)** access, or when constructing an **abstract-syntax-tree (AST)** for a function call.

## O(<7) implies near constant time

You may know that the **Big-O** notation is used to express the upper bound (worst case) of the efficiency of any algorithm. The variable  $n$  is used to express the number of elements in the algorithm. For example, a binary search on a sorted associative collection, such as a sorted vector, is a logarithmic time, that is an  $O(\log_2 n)$  or simply an  $O(\log n)$  algorithm. Since there can be a maximum of  $2^{32}$  (technically  $2^{31}$  due to a signed positive integer) elements in a Java collection and  $\log_2 2^{32}$  is 32, the binary search can be  $O(\leq 32)$  in the worst case. Similarly, though operations on persistent collections are  $O(\log_{32} n)$ , in the worst case they actually turn out to be  $O(\log_{32} 2^{32})$  at maximum, which is  $O(<7)$ . Note that this is much lower than logarithmic time and approaches near constant time. This implies not so bad performance for persistent collections even in the worst possible scenario.

## The concatenation of persistent data structures

While persistent data structures have excellent performance

characteristics, the concatenation of two persistent data structures has been a linear time  $O(N)$  operation, except for some recent developments. The `concat` function, as of Clojure 1.7, still provides linear time concatenation. Experimental work on **Relaxed Radix Balanced (RRB)** trees is going on in the **core.rrb-vector** contrib project (<https://github.com/clojure/core.rrb-vector>), which may provide logarithmic time  $O(\log N)$  concatenation. Readers interested in the details should refer to the following links:

- The RRB-trees paper at <http://infoscience.epfl.ch/record/169879/files/RMTrees.pdf>
- Phil Bagwell's talk at <http://www.youtube.com/watch?v=K2NYwP90bNs>
- Tiark Rompf's talk at <http://skillsmatter.com/podcast-scala/fast-concatenation-immutable-vectors>

# Sequences and laziness

*"A seq is like a logical cursor."*

--Rich Hickey

**Sequences** (commonly known as **seqs**) are a way to sequentially consume a succession of data. As with iterators, they let a user begin consuming elements from the head and proceed realizing one element after another. However, unlike iterators, sequences are immutable. Also, since sequences are only a view of the underlying data, they do not modify the storage structure of the data.

What makes sequences stand apart is they are not data structures per se; rather, they are a data abstraction over a stream of data. The data may be produced by an algorithm or a data source connected to an I/O operation. For example, the `resultset-seq` function accepts a `java.sql.ResultSet` JDBC instance as an argument and produces lazily realized rows of data as `seq`.

Clojure data structures can be turned into sequences using the `seq` function. For example, `(seq [:a :b :c :d])` returns a sequence. Calling `seq` over an empty collection returns `nil`.

Sequences can be consumed by the following functions:

- `first`: This returns the head of the sequence
- `rest`: This returns the remaining sequence, even if it's empty, after removing the head
- `next`: This returns the remaining sequence or `nil`, if it's empty, after removing the head

## Laziness

Clojure is a strict (as in, the opposite of "lazy") language, which can

choose to explicitly make use of laziness when required. Anybody can create a lazily evaluated sequence using the `lazy-seq` macro. Some Clojure operations over collections, such as `map`, `filter`, and more are intentionally lazy.

**Laziness** simply means that the value is not computed until actually required. Once the value is computed, it is cached so that any future reference to the value need not re-compute it. The caching of the value is called **memoization**. Laziness and memoization often go hand in hand.

## Laziness in data structure operations

Laziness and memoization together form an extremely useful combination to keep the single-threaded performance of functional algorithms comparable to its imperative counterparts. For an example, consider the following Java code:

```
List<String> titles = getTitles();
int goodCount = 0;
for (String each: titles) {
    String checksum = computeChecksum(each);
    if (verifyOK(checksum)) {
        goodCount++;
    }
}
```

As is clear from the preceding snippet, it has a linear time complexity, that is,  $O(n)$ , and the whole operation is performed in a single pass. The comparable Clojure code is as follows:

```
(->> (get-titles)
      (map compute-checksum)
      (filter verify-ok?))
      count)
```

Now, since we know `map` and `filter` are lazy, we can deduce that the Clojure version also has linear time complexity, that is,  $O(n)$ , and finishes the task in one pass with no significant memory overhead. Imagine, for a

moment, that `map` and `filter` are not lazy—what would be the complexity then? How many passes would it make? It's not just that `map` and `filter` would both have taken one pass, that is,  $O(n)$ , each; they would each have taken as much memory as the original collection in the worst case, due to storing the intermediate results.

It is important to know the value of laziness and memoization in an immutability-emphasizing functional language such as Clojure. They form a basis for **amortization** in persistent data structures, which is about focusing on the overall performance of a composite operation instead of microanalyzing the performance of each operation in it; the operations are tuned to perform faster in those operations that matter the most.

Another important bit of detail is that when a lazy sequence is realized, the data is memoized and stored. On the JVM, all the heap references that are reachable in some way are not garbage collected. So, as a consequence, the entire data structure is kept in the memory unless you lose the head of the sequence. When working with lazy sequences using local bindings, make sure you don't keep referring to the lazy sequence from any of the locals. When writing functions that may accept lazy sequence(s), take care that any reference to the lazy `seq` does not outlive the execution of the function in the form of a closure or some such.

## Constructing lazy sequences

Now that we know what lazy sequences are, let's try to create a retry counter that should return true only as many times as the retry can be performed. This is shown in the following code:

```
(defn retry? [n]
  (if (<= n 0)
    (cons false (lazy-seq (retry? 0)))
    (cons true (lazy-seq (retry? (dec n))))))
```

The `lazy-seq` macro makes sure that the stack is not used for recursion. We can see that this function would return endless values. Hence, in order

to inspect what it returns, we should limit the number of elements as shown in the following code:

```
user=> (take 7 (retry? 5))
(true true true true true false false)
```

Now, let's try using it in a mock fashion:

```
(loop [r (retry? 5)]
  (if-not (first r)
    (println "No more retries")
    (do
      (println "Retrying")
      (recur (rest r)))))
```

As expected, the output should print `Retrying` five times before printing `No more retries` and exiting as follows:

```
Retrying
Retrying
Retrying
Retrying
Retrying
Retrying
No more retries
nil
```

Let's take another simpler example of constructing a lazy sequence, which gives us a countdown from a specified number to zero:

```
(defn count-down [n]
  (if (<= n 0)
    '()
    (cons n (lazy-seq (count-down (dec n))))))
```

We can inspect the values it returns as follows:

```
user=> (count-down 8)
(8 7 6 5 4 3 2 1 0)
```

Lazy sequences can loop indefinitely without exhausting the stack and can come in handy when working with other lazy operations. To maintain a

balance between space-saving and performance, consuming lazy sequences results in the chunking of elements by a factor of 32. That means lazy seqs are realized in a chunk-size of 32, even though they are consumed sequentially.

## Custom chunking

The default chunk size 32 may not be optimum for all lazy sequences—you can override the chunking behavior when you need to. Consider the following snippet (adapted from Kevin Downey's public gist at <https://gist.github.com/hiredman/324145>):

```
(defn chunked-line-seq
  "Returns the lines of text from rdr as a chunked[size] sequence
  of strings.
  rdr must implement java.io.BufferedReader."
  [^java.io.BufferedReader rdr size]
  (lazy-seq
    (when-let [line (.readLine rdr)]
      (chunk-cons
        (let [buffer (chunk-buffer size)]
          (chunk-append buffer line)
          (dotimes [i (dec size)]
            (when-let [line (.readLine rdr)]
              (chunk-append buffer line)))
        (chunk buffer)))
    (chunked-line-seq rdr size)))))
```

As per the previous snippet, the user is allowed to pass a chunk size that is used to produce the lazy sequence. A larger chunk size may be useful when processing large text files, such as when processing CSV or log files. You would notice the following four less-known functions used in the snippet:

- clojure.core/chunk-cons
- clojure.core/chunk-buffer
- clojure.core/chunk-append
- clojure.core/chunk

While `chunk-cons` is the equivalent of `clojure.core/cons` for chunked sequences, `chunk-buffer` creates a mutable chunk buffer (controls the chunk size), `chunk-append` appends an item to the end of a mutable chunk buffer, and `chunk` turns a mutable chunk buffer into an immutable chunk.

The `clojure.core` namespace has several functions related to chunked sequences listed as follows:

- `chunk`
- `chunk-rest`
- `chunk-cons`
- `chunk-next`
- `chunk-first`
- `chunk-append`
- `chunked-seq?`
- `chunk-buffer`

These functions are not documented, so although I would encourage you to study their source code to understand what they do, I would advise you not to make any assumptions about their support in future Clojure versions.

## Macros and closures

Often, we define a macro so as to turn the parameter body of code into a closure and delegate it to a function. See the following example:

```
(defmacro do-something
  [& body]
  `(do-something* (fn [] ~@body)))
```

When using such code, if the body binds a local to a lazy sequence it may be retained longer than necessary, likely with bad consequences on memory consumption and performance. Fortunately, this can be easily fixed:

```
(defmacro do-something
```

```
[& body]
` (do-something* (^:once fn* [] ~@body))
```

Notice the `^:once` hint and the `fn*` macro, which make the Clojure compiler clear the closed-over references, thus avoiding the problem. Let's see this in action (Alan Malloy's example from

[https://groups.google.com/d/msg/clojure/Ys3kEz5c\\_eE/3St2AbIc3zMJ](https://groups.google.com/d/msg/clojure/Ys3kEz5c_eE/3St2AbIc3zMJ)):

```
user> (let [x (for [n (range)] (make-array Object 10000))
  f (^:once fn* [] (nth x 1e6))] ; using ^:once
  (f))
#<Object[] [Ljava.lang.Object;@402d3105>
user> (let [x (for [n (range)] (make-array Object 10000))
  f (fn* [] (nth x 1e6))] ; not using ^:once
  (f))
OutOfMemoryError GC overhead limit exceeded
```

The manifestation of the previous condition depends on the available heap space. This issue is tricky to detect as it only raises `OutOfMemoryError`, which is easy to misunderstand as a heap space issue instead of a memory leak. As a preventive measure, I would suggest using `^:once` with `fn*` in all cases where you close over any potentially lazy sequence.

# Transducers

Clojure 1.7 introduced a new abstraction called transducers for "composable algorithmic transformations", commonly used to apply a series of transformations over collections. The idea of transducers follows from the **reducing function**, which accepts arguments of the form (`result, input`) and returns `result`. A reducing function is what we typically use with `reduce`. A **transducer** accepts a reducing function, wraps/composes over its functionality to provide something extra, and returns another reducing function.

The functions in `clojure.core` that deal with collections have acquired an arity-1 variant, which returns a transducer, namely `map`, `cat`, `mapcat`, `filter`, `remove`, `take`, `take-while`, `take-nth`, `drop`, `drop-while`, `replace`, `partition-by`, `partition-all`, `keep`, `keep-indexed`, `dedupe` and `random-sample`.

Consider the following few examples, all of which do the same thing:

```
user=> (reduce ((filter odd?) +) [1 2 3 4 5])
9
user=> (transduce (filter odd?) + [1 2 3 4 5])
9
user=> (defn filter-odd? [xf]
           (fn
             ([] (xf))
             ([result] (xf result)))
             ([result input] (if (odd? input)
                               (xf result input)
                               result))))
# 'user/filter-odd?
user=> (reduce (filter-odd? +) [1 2 3 4 5])
9
```

Here, `(filter odd?)` returns a transducer—in the first example the transducer wraps over the reducer function `+` to return another combined reducing function. While we use the ordinary `reduce` function in the first

example, in the second example we use the `transduce` function that accepts a transducer as an argument. In the third example, we write a transducer `filter-odd?`, which emulates what (`filter odd?`) does. Let's see how the performance varies between traditional and transducer versions:

```
;; traditional way
user=> (time (dotimes [_ 10000] (reduce + (filter odd? (range 10000)))))
"Elapsed time: 2746.782033 msecs"
nil
;; using transducer
(def fodd? (filter odd?))
user=> (time (dotimes [_ 10000] (transduce fodd? + (range 10000))))
"Elapsed time: 1998.566463 msecs"
nil
```

## Performance characteristics

The key point behind transducers is how orthogonal each transformation is allowed to be, yet highly composable also. At the same time, transformations can happen in lockstep for the entire sequence instead of each operation producing lazy chunked sequences. This often causes significant performance benefits with transducers. Lazy sequences are still going to be useful when the final result is too large to realize at once—for other use cases transducers should fit the need aptly with improved performance. Since the core functions have been overhauled to work with transducers, it makes sense to model transformations more often than not in terms of transducers.

# Transients

Earlier in this chapter, we discussed the virtues of immutability and the pitfalls of mutability. However, even though mutability is fundamentally unsafe, it also has very good single-threaded performance. Now, what if there was a way to restrict the mutable operation in a local context in order to provide safety guarantees? That would be equivalent to combining the performance advantage and local safety guarantees. That is exactly the abstraction called **transients**, which is provided by Clojure.

Firstly, let's verify that it is safe (up to Clojure 1.6 only):

```
user=> (let [t (transient [:a])]
  @(future (conj! t :b)))
IllegalAccessError Transient used by non-owner thread
clojure.lang.PersistentVector$TransientVector.ensureEditable
(PersistentVector.java:463)
```

As we can see previously, up to Clojure 1.6, a transient created in one thread cannot be accessed by another. However, this operation is allowed in Clojure 1.7 in order for transducers to play well with the `core.async` (<https://github.com/clojure/core.async>) library —the developer should maintain operational consistency on transients across threads:

```
user=> (let [t (transient [:a])] (seq t))
IllegalArgumentException Don't know how to create ISeq from:
clojure.lang.PersistentVector$TransientVector
clojure.lang.RT.seqFrom (RT.java:505)
```

So, transients cannot be converted to seqs. Hence, they cannot participate in the birthing of new persistent data structures and leak out of the scope of execution. Consider the following code:

```
(let [t (transient [])]
  (conj! t :a)
  (persistent! t))
```

```
(conj! t :b))
IllegalAccessError Transient used after persistent! call
clojure.lang.PersistentVector$TransientVector.ensureEditable
(PersistentVector.java:464)
```

The `persistent!` function permanently converts `transient` into an equivalent persistent data structure. Effectively, transients are for one-time use only.

Conversion between `persistent` and `transient` data structures (the `transient` and `persistent!` functions) is constant time, that is, it is an  $O(1)$  operation. Transients can be created from unsorted maps, vectors, and sets only. The functions that mutate transients are: `conj!`, `disj!`, `pop!`, `assoc!`, and `dissoc!`. Read-only operations such as `get`, `nth`, `count`, and many more work as usual on transients, but functions such as `contains?` and those that imply seqs, such as `first`, `rest`, and `next`, do not.

## Fast repetition

The function `clojure.core/repeatedly` lets us execute a function many times and produces a lazy sequence of results. Peter Taoussanis, in his open source serialization library **Nippy** (<https://github.com/ptaoussanis/nippy>), wrote a transient-aware variant that performs significantly better. It is reproduced, as shown, with his permission (note that the arity of the function is not the same as `repeatedly`):

```
(defn repeatedly*
  "Like `repeatedly` but faster and returns given collection
  type."
  [coll n f]
  (if-not (instance? clojure.lang.IEditableCollection coll)
    (loop [v coll idx 0]
      (if (>= idx n)
        v
        (recur (conj v (f)) (inc idx))))
    (loop [v (transient coll) idx 0]
      (if (>= idx n)
```

```
(persistent! v)
(recur (conj! v (f)) (inc idx))))
```

# Performance miscellanea

Besides the major abstractions we saw earlier in the chapter, there are other smaller, but nevertheless very performance-critical, parts of Clojure that we will see in this section.

## Disabling assertions in production

Assertions are very useful to catch logical errors in the code during development, but they impose a runtime overhead that you may like to avoid in the production environment. Since `assert` is a compile time variable, the assertions can be silenced either by binding `assert` to false or by using `alter-var-root` before the code is loaded. Unfortunately, both the techniques are cumbersome to use. Paul Stadig's library called **assertions** (<https://github.com/pjstadig/assertions>) helps with this exact use-case by enabling or disabling assertions via the command-line argument `-ea` to the Java runtime.

To use it, you must include it in your Leiningen `project.clj` file as a dependency:

```
:dependencies [;; other dependencies...
               [pjstadig/assertions "0.1.0"]]
```

You must use this library's `assert` macro instead of Clojure's own, so each `ns` block in the application should look similar to this:

```
(ns example.core
  (:refer-clojure :exclude [assert])
  (:require [pjstadig.assertions :refer [assert]]))
```

When running the application, you should include the `-ea` argument to the JRE to enable assertions, whereas its exclusion implies no assertion at runtime:

```
$ JVM_OPTS=-ea lein run -m example.core  
$ java -ea -jar example.jar
```

Note that this usage will not automatically avoid assertions in the dependency libraries.

## Destructuring

**Destructuring** is one of Clojure's built-in mini languages and, arguably, a top productivity booster during development. This feature leads to the parsing of values to match the left-hand side of the binding forms. The more complicated the binding form, the more work there is that needs to be done. Not surprisingly, this has a little bit of performance overhead.

It is easy to avoid this overhead by using explicit functions to unravel data in the tight loops and other performance-critical code. After all, it all boils down to making the program work less and do more.

## Recursion and tail-call optimization (TCO)

Functional languages have this concept of tail-call optimization related to recursion. So, the idea is that when a recursive call is at the tail position, it does not take up space on the stack for recursion. Clojure supports a form of user-assisted recursive call to make sure the recursive calls do not blow the stack. This is kind of an imperative looping, but is extremely fast.

When carrying out computations, it may make a lot of sense to use `loop-recur` in the tight loops instead of iterating over synthetic numbers. For example, we want to add all odd integers from zero through to 1,000,000. Let's compare the code:

```
(defn oddsum-1 [n] ; using iteration  
  (->> (range (inc n))  
         (filter odd?)  
         (reduce +)))  
(defn oddsum-2 [n] ; using loop-recur  
  (loop [i 1 s 0]
```

```
(if (> i n)
    s
    (recur (+ i 2) (+ s i)))))
```

When we run the code, we get interesting results:

```
user=> (time (oddsum-1 1000000))
"Elapsed time: 109.314908 msecs"
```

```
250000000000
```

```
user=> (time (oddsum-2 1000000))
"Elapsed time: 42.18116 msecs"
```

```
250000000000
```

The `time` macro is far from perfect as the performance-benchmarking tool, but the relative numbers indicate a trend—in the subsequent chapters, we will look at the *Criterium* library for more scientific benchmarking. Here, we use `loop-recur` not only to iterate faster, but we are also able to change the algorithm itself by iterating only about half as many times as we did in the other example.

## Premature end of iteration

When accumulating over a collection, in some cases, we may want to end it prematurely. Prior to Clojure 1.5, `loop-recur` was the only way to do it. When using `reduce`, we can do just that using the `reduced` function introduced in Clojure 1.5 as shown:

```
;; let coll be a collection of numbers
(reduce (fn ([x] x) ([x y] (if (or (zero? x) (zero? y)) (reduced
0) (* x y)))) coll)
```

Here, we multiply all the numbers in a collection and, upon finding any of the numbers as zero, immediately return the result zero instead of continuing up to the last element.

The function `reduced?` helps detect when a reduced value is returned.

Clojure 1.7 introduces the `ensure-reduced` function to box up non-reduced values as reduced.

## Multimethods versus protocols

**Multimethods** are a fantastic expressive abstraction for a polymorphic dispatch on the dispatch function's return value. The `dispatch` functions associated with a multimethod are maintained at runtime and are looked up whenever a multimethod call is invoked. While multimethods provide a lot of flexibility in determining the dispatch, the performance overhead is simply too high compared to that of protocol implementations.

Protocols (`defprotocol`) are implemented using reify, records (`defrecord`), and types (`deftype`, `extend-type`) in Clojure. This is a big discussion topic —since we are discussing the performance characteristics, it should suffice to say that protocol implementations dispatch on polymorphic types and are significantly faster than multimethods. Protocols and types are generally the implementation detail of an API, so they are usually fronted by functions.

Due to the multimethods' flexibility, they still have a place. However, in performance-critical code it is advisable to use protocols, records, and types instead.

## Inlining

It is well known that macros are expanded inline at the call site and avoid a function call. As a consequence, there is a small performance benefit. There is also a `definline` macro that lets you write a function just like a normal macro. It creates an actual function that gets inlined at the call site:

```
(def PI Math/PI)
(definline circumference [radius]
  `(* 2 PI ~radius))
```

## Note

Note that the JVM also analyzes the code it runs and does its own inlining of code at runtime. While you may choose to inline the hot functions, this technique is known to give only a modest performance boost.

When we define a `var` object, its value is looked up each time it is used. When we define a `var` object using a `:const` meta pointing to a `long` or `double` value, it is inlined from wherever it is called:

```
(def ^:const PI Math/PI)
```

This is known to give a decent performance boost when applicable. See the following example:

```
user=> (def a 10)
user=> (def ^:const b 10)
user=> (def ^:dynamic c 10)
user=> (time (dotimes [_ 100000000] (inc a)))
"Elapsed time: 1023.745014 msecs"
nil
user=> (time (dotimes [_ 100000000] (inc b)))
"Elapsed time: 226.732942 msecs"
nil
user=> (time (dotimes [_ 100000000] (inc c)))
"Elapsed time: 1094.527193 msecs"
nil
```

# Summary

Performance is one of the cornerstones of Clojure's design. Abstractions in Clojure are designed for simplicity, power, and safety, with performance firmly in mind. We saw the performance characteristics of various abstractions and also how to make decisions about abstractions depending on performance use cases.

In the next chapter, we will see how Clojure interoperates with Java and how we can extract Java's power to derive optimum performance.

# Chapter 3. Leaning on Java

Being hosted on the JVM, there are several aspects of Clojure that really help to understand about the Java language and platform. The need is not only due to interoperability with Java or understanding its implementation, but also for performance reasons. In certain cases, Clojure may not generate optimized JVM bytecode by default; in some other cases, you may want to go beyond the performance that Clojure data structures offer —you can use the Java alternatives via Clojure to get better performance. This chapter discusses those aspects of Clojure. In this chapter we will discuss:

- Inspecting Java and bytecode generated from a Clojure source
- Numerics and primitives
- Working with arrays
- Reflection and type hinting

## Inspecting the equivalent Java source for Clojure code

Inspecting the equivalent Java source for a given Clojure code provides great insight into how that might impact its performance. However, Clojure generates only Java bytecodes at runtime unless we compile a namespace out to the disk. When developing with Leiningen, only selected namespaces under the `:aot` vector in the `project.clj` file are output as the compiled `.class` files containing bytecodes. Fortunately, an easy and quick way to know the equivalent Java source for the Clojure code is to AOT-compile namespaces and then decompile the bytecodes into equivalent Java sources, using a Java bytecode decompiler.

There are several commercial and open source Java bytecode decompilers available. One of the open source decompilers we will discuss here is **JD-GUI**, which you can download from its website (<http://jd.benow.ca/#jd>

[gui](#)). Use a version suitable for your operating system.

## Creating a new project

Let's see how exactly to arrive at the equivalent Java source code from Clojure. Create a new project using Leiningen: `lein new foo`. Then edit the `src/foo/core.clj` file with a `mul` function to find out the product of two numbers:

```
(ns foo.core)

(defn mul [x y]
  (* x y))
```

## Compiling the Clojure sources into Java bytecode

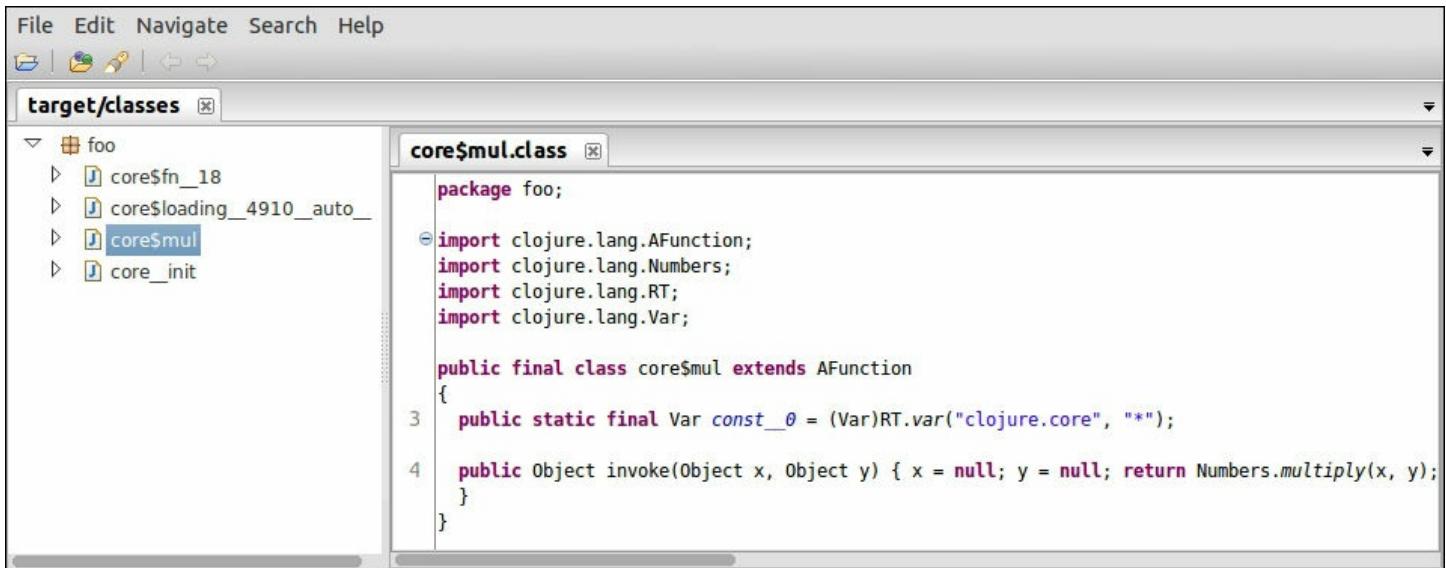
Now, to compile Clojure sources into bytecodes and output them as `.class` files, run the `lein compile :all` command. It creates the `.class` files in the `target/classes` directory of the project as follows:

```
target/classes/
`-- foo
    |-- core$fn_18.class
    |-- core_init.class
    |-- core$loading_4910_auto_.class
    `-- core$mul.class
```

You can see that the `foo.core` namespace has been compiled into four `.class` files.

## Decompiling the `.class` files into Java source

Assuming that you have already installed JD-GUI, decompiling the `.class` files is as simple as opening them using the JD-GUI application.



On inspection, the code for the `foo.core/mul` function looks as follows:

```

package foo;

import clojure.lang.AFunction;
import clojure.lang.Numbers;
import clojure.lang.RT;
import clojure.lang.Var;

public final class core$mul extends AFunction
{
    public static final Var const__0 = (Var)RT.var("clojure.core",
"**");

    public Object invoke(Object x, Object y) { x = null; y = null;
return Numbers.multiply(x, y);
}

```

It is easy to understand from the decompiled Java source that the `foo.core/mul` function is an instance of the `core$mul` class in the `foo` package extending the `clojure.lang.AFunction` class. We can also see that the argument types are of the `Object` type in method `invoke(Object, Object)`, which implies the numbers will be boxed. In a similar fashion, you

can decompile class files of any Clojure code to inspect the equivalent Java code. If you can combine this with knowledge about Java types and potential reflection and boxing, you can find the suboptimal spots in code and focus on what to improve upon.

## Compiling the Clojure source without locals clearing

Note the Java code in the method `invoke` where it says `x = null; y = null;` —how is it possible that the code throws away the arguments, sets them to null, and effectively multiplies two null objects? This misleading decompilation happens due to locals clearing, a feature of the JVM bytecode implementation of Clojure, which has no equivalent in the Java language.

Starting with Clojure 1.4, the compiler supports the `:disable-locals-clearing` key in the dynamic `clojure.core/*compiler-options*` var that we cannot configure in the `project.clj` file. So, we cannot use the `lein compile` command, but we can start a **REPL** with the `lein repl` command to compile the classes:

```
user=> (binding [*compiler-options* {:disable-locals-clearing true}] (compile 'foo.core))
foo.core
```

This generates the class files in the same location as we saw earlier in this section, but without `x = null; y = null;` because locals clearing is omitted.

# Numerics, boxing, and primitives

**Numerics** are scalars. The discussion on numerics was deferred till this chapter for the sole reason that the numerics implementation in Clojure has strong Java underpinnings. Since version 1.3, Clojure has settled with 64-bit numerics as the default. Now, `long` and `double` are idiomatic and the default numeric types. Note that these are primitive Java types, not objects. Primitives in Java lead to high performance and have several optimizations associated with them at compiler and runtime levels. A local primitive is created on the stack (hence does not contribute to heap allocation and GC) and can be accessed directly without any kind of dereferencing. In Java, there also exist object equivalents of the numeric primitives, known as **boxed numerics**—these are regular objects that are allocated on the heap. The boxed numerics are also immutable objects, which mean not only does the JVM need to dereference the stored value when reading it, but also needs to create a new boxed object when a new value needs to be created.

It should be obvious that boxed numerics are slower than their primitive equivalents. The Oracle HotSpot JVM, when started with the `-server` option, aggressively inlines those functions (on frequent invocation) that contain a call to primitive operations. Clojure automatically uses **primitive numerics** at several levels. In the `let` blocks, `loop` blocks, arrays, and arithmetic operations (`+`, `-`, `*`, `/`, `inc`, `dec`, `<`, `<=`, `>`, `>=`), primitive numerics are detected and retained. The following table describes the primitive numerics with their boxed equivalents:

Primitive numeric type	Boxed equivalent
byte (1 byte)	<code>java.lang.Byte</code>
short (2 bytes)	<code>java.lang.Short</code>

int (4 bytes)	java.lang.Integer
float (4 bytes)	java.lang.Float
long (8 bytes)	java.lang.Long
double (8 bytes)	java.lang.Double

In Clojure, sometimes you may find the numerics are passed or returned as boxed objects to or from functions due to the lack of type information at runtime. Even if you have no control over such functions, you can coerce the values to be treated as primitives. The `byte`, `short`, `int`, `float`, `long`, and `double` functions create primitive equivalents from given boxed numeric values.

One of the Lisp traditions is to provide correct ([http://en.wikipedia.org/wiki/Numerical\\_tower](http://en.wikipedia.org/wiki/Numerical_tower)) arithmetic implementation. A lower type should not truncate values when overflow or underflow happens, but rather should be promoted to construct a higher type to maintain correctness. Clojure follows this constraint and provides **autopromotion** via prime ([http://en.wikipedia.org/wiki/Prime\\_\(symbol\)](http://en.wikipedia.org/wiki/Prime_(symbol))) functions: `+`, `-`, `*`, `inc`', and `dec`'. Autopromotion provides correctness at the cost of some performance.

There are also arbitrary length or precision numeric types in Clojure that let us store unbounded numbers but have poorer performance compared to primitives. The `bignum` and `bigdec` functions let us create numbers of arbitrary length and precision.

If we try to carry out any operations with primitive numerics that may result in a number beyond its maximum capacity, the operation maintains correctness by throwing an exception. On the other hand, when we use the prime functions, they autopromote to provide correctness. There is another set of operations called unchecked operations, which do not check for

overflow or underflow and can potentially return incorrect results.

In some cases, they may be faster than regular and prime functions. Such functions are `unchecked-add`, `unchecked-subtract`, `unchecked-multiply`, `unchecked-divide`, `unchecked-inc`, and `unchecked-dec`. We can also enable unchecked math behavior for regular arithmetic functions using the `*unchecked-math*` var; simply include the following in your source code file:

```
(set! *unchecked-math* true)
```

One of the common needs in the arithmetic is the division used to find out the quotient and remainder after a natural number division. Clojure's `/` function provides a rational number division yielding a ratio, and the `mod` function provides a true modular arithmetic division. These functions are slower than the `quot` and `rem` functions that compute the division quotient and the remainder, respectively.

# Arrays

Besides objects and primitives, Java has a special type of collection storage structure called **arrays**. Once created, arrays cannot be grown or shrunk without copying data and creating another array to hold the result. Array elements are always homogeneous in type. The array elements are similar to places where you can mutate them to hold new values. Unlike collections such as list and vector, arrays can contain primitive elements, which make them a very fast storage mechanism without GC overhead.

Arrays often form a basis for mutable data structures. For example, Java's `java.lang.ArrayList` implementation uses arrays internally. In Clojure, arrays can be used for fast numeric storage and processing, efficient algorithms, and so on. Unlike collections, arrays can have one or more dimensions. So you could layout data in an array such as a matrix or cube. Let's see Clojure's support for arrays:

Description	Example	Notes
Create array	<code>(make-array Integer 20)</code>	Array of type (boxed) integer
	<code>(make-array Integer/TYPE 20)</code>	Array of primitive type integer
	<code>(make-array Long/TYPE 20 10)</code>	Two-dimensional array of primitive long
Create array of primitives	<code>(int-array 20)</code>	Array of primitive integer of size 20
	<code>(int-array [10 20 30 40])</code>	Array of primitive integer created from a vector
Create array from coll	<code>(to-array [10 20 30 40])</code>	Array from sequable

	(to-array-2d [[10 20 30][40 50 60]])	Two-dimensional array from collection
Clone an array	(aclone (to-array [:a :b :c]))	
Get array element	(aget array-object 0 3)	Get element at index [0][3] in a 2-D array
Mutate array element	(aset array-object 0 3 :foo)	Set obj :foo at index [0][3] in a 2-D array
Mutate primitive array element	(aset-int int-array-object 2 6 89)	Set value 89 at index [2][6] in 2-D array
Find length of array	(alength array-object)	alength is significantly faster than count
Map over an array	(def a (int-array [10 20 30 40 50 60]))  (seq (amap a idx ret (do (println idx (seq ret)) (inc (aget a idx))))	Unlike map, amap returns a non-lazy array, which is significantly faster over array elements. Note that amap is faster only when properly type hinted. See next section for type hinting.
Reduce over an array	(def a (int-array [10 20 30 40 50 60]))  (areduce a idx ret 0 (do (println idx ret) (+ ret idx)))	Unlike reduce, areduce is significantly faster over array elements. Note that reduce is faster only when properly type hinted. See next section for type hinting.

Cast to  
primitive  
arrays

(ints int-  
array-object)

Used with type hinting (see next section)

Like `int-array` and `ints`, there are functions for other types as well:

Array construction function	Primitive-array casting function	Type hinting (does not work for vars)	Generic array type hinting
boolean-array	booleans	^booleans	^" [ Z "
byte-array	bytes	^bytes	^" [ B "
short-array	shorts	^shorts	^" [ S "
char-array	chars	^chars	^" [ C "
int-array	ints	^ints	^" [ I "
long-array	longs	^longs	^" [ J "
float-array	floats	^floats	^" [ F "
double-array	doubles	^doubles	^" [ D "
object-array	—	^objects	^" [Ljava.lang.Object"

Arrays are favored over other data structures mainly due to performance, and sometimes due to interop. Extreme care should be taken to type hint the arrays and use the appropriate functions to work with them.

# Reflection and type hints

Sometimes, as Clojure is dynamically typed, the Clojure compiler is unable to figure out the type of object to invoke a certain method. In such cases, Clojure uses **reflection**, which is considerably slower than the direct method dispatch. Clojure's solution to this is something called **type hints**. Type hints are a way to annotate arguments and objects with static types, so that the Clojure compiler can emit bytecodes for efficient dispatch.

The easiest way to know where to put type hints is to turn on reflection warning in the code. Consider this code that determines the length of a string:

```
user=> (set! *warn-on-reflection* true)
true
user=> (def s "Hello, there")
#'user/s
user=> (.length s)
Reflection warning, NO_SOURCE_PATH:1 - reference to field length
can't be resolved.
12
user=> (defn str-len [^String s] (.length s))
#'user/str-len
user=> (str-len s)
12
user=> (.length ^String s) ; type hint when passing argument
12
user=> (def ^String t "Hello, there") ; type hint at var level
#'user/t
user=> (.length t) ; no more reflection warning
12
user=> (time (dotimes [_ 1000000] (.length s)))
Reflection warning,
/private/var/folders/cv/myzdv_vd675g417y92jx9bm5lflvxq/T/form-
init6904047906685577265.clj:1:28 - reference to field length
can't be resolved.
"Elapsed time: 2409.155848 msecs"
nil
user=> (time (dotimes [_ 1000000] (.length t)))
```

```
"Elapsed time: 12.991328 msecs"  
nil
```

In the previous snippet, we can clearly see there is a very big difference in performance in the code that uses reflection versus the code that does not. When working on a project, you may want reflection warning to be turned on for all files. You can do it easily in Leiningen. Just put the following entry in your `project.clj` file:

```
:profiles {:dev {:global-vars {*warn-on-reflection* true}}}
```

This will automatically turn on warning reflection every time you begin any kind of invocation via Leiningen in the dev workflow such as REPL and test.

## An array of primitives

Recall the examples on `amap` and `areduce` from the previous section. If we run them with reflection warning on, we'd be warned that it uses reflection. Let's type hint them:

```
(def a (int-array [10 20 30 40 50 60]))  
;; amap example  
(seq  
  (amap ^ints a idx ret  
    (do (println idx (seq ret))  
        (inc (aget ^ints a idx))))  
;; areduce example  
(areduce ^ints a idx ret 0  
  (do (println idx ret)  
      (+ ret idx)))
```

Note that the primitive array hint `^ints` does not work at the var level. So, it would not work if you defined the var `a`, as in the following:

```
(def ^ints a (int-array [10 20 30 40 50 60])) ; wrong, will  
complain later  
(def ^"[I" a (int-array [10 20 30 40 50 60])) ; correct  
(def ^{:tag 'ints} a (int-array [10 20 30 40 50 60])) ; correct
```

This notation is for an array of integers. Other primitive array types have similar type hints. Refer to the previous section for type hinting for various primitive array types.

## Primitives

The type hinting of primitive locals is neither required nor allowed. However, you can type hint function arguments as primitives. Clojure allows up to four arguments in functions to be type hinted:

```
(defn do-something
  [^long a ^long b ^long c ^long d]
  ...)
```

## Note

Boxing may result in something not always being a primitive. In those cases, you can coerce those using respective primitive types.

## Macros and metadata

In macros, type hinting does not work the way it does in the other parts of the code. Since macros are about transforming the **Abstract Syntax Tree (AST)**, we need to have a mental map of the transformation and we should add type hints as metadata in the code. For example, if `str-len` is a macro to find the length of a string, we make use of the following code:

```
(defmacro str-len
  [s]
  `(.length ~(with-meta s {:tag String})))
;; below is another way to write the same macro
(defmacro str-len
  [s]
  `(.length ~(vary-meta s assoc :tag `String)))
```

In the preceding code, we alter the metadata of the symbol `s` by tagging it with the type `String`, which happens to be the `java.lang.String` class in this case. For array types, we can use `[Ljava.lang.String` for an array of

string objects and similarly for others. If you try to use `str-len` listed previously, you may notice this works only when we pass the string bound to a local or a var, not as a string literal. To mitigate this, we can write the macro as follows:

```
(defmacro str-len
  [s]
  `(let [^String s# ~s] (.length s#)))
```

Here we bind the argument to a type-hinted gensym local, hence calling `.length` on it does not use reflection and there is no reflection warning emitted as such.

Type hinting via metadata also works with functions, albeit in a different notation:

```
(defn foo [] "Hello")
(defn foo ^String [] "Hello")
(defn foo (^String []) "Hello") (^String [x] (str "Hello, " x)))
```

Except for the first example in the preceding snippet, they are type hinted to return the `java.lang.String` type.

## String concatenation

The `str` function in Clojure is used to concatenate and convert to string tokens. In Java, when we write `"hello" + e`, the Java compiler translates this to an equivalent code that uses `StringBuilder` and is considerably faster than the `str` function in micro-benchmarks. To obtain close-to-Java performance, in Clojure we can use a similar mechanism with a macro directly using Java interop to avoid the indirection via the `str` function.

The **Stringer** (<https://github.com/kumarshantanu/stringer>) library adopts the same technique to come up with fast string concatenation in Clojure:

```
(require '[stringer.core :as s])
user=> (time (dotimes [_ 10000000] (str "foo" :bar 707 nil
'baz)))
"Elapsed time: 2044.284333 msecs"
```

```
nil
user=> (time (dotimes [_ 10000000] (s	strcat "foo" :bar 707 nil
'baz)))
"Elapsed time: 555.843271 msecs"
nil
```

Here, Stringer also aggressively concatenates the literals during the compile phase.

## Miscellaneous

In a type (as in `deftype`), the mutable instance variables can be optionally annotated as `^:volatile-mutable` or `^:unsynchronized-mutable`. For example:

```
(deftype Counter [^:volatile-mutable ^long now]
  ...)
```

Unlike `defprotocol`, the `definterface` macro lets us provide a return type hint for methods:

```
(definterface Foo
  (^long doSomething [^long a ^double b]))
```

The `proxy-super` macro (which is used inside the `proxy` macro) is a special case where you cannot directly apply a type hint. The reason being that it relies on the implicit `this` object that is automatically created by the `proxy` macro. In this case, you must explicitly bind `this` to a type:

```
(proxy [Object] []
  (equals [other]
    (let [^Object this this]
      (proxy-super equals other))))
```

Type hinting is quite important for performance in Clojure. Fortunately, we need to type hint only when required and it's easy to find out when. In many cases, a gain from type hinting overshadows the gains from code inlining.

# Using array/numeric libraries for efficiency

You may have noticed in the previous sections, when working with numerics, performance depends a lot on whether the data is based on arrays and primitives. It may take a lot of meticulousness on the programmer's part to correctly coerce data into primitives and arrays at all stages of the computation in order to achieve optimum efficiency. Fortunately, the high-performance enthusiasts from the Clojure community realized this issue early on and created some dedicated open source libraries to mitigate the problem.

## HipHip

**HipHip** is a Clojure library used to work with arrays of primitive types. It provides a safety net, that is, it strictly accepts only primitive array arguments to work with. As a result, passing silently boxed primitive arrays as arguments always results in an exception. HipHip macros and functions rarely need the programmer to type hint anything during the operations. It supports arrays of primitive types such as `int`, `long`, `float`, and `double`.

The HipHip project is available at <https://github.com/Prismatic/hiphip>.

As of writing, HipHip's most recent version is 0.2.0 that supports Clojure 1.5.x or above, and is tagged as an Alpha release. There is a standard set of operations provided by HipHip for arrays of all of the four primitive types: integer array operations are in the namespace `hiphip.int`; double precision array operations in `hiphip.double`; and so on. The operations are all type hinted for the respective types. All of the operations for `int`, `long`, `float`, and `double` in respective namespaces are essentially the same except for the array type:

Category	Function/macro	Description
----------	----------------	-------------

Core functions	aclone	Like <code>clojure.core/aclone</code> , for primitives
	alength	Like <code>clojure.core/alength</code> , for primitives
	aget	Like <code>clojure.core/aget</code> , for primitives
	aset	Like <code>clojure.core/aset</code> , for primitives
	ainc	Increment array element by specified value
Equiv hiphip.array operations	amake	Make a new array and fill values computed by expression
	areduce	Like <code>clojure.core/areduce</code> , with HipHip array bindings
	doarr	Like <code>clojure.core/doseq</code> , with HipHip array bindings
	amap	Like <code>clojure.core/for</code> , creates new array
	afill!	Like preceding <code>amap</code> , but overwrites array argument
Mathy operations	asum	Compute sum of array elements using expression
	aproduct	Compute product of array elements using expression
	amean	Compute mean over array elements
	dot-product	Compute dot product of two arrays
Finding minimum/maximum, Sorting	amax-index	Find maximum value in array and return the index
	amax	Find maximum value in array and return it

	amin-index	Find minimum value in array and return the index
	amin	Find minimum value in array and return it
	apartition!	Three-way partition of array: less, equal, greater than pivot
	aselect!	Gather smallest $k$ elements at the beginning of array
	asort!	Sort array in-place using Java's built-in implementation
	asort-max!	Partial in-place sort gathering top $k$ elements to the end
	asort-min!	Partial in-place sort gathering min $k$ elements to the top
	apartition-indices!	Like apartition! but mutates index-array instead of values
	aselect-indices!	Like aselect! but mutates index-array instead of values
	asort-indices!	Like asort! but mutates index-array instead of values
	amax-indices	Get index-array; last $k$ indices pointing to max $k$ values
	amin-indices	Get index-array; first $k$ indices pointing to min $k$ values

To include HipHip as a dependency in your Leiningen project, specify it in `project.clj`:

```
:dependencies [;; other dependencies
               [prismatic/hiphip "0.2.0"]]
```

As an example of how to use HipHip, let's see how to compute the normalized values of an array:

```
(require '[hiphip.double :as hd])

(def xs (double-array [12.3 23.4 34.5 45.6 56.7 67.8]))

(let [s (hd/asum xs)] (hd/amap [x xs] (/ x s)))
```

Unless we make sure that `xs` is an array of primitive doubles, HipHip will throw `ClassCastException` when the type is incorrect, and `IllegalArgumentException` in other cases. I recommend exploring the HipHip project to gain more insight into using it effectively.

## primitive-math

We can set `*warn-on-reflection*` to true to let Clojure warn us when the reflection is used at invocation boundaries. However, when Clojure has to implicitly use reflection to perform math, the only resort is to either use a profiler or compile the Clojure source down to bytecode, and analyze boxing and reflection with a decompiler. This is where the `primitive-math` library helps, by producing extra warnings and throwing exceptions.

The `primitive-math` library is available at  
<https://github.com/ztellman/primitive-math>.

As of writing, `primitive-math` is at version 0.1.4; you can include it as a dependency in your Leiningen project by editing `project.clj` as follows:

```
:dependencies [;; other dependencies
               [primitive-math "0.1.4"]]
```

The following code is how it can be used (recall the example from the *Decompiling the .class files into Java source* section):

```

;; must enable reflection warnings for extra warnings from
primitive-math
(set! *warn-on-reflection* true)
(require '[primitive-math :as pm])
(defn mul [x y] (pm/* x y)) ; primitive-math produces reflection
warning
(mul 10.3 2) ; throws exception
(defn mul [^long x ^long y] (pm/* x y)) ; no warning after type
hinting
(mul 10.3 2) ; returns 20

```

While `primitive-math` is a useful library, the problem it solves is mostly taken care of by the boxing detection feature in Clojure 1.7 (see next section *Detecting boxed math*). However, this library is still useful if you are unable to use Clojure 1.7 or higher.

## Detecting boxed math

**Boxed math** is hard to detect and is a source of performance issues. Clojure 1.7 introduces a way to warn the user when boxed math happens. This can be configured in the following way:

```

(set! *unchecked-math* :warn-on-boxed)

(defn sum-till [n] (/ (* n (inc n)) 2)) ; causes warning
Boxed math warning,
/private/var/folders/cv/myzdv_vd675g417y92jx9bm5lflvxq/T/form-
init3701519533014890866.clj:1:28 - call: public static
java.lang.Number
clojure.lang.Numbers.unchecked_inc(java.lang.Object).
Boxed math warning,
/private/var/folders/cv/myzdv_vd675g417y92jx9bm5lflvxq/T/form-
init3701519533014890866.clj:1:23 - call: public static
java.lang.Number
clojure.lang.Numbers.unchecked_multiply(java.lang.Object,java.lan-
g.Object).
Boxed math warning,
/private/var/folders/cv/myzdv_vd675g417y92jx9bm5lflvxq/T/form-
init3701519533014890866.clj:1:20 - call: public static
java.lang.Number
clojure.lang.Numbers.divide(java.lang.Object,long).

```

```
;; now we define again with type hint
(defn sum-till [^long n] (/ (* n (inc n)) 2))
```

When working with Leiningen, you can enable boxed math warnings by putting the following entry in the `project.clj` file:

```
:global-vars {*unchecked-math* :warn-on-boxed}
```

The math operations in `primitive-math` (like `HipHip`) are implemented via macros. Therefore, they cannot be used as higher order functions and, as a consequence, may not compose well with other code. I recommend exploring the project to see what suits your program use case. Adopting Clojure 1.7 obviates the boxing discovery issues by means of a boxed-warning feature.

# Resorting to Java and native code

In a handful of cases, where the lack of imperative, stack-based, mutable variables in Clojure may make the code not perform as well as Java, we may need to evaluate alternatives to make it faster. I would advise you to consider writing such code directly in Java for better performance.

Another consideration is to use native OS capabilities, such as memory-mapped buffers

(<http://docs.oracle.com/javase/7/docs/api/java/nio/MappedByteBuffer.html>) or files and unsafe operations

(<http://highlyscalable.wordpress.com/2012/02/02/direct-memory-access-in-java/>). Note that unsafe operations are potentially hazardous and not recommended in general. Such times are also an opportunity to consider writing performance-critical pieces of code in C or C++ and then access them via the **Java Native Interface (JNI)**.

## Proteus – mutable locals in Clojure

Proteus is an open source Clojure library that lets you treat a local as a local variable, thereby allowing its unsynchronized mutation within the local scope only. Note that this library depends on the internal implementation structure of Clojure as of Clojure 1.5.1. The **Proteus** project is available at <https://github.com/ztellman/proteus>.

You can include Proteus as a dependency in the Leiningen project by editing `project.clj`:

```
:dependencies [;; other dependencies
               [proteus "0.1.4"]]
```

Using Proteus in code is straightforward, as shown in the following code snippet:

```
(require '[proteus :as p])
```

```
(p/let-mutable [a 10]
  (println a)
  (set! a 20)
  (println a))
;; Output below:
;; 10
;; 20
```

Since Proteus allows mutation only in the local scope, the following throws an exception:

```
(p/let-mutable [a 10 add2! (fn [x] (set! x (+ 2 x)))]
  (add2! a)
  (println a))
```

The mutable locals are very fast and may be quite useful in tight loops. Proteus is unconventional by Clojure idioms, but it may give the required performance boost without having to write Java code.

# Summary

Clojure has strong Java interoperability and underpinning, due to which programmers can leverage the performance benefits nearing those of Java. For performance-critical code, it is sometimes necessary to understand how Clojure interacts with Java and how to turn the right knobs. Numerics is a key area where Java interoperability is required to get optimum performance. Type hints are another important performance trick that is frequently useful. There are several open source Clojure libraries that make such activities easier for the programmer.

In the next chapter, we will dig deeper below Java and see how the hardware and the JVM stack play a key role in offering the performance we get, what their constraints are, and how to use their understanding to get better performance.

# Chapter 4. Host Performance

In the previous chapters, we noted how Clojure interoperates with Java. In this chapter we will go a bit deeper to understand the internals better. We will touch upon several layers of the entire stack, but our major focus will be the JVM, in particular the Oracle HotSpot JVM, though there are several JVM vendors to choose from

([http://en.wikipedia.org/wiki/List\\_of\\_Java\\_virtual\\_machines](http://en.wikipedia.org/wiki/List_of_Java_virtual_machines)). At the time of writing this, Oracle JDK 1.8 is the latest stable release and early OpenJDK 1.9 builds are available. In this chapter we will discuss:

- How the hardware subsystems function from the performance viewpoint
- Organization of the JVM internals and how that is related to performance
- How to measure the amount of space occupied by various objects in the heap
- Profile Clojure code for latency using Criterium

## The hardware

There are various hardware components that may impact the performance of software in different ways. The processors, caches, memory subsystem, I/O subsystems, and so on, all have varying degrees of performance impact depending upon the use cases. In the following sections we look into each of those aspects.

## Processors

Since about the late 1980s, microprocessors have been employing pipelining and instruction-level parallelism to speed up their performance. Processing an instruction at the CPU level consists of typically four cycles: **fetch, decode, execute, and writeback**. Modern processors optimize the cycles by running them in parallel—while one instruction is executed, the

next instruction is being decoded, and the one after that is being fetched, and so on. This style is called **instruction pipelining**.

In practice, in order to speed up execution even further, the stages are subdivided into many shorter stages, thus leading to deeper super-pipeline architecture. The length of the longest stage in the pipeline limits the clock speed of the CPU. By splitting stages into substages, the processor can be run at a higher clock speed, where more cycles are required for each instruction, but the processor still completes one instruction per cycle. Since there are more cycles per second now, we get better performance in terms of throughput per second even though the latency of each instruction is now higher.

## Branch prediction

The processor must fetch and decode instructions in advance even when it encounters instructions of the conditional `if-then` form. Consider an equivalent of the `(if (test a) (foo a) (bar a))` Clojure expression. The processor must choose a branch to fetch and decode, the question is should it fetch the `if` branch or the `else` branch? Here, the processor makes a guess as to which instruction to fetch/decode. If the guess turns out to be correct, it is a performance gain as usual; otherwise, the processor has to throw away the result of the fetch/decode process and start on the other branch afresh.

Processors deal with branch prediction using an on-chip branch prediction table. It contains recent code branches and two bits per branch, indicating whether or not the branch was taken, while also accommodating one-off, not-taken occurrences.

Today, branch prediction is extremely important in processors for performance, so modern processors dedicate hardware resources and special predication instructions to improve the prediction accuracy and lower the cost of a mispredict penalty.

## Instruction scheduling

High-latency instructions and branching usually lead to empty cycles in the instruction pipeline known as **stalls** or **bubbles**. These cycles are often used to do other work by the means of instruction reordering. Instruction reordering is implemented at the hardware level via out of order execution and at the compiler level via compile time instruction scheduling (also called **static instruction scheduling**).

The processor needs to remember the dependencies between instructions when carrying out the out-of-order execution. This cost is somewhat mitigated by using renamed registers, wherein register values are stored into / loaded from memory locations, potentially on different physical registers, so that they can be executed in parallel. This necessitates that out-of-order processors always maintain a mapping of instructions and corresponding registers they use, which makes their design complex and power hungry. With a few exceptions, almost all high-performance CPUs today have out-of-order designs.

Good compilers are usually extremely aware of processors, and are capable of optimizing the code by rearranging processor instructions in a way that there are fewer bubbles in the processor instruction pipeline. A few high-performance CPUs still rely on only static instruction reordering instead of out-of-order instruction reordering and, in turn, save chip area due to simpler design—the saved area is used to accommodate extra cache or CPU cores. Low-power processors, such as those from the ARM and Atom family, use in-order design. Unlike most CPUs, the modern GPUs use in-order design with deep pipelines, which is compensated by very fast context switching. This leads to high latency and high throughput on GPUs.

## Threads and cores

Concurrency and parallelism via context switches, hardware threads, and cores are very common today and we have accepted them as a norm to

implement in our programs. However, we should understand why we needed such a design in the first place. Most of the real-world code we write today does not have more than a modest scope for instruction-level parallelism. Even with hardware-based, out-of-order execution and static instruction reordering, no more than two instructions per cycle are truly parallel. Hence, another potential source of instructions that can be pipelined and executed in parallel are the programs other than the currently running one.

The empty cycles in a pipeline can be dedicated to other running programs, which assume that there are other currently running programs that need the processor's attention. **Simultaneous multithreading (SMT)** is a hardware design that enables such kinds of parallelism. Intel implements SMT named as **HyperThreading** in some of its processors. While SMT presents a single physical processor as two or more logical processors, a true multiprocessor system executes one thread per processor, thus achieving simultaneous execution. A multicore processor includes two or more processors per chip, but has the properties of a multiprocessor system.

In general, multicore processors significantly outperform SMT processors. Performance on SMT processors can vary by the use case. It peaks in those cases where code is highly variable or threads do not compete for the same hardware resources, and dips when the threads are cache-bound on the same processor. What is also important is that some programs are simply not inherently parallel. In such cases it may be hard to make them go faster without the explicit use of threads in the program.

## Memory systems

It is important to understand the memory performance characteristics to know the likely impact on the programs we write. Data-intensive programs that are also inherently parallel, such as audio/video processing and scientific computation, are largely limited by memory bandwidth, not by

the processor. Adding processors would not make them faster unless the memory bandwidth is also increased. Consider another class of programs, such as 3D graphics rendering or database systems that are limited mainly by memory latency but not the memory bandwidth. SMT can be highly suitable for such programs, where threads do not compete for the same hardware resources.

Memory access roughly constitutes a quarter of all instructions executed by a processor. A code block typically begins with memory-load instructions and the remainder portion depends on the loaded data. This stalls the instructions and prevents large-scale, instruction-level parallelism. As if that was not bad enough, even superscalar processors (which can issue more than one instruction per clock cycle) can issue, at most, two memory instructions per cycle. Building fast memory systems is limited by natural factors such as the speed of light. It impacts the signal round trip to the RAM. This is a natural hard limit and any optimization can only work around it.

Data transfer between the processor and motherboard chipset is one of the factors that induce memory latency. This is countered using a **faster front-side bus (FSB)**. Nowadays, most modern processors fix this problem by integrating the memory controller directly at the chip level. The significant difference between the processor versus memory latencies is known as the **memory wall**. This has plateaued in recent times due to processor clock speeds hitting power and heat limits, but notwithstanding this, memory latency continues to be a significant problem.

Unlike CPUs, GPUs typically realize a sustained high-memory bandwidth. Due to latency hiding, they utilize the bandwidth even during a high number-crunching workload.

## Cache

To overcome the memory latency, modern processors employ a special type of very fast memory placed onto the processor chip or close to the

chip. The purpose of the cache is to store the most recently used data from the memory. Caches are of different levels: **L1** cache is located on the processor chip; **L2** cache is bigger and located farther away from the processor compared to L1. There is often an **L3** cache, which is even bigger and located farther from the processor than L2. In Intel's Haswell processor, the L1 cache is generally 64 kilobytes (32 KB instruction plus 32 KB data) in size, L2 is 256 KB per core, and L3 is 8 MB.

While memory latency is very bad, fortunately caches seem to work very well. The L1 cache is much faster than accessing the main memory. The reported cache hit rates in real-world programs is 90 percent, which makes a strong case for caches. A cache works like a dictionary of memory addresses to a block of data values. Since the value is a block of memory, the caching of adjacent memory locations has mostly no additional overhead. Note that L2 is slower and bigger than L1, and L3 is slower and bigger than L2. On Intel Sandybridge processors, register lookup is instantaneous; L1 cache lookup takes three clock cycles, L2 takes nine, L3 takes 21, and main memory access takes 150 to 400 clock cycles.

## Interconnect

A processor communicates with the memory and other processors via interconnect that are generally of two types of architecture: **Symmetric multiprocessing (SMP)** and **Non-uniform memory access (NUMA)**. In SMP, a bus interconnects processors and memory with the help of bus controllers. The bus acts as a broadcast device for the end points. The bus often becomes a bottleneck with a large number of processors and memory banks. SMP systems are cheaper to build and harder to scale to a large number of cores compared to NUMA. In a NUMA system, collections of processors and memory are connected point to point to other such groups of processors and memory. Every such group is called a node. Local memory of a node is accessible by other nodes and vice versa. Intel's **HyperTransport** and **QuickPath** interconnect technologies support NUMA.

# **Storage and networking**

Storage and networking are the most commonly used hardware components besides the processor, cache, and memory. Many of the real-world applications are more often I/O bound than execution-bound. Such I/O technologies are continuously advancing and there is a wide variety of components available in the market. The consideration of such devices should be based on the exact performance and reliability characteristics for the use case. Another important criterion is to know how well they are supported by the target operating system drivers. Current day storage technologies mostly build upon hard disks and solid state drives. The applicability of network devices and protocols vary widely as per the business use case. A detailed discussion of I/O hardware is beyond the scope of this book.

# The Java Virtual Machine

The Java Virtual Machine is a bytecode-oriented, garbage-collected virtual machine that specifies its own instruction set. The instructions have equivalent bytecodes that are interpreted and compiled to the underlying OS and hardware by the **Java Runtime Environment (JRE)**. Objects are referred to using symbolic references. The data types in the JVM are fully standardized as a single spec across all JVM implementations on all platforms and architectures. The JVM also follows the network byte order, which means communication between Java programs on different architectures can happen using the big-endian byte order. **Jvmtop** (<https://code.google.com/p/jvmtop/>) is a handy JVM monitoring tool similar to the top command in Unix-like systems.

## The just-in-time compiler

The **just-in-time (JIT)** compiler is part of the JVM. When the JVM starts up, the JIT compiler knows hardly anything about the running code so it simply interprets the JVM bytecodes. As the program keeps running, the JIT compiler starts profiling the code by collecting statistics and analyzing the call and bytecode patterns. When a method call count exceeds a certain threshold, the JIT compiler applies a number of optimizations to the code. Most common optimizations are inlining and native code generating. The final and static methods and classes are great candidates for inlining. JIT compilation does not come without a cost; it occupies memory to store the profiled code and sometimes it has to revert the wrong speculative optimization. However, JIT compilation is almost always amortized over a long duration of code execution. In rare cases, turning off JIT compilation may be useful if either the code is too large or there are no hotspots in the code due to infrequent execution.

A JRE has typically two kinds of JIT compilers: client and server. Which JIT compiler is used by default depends on the type of hardware and

platform. The client JIT compiler is meant for client programs such as command-line and desktop applications. We can start the JRE with the -server option to invoke the server JIT compiler, which is really meant for long-running programs on a server. The threshold for JIT compilation is higher in the server than the client. The difference in the two kinds of JIT compilers is that the client targets upfront, visible lower latency, and the server is assumed to be running on a high-resource hardware and tries to optimize for throughput.

The JIT compiler in the Oracle HotSpot JVM observes the code execution to determine the most frequently invoked methods, which are hotspots. Such hotspots are usually just a fraction of the entire code that can be cheap to focus on and optimize. The **HotSpot JIT** compiler is lazy and adaptive. It is lazy because it compiles only those methods to native code that have crossed a certain threshold, and not all the code that it encounters. Compiling to native code is a time-consuming process and compiling all code would be wasteful. It is adaptive to gradually increasing the aggressiveness of its compilation on frequently called code, which implies that the code is not optimized only once but many times over as the code gets executed repeatedly. After a method call crosses the first JIT compiler threshold, it is optimized and the counter is reset to zero. At the same time, the optimization count for the code is set to one. When the call exceeds the threshold yet again, the counter is reset to zero and the optimization count is incremented; and this time a more aggressive optimization is applied. This cycle continues until the code cannot be optimized anymore.

The HotSpot JIT compiler does a whole bunch of optimizations. Some of the most prominent ones are as follows:

- **Inlining:** Inlining of methods—very small methods, the static and final methods, methods in final classes, and small methods involving only primitive numerics are prime candidates for inlining.
- **Lock elimination:** Locking is a performance overhead. Fortunately, if the lock object monitor is not reachable from other threads, the lock is

eliminated.

- **Virtual call elimination:** Often, there is only one implementation for an interface in a program. The JIT compiler eliminates the virtual call and replaces that with a direct method call on the class implementation object.
- **Non-volatile memory write elimination:** The non-volatile data members and references in an object are not guaranteed to be visible by the threads other than the current thread. This criterion is utilized not to update such references in memory and rather use hardware registers or the stack via native code.
- **Native code generation:** The JIT compiler generates native code for frequently invoked methods together with the arguments. The generated native code is stored in the code cache.
- **Control flow and local optimizations:** The JIT compiler frequently reorders and splits the code for better performance. It also analyzes the branching of control and optimizes code based on that.

There should rarely be any reason to disable JIT compilation, but it can be done by passing the `-Djava.compiler=NONE` parameter when starting the JRE. The default compile threshold can be changed by passing `-XX:CompileThreshold=9800` to the JRE executable where 9800 is the example threshold. The `XX:+PrintCompilation` and `-XX:-CITime` options make the JIT compiler print the JIT statistics and time spent on JIT.

## Memory organization

The memory used by the JVM is divided into several segments. JVM, being a stack-based execution model, one of the memory segments is the stack area. Every thread is given a stack where the stack frames are stored in **Last-in-First-out (LIFO)** order. The stack includes a **program counter (PC)** that points to the instruction in the JVM memory currently being executed. When a method is called, a new stack frame is created containing the local variable array and the operand stack. Contrary to conventional stacks, the operand stack holds instructions to load local

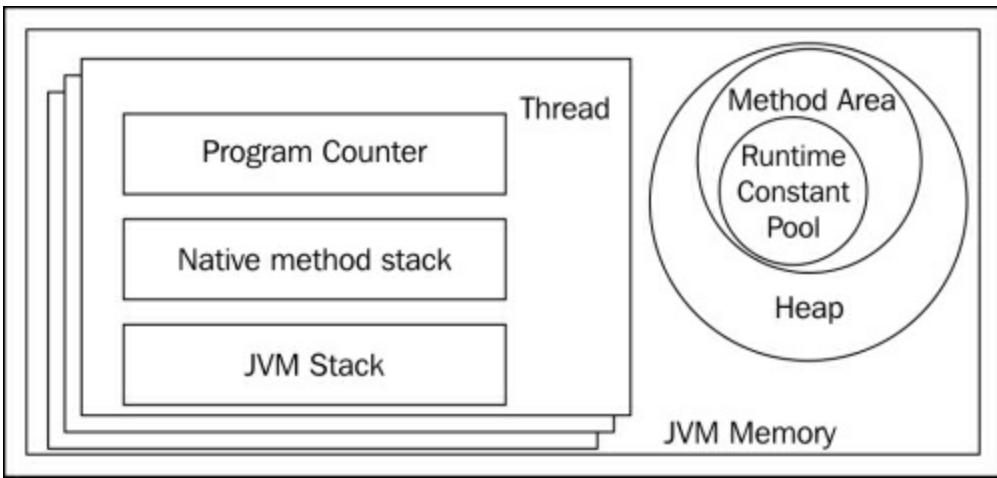
variable / field values and computation results—a mechanism that is also used to prepare method parameters before a call and to store the return value. The stack frame itself may be allocated on the heap. The easiest way to inspect the order of stack frames in the current thread is to execute the following code:

```
(require 'clojure.repl)
(clojure.repl/pst (Throwable.))
```

When a thread requires more stack space than what the JVM can provide, `StackOverflowError` is thrown.

The heap is the main memory area where the object and array allocations are done. It is shared across all JVM threads. The heap may be of a fixed size or expanding, depending on the arguments passed to the JRE on startup. Trying to allocate more heap space than what the JVM can make room for results in `OutOfMemoryError` to be thrown. The allocations in the heap are subject to garbage collection. When an object is no more reachable via any reference, it is garbage collected, with the notable exception of weak, soft, and phantom references. Objects pointed to by non-strong references take longer to GC.

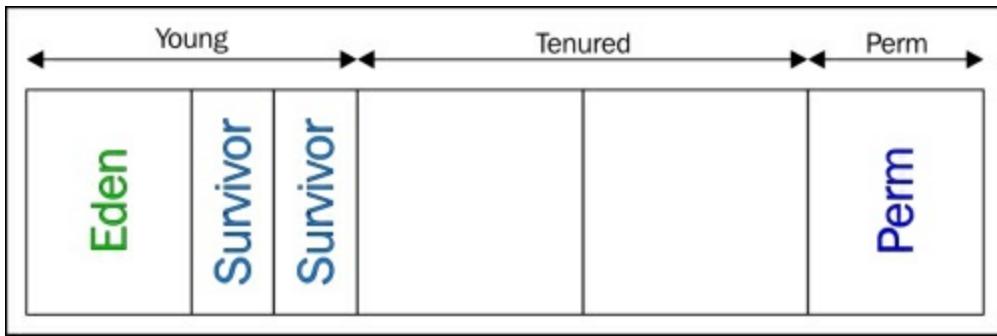
The method area is logically a part of the heap memory and contains per-class structures such as the field and method information, runtime constant pool, code for method, and constructor bodies. It is shared across all JVM threads. In the Oracle HotSpot JVM (up to Version 7), the method area is found in a memory area called the **permanent generation**. In HotSpot Java 8, the permanent generation is replaced by a native memory area called **Metaspace**.



The JVM contains the native code and the Java bytecode to be provided to the Java API implementation and the JVM implementation. The native code call stack is maintained separately for each thread stack. The JVM stack contains the Java method calls. Please note that the JVM spec for Java SE 7 and 8 does not imply a native method stack, but for Java SE 5 and 6, it does.

## HotSpot heap and garbage collection

The Oracle HotSpot JVM uses a generational heap. The three main generations are **Young**, **Tenured** (old), and **Perm** (permanent) (up to HotSpot JDK 1.7 only). As objects survive garbage collection, they move from **Eden** to **Survivor** and from **Survivor** to **Tenured** spaces. The new instances are allocated in the **Eden** segment, which is a very cheap operation (as cheap as a pointer bump, and faster than a C `malloc` call), if it already has sufficient free space. When the Eden area does not have enough free space, a minor GC is triggered. This copies the live objects from **Eden** into the **Survivor** space. In the same operation, live objects are checked in **Survivor-1** and copied over to **Survivor-2**, thus keeping the live objects only in **Survivor-2**. This scheme keeps **Eden** and **Survivor-1** empty and unfragmented to make new allocations, and is known as **copy collection**.



After a certain survival threshold in the young generation, the objects are moved to the tenured/old generation. If it is not possible to do a minor GC, a major GC is attempted. The major GC does not use copying, but rather relies on mark-and-sweep algorithms. We can use throughput collectors (**Serial**, **Parallel**, and **ParallelOld**) or low-pause collectors (**Concurrent** and **G1**) for the old generation. The following table shows a non-exhaustive list of options to be used for each collector type:

Collector name	JVM flag
Serial	-XX:+UseSerialGC
Parallel	-XX:+UseParallelGC
Parallel Compacting	-XX:+UseParallelOldGC
Concurrent	-XX:+UseConcMarkSweepGC -XX:+UseParNewGC -XX:+CMSParallelRemarkEnabled
G1	-XX:+UseG1GC

The previously mentioned flags can be used to start the Java runtime. For example, in the following command, we start the server JVM with a 4 GB

heap using Parallel compacting GC:

```
java \
-server \
-Xms4096m -Xmx4096m \
-XX:+UseParallelOldGC XX:ParallelGCThreads=4 \
-jar application-standalone.jar
```

Sometimes, due to running full GC multiple times, the tenured space may have become so fragmented that it may not be feasible to move objects from Survivor to Tenured spaces. In those cases, a full GC with compaction is triggered. During this period, the application may appear unresponsive due to the full GC in action.

## Measuring memory (heap/stack) usage

One of the prime reasons for a performance hit in the JVM is garbage collection. It certainly helps to know how heap memory is used by the objects we create and how to reduce the impact on GC by means of a lower footprint. Let's inspect how the representation of an object may lead to heap space.

Every (uncompressed) object or array reference on a 64-bit JVM is 16 bytes long. On a 32-bit JVM, every reference is 8 bytes long. As the 64-bit architecture is becoming more commonplace now, the 64-bit JVM is more likely to be used on servers. Fortunately, for a heap size of up to 32 GB, the JVM (Java 7) can use compressed pointers (default behavior) that are only 4 bytes in size. Java 8 VMs can address up to 64 GB heap size via compressed pointers as seen in the following table:

	Uncompressed	Compressed	32-bit
Reference (pointer)	8	4	4
Object header	16	12	8

Array header	24	16	12
Superclass padding	8	4	4

This table illustrates pointer sizes in different modes (reproduced with permission from Attila Szegedi:

<http://www.slideshare.net/aszegedi/everything-i-ever-learned-about-jvm-performance-tuning-twitter/20>).

We saw in the previous chapter how many bytes each primitive type takes. Let's see how the memory consumption of the composite types looks with compressed pointers (a common case) on a 64-bit JVM with a heap size smaller than 32 GB:

Java Expression	64-bit memory usage	Description (b = bytes, padding toward memory word size in approximate multiples of 8)
<code>new Object()</code>	16 bytes	12 b header + 4 b padding
<code>new byte[0]</code>	16 bytes	12 b <code>obj</code> header + 4 b <code>int</code> length = 16 b array header
<code>new String("foo")</code>	40 bytes (interned for literals)	12 b header + (12 b array header + 6 b char-array content + 4 b length + 2 b padding = 24 b) + 4 b hash
<code>new Integer(3)</code>	16 bytes (boxed integer)	12 b header + 4 b <code>int</code> value
<code>new Long(4)</code>	24 bytes (boxed long)	12 b header + 8 b <code>long</code> value + 4 b padding
<code>class A { byte x; }</code> <code>new A();</code>	16 bytes	12 b header + 1 b value + 3 b padding

class B extends A {byte y;}  new B();	24 bytes (subclass padding)	12 b reference + (1 b value + 7 b padding = 8 b) for A + 1 b for value of y + 3 b padding
clojure.lang.Symbol.intern("foo")  // clojure 'foo	104 bytes (40 bytes interned)	12 b header + 12 b ns reference + (12 b name reference + 40 b interned chars) + 4 b int hash + 12 b meta reference + (12 b _str reference + 40 b interned chars) – 40 b interned str
clojure.lang.Keyword.intern("foo")  // clojure :foo	184 bytes (fully interned by factory method)	12 b reference + (12 b symbol reference + 104 b interned value) + 4 b int hash + (12 b _str reference + 40 b interned char)

A comparison of space taken by a symbol and a keyword created from the same given string demonstrates that even though a keyword has slight overhead over a symbol, the keyword is fully interned and would provide better guard against memory consumption and thus GC over time. Moreover, the keyword is interned as a weak reference, which ensures that it is garbage collected when no keyword in memory is pointing to the interned value anymore.

## Determining program workload type

We often need to determine whether a program is CPU/cache bound, memory bound, I/O bound or contention bound. When a program is I/O or contention bound, the CPU usage is generally low. You may have to use a profiler (we will see this in [Chapter 7, Performance Optimization](#)) to find out whether threads are stuck due to resource contention. When a program is CPU/cache or memory bound, CPU usage may not be a clear indicator of the source of the bottleneck. In such cases, you may want to make an educated guess by inspecting cache misses in the program. On Linux systems tools such as **perf** (<https://perf.wiki.kernel.org/>), **cachegrind** (<http://valgrind.org/info/tools.html#cachegrind>) and **oprofile** (<http://oprofile.sourceforge.net/>) can help determine the volume of cache misses—a higher threshold may imply that the program is memory bound.

However, using these tools with Java is not straightforward because Java's JIT compiler needs a warm-up until meaningful behavior can be observed. The project **perf-map-agent** (<https://github.com/jrudolph/perf-map-agent>) can help generate method mappings that you can correlate using the `perf` utility.

## Tackling memory inefficiency

In earlier sections in this chapter we discussed that unchecked memory access may become a bottleneck. As of Java 8, due to the way the heap and object references work, we cannot fully control the object layout and memory access patterns. However, we can take care of the frequently executed blocks of code to consume less memory and attempt to make them cache-bound instead of memory-bound at runtime. We can consider a few techniques to lower memory consumption and randomness in access:

- Primitive locals (long, double, boolean, char, etc) in the JVM are created on the stack. The rest of the objects are created on the heap and only their references are stored in the stack. Primitives have a low overhead and do not require memory indirection for access, and are hence recommended.
- Data laid out in the main memory in a sequential fashion is faster to access than randomly laid out data. When we use a large (say more than eight elements) persistent map, the data stored in tries may not be sequentially laid out in memory, rather they would be randomly laid out in the heap. Moreover both keys and values are stored and accessed. When you use records (`defrecord`) and types (`deftype`), not only do they provide array/class semantics for the layout of fields within them, they do not store the keys, which is very efficient compared to regular maps.
- Reading large content from a disk or the network may have an adverse impact on performance due to random memory roundtrips. In [Chapter 3, \*Leaning on Java\*](#), we briefly discussed memory-mapped byte buffers. You can leverage memory-mapped buffers to minimize fragmented object allocation/access on the heap. While libraries such

as `nio` (<https://github.com/pjstadig/nio/>) and `clj-mmap` (<https://github.com/thebusby/clj-mmap>) help us deal with memory-mapped buffers, `bytebuffer` (<https://github.com/geoffsalmon/bytebuffer>), and `gloss` (<https://github.com/ztellman/gloss>) let us work with byte buffers. There are also alternate abstractions such as `iota` (<https://github.com/thebusby/iota>) that help us deal with large files as collections.

Given that memory bottleneck is a potential performance issue in data-intensive programs, lowering memory overhead goes a long way in avoiding performance risk. Understanding low-level details of the hardware, the JVM and Clojure's implementation helps us choose the appropriate techniques to tackle the memory bottleneck issue.

# Measuring latency with Criterium

Clojure has a neat little macro called `time` that evaluates the body of code passed to it, and then prints out the time it took and simply returns the value. However, we can note that often the time taken to execute the code varies quite a bit across various runs:

```
user=> (time (reduce + (range 100000)))
"Elapsed time: 112.480752 msecs"
4999950000
user=> (time (reduce + (range 1000000)))
"Elapsed time: 387.974799 msecs"
499999500000
```

There are several reasons associated to this variance in behavior. When cold started, the JVM has its heap segments empty and is unaware of the code path. As the JVM keeps running, the heap fills up and the GC patterns start becoming noticeable. The JIT compiler gets a chance to profile the different code paths and optimize them. Only after quite some GC and JIT compilation rounds, does the JVM performance become less unpredictable.

Criterium (<https://github.com/hugoduncan/criterium>) is a Clojure library to scientifically measure the latency of Clojure expressions on a machine. A summary of how it works can be found at the Criterium project page. The easiest way to use Criterium is to use it with Leiningen. If you want Criterium to be available only in the REPL and not as a project dependency, add the following entry to the `~/.lein/profiles.clj` file:

```
{:user {:plugins [[criterium "0.4.3"]]}}
```

Another way is to include `criterium` in your project in the `project.clj` file:

```
:dependencies [[org.clojure/clojure "1.7.0"]
              [criterium "0.4.3"]]
```

Once done with the editing of the file, launch REPL using `lein repl`:

```
user=> (require '[criterium.core :as c])
nil
user=> (c/bench (reduce + (range 100000)))
Evaluation count : 1980 in 60 samples of 33 calls.
          Execution time mean : 31.627742 ms
          Execution time std-deviation : 431.917981 us
          Execution time lower quantile : 30.884211 ms ( 2.5%)
          Execution time upper quantile : 32.129534 ms (97.5%)
nil
```

Now, we can see that, on average, the expression took 31.6 ms on a certain test machine.

## Criterium and Leiningen

By default, Leiningen starts the JVM in a low-tiered compilation mode, which causes it to start up faster but impacts the optimizations that the JRE can perform at runtime. To get the best effects when running tests with Criterium and Leiningen for a server-side use case, be sure to override the defaults in `project.clj` as follows:

```
:jvm-opts ^{:replace ["-server"]}
```

The `^{:replace}` hint causes Leiningen to replace its own defaults with what is provided under the `:jvm-opts` key. You may like to add more parameters as needed, such as a minimum and maximum heap size to run the tests.

# Summary

The performance of a software system is directly impacted by its hardware components, so understanding how the hardware works is crucial. The processor, caches, memory, and I/O subsystems have different performance behaviors. Clojure, being a hosted language, understanding the performance properties of the host, that is, the JVM, is equally important. The Criterium library is useful for measuring the latency of the Clojure code—we will discuss Criterium again in [Chapter 6, Measuring Performance](#). In the next chapter we will look at the concurrency primitives in Clojure and their performance characteristics.

# Chapter 5. Concurrency

Concurrency was one of the chief design goals of Clojure. Considering the concurrent programming model in Java (the comparison with Java is due to it being the predominant language on the JVM), it is not only too low level, but rather tricky to get right that without strictly following the patterns, one is more likely to shoot oneself in the foot. Locks, synchronization, and unguarded mutation are recipes for the concurrency pitfalls, unless exercised with extreme caution. Clojure's design choices deeply influence the way in which the concurrency patterns can be achieved in a safe and functional manner. In this chapter, we will discuss:

- The low level concurrency support at the hardware and JVM level
- The concurrency primitives of Clojure—atoms, agents, refs and vars
- The built-in concurrency that features in Java safe, and its usefulness with Clojure
- Parallelization with the Clojure features and reducers

## Low-level concurrency

Concurrency cannot be achieved without explicit hardware support. We discussed about SMT and the multi-core processors in the previous chapters. Recall that every processor core has its own L1 cache, and several cores share the L2 cache. The shared L2 cache provides a fast mechanism to the processor cores to coordinate their cache access, eliminating the comparatively expensive memory access. Additionally, a processor buffers the writes to memory into something known as a **dirty write-buffer**. This helps the processor to issue a batch of memory update requests, reorder the instructions, and determine the final value to write to memory, known as **write absorption**.

## Hardware memory barrier (fence) instructions

Memory access reordering is great for a sequential (single-threaded)

program performance, but it is hazardous for the concurrent programs where the order of memory access in one thread may disrupt the expectations in another thread. The processor needs the means of synchronizing the access in such a way that memory reordering is either compartmentalized in code segments that do not care, or is prevented where it might have undesirable consequences. The hardware supports such a safety measure in terms of a "memory barrier" (also known as "fence").

There are several kinds of memory barrier instructions found in different architectures, with potentially different performance characteristics. The compiler (or the JIT compiler in the case of the JVM) usually knows about the fence instructions on the architectures that it runs on. The common fence instructions are read, write, acquire, and release barrier, and more. The barriers do not guarantee the latest data, rather they only control the relative ordering of memory access. Barriers cause the write-buffer to be flushed after all the writes are issued, before the barrier is visible to the processor that issued it.

Read and write barriers control the order of reads and writes respectively. Writes happen via a write-buffer; but reads may happen out of order, or from the write-buffer. To guarantee the correct ordering, acquire, and release, blocks/barriers are used. Acquire and release are considered "half barriers"; both of them together (acquire and release) form a "full barrier". A full barrier is more expensive than a half barrier.

## Java support and the Clojure equivalent

In Java, the memory barrier instructions are inserted by the higher level coordination primitives. Even though fence instructions are expensive (hundreds of cycles) to run, they provide a safety net that makes accessing shared variables safe within the critical sections. In Java, the `synchronized` keyword marks a "critical section", which can be executed by only one thread at a time, thus making it a tool for "mutual exclusion". In Clojure,

the equivalent of Java's `synchronized` is the `locking` macro:

```
// Java example
synchronized (someObject) {
    // do something
}
;; Clojure example
(locking some-object
  ;; do something
)
```

The `locking` macro builds upon two special forms, `monitor-enter` and `monitor-exit`. Note that the `locking` macro is a low-level and imperative solution just like Java's `synchronized` – their use is not considered idiomatic Clojure. The special forms `monitor-enter` and `monitor-exit` respectively enter and exit the lock object's "monitor" – they are even lower level and not recommended for direct use.

Someone measuring the performance of the code that uses such locking should be aware of its single-threaded versus the multi-threaded latencies. Locking in a single thread is cheap. However, the performance penalty starts kicking in when there are two or more threads contending for a lock on the same object monitor. A lock is acquired on the monitor of an object called the "intrinsic" or "monitor" lock. Object equivalence (that is, when the `=` function returns as true) is never used for the purpose of locking. Make sure that the object references are the same (that is, when `identical?` returns as true) when locking from different threads.

Acquiring a monitor lock by a thread entails a read barrier, which invalidates the thread-local cached data, the corresponding processor registers, and the cache lines. This forces a reread from the memory. On the other hand, releasing the monitor lock results in a write barrier, which flushes all the changes to memory. These are expensive operations that impact parallelism, but they ensure consistency of data for all threads.

Java supports a `volatile` keyword for the data members in a class that

guarantees read and write to an attribute outside of a synchronized block that would not be reordered. It is interesting to note that unless an attribute is declared `volatile`, it is not guaranteed to be visible in all the threads that are accessing it. The Clojure equivalent of Java's `volatile` is the metadata called `^:volatile-mutable` that we discussed in [Chapter 3](#), *Leaning on Java*. An example of `volatile` in Java and Clojure is as follows:

```
// Java example
public class Person {
    volatile long age;
}
;; Clojure example
(deftype Person [^:volatile-mutable ^long age])
```

Reading and writing a `volatile` data requires read-acquire or write-release respectively, which means we need only a half-barrier to individually read or write the value. Note that due to a half-barrier, the read-followed-by-write operations are not guaranteed to be atomic. For example, the `age++` expression first reads the value, then increments and sets it. This makes two memory operations, which is no more a half-barrier.

Clojure 1.7 introduced a first class support for the volatile data using a new set of functions: `volatile!`, `vswap!`, `vreset!`, and `volatile?`. These functions define volatile (mutable) data and work with that. However, make a note that these functions do not work with the volatile fields in `deftype`. You can see how to use them as follows:

```
user=> (def a (volatile! 10))
#'user/a
user=> (vswap! a inc)
11
user=> @a
11
user=> (vreset! a 20)
20
user=> (volatile? a)
true
```

Operations on volatile data are not atomic, which is why even creating a volatile (using `volatile!`) is considered potentially unsafe. In general, volatiles may be useful where read consistency is not a high priority but writes must be fast, such as real-time trend analysis, or other such analytics reporting. Volatiles may also be very useful when writing stateful transducers (refer to [Chapter 2](#), *Clojure Abstractions*), serving as very fast state containers. In the next sub-section, we will see the other state abstractions that are safer (and mostly slower) than volatiles.

# Atomic updates and state

It is a common use case to read a data element, execute some logic, and update with a new value. For single-threaded programs, it bears no consequences; but for concurrent scenarios, the entire operation must be carried out in a lockstep, as an atomic operation. This case is so common that many processors support this at the hardware level using a special Compare-and-swap (CAS) instruction, which is much cheaper than locking. On x86/x64 architectures, the instruction is called CompareExchange (CMPXCHG).

Unfortunately, it is possible that another thread updates the variable with the same value that the thread, which is working on the atomic update, is going to compare the old value against. This is known as the "ABA" problem. The set of instructions such as "Load-linked" (LL) and "Store-conditional" (SC), which are found in some other architectures, provide an alternative to CAS without the ABA problem. After the LL instruction reads the value from an address, the SC instruction to update the address with a new value will only go through if the address has not been updated since the LL instruction was successful.

## Atomic updates in Java

Java has a bunch of built-in lock free, atomic, thread safe compare-and-swap abstractions for the state management. They live in the `java.util.concurrent.atomic` package. For primitive types, such as `boolean`, `integer`, and `long`, there are the `AtomicBoolean`, `AtomicInteger`, and `AtomicLong` classes respectively. The latter two classes support additional atomic add/subtract operations. For atomic reference updates, there are the `AtomicReference`, `AtomicMarkableReference`, and `AtomicStampedReference` classes for the arbitrary objects. There is also a support available for arrays where the array elements can be updated atomically—`AtomicIntegerArray`, `AtomicLongArray`, and

`AtomicReferenceArray`. They are easy to use; here is the example:

```
(import 'java.util.concurrent.atomic.AtomicReference)
(def ^AtomicReference x (AtomicReference. "foo"))
(.compareAndSet x "foo" "bar")
(import 'java.util.concurrent.atomic.AtomicInteger)
(def ^AtomicInteger y (AtomicInteger. 10))
(.getAndAdd y 5)
```

However, where and how to use it is subjected to the update points and the logic in the code. The atomic updates are not guaranteed to be non-blocking. Atomic updates are not a substitute to locking in Java, but rather a convenience, only when the scope is limited to a compare and swap operation for one mutable variable.

## Clojure's support for atomic updates

Clojure's atomic update abstraction is called "atom". It uses `AtomicReference` under the hood. An operation on `AtomicInteger` or `AtomicLong` may be slightly faster than on the Clojure `atom`, because the former uses primitives. But neither of them are too cheap, due to the compare-and-swap instruction that they use in the CPU. The speed really depends on how frequently the mutation happens, and how the JIT compiler optimizes the code. The benefit of speed may not show up until the code is run several hundred thousand times, and having an atom mutated very frequently will increase the latency due to the retries. Measuring the latency under actual (or similar to actual) load can tell better. An example of using an atom is as follows:

```
user=> (def a (atom 0))
#'user/a
user=> (swap! a inc)
1
user=> @a
1
user=> (compare-and-set! a 1 5)
true
user=> (reset! a 20)
```

The `swap!` function provides a notably different style of carrying out atomic updates than the `compareAndSwap(oldval, newval)` methods. While `compareAndSwap()` compares and sets the value, returning true if it's a success and false if it's a failure, `swap!` keeps on trying to update in an endless loop until it succeeds. This style is a popular pattern that is followed among Java developers. However, there is also a potential pitfall associated with the update-in-loop style. As the concurrency of the updaters gets higher, the performance of the update may gradually degrade. Then again, high concurrency on the atomic updates raises a question of whether or not uncoordinated updates was a good idea at all for the use-case. The `compare-and-set!` and `reset!` are pretty straightforward.

The function passed to `swap!` is required to be pure (as in side effect free), because it is retried several times in a loop during contention. If the function is not pure, the side effect may happen as many times as the retries.

It is noteworthy that atoms are not "coordinated", which means that when an atom is used concurrently by different threads, we cannot predict the order in which the operations work on it, and we cannot guarantee the end result as a consequence. The code we write around atoms should be designed with this constraint in mind. In many scenarios, atoms may not be a good fit due to the lack of coordination—watch out for that in the program design. Atoms support meta data and basic validation mechanism via extra arguments. The following examples illustrate these features:

```
user=> (def a (atom 0 :meta {:foo :bar}))
user=> (meta a)
{:foo :bar}
user=> (def age (atom 0 :validator (fn [x] (if (> x 200) false
true))))
user=> (reset! age 200)
200
```

```
user=> (swap! age inc)
IllegalStateException Invalid reference state
clojure.lang.ARef.validate (ARef.java:33)
```

The second important thing is that atoms support adding and removing watches on them. We will discuss watches later in the chapter.

## Faster writes with atom striping

We know that atoms present contention when multiple threads try to update the state at the same time. This implies that atoms have great performance when the writes are infrequent. There are some use cases, for example metrics counters, where the writes need to be fast and frequent, but the reads are fewer and can tolerate some inconsistency. For such use cases, instead of directing all the updates to a single atom, we can maintain a bunch of atoms where each thread updates a different atom, thus reducing contention. Reads from these atoms cannot be guaranteed to be consistent. Let's develop an example of such a counter:

```
(def ^:const n-cpu (.availableProcessors (Runtime/getRuntime)))
(def counters (vec (repeatedly n-cpu #(atom 0))))
(defn inc! []
  ;; consider java.util.concurrent.ThreadLocalRandom in Java 7+
  ;; which is faster than Math/random that rand-int is based on
  (let [i (rand-int n-cpu)]
    (swap! (get counters i) inc)))
(defn value []
  (transduce (map deref) + counters))
```

In the previous example, we created a vector called `counters` of the same size as the number of CPU cores in the computer, and initialize each element with an atom of initial value 0. The function called `inc!` updates the counter by picking up a random atom from `counters`, and incrementing the value by 1. We also assumed that `rand-int` distributes the picking up of atom uniformly across all the processor cores, so that we have almost zero contention. The `value` function simply walks over all the atoms and adds up their `deref`'ed values to return the counter value. The example uses `clojure.core/rand-int`, which depends on `java.lang.Math/random`.

(due to Java 6 support) to randomly find out the next counter atom. Let's see how we can optimize this when using Java 7 or above:

```
(import 'java.util.concurrent.ThreadLocalRandom)
(defn inc! []
  (let [i (.nextLong (ThreadLocalRandom/current) n-cpu)]
    (swap! (get counters i) inc)))
```

Here, we import the `java.util.concurrent.ThreadLocalRandom` class, and define the `inc!` function to pick up the next random atom using `ThreadLocalRandom`. Everything else remains the same.

# Asynchronous agents and state

While atoms are synchronous, agents are the asynchronous mechanism in Clojure to effect any change in the state. Every agent is associated with a mutable state. We pass a function (known as "action") to an agent with the optional additional arguments. This function gets queued for processing in another thread by the agent. All the agents share two common thread pools —one for the low-latency (potentially CPU-bound, cache-bound, or memory-bound) jobs, and one for the blocking (potentially I/O related or lengthy processing) jobs. Clojure provides the `send` function for the low-latency actions, `send-off` for blocking actions, and `send-via` to have the action executed on the user-specified thread-pool, instead of either of the preconfigured thread pools. All of `send`, `send-off`, and `send-via` return immediately. Here is how we can use them:

```
(def a (agent 0))
;; invoke (inc 0) in another thread and set state of a to result
(send a inc)
@a ; returns 1
;; invoke (+ 1 2 3) in another thread and set state of a to
result
(send a + 2 3)
@a ; returns 6

(shutdown-agents) ; shuts down the thread-pools
;; no execution of action anymore, hence no result update either
(send a inc)
@a ; returns 6
```

When we inspect the Clojure (as of version 1.7.0) source code, we can find that the thread-pool for the low-latency actions is named as `pooledExecutor` (a bounded thread-pool, initialized to max '2 + number of hardware processors' threads), and the thread-pool for the high-latency actions is named as `soloExecutor` (an unbounded thread pool). The premise of this default configuration is that the CPU/cache/memory-bound actions run most optimally on a bounded thread-pool, with the default

number of threads. The I/O bound tasks do not consume CPU resources. Hence, a relatively larger number of such tasks can execute at the same time, without significantly affecting the performance of the CPU/cache/memory-bound jobs. Here is how you can access and override the thread-pools:

```
(import 'clojure.lang.Agent)
Agent/pooledExecutor ; thread-pool for low latency actions
Agent/soloExecutor ; thread-pool for I/O actions
(import 'java.util.concurrent.Executors)
(def a-pool (Executors/newFixedThreadPool 10)) ; thread-pool
with 10 threads
(def b-pool (Executors/newFixedThreadPool 100)) ; 100 threads
pool
(def a (agent 0))
(send-via a-pool a inc) ; use 'a-pool' for the action
(set-agent-send-executor! a-pool) ; override default thread-pool
(set-agent-send-off-executor! b-pool) ; override default pool
```

If a program carries out a large number of I/O or blocking operations through agents, it probably makes sense to limit the number of threads dedicated for such actions. Overriding the `send-off` thread-pool using `set-agent-send-off-executor!` is the easiest way to limit the thread-pool size. A more granular way to isolate and limit the I/O actions on the agents is to use `send-via` with the thread-pools of appropriate sizes for various kinds of I/O and blocking operations.

