

Wednesday, December 28, 2011

Convert java.util.Properties to a Clojure Map

As I previously mentioned, a lot of the work I do involves [Clojure & Java interop](#). This work includes the occasional case of working with a java.util.Properties object from within Clojure. Working with a Properties object isn't a huge deal, but while in Clojure I prefer to use [destructuring](#) and the various functions (e.g. update-in, assoc, dissoc, etc) that are designed to work with Clojure maps.

The following example shows how easy it is to convert a Properties object to a Clojure map.

```
user=> (def prop-obj (doto (java.util.Properties.) (.putAll {"a" 1 "b" 2})))
user=> prop-obj
#<Properties {b=2, a=1}>

user=> (reduce (fn [x [y z]] (assoc x y z)) {} prop-obj)
{"a" 1, "b" 2}
```

That's fairly easy, but you'll quickly want your keys to be keywords if you plan on destructuring. You can drop in a quick call to keyword to convert the keys; however, you'll probably also want to dasherize the keys to allow for easy destructuring using keys and idiomatic names.

```
user=> (defn dash-match [[ _ g1 g2]]
      (str g1 "-" g2))

user=> (defn dasherize [k]
      (-> k
        (clojure.string/replace #"([A-Z]+)([A-Z][a-z])" dash-match)
        (clojure.string/replace #"([a-z\d])([A-Z])" dash-match)
        (clojure.string/lower-case)))#'user/dash-match)

user=> (def prop-obj (doto (java.util.Properties.) (.putAll {"FirstName" "Mike" "LastName"
"Green"})))
user=> (reduce (fn [x [y z]] (assoc x y z)) {} prop-obj)
{"LastName" "Green", "FirstName" "Mike"}
user=> (reduce (fn [x [y z]] (assoc x (-> y dasherize keyword) z)) {} prop-obj)
{:last-name "Green", :first-name "Mike"}
```

That looks good, but you might also find yourself working with a properties file that uses dots or dashes to group similar data. For example, you might find the following entry in your properties file (and the resulting Properties object).

```
person.name=Mike Green
person.age=26
person.sex=male
```

Loading this into a map works fine; however, it would be nice if the resulting map was nested (to keep common data together, and for easier access using get-in, update-in, etc).

The following code splits on dots and nests the values appropriately.

```
user=> (def prop-obj (doto (java.util.Properties.) (.putAll {"person.name" "Mike Green"
"person.age" "26" "person.sex" "male"})))
user=> prop-obj
```

```
#<Properties {person.name=Mike Green, person.age=26, person.sex=male}>
user=> (reduce (fn [x [y z]] (assoc-in x (-> y dasherize (clojure.string/split #"\.")) z))
{} prop-obj)
{"person" {"sex" "male", "age" "26", "name" "Mike Green"}}
```

```
user=> (-> (reduce (fn [x [y z]] (assoc-in x (-> y dasherize (clojure.string/split #"\."))
z)) {} prop-obj) clojure.walk/keywordize-keys)
{:person {:sex "male", :age "26", :name "Mike Green"}}
```


So, not too complicated, but much more complicated than what we started with. It turns out you'll likely want to do other things like parse integers, parse booleans, require keys, and add defaults when a key=value pair isn't specified.

We could go through the effort here of providing all that functionality, but instead I've created a small library that provides all of the above mentioned features: [propertea](#)

If you'd like to see the implementation of the above features, you'll just need to look in `propertea.core`. If you want examples of all of the features of `propertea`, checkout the tests in `github`.

Labels: [clojure](#), [java interop](#)

Share: [Email](#) | [Del.icio.us](#) | [Digg](#) | [Reddit](#) | [Shadow](#) | [Rojo](#)

posted by jaycfields : 9:15 PM [1 comments](#) 
[[\(www\)](#) [\(email\)](#) [\(work\)](#)]

Clojure & Java Interop

About a year ago I got a phone call asking if I wanted to join another team at [DRW](#). The team supports a (primarily) Java application, but the performance requirements would also allow it to be written in a higher level language. I'd been writing Clojure (basically) full-time at that point - so my response was simple: I'd love to join, but I'm going to want to do future development using Clojure.

A year later we still have plenty of Java, but the vast majority of the new code I add is Clojure. One of the big reasons I'm able to use Clojure so freely is the seamless interop with Java.

Execute Clojure from Java

Calling Clojure from Java is as simple as loading the `.clj` file and invoking a method from that file. I used the [same example](#) years ago, but I'll inline it here for simplicity.

```
; interop/core.clj
(ns interop.core)

(defn print-string [arg]
  (println arg))

// Java calling code
RT.loadResourceScript("interop/core.clj");
RT.var("interop.core", "print-string").invoke("hello world");
```

note: examples from this blog entry are available in [this](#) git repo. The commit with the code from the

previous example is available [here](#) and I'm running the example from the command line with:

```
lein jar && java -cp "interop-1.0.0.jar:lib/*" interop.Example
```

Execute Java from Clojure

At this point we have Java executing some Clojure code, and we also have Clojure using an object that was created in Java. Even though we're in Clojure we can easily call methods on any Java object.

```
(ns interop.core)

(defn print-string [arg]
  (println arg "is" (.length arg) "characters long"))
```

[commit](#)

The above code (using the length method of a String instance) produces the following output.

```
hello world is 11 characters long
```

Calling a Java method and passing in additional arguments is also easy in Clojure.

```
(ns interop.core)

(defn print-string [arg]
  (println (.replace arg "hello" "goodbye")))
```

[commit](#)

The above code produces the following output.

```
goodbye world
```

There are a few other things to know about calling Java from Clojure. The following examples show how to call static methods, use enums, and use inner classes.

```
(ns interop.core)

(defn print-string [arg]
  ;;; calling a static method
  (println (String/valueOf true))

  ;;; using an enum
  (println (java.util.concurrent.TimeUnit/SECONDS))

  ;;; using a Java nested (inner) class. Note, in Clojure you
  ;;; use a $ instead of a .
  (println (java.util.AbstractMap$SimpleEntry. "key" "val")))
```

[commit](#)

And, the output:

```
true
#< SECONDS>
#<SimpleEntry key=val>
```

Create Java objects in Clojure

When working with Clojure you'll likely want to interact with existing Java objects, but you'll probably also want to create new instances of Java objects. You might have noticed the dot at the end of `AbstractSimpleEntry` in the previous example - that's how you instruct Clojure to create an instance of a Java object. The following example shows the dot notation for calling a constructor of the `String` class.

```
(ns interop.core)

(defn print-string [arg]
  (println (String. arg)))
```

[commit](#)

At this point our output is back to the original output.

```
hello world
```

When creating Java objects it's often beneficial to know which Java interfaces the Clojure data structures implement. The following examples demonstrate how you can create Java objects while passing Clojure datastructures (and functions) as constructor arguments.

```
(ns interop.core)

(defn print-string [arg]
  ;;; pass a Clojure vector where Java expects a java.util.Collection
  (println (java.util.HashSet. ["1" "2"])))

  ;;; pass a Clojure map where Java expects a java.util.Map
  (println (java.util.LinkedHashMap. {1 "1" 2 "2"})))

  ;;; pass a Clojure function where Java expects a Runnable
  (println (Thread. (fn [] (println "clojure fns are runnables (and callables)")))))
```

[commit](#)

The output shows the constructed Java objects.

```
#<HashSet [2, 1]>
#<LinkedHashMap {1=1, 2=2}>
#<Thread Thread[Thread-1,5,main]>
```

Calling constructors in Clojure is very easy, but that's not always an option when creating a Java object. At times you will likely need to create an instance of a Java interface. Clojure provides both [proxy](#) and [reify](#) for creating instances of Java interfaces. The following example demonstrates the syntax for using either proxy or reify.

```
(ns interop.core)

(defn proxy-coll []
  (proxy [java.util.Collection] []
    (add [o]
      (println o)
      true)))

(defn reify-coll []
```

```

(reify java.util.Collection
  (add [this o]
    (println o)
    (println this)
    true)))

(defn main []
  (.add (proxy-coll) "this string is printed on proxied.add")
  (.add (reify-coll) "this string is printed on reified.add"))

```

[commit](#)

note, I also changed Example.java (the details are available in the above linked commit). The syntax for proxy and reify are fairly similar, and both offer additional options that are worth looking into. The primary differences between these two simple examples are:

- The proxy implementation requires an empty vector where we could specify constructor arguments (if this were an abstract class instead of an interface).
- The arg list for all methods of reify will specify the reified instance as the first argument. In our example the Collection.add method only takes one argument, but in our reify we also get the instance of the collection.

You might have also noticed that both implementations of add have "true" at the end - in our example we're hard-coding the return value of add to always return true. The following output is the result of running the current example code.

```

this string is printed on proxied.add
this string is printed on reified.add
#<core$reify_coll$reify__11 interop.core$reify_coll$reify__11@556917ee>

```

It's worth reading the docs to determine whether you want proxy or reify; however, if you don't see a clear choice I would opt for reify.

Returning objects from Clojure to Java

Our current Example.java returns something from the call to invoke on the clojure.lang.Var that is returned from RT.var("interop.core", "main"), but we're ignoring it so we have no idea what's returned.* Let's change the code and return something on purpose.

```

// interop/Example.java
package interop;

import clojure.lang.RT;

public class Example {
  public static void main(String[] args) throws Exception {
    RT.loadResourceScript("interop/core.clj");
    System.out.println(RT.var("interop.core", "main").invoke());
  }
}

; interop/core.clj
(ns interop.core)

(defn main []
  {:a "1" :b "2"})

```

Running our changes produces the following output.

```
{:a "1", :b "2"}
```

[commit](#)

At this point we are back in Java land after making a quick trip to Clojure to get a value. Returning most objects will be pretty straightforward; however, at some point you may want to return a Clojure function. This turns out to be fairly easy as well, since Clojure functions are instances of the IFn interface. The following code demonstrates how to return a Clojure function and call it from within Java.

```
// interop/Example.java
package interop;

import clojure.lang.RT;

public class Example {
    public static void main(String[] args) throws Exception {
        RT.loadResourceScript("interop/core.clj");
        clojure.lang.IFn f = (clojure.lang.IFn) RT.var("interop.core", "main").invoke();
        f.invoke("hello world");
    }
}

// interop/core.clj
(ns interop.core)

(defn main [] println)
```

[commit](#)

The above example returns the println function from interop.core/main and then invokes the println function from within Java. I only chose to pass one argument to invoke; however, the IFn.invoke method has various overrides to allow you to pass several arguments. The above code works, but it can be simplified to the following example.

```
package interop;

import clojure.lang.RT;

public class Example {
    public static void main(String[] args) throws Exception {
        clojure.lang.IFn f = (clojure.lang.IFn) RT.var("clojure.core", "println");
        f.invoke("hello world");
    }
}
```

[commit](#)


It seems like a fitting end that our final output is the same as our original output.

```
hello world
```

*actually, it's the last thing that's returned, or "true" for this specific case.

