Clojure-Java Interop

Stuart Halloway
stu@clojure.com
@stuarthalloway

Copyright 2007-2010 Relevance, Inc. This presentation is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States License. See http://creativecommons.org/licenses/by-nc-sa/3.0/us/
the plan

syntax

plumbing

Clojure calling Java
objects, Clojure style
Java calling Clojure

design principles

simplicity
direct access
bonus example
syntax
<table>
<thead>
<tr>
<th>type</th>
<th>example</th>
<th>java equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>&quot;foo&quot;</td>
<td>String</td>
</tr>
<tr>
<td>character</td>
<td>\f</td>
<td>Character</td>
</tr>
<tr>
<td>regex</td>
<td>#&quot;fo*&quot;</td>
<td>Pattern</td>
</tr>
<tr>
<td>integer</td>
<td>42</td>
<td>Long</td>
</tr>
<tr>
<td>a.r. integer</td>
<td>42N</td>
<td>BigInteger</td>
</tr>
<tr>
<td>double</td>
<td>3.14159</td>
<td>Double</td>
</tr>
<tr>
<td>a.p. double</td>
<td>3.14159M</td>
<td>BigDecimal</td>
</tr>
<tr>
<td>boolean</td>
<td>true</td>
<td>Boolean</td>
</tr>
<tr>
<td>nil</td>
<td>nil</td>
<td>null</td>
</tr>
<tr>
<td>ratio</td>
<td>22/7</td>
<td>N/A</td>
</tr>
<tr>
<td>symbol</td>
<td>foo, +</td>
<td>N/A</td>
</tr>
<tr>
<td>keyword</td>
<td>:foo, ::foo</td>
<td>N/A</td>
</tr>
</tbody>
</table>
## data literals

<table>
<thead>
<tr>
<th>type</th>
<th>properties</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>singly-linked, insert at front</td>
<td>(1 2 3)</td>
</tr>
<tr>
<td>vector</td>
<td>indexed, insert at rear</td>
<td>[1 2 3]</td>
</tr>
<tr>
<td>map</td>
<td>key/value</td>
<td>{:a 100 :b 90}</td>
</tr>
<tr>
<td>set</td>
<td>key</td>
<td>#{:a :b}</td>
</tr>
</tbody>
</table>
function call

semantics:

fn call

arg

(symbol

list

string

Hello World)
defn greet "Returns a friendly greeting" [your-name]
    (str "Hello, " your-name))
it's all data

(symbol symbol)

(definition greet)

"Returns a friendly greeting"

[your-name]

(str "Hello, " your-name))

(vector list string)
metadata

(prefix with ^

(class name or arbitrary map

(defn ^{String} greet
    "Returns a friendly greeting"
    [your-name]
    (str "Hello, " your-name))

onsdag, 2010 november 03
Clojure calling Java
<table>
<thead>
<tr>
<th>java</th>
<th>new Widget(&quot;foo&quot;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>clojure sugar</td>
<td>(Widget. &quot;red&quot;)</td>
</tr>
</tbody>
</table>
access static members

<table>
<thead>
<tr>
<th>java</th>
<th>Math.PI</th>
</tr>
</thead>
<tbody>
<tr>
<td>clojure sugar</td>
<td>Math/PI</td>
</tr>
<tr>
<td>java</td>
<td><code>rnd.nextInt()</code></td>
</tr>
<tr>
<td>------</td>
<td>-----------------</td>
</tr>
<tr>
<td>clojure sugar</td>
<td><code>.nextInt rnd)</code></td>
</tr>
</tbody>
</table>
## chaining access

<table>
<thead>
<tr>
<th>java</th>
<th>person.getAddress().getZipCode()</th>
</tr>
</thead>
<tbody>
<tr>
<td>clojure sugar</td>
<td>(.. person setAddress getZipCode)</td>
</tr>
</tbody>
</table>
(doto
  ((JFrame. "Foobar")
   (.add (proxy [JPanel] []))
   (.setSize 640 400)
   (.setVisible true)))
# parenthsis count

