Ivan Grishaev

Clojure
extended

Java interop

How to use Java features
in Clojure projects

One narrow topic
Five practical chapters

Clojure Extended: Java interop

Ivan Grishaev

This book is for sale at http://leanpub.com/clojure-java-interop

This version was published on 2018-09-03

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Introduction

I often hear that question especially from those who are new to Clojure. Phasing varies in time but the meaning is the same:

> I fell in love with Clojure and would like to program on it on daily basis. But I've got no Java experience in the past. Do I need to know Java? Should I spend some time learning Java first?

I clearly remember same thoughts have been worried me before I dove into the Clojure world. I knew Clojure relies on Java VM heavily but I've got poor Java experience in the past, unfortunately. Just some occasional work on legacy code and an Android app almost completely copy-pasted from public manuals.

Those bits of Java knowledge were not sufficient to feel confident with the platform, I thought. What if I need to call Java code from Clojure? Will my Java-skilled teammates be laughing at me pointing out I just conjure on magic stuff without any knowledge of what's going on under the hood?

Two answers

My eagerness for Clojure finally won, and today I'd like to summarize the following. The short answer to that question is **no**, you don't need to be proficient in Java when starting with Clojure. If you are new to that language, don't be afraid that the whole Java machinery suddenly comes down on you. Perhaps you'll touch Java guts one day, but it will happen in many days or months since the first line of code written.

I hope most of you who have been doubting feel more confident now.

By default, Clojure brings everything you need for a quick start just out from the box. There are great collections, shortcuts to read and write files and special reference types to keep state. Plenty of useful libraries might be connected to your project by adding a single line into a config file. Most of them provide clear API that usually takes a Clojure map and return a map as well.

Clojure has been developing for ten years so far and thus most of the problems you will be trying to solve have already been completed with great libraries. Serving HTTP requests, reading and writing any file formats, fetching data from a local or remote resource, etc might be set up with relative ease by re-using Clojure code developed by programmers all over the world.

Clojure idioms persuade you to develop simple APIs that are easy to use and *understand*. For example, to start a web-server, you declare a handler function that takes and returns a map. You only need to read the document describes the semantics of each field for those maps. The same

approach works for fetching data via HTTP protocol. Again, the input and output data are just Clojure structures consist of maps and vectors.

Let me stress it again: no, you don't need to be a Java expert. If Clojure beckons you, start your journey with no doubts.

The full response to the subject question takes a bit longer since it tries to be more realistic. If you program in Clojure regularly and it has become your primary language, sooner or later you will have to deal with Java. This is inevitable, period. The more business problems you are trying to solve, the more tools and levers you should operate on. The underlying Java world is full of such tools and you would be better to get closer to them.

Java is a platform has been developing for several decades. There are both positive and negative opinions on its nature, so arguing across the Internet never stops. But the facts are, Java is a really mature platform with incredible amounts of time and money invested into it. Rewriting everything that has been solved with Java before would be a crazy idea. Let's briefly go through a list of cases when you might need to refer to Java rather than coding from scratch.

Reusing Java code

For the whole history of Java, millions of lines of code have been written. A significant part of that code is open source. Thousands of people work every day on fixing bugs and adding features. Some of the libraries haven't been updating for several years not because they are abandoned but because they are complete. They solve problems they were aimed at without any claim.

In Clojure, one day you will face such a non-standard task as parsing some legacy binary format or communicate with a third party service. Every time you start writing your own solution for that, check if you are inventing the wheel first. Because most likely there is a Java library for that, being developed and debugged for several years. You might think your own solution is better and clearer, but no it is not. Qualitative software really takes years to become at least sufficient not even great. Such a simple task as parsing an RSS feed or User-Agent header conceals dozens of hidden tricks. For your own sanity, you'd better not to open Pandora box by writing everything from scratch.

Many well-known Clojure libraries are just thin wrappers around solid Java code. Inside, they might look ugly and full of mutable state and weird tricks, etc. But they work, which is most important. Reusing Java code really boosts the Clojure development process, consumes time and nerves.