## Asynchrony, queueing, and error handling

Sending an action to an agent returns immediately without blocking. If the agent is not already busy in executing any action, it "reacts" by enqueueing the action that triggers the execution of the action, in a thread, from the respective thread-pool. If the agent is busy in executing another action, the new action is simply enqueued. Once an action is executed from the action queue, the queue is checked for more entries and triggers the next action, if found. This whole "reactive" mechanism of triggering actions obviates the need of a message loop, polling the queue. This is only possible,

because the entry points to an agent's queue are controlled.

Actions are executed asynchronously on agents, which raises the question of how the errors are handled. Error cases need to be handled with an explicit, predefined function. When using a default agent construction, such as `(agent :foo)`, the agent is created without any error handler, and gets suspended in the event of any exception. It caches the exception, and refuses to accept any more actions. It throws the cached exception upon sending any action until the agent is restarted. A suspended agent can be reset using the `restart-agent` function. The objective of such suspension is safety and supervision. When the asynchronous actions are executed on an agent and suddenly an error occurs, it will require attention. Check out the following code:

```
(def g (agent 0))
(send g (partial / 10)) ; ArithmeticException due to divide-by-zero
@g ; returns 0, because the error did not change the old state
(send g inc) ; throws the cached ArithmeticException
(agent-error g) ; returns (doesn't throw) the exception object
(restart-agent g @g) ; clears the suspension of the agent
(agent-error g) ; returns nil
(send g inc) ; works now because we cleared the cached error
@g ; returns 1
(dotimes [_ 1000] (send-off g long-task))
;; block for 100ms or until all actions over (whichever earlier)
(await-for 100 g)
(await g) ; block until all actions dispatched till now are over
```

There are two optional parameters `:error-handler` and `:error-mode`, which we can configure on an agent to have finer control over the error handling and suspension as shown in the following code snippet:

```
(def g (agent 0 :error-handler (fn [x] (println "Found:" x)))) ;
incorrect arity
(send g (partial / 10)) ; no error encountered because error-
handler arity is wrong
(def g (agent 0 :error-handler (fn [ag x] (println "Found:" x)))) ;
correct arity
```

```
(send g (partial / 10)) ; prints the message
(set-error-handler! g (fn [ag x] (println "Found:" x))) ; equiv
of :error-handler arg
(def h (agent 0 :error-mode :continue))
(send h (partial / 10)) ; error encountered, but agent not
suspended
(send h inc)
@h ; returns 1
(set-error-mode! h :continue) ; equiv of :error-mode arg, other
possible value :fail
```

## Why you should use agents

Just as the "atom" implementation uses only compare-and-swap instead of locking, the underlying "agent" specific implementation uses mostly the compare-and-swap operations. The agent implementation uses locks only when dispatching action in a transaction (discussed in the next section), or when restarting an agent. All the actions are queued and dispatched serially in the agents, regardless of the concurrency level. The serial nature makes it possible to execute the actions in an independent and contention-free manner. For the same agent, there can never be more than one action being executed. Since there is no locking, reads (`deref` or `@`) on agents are never blocked due to writes. However, all the actions are independent of each other—there is no overlap in their execution.

The implementation goes so far as to ensure that the execution of an action blocks other actions, which follow in the queue. Even though the actions are executed in a thread-pool, actions for the same agent are never executed concurrently. This is an excellent ordering guarantee that also extends a natural coordination mechanism, due to its serial nature. However, note that this ordering coordination is limited to only a single agent. If an agent action sends actions to two other agents, they are not automatically coordinated. In this situation, you may want to use transactions (which will be covered in the next section).

Since agents distinguish between the low-latency and blocking jobs, the

jobs are executed in an appropriate kind of thread-pools. Actions on different agents may execute concurrently, thereby making optimum use of the threading resources. Unlike atoms, the performance of the agents is not impeded by high contention. In fact, for many cases, agents make a lot of sense due to the serial buffering of actions. In general, agents are great for high volume I/O tasks, or where the ordering of operations provides a win in the high contention scenarios.

## Nesting

When an agent action sends another action to the same agent, that is a case of nesting. This would have been nothing special if agents didn't participate in STM transactions (which will be covered in the next section). However, agents do participate in STM transactions and that places certain constraints on agent implementation that warrants a second-layer buffering of actions. For now, it should suffice to say that the nested sends are queued in a thread-local queue instead of the regular queue in the agent. The thread-local queue is visible only to the thread in which the action is executed. Upon executing an action, unless there was an error, the agent implicitly calls the equivalent of `release-pending-sends` function, which transfers the actions from second level thread-local queue to the normal action queue. Note that nesting is simply an implementation detail of agents and has no other impact.

# Coordinated transactional ref and state

We saw in an earlier section that an atom provides atomic read-and-update operation. What if we need to perform an atomic read-and-update operation across two or even more number of atoms? This clearly poses a coordination problem. Some entity has to watch over the process of reading and updating, so that the values are not corrupted. This is what a ref provides—a **Software Transactional Memory (STM)** based system that takes care of concurrent atomic read-and-update operations across multiple refs, such that either all the updates go through, or in the case of failure, none does. Like atoms, on failure, refs retry the whole operation from scratch with the new values.

Clojure's STM implementation is coarse grained. It works at the application level objects and aggregates (that is, references to aggregates), scoped to only all the refs in a program, constituting the "Ref world". Any update to a ref can only happen synchronously, in a transaction, in a `dosync` block of code, within the same thread. It cannot span beyond the current thread. The implementation detail reveals that a thread-local transaction context is maintained during a lifetime of a transaction. The same context ceases to be available, the moment the control reaches another thread.

Like the other reference types in Clojure, reads on a ref are never blocked by the updates, and vice versa. However, unlike the other reference types, the implementation of ref does not depend on a lock-free spinning, but rather, it internally uses locks, a low-level wait/notify, a deadlock detection, and the age-based barging.

The `alter` function is used to read-and-update the value of a ref, and `ref-set` is used to reset the value. Roughly, `alter` and `ref-set`, for the refs,

are analogous to `swap!` and `reset!` for the atoms. Just like `swap!`, `alter` accepts a function (and arguments) with no side effects, and may be retried several times during the contention. However, unlike with the atoms, not only `alter` but also `ref-set` and simple `deref`, may cause a transaction to be retried during the contention. Here is a very simple example on how we may use a transaction:

```
(def r1 (ref [:a :b :c]))
(def r2 (ref [1 2 3]))
(alter r1 conj :d) ; IllegalStateException No transaction
running...
(dosync (let [v (last @r1)] (alter r1 pop) (alter r2 conj v)))
@r1 ; returns [:a :b]
@r2 ; returns [1 2 3 :c]
(dosync (ref-set r1 (conj @r1 (last @r2))) (ref-set r2 (pop
@r2)))
@r1 ; returns [:a :b :c]
@r2 ; returns [1 2 3]
```

## Ref characteristics

Clojure maintains the **Atomicity, Consistency, and Isolation (ACI)** characteristics in a transaction. This overlaps with A, C, and I of the ACID guarantee that many databases provide. Atomicity implies that either all of the updates in a transaction will complete successfully or none of them do. Consistency means that the transaction must maintain general correctness, and should honor the constraints set by the validation—any exception or validation error should roll back the transaction. Unless a shared state is guarded, concurrent updates on it may lead a multi-step transaction into seeing different values at different steps. Isolation implies that all the steps in a transaction will see the same value, no matter how concurrent the updates are.

The Clojure refs use something known as **Multi Version Concurrency Control (MVCC)** to provide **Snapshot Isolation** to the transactions. In MVCC, instead of locking (which could block the transactions), the queues are maintained, so that each transaction can occur using its own

snapshot copy, taken at its "read point", independent of other transactions. The main benefit of this approach is that the read-only out-of-transaction operations can go through without any contention. Transactions without the ref contention go through concurrently. In a rough comparison with the database systems, the Clojure ref isolation level is "Read Committed" for reading a Ref outside of a transaction, and "Repeatable Read" by default when inside the transaction.

## Ref history and in-transaction deref operations

We discussed earlier that both, read and update operations, on a ref, may cause a transaction to be retried. The reads in a transaction can be configured to use the ref history in such a manner that the snapshot isolation instances are stored in the history queues, and are used by the read operations in the transactions. The default, which is not supposed to use the history queues, conserves heap space, and provides strong consistency (avoids the staleness of data) in the transactions.

Using the ref history reduces the likelihood of the transaction retries caused by read contention, thereby providing a weak consistency. Therefore, it is a tool for performance optimization, which comes at the cost of consistency. In many scenarios, programs do not need strong consistency—we can choose appropriately if we know the trade-off, and what we need. The snapshot isolation mechanism in the Clojure ref implementation is backed by the adaptive history queues. The history queues grow dynamically to meet the read requests, and do not overshoot the maximum limit that is set for the ref. By default, the history is not enabled, so we need to specify it during the initialization or set it later. Here is an example of how to use the history:

```
(def r (ref 0 :min-history 5 :max-history 10))
(ref-history-count r) ; returns 0, because no snapshot instances
are queued so far
(ref-min-history r) ; returns 5
(ref-max-history r) ; returns 10
(future (dosync (println "Sleeping 20 sec")) (Thread/sleep 20000))
```

```
(ref-set r 10))
(dosync (alter r inc)) ; enter this within few seconds after the
previous expression
;; The message "Sleeping 20 sec" should appear twice due to
transaction-retry
(ref-history-count r) ; returns 2, the number of snapshot
history elements
(.trimHistory ^clojure.lang.Ref r)
(ref-history-count r) ; returns 0 because we wiped the history
(ref-min-history r 10) ; reset the min history
(ref-max-history r 20) ; reset the max history count
```

Minimum/maximum history limits are proportional to the length of the staleness window of the data. It also depends on the relative latency difference of the update and read operations to see what the range of the min-history and the max-history works well on a given host system. It may take some amount of trial and error to get the range right. As a ballpark figure, read operations only need as many min-history elements to avoid the transaction retries, as many updates can go through during one read operation. The max-history elements can be a multiple of min-history to cover for any history overrun or underrun. If the relative latency difference is unpredictable, then we have to either plan a min-history for the worst case scenario, or consider other approaches.

## Transaction retries and barging

A transaction can internally be in one of the five distinct states—Running, Committing, Retry, Killed, and Committed. A transaction can be killed for various reasons. Exceptions are the common reasons for killing a transaction. But let's consider the corner case where a transaction is retried many times, but it does not appear to commit successfully—what is the resolution? Clojure supports age-based barging, wherein an older transaction automatically tries to abort a younger transaction, so that the younger transaction is retried later. If the barging still doesn't work, as a last resort, the transaction is killed after a hard limit of 10,000 retry attempts, and then the exception is thrown.

# Upping transaction consistency with ensure

Clojure's transactional consistency is a good balance between performance and safety. However, at times, we may need the **Serializable** consistency in order to preserve the correctness of the transaction. Concretely, in the face of the transaction retries, when a transaction's correctness depends on the state of a ref, in the transaction, wherein the ref is updated simultaneously in another transaction, we have a condition called "write skew". The Wikipedia entry on the write skew,

[https://en.wikipedia.org/wiki/Snapshot\\_isolation](https://en.wikipedia.org/wiki/Snapshot_isolation), describes it well, but let's see a more concrete example. Let's say we want to design a flight simulation system with two engines, and one of the system level constraints is not to switch off both engines at the same time. If we model each engine as a ref, and certain maneuvers do require us to switch off an engine, we must ensure that the other engine is on. We can do it with `ensure`. Usually, `ensure` is required when maintaining a consistent relationship (invariants) across the refs is necessary. This cannot be ensured by the validator functions, because they do not come into play until the transaction commits. The validator functions will see the same value hence cannot help.

The write-skew can be solved using the namesake `ensure` function that essentially prevents a ref from modification by other transactions. It is similar to a locking operation, but in practice, it provides better concurrency than the explicit read-and-update operations, when the retries are expensive. Using `ensure` is quite simple—`(ensure ref-object)`. However, it may be performance-wise expensive, due to the locks it holds during the transaction. Managing performance with `ensure` involves a trade-off between the retry latency, and the lost throughput due to the ensured state.

# Lesser transaction retries with commutative operations

Commutative operations are independent of the order in which they are applied. For example, incrementing a counter ref `c1` from transactions `t1` and `t2` would have the same effect irrespective of the order in which `t1` and `t2` commit their changes. Refs have a special optimization for changing functions that are commutative for transactions—the `commute` function, which is similar to `alter` (same syntax), but with different semantics. Like `alter`, the `commute` functions are applied atomically during the transaction commit. However, unlike `alter`, `commute` does not cause the transaction retry on contention, and there is no guarantee about the order in which the `commute` functions are applied. This effectively makes `commute` nearly useless for returning a meaningful value as a result of the operation. All the `commute` functions in a transaction are reapplied with the final in transaction ref values during the transaction commit.

As we can see, `commute` reduces the contention, thereby optimizing the performance of the overall transaction throughput. Once we know that an operation is commutative and we are not going to use its return value in a meaningful way, there is hardly any trade-off deciding on whether to use it—we should just go ahead and use it. In fact, a program design, with respect to the ref transactions, with `commute` in mind, is not a bad idea.

## Agents can participate in transactions

In the previous section on agents, we discussed how agents work with the queued change functions. Agents can also participate in the ref transactions, thereby making it possible to combine the use of refs and agents in the transactions. However, agents are not included in the "Ref world", hence a transaction scope is not extended till the execution of the change function in an agent. Rather, the transactions only make sure that the changes sent to the agents are queued until the transaction commit happens.

The *Nesting* sub-section, in the earlier section on agents, discusses about a second-layer thread-local queue. This thread-local queue is used during a

transaction to hold the sent changes to an agent until the commit. The thread-local queue does not block the other changes that are being sent to an agent. The out-of-transaction changes are never buffered in the thread-local queue; rather, they are added to the regular queue in the agent.

The participation of agents in the transactions provides an interesting angle of design, where the coordinated and independent/sequential operations can be pipelined as a workflow for better throughput and performance.

## Nested transactions

Clojure transactions are nesting aware and they compose well. But, why would one need a nested transaction? Often, independent units of code may have their own low-granularity transactions that a higher level code can make use of. When the higher level caller itself needs to wrap actions in a transaction, nested transactions occur. Nested transactions have their own lifecycle and run-state. However, an outer transaction can abort an inner transaction on the detection of failure.

The "ref world" snapshot ensures and commutes are shared among all (that is, outer and inner) levels of a nested transaction. Due to this, the inner transaction is treated as any other ref change operation (similar to alter, ref-set and so on) within an outer transaction. The watches and internal lock implementation are handled at the respective nesting level. The detection of contention in the inner transactions causes a restart of not only the inner but also the outer transaction. Commits at all the levels are effected as a global state finally when the outermost transaction commits. The watches, even though tracked at each individual transaction level, are finally effected during the commit. A closer look at the nested transaction implementation shows that nesting has little or no impact on the performance of transactions.

## Performance considerations

Clojure Ref is likely to be the most complex reference type implemented

yet. Due to its characteristics, especially its transaction retry mechanism, it may not be immediately apparent that such a system would have good performance during the high-contention scenarios.

Understanding its nuances and best ways of use should help:

- We do not use changes with the side effects in a transaction, except for possibly sending the I/O changes to agents, where the changes are buffered until the commit. So by definition, we do not carry out any expensive I/O work in a transaction. Hence, a retry of this work would be cheap as well.
- A change function for a transaction should be as small as possible. This lowers the latency and hence, the retries will also be cheaper.
- Any ref that is not updated along with at least one more ref simultaneously needs not be a ref—atoms would do just fine in this case. Now that the refs make sense only in a group, their contention is directly proportional to the group size. Small groups of refs used in the transactions lead to a low contention, lower latency, and a higher throughput.
- Commutative functions provide a good opportunity to enhance the transaction throughput without any penalty. Identifying such cases and designing with commute in mind can help performance significantly.
- Refs are very coarse grained—they work at the application aggregate level. Often a program may need to have more fine-grained control over the transaction resources. This can be enabled by Ref striping, such as Megaref (<https://github.com/cgrand/megaref>), by providing a scoped view on the associative refs, thereby allowing higher concurrency.
- In the high contention scenarios in which the ref group size in a transaction cannot be small, consider using agents, as they have no contention due to the serial nature. Agents may not be a replacement for the transactions, but rather we can employ a pipeline consisting of atoms, refs, and agents to ease out the contention versus latency concerns.

Refs and transactions have an intricate implementation. Fortunately, we can inspect the source code, and browse through available online and offline resources.

# Dynamic var binding and state

The fourth kind among the Clojure's reference types is the dynamic var. Since Clojure 1.3, all the vars are static by default. A var must be explicitly declared so in order to be dynamic. Once declared, a dynamic var can be bound to new values on per-thread basis. Binding on different threads do not block each other. An example is shown here:

```
(def ^:dynamic *foo* "bar")
(println *foo*) ; prints bar
(binding [*foo* "baz"] (println *foo*)) ; prints baz
(binding [*foo* "bar"] (set! *foo* "quux") (println *foo*)) ;
prints quux
```

As the dynamic binding is thread-local, it may be tricky to use in multi-threaded scenarios. Dynamic vars have been long abused by libraries and applications as a means to pass in a common argument to be used by several functions. However, this style is acknowledged to be an anti-pattern, and is discouraged. Typically, in the anti-pattern dynamic, vars are wrapped by a macro to contain the dynamic thread-local binding in the lexical scope. This causes problems with the multi-threading and lazy sequences.

So, how can the dynamic vars be used effectively? A dynamic var lookup is more expensive than looking up a static var. Even passing a function argument is performance-wise much cheaper than looking up a dynamic var. Binding a dynamic var incurs additional cost. Clearly, in performance sensitive code, dynamic vars are best not used at all. However, dynamic vars may prove to be useful to hold a temporary thread-local state in a complex, or recursive call-graph scenario, where the performance does not matter significantly, without being advertised or leaked into the public API. The dynamic var bindings can nest and unwind like a stack, which makes them both attractive and suitable for such tasks.

# Validating and watching the reference types

Vars (both static and dynamic), atoms, refs, and agents provide a way to validate the value being set as state—a `validator` function that accepts new value as argument, and returns the logical as true if it succeeds, or throws exception/returns logical as false (the `false` and `nil` values) if there's an error. They all honor what the validator function returns. If it is a success, the update goes through, and if an error, an exception is thrown instead. Here is the syntax on how the validators can be declared and associated with the reference types:

```
(def t (atom 1 :validator pos?))
(def g (agent 1 :validator pos?))
(def r (ref 1 :validator pos?))
(swap! t inc) ; goes through, because value after increment (2)
is positive
(swap! t (constantly -3)) ; throws exception
(def v 10)
(set-validator! (var v) pos?)
(set-validator! t (partial < 10)) ; throws exception
(set-validator! g (partial < 10)) ; throws exception
(set-validator! r #(< % 10)) ; works
```

Validators cause actual failure within a reference type while updating them. For vars and atoms, they simply prevent the update by throwing an exception. In an agent, a validation failure causes agent failure, and needs the agent to restart. Inside a ref, the validation failure causes the transaction to rollback and rethrow the exception.

Another mechanism to observe the changes to the reference types is a "watcher". Unlike validators, a watcher is passive—it is notified of the update after the fact. Hence, a watcher cannot prevent updates from going through, because it is only a notification mechanism. For transactions, a watcher is invoked only after the transaction commit. While only one

validator can be set on a reference type, it is possible to associate multiple watchers to a reference type on the other hand. Secondly, when adding a watch, we can specify a key, so that the notifications can be identified by the key, and be dealt accordingly by the watcher. Here is the syntax on how to use watchers:

```
(def t (atom 1))
(defn w [key iref oldv newv] (println "Key:" key "Old:" oldv
                                         "New:" newv))
(add-watch t :foo w)
(swap! t inc) ; prints "Key: :foo Old: 1 New: 2"
```

Like validators, the watchers are executed synchronously in the thread of the reference type. For atoms and refs, this may be fine, since the notification to the watchers goes on, the other threads may proceed with their updates. However in agents, the notification happens in the same thread where the update happens—this makes the update latency higher, and the throughput potentially lower.

# Java concurrent data structures

Java has a number of mutable data structures that are meant for concurrency and thread-safety, which implies that multiple callers can safely access these data structures at the same time, without blocking each other. When we need only the highly concurrent access without the state management, these data structures may be a very good fit. Several of these employ lock free algorithms. We discussed about the Java atomic state classes in the *Atomic updates and state section*, so we will not repeat them here. Rather, we will only discuss the concurrent queues and other collections.

All of these data structures live in the `java.util.concurrent` package. These concurrent data structures are tailored to leverage the JSR 133 "Java Memory Model and Thread Specification Revision" (<http://gee.cs.oswego.edu/dl/jmm/cookbook.html>) implementation that first appeared in Java 5.

## Concurrent maps

Java has a mutable concurrent hash map

—`java.util.concurrent.ConcurrentHashMap` (CHM in short). The concurrency level can be optionally specified when instantiating the class, which is 16 by default. The CHM implementation internally partitions the map entries into the hash buckets, and uses multiple locks to reduce the contention on each bucket. Reads are never blocked by writes, therefore they may be stale or inconsistent—this is countered by built-in detection of such situations, and issuing a lock in order to read the data again in the synchronized fashion. This is an optimization for the scenarios, where reads significantly outnumber writes. In CHM, all the individual operations are near constant-time unless stuck in a retry loop due to the lock contention.

In contrast with Clojure's persistent map, CHM cannot accept `null` (`nil`) as the key or value. Clojure's immutable scalars and collections are automatically well-suited for use with CHM. An important thing to note is that only the individual operations in CHM are atomic, and exhibit strong consistency. As CHM operations are concurrent, the aggregate operations provide a rather weak consistency than the true operation-level consistency. Here is how we can use CHM. The individual operations in CHM, which provide a better consistency, are safe to use. The aggregate operations should be reserved for when we know its consistency characteristics, and the related trade-off:

```
(import 'java.util.concurrent.ConcurrentHashMap)
(def ^ConcurrentHashMap m (ConcurrentHashMap.))
(.put m :english "hi")                      ; individual operation
(.get m :english)                           ; individual
operation
(.putIfAbsent m :spanish "alo")      ; individual operation
(.replace m :spanish "hola")        ; individual operation
(.replace m :english "hi" "hello")   ; individual compare-and-swap
atomic operation
(.remove m :english)                  ; individual operation
(.clear m)                         ; aggregate operation
(.size m)                          ; aggregate operation
(count m)                          ; internally uses the .size() method
;; aggregate operation
(.putAll m {:french "bonjour" :italian "buon giorno"})
(.keySet m)                        ; aggregate operation
(keys m)                           ; calls CHM.entrySet() and on each pair
java.util.Map.Entry.getKey()
(vals m)                           ; calls CHM.entrySet() and on each pair
java.util.Map.Entry.getValue()
```

The `java.util.concurrent.ConcurrentSkipListMap` class (CSLM in short) is another concurrent mutable map data structure in Java. The difference between CHM and CSLM is that CSLM offers a sorted view of the map at all times with the  $O(\log N)$  time complexity. The sorted view has the natural order of keys by default, which can be overridden by specifying a Comparator implementation when instantiating CSLM. The implementation of CSLM is based on the Skip List, and provides

navigation operations.

The `java.util.concurrent.ConcurrentSkipListSet` class (CSLS in short) is a concurrent mutable set based on the CSLM implementation. While CSLM offers the map API, CSLS behaves as a set data structure while borrowing features of CSLM.

## Concurrent queues

Java has a built-in implementation of several kinds of mutable and concurrent in-memory queues. The queue data structure is a useful tool for buffering, producer-consumer style implementation, and for pipelining such units together to form the high-performance workflows. We should not confuse them with durable queues that are used for similar purpose in the batch jobs for a high throughput. Java's in-memory queues are not transactional, but they provide atomicity and strong consistency guarantee for the individual queue operations only. Aggregate operations offer weaker consistency.

The `java.util.concurrent.ConcurrentLinkedQueue` (CLQ) is a lock-free, wait-free unbounded "First In First Out" (FIFO) queue. FIFO implies that the order of the queue elements will not change once added to the queue. CLQ's `size()` method is not a constant time operation; it depends on the concurrency level. Few examples of using CLQ are here:

```
(import 'java.util.concurrent.ConcurrentLinkedQueue)
(def ^ConcurrentLinkedQueue q (ConcurrentLinkedQueue.))
(.add q :foo)
(.add q :bar)
(.poll q) ; returns :foo
(.poll q) ; returns :bar
```

Queue	Blocking?	Bounded?	FIFO?	Fairness?	Notes
CLQ	No	No	Yes	No	Wait-free, but the size() is not constant time

ABQ	Yes	Yes	Yes	Optional	The capacity is fixed at instantiation
DQ	Yes	No	No	No	The elements implement the Delayed interface
LBQ	Yes	Optional	Yes	No	The capacity is flexible, but with no fairness option
PBQ	Yes	No	No	No	The elements are consumed in a priority order
SQ	Yes	—	—	Optional	It has no capacity; it serves as a channel

In the `java.util.concurrent` package, `ArrayBlockingQueue` (ABQ), `DelayQueue` (DQ), `LinkedBlockingQueue` (LBQ), `PriorityBlockingQueue` (PBQ), and `SynchronousQueue` (SQ) implement the `BlockingQueue` (BQ) interface. Its Javadoc describes the characteristics of its method calls. ABQ is a fixed-capacity, FIFO queue backed by an array. LBQ is also a FIFO queue, backed by the linked nodes, and is optionally bounded (default `Integer.MAX_VALUE`). ABQ and LBQ generate "Back pressure" by blocking the enqueue operations on full capacity. ABQ supports optional fairness (with performance overhead) in the order of the threads that access it.

DQ is an unbounded queue that accepts the elements associated with the delay. The queue elements cannot be null, and must implement the `java.util.concurrent.Delayed` interface. Elements are available for removal from the queue only after the delay has been expired. DQ can be very useful for scheduling the processing of the elements at different times.

PBQ is unbounded and blocking while letting elements be consumed from the queue as per priority. Elements have the natural ordering by default that can be overridden by specifying a `Comparator` implementation when instantiating the queue.

SQ is not really a queue at all. Rather, it's just a barrier for a producer or consumer thread. The producer blocks until a consumer removes the

element and vice versa. SQ does not have a capacity. However, SQ supports optional fairness (with performance overhead), in the order, in which the threads access it.

There are some new concurrent queue types introduced after Java 5. Since JDK 1.6, in the `java.util.concurrent` package Java has **BlockingDeque (BD)** with **LinkedBlockingDeque (LBD)** as the only available implementation. BD builds on BQ by adding the **Deque (double-ended queue)** operations, that is, the ability to add elements and consume the elements from both the ends of the queue. LBD can be instantiated with an optional capacity (bounded) to block the overflow. JDK 1.7 introduced **TransferQueue (TQ)** with **LinkedTransferQueue (LTQ)** as the only implementation. TQ extends the concept of SQ in such a way that the producers and consumers block a queue of elements. This will help utilize the producer and consumer threads better by keeping them busy. LTQ is an unbounded implementation of TQ where the `size()` method is not a constant time operation.

## Clojure support for concurrent queues

We covered the persistent queue in [Chapter 2, Clojure Abstractions](#) earlier. Clojure has a built-in `seqe` function that builds over a BQ implementation (LBQ by default) to expose a write-ahead sequence. The sequence is potentially lazy, and the write-ahead buffer throttles how many elements to realize. As opposed to the chunked sequences (of chunk size 32), the size of the write-ahead buffer is controllable and potentially populated at all times until the source sequence is exhausted. Unlike the chunked sequences, the realization doesn't happen suddenly for a chunk of 32 elements. It does so gradually and smoothly.

Under the hood, Clojure's `seqe` uses an agent to the backfill data in the write-ahead buffer. In the arity-2 variant of `seqe`, the first argument should either be a positive integer, or an instance of BQ (ABQ, LBQ, and more) that is preferably bounded.

# Concurrency with threads

On the JVM, threads are the de-facto fundamental instrument of concurrency. Multiple threads live in the same JVM; they share the heap space, and compete for the resources.

## JVM support for threads

The JVM threads are the Operating System threads. Java wraps an underlying OS thread as an instance of the `java.lang.Thread` class, and builds up an API around it to work with threads. A thread on the JVM has a number of states: New, Runnable, Blocked, Waiting, Timed\_Waiting, and Terminated. A thread is instantiated by overriding the `run()` method of the `Thread` class, or by passing an instance of the `java.lang.Runnable` interface to the constructor of the `Thread` class.

Invoking the `start()` method of a `Thread` instance starts its execution in a new thread. Even if just a single thread runs in the JVM, the JVM would not shut down. Calling the `setDaemon(boolean)` method of a thread with argument `true` tags the thread as a daemon that can be automatically shut down if no other non-daemon thread is running.

All Clojure functions implement the `java.lang.Runnable` interface. Therefore, invoking a function in a new thread is very easy:

```
(defn foo5 [] (dotimes [_ 5] (println "Foo")))
(defn barN [n] (dotimes [_ n] (println "Bar")))
(.start (Thread. foo5)) ; prints "Foo" 5 times
(.start (Thread. (partial barN 3))) ; prints "Bar" 3 times
```

The `run()` method does not accept any argument. We can work around it by creating a higher order function that needs no arguments, but internally applies the argument 3.

## Thread pools in the JVM

Creating threads leads to the Operating System API calls, which is not always a cheap operation. The general practice is to create a pool of threads that can be recycled for different tasks. Java has a built-in support for threads pools. The interface called

`java.util.concurrent.ExecutorService` represents the API for a thread pool. The most common way to create a thread pool is to use a factory method in the `java.util.concurrent.Executors` class:

```
(import 'java.util.concurrent.Executors)
(import 'java.util.concurrent.ExecutorService)
(def ^ExecutorService a (Executors/newSingleThreadExecutor)) ; bounded pool
(def ^ExecutorService b (Executors/newCachedThreadPool)) ; unbounded pool
(def ^ExecutorService c (Executors/newFixedThreadPool 5)) ; bounded pool
(.execute b #(dotimes [_ 5] (println "Foo")))) ; prints "Foo" 5 times
```

The previous example is equivalent of the examples with raw threads that we saw in the previous sub-section. Thread pools are also capable of helping to track the completion, and the return value of a function, executed in a new thread. An `ExecutorService` accepts an instance of the `java.util.concurrent.Callable` instance as an argument to several methods that launch a task, and return `java.util.concurrent.Future` to track the final result.

All the Clojure functions also implement the `Callable` interface, so we can use them as follows:

```
(import 'java.util.concurrent.Callable)
(import 'java.util.concurrent.Future)
(def ^ExecutorService e (Executors/newSingleThreadExecutor))
(def ^Future f (.submit e (cast Callable #(reduce + (range 10000000)))))
(.get f) ; blocks until result is processed, then returns it
```

The thread pools described here are the same as the ones that we saw

briefly in the Agents section earlier. Thread pools need to be shut down by calling the `shutdown()` method when no longer required.

## Clojure concurrency support

Clojure has some nifty built-in features to deal with concurrency. We already discussed about the agents, and how they use the thread pools, in an earlier section. There are some more concurrency features in Clojure to deal with the various use cases.

### Future

We saw earlier in this section how to use the Java API to launch a new thread, to execute a function. Also, we learned how to get the result back. Clojure has a built-in support called "futures" to do these things in a much smoother and integrated manner. The basis of the futures is the function `future-call` (it takes a no-arg function as an argument), and the macro `future` (it takes the body of code) that builds on the former. Both of them immediately start a thread to execute the supplied code. The following snippet illustrates the functions that work with the future, and how to use them:

```
; ; runs body in new thread
(def f (future (println "Calculating") (reduce + (range 1e7))))
(def g (future-call #(do (println "Calculating") (reduce + (range
1e7)))) ; takes no-arg fn
(future? f) ; returns true
(future-cancel g) ; cancels execution unless already over
(can stop mid-way)
(future-cancelled? g) ; returns true if canceled due to request
(future-done? f) ; returns true if terminated
successfully, or canceled
(realized? f) ; same as future-done? for futures
@f ; blocks if computation not yet
over (use deref for timeout)
```

One of the interesting aspects of `future-cancel` is that it can sometimes not only cancel tasks that haven't started yet, but may also abort those that

are halfway through execution:

```
(let [f (future (println "[f] Before sleep")
                 (Thread/sleep 2000)
                 (println "[f] After sleep")
                 2000)]
    (Thread/sleep 1000)
    (future-cancel f)
    (future-cancelled? f))
;; [f] Before sleep ← printed message (second message is never
printed)
;; true ← returned value (due to future-cancelled?)
```

The previous scenario happens because Clojure's `future-cancel` cancels a future in such a way that if the execution has already started, it may be interrupted causing `InterruptedException`, which, if not explicitly caught, would simply abort the block of code. Beware of exceptions arising from the code that is executed in a future, because, by default, they are not reported verbosely! Clojure futures use the "solo" thread pool (used to execute the potentially blocking actions) that we discussed earlier with respect to the agents.

## Promise

A promise is a placeholder for the result of a computation that may or may not have occurred. A promise is not directly associated with any computation. By definition, a promise does not imply when the computation might occur, hence realizing the promise.

Typically, a promise originates from one place in the code, and is realized by some other portion of the code that knows when and how to realize the promise. Very often, this happens in a multi-threaded code. If a promise is not realized yet, any attempt to read the value blocks all callers. If a promise is realized, then all the callers can read the value without being blocked. As with futures, a promise can be read with a timeout using `deref`.

Here is a very simple example showing how to use promises:

```
(def p (promise))  
(realized? p) ; returns false  
@p ; at this point, this will block until another thread  
delivers the promise  
(deliver p :foo)  
@p ; returns :foo (for timeout use deref)
```

A promise is a very powerful tool that can be passed around as function arguments. It can be stored in a reference type, or simply be used for a high level coordination.

# Clojure parallelization and the JVM

We observed in [Chapter 1](#), *Performance by Design* that parallelism is a function of the hardware, whereas concurrency is a function of the software, assisted by the hardware support. Except for the algorithms that are purely sequential by nature, concurrency is the favored means to facilitate parallelism, and achieve better performance. Immutable and stateless data is a catalyst to concurrency, as there is no contention between threads, due to absence of mutable data.

## Moore's law

In 1965, Intel's cofounder, Gordon Moore, made an observation that the number of transistors per square inch on Integrated Circuits doubles every 24 months. He also predicted that the trend would continue for 10 years, but in practice, it has continued till now, marking almost half a century. More transistors have resulted in more computing power. With a greater number of transistors in the same area, we need higher clock speed to transmit signals to all of the transistors. Secondly, transistors need to get smaller in size to fit in. Around 2006-2007, the clock speed that the circuitry could work with topped out at about 2.8GHz, due to the heating issues and the laws of physics. Then, the multi-core processors were born.

## Amdahl's law

The multi-core processors naturally require splitting up computation in order to achieve parallelization. Here begins a conflict—a program that was made to be run sequentially cannot make use of the parallelization features of the multi-core processors. The program must be altered to find the opportunity to split up computation at every step, while keeping the cost of coordination in mind. This results in a limitation that a program can

be no more faster than its longest sequential part (*contention*, or *seriality*), and the coordination overhead. This characteristic was described by Amdahl's law.

## Universal Scalability Law

Dr Neil Gunther's Universal Scalability Law (USL) is a superset of Amdahl's Law that makes both: *contention* ( $\alpha$ ) and *coherency* ( $\beta$ ) the first class concerns in quantifying the scalability very closely to the realistic parallel systems. Coherency implies the coordination overhead (latency) in making the result of one part of a parallelized program to be available to another. While Amdahl's Law states that contention (seriality) causes performance to level off, USL goes to show that the performance actually degrades with excessive parallelization. USL is described with the following formula:

$$C(N) = N / (1 + \alpha ((N - 1) + \beta N (N - 1)))$$

Here,  $C(N)$  implies relative capacity or throughput in terms of the source of concurrency, such as physical processors, or the users driving the software application.  $\alpha$  implies the degree of contention because of the shared data or the sequential code, and  $\beta$  implies penalty incurred for maintaining the consistency of shared data. I would encourage you to pursue USL further

(<http://www.perfdynamics.com/Manifesto/USLscalability.html>), as this is a very important resource for studying the impact of concurrency on scalability and the performance of the systems.

## Clojure support for parallelization

A program that relies on mutation cannot parallelize its parts without creating contention on the mutable state. It requires coordination overhead, which makes the situation worse. Clojure's immutable nature is better suited to parallelize the parts of a program. Clojure also has some constructs that are suited for parallelism by the virtue of Clojure's

consideration of available hardware resources. The result is, the operations execute optimized for certain use case scenarios.

## pmap

The `pmap` function (similar to `map`) accepts as arguments a function and one, or more collections of data elements. The function is applied to each of the data elements in such a way that some of the elements are processed by the function in parallel. The parallelism factor is chosen at runtime by the `pmap` implementation, as two greater than the total number of available processors. It still processes the elements lazily, but the realization factor is same as the parallelism factor.

Check out the following code:

```
(pmap (partial reduce +)
      [ (range 1000000)
        (range 1000001 2000000)
        (range 2000001 3000000) ])
```

To use `pmap` effectively, it is imperative that we understand what it is meant for. As the documentation says, it is meant for computationally intensive functions. It is optimized for CPU-bound and cache-bound jobs. High latency and low CPU tasks, such as blocking I/O, are a gross misfit for `pmap`. Another pitfall to be aware of is whether the function used in `pmap` performs a lot of memory operations or not. Since the same function will be applied across all the threads, all the processors (or cores) may compete for the memory interconnect and the sub-system bandwidth. If the parallel memory access becomes a bottleneck, `pmap` cannot make the operation truly parallel, due to the contention on the memory access.

Another concern is what happens when several `pmap` operations run concurrently? Clojure does not attempt to detect multiple `pmaps` running concurrently. The same number of threads will be launched afresh for every new `pmap` operation. The developer is responsible to ensure the performance characteristics, and the response time of the program

resulting from the concurrent pmap executions. Usually, when the latency reasons are paramount, it is advisable to limit the concurrent instances of pmap running in the program.

## pcalls

The pcalls function is built using pmap, so it borrows properties from the latter. However, the pcalls function accepts zero or more functions as arguments and executes them in parallel, returning the result values of the calls as a list.

## pvalues

The pvalues macro is built using pcalls, so it transitively shares the properties of pmap. Its behavior is similar to pcalls, but instead of functions, it accepts zero or more S-expressions that are evaluated in the parallel using pmap.

# Java 7's fork/join framework

Java 7 introduced a new framework for parallelism called "fork/join," based on divide-and-conquer and the work-stealing scheduler algorithms. The basic idea of how to use the fork/join framework is fairly simple—if the work is small enough, then do it directly in the same thread; otherwise, split the work into two pieces, invoke them in a fork/join thread pool, and wait for the results to combine.

This way, the job gets recursively split into smaller parts such as an inverted tree, until the smallest part can be carried out in just a single thread. When the leaf/subtree jobs return, the parent combines the result of all children, and returns the results.

The fork/join framework is implemented in Java 7 in terms of a special kind of thread pool; check out `java.util.concurrent.ForkJoinPool`. The specialty of this thread pool is that it accepts the jobs of `java.util.concurrent.ForkJoinTask` type, and whenever these jobs

block, waiting for the child jobs to finish, the threads used by the waiting jobs are allocated to the child jobs. When the child finishes its work, the thread is allocated back to the blocked parent jobs in order to continue. This style of dynamic thread allocation is described as "work-stealing". The fork/join framework can be used from within Clojure. The `ForkJoinTask` interface has two implementations: `RecursiveAction` and `RecursiveTask` in the `java.util.concurrent` package. Concretely, `RecursiveTask` maybe more useful with Clojure, as `RecursiveAction` is designed to work with mutable data, and does not return any value from its operation.

Using the fork-join framework entails choosing the batch size to split a job into, which is a crucial factor in parallelizing a long job. Too large a batch size may not utilize all the CPU cores enough; on the other hand, a small batch size may lead to a longer overhead, coordinating across the parent/child batches. As we will see in the next section, Clojure integrates with the Fork/join framework to parallelize the reducers implementation.

# Parallelism with reducers

Reducers are a new abstraction introduced in Clojure 1.5, and are likely to have a wider impact on the rest of the Clojure implementation in the future versions. They depict a different way of thinking about processing collections in Clojure—the key concept is to break down the notion that collections can be processed only sequentially, lazily, or producing a seq, and more. Moving away from such a behavior guarantee raises the potential for eager and parallel operations on one hand, whereas incurring constraints on the other. Reducers are compatible with the existing collections.

For an example, a keen observation of the regular `map` function reveals that its classic definition is tied to the mechanism (recursion), order (sequential), laziness (often), and representation (list/seq/other) aspects of producing the result. Most of this actually defines "how" the operation is performed, rather than "what" needs to be done. In the case of `map`, the "what" is all about applying a function to each element of its collection arguments. But since the collection types can be of various types (tree-structured, sequence, iterator, and more), the operating function cannot know how to navigate the collection. Reducers decouple the "what" and "how" parts of the operation.

## Reducible, reducer function, reduction transformation

Collections are of various kinds, hence only a collection knows how to navigate itself. In the reducers model at a fundamental level, an internal "reduce" operation in each collection type has access to its properties and behavior, and access to what it returns. This makes all the collection types essentially "reducible". All the operations that work with collections can be modeled in terms of the internal "reduce" operation. The new modeled form of such operations is a "reducing function", which is typically a

function of two arguments, the first argument being the accumulator, and the second being the new input.

How does it work when we need to layer several functions upon another, over the elements of a collection? For an example, let's say first we need to "filter", "map," and then "reduce". In such cases, a "transformation function" is used to model a reducer function (for example, for "filter") as another reducer function (for "map") in such a way that it adds the functionality during the transformation. This is called "reduction transformation".

## Realizing reducible collections

While the reducer functions retain the purity of the abstraction, they are not useful all by themselves. The reducer operations in the namespace called as `clojure.core.reducers` similar to `map`, `filter`, and more, basically return a reducible collection that embed the reducer functions within themselves. A reducible collection is not realized, not even lazily realized—rather, it is just a recipe that is ready to be realized. In order to realize a reducible collection, we must use one of the `reduce` or `fold` operations.

The `reduce` operation that realizes a reducible collection is strictly sequential, albeit with the performance gains compared to `clojure.core/reduce`, due to reduced object allocations on the heap. The `fold` operation, which realizes a reducible collection, is potentially parallel, and uses a "reduce-combine" approach over the fork-join framework. Unlike the traditional "map-reduce" style, the use of fork/join the reduce-combine approach reduces at the bottom, and subsequently combines by the means of reduction again. This makes the `fold` implementation less wasteful and better performing.

## Foldable collections and parallelism

Parallel reduction by `fold` puts certain constraints on the collections and

operations. The tree-based collection types (persistent map, persistent vector, and persistent set) are amenable to parallelization. At the same time, the sequences may not be parallelized by `fold`. Secondly, `fold` requires that the individual reducer functions should be "associative", that is, the order of the input arguments applied to the reducer function should not matter. The reason being, `fold` can segment the elements of the collection to process in parallel, and the order in which they may be combined is not known in advance.

The `fold` function accepts few extra arguments, such as the "combine function," and the partition batch size (default being 512) for the parallel processing. Choosing the optimum partition size depends on the jobs, host capabilities, and the performance benchmarking. There are certain functions that are foldable (that is, parallelizable by `fold`), and there are others that are not, as shown here. They live in the `clojure.core.reducers` namespace:

- **Foldable:** `map`, `mapcat`, `filter`, `remove`, and `flatten`
- **Non-foldable:** `take-while`, `take`, and `drop`
- **Combine functions:** `cat`, `foldcat`, and `monoid`

A notable aspect of reducers is that it is foldable in parallel only when the collection is a tree type. This implies that the entire data set must be loaded in the heap memory when folding over them. This has the downside of memory consumption during the high load on a system. On the other hand, a lazy sequence is a perfectly reasonable solution for such scenarios. When processing large amount of data, it may make sense to use a combination of lazy sequences and reducers for performance.

# Summary

Concurrency and parallelism are extremely important for performance in this multi-core age. Effective use of concurrency requires substantial understanding of the underlying principles and details. Fortunately, Clojure provides safe and elegant ways to deal with concurrency and state.

Clojure's new feature called "reducers" provides a way to achieve granular parallelism. In the coming years, we are likely to see more and more processor cores, and an increasing demand to write code that takes advantage of these. Clojure places us in the right spot to meet such challenges.

In the next chapter, we will look at the performance measurement, analysis, and monitoring.

# Chapter 6. Measuring Performance

Depending on the expected and actual performance, and the lack or presence of a measuring system, performance analysis and tuning can be a fairly elaborate process. Now we will discuss the analysis of performance characteristics and ways to measure and monitor them. In this chapter we will cover the following topics:

- Measuring performance and understanding the results
- What performance tests to carry out for different purposes
- Monitoring performance and obtaining metrics
- Profiling Clojure code to identify performance bottlenecks

## Performance measurement and statistics

Measuring performance is the stepping stone to performance analysis. As we noted earlier in this book, there are several performance parameters to be measured with respect to various scenarios. Clojure's built-in `time` macro is a tool to measure the amount of time elapsed while executing a body of code. Measuring performance factors is a much more involved process. The measured performance numbers may have linkages with each other that we need to analyze. It is a common practice to use statistical concepts to establish the linkage factors. We will discuss some basic statistical concepts in this section and use that to explain how the measured data gives us the bigger picture.

### A tiny statistics terminology primer

When we have a series of quantitative data, such as latency in milliseconds for the same operation (measured over a number of executions), we can

observe a number of things. First, and the most obvious, are the minimum and maximum values in the data. Let's take an example dataset to analyze further:

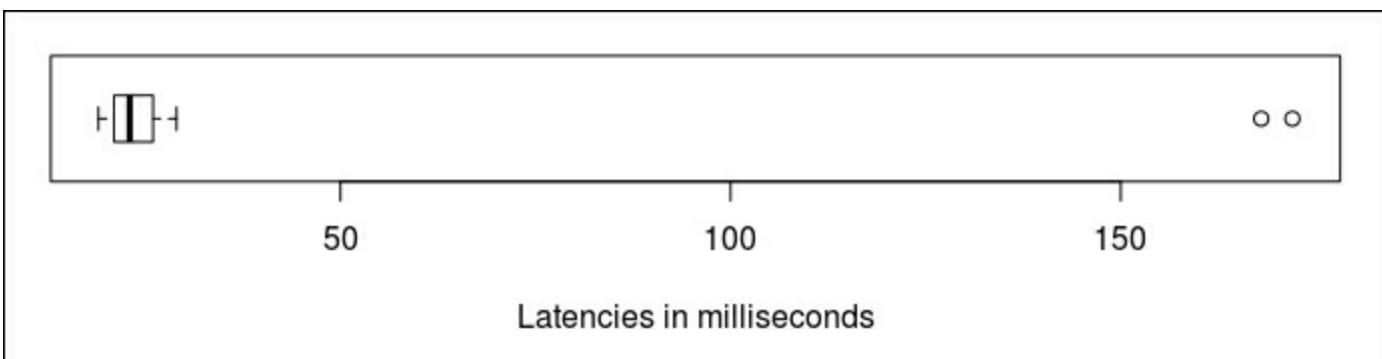
23	19	21	24	26	20	22	21	25	168	23	20	29	172	22	24	26
----	----	----	----	----	----	----	----	----	-----	----	----	----	-----	----	----	----

## Median, first quartile, third quartile

We can see that the minimum latency here is 19 ms whereas the maximum latency is 172ms. We can also observe that the average latency here is about 40ms. Let's sort this data in ascending order:

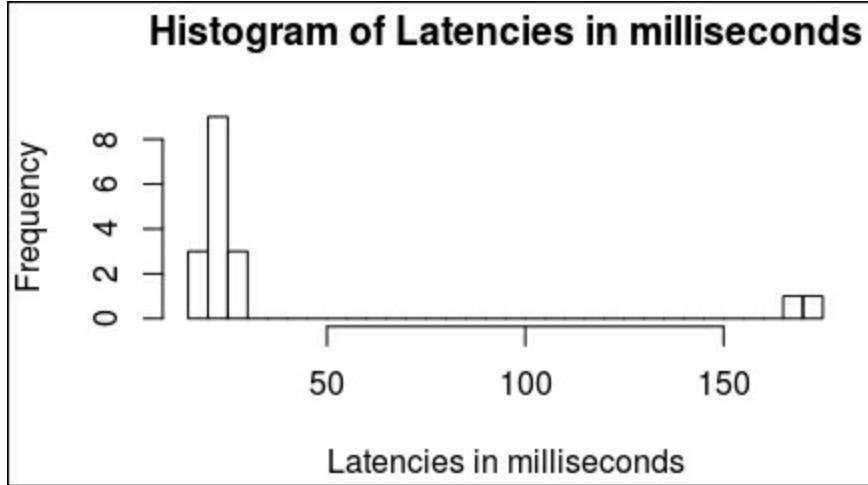
19	20	20	21	21	22	22	23	23	24	24	25	26	26	29	168	172
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	-----	-----

The center element of the previous dataset, that is the ninth element (value 23), is considered the **median** of the dataset. It is noteworthy that the median is a better representative of the center of the data than the **average** or **mean**. The center element of the left half, that is the fifth element (value 21), is considered the **first quartile**. Similarly, the value in the center of the right half, that is the 13th element (value 26), is considered the **third quartile** of the dataset. The difference between the third quartile and the first quartile is called **Inter Quartile Range (IQR)**, which is 5 in this case. This can be illustrated with a **boxplot** , as follows:



A boxplot highlights the first quartile, median and the third quartile of a

dataset. As you can see, two "outlier" latency numbers (168 and 172) are unusually higher than the others. Median makes no representation of outliers in a dataset, whereas the mean does.



A histogram (the diagram shown previously) is another way to display a dataset where we batch the data elements in **periods** and expose the **frequency** of such periods. A period contains the elements in a certain range. All periods in a histogram are generally the same size; however, it is not uncommon to omit certain periods when there is no data.

## Percentile

A **percentile** is expressed with a parameter, such as 99 percentile, or 95 percentile etc. A percentile is the value below which all the specified percentage of data elements exist. For example, 95 percentile means the value  $N$  among a dataset, such that 95 percent of elements in the dataset are below  $N$  in value. As a concrete example, 85 percentile from the dataset of latency numbers we discussed earlier in this section is 29, because out of 17 total elements, 14 (which is 85 percent of 17) other elements in the dataset have a value below 29. A quartile splits a dataset into chunks of 25 percent elements each. Therefore, the first quartile is actually 25 percentile, the median is 50 percentile, and the third quartile is 75 percentile.

# Variance and standard deviation

The spread of the data, that is, how far away the data elements are from the center value, gives us further insight into the data. Consider the  $i^{th}$  deviation as the difference between the  $i^{th}$  dataset element value (in statistics terms, a "variable" value) and its mean; we can represent it as  $x_i - \bar{x}$ . We can express its "variance" and "standard deviation" as follows:

$$\text{Variance} = \frac{\sum_{i=1}^a (x_i - \bar{x})^2}{n-1}, \text{ standard deviation } (\sigma) = \sqrt{\frac{\sum_{i=1}^a (x_i - \bar{x})^2}{n-1}}$$

Standard deviation is shown as the Greek letter "sigma", or simply "s". Consider the following Clojure code to determine variance and standard deviation:

```
(def tdata [23 19 21 24 26 20 22 21 25 168 23 20 29 172 22 24 26])

(defn var-std-dev
  "Return variance and standard deviation in a vector"
  [data]
  (let [size (count data)
        mean (/ (reduce + data) size)
        sum (->> data
                  (map #(let [v (- % mean)] (* v v))))
        (reduce +))
        variance (double (/ sum (dec size)))]
    [variance (Math/sqrt variance)]))

user=> (println (var-std-dev tdata))
[2390.345588235294 48.89116063497873]
```

You can use the Clojure-based platform Incanter (<http://incanter.org/>) for statistical computations. For example, you can find standard deviation using (incanter.stats/sd tdata) in Incanter.

The **empirical rule** states the relationship between the elements of a dataset and SD. It says that 68.3 percent of all elements in a dataset lie in the range of one (positive or negative) SD from the mean, 95.5 percent of all elements lie in two SDs from the mean, and 99.7 percent of all data elements lie in three SDs from the mean.

Looking at the latency dataset we started out with, one SD from the mean is  $40 \pm 49$  ( $40 \pm 98$  range -9 to 89) containing 88 percent of all elements. Two SDs from the mean is  $40 \pm 49$  range -58 to 138) containing the same 88 percent of all elements. However, three SDs from the mean is ( $40 \pm 147$  range -107 to 187) containing 100 percent of all elements. There is a mismatch between what the empirical rule states and the results we found, because the empirical rule applies generally to uniformly distributed datasets with a large number of elements.

## Understanding Criterium output

In [Chapter 4, Host Performance](#), we introduced the Clojure library *Criterium* to measure the latency of Clojure expressions. A sample benchmarking result is as follows:

```
user=> (require '[criterium.core :refer [bench]])
nil
user=> (bench (reduce + (range 1000)))
Evaluation count : 162600 in 60 samples of 2710 calls.
          Execution time mean : 376.756518 us
          Execution time std-deviation : 3.083305 us
          Execution time lower quantile : 373.021354 us ( 2.5%)
          Execution time upper quantile : 381.687904 us (97.5%)

Found 3 outliers in 60 samples (5.0000 %)
low-severe 2 (3.3333 %)
low-mild 1 (1.6667 %)
Variance from outliers : 1.6389 % Variance is slightly inflated
by outliers
```

We can see that the result has some familiar terms we discussed earlier in

this section. A high mean and low standard deviation indicate that there is not a lot of variation in the execution times. Likewise, the lower (first) and upper (third) quartiles indicate that they are not too far away from the mean. This result implies that the body of code is more or less stable in terms of response time. Criterium repeats the execution many times to collect the latency numbers.

However, why does Criterium attempt to do a statistical analysis of the execution time? What would be amiss if we simply calculate the mean? It turns out that the response times of all executions are not always stable and there is often disparity in how the response time shows up. Only upon running sufficient times under correctly simulated load we can get complete data and other indicators about the latency. A statistical analysis gives insight into whether there is something wrong with the benchmark.

## Guided performance objectives

We only briefly discussed performance objectives in [Chapter 1](#), *Performance by Design* because that discussion needs a reference to statistical concepts. Let's say we identified the functional scenarios and the corresponding response time. Should response time remain fixed? Can we constrain throughput in order to prefer a stipulated response time?

The performance objective should specify the worst-case response time, that is, maximum latency, the average response time and the maximum standard deviation. Similarly, the performance objective should also mention the worst-case throughput, maintenance window throughput, average throughput, and the maximum standard deviation.

# Performance testing

Testing for performance requires us to know what we are going to test, how we want to test, and what environment to set up for the tests to execute. There are several pitfalls to be aware of, such as a lack of near-real hardware and resources of production use, similar OS and software environments, diversity of representative data for test cases, and so on. Lack of diversity in test inputs may lead to a monotonic branch prediction, hence introducing a "bias" in test results.

## The test environment

Concerns about the test environment begin with the hardware representative of the production environment. Traditionally, the test environment hardware has been a scaled-down version of the production environment. The performance analysis done on non-representative hardware is almost certain to skew the results. Fortunately, in recent times, thanks to the commodity hardware and cloud systems, provisioning test environment hardware that is similar to the production environment is not too difficult.

The network and storage bandwidth, operating system, and software used for performance testing should of course be the same as in production. What is also important is to have a "load" representative of the test scenarios. The load comes in different combinations including the concurrency of requests, the throughput and standard deviation of requests, the current population level in the database or in the message queue, CPU and heap usage, and so on. It is important to simulate a representative load.

Testing often requires quite some work on the part of the piece of code that carries out the test. Be sure to keep its overhead to a minimum so that it does not impact the benchmark results. Wherever possible, use a system

other than the test target to generate requests.

## What to test

Any implementation of a non-trivial system typically involves many hardware and software components. Performance testing a certain feature or a service in the entire system needs to account for the way it interacts with the various sub-systems. For example, a web service call may touch multiple layers such as the web server (request/response marshaling and unmarshaling), URI-based routing, service handler, application-database connector, the database layer, logger component, and so on. Testing only the service handler would be a terrible mistake, because that is not exactly the performance what the web client will experience. The performance test should test at the perimeter of a system to keep the results realistic, preferably with a third-party observer.

The performance objectives state the criteria for testing. It would be useful to test what is not required by the objective, especially when the tests are run concurrently. Running meaningful performance tests may require a certain level of isolation.

## Measuring latency

The latency obtained by executing a body of code may vary slightly on each run. This necessitates that we execute the code many times and collect samples. The latency numbers may be impacted by the JVM warm-up time, garbage collection and the JIT compiler kicking in. So, the test and sample collection should ensure that these conditions do not impact the results. Criterium follows such methods to produce the results. When we test a very small piece of code this way, it is called a **Micro-benchmark**.

As the latency of some operations may vary during runs, it is important to collect all samples and segregate them into periods and frequencies turning up into a histogram. The maximum latency is an important metric when

measuring latency—it indicates the worst-case latency. Besides the maximum, the 99 percentile and 95 percentile latency numbers are also important to put things in perspective. It's important to actually collect the latency numbers instead of inferring them from standard deviation, as we noted earlier that the empirical rule works only for normal distributions without significant outliers.

The outliers are an important data point when measuring latency. A proportionately higher count of outliers indicates a possibility of degradation of service.

## Comparative latency measurement

When evaluating libraries for use in projects, or when coming up with alternate solutions against some baseline, comparative latency benchmarks are useful to determine the performance trade-offs. We will inspect two comparative benchmarking tools based on Criterium, called Perforate and Citius. Both make it easy to run Criterium benchmarks grouped by context, and to easily view the benchmark results.

Perforate (<https://github.com/davidsantiago/perforate>) is a Leiningen plugin that lets one define goals; a goal (defined using `perforate.core/defgoal`) is a common task or context having one or more benchmarks. Each benchmark is defined using `perforate.core/defcase`. As of version 0.3.4, a sample benchmark code may look like the following code snippet:

```
(ns foo.bench
  (:require [perforate.core :as p]))

(p/defgoal str-concat "String concat")
(p/defcase str-concat :apply
  [] (apply str ["foo" "bar" "baz"]))
(p/defcase str-concat :reduce
  [] (reduce str ["foo" "bar" "baz"]))

(p/defgoal sum-numbers "Sum numbers")
```

```
(p/defcase sum-numbers :apply
[] (apply + [1 2 3 4 5 6 7 8 9 0]))
(p/defcase sum-numbers :reduce
[] (reduce + [1 2 3 4 5 6 7 8 9 0]))
```

You can declare the test environments in `project.clj` and provide the setup/cleanup code when defining the goal. Perforate provides ways to run the benchmarks from the command-line.

Citius (<https://github.com/kumarshantanu/citius>) is a library that provides integration hooks for `clojure.test` and other forms of invocation. It imposes more rigid constraints than Perforate, and renders additional comparative information about the benchmarks. It presumes a fixed number of targets (cases) per test suite where there may be several goals.

As of version 0.2.1, a sample benchmark code may look like the following code snippet:

```
(ns foo.bench
  (:require [citius.core :as c]))

(c/with-bench-context ["Apply" "Reduce"]
  {:chart-title "Apply vs Reduce"
   :chart-filename "bench-simple.png"})
(c/compare-perf
  "concat strs"
  (apply str ["foo" "bar" "baz"]))
  (reduce str ["foo" "bar" "baz"]))
(c/compare-perf
  "sum numbers"
  (apply + [1 2 3 4 5 6 7 8 9 0]))
  (reduce + [1 2 3 4 5 6 7 8 9 0])))
```

In the previous example, the code runs the benchmarks, reports the comparative summary, and draws a bar chart image of the mean latencies.

## Latency measurement under concurrency

When we benchmark a piece of code with Criterium, it uses just a single

thread to determine results. That gives us a fair output in terms of single-threaded result, but there are many benchmarking scenarios where single-threaded latency is far from what we need. Under concurrency, the latency often differs quite a bit from single-threaded latency. Especially when we deal with *stateful* objects (e.g. drawing a connection from a JDBC connection pool, updating shared in-memory state etc.), the latency is likely to vary in proportion with the contention. In such scenarios it is useful to find out the latency patterns of the code under various concurrency levels.

The Citius library we discussed in the previous sub-section supports tunable concurrency levels. Consider the following benchmark of implementations of shared counters:

```
(ns foo.bench
  (:require
    [clojure.test :refer [deftest]]
    [citius.core :as c])
  (:import [java.util.concurrent.atomic AtomicLong]))

(def a (atom 0))
(def ^AtomicLong b (AtomicLong. 0))

(deftest test-counter
  (c/with-bench-context ["Atom" "AtomicLong"] {}
    (c/compare-perf "counter"
      (swap! a unchecked-inc) (.incrementAndGet b)))))

;; Under Unix-like systems run the following command in terminal:
;; CITIUS_CONCURRENCY=4,4 lein test
```

When I ran this benchmark on a 4th generation quad-core Intel Core i7 processor (Mac OSX 10.10), the mean latency at concurrency level 04 was 38 to 42 times the value of the mean latency at concurrency level 01. Since, in many cases, the JVM is used to run server-side applications, benchmarking under concurrency becomes all the more important.

## Measuring throughput

Throughput is expressed per unit of time. Coarse-grained throughput, that is, the throughput number collected over a long period of time, may hide the fact when the throughput is actually delivered in bursts instead of a uniform distribution. Granularity of the throughput test is subject to the nature of the operation. A batch process may process bursts of data, whereas a web service may deliver uniformly distributed throughput.

## Average throughput test

Though Citius (as of version 0.2.1) shows extrapolated throughput (per second, per thread) in benchmark results, that throughput number may not represent the actual throughput very well for a variety of reasons. Let's construct a simple throughput benchmark harness as follows, beginning with the helper functions:

```
(import '[java.util.concurrent ExecutorService Executors Future])
(defn concurrently
  ([n f]
   (concurrently n f #(mapv deref %)))
  ([n f g]
   (let [^ExecutorService
         thread-pool (Executors/newFixedThreadPool n)
         future-vals (transient [])]
     (dotimes [i n]
       (let [^Callable task (if (coll? f) (nth f i) f)
             ^Future each-future (.submit thread-pool task)]
         (conj! future-vals each-future)))
     (try
      (g (persistent! future-vals))
      (finally
        (.shutdown thread-pool))))))
  (try
   (g (persistent! future-vals))
   (finally
     (.shutdown thread-pool)))))

(defn call-count
  []
  (let [stats (atom 0)]
    (fn
      ([] (deref stats))
      ([k] (if (identical? :reset k)
              (reset! stats 0)
              (swap! stats unchecked-inc))))))
```

```
(defn wrap-call-stats
  [stats f]
  (fn [& args]
    (try
      (let [result (apply f args)]
        (stats :count)
        result)))))

(defn wait-until-millis
  ([^long timeout-millis]
   (wait-until-millis timeout-millis 100))
  ([^long timeout-millis ^long progress-millis]
   (while (< (System/currentTimeMillis) timeout-millis)
     (let [millis (min progress-millis
                        (- timeout-millis
                           (System/currentTimeMillis)))]
       (when (pos? millis)
         (try
           (Thread/sleep millis)
           (catch InterruptedException e
             (.interrupt ^Thread (Thread/currentThread)))))))
   (print \.)
   (flush))))))
```

Now that we have the helper functions defined, let's see the benchmarking code:

```

call-count (->> (fn [future-vals]
                     (print "\nWarming up")
                     (wait-until-millis (+ (now) warmup-
millis)))
                     (mapv #(reset) stats-coll) ; reset
count
                     (print "\nBenchmarking")
                     (wait-until-millis (+ (now) bench-
millis)))
                     (println)
                     (swap! exit? not)
                     (mapv deref future-vals))
                     (concurrently concurrency g-coll)
                     (apply +))]

{:concurrency concurrency
 :calls-count call-count
 :duration-millis bench-millis
 :calls-per-second (->> (/ bench-millis 1000)
                           double
                           (/ ^long call-count)
                           long)))}

(defmacro benchmark-throughput
  "Benchmark a body of code for average throughput."
  [concurrency warmup-millis bench-millis & body]
  `(benchmark-throughput*
    ~concurrency ~warmup-millis ~bench-millis (fn [] ~@body)))

```

Let's now see how to test some code for throughput using the harness:

```

(def a (atom 0))
(println
(benchmark-throughput
  4 20000 40000 (swap! a inc)))

```

This harness provides only a simple throughput test. To inspect the throughput pattern you may want to bucket the throughput across rolling fixed-duration windows (e.g. per second throughput.) However, that topic is beyond the scope of this text, though we will touch upon it in the *Performance monitoring* section later in this chapter.

# The load, stress, and endurance tests

One of the characteristics of tests is each run only represents the slice of time it is executed through. Repeated runs establish their general behavior. But how many runs should be enough? There may be several anticipated load scenarios for an operation. So, there is a need to repeat the tests at various load scenarios. Simple test runs may not always exhibit the long-term behavior and response of the operation. Running the tests under varying high load for longer duration allows us to observe them for any odd behavior that may not show up in a short test cycle.

When we test an operation at a load far beyond its anticipated latency and throughput objectives, that is **stress testing**. The intent of a stress test is to ascertain a reasonable behavior exhibited by the operation beyond the maximum load it was developed for. Another way to observe the behavior of an operation is to see how it behaves when run for a very long duration, typically for several days or weeks. Such prolonged tests are called **endurance tests**. While a stress test checks the graceful behavior of the operation, an endurance test checks the consistent behavior of the operation over a long period.

There are several tools that may help with load and stress testing. Engulf (<http://engulf-project.org/>) is a distributed HTTP-based, load-generation tool written in Clojure. Apache JMeter and Grinder are Java-based load-generation tools. Grinder can be scripted using Clojure. Apache Bench is a load-testing tool for web systems. Tsung is an extensible, high-performance, load-testing tool written in Erlang.

# Performance monitoring

During prolonged testing or after the application has gone to production, we need to monitor its performance to make sure the application continues to meet the performance objectives. There may be infrastructure or operational issues impacting the performance or availability of the application, or occasional spikes in latency or dips in throughput. Generally, monitoring alleviates such risk by generating a continuous feedback stream.

Roughly there are three kinds of components used to build a monitoring stack. A **collector** sends the numbers from each host that needs to be monitored. The collector gets host information and the performance numbers and sends them to an **aggregator**. An aggregator receives the data sent by the collector and persists them until asked by a **visualizer** on behalf of the user.

The project **metrics-clojure** (<https://github.com/sjl/metrics-clojure>) is a Clojure wrapper over the **Metrics** (<https://github.com/dropwizard/metrics>) Java framework, which acts as a collector. **Statsd** is a well-known aggregator that does not persist data by itself but passes it on to a variety of servers. One of the popular visualizer projects is **Graphite** that stores the data as well as produces graphs for requested periods. There are several other alternatives to these, notably **Riemann** (<http://riemann.io/>) that is written in Clojure and Ruby. Riemann is an event processing-based aggregator.

## Monitoring through logs

One of the popular performance monitoring approaches that has emerged in recent times is via logs. The idea is simple—the application emits metrics data as logs, which are shipped from the individual machine to a central log aggregation service. Then, those metrics data are aggregated for

each time window and further moved for archival and visualization.

As a high-level example of such a monitoring system, you may like to use **Logstash-forwarder** (<https://github.com/elastic/logstash-forwarder>) to grab the application logs from the local filesystem and ship to **Logstash** (<https://www.elastic.co/products/logstash>), where it forwards the metrics logs to **StatsD** (<https://github.com/etsy/statsd>) for metrics aggregation or to **Riemann** (<http://riemann.io/>) for events analysis and monitoring alerts. StatsD and/or Riemann can forward the metrics data to Graphite (<http://graphite.wikidot.com/>) for archival and graphing of the time-series metrics data. Often, people want to plug in a non-default time-series data store (such as **InfluxDB**: <https://influxdb.com/>) or a visualization layer (such as **Grafana**: <http://grafana.org/>) with Graphite.

A detailed discussion on this topic is out of the scope of this text, but I think exploring this area would serve you well.

## Ring (web) monitoring

If you develop web software using Ring (<https://github.com/ring-clojure/ring>) then you may find the Ring extension of the metrics-clojure library useful: <http://metrics-clojure.readthedocs.org/en/latest/ring.html> — this tracks a number of useful metrics that can be queried in JSON format and integrated with visualization via the web browser.

To emit a continuous stream of metrics data from the web layer, **Server-Sent Events (SSE)** may be a good idea due to its low overhead. Both **http-kit** (<http://www.http-kit.org/>) and **Aleph** (<http://aleph.io/>), which work with Ring, support SSE today.

## Introspection

Both Oracle JDK and OpenJDK provide two GUI tools called **JConsole** (executable name `jconsole`) and **JVisualVM** (executable name

jvisualvm) that we can use to introspect into running JVMs for instrumentation data. There are also some command-line tools (<http://docs.oracle.com/javase/8/docs/technotes/tools/>) in the JDK to peek into the inner details of the running JVMs.

A common way to introspect a running Clojure application is to have an **nREPL** (<https://github.com/clojure/tools.nrepl>) service running so that we can connect to it later using an nREPL client. Interactive introspection over nREPL using the Emacs editor (embedded nREPL client) is popular among some, whereas others prefer to script an nREPL client to carry out tasks.

## JVM instrumentation via JMX

The JVM has a built-in mechanism to introspect managed resources via the extensible **Java Management Extensions (JMX)** API. It provides a way for application maintainers to expose manageable resources as "MBeans". Clojure has an easy-to-use contrib library called `java.jmx` (<https://github.com/clojure/java.jmx>) to access JMX. There is a decent amount of open source tooling for visualization of JVM instrumentation data via JMX, such as `jmxtrans` and `jmxetric`, which integrate with Ganglia and Graphite.

Getting quick memory stats of the JVM is pretty easy using Clojure:

```
(let [^Runtime r (Runtime/getRuntime)]
  (println "Maximum memory" (.maxMemory r))
  (println "Total memory" (.totalMemory r))
  (println "Free memory" (.freeMemory r)))
```

Output:

```
Maximum memory 704643072
Total memory 291373056
Free memory 160529752
```

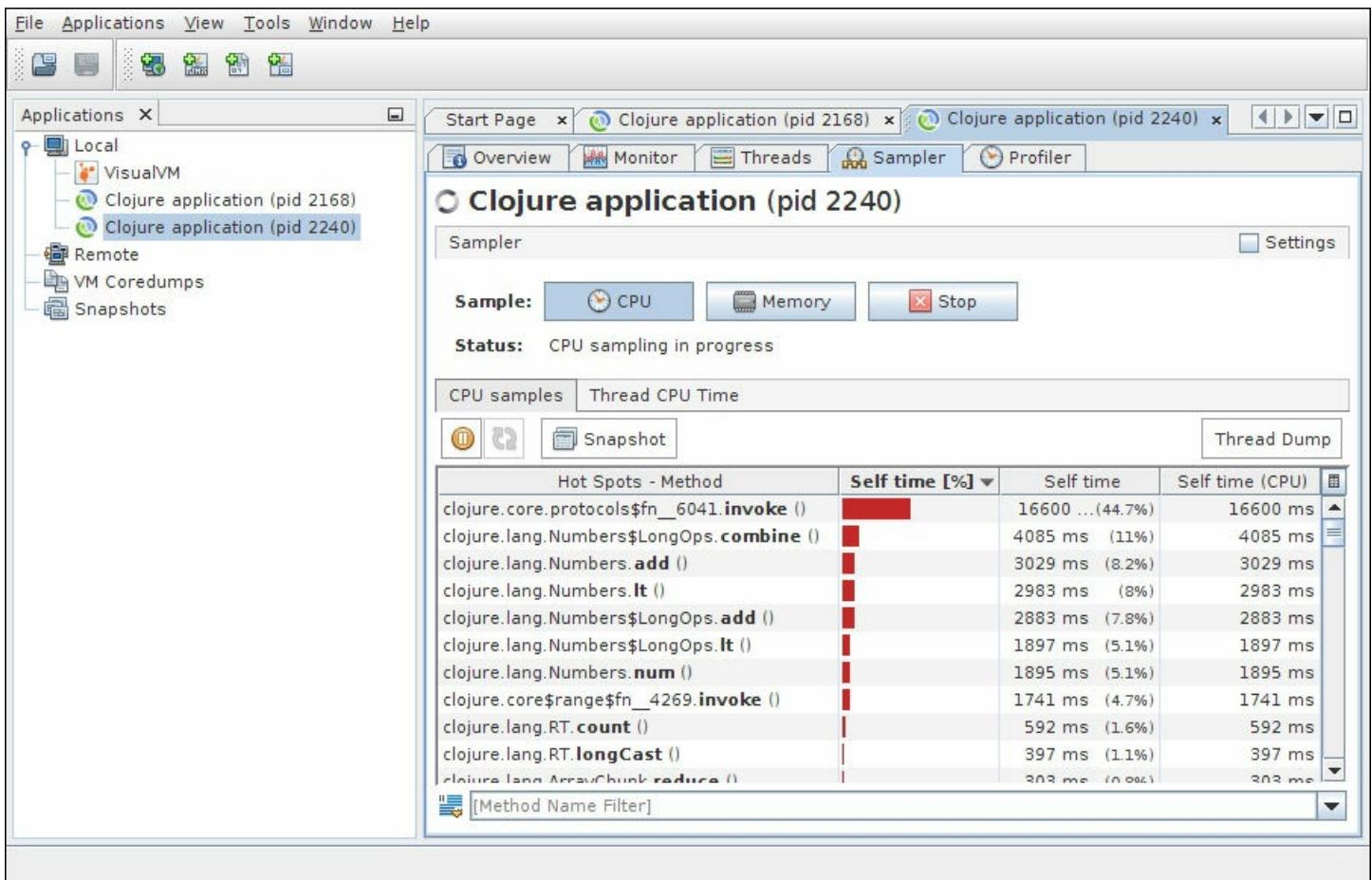
# Profiling

We briefly discussed profiler types in [Chapter 1, Performance by Design](#). The JVisualVM tool we discussed with respect to introspection in the previous section is also a CPU and memory profiler that comes bundled with the JDK. Let's see them in action— consider the following two Clojure functions that stress the CPU and memory respectively:

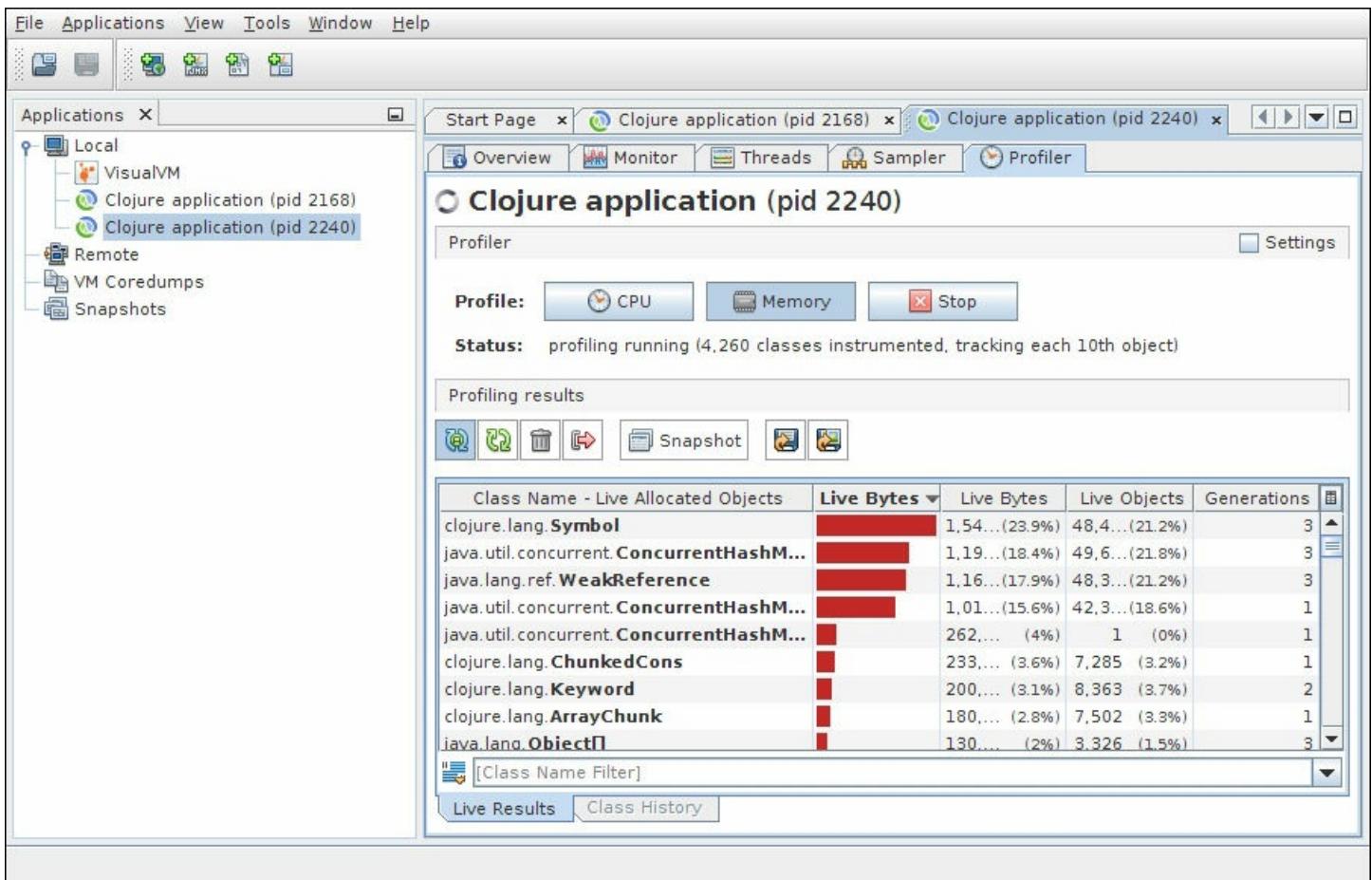
```
(defn cpu-work []
  (reduce + (range 100000000)))

(defn mem-work []
  (->> (range 1000000)
    (map str)
    vec
    (map keyword)
    count))
```

Using JVisualVM is pretty easy—open the Clojure JVM process from the left pane. It has sampler and regular profiler styles of profiling. Start profiling for CPU or memory use when the code is running and wait for it to collect enough data to plot on the screen.



The following shows memory profiling in action:



Note that JVisualVM is a very simple, entry-level profiler. There are several commercial JVM profilers on the market for sophisticated needs.

## OS and CPU/cache-level profiling

Profiling only the JVM may not always tell the whole story. Getting down to OS and hardware-level profiling often provides better insight into what is going on with the application. On Unix-like operating systems, command-line tools such as `top`, `htop`, `perf`, `iota`, `netstat`, `vista`, `upstate`, `pidstat` etc can help. Profiling the CPU for cache misses and other information is a useful source to catch performance issues. Among open source tools for Linux, **Likwid** (<http://code.google.com/p/likwid/> and <https://github.com/rrze-likwid/likwid>) is small yet effective for Intel and AMD processors; **i7z** (<https://code.google.com/p/i7z/> and

<https://github.com/ajaiantilal/i7z>) is specifically for Intel processors. There are also dedicated commercial tools such as **Intel VTune Analyzer** for more elaborate needs.

## I/O profiling

Profiling I/O may require special tools too. Besides `iota` and `blktrace`, `ioping` (<https://code.google.com/p/ioping/>) and <https://github.com/koc9i/ioping>) is useful to measure real-time I/O latency on Linux/Unix systems. The `vnStat` tool is useful to monitor and log network traffic on Linux. The IOPS of a storage device may not tell the whole truth unless it is accompanied by latency information for different operations, and how many reads and writes can simultaneously happen.

In an I/O bound workload one has to look for the read and write IOPS over time and set a threshold to achieve optimum performance. The application should throttle the I/O access so that the threshold is not crossed.

# Summary

Delivering high-performance applications not only requires care for performance but also systematic effort to measure, test, monitor, and optimize the performance of various components and subsystems. These activities often require the right skill and experience. Sometimes, performance considerations may even bring system design and architecture back to the drawing board. Early structured steps taken to achieve performance go a long way to ensuring that the performance objectives are being continuously met.

In the next chapter, we will look into performance optimization tools and techniques.

# Chapter 7. Performance Optimization

Performance optimization is additive by nature, as in it works by adding performance tuning to the knowledge of how the underlying system works, and to the result of performance measurement. This chapter builds on the previous ones that covered "how the underlying system works" and "performance measurement". Though you will notice some recipe-like sections in this chapter, you already know the pre-requisite in order to exploit those well. Performance tuning is an iterative process of measuring performance, determining bottlenecks, applying knowledge in order to experiment with tuning the code, and repeating it all until performance improves. In this chapter, we will cover:

- Setting up projects for better performance
- Identifying performance bottlenecks in the code
- Profiling code with VisualVM
- Performance tuning of Clojure code
- JVM performance tuning

## Project setup

While finding bottlenecks is essential to fixing performance problems in the code, there are several things one can do right from the start to ensure better performance.

## Software versions

Usually, new software versions include bug fixes, new features, and performance improvements. Unless advised to the contrary, it is better to use newer versions. For development with Clojure, consider the following software versions:

- **JVM version:** As of this writing, Java 8 (Oracle JDK, OpenJDK, Zulu) has been released as the latest stable production-ready version. It is not only stable, it also has better performance in several areas (especially concurrency) than the earlier versions. If you have a choice, choose Java 8 over the older versions of Java.
- **Clojure version:** As of this writing, Clojure 1.7.0 is the latest stable version that has several performance improvements over the older versions. There are also new features (transducers, volatile) that can make your code perform better. Choose Clojure 1.7 over the older versions unless you have no choice.

## Leiningen project.clj configuration

As of version 2.5.1, the default Leiningen template (`lein new foo`, `lein new app foo`) needs few tweaks to make the project amenable to performance. Ensure your Leiningen `project.clj` file has the following entries, as appropriate.

### Enable reflection warning

One of the most common pitfalls in Clojure programming is to inadvertently let the code resort to reflection. Recall that we discussed this in [Chapter 3, Leaning on Java. Enabling](#), reflection warning is quite easy, let's fix it by adding the following entry to `project.clj`:

```
:global-vars { *unchecked-math* :warn-on-boxed ; in Clojure 1.7+
               *warn-on-reflection* true }
```

In the previous configuration, the first setting `*unchecked-math* :warn-on-boxed` works only in Clojure 1.7—it emits numeric boxing warnings. The second setting `*warn-on-reflection* true` works on earlier Clojure versions as well as Clojure 1.7, and emits reflection warning messages in the code.

However, including these settings in `project.clj` may not be enough. Reflection warnings are emitted only when a namespace is loaded. You

need to ensure that all namespaces are loaded in order to search for reflection warnings throughout the project. This can be done by writing tests that refer to all namespaces, or via scripts that do so.

## Enable optimized JVM options when benchmarking

In [Chapter 4, Host Performance](#) we discussed that Leiningen enables tiered compilation by default, which provides low startup time at the cost of poor JIT compiler optimization. The default setting is quite misleading for performance benchmarking, so you should enable JVM options that are representative of what you would use in production:

```
:profiles {:perf {:test-paths ^:replace ["perf-test"]
                  :jvm-opts ^:replace ["-server"
                                       "-Xms2048m" "-Xmx2048m"]}}}
```

For example, the previous setting defines a Leiningen profile that overrides the default JVM options to configure a `server` Java runtime with 2 GB of fixed-size heap space. It also sets test paths to a directory `perf-test`. Now you can run performance tests as follows:

```
lein with-profile perf test
```

If your project has performance test suites that require different JVM options, you should define multiple profiles for running tests, as appropriate.

## Distinguish between initialization and runtime

Most non-trivial projects need a lot of context to be set up before they can function. Examples of such contexts could be app configuration, in-memory state, I/O resources, thread pools, caches, and so on. While many projects start with ad hoc configuration and initialization, eventually projects need to isolate the initialization phase from runtime. The purpose of this distinction is not only to sanitize the organization of code, but also to pre-compute as much as possible once before the runtime can take over

to repeatedly respond to demands. This distinction also allows the initialization phase to easily (and conditionally, based on configuration) instrument the initialized code for performance logging and monitoring.

Non-trivial programs are usually divided into layers, such as business logic, caching, messaging, database access, and so on. Each layer has a dependency relationship with one or more of the other layers. It is feasible to carry out the isolation of the initialization phase by writing code using first principles, and many projects actually do that. However, there are a few libraries that simplify this process by letting you declare the dependency relationship between layers. **Component** (<https://github.com/stuarts Sierra/component>) and **Prismatic Graph** (<https://github.com/Prismatic/plumbing>) are notable examples of such libraries.

The Component library is well documented. It may not be easily apparent how to use Prismatic Graph for dependency resolution; following is a contrived example for illustration:

```
(require '[plumbing.core :refer [fnk]])
(require '[plumbing.graph :as g])

(def layers
  { :db      (fnk [config] (let [pool (db-pool config)]
                                (reify IDatabase ...)))
    :cache   (fnk [config db] (let [cache-obj (mk-cache config)]
                                (reify ICache ...)))
    :service (fnk [config db cache] (reify IService ...))
    :web     (fnk [config service] (reify IWeb ...)))}

(defn resolve-layers
  "Return a map of reified layers"
  [app-config]
  (let [compiled (g/compile layers)]
    (compiled {:config app-config})))
```

This example merely shows the construction of a layer dependency graph, but often you may need different construction scope and order for testing.

In that case you may define different graphs and resolve them, as and when appropriate. If you need teardown logic for testing, you can add extra `fnk` entries for each teardown step and use those for teardown.

# Identifying performance bottlenecks

We discussed in previous chapters that random performance tuning of code rarely works, because we may not be tuning in the right place. It is crucial to find the performance bottlenecks before we can tune those areas in the code. Upon finding the bottleneck, we can experiment with alternate solutions around it. In this section we will look into finding the bottlenecks.

## Latency bottlenecks in Clojure code

Latency is the starting, and the most obvious, metric to drill-down in order to find bottlenecks. For Clojure code, we observed in [Chapter 6, Measuring Performance](#) that code profiling tools can help us find the areas of bottleneck. Profilers are, of course, very useful. Once you discover hotspots via profilers, you may find ways to tune those for latency to a certain extent.

Most profilers work on aggregates, a batch of runs, ranking the hotspots in code by resources consumed. However, often the opportunity to tune latency lies in the long tail that may not be highlighted by the profilers. In such circumstances, we may employ a direct drill-down technique. Let's see how to carry out such drill-down using **Espejito** (<https://github.com/kumarshantanu/espejito>), a Clojure library for measuring latency (as of version 0.1.0) across measurement points in single-threaded execution paths. There are two parts of using **Espejito**, both requiring change to your code—one to wrap the code being measured, and the other to report the collected measurement data. The following code illustrates a contrived E-commerce use case of adding an item to a cart:

```
(require '[espejito.core :as e])
```

```

;; in the top level handler (entry point to the use case)
(e/report e/print-table
  ...)

;; in web.clj
(e/measure "web/add-cart-item"
  (biz/add-item-to-cart (resolve-cart request) item-code qty)
  ...)

;; in service.clj (biz)
(defn add-item-to-cart
  [cart item qty]
  (e/measure "biz/add-cart-item"
    (db/save-cart-item (:id cart) (:id item) qty)
    ...))

;; in db.clj (db)
(defn save-cart-item
  [cart-id item-id qty]
  (e/measure "db/save-cart-item"
    ...))

```

Reporting a call is required to be made only once at the outermost (top-level) layer of the code. Measurement calls can be made at any number of places in the call path. Be careful not to put measurement calls inside tight loops, which may shoot memory consumption up. When this execution path is triggered, the functionality works as usual, while the latencies are measured and recorded alongside transparently in memory. The `e/report` call prints a table of recorded metrics. An example output (edited to fit) would be:

	:name	:cumulat	:cumul%	:indiv	:indiv%	:thrown?
web/add-cart-item	11.175ms	100.00%	2.476ms	22.16%		
biz/add-item-to-cart	8.699ms	77.84%	1.705ms	15.26%		
db/save-cart-item	6.994ms	62.59%	6.994ms	62.59%		

Here we can observe that the database call is the most expensive (individual latency), followed by the web layer. Our tuning preference may be guided by the order of expensiveness of the measurement points.

## Measure only when it is hot

One important aspect we did not cover in the drill-down measurement is whether the environment is ready for measurement. The `e/report` call is invoked unconditionally every time, which would not only have its own overhead (table printing), but the JVM may not be warmed up and the JIT compiler may not have kicked in to correctly report the latencies. To ensure that we report only meaningful latencies, let's trigger the `e/report` call on an example condition:

```
(defmacro report-when
  [test & body]
  `(~(if ~test
         (e/report e/print-table
                   ~@body)
         ~@body) )
```

Now, let's assume it is a **Ring**-based (<https://github.com/ring-clojure/ring>) web app and you want to trigger the reporting only when the web request contains a parameter `report` with a value `true`. In that case, your call might look like the following:

```
(report-when (= "true" (get-in request [:params "report"]))
  ....)
```

Condition-based invocation expects the JVM to be up across several calls, so it may not work with command-line apps.

This technique can also be used in performance tests, where non-reporting calls may be made during a certain warm-up period, followed by a reporting call that provides its own reporter function instead of `e/print-table`. You may even write a sampling reporter function that aggregates the samples over a duration and finally reports the latency metrics. Not

only for performance testing, you can use this for latency monitoring where the reporter function logs the metrics instead of printing a table, or sends the latency breakup to a metrics aggregation system.

## Garbage collection bottlenecks

Since Clojure runs on the JVM, one has to be aware of the GC behavior in the application. You can print out the GC details at runtime by specifying the respective JVM options in `project.clj` or on the Java command-line:

```
:jvm-options ^:replace [..other options..  
  "-verbose:gc" "-XX:+PrintGCDetails"  
  "-XX:+PrintGC" "-XX:+PrintGCTimeStamps"  
  ..other options..]
```

This causes a detailed summary of GC events to be printed as the application runs. To capture the output in a file, you can specify the following parameter:

```
:jvm-options ^:replace [..other options..  
  "-verbose:gc" "-XX:+PrintGCDetails"  
  "-XX:+PrintGC" "-XX:+PrintGCTimeStamps"  
  "-Xloggc:./memory.log"  
  ..other options..]
```

It is also useful to see the time between and during full GC events:

```
:jvm-options ^:replace [..other options..  
  "-verbose:gc" "-XX:+PrintGCDetails"  
  "-XX:+PrintGC" "-XX:+PrintGCTimeStamps"  
  "-XX:+PrintGCAfterApplicationStoppedTime"  
  "-XX:+PrintGCAfterApplicationConcurrentTime"  
  ..other options..]
```

The other useful options to troubleshoot GC are as follows:

- `-XX:+HeapDumpOnOutOfMemoryError`
- `-XX:+PrintTenuringDistribution`
- `-XX:+PrintHeapAtGC`

The output of the previous options may help you identify GC bottlenecks that you can try to fix by choosing the right garbage collector, other generational heap options, and code changes. For easy viewing of GC logs, you may like to use GUI tools such as **GCViewer** (<https://github.com/chewiebug/GCViewer>) for this purpose.

## Threads waiting at GC safepoint

When there is a long tight loop (without any I/O operation) in the code, the thread executing it cannot be brought to safepoint if GC happens when the loop ends or goes out of memory (for example, fails to allocate). This may have a disastrous effect of stalling other critical threads during GC. You can identify this category of bottleneck by enabling safepoint logs using the following JVM option:

```
:jvm-options ^:replace [..other options..  
    "-verbose:gc" "-XX:+PrintGCDetails"  
    "-XX:+PrintGC" "-XX:+PrintGCTimeStamps"  
    "-XX:+PrintSafepointStatistics"  
    ..other options..]
```

The safepoint logs emitted by the previous option may help you identify the impact of a tight-loop thread on other threads during GC.

## Using jstat to probe GC details

The Oracle JDK (also OpenJDK, Azul's Zulu) comes with a utility called **jstat** that can be handy to inspect GC details. You can find details on this utility at

<https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jstat.html> —the following examples show how to use it:

```
jstat -gc -t <process-id> 10000  
jstat -gccause -t <process-id> 10000
```

The first command mentioned previously monitors object allocations and freeing in various heap generations, together with other GC statistics, one

in every 10 seconds. The second command also prints the reason for GC, along with other details.

## Inspecting generated bytecode for Clojure source

We discussed in [Chapter 3](#), *Leaning on Java* how to see the generated equivalent Java code for any Clojure code. Sometimes, there may not be a direct correlation between the generated bytecode and Java, which is when inspecting the generated bytecode is very useful. Of course, it requires the reader to know at least a bit about the JVM instruction set (<http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html>). This tool can allow you to very effectively analyze the cost of the generated bytecode instructions.

The project **no.disassemble** (<https://github.com/gtrak/no.disassemble>) is a very useful tool to discover the generated bytecode. Include it in your `project.clj` file as a Leiningen plugin:

```
:plugins [[lein-nodisassemble "0.1.3"]]
```

Then, at the REPL, you can inspect the generated bytecodes one by one:

```
(require '[no.disassemble :as n])
(println (n/disassemble (map inc (range 10)))))
```

The previous snippet prints out the bytecode of the Clojure expression entered there.