<table>
<thead>
<tr>
<th></th>
<th>()</th>
<th>()</th>
<th>()</th>
<th>()</th>
</tr>
</thead>
<tbody>
<tr>
<td>java</td>
<td>( )</td>
<td>( )</td>
<td>( )</td>
<td>( )</td>
</tr>
<tr>
<td>clojure</td>
<td>( )</td>
<td>( )</td>
<td>( )</td>
<td>( )</td>
</tr>
</tbody>
</table>

onsdag, 2010 november 03
example:
refactor apache commons isBlank
public class StringUtils {
    public static boolean isBlank(String str) {
        int strLen;
        if (str == null || (strLen = str.length()) == 0) {
            return true;
        }
        for (int i = 0; i < strLen; i++) {
            if ((Character.isWhitespace(str.charAt(i)) == false)) {
                return false;
            }
        }
        return true;
    }
}

initial implementation
```java
public class StringUtils {
    public isBlank(str) {
        if (str == null || (strLen = str.length()) == 0) {
            return true;
        }
        for (i = 0; i < strLen; i++) {
            if (((Character.isWhitespace(str.charAt(i)) == false)) {
                return false;
            }
        }
        return true;
    }
}
```
public isBlank(str) {
    if (str == null || (strLen = str.length()) == 0) {
        return true;
    }
    for (i = 0; i < strLen; i++) {
        if ((Character.isWhitespace(str.charAt(i)) == false)) {
            return false;
        }
    }
    return true;
}
public isBlank(String str) {
    if (str == null || (strLength = str.length()) == 0) {
        return true;
    }
    for (char ch : str) {
        if (!Character.isWhitespace(ch)) {
            return false;
        }
    }
    return true;
}
- corner cases

```java
public isBlank(str) {
    every (ch in str) {
        Character.isWhitespace(ch);
    }
}
```
(defn blank? [s]
  (every? #(Character/isWhitespace %) s))
objects, Clojure style
records
(defrecord Foo [a b c])
-> user.Foo

defrecord named type
(with slots
(def f (Foo 1 2 3))
-> #'user/f

positional
constructor
(:b f)
-> 2

keyword access
(class f)
-> user.Foo

plain ol' class
(supers (class f))
-> #{clojure.lang.IObj clojure.lang.IKeywordLookup java.util.Map
clojure.lang.IPersistentMap clojure.lang.IMeta java.lang.Object
clojure.lang.Iterable clojure.lang.ILookup clojure.lang.Seqable
clojure.lang.Counted clojure.lang.IPersistentCollection
clojure.lang.Associative}
from maps...

```
(def stu {:fname "Stu" 
  :lname "Halloway" 
  :address {:street "200 N Mangum" 
    :city "Durham" 
    :state "NC" 
    :zip 27701}})
```

```
(:lname stu)  => "Halloway"
```

```
(assoc stu :fname "Stuart")
=> {:fname "Stuart", :lname "Halloway", 
    :address ...}
```

```
(update-in stu [:address :zip] inc)
=> {:address {:street "200 N Mangum", 
    :zip 27702 ...} ...}
```
...to records!

(defrecord Person [fname lname address])
(defrecord Address [street city state zip])
(def stu (Person. "Stu" "Halloway"
                   (Address. "200 N Mangum"
                             "Durham"
                             "NC"
                             27701)))

(:lname stu)
=> "Halloway"

(-> stu :address :city)
=> "Durham"

(assoc stu :fname "Stuart")
=> :user.Person{:fname "Stuart", :lname"Halloway",
                 :address ...}

(update-in stu [:address :zip] inc)
=> :user.Person{:address {:street "200 N Mangum",
                          :zip 27702 ...} ...}
protocols
defprotocol

(defprotocol Coercions
  "Coerce between various 'resource-namish' things."
  (as-file [x] "Coerce argument to a file.")
  (as-url [x] "Coerce argument to a URL.")
)

named set of generic functions
polymorphic on type of first argument
defines fns in same namespace as protocol
(defrecord Name [n]
  Coercions
  (as-file [_] (File. n)))

(def n (Name. "johndoe"))

(as-file n)
-> #<File johndoe>

not a good design choice in this case, better approach follows on next slide...
extend protocol to types...

