Enterprise Clojure Training

Timothy Pratley

Table of Contents

About . . . 1
- Overview . . . 1
- Duration . . . 1
- Learning Objectives . . . 1
- Prerequisites . . . 1
- Target Audience . . . 1
- Required setup . . . 1
- Pre-assessment . . . 2
- Required preparation . . . 2
- Optional reading . . . 2
- Introductions . . . 2
  - The instructor . . . 2
  - Clojure . . . 2
  - Syntax Summary . . . 3

1. The Clojure Ecosystem . . . 4
- 1.1. Leiningen . . . 4
- 1.2. The Read Eval Print Loop (REPL) . . . 4
- 1.3. Editor setup . . . 5
- 1.4. Exercises . . . 5
- 1.5. Answers . . . 6

2. Clojure Syntax . . . 7
- 2.1. Primitive data types . . . 7
- 2.2. Collections: lists, vectors, maps, and sets . . . 8
- 2.3. Invoking functions . . . 10
- 2.4. Defining vars . . . 10
- 2.5. Binding names with let . . . 11
- 2.6. Destructuring (also known as binding forms) . . . 12
- 2.7. Namespaces . . . 14
- 2.8. Regex . . . 15
- 2.9. Exercises . . . 15
- 2.10. Answers . . . 15

3. Functions . . . 17
- 3.1. Defining functions . . . 17
- 3.2. Pre- and post-conditions . . . 18
- 3.3. Anonymous functions . . . 19
- 3.4. Function literals . . . 21
- 3.5. Keyword and variadic arguments . . . 21
- 3.6. Exercises . . . 22

3.7. Answers . . . 22
4. Challenge 1: Corgi Cover eligibility . . . 24
4.1. Part 1: . . . 24
4.1.1. Test data: . . . 24
4.2. Part 2: . . . 24
4.3. Part 3: . . . 24
4.4. Part 4: . . . 25
5. Testing with clojure.test . . . 26
5.1. Defining tests with deftest . . . 26
5.2. lein-test-refresh . . . 27
5.3. Assertions with is and are . . . 27
5.4. Test fixtures . . . 29
5.5. Using with-redefs for mocking behavior . . . 29
5.6. Debugging . . . 30
5.7. Exercises . . . 31
5.8. Answers . . . 31
6. Control Flow . . . 32
6.1. Conditionals: if, when, cond . . . 32
6.2. Recursion . . . 33
6.3. Loops . . . 33
6.4. Exception handling . . . 33
6.5. Comments . . . 33
6.6. Exercises . . . 34
6.7. Answers . . . 34
7. Functional Programming . . . 36
7.1. Pure functions and side effects . . . 36
7.2. Apply, partial, and comp . . . 37
7.3. Functions on sequences: map, reduce, and friends . . . 38
7.4. Threading operators . . . 43
7.5. Data structures are functions! . . . 44
7.6. Exercises . . . 45
7.7. Answers . . . 45
8. Challenge 2: Processing files . . . 47
8.1. Part 1: . . . 47
8.2. Part 2: . . . 47
8.3. Part 3: . . . 47
8.4. Part 4: . . . 47
9. Macros . . . 48
9.1. Expanding macros . . . 48
9.2. Defining macros . . . 48
9.3. Syntax quoting . . . 49

9.4. Code as data 51
9.5. Exercises 51
9.6. Answers 52
10. Parallel Programming and Concurrency 53
10.1. Vars and dynamic scope 53
10.2. Delays, Futures, and Promises 54
10.2.1. Delays 54
10.2.2. Futures 55
10.2.3. Promises 56
10.3. Atoms, Refs, and Agents 56
11. Java Interop 58
11.1. Clojure syntax for Java constructors 58
11.2. Calling methods 58
11.3. reify 59
11.4. gen-class and proxy 59
11.5. Including Java classes in Clojure projects 59
12. Challenge 3: Mocking parallel web requests 60
12.1. Part 1: Mock a web request 60
12.2. Part 2: Report the how long it takes 60
12.3. Part 3: Make parallel requests 60
12.4. Part 4: Error handling 60
13. Polymorphism and Types 61
13.1. Multimethods 61
13.2. Protocols 62
13.3. Creating types with defrecord and deftype 64
13.3.1. Deftype 64
13.3.2. Defrecord 64
13.4. Specifications with clojure.spec 65
13.5. Validation 65
13.6. Conforming 66
13.7. Maps 67
13.8. A game of cards 68
13.9. Generators 69
13.10. Instrumentation and Testing 71
14. Interacting with a Database 75
14.1. Intro to clojure.java.jdbc 75
14.2. Inserting, updating and retrieving data 75
14.3. Solutions for SQL management 77
14.4. Exercises 77
14.5. Answers 77
15. Challenge 4: Corgi Cover Database 79

15.1. Part 1: Set up the schema . . . 79
15.2. Part 2: Populate the data . . . 79
15.3. Part 3: Write a spec . . . 79
15.4. Part 4: Extending to Poodle Protection . . . 79
16. Further reading . . . 80

About

Welcome to Enterprise Clojure Training. This course is for developers learning Clojure for the purpose of building enterprise software.

Overview

Clojure is a dynamic, general-purpose programming language, combining interactive development with an efficient and robust infrastructure for multithreaded programming. Clojure provides direct access to the Java frameworks. Clojure is a dialect of Lisp, and shares with Lisp the code-as-data philosophy and macro system. Clojure is predominantly a functional programming language, and features a rich set of immutable, persistent data structures.

In this course we will cover how to: install Clojure and related tools; interact with Clojure via the read-eval-print- loop (REPL); create functions, data-structures, macros, and types; use functional programming constructs like map and reduce; and design, implement, and test Clojure programs.

Each section will feature interactive exercises, and course material will be reinforced with guided case studies.

Duration

2days, 10 hr/day + 2 hour webinar after completion of workshop.

Learning Objectives

At the end of this course, you will be able to

- Write Clojure code
- Structure Clojure products
- Interact with Java
- Use Clojure’s parallel programming and concurrency facilities

Prerequisites

General programming knowledge.

Target Audience

Developers and Senior Developers.

Required setup

The following software must be installed on your laptop prior to the course:

- Java (https://java.com)
- Leiningen (https://leiningen.org)
- IntelliJ (https://www.jetbrains.com/idea)
- Cursive plugin for IntelliJ (https://cursive-ide.com)

Pre-assessment

- What programming languages have you used before?
- Do you have an interest in Clojure? If so what in particular interests you?
- What do you plan to do with Clojure?
- Name a scenario where you would use a HashMap data structure.
- When should you use a Vector instead of a List or an Array?

Required preparation

Complete the first 10 exercises on the 4Clojure website (http://www.4clojure.com).

Optional reading

If you would like to get a head-start, please read the official Clojure introduction tutorial (http://clojure-doc.org/articles/tutorials/introduction.html). This material will be covered as part of the course. Having read it before hand will allow you to focus on working through the exercises of the course.

Introductions

> Creativity is the ability to introduce order into the randomness of nature.
>
> — Eric Hoffer

The instructor

Timothy Pratley is the author of the book “Professional Clojure”, and a contributor to the Clojure core language. He has 18 years of professional software development experience in banking, robotics, logistics, and advertising. He spent the last 4 years exclusively using Clojure and ClojureScript developing enterprise systems for Fortune 500 companies. He enjoys making YouTube videos about Clojure, running, and reading books.

Clojure

During this course we will be examining the Clojure language up close. Sometimes a new language can feel different just for difference sake. So why is it worth learning Clojure?

Clojure is simple and data-oriented. Smart people want to use it. Clojure enables teams to build things fast. This makes it excellent for delivering value in enterprise development projects.

Throughout the course there will be time to reflect on what purpose the differences serve and what trade offs are being made. These are the Clojure language themes to watch out for as we move through the course:

Data

- Literals
- Sequences
- Transformations

Functions

- Act on general purpose data structures
- Pure

A tool for thought

- Concise
- Unadorned
- Abstract

Getting stuff done

- Access to libraries
- Performance

Syntax Summary

Table 1. Everything is a list with the operation at the front.

Java	Clojure
`int i = 5;`	`(def i 5)`

`if (x == 0)`
`  return y;`
`else`
`  return z;`
	`(if (zero? x)`
	`  y`
	`  z)`

`x * y * z;`	`(* x y z)`
`foo(x, y, z);`	`(foo x y z)`
`foo.bar(x);`	`(.bar foo x)`

Things that would be declarations, control structures, function calls, operators, are all just lists with an op at front.

Chapter 1. The Clojure Ecosystem

There are many Clojure libraries. Hosted on Maven and Clojars. Just jars, like any other Java artifact.

Clojure is itself a Java library. Clojure can make direct use of Java libraries. ClojureScript can make direct use of JavaScript libraries.

The Clojure compiler is a Java library, a clojure.jar file. The only required installation is that Java must be installed. Clojure is very simple to deploy due to the lack of dependencies.

You can use Java tooling to manage your project, but Clojure has some tools to make the process easier.

Please follow along on your laptop and ask questions at any time.

1.1. Leiningen

A popular project built tool that provides a convenient way to pull libraries for your project. Follow the installation instructions at (https://leiningen.org).

```
lein new training
cd training
tree
cat project.clj
cat src/training/core.clj
```

As you can see, Leiningen created a project with one dependency; Clojure itself.

```
lein repl
```

1.2. The Read Eval Print Loop (REPL)

When you type in this code:

```
(+ 1 2)
```

Clojure evaluates it immediately and returns a result:

```
=> 3
```

Pressing the up arrow moves through your history.

The REPL is convenient for experimenting and doing informal tests. But the default REPL is not ideal for editing code.

1.3. Editor setup

Most popular editors have plugins to send commands from the editor to a REPL, do syntax highlighting and manage parenthesis. These are useful features, but I encourage you to prioritize learning Clojure ahead of configuring and learning new editor key combinations. It is difficult to do both at once!

For this course I recommend using IntelliJ (https://www.jetbrains.com/idea) with the Cursive Clojure plugin (https://cursive-ide.com). The main feature that sets Cursive apart is that it does error highlighting in the editor itself (https://cursive-ide.com/userguide).

- Open the project we just created and launch a REPL.
- Click file → open and browse to the project.clj file in the directory.
- In the file navigator, right click the project.clj file and select launch REPL.
- Press control+shift+T to send a form to the REPL.
- Press control+shift+A to see all actions available.

Alternative: Lighttable (http://lighttable.com)

- Click File→open folder.
- Browse to the “training” project directory that we created with lein.
- Navigate to training/src/core.clj in the left hand tree view.
- Press control+enter to send a form to the REPL.
- Press control+space for a list of commands available.
- Note that println will show up in the bottom console, which is hidden to begin.

You can also open a REPL in your browser: (https://repl.it/languages/clojure).

For other editor options see (https://cb.codes/what-editor-ide-to-use-for-clojure).

1.4. Exercises

Evaluate some math expressions in the REPL:

- Find the sum of 2 and 3
- What is 31 times 79?
- Divide 10 by 2
- Divide 2 by 10

Create a new project called `training`. Open `src/training/core.clj` with your editor, write some expressions, and send them to the REPL:

- Find the sum of 1, 2, and 3

- Send (println "hello world")

1.5. Answers

```
(+ 2 3)
=> 5
```

```
(* 31 79)
=> 2449
```

```
(/ 10 2)
=> 5
```

```
(/ 2 10)
=> 1/5
```

```
(+ 1 2 3)
=> 6
```

```
(println "hello world")
=> "hello world"
```

Chapter 2. Clojure Syntax

> The rhythmical unit of the syllable is at the back of all of it - the word, the phrase, the sentence, the syntax, the paragraph, and the way the heart moves when you read it.
>
> — Ali Smith

2.1. Primitive data types

Strings are enclosed in double quotes

```
"This is a string."
```

Character literals are preceded by a backslash

```
\a \b \c \newline \tab
```

Numbers can be Long

```
1
```

Double

```
3.14
```

BigInteger, suffixed with N

```
1000000000000N
```

BigDecimal, suffixed with M

```
1000000000000.1M
```

Expressed as exponents

```
1e3
```

Or ratio

```
2/5
```

Numbers are automatically promoted if they overflow during arithmetic.

Booleans are represented as `true` and `false`.