## Throughput bottlenecks

The throughput bottlenecks usually arise from shared resources, which could be CPU, cache, memory, mutexes and locks, GC, disk, and other I/O devices. Each of these resources has a different way to find utilization, saturation, and load level. This also heavily depends on the operating

system in use, as it manages the resources. Delving into the OS-specific ways of determining those factors is beyond the scope of this text. However, we will look at profiling some of these for bottlenecks in the next section.

The net effect of throughput shows up as an inverse relationship with latency. This is natural as per Little's law—as we will see in the next chapter. We covered throughput testing and latency testing under concurrency in [Chapter 6, Measuring Performance](#). This should be roughly a good indicator of the throughput trend.

# Profiling code with VisualVM

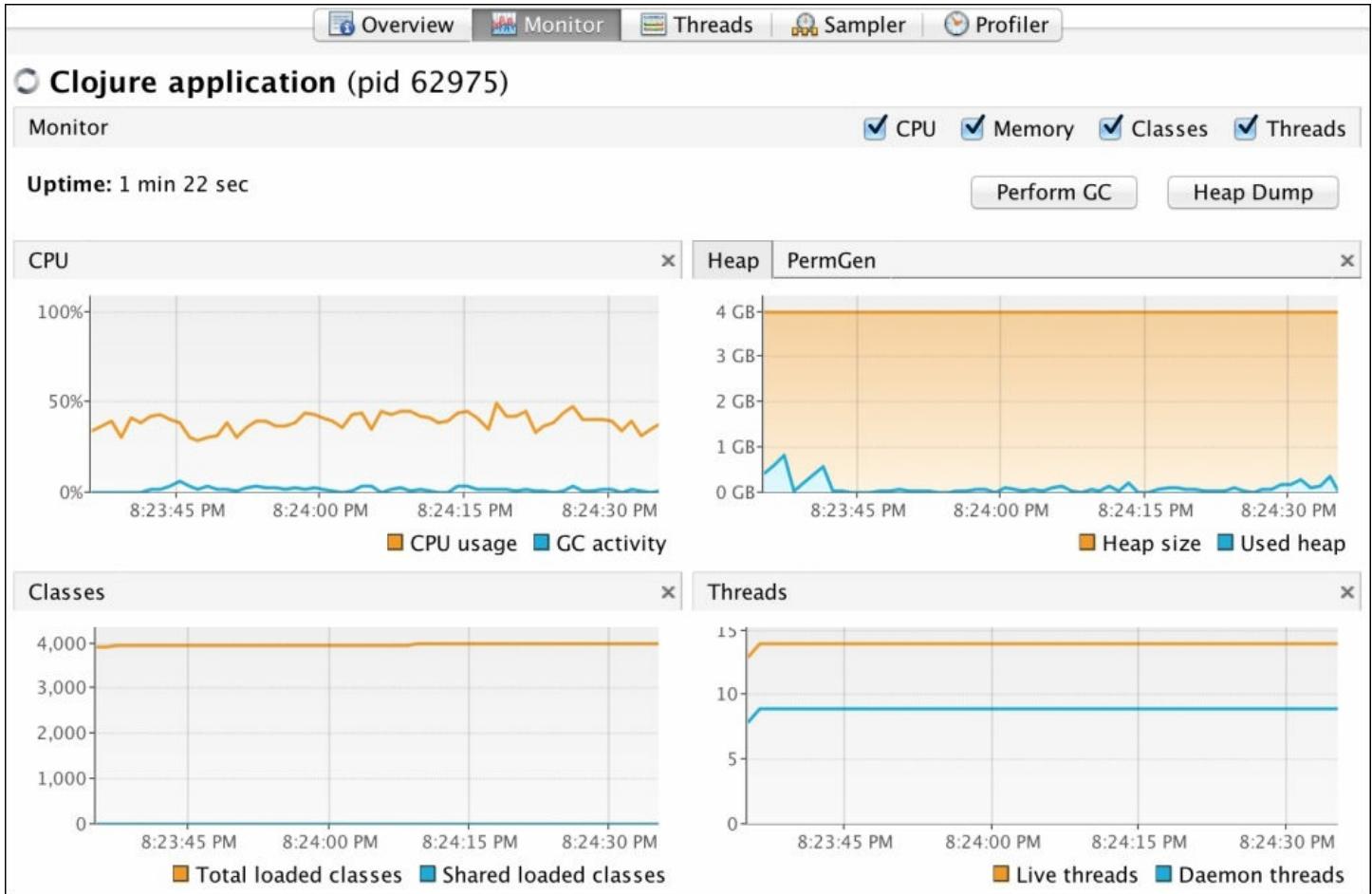
The Oracle JDK (also OpenJDK) comes with a powerful profiler called **VisualVM**; the distribution that comes with the JDK is known as Java VisualVM and can be invoked using the binary executable:

```
jvisualvm
```

This launches the GUI profiler app where you can connect to running instances of the JVM. The profiler has powerful features (<https://visualvm.java.net/features.html>) that can be useful for finding various bottlenecks in code. Besides analyzing heap dump and thread dump, VisualVM can interactively graph CPU and heap consumption, and thread status in near real time. It also has sampling and tracing profilers for both CPU and memory.

# The Monitor tab

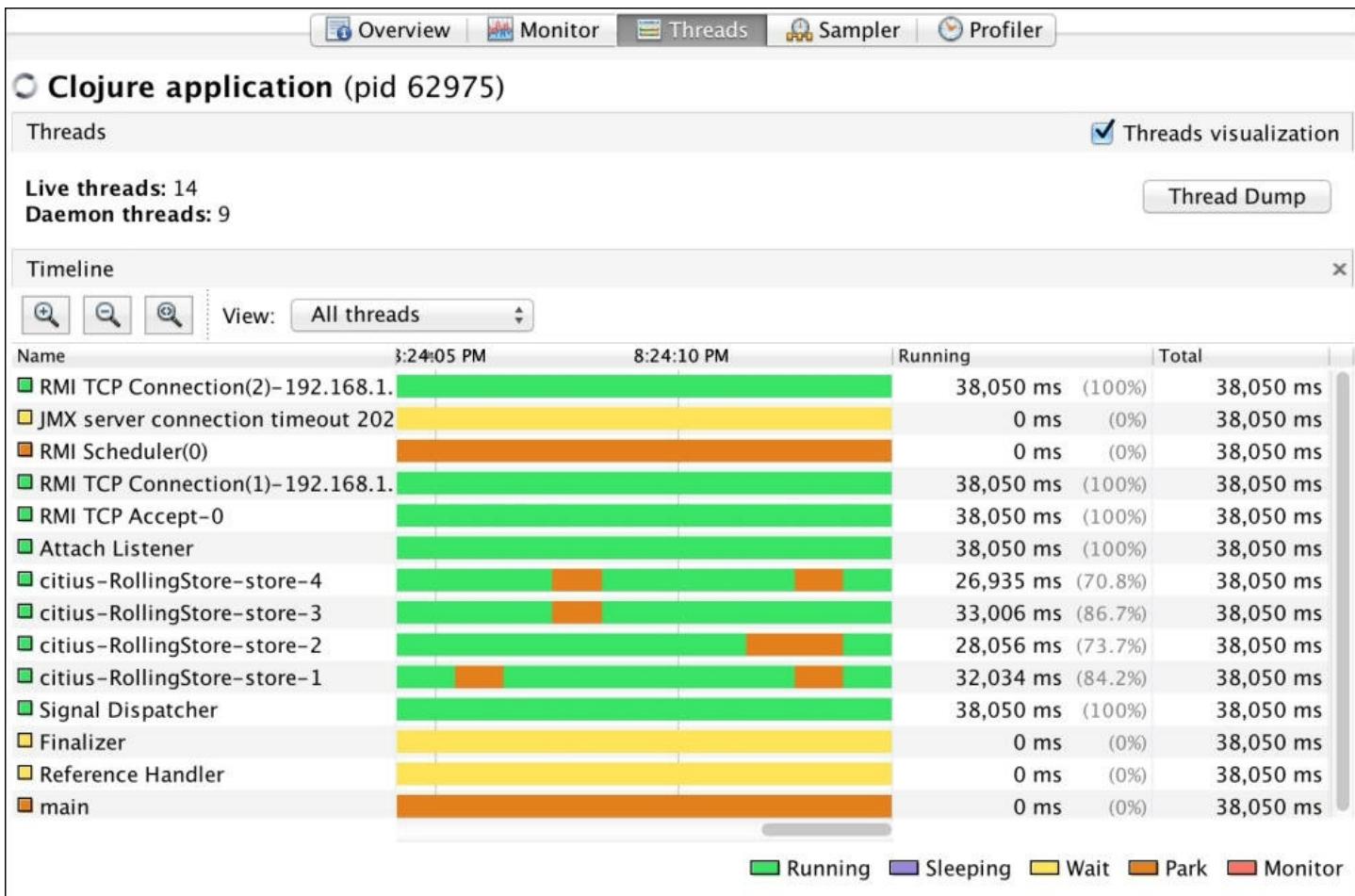
The **Monitor** tab has a graphical overview of the runtime, including CPU, heap, threads and loaded classes:



This tab is useful for "at a glance" information, leaving further drill-down for other tabs.

## The Threads tab

In the following screenshot, the **Threads** tab shows the status of all threads:



It is very useful to find out if any threads are undergoing contention, entering deadlock, are underutilized, or they are taking up more CPU. Especially in concurrent apps with in-memory state, and in apps that use limited I/O resources (such as connection pools, or network calls to other hosts) shared by threads, this feature provides a great insight if you set the thread names:

Notice the threads named **citus-RollingStore-store-1** through **citus-RollingStore-store - 4**. In an ideal no-contention scenario, those threads would have a green **Running** status. See the legend at the bottom right of the image, which explains thread state:

- **Running:** A thread is running, which is the ideal condition.
- **Sleeping:** A thread has yielded control temporarily.

- **Wait:** A thread is waiting for notification in a critical section.  
`Object.wait()` was called, and is now waiting for `Object.notify()` or `Object.notifyAll()` to wake it up.
- **Park:** A thread is parked on a permit (binary semaphore) waiting for some condition. Usually seen with concurrent blocking calls in the `java.util.concurrent` API.
- **Monitor:** A thread has reached object monitor waiting for some lock, perhaps waiting to enter or exit a critical section.

You can install the *Threads Inspector* plugin for details on threads of interest. To inspect thread dumps from the command line you can use the `jstack` or `kill -3` commands.

## The Sampler tab

The **Sampler** tab is the lightweight sampling profiler tab that can sample both CPU and memory consumption. You can easily find hotspots in code that may benefit from tuning. However, sampler profiling is limited by sampling period and frequency, inability to detect inlined code, and so on. It is a good general indicator of the bottlenecks and looks similar to the screenshots we saw in [Chapter 6, Measuring Performance](#). You can profile either CPU or memory at a time.

The **CPU** tab displays both the overall CPU time distribution and per-thread CPU consumption. You can take a thread dump while sampling is in progress and analyze the dump. There are several VisualVM plugins available for more analysis.

The **Memory** tab displays heap histogram metrics with distribution and instance count of objects. It also shows a PermGen histogram and per thread allocation data. It is a very good idea and highly recommended to set thread names in your project so that it is easy to locate those names in such tools. In this tab, you can force a GC, take a heap dump for analysis, and view memory metrics data in several ways.

## Setting the thread name

Setting a thread name in Clojure is quite straightforward using Java interop:

```
(.setName ^Thread (Thread/currentThread) "service-thread-12")
```

However, since threads often transcend several contexts, in most cases you should do so in a limited scope as follows:

```
(defmacro with-thread-name
  "Set current thread name; execute body of code in that
context."
  [new-name & body]
  `(~(let [^Thread thread# (Thread/currentThread)
          ^String t-name# thread#]
       (.setName thread# ~new-name))
    (~@body
     (finally
      (.setName thread# t-name#))))
```

Now you can use this macro to execute any body of code with a specified thread name:

```
(with-thread-name (str "process-order-" order-id)
  ;; business code
  )
```

This style of setting a thread name makes sure that the original name is restored before leaving the thread-local scope. If your code has various sections and you are setting a different thread name for each section, you can detect which code sections are causing contention by looking at the name when any contention appears on profiling and monitoring tools.

## The Profiler tab

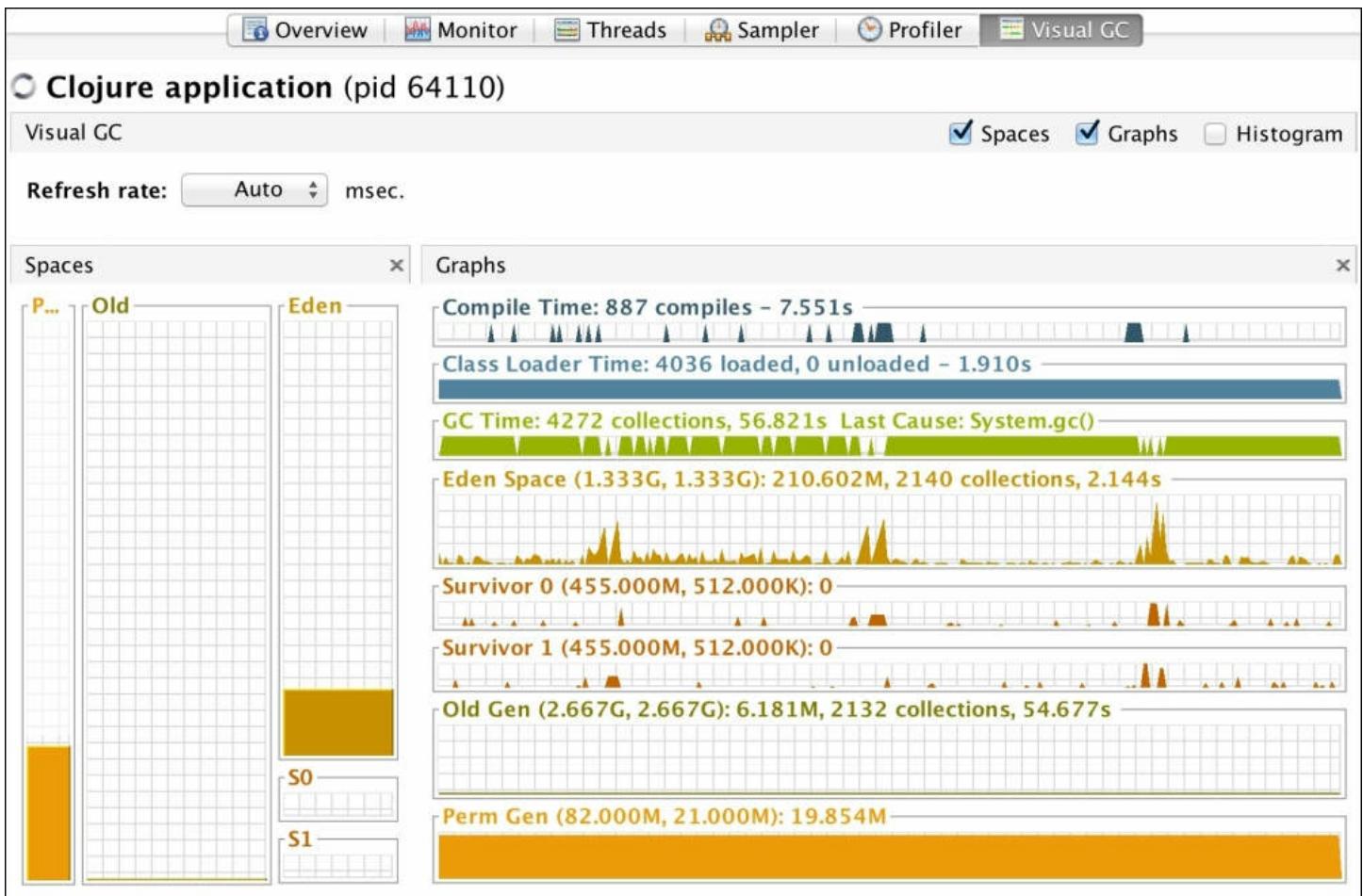
The **Profiler** tab lets you instrument the running code in the JVM, and profile both CPU and memory consumption. This option adds a larger

overhead than the **Sampler** tab, and poses a different trade off in terms of JIT compilation, inlining, and accuracy. This tab does not have as much diversity in visualization as the **Sampler** tab. The main difference this tab has with the **Sampler** tab is it changes the bytecode of the running code for accurate measurement. When you choose CPU profiling, it starts instrumenting the code for CPU profiling. If you switch from CPU to memory profiling, it re-instruments the running code for memory profiling, and re-instruments every time you want a different profiling. One downside of such instrumentation is that it may massively slow down everything if your code is deployed in application containers, such as Tomcat.

While you can get most of the common CPU bottleneck information from **Sampler**, you may need the **Profiler** to investigate hotspots already discovered by **Sampler** and other profiling techniques. You can selectively profile and drill-down only the known bottlenecks using the instrumenting profiler, thereby restricting its ill-effects to only small parts of the code.

## The Visual GC tab

The **Visual GC** is a VisualVM plugin that visually depicts the GC status in near real time.



If your application uses a lot of memory and potentially has GC bottlenecks, this plugin may be very useful for various troubleshooting purposes.

## The Alternate profilers

Besides VisualVM, there are several third-party profilers and performance-monitoring tools for the Java platform. Among open source tools, Prometheus (<http://prometheus.io/>) and Moskito (<http://www.moskito.org/>) are relatively popular. A non-exhaustive list of Open Source performance tools is here: <http://java-source.net/open-source/profilers>

There are several commercial proprietary profilers that you may want to know about. The YourKit (<https://www.yourkit.com/>) Java profiler is

probably the most notable profiler that many people have found much success with for profiling Clojure code. There are also other profiling tools for the JVM, such as JProfiler (<https://www.ej-technologies.com/products/jprofiler/overview.html>), which is a desktop-based profiler and web-based hosted solutions such as New Relic (<http://newrelic.com/>) and AppDynamics (<https://www.appdynamics.com/>).

# Performance tuning

Once we get insight into the code via testing and profiling results, we need to analyze the bottlenecks worth considering for optimization. A better approach is to find the most under-performing portion and optimize it, thereby eliminating the weakest link. We discussed performance aspects of hardware and JVM/Clojure in previous chapters. Optimization and tuning requires rethinking the design and code in light of those aspects, and then refactoring for performance objectives.

Once we establish the performance bottlenecks, we have to pinpoint the root cause and experiment with improvisations, one step at a time, to see what works. Tuning for performance is an iterative process that is backed by measurement, monitoring and experimentation.

## Tuning Clojure code

Identifying the nature of the performance bottleneck helps a lot in order to experiment with the right aspects of the code. The key is to determine the origin of cost and whether the cost is reasonable.

### CPU/cache bound

As we noted in the beginning of this chapter, setting up a project with the right JVM options and project settings informs us of reflection and boxing, the common sources of CPU-bound performance issues after poor design and algorithm choice. As a general rule, we have to see whether we are doing unnecessary or suboptimal operations, especially inside loops. For example, transducers are amenable to better performance than lazy sequences in CPU-bound operations.

While public functions are recommended to work with immutable data structures, the implementation details can afford to use transients and arrays when performance is necessary. Records are a great alternative to

maps, where appropriate, due to type hints and tight field layout in the former. Operations on primitive data types is faster (hence recommended) than their boxed equivalents.

In tight loops, besides transients and arrays you may prefer loop-recur with unchecked math for performance. You may also like to avoid using multi-methods and dynamic vars in tight loops, rather than pass arguments around. Using Java and macros may be the last resort, but still an option if there is such a need for performance.

## Memory bound

Allocating less memory in code is always going to reduce memory-related performance issues. Optimization of memory-bound code is not only about reducing memory consumption, but it is also about memory layout and utilizing the CPU and cache well. We have to see whether we are using the data types that fit well in CPU registers and cache lines. For cache and memory-bound code, we have to know whether there are cache misses and the reason—often the data might be too large to fit in a cache line. For memory-bound code we have to care about data locality, whether the code is hitting the interconnect too often, and whether memory representation of data can be slimmed down.

## Multi-threaded

Shared resources with side effects are the main source of contention and performance bottlenecks in multi-threaded code. As we saw in the *Profiling VisualVM code* section in this chapter, profiling the threads better informs us about the bottlenecks. The best way to improve performance of multi-threaded code is to reduce contention. The easy way to reduce contention is to increase the resources and reduce concurrency, though only optimal levels of resources and concurrency would be good for performance. While designing for concurrency, append only, single writer, and shared nothing approaches work well.

Another way to reduce contention may be to exploit thread-local queueing of data until resources are available. This technique is similar to what Clojure agents use, though it is an involved technique. [Chapter 5](#), *Concurrency* covers agents in some detail. I would encourage you to study the agents source code for better understanding. When using CPU-bound resources (for example `java.util.concurrent.atomic.AtomicLong`) you may use the contention-striping technique used by some Java 8 classes (such as `java.util.concurrent.atomic.LongAdder`, which also balances between memory consumption and contention striping across processors.) This technique is also quite involved and generic contention-striping solutions may have to trade off read consistency to allow fast updates.

## I/O bound

I/O-bound tasks could be limited by bandwidth or IOPS/latency. Any I/O bottleneck usually manifests in chatty I/O calls or unconstrained data serialization. Restricting I/O to only minimum required data is a common opportunity to minimize serialization and reduce latency. I/O operations can often be batched for higher throughput, for example *SpyMemcached* library employs an asynchronous batched operation for high throughput.

I/O-bound bottlenecks are often coupled with multi-threaded scenarios. When the I/O calls are synchronous (for example, the JDBC API), one naturally has to depend upon multiple threads working on a bounded resource pool. Asynchronous I/O can relieve our threads from blocking, letting the threads do other useful work until the I/O response arrives. In synchronous I/O, we pay the cost of having threads (each allocated with memory) block on I/O calls while the kernel schedules them.

## JVM tuning

Often Clojure applications might inherit bloat from Clojure/Java libraries or frameworks, which cause poor performance. Hunting down unnecessary abstractions and unnecessary layers of code may bring decent

performance gains. Reasoning about the performance of dependency libraries/frameworks before inclusion in a project is a good approach.

The JIT compiler, garbage collector and safepoint (in Oracle HotSpot JVM) have a significant impact on the performance of applications. We discussed the JIT compiler and garbage collector in [Chapter 4, Host Performance](#). When the HotSpot JVM reaches a point when it cannot carry out concurrent, incremental GC anymore, it needs to suspend the JVM safely in order to carry out a full GC. It is also called the stop-the-world GC pause that may run up to several minutes while the JVM appears frozen.

The Oracle and OpenJDK JVMs accept many command-line options when invoked, to tune and monitor the way components in the JVM behave. Tuning GC is common among people who want to extract optimum performance from the JVM.

You may like to experiment with the following JVM options (Oracle JVM or OpenJDK) for performance:

JVM option	Description
<code>-XX:+AggressiveOpts</code>	Aggressive options that enable compressed heap pointers
<code>-server</code>	Server class JIT thresholds (use <code>-client</code> for GUI apps)
<code>-XX:+UseParNewGC</code>	Use Parallel GC
<code>-Xms3g</code>	Specify min heap size (keep it less on desktop apps)
<code>-Xmx3g</code>	Specify max heap size (keep min/max same on servers)
<code>-XX:+UseLargePages</code>	Reduce Translation-Lookaside Buffer misses (if OS supports), see <a href="http://www.oracle.com/technetwork/java/javase/tech/largememory-jsp-137182.html">http://www.oracle.com/technetwork/java/javase/tech/largememory-jsp-137182.html</a> for details

On the Java 6 HotSpot JVM, the **Concurrent Mark and Sweep (CMS)** garbage collector is well regarded for its GC performance. On the Java 7 and Java 8 HotSpot JVM, the default GC is a parallel collector (for better throughput), whereas at the time of writing this, there is a proposal to use the G1 collector (for lower pauses) by default in the upcoming Java 9. Note that the JVM GC can be tuned for different objectives, hence the same exact configuration for one application may not work well for another. Refer to the documents Oracle published for tuning the JVM at the following links:

- <http://www.oracle.com/technetwork/java/tuning-139912.html>
- <https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/>

## Back pressure

It is not uncommon to see applications behaving poorly under load. Typically, the application server simply appears unresponsive, which is often a combined result of high resource utilization, GC pressure, more threads that lead to busier thread scheduling, and cache misses. If the capacity of a system is known, the solution is to apply **back pressure** by denying services after the capacity is reached. Note that back pressure cannot be applied optimally until the system is load-tested for optimum capacity. The capacity threshold that triggers back pressure may or may not be directly associated with individual services, but rather can be defined as load criteria.

# Summary

It is worth reiterating that performance optimization begins with learning about how the underlying system works, and measuring the performance of systems we build under representative hardware and load. The chief component of performance optimization is identifying the bottlenecks using various kinds of measurements and profiling. Thereafter, we can apply experiments to tune the performance of code and measure/profile once again to verify. The tuning mechanism varies depending on the type of bottleneck.

In the next chapter, we will see how to address performance concerns when building applications. Our focus will be on the several common patterns that impact performance.

# Chapter 8. Application Performance

The earliest computing devices were built to perform automatic computations and, as computers grew in power, they became increasingly popular because of how much and how fast they could compute. Even today, this essence lives on in our anticipation that computers can execute our business calculations faster than before by means of the applications we run on them.

Compared to performance analysis and optimization at a smaller component level, as we saw in previous chapters, it takes a holistic approach to improve performance at the application level. The higher-level concerns, such as serving a certain number of users in a day, or handling an identified quantum of load through a multi-layered system, requires us to think about how the components fit together and how the load is designed to flow through it. In this chapter, we will discuss such high-level concerns. Like the previous chapter, by and large this chapter applies to applications written in any JVM language, but with a focus on Clojure. In this chapter, we will discuss general performance techniques that apply to all layers of the code:

- Choosing libraries
- Logging
- Data sizing
- Resource pooling
- Fetch and compute in advance
- Staging and batching
- Little's law

## Choosing libraries

Most non-trivial applications depend a great deal on third-party libraries

for various functionality, such as logging, serving web requests, connecting to databases, writing to message queues, and so on. Many of these libraries not only carry out parts of critical business functionality but also appear in the performance-sensitive areas of our code, impacting the overall performance. It is imperative that we choose libraries wisely (with respect to features versus performance trade off) after due performance analysis.

The crucial factor in choosing libraries is not identifying which library to use, rather it is having a performance model of our applications and having the use cases benchmarked under representative load. Only benchmarks can tell us whether the performance is problematic or acceptable. If the performance is below expectation, a drill-down profiling can show us whether a third-party library is causing the performance issue. In [Chapter 6, Measuring Performance](#) and [Chapter 7, Performance Optimization](#) we discussed how to measure performance and identify bottlenecks. You can evaluate multiple libraries for performance-sensitive use cases and choose what suits.

Libraries often improve (or occasionally lose) performance with new releases, so measurement and profiling (comparative, across versions) should be an ongoing practice for the development and maintenance lifecycle of our applications. Another factor to note is that libraries may show different performance characteristics based on the use case, load, and the benchmark. The devil is in the benchmark details. Be sure that your benchmarks are as close as possible to the representative scenario for your application.

## Making a choice via benchmarks

Let's take a brief look at a few general use cases where performance of third-party libraries are exposed via benchmarks.

### Web servers

Web servers are typically subject to quite a bit of performance

benchmarking due to their generic nature and scope. One such benchmark for Clojure web servers exists here:

<https://github.com/ptaoussanis/clojure-web-server-benchmarks>

Web servers are complex pieces of software and they may exhibit different characteristics under various conditions. As you will notice, the performance numbers vary based on keep-alive versus non-keep-alive modes and request volume—at the time of writing, Immutant-2 came out better in keep-alive mode but fared poorly in the non-keep-alive benchmark. In production, people often front their application servers with reverse proxy servers, for example Nginx or HAProxy, which make keep-alive connections to application servers.

## Web routing libraries

There are several web routing libraries for Clojure, as listed here:

<https://github.com/juxt/bidi#comparison-with-other-routing-libraries>

The same document also shows a performance benchmark with **Compojure** as the baseline, in which (at the time of writing) Compojure turns out to be performing better than **Bidi**. However, another benchmark compares Compojure, **Clout** (the library that Compojure internally uses), and **CalfPath** routing here:

<https://github.com/kumarshantanu/calfpath#development>

In this benchmark, as of this writing, Clout performs better than Compojure, and CalfPath outperforms Clout. However, you should be aware of any caveats in the faster libraries.

## Data serialization

There are several ways to serialize data in Clojure, for example EDN and Fressian. Nippy is another serialization library with benchmarks to

demonstrate how well it performs over EDN and Fressian:

<https://github.com/ptaoussanis/nippy#performance>

We covered Nippy in [Chapter 2](#), *Clojure Abstractions* to show how it uses transients to speed up its internal computations. Even within Nippy, there are several flavors of serialization that have different features/performance trade-offs.

## JSON serialization

Parsing and generating JSON is a very common use case in RESTful services and web applications. The Clojure contrib library clojure/data.json (<https://github.com/clojure/data.json>) provides this functionality. However, many people have found out that the Cheshire library <https://github.com/dakrone/cheshire> performs much better than the former. The included benchmarks in Cheshire can be run using the following command:

```
lein with-profile dev,benchmark test
```

Cheshire internally uses the Jackson Java library <https://github.com/FasterXML/jackson>, which is known for its good performance.

## JDBC

JDBC access is another very common use case among applications using relational databases. The Clojure contrib library clojure/java.jdbc <https://github.com/clojure/java.jdbc> provides a Clojure JDBC API. Asphalt <https://github.com/kumarshantanu/asphalt> is an alternative JDBC library where the comparative benchmarks can be run as follows:

```
lein with-profile dev,c17,perf test
```

As of this writing, Asphalt outperforms clojure/java.jdbc by several micro seconds, which may be useful in low-latency applications. However,

note that JDBC performance is usually dominated by SQL queries/joins, database latency, connection pool parameters, and so on. We will discuss more about JDBC in later sections.

# Logging

Logging is a prevalent activity that almost all non-trivial applications do. Logging calls are quite frequent, hence it is important to make sure our logging configuration is tuned well for performance. If you are not familiar with logging systems (especially on the JVM), you may want to take some time to get familiar with those first. We will cover the use of `clojure/tools.logging`, **SLF4J** and **LogBack** libraries (as a combination) for logging, and look into how to make them perform well:

- Clojure/tools.logging <https://github.com/clojure/tools.logging>
- SLF4J: <http://www.slf4j.org/>
- LogBack: <http://logback.qos.ch/>

## Why SLF4J/LogBack?

Besides SLF4J/LogBack, there are several logging libraries to choose from in the Clojure application, for example Timbre, Log4j and `java.util.logging`. While there is nothing wrong with these libraries, we are often constrained into choosing something that covers most other third-party libraries (also including Java libraries) in our applications for logging purposes. SLF4J is a Java logger facade that detects any available implementation (LogBack, Log4j, and so on)—we choose LogBack simply because it performs well and is highly configurable. The library `clojure/tools.logging` provides a Clojure logging API that detects SLF4J, Log4j or `java.util.logging` (in that order) in the classpath and uses whichever implementation is found first.

## The setup

Let's walk through how to set up a logging system for your application using LogBack, SLF4J and `clojure/tools.logging` for a project built using Leiningen.

## Dependencies

Your project.clj file should have the LogBack, SLF4J and clojure/tools.logging dependencies under the :dependencies key:

```
[ch.qos.logback/logback-classic "1.1.2"]
[ch.qos.logback/logback-core      "1.1.2"]
[org.slf4j/slf4j-api              "1.7.9"]
[org.codehaus.janino/janino       "2.6.1"] ; for Logback-config
[org.clojure/tools.logging         "0.3.1"]
```

The previously mentioned versions are current and work as of the time of writing. You may want to use updated versions, if available.

## The logback configuration file

You need to create a logback.xml file in the resources directory:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>

    <appender name="FILE"
        class="ch.qos.logback.core.rolling.RollingFileAppender">
        <file>${logfile.general.name}:-logs/application.log</file>
        <rollingPolicy
            class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <!-- daily rollover -->
            <fileNamePattern>${logfile.general.name}:-logs/application.log}.%d{yyyy-MM-dd}.%i.gz</fileNamePattern>
            <timeBasedFileNamingAndTriggeringPolicy
                class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP">
                <!-- or whenever the file size reaches 100MB -->
                <maxFileSize>100MB</maxFileSize>
            </timeBasedFileNamingAndTriggeringPolicy>
            <!-- keep 30 days worth of history -->
            <maxHistory>30</maxHistory>
        </rollingPolicy>
        <append>true</append>
        <encoder
            class="ch.qos.logback.core.encoder.LayoutWrappingEncoder">
            <layout class="ch.qos.logback.classic.PatternLayout">
                <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36}
- %msg%n</pattern>
```

```

</layout>
<immediateFlush>false</immediateFlush>
</encoder>
</appender>

<appender name="AsyncFile"
class="ch.qos.logback.classic.AsyncAppender">
<queueSize>500</queueSize>
<discardingThreshold>0</discardingThreshold>
<appender-ref ref="FILE" />
</appender>

<!-- You may want to set the level to DEBUG in development -->
<root level="ERROR">
  <appender-ref ref="AsyncFile" />
</root>

<!-- Replace com.example with base namespace of your app -->
<logger name="com.example" additivity="false">
  <!-- You may want to set the level to DEBUG in development -->
  <level value="INFO"/>
  <appender-ref ref="AsyncFile" />
</logger>

</configuration>

```

The previous `logback.xml` file is simple on purpose (for illustration) and has just enough configuration to get you started with logging using LogBack.

## Optimization

The optimization points are highlighted in the `logback.xml` file we saw earlier in this section. We set the `immediateFlush` attribute to `false` such that the messages are buffered before flushing to the appender. We also wrapped the regular file appender with an asynchronous appender and edited the `queueSize` and `discardingThreshold` attributes, which gets us much better results than the default.

Unless optimized, logging configurations are usually a common source of suboptimal performance in many applications. Usually, the performance problems show up only at high load when the log volume is very high. The optimizations discussed previously are only a few of the many possible optimizations that one can experiment with. The chapters in LogBack documentation, such as **encoders** (<http://logback.qos.ch/manual/encoders.html>), **appenders** ([http://logback.qos.ch/manual-appenders.html](http://logback.qos.ch/manual	appenders.html)) and **configuration** (<http://logback.qos.ch/manual/configuration.html>) have useful **information**. There are also tips <http://blog.takipi.com/how-to-instantly-improve-your-java-logging-with-7-logback-tweaks/> on the Internet that may provide useful pointers.

# Data sizing

The cost of abstractions in terms of the data size plays an important role. For example, whether or not a data element can fit into a processor cache line depends directly upon its size. On a Linux system, we can find out the cache line size and other parameters by inspecting the values in the files under the `/sys/devices/system/cpu/cpu0/cache/` directory. Refer to [Chapter 4, Host Performance](#), where we discussed how to compute the size of primitives, objects, and data elements.

Another concern we generally find with data sizing is how much data we hold at any time in the heap. As we noted in earlier chapters, GC has direct consequences on the application performance. While processing data, often we do not really need all the data we hold on to. Consider the example of generating a summary report of sold items for a certain period (months) of time. After the subperiod (month-wise) summary data is computed, we do not need the item details anymore, hence it's better to remove the unwanted data while we add the summaries. See the following example:

```
(defn summarize [daily-data] ; daily-data is a map
  (let [s (items-summary (:items daily-data))]
    (-> daily-data
        (select-keys [:digest :invoices]) ; keep required k/v
        pairs
        (assoc :summary s)))))

;; now inside report generation code
(-> (fetch-items period-from period-to :interval-day)
     (map summarize)
     generate-report)
```

Had we not used `select-keys` in the previous `summarize` function, it would have returned a map with extra `:summary` data along with all other existing keys in the map. Now, such a thing is often combined with lazy sequences, so for this scheme to work it is important not to hold onto the head of the

lazy sequence. Recall that in [Chapter 2](#), *Clojure Abstractions* we discussed the perils of holding onto the head of a lazy sequence.

## Reduced serialization

We discussed in earlier chapters that serialization over an I/O channel is a common source of latency. The perils of over-serialization cannot be overstated. Whether we read or write data from a data source over an I/O channel, all of that data needs to be prepared, encoded, serialized, de-serialized, and parsed before being worked upon. The less data that is involved, the better it is for every step in order to lower the overhead. Where there is no I/O involved (such as in-process communication), it generally makes no sense to serialize.

A common example of over-serialization is when working with SQL databases. Often, there are common SQL query functions that fetch all columns of a table or a relation—they are called by various functions that implement business logic. Fetching data that we do not need is wasteful and detrimental to performance for the same reason that we discussed in the previous paragraph. While it may seem more work to write one SQL statement and one database-query function for each use case, it pays off with better performance. Code that uses NoSQL databases is also subject to this anti-pattern—we have to take care to fetch only what we need even though it may lead to additional code.

There's a pitfall to be aware of when reducing serialization. Often, some information needs to be inferred in the absence of serialized data. In such cases, where some of the serialization is dropped so that we can infer other information, we must compare the cost of inference versus the serialization overhead. The comparison may not necessarily be only per operation, but rather on the whole, such that we can consider the resources we can allocate in order to achieve capacities for various parts of our systems.

## Chunking to reduce memory pressure

What happens when we slurp a text file regardless of its size? The contents of the entire file will sit in the JVM heap. If the file is larger than the JVM heap capacity, the JVM will terminate, throwing `OutOfMemoryError`. If the file is large, but not enough to force the JVM into OOM error, it leaves relatively less JVM heap space for other operations to continue in the application. Similar situations take place when we carry out any operation disregarding the JVM heap capacity. Fortunately, this can be fixed by reading data in chunks and processing them before reading more. In [Chapter 3, \*Leaning on Java\*](#), we briefly discussed memory mapped buffers, which is another complementary solution that you may like to explore.

## Sizing for file/network operations

Let's take the example of a data ingestion process where a semi-automated job uploads large **Comma Separated File (CSV)** files via **File Transfer Protocol (FTP)** to a file server, and another automated job (written in Clojure) runs periodically to detect the arrival of files via a Network File System (NFS). After detecting a new file, the Clojure program processes the file, updates the result in a database, and archives the file. The program detects and processes several files concurrently. The size of the CSV files is not known in advance, but the format is predefined.

As per the previous description, one potential problem is, since there could be multiple files being processed concurrently, how do we distribute the JVM heap among the concurrent file-processing jobs? Another issue at hand could be that the operating system imposes a limit on how many files could be open at a time; on Unix-like systems you can use the `ulimit` command to extend the limit. We cannot arbitrarily slurp the CSV file contents—we must limit each job to a certain amount of memory, and also limit the number of jobs that can run concurrently. At the same time, we cannot read a very small number of rows from a file at a time because this may impact performance:

```
(def ^:const K 1024)
```

```

;; create the buffered reader using custom 128K buffer-size
(-> filename
  java.io.FileInputStream.
  java.io.InputStreamReader.
  (java.io.BufferedReader. (* K 128)))

```

Fortunately, we can specify the buffer size when reading from a file (or even from a network stream) so as to tune the memory usage and performance as appropriate. In the previous code example, we explicitly set the buffer size of the reader to facilitate the same.

## Sizing for JDBC query results

Java's interface standard for SQL databases, JDBC (which is technically not an acronym), supports *fetch size* for fetching query results via JDBC drivers. The default fetch size depends on the JDBC driver. Most of the JDBC drivers keep a low default value to avoid high memory usage and for internal performance optimization reasons. A notable exception to this norm is the MySQL JDBC driver that completely fetches and stores all rows in memory by default:

```

(require '[clojure.java.jdbc :as jdbc])

;; using prepare-statement directly
(with-open
  [stmt (jdbc/prepare-statement
          conn sql :fetch-size 1000 :max-rows 9000)
   rset (resultset-seq (.executeQuery stmt)) ]
  (vec rset))

;; using query
(jdbc/query db [{:fetch-size 1000}
                "SELECT empno FROM emp WHERE country=?"] )

```

When using the Clojure contrib library `java.jdbc` (<https://github.com/clojure/java.jdbc> as of version 0.3.7), the fetch size can be set while preparing a statement as shown in the previous example. Note that the fetch size does not guarantee proportional latency; however, it can

be used safely for memory sizing. We must test any performance-impacting latency changes due to fetch size at different loads and use cases for the particular database and JDBC driver. Another important factor to note is that the benefit of `:fetch-size` can be useful only if the query result set is consumed incrementally and lazily—if a function extracts all rows from a result set to create a vector, then the benefit of `:fetch-size` is nullified from a memory conservation point of view. Besides fetch size, we can also pass the `:max-rows` argument to limit the maximum rows to be returned by a query—however, this implies that the extra rows will be truncated from the result, and not whether the database will internally limit the number of rows to realize.

# Resource pooling

There are several types of resources on the JVM that are rather expensive to initialize. Examples are HTTP connections, execution threads, JDBC connections, and so on. The Java API recognizes such resources and has built-in support for creating a pool of some of those resources, such that the consumer code borrows a resource from a pool when required and at the end of the job simply returns it to the pool. Java's thread pools (discussed in [Chapter 5, Concurrency](#)) and JDBC data sources are prominent examples. The idea is to preserve the initialized objects for reuse. Even though Java does not support pooling of a resource type directly, one can always create a pool abstraction around custom expensive resources. Note that the pooling technique is common in I/O activities, but can be equally applicable to non-I/O purposes where initialization cost is high.

## JDBC resource pooling

Java supports the obtaining of JDBC connections via the `javax.sql.DataSource` interface, which can be pooled. A JDBC connection pool implements this interface. Typically, a JDBC connection pool is implemented by third-party libraries or a JDBC driver itself. Generally, very few JDBC drivers implement a connection pool, so Open Source third-party JDBC resource pooling libraries such as Apache DBCP, c3p0, BoneCP, HikariCP, and so on are popular. They also support validation queries for eviction of stale connections that might result from network timeouts and firewalls, and guard against connection leaks. Apache DBCP and HikariCP are accessible from Clojure via their respective Clojure wrapper libraries Clj-DBCP (<https://github.com/kumarshantanu/clj-dbcp>) and HikariCP (<https://github.com/tomekw/hikari-cp>), and there are Clojure examples describing how to construct C3P0 and BoneCP pools ([http://clojure-doc.org/articles/ecosystem/java\\_jdbc/connection\\_pooling.html](http://clojure-doc.org/articles/ecosystem/java_jdbc/connection_pooling.html)).

Connections are not the only JDBC resources that need to be pooled. Every time we create a new JDBC prepared statement, depending on the JDBC driver implementation, often the entire statement template is sent to the database server in order to obtain a reference to the prepared statement. As the database servers are generally deployed on separate hardware, there may be network latency involved. Hence, the pooling of prepared statements is a very desirable property of JDBC resource pooling libraries. Apache DBCP, C3P0, and BoneCP all support statement pooling, and the Clj-DBCP wrapper enables the pooling of prepared statements out-of-the-box for better performance. HikariCP has the opinion that statement pooling, nowadays, is already done internally by JDBC drivers, hence explicit pooling is not required. I would strongly advise running your benchmarks with the connection pooling libraries to determine whether or not it really works for your JDBC driver and application.

# I/O batching and throttling

It is well known that chatty I/O calls generally lead to poor performance. In general, the solution is to batch together several messages and send them in one payload. In databases and network calls, batching is a common and useful technique to improve throughput. On the other hand, large batch sizes may actually harm throughput as they tend to incur memory overhead, and components may not be ready to handle a large batch at once. Hence, sizing the batches and throttling are just as important as batching. I would strongly advise conducting your own tests to determine the optimum batch size under representative load.

## JDBC batch operations

JDBC has long had batch-update support in its API, which includes the INSERT, UPDATE, DELETE statements. The Clojure contrib library `java.jdbc` supports JDBC batch operations via its own API, as we can see as follows:

```
(require '[clojure.java.jdbc :as jdbc])

;; multiple SQL statements
(jdbc/db-do-commands
 db true
 ["INSERT INTO emp (name, countrycode) VALUES ('John Smith', 3)"
  "UPDATE emp SET countrycode=4 WHERE empid=1379"])

;; similar statements with only different parametrs
(jdbc/db-do-prepared
 db true
 "UPDATE emp SET countrycode=? WHERE empid=?"
 [4 1642]
 [9 1186]
 [2 1437])
```

Besides batch-update support, we can also batch JDBC queries. One of the common techniques is to use the SQL `WHERE` clause to avoid the `N+1` selects issue. The `N+1` issue indicates the situation when we execute one query in

another child table for every row in a rowset from a master table. A similar technique can be used to combine several similar queries on the same table into just one, and segregate the data in the program afterwards.

Consider the following example that uses clojure.java.jdbc 0.3.7 and the MySQL database:

```
(require '[clojure.java.jdbc :as j])

(def db {:subprotocol "mysql"
         :subname "//127.0.0.1:3306/clojure_test"
         :user "clojure_test" :password "clojure_test"})

;; the snippet below uses N+1 selects
;; (typically characterized by SELECT in a loop)
(def rq "select order_id from orders where status=?")
(def tq "select * from items where fk_order_id=?")
(doseq [order (j/query db [rq "pending"])]
  (let [items (j/query db [tq (:order_id order)])]
    ;; do something with items
    ...))

;; the snippet below avoids N+1 selects,
;; but requires fk_order_id to be indexed
(def jq "select t.* from orders r, items t
         where t.fk_order_id=r.order_id and r.status=? order by
t.fk_order_id")
(let [all-items (group-by :fk_order_id (j/query db [jq
"pending"]))]
  (doseq [[order-id items] all-items]
    ;; do something with items
    ...))
```

In the previous example there are two tables: `orders` and `items`. The first snippet reads all order IDs from the `orders` table, and then iterates through them to query corresponding entries in the `items` table in a loop. This is the `N+1` selects performance anti-pattern you should keep an eye on. The second snippet avoids `N+1` selects by issuing a single SQL query, but may not perform very well unless the column `fk_order_id` is indexed.

# Batch support at API level

When designing any service, it is very useful to provide an API for batch operations. This builds flexibility in the API such that batch sizing and throttling can be controlled in a fine-grained manner. Not surprisingly, it is also an effective recipe for building high-performance services. A common overhead we encounter when implementing batch operations is the identification of each item in the batch and their correlation across requests and responses. The problem becomes more prominent when requests are asynchronous.

The solution to the item identification issue is resolved either by assigning a canonical or global ID to each item in the request (batch), or by assigning every request (batch) a unique ID and each item in the request an ID that is local to the batch.

The choice of the exact solution usually depends on the implementation details. When requests are synchronous, you can do away with identification of each request item (see the Facebook API for reference: <http://developers.facebook.com/docs/reference/api/batch/>) where the items in response follow the same order as in the request. However, in asynchronous requests, items may have to be tracked via status-check call or callbacks. The desired tracking granularity typically guides the appropriate item identification strategy.

For example, if we have a batch API for order processing, every order would have a unique Order-ID that can be used in subsequent status-check calls. In another example, let's say there is a batch API for creating API keys for **Internet of Things (IoT)** devices—here, the API keys are not known beforehand, but they can be generated and returned in a synchronous response. However, if this has to be an asynchronous batch API, the service should respond with a batch request ID that can be used later to find the status of the request. In a batch response for the request ID, the server can include request item IDs (for example device IDs, which

may be unique for the client but not unique across all clients) with their respective status.

## Throttling requests to services

As every service can handle only a certain capacity, the rate at which we send requests to a service is important. The expectations about the service behavior are generally in terms of both throughput and latency. This requires us to send requests at a specified rate, as a rate lower than that may lead to under-utilization of the service, and a higher rate may overload the service or result in failure, thus leading to client-side under-utilization.

Let's say a third-party service can accept 100 requests per second. However, we may not know how robustly the service is implemented. Though sometimes it is not exactly specified, sending 100 requests at once (within 20ms, let's say) during each second may lead to lower throughput than expected. Evenly distributing the requests across the one-second duration, for example sending one request every 10ms ( $1000\text{ms} / 100 = 10\text{ms}$ ), may increase the chance of attaining the optimum throughput.

For throttling, **Token bucket** ([https://en.wikipedia.org/wiki/Token\\_bucket](https://en.wikipedia.org/wiki/Token_bucket)) and **Leaky bucket** ([https://en.wikipedia.org/wiki/Leaky\\_bucket](https://en.wikipedia.org/wiki/Leaky_bucket)) algorithms can be useful. Throttling at a very fine-grained level requires that we buffer the items so that we can maintain a uniform rate. Buffering consumes memory and often requires ordering; queues (covered in [Chapter 5, Concurrency](#)), pipeline and persistent storage usually serve that purpose well. Again, buffering and queuing may be subject to back pressure due to system constraints. We will discuss pipelines, back pressure and buffering in a later section in this chapter.

# Precomputing and caching

While processing data, we usually come across instances where few common computation steps precede several kinds of subsequent steps. That is to say, some amount of computation is common and the remaining is different. For high-latency common computations (I/O to access the data and memory/CPU to process it), it makes a lot of sense to compute them once and store in digest form, such that the subsequent steps can simply use the digest data and proceed from that point onward, thus resulting in reduced overall latency. This is also known as staging of semi-computed data and is a common technique to optimize processing of non-trivial data.

Clojure has decent support for caching. The built-in `clojure.core/memoize` function performs basic caching of computed results with no flexibility in using specific caching strategies and pluggable backends. The Clojure contrib library `core.memoize` offsets the lack of flexibility in `memoize` by providing several configuration options. Interestingly, the features in `core.memoize` are also useful as a separate caching library, so the common portion is factored out as a Clojure contrib library called `core.cache` on top of which `core.memoize` is implemented.

As many applications are deployed on multiple servers for availability, scaling and maintenance reasons, they need distributed caching that is fast and space efficient. The open source memcached project is a popular in-memory, distributed key-value/object store that can act as a caching server for web applications. It hashes the keys to identify the server to store the value on, and has no out-of-the-box replication or persistence. It is used to cache database query results, computation results, and so on. For Clojure, there is a memcached client library called SpyGlass (<https://github.com/clojurewerkz/spyglass>). Of course, memcached is not limited to just web applications; it can be used for other purposes too.

# Concurrent pipelines

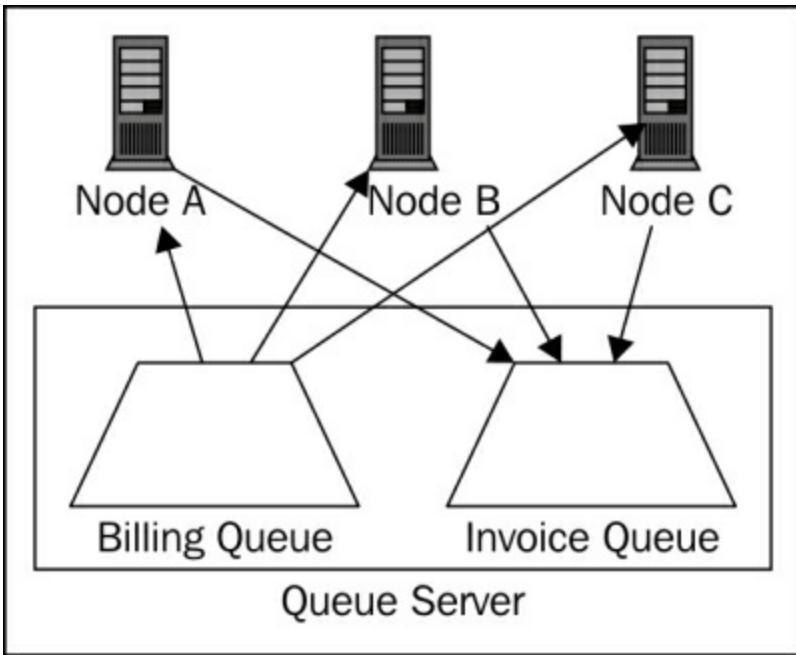
Imagine a situation where we have to carry out jobs at a certain throughput, such that each job includes the same sequence of differently sized I/O task (task A), a memory-bound task (task B) and, again, an I/O task (task C). A naïve approach would be to create a thread pool and run each job off it, but soon we realize that this is not optimum because we cannot ascertain the utilization of each I/O resource due to unpredictability of the threads being scheduled by the OS. We also observe that even though several concurrent jobs have similar I/O tasks, we are unable to batch them in our first approach.

As the next iteration, we split each job in stages (A, B, C), such that each stage corresponds to one task. Since the tasks are well known, we create one thread pool (of appropriate size) per stage and execute tasks in them. The result of task A is required by task B, and B's result is required by task C—we enable this communication via queues. Now, we can tune the thread pool size for each stage, batch the I/O tasks, and throttle them for an optimum throughput. This kind of an arrangement is a concurrent pipeline. Some readers may find this feebly resembling the actor model or **Staged Event Driven Architecture (SEDA)** model, which are more refined models for this kind of approach. Recall that we discussed several kinds of in-process queues in [Chapter 5, Concurrency](#).

# Distributed pipelines

With this approach, it is possible to scale out the job execution to multiple hosts in a cluster using network queues, thereby offloading memory consumption, durability, and delivery to the queue infrastructure. For example, in a given scenario there could be several nodes in a cluster, all of them running the same code and exchanging messages (requests and intermediate result data) via network queues.

The following diagram depicts how a simple invoice-generation system might be connected to network queues:



RabbitMQ, HornetQ, ActiveMQ, Kestrel and Kafka are some well-known Open Source queue systems. Once in a while, the jobs may require distributed state and coordination. The Avout (<http://avout.io/>) project implements the distributed version of Clojure's atom and ref, which can be used for this purpose. Tesser (<https://github.com/aphyr/tesser>) is another library for local and distributed parallelism using Clojure. The Storm (<http://storm-project.net/>) and Onyx (<http://www.onyxplatform.org/>) projects are distributed, real-time stream processing systems implemented using Clojure.

# Applying back pressure

We discussed back pressure briefly in the last chapter. Without back pressure we cannot build a reasonable load-tolerant system with predictable stability and performance. In this section, we will see how to apply back pressure in different scenarios in an application. At a fundamental level, we should have a threshold of a maximum number of concurrent jobs in the system and, based on that threshold, we should reject new requests above a certain arrival rate. The rejected messages may either be retried by the client or ignored if there is no control over the client. When applying back pressure to user-facing services, it may be useful to detect system load and deny auxiliary services first in order to conserve capacity and degrade gracefully in the face of high load.

## Thread pool queues

JVM thread pools are backed by queues, which means that when we submit a job into a thread pool that already has the maximum jobs running, the new job lands in the queue. The queue is by default an unbounded queue, which is not suitable for applying back pressure. So, we have to create the thread pool backed by a bounded queue:

```
(import 'java.util.concurrent.LinkedBlockingDeque)
(import 'java.util.concurrent.TimeUnit)
(import 'java.util.concurrent.ThreadPoolExecutor)
(import 'java.util.concurrent.ThreadPoolExecutor$AbortPolicy)
(def tpool
  (let [q (LinkedBlockingDeque. 100)
        p (ThreadPoolExecutor$AbortPolicy.)]
    (ThreadPoolExecutor. 1 10 30 TimeUnit/SECONDS q p)))
```

Now, on this pool, whenever there is an attempt to add more jobs than the capacity of the queue, it will throw an exception. The caller should treat the exception as a buffer-full condition and wait until the buffer has idle capacity again by periodically polling the

`java.util.concurrent.BlockingQueue.remainingCapacity()` method.

## Servlet containers such as Tomcat and Jetty

In the synchronous **Tomcat** and **Jetty** versions, each HTTP request is given a dedicated thread from a common thread pool that a user can configure. The number of simultaneous requests being served is limited by the thread pool size. A common way to control the arrival rate is to set the thread pool size of the server. The **Ring** library uses an embedded jetty server by default in development mode. The embedded Jetty adapter (in Ring) can be programmatically configured with a thread pool size.

In the asynchronous (Async Servlet 3.0) versions of Tomcat and Jetty beside the thread pool size, it is also possible to specify the timeout for processing each request. However, note that the thread pool size does not limit the number of requests in asynchronous versions in the way it does on synchronous versions. The request processing is transferred to an `ExecutorService` (thread pool), which may buffer requests until a thread is available. This buffering behavior is tricky because this may cause system overload—you can override the default behavior by defining your own thread pool instead of using the servlet container's thread pool to return a HTTP error at a certain threshold of waiting requests.

## HTTP Kit

**HTTP Kit** (<http://http-kit.org/>) is a high-performance asynchronous (based on Java NIO implementation) web server for Clojure. It has built-in support for applying back pressure to new requests via a specified queue length. As of HTTP Kit 2.1.19, see the following snippet:

```
(require '[org.httpkit.server :as hk])  
  
;; handler is a typical Ring handler  
(hk/run-server handler {:port 3000 :thread 32 :queue-size 600})
```

In the previous snippet, the worker thread pool size is 32 and the max

queue length is specified as 600. When not specified, 20480 is the default maximum queue length for applying back pressure.

## Aleph

Aleph (<http://aleph.io/>) is another high-performance asynchronous web server based on the Java Netty (<http://netty.io/>) library, which in turn is based on Java NIO. Aleph extends Netty with its own primitives compatible with Netty. The worker thread pool in Aleph is specified via an option, as we can see in the following snippet as of Aleph 0.4.0:

```
(require '[aleph.http :as a])  
  
;; handler is a typical Ring handler  
(a/start-server handler {:executor tpool})
```

Here, `tpool` refers to a bounded thread pool as discussed in the subsection *Thread pool queues*. By default, Aleph uses a dynamic thread pool capped at maximum 512 threads aimed at 90 percent system utilization via the **Dirigiste** (<https://github.com/ztellman/dirigiste>) library.

Back pressure not only involves enqueueing a limited number of jobs, but slows down the processing rate of a job when the peer is slow. Aleph deals with per-request back pressure (for example, when streaming response data) by "not accepting data until it runs out of memory" — it falls back to blocking instead of dropping data, or raising exceptions and closing connections

# Performance and queueing theory

If we observe the performance benchmark numbers across a number of runs, even though the hardware, loads and OS remain the same, the numbers are rarely exactly the same. The difference between each run may be as much as -8 percent to 8 percent for no apparent reason. This may seem surprising, but the deep-rooted reason is that the performances of computer systems are *stochastic* by nature. There are many small factors in a computer system that make performance unpredictable at any given point of time. At best, the performance variations can be explained by a series of probabilities over random variables.

The basic premise is that each subsystem is more or less like a queue where requests await their turn to be served. The CPU has an instruction queue with unpredictable fetch/decode/branch-predict timings, the memory access again depends on cache hit ratio and whether it needs to be dispatched via the interconnect, and the I/O subsystem works using interrupts that may again depend on mechanical factors of the I/O device. The OS schedules threads that wait while not executing. The software built on the top of all this basically waits in various queues to get the job done.

## Little's law

Little's law states that, over steady state, the following holds true:

$$\text{mean number of jobs in a system} = \text{mean arrival rate} \times \text{mean response time}$$

$$\text{mean number of jobs in the queue} = \text{mean arrival rate} \times \text{mean waiting time}$$

This is a rather important law that gives us insight into the system capacity

as it is independent of other factors. For an example, if the average time to satisfy a request is 200 ms and the service rate is about 70 per second, then the mean number of requests being served is  $70 \text{ req/second} \times 0.2 \text{ second} = 14 \text{ requests}$ .

Note that Little's law does not talk about spikes in request arrival rate or spikes in latency (due to GC and/or other bottlenecks) or system behavior in response to these factors. When the arrival rate spikes at one point, your system must have enough resources to handle the number of concurrent tasks required to serve the requests. We can infer here that Little's law is helpful to measure and tune average system behavior over a duration, but we cannot plan capacity based solely on this.

## **Performance tuning with respect to Little's law**

In order to maintain good throughput, we should strive to maintain an upper limit on the total number of tasks in the system. Since there can be many kinds of tasks in a system and lot of tasks can happily co-exist in the absence of bottlenecks, a better way to say it is to ensure that the system utilization and bottlenecks remain in limit.

Often, the arrival rate may not be within the control of a system. For such scenarios, the only option is to minimize the latency as much as possible and deny new requests after a certain threshold of total jobs in the system. You may be able to know the right threshold only through performance and load tests. If you can control the arrival rate, you can throttle the arrival (based on performance and load tests) so as to maintain a steady flow.

# Summary

Designing an application for performance should be based on the use cases and patterns of anticipated system load and behavior. Measuring performance is extremely important to guide optimization in the process. Fortunately, there are several well-known optimization patterns to tap into, such as resource pooling, data sizing, pre-fetch and pre-compute, staging, batching, and so on. As it turns out, application performance is not only a function of the use cases and patterns—the system as a whole is a continuous stochastic turn of events that can be assessed statistically and is guided by probability. Clojure is a fun language to do high-performance programming. This book prescribes many pointers and practices for performance, but there is no mantra that can solve everything. The devil is in the details. Know the idioms and patterns, experiment to see what works for your applications, and know which rules you can bend for performance.

# Part 3. Module 3

## *Mastering Clojure*

*Understand the philosophy of the Clojure language and dive into its inner workings to unlock its advanced features, methodologies, and constructs*

# Chapter 1. Working with Sequences and Patterns

In this chapter, we'll revisit a few basic programming techniques, such as recursion and sequences, with Clojure. As we will see, Clojure focuses on the use of higher-order functions to abstract computation, like any other functional programming language. This design can be observed in most, if not all, of the Clojure standard library. In this chapter, we will cover the following topics:

- Exploring recursion
- Learning about sequences and laziness
- Examining zippers
- Briefly studying pattern matching

## Defining recursive functions

**Recursion** is one of the central methodologies of computer science. It allows us to elegantly solve problems that have cumbersome non-recursive solutions. Yet, recursive functions are discouraged in quite a few imperative programming languages in favor of non-recursive functions. Clojure does no such thing and completely embraces recursion along with all its pros and cons. In this section, we will explore how to define recursive functions.

### Note

The following examples can be found in `src/m_c1j/c1/recur.clj` of the book's source code.

In general, a function can be made recursive by simply calling it again from within the body of the function. We can define a simple function to return the first  $n$  numbers of the Fibonacci sequence as shown in *Example*

## 1.1:

```
(defn fibo
  ([n]
   (fibo [0N 1N] n))
  ([xs n]
   (if (<= n (count xs))
       xs
       (let [x' (+ (last xs)
                    (nth xs (- (count xs) 2)))
             xs' (conj xs x')]
           (fibo xs' n))))
```

*Example 1.1: A simple recursive function*

## Note

The Fibonacci sequence is a series of numbers that can be defined as follows:

The first element  $F_0$  is 0 and the second element  $F_1$  is 1.

The rest of the numbers are the sum of the previous two numbers, that is the nth Fibonacci number  $F_n = F_{n-1} + F_{n-2}$ .

In the previously defined `fibo` function, the last two elements of the list are determined using the `nth` and `last` functions, and the sum of these two elements is appended to the list using the `conj` function. This is done in a recursive manner, and the function terminates when the length of the list, determined by the `count` function becomes equal to the supplied value `n`. Also, the values `0N` and `1N`, which represent `BigInteger` types, are used instead of the values 0 and 1. This is done because using long or integer values for such a computation could result in an arithmetic overflow error. We can try out this function in the REPL shown as follows:

```
user> (fibo 10)
[0N 1N 1N 2N 3N 5N 8N 13N 21N 34N]
user> (last (fibo 100))
```

218922995834555169026N

The `fibo` function returns a vector of the first `n` Fibonacci numbers as expected. However, for larger values of `n`, this function will cause a stack overflow:

```
user> (last (fibo 10000))
StackOverflowError  clojure.lang.Numbers.lt  (Numbers.java:219)
```

The reason for this error is that there were too many nested function calls. A call to any function requires an additional call stack. With recursion, we reach a point where all of the available stack space in a program is consumed and no more function calls can be performed. A *tail call* can overcome this limitation by using the existing call stack for a recursive call, which removes the need for allocating a new call stack. This is only possible when the return value of a function is the return value of a recursive call made by the function, in which case an additional call stack is not required to store the state of the function that performs the recursive call. This technique is termed as *tail call elimination*. In effect, a tail call optimized function consumes a constant amount of stack space.

In fact, the `fibo` function does indeed make a tail call, as the last expression in the body of the function is a recursive call. Still, it consumes stack space for each recursive call. This is due to the fact that the underlying virtual machine, the JVM, does not perform tail call elimination. In Clojure, tail call elimination has to be done explicitly using a `recur` form to perform a recursive call. The `fibo` function we defined earlier can be refined to be *tail recursive* by using a `recur` form, as shown in *Example 1.2*:

```
(defn fibo-recur
  ([n]
   (fibo-recur [0N 1N] n))
  ([xs n]
   (if (<= n (count xs))
       xs
       (let [x' (+ (last xs)
```

```

        (nth xs (- (count xs) 2)))
  xs' (conj xs x')])
(recur xs' n))))
```

Effectively, the `fibo-recur` function can perform an infinite number of nested recursive calls. We can observe that this function does not blow up the stack for large values of `n`, shown as follows:

```

user> (fibo-recur 10)
[0N 1N 1N 2N 3N 5N 8N 13N 21N 34N]
user> (last (fibo-recur 10000))
207936...230626N
```

We should note that a call to `fibo-recur` can take quite a while to terminate for large values of `n`. We can measure the time taken for a call to `fibo-recur` to complete and return a value, using the `time` macro, as follows:

```

user> (time (last (fibo-recur 10000)))
"Elapsed time: 1320.050942 msecs"
207936...230626N
```

The `fibo-recur` function can also be expressed using the `loop` and `recur` forms. This eliminates the need for using a second function arity to pass the `[0N 1N]` value around, as shown in the `fibo-loop` function defined in *Example 1.3*:

```

(defn fibo-loop [n]
  (loop [xs [0N 1N]
         n n]
    (if (<= n (count xs))
        xs
        (let [x' (+ (last xs)
                     (nth xs (- (count xs) 2)))
              xs' (conj xs x') ]
          (recur xs' n)))))
```

*Example 1.3: A recursive function defined using loop and recur*

Note that the `loop` macro requires a vector of bindings (pairs of names and

values) to be passed as its first argument. The second argument to the `loop` form must be an expression that uses the `recur` form. This nested `recur` form calls the surrounding expression recursively by passing in the new values for the declared bindings in the `loop` form. The `fibo-loop` function returns a value that is equal to that returned by the `fibo-recur` function, from *Example 1.2*, shown as follows:

```
user> (fibo-loop 10)
[0N 1N 1N 2N 3N 5N 8N 13N 21N 34N]
user> (last (fibo-loop 10000))
207936...230626N
```

Another way to handle recursion is by using the `trampoline` function. The `trampoline` function takes a function as its first argument, followed by the values of the parameters to be passed to the supplied function. A `trampoline` form expects the supplied function to return another function, and in such a case, the returned function will be invoked. Thus, a `trampoline` form manages recursion by obtaining a return value, and invoking the returned value again if it's a function. Thus, the `trampoline` function avoids using any stack space. Each time the supplied function is invoked, it returns and the result gets stored in the process heap. For example, consider the function in *Example 1.4* that calculates the first  $n$  numbers of the Fibonacci sequence using a `trampoline`:

```
(defn fibo-trampoline [n]
  (letfn [(fibo-fn [xs n]
            (if (<= n (count xs))
                xs
                (let [x' (+ (last xs)
                            (nth xs (- (count xs) 2)))
                     xs' (conj xs x') ]
                  #(fibo-fn xs' n))))]
    (trampoline fibo-fn [0N 1N] n)))
```

*Example 1.4: A recursive function defined using trampoline*

In the `fib-trampoline` function, the internal `fibo-fn` function returns either a sequence, denoted by `xs`, or a closure that takes no arguments,

represented by `#(fib-trampoline xs' n)`. This function is equivalent to the `fib-recur` function we defined earlier, even in terms of performance, shown as follows:

```
user> (fib-trampoline 10)
[0N 1N 1N 2N 3N 5N 8N 13N 21N 34N]
user> (time (last (fib-trampoline 10000)))
"Elapsed time: 1346.629108 msecs"
207936...230626N
```

*Mutual recursion* can also be handled effectively using a trampoline. In mutual recursion, two functions call each other in a recursive manner. For example, consider the function that utilizes two mutually recursive functions in *Example 1.5*:

```
(defn sqrt-div2-recur [n]
  (letfn [(sqrt [n]
            (if (< n 1)
                n
                (div2 (Math/sqrt n))))
          (div2 [n]
            (if (< n 1)
                n
                (sqrt (/ n 2))))]
    (sqrt n)))
```

### *Example 1.5: A simple function that uses mutual recursion*

The `sqrt-div2-recur` function from *Example 1.5* defines two mutually recursive functions internally, namely `sqrt` and `div2`, that repeatedly square root and halve a given value `n` until the calculated value is less than 1. The `sqrt-div2-recur` function declares these two functions using a `letfn` form and invokes the `sqrt` function. We can convert this to use a `trampoline` form as shown in *Example 1.6*:

```
(defn sqrt-div2-trampoline [n]
  (letfn [(sqrt [n]
            (if (< n 1)
                n
```

```

#(div2 (Math/sqrt n)))
(div2 [n]
  (if (< n 1)
    n
    #(sqrt (/ n 2))))]
(trampoline sqrt n)))

```

*Example 1.6: A function that uses mutual recursion using trampoline*

In the previous `sqrt-div2-trampoline` function shown, the functions `sqrt` and `div2` return closures instead of calling a function directly. The `trampoline` form in the body of the function calls the `sqrt` function while supplying the value `n`. Both the `sqrt-div2-recur` and `sqrt-div2-trampoline` functions take about the same time to return a value for the given value of `n`. Hence, using a `trampoline` form does not have any additional performance overhead, shown as follows:

```

user> (time (sqrt-div2-recur 1000000000N))
"Elapsed time: 0.327439 msecs"
0.5361105866719398
user> (time (sqrt-div2-trampoline 1000000000N))
"Elapsed time: 0.326081 msecs"
0.5361105866719398

```

As the preceding examples demonstrate, there are various ways to define recursive functions in Clojure. Recursive functions can be optimized using tail call elimination, by using `recur`, and mutual recursion, which is done using the `trampoline` function.

# Thinking in sequences

A **sequence**, shortened as a **seq**, is essentially an abstraction of a list. This abstraction provides a unified model or interface to interact with a collection of items. In Clojure, all the primitive data structures, namely strings, lists, vectors, maps, and sets can be treated as sequences. In practice, almost everything that involves iteration can be translated into a sequence of computations. A collection is termed as **seqable** if it implements the abstraction of a sequence. We will learn everything there is to know about sequences in this section.

Sequences can also be *lazy*. A lazy sequence can be thought of as a possibly infinite series of computed values. The computation of each value is deferred until it is actually needed. We should note that the computation of a recursive function can easily be represented as a lazy sequence. For example, the Fibonacci sequence can be computed by lazily adding the last two elements in the previously computed sequence. This can be implemented as shown in *Example 1.7*.

## Note

The following examples can be found in `src/m_c1j/c1/seq.clj` of the book's source code.

```
(defn fibo-lazy [n]
  (->> [0N 1N]
        (iterate (fn [[a b]] [b (+ a b)]))
        (map first)
        (take n)))
```

*Example 1.7: A lazy Fibonacci sequence*

## Note

The threading macro `->>` is used to pass the result of a given expression as

the last argument to the next expression, in a repetitive manner for all expressions in its body. Similarly, the threading macro `->` is used to pass the result of a given expression as the first argument to the subsequent expressions.

The `fibo-lazy` function from *Example 1.7* uses the `iterate`, `map`, and `take` functions to create a lazy sequence. We will study these functions in more detail later in this section. The `fibo-lazy` function takes a single argument `n`, which indicates the number of items to be returned by the function. In the `fibo-lazy` function, the values `0N` and `1N` are passed as a vector to the `iterate` function, which produces a lazy sequence. The function used for this iteration creates a new pair of values `b` and `(+ a b)` from the initial values `a` and `b`.

Next, the `map` function applies the `first` function to obtain the first element in each resulting vector. A `take` form is finally applied to the sequence returned by the `map` function to retrieve the first `n` values in the sequence. The `fibo-lazy` function does not cause any error even when passed relatively large values of `n`, shown as follows:

```
user> (fibo-lazy 10)
(0N 1N 1N 2N 3N 5N 8N 13N 21N 34N)
user> (last (fibo-lazy 10000))
207936...230626N
```

Interestingly, the `fibo-lazy` function in *Example 1.7* performs significantly better than the recursive functions from *Example 1.2* and *Example 1.3*, as shown here:

```
user> (time (last (fibo-lazy 10000)))
"Elapsed time: 18.593018 msecs"
207936...230626N
```

Also, binding the value returned by the `fibo-lazy` function to a variable does not really consume any time. This is because this returned value is lazy and not evaluated yet. Also, the type of the return value is

`clojure.lang.LazySeq`, as shown here:

```
user> (time (def fibo-xs (fibolazy 10000)))
"Elapsed time: 0.191981 msecs"
#'user/fibo-xs
user> (type fibo-xs)
clojure.lang.LazySeq
```

We can optimize the `fibo-lazy` function even further by using **memoization**, which essentially caches the value returned by a function for a given set of inputs. This can be done using the `memoize` function, as follows:

```
(def fibo-mem (memoize fibolazy))
```

The `fibo-mem` function is a memoized version of the `fibo-lazy` function. Hence, subsequent calls to the `fibo-mem` function for the same set of inputs will return values significantly faster, shown as follows:

```
user> (time (last (fibo-mem 10000)))
"Elapsed time: 19.776527 msecs"
207936...230626N
user> (time (last (fibo-mem 10000)))
"Elapsed time: 2.82709 msecs"
207936...230626N
```

Note that the `memoize` function can be applied to any function, and it is not really related to sequences. The function we pass to `memoize` must be free of side effects, or else any side effects will be invoked only the first time the memoized function is called with a given set of inputs.

## Using the seq library

Sequences are a truly ubiquitous abstraction in Clojure. The primary motivation behind using sequences is that any domain with sequence-like data in it can be easily modelled using the standard functions that operate on sequences. This infamous quote from the Lisp world reflects on this design:

*"It is better to have 100 functions operate on one data abstraction than 10 functions on 10 data structures."*

A sequence can be constructed using the `cons` function. We must provide an element and another sequence as arguments to the `cons` function. The `first` function is used to access the first element in a sequence, and similarly the `rest` function is used to obtain the other elements in the sequence, shown as follows:

```
user> (def xs (cons 0 '(1 2 3)))
#'user/xs
user> (first xs)
0
user> (rest xs)
(1 2 3)
```

## Note

The `first` and `rest` functions in Clojure are equivalent to the `car` and `cdr` functions, respectively, from traditional Lisps. The `cons` function carries on its traditional name.

In Clojure, an empty list is represented by the literal `()`. An empty list is considered as a *truthy* value, and does not equate to `nil`. This rule is true for any empty collection. An empty list does indeed have a type – it's a list. On the other hand, the `nil` literal signifies the absence of a value, of any type, and is not a truthy value. The second argument that is passed to `cons` could be empty, in which case the resulting sequence would contain a single element:

```
user> (cons 0 ())
()
user> (cons 0 nil)
()
user> (rest (cons 0 nil))
()
```

An interesting quirk is that `nil` can be treated as an empty collection, but

the converse is not true. We can use the `empty?` and `nil?` functions to test for an empty collection and a `nil` value, respectively. Note that `(empty? nil)` returns `true`, shown as follows:

```
user> (empty? ())
true
user> (empty? nil)
true
user> (nil? ())
false
user> (nil? nil)
true
```

## Note

By the *truthy* value, we mean to say a value that will test positive in a conditional expression such as an `if` or a `when` form.

The `rest` function will return an empty list when supplied an empty list. Thus, the value returned by `rest` is always truthy. The `seq` function can be used to obtain a sequence from a given collection. It will return `nil` for an empty list or collection. Hence, the `head`, `rest` and `seq` functions can be used to iterate over a sequence. The `next` function can also be used for iteration, and the expression `(seq (rest coll))` is equivalent to `(next coll)`, shown as follows:

```
user> (= (rest ()) nil)
false
user> (= (seq ()) nil)
true
user> (= (next ()) nil)
true
```

The `sequence` function can be used to create a list from a sequence. For example, `nil` can be converted into an empty list using the expression `(sequence nil)`. In Clojure, the `seq?` function is used to check whether a value implements the sequence interface, namely `clojure.lang.ISeq`. Only lists implement this interface, and other data structures such as

vectors, sets, and maps have to be converted into a sequence by using the `seq` function. Hence, `seq?` will return `true` only for lists. Note that the `list?`, `vector?`, `map?`, and `set?` functions can be used to check the concrete type of a given collection. The behavior of the `seq?` function with lists and vectors can be described as follows:

```
user> (seq? '(1 2 3))
true
user> (seq? [1 2 3])
false
user> (seq? (seq [1 2 3]))
true
```

Only lists and vectors provide a guarantee of sequential ordering among elements. In other words, lists and vectors will store their elements in the same order or sequence as they were created. This is in contrast to maps and sets, which can reorder their elements as needed. We can use the `sequential?` function to check whether a collection provides sequential ordering:

```
user> (sequential? '(1 2 3))
true
user> (sequential? [1 2 3])
true
user> (sequential? {:a 1 :b 2})
false
user> (sequential? #{:a :b})
false
```

The `associative?` function can be used to determine whether a collection or sequence associates a key with a particular value. Note that this function returns `true` only for maps and vectors:

```
user> (associative? '(1 2 3))
false
user> (associative? [1 2 3])
true
user> (associative? {:a 1 :b 2})
true
user> (associative? #{:a :b})
```

```
false
```

The behavior of the `associative?` function is fairly obvious for a map since a map is essentially a collection of key-value pairs. The fact that a vector is also associative is well justified too, as a vector has an implicit key for a given element, namely the index of the element in the vector. For example, the `[:a :b]` vector has two implicit keys, `0` and `1`, for the elements `:a` and `:b` respectively. This brings us to an interesting consequence – vectors and maps can be treated as functions that take a single argument, that is a key, and return an associated value, shown as follows:

```
user> ([:a :b] 1)
:b
user> ({:a 1 :b 2} :a)
1
```

Although they are not associative by nature, sets are also functions. Sets return a value contained in them, or `nil`, depending on the argument passed to them, shown as follows:

```
user> (#{} 1)
1
user> (#{} 0)
nil
```

Now that we have familiarized ourselves with the basics of sequences, let's have a look at the many functions that operate over sequences.

## Creating sequences

There are several ways to create sequences other than using the `cons` function. We have already encountered the `conj` function in the earlier examples of this chapter. The `conj` function takes a collection as its first argument, followed by any number of arguments to add to the collection. We must note that `conj` behaves differently for lists and vectors. When supplied a list, the `conj` function adds the other arguments at the head, or

start, of the list. In case of a vector, the `conj` function will insert the other arguments at the tail, or end, of the vector:

```
user> (conj [1 2 3] 4 5 6)
[1 2 3 4 5 6]
user> (conj '(1 2 3) 4 5 6)
(6 5 4 1 2 3)
```

The `concat` function can be used to join or *concatenate* any number of sequences in the order in which they are supplied, shown as follows:

```
user> (concat [1 2 3] [])
(1 2 3)
user> (concat [] [1 2 3])
(1 2 3)
user> (concat [1 2 3] [4 5 6] [7 8 9])
(1 2 3 4 5 6 7 8 9)
```

A given sequence can be reversed using the `reverse` function, shown as follows:

```
user> (reverse [1 2 3 4 5 6])
(6 5 4 3 2 1)
user> (reverse (reverse [1 2 3 4 5 6]))
(1 2 3 4 5 6)
```

The `range` function can be used to generate a sequence of values within a given integer range. The most general form of the `range` function takes three arguments—the first argument is the start of the range, the second argument is the end of the range, and the third argument is the step of the range. The step of the range defaults to 1, and the start of the range defaults to 0, as shown here:

```
user> (range 5)
(0 1 2 3 4)
user> (range 0 10 3)
(0 3 6 9)
user> (range 15 10 -1)
(15 14 13 12 11)
```

We must note that the `range` function expects the start of the range to be less than the end of the range. If the start of the range is greater than the end of the range and the step of the range is positive, the `range` function will return an empty list. For example, `(range 15 10)` will return `()`. Also, the `range` function can be called with no arguments, in which case it returns a lazy and infinite sequence starting at 0.

The `take` and `drop` functions can be used to take or drop elements in a sequence. Both functions take two arguments, representing the number of elements to take or drop from a sequence, and the sequence itself, as follows:

```
user> (take 5 (range 10))
(0 1 2 3 4)
user> (drop 5 (range 10))
(5 6 7 8 9)
```

To obtain an item at a particular position in the sequence, we should use the `nth` function. This function takes a sequence as its first argument, followed by the position of the item to be retrieved from the sequence as the second argument:

```
user> (nth (range 10) 0)
0
user> (nth (range 10) 9)
9
```

To repeat a given value, we can use the `repeat` function. This function takes two arguments and repeats the second argument the number of times indicated by the first argument:

```
user> (repeat 10 0)
(0 0 0 0 0 0 0 0 0 0)
user> (repeat 5 :x)
(:x :x :x :x :x)
```

The `repeat` function will evaluate the expression of the second argument and repeat it. To call a function a number of times, we can use the

`repeatedly` function, as follows:

```
user> (repeat 5 (rand-int 100))
(75 75 75 75 75)
user> (repeatedly 5 #(rand-int 100))
(88 80 17 52 32)
```

In this example, the `repeat` form first evaluates the `(rand-int 100)` form, before repeating it. Hence, a single value will be repeated several times. Note that the `rand-int` function simply returns a random integer between 0 and the supplied value. On the other hand, the `repeatedly` function invokes the supplied function a number of times, thus producing a new value every time the `rand-int` function is called.

A sequence can be repeated an infinite number of times using the `cycle` function. As you might have guessed, this function returns a lazy sequence to indicate an infinite series of values. The `take` function can be used to obtain a limited number of values from the resulting infinite sequence, shown as follows:

```
user> (take 5 (cycle [0]))
(0 0 0 0 0)
user> (take 5 (cycle (range 3)))
(0 1 2 0 1)
```

The `interleave` function can be used to combine any number of sequences. This function returns a sequence of the first item in each collection, followed by the second item, and so on. This combination of the supplied sequences is repeated until the shortest sequence is exhausted of values. Hence, we can easily combine a finite sequence with an infinite one to produce another finite sequence using the `interleave` function:

```
user> (interleave [0 1 2] [3 4 5 6] [7 8])
(0 3 7 1 4 8)
user> (interleave [1 2 3] (cycle [0]))
(1 0 2 0 3 0)
```

Another function that performs a similar operation is the `interpose`

function. The `interpose` function inserts a given element between the adjacent elements of a given sequence:

```
user> (interpose 0 [1 2 3])
(1 0 2 0 3)
```

The `iterate` function can also be used to create an infinite sequence. Note that we have already used the `iterate` function to create a lazy sequence in *Example 1.7*. This function takes a function `f` and an initial value `x` as its arguments. The value returned by the `iterate` function will have `(f x)` as the first element, `(f (f x))` as the second element, and so on. We can use the `iterate` function with any other function that takes a single argument, as follows:

```
user> (take 5 (iterate inc 5))
(5 6 7 8 9)
user> (take 5 (iterate #(+ 2 %) 0))
(0 2 4 6 8)
```

## Transforming sequences

There are also several functions to convert sequences into different representations or values. One of the most versatile of such functions is the `map` function. This function *maps* a given function over a given sequence, that is, it applies the function to each element in the sequence. Also, the value returned by `map` is implicitly lazy. The function to be applied to each element must be the first argument to `map`, and the sequence on which the function must be applied is the next argument:

```
user> (map inc [0 1 2 3])
(1 2 3 4)
user> (map #(* 2 %) [0 1 2 3])
(0 2 4 6)
```

Note that `map` can accept any number of collections or sequences as its arguments. In this case, the resulting sequence is obtained by passing the first items of the sequences as arguments to the given function, and then passing the second items of the sequences to the given function, and so on

until any of the supplied sequences are exhausted. For example, we can sum the corresponding elements of two sequences using the `map` and `+` functions, as shown here:

```
user> (map + [0 1 2 3] [4 5 6])
(4 6 8)
```

The `mapv` function has the same semantics of `map`, but returns a vector instead of a sequence, as shown here:

```
user> (mapv inc [0 1 2 3])
[1 2 3 4]
```

Another variant of the `map` function is the `map-indexed` function. This function expects that the supplied function will accept two arguments—one for the index of a given element and another for the actual element in the list:

```
user> (map-indexed (fn [i x] [i x]) "Hello")
([0 \H] [1 \e] [2 \l] [3 \l] [4 \o])
```

In this example, the function supplied to `map-indexed` simply returns its arguments as a vector. An interesting point that we can observe from the preceding example is that a string can be treated as a sequence of characters.

The `mapcat` function is a combination of the `map` and `concat` function. This function maps a given function over a sequence, and applies the `concat` function on the resulting sequence:

```
user> (require '[clojure.string :as cs])
nil
user> (map #(cs/split % #"\\d") ["aa1bb" "cc2dd" "ee3ff"])
(["aa" "bb"] ["cc" "dd"] ["ee" "ff"])
user> (mapcat #(cs/split % #"\\d") ["aa1bb" "cc2dd" "ee3ff"])
("aa" "bb" "cc" "dd" "ee" "ff")
```

In this example, we use the `split` function from the `clojure.string`

namespace to split a string using a regular expression, shown as `#"\d"`. The `split` function will return a vector of strings, and hence the `mapcat` function returns a sequence of strings instead of a sequence of vectors like the `map` function.

The `reduce` function is used to combine or *reduce* a sequence of items into a single value. The `reduce` function requires a function as its first argument and a sequence as its second argument. The function supplied to `reduce` must accept two arguments. The supplied function is first applied to the first two elements in the given sequence, and then applied to the previous result and the third element in the sequence, and so on until the sequence is exhausted. The `reduce` function also has a second arity, which accepts an initial value, and in this case, the supplied function is applied to the initial value and the first element in the sequence as the first step. The `reduce` function can be considered equivalent to loop-based iteration in imperative programming languages. For example, we can compute the sum of all elements in a sequence using `reduce`, as follows:

```
user> (reduce + [1 2 3 4 5])
15
user> (reduce + [])
0
user> (reduce + 1 [])
1
```

In this example, when the `reduce` function is supplied an empty collection, it returns 0, since `(+)` evaluates to 0. When an initial value of 1 is supplied to the `reduce` function, it returns 1, since `(+ 1)` returns 1.

A *list comprehension* can be created using the `for` macro. Note that a `for` form will be translated into an expression that uses the `map` function. The `for` macro needs to be supplied a vector of bindings to any number of collections, and an expression in the body. This macro binds the supplied symbol to each element in its corresponding collection and evaluates the body for each element. Note that the `for` macro also supports a `:let` clause to assign a value to a variable, and also a `:when` clause to filter out

values:

```
user> (for [x (range 3 7)]
          (* x x))
(9 16 25 36)
user> (for [x [0 1 2 3 4 5]
            :let [y (* x 3)]
            :when (even? y)
            y)
(0 6 12)
```

The `for` macro can also be used over a number of collections, as shown here:

```
user> (for [x ['a 'b 'c]
            y [1 2 3]]
            [x y])
([a 1] [a 2] [a 3] [b 1] [b 2] [b 3] [c 1] [c 2] [c 3])
```

The `doseq` macro has semantics similar to that of `for`, except for the fact that it always returns a `nil` value. This macro simply evaluates the body expression for all of the items in the given bindings. This is useful in forcing evaluation of an expression with side effects for all the items in a given collection:

```
user> (doseq [x (range 3 7)]
              (* x x))
nil
user> (doseq [x (range 3 7)]
              (println (* x x)))
9
16
25
36
nil
```

As shown in the preceding example, both the first and second `doseq` forms return `nil`. However, the second form prints the value of the expression `(* x x)`, which is a side effect, for all items in the sequence `(range 3 7)`.

The `into` function can be used to easily convert between types of collections. This function requires two collections to be supplied to it as arguments, and returns the first collection filled with all the items in the second collection. For example, we can convert a sequence of vectors into a map, and vice versa, using the `into` function, shown here:

```
user> (into {} [:a 1] [:c 3] [:b 2])
{:a 1, :c 3, :b 2}
user> (into [] {1 2 3 4})
[[1 2] [3 4]]
```

We should note that the `into` function is essentially a composition of the `reduce` and `conj` functions. As `conj` is used to fill the first collection, the value returned by the `into` function will depend on the type of the first collection. The `into` function will behave similar to `conj` with respect to lists and vectors, shown here:

```
user> (into [1 2 3] '(4 5 6))
[1 2 3 4 5 6]
user> (into '(1 2 3) '(4 5 6))
(6 5 4 1 2 3)
```

A sequence can be partitioned into smaller ones using the `partition`, `partition-all` and `partition-by` functions. Both the `partition` and `partition-all` functions take two arguments—one for the number of items `n` in the partitioned sequences and another for the sequence to be partitioned. However, the `partition-all` function will also return the items from the sequence, which have not been partitioned as a separate sequence, shown here:

```
user> (partition 2 (range 11))
((0 1) (2 3) (4 5) (6 7) (8 9))
user> (partition-all 2 (range 11))
((0 1) (2 3) (4 5) (6 7) (8 9) (10))
```

The `partition` and `partition-all` functions also accept a `step` argument, which defaults to the supplied number of items in the partitioned sequences, shown as follows:

```
user> (partition 3 2 (range 11))
((0 1 2) (2 3 4) (4 5 6) (6 7 8) (8 9 10))
user> (partition-all 3 2 (range 11))
((0 1 2) (2 3 4) (4 5 6) (6 7 8) (8 9 10) (10))
```

The `partition` function also takes a second sequence as an optional argument, which is used to pad the sequence to be partitioned in case there are items that are not partitioned. This second sequence has to be supplied after the step argument to the `partition` function. Note that the padding sequence is only used to create a single partition with the items that have not been partitioned, and the rest of the padding sequence is discarded. Also, the padding sequence is only used if there are any items that have not been partitioned. This can be illustrated in the following example:

```
user> (partition 3 (range 11))
((0 1 2) (3 4 5) (6 7 8))
user> (partition 3 3 (range 11 12) (range 11))
((0 1 2) (3 4 5) (6 7 8) (9 10 11))
user> (partition 3 3 (range 11 15) (range 11))
((0 1 2) (3 4 5) (6 7 8) (9 10 11))
user> (partition 3 4 (range 11 12) (range 11))
((0 1 2) (4 5 6) (8 9 10))
```

In this example, we first provide a padding sequence in the second statement as `(range 11 12)`, which only comprises of a single element. In the next statement, we supply a larger padding sequence, as `(range 11 15)`, but only the first item `11` from the padding sequence is actually used. In the last statement, we also supply a padding sequence but it is never used, as the `(range 11)` sequence is partitioned into sequences of 3 elements each with a step of 4, which will have no remaining items.

The `partition-by` function requires a higher-order function to be supplied to it as the first argument, and will partition items in the supplied sequence based on the return value of applying the given function to each element in the sequence. The sequence is essentially partitioned by `partition-by` whenever the given function returns a new value, as shown here:

```
user> (partition-by #(= 0 %) [-2 -1 0 1 2])
```

```
((-2 -1) (0) (1 2))
user> (partition-by identity [-2 -1 0 1 2])
((-2) (-1) (0) (1) (2))
```

In this example, the second statement partitions the given sequence into sequences that each contain a single item as we have used the `identity` function, which simply returns its argument. For the `[-2 -1 0 1 2]` sequence, the `identity` function returns a new value for each item in the sequence and hence the resulting partitioned sequences all have a single element.

The `sort` function can be used to change the ordering of elements in a sequence. The general form of this function requires a function to compare items and a sequence of items to sort. The supplied function defaults to the `compare` function, whose behavior changes depending on the actual type of the items being compared:

```
user> (sort [3 1 2 0])
(0 1 2 3)
user> (sort > [3 1 2 0])
(3 2 1 0)
user> (sort ["Carol" "Alice" "Bob"])
("Alice" "Bob" "Carol")
```

If we intend to apply a particular function to each item in a sequence before performing the comparison in a `sort` form, we should consider using the `sort-by` function for a more concise expression. The `sort-by` function also accepts a function to perform the actual comparison, similar to the `sort` function. The `sort-by` function can be demonstrated as follows:

```
user> (sort #(compare (first %1) (first %2)) [[1 1] [2 2] [3 3]])
([[1 1] [2 2] [3 3]])
user> (sort-by first [[1 1] [2 2] [3 3]])
([[1 1] [2 2] [3 3]])
user> (sort-by first > [[1 1] [2 2] [3 3]])
([3 3] [2 2] [1 1])
```

In this example, the first and second statements both compare items after applying the `first` function to each item in the given sequence. The last statement passes the `>` function to the `sort-by` function, which returns the reverse of the sequence returned by the first two statements.

## Filtering sequences

Sequences can also be *filtered*, that is transformed by removing some elements from the sequence. There are several standard functions to perform this task. The `keep` function can be used to remove values from a sequence that produces a `nil` value for a given function. The `keep` function requires a function and a sequence to be passed to it. The `keep` function will apply the given function to each item in the sequence and remove all values that produce `nil`, as shown here:

```
user> (keep #(if (odd? %) %) (range 10))
(1 3 5 7 9)
user> (keep seq [() [] '(1 2 3) [:a :b] nil])
((1 2 3) (:a :b))
```

In this example, the first statement removes all even numbers from the given sequence. In the second statement, the `seq` function is used to remove all empty collections from the given sequence.

A map or a set can also be passed as the first argument to the `keep` function since they can be treated as functions, as shown here:

```
user> (keep {:a 1, :b 2, :c 3} [:a :b :d])
(1 2)
user> (keep #{0 1 2 3} #{2 3 4 5})
(3 2)
```

The `filter` function can also be used to remove some elements from a given sequence. The `filter` function expects a predicate function to be passed to it along with the sequence to be filtered. The items for which the predicate function does not return a truthy value are removed from the result. The `filterv` function is identical to the `filter` function, except for

the fact that it returns a vector instead of a list:

```
user> (filter even? (range 10))
(0 2 4 6 8)
user> (filterv even? (range 10))
[0 2 4 6 8]
```

Both the `filter` and `keep` functions have similar semantics. However, the primary distinction is that the `filter` function returns a subset of the original elements, whereas `keep` returns a sequence of non `nil` values that are returned by the function supplied to it, as shown in the following example:

```
user> (keep #(if (odd? %) %) (range 10))
(1 3 5 7 9)
user> (filter odd? (range 10))
(1 3 5 7 9)
```

Note that in this example, if we passed the `odd?` function to the `keep` form, it would return a list of `true` and `false` values, as these values are returned by the `odd?` function.

Also, a `for` macro with a `:when` clause is translated into an expression that uses the `filter` function, and hence a `for` form can also be used to remove elements from a sequence:

```
user> (for [x (range 10) :when (odd? x)] x)
(1 3 5 7 9)
```

A vector can be *sliced* using the `subvec` function. By sliced, we mean to say that a smaller vector is selected from the original vector depending on the values passed to the `subvec` function. The `subvec` function takes a vector as its first argument, followed by the index indicating the start of the sliced vector, and finally another optional index that indicates the end of the sliced vector, as shown here:

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

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

Maps can be filtered by their keys using the `select-keys` function. This function requires a map as the first argument and a vector of keys as a second argument to be passed to it. The vector of keys passed to this function indicates the key-value pairs to be included in the resulting map, as shown here:

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

Another way to select key-value pairs from a map is to use the `find` function, as shown here:

```
user> (find {:a 1 :b 2} :a)
[:a 1]
```

`take-while` and `drop-while` are analogous to the `take` and `drop` functions, and require a predicate to be passed to them, instead of the number of elements to take or drop. The `take-while` function takes elements as long as the predicate function returns a truthy value, and similarly the `drop-while` function will drop elements for the same condition:

```
user> (take-while neg? [-2 -1 0 1 2])
(-2 -1)
user> (drop-while neg? [-2 -1 0 1 2])
(0 1 2)
```

## Lazy sequences

`lazy-seq` and `lazy-cat` are the most elementary constructs to create lazy sequences. The value returned by these functions will always have the type `clojure.lang.LazySeq`. The `lazy-seq` function is used to wrap a lazily computed expression in a `cons` form. This means that the rest of the sequence created by the `cons` form is lazily computed. For example, the

`lazy-seq` function can be used to construct a lazy sequence representing the Fibonacci sequence as shown in *Example 1.8*:

```
(defn fibo-cons [a b]
  (cons a (lazy-seq (fibo-cons b (+ a b)))))
```

*Example 1.8: A lazy sequence created using `lazy-seq`*

The `fibo-cons` function requires two initial values, `a` and `b`, to be passed to it as the initial values, and returns a lazy sequence comprising the first value `a` and a lazily computed expression that uses the next two values in the sequence, that is, `b` and `(+ a b)`. In this case, the `cons` form will return a lazy sequence, which can be handled using the `take` and `last` functions, as shown here:

```
user> (def fibo (fibo-cons 0N 1N))
#'user/fibo
user> (take 2 fibo)
(0N 1N)
user> (take 11 fibo)
(0N 1N 1N 2N 3N 5N 8N 13N 21N 34N 55N)
user> (last (take 10000 fibo))
207936...230626N
```

Note that the `fibo-cons` function from *Example 1.8* recursively calls itself without an explicit `recur` form, and yet it does not consume any stack space. This is because the values present in a lazy sequence are not stored in a call stack, and all the values are allocated on the process heap.

Another way to define a lazy Fibonacci sequence is by using the `lazy-cat` function. This function essentially concatenates all the sequences it is supplied in a lazy fashion. For example, consider the definition of the Fibonacci sequence in *Example 1.9*:

```
(def fibo-seq
  (lazy-cat [0N 1N] (map + fibo-seq (rest fibo-seq))))
```

*Example 1.9: A lazy sequence created using `lazy-cat`*

The `fibo-seq` variable from *Example 1.9* essentially calculates the Fibonacci sequence using a lazy composition of the `map`, `rest`, and `+` functions. Also, a sequence is required as the initial value, instead of a function as we saw in the definition of `fibo-cons` from *Example 1.8*. We can use the `nth` function to obtain a number from this sequence as follows:

```
user> (first fibo-seq)
ON
user> (nth fibo-seq 1)
1N
user> (nth fibo-seq 10)
55N
user> (nth fibo-seq 9999)
207936...230626N
```

As shown previously, `fibo-cons` and `fibo-seq` are concise and idiomatic representations of the infinite series of numbers in the Fibonacci sequence. Both of these definitions return identical values and do not cause an error due to stack consumption.

An interesting fact is that most of the standard functions that return sequences, such as `map` and `filter`, are inherently lazy. Any expression that is built using these functions is lazy, and hence never evaluated until needed. For example, consider the following expression that uses the `map` function:

```
user> (def xs (map println (range 3)))
#'user/xs
user> xs
0
1
2
(nil nil nil)
```

In this example, the `println` function is not called when we define the `xs` variable. However, once we try to print it in the REPL, the sequence is evaluated and the numbers are printed out by calling the `println` function. Note that `xs` evaluates to `(nil nil nil)` as the `println` function always

returns `nil`.

Sometimes, it is necessary to eagerly evaluate a lazy sequence. The `doall` and `dorun` functions are used for this exact purpose. The `doall` function essentially forces evaluation of a lazy sequence along with any side effects of the evaluation. The value returned by `doall` is a list of all the elements in the given lazy sequence. For example, let's wrap the `map` expression from the previous example in a `doall` form, shown as follows:

```
user> (def xs (doall (map println (range 3))))  
0  
1  
2  
# 'user/xs  
user> xs  
(nil nil nil)
```

Now, the numbers are printed out as soon as `xs` is defined, as we force evaluation using the `doall` function. The `dorun` function has similar semantics as the `doall` function, but it always returns `nil`. Hence, we can use the `dorun` function instead of `doall` when we are only interested in the side effects of evaluating the lazy sequence, and not the actual values in it. Another way to call a function with some side effects over all values in a collection is by using the `run!` function, which must be passed a function to call and a collection. The `run!` function always returns `nil`, just like the `dorun` form.

## Using zippers

Now that we are well versed with sequences, let's briefly examine **zippers**. Zippers are essentially data structures that help in traversing and manipulating *trees*. In Clojure, any collection that contains nested collections is termed as a tree. A zipper can be thought of as a structure that contains location information about a tree. Zippers are not an extension of trees, but rather can be used to traverse and realize a tree.

### Note

The following namespaces must be included in your namespace declaration for the upcoming examples:

```
(ns my-namespace
  (:require [clojure.zip :as z]
            [clojure.xml :as xml]))
```

The following examples can be found in `src/m_clj/c1/zippers.clj` of the book's source code.

We can define a simple tree using vector literals, as shown here:

```
(def tree [:a [1 2 3] :b :c])
```

The vector `tree` is a tree, comprised of the nodes `:a`, `[1 2 3]`, `:b`, and `:c`. We can use the `vector-zip` function to create a zipper from the vector `tree` as follows:

```
(def root (z/vector-zip tree))
```

The variable `root` defined previously is a zipper and contains location information for traversing the given tree. Note that the `vector-zip` function is simply a combination of the standard `seq` function and the `seq-zip` function from the `clojure.zip` namespace. Hence, for trees that are represented as sequences, we should use the `seq-zip` function instead. Also, all other functions in the `clojure.zip` namespace expect their first argument to be a zipper.

To traverse the zipper, we must use the `clojure.zip/next` function, which returns the next node in the zipper. We can easily iterate over all the nodes in the zipper using a composition of the `iterate` and `clojure.zip/next` functions, as shown here:

```
user> (def tree-nodes (iterate z/next root))
#'user/tree-nodes
user> (nth tree-nodes 0)
[[{:a [1 2 3] :b :c} nil]
user> (nth tree-nodes 1)
```

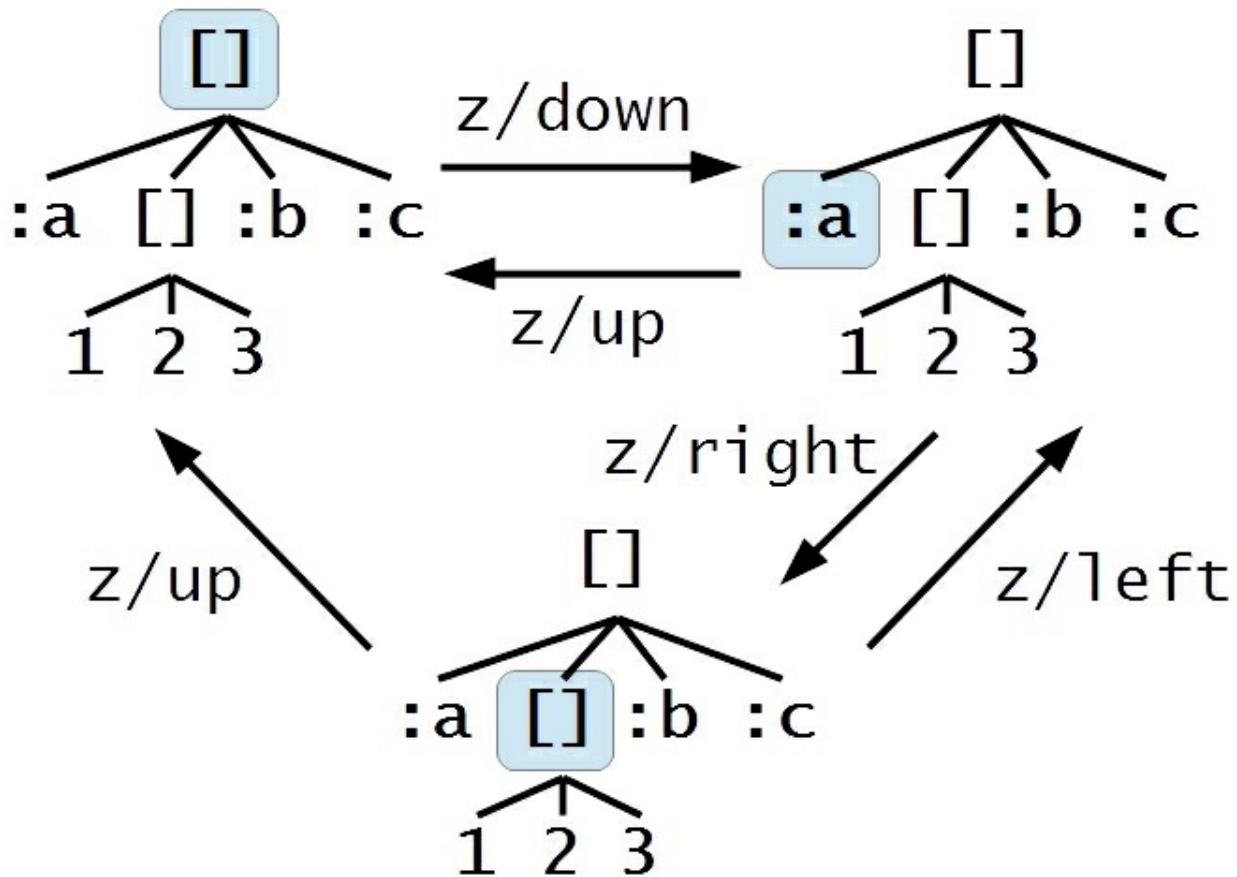
```
[:a {:l [], :pnodes ... }]  
user> (nth tree-nodes 2)  
[[1 2 3] {:l [:a], :pnodes ... }]  
user> (nth tree-nodes 3)  
[1 {:l [], :pnodes ... }]
```

As shown previously, the first node of the zipper represents the original tree itself. Also, the zipper will contain some extra information, other than the value contained in the current node, which is useful in navigating across the given tree. In fact, the return value of the `next` function is also a zipper. Once we have completely traversed the given tree, a zipper pointing to the root of the tree will be returned by the `next` function. Note that some information in a zipper has been truncated from the preceding REPL output for the sake of readability.

To navigate to the adjacent nodes in a given zipper, we can use the `down`, `up`, `left`, and `right` functions. All of these functions return a zipper, as shown here:

```
user> (-> root z/down)  
[:a {:l [], :pnodes ... }]  
user> (-> root z/down z/right)  
[[1 2 3] {:l [:a], :pnodes ... }]  
user> (-> root z/down z/right z/up)  
[[{:a [1 2 3] :b :c} nil]  
user> (-> root z/down z/right z/right)  
[:b {:l [:a [1 2 3]], :pnodes ... }]  
user> (-> root z/down z/right z/left)  
[:a {:l [], :pnodes ... }]
```

The `down`, `up`, `left`, and `right` functions change the location of the `root` zipper in the `[{:a [1 2 3] :b :c}]` tree, as shown in the following illustration:



The preceding diagram shows a zipper at three different locations in the given tree. Initially, the location of the zipper is at the root of the tree, which is the entire vector. The `down` function moves the location to the first child node in the tree. The `left` and `right` functions move the location of the zipper to other nodes at the same level or depth in the tree. The `up` function moves the zipper to the parent of the node pointed to by the zipper's current location.

To obtain the node representing the current location of a zipper in a tree, we must use the `node` function, as follows:

```

user> (-> root z/down z/right z/right z/node)
:b
user> (-> root z/down z/right z/left z/node)

```

```
:a
```

To navigate to the extreme left or right of a tree, we can use the `leftmost` and `rightmost` functions, respectively, as shown here:

```
user> (-> root z/down z/rightmost z/node)
:c
user> (-> root z/down z/rightmost z/leftmost z/node)
:a
```

The `lefts` and `rights` functions return the nodes that are present to the left and right, respectively, of a given zipper, as follows:

```
user> (-> root z/down z/rights)
([1 2 3] :b :c)
user> (-> root z/down z/lefts)
nil
```

As the `:a` node is the leftmost element in the tree, the `rights` function will return all of the other nodes in the tree when passed a zipper that has `:a` as the current location. Similarly, the `lefts` function for the zipper at the `:a` node will return an empty value, that is `nil`.

The `root` function can be used to obtain the root of a given zipper. It will return the original tree used to construct the zipper, as shown here:

```
user> (-> root z/down z/right z/root)
[:a [1 2 3] :b :c]
user> (-> root z/down z/right r/left z/root)
[:a [1 2 3] :b :c]
```

The `path` function can be used to obtain the path from the root element of a tree to the current location of a given zipper, as shown here:

```
user> (def e (-> root z/down z/right z/down))
#'user/e
user> (z/node e)
1
user> (z/path e)
[[[:a [1 2 3] :b :c]
```

```
[1 2 3]
```

In the preceding example, the path of the 1 node in `tree` is represented by a vector containing the entire tree and the subtree [1 2 3]. This means that to get to the 1 node, we must pass through the root and the subtree [1 2 3].

Now that we have covered the basics of navigating across trees, let's see how we can modify the original tree. The `insert-child` function can be used to insert a given element into a tree as follows:

```
user> (-> root (z	insert-child :d) z/root)
[:d :a [1 2 3] :b :c]
user> (-> root z/down z/right (z	insert-child 0) z/root)
[:a [0 1 2 3] :b :c]
```

We can also remove a node from the zipper using the `remove` function. Also, the `replace` function can be used to replace a given node in a zipper:

```
user> (-> root z/down z/remove z/root)
[[1 2 3] :b :c]
user> (-> root z/down (z/replace :d) z/root)
[:d [1 2 3] :b :c]
```

One of the most noteworthy examples of tree-like data is XML. Since zippers are great at handling trees, they also allow us to easily traverse and modify XML content. Note that Clojure already provides the `xml-seq` function to convert XML data into a sequence. However, treating an XML document as a sequence has many strange implications.

One of the main disadvantages of using `xml-seq` is that there is no easy way to get to the root of the document from a node if we are iterating over a sequence. Also, `xml-seq` only helps us iterate over the XML content; it doesn't deal with modifying it. These limitations can be overcome using zippers, as we will see in the upcoming example.

For example, consider the following XML document:

```

<countries>
  <country name="England">
    <city>Birmingham</city>
    <city>Leeds</city>
    <city capital="true">London</city>
  </country>
  <country name="Germany">
    <city capital="true">Berlin</city>
    <city>Frankfurt</city>
    <city>Munich</city>
  </country>
  <country name="France">
    <city>Cannes</city>
    <city>Lyon</city>
    <city capital="true">Paris</city>
  </country>
</countries>

```

The document shown above contains countries and cities represented as XML nodes. Each country has a number of cities, and a single city as its capital. Some information, such as the name of the country and a flag indicating whether a city is a capital, is encoded in the XML attributes of the nodes.

## Note

The following example expects the XML content shown previously to be present in the `resources/data/sample.xml` file, relative to the root of your Leiningen project.

Let's define a function to find out all the capital cities in the document, as shown in *Example 1.10*:

```

(defn is-capital-city? [n]
  (and (= (:tag n) :city)
       (= "true" (:capital (:attrs n)))))

(defn find-capitals [file-path]
  (let [xml-root (z/xml-zip (xml/parse file-path))
        xml-seq (iterate z/next (z/next xml-root))])

```

```
(->> xml-seq
  (take-while #(not= (z/root xml-root) (z/node %)))
  (map z/node)
  (filter is-capital-city?))
  (mapcat :content))))
```

### *Example 1.10: Querying XML with zippers*

Firstly, we must note that the `parse` function from the `clojure.xml` namespace reads an XML document and returns a map representing the document. Each node in this map is another map with the `:tag`, `:attrs`, and `:content` keys associated with the XML node's tag name, attributes, and content respectively.

In *Example 1.10*, we first define a simple function, `is-capital-city?`, to determine whether a given XML node has the `city` tag, represented as `:city`. The `is-capital-city?` function also checks whether the XML node contains the `capital` attribute, represented as `:capital`. If the value of the `capital` attribute of a given node is the "true" string, then the `is-capital-city?` function returns `true`.

The `find-capitals` function performs most of the heavy lifting in this example. This function first parses XML documents present at the supplied path `file-path`, and then converts it into a zipper using the `xml-zip` function. We then iterate over the zipper using the `next` function until we arrive back at the root node, which is checked by the `take-while` function. We then map the `node` function over the resulting sequence of zippers using the `map` function, and apply the `filter` function to find the capital cities among all the nodes. Finally, we use the `mapcat` function to obtain the XML content of the filtered nodes and flatten the resulting sequence of vectors into a single list.

When supplied a file containing the XML content we described earlier, the `find-capitals` function returns the names of all capital cities in the document:

```
user> (find-capitals "resources/data/sample.xml")
("London" "Berlin" "Paris")
```

As demonstrated previously, zippers are apt for dealing with trees and hierarchical data such as XML. More generally, sequences are a great abstraction for collections and several forms of data, and Clojure provides us with a huge toolkit for dealing with sequences. There are several more functions that handle sequences in the Clojure language, and you are encouraged to explore them on your own.

# Working with pattern matching

In this section, we will examine *pattern matching* in Clojure. Typically, functions that use conditional logic can be defined using the `if`, `when`, or `cond` forms. Pattern matching allows us to define such functions by declaring patterns of the literal values of their parameters. While this idea may appear quite rudimentary, it is a very useful and powerful one, as we shall see in the upcoming examples. Pattern matching is also a foundational programming construct in other functional programming languages.

In Clojure, there is no pattern matching support for functions and forms in the core language. However, it is a common notion among Lisp programmers that we can easily modify or extend the language using macros. Clojure takes this approach as well, and thus pattern matching is made possible using the `match` and `defun` macros. These macros are implemented in the `core.match` (<https://github.com/clojure/core.match>) and `defun` (<https://github.com/killme2008/defun>) community libraries. Both of these libraries are also supported on ClojureScript.

## Note

The following library dependencies are required for the upcoming examples:

```
[org.clojure/core.match "0.2.2"
 :exclusions [org.clojure/tools.analyzer.jvm]]
[defun "0.2.0-RC"]
```

Also, the following namespaces must be included in your namespace declaration:

```
(ns my-namespace
  (:require [clojure.core.match :as m]
            [defun :as f]))
```

The following examples can be found in `src/m_clj/c1/match.clj` of the book's source code.

Let's consider a simple example that we can model using pattern matching. The XOR logic function returns a true value only when its arguments are exclusive of each other, that is, when they have differing values. In other words, the XOR function will return false when both of its arguments have the same values. We can easily define such a function using the `match` macro, as shown in *Example 1.11*:

```
(defn xor [x y]
  (m/match [x y]
    [true true] false
    [false true] true
    [true false] true
    [false false] false))
```

*Example 1.11: Pattern matching using the match macro*

The `xor` function from *Example 1.11* simply matches its arguments, `x` and `y`, against a given set of patterns, such as `[true true]` and `[true false]`. If both the arguments are `true` or `false`, then the function returns `false`, or else it returns `true`. It's a concise definition that relies on the values of the supplied arguments, rather than the use of conditional forms such as `if` and `when`. The `xor` function can be defined alternatively, and even more concisely, by the `defun` macro, as shown in *Example 1.12*:

```
(f/defun xor
  ([true true] false)
  ([false true] true)
  ([true false] true)
  ([false false] false))
```

*Example 1.12: Pattern match using the defun macro*

The definition of the `xor` function that uses the `defun` macro simply declares the actual values as its arguments. The expression to be returned

is thus determined by the values of its inputs. Note that the `defun` macro rewrites the definition of the `xor` function to use the `match` macro. Hence, all patterns supported by the `match` macro can also be used with the `defun` macro. Both the definitions of the `xor` function, from *Example 1.11* and *Example 1.12*, work as expected, as shown here:

```
user> (xor true true)
false
user> (xor true false)
true
user> (xor false true)
true
user> (xor false false)
false
```

The `xor` function will throw an exception if we try to pass values that have not been declared as a pattern:

```
user> (xor 0 0)
IllegalArgumentException No matching clause: [0 0] user/xor ...
```

We can define a simple function to compute the  $n^{th}$  number of the Fibonacci sequence using the `defun` macro, as shown in *Example 1.13*:

```
(f/defun fibo
  ([0] 0N)
  ([1] 1N)
  ([n] (+ (fibo (- n 1))
            (fibo (- n 2)))))
```

Note the use of the variable `n` in the function's pattern rules. This signifies that any value other than `0` and `1` will match with the pattern definition that uses `n`. The `fibo` function defined in *Example 1.13* does indeed calculate the  $n^{th}$  Fibonacci sequence, as shown here:

```
user> (fibo 0)
0N
user> (fibo 1)
1N
```

```
user> (fibo 10)
```

```
55N
```

However, the definition of `fibo`, shown in *Example 1.13*, cannot be optimized by tail call elimination. This is due to the fact that the definition of `fibo` is tree recursive. By this, we mean to say that the expression `(+ (fibo ...) (fibo ...))` requires two recursive calls in order to be evaluated completely. In fact, if we replace the recursive calls to the `fibo` function with `recur` expressions, the resulting function won't compile. It is fairly simple to convert tree recursion into linear recursion, as shown in *Example 1.14*:

```
(f/defun fibo-recur
  ([a b 0] a)
  ([a b n] (recur b (+ a b) (dec n)))
  ([n] (recur 0N 1N n)))
```

*Example 1.14: A tail recursive function with pattern matching*

It is fairly obvious from the definition of the `fibo-recur` function, from *Example 1.14*, that it is indeed tail recursive. This function does not consume any stack space, and can be safely called with large values of `n`, as shown here:

```
user> (fibo-recur 0)
0N
user> (fibo-recur 1)
1N
user> (fibo-recur 10)
55N
user> (fibo-recur 9999)
207936...230626N
```

As the preceding examples show us, pattern matching is a powerful tool in functional programming. Functions that are defined using pattern matching are not only correct and expressive, but can also achieve good performance. In this respect, the `core.match` and `defun` libraries are indispensable tools in the Clojure ecosystem.

# Summary

In this chapter, we introduced a few programming constructs that can be used in the Clojure language. We've explored recursion using the `recur`, `loop`, and `trampoline` forms. We've also studied the basics of sequences and laziness, while describing the various functions in the Clojure language that are used in creating, transforming, and filtering sequences. Next, we had a look at zippers, and how they can be used to idiomatically handle trees and hierarchical data such as XML. Finally, we briefly explored pattern matching using the `core.match` and `defun` libraries.

In the next chapter, we will explore concurrency and parallelism. We will study the various data structures and functions that allow us to leverage these concepts in Clojure in ample detail.

# Chapter 2. Orchestrating Concurrency and Parallelism

Let's now examine how concurrent and parallel programming are supported in Clojure. The term *concurrent programming* refers to managing more than one task at the same time. *Parallel programming* or *parallelism*, on the other hand, deals with executing multiple tasks at the same time. The distinction between these two terms is that concurrency is about how we structure and synchronize multiple tasks, and parallelism is more about running multiple tasks in parallel over multiple cores. The main advantages of using concurrency and parallelism can be elaborated as follows:

- Concurrent programs can perform multiple tasks simultaneously. For example, a desktop application can have a single task for handling user interaction and another task for handling I/O and network communication. A single processor can be shared among several tasks. Processor utilization is thus more effective in concurrent programs.
- Parallel programs take advantage of having multiple processor cores. This means that such programs can be made to run faster by executing them on a system with more processor cores. Also, tasks that are computationally expensive can be parallelized to complete in a lesser amount of time.

In this chapter, we will:

- Explore how we can create and synchronize tasks that run concurrently
- See how to deal with a shared state between concurrent tasks
- Examine how computations can be parallelized and how we can control the amount of parallelism used to perform these computations

## Managing concurrent tasks

Clojure has a couple of handy constructs that allow us to define concurrent tasks. A *thread* is the most elementary abstraction of a task that runs in the background. In the formal sense, a thread is simply a sequence of instructions that can be scheduled for execution. A task that runs in the background of a program is said to execute on a separate thread. Threads will be scheduled for execution on a specific processor by the underlying operating system. Most modern operating systems allow a process to have several threads of execution. The technique of managing multiple threads in a single process is termed as *multithreading*.

While Clojure does support the use of threads, concurrent tasks can be modeled in more elegant ways using other constructs. Let's explore the different ways in which we can define concurrent tasks.

## Note

The following examples can be found in `src/m_clj/c2/concurrent.clj` of the book's source code.

## Using delays

A *delay* can be used to define a task whose execution is delayed, or *deferred*, until it is necessary. A delay is only run once, and its result is cached. We simply need to wrap the instructions of a given task in a `delay` form to define a delay, as shown in *Example 2.1*:

```
(def delayed-1
  (delay
    (Thread/sleep 3000)
    (println "3 seconds later ..."))
  1))
```

*Example 2.1: A delayed value*

## Note

The static `Thread/sleep` method suspends execution of the current thread of execution for a given number of milliseconds, which is passed as the first argument to this method. We can optionally specify the number of nanoseconds by which the current thread must be suspended as the second argument to the `Thread/sleep` method.

The `delay` form in *Example 2.1* simply sleeps for 3000 milliseconds, prints a string and returns the value 1. However, it is not yet *realized*, in the sense that, it has not been executed yet. The `realized?` predicate can be used to check whether a delay has been executed, as shown here:

```
user> (realized? delayed-1)
false
user> (realized? delayed-1) ; after 3 seconds
false
```

## Note

We can check whether a value is a delay using the `delay?` predicate.

The body expressions in a `delay` form will not be executed until the value returned by it is actually used. We can obtain the value contained in a delay by dereferencing it using the at-the-rate symbol (@):

```
user> @delayed-1
3 seconds later ...
1
user> (realized? delayed-1)
true
```

## Note

Using the at-the-rate symbol (@) to dereference a value is the same as using the `deref` function. For example, the expression `@x` is equivalent to `(deref x)`.

The `deref` function also has a variant form that accepts three arguments—a value to dereference, the number of milliseconds to wait before timing

out, and a value that will be returned in case of a timeout.

As shown previously, the expression `@delayed-1` returns the value `1`, after a pause of 3 seconds. Now, the call to `realized?` returns `true`. Also, the value returned by the expression `@delayed-1` will be cached, as shown here:

```
user> @delayed-1
1
```

It is thus evident that the expression `@delayed-1` will be blocked for 3 seconds, will print a string, and return a value only once.

## Note

Another way to execute a delay is by using the `force` function, which takes a delay as an argument. This function executes a given delay if needed, and returns the value of the delay's inner expression.

Delays are quite handy for representing values or tasks that need not be executed until required. However, a delay will always be executed in the same thread in which it is dereferenced. In other words, delays are *synchronous*. Hence, delays aren't really a solution for representing tasks that run in the background.

## Using futures and promises

As we mentioned earlier, threads are the most elementary way of dealing with background tasks. In Clojure, all functions implement the `clojure.lang.IFn` interface, which in turn extends the `java.lang.Runnable` interface. This means that any Clojure function can be invoked in a separate thread of execution. For example, consider the function in *Example 2.2*:

```
(defn wait-3-seconds []
  (Thread/sleep 3000)
  (println))
```

```
(println "3 seconds later ..."))
```

### *Example 2.2: A function that waits for 3 seconds*

The `wait-3-seconds` function in *Example 2.2* waits for 3000 milliseconds and prints a new line and a string. We can execute this function on a separate thread by constructing a `java.lang.Thread` object from it using the `Thread.` constructor. The resulting object can then be scheduled for execution in the background by invoking its `.start` method, as shown here:

```
user> (.start (Thread. wait-3-seconds))
nil
user>
3 seconds later ...
```

```
user>
```

The call to the `.start` method returns immediately to the REPL prompt. The `wait-3-seconds` function gets executed in the background, and prints to standard output in the REPL after 3 seconds. While using threads does indeed allow execution of tasks in the background, they have a couple shortcomings:

- There is no obvious way to obtain a return value from a function that is executed on a separate thread.
- Also, using the `Thread.` and `.start` functions is essentially interop with the underlying JVM. Thus, using these functions in a program's code would mean that the program could be run only on the JVM. We essentially lock our program into a single platform, and the program can't be run on any of the other platforms that Clojure supports.

A *future* is a more idiomatic way to represent a task that is executed in a separate thread. Futures can be concisely defined as values that will be realized in the future. A future represents a task that performs a certain computation and returns the result of the computation. We can create a future using the `future` form, as shown in *Example 2.3*:

```
(defn val-as-future [n secs]
  (future
    (Thread/sleep (* secs 1000))
    (println)
    (println (str secs " seconds later ..."))
    n))
```

### *Example 2.3: A future that sleeps for some time and returns a value*

The `val-as-future` function defined in *Example 2.3* invokes a future that waits for the number of seconds specified by the argument `secs`, prints a new line and a string, and finally returns the supplied value `n`. A call to the `val-as-future` function will return a future immediately, and a string will be printed after the specified number of seconds, as shown here:

```
user> (def future-1 (val-as-future 1 3))
#'user/future-1
user>
3 seconds later ...
```

```
user>
```

The `realized?` and `future-done?` predicates can be used to check whether a future has completed, as shown here:

```
user> (realized? future-1)
true
user> (future-done? future-1)
true
```

## Note

We can check whether a value is a future using the `future?` predicate.

A future that is being executed can be stopped by using the `future-cancel` function, which takes a future as its only argument and returns a Boolean value indicating whether the supplied future was cancelled, as depicted here:

```
user> (def future-10 (val-as-future 10 10))
```

```
#'user/future-10
user> (future-cancel future-10)
true
```

We can check whether a future has been cancelled using the `future-cancelled?` function. Also, dereferencing a future after it has been cancelled will cause an exception, as shown here:

```
user> (future-cancelled? future-10)
true
user> @future-10
CancellationException    java.util.concurrent.FutureTask.report
(FutureTask.java:121)
```

Now that we are familiar with the notion of representing tasks as futures, let's talk about how multiple futures can be synchronized. Firstly, we can use *promises* to synchronize two or more futures. A promise, created using the `promise` function, is simply a value that can be set only once. A promise is set, or *delivered*, using the `deliver` form. Subsequent calls to the `deliver` form on a promise that has been delivered will not have any effect, and will return `nil`. When a promise is not delivered, dereferencing it using the `@` symbol or the `deref` form will block the current thread of execution. Hence, a promise can be used with a future in order to pause the execution of the future until a certain value is available. The `promise` and `deliver` forms can be quickly demonstrated as follows:

```
user> (def p (promise))
#'user/p
user> (deliver p 100)
#<core$promise$reify__6363@1792b00: 100>
user> (deliver p 200)
nil
user> @p
100
```

As shown in the preceding output, the first call to the `deliver` form using the promise `p` sets the value of the promise to `100`, and the second call to the `deliver` form has no effect.

## Note

The `realized?` predicate can be used to check whether a promise instance has been delivered.

Another way to synchronize concurrent tasks is by using the `locking` form. The `locking` form allows only a single task to hold a lock variable, or a *monitor*, at any given point in time. Any value can be treated as a monitor. When a monitor is held, or *locked*, by a certain task, any other concurrent tasks that try to acquire the monitor are blocked until the monitor is available. We can thus use the `locking` form to synchronize two or more concurrent futures, as shown in *Example 2.4*:

```
(defn lock-for-2-seconds []
  (let [lock (Object.)]
    (task-1 (fn []
              (future
                (locking lock
                  (Thread/sleep 2000)
                  (println "Task 1 completed")))))
    (task-2 (fn []
              (future
                (locking lock
                  (Thread/sleep 1000)
                  (println "Task 2 completed")))))
  (task-1)
  (task-2)))
```

### *Example 2.4: Using the locking form*

The `lock-for-2-seconds` function in *Example 2.4* creates two functions, `task-1` and `task-2`, which both invoke futures that try to acquire a monitor, represented by the variable `lock`. In this example, we use a boring `java.lang.Object` instance as a monitor for synchronizing two futures. The future invoked by the `task-1` function sleeps for two seconds, whereas the future called by the `task-2` function sleeps for a single second. The future called by the `task-1` function is observed to complete

first as the future invoked by the `task-2` function will not be executed until the `locking` form in the future obtains the monitor `lock`, as shown in the following output:

```
user> (lock-for-2-seconds)
[#<core$future_call$reify_6320@19ed4e9: :pending>
 #<core$future_call$reify_6320@ac35d5: :pending>]
user>
Task 1 completed
Task 2 completed
```

We can thus use the `locking` form to synchronize multiple futures. However, the `locking` form must be used sparingly as careless use of it could result in a deadlock among concurrent tasks. Concurrent tasks are generally synchronized to pass around a shared state. Clojure allows us to avoid using the `locking` form and any possible deadlocks through the use of reference types to represent shared state, as we will examine in the following section.

# Managing state

A program can be divided into several parts which can execute concurrently. It is often necessary to share data or state among these concurrently running tasks. Thus, we arrive at the notion of having multiple observers for some data. If the data gets modified, we must ensure that the changes are visible to all observers. For example, suppose there are two threads that read data from a common variable. This data gets modified by one thread, and the change must be propagated to the other thread as soon as possible to avoid inconsistencies.

Programming languages that support mutability handle this problem by locking over a monitor, as we demonstrated with the `locking` form, and maintaining local copies of the data. In such languages, a variable is just a container for data. Whenever a concurrent task accesses a variable that is shared with other tasks, it copies the data from the variable. This is done in order to prevent unwanted overwriting of the variable by other tasks while a task is performing a computation on it. In case the variable is actually modified, a given task will still have its own copy of the shared data. If there are two concurrent tasks that access a given variable, they could simultaneously modify the variable and thus both of the tasks would have an inconsistent view of the data in the given variable. This problem is termed as a *race condition*, and must be avoided when dealing with concurrent tasks. For this reason, monitors are used to synchronize access to shared data. However, this methodology is not really *deterministic*, in the sense that we cannot easily reason about the actual data contained in a variable at a given point in time. This makes developing concurrent programs quite cumbersome in programming languages that use mutability.

Like other functional programming languages, Clojure tackles this problem using *immutability*—all values are immutable by default and cannot be changed. To model mutable state, there is the notion of *identity*, *state*, and *time*:

- An *identity* is anything that is associated with a changing state. At a given point in time, an identity has a single state.
- *State* is the value associated with an identity at a given point in time.
- *Time* defines an ordering between the states of an identity.

Programs that actually use state can thus be divided into two layers. One layer is purely functional and has nothing to do with state. The other layer constitutes parts of the program that actually require the use of mutable state. This decomposition allows us to isolate the parts of a program that actually require the use of mutable state.

There are several ways to define mutable state in Clojure, and the data structures used for this purpose are termed as *reference types*. A reference type is essentially a mutable reference to an immutable value. Hence, the reference has to be changed explicitly, and the actual value contained in a reference type cannot be modified in any way. Reference types can be characterized in the following ways:

- The change of state in some reference types can either be *synchronous* or *asynchronous*. For example, suppose we are writing data to a file. A synchronous write operation would block the caller until all data is written to the file. On the other hand, an asynchronous write operation would start off a background task to write all data to the file and return to the caller immediately.
- Mutation of a reference type can be performed in either a *coordinated* or an *independent* manner. By coordinated, we mean that state can only be modified within transactions that are managed by some underlying system, which is quite similar to the way a database works. A reference type that mutates independently, however, can be changed without the explicit use of a transaction.
- Changes in some state can be visible to only the thread in which the change occurs, or they could be visible to all threads in the current process.

We will now explore the various reference types that can be used to

represent mutable state in Clojure.

## Using vars

*Vars* are used to manage state that is changed within the scope of a thread. We essentially define vars that can have state, and then bind them to different values. The modified value of a var is only visible to the current thread of execution. Hence, vars are a form of the *thread-local* state.

### Note

The following examples can be found in `src/m_clj/c2/vars.clj` of the book's source code.

Dynamic vars are defined using the `def` form with the `:dynamic` meta keyword. If we omit the `:dynamic` metadata, it would be the same as defining an ordinary variable, or a static var, using a `def` form. It's a convention that all dynamic var names must start and end with the asterisk character (\*), but this is not mandatory. For example, let's define a dynamic variable shown as follows:

```
(def ^:dynamic *thread-local-state* [1 2 3])
```

The `*thread-local-state*` variable defined in *Example 2.5* represents a thread-local var that can change dynamically. We have initialized the var `*thread-local-state*` with the vector `[1 2 3]`, but it's not really required. In case an initial value is not supplied to a `def` form, then the resulting variable is termed as an *unbound* var. While the state of a var is confined to the current thread, its declaration is global to the current namespace. In other words, a var defined with the `def` form will be visible to all threads invoked from the current namespace, but the state of the variable is local to the thread in which it is changed. Thus, vars using the `def` form are also termed as *global vars*.

Normally, the `def` form creates a static var, which can only be redefined by using another `def` form. Static vars can also be redefined within a scope or

context using the `with-redefs` and `with-redefs-fn` forms. A dynamic var, however, can be set to a new value after it has been defined by using the `binding` form, shown as follows:

```
user> (binding [*thread-local-state* [10 20]]
           (map #(* % %) *thread-local-state*))
(100 400)
user> (map #(* % %) *thread-local-state*)
(1 4 9)
```

In this example, the `binding` form changes the value contained in the `*thread-local-state*` var to the vector `[10 20]`. This causes the `map` form in the example to return a different value when called without a `binding` form surrounding it. Thus, the `binding` form can be used to temporarily change the state of the vars supplied to it.

The Clojure namespace system will resolve free symbols, or rather variable names, to their values. This process of resolving a variable name to a namespace qualified symbol is termed as *interning*. Also, a `def` form will first look for an existing global var depending on the symbol it is passed, and will create one if it hasn't been defined yet. The `var` form can be used to obtain the fully qualified name of a variable, instead of its current value, as shown here:

```
user> *thread-local-state*
[1 2 3]
user> (var *thread-local-state*)
#'user/*thread-local-state*
```

## Note

Using the `#'` symbol is the same as using the `var` form. For example, `#'x` is equivalent to `(var x)`.

The `with-bindings` form is another way to rebind vars. This form accepts a map of var and value pairs as its first argument, followed by the body of the form, shown as follows:

```

user> (with-bindings {#'*thread-local-state* [10 20]}
  (map #(* % %) *thread-local-state*))
(100 400)
user> (with-bindings {(var *thread-local-state*) [10 20]}
  (map #(* % %) *thread-local-state*))
(100 400)

```

We can check if a var is bound to any value in the current thread of execution using the `thread-bound?` predicate, which requires a var to be passed as its only argument:

```

user> (def ^:dynamic *unbound-var*)
#'user/*unbound-var*
user> (thread-bound? (var *unbound-var*))
false
user> (binding [*unbound-var* 1]
  (thread-bound? (var *unbound-var*)))
true

```

We can also define vars that are not interned, or *local vars*, using the `with-local-vars` form. These vars will not be resolved by the namespace system, and have to be accessed manually using the `var-get` and `var-set` functions. These functions can thus be used to create and access mutable variables, as shown in *Example 2.5*.

## Note

Using the at-the-rate symbol (@) with a non-interned var is the same as using the `var-get` function. For example, if `x` is a non-interned var, `@x` is equivalent to `(var-get x)`.

```

(defn factorial [n]
  (with-local-vars [i n acc 1]
    (while (> @i 0)
      (var-set acc (* @acc @i))
      (var-set i (dec @i)))
    (var-get acc)))

```

*Example 2.5: Mutable variables using the with-local-vars form*

The `factorial` function defined in *Example 2.5* calculated the factorial of `n` using two mutable local vars `i` and `acc`, which are initialized with the values `n` and `1` respectively. Note that the code in this function exhibits an imperative style of programming, in which the state of the variables `i` and `acc` is manipulated using the `var-get` and `var-set` functions.

## Note

We can check whether a value has been created through a `with-local-vars` form using the `var?` predicate.

## Using refs

A **Software Transactional Memory (STM)** system can also be used to model mutable state. STM essentially treats mutable state as a tiny database that resides in a program's memory. Clojure provides an STM implementation through `refs`, and they can only be changed within a transaction. Refs are a reference type that represent *synchronous* and *coordinated* state.

## Note

The following examples can be found in `src/m_c1j/c2/refs.clj` of the book's source code.

We can create a ref by using the `ref` function, which requires a single argument to indicate the initial state of the ref. For example, we can create a ref as follows:

```
(def state (ref 0))
```

The variable `state` defined here represents a ref with the initial value of `0`. We can dereference `state` using `@` or `deref` to obtain the value contained in it.

In order to modify a ref, we must start a transaction by using the `dosync`

form. If two concurrent tasks invoke transactions using the `dosync` form simultaneously, then the transaction that completes first will update the ref successfully. The transaction which completes later will be retried until it completes successfully. Thus, I/O and other side-effects must be avoided within a `dosync` form, as it can be retried. Within a transaction, we can modify the value of a ref using the `ref-set` function. This function takes two arguments—a ref and the value that represents the new state of the ref. The `ref-set` function can be used to modify a ref as follows:

```
user> @state
0
user> (dosync (ref-set state 1))
1
user> @state
1
```

Initially, the expression `@state` returns `0`, which is the initial state of the ref `state`. The value returned by this expression changes after the call to `ref-set` within the `dosync` form.

We can obtain the latest value contained in a ref by using the `ensure` function. This function returns the latest value of a ref, and has to be called within a transaction. For example, the expression `(ensure state)`, when called within a transaction initiated by a `dosync` form, will return the latest value of the ref `state` in the transaction.

A more idiomatic way to modify a given ref is by using the `alter` and `commute` functions. Both these functions require a ref and a function to be passed to it as arguments. The `alter` and `commute` functions will apply the supplied function to the value contained in a given ref, and save the resulting value into the ref. We can also specify additional arguments to pass to the supplied function. For example, we can modify the state of the ref `state` using `alter` and `commute` as follows:

```
user> @state
1
user> (dosync (alter state + 2))
```

```
3
user> (dosync (commute state + 2))
5
```

The preceding transactions with the `alter` and `commute` forms will save the value `(+ @state 2)` into the ref `state`. The main difference between `alter` and `commute` is that a `commute` form must be preferred when the supplied function is *commutative*. This means two successive calls of the function supplied to a `commute` form must produce the same result regardless of the ordering among the two calls. Using the `commute` form is considered an optimization over the `alter` form in which we are not concerned with the ordering among concurrent transactions on a given ref.

## Note

The `ref-set`, `alter`, and `commute` functions all return the new value contained in the supplied ref. Also, these functions will throw an error if they are not called within a `dosync` form.

A mutation performed by the `alter` and `commute` forms can also be validated. This is achieved using the `:validator` key option when creating a ref, as shown here:

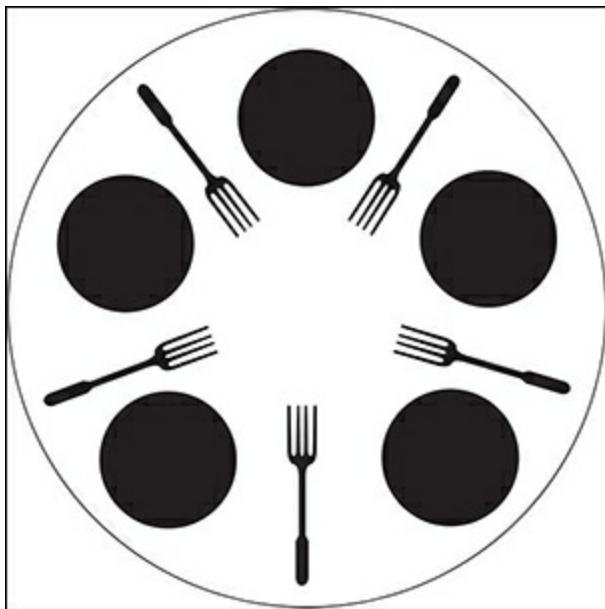
```
user> (def r (ref 1 :validator pos?))
#'user/r
user> (dosync (alter r (fn [_] -1)))
IllegalStateException Invalid reference state
clojure.lang.ARef.validate (ARef.java:33)
user> (dosync (alter r (fn [_] 2)))
2
```

As shown previously, the ref `r` throws an exception when we try to change its state to a negative value. This is because the `pos?` function is used to validate the new state of the ref. Note that the `:validator` key option can be used with other reference types as well. We can also set the validation function of a ref that was created without a `:validator` key option using the `set-validator!` function.

## Note

The `:validator` key option and the `set-validator!` function can be used with *all* reference types. The supplied validation function must return `false` or throw an exception to indicate a validation error.

The *dining philosophers problem* depicts the use of synchronization primitives to share resources. The problem can be defined as follows: five philosophers are seated on a round table to eat spaghetti, and each philosopher requires two forks to eat from his plate of spaghetti. There are five forks on the table, placed in between the five philosophers. A philosopher will first have to pick up a fork from his left side as well as one from his right side before he can eat. When a philosopher cannot obtain the two forks to his left and right side, he must wait until both the forks are available. After a philosopher is done eating his spaghetti, he will think for some time, thereby allowing the other philosophers to use the forks that he used. The solution to this problem requires that all philosophers share the forks among them, and none of the philosophers starve due to being unable to get two forks. The five philosophers' plates and forks are placed on the table as illustrated in the following diagram:



A philosopher must obtain exclusive access to the forks on his left and right side before he starts eating. If both the forks are unavailable, the philosopher must wait for some time for either one of the forks to be free, and retry obtaining the forks. This way, each philosopher can access the forks in tandem with the other philosophers and avoid starvation.

Generally, this solution can be implemented by using synchronization primitives to access the available forks. Refs allow us to implement a solution to the dining philosophers problem without the use of any synchronization primitives. We will now demonstrate how we can implement and simulate a solution to this problem in Clojure. Firstly, we will have to define the states of a fork and a philosopher as refs, as shown in *Example 2.6*:

```
(defn make-fork []
  (ref true))

(defn make-philosopher [name forks food]
  (ref {:name name
        :forks forks
        :eating? false
        :food food}))
```

*Example 2.6: The dining philosophers problem using refs*

The `make-fork` and `make-philosopher` functions create refs to represent the states of a fork and a philosopher, respectively. A fork is simply the state of a Boolean value, indicating whether it is available or not. And a philosopher, created by the `make-philosopher` function, is a map encapsulated as a state, which has the following keys:

- The `:name` key contains the name of a philosopher that is a string value.
- The `:forks` key points to the forks on the left and the right side of a philosopher. Each fork will be a ref created by the `make-fork` function.
- The `:eating?` key indicates whether a philosopher is eating at the moment. It is a Boolean value.

- The `:food` key represents the amount of food available to a philosopher. For simplicity, we will treat this value as an integer.

Now, let's define some primitive operations to help in handling forks, as shown in *Example 2.7*:

```
(defn has-forks? [p]
  (every? true? (map ensure (:forks @p)))))

(defn update-forks [p]
  (doseq [f (:forks @p)]
    (commute f not))
  p)
```

*Example 2.7: The dining philosophers problem using refs (continued)*

The `has-forks?` function defined previously checks whether both the forks that are placed to the left and right of a given philosopher ref `p` are available. The `update-forks` function will modify the state of both the associated forks of a philosopher ref `p` using a `commute` form, and returns the ref `p`. Obviously, these functions can only be called within a transaction created by the `dosync` form, since they use the `ensure` and `commute` functions. Next, we will have to define some functions to initiate transactions and invoke the `has-forks?` and `update-forks` functions for a given philosopher, as shown in *Example 2.8*:

```
(defn start-eating [p]
  (dosync
    (when (has-forks? p)
      (update-forks p)
      (commute p assoc :eating? true)
      (commute p update-in [:food] dec)))))

(defn stop-eating [p]
  (dosync
    (when (:eating? @p)
      (commute p assoc :eating? false)
      (update-forks p))))
```

```
(defn dine [p retry-ms max-eat-ms max-think-ms]
  (while (pos? (:food @p))
    (if (start-eating p)
        (do
          (Thread/sleep (rand-int max-eat-ms) )
          (stop-eating p)
          (Thread/sleep (rand-int max-think-ms)) )
        (Thread/sleep retry-ms))))
```

*Example 2.8: The dining philosophers problem using refs  
(continued)*

The heart of the solution to the dining philosophers problem is the `start-eating` function in *Example 2.8*. This function will check whether both the forks on either side of a philosopher are available, using the `has-forks?` function. The `start-eating` function will then proceed to update the states of these forks by calling the `update-forks` function. The `start-eating` function will also change the state of the philosopher ref `p` by invoking `commute` with the `assoc` and `update-in` functions, which both return a new map. Since the `start-eating` function uses a `when` form, it will return `nil` when any of the philosophers' forks are unavailable. These few steps are the solution; in a nutshell, a philosopher will eat only when both his forks are available.

The `stop-eating` function in *Example 2.8* reverses the state of a given philosopher ref after the `start-eating` function has been invoked on it. This function basically sets the `:eating` key of the map contained in the supplied philosopher ref `p` to `false` using a `commute` form, and then calls `update-forks` to reset the state of the associated forks of the philosopher ref `p`.

The `start-eating` and `stop-eating` function can be called repeatedly in a loop using a `while` form, as long as the `:food` key of a philosopher ref `p`, or rather the amount of available food, is a positive value. This is performed by the `dine` function in *Example 2.8*. This function will call the `start-`

eating function on a philosopher ref `p`, and will wait for some time if the philosopher's forks are being used by any other philosophers. The amount of time that a philosopher waits for is indicated by the `retry-ms` argument that is passed to the `dine` function. If a philosopher's forks are available, he eats for a random amount of time, as indicated by the expression `(rand-int max-eat-ms)`. Then, the `stop-eating` function is called to reset the state of the philosopher ref `p` and the forks that it contains. Finally, the `dine` function waits for a random amount of time, which is represented by the `(rand-int max-think-ms)` expression, to indicate that a philosopher is thinking.

Let's now define some function and actually create some refs representing philosophers and associated forks, as shown in *Example 2.9*:

```
(defn init-forks [nf]
  (repeatedly nf #'(make-fork)))

(defn init-philosophers [np food forks init-fn]
  (let [p-range (range np)
        p-names (map #'(str "Philosopher " (inc %))
                     p-range)
        p-forks (map #'(vector (nth forks %)
                               (nth forks (-> % inc (mod np)))))
                     p-range)
        p-food (cycle [food])]
    (map init-fn p-names p-forks p-food)))
```

*Example 2.9: The dining philosophers problem using refs  
(continued)*

The `init-forks` function from *Example 2.9* will simply invoke the `make-fork` function a number of times, as indicated by its argument `nf`. The `init-philosophers` function will create `np` number of philosophers and associate each of them with a vector of two forks and a certain amount of food. This is done by mapping the function `init-fn`, which is a function that matches the arity of the `make-philosopher` function in *Example 2.6*, over a range of philosopher names `p-names` and forks `p-forks`, and an

infinite range `p-food` of the value `food`.

We will now define a function to print the collective state of a sequence of philosophers. This can be done in a fairly simple manner using the `doseq` function, as shown in *Example 2.10*:

```
(defn check-philosophers [philosophers forks]
  (doseq [i (range (count philosophers))]
    (println (str "Fork:\t\t\t available=" @ (nth forks i)))
    (if-let [p @ (nth philosophers i)]
      (println (str (:name p)
                    ":\t\t eating=" (:eating? p)
                    " food=" (:food p))))))
```

*Example 2.10: The dining philosophers problem using refs (continued)*

The `check-philosophers` function in *Example 2.10* iterates through all of its supplied philosopher refs, represented by `philosophers`, and associated forks, represented by `forks`, and prints their state. The `if-let` form is used here to check if a dereferenced ref from the collection `philosophers` is not `nil`.

Now, let's define a function to concurrently invoke the `dine` function over a collection of philosopher. This function could also pass in values for the `retry-ms`, `max-eat-ms`, and `max-think-ms` arguments of the `dine` function. This is implemented in the `dine-philosophers` function in *Example 2.11*:

```
(defn dine-philosophers [philosophers]
  (doall (for [p philosophers]
            (future (dine p 10 100 100)))))
```

*Example 2.11: The dining philosophers problem using refs (continued)*

Finally, let's define five instances of `philosophers` and five associated forks for our simulation, using the `init-forks`, `init-philosophers`, and `make-`

philosopher functions, as shown in *Example 2.12* as follows:

```
(def all-forks (init-forks 5))

(def all-philosophers
  (init-philosophers 5 1000 all-forks make-philosopher))
```

*Example 2.12: The dining philosophers problem using refs  
(continued)*

We can now use the `check-philosopher` function to print the state of the philosopher and fork refs created in *Example 2.12*, as shown here:

```
user> (check-philosophers all-philosophers all-forks)
Fork:                                available=true
Philosopher 1:                          eating=false food=1000
Fork:                                available=true
Philosopher 2:                          eating=false food=1000
Fork:                                available=true
Philosopher 3:                          eating=false food=1000
Fork:                                available=true
Philosopher 4:                          eating=false food=1000
Fork:                                available=true
Philosopher 5:                          eating=false food=1000
nil
```

Initially, all of the forks are available and none of the philosophers are eating. To start the simulation, we must call the `dine-philosophers` function on the philosopher refs `all-philosophers` and the fork refs `all-forks`, as shown here:

```
user> (def philosophers-futures (dine-philosophers all-
philosophers))
#'user/philosophers-futures
user> (check-philosophers all-philosophers all-forks)
Fork:                                available=false
Philosopher 1:                        eating=true food=978
Fork:                                available=false
Philosopher 2:                        eating=false food=979
Fork:                                available=false
Philosopher 3:                        eating=true food=977
```

```
Fork: available=false
Philosopher 4: eating=false food=980
Fork: available=true
Philosopher 5: eating=false food=980
nil
```

After invoking the `dine-philosophers` function, each philosopher is observed to consume the allocated food, as shown in the output of the previous `check-philosophers` function. At any given point of time, one or two philosophers are observed to be eating, and the other philosophers will wait until they complete using the available forks. Subsequent calls to the `check-philosophers` function also indicate the same output, and the philosophers will eventually consume all of the allocated food:

```
user> (check-philosophers all-philosophers all-forks)
Fork: available=true
Philosopher 1: eating=false food=932
Fork: available=true
Philosopher 2: eating=false food=935
Fork: available=true
Philosopher 3: eating=false food=933
Fork: available=true
Philosopher 4: eating=false food=942
Fork: available=true
Philosopher 5: eating=false food=935
nil
```

We can pause the simulation by calling the `future-cancel` function, as shown here. Once the simulation is paused, it can be resumed by calling the `dine-philosophers` function again, as (`(dine-philosophers all-philosophers)`):

```
user> (map future-cancel philosophers-futures)
(true true true true true)
```

To summarize, the preceding example is a concise and working implementation of a solution to the dining philosophers problem using Clojure futures and refs.

# Using atoms

*Atoms* are used to handle state that changes atomically. Once an atom is modified, its new value is reflected in all concurrent threads. In this way, atoms represent *synchronous* and *independent* state. Let's quickly explore the functions that can be used to handle atoms.

## Note

The following examples can be found in `src/m_clj/c2/atoms.clj` of the book's source code.

We can define an atom using the `atom` function, which requires the initial state of the atom to be passed to it as the first argument, as shown here:

```
(def state (atom 0))
```

The `reset!` and `swap!` functions can be used to modify the state of an atom. The `reset!` function is used to directly set the state of an atom. This function takes two arguments—an atom and the value that represents the new state of the atom, as shown here:

```
user> @state
0
user> (reset! state 1)
1
user> @state
1
```

The `swap!` function requires a function and additional arguments to pass to the supplied function as arguments. The supplied function is applied to the value contained in the atom along with the other additional arguments specified to the `swap!` function. This function can thus be used to mutate an atom using a supplied function, as shown here:

```
user> @state
1
user> (swap! state + 2)
```

The call to the preceding `swap!` function sets the state of the atom to the result of the expression `(+ @state 2)`. The `swap!` function may call the function `+` multiple times due to concurrent calls to the `swap!` function on the atom `state`. Hence, functions that are passed to the `swap!` function must be free of I/O and other side effects.

## Note

The `reset!` and `swap!` functions both return the new value contained in the supplied atom.

We can watch for any change in an atom, and other reference types as well, using the `add-watch` function. This function will call a given function whenever the state of an atom is changed. The `add-watch` function takes three arguments—a reference, a key and a *watch function*, that is, a function that must be called whenever the state of the supplied reference type is changed. The function that is supplied to the `add-watch` function must accept four arguments—a key, the reference that was changed, the old value of the reference, and the new value of the reference. The value of the key argument that is passed to the `add-watch` function gets passed to the `watch` function as its first argument. A `watch` function can also be unlinked from a given reference type using the `remove-watch` function. The `remove-watch` function accepts two arguments—a reference and a key that was specified while adding a `watch` function to the reference. *Example 2.13* depicts how we can track the state of an atom using a `watch` function:

```
(defn make-state-with-watch []
  (let [state (atom 0)
        state-is-changed? (atom false)
        watch-fn (fn [key r old-value new-value]
                   (swap! state-is-changed? (fn [_] true)))
        (add-watch state nil watch-fn)
        [state
         state-is-changed?]))
```

### *Example 2.13: Using the add-watch function*

The `make-state-with-watch` function defined in *Example 2.13* returns a vector of two atoms. The second atom in this vector initially contains the value `false`. Whenever the state of the first atom in the vector returned by the `make-state-with-watch` function is changed, the state of the second atom in this vector is changed to the value `true`. This can be verified in the REPL, as shown here:

```
user> (def s (make-state-with-watch))
#'user/s
user> @(nth s 1)
false
user> (swap! (nth s 0) inc)
1
user> @(nth s 1)
true
```

Thus, watch functions can be used with the `add-watch` function to track the state of atoms and other reference types.

### **Note**

The `add-watch` function can be used with *all* reference types.

## **Using agents**

An *agent* is used to represent state that is associated with a queue of actions and a pool of worker threads. Any action that modifies the state of an agent must be sent to its queue, and the supplied function will be called by a thread selected from the agent's pool of worker threads. We can send actions asynchronously to agents as well. Thus, agents represent *asynchronous* and *independent* state.

### **Note**

The following examples can be found in `src/m_clj/c2/agents.clj` of the book's source code.

An agent is created using the `agent` function. For example, we can create an agent with an empty map as its initial value as follows:

```
(def state (agent {}))
```

We can modify the state of an agent by using the `send` and `send-off` functions. The `send` and `send-off` functions will send a supplied action and its additional arguments to an agent's queue in an asynchronous manner. Both these functions return the agent they are passed immediately.

The primary difference between the `send` and `send-off` functions is that the `send` function assigns actions to a thread selection from a pool of worker threads, whereas the `send-off` function creates a new dedicated thread to execute each action. Blocking actions that are sent to an agent using the `send` function could exhaust the agent's pool of threads. Thus, the `send-off` function is preferred for sending blocking actions to an agent.

To demonstrate the `send` and `send-off` functions, let's first define a function that returns a closure that sleeps for a certain amount of time, and then, call the `assoc` function, as shown in *Example 2.14*:

```
(defn set-value-in-ms [n ms]
  (fn [a]
    (Thread/sleep ms)
    (assoc a :value n)))
```

*Example 2.14: A function that returns a closure which sleeps and calls assoc*

A closure returned by the `set-value-in-ms` function, in *Example 2.14*, can be passed as an action to the `send` and `send-off` functions, as shown here:

```
user> (send state (set-value-in-ms 5 5000))
#<Agent@7fce18: {}>
user> (send-off state (set-value-in-ms 10 5000))
#<Agent@7fce18: {}>
user> @state
{ }
```

```
user> @state ; after 5 seconds
{:value 5}
user> @state ; after another 5 seconds
{:value 10}
```

The calls to the preceding `send` and `send-off` functions will call the closures returned by the `set-value-in-ms` function, from *Example 2.14*, asynchronously over the agent state. The agent's state changes over a period of 10 seconds, which is required to execute the closures returned by the `set-value-in-ms` function. The new key-value pair `{:value 5}` is observed to be saved into the agent state after five seconds, and the state of the agent again changes to `{:value 10}` after another five seconds.

Any action that is passed to the `send` and `send-off` functions can use the `*agent*` var to access the agent through which the action will be executed.

The `await` function can be used to wait for all actions in an agent's queue to be completed, as shown here:

```
user> (send-off state (set-value-in-ms 100 3000))
#<Agent@af9ac: {:value 10}>
user> (await state) ; will block
nil
user> @state
{:value 100}
```

The expression `(await state)` is observed to be blocked until the previous action that was sent to the agent `state` using the `send-off` function is completed. The `await-for` function is a variant of `await`, which waits for a certain number of milliseconds, indicated by its first argument, for all the actions on an agent, its second argument, to complete.

An agent also saves any error it encounters while performing the actions in its queue. An agent will throw the error it has encountered on any subsequent calls to the `send` and `send-off` functions. The error saved by an agent can be accessed using the `agent-error` function, and can be cleared using the `clear-agent-errors` function, as shown here:

```
user> (def a (agent 1))
#'user/a
user> (send a / 0)
#<Agent@5d29f1: 1>
user> (agent-error a)
#<ArithmaticException java.lang.ArithmaticException: Divide by
zero>
user> (clear-agent-errors a)
1
user> (agent-error a)
nil
user> @a
1
```

An agent that has encountered an error can also be restarted using the `restart-agent` function. This function takes an agent as its first argument and the new state of the agent as its second argument. All actions that were sent to an agent while it was failed will be executed once the `restart-agent` is called on the agent. We can avoid this behavior by passing the `:clear-actions true` optional argument to the `restart-agent` function. In this case, any actions held in an agent's queue are discarded before it is restarted.

To create a pool of threads, or a *threadpool*, to use with an agent, we must call the static `newFixedThreadPool` method of the `java.util.concurrent.Executors` class by passing the desired number of threads in the pool as an argument, as follows:

```
(def pool (java.util.concurrent.Executors/newFixedThreadPool 10))
```

The pool of threads defined previously can be used to execute the actions of an agent by using the `send-via` function. This function is a variant of the `send` function that accepts a pool of threads, such as the `pool` defined previously, as its first argument, as shown here:

```
user> (send-via pool state assoc :value 1000)
#<Agent@8efada: {:value 100}>
user> @state
{:value 1000}
```

We can also specify the thread pools to be used by all agents to execute actions sent to them using the `send` and `send-off` functions using the `set-agent-send-executor!` and `set-agent-send-off-executor!` functions respectively. Both of these functions accept a single argument representing a pool of threads.

All agents in the current process can be stopped by invoking the `(shutdown-agents)`. The `shutdown-agents` function should only be called before exiting a process, as there is no way to restart the agents in a process after calling this function.

Now, let's try implementing the dining philosophers problem using agents. We can reuse most of the functions from the previous implementation of the dining philosophers problem that was based on refs. Let's define some functions to model this problem using agents, as shown in *Example 2.15*:

```
(defn make-philosopher-agent [name forks food]
  (agent {:name name
          :forks forks
          :eating? false
          :food food}))  
  
(defn start-eating [max-eat-ms]
  (dosync (if (has-forks? *agent*)
            (do
              (-> *agent*
                  update-forks
                  (send assoc :eating? true)
                  (send update-in [:food] dec))
              (Thread/sleep (rand-int max-eat-ms))))))  
  
(defn stop-eating [max-think-ms]
  (dosync (-> *agent*
                (send assoc :eating? false)
                update-forks))
  (Thread/sleep (rand-int max-think-ms)))  
  
(def running? (atom true))  
  
(defn dine [p max-eat-ms max-think-ms]
```

```

	when (and p (pos? (:food p)))
	(if-not (:eating? p)
		(start-eating max-eat-ms)
		(stop-eating max-think-ms))
	(if-not @running?
		@*agent*
		@(send-off *agent* dine max-eat-ms max-think-ms)))))

(defn dine-philosophers [philosophers]
  (swap! running? (fn [_] true))
  (doall (for [p philosophers]
    (send-off p dine 100 100)))))

(defn stop-philosophers []
  (swap! running? (fn [_] false)))

```

*Example 2.15: The dining philosophers problem using agents*

In *Example 2.15*, the `make-philosopher-agent` function will create an agent representing a philosopher. The initial state of the resulting agent is a map of the keys `:name`, `:forks`, `:eating?`, and `:food`, as described in the previous implementation of the dining philosophers problem. Note that the forks in this implementation are still represented by refs.

The `start-eating` function in *Example 2.15* will start a transaction, check whether the forks placed to the left and right sides of a philosopher are available, changes the state of the forks and philosopher agent accordingly, and then suspends the current thread for some time to indicate that a philosopher is eating. The `stop-eating` function in *Example 2.15* will similarly update the state of a philosopher and the forks he had used, and then suspend the current thread for some time to indicate that a philosopher is thinking. Note that both the `start-eating` and `stop-eating` functions reuse the `has-forks?` and `update-forks` functions from *Example 2.7* of the previous implementation of the dining philosophers problem.

The `start-eating` and `stop-eating` functions are called by the `dine` function in *Example 2.15*. We can assume that this function will be passed as an action to a philosopher agent. This function checks the value of the

:eating? key contained in a philosopher agent to decide whether it must invoke the start-eating or stop-eating function in the current call. Next, the `dine` function invokes itself again using the `send-off` function and dereferencing the agent returned by the `send-off` function. The `dine` function also checks the state of the atom `running?` and does not invoke itself through the `send-off` function in case the expression `@running` returns `false`.

The `dine-philosophers` function in *Example 2.15* starts the simulation by setting the value of the `running?` atom to `true` and then invoking the `dine` function asynchronously through the `send-off` function for all the philosopher agents passed to it, represented by `philosophers`. The function `stop-philosophers` simply sets the value of the `running?` atom to `false`, thereby stopping the simulation.

Finally, let's define five instances of forks and philosophers using the `init-forks` and `init-philosophers` functions from *Example 2.9*, shown in *Example 2.16* as follows:

```
(def all-forks (init-forks 5))

(def all-philosophers
  (init-philosophers 5 1000 all-forks make-philosopher-agent))
```

*Example 2.16: The dining philosophers problem using agents (continued)*

We can now start the simulation by calling the `dine-philosophers` function. Also, we can print the collective state of the fork and philosopher instances in the simulation using the `check-philosophers` function defined in *Example 2.10*, as follows:

```
user> (def philosophers-agents (dine-philosophers all-
philosophers))
#'user/philosophers-agents
user> (check-philosophers all-philosophers all-forks)
Fork:           available=false
```

```

Philosopher 1:          eating=false food=936
Fork:                  available=false
Philosopher 2:          eating=false food=942
Fork:                  available=true
Philosopher 3:          eating=true food=942
Fork:                  available=true
Philosopher 4:          eating=false food=935
Fork:                  available=true
Philosopher 5:          eating=true food=943
nil

user> (check-philosophers all-philosophers all-forks)
Fork:                  available=false
Philosopher 1:          eating=true food=743
Fork:                  available=false
Philosopher 2:          eating=false food=747
Fork:                  available=true
Philosopher 3:          eating=false food=751
Fork:                  available=true
Philosopher 4:          eating=false food=741
Fork:                  available=true
Philosopher 5:          eating=false food=760
nil

```

As shown in the preceding output, all philosopher agents share the fork instances among themselves. In effect, they work in tandem to ensure that each philosopher eventually consumes all of their allocated food.

In summary, vars, refs, atoms, and agents can be used to represent mutable state that is shared among concurrently executing tasks.

# Executing tasks in parallel

The simultaneous execution of several computations is termed as *parallelism*. The use of parallelism tends to increase the overall performance of a computation, since the computation can be partitioned to execute on several cores or processors. Clojure has a couple of functions that can be used for the parallelization of a particular computation or task, and we will briefly examine them in this section.

## Note

The following examples can be found in `src/m_clj/c2/parallel.clj` of the book's source code.

Suppose we have a function that pauses the current thread for some time and then returns a computed value, as depicted in *Example 2.17*:

```
(defn square-slowly [x]
  (Thread/sleep 2000)
  (* x x))
```

*Example 2.17: A function that pauses the current thread*

The function `square-slowly` in *Example 2.17* requires a single argument `x`. This function pauses the current thread for two seconds and returns the square of its argument `x`. If the function `square-slowly` is invoked over a collection of three values using the `map` function, it takes three times as long to complete, as shown here:

```
user> (time (doall (map square-slowly (repeat 3 10))))
"Elapsed time: 6000.329702 msecs"
(100 100 100)
```

The previously shown `map` form returns a lazy sequence, and hence the `doall` form is required to realize the value returned by the `map` form. We could also use the `dorun` form to perform this realization of a lazy

sequence. The entire expression is evaluated in about six seconds, which is thrice the time taken by the `square-slowly` function to complete. We can parallelize the application of the `square-slowly` function using the `pmap` function instead of `map`, as shown here:

```
user> (time (doall (pmap square-slowly (repeat 3 10))))  
"Elapsed time: 2001.543439 msecs"  
(100 100 100)
```

The entire expression now evaluates in the same amount of time required for a single call to the `square-slowly` function. This is due to the `square-slowly` function being called in parallel over the supplied collection by the `pmap` form. Thus, the `pmap` form has the same semantics as that of the `map` form, except that it applies the supplied function in parallel.

The `pvalues` and `pcalls` forms can also be used to parallelize computations. The `pvalues` form evaluates the expressions passed to it in parallel, and returns a lazy sequence of the resulting values. Similarly, the `pcalls` form invokes all functions passed to it, which must take no arguments, in parallel and returns a lazy sequence of the values returned by these functions:

```
user> (time (doall (pvalues (square-slowly 10)  
                           (square-slowly 10)  
                           (square-slowly 10))))  
"Elapsed time: 2007.702703 msecs"  
(100 100 100)  
user> (time (doall (pcalls #(square-slowly 10)  
                           #(square-slowly 10)  
                           #(square-slowly 10))))  
"Elapsed time: 2005.683279 msecs"  
(100 100 100)
```

As shown in the preceding output, both expressions that use the `pvalues` and `pcalls` forms take the same amount of time to evaluate as a single call to the `square-slowly` function.

## Note

The `pmap`, `pvalues`, and `pcalls` forms *all* return lazy sequences that have to be realized using the `doall` or `dorun` form.

## Controlling parallelism with thread pools

The `pmap` form schedules parallel execution of the supplied function on the default threadpool. If we wish to configure or tweak the threadpool used by `pmap`, the `claypoole` library

(<https://github.com/TheClimateCorporation/claypoole>) is a good option.

This library provides an implementation of the `pmap` form that must be passed a configurable threadpool. We will now demonstrate how we can use this library to parallelize a given function.

### Note

The following library dependencies are required for the upcoming examples:

```
[com.climate/claypoole "1.0.0"]
```

Also, the following namespaces must be included in your namespace declaration:

```
(ns my-namespace
  (:require [com.climate.claypoole :as cp]
            [com.climate.claypoole.lazy :as cpl]))
```

The `pmap` function from the `com.climate.claypoole` namespace is essentially a variant of the standard `pmap` function to which we supply a threadpool instance to be used in parallelizing a given function. We can also supply the number of threads to be used by this variant of the `pmap` function in order to parallelize a given function, as shown here:

```
user> (time (doall (cpl/pmap 2 square-slowly [10 10 10])))
"Elapsed time: 4004.029789 msecs"
(100 100 100)
```

As previously shown, the `pmap` function from the `claypoole` library can be

used to parallelize the `square-slowly` function that we defined earlier in *Example 2.17* over a collection of three values. These three elements are computed over in two batches, in which each batch will parallelly apply the `square-slowly` function over two elements in two separate threads. Since the `square-slowly` function takes two seconds to complete, the total time taken to compute over the collection of three elements is around four seconds.

We can create an instance of a pool of threads using the `threadpool` function from the `claypoole` library. This threadpool instance can then be passed to the `pmap` function from the `claypoole` library. The `com.climate.claypoole` namespace also provides the `ncpus` function that returns the number of physical processors available to the current process. We can create a threadpool instance and pass it to this variant of the `pmap` function as shown here:

```
user> (def pool (cp/threadpool (cp/ncpus)))
#'user/pool
user> (time (doall (cp1/pmap pool square-slowly [10 10 10])))
"Elapsed time: 4002.05885 msecs"
(100 100 100)
```

Assuming that we are running the preceding code on a computer system that has two physical processors, the call to the `threadpool` function shown previously will create a threadpool of two threads. This threadpool instance can then be passed to the `pmap` function as shown in the preceding example.

## Note

We can fall back to the standard behavior of the `pmap` function by passing the `:builtin` keyword as the first argument to the `com.climate.claypoole/pmap` function. Similarly, if the keyword `:serial` is passed as the first argument to the `claypoole` version of the `pmap` function, the function behaves like the standard `map` function.

The `threadpool` function also supports a couple of useful key options. Firstly, we can create a pool of non-daemon threads using the `:daemon false` optional argument. Daemon threads are killed when the process exits, and all threadpools created by the `threadpool` function are pools of daemon threads by default. We can also name a threadpool using the `:name` key option of the `threadpool` function. The `:thread-priority` key option can be used to indicate the priority of the threads in the new threadpool.

Tasks can also be prioritized using the `pmap`, `priority-threadpool`, and `with-priority` forms from the `claypoole` library. A priority threadpool is created using the `priority-threadpool` function, and this new threadpool can be used along with the `with-priority` function to assign a priority to a task that must be parallelized using `pmap`, as shown here:

```
user> (def pool (cp/priority-threadpool (cp/ncpus))
#'user/pool
user> (def task-1 (cp/pmap (cp/with-priority pool 1000)
                           square-slowly [10 10 10]))
#'user/task-1
user> (def task-2 (cp/pmap (cp/with-priority pool 0)
                           square-slowly [5 5 5]))
#'user/task-2
```

Tasks with higher priority are assigned to threads first. Hence, the task represented by `task-1` will be assigned to a thread of execution before the task represented by `task-2` in the previous output.

To gracefully deallocate a given threadpool, we can call the `shutdown` function from the `com.climate.claypoole` namespace, which accepts a threadpool instance as its only argument. The `shutdown!` function from the same namespace will forcibly shut down the threads in a threadpool. The `shutdown!` function can also be called using the `with-shutdown!` macro. We specify the threadpools to be used for a series of computations as a vector of bindings to the `with-shutdown!` macro. This macro will implicitly call the `shutdown!` function on all of the threadpools that it has created once all the computations in the body of this macro are completed. For

example, we can define a function to create a threadpool, use it for a computation, and finally, shut down the threadpool, using the `with-shutdown!` function as shown in *Example 2.18*:

```
(defn square-slowly-with-pool [v]
  (cp/with-shutdown! [pool (cp/threadpool (cp/ncpus))]
    (doall (cp/pmap pool square-slowly v))))
```

### *Example 2.18: Using a priority threadpool*

The `square-slowly-with-pool` function defined in *Example 2.18* will create a new threadpool, represented by `pool`, and then use it to call the `pmap` function. The `shutdown!` function is implicitly called once the `doall` form completely evaluates the lazy sequence returned by the `pmap` function.

The `claypoole` library also supports *unordered parallelism*, in which results of individual threads of computation are used as soon as they are available in order to minimize latency. The `com.climate.claypoole/upmap` function is an unordered parallel version of the `pmap` function.

The `com.climate.claypoole` namespace also provides several other functions that use threadpools, as described here:

- The `com.climate.claypoole/pvalues` function is a threadpool-based implementation of the `pvalues` function. It will evaluate its arguments in parallel using a supplied threadpool and return a lazy sequence.
- The `com.climate.claypoole/pcalls` function is a threadpool-based version of the `pcalls` function, which invokes several no-argument functions to return a lazy sequence.
- A future that uses a given threadpool can be created using the `com.climate.claypoole/future` function.
- We can evaluate an expression in a parallel fashion over the items in a given collection using the `com.climate.claypoole/pfor` function.
- The `upvalues`, `upcalls`, and `upfor` functions in the `com.climate.claypoole` namespace are unordered parallel versions of

the `pvalues`, `pcalls`, and `pfor` functions, respectively, from the same namespace.

It is quite evident that the `pmap` function from the `com.climate.claypooke` namespace will eagerly evaluate the collection it is supplied. This may be undesirable when we intend to call `pmap` over an infinite sequence. The `com.climate.claypooke.lazy` namespace provides versions of `pmap` and other functions from the `com.climate.claypooke` namespace that preserve the laziness of a supplied collection. The lazy version of the `pmap` function can be demonstrated as follows:

```
user> (def lazy-pmap (cp1/pmap pool square-slowly (range)))
#'user/lazy-pmap
user> (time (doall (take 4 lazy-pmap)))
"Elapsed time: 4002.556548 msecs"
(0 1 4 9)
```

The previously defined `lazy-pmap` sequence is a lazy sequence created by mapping the `square-slowly` function over the infinite sequence `(range)`. As shown previously, the call to the `pmap` function returns immediately, and the first four elements of the resulting lazy sequence are realized in parallel using the `doall` and `take` functions.

To summarize, Clojure has the `pmap`, `pvalues`, and `pcalls` primitives to deal with parallel computations. If we intend to control the amount of parallelism utilized by these functions, we can use the `claypooke` library's implementations of these primitives. The `claypooke` library also supports other useful features such as prioritized threadpools and unordered parallelism.

# Summary

We have explored various constructs that can be used to create concurrent and parallel tasks in Clojure. You learned to handle shared mutable state through the use of reference types, namely vars, refs, atoms and agents. As we described earlier, the dining philosophers problem can be easily implemented using refs and agents. You also studied how tasks can be executed in parallel. Lastly, we explored the `claypool` library, which allows us to control the amount of parallelism used for a given computation.

In the next chapter, we will continue our exploration of parallelism in Clojure through the use of reducers.

# Chapter 3. Parallelization Using Reducers

*Reducers* are another way of looking at collections in Clojure. In this chapter, we will study this particular abstraction of collections, and how it is quite orthogonal to viewing collections as sequences. The motivation behind reducers is to increase the performance of computations over collections. This performance gain is achieved mainly through parallelization of such computations.

As we have seen in [Chapter 1, Working with Sequences and Patterns](#), sequences and laziness are a great way to handle collections. The Clojure standard library provides several functions to handle and manipulate sequences. However, abstracting a collection as a sequence has an unfortunate consequence; any computation performed over all the elements of a sequence is inherently sequential. Also, all of the standard sequence functions create a new collection that is similar to the collection passed to these functions. Interestingly, performing a computation over a collection without creating a similar collection, even as an intermediary result, is quite useful. For example, it is often required to reduce a given collection to a single value through a series of transformations in an iterative manner. This sort of computation does not necessarily require the intermediary results of each transformation to be saved.

A consequence of iteratively computing values from a collection is that we cannot parallelize it in a straightforward way. Modern *MapReduce* frameworks handle this kind of computation by pipelining the elements of a collection through several transformations in parallel, and finally, reducing the results into a single result. Of course, the result could as well be a new collection. A drawback of this methodology is that it produces concrete collections as intermediate results of each transformation, which is rather wasteful. For example, if we wanted to filter out values from a

collection, the MapReduce strategy would require creating empty collections to represent values that are left out of the reduction step to produce the final result.

This incurs unnecessary memory allocation and also creates additional work for the reduction step, which produces the final result. Hence, there's a scope for optimizing these sorts of computations.

This brings us to the notion of treating computations over collections as *reducers* to attain better performance. Of course, this doesn't mean that reducers are a replacement for sequences. Sequences and laziness are great for abstracting computations that create and manipulate collections, while reducers are a specialized high-performance abstraction of collections in which a collection needs to be piped through several transformations, and finally, combined to produce the final result. Reducers achieve a performance gain in the following ways:

- Reducing the amount of memory allocated to produce the desired result
- Parallelizing the process of reducing a collection into a single result, which could be an entirely new collection

The `clojure.core.reducers` namespace provides several functions to process collections using reducers. Let's now examine how reducers are implemented and a few examples that demonstrate how reducers can be used.

## Using reduce to transform collections

Sequences and functions that operate on sequences preserve the sequential ordering between elements. Lazy sequences avoid the unnecessary realization of elements in a collection until they are required for a computation, but the realization of these values is still performed in a

sequential manner. However, this characteristic of sequential ordering may not be desirable for all computations performed over it. For example, it's not possible to map a function over a vector and then lazily realize values in the resulting collection out of order; since the `map` function converts the supplied collection into a sequence. Also, functions such as `map` and `filter` are lazy, but still sequential by nature.

## What's wrong with sequences?

One of the limitations of sequences is that they are realized in *chunks*. Let's study a simple example to illustrate what this means. Consider a unary function, as shown in *Example 3.1*, which we intend to map over a given vector. The function must compute a value from the one it is supplied, and also perform a side effect so that we can observe its application over the elements in a collection.

### Note

The following examples can be found in `src/m_clj/c3/reducers.clj` of the book's source code.

```
(defn square-with-side-effect [x]
  (do
    (println (str "Side-effect: " x))
    (* x x)))
```

#### *Example 3.1: A simple unary function*

The `square-with-side-effect` function simply returns the square of a number `x` using the `*` function. This function also prints the value of `x` using a `println` form whenever it is called. Suppose this function is mapped over a given vector. The resulting collection would have to be realized completely if a computation has to be performed over it, even if all the elements from the resulting vector are not required. This can be demonstrated as follows:

```
user> (def mapped (map square-with-side-effect [0 1 2 3 4 5]))
```

```
#'user/mapped
user> (reduce + (take 3 mapped))
Side-effect: 0
Side-effect: 1
Side-effect: 2
Side-effect: 3
Side-effect: 4
Side-effect: 5
5
```

As shown previously, the `mapped` variable contains the result of mapping the `square-with-side-effect` function over a vector. If we try to sum the first three values in the resulting collection using the `reduce`, `take`, and `+` functions, all the values in the `[0 1 2 3 4 5]` vector are printed as a side effect, as shown in the preceding output. This means that the `square-with-side-effect` function was applied to all the elements in the initial vector, despite the fact that only the first three elements were actually required by the `reduce` form. Of course, this can be solved using the `seq` function to convert the vector to a sequence before mapping the `square-with-side-effect` function over it. But then, we lose the ability to efficiently access elements in a random order in the resulting collection.

To understand why this actually happens, we first need to understand how the standard `map` function is actually implemented. A simplified definition of the `map` function is shown in *Example 3.2*:

```
(defn map [f coll]
  (cons (f (first coll))
        (lazy-seq (map f (rest coll)))))
```

*Example 3.2: A simplified definition of the map function*

The definition of `map` in *Example 3.2* is a simplified and rather incomplete one, as it doesn't check for an empty collections and cannot be used over multiple collections. That aside, this definition of `map` does indeed apply a function `f` to all the elements in a collection `coll`. This is implemented using a composition of the `cons`, `first`, `rest`, and `lazy-seq` forms.

This implementation can be interpreted as "applying the function `f` to the first element in the collection `coll`, and then mapping `f` over the rest of the collection in a lazy manner". An interesting consequence of this implementation is that the `map` function has the following characteristics:

- The ordering among elements in the collection `coll` is preserved.
- This computation is performed recursively.
- The `lazy-seq` form is used to perform the computation in a lazy manner.
- The use of the `first` and `rest` forms indicate that `coll` must be a sequence, and the `cons` form will also produce a result that is a sequence. Hence, the `map` function accepts a sequence and builds a new one.

However, none of these properties of sequences are needed to transform a given collection into a result that is not a sequence. Another characteristic of lazy sequences is how they are realized. By the term realized, we mean to say a given lazy sequence is evaluated to produce concrete values. Lazy sequences are realized in *chunks*. Each chunk is comprised of 32 elements, and this is done as an optimization. Sequences that behave this way are termed as *chunked sequences*. Of course, not all sequences are chunked, and we can check whether a given sequence is chunked using the `chunked-seq?` predicate. The `range` function returns a chunked sequence, as shown here:

```
user> (first (map #'(do (print \!) %) (range 70)))
!!!!!!!!!!!!!!!!!!!!!!!
0
user> (nth (map #'(do (print \!) %) (range 70)) 32)
!!!!!!!!!!!!!!!!!!!!!!!
32
```

Both the statements in the preceding output select a single element from a sequence returned by the `map` function. The function passed to the `map` function in both the preceding statements prints the `!` character and returns the value supplied to it. In the first statement, the first 32 elements of the

resulting sequence are realized even though only the first element is required. Similarly, the second statement is observed to realize the first 64 elements of the resulting sequence when the element at the 32nd position is obtained using the `nth` function. But again, realizing a collection in chunks isn't required to perform a computation over the elements in the collection.

## Note

Chunked sequences have been an integral part of Clojure since version 1.1.

If we are to handle such computations efficiently, we cannot build on functions that return sequences, such as `map` and `filter`. Incidentally, the `reduce` function does not necessarily produce a sequence. It also has a couple of other interesting properties:

- The `reduce` function actually lets the supplied collection define how it is computed over or reduced. Thus, `reduce` is *collection independent*.
- Also, the `reduce` function is versatile enough to build a single value or an entirely new collection as well. For example, using `reduce` with the `*` or `+` functions will create a single valued result, while using it with the `cons` or `concat` functions can create a new collection as a result. Thus, `reduce` can *build anything*.

To summarize, the `reduce` function can be used as a premise to generalize any computation or transformation that has to be applied on a collection.

## Introducing reducers

A collection is said to be *reducible* when it defines its behavior with the `reduce` function. The binary function used by the `reduce` function along with a collection is also termed as a *reducing function*. A reducing function requires two arguments—one to represent the accumulated result of the reduction, and another to represent an input value that has to be combined into the result. Several reducing functions can be composed into

one, which effectively changes how the `reduce` function processes a given collection. This composition is done using *reducing function transformers*, or simply *reducers*.

The use of sequences and laziness can be compared to using reducers to perform a given computation by Rich Hickey's infamous pie-maker analogy. Suppose a pie-maker has been supplied a bag of apples, with an intent to *reduce* the apples to a pie. There are a couple transformations needed to perform this task. First, the stickers on all the apples have to be removed, as in we *map* a function to "take the sticker off" over the apples in the collection. Also, all the rotten apples will have to be removed, which is analogous to using the `filter` function to remove elements from a collection. Instead of performing this work herself, the pie-maker delegates it to her assistant. The assistant could first take the stickers off of all the apples, thus producing a new collection, and then take out the rotten apples to produce another new collection, which illustrates the use of lazy sequences. But then, the assistant would be doing unnecessary work by removing the stickers from the rotten apples, which will have to be discarded later.

On the other hand, the assistant could delay this work until the actual reduction of the processed apples into a pie is performed. Once the work is actually needed to be performed, the assistant will compose the two tasks of *mapping* and *filtering* the collection of apples, thus avoiding any unnecessary work. This case depicts the use of reducers to compose and transform the tasks needed to effectively reduce the collection of apples into a pie. Thus, the use of intermediary collections between each transformation is avoided, which is an optimization in terms of memory allocations performed to produce the result.

Of course, a smart assistant would simply discard the rotten apples first, which is essentially filtering the apples before mapping them. However, not all recipes are that trivial, and moreover, we can achieve a more interesting optimization through the use of reducers—parallelism. By using

reducers, we create a *recipe* of tasks to reduce a collection of apples into a pie that can be parallelized. Also, all processing is delayed until the final reduction, instead of dealing with collections as intermediary results of each task. This is the gist of how reducers achieve performance through function composition and parallelization.

## Note

The following namespaces must be included in your namespace declaration for the upcoming examples:

```
(ns my-namespace
  (:require [clojure.core.reducers :as r]))
```

The `clojure.core.reducers` namespace requires Java 6 with the `jsr166y.jar` JAR or Java 7+ for fork/join support.

Let's now briefly explore how reducers are actually implemented. Functions that operate on sequences use the `clojure.lang.ISeq` interface to abstract the behavior of a collection. In the case of reducers, the common interface that we must build upon is that of a reducing function. As we mentioned earlier, a reducing function is a two-arity function in which the first argument is the accumulated result so far and the second argument is the current input that has to be combined with the first argument. The process of performing a computation over a collection and producing some result can be generalized into three distinct cases. They can be described as follows:

- A new collection with the same number of elements as the collection it is supplied needs to be produced. This *one-to-one* case is analogous to using the `map` function.
- The computation *shrinks* the supplied collection by removing elements from it. This can be done using the `filter` function.
- The computation could also be *expansive*, in which case it produces a new collection that contains an increased number of elements. This is like what the `mapcat` function does.

These cases depict the different ways in which a collection can be transformed into the desired result. Any computation, or reduction, over a collection can be thought of as an arbitrary sequence of such transformations. These transformations are represented by *transformers*, which are essentially functions that transform a reducing function. They can be implemented as shown in *Example 3.3*:

```
(defn mapping [f]
  (fn [rf]
    (fn [result input]
      (rf result (f input)))))

(defn filtering [p?]
  (fn [rf]
    (fn [result input]
      (if (p? input)
          (rf result input)
          result)))))

(defn mapcatting [f]
  (fn [rf]
    (fn [result input]
      (reduce rf result (f input)))))
```

### *Example 3.3: Transformers*

The `mapping`, `filtering`, and `mapcatting` functions in *Example 3.3* represent the core logic of the `map`, `filter`, and `mapcat` functions respectively. All of these functions are transformers that take a single argument and return a new function. The returned function transforms a supplied reducing function, represented by `rf`, and returns a new reducing function, created using the expression `(fn [result input] ... )`. Functions returned by the `mapping`, `filtering`, and `mapcatting` functions are termed as *reducing function transformers*.

The `mapping` function applies the `f` function to the current input, represented by the `input` variable. The value returned by the function `f` is then combined with the accumulated result, represented by `result`, using

the reducing function `rf`. This transformer is a frighteningly pure abstraction of the standard `map` function that applies a function `f` over a collection. The `mapping` function makes no assumptions of the structure of the collection it is supplied or how the values returned by the function `f` are combined to produce the final result.

Similarly, the `filtering` function uses a predicate `p?` to check whether the current input of the reducing function `rf` must be combined into the final result, represented by `result`. If the predicate is not true, then the reducing function will simply return the value `result` without any modification. The `mapcatting` function uses the `reduce` function to combine the value `result` with the result of the expression `(f input)`. In this transformer, we can assume that the function `f` will return a new collection and the reducing function `rf` will somehow combine two collections.

One of the foundations of the `reducers` library is the `CollReduce` protocol defined in the `clojure.core.protocols` namespace. This protocol abstracts the behavior of a collection when it is passed as an argument to the `reduce` function, and is declared as shown in *Example 3.4*:

```
(defprotocol CollReduce
  (coll-reduce [coll rf init]))
```

#### *Example 3.4: The CollReduce protocol*

The `clojure.core.reducers` namespace defines a `reducer` function that creates a reducible collection by dynamically extending the `CollReduce` protocol, as shown in *Example 3.5*:

```
(defn reducer
  ([coll xf]
   (reify
     CollReduce
     (coll-reduce [_ rf init]
       (coll-reduce coll (xf rf) init)))))
```

#### *Example 3.5: The reducer function*

The `reducer` function combines a collection `coll` and a reducing function transformer `xf`, which is returned by the `mapping`, `filtering`, and `mapcatting` functions, to produce a new reducible collection. When `reduce` is invoked on a reducible collection, it will ultimately ask the collection to reduce itself using the reducing function returned by the expression `(xf rf)`. Using this mechanism, several reducing functions can be composed into a single computation to be performed over a given collection. Also, the `reducer` function needs to be defined only once, and the actual implementation of `coll-reduce` is provided by the collection supplied to the `reducer` function.

Now, we can redefine the `reduce` function to simply invoke the `coll-reduce` function implemented by a given collection, as shown in *Example 3.6*:

```
(defn reduce
  ([rf coll]
   (reduce rf (rf) coll))
  ([rf init coll]
   (coll-reduce coll rf init)))
```

### *Example 3.6: Redefining the reduce function*

As shown in *Example 3.6*, the `reduce` function delegates the job of reducing a collection to the collection itself using the `coll-reduce` function. Also, the `reduce` function will use the reducing function `rf` to also supply the `init` argument when it is not specified. An interesting consequence of this definition of `reduce` is that the function `rf` must produce an *identity value* when supplied no arguments. The standard `reduce` function also uses the `CollReduce` protocol to delegate the job of reducing a collection to the collection itself, but will also fall back on the default definition of `reduce` in case the supplied collection does not implement the `CollReduce` protocol.

## Note

Since Clojure 1.4, the `reduce` function allows a collection to define how it is reduced using the `clojure.core.CollReduce` protocol. Clojure 1.5 introduced the `clojure.core.reducers` namespace that extends the use of this protocol.

All of the standard Clojure collections, namely lists, vectors, sets, and maps, implement the `CollReduce` protocol. The `reducer` function can be used to build a sequence of transformations to be applied to a collection when it is passed as an argument to the `reduce` function. This can be demonstrated as follows:

```
user> (r/reduce + 0 (r/reducer [1 2 3 4] (mapping inc)))
14
user> (reduce + 0 (r/reducer [1 2 3 4] (mapping inc)))
14
```

In the preceding output, the `mapping` function is used with the `inc` function to create a reducing function transformer that increments all the elements in a given collection. This transformer is then combined with a vector using the `reducer` function to produce a reducible collection. The call to `reduce` in both of the preceding statements is transformed into the expression `(reduce + [2 3 4 5])`, thus producing the result 14. We can now redefine the `map`, `filter`, and `mapcat` functions using the `reducer` function, as shown in *Example 3.7*:

```
(defn map [f coll]
  (reducer coll (mapping f)))

(defn filter [p? coll]
  (reducer coll (filtering p?)))

