

Technical Explorations

by Keith R. Bennett

Dabbling in Clojure

Mar 16, 2009 • keithrbennett


Clojure (<http://www.clojure.org>) is a Lisp implementation running on the Java Virtual Machine, written primarily by Rich Hickey and designed for simple and reliable concurrent programming. Clojure is a radical departure from the more conventional languages such as C, C++, Java, and Ruby. As a functional language, using it requires a different outlook and thinking style.

Code written in Lisp or Clojure may appear to the uninitiated to be a confusing excess of parentheses and other seemingly cryptic text. In fact, I used to be one of the multitude who dismissed these languages with a mere “too many parentheses” as a feeble rationalization. However, after working with it for a bit I have instead come to view it with respect and even awe.

The Clojure community is a bright and helpful bunch; I often consulted them on the *#clojure* IRC channel, and the Google group at <http://groups.google.com/group/clojure> has had some very useful information as well.

The previous article ([JRuby — A Better Language for the Java Virtual Machine](#)) discussed using JRuby as a better language for programming on the JVM, and used as an example a Fahrenheit/Celsius temperature conversion Swing program. I’ve ported this program to Clojure so that we can contrast Clojure with Java and JRuby using a known quantity.

Here is an image of the application’s sole window. There are text fields for entering the temperature, and buttons and menu items to perform the conversions, clear the text fields, and exit the program.

The Fahrenheit Celsius temperature conversion Swing app window<figcaption class="caption wp-caption-text">the temperature conversion Swing app window</figcaption></figure>

The Git repository for this project is at <http://github.com/keithrbennett/multilanguage-swing/tree/master>. You can get to the *FrameInClojure.clj* source file by clicking the *src* folder, and then clicking *FrameInClojure.clj*. It would probably help to have the program open in another window (or printed out) while reading this article.

If you can access the REPL interactive Clojure command line, you can issue the following commands to run the example program and have access to the main-frame variable for inspection and manipulation:

```
(load-file "FrameInClojure.clj")
(println temp-converter/main-frame) ; illustrates access to the program's frame
(.setVisible temp-converter/main-frame false) ; illustrates that it can be manipulated
```

The strengths of Clojure are many, and a Swing application is not the best program with which to showcase them. Therefore, it would not be fair to judge Clojure based solely on the content of this article, because I am intentionally excluding its greatest strengths – they are outside of the scope of porting this Swing application, and I am not yet expert enough to be qualified to teach about them. However, this article *is* useful in illustrating some of the mechanics and philosophy of the language that can be used in general programming.

Firstly, Clojure (and, by extension, Lisp) struck me as a language that easily enables the writing of clear, concise, and logically structured code. I’m a big fan of this, as you will know if you have read my [Applying User Interface Design to Source Code](#) article.

Let

For example, `let` (see http://clojure.org/special_forms#let) is a construct that formalizes the definition of terms and their scope. Here’s an example of a function that uses it:

```
(defn center-on-screen
  "Centers a component on the screen based on the screen dimensions
  reported by the Java runtime."
  [component])
```

```
(let [
  screen-size    (.. Toolkit getDefaultToolkit getScreenSize)
  screen-width   (.getWidth screen-size)
  screen-height  (.getHeight screen-size)
  comp-width     (.getWidth component)
  comp-height    (.getHeight component)
  new-x          (/ (- screen-width comp-width) 2)
  new-y          (/ (- screen-height comp-height) 2)]

  (.setLocation component new-x new-y))
component)
```

The `let` clause above leaves no doubt 1) that those terms are inaccessible outside of the `let` clause, and 2) that all terms are colocated at the beginning of the `let` clause. The first feature is available in other languages such as Java, though it is (unfortunately) rarely used – I’m speaking of nesting some of a method’s code in curly braces without any enclosing `for`, `if`, `do`, or `while`, for the sole purpose of limiting the scope of the local variables used therein.

Variables in the `let` clause can use preceding variables, so it easily facilitates the use of intermediate variables for simplicity and clarity. For example, the `screen-width` and `screen-height` variables above use the `screen-size` variable defined before them.

In the above function, the `let` clause is the only content of this function. However, when this is not the case, and the function contains other code, the `let` clause clearly and cleanly displays for the writer a refactoring opportunity, namely the opportunity to extract the `let` clause into a function by itself.

Lambdas

As a functional language, the creation of a chunk of behavior that can be assigned to a variable (known as a lambda) is trivial. Here are two ways to create a lambda that returns the square of its argument...

```
(def square1 (fn [n] (* n n)))
(def square2 #( * % %))
```

... and here’s an example of using a lambda concisely to guarantee uniform creation of two similar objects:

```
(let [
  create-an-inner-panel #(JPanel. (GridLayout. 0 1 5 5))
  label-panel            (create-an-inner-panel)
  text-field-panel       (create-an-inner-panel)
  ;; ...
```

In the code above, a minimal lambda is created (the code between `#(` and `)`), and then used to create two panels immediately below. In Java you would need to create a method to do this, probably either another instance method of the same class, or a static utility method of the same or another class. The Clojure approach is superior because a) the lambda’s code is invisible outside of the function (and even the `let` clause) in which it is used, and b) it appears in the code immediately adjacent to its use.