`nil` means nothing and is considered false in logical tests.

2.2. Collections: lists, vectors, maps, and sets

Lists are forms enclosed in parentheses.

```
()
```

Lists are evaluated as function calls.

```
(inc 1)
=> 2
```

The first element in the list is the function, and any following elements are arguments. Here we are calling the inc function on 1, which will return 2.

Quote yields the unevaluated form.

```
(quote (1 2))
=> (1 2)
```

Apostrophe is a syntactic shortcut for quote.

```
‘(1 2)
=> (quote (1 2))
=> (1 2)
```

Clojure prints sequences and lists the same way.

```
(seq ‘(1 2 3))
=> (1 2 3)
```

Sequences are lazy. Their values are only created as they are consumed.

Symbols are resolved.

```
inc
=> #object[clojure.core$inc]
```

```
foo
=> Exception: Unable to resolve symbol foo
```

To create an unresolved symbol, quote it

```
‘foo
=> foo
```

Vectors are enclosed in square braces

```
[1 2 3 4]
```

Vectors have order 1 lookup by index and count. Vectors are used in preference to lists for cases where either could be used. Vectors do not require quoting and are visually distinct. You will rarely see or use lists.

Clojure compares by identity and by value. A vector with elements matching a sequence is equal to it.

```
(= [1 2 3] ‘(1 2 3))
=> true
```

Maps are key/value pairs

```
{"Language" "Clojure"
 "Version" 1.5
 "Author" "Rich Hickey"}
```

Maps have near constant time lookup by key. Maps are tuned to be fast. Maps are an excellent replacement for object fields.

Keywords are shorthand identifiers that do not need to be declared. Keywords begin with a colon.

```
:language
```

Keywords are often used as keys in hashmaps; similar to fields in an object.

```
{:language "Clojure"
 :version 1.5
 :author "Rich Hickey"}
```

Keywords can be namespaced.

```
:timothy.example/rect
```

Double colon is shorthand for a fully qualified keyword in the current namespace.

```
::rect
=> :timothy.example/rect
```

Sets are written as

```
#{1 2 3}
```

Sets have near constant time membership lookup, with a high branching factor.

Collections can be combined and nested

```
{[1 2] {:name "diamond" :type :treasure}
 [3 4] {:name "dragon" :type :monster}}
```

This is a map that has vector coordinates as keys and maps as values.

2.3. Invoking functions

To call a function, wrap it in parenthesis:

```
(inc 1)
=> 2
```

The first element in a list is a function to be called. The remaining elements are the arguments to the function.

2.4. Defining vars

A var is used to store a mutable reference to a value. Vars are unbound if no value is supplied.

```
(def x)
x
=> #object[clojure.lang.Var$Unbound "Unbound: #'user/x"]
```

It is more common to supply an initial value.

```
(def x 1)
x
=> 1
```

Def created a var named `x` which is bound to the value `1`. Vars are automatically dereferenced when evaluated.

To represent values that changes over time, you can use an atom.

```
(def a (atom 1))
(swap! a inc)
@a
=> 2
```

We defined `a` to be an atom with initial value `1`, then swapped the atom's value with the `inc` function. We retrieved the value of the atom by dereference it with `@`. The current value of `a` is now `2`, the increment of `1`. `@` is shorthand for `deref`.

```
(deref a)
=> 2
```

Atoms provide compare and set, which is suitable for non-transactional changes. Refs provide transactional change, which is suitable for multi-threaded change management. Agents provide update serialization as an alternative strategy for multi-threaded change.

Deref also blocks and gets the result of futures, promises and delays, which are operations that do not block until dereferenced.

2.5. Binding names with let

Symbols:

- begin with an alphabet character
- can contain numbers and punctuation
- are usually lowercase words separated with hyphens
- must be bound to values before they can be evaluated

Symbols can be bound to a value in a scope with let.

```
(let [x 1]
  (inc x))
=> 2
```

The symbol x is bound to the value 1, and the function inc is called on x, resulting in 2.

The binding scope is within the parentheses enclosing the let form, and will shadow any existing bindings. It is preferable to use let instead of def for values that can be contained in a scope. Vars can be changed, but you should almost never modify them directly. Instead Clojure provides local bindings, atoms, refs and agents for managing change.

2.6. Destructuring (also known as binding forms)

```
(let [[x y] [1 2]]
  (+ x y))
=> 3
```

Destructing is providing a literal data structure containing symbols that get bound to the respective parts of a value with a matching structure. Where we might otherwise bind the vector [1 2] to a single symbol, here we destructure two symbols x and y by providing a pattern that matches the vector.

```
(defn normalize
  "Divide all dimensions by the sum of squares"
  [[x y]]
  (let [length (Math/sqrt (+ (* x x) (* y y)))]
    [(/ x length) (/ y length)]))
```

Note that function arguments are already a destructured vector. The above case is an example of a vector of arguments which contains a vector of x and y.

Destructuring avoids us having to extract substructure manually:

```
(defn normalize1 [v]
  (let [x (first v)
        y (second v)
        length (Math/sqrt (+ (* x x) (* y y)))]
    [(/ x length) (/ y length)]))
```

```
(defn normalize2 [[x y]]
  (let [length (Math/sqrt (+ (* x x) (* y y)))]
    [(/ x length) (/ y length)]))
```

Destructuring is also useful in for comprehensions and loops:

```
(defn invert [m]
  (into {} (for [[k v] m]
             [v k])))
```

In Clojure, for expressions are a convenient syntax alternative to map which also allows additional constraints to be expressed.

```
(for [i (range 10)
      :when (odd? i)]
  (* i i))
=> (1 9 25 49 81)
```

There is no need to restrict normalize to use 2 dimensions, instead we can write a generic version:

```
(defn normalize
  "Divide all dimensions by the sum of squares"
  [dims]
  (let [squares (map * dims dims)
        length (Math/sqrt (reduce + squares))
        by-length #(/ % length)]
    (map by-length dims)))
(normalize [3 4]) -> (0.6 0.8)
(normalize [3 4 5]) -> (0.424 0.566 0.707)
```

Variadic functions are destructured using `&`. Variadic means variable number of arguments. Arity means number of arguments.

```
(defn sub [& vs]
  vs)
```

```
(sub 1 2 3 4)
=> (1 2 3 4)
```

Which produces a vector. Apply expands the vector arguments. Most mathematical functions are variadic:

```
(+ 1 2 3)
=> 6
```

Destructuring is nested, so you can use it to pull out sub-values without resorting to getter functions.

Common opportunities for destructuring are:

Values in a map:

```
(:field1 x)
(:field2 x)
```

```
{:keys [field1 field2]} x
```

Values in a sequence:

```
(first x)
(rest x)
```

```
[a & more]
```

Nested destructuring

```
(get-in x [:a :b])
```

```
{{b :b} :a}
```

2.7. Namespaces

Namespace forms occur at the start of files.

```
(ns training.core
  (:require [clojure.string :as string])
  (:import [java.util Date]))
```

```
(string/upper-case "shout")
```

The namespace must match the path and filename. The namespace training.core Must be defined in the `src/training/core.clj` file. Filename hyphens are replaced with underscores, and dot separators indicate directories.

The `ns` form allows us to require other namespaces and import java Classes. There are other valid `ns` forms which are best to be avoided and so are not shown here. If you do see them in other code, just know that you can and should achieve the same thing with the regular ns form described previously.

Clojure programs are written in expressions which are evaluated to results. If an expression needs to be compiled, it will be. Programs can be loaded from files or evaluated dynamically.

2.8. Regex

Regular expressions are written as `#"pattern"`

```
(re-seq #"\w+" "the quick brown fox")
=> ("the" "quick" "brown" "fox")
```

2.9. Exercises

Write code into a new file called `src/training/syntax.clj`, and send the lines to the REPL as you enter them.

- Set up the new namespace called `training.syntax`
- Define a var called `message` bound to the string `"greetings"`.
- Print out the value of the var `message`.
- Create a `let` binding that binds the symbol `message` to `"well hello there"`, and prints out `message` inside the `let` block.
- Print out message again, outside of the `let` block.
- Create a let binding that destructures the map `{:greeting "good morning", :tone "happy"}` and prints the greeting and tone inside the let block.
- Destructure a single map input containing `{:greeting "good morning", :tone "happy"}` and return a string combining greeting and tone. Use the `str` function.

2.10. Answers

```
(ns training.syntax)
=> nil
```

```
(def message "greetings")
=> #'hello-clojure/message
```

```
(prn message)
=> "greetings"
   nil
```

Note the prn and println behave slightly differently; prn keeps the quotes around strings. This is often useful when experimenting, because you can visually see the type of the values more clearly.

```
(let [message "well hello there"]
  (prn message))
=> "well hello there"
   nil
```

```
(prn message)
=> "greetings"
```

Note that the message global var is still the original value.

```
(def m {:greeting "good morning", :tone "happy"})
```

```
(let [{:keys [greeting tone]} m]
  (prn greeting tone))
=> "good morning" "happy"
```

```
(defn hi [{:keys [greeting tone]}]
  (str greeting " - " tone))
```

```
(hi m)
=> "good morning - happy"
```

Chapter 3. Functions

> The chief function of the body is to carry the brain around.
>
> — Thomas A. Edison

3.1. Defining functions

Functions are defined like this:

```
(defn square [x]
  (* x x))
```

All functions return a result, the result of the last expression in the form. Defn binds the symbol square to a var which refers to a function which returns the result of multiplying the input parameter x by itself.

```
(square 2)
=> 4
```

When evaluated, a list containing square in the first position causes the var bound to square to be automatically dereferenced to the function, which is called on the arguments.

Mathematical operators are regular functions which must be written in prefix notation.

```
(+ (square 2) (square 3))
=> 13
```

Function arguments are evaluated from left to right before the function is called.

Unnamed functions are written as

```
(fn [a]
  (inc a))
```

Unnamed functions are also called anonymous functions and Lambda expressions. There is a special syntax for creating unnamed functions.

```
#(inc %)
```

Is a function which increments a single argument.

```
(#(inc %) 1)
=> 2
```

Closures are functions that capture values from the environment.

```
(let [who "world"]
  (defn greet []
    (str "Hello " who))
(greet)
=> "Hello world"
```

Functions are values and can be passed as arguments to other functions. Functions that take a function as an argument are called higher order functions.

```
(defn higher-order-function [f]
  (f))
(higher-order-function greet)
=> "Hello world"
```

Map is function that calls a function on every element in a sequence

```
(map #(inc %) [1 2 3])
=> (2 3 4)
```

Map is a higher order function because the first argument is a function. Unnamed closures are useful as arguments to higher order functions.

```
(let [x 5]
  (map #(+ x %) [1 2 3]))
=> (6 7 8)
```

Here we have the symbol x bound to 5. We call the map function. Our first argument is an unnamed function that captures x from the environment; a closure. The closure is called on every element of the vector `1 2 3`, resulting in a sequence `6 7 8`. Higher order functions, closures, and unnamed functions are terms that describe specific uses of functions that allow concise expressions.

3.2. Pre- and post-conditions

You can make assertions about inputs and outputs of a function. Place a map after the arguments vector containing :pre and :post, which are a sequence of conditions which must hold true.

```
(defn f [x]
  {:pre [(pos? x)]
   :post [(neg? %) (int? %)]}
  (- x))
```

```
(f 1)
=> -1
```

```
(f -1)
=> AssertionError Assert failed: (pos? x)
```

```
(f 1.5)
=> AssertionError Assert failed: (int? %)
```

In practise pre and post are rarely used. It is more common to check for a condition and throw an exception:

```
(defn f [x]
  (when-not (pos? x)
    (throw (ex-info "bad input" {:x x}))
  (let [result (- x)]
    (if (and (neg? result) (int? result))
      result
      (throw (ex-info "bad result" {:x x}))))
```

Or to use a schema or spec (which will be covered later in the course).

While pre and post are more concise, they suffer the following drawbacks: Syntax is easy to get wrong, resulting in no assertion being made Assertions can be disabled Less control over error description and handling

3.3. Anonymous functions

We usually define functions with defn, which creates a global var to hold our function. But sometimes the function need not be globally available. We can specify functions without names like so:

```
(fn [x]
  (inc x))
```

But we would only do this if we wanted to make use of them in some way. The simplest way to use a function is to call it immediately:

```
((fn [x]
   (inc x)
 1)
=> 2
```

The function appears as the first thing in a list, so is called on the argument 1, and evaluates the body of the function to calculate 2.

