



Patrik Fredriksson is a consultant and mentor in software development at Citerus. Patrik is especially interested in design and architecture in Java, and in productivity in development projects.

patrik.fredriksson@citerus.se

From Java to Clojure

Patrik Fredriksson takes us on a tour from Java to Clojure, by using a step by step conversion from one language to the other. If you're a Java programmer who wants to know a little bit about Clojure, make sure you come along.

Earlier this year my colleague Peter Backlund showed us how to convert a piece of Java code to Groovy (in Swedish). In this article we will convert that same Java code, but this time we will convert it to clojure, one of the new languages developed to run on the JVM. Clojure is a general-purpose, functional language that is dynamically and strongly typed. If you have come in contact with Lisp, you will notice that Clojure is a Lisp dialect leveraging the code-as-data philosophy. It is said to be homioconic.. Clojure's emphasis on side-effect-free-functions and immutability together with explicit support for concurrent programming makes it an attractive language for scaling out code onto multiple cores.

Don't miss the online follow-up for this article at:

<http://weakreference.blogspot.com/2009/10/from-java-to-clojure-followup.html>

I will not spend more time on introducing Clojure and its potential here, instead I suggest you visit the Clojure website (<http://clojure.org/>) when you are ready to learn more. But now, let's start exploring Clojure and its Java interoperability features immediately by looking at some code. This code is written for Clojure 1.0.

The Java method we are about to convert to Clojure takes a list of character strings as input and returns a new list of the unique strings, sorted first by frequency and then, for entries with the same frequency, alphabetically. Our JUnit test looks like this:

```
public class OrdererTestUsingJava extends TestCase {

    private void doTestOrderByFreq(Orderer orderer) {
        List unordered = Arrays.asList("b", "d", "b", "b", "a", "c", "a");
        List ordered = orderer.orderByFreq(unordered);
        assertEquals(Arrays.asList("b","a","c","d"), ordered);
    }
}
```

```
public void testJavaImpl() {
    doTestOrderByFreq(new JavaOrderer());
}
}
```

...and the complete Java implementation, like this:

```
package pnehm;

import org.apache.commons.collections.Bag;
import org.apache.commons.collections.bag.HashBag;

import java.util.List;
import java.util.Comparator;
import java.util.Collections;
import java.util.ArrayList;

public class JavaOrderer implements Orderer {

    public List orderByFreq(List in) {
        final Bag bag = new HashBag(in);
        List out = new ArrayList(bag.uniqueSet());

        Collections.sort(out, new Comparator() {
            public int compare(String a, String b) {
                int freq = Integer.valueOf(bag.getCount(b))
                    .compareTo(bag.getCount(a));
                return freq != 0 ? freq : a.compareTo(b);
            }
        });
        return out;
    }
}
```

As you can see, the Java class implements an `Orderer` interface, this makes it easy for us to set up tests that can run multiple implementations. The `Orderer` interface looks like this:

```
package pnehm;
import java.util.List;
public interface Orderer {
    List orderByFreq(List in);
}
```

The first task on our way to an all-Clojure implementation is to create a Clojure script file that can be called from our Java tests. We create the file `clojure_orderer.clj` in the directory `pnehm`, and add the following code:

```
(ns step1.pnehm.clojure-orderer
  (:gen-class
   :name step1.pnehm.ClojureOrderer
   :implements [pnehm.Orderer]
  ))
```

The first form is a macro that sets the namespace, which pretty much is the Clojure version of Java's packages; `pnehm.clojure_orderer` corresponds to the newly created file `pnehm/clojure_orderer.clj`. Here we also use `:gen-class` to tell Clojure to generate a Java class that we can call from Java. The generated class will have the name `pnehm.ClojureOrderer` and implement the `pnehm.Orderer` interface. Going forward you may find it handy to reference the **Clojure API** (<http://www.clojure.org/api>).

With the first script file ready, it's now time to implement the interface. By default the generated class will look for a Clojure function with the same name as the interface method, prefixed by '-'. The prefix can be changed by supplying the option `:prefix` to `:gen-class`, but we will stick with the default for now.

