Thinking in Clojure for Java Programmers (Part 1 — A Gentle Intro)

December 14th, 2010 by Aaron Crow

Tagged with clojure

We do a lot of Java work at Factual. Most of our back-end data store is written in Java, and we use a ton of Java libraries and frameworks. We recently began experimenting with Clojure in certain parts of our data store. Clojure is a Lisp implementation that runs on the JVM and offers excellent Java interoperability.

Because our data store already had a custom Lisp-like query engine, using Clojure was a natural experiment. In certain situations we found it to be more expressive and more productive than what we'd have to do in Java to get the same work done.

But why should anyone be interested in a Lisp? Aren't there too many parenthesis? And isn't it just… weird?

Some consider Lisps to be the most powerful programming languages created so far. You don't have to agree, but one thing is hard to deny: Lisps in general, and Clojure specifically, offer some interesting ideas and compelling features. Working with a Lisp will inspire you to think differently about programming.

This post kicks off an ongoing series of discussions about Clojure. We'll look at Clojure from the perspective of a Java developer who may not be very familiar with Lisp, but would like to be more productive and is open to thinking about programming in a new way.

Clojure uses "prefix notation"… but don't be afraid!

As a Java programmer, one of the things that might immediately weird you out about Clojure is that it uses "prefix notation", meaning the first thing you see in a line of code is the name of the function being called. So, instead of…

```
myObj.doSomething(anArg);
```

… you'll see…

```
(do-something my-obj an-arg)
```

A little more formally, the syntax is like:

```
(fn-name arg1 arg2 arg3 ...)
```

The Clojure way to print out a String shouldn't look strange to you…

```
(println "Hello Clojure!")
```

But maybe the way to add two numbers in Clojure will seem odd to you:

```
(+ 2 3)
```

Lisps, including Clojure, hold consistently to this prefix notation. This might seem strange at first, and it may take some getting used to. But believe it or not, there is some power and beauty in this way of doing things. As a small example, consider doing a logical AND across 5 variables. In Java:

```
... (boolA && boolB && boolC && boolD && boolE) ...
```

In Clojure:

```
... (and boolA boolB boolC boolD boolE) ...
```

By putting functions at the head, the language can support multiple numbers of arguments in a cleaner fashion. Obviously, this example is not earth-shattering. However, at least you can see with a very simple example how prefix notation can make for cleaner code.

Clojure is functional

Lisps do a great job of allowing you to treat functions as first class objects. This means you can easily do useful things with functions, like passing them into other functions, returning them from other functions, and composing them together. Let's first look at simple function definition syntax in Clojure, then move on to treating functions as first class objects…

Defining named functions

Here's an example of a simple function definition in Clojure:

```
(defn do-something []
  (println "hello!"))
```

The above function takes no arguments. Here is a function that takes 3 arguments and prints out each one in succession:

```
(defn do-something [arg1 arg2 arg3]
  (println arg1)
  (println arg2)
  (println arg3))
```

When you run a function, the returned value of that function is the last thing evaluated. So in the above example, the return value of `do-something` will be the returned value of the third call to `println`. The `println` function prints its argument to standard out, then returns `nil`. Therefore, the normal return of `do-something` will always be nil. However, we could easily upgrade `do-something` to return something else if we wanted to, such as a String:

```
(defn do-something [arg1 arg2 arg3]
  (println arg1)
  (println arg2)
  (println arg3)
  "my returned String")
```

Defining anonymous functions

Many programming languages support the concept of anonymous functions (also called "closures" or "code blocks" in certain situations). These can be convenient and make our code more concise. If we're only going to use the function once, for example, there's no need to give it a name… just define it in the same place where you use it, don't name it, and therefore it's "anonymous".

In Java, we have a kind of "poor man's" version of anonymous functions and closures, where we use anonymous inner classes.

As an example, imagine you have a collection of ugly Strings and you need to transform them into pretty Strings. You could write the imperative code to do this "by hand", but Google's Guava library offers an alternative.

There's a `transform` method in their `Collections2` class that takes a Collection and a Function. It returns a new Collection where each item in the new Collection is the result of applying the Function to every item in the original Collection. So the Function allows you to define exactly how you want to transform each item in the Collection.

You could create a new Java class that implements `Function` and pass that to `transform`. But why bother creating a new named class, if you're only going to use it once? So instead, you can define the Function anonymously like this:

```
import com.google.common.base.Function;
...
  // an anonymous function to take an
  // ugly String and make it pretty:
  new Function<String, String>(){
    @Override
    public String apply(String ugly) {
      return ugly.replaceAll("ugly", "pretty");
    }
  }...
```

Imagine that the ellipses (…) represent the surrounding code that passes this anonymous function into `Collections2.transform()`. Since the Function implementation is defined and used exactly once, there's no real need to give it a name. (We'll look at the complete transformation implementation a little later.)

