Improving the Interoperability of Java and Clojure

Stephen Adams

University of Minnesota: Morris

April 14th, 2012

A little reminder about Java

An object-oriented language released by Sun Microsystems in 1995.

- Compiled to Java bytecode
- Java bytecode is run on the Java Virtual Machine
- Second language in TIOBE programming Community Index (as of April 2012)

Clojure

The Clojure programming language was released in 2007 by the software developer Rich Hickey. Clojure was designed with four features in mind:

Clojure

The Clojure programming language was released in 2007 by the software developer Rich Hickey. Clojure was designed with four features in mind:

The Four Features of Clojure

- A Lisp
- Functional programming
- Symbiosis with an established platform (Java)
- Designed for concurrency

LISP and Functional Programming

- Lisp was developed in 1958
- The earliest functional language
- Emphasizes the application of functions
- Imperative programming emphasizes changes in state

Functional Programming and concurrency

- Functional languages have immutable data by default
- A function will always return the same results given the same arguments
- Easier to predict behavior of a program in a concurrent environment

Who is this talk for?

- Need to parallelize your work?
- Are you tired of Java?

Who is this talk for?

- Need to parallelize your work?
- Are you tired of Java?
- Have a project already written in Java?
- Have a Java library you use a lot?

Who is this talk is for?

"Clojure does Java better than Java" - Stuart Halloway, at the Greater Atlanta Software Symposium, 2009.

Prefix Notation

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

Prefix Notation

```
(+ 2 3)
=> 5

(+ 2 3 4)
=> 9

(inc 4)
=> 5
```

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"]
Hashmap	{ :name "Stephen Adams" :phone 555555555 }

- Keywords are symbolic identifiers, denoted with a leading colon.
- Provide very fast equality tests

Variables and Functions

```
(def vect [1 2 3 4 5])

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

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

- Provides basic access to Java fields and methods
- Can also be read as “in the scope of."

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

The dot special form

- Provides basic access to Java fields and methods
- Can also be read as “in the scope of."

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

"fred".toUpperCase();
```

The dot special form cont.

```
(.toUpperCase "fred")
```

The dot special form cont.

```
(.toUpperCase "fred")
```

Accessing static methods and fields

```
(. 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

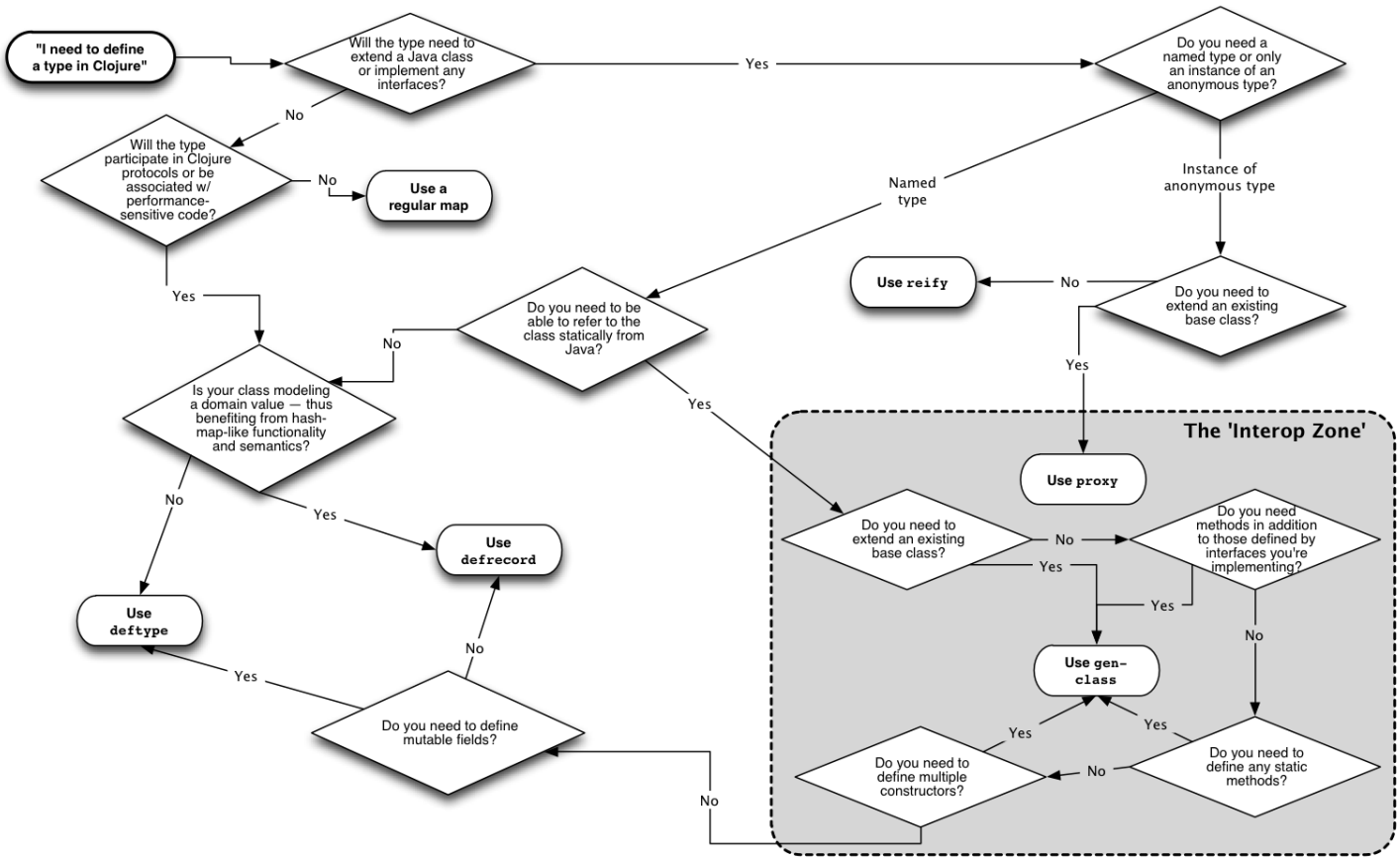
```
(new StringBuffer "fred")
=> #<StringBuffer fred>

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

Object Construction and modification cont.