Another way to make use of an anonymous function is to bind it in a let form:

```
(let [f (fn [x]
          (inc x))]
  (f 2))
=> 3
```

In Clojure it is very common to pass a function as the argument to another function:

```
(map inc [1 2 3])
=> (2 3 4)
```

So having a way to specify an anonymous functions is helpful:

```
(map (fn [x]
       (* x x))
     [1 2 3 4])
=> (1 4 9 16)
```

You can name a function without creating a global var:

```
(fn add-one [x]
  (inc x))
```

Naming a function has several benefits:

- The name serves as a summary of the purpose of the function
- The name will appear in stacktraces, giving a searchable clue in your code
- The function can call itself
- The name will not be available outside the function

Note that

```
(defn f [x]
  (inc x))
```

is shorthand for

```
(def f
  (fn [x]
    (inc x)))
```

3.4. Function literals

There is a special syntax for creating anonymous functions concisely:

```
#(inc %)
#(+ %1 %2)
```

This allows the construction of very terse but powerful expressions:

```
(map #(* % %) [1 2 3 4])
=> (1 4 9 16)
```

I encourage you to use the (fn) form as much as possible instead of the #() form, it is not much more typing and affords more opportunity to name parameters and functions in meaningful ways which will describe your program better. For example:

```
(map (fn square [x]
       (* x x))
     [1 2 3 4])
=> (1 4 9 16)
```

Is longer, but provides a semantic summary of the operation and a hint at the expected input values.

3.5. Keyword and variadic arguments

```
(defn f [& args]
  args)
(f 1 2 3)
=> (1 2 3)
```

Variadic arguments sometimes introduce two disadvantages: Causing callers to have to use apply Bypasses arity checking

An antipattern is

```
(defn f [x & [y]]
  (if y
    (+ x y)
    (inc x)))
```

Prefer instead

```
(defn f
  ([x] (inc x))
  ([x y] (+ x y)))
```

Clojure supports keyword arguments, but this style is discouraged because it prevents users from passing a map of options. We cannot apply a map to a keyword argument function, so use a map argument instead of keyword arguments.

3.6. Exercises

Create a new namespace called `fun-functions`. Define the following functions and call them with some test input:

- A function that computes the square of an input number. What is the square of 55?
- A function that takes a number as input, ensures that the number is less than 100, and returns the square of the square of the input.
- A function that takes two numbers as input, and returns a vector where the first element is the second input, and the second element is the sum of the first and second input.

3.7. Answers

```
(defn square [x]
  (* x x))
(square 55)
=> 3025
```

```
(defn square-of-square [x]
  (if (< x 100)
    (square (square x))
    (throw (ex-info "Input too large" {:x x}))))
(square-of-square 2)
=> 16
(square-of-square 123)
=> ExceptionInfo Input too large
```

```
(defn fib-step [a b]
  [b (+ a b)]))
(fib-step 1 1)
=> [1 2]
(fib-step 1 2)
=> [2 3]
(fib-step 2 3)
=> [3 5]
```

Chapter 4. Challenge 1: Corgi Cover eligibility

Insuricorp is about to launch a marketing campaign for a new “corgi cover” policy. Only certain people are eligible to register for “corgi cover”. To be eligible they must own a corgi, live in either Illinois (IL), Washington (WA), New York (NY), or Colorado (CO). You are tasked with building a system to validate applications for the policy.

4.1. Part 1:

Write a function that takes as input a state and corgi-count, and returns a boolean indicating the person’s eligibility for the “corgi cover” policy.

4.1.1. Test data:

Name	State	Corgi count	Existing policy count
Chloe	IL	1	0
Ethan	IL	4	2
Annabelle	WY	19	0
Logan	WA	2	1

See `if` `=`.

4.2. Part 2:

A focus group of corgi owners has revealed that “corgi cover” needs to be offered at 3 different tiers: “corgi cover silver”, “corgi cover gold”, and “corgi cover platinum”. Platinum is available when covering 7 or more corgis OR covering at least 3 corgis and also having one other policy with Insuricorp. Gold is available when covering at least 3 corgis. Silver is the original “corgi cover” policy. Create a new function that takes an additional argument policy-count and returns a keyword indicating their eligibility.

See `cond`.

4.3. Part 3:

The “corgi cover” applications Insuricorp collect contain more information than necessary to determine eligibility. Create a new function that takes as input a single map data structure as input instead of multiple inputs. It should pick out the values that it needs from the input map. Create some test data and feed it to your function. The data should look something like:

```
{:name "Chloe", :state "IL", :corgi-count 1, :policy-count 0}
```

4.4. Part 4:

Insuricorp just merged with Megacorp. Platinum level corgi cover is now offered to people with an existing Megacorp policy as well. Because the company is still restructuring, the policy-count input still only contains Insuricorp data. But a new input has been made available to you which is a map of people to policies.

```
{"Chloe" ["secure goldfish"]
 "Ethan" ["cool cats cover" "megasafe"]}
```

Create a new function that takes as inputs two maps: the application, and the existing policies. It should apply the same logic, but make use of the Megacorp data.

Chapter 5. Testing with clojure.test

> The greatest test of courage on earth is to bear defeat without losing heart.
>
> — Robert Green Ingersoll

5.1. Defining tests with deftest

You can define a test in any file, but it is common to put all test code in a separate “test” directory, and to create namespaces that mirror the “src” directory but have -test appended. So if we have a source file `src/my_namespace.clj` then we create a test file as `test/my_namespace_test.clj`.

Test namespaces are normal Clojure namespaces. Test related functions come from the `clojure.test` namespace, so it is common to refer all symbols from `clojure.test` for convenience:

```
(ns my-namespace-test
  (:require [clojure.test :refer :all]))
```

A test is just a function that takes no arguments and will be called by the Clojure test runner.

```
(deftest my-test
  (prn "My test ran"))
```

You can run the tests manually from the REPL:

```
(run-tests)
```

```
=> "My test ran"
Ran 0 tests containing 0 assertions.
0 failures, 0 errors.
{:test 0, :pass 0, :fail 0, :error 0, :type :summary}
```

To run all tests in a project from the command line:

```
$ lein test
```

```
=> "My test ran"
Ran 0 tests containing 0 assertions.
0 failures, 0 errors.
{:test 0, :pass 0, :fail 0, :error 0, :type :summary}
```

5.2. lein-test-refresh

Lein-test-refresh is a Leiningen plugin that reloads code and re-runs tests when you save a file. https://github.com/jakemcc/lein-test-refresh.

Add lein-test-refresh to your `~/.lein/profiles.clj`. It should look similar to below.

```
{:user {:plugins [[com.jakemccrary/lein-test-refresh "0.22.0"]]}}
```

Alternatively you may add it to your `project.clj`.

```
(defproject sample
  :dependencies [[org.clojure/clojure "1.8.0"]]
  :profiles
  {:dev
   {:plugins [[com.jakemccrary/lein-test-refresh "0.22.0"]]}})
```

Now you can watch for changes from the command line:

```
$ lein test-refresh
```

If you change `my-test` now to print a new message, the tests are re-run as soon as you save the file… giving immediate feedback on your change.

```
(deftest my-test
  (prn "My test ran immediately"))
```

Seeing as saving the file executes code, you can use lein-test-refresh like a REPL.

5.3. Assertions with is and are

Let’s begin with a false assertion:

```
(deftest my-test
  (is (= 1 (inc 1))))
```

```
=> FAIL in (my-test)
   expected: (= 1 (inc 1))
     actual: (not (= 1 2))
```

And then convert it to a true assertion:

```
(deftest my-test
  (is (= 2 (inc 1))))
```

```
=> Ran 1 tests containing 1 assertions.
   0 failures, 0 errors.
```

We have written a test that makes an assertion about the function `inc`. Most tests check for equality with the expected value first, and the actual value second. The expected value is a literal expression and the actual is a call to the function under test. However you are not limited to following this for every test case. You can use any truthy assertion. Here is an example that does not do equality checking:

```
(deftest my-test
  (is (odd? 1)))
```

If your assertion expression is not self explanatory, supply an optional string argument which describes the assertion:

```
(deftest my-test
  (is (= (* 5 5) (+ (* 3 3) (* 4 4)))
    "The square of the hypotenuse is equal to the sum of the squares of the other two
sides"))
```

And to group assertions into logical blocks, use the testing form:

```
(deftest math-test
  (testing "basic math"
    (is (odd? 1))
    (is (= 2 (inc 1))))
  (testing "pythagoras"
    (is (= (* 5 5) (+ (* 3 3) (* 4 4)))
    "The square of the hypotenuse is equal to the sum of the squares of the other two
sides"))
```

It is also possible to more concisely express multiple assertions using the are form:

```
(are [x y] (= x y)
     2 (+ 1 1)
     4 (* 2 2))
```

However I recommend you avoid this form. It is easy to make an error in the syntax, and can be confusing. Furthermore line numbers are not preserved, so a failing test case is harder to identify.

Occasionally we need to assert that an exception is thrown:

```
(defn bad [x]
  (throw (ex-info "oh no" {})))
```

```
(deftest test-exception
  (is (thrown-with-msg? Exception #"oh no"
        (bad 42))))
```

5.4. Test fixtures

Test fixtures are for setting up and tearing down resources required by your tests. We can specify `:once` fixtures that execute one time for all tests in the namespace, or `:each` fixtures that run around each test in the namespace.

A fixture is simply a function that takes a test and executes it. Recall that tests are functions.

```
(use-fixtures :once
  (fn print-enter-exit [tests]
    (println "before")
    (tests)
    (println "after")))
```

Now the test runner will print out “before”, execute the tests in the namespace, and then print out “after”.

```
(use-fixtures :every
  (fn capture-prints [f]
    (with-out-str (f))))
```

Here we prevent printing within our function from appearing in the console. Usually we want our tests to make assertions, but not produce output. Otherwise the test report can be cluttered.

Another common use case is when doing database tests, we can wrap the test execution inside a transaction and rollback after the test completes. This avoids cleaning up data after the tests run, as no data was created.

5.5. Using with-redefs for mocking behavior

Often when we are writing tests we want to isolate particular behaviors. Some parts of a function might not be appropriate to occur during the test. We can conveniently replace the definition of any var during a test using with-redefs:

```
(defn post [url]
  {:body (str "Hello world")})
```

```
(deftest test-post
  (with-redefs [str (fn [& args]
                      "Goodbye world")]
    (is (= {:body "Goodbye world"}
           (post "http://service.com/greet")))))
```

At first glance this is very similar to let, but notice that a let would not work in this example. We changed the behavior of the str function whose definition is outside the scope of the test. We replaced it with an anonymous function that always returns “Goodbye world” regardless of its inputs. Note that we could have used (constantly "Goodbye world") instead, which produces an anonymous function just like the one we defined.

5.6. Debugging

While working on a function, sometimes it is useful to print out an intermediary value. One way to accomplish this is using doto. Say that we were working on a complicated nested function:

```
(defn shazam [a b]
  (/ 1 (+ a b) (+ a (* a b))))
```

And we wanted to see what `(+ a (* a b))` was evaluating to in the context of the function call. We can temporarily wrap the expression in `(doto … (prn))`.

```
(defn shazam [a b]
  (/ 1 (+ a b) (doto (+ a (* a b)) (prn "***"))))
```

```
(shazam 1 2)
=> 3 "***"
   1/9
```

The difference from wrapping with just `prn` is that `prn` always returns `nil`, while `doto` will cause the `prn` side-effect to occur, but will return the original argument. This is also very useful when interacting with Java, because you can construct an object, call various methods on it, and return the object constructed.

```
(doto (new java.util.HashMap)
  (.put "a" 1)
  (.put "b" 2))
=> {"a" 1, "b" 2}
```

5.7. Exercises

- Start lein-test-refresh running in your existing project directory.
- Create a new namespace in the “test” directory called `training.core-test`
- Write a function called `pythag` that returns the square root of the sum of squares for two inputs.
- Write a test containing an assertion that exercises your function. Expect `5` when passing `4` and `3` as arguments.
- Write another test case with different inputs.
- Introduce a bug into pythag to make sure your tests discover the problem.
- Fix `pythag` so that all tests pass.
- Copy the test `test-post` from the "with-redefs" section and modify it so that it counts how many times `str` gets called. Call `post` several times and make an assertion about how many times `str` should get called.