Clojure has excellent support for anonymous functions and closures. Here's an example of creating an anonymous function in Clojure, which takes one argument and prints it out:

```
(fn [an-arg] (println "my arg is" an-arg))
```

Here's the Clojure equivalent of the previous Java code to prettify Strings:

```
...
  (fn [ugly] (re-gsub #"ugly" "pretty" ugly))...
```

re-gsub takes a regular expression, a replacement, and a String. It replaces all matches in the String with the specified replacement — effectively doing what **replaceAll** did for us in the Java code. Note that the regular expression is defined as `#"ugly"` — Clojures's way of compiling a regex.

Clojure also has a less verbose version of defining anonymous functions that doesn't require named arguments:

```
#(re-gsub #"ugly" "pretty" %)
```

The above is equivalent to the previous version using the explicit "`(fn [...`". The difference here is that we didn't need to give the `ugly` argument a name. Instead, we can use the `%` notation, which means "the one and only argument".

This may look cryptic at first, but don't worry. It's easy enough to break this apart and see exactly what's happening. First off, the `#(...)` syntax is Clojure shorthand for defining an anonymous function. The code contained in the "…" section is the implementation of the anonymous function. In this case, our implementation just applies **re-gsub** to the incoming argument. We define our regex, which in this case is just anything that is "ugly". Our replacement is "pretty", and then we apply that to the one and only incoming argument, signified with the `%`.

What would you do if you want an anonymous function that takes multiple arguments, but you don't want to name the arguments? Clojure lets you use the convenient `%` notation along with argument indices, like this:

```
#(println "my three args: " %1 %2 %3)
```

The above anonymous function expects 3 arguments, and prints them out on one line.

Consider that doing this with Guava's Function interface would be somewhat awkward — you'd need your function implementation to take some kind of Collection, array, or other first class object that grouped the arguments. Then your function would need to unpack those arguments explicitly.

The ability to treat functions as first class objects in this way offers power and flexibility. One huge win is the ability to pass functions into other functions, otherwise known as "composability". Let's take a look at three common functions provided by Clojure that work very well when composed with other functions…

Three Lispy workhorses: map, reduce, and filter

One area where Lisps are particularly strong is when you need to work with collections of things. This is good, because as programmers we are **constantly** working with collections of things – iterating over them, transforming them, summarizing them to a single result, and so on.

Three workhorses that Lisps use to deal with collections of things are the functions **map**, **reduce**, and **filter**. (You might recognize "map" and "reduce" from Google's seminal work on distributed batch processing, and you'd be right to make the connection. Their "map" and "reduce" idiom is a nod to these classic Lispy workhorses that have been with us for decades.)

Introducing map

It's very common to have a collection of things, and need to transform it into a collection of different things. Clojure's `map` function applies a given function to each element in a collection and returns a new collection with the results:

```
(map f coll)
```

The first argument **f** to map is the function we want applied to each element in the original collection. The second argument is our original collection that we want transformed. (Note: the map function is **not** related to hash maps or dictionaries!)

A simple example using map:

```
; Get the squares of some integers...
(map #(* % %) [0 1 2 3 4 5 6])
=> (0 1 4 9 16 25 36)
```

Let's break down this example, working from the inside out. Our second and last argument to map is the collection we want to transform — a handful of integers.

The first argument to map is the function we want applied across our collection of integers. In this case that's an anonymous function that takes a value and produces the square of that value. So we end up with a new collection: the squares of the items from the original collection.

Imperative pseudo-code to illustrate what `map` is doing:

```
// inputs are f and coll
new_coll = new Collection
for (each item i in coll) {
  new_coll.add( f(i) )
}
return new_coll
```

Transforming a collection: Java vs. Clojure

Say you're given a list of ugly Strings and you need to turn it into a list of pretty Strings. In Java, assuming you're a fan of Google's nifty Guava utilities for collections, you could do this:

```
private Collection prettyStrings(Collection uglyStrings) {
  return Collections2.transform(uglyStrings, new Function<String, String>(){
    @Override
    public String apply(String ugly) {
      return ugly.replaceAll("ugly", "pretty");
    }});
}
```

Using what we've learned so far, here's how you could do the same thing in Clojure:

```
(defn pretty-strings [ugly-strs]
  (map #(re-gsub #"ugly" "pretty" %) ugly-strs))
```

