Clojure offers the chance to write high quality, multi-core software faster than ever, without having to leave your current platform.

This book aims to unleash the true potential of the Clojure language so you can apply it in your projects. It begins with the installation and setup of the Clojure environment before moving on to explore the language in-depth. Get acquainted with its various features, such as functional programming, concurrency, and so on with the help of example projects. Additionally, you will also learn how the tool works, and how it interacts with the Java environment.

By the end of this book, you will have a firm grasp of Clojure and its features, and use them effectively to write more robust programs.

Who this book is written for
This book is intended for Java developers who are looking for a way to expand their skills and understand new paradigms of programming. Whether you know a little bit about functional languages, or you are just getting started, this book will get you up and running, using your existing skills in Clojure and functional programming.

What you will learn from this book
- Understand the tools of the Clojure world and how they relate to Java tools and standards (such as Maven)
- Get to grips with immutable data structures, and what makes them feasible for everyday programming
- Write simple multi-core programs using Clojure’s core concepts, such as atoms, agents and refs
- Understand that in Clojure, code is data, and take advantage of that fact by generating and manipulating code with macros
- Learn how Clojure interacts with Java, how class loaders work, and how to use Clojure from Java or the other way around
- Discover a new, more flexible meaning of polymorphism and understand that OOP is not the only way to get it
- Unravel the enigma of Leiningen – the building tool for Clojure

Clojure offers the chance to write high quality, multi-core software faster than ever, without having to leave your current platform.

This book aims to unleash the true potential of the Clojure language so you can apply it in your projects. It begins with the installation and setup of the Clojure environment before moving on to explore the language in-depth. Get acquainted with its various features, such as functional programming, concurrency, and so on with the help of example projects. Additionally, you will also learn how the tool works, and how it interacts with the Java environment.

By the end of this book, you will have a firm grasp of Clojure and its features, and use them effectively to write more robust programs.

Who this book is written for
This book is intended for Java developers who are looking for a way to expand their skills and understand new paradigms of programming. Whether you know a little bit about functional languages, or you are just getting started, this book will get you up and running, using your existing skills in Clojure and functional programming.

What you will learn from this book
- Understand the tools of the Clojure world and how they relate to Java tools and standards (such as Maven)
- Get to grips with immutable data structures, and what makes them feasible for everyday programming
- Write simple multi-core programs using Clojure’s core concepts, such as atoms, agents and refs
- Understand that in Clojure, code is data, and take advantage of that fact by generating and manipulating code with macros
- Learn how Clojure interacts with Java, how class loaders work, and how to use Clojure from Java or the other way around
- Discover a new, more flexible meaning of polymorphism and understand that OOP is not the only way to get it
- Unravel the enigma of Leiningen – the building tool for Clojure

Clojure offers the chance to write high quality, multi-core software faster than ever, without having to leave your current platform.

This book aims to unleash the true potential of the Clojure language so you can apply it in your projects. It begins with the installation and setup of the Clojure environment before moving on to explore the language in-depth. Get acquainted with its various features, such as functional programming, concurrency, and so on with the help of example projects. Additionally, you will also learn how the tool works, and how it interacts with the Java environment.

By the end of this book, you will have a firm grasp of Clojure and its features, and use them effectively to write more robust programs.

Who this book is written for
This book is intended for Java developers who are looking for a way to expand their skills and understand new paradigms of programming. Whether you know a little bit about functional languages, or you are just getting started, this book will get you up and running, using your existing skills in Clojure and functional programming.

What you will learn from this book
- Understand the tools of the Clojure world and how they relate to Java tools and standards (such as Maven)
- Get to grips with immutable data structures, and what makes them feasible for everyday programming
- Write simple multi-core programs using Clojure’s core concepts, such as atoms, agents and refs
- Understand that in Clojure, code is data, and take advantage of that fact by generating and manipulating code with macros
- Learn how Clojure interacts with Java, how class loaders work, and how to use Clojure from Java or the other way around
- Discover a new, more flexible meaning of polymorphism and understand that OOP is not the only way to get it
- Unravel the enigma of Leiningen – the building tool for Clojure
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Getting Started with Clojure'
- A synopsis of the book’s content
- More information on Clojure for Java Developers
**Eduardo Díaz** is a developer with a strong background in the Java language. He has a passion for functional programming and new programming paradigms. His work includes full stack development, systems design, and high volume real time data processing.