(defn mapcat [f coll]
  (reducer coll (mapcatting f)))
```

*Example 3.7: Redefining the map, filter and mapcat functions using the reducer form*

As shown in *Example 3.7*, the `map`, `filter`, and `mapcat` functions are now

simply compositions of the `reducer` form with the `mapping`, `filtering`, and `mapcatting` transformers respectively.

## Note

The definitions of `CollReduce`, `reducer`, `reduce`, `map`, `filter`, and `mapcat` as shown in this section are simplified versions of their actual definitions in the `clojure.core.reducers` namespace.

The definitions of the `map`, `filter`, and `mapcat` functions shown in *Example 3.7* have the same shape as the standard versions of these functions, as shown here:

```
user> (r/reduce + (r/map inc [1 2 3 4]))  
14  
user> (r/reduce + (r/filter even? [1 2 3 4]))  
6  
user> (r/reduce + (r/mapcat range [1 2 3 4]))  
10
```

Hence, the `map`, `filter`, and `mapcat` functions from the `clojure.core.reducers` namespace can be used in the same way as the standard versions of these functions. The `reducers` library also provides a `take` function that can be used as a replacement for the standard `take` function. We can use this function to reduce the number of calls to the `square-with-side-effect` function (from *Example 3.1*) when it is mapped over a given vector, as shown here:

```
user> (def mapped (r/map square-with-side-effect [0 1 2 3 4 5]))  
# 'user/mapped  
user> (reduce + (r/take 3 mapped))  
Side-effect: 0  
Side-effect: 1  
Side-effect: 2  
Side-effect: 3  
5
```

Thus, using the `map` and `take` functions from the `clojure.core.reducers`

namespace as shown here avoids applying the `square-with-side-effect` function to all five elements in the vector `[0 1 2 3 4 5]` as only the first three are required.

The `reducers` library also provides variants of the standard `take-while`, `drop`, `flatten`, and `remove` functions, which are based on reducers. Effectively, functions based on reducers will require a lesser number of allocations than sequence-based functions, thus leading to an improvement in performance. For example, consider the `process` and `process-with-reducer` functions shown in *Example 3.8*:

```
(defn process [nums]
  (reduce + (map inc (map inc (map inc nums)))))

(defn process-with-reducer [nums]
  (reduce + (r/map inc (r/map inc (r/map inc nums)))))
```

*Example 3.8: Functions to process a collection of numbers using sequences and reducers*

The `process` function in *Example 3.8* applies the `inc` function over a collection of numbers represented by `nums` using the `map` function. The `process-with-reducer` function performs the same action, but uses the reducer variant of the `map` function. The `process-with-reducer` function will take a lesser amount of time to produce its result from a large vector when compared to the `process` function, as shown here:

```
user> (def nums (vec (range 1000000)))
#'user/nums
user> (time (process nums))
"Elapsed time: 471.217086 msecs"
500002500000
user> (time (process-with-reducer nums))
"Elapsed time: 356.767024 msecs"
500002500000
```

The `process-with-reducer` function gets a slight performance boost as it requires a lesser number of memory allocations than the `process` function.

We should note that the available memory should be large enough to load the entire file, or else we could run out of memory. The performance of this computation can be improved by a greater scale if we can somehow parallelize it, and we shall examine how this can be done in the following section.

# Using fold to parallelize collections

A collection that implements the `CollReduce` protocol is still sequential by nature. Using the `reduce` function with `CollReduce` does have a certain amount of performance gain, but it still processes elements in a collection in a sequential order. The most obvious way to improve the performance of a computation that is performed over a collection is parallelization. Such computations can be parallelized if we ignore the ordering of elements in a given collection to produce the result of the computation. In the reducers library, this is implemented based on the *fork/join model* of parallelization from the `java.util.concurrent` namespace. The fork/join model essentially partitions a collection over which a computation has to be performed into two halves and processes each partition in parallel. This halving of the collection is done in a recursive manner. The granularity of the partitions affects the overall performance of a computation modeled using fork/join. This means that if a fork/join strategy is used to recursively partition a collection into smaller collections that contain a single element each, the overhead of the mechanics of fork/join would actually bring down the overall performance of the computation.

## Note

A fork/join based method of parallelization is actually implemented in the `clojure.core.reducers` namespace using the `ForkJoinTask` and `ForkJoinPool` classes from the `java.util.concurrent` namespace in Java 7. In Java 6, it is implemented in the `ForkJoinTask` and `ForkJoinPool` classes of the `jsr166y` namespace. For more information on the Java fork/join framework, visit

<https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html>.

The parallelization of such computations using reducers is quite different from how it is handled in MapReduce-based libraries. In case of reducers, the elements are first reduced through a number of transformations into a smaller number of elements and then finally, combined to create the result.

This contrasts with how a MapReduce strategy models such a computation, in which the elements of a collection are mapped through several transformations and a final reduction step is used to produce the final result. This distinguishes the MapReduce model of parallel computation with the *reduce-combine* model used by the `reducers` library. This methodology of parallelization using a reduce-combine strategy is implemented by the `fold` function in the `clojure.core.reducers` namespace.

## Note

In Clojure, the `fold` function refers to a parallelizable computation, which is very different from the traditional fold left (`foldl`) and fold right (`foldr`) functions in other functional programming languages such as Haskell and Erlang. The `reduce` function in Clojure actually has the same sequential nature and semantics as the `foldl` function in other languages.

The `fold` function parallelizes a given computation over a collection using fork/join based threads. It implements the reduce-combine strategy that we previously described and executes the `reduce` function in parallel over equally partitioned segments of a given collection. The results produced by these parallel executions of the `reduce` function are finally combined using a *combining function*. Of course, if the supplied collection is too small to actually gain any performance through fork/join based parallelization, a `fold` form will simply call the `reduce` function on a single thread of execution. The `fold` function thus represents a *potentially parallelizable* computation over a collection. Due to this nature of `fold`, we should avoid performing IO and other side effects based on sequential ordering when using the `fold` form.

The `fold` function allows a collection to define how it is *folded* into the result, which is similar to the semantics of the `reduce` function. A collection is said to be *foldable* if it implements the `CollFold` protocol from the `clojure.core.reducers` namespace. The `reducers` library extends the `CollFold` protocol for the standard vector and map collection

types. The parallelization of these implementations of `CollFold` is done using fork/join based parallelism. The definition of the `CollFold` protocol is shown in *Example 3.9*:

```
(defprotocol CollFold
  (coll-fold [coll n cf rf]))
```

*Example 3.9: The CollFold protocol*

The `CollFold` protocol defines a `coll-fold` function, which requires four arguments—a collection `coll`, the number of elements `n` in each segment or partition of the collection, a combining function `cf`, and a reducing function `rf`. A foldable collection must implement this protocol, as well as the `clojure.core.protocols.CollReduce` protocol, as a call to `fold` on a given collection may fall back to a single-threaded execution of the `reduce` function.

To create a foldable collection from a collection and a reduction function transformer, the reducers library defines a `folder` function with similar semantics as the `reducer` function. This function is implemented as shown in *Example 3.10*:

```
(defn folder
  ([coll xf]
   (reify
     CollReduce
     (coll-reduce [_ rf init]
       (coll-reduce coll (xf rf) init)))
     CollFold
     (coll-fold [_ n cf rf]
       (coll-fold coll n cf (xf rf))))))
```

*Example 3.10: The folder function*

The `folder` function creates a new foldable and reducible collection from the collection `coll` and the reduction function transformer `xf`. This composition of the `xf` and `rf` functions is analogous to that performed by

the `reducer` function described in *Example 3.5*. Apart from the `xf` and `rf` functions, the `coll-fold` function also requires a combining function `cf` with which the results of the potentially parallel executions of the `reduce` function are combined. Similar to the `reduce` function, the `fold` function passes on the responsibility of actually folding a given collection to the collections implementation of the `coll-fold` function. An implementation of the `fold` function is described in *Example 3.11*:

```
(defn fold
  ([rf coll]
   (fold rf rf coll))
  ([cf rf coll]
   (fold 512 cf rf coll))
  ([n cf rf coll]
   (coll-fold coll n cf rf)))
```

### *Example 3.11: The fold function*

As shown in *Example 3.11*, the `fold` function calls the `coll-fold` function of the collection `coll` using the reducing function `rf` and the combining function `cf`. The `fold` function can also specify the number of elements `n` in each segment processed by the `reduce` function, which defaults to 512 elements. We can also avoid specifying the combining function `cf` to the `fold` function, in which case the reducing function `rf` itself will be used as the combining function.

An interesting aspect of the combining and reducing functions used by the `fold` form is that they must be *associative* in nature. This guarantees that the result of the `fold` function will be independent of the order in which the elements in a given collection are combined to produce the given result. This allows us to parallelize the execution of the `fold` function over segments of a given collection. Also, analogous to the reducing function required by the `reduce` form, the `fold` function requires the combining and reducing functions to produce an *identity value* when invoked with no arguments. In functional programming, a function that is both associative and provides an identity value is termed as a **monoid**. The

`clojure.core.reducers` namespace provides the `monoid` function, described in *Example 3.12*, to create such a function that can be used as the combining function or the reducing function supplied to a `fold` form:

```
(defn monoid
  [op ctor]
  (fn
    ([] (ctor))
    ([a b] (op a b))))
```

### *Example 3.12: The monoid function*

The `monoid` function shown in *Example 3.12* produces a function that calls a function `op` when supplied with two arguments `a` and `b`. When the function returned by the `monoid` function is called with no arguments, it will produce an identity value of the operation by simply calling the `ctor` function with no arguments. This function allows us to easily create a combining function to be used with the `fold` function from any arbitrary functions `ctor` and `op`.

We can now redefine the `map`, `filter`, and `mapcat` operations as compositions of the `folder` function and the `mapping`, `filtering`, and `mapcatting` transformers defined in *Example 3.3*, as shown in *Example 3.13*:

```
(defn map [f coll]
  (folder coll (mapping f)))

(defn filter [p? coll]
  (folder coll (filtering p?)))

(defn mapcat [f coll]
  (folder coll (mapcatting f)))
```

### *Example 3.13: Redefining the map, filter and mapcat functions using the folder form*

## Note

The definitions of `folder`, `fold`, `monoid`, `map`, `filter`, and `mapcat` as shown in this section are simplified versions of their actual definitions in the `clojure.core.reducers` namespace.

The `reducers` library also defines the `foldcat` function. This function is a high-performance variant of the `reduce` and `conj` functions. In other words, the evaluation of the expression `(foldcat coll)` will be significantly faster than that of the expression `(reduce conj [] coll)`, where `coll` is a reducible or foldable collection. Also, the collection returned by the `foldcat` function will be a foldable collection.

Let's now use the `fold` and `map` functions to improve the performance of the `process` and `process-with-reducer` functions from *Example 3.8*. We can implement this as shown in *Example 3.14*:

```
(defn process-with-folder [nums]
  (r/fold + (r/map inc (r/map inc (r/map inc nums))))))
```

*Example 3.14: A function to process a collection of numbers using a fold form*

The performance of the `process-with-folder` function with a large vector can be compared to the `process` and `process-with-reducer` functions, as shown here:

```
user> (def nums (vec (range 1000000)))
#'user/nums
user> (time (process nums))
"Elapsed time: 474.240782 msecs"
500002500000
user> (time (process-with-reducer nums))
"Elapsed time: 364.945748 msecs"
500002500000
user> (time (process-with-folder nums))
"Elapsed time: 241.057025 msecs"
500002500000
```

It is observed from the preceding output that the `process-with-folder`

function performs significantly better than the process and process-with-reducer functions due to its inherent use of parallelism. In summary, reducers improve the performance of a computation that has to be performed over a collection using fork/join-based parallelism.

# Processing data with reducers

We will now study a simple example that depicts the use of reducers in efficiently processing large collections. For this example, we will use the `iota` library (<https://github.com/thebusby/iota>) to handle large memory-mapped files. The usage of the `iota` library with large files is encouraged as an efficient alternative to using concrete collections. For example, loading the records in a 1 GB TSV file as strings into a Clojure vector would consume over 10 GB of memory due to the inefficient storage of Java strings. The `iota` library avoids this by efficiently indexing and caching the contents of a large file, and this is done with much lower amount of memory overhead when compared to using concrete collections.

## Note

The following library dependencies are required for the upcoming examples:

```
[iota "1.1.2"]
```

Also, the following namespaces must be included in your namespace declaration:

```
(ns my-namespace
  (:require [iota :as i]
            [clojure.string :as cs]
            [clojure.core.reducers :as r]))
```

The following examples can be found in `src/m_clj/c3/io.clj` of the book's source code.

Suppose we have a large TSV file that contains several thousands of records. Each record represents a person, and can be assumed to have five fields, as shown in the following data:

```
brown brian :m :child :east
smith bill :f :child :south
jones jill :f :parent :west
```

Each record contains two strings and three keywords. The first two string fields of a record represent the last and first name of a person, the third column is a keyword that indicates the gender of a person, and the fourth column is a keyword that identifies a person as a parent or a child. Finally, the fifth column is a keyword that represents an arbitrary direction.

## Note

The following example expects the content shown previously to be present in the file `resources/data/sample.tsv`, relative to the root of your Leiningen project.

The `seq` and `vec` functions from the `iota` library can be used to create a sequence and a vector representation of a memory-mapped file. These objects can then be used to access the file in a performant way. Both of the `seq` and `vec` functions require a file path to be passed to them as the first argument. The `vec` function will index the supplied file in *chunks*, and we can specify the size of each chunk as the second argument to the `vec` function. The `seq` function performs buffered reads of the supplied file as required, similar to the way a lazy sequence is realized. The size of the buffer used by this resulting sequence can be specified as the second argument to the `seq` function. Both the `seq` and `vec` functions split the contents of a file by a predefined byte-separator into records represented as strings. These functions also accept an optional third argument to indicate the byte separator between records in the supplied file. The `vec` function is slower than the `seq` function as it must index the records in the file, which can be demonstrated as follows:

```
user> (time (def file-as-seq (i/seq
"resources/data/sample.tsv")))
"Elapsed time: 0.905326 msecs"
#'user/file-as-seq
user> (time (def file-as-vec (i/vec
```

```
"resources/data/sample.tsv")))
"Elapsed time: 4.95506 msecs"
#'user/file-as-vec
```

Both the statements shown here load the `sample.tsv` file into Clojure data structures. As expected, the `vec` function takes a bit more time than the `seq` function to return a value. The values returned by `seq` and `vec` can be treated just like any other collection. Naturally, iterating over a vector returned by the `vec` function is much faster than using a sequence, as shown here:

```
user> (time (def first-100-lines (doall (take 100 file-as-seq))))
"Elapsed time: 63.470598 msecs"
#'user/first-100-lines
user> (time (def first-100-lines (doall (take 100 file-as-vec))))
"Elapsed time: 0.984128 msecs"
#'user/first-100-lines
```

We will now demonstrate a couple of ways to query the data in the `sample.tsv` file using reducers and the `iota` library. We will need to first define a function that converts a collection of records into collections of columnar values from their string-based representations. This can be implemented using the reducer based `map` and `filter` functions, as shown in the `into-records` function in *Example 3.15*:

```
(defn into-records [file]
  (->> file
    (r/filter identity)
    (r/map #(cs/split % #"[\t]"))))
```

*Example 3.15: A function to convert a memory-mapped file into a reducible collection*

Now, let's say we need to compute the total number of females from the records in the `sample.tsv` file. We can implement a function to perform this computation using the `map` and `fold` functions, as shown in the `count-females` function in *Example 3.16*:

```
(defn count-females [coll]
  (-> coll
    (r/map #(-> (nth % 2)
                  ({"":m" 0 ":f" 1})))
    (r/fold +)))
```

We can query the total number of females in the `file-as-seq` and `file-as-vec` collections by composing the `into-records` and `count-females` functions. This can be done using the `->` threading form, as shown here:

```
user> (-> file-as-seq into-records count-females)
10090
user> (-> file-as-vec into-records count-females)
10090
```

Similarly, the reducer-based `map` and `filter` functions can be used to fetch the first names of all the children with the same last name or family in a given collection, as implemented by the `get-children-names-in-family` function in *Example 3.17*:

```
(defn get-children-names-in-family [coll family]
  (-> coll
    (r/filter #(and (= (nth % 0) family)
                  (= (nth % 3) ":child")))
    (r/map #(nth % 1))
    (into [])))
```

*Example 3.17: A function to get the first names of all children in a collection of persons*

The `into-records` and `get-children-names-in-family` functions can be composed together to query the first names of all children with the last name "brown" from the available data, as shown here:

```
user> (-> file-as-seq into-records
  (get-children-names-in-family "brown"))
["sue" "walter" ... "jill"]
user> (-> file-as-vec into-records
  (get-children-names-in-family "brown"))
["sue" "walter" ... "jill"]
```

The `iota` library provides a couple more useful functions to handle large text files:

- The `numbered-vec` function will create a vector representing a memory-mapped file in which each string representing a record will be prepended with its position in the given file.
- The `subvec` function of the `iota` library can be used to *slice* records from a memory-mapped file returned by the `vec` and `numbered-vec` functions. Its semantics are identical to the standard `subvec` function that operates on vectors.

Reducers and the `iota` library allow us to idiomatically and efficiently handle text files containing a large number of byte-separated records. There are also several other libraries and frameworks in the Clojure ecosystem that use reducers to handle large amounts of data, and the reader is encouraged to explore these libraries and frameworks on their own.

# Summary

In this chapter, we explored the `clojure.core.reducers` library in detail. We had a look at how reducers are implemented and also how we can use reducers to handle large collections of data in an efficient manner. We also briefly studied the `iota` library that can be used with reducers to handle large amounts of data stored in text files.

In the following chapter, we will explore Clojure macros.

# Chapter 4. Metaprogramming with Macros

Programmers often stumble into situations where they would like to add features or constructs to their programming language of choice. Generally, if a feature would have to be added to a language, the language's compiler or interpreter would need some modification. Alternatively, Clojure (and other Lisps as well) uses *macros* to solve this problem. The term *metaprogramming* is used to describe the ability to generate or manipulate a program's source code by using another program. Macros are a metaprogramming tool that allow programmers to easily add new features to their programming language.

Lisps are not the only languages with support for macro-based metaprogramming. For example, in C and C++, macros are handled by the compiler's preprocessor. In these languages, before a program is compiled, all macro calls in the program's source code are replaced by their definitions. In this sense, macros are used to generate code through a form of text substitution during the compilation phase of a program. On the other hand, Lisps allow programmers to transform or rewrite code when macros are interpreted or compiled. Macros can thus be used to concisely encapsulate recurring patterns in code. Of course, this can be done in languages without macros, as well, without much hassle. But macros allow us to encapsulate patterns in code in a clean and concise manner. As we will see ahead in this chapter, there's nothing equivalent to Lisp macros in other programming languages in terms of clarity, flexibility, and power. Lisps are truly leaps ahead of other programming languages in terms of metaprogramming capabilities.

The rabbit hole of macros in Lisps goes deep enough that there are entire books that talk about them. *Mastering Clojure Macros* by Colin Jones is one among these, and this publication describes the various patterns in

which macros can be used in great detail. In this chapter, we will explore the foundational concepts behind macros and their usage. We will:

- First, have a look at the basics of reading, evaluating, and transforming code in Clojure.
- Later on, we will examine how macros can be defined and used, and also study several examples based on macros. We will also describe how we can handle platform-specific code using *reader conditionals*.

# Understanding the reader

The reader is responsible for interpreting Clojure code. It performs several steps to translate source code in textual representation into executable machine code. In this section, we will briefly describe these steps performed by the reader to illustrate how the reader works.

Clojure and other languages from the Lisp family are **homoiconic**. In a homoiconic language, the source code of a program is represented as a plain data structure. This means that all the code written in a Lisp language is simply a bunch of nested lists. Thus, we can manipulate programs' code just like any other list of values. Clojure has a few more data structures, such as vectors and maps in its syntax, but they can be handled just as easily. In languages that are not homoiconic, any expression or statement in a program has to be translated into an internal data structure termed as a *parse tree*, or *syntax tree*, when the program is compiled or interpreted. In Lisps, however, an expression is already in the form of a syntax tree, since a tree is really just another name for a nested list. In other words, there is no distinction between an expression and the syntax tree it produces. One might also opine that this design tricks programmers into writing code directly as a syntax tree. This distinguishing aspect of Lisps is succinctly captured by the following axiom: *Code is Data*.

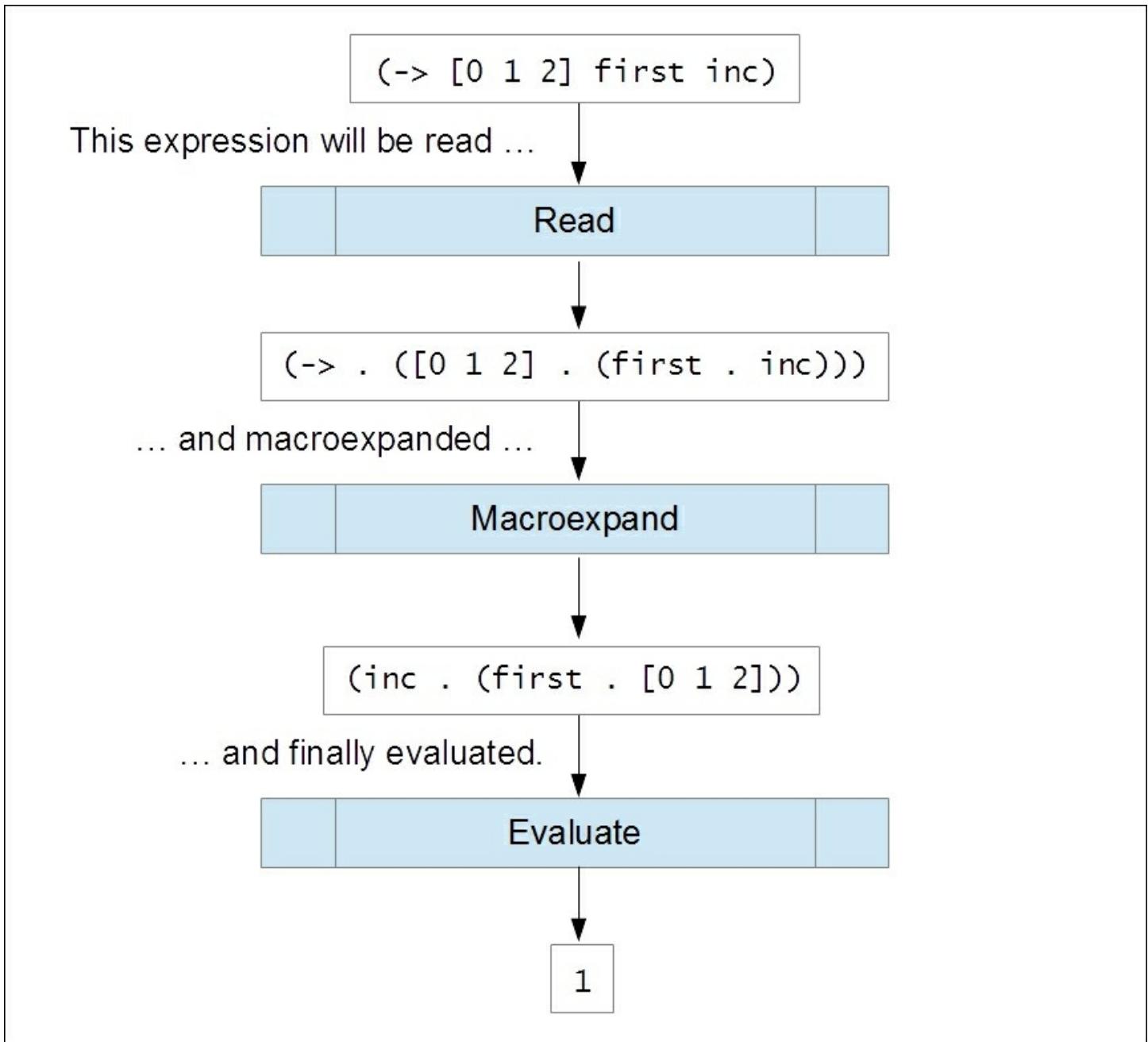
Let's first take a look at the most rudimentary representation of code and data in Lisps—an **s-expression**. Any expression comprises of *values* and *symbols*, where the symbols represent variables being used. A nested list of

symbols is known as a *symbolic expression*, *s-expression*, or *sexp*. All source code in Clojure is represented as s-expressions. A symbolic expression is formally defined as:

- An atom, which refers to a single symbol or literal value.
- A combination of two s-expressions  $x$  and  $y$ , represented as  $(x . y)$ . Here, the dot ( $.$ ) is used to signify a `cons` operation.

Using this recursive definition, a list of symbols  $(x y z)$  is represented by the s-expression,  $(x . (y . (z . nil)))$  or  $(x . (y . z))$ . When an s-expression is used to represent source code, the first element of the expression represents the function used, and the rest of the elements are the arguments to the function. Of course, this is just a theoretical representation and not really Clojure code. This representation is also called *prefix notation*. This recursive structure of s-expressions is flexible enough to represent both code as well as data. In fact, s-expressions are more-or-less the only form of syntax in Clojure (and other Lisps). For example, if we wanted to add two numbers, we would use an expression with the `+` function as the first symbol, followed by the values to be added. Similarly, if we wanted to define a function, we would have to write an expression with `defn` or `def` as the first symbol in the expression. In Clojure and other Lisps, we also represent data such as lists, vectors, and maps using s-expressions.

Let's look at a simple example that depicts how Clojure code is interpreted. The expression `(-> [0 1 2] first inc)` that uses a threading macro `(->)` will be interpreted in three distinct steps. This expression will be read, *macroexpanded*, and evaluated to the value `1`, as illustrated here:



The reader will first parse textual representations of s-expressions from a Clojure program's source code. Once a program's source code is read into s-expressions, all macro calls in the code are replaced by their definitions. This transformation of macro calls in a program is called *macroexpansion*. Lastly, the resulting s-expressions from the macroexpansion phase are evaluated by the Clojure runtime. In the evaluation phase, bytecode is generated from the supplied expressions, loaded into memory, and

executed. In short, code in a program's source code is read, transformed through macros, and finally evaluated. Also, macroexpansion happens immediately after a program's source code is parsed, thus allowing the program to internally transform itself before it is evaluated. This transformation of code is exactly what macros are used to achieve.

## Note

In Clojure, the reader only reads code and performs macroexpansion. The generation of bytecode is done by the analyzer and the emitter, and this generated bytecode is evaluated by the JVM.

All Clojure code is translated to *reader forms* and *special forms* before it is evaluated. Special forms are constructs, such as `quote` and `let*`, that are implemented directly as bytecode for the underlying runtime, such as the JVM for Clojure or the Rhino JavaScript runtime for ClojureScript. Interestingly, Clojure source code is composed mostly of reader forms, and these reader forms are implemented in Clojure itself. The reader also transforms certain characters and forms called *reader macros* as soon as they are read. There are several reader macros in the Clojure language, as described in the following table:

Reader macro	Usage
<code>\x</code>	This is a character literal.
<code>:</code>	This is used to comment. It ignores the rest of the line.
<code>(.method o)</code>	This is a native method call. It is rewritten to a dot (.) form as <code>(. o method)</code> . Also, <code>o</code> must be a native object.
<code>@x or @(...)</code>	This is the dereference operator. It is used with reference types and is rewritten to a <code>deref</code> form.
<code>^{ ... }</code>	This is the metadata map to be used with a form. It is rewritten to a <code>with-meta</code> form.

'x or '( ... )	This is a quote.
`x or `( ... )	This is a syntax quote.
~x or ~( ... )	This is used to unquote.
~@x or ~@( ... )	This is a splicing unquote.
#_x or #_( ... )	This ignores the next form. #_ should be preferred over the <code>comment</code> form to comment out code, since <code>comment</code> actually returns <code>nil</code> .
#'x	This is a var quote. It is equivalent to <code>(var x)</code> .
#=x or #=( ... )	This will read-evaluate an expression.
#?( ... )	This is a reader conditional form.
#?@( ... )	This is a reader conditional splicing form.

We have encountered quite a few of the preceding reader macros in the previous chapters. We will demonstrate the usage of several reader forms that are used with macros in this chapter.

## Note

At the time of writing this book, Clojure does not support user-defined reader macros.

Now that we have familiarized ourselves with the Clojure reader and how code is interpreted, let's explore the various metaprogramming constructs that help us read and evaluate code.

# Reading and evaluating code

Let's have a look at how code can be parsed and evaluated in Clojure. The most elementary way to convert text into an expression is by using the `read` function. This function accepts a `java.io.PushbackReader` instance as its first argument, as shown here:

```
user> (read (-> "(list 1 2 3)"
                     .toCharArray
                     java.io.CharArrayReader.
                     java.io.PushbackReader.))
(list 1 2 3)
```

## Note

These examples can be found in `src/m_c1j/c4/read_and_eval.clj` of the book's source code.

In this example, a string containing a valid expression is first converted into an instance of `java.io.PushbackReader` and then passed to the `read` function. It seems like a lot of unnecessary work to read a string, but it is due to the fact that the `read` function deals with streams and readers, and not strings. If no arguments are passed to the `read` function, it will create a reader from the standard input and prompt the user to enter an expression to be parsed. The `read` function has several other options as well, and you are encouraged to explore these options in the REPL on their own.

A simpler way to read an expression from a string is by using the `read-string` function. This function accepts a string as its only argument and converts the supplied string into an expression, as shown here:

```
user> (read-string "(list 1 2 3)")
(list 1 2 3)
```

The `read` and `read-string` forms can only convert strings into valid expressions. If we have to evaluate an expression, we must use the `eval`

function, as shown here:

```
user> (eval '(list 1 2 3))
(1 2 3)
user> (eval (list + 1 2 3))
6
user> (eval (read-string "(+ 1 2 3)"))
6
```

In the first statement in the preceding output, we prevent the expression `(list 1 2 3)` from being evaluated before it is passed to the `eval` function using the quote operator `'`. This technique is termed as *quoting* and we shall explore more of it later in this chapter. The `eval` function evaluates the expression `(list 1 2 3)` to the list `(1 2 3)`. Similarly, in the second statement, the expression `(list + 1 2 3)` is first evaluated as `(+ 1 2 3)` by the reader, and then the `eval` function evaluates this list to the value `6`. In the third statement, the string `"(+ 1 2 3)"` is first parsed by the `read-string` function and then evaluated by the `eval` function.

The read-evaluate macro `(#=)` can be used to force the `read` and `read-string` functions to evaluate an expression when it is parsed, as shown here:

```
user> (read (-> "#=(list 1 2 3)"
                     .toCharArray
                     java.io.CharArrayReader.
                     java.io.PushbackReader.))
(1 2 3)
user> (read-string "#=(list 1 2 3)")
(1 2 3)
```

In the preceding output, the `#=` reader macro evaluates the expression `(list 1 2 3)` when it is read by the `read` and `read-string` functions. If the `#=` macro was not used, both statements would return the expression `(list 1 2 3)` in verbatim. We can also use the `#=` macro without using `read` or `read-string`, in which case it would be equivalent to calling the `eval` function. Also, the calls to the `#=` macro can be nested any number of times, as shown here:

```
user> #=(list + 1 2 3)
6
user> (read-string "#=(list + 1 2 3)")
(+ 1 2 3)
user> (read-string "#=#=(list + 1 2 3)")
6
```

The `#=` macro makes it easy to evaluate expressions while they are being read. Oh wait! This is a potential security hazard as the `read` and `read-string` functions are evaluating arbitrary strings, even if they contain any malicious code. Thus, evaluation of code while it is being parsed is deemed unsafe. As a solution to this problem, the `*read-eval*` var can be set to `false` to prevent usage of the `#=` macro, as shown here:

```
user> (binding [*read-eval* false]
           (read-string (read-string "#=(list 1 2 3)")))
RuntimeException EvalReader not allowed when *read-eval* is
false. clojure.lang.Util.runtimeException (Util.java:221)
```

Thus, use of the `#=` macro in strings passed to the `read` and `read-string` functions will throw an error if `*read-eval*` is set to `false`. Obviously, the default value of this var is `true`. For this reason, we must avoid using the `#=` macro, or set the `*read-eval*` var to `false`, while processing the user input.

Another way to read and evaluate arbitrary strings is by using the `load-string` function. This function has the same arity as the `read-string` function, and is equivalent to calling the `eval` and `read-string` forms, as shown here:

```
user> (load-string "(+ 1 2 3)")
6
```

There are a couple of semantic differences between using the `load-string` form and a composition of the `eval` and `read-string` forms. Firstly, the behavior of the `load-string` function is not affected by the changing `*read-eval*` var, and is thus unsafe for use with arbitrary user input.

A more important difference is that the `read-string` function only parses the first expression it encounters in the string that it has passed. The `load-string` function will parse and evaluate all expressions passed to it, as shown here:

```
user> (eval (read-string "(println 1) (println 2)"))
1
nil
user> (load-string "(println 1) (println 2)")
1
2
nil
```

In the preceding output, the `read-string` form skips the second `println` form in the string that it is passed, thus printing the value `1` only. The `load-string` form, however, parses and evaluates both the `println` forms it is passed as a string, and prints both the values `1` and `2`.

The `load-reader` function is analogous to the `read` function, in the sense that it accepts a `java.io.PushbackReader` instance, from which it has to read and evaluate forms, as an argument. Another variant of `load-string` is the `load-file` function, to which we can pass the path of a file that contains source code. The `load-file` function will parse the file in the path that it is passed and evaluate all forms present in it.

## Note

Note that the `*file*` var can be used to obtain the path of the current file being executed.

So far, we have seen how code can be parsed and evaluated by the Clojure reader. There are several constructs that can be used to perform these tasks. However, evaluating arbitrary strings is not really a good idea, as the code being evaluated is insecure and may be malicious. In practice, we should always set the `*read-eval*` variable to `false` in order to prevent the evaluation of arbitrary code by functions such as `read` and `read-string`. Next, we will explore how *quoting* and *unquoting* can be used to

transform expressions.

# Quoting and unquoting code

We will now explore *quoting* and *unquoting*, which are techniques used to generate expressions based on a predefined template for an expression. These techniques are foundational in creating macros, and they help structure the code of a macro to look more like its macroexpanded form.

## Note

The following examples can be found in `src/m_clj/c4/quoting.clj` of the book's source code.

The `quote` form simply returns an expression without evaluating it. This may seem trivial, but preventing the evaluation of an expression is actually something that is not possible in all programming languages. The `quote` form is abbreviated using the apostrophe character ('). If we *quote* an expression, it is returned in verbatim, as shown here:

```
user> 'x
x
user> (quote x)
x
```

The `quote` form is quite historic in Lisp. It is one of the seven primitive operators in the original Lisp language, as described in John McCarthy's paper. Incidentally, `quote` is one among the rare special forms that are implemented in Java and not in Clojure itself. The `quote` form is used to handle variable names, or *symbols*, as values. In a nutshell, using the `quote` form, we can treat a given expression as a list of symbols and values. After all, *Code is Data*.

## Note

An apostrophe ('') represents a quoted expression only when it appears as the first character in the expression. For example, `x'` is just a variable name.

A syntax quote, written as a backtick character (`), will quote an expression and allows *unquoting* to be performed within it. This construct allows us to create expressions just like quoting, but also has the added benefit of letting us interpolate values and execute arbitrary code in a quoted form. This has the effect of treating a predefined expression as a template with some parts left blank to be filled in later. An expression within a syntax quoted form can be unquoted using the tidal character (~). Unquoting an expression will evaluate it and insert the result into the surrounding syntax quoted form. A *splicing unquote*, written as ~@, can be used to evaluate an expression that returns a list and use the returned list of values as arguments for a form. This is something like what the `apply` form does, except that it's within the context of a syntax quote. We must note that both of these unquoting operations (~ and ~@) can only be used within a syntax quoted form. We can try out these operations in the REPL, as shown here:

```
user> (def a 1)
#'user/a
user> `(list ~a 2 3)
(clojure.core/list 1 2 3)
user> `(list ~@[1 2 3])
(clojure.core/list 1 2 3)
```

As shown here, unquoting the variable `a` in the preceding syntax quoted `list` form returns the expression `(list 1 2 3)`. Similarly, using a splicing unquote with the vector `[1 2 3]` returns the same list. On the other hand, unquoting a variable in a quoted form will expand the unquote reader macro (~) to a `clojure.core/unquote` form, as shown here:

```
user> (def a 1)
#'user/a
user> `(list ~a 2 3)
(clojure.core/list 1 2 3)
user> '(list ~a 2 3)
(list (clojure.core/unquote a) 2 3)
```

A more interesting difference between using a quote and a syntax quote is

that the latter will resolve all variable names to namespace-qualified names. This applies to function names as well. For example, let's look at the following expressions:

```
user> `(vector x y z)
(clojure.core/vector user/x user/y user/z)
user> `(~'x ~'y ~'z)
(clojure.core/vector x y z)
```

As shown in the preceding output, the variables `x`, `y`, and `z` are resolved to `user/x`, `user/y`, and `user/z` respectively by the syntax quoted form, since `user` is the current namespace. Also, the `vector` function is translated to its namespace-qualified name, shown as `clojure.core/vector`. The unquote and quote operations in succession, shown as `~'`, can be used to bypass the resolution of a symbol to a namespace-qualified name.

Quoting is supported on data structures other than lists, such as vectors, sets, and maps, as well. The effect of a syntax quote is the same on all of the data structures; it allows expressions to be unquoted within it, thus transforming the quoted form. Also, quoted forms can be nested, as in a quoted forms can contain other quoted forms. In such a case, the deepest quoted form is processed first. Consider the following quoted vectors:

```
user> `[1 :b ~(+ 1 2)]
[1 :b 3]
user> `[1 :b ' ~(+ 1 2)]
[1 :b (quote 3)]
user> `[1 ~'b ~(+ 1 2)]
[1 b 3]
```

There are a lot of interesting aspects that can be inferred from the preceding output. Firstly, keywords are apparently not interned to namespace-qualified names such as symbols. In fact, this behavior is exhibited by any value that evaluates to itself, such as keywords, `nil`, `true`, and `false`, when used in a syntax quoted form. Other than that, unquoting followed by quoting an expression in a syntax quote, shown as `'~(+ 1 2)`, will evaluate the expression and wrap it in a quote. Conversely,

unquoting a quoted symbol, shown as `~'b`, will prevent it from being resolved to a namespace-qualified name as we mentioned earlier. Let's take a look at another example that uses nested quoting, as shown here:

```
user> (def ops ['first 'second])
#'user/ops
user> `{:a (~(nth ops 0) ~'xs)
          :b (~(nth ops 1) ~'xs)}
{:b (second xs),
 :a (first xs)}
```

In the preceding output, the variables `first`, `second`, and `xs` are prevented from being interned to a namespace-qualified names using the quote (`'`) and unquote (`~`) operations in tandem. Anyone who's used older Lisps is probably cringing at this point. In practice, usage of the `~'` operation should actually be avoided. This is because preventing the resolution of a variable to a namespace-qualified name isn't really a good idea. In fact, unlike Clojure, some Lisps completely disallow it. It causes a peculiar problem called *symbol capture*, which we will see ahead while we explore macros.

# Transforming code

As previously described in this chapter, it's trivial to read and evaluate code in Clojure using the `read` and `eval` functions and their variants. Instead of evaluating code right after it is parsed, we can use macros to first transform code programmatically using quoting and unquoting, and then evaluate it. Thus, macros help us define our own constructs that rewrite and transform expressions passed to them. In this section, we will explore the basics of creating and using macros.

## Expanding macros

Macros need to be *expanded* when they are called. All Clojure code is read, macroexpanded, and evaluated by the reader as we described earlier. Let's now take a look at how macroexpansion is performed. As you may have guessed already, this is done using plain Clojure functions.

Interestingly, the reader of the Clojure runtime also uses these functions to process a program's source code. As an example, we will examine how the `->` threading macro is macroexpanded. The `->` macro can be used as shown here:

```
user> (-> [0 1 2] first inc)
1
user> (-> [0 1 2] (-> first inc))
1
user> (-> (-> [0 1 2] first) inc)
1
```

### Note

These examples can be found in `src/m_c1j/c4/macroexpand.clj` of the book's source code.

All of the three expressions using the `->` macro in the preceding output will be evaluated to the value `1`. This is due to the fact that they are all

macroexpanded to produce the same final expression. How can we claim that? Well, we can prove it using the `macroexpand-1`, `macroexpand`, and `clojure.walk/macroexpand-all` functions. The `macroexpand` function returns the complete macroexpansion of a form, as shown here:

```
user> (macroexpand '(-> [0 1 2] first inc))
(inc (first [0 1 2]))
```

The expression using the `->` threading macro is thus transformed to the expression `(inc (first [0 1 2]))`, which evaluates to the value `1`. In this way, the `macroexpand` function allows us to inspect the macroexpanded form of an expression.

The `macroexpand-1` function returns the first expansion of a macro. In fact, the `macroexpand` function simply applies the `macroexpand-1` function repeatedly until no more macroexpansion can be performed. We can inspect how the expression `(-> [0 1 2] (-> first inc))` is macroexpanded using these functions:

```
user> (macroexpand-1 '(-> [0 1 2] (-> first inc)))
(-> [0 1 2] first inc)
user> (macroexpand '(-> [0 1 2] (-> first inc)))
(inc (first [0 1 2]))
```

The `macroexpand` function has a small limitation. It only repeatedly macroexpands an expression until the first form in the expression is a macro. Hence, the `macroexpand` function will not completely macroexpand the expression `(-> (-> [0 1 2] first) inc)`, as shown here:

```
user> (macroexpand-1 '(-> (-> [0 1 2] first) inc))
(inc (-> [0 1 2] first))
user> (macroexpand '(-> (-> [0 1 2] first) inc))
(inc (-> [0 1 2] first))
```

As shown in the preceding example, the `macroexpand` function will return the same macroexpansion as `macroexpand-1`. This is because the second call to the `->` macro is not the first form in the result of the first

macroexpansion for the previous expression. In such cases, we can use the `macroexpand-all` function from the `clojure.walk` namespace to macroexpand a given expression regardless of the positions of macro calls in it, as shown here:

```
user> (clojure.walk/macroexpand-all '(-> (-> [0 1 2] first) inc))
(inc (first [0 1 2]))
```

Thus, all three expressions using the `->` macro as examples are macroexpanded to the same expression `(inc (first [0 1 2]))`, which is evaluated to the value 1.

## Note

The `macroexpand-1`, `macroexpand`, and `clojure.walk/macroexpand-all` functions will have no effect on an expression that does not contain any macros.

The `macroexpand-1` and `macroexpand` functions are indispensable tools for debugging user-defined macros. Additionally, the `clojure.walk/macroexpand-all` function can be used in situations where the `macroexpand` function does not completely macroexpand a given expression. The Clojure reader also uses these functions for macroexpanding a program's source code.

## Creating macros

Macros are defined using the `defmacro` form. A macro name, a vector of arguments for the macro, an optional doc-string, and the body of the macro have to be passed to this form. We can also specify multiple arities for a macro. Its similarity to the `defn` form is quite obvious. Unlike a `defn` form, however, a macro defined using the `defmacro` form will not evaluate the arguments passed to it. In other words, the arguments passed to a macro are implicitly quoted. For example, we can create a couple of macros to rewrite an s-expression in infix and postfix notation, as shown in *Example 4.1*.

```
(defmacro to-infix [expr]
  (interpose (first expr) (rest expr)))

(defmacro to-postfix [expr]
  (concat (rest expr) [(first expr)]))
```

*Example 4.1: Macros to transform a prefix expression*

## Note

These examples can be found in `src/m_clj/c4/defmacro.clj` of the book's source code.

Each of the macros in *Example 4.1* describes an elegant way to rewrite an expression `expr` by treating it as a generic sequence. The function being called in the expression `expr` is extracted using the `first` form, and its arguments are obtained using the `rest` form. To convert the expression to its infix form, we use the `interpose` function. Similarly, the `postfix` form of the expression `expr` is generated using the `concat` form. We can use the `macroexpand` function to inspect the expression generated by the `to-infix` and `to-postfix` macros, as shown here:

```
user> (macroexpand '(to-infix (+ 0 1 2)))
(0 + 1 + 2)
user> (macroexpand '(to-postfix (+ 0 1 2)))
(0 1 2 +)
```

## Note

The expression `x + y` is said to be written in an *infix* notation. The *prefix* notation of this expression is `+ x y`, and its *postfix* notation is `x y +`.

In this way, by transforming expressions we can effectively modify the language. It's that simple! The basis of the `to-infix` and `to-postfix` macros in *Example 4.1* are that we can treat the terms of an expression as a sequence of elements and manipulate them by using sequence functions such as `interpose` and `concat`. Of course, the preceding example was

simple enough such that we could avoid the use of quoting altogether. The `defmacro` form can also be used in combination with quoting to easily rewrite more complex expressions. The same rule can be applied to *any* form of Clojure code.

Interestingly, macros are internally represented as functions, and this can be verified by dereferencing the fully qualified name of a macro and using the `fn?` function, as shown here:

```
user> (fn? #'to-infix)
true
user> (fn? #'to-postfix)
true
```

## Note

At the time of writing this book, ClojureScript only supports macros written in Clojure. Macros have to be referenced using the `:require-macros` keyword in a ClojureScript namespace declaration, as shown here:

```
(ns my-cljs-namespace
  (:require-macros [my-clj-macro-namespace :as macro]))
```

The `symbol` and `gensym` functions can be used to create temporary variables for use within the body of a macro. The `symbol` function returns a symbol from a name and an optional namespace, as shown here:

```
user> (symbol 'x)
x
user> (symbol "x")
x
user> (symbol "my-namespace" "x")
my-namespace/x
```

## Note

We can check whether a value is a symbol using the `symbol?` predicate.

The `gensym` function can be used to create a unique symbol name. We can

specify a prefix to be used for the returned symbol name to the `gensym` function. The prefix is defaulted to a capital G character followed by two underscores (`G__`). The `gensym` function can also be used to create a new unique keyword. We can try out the `gensym` function in the REPL, as shown here:

```
user> (gensym)
G__8090
user> (gensym 'x)
x8081
user> (gensym "x")
x8084
user> (gensym :x)
:x8087
```

As shown here, the `gensym` function creates a new symbol every time it is called. In a syntax quoted form, we can use an automatic symbol name created from a prefixed name and the `gensym` function by using the hash character (#), shown as follows:

```
user> `(let [x# 10] x#)
(clojure.core/let [x__8561__auto__ 10]
  x__8561__auto__)
user> (macroexpand `(let [x# 10] x#))
(let* [x__8910__auto__ 10]
  x__8910__auto__)
```

## Note

The `let` form is, in fact, a macro defined using the `let*` special form.

As shown in the preceding expression, all occurrences of the *auto-gensym* variable `x#` in the syntax quoted form are replaced with an automatically generated symbol name. We should note that only symbols, and not strings or keywords, can be used as a prefix for an auto-gensym symbol.

By generating unique symbols in this way, we can create *hygenic macros*, which avoid the possibility of *symbol capture* or *variable capture*, which

is an interesting problem that arises with the use of dynamically scoped variables and macros. To illustrate this problem, consider the macros defined in *Example 4.2*:

```
(defmacro to-list [x]
  `(list ~x))

(defmacro to-list-with-capture [x]
  `(list ~'x))
```

*Example 4.2: Macros to depict symbol capture*

The macros in *Example 4.2* create a new list using a `list` form and the value `x`. Of course, we wouldn't really need to use a macro here, but it is only done for the sake of demonstrating symbol capture. The `to-list-with-capture` macro *captures* the variable `x` from the surrounding scope by the use of the `~'` operation. If we use a `let` form to bind the variable name `x` with a value, we will get different results on calling the `to-list` and `to-list-with-capture` macros, as shown here:

```
user> (let [x 10]
          (to-list 20))
(20)
user> (let [x 10]
          (to-list-with-capture 20))
(10)
```

The `to-list-with-capture` function seems to dynamically obtain the value of `x` from the surrounding scope, and not from the parameter passed to it. As you may have guessed, this can lead to a number of subtle and bizarre bugs. In Clojure, the solution to this problem is simple; a syntax quoted form will resolve all free symbols to namespace-qualified names. This can be verified by macroexpanding the expression that uses the `to-list` function in the preceding example.

Let's say we would like to use a temporary variable using a `let` form with a macro that performs the same task as the `to-list` macro from *Example*

4.2. This may seem rather unnecessary, but it is only being done to demonstrate how symbols are resolved by a syntax quote. Such a macro can be implemented as shown in *Example 4.3*:

```
(defmacro to-list-with-error [x]
  `(let [y ~x]
    (list y)))
```

Calling the `to-list-with-error` macro will result in an error due to the use of the free symbol `y`, as shown here:

```
user> (to-list-with-error 10)
CompilerException java.lang.RuntimeException:
Can't let qualified name: user/y
```

This error can be quite annoying, as we simply intended to use a temporary variable in the body of the `to-list-with-error` macro. This error occurred because it is not clear where the variable `y` is resolved from. To get around this error, we can declare the variable `y` as an auto-gensym variable, as shown in *Example 4.4*:

```
(defmacro to-list-with-gensym [x]
  `(let [y# ~x]
    (list y#)))
```

*Example 4.4: A macro that uses a let form and an auto-gensym variable*

The `to-list-with-gensym` macro works as expected without any error, as shown here:

```
user> (to-list-with-gensym 10)
(10)
```

We can also inspect the expression generated by the `to-list-with-gensym` macro using the `macroexpand` and `macroexpand-1` forms, and the reader is encouraged to try this in the REPL.

To summarize, macros defined using the `defmacro` form can be used to rewrite and transform code. Syntax quote and auto-gensym variables can be used to write hygenic macros that avoid certain problems that can arise due the use of dynamic scope.

## Note

Syntax quote can actually be implemented as a user defined macro. Libraries such as `syntax-quote` (<https://github.com/hiredman/syntax-quote>) and `backtick` (<https://github.com/brandongbloom/backtick>) demonstrate how syntax quote can be implemented through macros.

## Encapsulating patterns in macros

In Clojure, macros can be used to rewrite expressions in terms of functions and special forms. However, in languages such as Java and C#, there is a lot of additional syntax added to the language for handling special forms. For example, consider the `if` construct in these languages, which is used to check whether an expression is true or not. This construct does have some special syntax. If a recurring pattern of usage of the `if` construct is found in a program written in these languages, there is no obvious way to automate this pattern. Languages such as Java and C# have the concept of *design patterns* that encapsulate these sort of patterns. But without the ability to rewrite expressions, encapsulating patterns in these languages can get a bit incomplete and cumbersome. The more special forms and syntax we add to a language, the harder it gets to programmatically generate code for the language. On the other hand, macros in Clojure and other Lisps can easily rewrite expressions to automate recurring patterns in code. Also, there is more-or-less no special syntax for code in Lisps, as code and data are one and the same. In a way, macros in Lispy languages allow us to concisely encapsulate design patterns by extending the language with our own hand-made constructs.

Let's explore a few examples that demonstrate how macros can be used to encapsulate patterns. The `->` and `->>` threading macros in Clojure are used

to compose several functions together by passing in an initial value. In other words, the initial value is *threaded* through the various forms that are passed as arguments to the `->` and `->>` macros. These macros are defined in the `clojure.core` namespace as part of the Clojure language, as shown in *Example 4.5*.

## Note

The following examples can be found in `src/m_clj/c4/threading.clj` of the book's source code.

```
(defmacro -> [x & forms]
  (loop [x x
         forms forms]
    (if forms
        (let [form (first forms)
              threaded (if (seq? form)
                         (with-meta
                           `(~(first form) ~x ~@(next form))
                           (meta form))
                         (list form x)))]
          (recur threaded (next forms)))
        x)))

(defmacro ->> [x & forms]
  (loop [x x
         forms forms]
    (if forms
        (let [form (first forms)
              threaded (if (seq? form)
                         (with-meta
                           `(~(first form) ~@ (next form) ~x)
                           (meta form))
                         (list form x)))]
          (recur threaded (next forms)))
        x)))
```

*Example 4.5: The `->` and `->>` threading macros*

The `->` and `->>` macros in *Example 4.5* use a `loop` form to recursively

thread a value  $x$  through the expressions represented by `forms`. The first symbol in a form, that is the function being called, is determined using the `first` function. The arguments to be passed in this function, other than  $x$ , are extracted using the `next` function. If a form is just a function name without any additional arguments, we create a new form using the expression `(list form x)`. The `with-meta` form is used to preserve any metadata specified with `form`. The `->` macro passes  $x$  as the first argument, whereas `->>` passes  $x$  as the last argument. This is done in a recursive manner for all the forms passed to these macros. Interestingly, syntax quoted forms are used sparingly by both of the `->` and `->>` macros. We can actually refactor out some parts of these macros into functions. This adds a slight advantage as functions can be tested quite easily compared to macros. The `->` and `->>` threading macros can be refactored as shown in *Example 4.6* and *Example 4.7*:

```
(defn thread-form [first? x form]
  (if (seq? form)
    (let [[f & xs] form
          xs (conj (if first? xs (vec xs)) x)]
      (apply list f xs))
    (list form x)))

(defn threading [first? x forms]
  (reduce #(thread-form first? %1 %2)
         x forms))
```

### *Example 4.6: Refactoring the `->` and `->>` threading macros*

The `thread-form` function in *Example 4.6* positions the value  $x$  in the expression form using the `conj` function. The premise here is that the `conj` function will add an element in the head of a list and at the end or tail of a vector. The `first?` argument is used to indicate whether the value  $x$  has to be passed as the first argument to `form`. The `threading` function simply applies the `thread-form` function to all the expressions passed to it, represented by `forms`. The macros `->` and `->>` can now be implemented using the `threading` function as shown in *Example 4.7*:

```
(defmacro -> [x & forms]
  (threading true x forms))

(defmacro ->> [x & forms]
  (threading false x forms))
```

*Example 4.7: Refactoring the -> and ->> threading macros  
(continued)*

The threading macros defined in *Example 4.7* work just as well as the ones in *Example 4.5*, and we can verify this in the REPL. This is left as an exercise for the reader.

A common pattern of usage of the `let` form is to repeatedly rebind a variable to new values by passing it through several functions. This kind of pattern can be encapsulated using the `as->` threading macro, which is defined as shown in *Example 4.8*.

```
(defmacro as-> [expr name & forms]
  `(let [~name ~expr
        ~@(interleave (repeat name) forms)]
    ~name))
```

*Example 4.8: Refactoring the -> and ->> threading macros*

Let's skip past explaining the details of the `as->` macro through words and simply describe the code it generates using the `macroexpand` function, as shown here:

```
user> (macroexpand '(as-> 1 x (+ 1 x) (+ x 1)))
(let* [x 1
       x (+ 1 x)
       x (+ x 1)]
  x)
user> (as-> 1 x (+ 1 x) (+ x 1))
3
```

The `as->` macro binds its first argument to a symbol represented by its second argument and generates a `let*` form as a result. This allows us to

define expressions that have to be threaded over in terms of an explicit symbol. One might even say it's a more flexible way to perform the threading of a value through several expressions, as compared to using the `->` and `->>` macros.

## Note

The `as->` form has been introduced in Clojure 1.5 along with several other threading macros.

Thus, macros are great tools in automating or encapsulating patterns in code. Several commonly used forms in the Clojure language are actually defined as macros, and we can just as easily define our own macros.

## Using reader conditionals

It is often necessary to interoperate with native objects in Clojure and its dialects such as ClojureScript. We can define platform-specific code using *reader conditionals*. Let's now briefly take a look at how we can use reader conditionals.

## Note

Reader conditionals have been introduced in Clojure 1.7. Prior to version 1.7, platform-specific Clojure/ClojureScript code had to be managed using the `cljs` library (<https://github.com/lynaghk/cljx>).

The *reader conditional form*, written as `#?( ... )`, allows us to define platform-specific code using the `:cljs`, `:clj`, `:clr`, and `:default` keywords. The *reader conditional splicing form*, written as `#?@( ... )`, has semantics similar to a reader conditional form. It can be used to splice a list of platform-specific values or expressions into a form. Both these conditional forms are processed when code is read, instead of when it is macroexpanded.

Since Clojure 1.7, the `read-string` function has a second arity in which we

can specify a map as an argument. This map can have two keys, `:read-cond` and `:features`. When a string containing a conditional form is passed to the `read-string` function, platform-specific code can be generated by specifying the platform as a set of keywords, represented by `:cljs`, `:clj`, or `:clr`, with the `:features` key in the map of options. In this case, the keyword `:allow` must be specified for the key `:read-cond` in the map passed to the `read-string` function, or else an exception will be thrown. We can try out the reader conditional form with the `read-string` function in the REPL as shown here:

```
user> (read-string {:read-cond :allow :features #{:clj}}
                     "#?(:cljs \"ClojureScript\" :clj
                     \"Clojure\")")
"Clojure"
user> (read-string {:read-cond :allow :features #{:cljs}}
                     "#?(:cljs \"ClojureScript\" :clj
                     \"Clojure\")")
"ClojureScript"
```

## Note

These examples can be found in

`src/m_clj/c4/reader_conditionals.cljc` of the book's source code.

Similarly, we can read a conditional splicing form into an expression with the `read-string` function as shown here:

```
user> (read-string {:read-cond :allow :features #{:clr}}
                     "[1 2 #?@(:cljs [3 4] :default [5 6]))")
[1 2 5 6]
user> (read-string {:read-cond :allow :features #{:clj}}
                     "[1 2 #?@(:cljs [3 4] :default [5 6]))")
[1 2 5 6]
user> (read-string {:read-cond :allow :features #{:cljs}}
                     "[1 2 #?@(:cljs [3 4] :default [5 6]))")
[1 2 3 4]
```

We can also prevent the transformation of conditional forms by specifying the `:preserve` keyword with the `:read-cond` key in the optional map

passed to the `read-string` function, as shown here:

```
user> (read-string {:read-cond :preserve}
                     "[1 2 #?@(:cljs [3 4] :clj [5 6]))]")
[1 2 #?@(:cljs [3 4] :clj [5 6]))]
```

However, wrapping conditional forms in a string is not really something we should be doing in practice. Generally, we should write all platform-specific code as reader conditional forms in source files with the `.cljc` extension. Once the top-level forms defined in the `.cljc` file are processed by the Clojure reader, we can use them just like any other reader forms. For example, consider the macro written using a reader conditional form in *Example 4.9*:

```
(defmacro get-milliseconds-since-epoch []
  `(.getTime #?(:cljs (js/Date.)
                      :clj (java.util.Date.))))
```

*Example 4.9: A macro using a reader conditional*

The `get-milliseconds-since-epoch` macro in *Example 4.9* calls the `.getTime` method on a new `java.util.Date` instance when called from the Clojure code. Also, this macro calls the `.getTime` method on a new JavaScript `Date` object when used in ClojureScript code. We can macroexpand a call to the `get-milliseconds-since-epoch` macro from the Clojure REPL to generate JVM-specific code, as shown here:

```
user> (macroexpand '(get-milliseconds-since-epoch))
(. (java.util.Date.) getTime)
```

Thus, reader conditionals help in encapsulating platform-specific code to be used in code that is agnostic of the underlying platform.

## Avoiding macros

Macros are an extremely flexible way of defining our own constructs in Clojure. However, careless use of macros in a program can become

complicated and lead to a number of strange bugs that are hidden from plain sight. As described in the book, *Programming Clojure* by *Stuart Halloway* and *Aaron Bedra*, the usage of macros in Clojure has two thumb rules:

- **Don't write macros:** Anytime we try to use a macro, we must think twice whether we could perform the same task using a function.
- **Write macros if it's the only way to encapsulate a pattern:** A macro must be used only if it is easier or more convenient than calling a function.

What's the problem with macros? Well, macros complicate a program's code in several ways:

- Macros cannot be composed like functions as they are not really values. It's not possible to pass a macro as an argument to the `map` or `apply` forms, for example.
- Macros cannot be tested as easily as functions. Though it can be done programmatically, the only way to test macros is by using macroexpansion functions and quoting.
- In some cases, code that calls a macro may have been written as a macro itself, thus adding more complexity to our code.
- Hidden bugs caused by problems such as symbol capture make macros a little tricky. Debugging macros isn't really easy either, especially in a large codebase.

For these reasons, macros have to be used carefully and responsibly. In fact, if we can solve a problem using macros as well functions, we should always prefer the solution that uses functions. If the use of a macro is indeed required, we should always strive to refactor out as much code as possible from a macro into a function.

That aside, macros make programming a lot of fun as they allow us to define our own constructs. They allow a degree of freedom and liberty that is not really possible in other languages. You may hear a lot of seasoned

Clojure programmers tell you that macros are evil and you should never use them, but don't let that stop you from exploring what is possible with macros. Once you encounter and tackle some of the problems that arise with the use of macros, you will have enough experience to be able to decide when macros can be used appropriately.

# Summary

We have explored how metaprogramming is possible with Clojure in this chapter. We discussed how code is read, macroexpanded, and evaluated, as well as the various primitive constructs that implement these operations. Macros can be used to encapsulate patterns in code, as we demonstrated in the various examples in this chapter. Toward the end of the chapter, we also talked about reader conditionals and pointed out the various complications that arise with the use of macros.

In the following chapter, we will explore how transducers can be used to process any data regardless of the source of the data.

# Chapter 5. Composing Transducers

Let's get back to our journey of performing computations over data in Clojure. We've already discussed how *reducers* can be used to process collections in [Chapter 3, Parallelization Using Reducers](#). Transducers are, in fact, a generalization of reducers that are independent of the source of data. Also, reducers are more about parallelization, while transducers are more focused on generalizing data transformations without restricting us to any particular source of data. Transducers capture the essence of the standard functions that operate on sequences, such as `map` and `filter`, for several sources of data. They allow us to define and compose transformations of data regardless of how the data is supplied to us.

Incidentally, in the context of physics, a transducer is a device that converts a signal from one form of energy into another form. In a way, Clojure transducers can be thought of as ways to capture the *energy* in functions, such as `map` and `filter`, and convert between different sources of data. These sources include collections, streams, and asynchronous channels. Transducers can also be extended to other sources of data. In this chapter, we will focus on how transducers can be used for sequences and collections, and will reserve discussing transducers with asynchronous channels until we talk about the `core.async` library in [Chapter 8, Leveraging Asynchronous Tasks](#). Later in this chapter, we will study how transducers are implemented in Clojure.

## Understanding transducers

Transducers are essentially a stack of transformations that can be composed and applied to *any* representation of data. They allow us to define transformations that are agnostic of implementation-specific details about the source of the supplied data. Transducers also have a significant

performance benefit. This is attributed to the avoidance of unnecessary memory allocations for arbitrary containers, such as sequences or other collections, to store intermediate results between transformations.

## Note

Transducers have been introduced in Clojure 1.7.

Transformations can be composed without the use of transducers as well. This can be done using the `comp` and `partial` forms. We can pass any number of transformations to the `comp` function, and the transformation returned by the `comp` function will be a composition of the supplied transformations in the right-to-left order. In Clojure, a transformation is conventionally denoted as `xf` or `xform`.

## Note

The following examples can be found in `src/m_clj/c5/transduce.clj` of the book's source code.

For example, the expression `(comp f g)` will return a function that applies the function `g` to its input and then applies the function `f` to the result. The `partial` function will bind a function to any number of arguments and return a new function. The `comp` function can be used with a `partial` form to compose the `map` and `filter` functions, as shown here:

```
user> (def xf-using-partial (comp
                                (partial filter even?)
                                (partial map inc)))
#'user/xf-using-partial
user> (xf-using-partial (vec (range 10)))
(2 4 6 8 10)
```

In the preceding output, the `partial` function is used to bind the `inc` and `even?` functions to the `map` and `filter` functions respectively. The functions returned by both the `partial` forms shown above will expect a collection to be passed to them. Thus, they represent transformations that

can be applied to a given collection. These two transformations are then composed with the `comp` function to create a new function `xf-using-partial`. This function is then applied to a vector of numbers to return a sequence of even numbers. There are a few issues with this code:

- The filtering of even numbers using the `even?` function is performed after applying the `inc` function. This proves that the transformations passed to the `comp` function are applied in the right-to-left order, which is the reverse of the order in which they are specified. This can be a little inconvenient at times.
- The value returned by the `xf-using-partial` function is a list and not a vector. This is because the `map` and `filter` function both return lazy sequences, which are ultimately converted into lists. Thus, the use of the `vec` function has no effect on the type of collection returned by the `xf-using-partial` function.
- Also, the transformation `(partial map inc)` applied by the `xf-using-partial` function will create a new sequence. This resulting sequence is then passed to the transformation `(partial filter even?)`. The intermediate use of a sequence is both unnecessary and wasteful in terms of memory if we have several transformations that must be composed.

This brings us to transducers, which address the preceding problems related to composing transformations using the `comp` and `partial` forms. In the formal sense, a transducer is a function that modifies a *step function*. This step function is analogous to a reducing function in the context of reducers. A step function combines an input value with the accumulated result of a given computation. A transducer accepts a step function as an argument and produces a modified version of it. In fact, the `xf` and `xform` notations are also used to represent a transducer; because a transducer is also a transformation, it transforms a step function. While it may be hard to illustrate without any code, this modification of a step function performed by a transducer actually depicts how some input data is consumed by a given computation to produce a result. Several transducers can also be composed together. In this way, transducers can be thought of

as a unified model to process data.

Several of the standard Clojure functions return a transducer when they are called with a single argument. These functions either:

- Accept a function along with a collection as arguments. Examples of such functions are `map`, `filter`, `mapcat`, and `partition-by`.
- Accept a value indicating the number of elements, usually specified as `n`, along with a collection. This category includes functions such as `take`, `drop`, and `partition-all`.

## Note

Visit <http://clojure.org/transducers> for the complete list of standard functions that implement transducers.

The use of transducers can be aptly depicted by Rich Hickey's baggage loading example. Suppose we intend to load several bags into an airplane. The bags will be supplied in pallets, which can be thought of as collections of bags. There are several steps that have to be performed to load the bags into the airplane. Firstly, the bags must be unbundled from the supplied pallets. Next, we must check whether a bag contains any food, and not process it any further if it does. Finally, all the bags must be weighed and labeled in case they are heavy. Note that these steps needed to load the bags into the airplane do not specify how the pallets are supplied to us, or how the labeled bags from the final step are transported to the plane.

We can model the process of loading the bags into the plane as shown in the `process-bags` function in *Example 5.1*, as follows:

```
(declare unbundle-pallet)
(declare non-food?)
(declare label-heavy)

(def process-bags
  (comp
    (partial map label-heavy))
```

```
(partial filter non-food?)  
(partial mapcat unbundle-pallet)))
```

### *Example 5.1: Loading bags into an airplane*

The functions `unbundle-pallet`, `non-food?`, and `label-heavy` in *Example 5.1* represent the three steps of loading bags into an airplane. These functions are applied to a collection of bags using the `map`, `filter`, and `mapcat` functions. Also, they can be composed using the `comp` and `partial` functions in a right-to-left order. As we described earlier, the `map`, `filter`, and `mapcat` functions will all produce sequences on being called, hence creating intermediate collections of bags between the three transformations. This intermediate use of sequences is analogous to putting all the bags on trollies after the step is performed. The supplied input and the final result would both be a bag of trollies. The use of trollies not only incurs additional work between the steps of our process, but the steps are now convoluted with the use of trollies. Thus, we would have to redefine these steps if we had to use, say, conveyer belts instead of trollies to transport the baggage. This means that the `map`, `filter`, and `mapcat` functions would have to be implemented again if we intend to produce a different type of collection as a final result. Alternatively, we can use transducers to implement the `process-bags` function without specifying the type of collection of either the input or the result, as shown in *Example 5.2*:

```
(def process-bags  
  (comp  
    (mapcat unbundle-pallet)  
    (filter non-food?)  
    (map label-heavy) ))
```

### *Example 5.2: Loading bags into an airplane using transducers*

The `process-bags` function in *Example 5.2* shows how transducers can be used to compose the `unbundle-pallet`, `non-food?`, and `label-heavy` functions in a left-to-right order. Each of the expressions passed to the

`comp` function in *Example 5.2* return a transducer. This implementation of the `process-bags` function does not create any intermediary collections when it is executed.

## Producing results from transducers

Transducers are only recipes for computations, and are not capable of performing any actual work on their own. A transducer can produce results when coupled with a source of data. There's also another vital piece of the puzzle, that is, the `step` function. To combine a transducer, a `step` function, and a source of data, we must use the `transduce` function.

The `step` function passed to `transduce` is also used to create the initial value of the result to be produced. This initial value of the result can also be specified as an argument to the `transduce` function. For example, the `transduce` function can be used with the `conj` form is shown as follows:

```
user> (def xf (map inc))
#'user/xf
user> (transduce xf conj [0 1 2])
[1 2 3]
user> (transduce xf conj () [0 1 2])
(3 2 1)
```

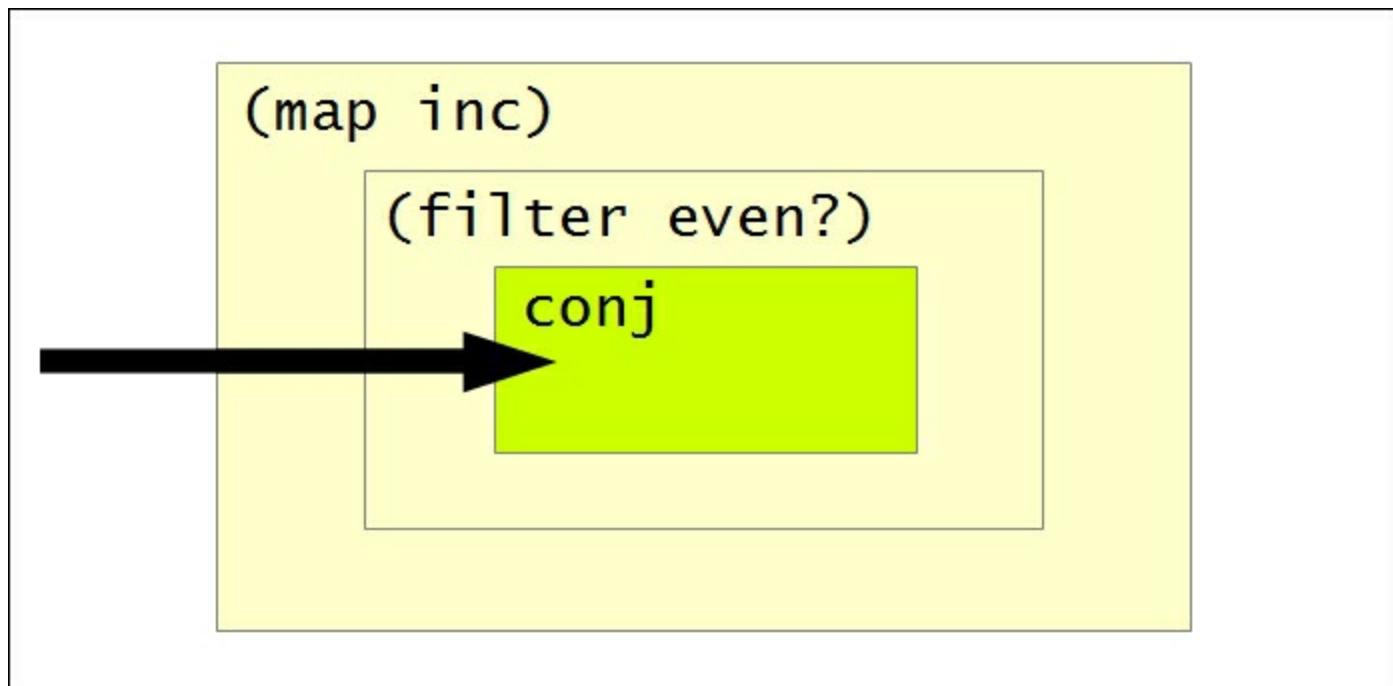
The `inc` function is coupled with the `map` function to create a transducer `xf`, as shown previously. The `transduce` function can be used to produce either a list or a vector from the transducer `xf` using the `conj` function. The order of elements in the results of both the `transduce` forms shown previously is different due to the fact that the `conj` function will add an element to the head of a list, as opposed to adding it at the end of a vector.

We can also compose several transducers together using the `comp` function, as shown here:

```
user> (def xf (comp
                  (map inc)
                  (filter even?)))
```

```
#'user/xf
user> (transduce xf conj (range 10))
[2 4 6 8 10]
```

The transducer `xf` in the preceding output encapsulates the application of the `inc` and `even?` functions using the `map` and `filter` forms respectively. This transducer will produce a vector of even numbers when used with the `transduce` and `conj` forms. Note that the `inc` function is indeed applied to the supplied collection (`range 10`), or else the value `10` would not show up in the final result. This computation using the transducer `xf` can be illustrated as follows:



The preceding diagram depicts how the transformations `(map inc)`, `(filter even?)`, and `conj` are composed in the transformation `xf`. The `map` form is applied first, followed by the `filter` form, and finally the `conj` form. In this manner, transducers can be used to compose a series of transformations for any source of data.

Another way to produce a collection from a transducer is by using the `into` function. The result of this function depends on the initial collection

supplied to it as the first argument, as shown here:

```
user> (into [] xf (range 10))
[2 4 6 8 10]
user> (into () xf (range 10))
(10 8 6 4 2)
```

The standard `sequence` function can also be used to produce a lazy sequence from a transducer. Of course, the returned lazy sequence will be converted to a list in the REPL, as shown here:

```
user> (sequence xf (range 10))
(2 4 6 8 10)
```

So far, we've composed transducers to produce collections with a finite number of elements. An infinite series of values could also be produced by a transducer when used with the `sequence` function. The `eduction` function can be used to represent this sort of computation. This function will transform a collection, specified as its last argument, to any number of transformations passed to it in right-to-left order. An `eduction` form may also require a fewer number of allocations compared to using a `sequence`.

For example, if we were to retrieve the 100th element in a sequence using the `nth` function, the first 99 elements would have to be realized and also discarded later as they are not needed. On the other hand, an `eduction` form can avoid this overhead. Consider the declaration of `simple-eduction` in *Example 5.3*:

```
(def simple-eduction (eduction (map inc)
                                (filter even?)
                                (range)))
```

### *Example 5.3: Using the eduction function*

The collection `simple-eduction` shown in *Example 5.3* will first filter out even values using the `even?` predicate from the infinite range (`range`) and then increment the resulting values using the `inc` function. We can retrieve

elements from the collection `simple-eduction` using the `nth` function. The same computation can also be modeled using lazy sequences, but transducers perform much better, as shown here:

```
user> (time (nth simple-eduction 100000))
"Elapsed time: 65.904434 msecs"
200001
user> (time (nth (map inc (filter even? (range))) 100000))
"Elapsed time: 159.039363 msecs"
200001
```

The `eduction` form using a transducer performs twice as fast compared to a sequence! From the output shown previously, it is quite clear that transducers perform significantly better than lazy sequences for composing a number of transformations. In summary, transducers created using functions such as `map` and `filter` can be easily composed to produce collections using functions such as `transduce`, `into`, and `eduction`. We can also use transducers with other sources of data such as streams, asynchronous channels, and observables.

# Comparing transducers and reducers

Both transducers and reducers, which were discussed in [Chapter 3, Parallelization Using Reducers](#), are ways to improve the performance of computations performed over collections. While transducers are a generalization of data processing for multiple data sources, there are a few other subtle differences between transducers and reducers, which are described as follows:

- Transducers are implemented as part of the Clojure language in the `clojure.core` namespace. However, reducers must be explicitly included in a program, as they are implemented in the `clojure.core.reducers` namespace.
- Transducers only create a collection when producing the final result of a series of transformations. There are no intermediary collections required to store the results of a transformation that constitutes a transducer. On the other hand, reducers produce intermediate collections to store results, and only avoid the creation of unnecessary empty collections.
- Transducers deal with efficient composition of a series of transformations. This is quite orthogonal to how reducers squeeze out performance from a computation performed over a collection through the use of parallelization. Transducers perform significantly better than both the `reduce` functions from the `clojure.core` and `clojure.core.reducers` namespaces. Of course, using the `clojure.core.reducers/fold` function is still a good way to implement a computation that can be parallelized.

These contrasts between transducers and reducers describe how these two methodologies of processing data are different. In practice, the performance of these techniques depends on the actual computation being implemented. Generally, if we intend to implement an algorithm to process

data in a performant way, we should use transducers. On the other hand, if we are dealing with a lot of data in the memory with no need for I/O and laziness, we should use reducers. The reader is encouraged to compare the performance of the `transduce` function with that of the `reduce` and `fold` functions of the `clojure.core.reducers` library for different computations and data sources.

# Transducers in action

In this section, we will examine how transducers are implemented. We will also get a basic idea of how our own *transducible* source of data can be implemented.

## Managing volatile references

Some transducers can internally use state. It turns out that the existing reference types, such as atoms and refs, aren't fast enough for the implementation of transducers. To circumvent this problem, transducers also introduce a new *volatile* reference type. A volatile reference represents a mutable variable that will not be copied into the thread-local cache. Also, volatile references are not atomic. They are implemented in Java using the `volatile` keyword with a `java.lang.Object` type.

### Note

The following examples can be found in `src/m_clj/c5/volatile.clj` of the book's source code.

We can create a new volatile reference using the `volatile!` function. The value contained in the volatile state can then be retrieved using the `@` reader macro or a `deref` form. The `vreset!` function can be used to set the state of a volatile reference, as shown here:

```
user> (def v (volatile! 0))
#'user/v
user> @v
0
user> (vreset! v 1)
1
```

In the preceding output, we encapsulate the value `0` in a volatile state, and then set its state to `1` using the `vreset!` function. We can also mutate the state contained in a volatile reference using the `vswap!` function. We will

have to pass a volatile reference and a function to be applied to the value contained in the reference to this function. We can also specify any other arguments for the supplied function as additional arguments to the `vswap!` function. The `vswap!` function can be used to change the state of the volatile reference `v` that we previously defined, as shown here:

```
user> (vswap! v inc)
2
user> (vswap! v + 3)
5
```

The first call to the `vswap!` function in the preceding output uses the `inc` function to increment the value stored in the reference `v`. Similarly, the subsequent call to the `vswap!` function adds the value `3` to the new value in the volatile reference `v`, thus producing the final value `5`.

## Note

We can check whether a value is a volatile using the `volatile?` predicate.

One may argue that the volatile reference type has the same semantics as that of an atom. The `vreset!` and `vswap!` functions have the exact same shape as the `reset!` and `swap!` functions that are used with atoms. However, there is an important difference between a volatile reference and an atom. Unlike an atom, a volatile reference does not guarantee atomicity of operations performed on it. Hence, it's recommended to use volatile references in a single thread.

## Creating transducers

As a transducer modifies a supplied step function, let's first define what a step function actually does. The following aspects need to be considered:

- The step function must be able to provide an initial value to the transformation it models. In other words, the step function must have an *identity* form that takes no arguments.
- Inputs must be combined with the result accumulated so far by the

computation. This is analogous to how a reducing function combines an input value with an accumulated result to produce a new result. The arity of this form is also the same as that of a reducing function; it requires two arguments to represent the current input and the accumulated result.

- The step function must also be able to complete the computation of the modeled process to return something. This can be implemented as a function that accepts a single argument that represents the accumulated result.

Thus, a step function is represented as a function with three arities, as described previously. *Early termination* may also be needed by some transducers to abruptly stop a computational process based on certain conditions.

Now, let's look at how some of the standard functions in the `clojure.core` namespace are implemented with transducers. The `map` function returns a transducer when called with a single argument.

## Note

The following examples can be found in `src/m_clj/c5/implenting_transducers.clj` of the book's source code.

The following *Example 5.4* describes how the `map` function is implemented:

```
(defn map
  ([f]
   (fn [step]
     (fn
       ([] (step))
       ([result] (step result)))
       ([result input]
         (step result (f input))))))
  ([f coll]
   (sequence (map f) coll)))
```

### *Example 5.4: The map function*

The 1-arity form of the `map` function returns a function that accepts a step function, represented by `step`, and returns another step function. The returned step function has three different arities, just like we described earlier in this section. The essence of the `map` function is described by the expression `(step result (f input))`, which translates to "apply the function `f` on the current input `input` and combine it with the accumulated result `result` using the function `step`". The returned step function also has two other arities—one that takes no arguments and another that takes one argument. These arities correspond to the other two cases of a step function that we described earlier.

The second arity of the `map` function, which returns a collection and not a transducer, is merely a composition of the `sequence` function and the transducer returned by the expression `(map f)`. The actual creation of a collection is done by the `sequence` function. The 1-arity form of the `map` function only describes how the function `f` is applied over a transducible context such as a collection.

Similarly, the `filter` function can be implemented using a transducer, as shown in *Example 5.5*, as follows:

```
(defn filter
  ([p?]
   (fn [step]
     (fn
       ([] (step))
       ([result] (step result)))
       ([result input]
        (if (p? input)
            (step result input)
            result)))))

  ([p? coll]
   (sequence (filter p?) coll)))
```

### *Example 5.5: The filter function*

The premise in the implementation of the `filter` function is that a predicate `p?` is used to conditionally combine the accumulated result and the current input, which are represented by `result` and `input` respectively. If the expression `(p? input)` does not return a truthy value, the accumulated result is returned without any modification. Similar to the `map` function in *Example 5.4*, the 2-arity form of the `filter` function is implemented using a sequence form and a transducer.

To handle early termination in transducers, we must use the `reduced` and `reduced?` functions. Calling `reduce` or a step function on a value that has been wrapped in a `reduced` form will simply return the contained value. The `reduced?` function checks whether a value is already *reduced*, that is, wrapped in a `reduced` form. The `reduced` and `reduced?` forms both accept a single argument, as shown here:

```
user> (def r (reduced 0))
#'user/r
user> (reduced? r)
true
```

Consider the following function `rf` in *Example 5.6* that uses a `reduced` form to ensure that the accumulated result is never more than 100 elements:

```
(defn rf [result input]
  (if (< result 100)
    (+ result input)
    (reduced :too-big)))
```

### *Example 5.6: Using the reduced function*

The function `rf` merely sums up all inputs to produce a result. If the `rf` function is passed to the `reduce` function along with a sufficiently large collection, the `:too-big` value is returned as the final result, as shown here:

```
user> (reduce rf (range 3))
3
user> (reduce rf (range 100))
```

:too-big

A value wrapped in a `reduced` form can be extracted using the `unreduced` function or the `@` reader macro. Also, the `ensure-reduced` function can be used instead of `reduced` to avoid re-applying a `reduced` form to a value that has already been reduced.

The standard `take-while` function can be implemented using a `reduced` form and a transducer, as shown in the following *Example 5.7*:

```
(defn take-while [p?]
  (fn [step]
    (fn
      ([] (step))
      ([result] (step result)))
      ([result input]
        (if (p? input)
            (step result input)
            (reduced result))))))
```

### *Example 5.7: The take-while function*

Note that only the 1-arity form of the `take-while` function is described in *Example 5.7*. The step function returned by the `take-while` function uses the expression `(p? input)` to check if the accumulated result has to be combined with the current input. If the `p?` predicate does not return a truthy value, the accumulated result is returned by wrapping it in a `reduced` form. This prevents any other transformations, which may be composed with the transformation returned by the `take-while` function, from modifying the accumulated result any further. In this way, the `reduced` form can be used to wrap the result of a transducer and perform early termination based on some conditional logic.

Let's look at how a stateful transducer is implemented. The `take` function returns a transducer that maintains an internal state. This state is used to keep a track of the number of items that have been processed so far, since the `take` function must only return a certain number of items from a

collection or any other transducible context by definition. *Example 5.8* describes how the `take` function is implemented using a volatile reference to maintain state:

```
(defn take [n]
  (fn [step]
    (let [nv (volatile! n)]
      (fn
        ([] (step))
        ([result] (step result))
        ([result input]
          (let [n @nv
                nn (vswap! nv dec)
                result (if (pos? n)
                          (step result input)
                          result)]
            (if (not (pos? nn))
                (ensure-reduced result)
                result)))))))
```

### *Example 5.8: The take function*

The transducer returned by the `take` function will first create a volatile reference `nv` from the supplied value `n` to track the number of items to be processed. The returned `step` function then decrements the volatile reference `nv` and combines the result with the input using the `step` function. This is done repeatedly until the value contained in the reference `nv` is positive. Once all `n` items have been processed, the result is wrapped in an `ensure-reduced` form to signal early termination. Here, the `ensure-reduced` function is used to prevent wrapping the value `result` in another reduced form, since the expression `(step result input)` could return a value that is already reduced.

Finally, let's take a quick look at how the `transduce` function is implemented, as shown in *Example 5.9*:

```
(defn transduce
  ([xform f coll] (transduce xform f (f) coll))
  ([xform f init coll]
```

```

(let [xf (xform f)
      ret (if (instance? clojure.lang.IReduceInit coll)
             (.reduce ^clojure.lang.IReduceInit coll xf init)
             (clojure.core.protocols/coll-reduce coll xf
               init)))]
  (xf ret)))

```

### *Example 5.9: The transduce function*

The `transduce` function has two arities. The 4-arity form of the `transduce` function calls the `.reduce` method of the transducible context `coll` if it is an instance of the `clojure.lang.IReduceInit` interface. This interface defines a single method `reduce` that represents how a data source is reduced using a given function and an initial value. If the variable `coll` does not implement this interface, the `transduce` function will fall back on the `coll-reduce` function to process the data source represented by `coll`. In a nutshell, the `transduce` function will try to process a transducible context in the fastest possible way. The `clojure.lang.IReduceInit` interface must be implemented by all data sources that must support the use of `transduce`.

The 3-arity form of the `transduce` function produces the initial value for the transduction by invoking the supplied function `f` without any arguments. Thus, this arity of the `transduce` function can only be used with functions that provide an identity value.

### **Note**

The definitions of the `map`, `filter`, `take`, and `take-while` functions, as shown in this section, are simplified versions of their actual definitions. However, the `transduce` function is shown exactly as it is implemented in the `clojure.core` namespace.

This depicts how transducers and the `transduce` function are implemented. If we need to implement our own transducible source of data, the implementations described in this section can be used as a guideline.

# Summary

So far, we have seen how we can process data using sequences, reducers, and transducers. In this chapter, we described how transducers can be used for performant computations. We also briefly studied how transducers are implemented in the Clojure language.

In the following chapter, we will explore algebraic data structures, such as functors, applicatives, and monads, in Clojure. These concepts will deepen our understanding of functional composition, which is the keystone of functional programming.

# Chapter 6. Exploring Category Theory

On a journey into functional programming, a programmer will eventually stumble upon *category theory*. First off, let's just say that the study of category theory is not really needed to write better code. It's more prevalent in the internals of pure functional programming languages, such as Haskell and Idris, in which functions are *pure* and more like mathematical functions that do not have implicit side effects such as I/O and mutation. However, category theory helps us reason about a very fundamental and practical aspect of computation: *composition*. Functions in Clojure, unlike in pure functional programming languages, are quite different from mathematical functions as they can perform I/O and other side effects. Of course, they can be pure under certain circumstances, and thus concepts from category theory are still useful in Clojure for writing reusable and composable code based on pure functions.

Category theory can be thought of as a mathematical framework for modeling composition. In this chapter, we will discuss a few concepts from category theory using Clojure. We will also study a few algebraic types, such as functors, monoids, and monads.

## Demystifying category theory

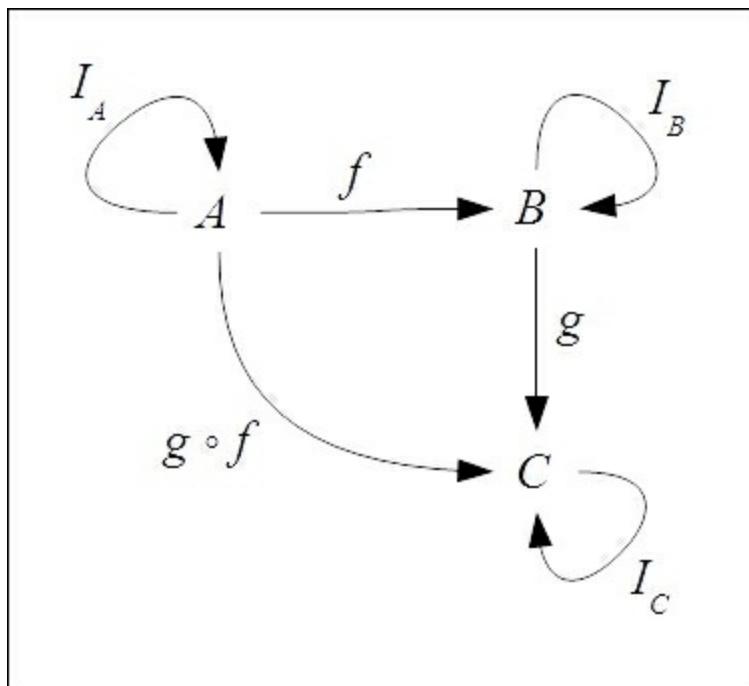
Category theory has its own share of quirky notations and conventions. Let's start off by exploring some of the terminology used in category theory, in a language understandable by us mortal programmers.

A *category* is formally defined as a collection of **objects** and **morphisms**. In simple terms, objects represent abstract types, and morphisms represent functions that convert between these types. A category is thus analogous to a programming language that has a few types and functions, and has two

basic properties:

- There exists an *identity morphism* for each object in the category. In practice, a single identity function can be used to represent the identity morphism for all given objects, but this is not mandatory.
- Morphisms in a category can be composed together into a new morphism. In fact, a composition of two or more morphisms is an optimization of applying the individual morphisms one at a time. In this way, the composition of several morphisms is said to *commute* with applying the constituting morphisms.

Morphisms in a category can be composed as illustrated by the following diagram:



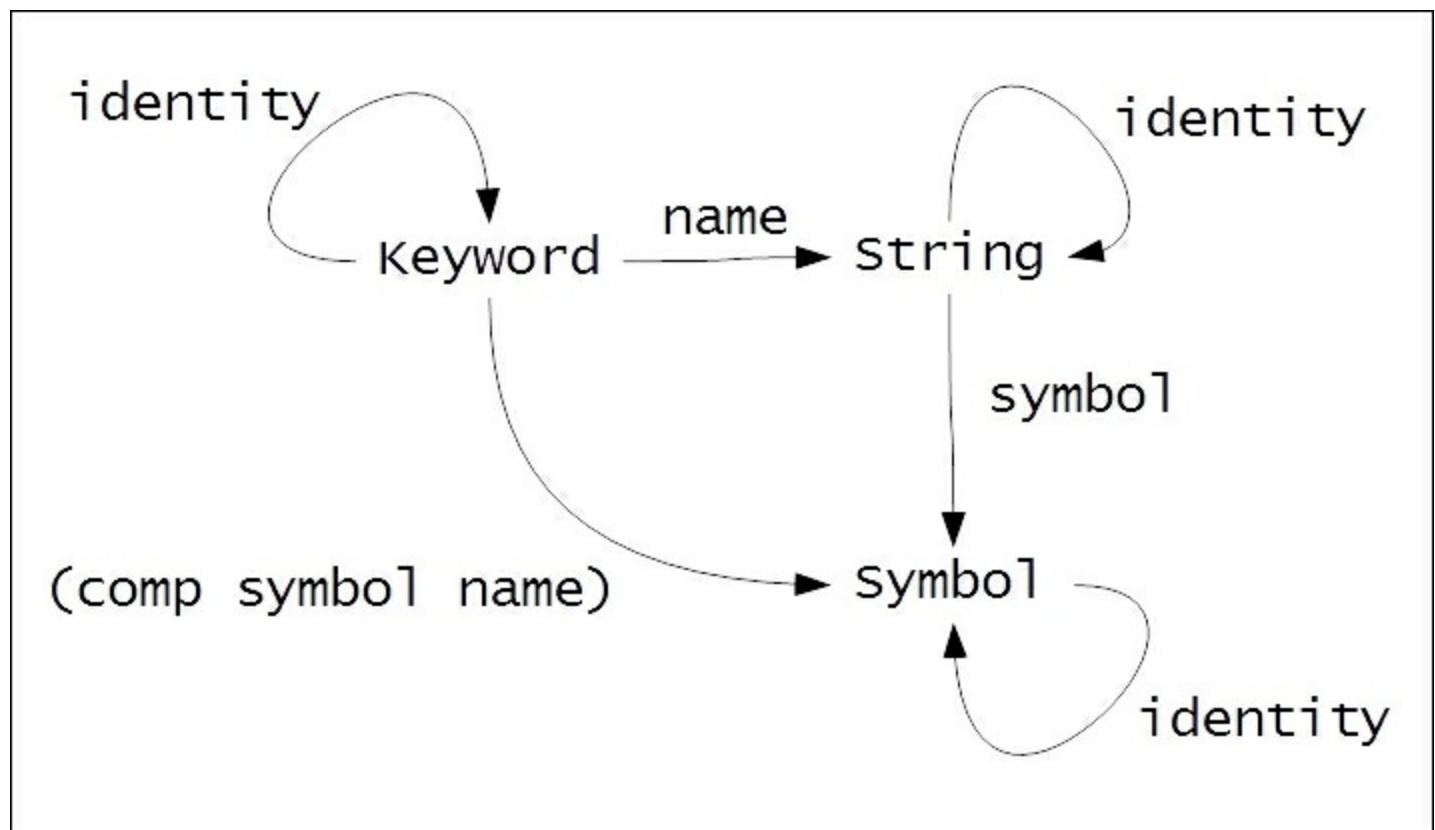
In the preceding diagram, the vertices  $A$ ,  $B$ , and  $C$  are the objects and the arrows are morphisms between these objects. The morphisms  $I_A$ ,  $I_B$ , and  $I_C$  are identity morphisms that map the objects  $A$ ,  $B$ , and  $C$  to themselves. The morphism  $f$  maps  $A$  to  $B$ , and similarly the morphism  $g$  maps  $B$  to  $C$ . These two morphisms can be composed together, as represented by the arrow  $g \circ f$  from  $A$  to  $C$ .

morphism  $g \circ f$  that maps  $A$  directly to  $C$ , and hence the morphism  $g \circ f$  commutes with the morphisms  $f$  and  $g$ . For this reason, the preceding diagram is termed as a *commutative diagram*. Note that identity morphisms in a commutative diagram are generally not shown, unlike in the preceding diagram.

## Note

The following examples can be found in `src/m_clj/c6/demystifying_cat_theory.clj` of the book's source code.

Now, let's translate the previous diagram to Clojure. We shall use the built-in string, symbol and keyword types to depict how morphisms, or rather functions, between these types can be composed together using the `comp` function:



As shown in the preceding diagram, the `name` function converts a keyword to a string, and the `symbol` function converts a string to a symbol. These two functions can be composed into a function that converts a keyword directly to a symbol, represented by the `(comp symbol name)` function. Also, the identity morphisms for each category translate to the `identity` function.

## Note

Internally, the string, symbol, and keyword types are represented by the `java.lang.String`, `clojure.lang.Symbol` and `clojure.lang.Keyword` classes respectively.

We can verify that the `name` and `symbol` functions can be composed together using the `comp` form, as shown in the following REPL output:

```
user> (name :x)
"x"
user> (symbol "x")
x
user> ((comp symbol name) :x)
x
```

This establishes the fact that concepts from category theory have equivalent representations in Clojure, and other programming languages as well. Although it is perfectly valid to think about objects in a category as concrete types like we just described, *algebraic structures* are a more practical substitute for objects. Algebraic structures describe abstract properties of types, rather than what data is contained in a type or how data is structured by a type, and are more like abstract types. Thus, category theory is all about composing functions that operate on abstract types with certain properties.

In Clojure, algebraic structures can be thought of as protocols. Concrete types can implement protocols, and hence a type can represent more than one algebraic structure. The `cats` library (<https://github.com/funcool/cats>)

takes this approach and provides protocol-based definitions of a few interesting algebraic structures. The `cats` library also provides types that implement these protocols. Additionally, this library extends some of the built-in types through these protocols allowing us to treat them as algebraic structures. Although there are several alternatives, `cats` is the only library compatible with ClojureScript.

## Note

The following library dependencies are required for the upcoming examples:

```
[funcool/cats "1.0.0"]
```

Also, the following namespaces must be included in your namespace declaration:

```
(ns my-namespace
  (:require [cats.core :as cc]
            [cats.builtin :as cb]
            [cats.applicative.validation :as cav]
            [cats.monad.maybe :as cmm]
            [cats.monad.identity :as cmi]
            [cats.monad.exception :as cme]))
```

Now, let's study some of the algebraic structures from the `cats` library.

# Using monoids

Let's start by exploring **monoids**. In order to define a monoid, we must first understand what a semigroup is.

## Note

The following examples can be found in `src/m_clj/c6/monoids.clj` of the book's source code.

A *semigroup* is an algebraic structure that supports an associative binary operation. A binary operation, say  $\oplus$ , is termed *associative* if the operation  $((a \oplus b) \oplus c)$  produces the same result as the operation  $(a \oplus (b \oplus c))$ . A monoid is in fact a semigroup with an additional property, as we will see ahead.

The `mappend` function from the `cats.core` namespace will associatively combine a number of instances of the same type and return a new instance of the given type. If we are dealing with strings or vectors, the `mappend` operation is implemented by the standard `concat` function. Thus, strings and vectors can be combined using the `mappend` function, as shown here:

```
user> (cc/mappend "12" "34" "56")
"123456"
user> (cc/mappend [1 2] [3 4] [5 6])
[1 2 3 4 5 6]
```

As strings and vectors support the associative `mappend` operation, they are semigroups. They are also *monoids*, which are simply semigroups that have an *identity element*. It's fairly obvious that the identity element for strings is an empty string, and for vectors it's an empty vector.

This is a good time to introduce a versatile concrete type from the world of

functional programming—the `Maybe` type. The `Maybe` type represents an optional value, and can either be empty or contain a value. It can be thought of as a value in a context or a container. The `just` and `nothing` functions from the `cats.monads.maybe` namespace can be used to construct an instance of the `Maybe` type. The `just` function constructs an instance with a contained value, and the `nothing` function creates an empty `Maybe` value. The value contained in a `Maybe` instance can be obtained by either passing it to the `cats.monads.maybe/from-maybe` function, or dereferencing it (using the `deref` form or the `@` reader macro).

Incidentally, the `Maybe` type is also a monoid, since an empty `Maybe` value, created using the `nothing` function, is analogous to an identity element. We can use the `mappend` function to combine values of the `Maybe` type, just like any other monoid, as shown here:

```
user> @ (cc/mappend (cmm/just "123")
                     (cmm/just "456"))
"123456"
user> @ (cc/mappend (cmm/just "123")
                     (cmm/nothing)
                     (cmm/just "456"))
"123456"
```

The `mappend` function can thus be used to associatively combine any values that are monoids.

# Using functors

Next, let's take a look at **functors**. A functor is essentially a value in a container or a computational context. The `fmap` function must be implemented by a functor. This function applies a supplied function to the value contained in a functor. In object-oriented terminology, a functor can be thought of as a generic type with a single abstract method `fmap`. In a way, reference types, such as refs and atoms, can be thought of as functors that save results, as a reference type applies a function to its contained value in order to obtain the new value that should be stored in it.

## Note

The following examples can be found in `src/m_clj/c6/ functors.clj` of the book's source code.

The `fmap` function from the `cats.core` namespace takes two arguments: a function and a functor. A functor itself defines what happens when an instance of the functor is passed to the `fmap` function. The `cats` library extends vectors as functors. When a vector is passed to the `fmap` function along with a function, the supplied function is applied to all elements in the vector. Wait a minute! Isn't that what the `map` function does? Well, yes, but the `map` function always returns a lazy sequence. On the other hand, the `fmap` function will return a value with the same concrete type as the functor that is passed. The behavior of the `map` and `fmap` functions can be compared as follows:

```
user> (map inc [0 1 2])
(1 2 3)
user> (cc/fmap inc [0 1 2])
[1 2 3]
```

As shown above, the `map` function produces a lazy sequence, which gets realized into a list in the REPL, when it is passed a vector along with the `inc` function. The `fmap` function, however, produces a vector when passed

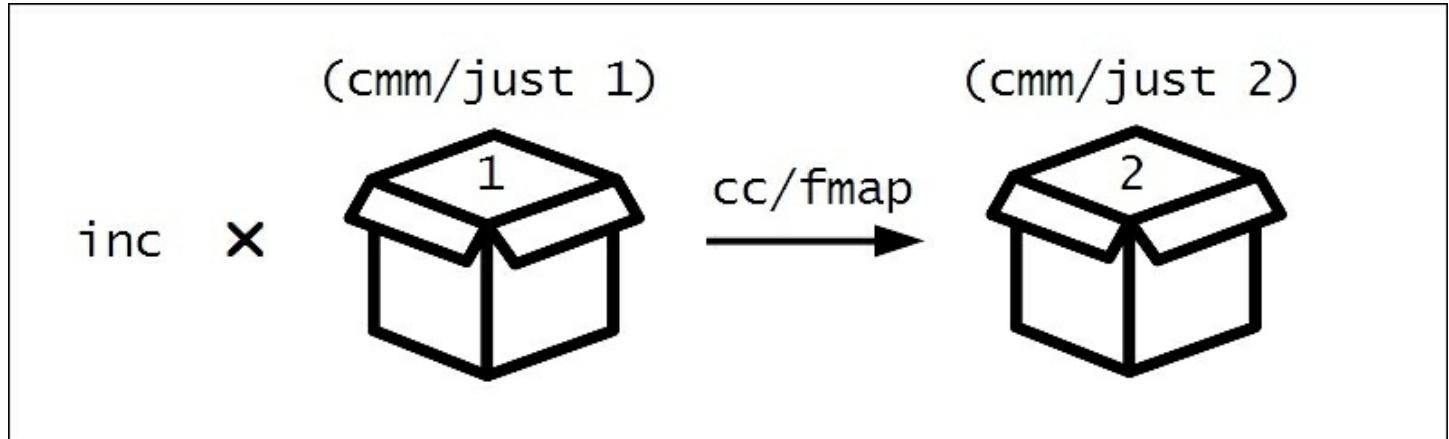
the same arguments. We should note that the `fmap` function is also aliased as `<$>`. Lazy sequences and sets can also be treated as functors, as shown here:

```
user> (cc/<$> inc (lazy-seq '(1)))
(2)
user> (cc/<$> inc #{1})
#{2}
```

The `Maybe` type is also a functor. The `fmap` function returns a *maybe* when it is passed a *maybe*, as shown here:

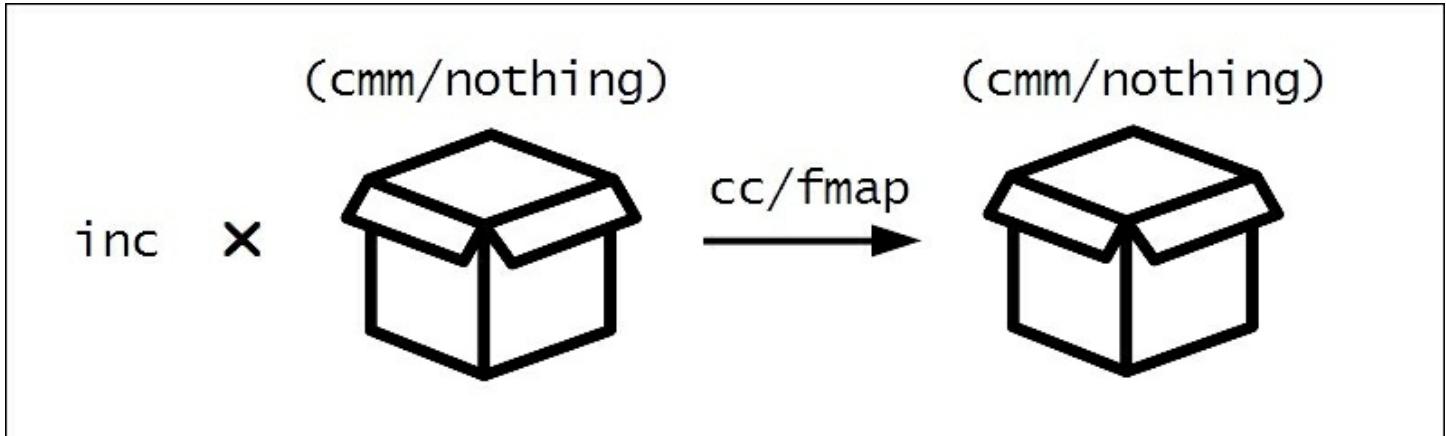
```
user> (cc/fmap inc (cmm/just 1))
#<Just@ff5df0: 2>
user> (cc/fmap inc (cmm/nothing))
#<Nothing@d4fb58: nil>
```

The `fmap` function applies the function `inc` to a `Maybe` value only when it contains a value. This behavior of the `fmap` function can be illustrated by the following diagram:



The preceding diagram depicts how the `fmap` function is passed the `inc` function and the expression `(cmm/just 1)`, and returns a new functor instance. The `fmap` function extracts the value from this `Maybe` value, applies the `inc` function to the value, and creates a new `Maybe` value with the result. On the other hand, the `fmap` function will simply return an

empty `Maybe` instance, created using the `nothing` function, without touching it, as shown in the following diagram:



This behavior of the `fmap` function is defined by the implementation of the `Maybe` type. This is because a functor itself gets to define how the `fmap` function behaves with it. Of course, implementing the `fmap` function is not enough to qualify a type as a functor. There are also functor laws that have to be satisfied by any plausible implementation of a functor. The functor laws can be described as follows:

1. Passing an identity morphism and a functor  $F$  to `fmap` must return the functor  $F$  without any modification. We can translate this into Clojure using the `identity` function, as follows:

```
user> (cc/<$> identity [0 1 2])
[0 1 2]
```

2. Passing a functor  $F$  and a morphism  $f$  to `fmap`, followed by passing the result and another morphism  $g$  to `fmap`, must be equivalent to calling `fmap` with the functor  $F$  and the composition  $g \circ f$ . We can verify this using the `comp` function, as shown here:

```
user> (->> [0 1 2]
  (cc/<$> inc)
  (cc/<$> (partial + 2)))
[3 4 5]
```

```
user> (cc/<$> (comp (partial + 2) inc) [0 1 2])
[3 4 5]
```

The first law describes identity morphisms, and the second law upholds the composition of morphisms. These laws can be thought of as optimizations that can be performed by the `fmap` function when used with valid functors.

# Using applicative functors

Applicative functors are a subset of functors with a few additional requirements imposed on them, thus making them a bit more useful. Similar to functors, applicative functors are computational contexts that are capable of applying a function to the value contained in them. The only difference is that the function to be applied to an applicative functor must itself be wrapped in the context of an applicative functor. Applicative functors also have a different interface of functions associated with them. An applicative functor, in `cats`, is manipulated using two functions: `fapply` and `pure`.

## Note

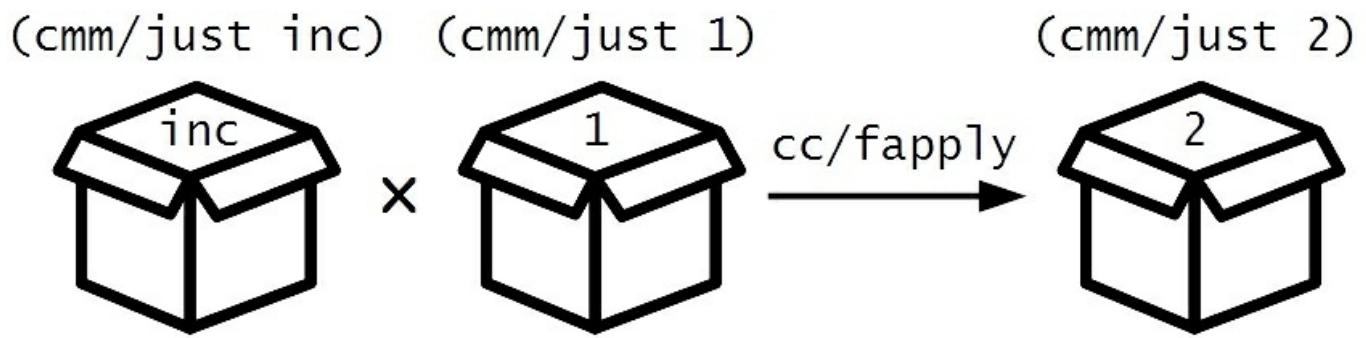
The following examples can be found in `src/m_clj/c6/applicatives.clj` of the book's source code.

The `fapply` function from the `cats.core` namespace can be called with an applicative functor, as follows:

```
user> @(cc/fapply (cmm/just inc)
                     (cmm/just 1))
```

2

Here, we again use the `Maybe` type, this time as an applicative functor. The `fapply` function unwraps the `inc` function and the value `1` from the `Maybe` values, combines them and returns the result `2` in a new `Maybe` instance. This can be illustrated with the following diagram:



The `cats.core/pure` function is used to create a new instance of an applicative functor. We must pass an implementation-specific context, such as `cats.monads.maybe/context`, and a value to the `pure` function, as follows:

```
user> (cc/pure cmm/context 1)
#<Just@cefb4d: 1>
```

The `cats` library provides an `alet` form to easily compose applicative functors. Its syntax is similar to that of the `let` form, as follows:

```
user> @(cc/alet [a (cmm/just [1 2 3])
                  b (cmm/just [4 5 6])]
                  (cc/mappend a b))
[1 2 3 4 5 6]
```

The value returned by the body of the `alet` form, shown previously, is wrapped in a new applicative functor instance and returned. The surrounding `alet` form is dereferenced, and thus the entire expression returns a vector.

The `<*>` function from the `cats.core` namespace is a variadic form of the `fapply` function. It accepts a value representing an applicative functor followed by any number of functions that produce applicative functors. The `cats` library also provides the `Validation` applicative functor type for

validating properties of a given object. This type can be constructed using the `ok` and `fail` forms in the `cats.applicative.validation` namespace. Let's say we want to validate a map representing a page with some textual content. A page must have a page number and an author. This validation can be implemented as shown in *Example 6.1*:

```
(defn validate-page-author [page]
  (if (nil? (:author page))
    (cav/fail {:author "No author"})
    (cav/ok page)))

(defn validate-page-number [page]
  (if (nil? (:number page))
    (cav/fail {:number "No page number"})
    (cav/ok page)))

(defn validate-page [page]
  (cc/let [a (validate-page-author page)
          b (validate-page-number page)]
    (cc/<*> (cc/pure cav/context page)
              a b)))
```

### *Example 6.1: The `cats.applicative.validation` type*

The `validate-page-author` and `validate-page-number` functions in *Example 6.1* check whether a map contains the `:author` and `:number` keys respectively. These functions create an instance of the `Validation` type using the `ok` function, and similarly use the `fail` function to create a `Validation` instance that represents a validation failure. Both the `validate-page-author` and `validate-page-number` functions are composed together using the `<*>` function. The first argument passed to `<*>` will have to be an instance of the `Validation` type created using the `pure` function. The `validate-page` function can thus validate maps representing pages, as shown here:

```
user> (validate-page {:text "Some text"})
<Fail@1203b6a: {:author "No author", :number "No page number"}>
user> (validate-page {:text "Some text" :author "John" :number 1})
```

```
#<Ok@161b2f8: { :text "Some text", :author "John", :number 1 } >
```

A successful validation will return a `Validation` instance containing the `page` object, and an unsuccessful one will return an instance of the `Validation` type with the appropriate validation messages as a map. The concrete types for these two cases are `Ok` and `Fail`, as shown in the preceding output.

Applicative functors must themselves define the behavior of the `fapply` and `pure` functions with them. Of course, there are laws that applicative functors must obey too. In addition to the identity and composition laws of functors, applicative functors also conform to the *homomorphism* and *interchange* laws. The reader is encouraged to find out more about these laws before implementing their own applicative functors.

# Using monads

Finally, let's take a look at an algebraic structure that helps us build and compose a sequence of computations: a **monad**. There are countless tutorials and articles on the web that explain monads and how they can be used. In this section, we will look at monads in our own unique and Clojure-y way.

In category theory, a monad is a morphism between functors. This means that a monad transforms the context of a contained value into another context. In pure functional programming languages, monads are data structures used to represent computations that are defined in steps. Each step is represented by an operation on a monad, and several of these steps can be chained together. Essentially, a monad is a composable abstraction of a step of any computation. A distinct feature of monads is that they allow us to model impure side effects, which may be performed in the various steps of a given computation, using pure functions.

Monads abstract the way a function binds values to arguments and returns a value. Formally, a monad is an algebraic structure that implements two functions: `bind` and `return`. The `bind` function is used to apply a function to the value contained in a monad, and the `return` function can be thought of as a construct for wrapping values in a new monad instance. The type signatures of the `bind` and `return` functions can be described by the following pseudo code:

```
bind : (Monad A a, [A -> Monad B] f) -> Monad B  
return : (A a) -> Monad A
```

The type signature of the `bind` function states that it accepts a value of type `Monad A` and a function that converts a value of type `A` to another value of type `Monad B`, which is simply a monad containing a value of type `B`. Also, the `bind` function returns a type `Monad B`. The `return` function's type signature shows that it takes a value of type `A` and returns a type

`Monad A`. Implementing these two functions allows a monad to execute any code, defined in its `bind` implementation, before the supplied function `f` is applied to the value contained in the monad. A monad can also define code to be executed when the supplied function `f` returns a value, as defined by the monad's implementation of the `return` function.

Due to the fact that a monad can do more than just call a function over its contained value when passed to the `bind` function, a monad is used to indicate side effects in pure functional programming languages. Let's say, we have a function that maps type `A` to `B`. A function that maps type `A` to `Monad B` can be used to model the side effects that can occur when a value of type `A` is converted to another value of type `B`. In this way, monads can be used to represent side effects, such as IO, change of state, exceptions, and transactions.

Some programmers may even argue that monads are unnecessary in a language with macros. This is true in some sense, because macros can encapsulate side effects in them. However, monads help us to be explicit about any side-effects, which is quite useful. In fact, monads are the only way to model side effects in pure functional programming languages. Because monads can represent side effects, they allow us to write imperative-style code, which is all about mutation of state, in a pure functional programming language.

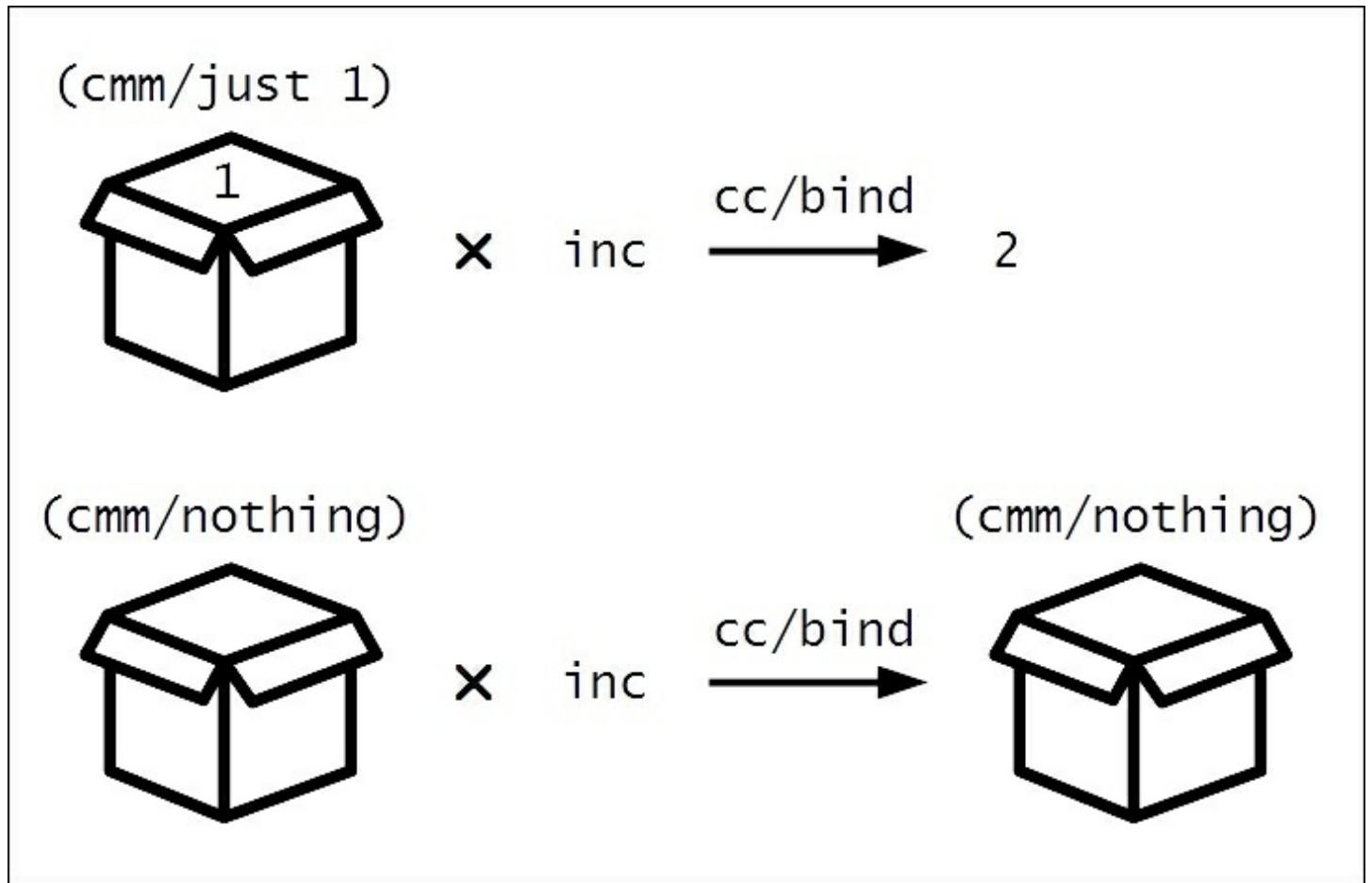
## Note

The following examples can be found in `src/m_clojure/c6/monads.clj` of the book's source code.

Let's now look at how the `Maybe` type from the `cats` library can take the form of a monad. We can pass a `Maybe` value along with a function to the `cats.core/bind` function to call the supplied function with the contained value in the monad. This function is aliased as `>>=`. The behavior of the `bind` function with a `Maybe` type is shown here:

```
user> (cc/bind (cmm/just 1) inc)
2
user> (cc/bind (cmm/nothing) inc)
#<Nothing@24e44b: nil>
```

In this way, we can *bind* the `inc` function to a `Maybe` monad. The expression shown in the preceding output can be depicted by the following diagram:



The `inc` function is applied to a `Maybe` monad only if it contains a value. When a `Maybe` monad does contain a value, applying the `inc` function to it using the `bind` function will simply return `2`, and not a monad containing `2`. This is because the standard `inc` function does not return a monad. On the other hand, an empty `Maybe` value is returned untouched. To return a monad in both the preceding cases, we can use the `return` function from

the `cats.core` namespace, as shown here:

```
user> (cc/bind (cmm/just 1) #(-> % inc cc/return))
#<Just@208e3: 1>
user> (cc/bind (cmm/nothing) #(-> % inc cc/return))
#<Nothing@1e7075b: nil>
```

The `lift-m` form can be used to *lift* a function that returns a type `A` to return a monad containing a type `A`. The concrete type of the return value of a lifted function depends on the monad context passed to it. If we pass a `Maybe` monad to a lifted version of `inc`, it will return a new instance of the `Maybe` monad, as shown here:

```
user> ((cc/lift-m inc) (cmm/just 1))
#<Just@1eaaab: 2>
```

We can also compose several calls to the `bind` function, as long as the function passed to the `bind` function produces a monad, as shown here:

```
user> (cc/>>= (cc/>>= (cmm/just 1)
                     #(-> % inc cmm/just))
                     #(-> % dec cmm/just))
#<Just@91ea3c: 1>
```

Of course, we can also compose calls to the `bind` function to change the type of monad. For example, we can map a `Maybe` monad to an `Identity` monad, which is constructed using the `cats.monads.identity/identity` function. We can modify the preceding expression to return an `Identity` monad as shown here:

```
user> (cc/>>= (cc/>>= (cmm/just 1)
                     #(-> % inc cmm/just))
                     #(-> % dec cmi/identity))
#<Identity@dd6793: 1>
```

As shown in the preceding output, calling the `bind` function multiple times can get a bit cumbersome. The `mlet` form lets us compose expressions that return monads, as shown in *Example 6.2*:

```
(defn process-with-maybe [x]
  (cc/mlet [a (if (even? x)
                    (cmm/just x)
                    (cmm/nothing) )
            b (do
                (println (str "Incrementing " a))
                (-> a inc cmm/just)) ]
    b))
```

### *Example 6.2. The mlet form*

In short, the `process-with-maybe` function defined in *Example 6.2* checks whether a number is even, then prints a line and increments the number. As we use the `Maybe` type, the last two steps of printing a line and incrementing a value are performed only if the input `x` is even. In this way, an empty `Maybe` monad, created using the `nothing` function, can be used to short-circuit a composition of monads. We can verify this behavior of the `process-with-maybe` function in the REPL, as shown here:

```
user> (process-with-maybe 2)
Incrementing 2
3
user> (process-with-maybe 3)
#<Nothing@1ebd3fe: nil>
```

As shown here, the `process-with-maybe` function prints a line only when the supplied value `x` is an even number. If not, an empty `Maybe` monad instance is returned.

The previous examples describe how we can use the `Maybe` monad. The `cats` library also provides implementations of the `Either` and `Exception` monads, in the `cats.monads.either` and `cats.monads.exception` namespaces respectively. Let's explore a few constructs from the `cats.monads.exception` namespace.

We can create a new `Exception` monad instance using the `success` and `failure` functions. The `success` form can be passed any value, and it returns a monad that represents a successful step in a computation. On the

other hand, the `failure` function must be passed a map containing an `:error` key that points to an exception, and returns a monad that represents a failure in a computation. The value or exception contained in an `Exception` monad can be obtained by dereferencing it (using the `deref` form or the `@` reader macro). Another way to create an `Exception` monad instance is by using the `try-on` macro. The following output describes how these constructs can be used to create an instance of the `Exception` monad:

```
user> (cme/success 1)
#<Success@441a312 [1]>
user> (cme/failure { :error (Exception.) })
#<Failure@4812b43 [#<java.lang.Exception>]>
user> (cme/try-on 1)
#<Success@5141a5 [1]>
user> @(cme/try-on 1)
1
```

The `try-on` macro will return a failure instance of the `Exception` monad if the expression passed to it throws an error, as shown here:

```
user> (cme/try-on (/ 1 0))
#<Failure@bc1115 [#<java.lang.ArithmaticException>]>
user> (cme/try-on (-> 1 (/ 0) inc))
#<Failure@f2d11a [#<java.lang.ArithmaticException>]>
```

A failure instance of an `Exception` monad can be used to short-circuit a composition of monads. This means that binding an `Exception` monad to a function will not call the supplied function if the monad contains an error. This is similar to how exceptions are used to halt computations. We can verify this using the `bind` function, as shown here:

```
user> (cc/bind (cme/try-on (/ 1 1)) #(-> % inc cc/return))
#<Success@116ea43 [2]>
user> (cc/bind (cme/try-on (/ 1 0)) #(-> % inc cc/return))
#<Failure@0x1c90acb [#<java.lang.ArithmaticException>]>
```

Instances of the `Exception` monad can also be created using the `try-or-else` and `try-or-recover` macros from the `cats.monads.exception`

namespace. The `try-or-else` form must be passed an expression and a default value. If the expression passed to this form throws an exception, the default value is wrapped in an `Exception` monad instance and returned. The `try-or-recover` form must be passed a 1-arity function in place of the default value. In case an error is encountered, the `try-or-recover` macro will invoke the supplied function and relay the value returned by it. The `try-or-else` and `try-or-recover` forms are demonstrated as follows:

```
user> (cme/try-or-else (/ 1 0) 0)
#<Success@bd15e6 [0]>
user> (cme/try-or-recover (/ 1 0)
                           (fn [e]
                             (if (instance? ArithmeticException e)
                                 0
                                 :error)))
0
```

In this way, monads can be used to model side effects using pure functions. We've demonstrated how we can use the `Maybe` and `Exception` monad types. The `cats` library also implements other interesting monad types. There are monad laws as well, and any monad that we implement must conform to these laws. You are encouraged to learn more about the monad laws on your own.

# Summary

In this chapter, we talked about the notations and terminology used in category theory. We also discussed several algebraic types from category theory. Each of these abstractions have laws that must be satisfied by their implementations, and these laws can be thought of as optimizations for computations that use these algebraic types.

In the next chapter, we will look at a different paradigm of programming altogether—logic programming.

# Chapter 7. Programming with Logic

We will now take a step back from the realm of functional programming and explore a completely different paradigm—**logic programming**. Logic programming has its own unique way of solving computational problems. Of course, logic programming isn't the only way to solve a problem, but it's interesting to see what kind of problems can be easily solved with it.

Although logic programming and functional programming are two completely different paradigms, they do have a few commonalities. Firstly, both of these paradigms are forms of *declarative programming*. Studies and papers have also shown that it is possible to implement the semantics of logic programming within a functional programming language. Hence, logic programming operates at a much higher degree of abstraction than functional programming. Logic programming is more suited for problems in which we have a set of rules, and we intend to find all the possible values that conform to these rules.

In this chapter, we look at logic programming in Clojure through the `core.logic` library. We will also study a few computational problems and how we can solve them in a concise and elegant manner using logic programming.

## Diving into logic programming

In Clojure, logic programming can be done using the `core.logic` library (<https://github.com/clojure/core.logic/>). This library is a port of **miniKanren**, which is a domain-specific language for logic programming. **miniKanren** defines a set of simple constructs for creating logical relations and generating results from them.

## Note

miniKanren was originally implemented in the Scheme programming language. You can find out more about miniKanren at <http://minikanren.org/>.

A program written using logic programming can be thought of as a set of logical relations. **Logical relations** are the elementary building blocks of logic programming, just as functions are for functional programming. The terms *relation* and *constraint* are used interchangeably to refer to a logical relation. The `core.logic` library is in fact an implementation of constraint-based logic programming.

A relation can be thought of as a function that returns a goal, and a goal can either be a success or a failure. In the `core.logic` library, a goal is represented by the `succeed` and `fail` constants. Another interesting aspect of relations is that they can return multiple results, or even no results. This is analogous to a function that produces a sequence of values, which could be empty, as a result. Functions such as `keep` and `filter` fit this description perfectly.

## Note

The following library dependencies are required for the upcoming examples:

```
[org.clojure/core.logic "0.8.10"]
```

Also, the following namespaces must be included in your namespace declaration:

```
(ns my-namespace
  (:require [clojure.core.logic :as l]
            [clojure.core.logic.fd :as fd]))
```

The following examples can be found in

`src/m_clj/c7/diving_into_logic.clj` of the book's source code.

## Solving logical relations

As a convention, relations have their name suffixed with an "o". For example, the `consO` construct from the `clojure.core.logic` namespace is a relation that represents the behavior of the `cons` function. Logical programming constructs that use multiple logical relations, such as `conde` and `matche`, end with an "e". We will explore these constructs later on in this chapter. Let's now focus on how we can solve problems with logical relations.

The `run*` macro, from the `clojure.core.logic` namespace, processes a number of goals to generate all possible results. The semantics of the `run*` form allow us to declare a number of logical variables that can be used in relations to return goals. The `run*` form returns a list of possible values for the logical variables it defines. An expression using the `run*` form and a set of relations is essentially a way of asking the question "What must the universe look like for these relations to be true?" to a computer and asking it to find the answer.

An equality test can be performed using the `run*` macro in combination with the `clojure.core.logic/==` form, as shown here:

```
user> (l/run* [x]
                 (l/== x 1))
(1)
user> (l/run* [x]
                 (l/== 1 0))
()
```

Both the statements using the `run*` form in the preceding output find all possible values of the logical variable `x`. The relation `(l/== x 1)` returns a goal that succeeds when the value of `x` is equal to 1. Obviously, the only value that `x` can have for this relation to succeed is 1. The `run*` form evaluates this relation to return 1 in a list. On the other hand, the relation

`(1/== 1 0)` is logically false, and thus produces no results when passed to the `run*` form. This means that there are no values of `x` for which `1` is equal to `0`.

A relation built using the `==` form from the `clojure.core.logic` namespace is called *unification*. Unification is often used in logic programming like variable assignment from other paradigms, as it's used to assign values to variables. Conversely, a *disequality* represents a relation in which a logical variable cannot be equal to a given value. The `clojure.core.logic/!=` form is used to construct a disequality relation, as shown here:

```
user> (1/run* [x]
  (1/!= 1 1))
()
user> (1/run* [x]
  (1/== 1 1))
(_0)
```

The first statement in the preceding output produces no results since the relation `(1/!= 1 1)` is logically false. An interesting quirk is that the second statement, which has a goal that always succeeds produces a single result `_0`, which represents an *unbound* logical variable. As we don't assign a value to `x` through unification, its value is said to be unbound. The symbols `_0`, `_1`, `_2`, and so on (also written as `_.0`, `_.1`, `_.2`, and so on) represent unbound logical variables in the context of a `run*` form.

The `clojure.core.logic/conso` form is useful in modeling the behavior of the standard `cons` function as a relation. It takes three arguments, of which two are the same as the `cons` function. The first two arguments passed to a `conso` form represent the head and tail of a sequence.

The third argument is a sequence that is expected to be returned on applying the `cons` function on the first two arguments. The `conso` relation can be demonstrated as follows:

```
user> (1/run* [x]
  (1/conso 1 [2 x])
```

```
[1 2 3]))
```

(3)

The expression using a `conso` relation, in the preceding output, will solve for values of `x` that produce the value `[1 2 3]` when a `cons` form is applied to the values `1` and `[2 x]`. Obviously, `x` must be `3` for this relation to be true, and hence the result `3` is produced.

Logical variables can be created without the use of a `run*` form using the `lvar` function from the `clojure.core.logic` namespace. Within a `run*` form, we can create local logical variables using the `clojure.core.logic/fresh` macro. Variables declared using a `fresh` form will not be part of the final result produced by the surrounding `run*` form. For example, consider the expressions using the `run*` forms in the following output:

```
user> (l/run* [x y]
  (l/== x y)
  (l/== y 1))
([1 1])
user> (l/run* [x]
  (l/fresh [y]
    (l/== x y)
    (l/== y 1)))
(1)
```

The first expression shown previously produces the vector `[1 1]` as a result, whereas the second expression produces `1` as the result. This is because we specify a single logical variable `x` and use a `fresh` form to internally declare the logical variable `y` in the second expression.

The `run*` form searches for results exhaustively from the set of relations provided to it. If we intend to find a limited number of results and avoid performing any additional computation to find any more results, we should use the `run` macro from the `clojure.core.logic` namespace instead. The `run` form has the same semantics of a `run*` form, but additionally requires the number of desired results to be passed to it as the first argument.

The `clojure.core.logic.fd` namespace provides us with several constructs to deal with relations that are constrained over a finite range of values. For example, suppose we wanted to find values within the range of 0 to 100 that are greater than 10. We can easily express this relation using the `>`, `in`, and `interval` forms from the `clojure.core.logic.fd` namespace and extract the first five values from it using a `run` form, as shown here:

```
user> (l/run 5 [x]
  (fd/in x (fd/interval 0 100))
  (fd/> x 10))
(11 12 13 14 15)
```

The preceding expression uses the `in` and `interval` forms to constrain the value of the variable `x`. The expression using these two forms ensures that `x` is within the range of 0 and 100. Also, the `clojure.core.logic.fd/>` function defines a relation in which `x` must be greater than 10. The surrounding `run` form simply extracts the first five possible values of `x` from the relations supplied to it. There are also several other arithmetic comparison operators, namely `<`, `<=`, and `>=`, implemented in the `clojure.core.logic.fd` namespace. Instead of specifying a range of values to the `in` macro, we can also enumerate the possible values of a variable by using the `clojure.core.logic.fd/domain` form.

The `firsto` form can be used to describe a relation in which the value in a given variable must be the first element in a collection. We can try out both the `domain` and `firsto` forms in the REPL as shown here:

```
user> (l/run 1 [v a b x]
  (l== v [a b])
  (fd/in a b x (fd/domain 0 1 2))
  (fd/< a b)
  (l/firsto v x))
([[0 1] 0 1 0])
```

In the preceding expression, we solve for the first set of values of `v`, `a`, `b`, and `x` that satisfy the following relations. The value of `a` must be less than that of `b`, which is shown using the `<` form, and both `a` and `b` must

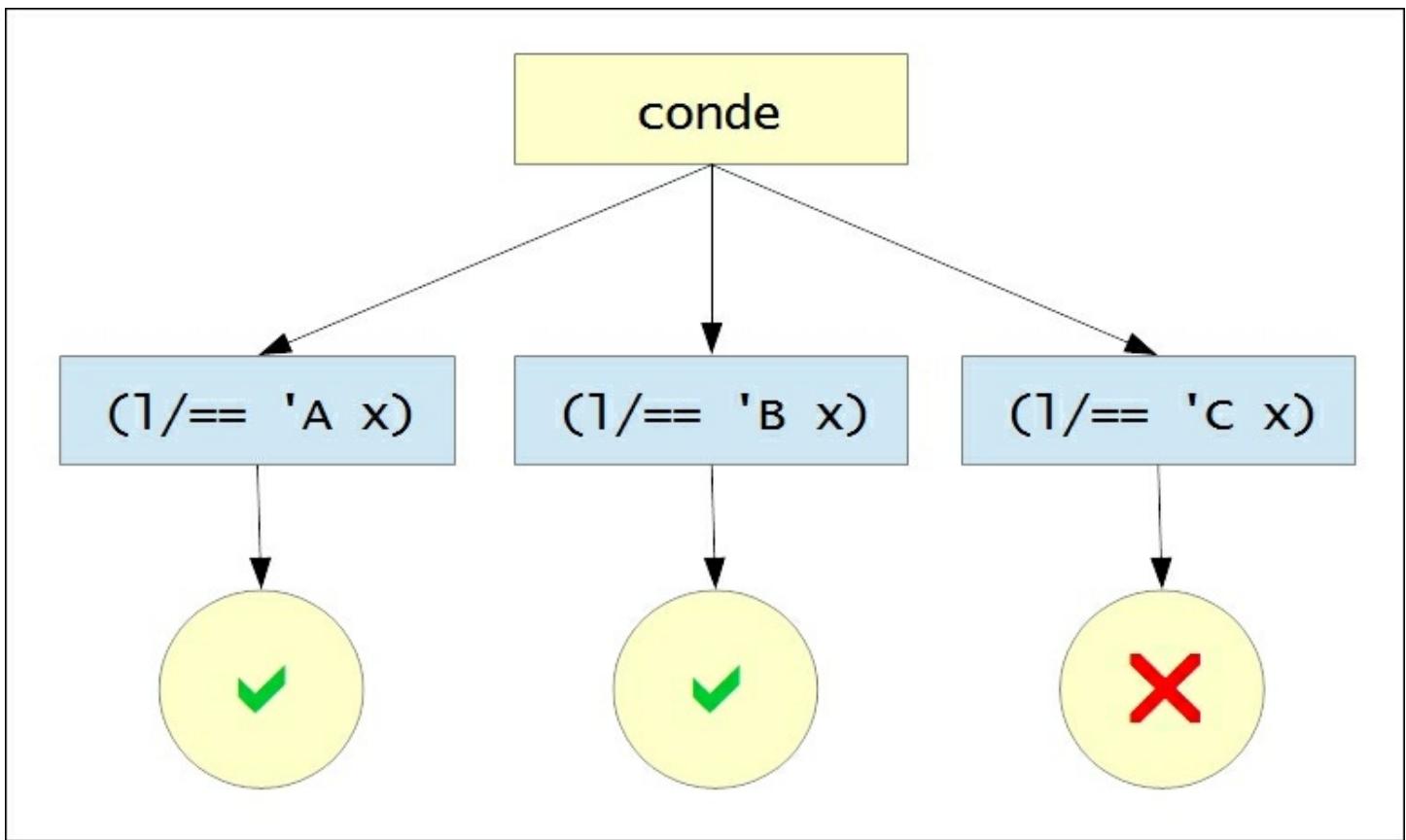
constitute the elements of a vector `v`, which is shown using the `==` form. Also, `a`, `b`, and `x` must be equal to either `0`, `1`, or `2`, as described by a composition of the `in` and `domain` forms. Lastly, the first element of the vector `v` must be equal to the value `x`. These relations generate the vector `[0 1]` and the values `0`, `1`, and `0` for `a`, `b`, and `x` respectively. Note the arity of the `in` form in the previous expression that allows multiple logical variables to be passed to it along with a constraint.

## Combining logical relations

The `clojure.core.logic/conde` form allows us to specify multiple relations, and is a bit similar to the standard `cond` form. For example, consider the following expression that uses the `conde` form:

```
user> (l/run* [x]
  (l/conde
    ((l/== 'A x) l/succeed)
    ((l/== 'B x) l/succeed)
    ((l/== 'C x) l/fail)))
(A B)
```

The preceding expression, which uses a `conde` form, performs equality checks for the symbols `A`, `B`, and `C` against the logical variable `x`. Only two of these checks produce a goal that succeeds, which is described using the `succeed` and `fail` constants in the clauses of the `conde` form. This logical branching by the `conde` form in the preceding expression can be illustrated through the following diagram:



The `conde` form in our previous example creates a conditional check for three clauses. Out of these three clauses, only two succeed, and hence the symbols `A` and `B` are returned as results. We should note that the clauses defined in a `conde` form can take any number of relations. Also, the use of the `1/succeed` constant is implicit, and we only need to use the `1/fail` constant to represent a goal that fails.

Another way to perform equality checks is by pattern matching. This can be done using the `clojure.core.logic/matche` form. The `matche` form is thus a more idiomatic way to define conditional branches involving logical variables, as shown here:

```

user> (1/run* [x]
  (1/conde
    ((1/== 'A x) 1/succeed)
    ((1/== 'B x) 1/succeed)))
(A B)
  
```

```

user> (l/run* [x]
  (l/matche [x]
    ([ 'A] l/succeed)
    ([ 'B] l/succeed)))
(A B)

```

Both of the preceding expressions produce the same result. The only difference between these expressions is that the first one uses a `conde` form and the second one performs a pattern match using a `matche` form. Also, the `l/succeed` constant is implicit and does not need to be specified, similar to a `conde` form. The `_` wildcard is also supported by the `matche` form, as shown here:

```

user> (l/run* [x]
  (l/matche [x]
    ([ 'A])
    ([_] l/fail)))
(A)

```

In the preceding expression, we solve for all values of `x` that match the pattern '`A`'. All other cases fail, which is described using the `_` wildcard and the `l/fail` constant. Of course, the pattern using the `_` wildcard is implicit and is only shown to describe how it can be used in a `matche` form.

The `matche` construct also supports destructuring of sequences. A sequence can be destructured by a `matche` form using a dot (`.`) to delimit the head and tail of the sequence, as shown here:

```

user> (l/run* [x]
  (l/fresh [y]
    (l/== y [1 2 3])
    (l/matche [y]
      ([[1 . x]]))))
((2 3))

```

In the preceding expression, the logical variable `x` must be `(2 3)` for the relation defined using the `matche` form to succeed. We can define relations using a syntax similar to the `defn` form using the `defne` macro from the

`clojure.core.logic` namespace. The `defne` form allows us to define relations in pattern matching style. Incidentally, a lot of constructs in the `core.logic` library are defined using the `defne` form. For example, consider the definition of the `membero` relation in *Example 7.1*:

```
(1/defne membero [x xs]
  ([_ [x . ys]])
  ([_ [y . ys]]
   (membero x ys)))
```

*Example 7.1: Defining the membero relation using the defne macro*

The `membero` relation is used to ensure that a value `x` is a member of the collections `xs`. The implementation of this relation destructures the collection `xs` into its head and tail parts. If the value `x` is the head of the collection `xs`, the relation succeeds, otherwise the relation is called recursively with the value `x` and the tail of the destructured list `ys`. We can try out this relation with the `run*` form in the REPL, as shown here:

```
user> (1/run* [x]
  (membero x (range 5))
  (membero x (range 3 10)))
(3 4)
```

The preceding expression solves for values of `x` that are contained in the range 0 to 5 as well as in the range 3 to 10. The results 3 and 4 are produced from these two relations that use the `membero` form.

While dealing with logical variables, it's important to note that we cannot use standard functions to perform any computation with them. In order to extract values from a bunch of logical variables, we have to use the `clojure.core.logic/project` form. For example, consider the following statement:

```
user> (1/run 2 [x y]
  (1/membero x (range 1 10))
  (1/membero y (range 1 10))
  (1/project [x y])
```

```
(1/== (+ x y) 5))  
([1 4] [2 3])
```

The preceding statement solves for two values of  $x$  and  $y$  such that they are both in the range 1 to 10 and their sum is equal to 5. The values [1 4] and [2 3] are returned as results. The `project` form is used to extract the values of  $x$  and  $y$ , or else the `+` function would throw an exception.

Thus, the `core.logic` library provides us with a handful of constructs that can be used to define logical relations, combine them, and generate results from them.

# Thinking in logical relations

Now that we are well versed with the various constructs from the `core.logic` library, let's look at some real world problems that can be solved through logic programming.

## Solving the n-queens problem

The **n-queens problem** is an interesting problem that can be implemented using logical relations. The objective of the n-queens problem is to place  $n$  queens on an  $n \times n$  sized chessboard such that no two queens are a threat to each other. This problem is a generalization of the *eight queens problem* published by Max Bezzel in 1848, which involves eight queens. In fact, we can actually solve the n-queens problem for any number of queens, as long as we are dealing with four or more queens. Traditionally, this problem can be solved using an algorithmic technique called *backtracking*, which is essentially an exhaustive search for all possible solutions to a given problem. However, in this section, we will solve it using logical relations.

Let's first define how a queen can be used. As we all know, a queen can move as she wishes! A queen can move horizontally, vertically, or diagonally on a chessboard. If any other chess piece is on the same path on which a queen can be moved, then the queen is a threat to it. The position of a chess piece on the chessboard can be specified using a pair of integers, just like how Cartesian coordinates can be used to represent the position of a point on a plane. Suppose the pairs  $(x_1, y_1)$  and  $(x_2, y_2)$  represent the positions of two queens on the chessboard. As they can threaten each other horizontally, vertically, or diagonally, there are three distinct cases we must avoid:

- The queens cannot be on the same vertical path, that is,  $x_1$  equal to  $x_2$ .
- Similarly, the queens cannot be on the same horizontal path, that is,  $y_1$  equal to  $y_2$ .

- The queens cannot be on the same diagonal path, in which case the ratio vertical and horizontal distance between them is either 1 or -1. This is actually a trick from coordinate geometry, and its proof is way out of the scope of our discussion. This case can be concisely represented by the following equations:

$$\frac{y_2 - y_1}{x_2 - x_1} = 1 \text{ or } \frac{y_2 - y_1}{x_2 - x_1} = -1$$

These are the only rules that determine whether two queens threaten each other. Yet if you think about them from a procedural or object-oriented perspective, implementing them could require a good amount of code. On the contrary, if we think in terms of relations, we can implement these three rules fairly easily using the `core.logic` library, as shown in the following *Example 7.2*:

## Note

The following examples can be found in `src/m_clj/c7/nqueens.clj` of the book's source code. This example is based on code from *n-queens with core.logic* by Martin Trojer (<http://martinsprogrammingblog.blogspot.in/2012/07/n-queens-with-corelogic-take-2.html>).

```
(1/defne safeo [q qs]
  ([_ ()])
  ([[x1 y1] [[x2 y2] . t]])
    (l/!= x1 x2)
    (l/!= y1 y2)
    (l/project [x1 x2 y1 y2]
      (l/!= (- x2 x1) (- y2 y1))
      (l/!= (- x1 x2) (- y2 y1)))
    (safeo [x1 y1] t)))
  )

(1/defne nqueenso [n qs]
```

```

([_ ()])
([n [[x y] . t]]
  (nqueenso n t)
  (l/membero x (range n))
  (safeo [x y] t)))

(defn solve-nqueens [n]
  (l/run* [qs]
    (l/== qs (map vector (repeatedly l/lvar) (range n)))
    (nqueenso n qs)))

```

*Example 7.2: The n-queens problem*

In *Example 7.2*, we define two relations, namely `safeo` and `nqueenso`, to describe the n-queens problem. Both of these relations must be passed a list `qs` as an argument, where `qs` contains coordinate pairs that represent the positions of queens placed on the chessboard. They are interestingly recursive relations, and the termination is specified by the case in which `qs` is empty.

The `safeo` relation is an implementation of the three rules that determine whether two queens threaten each other. Note the way this relation uses a project form to extract the values `x1`, `y1`, `x2`, and `y2` to handle the case in which two queens are on the same diagonal path. The `nqueenso` relation processes all positions of queens from the list `qs` and ensures that each queen is safe. The `solve-queens` function initializes `n` logical variables using the `clojure.core.logic/lvar` form.

The value `qs` is initialized a list of vector pairs that each contain a logical variable and a number within the range of  $0$  to  $n$ . In effect, we initialize the  $y$  coordinates of all pairs, and solve for the  $x$  coordinates. The reasoning behind this is that as we are solving for  $n$  queens on a board with  $n$  columns and  $n$  rows, and each row will have a queen placed on it.

The `solve-nqueens` function returns a list of solutions that each contain a list of coordinate pairs. We can print this data in a more intuitive representation by using the `partition` and `clojure.pprint/pprint`

functions, as shown in *Example 7.3*:

```
(defn print-nqueens-solution [solution n]
  (let [solution-set (set solution)
        positions (for [x (range n)
                        y (range n)]
                    (if (contains? solution-set [x y]) 1 0))]
    (binding [*print-right-margin* (* n n)]
      (clojure pprint/partition n positions)))))

(defn print-all-nqueens-solutions [solutions n]
  (dorun (for [i (-> solutions count range)
              :let [s (nth solutions i)]]
            (do
              (println (str "\nSolution " (inc i) ":"))
              (print-nqueens-solution s n)))))

(defn solve-and-print-nqueens [n]
  (-> (solve-nqueens n)
       (print-all-nqueens-solutions n)))
```

### *Example 7.3: The n-queens problem (continued)*

Now, we just need to call the `solve-and-print-nqueens` function by passing it the number of queens. Let's try to use this function to solve the n-queens problem for four queens, as shown here:

```
user> (solve-and-print-nqueens 4)
```

Solution 1:

```
((0 1 0 0)
 (0 0 0 1)
 (1 0 0 0)
 (0 0 1 0))
```

Solution 2:

```
((0 0 1 0)
 (1 0 0 0)
 (0 0 0 1)
 (0 1 0 0))
```

nil

The `solve-and-print-nqueens` function prints two solutions for four queens. Each solution is printed as a bunch of nested lists, in which each inner list represents a row on the chessboard. The value `1` indicates that a queen is placed on that position on the chessboard. As you can see, none of the four queens threaten each other in either of the two solutions.

In this way, the `solve-nqueens` function uses relations to solve the n-queens problem. We mentioned earlier that the n-queens problem originally involved eight queens. There are totally 92 distinct solutions for eight queens, and the `solve-nqueens` function can find every single one of them. You are encouraged to try this out by passing the value `8` to the `solve-and-print-nqueens` function and verifying the solutions it prints.

## Solving a Sudoku puzzle

Some of us may already be in love with the intuitive and mesmerizing Sudoku puzzles that we find in newspapers and magazines. This is another problem that involves logical rules. A Sudoku board is a  $9 \times 9$  grid on which we can place digits. The grid is divided into nine smaller grids, each of which is further divided into  $3 \times 3$  grids that contain digits. These smaller grids are also called *squares* or *boxes*. Some of the squares will be filled with boxes. The goal is to place digits on all positions on the grid such that each row, each column, and each of the smaller grids all contain distinct digits in the range 1 through 9.

Let's implement the rules of a Sudoku puzzle in this way. We will create a logical variable for every possible position of a digit on a Sudoku board and solve for their values using the rules of the puzzle. The initial values of the digits on a Sudoku board can be provided as a single vector comprising of 81 numbers. In this implementation, we introduce a couple of new constructs that are useful in concisely describing the rules of a Sudoku puzzle. The `everyg` function from the `clojure.core.logic` namespace can be used to apply a relation over a list of logical variables, thus ensuring that a relation is true for all the supplied logical variables. We must also

ensure that the logical variables in a row, column, and  $3 \times 3$  sized grid in a Sudoku puzzle are distinct. This can done by using the `clojure.core.logic.fd/distinct` function. An implementation of this design of a Sudoku solver is shown in *Example 7.4*.

## Note

The following examples can be found in `src/m_clj/c7/sudoku.clj` of the book's source code.

```
(l/defn init-sudoku-board [vars puzzle]
  ([[[] []])
  ([[_ . vs] [0 . ps]] (init-sudoku-board vs ps))
  ([[_ . vs] [n . ps]] (init-sudoku-board vs ps)))

(defn solve-sudoku [puzzle]
  (let [board (repeatedly 81 l/lvar)
        rows (into [] (map vec (partition 9 board)))
        cols (apply map vector rows)
        val-range (range 1 10)
        in-range (fn [x]
                   (fd/in x (apply fd/domain val-range)))
        get-square (fn [x y]
                     (for [x (range x (+ x 3))
                           y (range y (+ y 3))]
                       (get-in rows [x y])))
        squares (for [x (range 0 9 3)
                      y (range 0 9 3)]
                  (get-square x y))]
    (l/run* [q]
      (l/== q board)
      (l/everyg in-range board)
      (init-sudoku-board board puzzle)
      (l/everyg fd/distinct rows)
      (l/everyg fd/distinct cols)
      (l/everyg fd/distinct squares))))
```

### *Example 7.4: A Sudoku solver*

In *Example 7.4*, the `init-sudoku-board` relation initializes the logical variables `vars` from the `puzzle` `puzzle`, and the `solve-sudoku` function

finds all possible solutions of the given puzzle. The `solve-sudoku` function creates the logical variables through a composition of the `repeatedly` and `clojure.core.logic/lvar` forms. These variables are then partitioned into rows, columns, and squares, represented by the variables `rows`, `cols`, and `squares` respectively. The `solve-sudoku` function then initializes the logical variables using the `init-sudoku-board` form, and uses a composition of the `everyg` and `distinct` forms to ensure that the rows, columns, and squares of a solution contain distinct values. All the logical variables are also bound to the range 1 through 9 using the internally defined `in-range` function.

The `solve-sudoku` function defined in *Example 7.4* takes a vector of values representing the initial state of a Sudoku board as an argument and returns a list of solutions in which each solution is a vector. As a plain vector isn't really an intuitive representation of a Sudoku board, let's define a simple function to find all solutions of a given puzzle and print the solutions, as shown in *Example 7.5*:

```
(defn solve-and-print-sudoku [puzzle]
  (let [solutions (solve-sudoku puzzle)]
    (dorun (for [i (-> solutions count range)
                :let [s (nth solutions i)]]
              (do
                (println (str "\nSolution " (inc i) ":"))
                (clojure.pprint/pprint
                  (partition 9 s)))))))
```

### *Example 7.5: A Sudoku solver (continued)*

The `solve-and-print-sudoku` function in *Example 7.5* calls the `solve-sudoku` function to determine all possible solutions to a given Sudoku puzzle and prints the results using the `partition` and `clojure.pprint/pprint` functions. Now, let's define a simple Sudoku puzzle to solve, as shown in *Example 7.6*.

```
(def puzzle-1
  [0 9 0 0 0 0 0 5 0]
```

```
6 0 0 0 5 0 0 0 2
1 0 0 8 0 4 0 0 6
0 7 0 0 8 0 0 3 0
8 0 3 0 0 0 2 0 9
0 5 0 0 3 0 0 7 0
7 0 0 3 0 2 0 0 5
3 0 0 0 6 0 0 0 7
0 1 0 0 0 0 4 0])
```

### *Example 7.6: A Sudoku solver (continued)*

Now, let's pass the vector `puzzle-1` to the `solve-and-print-sudoku` function to print all possible solutions to it, as shown here:

```
user> (solve-and-print-sudoku puzzle-1)
```

Solution 1:

```
((4 9 8 6 2 3 7 5 1)
 (6 3 7 9 5 1 4 8 2)
 (1 2 5 8 7 4 3 9 6)
 (9 7 1 2 8 6 5 3 4)
 (8 4 3 5 1 7 2 6 9)
 (2 5 6 4 3 9 1 7 8)
 (7 6 9 3 4 2 8 1 5)
 (3 8 4 1 6 5 9 2 7)
 (5 1 2 7 9 8 6 4 3))
```

nil

The `solve-sudoku` function finds a single solution to the Sudoku puzzle represented by the vector `puzzle-1` as shown previously. The puzzle represented by `puzzle-1` and its solution are shown on a Sudoku board in the following illustration:

The diagram illustrates a Sudoku solving process. On the left, there is an initial 9x9 Sudoku grid with some numbers filled in. An arrow points from this grid to a completed 9x9 grid on the right, representing the solution.

	9				5			
6			5			2		
1			8	4		6		
	7		8		3			
8	3			2	9			
5			3		7			
7			3	2		5		
3			6			7		
1					4			

4	9	8	6	2	3	7	5	1
6	3	7	9	5	1	4	8	2
1	2	5	8	7	4	3	9	6
9	7	1	2	8	6	5	3	4
8	4	3	5	1	7	2	6	9
2	5	6	4	3	9	1	7	8
7	6	9	3	4	2	8	1	5
3	8	4	1	6	5	9	2	7
5	1	2	7	9	8	6	4	3

### *Example 7.7: A Sudoku solver (continued)*

It is very likely that a Sudoku puzzle has multiple solutions. For example, the Sudoku puzzle represented by `puzzle-2` in *Example 7.7* has eight distinct solutions. You're more than welcome to find the solutions to this puzzle using the `solve-and-print-sudoku` function:

```
(def puzzle-2
  [0 8 0 0 0 9 7 4 3
   0 5 0 0 0 8 0 1 0
   0 1 0 0 0 0 0 0 0
   8 0 0 0 0 5 0 0 0
   0 0 0 8 0 4 0 0 0
   0 0 0 3 0 0 0 0 6
   0 0 0 0 0 0 0 7 0
   0 3 0 5 0 0 0 8 0
   9 7 2 4 0 0 0 5 0])
```

### *Example 7.7: A Sudoku solver (continued)*

In conclusion, we can implement the rules of a Sudoku puzzle as logical relations using the `core.logic` library.

# Summary

In this chapter, we looked at how Clojure can be used for logic programming. We introduced the `core.logic` library by exploring the various constructs provided by this library. We also studied how we can implement solutions to the n-queens problem and a Sudoku puzzle using the `core.logic` library.

In the following chapter, we will get back on our journey through functional programming and talk about handling asynchronous tasks in Clojure.

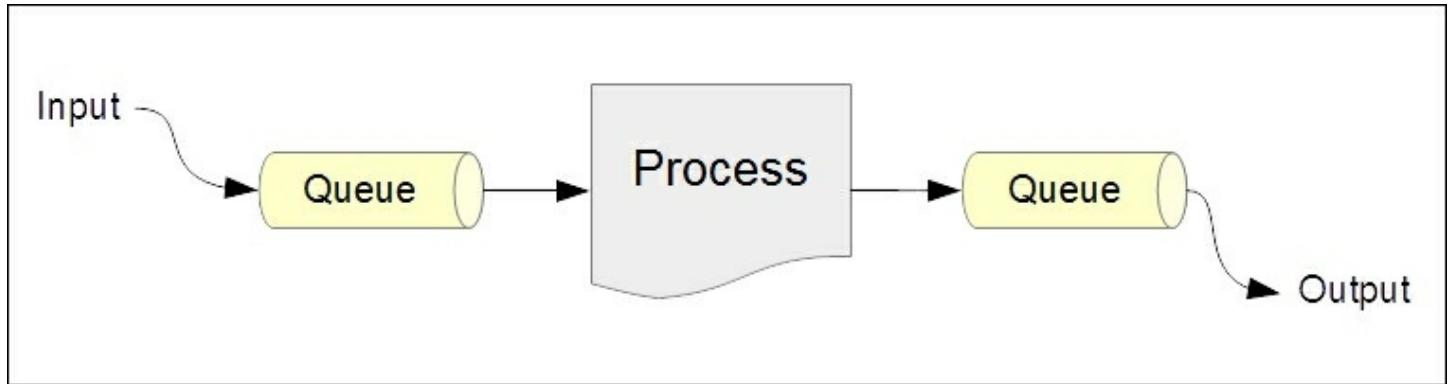
# Chapter 8. Leveraging Asynchronous Tasks

The term **asynchronous programming** refers to defining tasks that are executed *asynchronously* on different threads of execution. While this is similar to multithreading, there are a few subtle differences. Firstly, a thread or a future will remain allocated to a single operating system thread until completion. This leads to the fact that there can only be a limited number of futures that can be executed concurrently, depending on the number of processing cores available. On the other hand, asynchronous tasks are scheduled for execution on threads from a thread pool. This way, a program can have thousands, or even millions of asynchronous tasks running concurrently. An asynchronous task can be suspended at any time, or *parked*, and the underlying thread of execution can be reallocated to another task. Asynchronous programming constructs also allow the definition of an asynchronous task to look like a sequence of synchronous calls, but each call could potentially be executed asynchronously.

In this chapter, we will explore various libraries and constructs that can be used for asynchronous programming in Clojure. First off, we will take a look at *processes* and *channels* in the `core.async` library for asynchronous programming. Later, we will explore *actors* from the *Pulsar* library. Processes and channels are constructs similar to *go-routines* in the Go programming language. On the other hand, actors were first popularized in the Erlang programming language. All of these techniques are different ways of structuring code that executes asynchronously. We must understand that the theory behind these concepts isn't really novel, and more implementations of these theories have been springing up since the rise of distributed and multi-core architectures. With that in mind, let's start off on our journey into asynchronous programming.

## Using channels

The `core.async` library (<https://github.com/clojure/core.async>) facilitates asynchronous programming in Clojure. Through this library, we can use asynchronous constructs that run on both the JVM and web browsers without dealing with how they are scheduled for execution on low-level threads. This library is an implementation of the theory in the paper **Communicating Sequential Processes (CSPs)**, originally published in the late '70s by C. A. R. Hoare. The bottom line of CSPs is that any system that processes some input and provides an output can be comprised of smaller subsystems, and each subsystem can be defined in terms of *processes* and *queues*. A queue simply buffers data, and a process can read from and write to several queues. Here, we shouldn't confuse the term *process* with an operating system process. In the context of CSPs, a process is simply a sequence of instructions that interacts with some data stored in queues. Several processes may exist in a given system and queues are a means of conveying data between them. A process that takes data from a single queue and outputs data to another queue can be illustrated as follows:



As shown in the preceding diagram, input data goes into a queue, a process manipulates this data through the queue, and finally, writes the output data to another queue. The `core.async` library essentially provides first-class support for creating processes and queues. Queues are dubbed as **channels** in the `core.async` library, and can be created using the `chan` function. Processes can be created using `go` and `thread` macros. Let's dive a bit

deeper into the details.

## Note

The following library dependencies are required for the upcoming examples:

```
[org.clojure/core.async "0.1.346.0-17112a-alpha"]
```

Also, the following namespaces must be included in your namespace declaration:

```
(ns my-namespace
  (:require [clojure.core.async :as a]))
```

Processes, created using the `thread` and `go` forms, are scheduled for execution on thread pools. In fact, we can create several thousands of these processes in a program as they do not require their own dedicated thread. On the other hand, creating a large number of threads or futures would result in the excessive jobs being queued for execution. This imposes a practical limit on the number of threads or futures we can run concurrently. Hence, the `core.async` library, and CSPs in general, allows us to model a system as a large number of lightweight and concurrent processes.

Channels can be thought of as data structures for managing the state between processes. The `chan` function from the `core.async` namespace returns a channel that can be read from and written to by several concurrent processes. Channels are *unbuffered* by default, which means a write operation to a channel will not complete until a read operation is invoked on it concurrently and vice versa. We can also create a *buffered* channel by specifying a number to the `chan` function to indicate the buffer size. A buffered channel will allow a certain number of values to be written to it without blocking, and the buffered values can then be read. A channel can be closed using the `close!` function from the `core.async` namespace.

We can also create a buffered channel by passing a buffer object to the `chan` function. A buffer object can be created using the `buffer`, `dropping-buffer`, or `sliding-buffer` functions, and these functions require a number, indicating the size of the buffer, as an argument. Either of the expressions `(a/chan (a/buffer n))` or `(a/chan n)` can be used to create a channel that can buffer `n` values, and the channel will block write operations to it once it is filled with `n` values. The `dropping-buffer` function creates a buffer that drops newly added values once it's full. Conversely, a buffer created using the `sliding-buffer` function will drop the oldest values added to it once it is completely filled.

The `core.async` library provides a handful of constructs for reading from and writing to channels, and the values passed to and returned by these constructs conform to a few simple rules. A read operation on a channel returns a value from the channel, or `nil` if the channel is closed. A write operation will return `true` if it succeeds, or `false` if the channel is closed and the write operation couldn't be completed. We can read the buffered data from a closed channel, but once the channel is exhausted of data, a read operation on it will return `nil`. The arguments to the read and write operations for channels conform to the following pattern:

1. The first argument to any operation is a channel.
2. A write operation must be passed a value to put onto a channel in addition to the channel itself.

At this point, we should note that in the context of channels, the terms "write" and "put" can be used interchangeably, and similarly, the terms "read" and "take" refer to the same operation. The `take!` and `put!` functions take data and put data onto a queue. Both these functions return immediately, and can be passed a callback function as an argument in addition to the usual parameters. Similarly, the `<!!` and `>!!` functions can be used to read from and write to a channel, respectively. However, the `<!!` operation can block the calling thread if there is no data in the supplied channel and the `>!!` operation will be blocked if there is no more buffer

space available in a given channel. These two operations are meant to be used within a `thread` form. Finally, the parking read and write functions, namely `<!`` and `>!``, can be used within a `go` form to interact with a channel. Both the `<!`` and `>!`` operations will park the state of the task and release the underlying thread of execution if an operation cannot be completed immediately.

Let's move on to the details of creating processes using the `core.async` library. The `core.async/thread` macro is used to create a single-threaded process. It is similar to the `future` form in the sense that the body of a `thread` form is executed on a new thread and a call to a `thread` form returns immediately. A `thread` form returns a channel from which the output of its body can be read. This makes the `thread` form a bit more convenient than the standard `future` form for interacting with channels, and is thus preferred over a `future` form. The `<!!` and `>!!` functions can be used within a `thread` form to interact with a channel.

To create an asynchronous process that can be parked and scheduled for execution, we must use the `go` macro from the `core.async` namespace. Similar to the `thread` form, it returns a channel from which the output of its body can be read. All channel operations within the body of the `go` form will park, rather than blocking the underlying thread of execution. This implies that the executing thread will not be blocked and can be reallocated to another asynchronous process. Thus, the execution of a number of `go` forms can be interleaved over a much lesser number of actual threads. We must ensure that no thread-specific operations, such as `Thread/sleep`, are made within a `go` form, as such operations affect the underlying thread of execution. Within a `go` form, we must always use the `<!`` and `>!`` parking forms to read from and write to a channel.

## Note

Visit <https://clojure.github.io/core.async/> for the complete documentation on all the functions and macros in the `core.async` library.

The `go-loop` macro is an asynchronous version of the `loop` form, and accepts a vector of bindings as its first argument, followed by any number of forms that must be executed. The body of a `go-loop` form will be internally executed within a `go` form. The `go-loop` construct is often used to create asynchronous event loops that have their own localized state. As an example, let's consider the simple `wait-and-print` function that sets off a process that reads from a given channel, as shown in *Example 8.1*.

## Note

The following examples can be found in `src/m_clj/c8/async.clj` of the book's source code.

```
(defn wait-and-print [c]
  (a/go-loop [n 1]
    (let [v (a/<! c)]
      (when v
        (println (str "Got a message: " v))
        (println (str "Got " n " messages so far!"))
        (recur (inc n))))))
  (println "Waiting..."))
```

*Example 8.1: A function that asynchronously reads from a channel*

The `wait-and-print` function shown previously will repeatedly read from the channel `c` passed to it. The `when` form is used to check if the value read from channel, represented by `v`, is not `nil`, since `nil` could be returned from the `<!` form if the channel `c` is closed. The `go-loop` form in the previous example also counts the number of values read from the channel using the variable `n`. On receiving a value from the channel, some information is printed and the body is looped over using a `recur` form. We can create a channel, pass it to the `wait-and-print` function and observe the output of sending values to the channel, as shown here:

```
user> (def c (a-chan))
#'user/c
user> (wait-and-print c)
Waiting...
```

```
nil
user> (a/>!! c :foo)
true
Got a message: :foo
Got 1 messages so far!
user> (a/>!! c :bar)
true
Got a message: :bar
Got 2 messages so far!
```

As shown previously, a call to the `wait-and-print` function starts an asynchronous event loop that reads from the channel `c`. On sending a value to the channel `c` using a `>!!` form, the value gets printed along with a total count of values sent to the channel. Also, calls to the `>!!` form return the value `true` immediately. Now, let's see what happens when we close the channel `c` using the `close!` function, shown as follows:

```
user> (a/close! c)
nil
user> (a/>!! c :foo)
false
```

After closing the channel `c`, the `>!!` form returns `false` when it is applied to the channel, which implies that the channel `c` doesn't allow any more values to be put into it. Also, nothing gets printed, which means that the asynchronous routine that was trying to take values from the channel `c` has terminated.

Another way to send values into a channel is by using the `onto-chan` function from the `core.async` namespace. This function must be passed a channel and a collection of values to put *onto* the channel, as shown here:

```
user> (def c (a/chanc 4))
#'user/c
user> (a/onto-chan c (range 4))
#<ManyToManyChannel@0x86f03a>
user> (repeatedly 4 #(-> c a/<!!))
(0 1 2 3)
```

The `onto-chan` function will close the channel it has been passed once the supplied collection of values is entirely put onto the channel. To avoid closing the channel, we can specify `false` as an additional argument to the `onto-chan` function.

The `alts!` and `alts!!` functions from the `core.async` namespace can be used to wait for completion of one of several channel operations. The main distinction between these functions is that the `alts!` function is intended for use within a `go` form and will park the current thread, unlike the `alts!!` function that blocks the current thread and must be used in a `thread` form. Both these functions must be passed a vector of channels and return a vector of two elements. The first element in the returned vector represents the value for a take operation or a Boolean value for a put operation, and the second one indicates the channel on which the operation completed. We can also specify a default value as a keyword argument with the key `:default` to the `alts!` and `alts!!` functions. The default value will be returned if none of the operations supplied to the `alts!` or `alts!!` forms have completed.

The `core.async` library provides two versatile macros, namely `alt!` and `alt!!`, to wait for one among several channel operations to be complete. As you may have already guessed, an `alt!` form parks the current task, and an `alt!!` form blocks the current thread. Both these forms can also return a default value when used with the `:default` keyword argument. We can pass several clauses to the `alt!` and `alt!!` forms for reading from and writing to several channels. The `alt!` form in *Example 8.2* describes the clauses supported by the `alt!` and `alt!!` macros:

```
(defn process-channels [c0 c1 c2 c3 c4 c5]
  (a/go
    (a/alt!
      ;; read from c0, c1, c2, c3
      c0 :r
      c1 ([v] (str v))
      [c2 c3] ([v c] (str v)))
      ;; write to c4, c5
```

```
[ [c4 :v1] [c5 :v2]] :w) ))
```

*Example 8.2: An asynchronous process implemented using the alt! form*

The preceding `process-channels` function takes six channels as its arguments, and uses an `alt!` form within a `go` form to perform asynchronous operations on these channels. The channels `c0`, `c1`, `c2`, and `c3` are read, and the channels `c4` and `c5` are written to. The `alt!` form tries to read from the channel `c0` and returns the keyword `:r` if the operation completes first. The channel `c1` is also read from, but the right hand side of its clause contains a parameterized expression with the argument `v`, where `v` is the value read from the channel. The channels `c2` and `c3` are passed as a vector in one of the clauses of the `alt!` form shown previously, and this clause uses a parameterized expression with the arguments `v` and `c`, where `c` is the channel on which the read operation completed first and `v` is the value read from the channel. Write operations are specified in an `alt!` form as a nested vector, where each inner vector contains a channel and a value to put onto the channel. The channels `c4` and `c5` are written to in the previous `alt!` form, and the value `:w` is returned if either of the two write operations completes. In this way, we can specify clauses to the `alt!` and `alt!!` forms to read to and write from several channels, and return a value based on which channel operation completes first.

Another facility that is often required in asynchronous programming is the ability to specify a *timeout* with a given operation. By the term *timeout*, we mean a specified amount of time after which the current operation is aborted. The `core.async` has an intuitive method for specifying operations with timeouts. This is done using the `core.async/timeout` function, which must be supplied a time interval in milliseconds and returns a channel that closes after the specified amount of time. If we intend to perform an operation with a timeout, we use one of the `alt*` forms with a channel returned by the `timeout` function.

This way, an operation with a channel returned by a `timeout` form will

surely complete after the specified amount of time. The `timeout` form is also useful in parking or blocking the current thread of execution for a given amount of time. For example, a blocking read operation from a channel returned by a `timeout` form will block the current thread for the specified time interval, as shown here:

```
user> (time (a/<!! (a/timeout 1000)))
"Elapsed time: 1029.502223 msecs"
nil
```

We have now covered the basics of processes and channels in the `core.async` library.

## Customizing channels

Channels can also be programmed to modify or compute values from those put into them. A read operation, for instance, on a channel could invoke a computation using the values buffered in the same channel, or even other channels. The `reduce` function from the `core.async` namespace can be used to compute values from channels and has more-or-less the same semantics as that of the standard `reduce` function. This variant of the `reduce` function requires a reducing operation, an initial value for the reduction operation and a channel to be passed to it, and it will return a channel from which the result can be read. Also, this function only produces values once the channel passed to it is closed. For example, consider the following code that computes a string from the values in a channel using the `core.async/reduce` function:

```
user> (def c (a/chanc 5))
#'user/c
user> (a/onto-chan c (range 5))
#<ManyToManyChannel@0x4adadd>
user> (def rc (a/reduce #(str %1 %2 " ") "" c))
#'user/rc
user> (a/<!! rc)
"0 1 2 3 4 "
```

In the preceding example, the sequence generated by the expression `(range 5)` is put onto the channel `c` using an `onto-chan` form, and values from the channel are computed over using the channel-based variant of the `reduce` function. A single value is read from the resulting channel `rc`, thus producing a string containing all the values from the channel `c`. Note that the `reduce` form in this example produced a result without explicitly calling the `close!` function, as the `onto-chan` function closes the supplied channel after it completes putting values onto it.

A more powerful and intuitive way to compute values from a channel is by using a transducer. We have already discussed transducers in some detail in [Chapter 5, Composing Transducers](#), and we will now have a look at how transducers can be used with channels. Essentially, a channel can be associated with a transducer by specifying the transducer as a second argument to the `core.async/chan` function. Let's consider the simple transducer `xform` shown in *Example 8.3*.

```
(def xform
  (comp
    (map inc)
    (map #(* % 2))))
```

### *Example 8.3: A simple transducer to use with a channel*

The transducer `xform` shown is a trivial composition of mapping the functions `inc` and `#(* % 2)`. It will simply increment all values in a source of data, or rather a channel, and then double all of the results from the previous step. Let's create a channel using this transducer and observe its behavior, as shown here:

```
user> (def xc (a/chan 10 xform))
#'user/xc
user> (a/onto-chan xc (range 10) false)
#<ManyToManyChannel@0x17d6a37>
user> (repeatedly 10 #(-> xc a/<!!))
(2 4 6 8 10 12 14 16 18 20)
```

The channel `xc` will apply the transducer `xform` to each value contained in it. The result of repeatedly taking values from the channel `xc` is thus a sequence of even numbers, which is produced by applying the functions `inc` and `#(* % 2)` to each number in the range `(range 10)`. Note that the `onto-chan` form in the previous example does not close the channel `xc` as we pass `false` as its last argument.

A transducer associated with a channel could encounter an exception. To handle errors, we can pass a function as an additional argument to the `chan` form. This function must take exactly one argument, and will be passed any exception that is encountered by a transducer while transforming the values in a channel. For example, the expression `(a/chan 10 xform ex-handler)` creates a channel with a transducer `xform` and an exception handler `ex-handler`.

In this way, the `core.async/reduce` form and transducers can be used to perform computations on the values contained in channels.

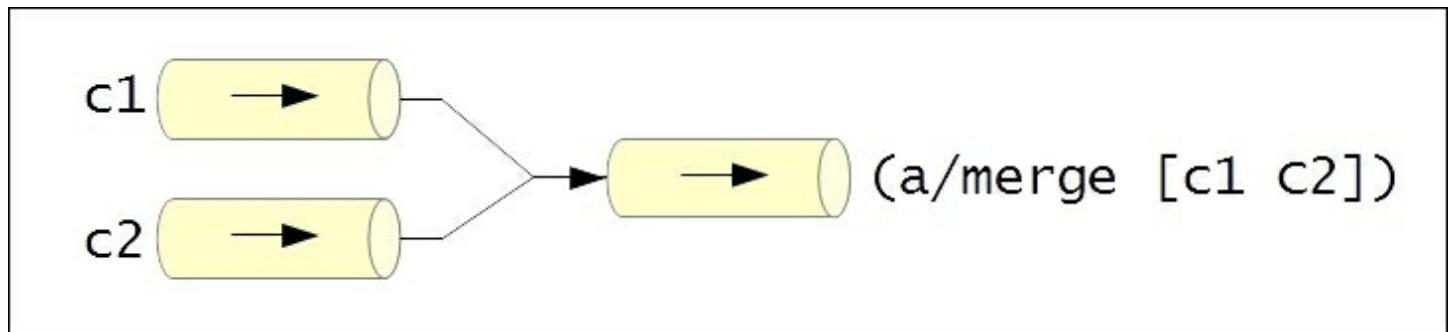
## Connecting channels

Now that we are familiar with the basics of channels and processes in the `core.async` library, let's explore the different ways in which channels can be connected together. Connecting two or more channels is useful for aggregating and distributing data among them. A connection between two or more channels is called a *joint fitting*, or simply a *joint*. We will use diagrams to describe some of the more complex joint fittings in this section. Keep in mind that the arrows in these diagrams indicate the direction of the flow of data in a given channel.

The simplest way to connect two channels is by using a *pipe*, which is implemented by the `core.async/pipe` function. This function will take values from the first channel provided to it, and supplies these values to the second channel passed to it. In this way, a pipe between channels is similar to UNIX-style pipes between streams. For example, the expression

(`a/pipe from to`) will take values from the channel `from` and put them onto the channel `to`. The `pipe` function also takes an optional third argument, which indicates whether the destination channel will be closed when the source channel closes, and this argument defaults to `true`. We can also connect two channels using a *pipeline*, using the `pipeline` function from the `core.async` namespace. The `pipeline` function will essentially apply a transducer to the values in a channel before they are put into another channel. The supplied transducer will also be invoked in parallel for each element in the supplied channel by the `pipeline` function.

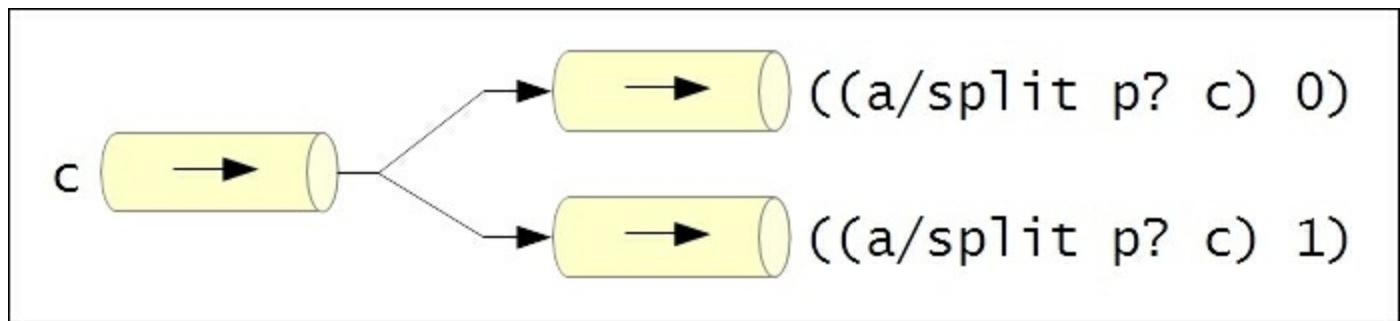
The `merge` function from the `core.async` namespace can be used to combine several channels. This function must be passed a vector of channels, and returns a channel from which the values from all of the supplied channels can be read. The returned channel is unbuffered by default, and we specify the size of the channel's buffer by passing a number as an additional argument to the `merge` function. Also, the channel returned by a `merge` form will be closed once all the source channels have been closed. The operation of the `merge` function with two channels can be illustrated as follows:



A channel can be split into two channels using the `core.async/split` function. The `split` function must be passed a predicate `p?` and a channel `c`, and returns a vector of two channels. The predicate `p?` is used to decide the channel on which a value from the channel `c` must be put. All values from the channel `c` that return a truthy value when passed to the predicate

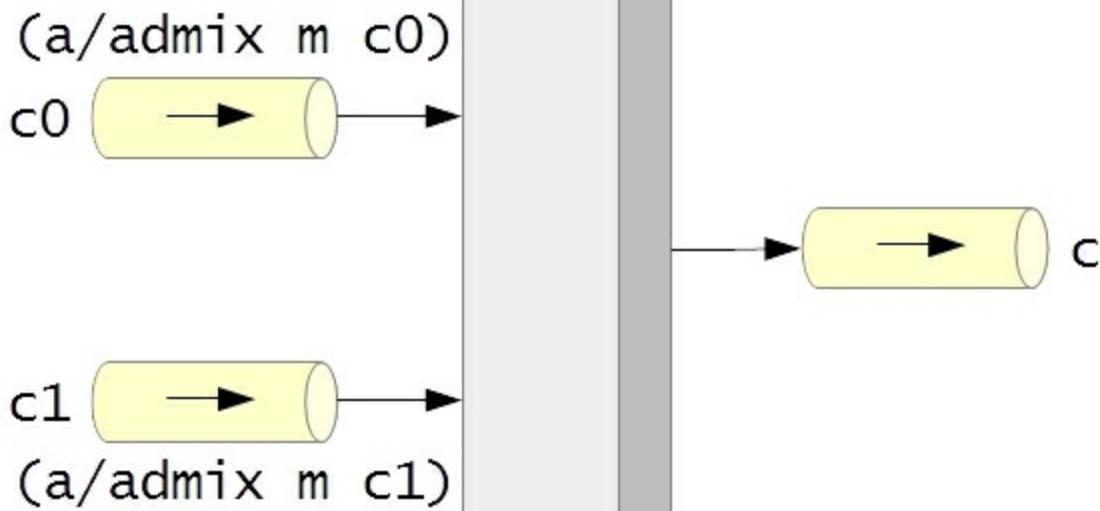
`p?` will be put onto the first channel in the vector returned by the `split` function.

Conversely, the second channel in the returned vector will contain all values that return `false` or `nil` when `p?` is applied to these values. Both the channels returned by this function will be unbuffered by default, and the buffer size of both these channels can be specified as additional arguments to a `split` form. The `split` function can be depicted by the following illustration:



A more dynamic way to combine several channels, compared to the `merge` function, is by using the `mix`, `admix`, and `unmix` functions from the `core.async` namespace. The `mix` function creates a *mix*, to which channels with incoming data can be connected to using the `admix` function. The `mix` function takes a channel as an argument, and the supplied channel will contain values from all the source channels added by the `admix` function. A source channel can be removed from a mixer by using the `unmix` function. The `admix` and `unmix` functions both accept a *mix*, which is returned by the `mix` function, and a source channel as arguments. To remove all channels from a *mix*, we simply pass the *mix* as an argument to the `unmix-all` function. The gist of a *mix* is that it allows us to dynamically add and remove source channels that send data to a given output channel. A *mix*, its output channel, and source channels can be illustrated as follows:

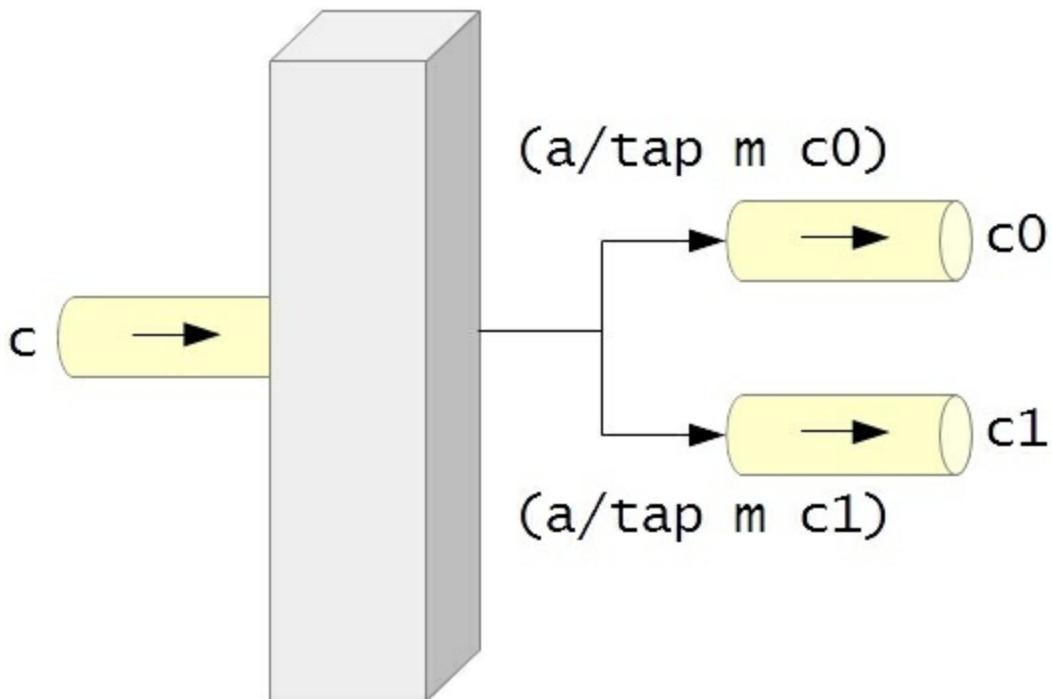
```
(def m (a/mix c))
```



In the preceding illustration, the channel `c` is used as the output channel of the `mix m`, and the channels `c0` and `c1` are added as source channels to the `mix m` using the `admix` function.

The `core.async/mult` function creates a *multiple* of a given channel. The data from a multiple can be *tapped into* from another channel using the `tap` function. The channel supplied to the `tap` function will receive copies of all data sent to the source channel of a multiple. The `untap` function is used to disconnect a channel from a multiple, and the `untap-all` function will disconnect all channels from a multiple. A multiple essentially allows us to dynamically add and remove output channels that read values from a given source channel. The `mult` and `tap` functions can be described by the following diagram:

```
(def m (a/mult c))
```

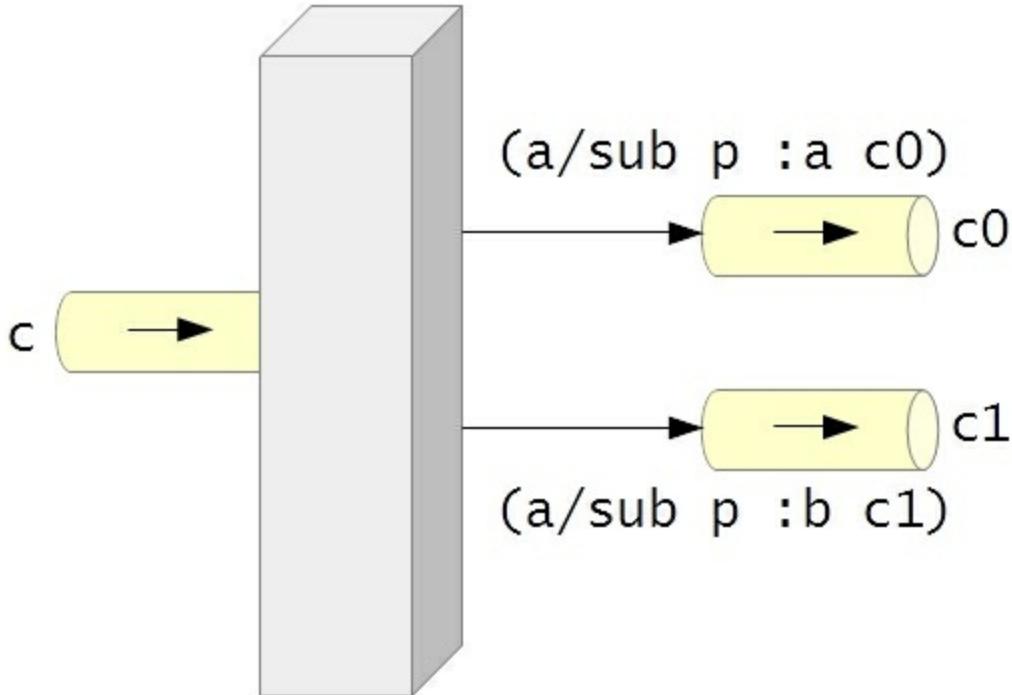


In the preceding illustration, the channel `c` is used as the source channel by the multiple `m`, and the channels `c0` and `c1` are passed to the `tap` function so that they effectively receive copies of the data sent to the channel `c`.

The `core.async` library also supports a *publish-subscribe* model of transferring data. This can be done using a *publication*, which is created using the `core.async/pub` function. This function must be supplied a source channel and a function to decide the topic of a given value in the publication. Here, a topic can be any literal value, such as a string or a keyword, which is returned by the function supplied to the `pub` form. Channels can subscribe to a publication and a topic via the `sub` function, and a channel can unsubscribe from a publication using the `unsub` function. The `sub` and `unsub` functions must be passed a publication, a topic value and a channel. Also, the `unsub-all` function can be used to disconnect all

channels that have subscribed to a publication. This function can optionally be passed a topic value, and will disconnect all channels that have subscribed to the given topic. A publication with two channels subscribed to it is depicted in following diagram:

```
(def p (a/pub c topic-fn))
```



In the preceding illustration, the publication `p` is created using the channel `c` and the function `topic-fn`. The channel `c0` subscribes to the publication `p` and the topic `:a`, while the channel `c1` subscribes to the same publication but for the topic `:b`. When a value is received on the channel `c`, it will either be sent to the channel `c0` if the function `topic-fn` returns `:a` for the given value, or to channel `c1` if the function `topic-fn` returns `:b` for the given value. Note that the values `:a` and `:b` in the preceding diagram are just arbitrary literals, and we could have used any other literal values just as easily.

In summary, the `core.async` library provides several constructs to create joints between channels. These constructs help in modelling different ways in which data flows from any number of source channels into any number of output channels.

## Revisiting the dining philosophers problem

Now, let's try to implement a solution to the **dining philosophers problem** using the `core.async` library. We have already implemented two solutions to the dining philosophers problem in [Chapter 2, Orchestrating Concurrency and Parallelism](#) of which one solution used refs and the other one used agents. In this section, we will use channels to implement a solution to the dining philosophers problem.

The dining philosophers problem can be concisely described as follows. Five philosophers are seated on a table with five forks placed between them. Each philosopher requires two forks to eat. The philosophers must somehow share access to the forks placed in between them to consume the food allocated to them, and none of the philosophers must starve due to being unable to acquire two forks. In this implementation, we will use channels to maintain the state of the forks as well as the philosophers on the table.

### Note

The following namespaces must be included in your namespace declaration for the upcoming examples:

```
(ns my-namespace
  (:require [clojure.core.async :as a]
            [m-clj.c2.refs :as c2]))
```

The following examples can be found in `src/m_clj/c8/dining_philosophers_async.clj` of the book's source code. Some of these examples are based on code from *A Dining Philosophers solver* by Pepijn de Vos (<http://pepijndevos.nl/2013/07/11/dining-philosophers-in->

[coreeasy.html](#)).

Let's first define a couple of functions to initialize all the philosophers and forks we are dealing with, as shown in *Example 8.4*:

```
(defn make-philosopher [name forks food]
  {:name name
   :forks forks
   :food food})

(defn make-forks [nf]
  (let [forks (repeatedly nf #(a-chan 1))]
    (doseq [f forks]
      (a/>! ! f :fork))
    forks))
```

#### *Example 8.4: The dining philosophers problem*

The `make-philosopher` function defined in *Example 8.4* creates a map representing the state of a philosopher. The argument `name` will be a string, the argument `forks` will be a vector of two fork channels, and the argument `food` will be a number indicating the amount of food served to a philosopher. The two forks represent the forks on the left- and right-hand side of a philosopher. These forks will be allocated and passed to the `make-philosopher` function by the `init-philosophers` function that we previously defined in [Chapter 2](#), Orchestrating Concurrency and Parallelism. The `make-forks` function shown previously creates a specified number of channels, puts the value `:fork` onto each of them, and finally returns the new channels.

Next, let's define the routine of a philosopher as a process. A philosopher must try to acquire the forks on his left and right side, eat his food if he acquires both forks, and finally release any forks that he successfully acquired. Also, since the state of all the philosophers in our simulation is captured in a channel, we will have to take a philosopher out of a channel, perform the routine of a philosopher, and then put the philosopher's state back onto the channel. This routine is implemented by the `philosopher-`

process function in *Example 8.5*:

```
(defn philosopher-process [p-chan max-eat-ms max-think-ms]
  (a/go-loop []
    (let [p (a/<! p-chan)
          food (:food p)
          fork-1 ((:forks p) 0)
          fork-2 ((:forks p) 1)
          ;; take forks
          fork-1-result (a/alt!
                          (a/timeout 100) :timeout
                          fork-1 :fork-1)
          fork-2-result (a/alt!
                          (a/timeout 100) :timeout
                          fork-2 :fork-2)]
      (if (and (= fork-1-result :fork-1)
                (= fork-2-result :fork-2))
          (do
            ;; eat
            (a/<! (a/timeout (rand-int max-eat-ms)))
            ;; put down both acquired forks
            (a/>! fork-1 :fork)
            (a/>! fork-2 :fork)
            ;; think
            (a/<! (a/timeout (rand-int max-think-ms)))
            (a/>! p-chan (assoc p :food (dec food))))
          (do
            ;; put down any acquired forks
            (if (= fork-1-result :fork-1)
                (a/>! fork-1 :fork))
            (if (= fork-2-result :fork-2)
                (a/>! fork-2 :fork))
            (a/>! p-chan p)))
        ;; recur
        (when (pos? (dec food)) (recur))))
```

*Example 8.5: The dining philosophers problem (continued)*

The preceding `philosopher-process` function starts an asynchronous process using the `go-loop` macro. The arguments `p-chan`, `max-eat-ms`, and `max-think-ms` represent the channel containing the state of all

philosophers, the maximum amount of time a philosopher can spend eating, and the maximum amount of time a philosopher can think, respectively. The asynchronous task started by the `philosopher-process` function will try to take values from the forks `fork-1` and `fork-2` of a philosopher with a timeout of 100 milliseconds. This is done using a combination of the `alt!` and `timeout` functions. If a philosopher is able to acquire two forks, he will eat for some time, put down or release both forks, spend some time thinking, and repeat the same process. If he is unable to get two forks, the philosopher will release any acquired forks and restart the same process. The state of the philosopher is always put back onto the channel `p-chan`. This asynchronous process is repeated until a philosopher has any remaining food. Next, let's define a couple of functions to start and print the philosophers in our simulation, as shown in *Example 8.6*:

```
(defn start-philosophers [p-chan philosophers]
  (a/onto-chan p-chan philosophers false)
  (dorun (repeatedly (count philosophers)
    #(philosopher-process p-chan 100 100)))))

(defn print-philosophers [p-chan n]
  (let [philosophers (repeatedly n #(a/<!! p-chan) )]
    (doseq [p philosophers]
      (println (str (:name p) ":\t food=" (:food p)))
      (a/>!! p-chan p))))
```

### *Example 8.6: The dining philosophers problem (continued)*

The preceding `start-philosophers` function will put a sequence of philosophers, represented by the argument `philosophers`, onto the channel `p-chan`, and then call the `philosopher-process` function for each philosopher in the sequence `philosophers`. The `print-philosophers` function uses the blocking channel read and write functions, namely `<!!` and `>!!`, to read `n` philosophers from the channel `p-chan` and print the amount of food remaining on each philosopher's plate.

Finally, let's create some instances of `philosophers` and associated forks by

using the `make-philosopher` and `make-forks` functions. We will also use the `init-philosophers` function from [Chapter 2](#), *Orchestrating Concurrency and Parallelism*, to create philosopher objects, using the `make-philosopher` function, and assign two forks to each philosopher. These top-level definitions of the philosophers and forks in our simulation are shown in *Example 8.7*.