Introducing reduce

We often need to take a collection and summarize it down to just one result. For example, suppose you have a collection of numbers and you want to get the grand total of all the numbers. In Java we might do this:

```
private long mySum(Collection<Integer> nums) {
  long sum = 0;
  for(int num : nums) {
    sum += num;
  }
  return sum;
}
```

In Clojure, we'd use the `reduce` function, and do this:

```
(defn my-sum [nums] (reduce + nums))
```

And actually, since the Clojure code to do the summing is so short, you probably wouldn't even define it as a named function. Instead, when you need to do it, you'd just write:

```
(reduce + nums)
```

That's so concise, it seems like cheating. In a way it *is* cheating. Lispy, functional languages like Clojure natively support this very common concept. This often means that common tasks can be done very concisely in Clojure, compared to an imperative language like Java.

There are two versions of `reduce`; the simple version used above is like this:

```
(reduce f coll)
```

This simple version of `reduce` applies a function **f** to an accumulator value and each element of the collection **coll**. The function **f** is expected to take 2 arguments. `reduce` supplies these arguments on each iteration: it plugs in the accumulator ("the running total") as the first argument, and it plugs in the next item in the iteration as the second argument. The accumulator starts with the first value of the collection, and the iteration starts on the next item.

A more flexible version of `reduce` allows you to specify the initial accumulator value, **val**:

```
(reduce f val coll)
```

The accumulator value could be anything you design it to be, such as a number, another collection, or any arbitrary data structure.

Another way to understand `reduce` is to consider this imperative pseudo code:

```
iterator = coll.iterator()
if (val is specified?) {
  accum = val
} else {
  accum = iterator.next()
}

while(iterator.hasNext()) {
  accum = f(accum, iterator.next())
}

return accum
```

In the Clojure summation example, our function **f** is simply the + function. (Note that the + function can naturally take two arguments.) We didn't make use of the optional **val** argument to `reduce`. However, it's worthwhile to note that the optional **val** adds important flexibility. It can be very handy to supply an explicit custom starting value, e.g. a pre-populated collection or an empty hash-map.

A simple histogram example using reduce

Let's use `reduce` to build a word histogram. We'll define a function called `counts` that takes a collection of words and builds a hash-map where each key is a unique word, and each value is the count of how many times that word appears in the collection:

```
(defn counts [words]
  (reduce
    (fn [map-so-far word]
      (assoc map-so-far word (inc (get map-so-far word 0))))
    {}
    words))
```

Our call to **reduce** runs across **words**, and it's initial accumulator is an empty hash-map (the {}). The core work of our function is our first argument to **reduce**. It's this anonymous function…

```
(fn [map-so-far word]
  (assoc map-so-far word (inc (get map-so-far word 0))))
```

…which contains the logic to populate the hash-map. As `reduce` runs over a collection and passes in arguments to **f**, the first argument is the "running total" (in our case, **`map-so-far`**), and the second argument is the next item in the collection (in our case, **`word`**). **assoc** is the Clojure way to take a hash-map and assign a value to a key in that hash-map (like Java's `Map.put()`). So `(assoc map-so-far word ...)` is saying, "in the current hash-map of counts, associate this word with X". In our case, X is obtained by the call to **inc**.

The **inc** call is what implements the word counting logic. Here, **get** is like Java's **Map.get**, except you can supply an optional argument to indicate what to return should the requested key not exist. So our call to **get** says, "return the associated value of this word in the current hash-map, or 0 if the word isn't in the hash-map. Our **inc** call simply increments the value returned from **get**. Stringed together, we get the exact logic we need to keep running word counts.

Now, if we wanted to we could shorten this up by using the `%` notation to avoid needing to name our arguments to the anonymous function:

```
(defn counts [words]
  (reduce
    #(assoc %1 %2 (inc (get %1 %2 0)))
    {}
    words))
```

I double-dog dare you to write the Java code that does this, and compare.

Introducing filter

We don't always want to use every item in a collection. Sometimes we wish to skip certain items that don't meet our criteria. Clojure's `filter` function helps with this.

```
(filter pred coll)
```

The first argument to `filter` is **pred**, which must be a function that takes one argument. **pred** is expected to have logic that answers the question, "should this item be kept?". The second and last argument to `filter` is the original collection you want to filter based on **pred**. For example, say you have some numbers but you only want the positive ones. Here's some Clojure for that:

```
(filter #(> % 0) [8 -3 4 2 -1 -9 -12 7])
```

The first argument to `filter`, our **pred**, is an anonymous function that asks, "is this item **%** larger than zero?". The result from running this filter is exactly what we'd expect:

```
=> (8 4 2 7)
```

Now, in the case of finding positive numbers, Clojure has the `pos?` function built-in. So we could have also done:

```
(filter pos? [8 -3 4 2 -1 -9 -12 7])
```

And of course you're free to use custom predicates as well, and define whatever arbitrary filter logic you want:

```
(defn my-pred? [num]
  (> num 6))
```

Here we've defined a custom predicate `my-pred` which only returns true when its argument is greater than 6. So:

```
(filter my-pred? [8 -3 4 2 -1 -9 -12 7])
=> (8 7)
```

Thinking in Clojure with functional composition

When you have a collection of things and need to turn that into a collection of different things, consider using the `map` function. The above examples are somewhat simple, but bear in mind that you can make the function you pass into `map` be as complex as you need it to be.