IO capabilities

Another reason for writing low-level Java code is to take control over IO operations. Java provides mighty capabilities for sending and accepting data to and from no matter if we deal with files, network or processes. Instead, Clojure was designed in terms of being just a guest system. It reuses the host environment (Java in our case) to provide high-level abstractions, e.g flexible collections,

smart multimethod dispatching, etc. But it doesn't try to overshadow such the original low-level features as input and output or interaction with the operating system.

Clojure ships some wrappers around Java's native IO subsystem and usually they are enough to write or read a file. But one day you may face a tough situation when plenty of streams are involved all together so you need to orchestrate them. This is the case you definitely need Java interop.

Source code problems

It might happen that the project you are working on relies on a special feature that's difficult to implement. Say, analysing fingerprints, face or voice recognition. Your customer may own a licence for a commercial library written in Java. Probably they've been using this library in all their Java projects so the Clojure solution should also follow it to make results relevant. So you've got a `*.jar` file without sources and an HTML file next to it describes classes and methods available from the library.

Surely you can decompile it and spend some time looking at its guts, but probably it won't change anything. Sometimes, you are really clamped by business demands so writing your own pure Clojure solution is out of a discussion. In that case, you have to go down to the Java basement.

Your career growth

The final and the most general reason for inventing time in Java is your career growth. If you program with Clojure on a daily basis, the businesses will offer more and more opportunities for you. To deal with them, you need to broaden your horizons. Diving into low-level Java world is exactly such a thing that raises your skills and thus boosts the career growth. If you in love with Clojure and feel confident with your intentions, you cannot bypass Java interop.

That was the long answer to the question if you need to know Java. So far, I hope you feel more confident than before and nothing prevents you from getting started with Clojure.

What is that book about

This paper is focused on one narrow topic: how to use Java capabilities in Clojure. There are plenty of tutorials and books about Clojure so far, but most of them consider their readers as new ones in programming. They consist basically of defining functions and transforming collections. There is no room for extended topics as the result.

Java interop is one of such topics. When you learn Clojure, usually you are focused primarily in the language by its own: functions and namespaces, atoms, etc. Everything looks simple and elegant so far. But once you've got a real Clojure job, it turns out the codebase is full of tricky calls to Java objects. Here, *the real work* starts.

You feel confused because the code doesn't look like the tutorials you read. So the aim of that book is to get you prepared for such a situation. Together, we will go through a set of chapters each of them expounds something useful on Java interop.

The structure

The **initial chapter** explains the basics on how to use Java classes in a Clojure project. There, I provide examples of importing such basic Java files as `java.io.File` or `java.util.Date` and performing simple operations on them. Yet it is not another Java tutorial. Everything we talk about is looked through a Clojure prism. The chapter also shares advice and common practices about organizing namespaces. At the end of the chapter, I expect a reader is familiar with Java interop even he or she has never worked with it (or at least they are not afraid anymore).

Then, a set of code-driven sessions follows. I would like to take apart several business cases. In each of them, you are going to solve a problem by writing Clojure code that relies on Java capabilities. What I'd like to highlight is all the cases are real and borrowed from the projects I've been worked on personally. None of them was invented just to make my statements louder, no. Everything described in that book is for real.

The **first coding session** shows how to extract as much information as possible from a User-Agent HTTP header comes to the server. This is definitely such a task that is better never be done with bare hands. I'll show how a Java library might help by calling proper classes and turning the result into plain Clojure data.

The **second task** affects an interesting topic: how to clean HTML markup out from unwanted and malicious data. It doesn't occur often in our daily job, but once it does, the days and weeks might be spent to overcome the problem. Again, we will rely on proven Java solutions. I'll show how to deal with tricky business rules to prevent certain media resources.

In the **third chapter**, let's talk about how to extend the JDBC driver to let it work with non-primitive data types. I consider this chapter is quite important since a database is the most valuable part of a project. The more features a database can perform, the less code you'll need for the business logic. I'll demonstrate how to build a seamless connection between native Clojure data and PostgreSQL backend.