He has worked on every technology related problem you can imagine, as a consultant solving anything related to Java, UNIX, C, or any other strange problem you might have had.

As a developer, he has been working for around 10 years on Java, Python, Scala, Clojure, in the media, bank, and primarily communications industries.

He is currently working at Grupo Expansion, a media company, where he helps design and implement a new content delivery platform aiming to empower content editors and encourage developers to find new ways to use data.
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.

Over the course of this book, you will learn about 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 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.

It is important to note that we will build all this knowledge on top of what you already know; this book assumes that you understand the Java language and a bit of how it works. It assumes that you understand how to create classes and objects, how to call methods, and also a bit about the JVM. We will find similarities and differences from the Java language that you already know, so you can understand how the Clojure world works.

It is said that a great programmer is not the one who knows a lot of different programming languages, it is someone who knows different programming paradigms.

Clojure brings ideas from Lisp and functional programming, which are different to what you are used to. This book will help you understand the power behind these ideas and why a language so old still exists and has a place.

Learning Lisp is one of the great pleasures of life, I hope you enjoy it!
What this book covers

Chapter 1, Getting Started with Clojure, is your first step with Clojure, from how to install an interpreter, how to use the IntelliJ plugin Cursive Clojure, and how to start writing your first lines of code in Clojure.

Chapter 2, Namespaces, Packages, and Tests, deals with how every other language needs a way to organize our code. We do it with namespaces, and we start learning by doing little experiments and by comparing with the Java packages we already know.

Chapter 3, Interacting with Java, discusses the fact that one of the most useful features of Clojure is that it can be hosted on top of other platforms. Java is probably one of the most common platforms, and here we learn how to interact with Java code and libraries and how to expose our Clojure code to Java.

Chapter 4, Collections and Functional Programming, tells us that functional programming and immutable data structures (or collections) are fundamental to programming in the Clojure language; here we understand how we can write meaningful programs using immutable data structures and how functional programming is ideal for that.

Chapter 5, Multimethods and Protocols, introduces new tools and ideas of Clojure that help you write much more flexible and simple codes. Destructuring allows you to gain instant access to the data you need from a data structure. Multimethods and protocols are similar to Java’s polymorphism but give you a whole new level of flexibility that you could only dream about.

Chapter 6, Concurrency, tells us about how in the modern world concurrency plays an extremely important part. This chapter also tells us about Clojure’s native primitives for concurrency and again you will learn why immutable data structures are a great ally when writing concurrent programs.

Chapter 7, Macros in Clojure, deals with the fact that these are not the Excel macros you are used to, they are a way in which you can modify your program’s source code at compile time. This is one of Lisp’s big ideas and one that gives you immense flexibility. Here we understand a little of how to use this feature.
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); 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://leiningen.org/. If you want to get a Leiningen installation up and running, this is what you would have to do:

```bash
curl -O https://raw.githubusercontent.com/technomancy/leiningen/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:

```bash
lein repl
```
This is what you’ll get after running the preceding command:

```
# iamedu at Eduardo-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 1.5.0
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:

![Screenshot of Leiningen nREPL client](image)

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

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:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>doc</td>
<td>Prints out a function's docstring</td>
<td>(doc println)</td>
</tr>
<tr>
<td>source</td>
<td>Prints a function's source code, it must be written in Clojure</td>
<td>(source println)</td>
</tr>
<tr>
<td>javadoc</td>
<td>Open the javadoc for a class in the browser</td>
<td>(javadoc java.lang.Integer)</td>
</tr>
</tbody>
</table>

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
```

**Downloading the example code**

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

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:

```clojure
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
ArithmeticException Divide by zero clojure.lang.Numbers.divide
(Numbers.java:156)
user=> *e
#<ArithmeticException java.lang.ArithmeticException: Divide by zero>

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

*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:
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 Leinen 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:

<table>
<thead>
<tr>
<th>IDE</th>
<th>Plugins</th>
</tr>
</thead>
<tbody>
<tr>
<td>IntelliJ</td>
<td>Cursive Clojure, La Clojure</td>
</tr>
<tr>
<td>NetBeans</td>
<td>NetBeans Clojure (works with NetBeans 7.4)</td>
</tr>
<tr>
<td>Eclipse</td>
<td>CounterClockwise</td>
</tr>
<tr>
<td>Emacs</td>
<td>Cider</td>
</tr>
<tr>
<td>VIM</td>
<td>vim-fireplace, vim-leiningen</td>
</tr>
</tbody>
</table>
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.

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:

```clojure
(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.

```clojure
(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:

```clojure
(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:

```clojure
(* 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:

\[
(\text{op parameter-1 parameter-2 ...})
\]

\text{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:

\[
(\text{map inc [1 2 3 4 5 6]})
\]

In this, we are asking to execute the \text{clojure.core/map} function with two parameters:

- \text{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 \text{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:

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

The \text{defn} is the operation that we are asking for. It has several parameters, such as:

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

```clojure
(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:

```clojure
(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.

<table>
<thead>
<tr>
<th>Clojure data type</th>
<th>Java data type</th>
<th>Sample</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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    | Character values, they are java.lang.Character instances, you can define Unicode characters |
|                   |                | \u0045 ;; | |
|                   |                | Unicode char 45 E | |
| Keywords          | Doesn't exist in java | :key
|                   |                | :sample | They evaluate themselves and they are often used as keys. They are also functions that look for themselves in a map. |
|                   |                | :some-keyword | |
| Number            | Numbers are automatically handled as BigDecimal, BigInteger or lower precision depending on what's necessary | 42N ;;Big Integer
<p>|                   |                | 42 ;;long | It is important to remember the trade-offs of Java numbers, if precision is important, you should always use big decimals and big integers. |
|                   |                | 0.1M ;;BigDecimal | |</p>
<table>
<thead>
<tr>
<th>Clojure data type</th>
<th>Java data type</th>
<th>Sample</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ratio</td>
<td>Doesn't exist</td>
<td>$\frac{22}{7}$</td>
<td>Clojure provides great numerical precision; if necessary it can retain the ration and execute exact operation. The tradeoff when using ratios is speed.</td>
</tr>
<tr>
<td>Symbol</td>
<td>Doesn't exist</td>
<td>some-name</td>
<td>Symbols are identifiers in Clojure, very similar to a variable name in Java.</td>
</tr>
<tr>
<td>nil</td>
<td>null</td>
<td>nil</td>
<td>The null value</td>
</tr>
<tr>
<td>Regular expressions</td>
<td>java.util.regex.Pattern</td>
<td>#&quot;\d*&quot;</td>
<td>Regular expressions, in Clojure you get free syntax to define regular expressions, but in the end it is a plain old Java reggae Pattern</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Clojure data type</th>
<th>Java data type</th>
<th>Sample</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td>List</td>
<td>(1 2 3 4 5)</td>
<td>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</td>
</tr>
<tr>
<td>Vector</td>
<td>Array</td>
<td>[1 2 3 4 5]</td>
<td>It is the main workhorse in Clojure, it is similar to an array because you can access elements in a random order</td>
</tr>
<tr>
<td>Set</td>
<td>HashSet</td>
<td>#{1 2 3 4}</td>
<td>A normal Java hash set</td>
</tr>
<tr>
<td>Map</td>
<td>HashMap</td>
<td>{:key 5 :key-2 &quot;red&quot;}</td>
<td>A Clojure map</td>
</tr>
</tbody>
</table>

**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.
Where to buy this book

You can buy Clojure for Java Developers from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.