Clojure offers powerful functions to customize your iteration over the collection, like skipping elements based

on arbitrary filter logic; taking only each nth element; partitioning the elements into groups; and so on.

When you have a collection of things and need to calculate a single result from that collection, consider using the `reduce` function. Remember that you can make the function you pass into `reduce` be as complex as you need it to be.

When you have a collection but only need some of the items, `filter` is probably the first thing you need to apply to the collection before doing other things.

With `map`, `reduce`, and `filter`, you can begin to see some of the power of function composition. Each of these three functions are designed to take another function as an argument and use it in their work. The passed-in function could be a built-in Clojure function, a function provided by someone else's library, or a custom function you've written yourself.

In Java, we tend to think imperatively. We like to design our code in terms of setting up variables, looping through collections to yank out items, changing the state of our variables, and so on. In a Lispy language like Clojure, we want to think more functionally. That is, we want to minimize side-effects and strive for the expressiveness of cleanly applying functions to things.

Imagine we want to add up the squares of all the odd numbers in a collection. In Java, we might do something like this:

```
long sum = 0;
for(int i : coll) {
  if(i % 2 != 0) {
    sum += i * i;
  }
}
```

In Clojure, we can compose three of our favourite Lispy workhorses to do this a little more expressively:

```
(reduce + (map #(* % %) (filter odd? coll)))
```

The best way to read the Java code is probably top to bottom. To best read the Clojure code, try working from the inside out. The first thing we do is get only the odd numbers in **coll**, by applying the call to `filter`. (Note that Clojure has the `odd?` function built in.) Next we use `map` to transform our collection of odd numbers into a new collection of the squares of those odd numbers. Finally, we run a `reduce` over the squares, using + so that our final result is their summation.

Hopefully you can see it's a different way of thinking! To write the Clojure code, we didn't think in terms of, "how do I define a variable and then mutate that variable?" Instead, we asked, "which functions can we compose together, apply to our collection and get to where we're going?"

Lispy languages like Clojure heavily leverage functional composition. `map`, `reduce`, and `filter` are just the tip of the iceberg. Clojure has a wealth of powerful and versatile functions.

Stay tuned…

This post aimed to gently introduce you to some basic Clojure concepts, starting from a Java perspective. In future posts we'll further explore the power and flexibility offered by Clojure. Please stay tuned…!

http://twitter.com/jneira Javier Neira
Great article, to change "from left to right" you can use a _macro_ (other powerful but double edged weapon of lisp) which makes a thrush combination. You last example would be:

user> (->> [1 2 3] (filter odd?) (map #(* % %)) (reduce +))
10

http://twitter.com/tchklovski tchklovski
very nice post, thanks!
if anyone's interested, here are what some of the examples would look like in ruby (1.9)
(the second lines are ruby). Overall, I'd say it's close, and both are gloriously more clear than
equivalent Java code. Particularly nice, to me, are Clojure's % and %1.
also, this example in particular highlights the cleanliness of lisp/clojure (esp w/ clean array syntax)
clojure: (reduce + [1,2,3])
ruby: [1,2,3].inject{|sum,v| sum + v}

Anyhow, here are the examples:

(map #(* % %) [0 1 2 3 4 5 6])
[0 1 2 3 4 5 6].map{|v| v*v }

(reduce + [1,2,3])
[1,2,3].inject{|sum,v| sum + v}

#"ugly"

/ugly/

(defn counts [words]
(reduce #(assoc %1 %2 (inc (get %1 %2 0))) {} words))

def counts(words)
words.inject(Hash.new(0)){|h, w| h[w]+=1; h}
end

(filter #(> % 0) [8 -3 4 2 -1 -9 -12 7])
[8 -3 4 2 -1 -9 -12 7].find_all{|v| v > 0}
[8 -3 4 2 -1 -9 -12 7].reject{|v| v 0 }

(defn my-pred? [num] (> num 6))
my-pred?=->(num){num>6}

(the above is a ruby1.9ism)

(reduce + (map #(* % %) (filter odd? coll)))
coll.find_all(&:odd?).map{|x| x*x }.inject{|sum,v| sum+v }

awmross
“I double-dog dare you to write the Java code that does this, and compare.”

