Improving the Interoperability of Java and Clojure

Stephen Adams

University of Minnesota: Morris

April 14th, 2012

Clojure was released in 2007 by Rich Hickey. Clojure was designed with four features in mind:

Clojure was released in 2007 by Rich Hickey. Clojure was designed with four features in mind:

The Four Features of Clojure

- a LISP
- functional programming
- Symbiosis with an established platform
- Designed for concurrency

Introduction

- Compiles Java, and now Clojure code into Java bytecode
- Does just in time compilation

Prefix Notation

```
(+ 2 3)
=> 5
```

Prefix Notation

```
 (+ 2 3)
=> 5

 (+ 2 3 4)
=> 9
```

Clojure Data Structures

- Many of Clojure's data structures are just Java data structures; strings, characters, and all numbers are just Java types.

Clojure Data Structures

- Many of Clojure's data structures are just Java data structures; strings, characters, and all numbers are just Java types.
- Clojure provides its own collections.

Every Clojure collection is denoted by a different literal symbol pair.

Collection Literals

List	(1 2 3 4)
Vector	["apple" "banana" "orange"]
Set	#{67 2 8.8 -78 }
Hashmap	{ :name "Stephen Adams." :phone 555555555 }

Keywords

- Symbolic identifiers, denoted with a leading colon
- The colon is not part of the name
- Keywords evaluate to themselves
- Very fast equality tests

```
\{ :name "Stephen Adams." :phone 555555555 \}
```

Functions

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

Namespaces

```
(ns some.SampleNamespace
    (:import javax.swing.JFrame))
```

- Provide a symbolic identifier for pieces of Clojure code
- The “ns" symbol is a macro

Macros

Textual transformations that happen before evaluation.
Macro expands one piece of code to another.

Unless Macro

```
(ns macro.Example)

(defmacro unless [conditional caseA caseB]
   `(if (not ~conditional) ~caseA ~caseB))

    (unless false (println "Will print")
       (println "Will not print"))

(if (not false) (println "Will print")
   (println "Will not print"))
```

The functional features of Clojure

Functional programming primarily refers to two language features:

- First class functions
- Anonymous functions

Clojure supports both of these features.

First Class Functions
Passing functions to other functions

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

First Class Functions
Passing functions to other functions

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

(map square [1 2 3 4 5])
=> [1 4 9 16 25]
```

First Class Functions
Passing functions to other functions

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

(map square [1 2 3 4 5])
=> [1 4 9 16 25]

(reduce + [1 2 3 4 5])
=> 15
```

Anonymous Functions

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

Introduction to Java Interop

The idea for Clojure always involved interoperability with an existing language. Java was chosen for various reasons:

- Access to previously written Java libraries
- Already implemented, garbage collection and other memory & resource management tools.
- JVM is OS agnostic.

The dot special form

```
(. "fred" toUpperCase)
=> "FRED"
```

The dot special form

```
(. "fred" toUpperCase)
=> "FRED"

(.toUpperCase "fred")
```

The dot special form

```
(. "fred" toUpperCase)
=> "FRED"

(.toUpperCase "fred")

(. Math PI)
=> 3.141592653589793
```

The dot special form

```
(. "fred" toUpperCase)
=> "FRED"

(.toUpperCase "fred")

(. Math PI)
=> 3.141592653589793

(Math/PI)
=> 3.141592653589793
(Math/abs -2)
=> 2
```

Object Construction and modification

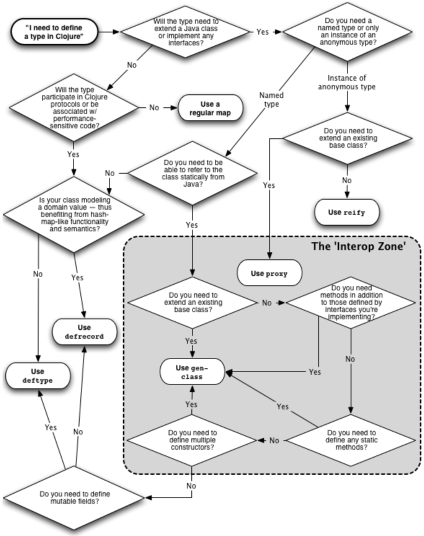
```
(new StringBuffer "fred")
=> #<StringBuffer fred>
```