(as-file "data.txt")
-> java.lang.IllegalArgumentException

(extend-protocol Coercions
 nil
 (as-file [ ] nil)
 (as-url [ ] nil)

String
 (as-file [s] (File. s))
 (as-url [s] (URL. s))

(as-file "data.txt")
-> <File data.txt>
...or type to protocols

(extend-type String Coercions
  (as-file [s] (File. s))
  (as-url [s] (URL. s))

EqualityPartition
  (equality-partition [x] :atom)
extending a protocol

inline

extend protocol to multiple types
extend type to multiple protocols
build directly from fns and maps

extension happens in the protocol fns, not in the types
(let [x 42]
  (reify AProtocol
    (bar [this b] "reify bar")
    (baz [this ] (str "reify baz " x))))
(baz r))

=> "reify baz 42"

instantiate an unnamed type
implement 0 or more protocols or interfaces
closes over environment like fn
Java calling Clojure
(compile 'examples.clojure)

(defproject clojure.examples "0.0.1"
  :aot [examples.contacts]
  #"...more maveny stuff")

from a build tool/IDE
(e.g. maven, leiningen, gradle, Eclipse, NetBeans, IDEA, emacs, vim)
Clojure data from Java

Address addr = new Address("200 N Mangum",
    "Durham", "NC", 27701);
Person person = new Person("Stuart", "Halloway",
    addr);

System.out.println("As person " + person.fname);

System.out.println("As map");
for (Iterator it = person.keySet().iterator();
    it.hasNext();) {
    Object key = it.next();
    System.out.println(key + ": " + person.get(key));
}
gen-class

(ns examples.tasklist
   (:gen-class
      :extends org.xml.sax.helpers.DefaultHandler
      :init init
      :state state))
mapping gen-class methods

(defn -init []
    [[] (atom [])])

(defn -startElement
    [this uri local qname atts]
    (when (= qname "target")
        (swap! (.state this)
            conj (.getValue atts "name"))))

(defn -main [& args]
    (doseq [arg args]
        (println (task-list arg))))
gen-class: for interop only

gen-class meets Java in ugly places
prefer defprotocol, defrecord, reify, deftype
previous slides were examples only
xml doesn’t have to be that tedious!
(defn tasklist
  [resource]
  (-> (read-html resource)
       (select [:target])
       (attr :name)))

clojure data

xml done better

declarative, standards based query
(courtesy cgrand’s enlive)
where are we

Clojure can call Java
Java can call Clojure
Clojure makes things better

so what?
simplicity
define simple
keeping count?
“if I wanted to deploy a simple web app ... how many lines of text do I have to deal with? Most importantly, how many of those are program text, and how many are framework boilerplate? Finally, how many tools do I have to use to get it deployed?”

simple things should be simple thread
Clojure google group
simple
still simple
complex
beginner-friendly?
“The most important gauge of any programming language is how easily a novice programmer can maintain a significant program in any problem domain.”

http://www.oreillynet.com/onlamp/blog/2006/03/the_worlds_most_maintainable_p.html
“Simplicity: Should be fairly simple to read, learn and understand. Python is a good example. PHP is a great example (at a simpler class of problems). C++ and Scala not good examples.”

familiar syntax?
“Constructs that are natural to humans not mathematics : This is actually a sub point to Simplicity. The constructs should be consistent with the normal average non mathematically trained brains.”
minimal?
“If the semantic model of a language is clear - if its rules are simple and precise - the language will be more productive to use. Less is more.”
uncomplicated?
A TYPICAL APPLE PRODUCT

A GOOGLE PRODUCT

YOUR COMPANY'S APP.

http://twittch.com/26/
valuable, but not simple

convenient
newb-friendly
easy
familiar syntax
minimal
uncomplicated
hopelessly subjective?
“I’m coming to the conclusion that one man’s simple is another man’s complex. Fowler evidently finds it easier to read many small methods than a few larger ones; I find the opposite.”