Multiset counts(List words){
Multiset result=HashMultiset.create();
for(String word:words) result.add(word);
return result;
}

vs

(defn counts [words]
(reduce
#(assoc %1 %2 (inc (get %1 %2 0)))
{}
words))

http://pulse.yahoo.com/_UJPBYO4U32MQEVRKKVEYU6IS4Y Aaron
Hi awmross,

Cool. That may be as clean as you’re going to get, w/o a custom data structure.

One thing to note: your function doesn’t end up with the full result that the Clojure version does. That is, after running your function, I *still* need to call the count() method on the result, for each key I’m interested in. Whereas the Clojure version already has the counts as the values in the hash map. But, arguably that’s a finer point.

Something else I noticed when testing your code: I had to do this:
List words = Lists.newArrayList(“one”, “two”, “two”, “three”);

Whereas, in Clojure, I’d just do this:
(def words ["one" "two" "two" "three"])

But again, a finer point, and not part of the dare. Thanks for your post!

Alan
As of Clojure 1.2, there’s the ‘frequencies’ function that does exactly that. The implementation uses transients:
https://github.com/clojure/clojure/blob/b578c69d7480f621841ebcafdfa98e33fcb765f6/src/clj/clojure/core.clj#L5566

Fuse
Isn’t

(reduce f val coll)

equivalent to

(reduce f (cons val coll))

?

José Luis ROmero
Hey thanks a lot for the article!! Learning clojure here and this post really really helped me a lot..

http://www.thefreshuk.com/blog James W. Dunne
#(re-gsub #”ugly” “pretty” %)

This line in particular stuck out for me, in a very, very good way. I cannot possibly come near to describing the feelings of amazement, excitement and whatever else was involved when reading it. It's just so beautiful. If cleanliness is godliness, Clojure must be heavenly.

Although I have toyed around with Scheme and CL before, I have never gone far enough to be able to use it but the examples provided in this article have pretty much demolished any procrastination I was experiencing before.

San
It's easier to think in clojure than to *work in clojure*, I suppose.

Let's recall that Java as a language is nothing special, but infrastructure built on top of it is large. I don't mean libraries, i mean IDE with its autocompletion, code navigation and stuff (also: UI designers etc).

Please explain this to us, java folks, how do you 1) safely rename function globally 2) navigate to function or symbol or find usages 3) change data type (remove field in a record or rename) and see all places that it affected.

Dynamic languages people seem to cover 100% of code with test cases, just to work around loss of type system. In java, you need to cover only important logic, with different goals. Why does clojure force us to write extra tests?

All this is said not to start a flame war, because these facts are known for ages, I just want honest answer:

How come you can trade strong type system (java) for (good) language features? Have you written medium to large project on it already? How it goes? How many devs on it? Why do you advertise clojure, if not?

I see that clojure may be good for expressing some pure logic, more elegant way, but in no way the whole backend or large framework can be written in it). Please comment.

(P.S. I develop in haskell and java for living, so functional paradigm is not new for me).

Tero Salomaa
Excellent !!! First time I found something which really explains some of the power of Clojure!

Thanks a lot.

Tero

Miki Tebeka
Hi San,

Clojure is a young language and IDEs for it are still catching up. Having said that – I've been working with Python for more than 10 years and never found the need for all the IDE features people can't live without on Java. This probably stems from the fact that codes in dynamic languages tend to be much shorter than the equivalent one on Java/C++ …

Code navigation is not a problem as well for me, VIm/Emacs/Eclipse … all have good support for code navigation for many languages (including Clojure).

As for "Static vs Dynamic", we are not going to decide it here 😃
I highly recommend you read Bruce Eckel's "Strong Typing vs. Strong Testing" (https://docs.google.com/View?id=dcsvntt2_25wpjvbbhk).

If find that dynamic languages work better for me, the code tend to be much shorter than it's Java/C++ equivalent (and I've written plenty of C++ and Java code). It's hard to compare project sizes since as said the code tend to be much shorter. Having written many things in Python – linkers, hardware simulators, news systems, trading systems, web sites … Collaboration was never a problem and the dynamic nature of the language helped for code reuse. My (short) experience with Clojure so far have been the same.

As I see it, a language is a tool and you need to pick the best tool *for you* to solve the problem. If you are more productive in strongly typed languages, maybe some other JVM base language (Scala?) will be better for you.

All the best,
–
Miki