Labels: [clojure](#), [clojure functions](#), [functions](#), [java](#), [java interop](#)

Share: [Email](#) | [Delicio.us](#) | [Digg](#) | [Reddit](#) | [Shadow](#) | [Rojo](#)

posted by jaycfields : 6:46 AM [3 comments](#) 
[([www](#)) ([email](#)) ([work](#))]

Saturday, November 19, 2011

Clojure: expectations - scenarios

When I set out to write [expectations](#) I wanted to create a simple unit testing framework. I'm happy with what expectations provides for unit testing; however, I also need to write the occasional test that changes values or causes a side effect. There's no way I could go back to clojure.test after enjoying better failure messages, trimmed stack traces, automatic testing running, etc. Thus, expectations.scenarios was born.

Using expectations.scenarios should be fairly natural if you already use expectations. The following example is a simple scenario (which could be a unit test, but we'll start here for simplicity).

```
(ns example.scenarios
  (:use expectations.scenarios))

(scenario
  (expect nil? nil))
```

A quick trip to the command line shows us that everything is working as expected.

```
Ran 1 tests containing 1 assertions in 4 msec
0 failures, 0 errors.
```

As I said above, you could write this test as a unit test. However, expectations.scenarios was created for the cases in which you want to verify a value, make a change, and verify a value again. The following example shows multiple expectations verifying changing values in the same scenario.

```
(scenario
  (let [a (atom 0)]
    (swap! a inc)
    (expect 1 @a)
    (swap! a inc)
    (expect 2 @a)))
```

In expectations (unit tests) you can only have one expect (or given) so failures are captured, but they do not stop execution. However, due to the procedural nature of scenarios, the first failing expect stops execution.

```
(scenario
  (let [a (atom 0)]
    (swap! a inc)
    (expect 2 @a)
    (println "you'll never see this")))
```

```
failure in (scenarios.clj:4) : example.scenarios
  (expect 2 (clojure.core/deref a))
  expected: 2
  was: 1
  on (scenarios.clj:7)
Ran 1 tests containing 1 assertions in 81 msec
1 failures, 0 errors.
```

expectations.scenarios also allows you to easily verify calls to any function. I generally use interaction expects when I need to verify some type of side effect (e.g. logging or message publishing).

```
(scenario
  (println "1")
  (expect (interaction (println "1"))))
```

It's important to note the ordering of this scenario. You don't 'setup' an expectation and then call the function. Exactly the opposite is true - you call the function the same way you would in production, then you expect the interaction to have occurred. You may find this jarring if you're used to setting up your mocks ahead of time; However, I think this syntax is the least intrusive - and I think you'll prefer it in the long term.

The above example calls println directly, but your tests are much more likely to look something like this.

```
(defn foo [x] (println x))

(scenario
  (foo "1")
  (expect (interaction (println "1"))))
```

Similar to all other mocking frameworks (that I know of) the expect is using an implicit "once" argument. You can also specify :twice and :never if you find yourself needing those interaction tests.

```
(defn foo [x] (println x))

(scenario
  (foo "1")
  (foo "1")
  (expect (interaction (println "1")) :twice)
  (expect (interaction (identity 1)) :never))
```

On occasion you may find yourself interested in verifying 2 out of 3 arguments - expectations.scenarios provides the 'anything' var that can be used for arguments you don't care about.


```
(defn foo [x y z] (println x y z))

(scenario
  (foo "1" 2 :a)
  (expect (interaction (println "1" anything :a))))
```

That's about all there is to expectations.scenarios, hopefully it fills the gap for tests you want to write that simply can't be done as unit tests.

Labels: [clojure](#), [expectations](#), [functional testing](#), [testing](#)

Share: [Email](#) | [Del.icio.us](#) | [Digg](#) | [Reddit](#) | [Shadow](#) | [Rojo](#)

posted by jaycfields : 8:24 AM [0 comments](#) 
[([www](#)) ([email](#)) ([work](#))]

Tuesday, November 01, 2011

Clojure: expectations unit testing wrap-up

[Clojure Unit Testing with Expectations Part One](#)

[Clojure Unit Testing with Expectations Part Two](#)

[Clojure Unit Testing with Expectations Part Three](#)

[Clojure Unit Testing with Expectations Part Four](#)

[Clojure Unit Testing with Expectations Part Five](#)

[Clojure Unit Testing with Expectations Part Six \(this entry\)](#)

The previous blog posts on [expectations](#) unit testing syntax cover all of the various ways that expectations can be used to write tests and what you can expect when your tests fail. However, there are a few other things worth knowing about expectations.

Stacktraces

expectations aggressively removes lines from the stacktraces. Just like many other aspects of expectations, the focus is on more signal and less noise. Any line in the stacktrace from clojure.core, clojure.lang, clojure.main, and java.lang will be removed. As a result any line appearing in your stacktrace should be relevant to your application or a third-party lib you're using. expectations also removes any duplicates that can occasionally appear when anonymous functions are part of the stacktrace. Again, it's all about improving signal by removing noise. Speaking of noise...

Test Names

You might have noticed that expectations does not require you to create a test name. This is a reflection of my personal opinion that [test names are nothing more than comments](#) and shouldn't be required. If you desire test names, feel free to drop a comment above each test. Truthfully, this is probably a better solution anyway, since you can use spaces (instead of dashes) to separate words in a comment. Comments are good when used properly, but they can become noise when they are required. The decision to simply use comments for test names is another example of improving signal by removing noise.

Running Focused Expectations

Sometimes you'll have a file full of expectations, but you only want to run a specific expectation - expectations solves this problem by giving you 'expect-focused'. If you use expect-focused only expectations that are defined using expect-focused will be run.

For example, if you have the following expectations in a file you should see the following results from 'lein expectations'.

```
(ns sample.test.core
  (:use [expectations]))
(expect zero? 0)
(expect zero? 1)
```

```
(expect-focused nil? nil)
jfields$ lein expectations
Ran 1 tests containing 1 assertions in 2 msec
IGNORED 2 EXPECTATIONS
0 failures, 0 errors.
```

As you can see, expectations only ran one test - the expect-focused on line 6. If the other tests had been run the test on line 5 would have created a failure. It can be easy to accidentally leave a few expect-focused calls in, so expectations prints the number of ignored expectations in capital letters as a reminder. Focused expectation running is yet another way to remove noise while working through a problem.

Tests Running

If you always use 'lein expectations' to run your tests you'll never even care; however, if you ever want to run individual test files it's important to know that your tests run by default on JVM shutdown. When I'm working with Clojure and Java I usually end up using IntelliJ, and therefore have the ability to easily run individual files. When I switched from clojure.test to expectations I wanted to make test running as simple as possible - so I removed the need to specify (run-all-tests). Of course, if you don't want expectations to run for some reason you can disable this feature by calling (expectations/disable-run-on-shutdown).

JUnit Integration

Lack of JUnit integration was a deal breaker for my team in the early days, so expectations comes with an easy way to run all tests as part of JUnit. If you want all of your tests to run in JUnit all you need to do is implement ExpectationsTestRunner.TestSource. The following example is what I use to run all the tests in expectations with JUnit.


```
import expectations.junit.ExpectationsTestRunner;
import org.junit.runner.RunWith;
@RunWith(expectations.junit.ExpectationsTestRunner.class)
public class SuccessTest implements ExpectationsTestRunner.TestSource{
    public String testPath() {
        return "test/clojure/success";
    }
}
```

As you can see from the example above, all you need to do is tell the test runner where to find your Clojure files.

That should be everything you need to know about expectations for unit testing use. If anything is unclear, please drop me a line in the comments.

Labels: [clojure](#), [expectations](#), [testing](#)

Share: [Email](#) | [Del.icio.us](#) | [Digg](#) | [Reddit](#) | [Shadow](#) | [Rojo](#)

posted by jaycfields : 6:59 AM [4 comments](#) 

[[\(www\)](#) [\(email\)](#) [\(work\)](#)]

Clojure: expectations - removing duplication with given

[Clojure Unit Testing with Expectations Part One](#)

[Clojure Unit Testing with Expectations Part Two](#)

[Clojure Unit Testing with Expectations Part Three](#)

[Clojure Unit Testing with Expectations Part Four](#)

[Clojure Unit Testing with Expectations Part Five \(this entry\)](#)

[Clojure Unit Testing with Expectations Part Six](#)

[expectations](#) obviously has a bias towards [one assertion per test](#); however, there are times that verifying several things at the same time does make sense. For example, if you want to verify a few different properties of the same Java object it probably makes sense to make multiple assertions on the same instance.

One of the biggest problems with multiple assertions per test is when your test follows this pattern:

1. create some state
2. verify a bit about the state
3. alter the state
4. verify more about the state

Part of the problem is that the assertions that occurred before the altering of the state may or may not be relevant after the alteration. Additionally, if any of the assertions fail you have to stop running the entire test - thus some of your assertions will not be run (and you'll be lacking some information).

[expectations](#) takes an alternate route - embracing the idea of multiple assertions by providing a specific syntax that allows multiple verifications and the least amount of duplication.

The following example shows how you can test multiple properties of a Java object using the 'given' syntax.

```
(given (java.util.ArrayList.)
      (expect
        .size 0
        .isEmpty true))
jfields$ lein expectations
Ran 2 tests containing 2 assertions in 4 msec
0 failures, 0 errors.
```

The syntax is simple enough: (given an-object (expect method return-value [method return-value]))
note: [method return-value] may be repeated any number of times.

This syntax allows us to expect return-values from as many methods as we care to verify, but encourages us not to change any state between our various assertions. This syntax also allows us to run each assertion regardless of the outcome of any previous assertion.