Macros

Lisp and Clojure support macros that can greatly increase the power and flexibility of the language. For example, I found a need for a function that lazily initializes an object. In Java, it would have been:

```
>private MyClass foo;

public MyClass getFoo() {
  if (foo == null) {
    foo = new MyClass(); // or other way of initializing it
  }
  return foo;
}
```

I coded the equivalent Clojure code, but (like the Java code) it smelled of verbosity, not to mention the redundancy when doing the same with several objects:

```
>(def fahr-text-field nil)

(defn get-fahr-text-field []
  (if fahr-text-field
    fahr-text-field
    (do
```

```
(def fahr-text-field (create-a-text-field))
fahr-text-field))
```

So I asked the folks on the #clojure IRC channel. I was pointed to the `_delay_` macro. So using that, I could eliminate the need to define the variable, and do:

```
>(def get-fahr-text-field
  (let [x (delay (create-a-text-field))]
    #(force x)))
```

However, even that seemed like too much ceremony, so I asked them if there was a way to reduce that even further. Minutes later, one of the generous IRC folks offered the following macro:

```
>(defmacro lazy-init [f & args]
  `(let [x# (delay (~f ~@args))]
    #(force x#)))
```

Using this macro, definitions of lazily initialized objects were as easy as:

```
>(def get-fahr-text-field (lazy-init create-a-text-field))
```

Seeing that, it struck me that Clojure enables writing code that's DRYer than that of any other language with which I've worked. Through the use of macros, all the mechanics of the lazy initialization could be isolated down to two short, testable macros by an expert programmer or two, and then ignored by the rest of the world. Although I've done only a little Ruby metaprogramming, my sense is that Clojure's macros make it more powerful than Ruby in this respect. I invite you to discuss this in comments to this article; I for one would like to learn more about this.

As with JRuby, the Clojure implementation of `SimpleDocumentListener` is concise and straightforward:

```
>(defn create-simple-document-listener
  "Returns a DocumentListener that performs the specified behavior
  identically regardless of type of document change."
  [behavior]

  (proxy [DocumentListener] []
    (changedUpdate [event] (behavior event))
    (insertUpdate [event] (behavior event))
    (removeUpdate [event] (behavior event))))
```

Furthermore, the JRuby and Clojure implementations are superior to the Java implementation, because Java requires subclassing the `SimpleDocumentListener` for each type of behavior, and JRuby and Clojure do not. Not only does this reduce class clutter, it is conceptually simpler since the behavior is the only variable.

Unnamed Arguments

Note the underscore below:

```
>(def clear-action (create-action "Clear"
  (fn [_]
    (.setText (get-fahr-text-field) "")
    (.setText (get-cels-text-field) ""))
  ;; ...
```

All Swing Action implementations must include an *actionPerformed* function that takes a single `ActionEvent` parameter. However, in many cases, access to that event parameter is not needed. In Clojure, the underscore is used idiomatically to indicate that the argument it identifies is not subsequently used. Although it is a legal identifier, its unique appearance stands out, so it is effective in communicating that it has a special meaning. Interestingly, this feature is available in C++ by specifying the data type without a variable name, but to my knowledge was never implemented in the Java language. The underscore is a valid identifier in Java, so there is nothing preventing using it as a convention. However, Java does not support using the same identifier for more than one parameter, so one would have to kludge it, for example by naming them `_0`, `_1`, etc.

Different Meaning of Parentheses

One thing that threw me off several times while learning Clojure was the need to unlearn the idea that redundant parentheses were benign. In Clojure, wrapping an expression in parentheses means 'call the function that is the first expression, and pass the rest of the expressions, if any, as parameters'.

Therefore, unlike the other languages with which I've worked, `3` does not evaluate to the same value as `(3)`, as evidenced by REPL's responses below:

```
>user=> 3
3
user=> (3)
java.lang.ClassCastException: java.lang.Integer cannot be cast to clojure.lang.IFn (NO_SOURCE_FILE:0)
```

Conclusion

I hope this article has been of interest to you. Learning Clojure has been a fascinating experience to me, stretching my brain in ways it had never been stretched before. I've mentioned here some things that have caught my interest along the way. However, I want to reiterate that Clojure is a very powerful language, and I have truly only skimmed the very top here. If you are interested in learning more, you can visit the sites listed above. There is also an excellent book by Stuart Halloway called "Programming Clojure" at <http://pragprog.com/titles/shcloj/programming-clojure>. At the time of this writing, it is only available as a beta PDF, but is an excellent learning resource.

Published with [GitHub Pages](#)