http://reprog.wordpress.com/2010/03/28/what-is-simplicity-in-programming/
don’t give up

simple has an objective definition

simplicity informs design

simplicity is fundamental
simple: not compound
Java delivers compounds
objects provide...
methods

Object
namespaces
uncontrolled mutation

Uncontrolled Mutation

Interface

Interface

Base Class

struct

xxxx

yyyy

zzzz
pojo is an oxymoron
Java* style:
compound, complex, complicated

features are delivered as compounds
compounds contain more than you need
snowballs from language into libs
Clojure delivers simples & composites
polymorphism

identity

structure

namespaces

struct
xxxx
yyyy
zzzz

Vars

values

functions

generic data access

foundaton
superstructure

polymorphism

types

structure

namespaces

identity

perception

values

functions

generic data access

struct
xxx
yyy
zzzz

V

vars
examples
polymorphism

types

structure

namespaces

identity

perception

values

functions

generic data access

polymorphism a la carte
Coercions

(defprotocol Coercions
  "Coerce between various 'resource-namish' things."
  (as-file [x] "Coerce argument to a file."
  (as-url [x] "Coerce argument to a URL.")

(extend-protocol Coercions
  nil
  (as-file [_] nil)
  (as-url [_] nil)

String
  (as-file [s] (File. s))
  (as-url [s] (URL. s))
extension options revisited

inline
extend types to protocol
extend protocol to types
build directly from fns and maps
implementation

reuse
(extend BufferedInputStream IOFactory

(assoc default-streams-impl

  :make-input-stream (fn [x opts] x)
  :make-reader inputstream->reader))

no special mixin
API: use generic
collection fns

plain ol’
keywords and fns
mere maps

;;; elided from clojure.java.io
(def default-streams-impl
  {:make-reader (fn [x opts] ...)
   :make-writer (fn [x opts] ...)
   :make-input-stream (fn [x opts] ...)
   :make-output-stream (fn [x opts] ...)})

make a map of names -> fns...
simple syntax:
singular
interpretation
<table>
<thead>
<tr>
<th>form</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>function</td>
<td><code>(println &quot;hello&quot;)</code></td>
</tr>
<tr>
<td>operator</td>
<td><code>( + 1 2)</code></td>
</tr>
<tr>
<td>method call</td>
<td><code>(.trim &quot; hello &quot;)</code></td>
</tr>
<tr>
<td>import</td>
<td><code>(require 'mylib)</code></td>
</tr>
<tr>
<td>metadata</td>
<td><code>(with-meta obj m)</code></td>
</tr>
<tr>
<td>control flow</td>
<td><code>(when valid? (proceed))</code></td>
</tr>
<tr>
<td>scope</td>
<td><code>(dosync (alter ...))</code></td>
</tr>
</tbody>
</table>
# Bindings

<table>
<thead>
<tr>
<th>form</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>fn</td>
<td><code>(fn [x] (* x 2))</code></td>
</tr>
<tr>
<td>named fn</td>
<td><code>(defn hi [s] (str &quot;hi, &quot; s))</code></td>
</tr>
<tr>
<td>lexical bind</td>
<td><code>(let [[x y] (range) ...])</code></td>
</tr>
<tr>
<td>dynamic bind</td>
<td><code>(binding [answer 42] ...)</code></td>
</tr>
<tr>
<td>comprehension</td>
<td><code>(for [x (range) ...] ...)</code></td>
</tr>
<tr>
<td>side effects</td>
<td><code>(doseq [{m :msg} warnings]...)</code></td>
</tr>
</tbody>
</table>
polymorphism

generic data access

types

structure

namespaces

identity

perception

values

functions

generic data access

generic data access
some data

lunch-companions
-> ({{:fname "Neal", :lname "Ford"}
    {{:fname "Stu", :lname "Halloway"}
    {{:fname "Dan", :lname "North"}})}
“getter” function

```
(defn last-name [x]
  (get x :last-name))
```

fn name

arg list (vector)

body
pass fn to fn

invoke this fn

(fn arg)
(data arg)

(sort-by
   first-name
   lunch-companions)
-> ({:fname "Dan", :lname "North"}
   {:fname "Neal", :lname "Ford"}
   {:fname "Stu", :lname "Halloway"})
anonymous fn

\[(\text{sort-by} (\text{fn} [n] (\text{get} n :\text{fname})) \text{lunch-companions}))\]
anonymous #()
maps are functions

(map is fn!