Obviously you could call methods that change the internal state of the object and at that point you're on your own. I definitely wouldn't recommend testing that way. However, as long as you call methods that don't change any state 'given' can help you write succinct tests that verify as many aspects of an object as you need to test.

As usual, I'll show the output for tests that fail using this syntax.

```
(given (java.util.ArrayList.)
  (expect
    .size 1
    .isEmpty false))
jfields$ lein expectations
failure in (core.clj:4) : sample.test.core
  (expect 1 (.size (java.util.ArrayList.)))
  expected: 1
  was: 0
failure in (core.clj:4) : sample.test.core
  (expect false (.isEmpty (java.util.ArrayList.)))
  expected: false
  was: true
```

This specific syntax was created for testing Java objects, but an interesting side effect is that it actually works on any value and you can substitute method calls with any function. For example, you can test a vector or a map using the examples below as a template.

```
(given [1 2 3]
  (expect
    first 1
    last 3))
(given {:a 2 :b 4}
  (expect
    :a 2
    :b 4))
jfields$ lein expectations
Ran 4 tests containing 4 assertions in 8 msec
0 failures, 0 errors.
```

And, of course, the failures.

```
(given [1 2 3]
  (expect
    first 2
    last 1))
(given {:a 2 :b 4}
  (expect
    :a 1
    :b 1))
jfields$ lein expectations
failure in (core.clj:4) : sample.test.core
  (expect 2 (first [1 2 3]))
  expected: 2
  was: 1
failure in (core.clj:4) : sample.test.core
  (expect 1 (last [1 2 3]))
  expected: 1
  was: 3
failure in (core.clj:9) : sample.test.core
  (expect 1 (:a {:a 2, :b 4}))
  expected: 1
  was: 2
failure in (core.clj:9) : sample.test.core
  (expect 1 (:b {:a 2, :b 4}))
  expected: 1
```

```
was: 4
Ran 4 tests containing 4 assertions in 14 msec
4 failures, 0 errors.
```

When you want to call methods on a Java object or call functions with the same instance over and over the previous given syntax is really the simplest solution. However, there are times where you want something more flexible.

expectations also has a 'given' syntax that allows you to specify a template - thus reducing code duplication. The following example shows a test that verifies + with various arguments.

```
(given [x y] (expect 10 (+ x y))
  4 6
  6 4
  12 -2)
jfields$ lein expectations
Ran 3 tests containing 3 assertions in 5 msec
0 failures, 0 errors.
```

The syntax for this flavor of given is: (given bindings template-form values-to-be-bound). The template form can be anything you need - just remember to put the expect in there.


Here's another example where we combine given with in to test a few different things. This example shows both the successful and failing versions.

```
;; successful
(given [x y] (expect x (in y))
  :a #{:a :b}
  {:a :b} {:a :b :c :d})
;; failure
(given [x y] (expect x (in y))
  :c #{:a :b}
  {:a :d} {:a :b :c :d})
lein expectations
failure in (core.clj:8) : sample.test.core
  (expect :c (in #{:a :b}))
  key :c not found in #{:a :b}
failure in (core.clj:8) : sample.test.core
  (expect {:a :d} (in {:a :b, :c :d}))
  expected: {:a :d}
  in: {:a :b, :c :d}
  :a expected: :d
  was: :b
Ran 4 tests containing 4 assertions in 13 msec
2 failures, 0 errors.
```

That's basically it for 'given' syntax within expectations. There are times that I use all of the various versions of given; however, there seems to be a connection with using given and interacting with Java objects. If you don't find yourself using Java objects very often then you probably won't have a strong need for given.

Labels: [clojure](#), [expectations](#), [testing](#)

Share: [Email](#) | [Delicio.us](#) | [Digg](#) | [Reddit](#) | [Shadow](#) | [Rojo](#)

posted by jaycfields : 6:58 AM [0 comments](#) 
[([www](#)) ([email](#)) ([work](#))]

Clojure: expectations and Double/NaN

[Clojure Unit Testing with Expectations Part One](#)

[Clojure Unit Testing with Expectations Part Two](#)

[Clojure Unit Testing with Expectations Part Three](#)

[Clojure Unit Testing with Expectations Part Four \(this entry\)](#)

[Clojure Unit Testing with Expectations Part Five](#)

[Clojure Unit Testing with Expectations Part Six](#)

I'm not really a fan of Double/NaN in general, but sometimes it seems like the least evil choice. When I find myself in one of those cases I always hate having to write tests in a way that differs from all the other tests in the codebase. A goal I've always had with expectations is to keep the syntax consistent, and as a result I've chosen to treat Double/NaN as equal to Double/NaN when in the various Clojure data structures.

The following examples demonstrate Double/NaN being treated as equal

```
;; allow Double/NaN equality in a map
(expect {:a Double/NaN :b {:c Double/NaN}} {:a Double/NaN :b {:c Double/NaN}})
;; allow Double/NaN equality in a set
(expect #{1 Double/NaN} #{1 Double/NaN})
;; allow Double/NaN equality in a list
(expect [1 Double/NaN] [1 Double/NaN])
jfields$ lein expectations
Ran 3 tests containing 3 assertions in 32 msec
0 failures, 0 errors.
```

As you would expect, you can also count on Double/NaN being considered equal even if you are using the 'in' function.

```
;; allow Double/NaN equality when verifying values are in a map
(expect {:a Double/NaN :b {:c Double/NaN}} (in {:a Double/NaN :b {:c Double/NaN} :d "other stuff"}))
;; allow Double/NaN equality when verifying it is in a set
(expect Double/NaN (in #{1 Double/NaN}))
;; allow Double/NaN equality when verifying it's existence in a list
(expect Double/NaN (in [1 Double/NaN]))
jfields$ lein expectations
Ran 3 tests containing 3 assertions in 32 msec
0 failures, 0 errors.
```

For completeness I'll also show the examples of each of these examples failing.

```
;; allow Double/NaN equality in a map
(expect {:a Double/NaN :b {:c Double/NaN}} {:a nil :b {:c Double/NaN}})
;; allow Double/NaN equality with in fn and map
(expect {:a Double/NaN :b {:c nil}} (in {:a Double/NaN :b {:c Double/NaN} :d "other stuff"}))
;; allow Double/NaN equality in a set
(expect #{1 Double/NaN} #{1 nil})
```

```


;; allow Double/NaN equality with in fn and set
(expect Double/NaN (in #{1 nil}))
;; allow Double/NaN equality in a list
(expect [1 Double/NaN] [1 nil])
;; allow Double/NaN equality with in fn and list
(expect Double/NaN (in [1 nil]))
jfields$ lein expectations
failure in (core.clj:5) : sample.test.core
  (expect {:a Double/NaN, :b {:c Double/NaN}}
    {:a nil, :b {:c Double/NaN}})
  expected: {:a NaN, :b {:c NaN}}
    was: {:a nil, :b {:c NaN}}
  :a expected: NaN
    was: nil
failure in (core.clj:8) : sample.test.core
  (expect {:a Double/NaN, :b {:c nil}} (in {:a Double/NaN, :b {:c Double/NaN}, :d
"other stuff"}))
  expected: {:a NaN, :b {:c nil}}
    in: {:a NaN, :b {:c NaN}, :d "other stuff"}
  :b {:c expected: nil
    was: NaN
failure in (core.clj:11) : sample.test.core
  (expect #{1 Double/NaN} #{nil 1})
  expected: #{NaN 1}
    was: #{nil 1}
  nil are in actual, but not in expected
  NaN are in expected, but not in actual
failure in (core.clj:14) : sample.test.core
  (expect Double/NaN (in #{nil 1}))
  key NaN not found in #{nil 1}
failure in (core.clj:17) : sample.test.core
  (expect [1 Double/NaN] [1 nil])
  expected: [1 NaN]
    was: [1 nil]
  nil are in actual, but not in expected
  NaN are in expected, but not in actual
failure in (core.clj:20) : sample.test.core
  (expect Double/NaN (in [1 nil]))
  value NaN not found in [1 nil]
Ran 6 tests containing 6 assertions in 66 msec
6 failures, 0 errors.

```

There always seems to be downsides to using NaN, so I tend to look for the least painful path. Hopefully expectations provides the most pain-free path when your tests end up needing to include NaN.

Labels: [closure](#), [expectations](#), [testing](#)

Share: [Email](#) | [Delicio.us](#) | [Digg](#) | [Reddit](#) | [Shadow](#) | [Rojo](#)

posted by jaycfields : 6:57 AM [0 comments](#) 
[([www](#)) ([email](#)) ([work](#))]

Clojure: expectations with values in vectors, sets, and maps

[Clojure Unit Testing with Expectations Part One](#)

[Clojure Unit Testing with Expectations Part Two](#)

[Clojure Unit Testing with Expectations Part Three \(this entry\)](#)

[Clojure Unit Testing with Expectations Part Four](#)

[Clojure Unit Testing with Expectations Part Five](#)

[Clojure Unit Testing with Expectations Part Six](#)

I've previously written about verifying equality and the various non-equality [expectations](#) that are available. This entry will focus on another type of comparison that is allowed in expectations - verifying that an 'expected' value is in an 'actual' value.

A quick recap - expectations generally look like this: (expect expected actual)

verifying an expected value is in an actual value is straightforward and hopefully not a surprising syntax: (expect expected (in actual))

If that's not clear, these examples should make the concept completely clear.

```
;; expect a k/v pair in a map.
(expect {:foo 1} (in {:foo 1 :cat 4}))
;; expect a key in a set
(expect :foo (in #{:foo :bar}))
;; expect a val in a list
(expect :foo (in [:foo :bar]))
```

As you would expect, running these expectations results in 3 passing tests.

```
jfields$ lein expectations
Ran 3 tests containing 3 assertions in 8 msecs
0 failures, 0 errors.
```

As usual, I'll show the failures as well.


```
;; expect a k/v pair in a map.
(expect {:foo 2} (in {:foo 1 :cat 4}))
;; expect a key in a set
(expect :baz (in #{:foo :bar}))
;; expect a val in a list
(expect :baz (in [:foo :bar]))
jfields$ lein expectations
failure in (core.clj:18) : sample.test.core
      (expect {:foo 2} (in {:foo 1, :cat 4}))
      expected: {:foo 2}
      in: {:foo 1, :cat 4}
      :foo expected: 2
      was: 1
failure in (core.clj:21) : sample.test.core
      (expect :baz (in #{:foo :bar}))
      key :baz not found in #{:foo :bar}
failure in (core.clj:24) : sample.test.core
      (expect :baz (in [:foo :bar]))
      value :baz not found in [:foo :bar]
```


expectations does it's best to provide you with any additional info that might be helpful. In the case of the vector and the set there's not much else that can be said; however, the map failure gives you additional information that can be used to track down the issue.

