Improving the Interoperability between Java and Clojure

Stephen Adams
Computer Science Discipline
University of Minnesota Morris
Morris, MN 56267
adams601@morris.umn.edu

Abstract

1 An Introduction to Clojure

Clojure is a programming language released in 2007 by Rich Hickey [5]. Clojure is part of the LISP family of languages. Some brief examples of LISP-like syntax are:

```
(+ 2 3)
```

Clojure uses prefix notation. To call a function you open a new list, the first thing in this list is the function name followed by the arguments. In the above case the function ”+” takes in the two ints 2 and 3 and will return 5. Also note that:

```
(+ 2 3 4)
```

is a valid function call that returns 9. Clojure supports defining functions with a variable number of arguments.

Many of Clojure's data structures are the corresponding Java data structure; strings, characters, and numbers are just the Java implementations[4]. Clojure collections, however, are not Java collections. Clojure implements its own collections, each of these collections are denoted by different bracket types, table 1 shows which literals correspond to each collection.

Clojure code is also Clojure data types. Function calls are just lists that consist of a symbol corresponding to a function name and the arguments to that function.

Clojure also includes several very powerful features for working with functions. The first allows for functions to be passed to other functions.

```
(defn square [x]
  (* x x))

(map square [1 2 3 4 5])
```

The example above does a few things. First a new function named square is defined; square takes in one argument named x and multiplies it by itself. Then the built in Clojure function map is called. Map takes in a function and a collection and applies the function to every element of the collection. In the example the call to map will return the vector [1 4 9 16 25].

Clojure also supports anonymous functions.

```
(defn all-same [vect]
```

Table 1: Clojure collection literals

Collection Type	Literal
List	(1 2 3 4)
Vector	[”apple” ”banana” ”orange”]
Hashmap	{ :address ”1234 Baker St.” :City Springfield :Zip 12345 }
Set	#{67 2 8.8 -78 }

```
    (if
      (empty? vect) true
      (every? (fn [x] (= (first vect) x)) (rest vect))
    )
  )
```

This example function takes in a vector and checks if every element in that vector is the same. Clojure's if is a function that takes in three arguments the first is the condition which if it evaluates to true the second argument is returned otherwise the third argument is returned. The "fn" function tells the compiler that a new function is being defined within the current scope (in this case the scope is the function all-same.) In this case the anonymous function takes the first element of vect and checks if it is equal to the argument x. The outer function every? takes a function and a collection, and returns true if the function returns true for every element in the collection. Using an anonymous function here allows for the function passed to every? to use the first element in vect which would not be available to another function because it would be out of scope. First and rest just return the first element in the vector and the vector without the first element respectively.

2 Current Java Interoperability

A major feature of Clojure is that it runs on the JVM. One of the main features that Rich Hickey wanted to have when he was first designing Clojure was complete compatibility with Java and its extensive library. Since Clojure and Java code both compile down to JVM bytecode Clojure can call code written in Java. I will now briefly go over the primary methods for calling Java code.

2.1 The Dot Special form

The simplest way to call Java methods is with the dot(".") special form. The dot special form will first take a class member or class name then a method name then the arguments the method takes [3]. Here are a few examples of the dot in use.

```
(. "fred" toUpperCase)
;;This will return "FRED".
(.toUpperCase "fred")
;;This also returns "FRED".
```

The second case will actually expand into the first call and is just a shortcut for accessing fields or methods. There are a few more shortcuts to make accessing static Java methods and objects.

```
(Math/PI)
;;Returns 3.141592653589793
(Math/abs -2)
```

```
;;Returns 2
```

2.2 Using Objects in Clojure

Static methods are only a small part of Java. Without effective ways of dealing with objects the usefulness of Java Clojure interop would be severely limited. Fortunately Clojure provides a full set of features that makes working with Java fairly easy.

To construct a new object you just use the ”new” special form.

```
(new StringBuffer "fred")
;;Returns #<StringBuffer fred>
```

To quickly modify an object the macro doto is available. A macro is a textual transformation. Essentially a macro expands from one piece of code to another. The doto first takes a call to new, then performs the rest of the supplied arguments on that object, and finally returns the resulting object.

```
(doto (new StringBuffer "fred")
  (.setCharAt 0 \F) (.append " is a nice guy!"))
;;Returns #<StringBuffer Fred is a nice guy!>
```

Doto is very handy for doing sequences of set methods right after you construct an object.

2.3 Making Objects in Clojure

There are problems that cannot be effectively solved without creating custom classes. In particular gui programming in swing, the primary Java gui tool, is quite difficult without the ability to extend the standard swing classes. Proxy and gen-class are the two functions provided by Clojure to implement, extend, or create Java classes.

2.3.1 Proxy

It has been said that Clojure does Java better than Java[2]. I think this is most true when using proxy. It is very common when programming in Swing in particular. When creating listeners to attach to buttons or other interactive interface devices it is very common to implement many different listeners. Proxy simplifies this process by allowing you to define a single superclass with methods that will work for your interface in most circumstances; and when you need to attach a listener instead of creating an entirely new subclass use proxy to create a new class that overwrites a few methods and inherits the rest.

Proxy is a macro that takes in two vectors and then a variable number of method declarations and returns a new object. The first vector declares which Java classes the proxy object will extend and which interfaces it implements. The second vector is a list of arguments

(which can be empty) to be passed to the superclass constructor. Any other arguments that are passed to proxy are method declarations[3].

For the following proxy examples I will be using the following basic java interface:

```
public interface TestInterface {
int square(int x);
}
```

This interface is in a separate interface file and is then imported into the following Clojure file.

```
(def test-inter
  (proxy [TestInterface] [] (square [x] (* x x))))
(. test-inter square 5)
;;Returns 25
```

There can be more functions than just square but they have to be defined in the interface as well.

2.3.2 Gen-class

Proxy is limited by what is defined in a super class or Java interface, on the other hand, gen-class is designed for implementing fully featured Java classes. Gen-class consists of a set of options which are represented as key value pairs. The keys consist of Clojure keywords which are just symbols that begin with a ":." Gen-class currently supports fifteen different options. I will only cover a select few of these options here, the complete specification can be found in [1] or [**?**]. The example below was take from [1]:

```
(ns some.Example)
(gen-class
  :prefix method-)

(defn method-toString
  [this]
  "Hello, world!")
```

After the call to gen-class itself are the methods that are associated with the type defined by the gen-class call. This is a fairly simple example but it gives the basic idea of how gen-class works. The only option that this class uses is the prefix option. This option defines what each method that corresponds to the class must begin with. When you want to call the method on the object you would use the following code:

```
(def aClass (new some.Example))
;;This constructs a new instance of the object
(.toString aClass)
;;This returns the string "Hello, world!" because the method-toString functi
```

```
;;in the class definition has the correct prefix for the object.
```

3 Suggestions

4 Conclusion and Future Work

References

[1] CLOJUREDOCS.ORG. gen-class, Mar. 2012.

[2] HALLOWAY, S. Clojure-java interop: A better java than java. QCon.

[3] HICKEY, R. Java interop, Mar. 2012.

[4] HICKEY, R. Rationale, Mar. 2012.

[5] WIKIPEDIA. Clojure — wikipedia, the free encyclopedia, 2012. [Online; accessed 13-March-2012].