The **fourth code sample** highlights Java IO capabilities, namely different types of input streams and how to conjoin them together. We will build a pipeline that pulls out a huge amount of data from a zipped CSV file without touching a hard drive or consuming the whole available memory.

The **fifth lesson** explains how to take control over Java processes. Although most of the business tasks might be solved with Clojure or Java code, sometimes you need to communicate with a third-party programme installed on your computer. Clojure brings poor capabilities for running external programs so we'll take a romp with the subject. For a bonus, I'll show how to control your browser with Clojure functions.

There has been said enough to raise your impatience. I hope you cannot wait to go further. So am I.

Java interoperability basics

This chapter explains basic Clojure capabilities to interoperate with Java. We will use them in further practical chapters a lot. I'm going to highlight how to operate on Java objects as well as share some good practices on the subject.

Importing classes

To use a class, you've got to import it first. There are two ways to import a class: in a namespace declaration which is desirable and in runtime using `import` function. For example:

```
(ns com.project.module
  (:import java.util.File))
```

or

```
(import 'java.util.File)
```

In the first example, we specify a class without putting a leading quote. This is because the whole `ns` statement is a macro so everything inside it is quoted. Instead, when calling the `import` function in runtime we pass a symbol which's name stands for a class we need.

Importing classes on the fly might be confusing and not obvious. Thus, it's better to keep all the imports and requirements at the top of a file in `ns` declaration.

To be imported, a class should present in your classpath. A classpath is a special parameter of Java virtual machine. It is a list of locations separated with colons where the machine searches classes for. Usually, you don't worry about configuring classpath manually because modern development tools like `lein` or `deps.edn` take care of that by their own.

The tools mentioned above scan through a list of dependencies declared in a config file. They download required artefacts into a special folder on your machine and start JVM passing with a classpath extended with everything you specified.

Classes that belong to the `java.lang` package are not required to being imported. They are available by default, so `java.lang.String` might be reduced to just `String`.

How to initiate an object

Creating an instance of a class works in two ways: `new` or trailing dot macros. The `new` form takes a class followed by its arguments. It looks like the standard `new` operator in Java. The trailing dot macro requires putting a dot at the end of a class name skipping `new` at the begging. As the result, it's a bit shorter than the first `new` form.

Let's demonstrate everything told so far with examples. We will use the standard classes distributed with Java SDK.

Importing a single class:

```
(ns project.into
  (:import java.io.File))
```

Importing several classes from the same package at once:

```
(ns project.into
  (:import java.io.File
           (java.util Date UUID)))
```

Pay attention at extra parens around the `java.util` path. It's mandatory to put them here. They help Clojure reader to not get confused when parsing imports.

Let's initiate some of the classes we've imported:

```
(def file (File. "/Users/ivan/.emacs"))

;; evaluating the `file` variable in repl prints
#object[java.io.File 0xbfef89d "/Users/ivan/.emacs"]

(def date (new Date))

;; evaluating `date` returns
#inst "2018-08-25T08:20:40.412-00:00"
```

Static methods

Some Java classes provide static methods. Calling such a method doesn't require to create an instance of a class. Access them via slash as follows:

```
(def uuid (UUID/randomUUID))

#uuid "fb5876a6-b3c6-47dc-89d6-10dafcaf0888"
```

The standard System class[1] carries plenty of static methods useful for general purposes. This class belongs to `java.lang` package and thus is available from everywhere without importing it.

To stop the program completely call its `exit` method passing an exit code. Running it in a REPL session will terminate it:

```
(System/exit 0)
```

The `getenv` static methods either return either a single environ variable or the whole map of them depending on arity (the number of passed parameters):

```
(System/getenv "HOME")
"/Users/ivan"

(into {} (System/getenv))
{"LEIN_VERSION" "2.6.1"
 "HOME"         "/Users/ivan"
 "USER"         "ivan"
 "LEIN_HOME"    "/Users/ivan/.lein"
 ;; truncated
 }
```

In the second case, we convent a Java native map into its Clojure counterpart to make the output look better.