There's nothing magical going on with 'in' expectations and you could easily do the equivalent with select-keys, contains?, or some, but expectations allows you to get that behavior while keeping your tests succinct.

Labels: [closure](#), [expectations](#), [testing](#)

Share: [Email](#) | [Del.icio.us](#) | [Digg](#) | [Reddit](#) | [Shadow](#) | [Rojo](#)

posted by jaycfields : 6:56 AM [0 comments](#) 
[([www](#)) ([email](#)) ([work](#))]

Closure: Non-equality expectations

[Closure Unit Testing with Expectations Part One](#)

[Closure Unit Testing with Expectations Part Two \(this entry\)](#)

[Closure Unit Testing with Expectations Part Three](#)

[Closure Unit Testing with Expectations Part Four](#)

[Closure Unit Testing with Expectations Part Five](#)

[Closure Unit Testing with Expectations Part Six](#)

In my last blog post I gave examples of how to use [expectations](#) to test for equality. This entry will focus on non-equality expectations that are also available.

Regex

expectations allows you to specify that you expect a regex, and if the string matches that regex the expectation passes. The following example shows both the successful and failing expectations that use regexes.

```
(expect #"in 14" "in 1400 and 92")
jfields$ lein expectations
Ran 1 tests containing 1 assertions in 4 msec
0 failures, 0 errors.
(expect #"in 14" "in 1300 and 92")
jfields$ lein expectations
failure in (core.clj:17) : sample.test.core
      (expect in 14 in 1300 and 92)
      regex #"in 14" not found in "in 1300 and 92"
Ran 1 tests containing 1 assertions in 5 msec
1 failures, 0 errors.
```

As you can see from the previous example, writing an expectation using a regex is syntactically the same as writing an equality expectation - and this is true for all of the non-equality expectations. In expectations there is only one syntax for expect - it's always (expect expected actual).

Testing for a certain type

I basically never write tests that verify the result of a function is a certain type. However, for the once

in a blue moon case where that's what I need, expectations allows me to verify that the result of a function call is a certain type simply by using that type as the expected value. The example below shows the successful and failing examples of testing that the actual is an instance of the expected type.

```
(expect String "in 1300 and 92")
jfields$ lein expectations
Ran 1 tests containing 1 assertions in 6 msec
0 failures, 0 errors.
(expect Integer "in 1300 and 92")
jfields$ lein expectations
failure in (core.clj:17) : sample.test.core
      (expect Integer in 1300 and 92)
      in 1300 and 92 is not an instance of class java.lang.Integer
Ran 1 tests containing 1 assertions in 5 msec
1 failures, 0 errors.
```

Expected Exceptions

Expected exceptions are another test that I rarely write; however, when I find myself in need - expectations has me covered.

```
(expect ArithmeticException (/ 12 0))
jfields$ lein expectations
Ran 1 tests containing 1 assertions in 6 msec
0 failures, 0 errors.
(expect ClassCastException (/12 0))
jfields$ lein expectations
failure in (core.clj:19) : sample.test.core
      (expect ClassCastException (/ 12 0))
      (/ 12 0) did not throw ClassCastException
Ran 1 tests containing 1 assertions in 4 msec
1 failures, 0 errors.
```


There's another non-equality expectation that I do use fairly often - an expectation where the 'expected' value is a function. The following simple examples demonstrate that if you pass a function as the first argument to expect it will be called with the 'actual' value and it will pass or fail based on what the function returns. ([truthy](#) results pass, [falsey](#) results fail).

```
(expect nil? nil)
(expect true? true)
(expect false? true)
jfields$ lein expectations
failure in (core.clj:19) : sample.test.core
      (expect false? true)
      true is not false?
Ran 3 tests containing 3 assertions in 4 msec
1 failures, 0 errors.
```

These are the majority of the non-equality expectations; however, there is one remaining non-equality expectation - in. Using 'in' is fairly straightforward, but since it has examples for vectors, sets, and maps I felt it deserved it's own blog post - coming soon.

Labels: [clojure](#), [expectations](#), [testing](#)

Share: [Email](#) | [Del.icio.us](#) | [Digg](#) | [Reddit](#) | [Shadow](#) | [Rojo](#)

posted by jaycfields : 6:55 AM [2 comments](#) 
[([www](#)) ([email](#)) ([work](#))]

Clojure: expectations Introduction

[Clojure Unit Testing with Expectations Part One \(this entry\)](#)

[Clojure Unit Testing with Expectations Part Two](#)

[Clojure Unit Testing with Expectations Part Three](#)

[Clojure Unit Testing with Expectations Part Four](#)

[Clojure Unit Testing with Expectations Part Five](#)

[Clojure Unit Testing with Expectations Part Six](#)

A bit of history

Over a year ago [I blogged](#) that I'd written a testing framework for Clojure - [expectations](#). I wrote expectations to test my production code, but made it open source in case anyone else wanted to give it a shot. I've put zero effort into advertising expectations; however, I've been quietly adding features and expanding it's use on my own projects. At this point it's been stable for quite awhile, and I think it's worth looking at if you're currently using clojure.test.

Getting expectations

Setting up expectations is easy if you use [lein](#). In your project you'll want to add:

```
:dev-dependencies [[lein-expectations "0.0.1"]  
                  [expectations "1.1.0"]]
```

After adding both dependencies you can do a "lein deps" and then do a "lein expectations" and you should see the following output.

```
Ran 0 tests containing 0 assertions in 0 msec  
0 failures, 0 errors.
```

At this point, you're ready to start writing your tests using expectations.

Unit Testing using expectations

expectations is build with the idea that unit tests should contain [one assertion per test](#). A result of this design choice is that expectations has very minimal syntax.

For example, if you want to verify the result of a function call, all you need to do is specify what return value you expect from the function call.

```
(expect 2 (+ 1 1))
```

note: you'll want to (:use expectations); however, no other setup is required for using expectations. I created a sample project for this blog post and the entire test file looks like this (at this point):

```
(ns sample.test.core  
  (:use [expectations]))  
(expect 2 (+ 1 1))
```

Again, we use lein to run our expectations.

```
jfields$ lein expectations  
Ran 1 tests containing 1 assertions in 2 msec
```

0 failures, 0 errors.

That's the simplest, and most often used expectation - an equality comparison. The equality comparison works across all Clojure types - vectors, sets, maps, etc and any Java instances that return true when given to Clojure's = function.

```
(ns sample.test.core
  (:use [expectations]))
(expect 2 (+ 1 1))
(expect [1 2] (conj [] 1 2))
(expect #{1 2} (conj #{} 1 2))
(expect {1 2} (assoc {} 1 2))
```

Running the previous expectations produces similar output as before.

```
jfields$ lein expectations
Ran 4 tests containing 4 assertions in 26 msecs
0 failures, 0 errors.
```

Successful equality comparison isn't very exciting; however, expectations really begins to prove it's worth with it's failure messages. When comparing two numbers there's not much additional information that expectations can provide. Therefore, the following output is what you would expect when your expectation fails.

```
(expect 2 (+ 1 3))
jfields$ lein expectations
failure in (core.clj:4) : sample.test.core
  (expect 2 (+ 1 3))
  expected: 2
  was: 4
```

expectations gives you the namespace, file name, and line number along with the expectation you specified, the expected value, and the actual value. Again, nothing surprising. However, when you compare vectors, sets, and maps expectations does a bit of additional work to give you clues on what the problem might be.

The following 3 expectations using vectors will all fail, and expectations provides detailed information on what exactly failed.

```
(expect [1 2] (conj [] 1))
(expect [1 2] (conj [] 2 1))
(expect [1 2] (conj [] 1 3))
jfields$ lein expectations
failure in (core.clj:5) : sample.test.core
  (expect [1 2] (conj [] 1))
  expected: [1 2]
  was: [1]
  2 are in expected, but not in actual
  expected is larger than actual
failure in (core.clj:6) : sample.test.core
  (expect [1 2] (conj [] 2 1))
  expected: [1 2]
  was: [2 1]
  lists appears to contain the same items with different ordering
failure in (core.clj:7) : sample.test.core
  (expect [1 2] (conj [] 1 3))
```

```
    expected: [1 2]
      was: [1 3]
    3 are in actual, but not in expected
    2 are in expected, but not in actual
Ran 3 tests containing 3 assertions in 22 msecs
3 failures, 0 errors.
```

In these simple examples it's easy to see what the issue is; however, when working with larger lists expectations can save you a lot of time by telling you which specific elements in the list are causing the equality to fail.

Failure reporting on sets looks very similar:

```
(expect #{1 2} (conj #{} 1))
(expect #{1 2} (conj #{} 1 3))
jfields$ lein expectations
failure in (core.clj:9) : sample.test.core
  (expect #{1 2} (conj #{} 1))
  expected: #{1 2}
  was: #{1}
  2 are in expected, but not in actual
failure in (core.clj:10) : sample.test.core
  (expect #{1 2} (conj #{} 1 3))
  expected: #{1 2}
  was: #{1 3}
  3 are in actual, but not in expected
  2 are in expected, but not in actual
Ran 2 tests containing 2 assertions in 15 msecs
2 failures, 0 errors.
```

expectations does this type of detailed failure reporting for maps as well, and this might be one of the biggest advantages expectations has over clojure.test - especially when dealing with nested maps.

```
(expect {:one 1 :many {:two 2}}
  (assoc {} :one 2 :many {:three 3}))
jfields$ lein expectations
failure in (core.clj:13) : sample.test.core
  (expect {:one 1, :many {:two 2}} (assoc {} :one 2 :many {:three 3}))
  expected: {:one 1, :many {:two 2}}
  was: {:many {:three 3}, :one 2}
  :many {:three with val 3 is in actual, but not in expected
  :many {:two with val 2 is in expected, but not in actual
  :one expected: 1
  was: 2
Ran 1 tests containing 1 assertions in 19 msecs
1 failures, 0 errors.
```

expectations also provides a bit of additional help when comparing the equality of strings.

```
(expect "in 1400 and 92" "in 1400 and 92")
jfields$ lein expectations
failure in (core.clj:17) : sample.test.core
  (expect in 1400 and 92 in 1400 and 92)
  expected: "in 1400 and 92"
  was: "in 1400 and 92"
  matches: "in 14"
  diverges: "00 and 92"
```


&: "00 and 92"

Ran 1 tests containing 1 assertions in 8 msec
1 failures, 0 errors.