Our first iteration will be a direct port of the Java version. In other words, we will use the same Java classes, but call them from Clojure. This may seem like a weird thing to do; if we want to write Java, why take on extra work by doing it from Clojure? Well, we have to start somewhere, we have the Java version and we know that it works, and it gives us a nice way of exploring how Java code can be called from Clojure.

So, our next step will be to import the Java classes that we need. `java.lang` is already imported for us, just as in Java, but if we want more stuff we have to explicitly specify it. Let's add the imports we need to our `ns` macro call:

```
(ns step1.pnehm.clojure-orderer
  (:gen-class
   :name step1.pnehm.ClojureOrderer
   :implements [pnehm.Orderer]
  )
  (:import [org.apache.commons.collections Bag]
           [org.apache.commons.collections.bag HashBag]
           [java.util List]
           [java.util Comparator]
           [java.util Collections]
           [java.util ArrayList]))
```

Now we can add our function definition that implements the `Orderer.orderByFreq(List in)` method:

```
(defn -orderByFreq [_ arg]
  ;function's body goes here
```

)

defn is a macro that is used to define named functions. The name of the function is `-orderByFreq`, to match our Java interface. The leading hyphen is the default way of mapping a function to Java. Functions that implement instance methods called by Java takes first a special parameter that references the Java object on which the method is called, known to Java developers as *this*. After that come the Java method parameters. The underscore inside the brackets tells Clojure that we expect that first parameter to be there, but we don't care much for it. The parameters we *do* care about, our list of character strings, are bound to the name `arg`.

With the function definition taken care of, all that remains is to write the function's body, and we're done. This is what the body looks like:

```
(defn -orderByFreq [_ arg]

  (let [bag (HashBag. arg)
        out (ArrayList. (.uniqueSet bag))
        cmpr (proxy [Comparator] []
                 (compare [a b]
                          (let [freq (.compareTo (.getCount bag b) (.getCount bag a))]
                            (if-not (zero? freq) freq (.compareTo a b))))))]

    (do
      (Collections/sort out, cmpr)
      out)))
```

This code is quite different from what we might be accustomed to as Java developers. It takes some time to get used to, but it really isn't that scary. If you feel the need for an introduction to Clojure before we proceed, sneak a peek at my Clojure Crash Course for Java Developers. (<http://weakreference.blogspot.com/2009/09/clojure-crash-course-for-java.html>)

First we use the *let* macro to bind a new `HashBag` instance to the name `bag`, and the `ArrayList` of unique character strings to the name `out`, this corresponds to the first two lines of our Java implementation. `(HashBag.)` is Clojure for `new HashBag()`, And `(ArrayList. (.uniqueSet bag))` calls the method `uniqueSet()` on `bag`, and passes the result into the `ArrayList` constructor.

The third `let` binding is `cmpr`. `cmpr` is our `java.util.Comparator` implementation, here generated as a Dynamic Proxy by Clojure. Dynamic Proxies are commonly used in Clojure to generate Java code "on the fly", which we use here to implement the Java interface `java.util.Compare` in Clojure. `(compare [a b]...)` is our Clojure implementation of `Comparator.compare(T o1, T o2)`.

if is a Clojure special form for branching. If the test evaluates to 'true', then the first expression is evaluated and returned, otherwise the second expression is evaluated and returned.

So, with our `ArrayList` set up as `out` and the `Comparator` in place as `cmpr`, we can use the `java.util.Collections.sort()` method to sort our list according to `cmpr`. The Clojure special

form *do* evaluates the expressions in order, and returns the result of the last expression. We need to use it here since `Collection.sort()` doesn't return the sorted collection.

And with that, we have a working implementation. The solution, however, doesn't feel very Clojure-ish. Let's continue and see if we can approach some more idiomatic Clojure.

First let us try to get rid of the `HashBag` and `ArrayList`. Functional languages like Clojure typically have very strong support for manipulating lists. In our alternative implementation we create a map with frequency as keys and the character string as value. We can then sort it with a slightly modified version of our comparator. We start by creating a function called `count-words`:

```
(defn count-words [coll]
  (reduce #(merge-with + %1 {%2 1}) {} coll))
```

The function takes a collection as input and calls *reduce* with the collection as input. *reduce* is a Clojure function that in this example takes another function, a value and a collection as parameters. *reduce* applies the given function to the value and the first item of the collection. The result of this is then used together with the second item in the collection, and the function is applied again. This will go on until all items in the collection have been used, i.e. the collection has been reduced, and the result of the final function call is returned.