5.8. Answers

```
(defn pythag [a b]
  (Math/sqrt (+ (* a a) (* b b))))
```

```
(deftest test-pythag
  (is (= 5 (pythag 4 3)))
  (is (= 13 (pythag 12 5))))
```

```
(defn post [url]
  {:body (str "Hello world")})
```

```
(deftest test-post
  (let [c (atom 0)]
    (with-redefs [str (fn [& args]
                        (swap! c inc)
                        "Goodbye world")]
      (post "http://service.com/greet")
      (post "http://service.com/greet")
      (post "http://service.com/greet")
      (is (= 3 @c)))))
```

Chapter 6. Control Flow

> Goals allow you to control the direction of change in your favor.
>
> — Brian Tracy

Clojure provides special forms for control flow. Special forms are built in primitives that behave differently from functions. We already saw several special forms in action: `def`, `let`, `quote` and `fn` are all special forms. The main thing that is different about them is that they don't evaluate all their arguments like a regular function call.

6.1. Conditionals: if, when, cond

Another special form is if which chooses between two options.

```
(if (pos? 1)
  (println "one is positive")
  (println "or is it?"))
=> "one is positive"
```

Only one branch is evaluated, whereas a function call evaluates all arguments.

Often we want to execute some code only when a condition is met:

```
(when (pos? 1)
  (println "one is positive")
  (println "multiple expressions allowed"))
=> "one is positive"
   "multiple expressions allowed"
```

When the test fails, nothing is evaluated, when it passes, everything in the body is evaluated.

Cond allows for multiple branches.

```
(def x {:cake 1})
(cond (= x 1) "one"
      (= x :cake) "the cake is a lie"
      (map? x) "it's a map!"
      :else "not sure what it is")
=> "it's a map!"
```

Note that `:else` is not a special keyword, it just happens to be a truthy value.

6.2. Recursion

Functions that call themselves are called recursive. Here is an example of recursion:

```
(defn sum-up [coll result]
  (if (empty? coll)
    result
    (sum-up (rest coll) (+ result (first coll)))))
```

In Clojure there is a special way to do recursion which avoids consuming the stack:

```
(defn sum-up-with-recur [coll result]
  (if (empty? coll)
    result
    (recur (rest coll) (+ result (first coll)))))
```

Recur can only occur at the last position of a function (where scope can be discarded).

6.3. Loops

Loop establishes bindings, and allows you to recur back to the start of the loop with new values.

```
(loop [a 0
       b 1]
  (if (< b 1000)
    (recur b (+ a b))
    a))
=> fib number below 1000
```

6.4. Exception handling

You can work with exceptions using try catch finally and throw.

```
(try
  (inc "cat")
  (catch Exception e
    (println "cat cannot be incremented")))
```

6.5. Comments

Anything following a semicolon is a comment

```
; this is an inline comment
;; this is a function level comment
```

Less common is the comment form:

```
(comment anything)
```

And a special form for complete removal of any form it is prefixed to

```
#_(this form is removed)
```

Which is handy for temporarily removing a form when modifying code. You can use hash-underscore multiple times to comment out multiple forms.

```
#_#_ ignored-1 ignored-2
```

I call this the bug eyes operator, because it looks like a bug emoji.

Commas are optional and treated as whitespace.

```
(= {:a 1, :b 2, :c 3} {:a 1 :b 2 :c 3})
```

6.6. Exercises

- Create a function that given a test score between 0 and 100 returns a grade A B C D or F for fail.
- Write a function that takes a number and uses a loop to calculate the factorial of that number. Factorial 5 is 1*2*3*4*5.
- Write a new version of factorial that does not use a loop but recursively calls itself.
- Write a loop for the Fibonacci sequence (1 1 2 3 5 8 13) that finds the maximum Fibonacci number less than 100. The sequence is defined by n2 = n1 + n0.

6.7. Answers

```
(def grade [score]
  (cond (>= score 90) "A"
        (>= score 80) "B"
        (>= score 70) "C"
        (>= score 60) "D"
        :else "F"))
```

```
(defn factorial [n]
  (loop [acc 1
         x n]
    (if (<= x 1)
      acc
      (recur (* acc x) (dec x)))))
(deftest factorial-test
  (is (= 120 (factorial 5))))
```

```
(defn factorial2
  ([n] (factorial 1 n))
  ([acc n]
   (if (<= n 1)
     acc
     (recur (* acc n) (dec n)))))
(deftest factorial2-test
  (is (= 120 (factorial2 5))))
```

```
(defn fib [limit]
  (loop [a 1
         b 1]
    (if (>= b limit)
      a
      (recur b (+ a b)))))
(deftest fib-test
  (is (= 89 (fib 100))))
```

Chapter 7. Functional Programming

> If you don't love something, it's not functional, in my opinion.
>
> — Yves Behar

7.1. Pure functions and side effects

You have probably noticed that Clojure functions always return a value. Moreover they usually return a useful result, not just a nil. There is a distinction to be made between functions which produce useful result values from functions which cause side-effects.

Functions that produces side effects are often called in a way that discards their result. For example calling `(println "hi")` is done not because we want a result. `println` returns `nil`, which is useless. What we want is to print to System out the string `"hi"`, which occurs as a side-effect of us calling the function. Contrast that with calling `(str "hi" "there")`, which returns a new string `"hithere"`; no side-effects occur.

A function with no side-effects is a pure function. Calling pure functions with a given input always results with the same corresponding output. Note that `rand` is not a pure function even though it returns a useful result, because it produces a different output every time.

Pure functions are desirable because they are:

- easier to reason about
- easier to combine
- easier to test
- easier to debug
- easier to parallelize

The Clojure api provides many pure functions. For example `conj` does not add something to a vector, it returns a completely new vector!

```
(def v [1 2])
(conj v 3)
=> [1 2 3]
```

```
v
=> [1 2]
```

In this example we can see that v remained unchanged. Clojure implements data structures that enable this to happen efficiently. Using a regular Java vector would require duplicating the vector, but Clojure makes use of a technique called shared structure to provide immutable data structures that don't require the entire object to be duplicated.

Clojure does allow side-effects, indeed they are very useful. It is good style to keep side-effects co-located instead of having them occur throughout various parts of the code. We will see some good examples of this philosophy in action later in the course when we get to atoms. We can use pure function to calculate the next value to be assigned to an atom given the current value. The logic is separate from the side effect.

7.2. Apply, partial, and comp

If you have 4 numbers and want the max, you can call

```
(max 1 2 5 3)
=> 5
```

But what if you have a sequence of many numbers? What if you don't know how many numbers there will be? Fortunately there is a way to convert a sequence of arguments into a function call:

```
(apply max [1 2 5 3])
=> 5
```

This is especially useful when calling variadic functions like max. Note that we could have alternatively reduced over the sequence, but apply is much more concise and clear about the intent.

In Clojure we often pass functions as values, so there is a convenient way to create a function that consumes some arguments that can be used with additional arguments later:

```
(partial + 1)
```

Creates a function that adds 1 to any number of arguments supplied. It returns a function that is equivalent to:

```
(fn [& args]
  (apply + 1 args))
```

So let's see how we might make use of that:

```
((partial + 1) 2 3)
=> 6
```

```
(map (partial / 1) (range 1 5))
=> (1 1/2 1/3 1/4)
```

In the previous example, we could have instead written:

```
(map #(/ 1 %) (range 1 5))
=> (1 1/2 1/3 1/4)
```

7.3. Functions on sequences: map, reduce, and friends

To really embrace Clojure is to think in terms of sequences and data structures.

The most basic way to construct a sequence is like so:

```
(cons 1 ())
=> (1)
```

```
(cons 3 (cons 2 (cons 1 ())))
=> (3 2 1)
```

But Clojure provides several easier ways to create a sequence:

```
(range 10)
=> (0 1 2 3 4 5 6 7 8 9)
```

Be careful though, Clojure can produce infinite sequences (don't do this in a REPL):

```
(range)
```

This would attempt to keep producing numbers forever. (Press control-c to cancel the REPL if you did try this). There is a way to limit the amount of values to take:

```
(take 5 (range))
=> (0 1 2 3 4)
```

```
(drop 5 (take 5 (range)))
=> (5 6 7 8 9)
```

Clojure has an excellent sequence abstraction that fits naturally into the language. From a vector `[1 2 3 4]` we can find the odd numbers by calling the filter function:

```
(filter odd? [1 2 3 4])
=> (1 3)
```

Here we called the `filter` function with two arguments: the `odd?` function and a vector of integers.

filter is a higher order function, since it takes an input function to use in its computation. The result is a sequence of odd values. Functions like filter that operate on sequences call seq on their arguments to convert collections to sequences. The underlying mechanism is the ISeq interface, which allows many collection data structures to provide access to their elements.

map is a function that calls another function for every element in a sequence:

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

The result is a sequence of the increment of each number in [1 2 3 4].

Sequences can be used as input arguments to other functions as shown here:

```
(filter odd? (map inc [1 2 3 4]))
=> (3 5)
```

Here we filtered by odd? the values from (2 3 4 5), which was the result of calling map.

To aggregate across a sequence, use reduce:

```
(reduce * [1 2 3 4])
=> 24
```

For each element in the sequence, reduce computes (* aggregate element) and passes the result of that as the aggregate for the next calculation. The first element 1 is used as the initial value of aggregate. The final result is 1 * 2 * 3 * 4.

Clojure provides a built-in function for grouped aggregates:

```
(group-by count ["the" "quick" "brown" "fox"])
=> {3 ["the" "fox"], 5 ["quick" "brown"]}
```

3 letter words are "the" and "fox", whereas 5 letter words are "quick" and "brown".

filter is like a Java loop:

```
for (i=0; i < vector.length; i++)
 if (condition)
     result.append(vector[i]);
```

map is like a Java loop:

```
for (i=0; i < vector.length; i++)
    result[i] = func(vector[i]);
```

`reduce` is like a Java loop:

```
for (i=0; i < vector.length; i++)
    result = func(result, vector[i]);
```

Sequence abstractions are like names for loops that you can add to your vocabulary to talk about and recognize different kinds of loops. Learning the names of the abstractions and patterns that replace loops is an effort, but it adds powerful words to a programmer's vocabulary. A large vocabulary facilitates reasoning more succinctly, communicating more effectively, and writing less code that does more.

Clojure provides a special form `#()` to create an anonymous function:

```
#(< % 3)
```

The `%` symbol is an implied input argument. This function takes one argument and returns `true` if the input argument is less than `3`, otherwise it is `false`. Anonymous functions are handy for adding small snippets of logic:

```
(filter #(< % 3) [1 2 3 4 5]))
=> (0 1 2)
```

This keeps only numbers less than `3`. Now let's create a sequence of odd/even labels for each number in the vector:

```
(map #(if (odd? %) "odd" "even") [1 2 3 4 5])
=> ("odd" "even" "odd" "even" "odd")
```

Sequence abstractions are more concise and descriptive than loops, especially when filtering multiple conditions, or performing multiple operations.

Clojure also has useful functions for constructing sequences:

```
(range 5)
=> (0 1 2 3 4)
```

```
(repeat 3 1)
=> (1 1 1)
```

```
(partition 3 (range 9))
=> ((0 1 2) (3 4 5) (6 7 8))
```

One situation that appears difficult to use a sequence abstraction in is when we have a vector of numbers and wish to perform a sequence operation that relies upon the previous value visited. For example, think about finding the sum of each pair in `[1 2 3 4 5]`. Using an imperative style loop we can peek into the vector at the previous value:

```
for (i=1; i < v.length; i++)
    print v[i] + v[i-1];
=> 3 5 7 9
```

Can we represent this as a sequence? Yes! Imagine two identical sequences offset slightly:

```
  [1 2 3 4 5]
[1 2 3 4 5]
```

The overlapping values are the pairs we want.

`map` can take multiple sequences from which to pull arguments for the input function:

```
(map + [1 3]
       [2 4])
=> (3 7)
```

Here `1` adds to `2` to make `3`, and `3` adds to `4` to make `7`.