That's basically all you'll need to know for using expectations to equality test. I'll be following up this blog post with more examples of using expectations with regexs, expected exceptions and type checking; however, if you don't want to wait you can take a quick look at the [success tests](#) that are found within the framework.

Labels: [clojure](#), [expectations](#), [testing](#)

Share: [Email](#) | [Del.icio.us](#) | [Digg](#) | [Reddit](#) | [Shadow](#) | [Rojo](#)

posted by jaycfields : 6:54 AM [2 comments](#) 
[([www](#)) ([email](#)) ([work](#))]

Tuesday, August 23, 2011

Clojure: Check For nil In a List

The `every?` function in Clojure is very helpful for determining if every element of a list passes a predicate. From the docs:

Usage: (every? pred coll)

Returns true if (pred x) is logical true for every x in coll, else false.

The usage of `every?` is very straightforward, but a quick REPL session is always nice to verify our assumptions.

```
Clojure 1.2.0  
user=> (every? nil? [nil nil nil])  
true  
user=> (every? nil? [nil 1])  
false
```

As expected, `every?` works well when you know exactly what predicate you need to use.

Yesterday, I was working on some code that included checking for nil - similar to the example below.

```
user=> (def front 1)  
#'user/front  
user=> (def back 2)  
#'user/back  
user=> (when (and front back) [front back])  
[1 2]
```

This code works perfectly if you have the individual elements "front" and "back", but as the code evolved I ended up representing "front" and "back" simply as a list of elements. Changing to a list required a way to verify that each entry in the list was not nil.

I was 99% sure that "and" was a macro; therefore, combining it with apply wasn't an option. A quick REPL reference verified my suspicion.

```
user=> (def legs [front back])
#'user/legs
user=> (when (apply and legs) legs)
java.lang.Exception: Can't take value of a macro: #'clojure.core/and (NO_SOURCE_FILE:8)
```

Several other options came to mind, an anonymous function that checked for (not (nil? %)), map the values to (not (nil? %)) and use every? with true?; however, because of [Clojure's truthiness](#) the [identity](#) function is really all you need. The following REPL session shows how identity works perfectly as our predicate for this example.

```
(when (every? identity legs) legs)
[1 2]
```


For a few more looks at behavior, here's a few examples that include nil and false.

```
user=> (every? identity [1 2 3 4])
true
user=> (every? identity [1 2 nil 4])
false
user=> (every? identity [1 false 4])
false
```

As you can see, using identity will cause every? to fail if any element is falsey (nil or false). In my case the elements are integers or nil, so this works perfectly; however, it's worth noting so you don't see unexpected results if booleans ever end up in your list.

Labels: [clojure](#), [clojure functions](#)

Share: [Email](#) | [Del.icio.us](#) | [Digg](#) | [Reddit](#) | [Shadow](#) | [Rojo](#)

posted by jaycfields : 6:20 AM [5 comments](#) 
[([www](#)) ([email](#)) ([work](#))]

Clojure: partition-by, split-with, group-by, and juxt

Today I ran into a common situation: I needed to split a list into 2 sublists - elements that passed a predicate and elements that failed a predicate. I'm sure I've run into this problem several times, but it's been awhile and I'd forgotten what options were available to me. A quick look at <http://clojure.github.com/clojure/> reveals several potential functions: [partition-by](#), [split-with](#), and [group-by](#).

partition-by

From the docs:

Usage: (partition-by f coll)

Applies f to each value in coll, splitting it each time f returns a new value. Returns a lazy seq of partitions.

Let's assume we have a collection of ints and we want to split them into a list of evens and a list of odds. The following REPL session shows the result of calling `partition-by` with our list of ints.

```
user=> (partition-by even? [1 2 4 3 5 6])
((1) (2 4) (3 5) (6))
```

The `partition-by` function works as described; unfortunately, it's not exactly what I'm looking for. I need a function that returns `((1 3 5) (2 4 6))`.

split-with

From the docs:

Usage: `(split-with pred coll)`

Returns a vector of `[(take-while pred coll) (drop-while pred coll)]`

The `split-with` function sounds promising, but a quick REPL session shows it's not what we're looking for.

```
user=> (split-with even? [1 2 4 3 5 6])
[() (1 2 4 3 5 6)]
```

As the docs state, the collection is split on the first item that fails the predicate - `(even? 1)`.

group-by

From the docs:

Usage: `(group-by f coll)`

Returns a map of the elements of `coll` keyed by the result of `f` on each element. The value at each key will be a vector of the corresponding elements, in the order they appeared in `coll`.

The `group-by` function works, but it gives us a bit more than we're looking for.

```
user=> (group-by even? [1 2 4 3 5 6])
{false [1 3 5], true [2 4 6]}
```

The result as a map isn't exactly what we desire, but using a bit of [destructuring](#) allows us to grab the values we're looking for.

```
user=> (let [{evens true odds false} (group-by even? [1 2 4 3 5 6])]
 [evens odds])
[[2 4 6] [1 3 5]]
```

The `group-by` results mixed with destructuring do the trick, but there's another option.

[juxt](#)

From the docs:

Usage: `(juxt f)`
`(juxt f g)`
`(juxt f g h)`
`(juxt f g h & fs)`

Alpha - name subject to change.

Takes a set of functions and returns a fn that is the juxtaposition of those fns. The returned fn takes a variable number of args, and returns a vector containing the result of applying each fn to the args (left-to-right).

```
((juxt a b c) x) => [(a x) (b x) (c x)]
```


The first time I ran into `juxt` I found it a bit intimidating. I couldn't tell you why, but if you feel the same way - don't feel bad. It turns out, `juxt` is exactly what we're looking for. The following REPL session shows how to combine `juxt` with [filter](#) and [remove](#) to produce the desired results.

```
user=> ((juxt filter remove) even? [1 2 4 3 5 6])
[(2 4 6) (1 3 5)]
```

There's one catch to using `juxt` in this way, the entire list is processed with `filter` and `remove`. In general this is acceptable; however, it's something worth considering when writing performance sensitive code.

Labels: [clojure](#), [clojure functions](#), [maps](#)

Share: [Email](#) | [Del.icio.us](#) | [Digg](#) | [Reddit](#) | [Shadow](#) | [Rojo](#)

posted by jaycfields : 6:07 AM [5 comments](#) 
[([www](#)) ([email](#)) ([work](#))]

Saturday, August 20, 2011

Clojure: Apply a Function To Each Value of a Map

I recently needed to update all of the values of map, and couldn't find an individual function in [clojure.core](#) that did the trick. Luckily, implementing an update-values function is actually very straightforward. The following function reduces the original map to a new map with the same elements, except the values are all the result of applying the specified function and any additional args.

```
(defn update-values [m f & args]
  (reduce (fn [r [k v]] (assoc r k (apply f v args))) {} m))
```

The code is concise, but perhaps a bit terse. Still, it does the trick, as the REPL session below demonstrates.

```
Clojure 1.2.0
user=> (defn update-values [m f & args]
  (reduce (fn [r [k v]] (assoc r k (apply f v args))) {} m))
#'user/update-values
user=> (update-values {:a 1 :b 2 :c 3} inc)
{:c 4, :b 3, :a 2}
user=> (update-values {:a 1 :b 2 :c 3} + 10)
{:c 13, :b 12, :a 11}
user=> (update-values {:a {:z 1} :b {:z 1} :c {:z 1}} dissoc :z)
{:c {}, :b {}, :a {}}
```

The last example is the specific problem I was looking to solve: remove a key-value pair from all the

values of a map. However, the map I was working with actually had a bit of nesting. Let's define an example map that is similar to what I was actually working with.

```
user=> (def data {:boxes {"jay" {:linux 2 :win 1} "mike" {:linux 2 :win 2}}})
#'user/data
```

Easy enough, I have 2 linux boxes and 1 windows box, and Mike has 2 linux and 2 windows. But, now the company decides to discard all of our windows boxes, and we'll need to update each user. A quick combo of update-in with update-values does the trick.


```
user=> (update-in data [:boxes] update-values dissoc :win)
{:boxes {"mike" {:linux 2}, "jay" {:linux 2}}}
```

As you can see, the update-values function plays nice with existing Clojure functions as well.

Disclosure: I am long AAPL and own every device Apple puts out. Don't take my example numbers to literally. Also, much to my dismay, [DRW](#) has yet to discard all of our windows boxes.

Labels: [clojure](#), [maps](#)

Share: [Email](#) | [Del.icio.us](#) | [Digg](#) | [Reddit](#) | [Shadow](#) | [Rojo](#)

posted by jaycfields : 3:33 PM [5 comments](#) 
[([www](#)) ([email](#)) ([work](#))]

Monday, August 01, 2011

Clojure: memfn

The other day I stumbled upon Clojure's [memfn](#) macro.

The memfn macro expands into code that creates a fn that expects to be passed an object and any args and calls the named instance method on the object passing the args. Use when you want to treat a Java method as a first-class fn.

```
(map (memfn charAt i) ["fred" "ethel" "lucy"] [1 2 3])
-> (\r \h \y)
```

-- clojure.org

At first glance it appeared to be something nice, but even the documentation states that "*...it is almost always preferable to do this directly now...*" - with an anonymous function.

```
(map #(.charAt %1 %2) ["fred" "ethel" "lucy"] [1 2 3])
-> (\r \h \y)
```

-- clojure.org, again

I pondered memfn. If it's almost always preferable to use an anonymous function, when is it preferable to use memfn? Nothing came to mind, so I moved on and never really gave memfn another thought.

Then the day came where I needed to test some Clojure code that called some very ugly and complex Java.