A simple example with *reduce*, the function `+`, and a vector could be: `(reduce + 0 [1 2 3 4 5])`. The result of this would be 15. In this case the init value 0 doesn't really contribute anything, so we can just skip it, using a different version of *reduce* to get the same result: `(reduce + [1 2 3 4 5])`.

Our supplied init value in the code fragment above is `{}`, an empty map. The function, `#(merge-with + %1 {%2 1})` , is an anonymous function using *merge-with*. *merge-with* merges any number of maps into one map, from left to right. In the case of a key collision, the supplied function is used to calculate a new key using the values of the two original keys. In our anonymous function `%1` refers to the first parameter and `%2` to the second, as supplied by *reduce*.

Let us see what happens when the `count-words` function is called on our test list: `"b", "d", "b", "b", "a", "c", "a"`.

The very first call to *merge-with* will get the empty map and the first item in our collection, `"b"`, as parameters by *reduce*. The empty map is bound to `%1` and the `"b"` is bound to `%2`. The expression `{%2 1}` evaluates to a new map with the `"b"` as key, and 1 as value. *merge-with* will merge these two into a new map: `{"b" 1}`. The second time *merge-with* is called, it will be with our new map, `{"b" 1}`, and the next item from our collection, `"d" 1}`, and create a new map: `{"b" 1, "d" 1}`. The third time, `{"b" 1, "d" 1}` will be merged with `{"b" 1}`, this time a key collision occurs, since we already have a key `"b"`. So, the function supplied to *merge-with* will be used, in our example that function is `+. 1 + 1` is 2 even in Clojure, so what we get is the following map: `{"b" 2, "d" 1}`. This

will go on until all items are processed and we have a complete map with character strings mapped to frequencies. With a slightly modified Comparator, and making sure we only return the keys from our sorted map, we're good to go. We have to replace `Collections/sort` with the Clojure equivalent `sort` since it is able to sort our map:

```
(defn count-words [coll]
  (reduce #(merge-with + %1 {%2 1}) {} coll))

(def cmpr
  (proxy [Comparator] []
    (compare [a b]
      (let [freq (.compareTo (.val b) (.val a))]
        (if-not (zero? freq) freq (.compareTo (.key a) (.key b)))))))

(defn -orderByFreq [_ arg]
  (let [out (count-words arg)]
    (if (empty? out) () (keys (sort cmpr out)))))
```

This is better. But let us clean up the comparator:

```
(defn cmpr [[val1 freq1] [val2 freq2]]
  (let [freq (compare freq2 freq1)]
    (if-not (zero? freq) freq (compare val1 val2))))

(defn -orderByFreq [_ coll]
  (if (empty? coll) () (keys (sort cmpr (count-words coll)))))
```

By using the Clojure equivalents to the Java methods we can drop the Proxy. Since we no longer use any Java code, we can also drop all our imports. The last thing that remains for us to look at before we are completely done, is testing in Clojure.

There is a unit-testing framework in the Clojure user contrib

(<http://www.assembla.com/spaces/dashboard/index/clojure-contrib>) libraries called `test-is`. It is not part of Clojure 1.0 core and must be downloaded and built separately from `clojure-contrib`. In the next release of Clojure the testing framework is moved into core and can then be found in the `clojure.test` namespace. Re-implementing our test with `test-is` looks like this:

```
(ns step3.pnehm.clojure-orderer-test
  (:use step3.pnehm.clojure-orderer clojure.contrib.test-is))

(deftest test-order-by-freq-1
  (is (= ["b", "a", "c", "d"] (-orderByFreq :a ["b", "d", "b", "b", "a", "c", "a"]))))
```

And if we would like to test our Java implementation with `test-is`, that can of course also be done:

```
(ns step5.pnehm.java-orderer-test
```

```
(:use clojure.contrib.test-is)
(:import [pnehm JavaOrderer]))

(deftest test-order-by-freq-1
  (is (= ["b","a","c","d"] (.orderByFreq (JavaOrderer.) ["b", "d", "b", "b", "a", "c",
"a"]))))
```