A simple wrapper to get the current number of seconds since 1 Jan 1970 which is also known as Unix timestamp or epoch:

```
(defn epoch
  []
  (quot (System/currentTimeMillis) 1000))

(epoch)
1535186375
```

[1] https://docs.oracle.com/javase/9/docs/api/java/lang/System.html

Object methods

Having an object, usually, you are interested in calling its methods. To access a non-static method, put its name with leading dot at the first place of a lisp form followed by an object and the rest arguments.

Here is how you may know a file's absolute path:

```
(def file (File. "book.txt"))

(.getAbsolutePath file)
"/Users/ivan/drafts/project/book.txt"
```

To check if it really exists:

```
(.exists file)
true
```

Or to rename (move) it:

```
(.renameTo file (File. "/Users/ivan/ready/project/book-ready.txt"))
true
```

Nested classes

Sometimes, a class declares a nested class. To access it, put a dollar sign between their names. For example, if a class `Foo` carries a nested class `Bar`, a syntax to reach it will be:

```
(ns project.into
  (:import com.project.Foo$Bar))

(def bar (Foo$Bar. param1 param2))
```

Universal Java access

There is a special Dot form that acts like a universal access to Java properties. It keeps the same syntax even for different cases.

To read a static field:

```
(. File pathSeparator)
":"
```

To call a static method:

```
(. File createTempFile "temp" ".txt")
#object[java.io.File 0x4c2544bc "/var/folders/94/...6500566792.txt"]
```

To call a method of an instance:

```
(. file getAbsolutePath)
"/Users/ivan/drafts/project/book.txt"
```

Or to read a field of an instance. Pay attention at the leading hyphen:

```
(. obj -value)
;; the same as `obj.value` in pure Java
```

There is also a `set!` form that works in pair with the Dot macro. Use it to write a new value to a field:

```
(set! (. obj -value) 42)
;; the same as `obj.value = 42` in pure Java
```

The last two cases with reading and writing a field are not common due to Java design. Exposing fields to the outer world is considered as bad practice. Instead, most Java programmers provide special methods to regulate how a certain field is being read or set. Thus, you call for `(.getValue obj)` or `(.setValue obj 42)` more often than accessing raw fields.

The Double Dot macros acts similar to its Single Dot brother. It chains results between multiple expressions so the next form takes a value produced by a previous form. It is similar to the threading `->` macro that probably you are familiar with.

```
(.. file toPath getFileSystem getClass getName)
"sun.nio.fs.MacOSXFileSystem"
```

Under the hood, it turns into a nested extression like:

```
(. (. (. (. (. file toPath) toPath) getFileSystem) getClass) getName)
```

which is difficult to read since your eyes constantly jump to and fro.

Each method in a chain is called from an object received from a previous unit. If any extra arguments are required, the method expression is put into parens:

```
(.. obj (some-method "foo") (other-method "bar" 42))
```

Reflection

To know what class an object belongs to, call `class` function:

```
(class file)
java.io.File
```

The `type` function acts similar but it checks for a variable's metadata first. Let's discuss how it works. If you define a variable with a type hint as shown below:

```
(def ^File file (File. "some/path.txt"))
```

the metadata of `#'file` variable will contain a `:tag` field with `java.io.File` value:

```
(meta #'file)

{:tag java.io.File,
 :line 1778,
 :column 15,
 :file "*cider-repl*",
 :name file,
 :ns #namespace[project.into]}
```

Probably you would like to divide logic into different branches depending on the variable's type. For example, a `file` parameter in a function might be of both `String` and `File` types. The function `instance?` checks if an object is an instance of a certain class. So instead of comparing types:

```
(case (class source)

  java.io.File
  ;; do this

  String
  ;; do that
)
```

you declare predicates:

```
(def file? (partial instance? java.io.File))

(cond
  (file? source)
  ;; do this

  (string? source)
  ;; do that
  )
```

which is more readable and neat.