In production we have an object that is created in Java and passed directly to Clojure. Interacting with this object is easy (in production); however, creating an instance of that class (while testing) is an entirely different task. My interaction with the instance is minimal, only one method call, but it's an important method call. It needs to work perfectly today and every day forward.

I tried to construct the object myself. I wanted to test my interaction with this object from Clojure, but creating an instance turned out to be quite a significant task. After failing to easily create an instance after 15 minutes I decided to see if `memfn` could provide a solution. I'd never actually used `memfn`, but the documentation seemed promising.

In order to verify the behavior I was looking for, all I'll I needed was a function that I could rebound to return an expected value. The `memfn` macro provided exactly what I needed.

As a (contrived) example, let's assume you want to create a new order with a sequence id generated by `incrementAndGet` on [AtomicLong](#). In production you'll use an actual `AtomicLong` and you might see something like the example below.

```
(def sequence-generator (AtomicLong.))
(defn new-order []
  (hash-map :id (.incrementAndGet sequence-generator)))
(println (new-order)) ; => {:id 1}
(println (new-order)) ; => {:id 2}
```

While that might be exactly what you need in production, it's generally preferable to use something more explicit while testing. I haven't found an easy way to rebound a Java method (`.incrementAndGet` in our example); however, if I use `memfn` I can create a first-class function that is easily rebound.

```
(def sequence-generator (AtomicLong.))
(def inc&get (memfn incrementAndGet))
(defn new-order []
  (hash-map :id (inc&get sequence-generator)))
(println (new-order)) ; => {:id 1}
(println (new-order)) ; => {:id 2}
```


At this point we can see that `memfn` is calling our `AtomicLong` and our results haven't been altered in anyway. The final example shows a version that uses binding to ensure that `inc&get` always returns 10.

```
(def sequence-generator (AtomicLong.))
(def inc&get (memfn incrementAndGet))
(defn new-order []
  (hash-map :id (inc&get sequence-generator)))
(println (new-order)) ; => 1
(println (new-order)) ; => 2
(binding [inc&get (fn [_] 10)])
  (println (new-order)) ; => 10
  (println (new-order))) ; => 10
```

With `inc&get` being constant, we can now easily test our `new-order` function.

Labels: [clojure](#), [clojure functions](#), [memfn](#), [testing](#)

Share: [Email](#) | [Delicio.us](#) | [Digg](#) | [Reddit](#) | [Shadow](#) | [Rojo](#)

posted by jaycfields : 6:47 PM [2 comments](#) 
[([www](#)) ([email](#)) ([work](#))]

Wednesday, May 04, 2011

Clojure: Get All Nested Map Values (N levels deep)

I recently needed to pull all the values from a nested map. The map I was working with was designed to give easy access to data like so (formatted):

```
user=>
(def my-data {"Jay"
              {"clojure"
               {:name "Jay", :language "clojure", :enjoys true},
               "ruby"
               {:name "Jay", :language "ruby", :enjoys true}}
              "Jon"
              {"java"
               {:name "Jon", :language "java", :enjoys true}}})
#'user/my-data
user=> (get-in my-data ["Jon" "java"])
{:name "Jon", :language "java", :enjoys true}
user=> (get-in my-data ["Jay" "ruby"])
{:name "Jay", :language "ruby", :enjoys true}
```

This worked for all of the ways the data was accessed, until someone asked for a list of all people and all languages.

I needed a function that grabbed all the values nested to a point in the map (grabbing only the values instead of the deepest map wouldn't have provided valuable information). I created the following function that should grab all the values up to the nesting level specified.

```
(defn nth-vals* [a i m]
  (if (and (map? m) (> i 0))
      (reduce into a (map (fn [v] (nth-vals* a (dec i) v)) (vals m)))
      (conj a m)))
(defn nth-vals [i m]
  (if (nil? m)
      {}
      (nth-vals* [] i m)))
```

The nth-vals function can be used as the following example shows. (assuming the same map for my-data) (formatted)

```
user=> (nth-vals 2 my-data)
[{:name "Jay", :language "clojure", :enjoys true}
 {:name "Jay", :language "ruby", :enjoys true}
 {:name "Jon", :language "java", :enjoys true}]
```

For reference, here's what's returned if we reduce the map all the way to it's values.

```
user=> (nth-vals 3 my-data)
["Jay" "clojure" true "Jay" "ruby" true "Jon" "java" true]
```

That list of values may be helpful for someone else, but it wouldn't have solved our current problem.


And, if you're interested, here's what's returned if you ask for 1 level deep of values. (formatted, again)

```
user=> (nth-vals 1 my-data)
[{"clojure" {:name "Jay", :language "clojure", :enjoys true},
  "ruby" {:name "Jay", :language "ruby", :enjoys true}}
 {"java" {:name "Jon", :language "java", :enjoys true}}]
```

I wouldn't at all be surprised if something like this already exists in Clojure core, but I haven't found it yet.

Labels: [clojure](#), [maps](#)

Share: [Email](#) | [Delicio.us](#) | [Digg](#) | [Reddit](#) | [Shadow](#) | [Rojo](#)

posted by Jay Fields : 8:26 PM [1 comments](#) 
[([www](#)) ([email](#)) ([work](#))]

Monday, May 02, 2011

Clojure: Converting a Java Object to a Clojure Map

Clojure has a great library that helps facilitate most things you'd want to do, but it's also easy to roll your own solution if you need something more specific. This blog entry is about converting an object to a map, something Clojure already has a function for: [bean](#). From the docs:

Takes a Java object and returns a read-only implementation of the map abstraction based upon its JavaBean properties.

The bean function is easy enough to use, as the following example shows (formatted for readability).

```
user=> (import 'java.util.Calendar)
java.util.Calendar
user=> (-> (Calendar/getInstance) .getTime bean)
{:seconds 16,
 :date 2,
 :class java.util.Date,
 :minutes 35,
 :hours 12,
 :year 111,
 :timezoneOffset 240,
 :month 4,
 :day 1,
 :time 1304354116038}
```

The bean function is simple and solves 99% of the problems I encounter. Generally I'm [destructuring](#) a value that I pull from a map that was just created from an object; bean works perfectly for this.

However, I have run across two cases where I needed something more specialized.

The first case involved processing a large amount of data in a very timely fashion. We found that garbage collection pauses were slowing us down too much, and beaming the objects that were coming in was causing a large portion of the garbage. Since we were only dealing with one type of object we ended up writing something specific that only grabbed the fields we cared about.

The second case involved working with less friendly Java objects that needed to be converted to maps that were going to be used many times throughout the system. In general it isn't a big deal working with ugly Java objects. You can destructure the map into nice var names, or you can use the [rename-keys](#) function if you plan on using the map more than once. This probably would have worked for me, but I had an additional requirement. The particular Java objects I was working with had many methods/fields and the maps contained about 80% more information than I needed. So, I could have used bean, then rename keys, then dissoc. But, this felt similar enough to the situation in the past where I was working with specific method names and I thought I'd look for a more general solution.

The following example shows the definition of `obj->map` and the [REPL](#) output of using it.


```
user=> (import 'java.util.Date)
java.util.Date
user=> (defmacro obj->map [o & bindings]
  (let [s (gensym "local")]
    `(let [~s ~o]
      ~(->> (partition 2 bindings)
        (map (fn [[k v]]
              (if (vector? v)
                  [k (list (last v) (list (first v) s))]
                  [k (list v s)]))))
          (into {}))))))
#'user/obj->map
user=> (obj->map (Date. 2012 1 31)
      :month .getMonth
      :year [ .getYear #(* 2 %)]
      :day .getDate)
{:month 2, :year 4024, :day 2}
```

As you can see from the output, the macro grabs the values of the methods you specify and stores them in a map. At one point I also needed to do a bit of massaging of a value, so I included the ability to use a vector to specify the method you care about and a function to apply to the value (the result of the function will become the value). In the example, you can see that the `:year` is double the value of what `.getYear` returns.

In general you should stick with bean, but if you need something more discriminating, `obj->map` might work for you.

Labels: [clojure](#), [maps](#), [object](#)

Share: [Email](#) | [Del.icio.us](#) | [Digg](#) | [Reddit](#) | [Shadow](#) | [Rojo](#)

posted by Jay Fields : 12:15 PM [2 comments](#) 
[([www](#)) ([email](#)) ([work](#))]

Saturday, April 09, 2011

Clojure: State Management

Those unfamiliar with Clojure are often interested in how you manage changing state within your applications. If you've heard a few things about Clojure but haven't really looked at it, I wouldn't be surprised if you thought it was impossible to write a "real" application with Clojure since "everything is immutable". I've even heard a developer that I respect make the mistake of saying: we're not going to use Clojure because it doesn't handle state well.

Clearly, state management in Clojure is greatly misunderstood.

I actually had a hard time not calling this blog entry "Clojure, it's about state". I think state shapes Clojure more than any other influence; it's the core of the language (as far as I can tell). Rich Hickey has clearly spent a lot of time thinking about state - there's an essay at <http://clojure.org/state> which describes common problems with a traditional approach to state management and Clojure's solutions.

Rich's essay does a good job of succinctly discussing his views on state; you should read it before you continue with this entry. The remainder of this entry will give examples of how you can manage state using Clojure's functions.

At the end of Rich's essay he says:

In the local case, since Clojure does not have mutable local variables, instead of building up values in a mutating loop, you can instead do it functionally with recur or reduce.

Before we get to reduce, let's start with the simplest example. You have an array of ints and you want to double each integer. In a language with mutable state you can loop through the array and build a new array with each integer doubled.

```
for (int i=0; i < nums.length; i++) {
    result.add(nums[i] * 2);
}
```

In Clojure you would build the new array by calling the [map](#) function with a function that doubles each value. (I'm using Clojure 1.2)

```
user=> (map (fn [i] (* i 2)) [1 2 3])
(2 4 6)
```

If you're new to Clojure there's a few things worth mentioning. "user=>" is a [REPL](#) prompt. You enter some text and hit enter and the text is evaluated. If you've completed the list (closed the parenthesis), the results of evaluating that list will be printed to the following line.