```
(def all-forks (make-forks 5))  
(def all-philosophers  
  (c2/init-philosophers 5 1000 all-forks make-philosopher))  
  
(def philosopher-chan (a/chan 5))
```

### *Example 8.7: The dining philosophers problem (continued)*

As shown here, we define five forks and philosophers, and create a channel to represent the state of all philosophers we have created. Note that the channel we use for the philosophers has a buffer size of 5. The simulation can be started by calling the `start-philosophers` function, and the state of the philosophers can be printed using the `print-philosophers` function, as shown here:

```
user> (start-philosophers philosopher-chan all-philosophers)  
nil  
user> (print-philosophers philosopher-chan 5)  
Philosopher 3: food=937  
Philosopher 2: food=938  
Philosopher 1: food=938  
Philosopher 5: food=938  
Philosopher 4: food=937  
nil  
user> (print-philosophers philosopher-chan 5)  
Philosopher 4: food=729  
Philosopher 1: food=729  
Philosopher 2: food=729  
Philosopher 5: food=730  
Philosopher 3: food=728  
nil
```

As the preceding output shows us, the five philosophers share access to the forks among themselves and consume their food at the same rate. All philosophers get a chance to eat their food, and thus no one starves. Note that the order of the philosophers printed by the `print-philosophers` function may differ from time to time, and some philosophers may also be printed twice by this function.

In this way, we can solve a given problem using channels and processes from the `core.async` library. Also, we can create any number of such processes without bothering about the available number of operating system level threads.

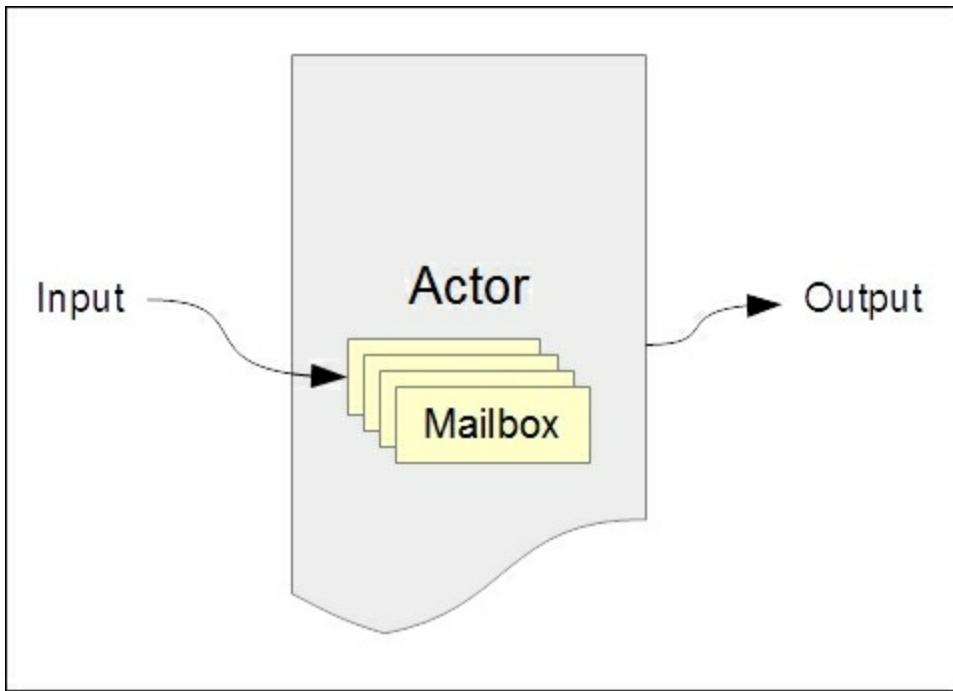
# Using actors

**Actors** are another way of modeling a system as a large number of concurrently running processes. Each process in *the actor model* is termed as an actor, and this model is based on the philosophy that every piece of logic in a system can be represented as an actor. The theory behind actors was first published by Carl Hewitt in the early '70s. Before we explore actors, we must note that the core Clojure language and libraries do not provide an implementation of the actor model. In fact, it is a widely accepted notion in the Clojure community that processes and channels are a much better methodology to model concurrently running processes compared to actors. That aside, actors can be used to provide more resilient error handling and recovery, and it is possible to use actors in Clojure through the Pulsar library (<https://github.com/puniverse/pulsar>).

## Note

To find out more about why processes and channels are preferred over actors in Clojure, take a look at *Clojure core.async Channels* by Rich Hickey (<http://clojure.com/blog/2013/06/28/clojure-core-async-channels>).

The actor model describes actors as concurrent processes that perform some computation on receiving messages. An actor can also send messages to other actors, create more actors, and change its own behavior depending on the messages it receives. Actors can also have their own internal state. In fact, actors were originally described as independent processors with their own local memory that interact with each other through a high-speed communication network. Every actor has its own *mailbox* to receive messages, and messages are the only means of conveying data between actors. The following diagram depicts an actor as an entity that receives some input as messages and performs computations to produce some output:



The Pulsar library provides a comprehensive implementation of the actor model. In this library, actors are scheduled to execute on **fibers**, which are similar to asynchronous tasks created using the `go` form from the `core.async` library. Fibers are scheduled to run on fork-join thread pools, unlike regular thread pools that are used in the `core.async` library. Due to this design, the Pulsar library is available only on the JVM, and not in the browser through ClojureScript.

Fibers communicate with each other through the Pulsar library's own implementation of *promises* and *channels*. Interestingly, the Pulsar library also has several thin wrappers around its implementation of channels, to provide an API that is fully compatible with that of the `core.async` library. Although we won't discuss fibers, promises, and channels from the Pulsar library any further in this section, we must understand that channels are quite relevant to actors, since an actor's mailbox is implemented using a channel. Now, let's explore the basics of actors in the Pulsar library.

## Creating actors

The `spawn` macro, from the `co.paralleluniverse.pulsar.actors` namespace, creates a new actor and must be passed a function that takes no arguments. We can specify the buffer size of an actor's mailbox using the `:mailbox-size` keyword argument of the `spawn` macro. There are several other interesting keyword arguments that can be passed to the `spawn` form, and you are encouraged to find out more about them on your own.

## Note

The following library dependencies are required for the upcoming examples:

```
[co.paralleluniverse/quasar-core "0.7.3"]
[co.paralleluniverse/pulsar "0.7.3"]
```

Your `project.clj` file must also contain the following entries:

```
:java-agents
[[co.paralleluniverse/quasar-core "0.7.3"]]
:jvm-opts
["-Dco.paralleluniverse.pulsar.instrument.auto=all"]
```

Also, the following namespaces must be included in your namespace declaration:

```
(ns my-namespace
  (:require [co.paralleluniverse.pulsar.core :as pc]
            [co.paralleluniverse.pulsar.actors :as pa]))
```

The function supplied to the `spawn` macro must use the `receive` macro, from the `co.paralleluniverse.pulsar.actors` namespace, to process messages received by the actor. Within this supplied function, we can use the expression `@self` to refer to the actor executing it. The `receive` form also supports pattern matching, which is implemented through the `core.match` library. We can also call the `receive` macro with no arguments, in which case it will return a message from the actor's mailbox. The `receive` form will also park the fiber on which it is executed.

To send messages to actors, we can use either the `!` or `!!` macros from the `co.paralleluniverse.pulsar.actors` namespace. Both these macros must be passed an actor and an expression that returns a value, and both of these forms return `nil`. The only difference between these two forms is that `!` is asynchronous, while `!!` is synchronous and may block the current thread of execution if the actor's mailbox is full. An actor may terminate on receiving a particular message, and we can check whether an actor is still active using the `done?` function from the `co.paralleluniverse.pulsar.actors` namespace. Once an actor terminates, we can obtain the final value returned by the actor using the `join` function from the `co.paralleluniverse.pulsar.core` namespace. For example, consider the actor created using the `spawn` and `receive` forms in *Example 8.8*.

## Note

The following examples can be found in `src/m_clj/c8/actors.clj` of the book's source code. Some of these examples are based on code from the official Pulsar documentation (<http://docs.paralleluniverse.co/pulsar/>).

```
(def actor (pa/spawn
  # (pa/receive
      :finish (println "Finished")
      m (do
          (println (str "Received: " m))
          (recur))))
```

### *Example 8.8: An actor created using the spawn macro*

The actor, represented by the preceding variable `actor`, will receive a message, print it and loop using a `recur` form. If the message `:finish` is received, it will print a string and terminate. The following code demonstrates how we can send a message to the actor:

```
user> (pa/! actor :foo)
nil
```

```
Received: :foo
user> (pa/done? actor)
false
```

As shown here, sending the value `:foo` to the actor returns `nil` immediately, and the message gets printed from another thread. As the `done?` function returns `false` when passed the variable `actor`, it is evident that the actor does not terminate on receiving the value `:foo` as a message. On the other hand, if we send the value `:finish` to the actor, it will terminate, as shown here:

```
user> (pa/! actor :finish)
nil
Finished
user> (pa/done? actor)
true
```

After being sent the value `:finish`, the `done?` function returns `true` when applied to the actor, which implies that the actor has terminated. The value returned by an actor before termination can be obtained using the `join` function from the `co.paralleluniverse.pulsar.core` namespace. We must note that the `join` function actually returns the result of any fiber, and will block the calling thread of execution until the fiber completes or terminates. For example, consider the actor in *Example 8.9* that divides a number by another number:

```
(def divide-actor
  (pa/spawn
    #(loop [c 0]
       (pa/receive
         :result c
         [a b] (recur (/ a b))))))
```

*Example 8.9: An actor that performs division of a number by another*

We can send messages to the actor `divide-actor` defined in *Example 8.9*, and obtain the final result from it using the `join` function, as shown here:

```

user> (pa/! divide-actor 30 10)
nil
user> (pa/! divide-actor :result)
nil
user> (pc/join divide-actor)
3

```

The preceding code shows that we can send two numbers to the actor `divide-actor`, and send it the value `:result` to terminate it. After termination, we can obtain the result of the actor, that is `3`, by passing the actor to the `join` function.

Actors can be registered with meaningful names that can be used to locate them. This is done using the `register!` function from the `co.paralleluniverse.pulsar.actors` namespace, which must be passed an actor instance and a name to register for the supplied actor. We can then send messages to a registered actor by specifying the actor's name to either the `!` or `!!` functions. For example, suppose the variable `actor` represents an actor instance created using the `spawn` macro. After registering the actor with the name `:my-actor` by calling `(pa/register! actor :my-actor)`, we can send the value `:foo` to the actor by calling `(pa/! :my-actor :foo)`.

## Passing messages between actors

Now, let's build a simple simulation of a ping pong game with two actors. These two actors will send the messages `:ping` and `:pong` to each other for a specified number of times. The code for this simulation is shown in *Example 8.10* as follows:

```

(defn ping-fn [n pong]
  (if (= n 0)
    (do
      (pa/! pong :finished)
      (println "Ping finished"))
    (do
      (pa/! pong [:ping @pa/self])
      (pa/receive
        (fn [pong]
          (when (= (:pong pong) 0)
            (pa/! pong :finished)
            (println "Pong finished"))
          (ping-fn (- n 1) pong)))))))

```

```

:pong (println "Ping received pong"))
(recur (dec n) pong)))))

(defn pong-fn []
  (pa/receive
   :finished (println "Pong finished")
   [:ping ping] (do
                  (println "Pong received ping")
                  (pa/! ping :pong)
                  (recur)))))

(defn start-ping-pong [n]
  (let [pong (pa/spawn pong-fn)
        ping (pa/spawn ping-fn n pong)]
    (pc/join pong)
    (pc/join ping)
    :finished))

```

*Example 8.10: Two actors playing a game of ping-pong*

The `ping-fn` and `pong-fn` functions shown in *Example 8.10* implement the logic of two actors playing a game of ping pong. The `ping-fn` will essentially send a vector containing the keyword `:ping` and the current actor instance to the actor represented by the argument `pong`. This is done `n` times, and finally the message `:finished` is sent to the actor `pong`. The function `pong-fn` will receive the vector `[:ping ping]`, where `ping` will be the actor sending the message. An actor created with the `pong-fn` will terminate once it receives the message `:finished`. The `start-ping-pong` function simply creates two actors using the functions `ping-fn` and `pong-fn` and waits until they are both finished using the `join` function. We can call the `start-ping-pong` function by passing in the number of times each of the two actors must send messages to each other, as shown here:

**user> (`start-ping-pong` 3)**

```

Pong received ping
Ping received pong
Pong received ping
Ping received pong
Pong received ping
Ping received pong

```

```
Ping finished
Pong finished
:finished
```

The two actors created by the `start-ping-pong` function pass messages between themselves to simulate a game of ping pong, as demonstrated by the preceding output. In conclusion, actors from the Pulsar library can be used to implement concurrently executing processes.

## Handling errors with actors

Actors support some interesting methods for error handling. If an actor encounters an error while processing a received message, it will terminate. The exception that was raised within the fiber executing the actor will be saved and thrown again when we pass the actor to the `join` function. In effect, we don't need to handle exceptions within the function passed to the `spawn` macro, and instead we must catch exceptions when the `join` function is called.

This brings us to an interesting consequence of actors. If an actor could encounter an error and fail, we can have another actor that monitors the first actor, and restart it in case of failure. Thus, an actor can be notified when another actor in the system terminates. This principle allows actors to recover from errors in an automated fashion. In the Pulsar library, this sort of error handling is done through the `watch!` and `link!` functions from the `co.paralleluniverse.pulsar.actors` namespace.

An actor can *watch* or *monitor* another actor by calling the `watch!` function from within its body. For example, we must call `(watch! A)` within the body of an actor to watch the actor `A`. If the actor being watched encounters an exception, the same exception will be thrown from the `receive` form of the monitoring actor. The monitoring actor must catch the exception, or else it will be terminated along with the actor from which the exception originated. Also, the monitoring actor could restart the terminated actor by calling the `spawn` macro. To stop watching an actor, we

must pass the watched actor to the `unwatch!` function from within the body of the monitoring actor.

Two actors could also be *linked* by passing them to the `link!` function. If two actors are linked together, an exception encountered in either of the two actors will be caught by the other one. In this way, linking two actors is a symmetrical way of monitoring them for errors. The `link!` function can also be called within the function passed to a `spawn` form, in which case it must be passed the actor to be linked. To unlink two actors, we can use the `unlink!` function.

Thus, the Pulsar library provides some interesting ways to watch and link actors to perform error handling and recovery.

## Managing state with actors

As we mentioned earlier, actors can have their own internal mutable state. Of course, accessing this state from other actors is not allowed, and immutable messages are the only way an actor can communicate with other actors. Another way that an actor can maintain or manage its state is by changing its behavior depending on the messages it receives, and this technique is called a *selective receive*.

Every actor created using the `spawn` function can read its internal state using the expression `@state`, and can also write to this state using the `set-state!` function. The `set-state!` function will also return the new state of the actor, as returned by the expression `@state`. Note that both of these forms are implemented in the `co.paralleluniverse.pulsar.actors` namespace.

Consider the `add-using-state` function in *Example 8.11* that uses an actor to add two numbers. Of course, we would never really need such a function in the real world, and it is only demonstrated here to depict how an actor can change its internal state.

```
(defn add-using-state [a b]
  (let [actor (pa/spawn
    # (do
        (pa/set-state! 0)
        (pa/set-state! (+ @pa/state (pa/receive)))
        (pa/set-state! (+ @pa/state (pa/receive))))))
    (pa/! actor a)
    (pa/! actor b)
    (pc/join actor)))
```

*Example 8.11: A function to add two numbers using an actor*

The `add-using-state` function shown in *Example 8.11* creates an actor that sets its state to 0, and adds the first two messages it receives to its state. The actor will return the latest state of the actor, as returned by the last call to `set-state!` in the function passed to the `spawn` macro. On calling the `add-using-state` function with two numbers, it produces their sum as its output, shown as follows:

```
user> (add-using-state 10 20)
30
```

Another way in which an actor can modify its state is through a selective receive, in which the actor modifies its behavior on receiving a particular message. This is done by calling a `receive` form within the body of another `receive` form, as shown in *Example 8.12*:

```
(defn add-using-selective-receive [a b]
  (let [actor (pa/spawn
    # (do
        (pa/set-state! 0)
        (pa/receive
          m (pa/receive
            n (pa/set-state! (+ n m))))))
    (pa/! actor a)
    (pa/! actor b)
    (pc/join actor)))
```

*Example 8.12: A function to add two numbers using an actor with*

## *selective receive*

The `add-using-selective-receive` function shown previously will set its state to 0, receive the messages `m` and `n` through a selective receive, and add these messages. This function produces identical results as the `add-using-state` function from *Example 8.11*, as shown here:

```
user> (add-using-selective-receive 10 20)
30
```

In this way, actors can change their internal state and behavior based on the messages sent to them.

## Comparing processes and actors

CSPs and actors are two distinct approaches to modeling a system as a large number of concurrent processes that execute and interact asynchronously. The logic of an asynchronous task can reside within a process created using a `go` block, or within the function passed to the `spawn` macro that creates an actor. However, there are some subtle contrasts between these two approaches:

- Processes created using the `go` and `thread` forms encourage us to put all states onto channels. Actors, on the other hand, can have their own internal state, in addition to the state in the form of messages sent to them. Thus, actors are more like objects with encapsulated state, while processes are more like functions that operate on states stored in channels.
- Tasks created using the `go` or `thread` macros have no implicit error handling, and we must handle exceptions using the `try` and `catch` forms in the body of the `go` and `thread` macros. Of course, channels do support error handlers, but only when combined with a transducer. Actors, however, will save any exception they run into until we apply the `join` function on the actor. Also, actors can be linked and monitored to provide a form of automated error recovery. In this way, actors are more focused on building fault-tolerant systems.

These distinguishing factors between CSPs and the actor model give us an idea about which approach is more suitable for implementing asynchronous tasks in a given problem.

# Summary

In this chapter, we looked at how we can create concurrent and asynchronous tasks using the `core.async` and Pulsar libraries. The `core.async` library provides an implementation of CSPs, and is supported in both Clojure and ClojureScript. We studied the various constructs in the `core.async` library and also demonstrated how a solution to the dining philosophers problem can be implemented using this library. Later on, we explored actors through the Pulsar library.

We will explore reactive programming in the following chapter. As we will see ahead, reactive programming can be thought of as an extension of asynchronous programming for handling data and events.

# Chapter 9. Reactive Programming

One of the many interesting applications of programming with asynchronous tasks is *reactive programming*. This methodology of programming is all about asynchronously reacting to changes in state. In reactive programming, code is structured in such a way that it *reacts* to changes. Generally, this is implemented using asynchronous data streams, in which data and events are propagated asynchronously through a program. In fact, there are quite a few interesting variants of reactive programming.

Reactive programming is particularly useful in designing graphical user interfaces in frontend development, where changes in the internal state of an application must asynchronously trickle down to the user interface. A program is thus segregated into events and logic that is executed on those events. For programmers used to imperative and object-oriented programming techniques, the hardest part of reactive programming is thinking in reactive abstractions and letting go of old habits like using the mutable state. However, if you've been paying attention so far and have started thinking with immutability and functions, you'll find reactive programming quite natural. In the JavaScript world, reactive programming with *observables* can be thought of as a contrasting alternative to using promises to manage asynchronous events and actions.

In this chapter, we will explore a few interesting forms of reactive programming through Clojure and ClojureScript libraries. Later on, we will also demonstrate how we can build dynamic user interfaces using reactive programming.

## Reactive programming with fibers and dataflow variables

*Dataflow programming* is one of the simplest forms of reactive programming. In dataflow programming, computations are described by composing variables without bothering about when these variables are set to a value. Such variables are also called **dataflow variables**, and they will trigger computations that refer to them once they are set. The *Pulsar library* (<https://github.com/puniverse/pulsar>) provides a few useful constructs for dataflow programming. These constructs can also be used with Pulsar **fibers**, which we briefly talked about in [Chapter 8, Leveraging Asynchronous Tasks](#). In this section, we will explore the basics of fibers and dataflow variables from the Pulsar library.

## Note

The following library dependencies are required for the upcoming examples:

```
[co.paralleluniverse/quasar-core "0.7.3"]
[co.paralleluniverse/pulsar "0.7.3"]
```

Your `project.clj` file must also contain the following entries:

```
:java-agents
[[co.paralleluniverse/quasar-core "0.7.3"]]
:jvm-opts
["-Dco.paralleluniverse.pulsar.instrument.auto=all"]
```

Also, the following namespaces must be included in your namespace declaration:

```
(ns my-namespace
  (:require [co.paralleluniverse.pulsar.core :as pc]
            [co.paralleluniverse.pulsar.dataflow
             :as pd]))
```

The elementary abstraction of an asynchronous task in the Pulsar library is a fiber. Fibers are scheduled for execution on fork-join based thread pools, and we can create a large number of fibers without bothering about the number of available processing cores. Fibers can be created using the

spawn-fiber and fiber macros from the co.paralleluniverse.pulsar.core namespace. The spawn-fiber macro must be passed a function that takes no arguments, and the fiber form must be passed a body of expressions. The body of both these forms will be executed on a new fiber. The join function from the co.paralleluniverse.pulsar.core namespace can be used to retrieve the value returned by a fiber.

An important rule we must keep in mind while dealing with fibers is that we must never call methods or functions that manipulate the current thread of execution from within a fiber. Instead, we must use fiber-specific functions from the co.paralleluniverse.pulsar.core namespace to perform these operations. For example, calling the java.lang.Thread/sleep method in a fiber must be avoided. Instead, the sleep function from the co.paralleluniverse.pulsar.core namespace can be used to suspend the current fiber for a given number of milliseconds.

## Note

The following examples can be found in `src/m_c1j/c9/fibers.clj` of the book's source code. Some of these examples are based on code from the official Pulsar documentation (<http://docs.paralleluniverse.co/pulsar/>).

For example, we can add two numbers using a fiber as shown in *Example 9.1*. Of course, using a fiber for such a trivial operation has no practical use, and it is only shown here to demonstrate how we can create a fiber and obtain its return value:

```
(defn add-with-fiber [a b]
  (let [f (pc/spawn-fiber
           (fn []
             (pc/sleep 100)
             (+ a b)))]
    (pc/join f)))
```

### *Example 9.1: Adding two numbers with a fiber*

The preceding `add-with-fiber` function creates a fiber `f` using the `spawn-fiber` macro and fetches the return value of the fiber using the `join` function. The fiber `f` will suspend itself for 100 milliseconds using the `sleep` function and then return the sum of `a` and `b`.

Let's talk a bit about dataflow variables. We can create dataflow variables using the `df-val` and `df-var` functions from the

`co.paralleluniverse.pulsar.dataflow` namespace. A dataflow variable created using these functions can be set by calling it like a function and passing it a value. Also, the value of a dataflow variable can be obtained by dereferencing it using the `@` operator or the `deref` form. A dataflow variable declared using the `df-val` function can only be set once, whereas one created using the `df-var` function can be set several times.

The `df-var` function can also be passed a function that takes no arguments and refers to other dataflow variables in the current scope. This way, the value of such a dataflow variable will be recomputed when the values of referenced variables are changed. For example, two numbers can be added using dataflow variables as shown in the `df-add` function defined in

### *Example 9.2:*

```
(defn df-add [a b]
  (let [x (pd/df-val)
        y (pd/df-val)
        sum (pd/df-var #(+ @x @y) )]
    (x a)
    (y b)
    @sum))
```

### *Example 9.2: Adding two numbers with dataflow variables*

The value of the dataflow variable `sum`, declared in the preceding `df-add` function, will be recalculated when the referenced dataflow variables `x` and `y` are set to a value. The variables `x` and `y` are set by calling them like

functions. Similarly, we can add a number to each element in a range of numbers using the `df-val` and `df-var` functions as shown in the following *Example 9.3*:

```
(defn df-add-to-range [a r]
  (let [x (pd/df-val)
        y (pd/df-var)
        sum (pd/df-var #(+ @x @y))
        f (pc/fiber
            (for [i r]
              (do
                (y i)
                (pc/sleep 10)
                @sum)))
        (x a)
        (pc/join f)))
```

*Example 9.3: Adding a number to a range of number with dataflow variables*

The `df-add-to-range` function shown previously defines the dataflow variables `x`, `y`, and `sum`, where `sum` is dependent on `x` and `y`. The function then creates a fiber `f` that uses the `for` macro to return a sequence of values. Within the body of the `for` macro, the dataflow variable `y` is set to a value from the range `r`, and the value `@sum` is returned. The fiber thus returns the result of adding `a` to all elements in the range `r`, as shown in the following output:

```
user> (df-add-to-range 2 (range 10))
(2 3 4 5 6 7 8 9 10 11)
```

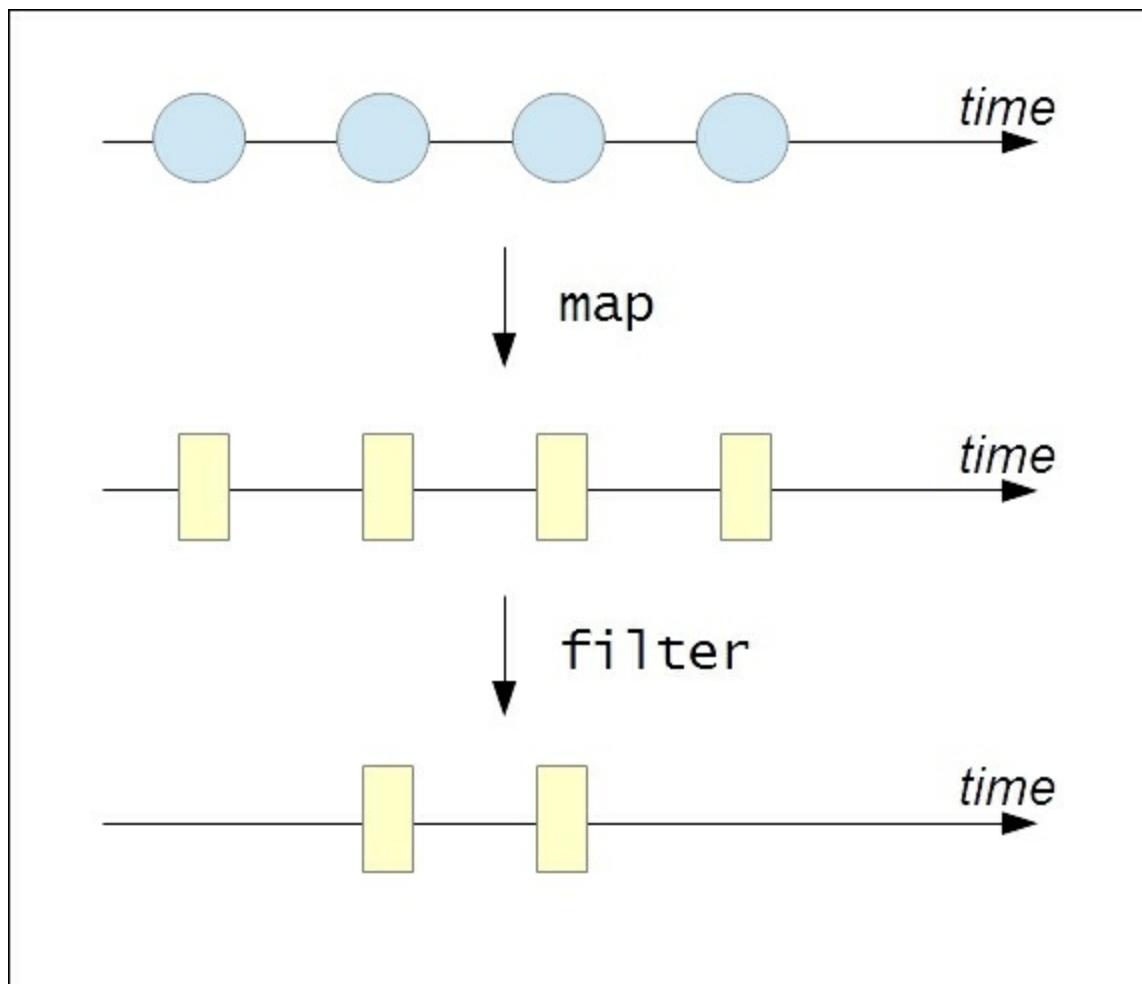
In conclusion, we can use the `df-val` and `df-var` functions to define dataflow variables, whose value can be recomputed when its referenced variables are changed. Effectively, changing the state of a dataflow variable may cause other dataflow variables to *react* to the change.

We should note that the Pulsar library also implements channels, which are analogous to channels from the `core.async` library. In a nutshell, channels

can be used to exchange data with fibers. The Pulsar library also provides constructs for reactive programming with channels, through the `co.paralleluniverse.pulsar.rx` namespace. These constructs are termed as *reactive extensions*, and are very similar to transducers, in the sense that they perform some computation on the values in a channel. Reactive extensions are also implemented by the *RxClojure* library. We should note that one of the limitations of both the Pulsar and RxClojure libraries is that they are available only on the JVM, and can't be used in ClojureScript programs. Thus, using `core.async` channels with transducers is a more feasible option in ClojureScript. Nevertheless, we will briefly explore reactive extensions through the RxClojure library in the following section.

# Using Reactive Extensions

**Reactive Extensions** (written as Rx) are a generalized implementation of reactive programming that can be used to model event and data streams. Rx can be thought of as an object-oriented approach to reactive programming, in the sense that an event stream is an object with certain methods and properties. In Rx, asynchronous event streams are termed as *observables*. An entity or object that subscribes to events from an observable is called an *observer*. Reactive extensions are essentially a library of functions, or methods, to manipulate observables and create objects that conform to the observer-observable pattern. For example, an observable can be transformed using the Rx variants of the `map` and `filter` functions, as shown in the following illustration:



As shown previously, an observable can be described as a collection of values that vary over a period of time. It's quite evident that observables can be treated as a sequence of values using the Rx-flavored variants of the `map` and `filter` functions. An observable can also be subscribed to by an observer, and the observer will be asynchronously invoked for any value produced by an observable.

We will now discuss the various constructs of the RxClojure library (<https://github.com/ReactiveX/RxClojure>). There are several implementations of Rx across multiple languages, such as C#, Java, and PHP. The Java library for reactive extensions is RxJava, and the RxClojure library provides Clojure bindings to RxJava. As we mentioned earlier, it's important to note that RxClojure can only be used on the JVM. Also, the RxClojure library predates the implementation of transducers in Clojure, and thus channels and transducers are a more portable and generic approach to reactive programming.

## Note

The following library dependencies are required for the upcoming examples:

```
[io.reactivex/rxclojure "1.0.0"]
```

Also, the following namespaces must be included in your namespace declaration:

```
(ns my-namespace
  (:require [rx.lang.clojure.core :as rx]
            [rx.lang.clojure.blocking :as rxb]
            [rx.lang.clojure.interop :as rxj]))
```

The `rx.lang.clojure.core` namespace contains functions for creating and manipulating observables. Observables are internally represented as collections of values. To extract values from observables, we can use functions from the `rx.lang.clojure.blocking` namespace. However, we must note that functions from the `rx.lang.clojure.blocking` namespace

must be avoided in a program and used only for testing purposes. The `rx.lang.clojure.interop` namespace contains functions for performing Java interop with the underlying RxJava library.

## Note

The following examples can be found in `src/m_clj/c9/rx.clj` of the book's source code.

A value can be converted to an observable using the `return` function from the `rx.lang.clojure.core` namespace. An observable can be converted to a vector of values using the `rx.lang.clojure.blocking/into` function, and similarly, we can obtain the first value of an observable using the `rx.lang.clojure.blocking/first` function. These functions are demonstrated in the following output:

```
user> (def o (rx/return 0))
#'user/o
user> (rxb/into [] o)
[0]
user> (rxb/first o)
0
```

A sequence of values can be converted to an observable using the `seq->o` function from the `rx.lang.clojure.core` namespace. To convert the observable back to a sequence, we pass it to the `o->seq` function from the `rx.lang.clojure.blocking` namespace. For example, we can convert the vector `[1 2 3]` to an observable and back to a sequence, as shown here:

```
user> (def o (rx/seq->o [1 2 3]))
#'user/o
user> (rxb/o->seq o)
(1 2 3)
```

Another way of creating an observable is by using the `cons` and `empty` functions from the `rx.lang.clojure.core` namespace. The `empty` function creates an observable with no values, and the `cons` function adds or combines a value and an observable into a new, observable, similar to the

standard `cons` function. We can create an observable containing the value `0` using the `cons` and `empty` functions as follows:

```
user> (def o (rx/cons 0 (rx/empty)))
#'user/o
user> (rxb/first o)
0
```

As we mentioned earlier, observers can subscribe to events from observables. Observers can be defined by implementing the `rx.lang.clojure.Observer` interface. This interface defines three methods, namely `onNext`, `onError`, and `onCompleted`. The `onNext` method is called whenever an observable produces a new value, and the `onCompleted` method is called when an observable is done producing values. The `onError` method will be called in case an exception is encountered. Interestingly, all of these methods will be invoked asynchronously from an observable. For example, we can create an observer using the `reify` form to implement the `Observer` interface as shown in *Example 9.4*:

```
(def observer
  (reify rx.Observer
    (onNext [this v] (println (str "Got value: " v "!")))
    (onError [this e] (println e))
    (onCompleted [this] (println "Done!"))))
```

#### *Example 9.4: Implementing the rx.lang.clojure.Observer interface*

An observable can call the methods of all its subscribed observers using the `on-next`, `on-error` and `on-completed` functions from the `rx.lang.clojure.core` namespace. We can also define an observable using these functions and the `observable*` form from the `rx.lang.clojure.core` namespace. The `observable*` form must be passed a function that takes a single argument, which represents an observer. For example, we can define a function to create an observable of two values using the `observable*` form as shown in *Example 9.5*:

```
(defn make-observable []
  (rx/observable* (fn [s]
```

```
(-> s
     (rx/on-next :a)
     (rx/on-next :b)
     rx/on-completed))))
```

### *Example 9.5: Creating an observable using the observable\* form*

The function passed to the `observable*` form, shown previously, calls the `on-next` and `on-completed` functions to produce an observable of two values. We can convert this observable into a vector using the `into` function from the `rx.lang.clojure.blocking` namespace, as shown here:

```
user> (def o (make-observable))
#'user/o
user> (rxb/into [] o)
[:a :b]
```

An observer can also be created using the `subscribe` function from the `rx.lang.clojure.core` namespace. This function must be passed a function that takes a single value, and an observer will be created by implementing the `onNext` method using the supplied function. We can also pass a second argument representing the `onError` method, as well as a third argument that represents the `onCompleted` method, to the `subscribe` function. For example, we can subscribe to an observable using the `subscribe` function, and apply a function to all values in the observable using the `rx.lang.clojure.core/map` function, as shown in *Example 9.6*:

```
(defn rx-inc [o]
  (rx/subscribe o (fn [v] (println (str "Got value: " v "!"))))
  (rx/map inc o))
```

### *Example 9.6: Subscribing to an observable using the subscribe function*

We can create an observable and pass it to the `rx-inc` function defined in *Example 9.6*, as shown here:

```
user> (def o (rx/seq->o [0 1 2]))
```

```
#'user/o
user> (rx-inc o)
Got value: 0!
Got value: 1!
Got value: 2!
#<rx.Observable 0xc3fae8>
```

The function passed to the `subscribe` form in *Example 9.6* is executed every time the `inc` function is applied to a value in the observable `o`. We could as well define the `rxj-inc` function using Java interop with RxJava, as shown in *Example 9.7*:

```
(defn rxj-inc [o]
  (.subscribe o (rxj/action [v]
    (println (str "Got value: " v "!")))
  (.map o (rxj/fn [v] (inc v))))
```

### *Example 9.7: Subscribing to an observable using the Java interop*

It's quite clear that using the RxJava library through Java interop isn't pretty, as we would have to wrap all the functions in the `action` and `fn` forms from the `rx.lang.clojure.interop` namespace. The `action` macro is used to represent a function that performs a side-effect, whereas the `fn` macro is used to wrap functions that return values. Observables can also be created using the Java interop. This is done using the `from` static method from the `rx.lang.clojure.core.Observable` class. The following output demonstrates this method as well as the `rxj-inc` function defined in *Example 9.7*:

```
user> (def o (rx.Observable/from [0 1 2]))
#'user/o
user> (rxj-inc o)
Got value: 0!
Got value: 1!
Got value: 2!
#<rx.Observable 0x16459ef>
```

Of course, we should prefer to use functions from the `rx.lang.clojure.core` namespace, and we are using Java interop here

only to show that it is possible. Similar to the `map` function used in *Example 9.6*, there are several other functions in the `rx.lang.clojure.core` namespace that allow us to treat observables as sequences. Thus, functions such as `map`, `filter`, and `mapcat` comprise the interface of observables, and describe the many ways in which we can interact with them. For example, the following output demonstrates the Rx variants of the `take`, `cycle`, and `range` functions:

```
user> (rxb/into [] (->> (rx/range)
                               (rx/take 10)))
[0 1 2 3 4 5 6 7 8 9]
user> (rxb/into [] (->> (rx/cycle (rx/return 1))
                               (rx/take 5)))
[1 1 1 1 1]
```

The `rx.lang.clojure.core` namespace also provides a `filter` function that can be used with an observable and a predicate, as shown here:

```
user> (rxb/into [] (->> (rx/seq->o [:a :b :c :d :e])
                               (rx/filter #{:b :c})))
[:b :c]
```

The `group-by` and `mapcat` functions from the `rx.lang.clojure.core` namespace have the same semantics as the standard versions of these functions. For example, let's define a function that uses the `group-by` and `mapcat` functions, as shown in *Example 9.8*:

```
(defn group-maps [ms]
  (->> ms
        (rx/seq->o)
        (rx/group-by :k)
        (rx/mapcat (fn [[k vs :as m]]
                     (rx/map #(vector k %) vs)))
        (rxb/into [])))
```

### *Example 9.8: Using the group-by and mapcat functions*

The `group-maps` function, defined previously, will transform a number of maps into an observable, group the maps by their values for the key `:k`,

and create a number of vectors using the `mapcat` and `map` functions. Of course, we wouldn't really need such a function in practice, and it's only shown here to demonstrate how the `group-by` and `mapcat` functions can be used. We can pass a vector of maps to the `group-maps` function to produce a sequence of vectors, as shown here:

```
user> (group-maps [{:k :a :v 1}
                     {:k :b :v 2}
                     {:k :a :v 3}
                     {:k :c :v 4}])
[[{:a {:k :a, :v 1}}
 {:a {:k :a, :v 3}}
 {:b {:k :b, :v 2}}
 {:c {:k :c, :v 4}}]]
```

Several observables can be combined using the `merge` function from the `rx.lang.clojure.core` namespace. The `merge` function can be passed any number of observables, as shown here:

```
user> (let [o1 (rx/seq->o (range 5))
            o2 (rx/seq->o (range 5 10))
            o (rx/merge o1 o2)]
        (rxb/into [] o))
[0 1 2 3 4 5 6 7 8 9]
```

An observable can also be split up into two observables using the `split-with` function from the `rx.lang.clojure.core` namespace. This function must be passed an observable and a predicate function, as shown here:

```
user> (->> (range 6)
              rx/seq->o
              (rx/split-with (partial >= 3))
              rxb/first
              (map (partial rxb/into [])))
([0 1 2 3] [4 5])
```

In summary, the RxClojure library provides us with several constructs for creating and manipulating observables. We can also easily create observers that asynchronously *react* to observables using the `subscribe` function

from this library. Also, the constructs from the `rx.lang.clojure.core` namespace have semantics similar to that of standard functions such as `map`, `filter`, and `mapcat`. There are several functions in the `rx.lang.clojure.core` namespace that we haven't talked about in this section, and you're encouraged to explore them on your own.

# Using functional reactive programming

A more functional flavor of reactive programming is **functional reactive programming** (abbreviated as **FRP**). FRP was first described in the late '90s by Conal Elliott, who was a member of the Microsoft Graphics Research Group at the time, and Paul Hudak, a major contributor to the Haskell programming language. FRP was originally described as a bunch of functions to interact with *events* and *behaviors*. Both events and behaviors represent values that change over time. The major difference between these two is that events are values that change discretely over time, whereas behaviors are continuously changing values. There is no mention of an observer-observable pattern in FRP. Also, programs in FRP are written as composable transformations of events and behaviors, and are also termed as **compositional event systems (CESs)**.

Modern implementations of FRP provide constructs to create and transform asynchronous event streams. Also, any form of state change is represented as an event stream. In this perspective, a click of a button, a request made to a server, and mutating a variable, can all be treated as event streams. The *Bacon.js* library (<https://github.com/baconjs/bacon.js/>) is a JavaScript implementation of FRP, and the *Yolk* library (<https://github.com/Cicayda/yolk>) provides ClojureScript bindings to the Bacon.js library. In this section, we will briefly study the constructs provided by the Yolk library.

## Note

The following library dependencies are required for the upcoming examples:

```
[yolk "0.9.0"]
```

Also, the following namespaces must be included in your namespace declaration:

```
(ns my-namespace
  (:require [yolk.bacon :as y]))
```

In addition to the preceding dependencies, the following examples also use the `set-html!` and `by-id` functions from `src/m_clj/c9/common.cljs`. These functions are defined as follows:

```
(defn ^:export by-id [id]
  (.getElementById js/document id))

(defn ^:export set-html! [el s]
  (set! (.-innerHTML el) s))
```

Ensure that the code in the following ClojureScript examples is compiled, using the following command:

```
$ lein cljsbuild once
```

The `yolk.bacon` namespace provides several functions to create event streams, such as the `later` and `interval` functions. The `later` function creates an event stream that produces a single value after a given delay. The `interval` function can infinitely repeat a value with a given time interval. Both these functions must be passed a number of milliseconds as the first argument and a value to produce as the second argument.

Event streams in the Yolk library may produce an infinite number of values. We can limit the number of values produced by an event stream by using the `yolk.bacon/sliding-window` function, which creates an event stream that drops older values once it's full. This function must be passed an event stream and a number indicating the capacity of the event stream returned by it.

We can also create an *event bus*, onto which we can arbitrarily push values, using the `bus` function from the `yolk.bacon` namespace. The `push`

function puts a value onto an event bus, and the `plug` function connects an event bus to another event stream.

To listen to values produced from event streams, we can use the `on-value`, `on-error`, and `on-end` functions. The `on-value` and `on-error` functions will call a supplied 1-arity function whenever a given event stream produces a value or an error, respectively. The `on-end` function will call a supplied function that takes no arguments whenever a stream ends. This function is often used with the `yolk.bacon/never` function, which creates an event stream that ends immediately without producing a value.

Event streams can also be combined in several ways. The `merge-all` function combines a vector of several event streams into a single one. Another function that can collect values from several event streams in this way is the `flat-map` function. Alternatively, the `combine-array` function can be used to create a single event stream that produces arrays of the values from the supplied streams. The `yolk.bacon/when` function can be used to conditionally combine several channels. This function must be passed a number of clauses, similar to the `cond` form. Each clause must have two parts—a vector of event streams and an expression that will be invoked when all the supplied event streams produce values.

The `yolk.bacon` namespace also provides event stream based variants of the standard `map`, `filter`, and `take` functions. These functions take an event stream as the first argument, which is a little different from the semantics of the standard versions of these functions.

Using these functions from the Yolk library, we can implement a simplified ClojureScript based solution to the dining philosophers problem, which we described in the previous chapters. For a detailed explanation of the dining philosophers problem and its solution, refer to [Chapter 2, Orchestrating Concurrency and Parallelism](#) and [Chapter 8, Leveraging Asynchronous Tasks](#).

## Note

The following examples can be found in `src/m_clj/c9/yolk/core.cljs` of the book's source code. Also, the HTML page for the following ClojureScript examples can be found in `resources/html/yolk.html`. The following scripts will be included in this page:

```
<script type="text/javascript" src="../js/bacon.js">
</script>
<script type="text/javascript" src="../js/out/yolk.js">
</script>
```

In this implementation of the dining philosophers problem, we will represent the state of the philosophers and the forks on the table using event buses. The event buses can then be combined using the `when` function from the Yolk library. We won't maintain much state about the philosophers for the sake of simplicity. Let's first define functions to print the philosophers and represent the routine of a philosopher, as shown in the following *Example 9.9*:

```
(defn render-philosophers [philosophers]
  (apply str
    (for [p (reverse philosophers)]
      (str "<div>" p "</div>"))))

(defn philosopher-fn [i n forks philosophers wait-ms]
  (let [p (nth philosophers i)
        fork-1 (nth forks i)
        fork-2 (nth forks (-> i inc (mod n)))]
    (fn []
      (js/setTimeout
        (fn []
          (y/push fork-1 :fork)
          (y/push fork-2 :fork)
          (y/push p {}))
        wait-ms)
      (str "Philosopher " (inc i) " ate!"))))
```

*Example 9.9: Solving the dining philosophers problem with event streams*

The preceding `render-philosophers` function will wrap each philosopher in a `div` tag, which will be displayed on a web page. The `philosopher-fn` function returns a function that represents the routine of a philosopher. The function returned by the `philosopher-fn` function sets off a task, using the `setTimeout` JavaScript function, to push values representing a particular philosopher and his associated forks into the event buses. This function will finally return a string indicating that the given philosopher was able to eat the food supplied to him. Using these functions, we can create a simulation of the dining philosophers problem in a web page, as shown in the following *Example 9.10*:

```
(let [out (by-id "ex-9-10-out")
      n 5
      [f1 f2 f3 f4 f5 :as forks] (repeatedly n #(y/bus))
      [p1 p2 p3 p4 p5 :as philosophers] (repeatedly n #(y/bus))
      eat #(philosopher-fn % n forks philosophers 1000)
      events (y/when [p1 f1 f2] (eat 0)
                  [p2 f2 f3] (eat 1)
                  [p3 f3 f4] (eat 2)
                  [p4 f4 f5] (eat 3)
                  [p5 f5 f1] (eat 4))]

  (-> events
    (y/sliding-window n)
    (y/on-value
      #(set-html! out (render-philosophers %))))
  (doseq [f forks]
    (y/push f :fork))
  (doseq [p philosophers]
    (y/push p {}))))
```

*Example 9.10: Solving the dining philosophers problem with event streams (continued)*

In the `let` form shown in *Example 9.10*, we created the philosophers and forks in our simulation using the `bus` function from the Yolk library. The values produced by these event buses are then combined using a `when` form. The `when` function in the preceding code will check for events from a philosopher and the forks on his left- and right-hand side. The

combinations of philosophers and forks are, in fact, hardcoded into the clauses of the `when` form. Of course, we must understand that the clauses of the `when` form shown previously could have easily been generated using a macro. Values are then placed onto the event buses representing the philosophers and forks using the `push` function, to start the simulation. The last five philosophers who could eat are rendered in the web page, as shown here:

```
Philosopher 5 ate!
Philosopher 3 ate!
Philosopher 1 ate!
Philosopher 4 ate!
Philosopher 2 ate!
```

In summary, the Yolk library provides several constructs to handle event streams. There are several functions from this library that we haven't discussed, and you should explore them on your own. In the following section, we will provide examples that demonstrate the other functions from the Yolk library.

## Note

Some of the preceding examples are based on code from *Yolk examples* by Wilkes Joiner (<https://github.com/Cicayda/yolk-examples>).

# Building reactive user interfaces

One of the primary applications of reactive programming is frontend development, where we must create user interface components that react asynchronously to changes in state. In this section, we will describe a few examples implemented using the `core.async` library and the `Yolk` library. This is meant to give you a comparison between channels and event streams, and also demonstrate how we can design solutions to problems using both these concepts. Note that only the overall design and code for these examples will be described, and you should be able to fill in the details on your own.

## Note

The following library dependencies are required for the upcoming examples:

```
[yolk "0.9.0"]
[org.clojure/core.async "0.1.346.0-17112a-alpha"]
```

Also, the following namespaces must be included in your namespace declaration:

```
(ns my-namespace
  (:require [goog.events :as events]
            [goog.events.EventType]
            [goog.style :as style]
            [cljs.core.async :as a]
            [yolk.bacon :as y])
  (:require-macros [cljs.core.async.macros
                    :refer [go go-loop alt!]]))
```

In addition to the preceding dependencies, the following examples also use the `set-html!` and `by-id` functions from `src/m_clj/c9/common.cljs`. Ensure that the code in the following ClojureScript examples is compiled, using the following command:

```
$ lein cljsbuild once
```

As a first example, let's create three asynchronous tasks that each produce values at different time intervals. We must fetch all the values produced by these tasks and render them on a web page in the same order.

## Note

The following examples can be found in

src/m\_clj/c9/reactive/core.cljs of the book's source code. Also, the [HTML page](#) for the following ClojureScript examples can be found in resources/html/reactive.html. The following scripts will be included in this page:

```
<script type="text/javascript" src="../js/bacon.js">
</script>
<script type="text/javascript" src="../js/out/reactive.js">
</script>
```

We could implement this using processes and channels from the core.async library. In this case, channels will convey the values produced by three processes, and we will use a merge operation to combine these channels, as shown in the following *Example 9.11*:

```
(defn render-div [q]
  (apply str
    (for [p (reverse q)]
      (str "<div class='proc-" p "'>Process " p "</div>"))))

(defn start-process [v t]
  (let [c (a/chan)]
    (go (while true
          (a/<! (a/timeout t))
          (a/>! c v)))
    c))

(let [out (by-id "ex-9-11-out")]
  c1 (start-process 1 250)
  c2 (start-process 2 1000)
  c3 (start-process 3 1500))
```

```

c (a/merge [c1 c2 c3])
firstn (fn [v n]
  (if (<= (count v) n)
    v
    (subvec v (- (count v) n))))]
(go-loop [q []]
  (set-html! out (render-div q))
  (recur (-> (conj q (a/<! c))
    (firstn 10)))))
```

*Example 9.11: Three asynchronous tasks using channels*

The preceding `start-process` function will create a process that periodically produces values using the `go` form, and returns a channel from which the values can be read. The `render-div` function will generate HTML for the values produced by the three tasks. Only the ten most recent values will be shown. This code will produce the following output:

```

Process 1
Process 2
Process 1
Process 1
Process 1
Process 1
Process 1
Process 2
Process 3
Process 1
Process 1
```

We could also implement the preceding example using FRP, in which values produced by each of the three tasks are represented as event streams. The `merge-all` function from the `yolk.bacon` namespace can be used to combine these event streams, and the `sliding-window` function can obtain the ten most recent values produced by the resulting stream. The `render-div` function from *Example 9.11* can be reused to render the values. This is implemented in *Example 9.12*, and produces the same output as *Example 9.11*:

```
(let [out (by-id "ex-9-12-out")
      events [(y/interval 250 1)
                (y/interval 1000 2)
                (y/interval 1500 3)]]]
  (-> events
      y/merge-all
      (y/sliding-window 10)
      (y/on-value
        #(set-html! out (render-div %)))))
```

*Example 9.12: Three asynchronous tasks using FRP*

Next, let's try to capture mouse events from a particular `div` tag, and display the page offset values of the locations of these events. We can do this with channels, but we would first need a function to convey DOM events onto a channel. We can implement this using the `goog.events/listen` and `cljs.core.async/put!` functions, as shown in *Example 9.13*:

```
(defn listen
  ([el type] (listen el type nil))
  ([el type f] (listen el type f (a/chanc)))
  ([el type f out]
   (events/listen el type
                 (fn [e] (when f (f e)) (a/put! out e)))
   out))
```

*Example 9.13: A function to convey events onto a channel*

We can now use the `listen` function defined previously to listen to the `goog.events.EventType.MOUSEMOVE` event type from a particular `div` tag. The values will have to be converted to page offsets, and this can be done using the `getPageOffsetLeft` and `getPageOffsetTop` functions from the `goog.style` namespace. This implementation is described in *Example 9.14*:

```
(defn offset [el]
  [(style/getPageOffsetLeft el) (style/getPageOffsetTop el)])
(let [el (by-id "ex-9-14")]
```

```

out (by-id "ex-9-14-out")
events-chan (listen el goog.events.EventType.MOUSEMOVE)
[left top] (offset el)
location (fn [e]
           { :x (+ (. -offsetX e) (int left))
             :y (+ (. -offsetY e) (int top)) })
(go-loop []
  (if-let [e (a/<! events-chan)]
    (let [loc (location e)]
      (set-html! out (str (:x loc) ", " (:y loc)))
      (recur))))
```

*Example 9.14: Mouse events using channels*

We can also implement a solution to this problem using the `from-event-stream` and `map` functions from the Yolk library. Interestingly, the events produced by the stream returned by the `from-event-target` function will have page offsets of the event stored as the `pageX` and `pageY` properties. This allows us to have a much simpler implementation, as shown in *Example 9.15*:

```

(let [el (by-id "ex-9-15")
      out (by-id "ex-9-15-out")
      events (y/from-event-target el "mousemove")]
  (-> events
        (y/map (juxt (fn [e] (.-pageX e))
                      (fn [e] (.-pageY e))))
        (y/map (fn [[x y]] (str x ", " y)))
        (y/on-value
          #(set-html! out %))))
```

*Example 9.15: Mouse events using FRP*

Both of the implementations shown in *Example 9.14* and *Example 9.15* work as expected, and produce the following output:

686, 915

As a final example, we will simulate several search queries being performed and display the results from the first three queries that return results. The queries can be described as: two queries for web results, two queries for image results, and two queries for video results. We can implement these simulated queries as shown in *Example 9.16*:

```
(defn chan-search [kind]
  (fn [query]
    (go
      (a/<! (a/timeout (rand-int 100)))
      [kind query])))

(def chan-web1 (chan-search :web1))
(def chan-web2 (chan-search :web2))
(def chan-image1 (chan-search :image1))
(def chan-image2 (chan-search :image2))
(def chan-video1 (chan-search :video1))
(def chan-video2 (chan-search :video2))
```

*Example 9.16: Simulating search queries with channels*

The `chan-search` function returns a function that uses the `cljs.core.async/timeout` function to simulate a search query by parking the current task for a random number of milliseconds. Using the `chan-search` function, we create several queries for the different kinds of results we are interested in. Using these functions, we can implement a function to perform all the queries and return the first three results, as shown in *Example 9.17*:

```
(defn chan-search-all [query & searches]
  (let [cs (for [s searches]
             (s query))]
    (-> cs vec a/merge)))

(defn chan-search-fastest [query]
  (let [t (a/timeout 80)
        c1 (chan-search-all query chan-web1 chan-web2)
        c2 (chan-search-all query chan-image1 chan-image2)
        c3 (chan-search-all query chan-video1 chan-video2)
        c (a/merge [c1 c2 c3])]
    (go (loop [i 0
              ret []]
          (if (= i 3)
              ret
              (recur (inc i)
                     (conj ret (alt!
                               [c t] ([v] v))))))))
```

*Example 9.17: Simulating search queries with channels (continued)*

As shown in the preceding example, the `merge` function can be used to combine channels that produce the results of the search queries. Note that the queries to all three types of results, namely web, images, and videos, are timed out after 80 milliseconds. We can bind the `chan-search-fastest` function to the click of a mouse button using the `listen` function we defined earlier, as shown in *Example 9.18*:

```
(let [out (by-id "ex-9-18-out")
      button (by-id "search-1")
      c (listen button goog.events.EventType.CLICK)]
  (go (while true
        (let [e (a/<! c)
              result (a/<! (chan-search-fastest "channels"))
              s (str result)]
          (set-html! out s))))
```

*Example 9.18: Simulating search queries with channels (continued)*

Clicking on the button bound to the `chan-search-fastest` function will

show the following output. Note that the `nil` value in the following output indicates a timeout of all queries for a particular search result type.

```
[[[:image2 "channels"] [:web1 "channels"] nil]]
```

We can just as easily implement an FRP version of the simulation of search queries that was previously described. The queries for the various sources of data are defined as shown in the following *Example 9.19*:

```
(defn frp-search [kind]
  (fn [query]
    (y/later (rand-int 100) [kind query])))

(def frp-web1 (frp-search :web1))
(def frp-web2 (frp-search :web2))
(def frp-image1 (frp-search :image1))
(def frp-image2 (frp-search :image2))
(def frp-video1 (frp-search :video1))
(def frp-video2 (frp-search :video2))
```

### *Example 9.19: Simulating search queries with FRP*

The preceding functions all return event streams for search results. The search results produced can be combined with timeouts using the `later`, `merge`, and `combine-as-array` functions from the `yolk.bacon` namespace, as shown in *Example 9.20*:

```
(defn frp-search-all [query & searches]
  (let [results (map #(% query) searches)
        events (cons (y/later 80 "nil") results)]
    (-> (apply y/merge events)
        (y/take 1)))

(defn frp-search-fastest [query]
  (y/combine-as-array
   (frp-search-all query frp-web1 frp-web2)))
```

```
(frp-search-all query frp-image1 frp-image2)
(frp-search-all query frp-video1 frp-video2)))
```

### *Example 9.20: Simulating search queries with FRP (continued)*

The `frp-search-fastest` function can be invoked on clicking a button, as shown in *Example 9.21*:

```
(let [out (by-id "ex-9-21-out")
      button (by-id "search-2")
      events (y/from-event-target button "click")]
  (-> events
      (y/flat-map-latest #(frp-search-fastest "events"))
      (y/on-value
       #(set-html! out %))))
```

### *Example 9.21: Simulating search queries with FRP (continued)*

The preceding example produces the following output when the search button is clicked:

```
nil, [:image2 "events"], [:video2 "events"]
```

In conclusion, we can use both channels and event streams to implement interactive interfaces in web pages. Although the FRP implementations of the preceding examples are slightly shorter, we can say that both the `core.async` and Yolk libraries have their own elegance.

## Note

The preceding examples are based on code from *Communicating Sequential Processes* by David Nolen (<http://swannodette.github.io/2013/07/12/communicating-sequential-processes/>) and *CSP vs. FRP* by Draco Dormiens (<http://potetm.github.io/2014/01/07/frp.html>).

# Introducing Om

The *Om* library (<https://github.com/omcljs/om>) is a great tool for building dynamic user interfaces in ClojureScript. In fact, it's an interface to *React.js* (<http://facebook.github.io/react/>), which is a JavaScript library for creating interactive user interface components. Om lets us define a user interface as a hierarchy of components, and each component reactively modifies its appearance based on changes to the component's state. In this way, Om components *react* to changes in their state.

## Note

The following library dependencies are required for the upcoming examples:

```
[org.omcljs/om "0.8.8"]
```

Also, the following namespaces must be included in your namespace declaration:

```
(ns my-namespace
  (:require [om.core :as om :include-macros true]
            [om.dom :as dom :include-macros true]))
```

In addition to the preceding dependencies, the following examples also use the `by-id` function from `src/m_clj/c9/common.cljs`. Ensure that the code in the following ClojureScript examples is compiled, using the following command:

```
$ lein cljsbuild once
```

The Om components are generally defined by implementing the `IRender` and `IRenderState` protocols from the `om.core` namespace. The `IRender` protocol declares a single function `render`, and similarly the `IRenderState` protocol declares the `render-state` function. The `render` and `render-state` functions define how a component that implements either of these

protocols is converted to DOM, which can be rendered by a web browser. The implementations of these functions must return a DOM object constructed using functions from the `om.dom` namespace. There are also several other protocols in the `om.core` namespace that allow us to define a component's behavior. Internally, Om uses React.js to perform batched updates to the DOM for the sake of performance, and uses *virtual DOM* to maintain the state of the DOM to be rendered.

## Note

The following examples can be found in `src/m_clj/c9/om/core.cljs` of the book's source code. Also, the HTML page for the following ClojureScript examples can be found in `resources/html/om.html`. The following scripts will be included in this page:

```
<script type="text/javascript" src="../js/out/om.js">
</script>
```

Let's now build a simple component using Om. Suppose we want to build a web application. One of the first steps in doing so is creating a login page for our application. As an example, let's create a simple login form with Om. A user will enter their username and password in this form. The only requirement is that the submit button of this form must be enabled only if the user has entered a username and password. Let's start off by defining some functions to create an input field of a form, as shown in *Example 9.22*:

```
(defn update-input-value-fn [owner]
  (fn [e]
    (let [target (.-target e)
          val (.-value target)
          id (keyword (.-id target))]
      (om/set-state! owner id val)))))

(defn input-field [text owner attrs]
  (let [handler (update-input-value-fn owner)
        event-attr {:onChange handler}
        js-attrs (-> attrs (merge event-attr) clj->js)])
```

```
(dom/div
  nil
  (dom/div nil text)
  (dom/input js-attrs))))
```

### *Example 9.22: A login form using Om*

The `update-input-value-fn` function defined in *Example 9.22* accepts a component `owner` as an argument and returns a function that we can bind to a DOM event. The returned function updates the state of the component with the value of the `.-value` property using the `set-state!` function from the `om.core` namespace. The `input-field` function returns a DOM object for an input field with some associated properties. The `input-field` function also creates an event handler using the `update-input-value-fn` function and binds it to the `onChange` event of the input field.

## Note

Note that a component can change its state or the global application state by using the `set-state!`, `update-state!`, `update!`, or `transact!` functions from the `om.core` namespace.

Next, let's define a form as a component using the `om.core/IRenderState` protocol and `input-field` function, as shown in *Example 9.23*:

```
(defn form [data owner]
  (reify
    om/IInitState
    (init-state [_]
      {:username "" :password ""})
    om/IRenderState
    (render-state [_ state]
      (dom/form
        nil
        (input-field "Username" owner
          {:type "text"
           :id "username"
           :value (:username state)})
        (input-field "Password" owner
```

```

  { :type "password"
    :id "password"
    :value (:password state) })
(dom/br nil)
(dom/input
#js { :type "submit"
      :value "Login"
      :disabled (or (-> state :username empty? )
                    (-> state :password empty? ))))))))

(om/root form nil {:target (by-id "ex-9-23") })

```

*Example 9.23: A login form using Om (continued)*

The preceding `form` function creates a component by implementing the `render-state` function of the `IRenderState` protocol. This component also implements the `IInitState` protocol to define the initial state of the component. The `form` function will render a login form with two input fields, for a username and password, and a login button. The button is enabled only when the username and password are entered. Also, the component is mounted onto a `div` using the `om.core/root` function. The following output in a web page describes the behavior of the component defined by the `form` function:

The figure consists of two side-by-side screenshots of a web application. Both screenshots show a login form within a light gray border. The left screenshot shows an empty form with two input fields labeled 'Username' and 'Password', and a single 'Login' button at the bottom. The right screenshot shows the same form but with values entered: 'user' in the 'Username' field and '\*\*\*' in the 'Password' field. The 'Login' button is now highlighted with a blue border, indicating it is enabled.

The preceding output describes two states of the login form component defined by the `form` function. The login button is observed to be disabled when either the username or password fields are empty, and is enabled only when the user enters values in both of these input fields. In this way,

the login form *reacts* to changes in the state of its input fields.

## Note

Visit <https://github.com/omcljs/om/wiki/Documentation> for complete documentation on all the protocols, functions, and macros in the Om library.

Thus, the Om library provides us with several constructs for creating interactive and stateful components.

# Summary

So far, we have discussed reactive programming through the Pulsar, RxClojure, and Yolk libraries. We have also described several ClojureScript examples that compare channels from the `core.async` library to reactive event streams from the Yolk library. We also demonstrated how we can leverage the Om library to build dynamic user interfaces.

In the following chapter, we will explore how we can test our Clojure programs.

# Chapter 10. Testing Your Code

Testing is an integral part of developing software. Alongside implementing functionality in our software, it is imperative to simultaneously define tests to verify several aspects of it. The Clojure standard library provides several constructs to define tests and mock data. There are also several community libraries that allow us to verify different aspects of the code being tested.

The main advantage of using tests is that they allow us to identify the overall impact of a particular change in a program's code. If we have tests to check the functionality of a program, we can refactor the program with confidence and without the fear of losing any functionality. If there's something that we unavoidably missed while refactoring a program, it will surely be brought to our attention when we run the program's tests. Thus, tests are indispensable tools for keeping code maintainable.

In this chapter, we will study the different ways in which we can write tests in Clojure. We will also discuss how we can perform type checking in Clojure. Although we describe several libraries for writing tests in this chapter, we must note that there are several more available in the Clojure ecosystem. That aside, the libraries described in this chapter are the most mature and battle-hardened tools for testing our code.

## Writing tests

Being a thoughtfully designed language, Clojure has a built-in unit testing library, namely `clojure.test`. Apart from that, there are a couple constructs in the core language that are helpful with regard to testing. Of course, these constructs don't allow us to define and run any tests in the formal sense, and the constructs from the `clojure.test` namespace must be preferred for that purpose.

Let's start off by briefly discussing the constructs from the core language that can be used for unit testing. The `assert` function checks whether an expression evaluates to a truthy value at runtime. This function will throw an exception if the expression passed to it does not evaluate to a truthy value, and the message of this exception can be optionally specified as a second argument to the `assert` form. We can effectively disable all the `assert` forms in a given program by using the global `*assert*` compile time `var`. This variable can only be changed by a top-level `set!` form in a given program or namespace.

Another interesting aspect of testing that is easily tackled by the core language is *mocking* and *stubbing*. In a nutshell, these techniques allow us to redefine the behavior of certain functions within the context of a test case. This is useful in preventing functions from performing unwanted side effects or using unavailable resources. In the Clojure language, this can be done using the `with-redefs` function. This form can be used within tests as well as plain functions, but its usage outside of the scope of tests is not really encouraged. Its semantics are similar to that of the standard `let` form, and you are encouraged to go through the Clojure docs for examples on the `with-redefs` form.

Now, let's explore how we can actually define tests using constructs from the `clojure.test` namespace.

## Defining unit tests

Clojure has support for defining unit tests baked into it. The `clojure.test` namespace, which requires no additional dependencies whatsoever, provides several constructs for testing our code. Let's explore a few of them.

### Note

The following namespaces must be included in your namespace declaration for the upcoming examples:

```
(ns my-namespace
  (:require [clojure.test :refer :all]))
```

The following examples can be found in `test/m_clj/c10/test.clj` of the book's source code.

Tests can be defined using the `deftest` macro. This form must be passed a symbol, indicating the name of the defined test, and any number of expressions. Generally, `is` and `are` forms are used within the `deftest` macro. The `is` form must be passed an expression, and will fail the test if the supplied expression does not return a truthy value. The `are` form must be passed a vector of variable names, a condition to test, and values for the defined variables. For example, the standard `*` function can be tested as shown in *Example 10.1*:

```
(deftest test-*
  (is (= 6 (* 2 3)))
  (is (= 4 (* 1 4)))
  (is (= 6 (* 3 2)))))

(deftest test-*-with-are
  (are [x y] (= 6 (* x y))
    2 3
    1 6
    3 2))
```

### *Example 10.1: Defining tests using the clojure.test namespace*

The preceding code defines two tests using the `is` and `are` forms. We can run tests using the `run-tests` and `run-all-tests` functions from the `clojure.test` namespace. The `run-tests` function can be passed any number of namespaces, and will run all the tests defined in them. Also, this form can be called without passing any arguments, in which case it will run all the tests in the current namespace. The `run-all-tests` function will run all the tests in all namespaces of the current project. It can optionally be passed a regular expression, and will only run the tests from matching namespaces if this argument is supplied. In fact, an IDE with integrated

support for running tests will call these functions. For example, we can run the tests we defined in *Example 10.1* using the `run-tests` function shown here:

```
user> (run-tests)
```

Testing ...

```
Ran 2 tests containing 6 assertions.  
0 failures, 0 errors.  
{:test 2, :pass 6, :fail 0, :error 0, :type :summary}
```

As shown in the preceding output, the `run-tests` function executes both the tests, and both of them pass. Let's now define a test that will fail, although we shouldn't really be doing this unless we have a good reason:

```
(deftest test-*-fails  
  (is (= 5 (* 2 3))))
```

### *Example 10.2: A test that fails*

The test `test-*-fails` shown in *Example 10.2* will fail when it is run, as shown here:

```
user> (run-tests)
```

Testing ...

```
FAIL in (test-*-fails) (test.clj:24)  
expected: (= 5 (* 2 3))  
actual: (not (= 5 6))
```

```
Ran 3 tests containing 7 assertions.  
1 failures, 0 errors.  
{:test 3, :pass 6, :fail 1, :error 0, :type :summary}
```

In fact, defining tests that fail should be considered a part and parcel of developing a program. To start a feature or fix a bug in a program, we must first define a test that validates this change (by failing!). We should then proceed to implement the feature or fix, such that all the newly defined

tests pass. These two steps are then repeated, until all the requirements of our feature or fix are met. This is the essence of **test-driven development (TDD)**.

## Note

We can also run the tests defined in a given namespace using the following command:

```
$ lein test my-namespace
```

The `clojure.test` namespace must be used for testing programs written strictly in Clojure. For testing ClojureScript programs in the same way, we can use the *doo* library (<https://github.com/bensu/doo>), which provides ClojureScript implementations of the `deftest`, `is`, and `are` constructs.

## Using top-down testing

A more powerful way to define tests in Clojure is by using the *Midje* library (<https://github.com/marick/Midje>). This library provides several constructs that allow us to easily define unit tests by describing relationships between several functions, rather than describing the implementation of the functions themselves. This approach is also called *top-down testing*, and Midje champions this kind of testing methodology. Let's dive into the details of the Midje library.

## Note

The following library dependencies are required for the upcoming examples:

```
[midje "1.8.2"]
```

We must also include the following dependencies in the `:plugins` section of your `project.clj` file:

```
[lein-midje "3.1.3"]
```

Also, the following namespaces must be included in your namespace declaration:

```
(ns my-namespace
  (:require [midje.sweet :refer :all]
            [midje.repl :as mr]))
```

The following examples can be found in `test/m_clj/c10/midje.clj` of the book's source code.

Firstly, let's define a simple function that we intend to test, as shown in *Example 10.3*:

```
(defn first-element [sequence default]
  (if (empty? sequence)
    default
    (first sequence)))
```

*Example 10.3: A simple function to test*

We can define tests for the `first-element` function using the `facts` and `fact` constructs from the `midje.sweet` namespace, as shown in *Example 10.4*.

```
(facts "about first-element"
  (fact "it returns the first element of a collection"
    (first-element [1 2 3] :default) => 1
    (first-element '(1 2 3) :default) => 1)

  (fact "it returns the default value for empty collections"
    (first-element [] :default) => :default
    (first-element '() :default) => :default
    (first-element nil :default) => :default
    (first-element
      (filter even? [1 3 5])
      :default) => :default))
```

*Example 10.4: Tests for the first-element function*

As shown in the preceding code, the `fact` form describes a test, and can be

passed any number of clauses. Each clause is comprised of an expression, a `=>` symbol, and the expected return value of the supplied expression. The `facts` form is simply used to group together several `fact` forms. It's quite apparent that instead of checking logical conditions, we use `fact` forms to check expressions and the values returned by them.

The `provided` form can be used to mock function calls. The Midje library allows us to use *metaconstants* in our tests, and they are often used with the `provided` form. Metaconstants can be thought of as generic placeholders for values and functions. All metaconstants should start and end with two or more dots (.) or hyphens (-); hyphens are more suitable for metaconstants representing functions. For example, we can test the `first-element` function we defined earlier using metaconstants and the `provided` form as shown in *Example 10.5*:

```
(fact "first-element returns the first element of a collection"
      (first-element ..seq.. :default) => :default
      (provided
        (empty? ..seq..) => true))
```

### *Example 10.5: Using the provided form and metaconstants*

In the test shown previously, the metaconstant `..seq..` is used to indicate the first argument passed to the `first-element` function, and the `provided` form mocks the call to the `empty?` function. This way, we can implement tests without completely implementing the functions being tested. Of course, we should avoid mocking or redefining standard functions in the `provided` form. For example, suppose we have three partially implemented functions, as shown in *Example 10.6*.

```
(defn is-diesel? [car])

(defn cost-of-car [car])

(defn overall-cost-of-car [car]
  (if (is-diesel? car)
    (* (cost-of-car car) 1.4))
```

```
(cost-of-car car)))
```

### *Example 10.6: Partially implemented functions to test*

Notice that only the `overall-cost-of-car` function is completely implemented in the preceding code. Nevertheless, we can still test the relation between these three functions using the Midje library, as shown in *Example 10.7*.

```
(fact
  (overall-cost-of-car ..car..) => (* 5000 1.4)
  (provided
    (cost-of-car ..car..) => 5000
    (is-diesel? ..car..) => true))
```

### *Example 10.7: Testing the `is-diesel?`, `cost-of-car` and `overall-cost-of-car` functions*

In the test shown previously, the `cost-of-car` and `is-diesel?` functions are mocked using the `provided` form and the `..car..` metaconstant, and the value returned by the `overall-cost-of-car` function is checked. We can run all of the tests we have defined so far using the `autotest` function from the `midje.repl` namespace, as shown here:

```
user> (mr/autotest :files "test")
=====
===
Loading ( ... )
>>> Output from clojure.test tests:

0 failures, 0 errors.
>>> Midje summary:
All checks (8) succeeded.
[Completed at ... ]
```

## Note

We can also run the tests defined in a given namespace using the following

command. Note that the following command will watch your project for file changes, and will run the tests in any files once they are changed:

```
$ lein midje :autotest test
```

In this way, we can use the Midje library to write tests, even for functions that haven't been completely implemented. Midje allows us to describe tests as relations between functions using metaconstants. In summary, the `clojure.test` and Midje libraries are great tools for defining unit tests.

# Testing with specs

We will now take a look at the Speclj, pronounced *speckle*, library (<https://github.com/slagyrl/speclj>), which is used to write *specs*. Specs are similar to unit tests, but are focused on the behavior of functions being tested, rather than their internal implementation. In fact, **behavior-driven development (BDD)** is centered about writing specs.

The main difference between TDD and BDD is that BDD focuses on the behavior or specifications of functions, rather than their implementation. From this perspective, if we change the internal implementation of a function that has been previously tested, there is a smaller chance that we have to modify the tests, or rather specs, associated with the function. BDD can also be thought of as a refined approach to TDD, in which the interface and behavior of a function is more important than its internal implementation. Now, let's study the various constructs of the Speclj library.

## Note

The following library dependencies are required for the upcoming examples. We must also include the following dependencies in the :plugins section of your project.clj file:

```
[speclj "3.3.1"]
```

Also, the following namespaces must be included in your namespace declaration:

```
(ns my-namespace
  (:require [speclj.core :refer :all]))
```

The `describe`, `it`, and `should` forms, from the `speclj.core` namespace, are used to define specs for a given function. The `it` form represents a single specification for the function being tested, and the `describe` form is used

to group together several specs together. Assertions within an `it` form can be expressed using the `should` form and its variants. For example, we can write a spec for the behavior of the standard `*` function, as shown in the following *Example 10.8*.

## Note

The following examples can be found in `spec/m_clj/c10/specclj.clj` of the book's source code.

```
(describe "*"
  (it "2 times 3 is 6"
    (should (= 6 (* 2 3)))))
```

### *Example 10.8: A spec for the `*` function*

The spec shown previously checks a single condition using the `should` and `=` forms. There are several variants of the `should` form, such as `should=`, `should-not`, `should-fail`, and `should-throw`. These forms are pretty much self-explanatory, and you are encouraged to go through the Speclj docs for more details. We can describe some specs for the standard `/` function, as shown in *Example 10.9*.

```
(describe "/"
  (it "5 divided by 5 is 1"
    (should= 1 (/ 5 5)))
  (it "5 divided by 5 is not 0"
    (should-not= 0 (/ 5 5)))
  (it "fail if 5 divided by 5 is not 1"
    (if (not= 1 (/ 5 5))
      (should-fail "divide not working")))
  (it "throw an error if 5 is divided by 0"
    (should-throw ArithmeticException
      (/ 5 0))))
```

### *Example 10.9: Specs for the `/` function using several `it` forms*

Within a `describe` form, we can use the `before` and `after` forms to execute arbitrary code before or after each `it` form is checked. Similarly,

the `before-all` and `after-all` forms can specify what to execute before and after all the specs are checked in a `describe` form.

Input and output performed by a certain function can be described using specs. This is done using the `with-out-str` and `with-in-str` forms. The `with-out-str` form returns whatever data is sent to standard output by a given expression. Conversely, the `with-in-str` form must be passed a string and an expression, and the supplied string will be sent to the standard input once the supplied expression is called. For example, let's say we have a simple function that reads a string and prints it. We can write a spec for such a function using the `with-out-str` and `with-in-str` forms as shown in *Example 10.10*:

```
(defn echo []
  (let [s (read-line)]
    (println (str "Echo: " s)))))

(describe "echo"
  (it "reads a line and prints it"
    (should= "Echo: Hello!\r\n"
      (with-out-str
        (with-in-str "Hello!"
          (echo))))))
```

*Example 10.10: A spec for a function that reads a string and prints it*

We can also mock function calls within an `it` form using the standard `with-redefs` macro we described earlier. For example, we can write a spec for the `echo` function described in *Example 10.10* by mocking the `read-line` and `println` functions as shown in *Example 10.11*. Obviously, it's not advisable to mock standard functions, and it's only done here to depict the usage of the `with-redefs` macro within a spec.

```
(describe "echo"
  (it "reads a line and prints it"
    (with-redefs [read-line (fn [] "Hello!")
                 println (fn [x] x)]
      (should= "Echo: Hello!" (echo)))))
```

### *Example 10.11: Using the with-redefs macro within a spec*

To run all the specs defined in a given project, we can call the `run-specs` macro, as shown here:

```
user> (run-specs)
...
Finished in 0.00547 seconds
7 examples, 0 failures
#<speclj.run.standard.StandardRunner 0x10999>
```

## Note

We can also run the specs defined in a given namespace using the following command. Note that the following command will watch your project for file changes, and will run the specs in any files once they are changed:

```
$ lein spec -a
```

To summarize, the Speclj library provides us with several constructs to define specs for BDD. Specs for a given function should be modified only when the required functionality or behavior of a function must be changed. With specs, there's less of a chance that modifying the underlying implementation of a function will require a change in its associated specs. Of course, the question of whether you should use specs or tests in your project is a subjective one. Some projects do fine with simple tests, and others prefer to use specs.

# Generative testing

Another form of testing is **generative testing**, in which we define properties of functions that must hold true for all inputs. This is quite different compared to enumerating the expected inputs and outputs of functions, which is essentially what unit tests and specs do. In Clojure, generative testing can be done using the `test.check` library (<https://github.com/clojure/test.check>). This library is inspired by Haskell's QuickCheck library, and provides similar constructs for testing properties of functions.

## Note

The following library dependencies are required for the upcoming examples:

```
[org.clojure/test.check "0.9.0"]
```

Also, the following namespaces must be included in your namespace declaration:

```
(ns my-namespace
  (:require [clojure.test.check :as tc]
            [clojure.test.check.generators :as gen]
            [clojure.test.check.properties :as prop]
            [clojure.test.check.clojure-test
             :refer [defspec]]))
```

The following examples can be found in `src/m_clj/c10/check.clj` of the book's source code.

To define a property to check, we can use the `for-all` macro from the `clojure.test.check.properties` namespace. This macro must be passed a vector of generator bindings, which can be created using constructs from the `clojure.test.check.generators` namespace, along with a property to verify. For example, consider the properties defined in *Example 10.12*:

```
(def commutative-mult-prop
  (prop/for-all [a gen/int
                b gen/int]
    (= (* a b)
       (* b a)))))

(def first-is-min-after-sort-prop
  (prop/for-all [v (gen/not-empty (gen/vector gen/int)) ]
    (= (apply min v)
       (first (sort v)))))
```

*Example 10.12: Simple properties defined using the test.check library*

In the preceding code, we have defined two properties, namely `commutative-mult-prop` and `first-is-min-after-sort-prop`. The `commutative-mult-prop` property asserts that a multiplication operation using the `*` function is commutative, and the `first-is-min-after-sort-prop` function checks whether the first element of a vector of integers sorted using the `sort` function is the smallest value in the vector. Note the use of the `int`, `vector` and `non-empty` functions from the `clojure.test.check.generators` namespace. We can verify that these properties are true using the `quick-check` function from the `clojure.test.check` namespace, as shown here:

```
user> (tc/quick-check 100 commutative-mult-prop)
{:result true, :num-tests 100, :seed 1449998010193}
user> (tc/quick-check 100 first-is-min-after-sort-prop)
{:result true, :num-tests 100, :seed 1449998014634}
```

As shown previously, the `quick-check` function must be passed the number of checks to run and a property to verify. This function returns a map describing the checks performed on the supplied properties, in which the value of the `:result` key indicates the outcome of the test. It's fairly evident that both of the properties `commutative-mult-prop` and `first-is-min-after-sort-prop` hold true for the specified type of inputs. Now, let's define a property that is not true, as shown in *Example 10.13*:

```
(def commutative-minus-prop
  (prop/for-all [a gen/int
                b gen/int]
    (= (- a b)
       (- b a)))))
```

*Example 10.13: A property that won't be true defined using the test.check library*

Running the preceding check will obviously fail, as shown in the following output:

```
user> (tc/quick-check 100 commutative-minus-prop)
{:result false, :seed 1449998165908,
 :failing-size 1, :num-tests 2, :fail [0 -1],
 :shrunk {:total-nodes-visited 1, :depth 0, :result false,
           :smallest [0 -1]}}
```

We can also define specs based on generative testing using the `defspec` macro from the `clojure.test.check.clojure-test` namespace. This form must be passed the number of checks to perform and a property, which is analogous to the `quick-check` function. Specs defined using the `defspec` form will be checked by the standard `clojure.test` runner. For example, we can define the `commutative-mult-prop` property as a spec as shown in *Example 10.14*:

```
(defspec commutative-mult 100
  (prop/for-all [a gen/int
                b gen/int]
    (= (* a b)
       (* b a))))
```

*Example 10.14: A spec defined using the defspec macro*

The spec defined in the preceding code can be verified by calling the `run-tests` or `run-all-tests` functions from the `clojure.test` namespace, or by running the `lein test` Leiningen command. In conclusion, generative testing through the `test.check` library is yet another way to test our code.

It focuses on specifying properties of functions rather than describing the expected output of functions for some input.

# Testing with types

Type checking is something that is often taken for granted in statically typed languages. With type checking, type errors can be found at compile time, rather than during runtime. In some dynamic languages such as Clojure, type signatures can be declared wherever and whenever they are required, and this technique is termed as *optional typing*. Type checking can be done using the `core.typed` library

(<https://github.com/clojure/core.typed>). Using `core.typed`, the type signature of a var can be checked using *type annotations*. Type annotations can be declared for any var, which includes values created using a `def` form, a `binding` form, or any other construct that creates a var. In this section, we will explore the details of this library.

## Note

The following library dependencies are required for the upcoming examples.

```
[org.clojure/core.typed "0.3.0"]
```

Also, the following namespaces must be included in your namespace declaration.

```
(ns my-namespace
  (:require [clojure.core.typed :as t]))
```

The following examples can be found in `src/m_clj/c10/typed.clj` of the book's source code.

Type annotations for vars are declared using the `ann` macro from the `clojure.core.typed` namespace. This form must be passed an expression to annotate and a vector of types. For example, a type annotation for a function that accepts two numbers as arguments and returns a number is shown in *Example 10.15*.

```
(t/ann add [Number Number -> Number])
(defn add [a b]
  (+ a b))
```

*Example 10.15: A type annotation for a function that accepts two numbers and returns a number*

To check all the type annotations in the given namespace, we must call the `clojure.core.typed/check-ns` function by passing it the namespace to be checked, as shown here:

```
user> (t/check-ns 'my-namespace)
Start collecting my-namespace
Finished collecting my-namespace
Collected 2 namespaces in 200.965982 msecs
Start checking my-namespace
Checked my-namespace in 447.580402 msecs
Checked 2 namespaces in 650.979682 msecs
:ok
```

As shown previously, the `check-ns` function prints some information about the namespaces being checked, and returns the keyword `:ok` if all type checks in the specified namespace have passed. Now, let's change the definition of the `add` function we previously defined as shown in *Example 10.16*:

```
(defn add [a b]
  (str (+ a b)))
```

*Example 10.16: Redefining the add function*

Although the preceding definition is valid, it will not be passed by the type checker, as shown here:

```
user> (t/check-ns 'my-namespace)
Start collecting my-namespace
Finished collecting my-namespace
Collected 2 namespaces in 215.705251 msecs
Start checking my-namespace
Checked my-namespace in 493.669488 msecs
```

```
Checked 2 namespaces in 711.644548 msecs
Type Error (m_clj/c1/typed.clj:23:3) Type mismatch:

Expected: Number

Actual: String
in: (str (clojure.lang.Numbers/add a b))
ExceptionInfo Type Checker: Found 1 error clojure.core/ex-info
(core.clj:4403)
```

The `check-ns` function throws an error stating that a `String` type was found where a `Number` type was expected. In this way, the `check-ns` function can find type errors in functions that have been annotated with the `ann` macro. Functions with multiple arities can be annotated using the `IFn` construct from the `clojure.core.typed` namespace, as shown in *Example 10.17*:

```
(t/ann add-abc
        (t/IFn [Number Number -> Number]
               [Number Number Number -> Number]))
(defn add-abc
  ([a b]
   (+ a b))
  ([a b c]
   (+ a b c)))
```

### *Example 10.17: Annotating functions with multiple arities*

We can also annotate functions with variadic arguments using the `*` symbol in the vector of types passed to the `ann` macro, as shown in *Example 10.18*.