Rest Java arguments

Some Java methods accept an arbitrary number of arguments. They are marked with ellipsis in signatures and represent an array of objects when accessing them. A good example is a `format` method of a `String` class:

```
static String    format(String format, Object... args)
```

In Clojure, calling such a method in this way won't work:

```
(String/format "%s %s %s" "foo" "bar" "baz")
```

An exception will raise saying `No matching method: format`.

This is because, in Clojure terms, the `args` parameters should be passed as a single array. It must be exactly a native Java array but not a Clojure sequence. To make your life a bit easier, there is already a `make-array` function that turns a Clojure collection into a Java typed array:

```
(String/format "%s %s %s" (into-array ["foo" "bar" "baz"]))
"foo bar baz"
```

By default, the function builds an array of `Objects` that satisfies the method's signature in our case. When you need an array of some certain type, you pass a class as the first parameter to that function:

```
(String/format "%s %s %s" (into-array String ["foo" "bar" "baz"]))
```

Type hints

Yet Clojure is a dynamic language its runtime relies on types a lot. When it knows what type an object belongs to, it doesn't spend extra time on reflection and thus performs faster. You can help the compiler to guess types by adding hints. A type hint takes its place in front of a variable in function signatures or `let` bindings.

Here is an example of a function that stops a process which is an instance of `java.lang.Process` class:

```
(defn stop-process
  [^Process p]
  (when (.isAlive p)
    (.destroy p)))
```

If you remove the `^Process` clause in the signature, the function will still work. But without a hint, Clojure spends extra time to figure out what the kind of an object is there. That's normal but reduces performance. To highlight all such problem places, set a special global variable to true:

```
(set! *warn-on-reflection* true)
```

If you compile any code that suffers from missing hints, you will see warnings in the REPL:

```
Reflection warning, /src/project/proc.clj:136:5 - reference to field destroy can't b\
e resolved.
```

Type hints should definitely be put in bottleneck functions called often. It is a good practice to scan the codebase for missing hints from time to time. Yet there is no need to put tags everywhere you physically can. The compiler is smart enough to guess a further type if it has enough data.

A type hint may be any class imported before. Sometimes, some complex Java signatures are required to specify an array of a certain type for example. So a hint takes a form of a string:

```
(defn ^"[Ljava.lang.String;"
  args->command
  [args]
  (into-array String (map str args)))
```

Best practices

One more benefit tags bring to a project is they are useful to get into the codebase quickly. When you operate on ordinary Java primitives and collections, a type of a variable might be guessed by its name with ease. For example, `opt` and `params` are usually maps, `items` and `users` are sequences, `url` and `path` are strings.

That's fine until you pass an object belongs to a third-party library, e.g. a parsed HTML document which is an instance of `org.jsoup.nodes.Document` class. A lonely name `doc` won't say anything to a programmer who is maintaining the code. Is it a map, a vector or something else? Instead, the `org.jsoup.nodes.Document doc` declaration clearly expresses the very nature of the parameter so it saves time.

Try to isolate calls to Java classes in a separate namespace or even a library. Expose only Clojure high-level functions that conceal the inner Java-based logic. Let those functions accept and return native Clojure structures like maps and vectors. Java interop is considered as a low-level feature so it is always a subject to alter. If anything changes in a namespace's guts, none if other modules will fail.

Clojure code that relies on Java a lot grows in size because of `longJavaNamingRules`. As the result, it looks clumsy and might be difficult to read. Keeping such a code into a separate namespace reduces noise in the business logic.

Conclusion

So far, we have passed through the most part of Clojure capabilities for Java interop. We didn't touch some topics like proxying and reifying interfaces or something else. But this book is not aimed at just enumerating features. Instead, everything in this book is about practice. I cannot guarantee we will cover all the possible Java features, but everything you will learn will be done for sure. I persuade you to start coding now. If any unmentioned feature appears, we will deal with it on the fly.