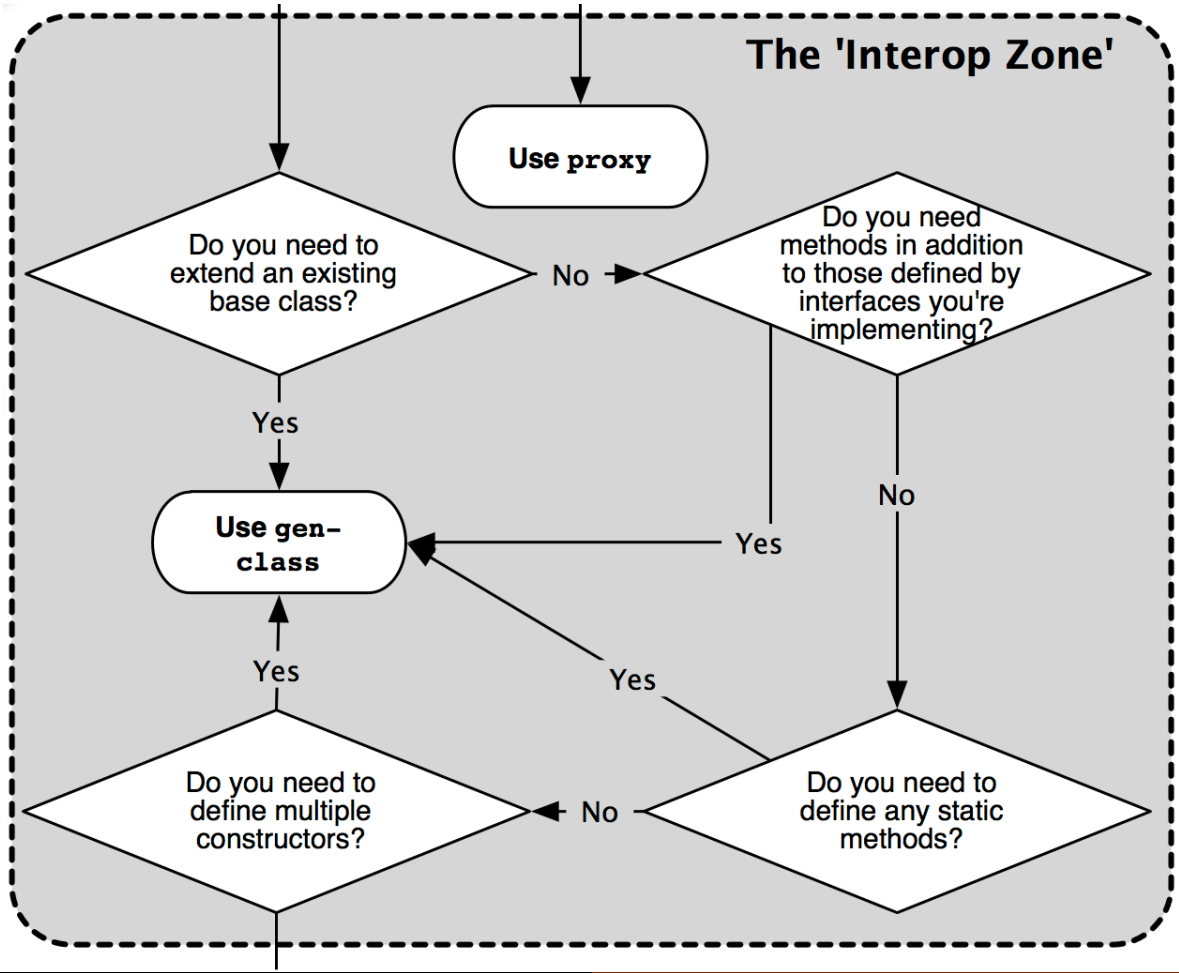
```
public class ExampleBuffer {
    public static void main(String[] args){
        StringBuffer buff = new StringBuffer("fred");
        buff.setCharAt(0, "F");
        buff.append(" is a nice guy");
    }
}
```

Many ways of defining a type

By Chas Emerick, from: http://bit.ly/liozRP



Many ways of defining a type



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.

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

Gen-class

- Proxy will only allow you to do so much because you can only overload functions defined by a super class or interface.
- Proxy also wont create a named type.

Gen-class

- Proxy will only allow you to do so much because you can only overload functions defined by a super class or interface.
- Proxy also wont create a named type.
- Without a named type Java is unable to call Clojure code.

Gen-class cont.

```
  (gen-class
      :name    Example
      :prefix "example-"))

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

Gen-class cont.

```
  (gen-class
      :name   Example
      :prefix "example-"))

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


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

Wrapping up the interop zone

- Proxy creates a single instance of an anonymous class
- A class defined by proxy is not visable to external files
- Gen-class creates a named type which Java will be able to see

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/IiozRP).

Recommendations

- Push Clojure's native abstractions into interop zone, possibly macros.
- Centralize documentation sources.
- Streamline IDE setup for beginners.

Why this matters

"The killer app for Clojure is the JVM itself. Everyone's fed up with the Java language, but understandably we don't want to abandon our investment in the Java Virtual Machine and its capabilities: the libraries, the configuration, the monitoring, and all the other entirely valid reasons we still use it."
Steve Yegge, in the foreward to Joy of Clojure.

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.