`rest` is a function which returns the input sequence without its first element:

```
(def v [1 2 3 4 5])
(rest v)
=> (2 3 4 5)
```

Putting them together:

```
(map + v (rest v))
=> (3 5 7 9)
```

We called map on the addition function over both input sequences:

```
v        => (1 2 3 4 5)
(rest v) => (2 3 4 5)
```

The input sequences were of different lengths, so map stopped when the smallest sequence was exhausted. The result was a new sequence of the pairwise sums:

```
(3 5 7 9)
```

Why are sequence abstractions better than loops? When reading a loop you must comprehend the entire block of code to know what it does. As the loop body grows and changes you must mentally keep track of more complexity. Mistakes like “off by one” are hard to spot, and can creep in as the code changes. Testing requires the invasion of the loop with breakpoints. You may find yourself duplicating a loop to customize some similar operation. The loop abstraction is very easy to understand and use, but it does not provide leverage.

Imagine discovering a new requirement where you need to multiply all of those numbers together. The change is invasive to the imperative loop:

```
result = 1;
for (i=1; i < v.length; i++)
    result *= (v[i] + v[i-1]);
=> 945
```

The change occurs inside the loop with the addition and multiplication intertwined.

Contrast this with modifying the Clojure sequence. We compose a reduce with the original map expression:

```
(reduce * (map + v (rest v)))
=> 945
```

- `reduce`: Aggregate by multiplication the sequence
- `map`: adding items together from two sequences
- `pairing`: the sequence of elements in v, adjacent to the rest of v

This is dense, but descriptive code… if you know the vocabulary.

With a sequence you can write unit tests for the component sequences and operations, reuse the same sequence without writing new code, and reason about the transformations as composable parts.

Look out for opportunities to name your steps by identifying long expressions and creating a named function out of them.

Clojure exposes a sequence interface over data collections to a rich set of functions that compose well. Three important functional sequence concepts are: `filter`, which retains each item in a sequence where some function evaluates to be truthy; `map`, which selects new values by calling a function over input sequence(s) to create a new sequence; and `reduce`, which aggregates a sequence and returns a single value.

I invite you to take the “no loops” challenge. The next time you spot a loop stop and think about what sequence operation the loop represents. Think about how to rewrite the loop as sequence operations instead. It will take time and mental effort, but you will be rewarded with a deeper understanding of the problem being solved. Whenever you see a loop, think about how it could be expressed as a sequence. Sequences are loop abstractions that allow you to ignore the implementation details.

7.4. Threading operators

By now, you should be feeling the combinatorial power functions offer. Simple functions compose sequence operations together to build transforms. Clojure has almost one hundred functions related to sequences, so you should also be feeling wary of such dense code. If we keep adding layers of function calls, the code becomes cryptic:

```
(reduce * (filter odd? (map inc v)))
=> 15
```

With three layers of function calls, things are getting hard to keep in our head all at once. This expression may be easier to mentally process by starting from the innermost map, working out to filter, and then out to reduce last. But that is the opposite of our reading direction and locating the true starting point is difficult.

The presentation of sequence operations is clearer if you name intermediary results:

```
(let [incs (map inc v)
      odd-incs (filter odd? incs)]
  (reduce * odd-incs))
=> 15
```

Or use a thread last:

```
(->> v
    (map inc)
    (filter odd?)
    (reduce *))
=> 15
```

Threading is good for unwrapping deeply nested function calls, or avoiding naming intermediary steps that don’t have a natural name.

Thread first is similar, but passes the value in the first position

```
(-> 42 (/ 2) (inc))
=> 22
```

Note that for empty expressions, the parenthesis are optional.

```
(-> 42 (/ 2) inc)
=> 22
```

7.5. Data structures are functions!

Maps sets vectors and keywords are functions. They delegate to get. While it is possible to use get to access collections, calling the collection directly is more common.

```
(get {:a 1 :b 2} :a)
=> 1
```

```
({:a 1 :b 2} :a)
=> 1
```

```
(:a {:a 1 :b 2})
=> 1
```

This is useful because you don’t need to create a function to call get.

```
(map (fn [m] (get m :a)) [{:a 1} {:a 2} {:a 3}])
=> (1 2 3)
```

Can instead be written as:

```
(map :a [{:a 1} {:a 2} {:a 3}])
=> (1 2 3)
```

Where we are looking up the value associated with :a for each element in a vector of maps.

Sets implement get:

```
(get #{1 2 3} 2)
=> 2
```

```
(#{1 2 3} 2)
=> 2
```

```
(remove #{nil "bad"} [:a nil :b "bad" "good"])
```

And so do vectors:

```
(get [1 2 3] 0)
=> 1
```

```
([1 2 3] 0)
=> 1
```

7.6. Exercises

- Write a function that takes two inputs, and returns the sum of the numbers in a range between two input integers, including the two input numbers.
- Write a function that produces a sequence of powers of 2: (1 2 4 8 16 …)
- Write a function that takes a string and produces a sequence of characters with no vowels.
- Write a function that produces a sequence: (1 ½   ¼ …)
- Write a function that produces a sequence: (1 ½ ¼   …)
- Write a function that produces the Fibonacci sequence (1 1 2 3 5 8 13 21)

7.7. Answers

```
(defn sum-between [a b]
  (apply + (range a (inc b))))
(sum-between 3 5)
=> 12
```

```
(defn powers-of [n]
  (iterate #(* % n) 1))
(take 5 (powers-of 2))
=> (1 2 4 8 16)
```

```
(defn shorten [s]
  (remove #{\a \e \i \o \u} s))
(apply str (shorten "Clojure sets are functions"))
=> "Cljr sts r fnctns"
```

```
(defn fractions []
  (map / (repeat 1) (rest (range))))
(take 5 (fractions))
=> (1 1/2 1/3 1/4 1/5)
```

```
(defn fraction-powers [n]
  (map / (repeat 1) (powers-of n)))
(take 5 (fraction-powers 2))
=> (1 1/2 1/4 1/8 1/16)
```

```
(defn fib-step [[a b]]
  [b (+ a b)])
(defn fib-seq []
  (map first (iterate fib-step [1 1])))
(take 10 (fib-seq))
=> (1 1 2 3 5 8 13 21 34 55)
```

Chapter 8. Challenge 2: Processing files

Insuricorp branches collect applications for the “corgi cover” policy and periodically send them to headquarters in a large comma separated text file. You have been tasked with processing the files using the validation logic you built earlier.

8.1. Part 1:

Create a function that opens a file called corgi-cover-applications.csv and converts every row into a data structure and prints it. Next use that data structure as an input to your validation function and print the result. See `slurp` `line-seq` `clojure.string/split`.

8.2. Part 2:

The downstream Insuricorp systems will only be operating on corgi cover applications that pass your eligibility check. But the invalid corgi cover applications need to be sent back to the branches so that they can follow up with the customers on why they are not eligible. Create a new function that opens two output files and writes to them based upon your eligibility check. The files should be called `eligible-corgi-cover-applications.csv` and `ineligible-corgi-cover-applications.csv`.

8.3. Part 3:

A request has come in from several Insuricorp branches that if a person is ineligible for corgi cover, a short reason be supplied. That way the sales reps don’t have to spend time figuring out what they need to tell the customer. Create a new validation function that instead of returning a boolean, returns nil if no problems are found, or returns a string with the reason if a problem is found. Create a new processing function that splits the applications into two files based on the new validator.

8.4. Part 4:

As part of the Megacorp merger, the downstream systems are converting to JSON format. Create a new function that writes JSON data to an `eligible-corgi-cover-applications.json` file.

Chapter 9. Macros

> I never think about myself as an artist working in this time. I think about it in macro.
>
> — Frank Ocean

Macros manipulate the operand forms instead of evaluating them as input arguments. They are not functions, and cannot be used as values or arguments to functions. We already used a macro; defn is a macro for conveniently defining functions.

```
(defn square [x] (* x x))
```

Actually expands to a def and fn form:

```
(def square (fn [x] (* x x)))
```

The difference between macros and functions is that macro arguments are manipulated at compile time instead of evaluated. Macros allow the user to extend the syntax of Clojure, but macros are less useful than functions as they cannot be used as values or arguments to higher order functions.

9.1. Expanding macros

Macros provide syntactic sugar. Macros first expand to produce new code that then gets compiled. The form is expanded at compile time through manipulation of the form. You can examine the expansion using `macroexpand-1`:

```
(macroexpand-1 '(defn square [x] (* x x)))
=> (def my-namespace/square
     (clojure.core/fn
       ([my-namespace/x]
        (clojure.core/* my-namespace/x my-namespace/x))))
```

9.2. Defining macros

Consider two different definitions of zen:

```
(defmacro zen1 [x]
  (println "x:" x) x)
```

and

```
(defn zen2 [x]
  (println "x:" x) x)
```

Now call

```
(zen1 (+ 1 2))
=> x:(+ 1 2)
3
```

```
(zen2 (+ 1 2))
=> x:3
3
```

The final result is the same, but notice that the input to `zen1` was a list, where as the input to `zen2` was the result of evaluating the list. That's the key difference between a macro and a function.

Macros themselves are really just functions with a `:macro` flag set in their metadata, which causes them to be passed in the input forms unevaluated, and caused the result to be evaluated. This last part is less obvious… but think back to `zen1`… `x` was a list, we returned `x`, but the final result wasn't a list… it was `3`. The list was evaluated as a function call to `+`, resulting in `3`.

9.3. Syntax quoting

To help write macros there is a special quoting form called syntax-quote.

Back-quote (`` ` ``) Unquote (`~`) and Unquote-splicing (`~@`)

```
‘(1 2 ~(+ 1 2) ~@(map inc [3 4 5]))
=> (1 2 3 4 5 6)
```

All symbols in a syntax-quote form get fully qualified.

```
`(inc 1)
=> (clojure.core/inc 1)
```

Fully qualified symbols is desirable when creating macros, otherwise symbols may have another meaning in the context that the macro is expanded in:

```
(defmacro m1 []
  '(inc 1))
(defmacro m2 []
  `(inc 1))
(let [inc dec]
  {:m1 (m1)
   :m2 (m2)})
=> {:m1 0, :m2 2}
```

Within the `let` block, the symbol `inc` has a different meaning than normal. Because `m2` uses syntax quote, `inc` gets fully qualified to `clojure.core/inc` which does not collide with the `let` binding.

Fully qualified symbols avoids one source of collisions, but there is another:

```
(defmacro bad [expr]
  (list 'let '[a 1]
    (list 'inc expr)))
```

```
(bad 0)
=> 1
```

```
(def a 0)
(bad a)
=> 2
```

This might seem confusing, unless you notice that:

```
(macroexpand-1 '(bad a))
=> (let [a 1] (inc a))
```

Instead of inc operating on the input parameter, it is operating on an internal let bound value. To avoid this situation Clojure provides a let gensyms form which will produce a randomly named binding:

```
(defmacro good [expr]
  `(let [a# 1]
     (inc ~expr)))
```

```
(good a)
=> 1
```

```
(good 0)
=> 1
```

```
(macroexpand-1 '(good a))
=> (clojure.core/let [a__6500__auto__ 1] (clojure.core/inc a))
```

The `let` binding `a#` expands out to a randomly generated symbol unlikely to collide with existing symbols.

9.4. Code as data

You may have noticed when we write a macro, we are really writing a function that produces code. The output is code… as data, and we manipulate code… as data. Homoiconic means that the language text has the same structure as its abstract syntax tree (AST). This allows all code in the language to be accessed and transformed as data, using the same representation. Nested code is well represented as a data structure.

When working on a non-trivial macro a good strategy is:

- Step 1: Write a function!
- Step 2: Call your function from the macro.

Stated another way; keep the macro as small as possible, and offload transformations to functions.