Now we no longer need the Java bindings and our final, pure Clojure version looks like this:

```
(ns step4.pnehm.clojure-orderer)

(defn count-words [coll]
  (reduce #(merge-with + %1 {%2 1}) {} coll))

(defn cmpr [[val1 freq1] [val2 freq2]]
  (let [freq (compare freq2 freq1)]
    (if-not (zero? freq) freq (compare val1 val2))))

(defn order-by-freq [coll]
  (keys (sort cmpr (count-words coll))))
```

Compared to the Java version, the final version no longer returns an empty list if given an empty list, instead it returns `nil`, which is more idiomatic Clojure.

Rerunning the micro benchmark from Peter Backlund's previous article shows that the Clojure implementation gives the following numbers for sorting 100 characters with 10000 samples on my Core 2 Duo:

- ▶ Java - 111 ms
- ▶ Groovy - 436 ms
- ▶ Clojure - 970 ms

So, the Java implementation is almost nine times as fast as the Clojure implementation, and four times as fast as the Groovy implementation. As always, take these kind of measurements with a grain of salt.

By using functions from Clojure-contrib the performance of the Clojure implementation can be increased to match the Groovy implementation, see followup blog entry (<http://weakreference.blogspot.com/2009/10/from-java-to-clojure-followup.html>).

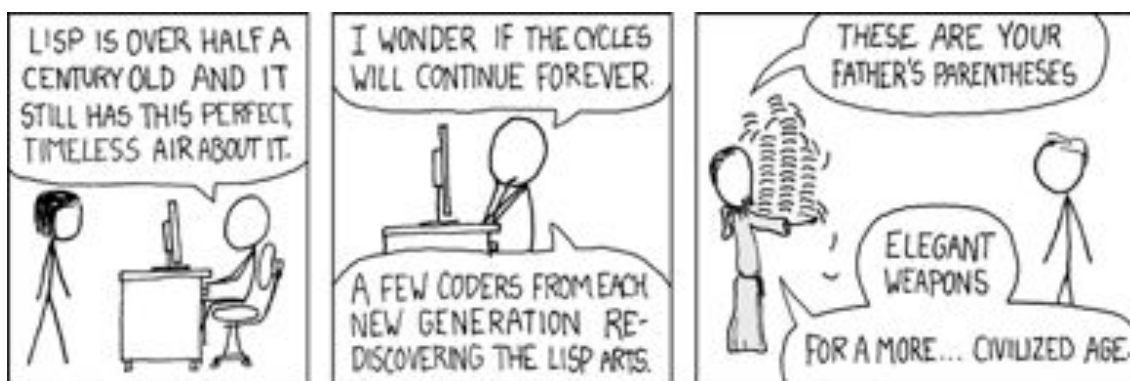
All code used in this article can be found in this Google code project: <http://code.google.com/p/pnehm-java-to-cool-language-x/> Here you will also find the Groovy implementation from the previous article.

Many thanks to the friendly members of the Clojure Google group <http://groups.google.com/group/clojure>

Clojure development is an ongoing effort. If you are curious about what is in store for the future, have a look at <http://clojure.org/todo> and the Clojure space at Assembla <http://www.assembla.com/spaces/clojure/milestones/>

Tool support

A final note about tool support. There is a number of tools out there to support Clojure development. Being an IntelliJ user I typically use IntelliJ and its La Clojure plug-in, but there are many other choices, including plug-ins for NetBeans, Eclipse, and of course Emacs. Clojure being a young language most plug-ins are more or less feature-rich, and most are under heavy development. For interactively working with Clojure from the command prompt, try the Clojure REPL (Read, Evaluate, Print, Loop) provided in the Clojure download.



Fro xkcd.com. Big thanks!

Further reading

- Online follow-up: <http://weakreference.blogspot.com/2009/10/from-java-to-clojure-followup.html>
- Clojure: <http://www.clojure.org/>
- Clojure user contrib <http://www.assembla.com/spaces/dashboard/index/clojure-contrib>
- Clojure Google Group <http://groups.google.com/group/clojure>