MAR 26TH, 2012 | COMMENTS

How to Call Clojure Functions From Java

I'm fairly new to Clojure and possibly even less familiar with the state of 21st-century Java, so when I wanted to generate a Clojure jar file that someone else could call from ordinary Java code, in an ordinary way, I didn't have much to start with. I hit up Google and Stack Overflow fairly hard with permutations of the bag of words **{call invoke Clojure from Java interop WTF}**. But I didn't find more than one or two relevant, recent examples, of a Clojure 1.3 (or later) vintage. The official document on Clojure compilation is fine as far as it goes, and a good reference, but was not quite enough to actually set my feet on the path.

My goal was to generate a jar file that could be executed standalone (e.g. double-clicked), *OR* used as a library, and no user would need to know there was Clojure code inside.

Here, then, I'm presenting a complete walkthrough that will get things up and running with minimal tool dependencies — no Eclipse, ant, maven, CCW, et cetera. We're going to create just a simple "Hello, Clojure" application in Clojure, a "Hello, Java" application, and then mix the two.

(Don't have Clojure? This will only take a minute. If you're a Mac user and not using Macports, I've had very good experiences with the Homebrew package manager. Get homebrew and then the not just excellent but indispensable leiningen with a couple quick lines)::

```
1  $  /usr/bin/ruby -e "$(/usr/bin/curl -fksSL https://raw.github.com/mxcl/homeb
2  $  brew install leiningen
```

My working version of leiningen here is 1.7.0. (**Update**: Use `brew install –devel leiningen` to get the leiningen 2 preview release instead.) To start the Clojure project:

```
1  $  lein new ThingOne
2  $  cd ThingOne
```

Edit the **project.clj** file to look like this:

```
(defproject ThingOne "1.0.0-SNAPSHOT"
  :description "Hello, Clojure"
  :dependencies [[org.clojure/clojure "1.3.0"]]
  :aot [ThingOne.core]
  :main ThingOne.core)
```

The :aot incantation indicates that we're going to ahead-of-time compile one of the classes, for use directly in Java. (If we don't do AOT, our Clojure project *can* still be invoked in Java, but only by making the user work through clojure.lang.RT. Again, that approach does work (here are some examples from Jay Fields), but it's not what I wanted here.

Specifying :main here seems to be necessary to ensure that the jar file will still have a 'main' that can be found and run when the jar is executed with 'java -jar', or simply double-clicked in a sufficiently modern OS.

So now we're ready to edit **src/core.clj** into this sort of shape:

```
(ns ThingOne.core
 (:gen-class
    :methods [#^{:static true} [foo [int] void]]))

(defn -foo [i] (println "Hello from Clojure. My input was " i))

(defn -main [] (println "Hello from Clojure -main." ))
```

Using gen-class incantations I'm not yet entirely comfortable with and completely stole from the Internet, we declare the **-foo**function as a static function, taking an int as input, and returning void. This is so that it can be called from a static function — in this case, a main function, below. If it's not static, this magic can be omitted.

In the case of an instance method (which I'm not covering here), we would need to include an extra first parameter, conventionally called 'this', to account for the fact that in an instance method call in Java (or C++, for that matter), the object itself is provided by the compiler as an implicit first parameter.

Now for the jar. Leiningen can pull in all dependencies (including Clojure itself), do the AOT compilation, and generate the all-inclusive jar file we want: the coveted uberjar.

```
$  lein compile
$  lein uberjar
```

(With leiningen 1.7.0, I found I need to do these steps separately, though the 'uberjar' command really should do it all.)

The uberjar embraces the whole project as well as all of its dependencies — so all of Clojure lives in that jar. (On my machine, the jar file weighs in at about 3.3 MB.) Create a new directory for the Java project, and copy the standalone uberjar into it for convenience.

```
$  cd ..
$  mkdir HelloJava && cd HelloJava
$  cp ../ThingOne/ThingOne-1.0.0-SNAPSHOT-standalone.jar .
```

Now to write the simplest possible Java program that will work,**HelloJava.java**.

```
import ThingOne.*;

class HelloJava {
    public static void main(String[] args) {
        System.out.println("Hello from Java!");
        core.foo (12345);
    }
}
```

We indiscriminately imported everything in the ThingOne namespace here, and then

Compile the Java file into a .class file, being extremely clear & explicit about what's on the classpath with the -cp option:

```
$  javac -cp '.:ThingOne-1.0.0-SNAPSHOT-standalone.jar' HelloJava.java
```

To actually run the thing, we still have to be clear about the classpath, which tripped me up for a while:

```
$  java -cp '.:ThingOne-1.0.0-SNAPSHOT-standalone.jar' HelloJava
Hello from Java!
Hello from Clojure. My input was  12345
```

Success!

And we can run the jar standalone as well:

```
$  java -jar ThingOne-1.0.0-SNAPSHOT-standalone.jar
Hello from Clojure -main.
```

I've posted the source code for this example on github.