```
(t/ann add-xs [Number * -> Number])
(defn add-xs [& xs]
  (apply + xs))
```

### *Example 10.18: Annotating functions with variadic arguments*

In the REPL, we can determine the inferred type of an expression or a

value using the `cf` macro from the `clojure.core.typed` namespace. This macro can also be passed the expected type as the second argument. Note that the `cf` form is only for experimentation and should not be used in type annotations. The `cf` form returns an inferred type, along with a structure called a *filter set*, which is represented as a map. For example, the type and filter sets of the values `nil`, `true`, and `false` can be inferred using the `cf` form as shown here:

```
user> (t/cf nil)
[nil {:then ff, :else tt}]
user> (t/cf true)
[true {:then tt, :else ff}]
user> (t/cf false)
[false {:then ff, :else tt}]
```

In the preceding output, the second value in each of the vectors returned by the `cf` macro represents the filter set derived from the supplied expression. A filter set can be described as a collection of the two filters:

- The `:then` filter, which is true if the expression is a truthy value
- The `:else` filter, which is true if the expression is not a truthy value

In the context of filter sets, there are two *trivial filters*, namely `tt` and `ff`, which can be described as follows:

- `tt`, which translates to trivially true and means the value is truthy.
- `ff`, which translates to *forever false* and means the value is not truthy. This filter is also termed as the *impossible filter*.

In this perspective, the filter set `{:then tt, :else ff}` translates into "the expression could be a truthy value, but it is impossible for it to be a non-truthy value". Thus, false values such as `nil` and `false` are never true as inferred by the `cf` form, which agrees with the semantics of these values in Clojure. Truthy values will always have `tt` as the `:then` filter, as shown in the following output:

```
user> (t/cf "Hello")
[(t/Val "Hello") {:then tt, :else ff}]
```

```
user> (t/cf 1)
[(t/Val 1) {:then tt, :else ff}]
user> (t/cf :key)
[(t/Val :key) {:then tt, :else ff}]
```

The `cf` macro can also be used to check the type signature of functions, as shown here:

```
user> (t/cf str)
[t/Any * -> String]
user> (t/cf +)
(t/IFn [Long * -> Long]
  [(t/U Double Long) * -> Double]
  [t/AnyInteger * -> t/AnyInteger]
  [Number * -> Number])
```

A form or expression can be annotated with an expected type using the `ann-form` macro, as shown here:

```
user> (t/cf (t/ann-form #(inc %) [Number -> Number]))
[[Number -> Number] {:then tt, :else ff}]
user> (t/cf (t/ann-form #(str %) [t/Any -> String]))
[[t/Any -> String] {:then tt, :else ff}]
```

Aggregate types such as lists and vectors also have types defined for them in the `clojure.core.typed` namespace. We can infer the types of these data structures using the `cf` macro, as shown here:

```
user> (t/cf (list 0 1 2))
(PersistentList (t/U (t/Val 1) (t/Val 0) (t/Val 2)))
user> (t/cf [0 1 2])
[(t/HVec [(t/Val 0) (t/Val 1) (t/Val 2)]) {:then tt, :else ff}]
```

The types `PersistentList` and `HVec` in the preceding output are concrete types for a list and a vector respectively. We can also pass the expected type as an extra argument to the `cf` form as shown here:

```
user> (t/cf (list 0 1 2) (t/List t/Num))
(t/List t/Num)
user> (t/cf [0 1 2] (t/Vec t/Num))
(t/Vec t/Num)
```

```
user> (t/cf {:a 1 :b 2} (t/Map t/Keyword t/Int))
(t/Map t/Keyword t/Int)
user> (t/cf #{0 1 2} (t/Set t/Int))
(t/Set t/Int)
```

The `core.typed` library also supports *parameterized types*, *union types*, and *intersection types*. Union types are declared using the `U` construct, and intersection types are declared using the `I` construct. Intersection types are meant to be used with protocols, which implies that the intersection type `(I A B)` must implement both the protocols `A` and `B`. On the other hand, union types can be defined using concrete types. For example, the `clojure.core.typed` namespace defines a parameterized `Option` type, which is simply a union of `nil` and the parameterized type. In other words, the type `(Option x)` is defined as the union type `(U x nil)`. Another good example of a union type is the `AnyInteger` type, which represents a whole number, and is defined in the `clojure.core.typed` namespace as shown in *Example 10.19*.

```
(defalias AnyInteger
  (U Integer Long clojure.lang.BigInt BigInteger Short Byte))
```

### *Example 10.19: The AnyInteger union type*

*Polymorphic types* are also supported by the `core.typed` library, which allow us to specify generalized types. For example, the `identity` and `iterate` functions have polymorphic type signatures, as shown here:

```
user> (t/cf identity)
(t/All [x] [x -> x :filters ... ])
user> (t/cf iterate)
(t/All [x] [[x -> x] x -> (t/ASeq x)])
```

We can annotate functions with polymorphic type signatures using the `All` construct from the `clojure.core.typed` namespace, as shown in *Example 10.20*.

```
(t/ann make-map (t/All [x] [x -> (t/Map t/Keyword x)]))
(defn make-map [a]
```

```
{ :x a })
```

### *Example 10.20: Defining a polymorphic type signature*

In conclusion, the `core.typed` library provides several constructs to define and verify type signatures of vars. There are also several constructs for determining the type signature of a given expression. Using `core.typed`, you can find logical type errors in your code before it is executed at runtime. Type annotations can also be thought of as a form of documentation, which concisely describe the types of functions and vars. Thus, there are several benefits of using types through the `core.typed` library in Clojure.

# Summary

So far, we have discussed several libraries that can help us test and verify our code. We talked about the `clojure.test` and Midje libraries for defining tests. We also explored how we can define specs in the spirit of BDD using the Speclj library. Generative testing is another approach to testing, and we demonstrated how it can be done using the `test.check` library. Lastly, we talked about how we can perform type checking in Clojure using the `core.typed` library. Hence, there is a wide array of options for testing our code in Clojure.

In the next and final chapter, we will talk about how we can troubleshoot our code, as well as some good practices for developing applications in Clojure.

# Chapter 11. Troubleshooting and Best Practices

By now, you must be aware of all the features and most of the constructs of the Clojure language. Before you start building your own applications and libraries in Clojure, we will briefly discuss a few techniques to troubleshoot your code and some practices that you should incorporate in your projects.

## Debugging your code

Along your journey of building applications and libraries in Clojure, you'll surely run into situations where it would be helpful to debug your code. The usual response to such a situation is to use an **Integrated Development Environment (IDE)** with a debugger. And while Clojure IDEs such as *CIDER* (<https://github.com/clojure-emacs/cider>) and *Counterclockwise* (<http://doc.ccw-ide.org>) do support debugging, there are a few simpler constructs and tools that we can use to troubleshoot our code. Let's have a look at a few of them.

One of the easiest ways to debug your code is by printing the value of some variables used within a function. We could use the standard `println` function for this purpose, but it doesn't always produce the most readable output for complex data types. As a convention, we should use the `clojure.pprint/pprint` function to print variables to the console. This function is the standard pretty-printer of the Clojure language.

### Note

Macros can be quite bewildering to debug. As we mentioned in [Chapter 4, Metaprogramming with Macros](#), macros should be used sparingly and we can debug macros using macroexpansion constructs such as `macroexpand`

and `macroexpand-all`.

Apart from these built-in constructs, there are a couple of useful libraries that we can add to our debugging toolkit.

## Note

The following examples can be found in `test/m_clj/c11/ debugging.clj` of the book's source code.

# Using tracing

*Tracing* can be used to determine when and how a form is called. The `tools.trace` contrib library (<https://github.com/clojure/tools.trace>) provides some handy constructs for tracing our code.

## Note

The following library dependencies are required for the upcoming examples:

```
[org.clojure/tools.trace "0.7.9"]
```

Also, the following namespaces must be included in your namespace declaration:

```
(ns my-namespace
  (:require [clojure.tools.trace :as tr]))
```

The `trace` function, from the `clojure.tools.trace` namespace, is the most elementary way to trace an expression. It will simply print the value returned by the expression passed to it. The `trace` construct can also be passed a string, with which the trace can be tagged, as an additional argument. For example, suppose we have to trace the function defined in *Example 11.1*:

```
(defn make-vector [x]
  [x])
```

### *Example 11.1: A simple function to trace*

We can trace the expression `(make-vector 0)` using the `trace` function shown here:

```
user> (tr/trace (make-vector 0))
TRACE: [0]
[0]
user> (tr/trace "my-tag" (make-vector 0))
TRACE my-tag: [0]
[0]
```

We can trace all the functions defined in a namespace by passing the namespace to the `trace-ns` macro, which is defined in the `clojure.tools.trace` namespace. Similarly, specific functions or vars in a namespace can be traced using the `trace-vars` macro. Traces added using these forms can be removed using the `untrace-ns` and `untrace-vars` macros. If we want to determine which expression among several ones is failing, we can pass the expressions to the `trace-forms` macro, shown here:

```
user> (tr/trace-forms (+ 10 20) (* 2 3) (/ 10 0))
ArithmaticException Divide by zero
  Form failed: (/ 10 0)
clojure.lang.Numbers.divide (Numbers.java:158)
```

As the preceding output shows, the `trace-forms` macro will print the form that causes the error. A more informative way to trace a function is by replacing the `defn` symbol in its definition by `clojure.tools.trace/deftrace`, which simply defines a function whose arguments and return value will be traced. For example, consider the function defined in the following *Example 11.2*:

```
(tr/deftrace add-into-vector [& xs]
  (let [sum (apply + xs)]
    [sum]))
```

### *Example 11.2: Tracing a function using the deftrace macro*

On calling the `add-into-vector` function defined previously, the following trace will be printed:

```
user> (add-into-vector 10 20)
TRACE t9083: (add-into-vector 10 20)
TRACE t9083: => [30]
[30]
```

In this way, tracing can be used to find the value returned by an expression during the execution of a program. The tracing constructs from the `tools.trace` namespace allow us to determine when a function is called, as well as what its return value and arguments are.

## Using Spyscope

As you may have already been thinking, you can easily implement your own debugging constructs using macros. The Spyscope library (<https://github.com/dgrnbrg/spyscope>) takes this approach and implements a few reader macros for debugging code. The use of reader macros for debugging is a more favorable approach for languages with the parentheses-flavored syntax of Lisps. This is because, in these languages, reader macros that print debugging information can be added more easily to an existing program compared to forms such as `trace` and `deftrace`. Let's explore the constructs of the Spyscope library to get a clearer idea of the advantage of debugging code with reader macros.

The Spyscope library provides the `#spy/p`, `#spy/d`, and `#spy/t` reader macros, which can all be used by writing them immediately before expressions that have to be debugged. It is a common practice to have these forms made available in the REPL using the `:injections` section of the `project.clj` file in a Leiningen project.

### Note

The following library dependencies are required for the upcoming examples:

```
[spyscope "0.1.5"]
```

We must also include the following forms as a vector in the :injections section of your `project.clj` file:

```
(require 'spyscope.core)
```

Also, the following namespaces must be included in your namespace declaration:

```
(ns my-namespace
  (:require [spyscope.repl :as sr]))
```

The `#spy/p` reader macro can be used to print a value that is used within an expression. An interesting point about this construct is that it is implemented using the `clojure pprint/pprint` function. For example, we can print out the intermediate values produced by a `take` form shown here:

```
user> (take 5 (repeatedly
                  #(let [r (rand-int 100)]
                     #spy/p r)))
95
36
61
99
73
(95 36 61 99 73)
```

To produce more detailed information, such as the call stack and the form that returns a value, we can use the `#spy/d` reader macro. For example, we can use this construct to produce the following information:

```
user> (take 5 (repeatedly
                  #(let [r (rand-int 100)]
                     #spy/d (/ r 10.0)))
user$eval9408$fn__9409.invoke(form-init1..0.clj:2) (/ r 10.0) =>
4.6
user$eval9408$fn__9409.invoke(form-init1..0.clj:2) (/ r 10.0) =>
4.4
user$eval9408$fn__9409.invoke(form-init1..0.clj:2) (/ r 10.0) =>
```

```
5.0
user$eval9408$fn__9409.invoke(form-init1..0.clj:2) (/ r 10.0) =>
7.8
user$eval9408$fn__9409.invoke(form-init1..0.clj:2) (/ r 10.0) =>
3.1
(4.6 4.4 5.0 7.8 3.1)
```

The `#spy/d` reader macro also supports several options, which can be passed to it as metadata. The `:fs` key of this metadata map specifies the number of stack frames to display. Also, the `:marker` key can be used to declare a string tag for a form. We can use these options to display information from the call stack of a form, shown here:

```
user> (take 5 (repeat #spy/d ^{:fs 3 :marker "add"}
(+ 0 1 2)))
-----
clojure.lang.Compiler.eval(Compiler.java:6745)
clojure.lang.Compiler.eval(Compiler.java:6782)
user$eval9476.invoke(form-init1..0.clj:1) add (+ 0 1 2) => 3
(3 3 3 3 3)
```

The preceding output shows the top three stack frames of a call to the `+` form. We can also filter out stack frames from the call stack information using the `:nses` key with a regular expression, shown here:

```
user> (take 5 (repeat #spy/d ^{:fs 3 :nses #"core|user"}
(+ 0 1 2)))
-----
clojure.core$apply.invoke(core.clj:630)
clojure.core$eval.invoke(core.clj:3081)
user$eval9509.invoke(form-init1..0.clj:1) (+ 0 1 2) => 3
(3 3 3 3 3)
```

To skip printing the form being debugged, we can specify the `:form` key with a `false` value in the metadata map specified to the `#spy/d` reader macro, and this key defaults to `true`. We can also print out the time at which a form is called using the `:time` key. The value for this key can either be `true`, in which case the default time format is used, or a string such as `"hh:mm:ss"`, which represents the timestamp format with which the

time must be displayed.

The `#spy/t` reader macro is used for tracing a form, and this construct supports the same options as the `#spy/d` reader macro. The trace is not printed immediately, and can be displayed using the `trace-query` function from the `spyscope.repl` namespace. For example, consider the function in *Example 11.3* that adds a number of values in a future:

```
(defn add-in-future [& xs]
  (future
    #spy/t (apply + xs)))
```

*Example 11.3: Tracing a function that adds numbers in a future*

Once the `add-in-future` function is called, we can display a trace of the call using the `trace-query` function, shown here:

```
user> (sr/trace-next)
nil
user> (def f1 (add-in-future 10 20))
#'user/f1
user> (def f2 (add-in-future 20 30))
#'user/f2
user> (sr/trace-query)
user$add_in_future$fn_..7.invoke(debugging.cljs:66) (apply + xs)
=> 30
-----
user$add_in_future$fn_..7.invoke(debugging.cljs:66) (apply + xs)
=> 50
nil
```

In the preceding output, the `trace-next` function is used to start a new *generation* of traces. Traces in the Spyscope library are grouped into generations, and a new generation can be started using the `spyscope.repl/trace-next` function. All trace information from all generations can be cleared using the `trace-clear` function from the `spyscope.repl` namespace. We can also pass an argument to the `trace-query` function to filter out results. This argument can be either a number,

which represents the number of recent generations to show, or a regex to filter traces by their namespaces.

To summarize, there are several ways to debug your code in Clojure without the use of a debugger. The `tools.trace` and `Spyscope` libraries have several useful and simple constructs for debugging and tracing the execution of Clojure code.

# Logging errors in your application

Another way to analyze what went wrong in an application is by using logs. Logging can be done using the `tools.logging` contrib library. This library lets us use multiple logging implementations through an agnostic interface, and the implementations to choose from include `slf4j`, `log4j`, and `logback`. Let's quickly skim over how we can add logging to any Clojure program using the `tools.logging` library and `logback`, which is arguably the most recent and configurable implementation to use with this library.

## Note

The following library dependencies are required for the upcoming examples:

```
[org.clojure/tools.logging "0.3.1"]
[ch.qos.logback/logback-classic "1.1.3"]
```

Also, the following namespaces must be included in your namespace declaration:

```
(ns my-namespace
  (:require [clojure.tools.logging :as log]))
```

The following examples can be found in `test/m_clj/c11/logging.clj` of the book's source code.

All the logging macros implemented in the `clojure.tools.logging` namespace fall into two categories. The first category of macros require arguments like those which are passed to the `println` form. All of these arguments are concatenated and written to the log. The other category of macros must be passed a format string and values to interpolate into the specified format. This second category of macros are generally suffixed with an `f` character, such as `debugf` or `infof`. The logging macros in the

`tools.logging` library can be passed an exception followed by the other usual arguments.

The macros in the `tools.logging` library write log messages at differing log levels. For example, the `debug` and `debugf` forms write log messages at the `DEBUG` level, and similarly, the `error` and `errorf` macros log at the `ERROR` level. In addition, the `spy` and `spfy` macros will evaluate and return the value of an expression, and may log the result if the current log level is equal to or below the log level specified to it, which defaults to `DEBUG`.

For example, the `divide` function, shown in the following *Example 11.4*, logs some information, using the `info`, `spfy`, and `error` macros, while performing integer division:

```
(defn divide [a b]
  (log/info "Dividing" a "by" b)
  (try
    (log/spfy "Result: %s" (/ a b))
    (catch Exception e
      (log/error e "There was an error!"))))
```

*Example 11.4: A function that logs information using the tools.logging library*

The following log messages will be written when the `divide` function is called:

```
user> (divide 10 1)
INFO - Dividing 10 by 1
DEBUG - Result: 10
10
user> (divide 10 0)
INFO - Dividing 10 by 0
ERROR - There was an error!
java.lang.ArithmetricException: Divide by zero
at clojure.lang.Numbers.divide(Numbers.java:158) ~[clojure-1.7.0.jar:na]
...
at java.lang.Thread.run(Thread.java:744) [na:1.7.0_45]
```

nil

As shown previously, the `divide` function writes several log messages at different log levels when it is called. The logging configuration for `logback` must be saved in a file named `logback.xml`, which can reside in either the `src/` or `resources/` directories of a Leiningen project. We can specify the default log level and several other options for `logback` in this file.

## Note

If you're interested in the logging configuration for the previous examples, take a look at the `src/logback.xml` file in the book's source code. For detailed configuration options, visit

<http://logback.qos.ch/manual/configuration.html>.

It is also handy to have a global exception handler that logs exceptions for all threads in a program. This can be particularly useful for checking errors that are encountered during the execution of `go` and `thread` macros from the `core.async` library. Such a global exception handler can be defined using the `setDefaultUncaughtExceptionHandler` method from the `java.lang.Thread` class, as shown in *Example 11.5*:

```
(Thread/setDefaultUncaughtExceptionHandler
  (reify Thread$UncaughtExceptionHandler
    (uncaughtException [_ thread ex]
      (log/error ex "Uncaught exception on" (.getName thread)))))
```

*Example 11.5: A global exception handler that logs all errors*

## Note

You can also use *Timbre* (<https://github.com/ptaoussanis/timbre>) for logging, which can be configured without the use of XML and is also supported on ClojureScript.

In conclusion, there are several options for logging available to us through

the `tools.logging` library. This library also supports several logging implementations that each have their own set of configuration options.

# Thinking in Clojure

Let's briefly discuss a handful of good practices for building real world applications in Clojure. Of course, these practices are only guidelines, and you should eventually try to establish your own set of rules and practices for writing code in Clojure:

- **Minimize state and use pure functions:** Most applications must inevitably use some form of state. You must always strive to reduce the amount of state you're dealing with, and implement most of the heavy lifting in pure functions. State can be managed using reference types, channels, or even monads in Clojure, thus giving us a lot of proven options. In this way, we can reduce the number of conditions that can cause any unexpected behavior in a program. Pure functions are also easier to compose and test.
- **Don't forget about laziness:** Laziness can be used as an alternative to solve problems that have solutions based on recursion. Although laziness does tend to simplify several aspects of functional programming, it also incurs additional memory usage in certain situations, such as holding on to the head of a lazy sequence. Take a look at [http://clojure.org/reference/lazy#\\_don\\_t\\_hang\\_onto\\_your\\_head](http://clojure.org/reference/lazy#_don_t_hang_onto_your_head) for more information on how laziness can increase the memory usage of your program. Most of the standard functions in Clojure return lazy sequences as results, and you must always consider laziness when working with them.
- **Model your program as transformations of data:** It is unavoidable to think in steps as humans, and you must always try to model your code as steps of transforming data. Try to avoid thinking in steps that mutate state, but rather in transformations of data. This leads to a more composable design, which makes combining a handful of transformations very easy.
- **Use the threading macros -> and ->> to avoid nesting expressions:**

You must have seen quite a few examples in this book that have used these macros, and have probably started enjoying their presence in your own code as well. The `->` and `->>` macros improve readability greatly, and must be used wherever possible. Don't hesitate to use these macros even if it avoids a couple of levels of nesting. There are several other threading macros, such as `cond->` and `as->`, that can often be useful.

- **Parallelism is at your fingertips:** There are several ways to write programs that benefit through the use of parallelism in Clojure. You can choose between futures, reducers, `core.async` processes, and several other constructs to model concurrent and parallel operations. Also, most of the state management constructs, such as atoms, agents, and channels, have been designed with concurrency in mind. So, don't hesitate to use them when you're dealing with concurrent tasks and state.
- **Live in the REPL:** It's an indispensable tool for experimenting with code and prototyping your programs. After writing a function or a macro, the first thing you should do is play with it in the REPL. You can use the `load-file` function to quickly reload changes in your source files without ever restarting the REPL. Keep in mind that reloading a source file with the `load-file` form will erase any modifications or redefinitions in the namespace of the source file that have been made through the REPL.
- **Embed a Clojure REPL in your application:** It is possible to embed the REPL into an application, thus allowing us to connect to it and modify its behavior at runtime as we desire. For more information on how to do this, take a look at the constructs in the `clojure.core.server.repl` namespace or the `tools.nrepl` library (<https://github.com/clojure/tools.nrepl>). But, this is a possible security risk, and should be used with caution.
- **Use the standard coding style with consistency:** Maintaining a good coding style is important in any project or programming language. All of the examples in this book are formatted in a standard way, as defined by the Clojure style guide (<https://github.com/bbatsov/clojure-style-guide>)

[style-guide](#)).

# Summary

So far, we talked about several ways to troubleshoot our code. The `tools.trace` and `Spyscope` libraries are useful in interactive debugging, while the `tools.logging` library can be used to log information in running applications. We also discussed a handful of good practices for developing applications and libraries in Clojure.

You must be quite anxious by now to write your own applications in Clojure. If you've been paying attention so far, you must have noticed that Clojure is indeed a simple language. Yet, through its simplicity, we are empowered to create elegant and scalable solutions to a lot of interesting problems. On your journey with Clojure ahead, always strive to make things simpler, if they aren't simple enough already. We'll leave you with a few thought provoking quotes as you go onwards to realize the possibilities of this elegant, powerful, and simple programming language.

*"Composing simple components is the way we write robust software."*

--Rich Hickey

*"Simplicity is prerequisite for reliability."*

--Edsger W. Dijkstra

*"Simplicity is the ultimate sophistication."*

--Leonardo da Vinci

# Appendix A. References

- *Anatomy of a Reducer*, Rich Hickey (2012):  
<http://clojure.com/blog/2012/05/15/anatomy-of-reducer.html>
- *Transducers are Coming*, Rich Hickey (2014):  
<http://blog.cognitect.com/blog/2014/8/6/transducers-are-coming>
- *Introduction to Logic Programming with Clojure*, Ambrose Bonnaire-Sergeant (2011): <http://github.com/frenchy64/Logic-Starter/wiki>
- *N-Queens with core.logic*, Martin Trojer (2012):  
<http://martinsprogrammingblog.blogspot.in/2012/07/n-queens-with-corelogic-take-2.html>
- *Clojure core.async Channels*, Rich Hickey (2013):  
<http://clojure.com/blog/2013/06/28/clojure-core-async-channels>
- *Communicating Sequential Processes*, C. A. R. Hoare (1978):  
<http://www.cs.ucf.edu/courses/cop4020/sum2009/CSP-hoare.pdf>
- *Communicating Sequential Processes*, David Nolen (2013):  
<http://swannodette.github.io/2013/07/12/communicating-sequential-processes/>
- *A Dining Philosophers solver*, Pepijn de Vos (2013):  
<http://pepijndevos.nl/2013/07/11/dining-philosophers-in-coreasync.html>
- *CSP vs. FRP*, Draco Dormiens (2014):  
<http://potetm.github.io/2014/01/07/frp.html>
- *Functional Reactive Animation*, Conal Elliott and Paul Hudak (1997):  
<http://conal.net/papers/icfp97/icfp97.pdf>
- *Yolk examples*, Wilkes Joiner (2013): <https://github.com/Cicayda/yolk-examples>
- *The Clojure docs*: <http://clojure.org/>
- *Cats Documentation*: <http://funcool.github.io/cats/latest>
- *The core.logic wiki*: <http://github.com/clojure/core.logic/wiki>
- *The Pulsar docs*: <http://docs.paralleluniverse.co/pulsar/>
- *The Midje wiki*: <http://github.com/marick/Midje/wiki>
- *Getting Started with Speclj*: <http://speclj.com/tutorial>

- *The core.typed wiki:* <http://github.com/clojure/core.typed/wiki>

# Bibliography

This learning path has been prepared for you to explore Clojure's features and learn to use them in your existing projects. It comprises of the following Packt products:

*Clojure for Java Developers, Eduardo Díaz*

*Clojure High Performance Programming, Second Edition, Shantanu Kumar*

*Mastering Clojure, Akhil Wali*

# Index

## A

- abstract-syntax-tree (AST)
  - about / [Complexity guarantee](#)
- abstract syntax tree / [Modifying code in Groovy](#)
- Abstract Syntax Tree (AST) / [Macros and metadata](#)
- actors
  - using / [Using actors](#)
  - about / [Using actors](#)
  - creating / [Creating actors](#)
  - messages, passing between / [Passing messages between actors](#)
  - errors, handling with / [Handling errors with actors](#)
  - state, managing with / [Managing state with actors](#)
  - comparing with processes / [Comparing processes and actors](#)
- agents
  - about / [Agents](#)
  - validators / [Validators](#)
  - watchers / [Watchers](#)
  - used, for managing state / [Using agents](#)
  - asynchronous state / [Using agents](#)
  - independent state / [Using agents](#)
- Aleph
  - reference link / [Ring \(web\) monitoring](#)
  - URL / [Aleph](#)
- algebraic structures / [Demystifying category theory](#)
- alts!! functions / [Using channels](#)
- alts! function / [Using channels](#)
- Amdahl's Law
  - about / [Amdahl's law](#)
- amortization
  - about / [Laziness in data structure operations](#)
- AppDynamics

- URL / [The Alternate profilers](#)
- application
  - errors, logging in / [Logging errors in your application](#)
- applicative functors
  - using / [Using applicative functors](#)
- array
  - using, for efficiency / [Using array/numeric libraries for efficiency](#)
- array of primitives
  - about / [An array of primitives](#)
- arrays
  - about / [Arrays](#)
- Asphalt
  - URL / [JDBC](#)
- assert function / [Writing tests](#)
- assertions
  - disabling, in production / [Disabling assertions in production](#)
  - reference link / [Disabling assertions in production](#)
- associative / [Using monoids](#)
- associative destructuring
  - about / [Associative destructuring](#)
- Asynchronous Agents
  - about / [Asynchronous agents and state](#)
  - state / [Asynchronous agents and state](#)
  - using / [Why you should use agents](#)
- asynchrony
  - about / [Asynchrony, queueing, and error handling](#)
- atomic updates
  - about / [Atomic updates and state](#)
  - and state / [Atomic updates and state](#)
  - in Java / [Atomic updates in Java](#)
  - support, Clojure / [Clojure's support for atomic updates, Faster writes with atom striping](#)
  - faster writes, with atom striping / [Faster writes with atom striping](#)
- atoms

- about / [Atoms](#)
- synchronous state / [Using atoms](#)
- used, for managing state / [Using atoms](#)
- independent state / [Using atoms](#)
- auto-gensym variable / [Creating macros](#)
- autopromotion
  - about / [Numerics, boxing, and primitives](#)
- Avout
  - URL / [Distributed pipelines](#)

## B

- back pressure, applying
  - about / [Applying back pressure](#)
  - thread pool queues / [Thread pool queues](#)
  - Tomcat / [Servlet containers such as Tomcat and Jetty](#)
  - Jetty / [Servlet containers such as Tomcat and Jetty](#)
  - HTTP Kit / [HTTP Kit](#)
  - Aleph / [Aleph](#)
- backtick
  - URL / [Creating macros](#)
- Bacon.js library
  - reference library / [Using functional reactive programming](#)
- bandwidth
  - about / [Bandwidth](#)
- baseline
  - about / [Baseline and benchmark](#)
- batch processing
  - about / [Batch processing](#)
- behavior-driven development (BDD) / [Testing with specs](#)
- benchmark
  - about / [Baseline and benchmark](#)
- best practices, Clojure / [Thinking in Clojure](#)
- Big-O notation
  - about / [O\(<7\) implies near constant time](#)

- boxed math
  - detecting / [Detecting boxed math](#)
- boxed numerics
  - about / [Numerics, boxing, and primitives](#)
- boxplot
  - about / [Median, first quartile, third quartile](#)
- branch prediction
  - about / [Branch prediction](#)
- bubbles
  - about / [Instruction scheduling](#)
- buffered channel / [Using channels](#)
- bytebuffer
  - URL / [Tackling memory inefficiency](#)

## C

- .class files
  - decompiling, into Java source / [Decompiling the .class files into Java source](#)
- cache
  - about / [Cache](#)
- cache-oblivious algorithms
  - about / [A cache bound task](#)
- cache bound task
  - about / [A cache bound task](#)
- cachegrind
  - URL / [Determining program workload type](#)
- caching
  - about / [Precomputing and caching](#)
- CalfPath
  - about / [Web routing libraries](#)
  - URL / [Web routing libraries](#)
- capacity planning
  - about / [A structured approach to the performance](#)
- category theory

- demystifying / [Demystifying category theory](#)
- cats library
  - URL / [Demystifying category theory](#)
- chan function / [Using channels](#)
- channels
  - about / [Why lightweight threads?, Channels, Using actors](#)
  - using / [Using channels](#)
  - customizing / [Customizing channels](#)
  - connecting / [Connecting channels](#)
  - dining philosophers problem / [Revisiting the dining philosophers problem](#)
- characters
  - about / [Non-numeric scalars and interning](#)
- CharSequence box
  - about / [Collection types](#)
- Cheshire library
  - URL / [JSON serialization](#)
- chunked sequences / [What's wrong with sequences?](#)
- chunking, for reducing memory pressure
  - about / [Chunking to reduce memory pressure](#)
- chunks / [What's wrong with sequences?](#)
- CIDER
  - reference link / [Debugging your code](#)
- Citius
  - about / [Comparative latency measurement](#)
  - URL / [Comparative latency measurement](#)
- classloader
  - about / [The classpath and the classloader](#)
- classpath
  - about / [The classpath and the classloader](#)
- claypoole library
  - URL / [Controlling parallelism with thread pools](#)
- Clj-DBCP
  - URL / [JDBC resource pooling](#)

- clj-mmap
  - URL / [Tackling memory inefficiency](#)
- cljx library
  - URL / [Using reader conditionals](#)
- Clojure
  - about / [Getting to know Clojure](#), [The Clojure model of state and identity](#)
  - code / [Getting started with Clojure code and data](#)
  - data / [Getting started with Clojure code and data](#)
  - lists / [Lists in Clojure](#)
  - operations / [Operations in Clojure](#)
  - functions / [Functions in Clojure](#)
  - data types / [Clojure's data types](#)
  - testing in / [Testing in Clojure](#)
  - destructuring / [Destructuring in Clojure](#)
  - types of collections / [Types of collections in Clojure](#)
  - sequence abstraction / [The sequence abstraction](#)
  - specific collection types / [Specific collection types in Clojure](#)
  - identity / [The Clojure model of state and identity](#)
  - state / [The Clojure model of state and identity](#)
  - reference link / [Using actors](#)
  - best practices / [Thinking in Clojure](#)
- Clojure code
  - equivalent Java source, inspecting for / [Inspecting the equivalent Java source for Clojure code](#)
  - latency bottlenecks / [Latency bottlenecks in Clojure code](#)
  - latency bottlenecks / [Latency bottlenecks in Clojure code](#)
  - aspects, for drill-down measurement / [Measure only when it is hot](#)
  - tuning / [Tuning Clojure code](#)
- Clojure code, tuning
  - tuning / [Tuning Clojure code](#)
  - CPU/cache bound / [CPU/cache bound](#)
  - memory bound / [Memory bound](#)
  - multi-threaded / [Multi-threaded](#)

- Clojure concurrency support
  - Future / [Future](#)
  - promise / [Promise](#)
- Clojure interop syntax
  - about / [Clojure interop syntax](#)
  - object, creating / [Creating an object](#)
  - instance method, calling / [Calling an instance method](#)
  - static method, calling / [Calling a static method or function](#)
  - static function, calling / [Calling a static method or function](#)
  - inner classes, accessing / [Accessing inner classes](#)
  - simple image namespace, writing / [Writing a simple image namespace](#)
- Clojure Parallelization
  - about / [Clojure parallelization and the JVM](#)
- Clojure source
  - compiling, without locals clearing / [Compiling the Clojure source without locals clearing](#)
- Clojure sources
  - compiling, into Java bytecode / [Compiling the Clojure sources into Java bytecode](#)
- Clojure style guide
  - reference link / [Thinking in Clojure](#)
- Clojure support, for parallelization
  - about / [Clojure support for parallelization](#)
  - pmap / [pmap](#)
  - pcalls / [pcalls](#)
  - pvalues / [pvalues](#)
- Clojure web servers
  - URL / [Web servers](#)
- closures
  - about / [Macros and closures](#)
- code
  - exposing, to Java / [Exposing your code to Java](#)
  - evaluating / [Reading and evaluating code](#)

- reading / [Reading and evaluating code](#)
  - quoting / [Quoting and unquoting code](#)
  - unquoting / [Quoting and unquoting code](#)
  - transforming / [Transforming code](#)
  - debugging / [Debugging your code](#)
- code, modifying in Groovy
  - about / [Modifying code in Groovy](#)
  - @ToString annotation / [The @ToString annotation](#)
  - @TupleConstructor annotation / [The @TupleConstructor annotation](#)
  - @Slf4j annotation / [The @Slf4j annotation](#)
- code debugging
  - about / [Debugging your code](#)
  - tracing, using / [Using tracing](#)
  - Spyscope, using / [Using Spyscope](#)
- collections
  - transforming, with reduce functions / [Using reduce to transform collections](#), [What's wrong with sequences?](#)
  - parallelizing, with fold / [Using fold to parallelize collections](#)
- collection types
  - about / [Collection types](#)
- combining function / [Using fold to parallelize collections](#)
- Comma Separated File (CSV) / [Sizing for file/network operations](#)
- Communicating Sequential Processes
  - reference link / [Building reactive user interfaces](#)
- Communicating Sequential Processes (CSPs) / [Using channels](#)
- complexity guarantees
  - about / [Complexity guarantee](#)
- Compojure
  - about / [Web routing libraries](#)
- Component
  - URL / [Distinguish between initialization and runtime](#)
- compositional event systems (CESs) / [Using functional reactive programming](#)

- computational and data-processing tasks
  - about / [Computational and data-processing tasks](#)
  - CPU bound computation / [A CPU bound computation](#)
  - memory bound task / [A memory bound task](#)
  - cache bound task / [A cache bound task](#)
  - input/output (I/O) bound task / [An input/output bound task](#)
- concatenation, of persistent data structures / [The concatenation of persistent data structures](#)
- concurrency
  - about / [Concurrency and parallelism](#)
- concurrency, with threads
  - defining / [Concurrency with threads](#)
  - JVM support, for threads / [JVM support for threads](#)
  - thread pools, in JVM / [Thread pools in the JVM](#)
  - Clojure concurrency support / [Clojure concurrency support](#)
- Concurrent Mark and Sweep (CMS) / [JVM tuning](#)
- concurrent pipelines
  - about / [Concurrent pipelines](#)
  - distributed pipelines / [Distributed pipelines](#)
- concurrent tasks
  - managing / [Managing concurrent tasks](#)
  - delays, using / [Using delays](#)
  - futures, using / [Using futures and promises](#)
  - promises, using / [Using futures and promises](#)
- Configuration
  - URL / [Optimization](#)
- cons function / [The sequence abstraction](#)
- Coordinated Transactional Ref
  - about / [Coordinated transactional ref and state](#)
  - and state / [Coordinated transactional ref and state](#)
  - characteristics / [Ref characteristics](#)
  - history / [Ref history and in-transaction deref operations](#)
  - in-transaction deref operations / [Ref history and in-transaction deref operations](#)

- transaction retries / [Transaction retries and barging](#)
  - barging / [Transaction retries and barging](#)
  - transaction consistency, upping with ensure / [Upping transaction consistency with ensure](#)
  - lesser transaction retries, with commutative operations / [Lesser transaction retries with commutative operations](#)
  - agents, in transactions / [Agents can participate in transactions](#)
  - nested transactions / [Nested transactions](#)
  - performance considerations / [Performance considerations](#)
- copy collection / [HotSpot heap and garbage collection](#)
- core.async
  - about / [core.async](#)
  - lightweight threads / [Why lightweight threads?](#)
  - goblocks / [Goblocks](#)
  - channels / [Channels](#)
  - transducers / [Transducers](#)
- core.async library / [Macros in the real world](#)
  - URL / [Transients](#)
  - reference link / [Using channels](#)
- core.logic library
  - reference link / [Diving into logic programming](#)
- core.match
  - URL / [Working with pattern matching](#)
- core.rrb-vector contrib project
  - about / [The concatenation of persistent data structures](#)
- core.typed library / [Macros in the real world](#)
  - reference link / [Testing with types](#)
  - union types / [Testing with types](#)
  - parameterized types / [Testing with types](#)
  - intersection types / [Testing with types](#)
- cores / [Threads and cores](#)
- Counterclockwise
  - reference link / [Debugging your code](#)
- CPU bound computation

- about / [A CPU bound computation](#)
- CPU level, cycles
  - fetch / [Processors](#)
  - decode / [Processors](#)
  - execute / [Processors](#)
  - writeback / [Processors](#)
- Criterium
  - latency, measuring with / [Measuring latency with Criterium](#)
  - URL / [Measuring latency with Criterium](#)
- Criterium output
  - about / [Understanding Criterium output](#)
- CSP vs. FRP
  - reference link / [Building reactive user interfaces](#)
- Cursive Clojure
  - using / [Using Cursive Clojure](#)
  - plugins / [Using Cursive Clojure](#)
  - installing / [Installing Cursive Clojure](#)

## D

- data
  - processing, with reducers / [Processing data with reducers](#)
- dataflow programming / [Reactive programming with fibers and dataflow variables](#)
- dataflow variables
  - used, for reactive programming / [Reactive programming with fibers and dataflow variables](#)
  - about / [Reactive programming with fibers and dataflow variables](#)
- data serialization
  - about / [Data serialization](#)
- data sizing
  - about / [Data sizing](#)
  - reduced serialization / [Reduced serialization](#)
  - chunking, for reducing memory pressure / [Chunking to reduce memory pressure](#)

- for file/network operations / [Sizing for file/network operations](#)
  - for JDBC query results / [Sizing for JDBC query results](#)
- data structure operations
  - laziness / [Laziness in data structure operations](#)
- data types, Clojure
  - about / [Clojure's data types](#)
  - scalars / [Scalars](#)
  - collection data types / [Collection data types](#)
- defun
  - URL / [Working with pattern matching](#)
- delay
  - using / [Using delays](#)
  - synchronous / [Using delays](#)
- deref function / [Using delays](#)
- design patterns / [Encapsulating patterns in macros](#)
- destructuring
  - about / [Destructuring in Clojure](#), [Destructuring](#)
  - sequential destructuring / [Destructuring in Clojure](#)
  - associative destructuring / [Destructuring in Clojure](#)
- deterministic methodology / [Managing state](#)
- dining philosophers problem / [Using refs](#)
  - revisiting / [Revisiting the dining philosophers problem](#)
  - reference link / [Revisiting the dining philosophers problem](#)
- Dirigiste
  - URL / [Aleph](#)
- distinct function / [The sequence abstraction](#)
- distributed pipelines
  - about / [Distributed pipelines](#)
- doo library
  - reference link / [Defining unit tests](#)
- dropping-buffer function / [Using channels](#)
- dynamic vars
  - binding / [Dynamic var binding and state](#)
  - and state / [Dynamic var binding and state](#)

# E

- EDN
  - URL / [Non-numeric scalars and interning](#)
- empirical rule
  - about / [Variance and standard deviation](#)
- Encoders
  - URL / [Optimization](#)
- endurance tests
  - about / [The load, stress, and endurance tests](#)
- Engulf
  - about / [The load, stress, and endurance tests](#)
  - URL / [The load, stress, and endurance tests](#)
- epochal time model
  - about / [Identity, value, and epochal time model](#)
- equivalent Java source
  - inspecting, for Clojure code / [Inspecting the equivalent Java source for Clojure code](#)
- error handling
  - about / [Asynchrony, queueing, and error handling](#)
- errors
  - handling, with actors / [Handling errors with actors](#)
  - logging, in application / [Logging errors in your application](#)
- Espejito
  - URL / [Latency bottlenecks in Clojure code](#)
  - using / [Latency bottlenecks in Clojure code](#)
- event-based profilers
  - about / [Profiling](#)
- event bus / [Using functional reactive programming](#)

# F

- Facebook API
  - reference link / [Batch support at API level](#)
- faster front-side bus (FSB) / [Memory systems](#)

- fibers / [Using actors](#)
  - used, for reactive programming / [Reactive programming with fibers and dataflow variables](#)
  - about / [Reactive programming with fibers and dataflow variables](#)
- File Transfer Protocol (FTP) / [Sizing for file/network operations](#)
- first function / [The sequence abstraction](#)
- first level citizens / [The functional paradigm](#)
- first quartile
  - about / [Median, first quartile, third quartile](#)
- fold
  - used, for parallelizing collections / [Using fold to parallelize collections](#)
- fold function / [Using fold to parallelize collections](#)
- force function / [Using delays](#)
- Fork/Join framework, Java 7
  - defining / [Java 7's fork/join framework](#)
- fork/join model / [Using fold to parallelize collections](#)
- frequency
  - about / [Median, first quartile, third quartile](#)
- functional programming
  - about / [Basics of functional programming](#)
  - applying, to collections / [Applying functional programming to collections](#)
  - imperative programming model / [The imperative programming model](#)
  - functional paradigm / [The functional paradigm](#)
  - immutability / [Functional programming and immutability](#)
  - laziness / [Laziness](#)
- functional reactive programming (FRP)
  - about / [Using functional reactive programming](#)
  - using / [Using functional reactive programming](#)
- functions, Clojure
  - about / [Functions in Clojure](#)
- functions, using with channels

- chan / [Channels](#)
- >! / [Channels](#)
- <! / [Channels](#)
- >!! / [Channels](#)
- <!! / [Channels](#)
- functors
  - using / [Using functors](#)
- futures
  - using / [Futures](#), [Using futures and promises](#)

## G

- garbage collection / [HotSpot heap and garbage collection](#)
- Garbage Collection (GC)
  - about / [Non-numeric scalars and interning](#)
- garbage collection bottlenecks
  - about / [Garbage collection bottlenecks](#)
  - threads, waiting at GC safepoint / [Threads waiting at GC safepoint](#)
  - jstat, using / [Using jstat to probe GC details](#)
- GCViewer
  - URL / [Garbage collection bottlenecks](#)
- generative testing
  - about / [Generative testing](#)
  - reference link / [Generative testing](#)
- global vars / [Using vars](#)
- gloss
  - URL / [Tackling memory inefficiency](#)
- goblocks
  - about / [Why lightweight threads?](#), [Goblocks](#)
- Graphite
  - about / [Performance monitoring](#)
  - URL / [Monitoring through logs](#)
- Groovy
  - testing from / [Testing from Groovy](#)
- guided performance objectives

- about / [Guided performance objectives](#)

## H

- hardware
  - about / [The hardware](#)
  - processors / [Processors](#)
  - memory systems / [Memory systems](#)
  - storage / [Storage and networking](#)
  - networking / [Storage and networking](#)
- hash maps
  - about / [Sorted maps and hash maps](#)
- Hello threaded world
  - about / [Using your Java knowledge](#)
- higher order functions / [The functional paradigm](#)
- HikariCP
  - URL / [JDBC resource pooling](#)
- HipHip
  - overview / [HipHip](#)
  - URL / [HipHip](#)
- homoiconic language / [Understanding the reader](#)
- homomorphism / [Using applicative functors](#)
- HotSpot heap / [HotSpot heap and garbage collection](#)
- HotSpot JIT compiler
  - about / [The just-in-time compiler](#)
  - optimizations / [The just-in-time compiler](#)
- http-kit
  - reference link / [Ring \(web\) monitoring](#)
- HTTP Kit
  - URL / [HTTP Kit](#)
- hygenic macros / [Creating macros](#)
- HyperThreading / [Threads and cores](#)
- HyperTransport / [Interconnect](#)

## I

- I/O batching
  - about / [I/O batching and throttling](#)
  - batch support, at API level / [Batch support at API level](#)
- I/O profiling
  - about / [I/O profiling](#)
- i7z
  - URL / [OS and CPU/cache-level profiling](#)
- identity
  - about / [The Clojure model of state and identity, Identity, value, and epochal time model](#)
- identity element / [Using monoids](#)
- identity morphism / [Demystifying category theory](#)
- identity value / [Using fold to parallelize collections](#)
- ImageScalr javadoc
  - reference / [Writing a simple image namespace](#)
- immutability
  - about / [Variables and mutation](#)
- Incanter
  - URL / [Variance and standard deviation](#)
- infix notation / [Creating macros](#)
- inlining
  - about / [Inlining](#)
- Input / [Using actors](#)
- input/output (I/O) bound task
  - about / [An input/output bound task](#)
- instruction pipelining
  - about / [Processors](#)
- instruction scheduling / [Instruction scheduling](#)
- instrumenting profilers
  - about / [Profiling](#)
- Integrated Development Environment (IDE) / [Debugging your code](#)
- IntelliJ
  - URL / [Installing Cursive Clojure](#)
- Intel VTune Analyzer

- about / [OS and CPU/cache-level profiling](#)
- interchange / [Using applicative functors](#)
- Internet of Things (IoT)
  - about / [Batch support at API level](#)
- interning
  - about / [Non-numeric scalars and interning](#)  
/ [Using vars](#)
- Inter Quartile Range (IQR)
  - about / [Median, first quartile, third quartile](#)
- introspection, performance monitoring
  - about / [Introspection](#)
  - JVM instrumentation, via JMX / [JVM instrumentation via JMX](#)
- IOPS (Input-output per second)
  - about / [Bandwidth](#)
- iota
  - URL / [Tackling memory inefficiency](#)
- iota library
  - URL / [Processing data with reducers](#)

## J

- Jackson Java library
  - URL / [JSON serialization](#)
- Java
  - about / [Using your Java knowledge](#)
  - restoring to / [Resorting to Java and native code](#)
- java.jdbc
  - reference link / [Sizing for JDBC query results](#)
- java.jmx
  - reference link / [JVM instrumentation via JMX](#)
- java.lang.Thread
  - about / [Using your Java knowledge](#)
- Java bytecode
  - Clojure sources, compiling into / [Compiling the Clojure sources into Java bytecode](#)

- Java concurrent data structures
  - defining / [Java concurrent data structures](#)
  - Concurrent Maps / [Concurrent maps](#)
  - Concurrent Queues / [Concurrent queues](#)
  - Clojure support, for concurrent queues / [Clojure support for concurrent queues](#)
- Java fork/join framework
  - URL / [Using fold to parallelize collections](#)
- Java Management Extensions (JMX)
  - about / [JVM instrumentation via JMX](#)
- Java Memory Model
  - URL / [Java concurrent data structures](#)
- Java Native Interface (JNI) / [Resorting to Java and native code](#)
- Java Runtime Environment (JRE) / [The Java Virtual Machine](#)
- Java source
  - .class files, decompiling into / [Decompiling the .class files into Java source](#)
- Java Virtual Machine
  - about / [The Java Virtual Machine](#)
  - memory organization / [Memory organization](#)
  - HotSpot heap / [HotSpot heap and garbage collection](#)
  - garbage collection / [HotSpot heap and garbage collection](#)
  - memory (heap/stack) usage, measuring / [Measuring memory \(heap/stack\) usage](#)
  - memory inefficiency, tackling / [Tackling memory inefficiency](#)
- jcenter / [Using Maven dependencies](#)
- JConsole
  - about / [Introspection](#)
- JD-GUI
  - about / [Inspecting the equivalent Java source for Clojure code](#)
  - URL / [Inspecting the equivalent Java source for Clojure code](#)
- JDBC
  - about / [JDBC](#)
- JDBC batch operations

- about / [JDBC batch operations](#)
- JDBC resource pooling
  - about / [JDBC resource pooling](#)
- jdeferred
  - reference / [Promises](#)
- Jetty
  - about / [Servlet containers such as Tomcat and Jetty](#)
- joint / [Connecting channels](#)
- joint fitting / [Connecting channels](#)
- JProfiler
  - URL / [The Alternate profilers](#)
- JSON serialization
  - about / [JSON serialization](#)
- jstat
  - URL / [Using jstat to probe GC details](#)
- just-in-time (JIT) compiler / [The just-in-time compiler](#)
- JVM
  - about / [Clojure parallelization and the JVM](#)
  - reference links / [JVM tuning](#)
- JVM instruction
  - URL / [Inspecting generated bytecode for Clojure source](#)
- JVM options
  - XX\*+AggressiveOpts / [JVM tuning](#)
  - -server / [JVM tuning](#)
  - -Xms3g / [JVM tuning](#)
  - -Xmx3g / [JVM tuning](#)
  - -XX\*+UseLargePages / [JVM tuning](#)
- JVM options\*-XX
  - +UseParNewGC / [JVM tuning](#)
- Jvmtop
  - about / [The Java Virtual Machine](#)
  - URL / [The Java Virtual Machine](#)

K

- keyword hierarchies, multimethods
  - isa function / [isa?](#)
  - parent function / [parents](#)
  - descendants function / [descendants](#)
  - underive function / [underive](#)
- keywords
  - about / [Non-numeric scalars and interning](#)

## L

- la carte dispatch functions
  - about / [A la carte dispatch functions](#)
- last-in-first-out (LIFO)
  - about / [Complexity guarantee](#)
- Last-in-First-out (LIFO) / [Memory organization](#)
- latency
  - about / [Latency](#)
  - measuring, with Criterium / [Measuring latency with Criterium](#)
- latency, performance testing
  - measuring / [Measuring latency](#)
  - comparative latency measurement / [Comparative latency measurement](#)
  - latency measurement under concurrency / [Latency measurement under concurrency](#)
- latency numbers
  - about / [The latency numbers that every programmer should know](#)
  - reference link / [The latency numbers that every programmer should know](#)
- laziness
  - about / [Laziness](#)
  - in data structure operations / [Laziness in data structure operations](#)
  - reference link / [Thinking in Clojure](#)
- lazy sequences
  - constructing / [Constructing lazy sequences](#)
  - custom chunking / [Custom chunking](#)

- macros / [Macros and closures](#)
  - closures / [Macros and closures](#)
- / [Thinking in sequences, Lazy sequences](#)
- Leaky Bucket
  - URL / [Throttling requests to services](#)
- Leiningen
  - installing / [Installing Leiningen](#)
  - wiki URL / [Installing Leiningen](#)
  - URL / [Installing Leiningen](#)
  - about / [Criterium and Leiningen](#)
- Leiningen project.clj configuration
  - about / [Leiningen project.clj configuration](#)
  - reflection warning, enabling / [Enable reflection warning](#)
  - optimized JVM options, enabling / [Enable optimized JVM options when benchmarking](#)
- less-used data structures
  - constructing / [Constructing lesser-used data structures](#)
- Let over Lambda
  - reference / [References](#)
- let statement
  - about / [The let statement](#)
- libraries selection
  - about / [Choosing libraries](#)
- libraries selection, via benchmarks
  - about / [Making a choice via benchmarks](#)
  - web server / [Web servers](#)
  - web-routing libraries / [Web routing libraries](#)
  - data serialization / [Data serialization](#)
  - JSON serialization / [JSON serialization](#)
  - JDBC / [JDBC](#)
- lightweight threads
  - about / [Why lightweight threads?](#)
- Likwid
  - URL / [OS and CPU/cache-level profiling](#)

- Lisp
  - about / [Lisp's foundational ideas](#)
  - foundational ideas / [Lisp's foundational ideas](#)
- list
  - about / [Complexity guarantee](#)
- list comprehension
  - creating / [Transforming sequences](#)
- lists
  - about / [Lists](#)
  - properties / [Lists](#)
- lists, Clojure
  - about / [Lists in Clojure](#)
- Little's law
  - about / [Little's law](#)
  - performance tuning / [Performance tuning with respect to Little's law](#)
- local vars / [Using vars](#)
- LogBack
  - URL / [Logging](#)
  - about / [Logging](#)
  - need for / [Why SLF4J/LogBack?](#)
- logging
  - about / [Logging](#)
  - setup / [The setup](#)
- logging configuration
  - reference link / [Logging errors in your application](#)
- logical relations
  - about / [Diving into logic programming](#)
  - solving / [Solving logical relations](#)
  - combining / [Combining logical relations](#)
  - solutions, to problems / [Thinking in logical relations](#)
  - n-queens problem, solving / [Solving the n-queens problem](#)
  - Sudoku puzzle, solving / [Solving a Sudoku puzzle](#)
- logic programming

- exploring / [Diving into logic programming](#)
- logical relations, solving / [Solving logical relations](#)
- logical relations, combining / [Combining logical relations](#)
- Logstash
  - URL / [Monitoring through logs](#)
- Logstash-forwarder
  - URL / [Monitoring through logs](#)
- low-level concurrency
  - about / [Low-level concurrency](#)
  - Hardware Memory Barrier (Fence) instructions / [Hardware memory barrier \(fence\) instructions](#)
  - Java support / [Java support and the Clojure equivalent](#)
  - Clojure equivalent / [Java support and the Clojure equivalent](#)

# M

- macro
  - writing / [Writing your first macro](#)
  - debugging / [Debugging your first macro](#)
  - quoting / [Quote, syntax quote, and unquoting](#)
  - syntax quote / [Quote, syntax quote, and unquoting](#)
  - unquoting / [Quote, syntax quote, and unquoting](#)
  - unquote splicing / [Unquote splicing](#)
  - gensym function / [gensym](#)
  - in real world / [Macros in the real world](#)
- macroexpanded / [Understanding the reader](#)
- macroexpansion / [Understanding the reader](#)
- macros
  - about / [Macros and closures, Macros and metadata](#)
  - expanding / [Expanding macros](#)
  - creating / [Creating macros](#)
  - patterns, encapsulating / [Encapsulating patterns in macros](#)
  - avoiding / [Avoiding macros](#)
  - thumb rules / [Avoiding macros](#)
  - issues / [Avoiding macros](#)

## / [Debugging your code](#)

- macros, as code modification tools
  - about / [Macros as code modification tools](#)
  - code, modifying in Java / [Modifying code in Java](#)
  - code, modifying in Groovy / [Modifying code in Groovy](#)
- map function / [The sequence abstraction](#)
  - characteristics / [What's wrong with sequences?](#)
- maps
  - about / [Maps](#)
- Mastering Clojure Macros
  - reference / [References](#)
- Maven Central
  - URL / [Using Maven dependencies](#)
- maven dependencies
  - using / [Using Maven dependencies](#)
- mean
  - about / [Median, first quartile, third quartile](#)
- median
  - about / [Median, first quartile, third quartile](#)
- Megaref
  - URL / [Performance considerations](#)
- memoization
  - about / [Laziness](#)
- memory (heap/stack) usage
  - measuring / [Measuring memory \(heap/stack\) usage](#)
- memory-mapped buffers
  - reference link / [Resorting to Java and native code](#)
- memory bound task
  - about / [A memory bound task](#)
- memory organization / [Memory organization](#)
- memory systems
  - about / [Memory systems](#)
  - cache / [Cache](#)
  - interconnect / [Interconnect](#)

- memory wall / [Memory systems](#)
- Mersenne prime
  - about / [A CPU bound computation](#)
- metaconstants / [Using top-down testing](#)
- metadata
  - about / [Macros and metadata](#)
- Metaspace / [Memory organization](#)
- Metrics
  - reference link / [Performance monitoring](#)
- metrics-clojure
  - reference link / [Performance monitoring](#)
- metrics-clojure library
  - reference link / [Ring \(web\) monitoring](#)
- Micro-benchmark
  - about / [Measuring latency](#)
- Midje library
  - reference link / [Using top-down testing](#)
- miniKanren
  - about / [Diving into logic programming](#)
  - reference link / [Diving into logic programming](#)
- mix function / [Connecting channels](#)
- mocking / [Writing tests](#)
- monads
  - using / [Using monads](#)
- Monitor tab
  - about / [The Monitor tab](#)
  - Threads tab / [The Sampler tab](#)
  - Profiler tab / [The Profiler tab](#)
  - Visual GC tab / [The Visual GC tab](#)
  - alternate profilers / [The Alternate profilers](#)
- monoid / [Using fold to parallelize collections](#)
- monoids
  - using / [Using monoids](#)
  - about / [Using monoids](#)

- Moore's Law
  - about / [Moore's law](#)
- morphisms / [Demystifying category theory](#)
- Moskito
  - URL / [The Alternate profilers](#)
- multimethods
  - about / [Multimethods in Clojure](#)
  - keyword hierarchies / [Keyword hierarchies](#)
  - la carte dispatch functions / [A la carte dispatch functions](#)
  - versus protocols / [Multimethods versus protocols](#)
- multithreading / [Managing concurrent tasks](#)
- multiversion concurrency control (MVCC)
  - about / [Software transactional memory and refs](#)
- mutable locals, Clojure / [Proteus – mutable locals in Clojure](#)
- mutable state, modeling
  - identity / [Managing state](#)
  - state / [Managing state](#)
  - time / [Managing state](#)
- mutation
  - about / [Variables and mutation](#)

## N

- n-queens problem
  - solving / [Solving the n-queens problem](#)
  - reference link / [Solving the n-queens problem](#)
- namespaces
  - about / [Namespaces in Clojure, Back to Clojure namespaces](#)
  - playing with / [Playing with namespaces](#)
  - creating / [Creating a new namespace](#)
  - working, with on REPL / [Working with namespaces on the REPL](#)
- native code
  - restoring to / [Resorting to Java and native code](#)
- nesting
  - about / [Nesting](#)

- Netty
  - URL / [Aleph](#)
- networking
  - about / [Storage and networking](#)
- New Relic
  - URL / [The Alternate profilers](#)
- nio
  - URL / [Tackling memory inefficiency](#)
- Nippy
  - URL / [Fast repetition, Data serialization](#)
- no.disassemble project
  - URL / [Inspecting generated bytecode for Clojure source](#)
- non-numeric scalars
  - about / [Non-numeric scalars and interning](#)
  - performance characteristics / [Non-numeric scalars and interning](#)
- Non-uniform memory access (NUMA) / [Interconnect](#)
- nREPL
  - URL / [Introspection](#)
- numerical tower
  - reference link / [Numerics, boxing, and primitives](#)
- numeric libraries
  - using, for efficiency / [Using array/numeric libraries for efficiency](#)
- numerics
  - about / [Numerics, boxing, and primitives](#)

## O

- objects / [Demystifying category theory](#)
- observables / [Using Reactive Extensions](#)
- observer / [Using Reactive Extensions](#)
- Om library
  - reference link / [Introducing Om](#)
  - about / [Introducing Om](#)
- online analytical processing (OLAP)
  - about / [Online analytical processing](#)

- online transaction processing (OLTP)
  - about / [Online transaction processing](#)
- Onyx
  - URL / [Distributed pipelines](#)
- Open Source performance
  - tools, URL / [The Alternate profilers](#)
- operations, Clojure
  - about / [Operations in Clojure](#)
- oprofile
  - URL / [Determining program workload type](#)
- optimization
  - about / [Performance optimization](#)
- optional typing / [Testing with types](#)
- Output / [Using actors](#)

## P

- packages
  - about / [Packages in Clojure](#)
  - java.lang / [Packages in Clojure](#)
  - java.io / [Packages in Clojure](#)
  - java.util / [Packages in Clojure](#)
  - java.util.logging / [Packages in Clojure](#)
  - java.text / [Packages in Clojure](#)
  - javax.servlet / [Packages in Clojure](#)
- parallelism
  - about / [Concurrency and parallelism, Executing tasks in parallel](#)
  - controlling, with thread pools / [Controlling parallelism with thread pools](#)
- parallelism, with Reducers
  - about / [Parallelism with reducers](#)
  - Reducible / [Reducible, reducer function, reduction transformation](#)
  - Reducer function / [Reducible, reducer function, reduction transformation](#)
  - Reduction transformation / [Reducible, reducer function, reduction](#)

## [transformation](#)

- reducible collections, realizing / [Realizing reducible collections](#)
- Foldable collections / [Foldable collections and parallelism](#)
- Parallelism / [Foldable collections and parallelism](#)
- parse tree / [Understanding the reader](#)
- pattern matching
  - working with / [Working with pattern matching](#)
- percentile
  - about / [Percentile](#)
- perf
  - URL / [Determining program workload type](#)
- perf-map-agent
  - URL / [Determining program workload type](#)
- Perforate
  - about / [Comparative latency measurement](#)
  - URL / [Comparative latency measurement](#)
- performance
  - structured approach / [A structured approach to the performance](#)
  - about / [Performance and queueing theory](#)
- performance bottlenecks
  - identifying / [Identifying performance bottlenecks](#)
  - latency bottlenecks, in Clojure code / [Latency bottlenecks in Clojure code](#)
  - garbage collection bottlenecks / [Garbage collection bottlenecks](#)
  - generated bytecode, inspecting for Clojure source / [Inspecting generated bytecode for Clojure source](#)
  - throughput bottlenecks / [Throughput bottlenecks](#)
- performance characteristics, transducer
  - about / [Performance characteristics](#)
- performance measurement
  - about / [Performance measurement and statistics](#)
- performance miscellanea
  - about / [Performance miscellanea](#)
  - assertions, disabling in production / [Disabling assertions in](#)

## production

- destructuring / [Destructuring](#)
- tail-call optimization (TCO) / [Recursion and tail-call optimization \(TCO\)](#)
- recursion / [Recursion and tail-call optimization \(TCO\)](#)
- multimethods, versus protocols / [Multimethods versus protocols](#)
- inlining / [Inlining](#)
- performance modeling
  - about / [A structured approach to the performance](#)
- performance monitoring
  - about / [Performance monitoring](#)
  - through logs / [Monitoring through logs](#)
  - ring (web) monitoring / [Ring \(web\) monitoring](#)
  - introspection / [Introspection](#)
- performance optimization
  - about / [Performance optimization](#)
  - project, setting up / [Project setup](#)
- performance statistics
  - about / [Performance measurement and statistics](#)
- performance testing
  - about / [Performance testing](#)
  - environment / [The test environment](#)
  - latency, measuring / [Measuring latency](#)
  - throughput, measuring / [Measuring throughput](#)
  - load testing / [The load, stress, and endurance tests](#)
  - stress testing / [The load, stress, and endurance tests](#)
  - endurance tests / [The load, stress, and endurance tests](#)
- performance tuning
  - about / [Performance tuning](#)
  - Clojure code, tuning / [CPU/cache bound](#)
  - I/O bound / [I/O bound](#)
  - JVM Tuning / [JVM tuning](#)
  - back pressure / [Back pressure](#)
- performance tuning, Little's law / [Performance tuning with respect to](#)

## Little's law

- performance vocabulary
  - about / [The performance vocabulary](#)
  - latency / [Latency](#)
  - throughput / [Throughput](#)
  - bandwidth / [Bandwidth](#)
  - baseline / [Baseline and benchmark](#)
  - benchmark / [Baseline and benchmark](#)
  - profiling / [Profiling](#)
  - optimization / [Performance optimization](#)
  - concurrency / [Concurrency and parallelism](#)
  - parallelism / [Concurrency and parallelism](#)
  - resource utilization / [Resource utilization](#)
  - workload / [Workload](#)
- periods
  - about / [Median, first quartile, third quartile](#)
- permanent generation / [Memory organization](#)
- persistent collections
  - about / [Persistent collections](#)
- persistent data structures
  - about / [Persistent data structures](#)
  - concatenation / [The concatenation of persistent data structures](#)
- persistent hash-maps
  - about / [Complexity guarantee](#)
- persistent hash-vectors
  - about / [Complexity guarantee](#)
- persistent tree-maps
  - about / [Complexity guarantee](#)
- persistent tree-sets
  - about / [Complexity guarantee](#)
- pipe / [Connecting channels](#)
- polymorphism
  - using, in Java / [Polymorphism in Java](#)
- postfix notation / [Creating macros](#)

- precomputing
  - about / [Precomputing and caching](#)
- prefix notation / [Understanding the reader](#)
- premature end of collection
  - about / [Premature end of iteration](#)
- prime (symbol)
  - reference link / [Numerics, boxing, and primitives](#)
- primitive-math
  - about / [primitive-math](#)
  - URL / [primitive-math](#)
- primitive numerics
  - about / [Numerics, boxing, and primitives](#)
- primitives
  - about / [Primitives](#)
- Prismatic Graph
  - URL / [Distinguish between initialization and runtime](#)
- processes
  - about / [Using channels](#)
  - comparing, with actors / [Comparing processes and actors](#)
- processors
  - about / [Processors](#)
  - branch prediction / [Branch prediction](#)
  - instruction scheduling / [Instruction scheduling](#)
  - threads / [Threads and cores](#)
  - cores / [Threads and cores](#)
- production
  - assertions, disabling in / [Disabling assertions in production](#)
- profile
  - about / [Profiling](#)
- profilers
  - event-based profilers / [Profiling](#)
  - instrumenting profilers / [Profiling](#)
  - sampling profilers / [Profiling](#)
- Profiler tab / [The Profiler tab](#)

- profiling
  - about / [Profiling, Profiling](#)
  - OS and CPU/cache-level profiling / [OS and CPU/cache-level profiling](#)
  - I/O profiling / [I/O profiling](#)
- program counter (PC) / [Memory organization](#)
- program workload type
  - determining / [Determining program workload type](#)
- project
  - creating / [Creating a new project](#)
- project setup, performance optimization
  - software versions / [Software versions](#)
  - JVM version / [Software versions](#)
  - Clojure version / [Software versions](#)
  - Leiningen project.clj configuration / [Leiningen project.clj configuration](#)
  - initialization and runtime, differentiating between / [Distinguish between initialization and runtime](#)
- project structure, REPL
  - resources folder / [Project structure](#)
  - src folder / [Project structure](#)
  - dev-resources folder / [Project structure](#)
  - test folder / [Project structure](#)
- Prometheus
  - URL / [The Alternate profilers](#)
- promises
  - about / [Promises](#)
  - pulsar thread / [Pulsar and lightweight threads](#)
  - lightweight thread / [Pulsar and lightweight threads](#)
  - using / [Using futures and promises](#)
- Using actors
  - [Using actors](#)
- Proteus
  - about / [Proteus – mutable locals in Clojure](#)
  - URL / [Proteus – mutable locals in Clojure](#)

- protocols
  - about / [Protocols in Clojure](#)
  - versus multimethods / [Multimethods versus protocols](#)
- proxy
  - about / [Proxy and reify](#)
- proxy-super macro / [Miscellaneous](#)
- publication / [Connecting channels](#)
- Pulsar
  - reference / [Pulsar and lightweight threads](#)
- Pulsar library
  - reference link / [Using actors, Reactive programming with fibers and dataflow variables](#)

## Q

- Quasar
  - reference / [Pulsar and lightweight threads](#)
- queueing
  - about / [Asynchrony, queueing, and error handling, Performance and queueing theory](#)
- queues
  - about / [Using channels](#)
- QuickPath / [Interconnect](#)
- quoting / [Reading and evaluating code](#)

## R

- race condition / [Managing state](#)
- React.js
  - reference link / [Introducing Om](#)
- reactive extensions (Rx)
  - using / [Using Reactive Extensions](#)
- reactive programming
  - fibers, using / [Reactive programming with fibers and dataflow variables](#)
  - dataflow variables, using / [Reactive programming with fibers and](#)

## [dataflow variables](#)

- reactive user interfaces
  - building / [Building reactive user interfaces](#)
- reader conditional form / [Using reader conditionals](#)
- reader conditionals
  - using / [Using reader conditionals](#)
- reader conditional splicing form / [Using reader conditionals](#)
- reader macro
  - about / [Understanding the reader](#)
  - \x / [Understanding the reader](#)
  - ; / [Understanding the reader](#)
  - (.method o) / [Understanding the reader](#)
  - @(... ) / [Understanding the reader](#)
  - @x / [Understanding the reader](#)
  - ^{ ... } / [Understanding the reader](#)
  - 'x or '( ... ) / [Understanding the reader](#)
  - `x or `(... ) / [Understanding the reader](#)
  - ~x or ~( ... ) / [Understanding the reader](#)
  - ~@x or ~@(... ) / [Understanding the reader](#)
  - #'x / [Understanding the reader](#)
  - #=x or #=( ... ) / [Understanding the reader](#)
  - #?( ... ) / [Understanding the reader](#)
  - #?@(... ) / [Understanding the reader](#)
- reader macros / [Understanding the reader](#)
- records
  - about / [Records in Clojure](#)
- recursion
  - about / [Recursion and tail-call optimization \(TCO\)](#)
- recursive functions
  - defining / [Defining recursive functions](#)
- reduce functions
  - used, for transforming collections / [Using reduce to transform collections, What's wrong with sequences?](#)
- reducers

- about / [Introducing reducers](#)
  - used, for processing data / [Processing data with reducers](#)
  - comparing, with transducers / [Comparing transducers and reducers](#)
- reducing function
  - about / [Transducers](#) / [Introducing reducers](#)
- reducing function transformers / [Introducing reducers](#)
- reference types
  - validating / [Validating and watching the reference types](#)
  - watching / [Validating and watching the reference types](#)
  - about / [Managing state](#)
  - characterizing, ways / [Managing state](#)
  - asynchronous / [Managing state](#)
  - synchronous / [Managing state](#)
  - mutation / [Managing state](#)
- reflection
  - about / [Reflection and type hints](#)
- refs
  - used, for managing state / [Using refs](#)
  - coordinated states / [Using refs](#)
  - synchronous state / [Using refs](#)
- reify
  - about / [Proxy and reify](#)
- Relaxed Radix Balanced (RRB)
  - about / [The concatenation of persistent data structures](#)
- Relic
  - URL / [The Alternate profilers](#)
- REPL
  - using / [Using a REPL](#)
  - about / [Using a REPL](#)
  - nREPL protocol / [The nREPL protocol](#)
  - Hello world / [Hello world](#)
  - utilities / [REPL utilities and conventions](#)

- conventions / [REPL utilities and conventions](#), [Creating a new project](#)
  - project structure / [Project structure](#)
  - standalone app, creating / [Creating a standalone app](#)  
/ [Compiling the Clojure source without locals clearing](#)
- requests
  - throttling, to services / [Throttling requests to services](#)
- resource pooling
  - about / [Resource pooling](#)
  - JDBC resource pooling / [JDBC resource pooling](#)
- resource utilization
  - about / [Resource utilization](#)
- rest function / [The sequence abstraction](#)
- Riemann
  - URL / [Performance monitoring](#), [Monitoring through logs](#)
- Ring
  - reference link / [Ring \(web\) monitoring](#)
- Ring-based
  - URL / [Measure only when it is hot](#)
- Ring library
  - about / [Servlet containers such as Tomcat and Jetty](#)
- RRB-trees paper
  - references / [The concatenation of persistent data structures](#)
- RxClojure library
  - reference library / [Using Reactive Extensions](#)

## S

- @Slf4j annotation
  - about / [The @Slf4j annotation](#)
- s-expression / [Understanding the reader](#)
- Sampler tab / [The Sampler tab](#)
  - about / [The Sampler tab](#)
  - thread name, setting / [Setting the thread name](#)
- sampling profilers

- about / [Profiling](#)
- semigroup / [Using monoids](#)
- seqable / [Thinking in sequences](#)
- seq library
  - using / [Using the seq library](#)
- sequence (seq)
  - about / [Thinking in sequences](#)
  - lazy sequences / [Thinking in sequences](#), [Lazy sequences](#)
  - creating / [Creating sequences](#)
  - transforming / [Transforming sequences](#)
  - filtering / [Filtering sequences](#)
  - zippers, using / [Using zippers](#)
- sequence abstraction
  - about / [The sequence abstraction](#)
- sequences
  - about / [Sequences and laziness](#)
  - first function / [Sequences and laziness](#)
  - rest function / [Sequences and laziness](#)
  - next function / [Sequences and laziness](#)
  - limitations / [What's wrong with sequences?](#)
- sequential destructuring
  - about / [Sequential destructuring](#)
- services
  - requests, throttling to / [Throttling requests to services](#)
- sets
  - about / [Sets](#)
  - hash-sets / [Sorted sets and hash sets](#)
  - sorted-sets / [Sorted sets and hash sets](#)
  - common properties / [Common properties](#)
  - union / [Union, difference, and intersection](#)
  - difference / [Union, difference, and intersection](#)
  - intersection / [Union, difference, and intersection](#)
- setup, logging
  - about / [The setup](#)

- dependencies / [Dependencies](#)
  - logback configuration file / [The logback configuration file](#)
  - optimization / [Optimization](#)
- SEXP / [Understanding the reader](#)
- Simultaneous multithreading (SMT) / [Threads and cores](#)
- SLF4J
  - about / [Logging](#)
  - URL / [Logging](#)
  - need for / [Why SLF4J/LogBack?](#)
- sliding-buffer function / [Using channels](#)
- Software Transactional Memory (STM) / [Using refs](#)
- software transactional memory (STM)
  - about / [Software transactional memory and refs](#)
- sorted maps
  - about / [Sorted maps and hash maps](#)
- span-thread / [Pulsar and lightweight threads](#)
- spawn-fiber / [Pulsar and lightweight threads](#)
- specific collection types, Clojure
  - about / [Specific collection types in Clojure](#)
  - vectors / [Vectors](#)
  - vector, using as function / [Vectors](#)
  - nth function / [Vectors](#)
  - get function / [Vectors](#)
  - lists / [Lists](#)
  - maps / [Maps](#)
  - hash maps / [Sorted maps and hash maps](#)
  - sorted maps / [Sorted maps and hash maps](#)
  - common properties / [Common properties](#)
  - sets / [Sets](#)
  - functional programming, applying to collections / [Applying functional programming to collections](#)
- Speclj
  - reference link / [Testing with specs](#)
- Specs

- reference link / [Testing with specs](#)
  - used, for testing / [Testing with specs](#)
- SpyGlass
  - URL / [Precomputing and caching](#)
- Spyscope
  - reference library / [Using Spyscope](#)
  - using / [Using Spyscope](#)
- Staged Event Driven Architecture (SEDA) / [Concurrent pipelines](#)
- stalls
  - about / [Instruction scheduling](#)
- standalone app
  - creating / [Creating a standalone app](#)
- standard deviation
  - about / [Variance and standard deviation](#)
- state
  - about / [The Clojure model of state and identity](#)
  - managing / [Managing state](#)
  - managing, with vars / [Using vars](#)
  - managing, with refs / [Using refs](#)
  - managing, with atoms / [Using atoms](#)
  - managing, with agents / [Using agents](#)
  - managing, with actors / [Managing state with actors](#)
- static instruction scheduling
  - about / [Instruction scheduling](#)
- statistics terminology primer
  - about / [A tiny statistics terminology primer](#)
  - median / [Median, first quartile, third quartile](#)
  - mean / [Median, first quartile, third quartile](#)
  - first quartile / [Median, first quartile, third quartile](#)
  - third quartile / [Median, first quartile, third quartile](#)
  - boxplot / [Median, first quartile, third quartile](#)
  - periods / [Median, first quartile, third quartile](#)
  - frequency / [Median, first quartile, third quartile](#)
  - percentile / [Percentile](#)

- standard deviation / [Variance and standard deviation](#)
  - variance / [Variance and standard deviation](#)
- Statsd
  - about / [Performance monitoring](#)
- StatsD
  - URL / [Monitoring through logs](#)
- storage / [Storage and networking](#)
- Storm
  - URL / [Distributed pipelines](#)
- stress testing
  - about / [The load, stress, and endurance tests](#)
- string concatenation / [String concatenation](#)
- Stringer
  - URL / [String concatenation](#)
- string interning
  - about / [Non-numeric scalars and interning](#)
- strings
  - about / [Non-numeric scalars and interning](#)
- structured approach, to performance / [A structured approach to the performance](#)
- stubbing / [Writing tests](#)
- Sudoku puzzle
  - solving / [Solving a Sudoku puzzle](#)
- symbol capture / [Creating macros](#)
- symbolic expression / [Understanding the reader](#)
- symbols
  - about / [Non-numeric scalars and interning](#)
  - / [Understanding the reader](#)
- Symmetric multiprocessing (SMP) / [Interconnect](#)
- syntax-quote
  - URL / [Creating macros](#)
- syntax tree / [Understanding the reader](#)

T

- @ToString annotation
  - about / [The @ToString annotation](#)
- @TupleConstructor annotation
  - about / [The @TupleConstructor annotation](#)
- tail-call optimization (TCO)
  - about / [Recursion and tail-call optimization \(TCO\)](#)
- take function / [The sequence abstraction](#)
- tasks
  - executing, in parallel / [Executing tasks in parallel](#)
- Tesser
  - URL / [Distributed pipelines](#)
- test-driven development (TDD) / [Defining unit tests](#)
- testing
  - about / [Testing in Clojure](#)
  - from command line / [Testing from the command line](#)
  - in IntelliJ / [Testing in IntelliJ](#)
  - with Specs / [Testing with specs](#)
  - with types / [Testing with types](#)
- tests
  - writing / [Writing the tests](#), [Writing tests](#)
  - unit tests, defining / [Defining unit tests](#)
  - top-down testing, using / [Using top-down testing](#)
- third quartile
  - about / [Median, first quartile, third quartile](#)
- Thread/sleep method / [Using delays](#)
- threadpool / [Using agents](#)
- thread pools
  - used, for controlling parallelism / [Controlling parallelism with thread pools](#)
- threads / [Threads and cores](#)
- Threads tab
  - about / [The Threads tab](#)
  - running state / [The Threads tab](#)
  - Sleeping state / [The Threads tab](#)

- wait state / [The Threads tab](#)
  - park state / [The Threads tab](#)
  - monitor state / [The Threads tab](#)
- throttling
  - about / [I/O batching and throttling](#)
- throughput
  - about / [Throughput](#)
- throughput, performance testing
  - measuring / [Measuring throughput](#)
  - average throughput test / [Average throughput test](#)
- Timbre
  - reference link / [Logging errors in your application](#)
- timeout / [Using channels](#)
- Token Bucket
  - URL / [Throttling requests to services](#)
- Tomcat
  - about / [Servlet containers such as Tomcat and Jetty](#)
- tools.nrepl library
  - reference link / [Thinking in Clojure](#)
- tools.trace contrib library
  - reference library / [Using tracing](#)
- top-down testing
  - about / [Using top-down testing](#)
  - using / [Using top-down testing](#)
- transducer
  - about / [Transducers](#)
  - performance characteristics / [Performance characteristics](#)
- transducers
  - about / [Understanding transducers](#)
  - standard functions, URL / [Understanding transducers](#)
  - results. producing / [Producing results from transducers](#)
  - comparing, with reducers / [Comparing transducers and reducers](#)
  - using / [Transducers in action](#)
  - used, for managing volatile references / [Managing volatile](#)

## [references](#)

- creating / [Creating transducers](#)
- early termination / [Creating transducers](#)
- transformers / [Introducing reducers](#)
- transients
  - about / [Transients](#)
  - fast repetition / [Fast repetition](#)
- truthy value / [Using the seq library](#)
- type checking / [Testing with types](#)
- type hints
  - about / [Reflection and type hints](#)
- types of collections, Clojure
  - about / [Types of collections in Clojure](#)
  - counted collection / [Types of collections in Clojure](#)
  - sequential collection / [Types of collections in Clojure](#)
  - associative collections / [Types of collections in Clojure](#)

## U

- unbound var / [Using vars](#)
- unbuffered / [Using channels](#)
- unit tests
  - defining / [Defining unit tests](#)
- Universal Scalability Law
  - about / [Universal Scalability Law](#)
- unquote splicing
  - about / [Unquote splicing](#)
- use case classification
  - about / [Use case classification](#)
  - user-facing software / [The user-facing software](#)
  - computational and data-processing tasks / [Computational and data-processing tasks](#)
  - online transaction processing (OLTP) / [Online transaction processing](#)
  - online analytical processing (OLAP) / [Online analytical processing](#)

- batch processing / [Batch processing](#)
- user-facing software
  - about / [The user-facing software](#)
- USL
  - URL / [Universal Scalability Law](#)

## V

- value
  - about / [Identity, value, and epochal time model](#)
- values / [Understanding the reader](#)
- varargs / [Creating a new namespace](#)
- variable
  - about / [Variables and mutation](#)
- variable capture / [Creating macros](#)
- variadic functions / [Creating a new namespace](#)
- variance
  - about / [Variance and standard deviation](#)
- vars
  - used, for managing state / [Using vars](#)
- vectors
  - about / [Vectors](#)
  - properties / [Vectors](#)
- Visual GC tab / [The Visual GC tab](#)
- VisualVM
  - used, for profiling code / [Profiling code with VisualVM](#)
  - URL / [Profiling code with VisualVM](#)
- vnStat tool
  - about / [I/O profiling](#)
- volatile references
  - managing / [Managing volatile references](#)

## W

- watch function / [Using atoms](#)
- web-routing libraries

- about / [Web routing libraries](#)
- web server
  - about / [Web servers](#)
- workload
  - about / [Workload](#)
- write-buffer
  - about / [Low-level concurrency](#)
- write absorption
  - about / [Low-level concurrency](#)
- write skew
  - URL / [Upping transaction consistency with ensure](#)

## Y

- yesql library / [Macros in the real world](#)
- Yolk examples
  - reference link / [Using functional reactive programming](#)
- Yolk library
  - reference library / [Using functional reactive programming](#)
- YourKit
  - URL / [The Alternate profilers](#)

## Z

- zippers
  - using / [Using zippers](#)

# Содержание

Clojure: High Performance JVM Programming	18
Credits	20
Preface	21
What this learning path covers	21
What you need for this learning path	23
Who this learning path is for	24
Reader feedback	25
Customer support	26
Downloading the example code	26
Errata	27
Piracy	27
Questions	28
1. Module 1	29
1. Getting Started with Clojure	30
Getting to know Clojure	30
Installing Leiningen	32
Using a REPL	34
The nREPL protocol	35
Hello world	35
REPL utilities and conventions	37
Creating a new project	40
Project structure	40
Creating a standalone app	41
Using Cursive Clojure	43
Installing Cursive Clojure	43
Getting started with Clojure code and data	46
Lists in Clojure	46
Operations in Clojure	47

Functions in Clojure	48
Clojure's data types	50
Scalars	50
Collection data types	51
Summary	53
2. Namespaces, Packages, and Tests	54
Namespaces in Clojure	54
Packages in Clojure	54
The classpath and the classloader	57
Back to Clojure namespaces	58
Playing with namespaces	59
Creating a new namespace	61
Working with namespaces on the REPL	63
Testing in Clojure	68
Testing from the command line	68
Testing in IntelliJ	69
Summary	76
3. Interacting with Java	77
Using Maven dependencies	77
Clojure interop syntax	80
Creating an object	80
Calling an instance method	81
Calling a static method or function	81
Accessing inner classes	82
Writing a simple image namespace	84
Writing the tests	87
The let statement	88
Destructuring in Clojure	91
Sequential destructuring	92
Associative destructuring	92

Exposing your code to Java	95
Testing from Groovy	98
Proxy and reify	100
Summary	103
4. Collections and Functional Programming	104
Basics of functional programming	104
Persistent collections	107
Types of collections in Clojure	109
The sequence abstraction	111
Specific collection types in Clojure	113
Vectors	113
Lists	115
Maps	116
Sorted maps and hash maps	116
Common properties	117
Sets	118
Sorted sets and hash sets	118
Common properties	119
Union, difference, and intersection	119
Applying functional programming to collections	120
The imperative programming model	120
The functional paradigm	122
Functional programming and immutability	124
Laziness	124
Summary	128
5. Multimethods and Protocols	129
Polymorphism in Java	129
Multimethods in Clojure	133
Keyword hierarchies	136
isa?	138

parents	138
descendants	138
underive	139
A la carte dispatch functions	139
Protocols in Clojure	142
Records in Clojure	142
Summary	147
<b>6. Concurrency</b>	148
Using your Java knowledge	148
The Clojure model of state and identity	151
Promises	154
Pulsar and lightweight threads	156
Futures	159
Software transactional memory and refs	161
Atoms	168
Agents	170
Validators	170
Watchers	171
core.async	173
Why lightweight threads?	173
Goblocks	174
Channels	174
Transducers	177
Summary	178
<b>7. Macros in Clojure</b>	179
Lisp's foundational ideas	179
Macros as code modification tools	181
Modifying code in Java	181
Modifying code in Groovy	182
The @ToString annotation	183

The <code>@TupleConstructor</code> annotation	183
The <code>@Slf4j</code> annotation	183
Writing your first macro	188
Debugging your first macro	192
Quote, syntax quote, and unquoting	193
Unquote splicing	194
gensym	196
Macros in the real world	199
References	200
Summary	201
2. Module 2	202
1. Performance by Design	203
Use case classification	203
The user-facing software	203
Computational and data-processing tasks	204
A CPU bound computation	204
A memory bound task	205
A cache bound task	205
An input/output bound task	206
Online transaction processing	206
Online analytical processing	207
Batch processing	207
A structured approach to the performance	209
The performance vocabulary	211
Latency	211
Throughput	212
Bandwidth	212
Baseline and benchmark	213
Profiling	214
Performance optimization	215

Concurrency and parallelism	215
Resource utilization	216
Workload	216
The latency numbers that every programmer should know	218
Summary	220
2. Clojure Abstractions	221
Non-numeric scalars and interning	221
Identity, value, and epochal time model	225
Variables and mutation	225
Collection types	226
Persistent data structures	228
Constructing lesser-used data structures	228
Complexity guarantee	229
$O(<7)$ implies near constant time	231
The concatenation of persistent data structures	231
Sequences and laziness	233
Laziness	233
Laziness in data structure operations	234
Constructing lazy sequences	235
Custom chunking	237
Macros and closures	238
Transducers	240
Performance characteristics	241
Transients	242
Fast repetition	243
Performance miscellanea	245
Disabling assertions in production	245
Destructuring	246
Recursion and tail-call optimization (TCO)	246
Premature end of iteration	247

Multimethods versus protocols	248
Inlining	248
Summary	250
3. Leaning on Java	251
Inspecting the equivalent Java source for Clojure code	251
Creating a new project	252
Compiling the Clojure sources into Java bytecode	252
Decompiling the .class files into Java source	252
Compiling the Clojure source without locals clearing	254
Numerics, boxing, and primitives	255
Arrays	258
Reflection and type hints	261
An array of primitives	262
Primitives	263
Macros and metadata	263
String concatenation	264
Miscellaneous	265
Using array/numeric libraries for efficiency	266
HipHip	266
primitive-math	269
Detecting boxed math	270
Resorting to Java and native code	272
Proteus – mutable locals in Clojure	272
Summary	274
4. Host Performance	275
The hardware	275
Processors	275
Branch prediction	276
Instruction scheduling	277
Threads and cores	277

Memory systems	278
Cache	279
Interconnect	280
Storage and networking	281
The Java Virtual Machine	282
The just-in-time compiler	282
Memory organization	284
HotSpot heap and garbage collection	286
Measuring memory (heap/stack) usage	288
Determining program workload type	290
Tackling memory inefficiency	291
Measuring latency with Criterium	293
Criterium and Leiningen	294
Summary	295
5. Concurrency	296
Low-level concurrency	296
Hardware memory barrier (fence) instructions	296
Java support and the Clojure equivalent	297
Atomic updates and state	301
Atomic updates in Java	301
Clojure's support for atomic updates	302
Faster writes with atom striping	304
Asynchronous agents and state	306
Asynchrony, queueing, and error handling	307
Why you should use agents	309
Nesting	310
Coordinated transactional ref and state	311
Ref characteristics	312
Ref history and in-transaction deref operations	313
Transaction retries and barging	314

Upping transaction consistency with ensure	315
Lesser transaction retries with commutative operations	315
Agents can participate in transactions	316
Nested transactions	317
Performance considerations	317
Dynamic var binding and state	320
Validating and watching the reference types	321
Java concurrent data structures	323
Concurrent maps	323
Concurrent queues	325
Clojure support for concurrent queues	327
Concurrency with threads	328
JVM support for threads	328
Thread pools in the JVM	328
Clojure concurrency support	330
Future	330
Promise	331
Clojure parallelization and the JVM	333
Moore's law	333
Amdahl's law	333
Universal Scalability Law	334
Clojure support for parallelization	334
pmap	335
pcalls	336
pvalues	336
Java 7's fork/join framework	336
Parallelism with reducers	338
Reducible, reducer function, reduction transformation	338
Realizing reducible collections	339
Foldable collections and parallelism	339

Summary	341
6. Measuring Performance	342
Performance measurement and statistics	342
A tiny statistics terminology primer	342
Median, first quartile, third quartile	343
Percentile	344
Variance and standard deviation	345
Understanding Criterium output	346
Guided performance objectives	347
Performance testing	348
The test environment	348
What to test	349
Measuring latency	349
Comparative latency measurement	350
Latency measurement under concurrency	351
Measuring throughput	352
Average throughput test	353
The load, stress, and endurance tests	356
Performance monitoring	357
Monitoring through logs	357
Ring (web) monitoring	358
Introspection	358
JVM instrumentation via JMX	359
Profiling	360
OS and CPU/cache-level profiling	362
I/O profiling	363
Summary	364
7. Performance Optimization	365
Project setup	365
Software versions	365

Leiningen project.clj configuration	366
Enable reflection warning	366
Enable optimized JVM options when benchmarking	367
Distinguish between initialization and runtime	367
Identifying performance bottlenecks	370
Latency bottlenecks in Clojure code	370
Measure only when it is hot	372
Garbage collection bottlenecks	373
Threads waiting at GC safepoint	374
Using jstat to probe GC details	374
Inspecting generated bytecode for Clojure source	375
Throughput bottlenecks	375
Profiling code with VisualVM	377
The Monitor tab	378
The Threads tab	378
The Sampler tab	380
Setting the thread name	381
The Profiler tab	381
The Visual GC tab	382
The Alternate profilers	383
Performance tuning	385
Tuning Clojure code	385
CPU/cache bound	385
Memory bound	386
Multi-threaded	386
I/O bound	387
JVM tuning	387
Back pressure	389
Summary	390
8. Application Performance	391

Choosing libraries	391
Making a choice via benchmarks	392
Web servers	392
Web routing libraries	393
Data serialization	393
JSON serialization	394
JDBC	394
Logging	396
Why SLF4J/LogBack?	396
The setup	396
Dependencies	396
The logback configuration file	397
Optimization	398
Data sizing	400
Reduced serialization	401
Chunking to reduce memory pressure	401
Sizing for file/network operations	402
Sizing for JDBC query results	403
Resource pooling	405
JDBC resource pooling	405
I/O batching and throttling	407
JDBC batch operations	407
Batch support at API level	409
Throttling requests to services	410
Precomputing and caching	411
Concurrent pipelines	412
Distributed pipelines	412
Applying back pressure	414
Thread pool queues	414
Servlet containers such as Tomcat and Jetty	415

HTTP Kit	415
Aleph	416
Performance and queueing theory	417
Little's law	417
Performance tuning with respect to Little's law	418
Summary	419
3. Module 3	420
1. Working with Sequences and Patterns	421
Defining recursive functions	421
Thinking in sequences	428
Using the seq library	430
Creating sequences	434
Transforming sequences	438
Filtering sequences	445
Lazy sequences	447
Using zippers	450
Working with pattern matching	459
Summary	463
2. Orchestrating Concurrency and Parallelism	464
Managing concurrent tasks	464
Using delays	465
Using futures and promises	467
Managing state	473
Using vars	475
Using refs	478
Using atoms	489
Using agents	491
Executing tasks in parallel	499
Controlling parallelism with thread pools	501
Summary	506

3. Parallelization Using Reducers	507
Using reduce to transform collections	508
What's wrong with sequences?	509
Introducing reducers	512
Using fold to parallelize collections	522
Processing data with reducers	529
Summary	534
4. Metaprogramming with Macros	535
Understanding the reader	536
Reading and evaluating code	541
Quoting and unquoting code	546
Transforming code	550
Expanding macros	550
Creating macros	552
Encapsulating patterns in macros	558
Using reader conditionals	562
Avoiding macros	564
Summary	567
5. Composing Transducers	568
Understanding transducers	568
Producing results from transducers	573
Comparing transducers and reducers	577
Transducers in action	579
Managing volatile references	579
Creating transducers	580
Summary	587
6. Exploring Category Theory	588
Demystifying category theory	588
Using monoids	593
Using functors	595

Using applicative functors	599
Using monads	603
Summary	610
7. Programming with Logic	611
Diving into logic programming	611
Solving logical relations	613
Combining logical relations	617
Thinking in logical relations	622
Solving the n-queens problem	622
Solving a Sudoku puzzle	626
Summary	631
8. Leveraging Asynchronous Tasks	632
Using channels	632
Customizing channels	641
Connecting channels	643
Revisiting the dining philosophers problem	649
Using actors	655
Creating actors	656
Passing messages between actors	660
Handling errors with actors	662
Managing state with actors	663
Comparing processes and actors	665
Summary	667
9. Reactive Programming	668
Reactive programming with fibers and dataflow variables	668
Using Reactive Extensions	674
Using functional reactive programming	683
Building reactive user interfaces	689
Introducing Om	698
Summary	703

10. Testing Your Code	704
Writing tests	704
Defining unit tests	705
Using top-down testing	708
Testing with specs	713
Generative testing	717
Testing with types	721
Summary	728
11. Troubleshooting and Best Practices	729
Debugging your code	729
Using tracing	730
Using Spyscope	732
Logging errors in your application	737
Thinking in Clojure	741
Summary	744
A. References	745
Bibliography	747
Index	748