I remember what I thought the first time I looked at a lisp, and I know the code might not look like readable code, so here's a version that breaks up a few of the concepts and might make it easier to digest the example.

```
user=> (defn double-int [i] (* i 2))
#'user/double-int
user=> (def the-array [1 2 3])
#'user/the-array
```

```
user=> (map double-int the-array)
(2 4 6)
```

In the first Clojure example you call the [fn](#) function to create an anonymous function, that was then passed to the map function (to be applied to each element of the array). The map function is a [high order function](#) that can take an anonymous function (example 1) or a named function (double-int, example 2). In Clojure (def ...) is a [special form](#) that allows you to define a var and [defn](#) is a function that allows you to easily define a function and assign it to a var. The syntax for defn is pretty straightforward, the first argument is the name, the second argument is the argument list of the new function, and any additional forms are the body of the function you are defining.

Once you get used to Clojure's syntax you can even have a bit of fun with your function naming that might result in concise and maintainable code.

```
user=> (defn *2 [i] (* 2 i))
#'user/*2
user=> (map *2 [1 2 3])
(2 4 6)
```

but, I digress.

Similarly, you may want to sum the numbers from an array.

```
for (int i = 0; i < nums.length; i++) {
    result += nums[i];
}
```

You can achieve goal of reducing an array to a single value in Clojure using the [reduce](#) function.

```
user=> (reduce + [1 2 3])
6
```

Clojure has several functions that allow you to create new values from existing values, which should be enough to solve any problem where you would traditionally use local mutable variables.

For non-local mutable state you generally have 3 options: [atoms](#), [refs](#), and [agents](#).

When I started programming in Clojure, atoms were my primary choice for mutable state. Atoms are very easy to use and only require that you know a few functions to interact with them. Let's assume we're building a trading application that needs to keep around the current price of Apple. Our application will call our `apple-price-update` function when a new price is received and we'll need to keep that price around for (possible) later usage. The example below shows how you can use an atom to track the current price of Apple.

```
user=> (def apple-price (atom nil))
#'user/apple-price
user=> (defn update-apple-price [new-price] (reset! apple-price new-price))
#'user/update-apple-price
user=> @apple-price
nil
user=> (update-apple-price 300.00)
300.0
user=> @apple-price
300.0
```



```
user=> (update-apple-price 301.00)
301.0
user=> (update-apple-price 302.00)
302.0
user=> @apple-price
302.0
```

The above example demonstrates how you can create a new atom and reset its value with each price update. The [reset!](#) function sets the value of the atom synchronously and returns its new value. You can also query the price of apple at any time using [@ \(or deref\)](#).

If you're coming from a Java background the example above should be the easiest to relate to. Each time we call the update-apple-price function our state is set to a new value. However, atoms provide much more value than simply being a variable that you can reset.

You may remember the following example from [Java Concurrency in Practice](#).

```
@NotThreadSafe
public class UnsafeSequence {
    private int value;
    /**
     * Returns a unique value.
     */
    public int getNext() {
        return value++;
    }
}
```

The book explains why this could cause potential problems.

The problem with UnsafeSequence is that with some unlucky timing, two threads could call getNext and receive the same value. The increment notation, nextValue++, may appear to be a single operation, but is in fact three separate operations: read the value, add one to it, and write out the new value. Since operations in multiple threads may be arbitrarily interleaved by the runtime, it is possible for two threads to read the value at the same time, both see the same value, and then both add one to it. The result is that the same sequence number is returned from multiple calls in different threads.

We could write a get-next function using a Clojure atom and the same race condition would not be a concern.

```
user=> (def uniq-id (atom 0))
#'user/uniq-id
user=> (defn get-next [] (swap! uniq-id inc))
#'user/get-next
user=> (get-next)
1
user=> (get-next)
2
```

The above code demonstrates the result of calling get-next multiple times (the [inc](#) function just adds one to the value passed in). Since we aren't in a multithreaded environment the example isn't exactly breathtaking; however, what's actually happening under the covers is described very well on clojure.org/atoms -

[Y]ou 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.

Also, remember that changes to atoms are synchronous, so our `get-next` function will never return the same value twice.

(note: while Java already provides an `AtomicInteger` class for handling this issue - that's not the point. The point of the example is to show that an `Atom` is safe to use across threads.)

If you're truly interested in verifying that an atom is safe across threads, [The Joy of Clojure](#) provides the following snippet of code (as well as a wonderful explanation of all things Clojure, including mutability).

```
(import '(java.util.concurrent Executors))
(def *pool* (Executors/newFixedThreadPool
  (+ 2 (.availableProcessors (Runtime/getRuntime)))))
(defn dothreads [f & {thread-count :threads exec-count :times :or {thread-count 1 exec-count 1}}]
  (dotimes [t thread-count]
    (.submit *pool* #(dotimes [_ exec-count] (f)))))
(def ticks (atom 0))
(defn tick [] (swap! ticks inc))
(dothreads tick :threads 1000 :times 100)
@ticks ;=> 100000
```

There you have it, 1000 threads updated ticks 100 times without issue.

Atoms work wonderfully when you want to insure atomic updates to an individual piece of state; however, it probably won't be long before you find yourself wanting to coordinate some type of state update. For example, if you're running an online store, when a customer cancels an order the order is either active or cancelled; however, the order should never be active and cancelled. If you were to keep a set of active orders and a set of cancelled orders, you would never want to have an order be in both sets at the same time. Clojure addresses this issue by using refs. Refs are similar to atoms, but they also participate in coordinated updates.

The following example shows the `cancel-order` function moving an order-id from the active orders set into the cancelled orders set.

```
user=> (def active-orders (ref #{2 3 4}))
#'user/active-orders
user=> (def cancelled-orders (ref #{1}))
#'user/cancelled-orders
user=> (defn cancel-order [id]
  (dosync
    (commute active-orders disj id)
    (commute cancelled-orders conj id)))
#'user/cancel-order
user=> (cancel-order 2)
#{1 2}
```

```
user=> @active-orders
#{3 4}
user=> @cancelled-orders
#{1 2}
```

As you can see from the example, we're moving an order id from active to cancelled. Again, our [REPL](#) session doesn't show the power of what's going on with a ref, but clojure.org/refs contains a good explanation -

All changes made to Refs during a transaction (via `ref-set`, `alter` or `commute`) will appear to occur at a single point in the 'Ref world' timeline (its 'write point').

The above quote is actually only 1 item in a 10 point list that discusses what's actually going on. It's worth reviewing the list a few times until you feel comfortable with everything that's going on. But, you don't need to completely understand everything to get started. You can begin to experiment with refs anytime you know you need coordinated changes to more than one piece of state.

When you first begin to look at refs you may wonder if you should use `commute` or `alter`. For most cases `commute` will provide more concurrency and is preferred; however, you may need to guarantee that the ref has not been updated during the life of the current transaction. This is generally the case where `alter` comes into play. The following example shows using `commute` to update two values. The example demonstrates that the pairs are always updated only once; however, it also shows that the function is simply applied to the current value, so the incrementing is not sequential and `@uid` can be dereferenced to the same value multiple times.

```
user=> (def uid (ref 0))
#'user/uid
user=> (def used-id (ref []))
#'user/used-id
user=> (defn use-id []
      (dosync
        (commute uid inc)
        (commute used-id conj @uid)))
#'user/use-id
user=> (dothreads use-id :threads 10 :times 10)
nil
user=> @used-id
[1 2 3 4 5 6 7 8 9 10 ... 89 92 92 94 93 94 97 97 99 100]
```

The above example shows that `commute` simply applies regardless of the underlying value. As a result, you may see duplicate values and gaps in your sequence (shown in the 90s in our output). If you wanted to ensure that the value didn't change during your transaction you could switch to `alter`. The following example shows the behavior of changing from `commute` to `alter`.

```
user=> (def uid (ref 0))
#'user/uid
user=> (def used-id (ref []))
#'user/used-id
user=> (defn use-id []
      (dosync
        (alter uid inc)
        (alter used-id conj @uid)))
#'user/use-id
user=> (dothreads use-id :threads 10 :times 10)
nil
```

```
user=> @used-id
[1 2 3 4 5 6 7 8 9 10 ... 91 92 93 94 95 96 97 98 99 100]
```

There are more advanced examples using refs in The Joy of Clojure for those of you looking to discuss corner case conditions.

Last, but not least, agents are also available. From clojure.org/agents -

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.


While I understand agents conceptually, I haven't used them much in practice. Some people [love them](#), and the last team I was on switched to using agents heavily in one of our applications shortly after I left. But, I personally don't have enough experience to say exactly where I think they fit in. I'm sure that will be a topic for a future blog post.

Between Rich's essay and the examples above I hope a few things have become clear:

- Clojure has plenty of support for managing state
- Rich's distinction between identity and value allows Clojure to benefit from immutable structures while also allowing identity reassignment.
- Clojure, it's about state.

Labels: [clojure](#), [state](#)

Share: [Email](#) | [Del.icio.us](#) | [Digg](#) | [Reddit](#) | [Shadow](#) | [Rojo](#)

posted by Jay Fields : 10:41 AM [4 comments](#) 
[([www](#)) ([email](#)) ([work](#))]

Thursday, March 24, 2011

Readable Clojure Without a Java Equivalent?

I've recently joined a new team and we've been doing a bit of Clojure. If you've never done Lisp (and I hadn't before I found Clojure) it's natural to ask the question: Will programming in Lisp, especially [Prefix Notation](#), ever feel natural? The question came up a few weeks ago and I had two answers

First of all, I've never been really upset about parenthesis. In fact, I've been doing a bit of Java these days and I don't see much difference between `verify(foo).bar(eq(100), any(Baz.class), eq("Cat"))` and `(-> foo verify (.bar (eq 100) (any Baz) (eq "Cat")))`. By my count it's the same number of parenthesis. Where they're located moves around a bit, but I don't consider that to be a good or a bad thing.

People also like to bring up examples like this:

```
(apply merge-with +
  (pmap count-lines
    (partition-all *batch-size*
```

```
(line-seq (reader filename))))))
```

Stefan Tilkov addressed this in a previous [blog entry](#), and (in the comments of Stefan's entry) Rich Hickey points out that you can use a few different versions if you prefer to lay your code out in a different manner. Below are two alternative solutions Rich provided.

```
; The same code in a pipelined Clojure style:
(->> (line-seq (reader filename))
      (partition-all *batch-size*)
      (pmap count-lines)
      (apply merge-with +))
; and in a step-wise Clojure style with labeled interim results (a la Adrian's comment):
(let [lines (line-seq (reader filename))
      processed-data (pmap count-lines
                           (partition-all *batch-size* lines))]
      (apply merge-with + processed-data))
```

I've felt the same way for awhile. You can write Cobol in any language. The real question is, are you up for learning how to solve problems elegantly and idiomatically in a new language. If you're willing to invest the time, you'll be able to find out for yourself if it feels natural to write idiomatic Clojure code.