Object Construction and modification

```
(new StringBuffer "fred")
=> #<StringBuffer fred>

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

Many ways of defining a type


"I need to define a type in Clojure"
Will the type need to extend a Java class or implement any interfaces?
Yes
Do you need a named type or only an instance of an anonymous type?
No
Will the type participate in Clojure protocols or be associated w/ performance-sensitive code?
No
Use a regular map
Named type
Instance of anonymous type
Do you need to extend an existing base class?
Yes
No
Use reify
Do you need to be able to refer to the class statically from Java?
No
Yes
Is your class modeling a domain value — thus benefiting from hash-map-like functionality and semantics?
The 'Interop Zone'
Use proxy
No
Yes
Do you need to extend an existing base class?
No
Do you need methods in addition to those defined by interfaces you're implementing?
Use defrecord
Yes
No
Use deftype
Use gen-class
Yes
Yes
Yes
Yes
No
Do you need to define multiple constructors?
No
Do you need to define any static methods?
No
Do you need to define mutable fields?

The Java interop zone

Two Clojure functions are designed to work with existing Java code.

The interop zone

- proxy
- gen-class

Proxy

- Must implement a Java interface or extend a Java class
- Creates a single instance of an anonymous Java class
- Cannot define methods not declared by a superclass or interface

Proxy cont.

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

Proxy cont.

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

(def test-inter
  (proxy [TestInterface] [] (square [x] (* x x))))
```

Proxy cont.

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

(def test-inter
  (proxy [TestInterface] [] (square [x] (* x x))))

(. test-inter square 5)
=> 25
```

Proxy cont.

```
(defn add-mousepressed-listener
     [component f & args]
      (let [listener (proxy [MouseAdapter] []
                     (mousePressed [event]
                        (apply f event args)))]
      (.addMouseListener component listener)
    listener))
```

Gen-class

Proxy will only allow you to do so much.

Gen-class

Proxy will only allow you to do so much.
There are cases when defining your own Java methods and objects is necessary.
E.G. working with a library that requires you to extend some object.

Gen-class cont.

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

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

Gen-class cont.

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

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


(def aClass (new some.Example))
(.toString aClass)
=> "Hello, world!"
```

Custom types in Clojure

```
(ns some.Example)

(gen-class
  :name  some.Example.classA
  :prefix classA- )

(gen-class
  :name some.Example.classB
  :prefix classB- )

(defn classA-toString
  [this]
  "I'm an A.")

(defn classB-toString
  [this]
  "I'm an B.")
```

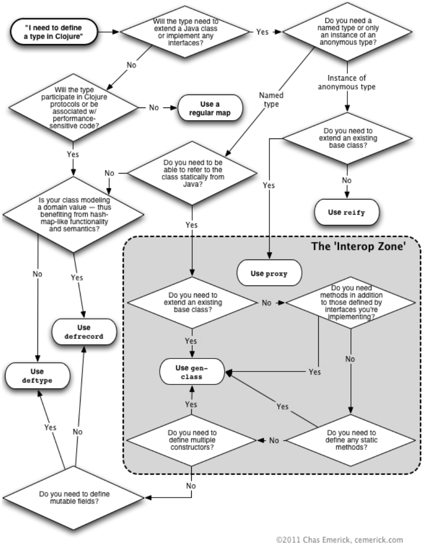
Java - Clojure Relationship

"Clojure does Java better than Java" - Stuart Halloway, at the Greater Atlanta Software Symposium, 2009.
Proxy is an example of this.

Java - Clojure Relationship

- You want to program Clojure in Clojure, not Java.
- Gen-class and, to some extent, proxy break from Clojure-like syntax.
- These functions should be used sparingly.
- C. Emerick's figure (http://bit.ly/liozRP).

Many ways of defining a type


"I need to define a type in Clojure"
Will the type need to extend a Java class or implement any interfaces?
Yes
Do you need a named type or only an instance of an anonymous type?
No
Will the type participate in Clojure protocols or be associated w/ performance-sensitive code?
No
Use a regular map
Named type
Instance of anonymous type
Do you need to extend an existing base class?
Yes
Do you need to be able to refer to the class statically from Java?
No
Yes
No
Use reify
Is your class modeling a domain value — thus benefiting from hash-map-like functionality and semantics?
Yes
The 'Interop Zone'
Use proxy
No
Yes
Do you need to extend an existing base class?
No
Do you need methods in addition to those defined by interfaces you're implementing?
Use defrecord
Yes
No
Use deftype
Use gen-class
Yes
Yes
Yes
Yes
No
Do you need to define multiple constructors?
No
Do you need to define any static methods?
No
Do you need to define mutable fields?
©2011 Chas Emerick, cemerick.com

Recomendations

- Push Clojure’s native abstractions into interop zone.
- Centralize documentation sources.
- Streamline IDE setup for beginners.

References

- CLOJUREDOCS.ORG. Mar. 2012. [Online; accessed March-2012].
- CLOJURE.ORG. Mar. 2012. [Online; accessed March-2012].
- Fogus, M., and Houser, C. The Joy of CLojure, Manning Publications.
- Halloway, S. Clojure-Java interop: A better java than java. QCon.