(sort-by
  #(\% :fname)
  lunch-companions)
keywords are functions

(keyword is fn!
(sort-by 
  #( :fname %
      lunch-companions))
beautiful

(sort-by :fname lunch-companions)
direct access to the jvm
living on the jvm

performance

power

reach
jvm tradeoffs

primitive operations

collection conformance

absence of tco

oo interop (genclass, proxy)

alternate numeric ops
(def nums (make-array Integer/TYP 10))
-> #'user/nums

(aset nums 4 1000)
-> 1000

(aget nums 4)
-> 1000

(seq nums)
-> (0 0 0 0 1000 0 0 0 0 0 0)
converting to arrays

(def nums (range 3))
(defn member-type [arr]
  (-> arr class (.getComponentType)))

(-> (to-array nums) member-type)
  -> java.lang.Object

(-> (into-array nums) member-type)
  -> java.lang.Integer

(-> (into-array Comparable nums) member-type)
  -> java.lang.Comparable
unboxed math (1.3+)

(defn fib [n]
  (if (<= n 1)
    1
    (+ (fib (dec n)) (fib (- n 2)))))

(time (fib 38))
"Elapsed time: 3565.579 msecs"

;; hint arg and return
(defn fib ^long [^long n]
  (if (<= n 1)
    1
    (+ (fib (dec n)) (fib (- n 2)))))

(time (fib 38))
"Elapsed time: 395.365 msecs"
(defn zipm [keys vals]
  (loop [m {}]
    ks (seq keys)
    vs (seq vals)]
  (if (and ks vs)
    (recur (assoc m (first ks) (first vs))
      (next ks)
      (next vs))
    m)))

(zipm [:a :b :c] [1 2 3]) => {:a 1, :b 2, :c 3}
loops are rare

(loop [m {}]
  [k & ks :as keys] (seq keys)
  [v & vs :as vals] (seq vals)]
  (if (and keys vals)
      (recur (assoc m k v) ks vs)
      m))

;reduce with adder fn
(reduce (fn [m [k v]] (assoc m k v))
  {[} (map vector keys vals))

;apply data constructor fn
(apply hash-map (interleave keys vals))

;map into empty (or not!) structure
(into {} (map vector keys vals))

;get lucky
(zipmap keys vals)
Annotations

```clojure
(deftype ^{:Deprecated true}
  Foo
  [
    ^{:Retention RetentionPolicy/RUNTIME} a b])
```

plain ol’ Clojure metadata

with keys and values as required by annotation
### numeric options

**Arbitrary**
**Fast**
**Limited**
**No**
**Not Applicable**
**Blow**
**Very Fast**
**Yes**
* (caller choose one)

<table>
<thead>
<tr>
<th></th>
<th>safety?</th>
<th>speed</th>
<th>canonical?</th>
<th>precision</th>
<th>contagion?</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>java</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>default</td>
<td>N</td>
<td>VF</td>
<td>N</td>
<td>L</td>
<td>N</td>
</tr>
<tr>
<td>big</td>
<td>Y</td>
<td></td>
<td>S</td>
<td>N/A</td>
<td>A</td>
</tr>
<tr>
<td><strong>clojure</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>default</td>
<td>Y</td>
<td>F*</td>
<td>Y</td>
<td>L</td>
<td>Y*</td>
</tr>
<tr>
<td>big</td>
<td>Y</td>
<td></td>
<td>S</td>
<td>N/A</td>
<td>A</td>
</tr>
<tr>
<td>unchecked</td>
<td>N</td>
<td>VF</td>
<td>Y</td>
<td>L</td>
<td>N</td>
</tr>
</tbody>
</table>
example:
reflection
sure wish we could...