9.5. Exercises

Create the following macros and test cases:

- Create a macro called ignore which accepts any number of expressions, does absolutely nothing, and always returns `nil`.

  ```
  (ignore (println "hello???") (inc 42))
  ```

- Define your own version of the when macro. When is like if, but only has one branch and allows multiple statements.

  ```
  (when2 (pos? x)
    (println "Positive:" x)
    (inc x))
  ```

- Write a spy macro. Spy wraps an expression and prints out its value.

```
(* (spy (+ 1 2)) 3)
=> Expression (+ 1 2) has value 3
   9
```

- Write your own version of the `or` macro

```
(or2 (pos? 1) (println "does not execute"))
```

9.6. Answers

```
(defmacro ignore  [expr]  nil)
```

```
(defmacro when2 [test & body]
  (list 'if test (cons ‘do body))
```

```
(defmacro spy [expr]
  `(let [result# ~expr]
     (println "Expression" '~expr "has value" result#)
     result#))
(macroexpand-1 '(spy (* 2 3)))
=> (clojure.core/let [result__6418__auto__ (* 2 3)]
     (clojure.core/println
       "Expression" (quote (* 2 3))
       "has value" result__6418__auto__)
     result__6418__auto__)
(+ 1 (spy (* 2 3)))
=> Expression (* 2 3) has value 6
   7
```

```
(defmacro or2
  ([] nil)
  ([x] x)
  ([x & next]
      `(let [or# ~x]
         (if or# or# (or ~@next)))))
```

Chapter 10. Parallel Programming and Concurrency

> Our moral traditions developed concurrently with our reason, not as its product.
>
> — Friedrich August von Hayek

10.1. Vars and dynamic scope

Vars are automatically derefed when evaluated, so it can seem like they are just a variable. But you can “see” the var itself using the var function or #' shorthand.

```
(def one-hundred 100)
=> #'training.core-test/one-hundred
```

```
(var one-hundred)
=> #'training.core-test/one-hundred
```

```
(deref #'one-hundred)
=> 100
```

The most common reason you would want to do that is to examine the metadata of a var:

```
(meta #'one-hundred)
=> {:line 73, :column 1, ...}
```

Metadata may be provided using `^{}`

```
(def x ^{:private true} 1)
```

You can attach whatever metadata you wish. These are the keys the compiler looks for:

```
:private
:doc
:author
:type
```

By default Vars are static. But Vars can be marked as dynamic to allow per-thread bindings. Within each thread they obey a stack discipline:

```
(def ^:dynamic x 1)
(def ^:dynamic y 1)
(+ x y)
=> 2
```

```
(binding [x 2 y 3]
         (+ x y))
=> 5
```

```
(+ x y)
=> 2
```

Bindings created with binding cannot be seen by any other thread. Likewise, bindings created with binding can be assigned to, which provides a means for a nested context to communicate with code before it on the call stack. This capability is opt-in only by setting a metadata tag: dynamic to true as in the code block above.

Functions defined with defn are stored in Vars, allowing for the re-definition of functions in a running program. This also enables many of the possibilities of aspect- or context-oriented programming. For instance, you could wrap a function with logging behavior only in certain call contexts or threads.

10.2. Delays, Futures, and Promises

10.2.1. Delays

Delays wrap an arbitrary body of code for evaluation at a later stage so that the code in question is not run unless the answer is asked for. Delays also cache the result value to prevent another execution. The body code will only run once, even if dereferenced concurrently.

```
(def d (delay (println "Hello world!") 42))
```

```
d
=> #object[clojure.lang.Delay {:status :pending, :val nil}]
```

```
(realized? d)
=> false
```

```
@d
=> Hello world!
   42
```

```
@d
=> 42
```

```
(realized? d)
=> true
```

We assign the delay to a var called `d`. We see that it starts in a pending state. Dereferencing `d` with `@` causes the code to run, printing `"Hello world!"` and returning `42`. Notice that the second dereference with `@` does not print `"Hello world!"` again, it only returns the already realized value of `42`.

10.2.2. Futures

Futures provide an easy way to spin off a new thread to do some computation or I/O that you will need access to in the future. The call style is compatible with delay. The difference is that the work begins immediately on another thread. The flow of control is not blocked. If you dereference a future, it will block until the value is available:

```
(def f
  (future (Thread/sleep 10000) 42))
```

```
f
=> #object[clojure.core$future_call {:status :pending, :val nil}]
```

```
(realized? f)
=> false
```

--- 10 seconds pass ---

```
(realized? f)
=> true
```

```
@f
=> 42
```

```
f
#object[clojure.core$future_call {:status :ready, :val 42}]
```

10.2.3. Promises

Promises are used in a similar way to delay or future in that you dereference them for a value, can check if they have a value with `realized?` and they block when you dereference them if they don't have a value until they do. Where they differ is that you don't immediately give them a value, but provide them with one by calling deliver:

```
(def p (promise))
(realized? p)
=> false
```

```
(deliver p "as-promised")
(realized? p)
=> true
```

```
@p
=> "as-promised"
```

Dereferencing works on futures, delays, promises, atoms, agents refs and vars.

10.3. Atoms, Refs, and Agents

Atoms provide a way to manage shared, synchronous, independent state. They are a reference type like refs and vars. You create an atom with atom, and can access its state with `deref`/`@`. Like refs and agents, atoms support validators. To change the value of an atom, you can use `swap!`. A lower-level `compare-and-set!` is also provided. Changes to atoms are always free of race conditions.

As with all reference types, the intended use of atom is to hold one of Clojure's immutable data structures. And, similar to ref's alter and agent's send, you change the value by applying a function to the old value. This is done in an atomic manner by `swap!` Internally, `swap!` reads the current value, applies the function to it, and attempts to `compare-and-set!` it in. Since another thread may have changed the value in the intervening time, it may have to retry, and does so in a spin loop. The net effect is that the value will always be the result of the application of the supplied function to a current value, atomically. However, because the function might be called multiple times, it must be free of side effects.

Atoms are an efficient way to represent some state that will never need to be coordinated with any other, and for which you wish to make synchronous changes (unlike agents, which are similarly independent but asynchronous).

While Vars ensure safe use of mutable storage locations via thread isolation, transactional references (Refs) ensure safe shared use of mutable storage locations via a software transactional memory (STM) system. Refs are bound to a single storage location for their lifetime, and only allow mutation of that location to occur within a transaction. In practise Refs are rarely used.

Like Refs, Agents provide shared access to mutable state. Where Refs support coordinated, synchronous change of multiple locations, Agents provide independent, asynchronous change of individual locations. Agents are bound to a single storage location for their lifetime, and only allow mutation of that location (to a new state) to occur as a result of an action. Actions are functions (with, optionally, additional arguments) that are asynchronously applied to an Agent's state and whose return value becomes the Agent's new state. Because actions are functions they can also be multimethods and therefore actions are potentially polymorphic. Also, because the set of functions is open, the set of actions supported by an Agent is also open, a sharp contrast to pattern matching message handling loops provided by some other languages.

Clojure's Agents are reactive, not autonomous - there is no imperative message loop and no blocking receive. The state of an Agent should be itself immutable (preferably an instance of one of Clojure's persistent collections), and the state of an Agent is always immediately available for reading by any thread (using the deref function or reader macro @) without any messages, i.e. observation does not require cooperation or coordination.

Agent action dispatches take the form (send agent fn args*). send (and send-off) always returns immediately. At some point later, in another thread, the following will happen:

- The given fn will be applied to the state of the Agent and the args, if any were supplied. The return value of the given fn will become the new state of the Agent.
- If any watchers were added to the Agent, they will be called. See add-watch for details.
- If during the function execution any other dispatches are made (directly or indirectly), they will be held until after the state of the Agent has been changed.
- If any exceptions are thrown by an action function, no nested dispatches will occur, and the exception will be cached in the Agent itself. When an Agent has errors cached, any subsequent interactions will immediately throw an exception, until the agent's errors are cleared. Agent errors can be examined with agent-error and the agent restarted with restart-agent.

The actions of all Agents get interleaved amongst threads in a thread pool. At any point in time, at most one action for each Agent is being executed. Actions dispatched to an agent from another single agent or thread will occur in the order they were sent, potentially interleaved with actions dispatched to the same agent from other sources. send should be used for actions that are CPU limited, while send-off is appropriate for actions that may block on IO.

Agents are integrated with the STM - any dispatches made in a transaction are held until it commits, and are discarded if it is retried or aborted. No user-code locking is involved.

Note that use of Agents starts a pool of non-daemon background threads that will prevent shutdown of the JVM. Use shutdown-agents to terminate these threads and allow shutdown.

Chapter 11. Java Interop

> Sitting in my favorite coffeehouse with a new notebook and a hot cup of java is my idea of Heaven.
>
> — Libba Bray

11.1. Clojure syntax for Java constructors

Constructing a Java object is done by appending a period to the class identifier:

```
(ns training.core
  (:import [java.util Date]))
```

```
(Date.)
(Date. 2018 02 17)
```

Which is equivalent to the less used variant:

```
(new Date)
(new Date 2018 02 17)
```

11.2. Calling methods

Calling a method on a Java object done by prepending a leading period:

```
(.length "hello world")
(.isDirectory (java.io.File. "my-dir"))
```

Which is equivalent to the less used variant:

```
(. "hello world" length)
(. (java.io.File. "my-dir") isDirectory)
```

Java static method calls are accessed by slash:

```
(Math/pow 1 2)
(.print System/out "hi")
```

Inner classes can be accessed using the dollar symbol:

```
java.nio.channels.FileChannel$MapMode/READ_ONLY
```

11.3. reify

`reify` creates an object that conforms to an interface:

```
(.listFiles (java.io.File. ".")
  (reify
    java.io.FileFilter
    (accept [this f]
      (.isDirectory f))))
```

Notice that we didn't define a class? We directly created an object that conforms to the `FileFilter` interface. `reify` is a convenient way to provide a concrete implementation of an interface.

11.4. gen-class and proxy

`gen-class` creates a class. In practice the need to create a class from within Clojure is rare, so we won't be covering the syntax. (see https://kotka.de/blog/2010/02/gen-class_how_it_works_and_how_to_use_it.html if you want to explore this further)

`proxy` can be used to extend a concrete superclass. Again the need for this is rare. (see https://kotka.de/blog/2010/03/proxy_gen-class_little_brother.html if you want to explore this further)

11.5. Including Java classes in Clojure projects

You can define Java classes in Java in a separate directory and add

```
:java-source-paths ["src/java"]
```

To your `project.clj` file (See https://github.com/technomancy/leiningen/blob/master/doc/MIXED_PROJECTS.md for more other options.)

Chapter 12. Challenge 3: Mocking parallel web requests

Insuricorp and Megacorp are integrating their IT systems. As part of this effort you need to modify the “Corgi cover” eligibility logic to call a remote web service. Your task is to set up the code and tests.

12.1. Part 1: Mock a web request

Every Insuricorp “Corgi cover” policy application needs to be cross referenced with Megacorp to see if the customer has a Megacorp policy already via a remote web service. The web service is not available for you to test against yet. Set up a function called fetch-megacorp-policies to do the web request but leave the implementation empty. Create a test that changes the behavior of fetch-megacorp-policies to behave as though it were a web request; make it pause for 100ms before returning the policies that the person has. Set up a test that exercises the eligibility checks using the mocked version of a web request.

12.2. Part 2: Report the how long it takes

In Java you might write something like this:

```
long startTime = System.nanoTime();
// ... the code being measured ...
long estimatedTime = System.nanoTime() - startTime;
```

Implement a similar solution in Clojure.

12.3. Part 3: Make parallel requests

The web service you are using can handle multiple requests faster than a series of requests. It operates fastest with up to 20 connections. Modify your code such that multiple requests are made simultaneously. Compare the timing results to confirm the operations are happening in parallel.

12.4. Part 4: Error handling

Modify your mock of fetch-megacorp-policies such that it throws an exception randomly about 10% of the time. Make sure your tests report a failure. Now update your logic to handle the errors and retry up to 10 times. The tests should pass. Then create another test where the exception is thrown 100% of the time, and the max tries occurs.