That was my answer, until today. While working on some Java code I stumbled on the following Java method.

```
public void onFill(int fillQty) {
    this.qty = Math.max(this.qty - fillQty, 0);
}
```

This is a simple Java method that is decrementing the outstanding quantity state of an order by the amount of the order that just been filled. While reading the line I couldn't help but feel like there should be a more elegant way to express the logic. You want to set the outstanding quantity state to the current outstanding quantity minus what's just been filled, but you also never want the outstanding quantity to go below zero. I read from left to right, and I really wanted a way to express this logic in a way that followed the left to right pattern.

In Clojure, this is easy to do:

```
(swap! qty #(-> % (- fill-qty) (Math/max 0)))
```

For readers who are less familiar with Clojure's [dispatch reader macro](#), the above example can also be written as:

```
(swap! qty (fn [current-qty] (-> current-qty (- fill-qty) (Math/max 0))))
```

In the example above `swap!` is setting the `qty` state with the return value of the function.

If you're really new to Clojure, that might still be too much to take, so we can reduce the example and remove the state setting. Here's the version in Java that ignores setting state.

```
Math.max(this.qty - fillQty, 0);
```

The example below is a logical equivalent in Clojure.

```
(-> qty (- fill-qty) (Math/max 0))
```

When reading the above Java example I'm forced to put `Math.max` on my mental stack, evaluate `this.qty - fillQty`, and then mentally evaluate the method I put on the stack with my new result and the additional args. This isn't rocket science, but it's also not how I naturally read (left to right). On the other hand, when I read the Clojure version I think - take the current quantity, subtract the fill quantity, then take the max of that and zero. The code reads in small, logical chunks that are easy for me to digest.

Obviously, the Java code can also be rewritten in a few other ways. Here's an example of Java code that reads left to right and top to bottom.

```
public void onFill(int fillQty) {
    this.qty -= fillQty
    this.qty = Math.max(this.qty, 0);
}
```

And, I can do something similar in Clojure, if I want.

```
(swap! qty #(- % fill-qty))
(swap! qty #(Math/max % 0))
```

While it's possible to write Clojure similar to the above example, it's much more likely that you'd use a `let` statement if you wanted to break up the two operations.

```
(defn update-qty [current fill-qty]
  (let [total-qty (- current fill-qty)]
    (Math/max total-qty 0)))
(swap! qty update-qty fill-qty)
```

The above example is probably about equivalent to the following Java snippet.

```
public void onFill(int fillQty) {
    int newTotalQty = this.qty - fillQty
    this.qty = Math.max(newTotalQty, 0);
}
```

So, I can write code that is similar to my options in Java, but I'm still left wanting a Java version that is similar to this Clojure example:

```
(swap! qty #(-> % (- fill-qty) (Math/max 0)))
```

The only thing that springs to mind is some type of fluent interface that allows me to say `this.qty = this.qty.minus(fillQty).maxOfIdentityOrZero()`, but I can't think of a realistic way to create that API without quite a bit of infrastructure code (including my own `Integer` class).

(note, you could extend `Integer` in a language with open classes, but that's outside the scope of this discussion)

The last Clojure example is definitely the version of the code I would prefer. My preference is based on the way the code reads in concise, logical chunks from left to write. I don't have to solve inside out like the original java version forces me to, and I don't have to split my work across two lines.


I'm sure there are situations where Java allowed me to create an elegant solution that wouldn't have been possible in Clojure. This entry isn't designed to send a "Clojure is better than Java" message. I don't believe that. However, before today I've held the opinion that you can write Clojure that logically

breaks up problems in ways very similar to what you do using Java. However, I've now also expanded my opinion to include the fact that in certain situations Clojure can also allow me to solve problems in a way that I find superior to my options within Java.

And, yes, after a bit of time getting used to Lisp syntax, it definitely does feel perfectly natural to me when I'm developing using Clojure.

Labels: [clojure](#), [java](#)

Share: [Email](#) | [Del.icio.us](#) | [Digg](#) | [Reddit](#) | [Shadow](#) | [Rojo](#)

posted by Jay Fields : 9:48 PM [6 comments](#) 
[([www](#)) ([email](#)) ([work](#))]

Wednesday, March 02, 2011

Clojure: Eval-ing a String in Clojure

I recently needed to eval a string into Clojure and found it was easy, but it wasn't accomplished in the way I expected. My recent experience with eval was in Ruby, so I naturally reached for Clojure's [eval](#) function. I quickly found that Clojure's eval was not what I was looking for, well, not exactly what I was looking for.

```
(eval "(+ 1 1)")  
"+ 1 1)"
```

This quick trip to the [REPL](#) reminded me of the last time I needed to eval a string, and the previous experiences I'd had with [read-string](#) and [load-string](#). It turns out load-string does the trick, and read-string + eval can also get the job done.

```
user=> (load-string "(+ 1 1)")  
2  
user=> (eval (read-string "(+ 1 1)"))  
2
```

Looking at the documentation for read-string and load-string can help point you at which you might need for your given situation.

read-string: Reads one object from the string [argument]
load-string: Sequentially read[s] and evaluate[s] the set of forms contained in the string [argument]

Okay, it looks like you'd probably want load-string if you had multiple forms. Back to the REPL.

```
user=> (load-string "(println 1) (println 2)")  
1  
2  
user=> (eval (read-string "(println 1) (println 2)"))  
1
```

As you can see, load-string evaluated the entire string and read-string only returned the first form to

eval. The last examples show printing values (and I removed the nil return values), but it's more likely that you'll be concerned with returning something from read-string or load-string. The following example shows what's returned by both functions.

```
user=> (load-string "1 2")
2
user=> (eval (read-string "1 2"))
1
```

Given our previous results, I'd say load-string and read-string are returning the values you'd expect. It seems like it's easy to default to load-string, but that's not necessarily the case if evaluation time is important to you. Here's a quick snippet to get the point across.


```
user=> (time (dotimes [_ 100] (eval (read-string "1 2"))))
"Elapsed time: 12.277 msecs"
nil
user=> (time (dotimes [_ 100] (load-string "1 2")))
"Elapsed time: 33.024 msecs"
nil
```

One of the applications I previously worked on would write a large number of events to a log each day; the format used to log an event was a Clojure map. If the application was restarted the log would be read and each event would be replayed to get the internal state back to current. We originally started by using load-string; however, start-up time was becoming a problem, and a switch to read-string completely removed the issue.

Like so many examples in programming, there's a reason that two versions exist. It's worth taking the time to see which solution best fits your problem.

Labels: [clojure](#), [eval](#), [strings](#)

Share: [Email](#) | [Del.icio.us](#) | [Digg](#) | [Reddit](#) | [Shadow](#) | [Rojo](#)

posted by Jay Fields : 4:45 AM [1 comments](#) 
[([www](#)) ([email](#)) ([work](#))]

Tuesday, March 01, 2011

Clojure: if-let and when-let

I'm a fan of [if-let](#) and [when-let](#). Both can be helpful in creating succinct, readable Clojure code. From the documentation:

If test is true, evaluates then with binding-form bound to the value of test, if not, yields else

A quick example should demonstrate the behavior.

```
user=> (if-let [a :anything] a "no")
:anything
user=> (if-let [a nil] a "no")
"no"
user=> (if-let [a false] a "no")
```


"no"

The above example demonstrates a simple case where a [truthy](#) value causes the "then" form to be returned, and a [falsey](#) value (nil or false) causes the "else" form to be returned. The example also shows that "a" is bound to the value :anything and can be used within the "then".

The following example shows that you can put code in the "then" and "else" statement's as well.

```
user=> (if-let [a 4] (+ a 4) (+ 10 10))
8
user=> (if-let [a nil] (+ a 4) (+ 10 10))
20
```

Just like above, when "a" is truthy the value is bound and it can be used as desired in the "then".

The when-let function behaves basically the same way, except there is no "else".

```
user=> (when-let [a 9] (+ a 4))
13
user=> (when-let [a nil] (+ a 4))
nil
user=> (when-let [a 9] (println a) (+ a 4))
9
13
```

The example demonstrates that then "then" case is very similar to if-let; however, if the test fails the when-let function simply returns nil. The last when-let example also shows how it slightly differs from an if-let: you can pass as many forms to the "then" as you'd like. If the test evaluates to true, all of the additional forms will be evaluated.

There are a few things worth knowing about if-let and when-let with respect to the bindings. Both if-let and when-let require a vector for their bindings, and there must be exactly two forms in the binding vector. The following example shows Clojure's response if you choose not to follow those rules.

```
user=> (when-let (a 9) (println a))
java.lang.IllegalArgumentException: when-let requires a vector for its binding
(NO_SOURCE_FILE:0)
user=> (when-let nil (println "hi"))
java.lang.IllegalArgumentException: when-let requires a vector for its binding
(NO_SOURCE_FILE:0)
user=> (when-let [a 9 b nil] (println a b))
java.lang.IllegalArgumentException: when-let requires exactly 2 forms in binding vector
(NO_SOURCE_FILE:0)
```

It's nice to know if-let and when-let, but they aren't exactly hard concepts to follow. Once someone points them out to you, I expect you'll be able to easily use them in your code without much effort.

However, what is the impact of [destructuring](#) on if-let and when-let? I know what I expected, but I thought it was worth a quick trip to the [REPL](#) to make sure my assumptions were correct. The following code shows that as long as the value being bound is truthy the "then" will be executed - destructuring values have their normal behavior and do no effect execution flow.

```
user=> (when-let [{a :a} {:a 1}] [a])
[1]
```

```
user=> (when-let [{a :a} {}] [a])
[nil]
user=> (when-let [{a :a} {:a false}] [a])
[false]
user=> (when-let [{a :a} nil] [a])
nil
```

As you would expect the when-let is consistent and the destructuring behavior is also consistent.

if-let and when-let are good, give them a shot.

Labels: [clojure](#), [destructuring](#), [if](#)