(reflect "Foo")
(reflect [1 2 3])
(reflect FileInputStream)
(show "foo" #"last")

=== public final java.lang.String ===

[56] lastIndexOf : int (String)
[57] lastIndexOf : int (String,int)
[58] lastIndexOf : int (int)
[59] lastIndexOf : int (int,int)
java reflection is compound

query
describe
invoke
class in hand
class in context
approach

1. single api

2. generic wiring

reflect

Reflector

3. generic predicates

(predicates) (clojure.data.match TBD)

4. generic printing

print-table
usage

```java
(->> (reflect java.lang.String)
  :members
  (filter #(.startsWith (str (:name %)) "last"))
  (print-table))
```

<table>
<thead>
<tr>
<th>:name</th>
<th>:return-type</th>
<th>:declaring-class</th>
<th>:parameter-types</th>
<th>:exception-types</th>
<th>:flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>lastIndexOf</td>
<td>int</td>
<td>java.lang.String</td>
<td>[java.lang.String]</td>
<td>[]</td>
<td>#{:public}</td>
</tr>
<tr>
<td>lastIndexOf</td>
<td>int</td>
<td>java.lang.String</td>
<td>[char&lt;&gt; int int char&lt;&gt; int int int]</td>
<td>[]</td>
<td>#{:static}</td>
</tr>
<tr>
<td>lastIndexOf</td>
<td>int</td>
<td>java.lang.String</td>
<td>[int]</td>
<td>[]</td>
<td>#{:public}</td>
</tr>
<tr>
<td>lastIndexOf</td>
<td>int</td>
<td>java.lang.String</td>
<td>[java.lang.String int]</td>
<td>[]</td>
<td>#{:public}</td>
</tr>
<tr>
<td>lastIndexOf</td>
<td>int</td>
<td>java.lang.String</td>
<td>[int int]</td>
<td>[]</td>
<td>#{:public}</td>
</tr>
</tbody>
</table>
```java
// from java.lang.reflect.Modifier
public static final int PUBLIC           = 0x00000001;
public static final int PRIVATE          = 0x00000002;
public static final int PROTECTED        = 0x00000004;
public static final int STATIC           = 0x00000008;
public static final int FINAL            = 0x00000010;
public static final int SYNCHRONIZED     = 0x00000020;
public static final int VOLATILE         = 0x00000040;
public static final int TRANSIENT        = 0x00000080;
```
problems

classes are poor containers
no data literals to use instead
modifiers, types add no value

_data doesn’t capture the model_
(defn- access-flag
   [[name flag & contexts]]
   {:name name
    :flag flag
    :contexts (set (map keyword contexts))})

(def flag-descriptors
   (vec
    (map access-flag
       [[:public 0x0001 :class :field :method]
        [:private 0x002 :class :field :method]
        [:protected 0x0004 :class :field :method]
        [:static 0x0008 :field :method]
        [:final 0x0010 :class :field :method]
        [:synchronized 0x0020 :method]
        [:volatile 0x0040 :field]
        [:bridge 0x0040 :method]])))
irrelevant destructions

field
method
constructor
member
modifier
(defprotocol Reflector
  "Protocol for reflection implementers."
  (do-reflect [reflector typeref]))

(defprotocol TypeReference
  "A TypeReference can be unambiguously converted to a type name on the host platform."
  (typename [o]))
abstractions != api

(defn reflect
  [obj & options]
  (apply type-reflect
    (if (class? obj) obj (class obj)) options))
information ~ maps

```java
(->> (reflect java.lang.String)
    :members
    (filter #(.startsWith (str (:name %)) "last"))
    pprint)

{:name lastIndexOf,
 :return-type int,
 :declaring-class java.lang.String,
 :parameter-types [java.lang.String],
 :exception-types [],
 :flags #{:public}}
{:name lastIndexOf,
 :return-type int,
 :declaring-class java.lang.String,
 :parameter-types [char<> int int char<> int int int],
 :exception-types [],
 :flags #{:static}}
...)
```
Clojure fits if you value

simplicity and directness

over

convenience and familiarity
thanks!

http://clojure.org