Chapter 13. Polymorphism and Types

13.1. Multimethods

Polymorphic dispatch. First we define the name of the multimethod, and the dispatch function:

```
(defmulti encounter
  (fn dispatch [x y]
    [(:species x) (:species y)]))
```

In this case the dispatch function returns a vector pair of the species of input `x` and the species of input `y`. Now we can provide methods implementing functions to execute for a given dispatch value:

```
(defmethod encounter [:bunny :lion] [x y] :run-away)
(defmethod encounter [:lion :bunny] [x y] :eat)
(defmethod encounter [:lion :lion] [x y] :fight)
(defmethod encounter [:bunny :bunny] [x y] :mate)
```

These are somewhere between a case statement and a function definition. They give the conditions under which to be called, and a function definition. Given a dispatch result of `[:bunny :lion]`, the first method will be called on the `x` and `y` inputs, and the method here does nothing but return a value `:run-away`. Let's set up some test inputs:

```
(def bunny1 {:species :bunny, :other :stuff})
(def bunny2 {:species :bunny, :other :stuff})
(def lion1 {:species :lion, :other :stuff})
(def lion2 {:species :lion, :other :stuff})
```

Now we can call encounter on the data to see what it does…

```
(encounter bunny1 bunny2)
=> :mate
(encounter bunny1 lion1)
=> :run-away
(encounter lion1 bunny1)
=> :eat
(encounter lion1 lion2)
=> :fight
```

Because keywords are functions, it's quite common to use a keyword as a dispatch function.

```
(defmulti draw :shape)
```

13.2. Protocols

A protocol is a named set of named methods and their signatures, defined using defprotocol:

```
(defprotocol AProtocol
  "A doc string for AProtocol abstraction"
  (bar [a b] "bar docs")
  (baz [a] [a b] [a b c] "baz docs"))
```

No implementations are provided. Docs can be specified for the protocol and the functions. The above yields a set of polymorphic functions and a protocol object. All are namespace-qualified by the namespace enclosing the definition.

The resulting functions dispatch on the type of their first argument, and thus must have at least one argument. defprotocol is dynamic, and does not require AOT compilation. defprotocol will automatically generate a corresponding interface, with the same name as the protocol, e.g. given a protocol my.ns/Protocol, an interface my.ns.Protocol. The interface will have methods corresponding to the protocol functions, and the protocol will automatically work with instances of the interface.

Note that you do not need to use this interface with deftype, defrecord, or reify, as they support protocols directly:

```
(defprotocol P
  (foo [x])
  (bar-me [x] [x y]))
```

```
(deftype Foo [a b c]
  P
  (foo [x] a)
  (bar-me [x] b)
  (bar-me [x y] (+ c y)))
```

```
(bar-me (Foo. 1 2 3) 42)
=> 45
```

```
(foo
 (let [x 42]
   (reify P
     (foo [this] 17)
     (bar-me [this] x)
     (bar-me [this y] x))))
=> 17
```

A Java client looking to participate in the protocol can do so most efficiently by implementing the protocol-generated interface. External implementations of the protocol (which are needed when you want a class or type not in your control to participate in the protocol) can be provided using the extend construct:

```
(extend AType
  AProtocol
   {:foo an-existing-fn
    :bar (fn [a b] ...)
    :baz (fn ([a]...) ([a b] ...)...)}
  BProtocol
    {...}
...)
```

`extend` takes a type/class (or interface, see below), a one or more protocol + function map (evaluated) pairs. Will extend the polymorphism of the protocol's methods to call the supplied functions when an AType is provided as the first argument. Function maps are maps of the keywordized method names to ordinary fns. This facilitates easy reuse of existing fns and maps, for code reuse/mixins without derivation or composition.

You can implement a protocol on an interface. This is primarily to facilitate interop with the host (e.g. Java) but opens the door to incidental multiple inheritance of implementation since a class can inherit from more than one interface, both of which implement the protocol. If one interface is derived from the other, the more derived is used, else which one is used is unspecified.

The implementing `fn` can presume first argument is instanceof `AType`. You can implement a protocol on `nil`. To define a default implementation of protocol (for other than `nil`) just use `Object`. Protocols are fully reified and support reflective capabilities via `extends?`, `extenders`, and `satisfies?`. Note the convenience macros `extend-type`, and `extend-protocol`.

If you are providing external definitions inline, these will be more convenient than using extend directly

```
(extend-type MyType
  Countable
    (cnt [c] ...)
  Foo
    (bar [x y] ...)
    (baz ([x] ...) ([x y zs] ...)))
```

Expands into:

```
(extend MyType
  Countable
   {:cnt (fn [c] ...)}
  Foo
   {:baz (fn ([x] ...) ([x y zs] ...))
    :bar (fn [x y] ...)})
```

13.3. Creating types with defrecord and deftype

`deftype`, `defrecord`, and `reify` provide the mechanism for defining implementations of abstractions, and instances of those implementations. Resist the urge to use them to define ‘structured data’ as you would define classes or structures in other languages. It is preferred to use the built-in datatypes (vectors, maps, sets) to represent structured data.

13.3.1. Deftype

```
(deftype Circle [radius])
(deftype Square [length width])
```

```
(Circle. 10)
(Square. 5 11)
```

```
(->Circle 10)
(->Square 5 11)
```

13.3.2. Defrecord

This example shows how to implement a Java interface in defrecord.

```
(import java.net.FileNameMap)
```

To define a record named `Thing` with a single field `a`, implement `FileNameMap` interface and provide an implementation for the single method: `String getContentTypeFor(String fileName)`.

```
(defrecord Thing [a]
  FileNameMap
  (getContentTypeFor [this fileName] (str a "-" fileName)))
```

Construct an instance of the record:

```
(def thing (Thing. "foo"))
```

Check that the instance implements the interface:

```
(instance? FileNameMap thing)
```

Call the method on the `thing` instance and pass `"bar"`:

```
(.getContentTypeFor thing "bar")
```

13.4. Specifications with clojure.spec

The spec library specifies the structure of data, validates or destructures it, and can generate data based on the spec. Spec was introduced into Clojure 1.9.0, so update your `project.clj` to the right version:

```
[org.clojure/clojure "1.9.0"]
```

To start working with spec, require the `clojure.spec.alpha` namespace at the REPL:

```
(ns my.ns
  (:require [clojure.spec.alpha :as s]))
```

13.5. Validation

Any function that takes a single argument and returns a truthy value is a valid predicate spec.

```
(s/valid? even? 10)
=> true
```

```
(s/valid? string? 0)
=> false
```

Sets are functions, so can be used as predicates that match one or more literal values:

```
(s/valid? #{:club :diamond :heart :spade} :club)
=> true
```

Specs are registered using `s/def`.

```
(s/def ::suit #{:club :diamond :heart :spade})
```

A registered spec identifier can be used in place of a spec definition.

```
(s/valid? ::suit :club)
=> true
```

The simplest way to compose specs is with `and` and `or`. Let's create a spec that combines several predicates into a composite spec with `s/and`:

```
(s/def ::big-even (s/and int? even? #(> % 1000)))
```

```
(s/valid? ::big-even 10)
=> false
```

```
(s/valid? ::big-even 100000)
=> true
```

13.6. Conforming

We can also use `s/or` to specify two alternatives:

```
(s/def ::name-or-id (s/or :name string? :id int?))
```

This `or` spec is the first case we've seen that involves a choice during validity checking. Each choice is annotated with a tag (here, between `:name` and `:id`) and those tags give the branches names that can be used to understand or enrich the data returned from conform and other spec functions.

```
(s/conform ::name-or-id "abc")
=> [:name "abc"]
```

```
(s/conform ::name-or-id 100)
=> [:id 100]
```

Many predicates that check an instance's type do not allow nil as a valid value (`string?`, `number?`, `keyword?`, etc). To include `nil` as a valid value, use the provided function nilable to make a spec:

```
(s/nilable string?)
```

Explain can be used to report why a value does not conform to a spec.

```
(s/explain ::big-even 5)
=> val: 5 fails spec: ::big-even predicate: even?
```

In addition to `explain`, you can use `explain-str` to receive the error messages as a string or `explain-data` to receive the errors as data.

13.7. Maps

Clojure programs rely heavily on passing around maps of data. Entity maps in spec are defined with keys:

```
(def email-regex
  #"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,63}$")
(s/def ::email-type (s/and string? #(re-matches email-regex %)))
(s/def ::acctid int?)
(s/def ::first-name string?)
(s/def ::last-name string?)
(s/def ::email ::email-type)
```

```
(s/def ::person (s/keys :req [::first-name ::last-name ::email]
                        :opt [::phone]))
```

Validation checks that the required attributes are included, and that every registered key has a conforming value.

```
(s/valid? ::person
  {::first-name "Elon"
   ::last-name "Musk"
   ::email "elon@example.com"})
=> true
```

Much existing Clojure code does not use maps with namespaced keys and so keys can also specify `:req-un` and `:opt-un` for required and optional unqualified keys. These variants specify namespaced keys used to find their specification, but the map only checks for the unqualified version of the keys.

```
(s/def :unq/person
  (s/keys :req-un [::first-name ::last-name ::email]
          :opt-un [::phone]))
```

```
(s/valid? :unq/person
  {:first-name "Elon"
   :last-name "Musk"
   :email "elon@example.com"})
=> true
```

In addition to the support for information maps via keys, spec also provides map-of for maps with homogenous key and value predicates.

```
(s/def ::scores (s/map-of string? int?))
(s/valid? ::scores {"Sally" 1000, "Joe" 500})
=> true
```

Spec has explicit support for pre and post conditions using `fdef`.

```
(defn adder [x] #(+ x %))
(s/fdef adder
  :args (s/cat :x number?)
  :ret (s/fspec :args (s/cat :y number?)
                :ret number?)
  :fn #(= (-> % :args :x) ((:ret %) 0)))
```

The `:ret` spec uses fspec to declare that the returning function takes and returns a number. Even more interesting, the `:fn` spec can state a general property that relates the `:args` (where we know `x`) and the result we get from invoking the function returned from adder, namely that adding `0` to it should return `x`.

13.8. A game of cards

Here's a bigger set of specs to model a game of cards:

```
(def suit? #{:club :diamond :heart :spade})
(def rank? (into #{:jack :queen :king :ace} (range 2 11)))
(def deck (for [suit suit? rank rank?] [rank suit]))
```

```
(s/def ::card (s/tuple rank? suit?))
(s/def ::hand (s/* ::card))
```

```
(s/def ::name string?)
(s/def ::score int?)
(s/def ::player (s/keys :req [::name ::score ::hand]))
```

```
(s/def ::players (s/* ::player))
(s/def ::deck (s/* ::card))
(s/def ::game (s/keys :req [::players ::deck]))
```

```
(def kenny
  {::name "Kenny Rogers"
   ::score 100
   ::hand []})
(s/valid? ::player kenny)
=> true
```

Bad data produces errors

```
(s/explain ::game
  {::deck deck
   ::players [{::name "Kenny Rogers"
               ::score 100
               ::hand [[2 :banana]]}]})
=> In: [::players 0 ::hand 0 1]
   val: :banana fails spec: ::card
   at: [::players ::hand 1]
   predicate: suit?
```

If we have a function deal that doles out some cards to the players we can spec that function to verify the arg and return value are both suitable data values. We can also specify a :fn spec to verify that the count of cards in the game before the deal equals the count of cards after the deal.

```
(defn total-cards [{:keys [::deck ::players] :as game}]
  (apply + (count deck)
    (map #(-> % ::hand count) players)))
```

```
(defn deal [game] ...)
```

```
(s/fdef deal
  :args (s/cat :game ::game)
  :ret ::game
  :fn #(= (total-cards (-> % :args :game))
          (total-cards (-> % :ret))))
```

13.9. Generators

A key design constraint of spec is that all specs are also designed to act as generators of sample data that conforms to the spec (a critical requirement for property-based testing).

Spec generators rely on the Clojure property testing library test.check. However, this dependency is dynamically loaded and you can use the parts of spec other than gen, exercise, and testing without declaring test.check as a runtime dependency. When we wish to use these parts of spec (typically during testing), we need to declare a dev dependency on `test.check` in our `project.clj`:

```
:profiles {:dev {:dependencies [[org.clojure/test.check "0.9.0"]]}}
```

The dev profile dependencies are included during testing but not published as a dependency or included in uber jars.

We require `clojure.spec.gen.alpha` in the `ns` form:

```
(ns my-ns.my-test
  (:require [clojure.spec.gen.alpha :as gen]))
```

The `gen` function can be used to obtain the generator for any spec.

Once you have obtained a generator with `gen`, there are several ways to use it. You can generate a single sample value with generate or a series of samples with sample. Let’s see some basic examples:

```
(gen/generate (s/gen int?))
=> -959
```

```
(gen/sample (s/gen string?))
=> ("" "" "" "" "8" "W" "" "G74SmCm" "K9sL9" "82vC")
```

```
(gen/sample (s/gen #{:club :diamond :heart :spade}))
=> (:heart :diamond :heart :heart :heart :diamond :spade :spade :spade :club)
```

What about generating a random player in our card game?

```
(gen/generate (s/gen ::player))
=> {:spec.examples.guide/name "sAt8r6t",
    :spec.examples.guide/score 233843,
    :spec.examples.guide/hand ([8 :spade] [5 :heart] [9 :club] [3 :heart])}
```

We can even generate an entire game:

```
(gen/generate (s/gen ::game))
```

It's useful to spec (and generate) values in a range. For example, in the case of a range of integer values, use `int-in` to spec a range:

```
(s/def ::roll (s/int-in 0 11))
(gen/sample (s/gen ::roll))
=> (1 0 0 3 1 7 10 1 5 0)
```

Spec also includes `inst-in` for a range of Dates, and `double-in` for double ranges.

To learn more about generators, read the test.check tutorial https://clojure.github.io/test.check/intro.html.

13.10. Instrumentation and Testing

Spec provides a set of development and testing functionality in the clojure.spec.test.alpha namespace, which we can include with:

```
(ns my-ns.core
  (:require [clojure.spec.test.alpha :as stest]))
```

Instrumentation validates that the :args spec is being invoked on instrumented functions and thus provides validation for external uses of a function.

```
(defn ranged-rand
  "Returns random int in range start <= rand < end"
  [start end]
  (+ start (long (rand (- end start)))))
```

```
(stest/instrument `ranged-rand)
```

Instrument takes a fully-qualified symbol so we use `` ` `` here to resolve it in the context of the current namespace. If the function is invoked with args that do not conform with the `:args` spec you will see an error like this:

```
(ranged-rand 8 5)
=> CompilerException clojure.lang.ExceptionInfo: Call to #'spec.examples.guide/ranged-
rand did not conform to spec
```

Instrumentation can be turned off using the complementary function unstrument. Instrumentation is useful at both development time and during testing to discover errors in calling code. It is not recommended to use instrumentation in production due to the overhead involved with checking args specs.

We mentioned earlier that `clojure.spec.test.alpha` provides tools for automatically testing functions. When functions have specs, we can use check, to automatically generate tests that check the function using the specs.

check will generate arguments based on the :args spec for a function, invoke the function, and check that the :ret and :fn specs were satisfied.

```
(ns my-ns.core
  (:require [clojure.spec.test.alpha :as stest]))
```

```
(stest/check `ranged-rand)
=> ({:spec #object[clojure.spec.alpha$fspec_impl ...],
     :clojure.spec.test.check/ret {:result true, :num-tests 1000, :seed
1466805740290},
     :sym spec.examples.guide/ranged-rand,
     :result true})
```

A keen observer will notice that ranged-rand contains a subtle bug. If the difference between start and end is very large (larger than is representable by Long/MAX_VALUE), then ranged-rand will produce an IntegerOverflowException. If you run check several times you will eventually cause this case to occur.

check also takes a number of options that can be passed to test.check to influence the test run, as well as the option to override generators for parts of the spec, by either name or path.

Imagine instead that we made an error in the ranged-rand code and swapped start and end:

(defn ranged-rand

BROKEN! "Returns random int in range start <= rand < end" [start end] (+ start (long (rand (- start end)))))

This broken function will still create random integers, just not in the expected range. Our :fn spec will detect the problem when checking the var:

```
(stest/abbrev-result (first (stest/check `ranged-rand)))
=> ({...
     :result {...
              :clojure.spec.alpha/failure :test-failed}}
```

check has reported an error in the :fn spec. We can see the arguments passed were -3 and 0 and the return value was -5, which is out of the expected range.

To test all of the spec'ed functions in a namespace (or multiple namespaces), use enumerate-namespace to generate the set of symbols naming vars in the namespace:

```
(-> (stest/enumerate-namespace 'user) stest/check)
```

And you can check all of the spec'ed functions by calling `stest/check` without any arguments.

While both instrument (for enabling `:args` checking) and check (for generating tests of a function) are useful tools, they can be combined to provide even deeper levels of test coverage.

instrument takes a number of options for changing the behavior of instrumented functions, including support for swapping in alternate (narrower) specs, stubbing functions (by using the `:ret` spec to generate results), or replacing functions with an alternate implementation.

Consider the case where we have a low-level function that invokes a remote service and a higher-level function that calls it.

```
(defn invoke-service [service request])
```

```
(defn run-query [service query]
  (let [{::keys [result error]} (invoke-service service
                                  {::query query})]
    (or result error)))
```

We can spec these functions using the following specs:

```
(s/def ::query string?)
(s/def ::request (s/keys :req [::query]))
(s/def ::result (s/coll-of string? :gen-max 3))
(s/def ::error int?)
(s/def ::response (s/or :ok (s/keys :req [::result])
                        :err (s/keys :req [::error])))
```

```
(s/fdef invoke-service
  :args (s/cat :service any? :request ::request)
  :ret ::response)
```

```
(s/fdef run-query
  :args (s/cat :service any? :query string?)
  :ret (s/or :ok ::result :err ::error))
```

And then we want to test the behavior of run-query while stubbing out invoke-service with instrument so that the remote service is not invoked:

```
(stest/instrument `invoke-service {:stub #{`invoke-service}})
=> [spec.examples.guide/invoke-service]
```

```
(invoke-service nil {::query "test"})
=> #:spec.examples.guide{:error -11}
```

```
(invoke-service nil {::query "test"})
=> #:spec.examples.guide{:result ["kq0H4yv08pLl4QkVH8"
"in6gH64gI0ARefv3k9Z5Fi23720gc"]}
```

```
(stest/summarize-results (stest/check `run-query))
=> {:total 1, :check-passed 1}
```

The first call here instruments and stubs `invoke-service`. The second and third calls demonstrate that calls to `invoke-service` now return generated results (rather than hitting a service). Finally, we can use check on the higher level function to test that it behaves properly based on the generated stub results returned from `invoke-service`.

There is even more to spec! Once you are comfortable with the basics you can learn more at https://clojure.org/guides/spec.

Chapter 14. Interacting with a Database

> You can have data without information, but you cannot have information without data.
>
> — Daniel Keys Moran

14.1. Intro to clojure.java.jdbc

Database persistence is important for many applications. We can use clojure.java.jdbc to interact with a database.

To start, create a new project

```
$ lein new messenger
```

and add dependencies to your `project.clj` file:

```
[org.clojure/java.jdbc "0.7.5"]
[hsqldb/hsqldb "1.8.0.10"]
```

Note that we need the driver we plan to use to connect to a database. In this case we are using an in memory HSQL database.

In the Clojure project we require jdbc, and set up a db connection url.

```
(ns messenger.core
  (:require [clojure.java.jdbc :as jdbc]))
```

```
(def db "jdbc:hsqldb:mem:testdb")
```

Now we are all set to start doing queries.

14.2. Inserting, updating and retrieving data

First we will create a table called testing inside the database with a text field named data, and then insert some rows.

```
(jdbc/execute! db
  "create table messages (message varchar(1024))")
```

```
(jdbc/insert-multi! db :messages
                    [{:message "Hello World"}
                     {:message "How now?"}])
```

And we can query the data back:

```
(jdbc/query db ["select * from messages"])
=> ({:message "Hello World"}
    {:message "How now?"})
```

To selectively delete some data:

```
(jdbc/delete! db :messages ["message like '%World%'"])
```

And now there is only one row remaining.

```
(jdbc/query db ["select * from messages"])
=> ({:message "Hello World"})
```

Let’s add some more data…

```
(jdbc/insert-multi! db :messages
                [{:message "Nobody panic!!!"}
                 {:message "What in the world?"}
                 {:message "All is well."}])
```

And now we create a function to do a parameterized query.

```
(defn search [s]
  (jdbc/query db
    ["select * from messages where message like ?" s]))
```

```
(search "%How%")
=> ({:message "How now?"})
```

It is important to use parameterized queries instead of string concatenation in this example because it protects us from SQL injection. Parameters are not part of the query, so they cannot perform SQL from malicious input.

If you want to redo any steps, remember that you can always drop the table and start again.

```
(jdbc/execute! db "drop table messages")
```

14.3. Solutions for SQL management

HoneySQL https://github.com/jkk/honeysql can be used to build SQL statements from data structures. This is useful when you have to programmatically combine clauses to produce a final SQL statement. For example if the user can check a checkbox to enable an additional clause in a search. In such cases it is more convenient to use Clojure's capabilities for manipulating data structures. However if you do not need to do such manipulation, I recommend using plain old SQL queries in their original text form, as you can run them interactively from an SQL prompt much easier that way.

14.4. Exercises

- Create and populate a table `person` with two columns; `id`, `name`.
- Create and populate a table `policy` with two columns; `id`, `name`
- Create and populate a table `person_policy` with two columns; `person_id`, `policy_id`
- Write a function that given a person name queries all the policies associated with them.

14.5. Answers

```
(ns messenger.core
  (:require [clojure.java.jdbc :as jdbc]))
```

```
(def db "jdbc:hsqldb:mem:testdb")
```

```
(jdbc/execute! db
  "create table person (id bigint, name varchar(1024))")
(jdbc/execute! db
  "create table policy (id bigint, name varchar(1024))")
(jdbc/execute! db
  "create table person_policy
  (person_id bigint, policy_id bigint)")
(jdbc/insert-multi! db :person
                    [{:id 1 :name "Sally"}
                     {:id 2 :name "Billy"}])
(jdbc/insert-multi! db :policy
                    [{:id 1 :name "Corgi Cover"}
                     {:id 2 :name "Poodle Protection"}])
(jdbc/insert-multi! db :person_policy
                    [{:person_id 1 :policy_id 1}
                     {:person_id 1 :policy_id 2}
                     {:person_id 2 :policy_id 1}])
```

```
(defn find-policies [person-name]
  (jdbc/query db ["select a.name
                  from policy a
                  inner join person_policy b
                  on a.id = b.policy_id
                  inner join person c
                  on b.person_id = c.id
                  where c.name = ?"
                  person-name]))
```

```
(find-policies "Sally")
=> ({:name "Corgi Cover"} {:name "Poodle Protection"})
(find-policies "Jane")
=> ()
(find-policies "Billy")
=> ({:name "Corgi Cover"})
```

Chapter 15. Challenge 4: Corgi Cover Database

Sending files around is proving to be problematic. Sometimes applications are lost or the results of the eligibility check are not communicated back to the customer. You have been tasked with creating a central source of truth that can be queried as to what applications have been submitted and processed.

15.1. Part 1: Set up the schema

Using the database of your choice, set up an initial database for the Corgi Cover project. In the code, connect to the database and create the initial table required. You can use whatever schema you like, but the first requirement is to store the applications with exactly the same data as was retrieved from the file format in Challenge 2.

15.2. Part 2: Populate the data

Modify the code to store the applications as they are processed, and the result of the eligibility check.

15.3. Part 3: Write a spec

Ensure that all records processed from the files meets your expectations for required fields. Write a spec that explicitly defines what should be in the applications. Validate the spec on the incoming records.

15.4. Part 4: Extending to Poodle Protection

Insuricorp is about to launch a new policy called “Poodle Protection”. Soon they will be processing applications with completely new rules. Set up a multimethod to handle “Poodle Protection” applications differently from “Corgi Cover” applications. For now the only difference with the rules from “Corgi Cover” is that “Poodle Protection” is available in different states: California (CA), Florida (FL), Wyoming (WY), and Hawaii (HI).

Chapter 16. Further reading

Writing Clojure code requires more thinking and less typing than other languages. Don't feel frustrated if the code comes slowly at first. Being a great programmer requires thinking. You will only reach your true potential expressing code in ways that empower you rather than constrain you.

Further exercises: https://www.4clojure.com/

Clojure for Java Programmers - Rich Hickey

- Part 1: https://www.youtube.com/watch?v=P76Vbsk_3J0
- Part 2: https://www.youtube.com/watch?v=hb3rurFxrZ8

Hadoop libraries

- https://github.com/nathanmarz/cascalog
- https://github.com/damballa/parkour
- https://github.com/r0man/hdfs-clj

Spark libraries

- https://github.com/yieldbot/flambo