Clojure: Functional Concurrency for the JVM

Howard M. Lewis Ship
Director of Open Source Technology
Formos Software Development

howard.lewis.ship@formos.com
Lisp is over half a century old and it still has this perfect, timeless aura about it.

I wonder if the cycles will continue forever.

A few coders from each new generation re-discovering the Lisp arts.

These are your father’s parentheses.

Elegant weapons for a more... civilized age.

http://xkcd.com/297/

Clojure: The Language
Rich Hickey

Chunked Sequences

Leveraging the structure of data when sequencing

Rich Hickey
Code is Data

```(1 2 3)```

Quoted list of numbers

```(biggest 5 42)```

Function call

```(defn biggest
  "Find the maximum of two numbers"
  [x y]
  (if (> x y) x y))```
Read Eval Print Loop

```clojure
user=> (defn biggest 
    "Find the maximum of two numbers" 
    [x y] 
    (if (> x y) x y))
#=(var user/biggest)
user=> (biggest 5 42)
42
user=> (doc biggest)
-------------------------------
user/biggest
[[x y]]
  Find the maximum of two numbers
nil
user=> '(1 2 3)
(1 2 3)
user=> '(biggest 5 42)
(biggest 5 42)
user=> (first '(biggest 5 42))
biggest
user=> (eval '(biggest 5 42))
42
```
There Is No Interpreter
Clojure Literals

user=> 42
42
user=> "A Clojure String"
"A Clojure String"
user=> nil
nil
user=> :balance
:balance
user=> true
true
user=> false
false
Clojure Literals

user=> 5
5
user=> 5.001
5.001
user=> 22/7
22/7
user=> (* 2 22/7)
44/7
user=> (* 100000 100000 100000)
1000000000000000000
user=> (+ 5. 0.000000000000000001)
5.0
user=> (+ 5.0M 0.000000000000000001M)
5.000000000000000001M
Java Interop

```java
factory.setNamespaceAware(true) // (.setNamespaceAware factory true)

new StringBuffer() // (new StringBuffer)

(new StringBuffer) // (StringBuffer.)

factory.newSAXParser().parse(src, handler) // (. factory newSAXParser (parse src handler))

myObject.ivar = "foo"; // (set! (. myObject ivar) "foo")
```
Java Interop

```java
frame = new JFrame();
frame.add(panel, BorderLayout.CENTER);
frame.add(greetButton, BorderLayout.SOUTH);
frame.pack();
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setVisible(true);
```

(doto (JFrame.)
 (.add panel BorderLayout/CENTER)
 (.add greet-button BorderLayout/SOUTH)
 (.pack)
 (.setDefaultCloseOperation JFrame/EXIT_ON_CLOSE)
 (.setVisible true))

© 2009 Formos Software Development
Clojure Collections: Lists

user=> (def lst `(1 2 3))
#=(var user/lst)
user=> lst
(1 2 3)
user=> (first lst)
1
user=> (rest lst)
(2 3)
user=> (conj lst 4)
(4 1 2 3)
user=> (cons 4 lst)
(4 1 2 3)
Clojure Collections: Vectors

```
user=> (def v [:moe :larry :curly])
#=(var user/v)
user=> v
[:moe :larry :curly]
user=> (first v)
:moe
user=> (rest v)
[:larry :curly]
user=> (conj v :shemp)
[:moe :larry :curly :shemp]
user=> (cons :shemp v)
[:shemp :moe :larry :curly]
user=> v
[:moe :larry :curly]
user=> (v 1)
:larry
```

vector is a function of its indexes
Clojure Collections: Map

```
user=> (def m {:first-name "Howard" :last-name "Lewis Ship"})
#=(var user/m)
user=> m
{:last-name "Lewis Ship", :first-name "Howard"}
user=> (get m :last-name)
"Lewis Ship"
user=> (m :last-name)
"Lewis Ship"
user=> (assoc m :company "Formos")
{:company "Formos", :last-name "Lewis Ship", :first-name "Howard"}
user=> m
{:last-name "Lewis Ship", :first-name "Howard"}
user=> (:first-name m)
"Howard"
user=> (:ssn m)
nil
```

map is a function of its keys

Keywords are functions, too!
Clojure Collections: Sets

```clojure
defn s #{"Howard" "Suzanne" "Molly" "Jim"}
#=(var user/s)
user=> s
#{"Howard" "Jim" "Molly" "Suzanne"}
user=> (contains? s "Howard")
true
user=> (contains? s "howard")
false
user=> (s "Howard")
"Howard"
user=> (s "Rhys")
nil
user=> (conj s "Howard")
#{"Howard" "Jim" "Molly" "Suzanne"}
user=> (conj s "Scott")
#{"Howard" "Jim" "Molly" "Suzanne" "Scott"}
```
“For alumni of other languages, beginning to use Lisp may be like stepping onto a skating rink for the first time. It’s actually much easier to get around on ice than it is on dry land—if you use skates. Till then you will be left wondering what people see in this sport.”

Paul Graham
Functional Programming

Computer Holy Wars

HOLD IT RIGHT THERE, BUDDY.

That scruffy beard... those suspenders... that smug expression...

You're one of those condescending functional programmers.

Here's a nickel, kid. Get yourself a better paradigm.
No Mutable State
No Side Effects
First Class Functions
Functional Composition
Functional Programming in Java

```java
public void saveOrUpdate(final Employee employee)
{
  HibernateCallback callback = new HibernateCallback()
  {
    public Object doInHibernate(Session session)
      throws HibernateException, SQLException
    {
      session.saveOrUpdate(employee);
      return null;
    }
  };

  hibernateTemplate.execute(callback);
}
```

```java
SwingUtilities.invokeLater(new Runnable()
{  
  public void run()
  {  
    progressBar.setValue(progressBar.getValue() + 1);
  }
});
```

Outer function controls the context:
- Thread
- Exception handling
- Parameters
public interface Predicate<T> {
  boolean accept(T value);
}

public static <T> Collection<T> filter(Predicate<T> pred, Collection<T> coll) {
  Collection<T> out = new ArrayList<T>();
  for (T item : coll) {
    if (pred.accept(item))
      out.add(item);
  }
  return out;
}

return CollectionUtils.filter(new Predicate<String>()
{
  public boolean accept(String value)
  {
    return !value.startsWith(".");
  }
}, names);
Functional Clojure Collections

```
user=> (def names ["fred" "barney" ".hidden" "wilma"])
#=(var user/names)
user=> (filter #(not (.startsWith % ".")) names)
("fred" "barney" "wilma")
user=> (remove #(.startsWith % ".") names)
("fred" "barney" "wilma")
user=>
```
First Class Functions

(filter #(not (.startsWith % "")) names)

(defun require-extension [ext]
  (fn [file-name]
    (= ext (last (split-string file-name "."))))

(defun filter-by-extension [ext coll]
  (filter (require-extension ext) coll))

function as parameter to function

function as return value

composing functions
Life without the for loop

```java
public static int sum(int[] vals)
{
    int total = 0;
    for (int val : vals) total += val;
    return total;
}
```

```clojure
(defn sum
  [col]
  (reduce + 0 col))
```

```
0 + col[0] + col[1] + col[2] ...
```
Life without the for loop

```java
public static String[] formatDoubles(double[] inputs) {
    String[] output = new String[inputs.length];
    for (int i = 0; i < input.length; i++)
        output[i] = String.format("%9.2f", inputs[i]);
    return output;
}
```

```
(defn format-doubles [col]
  (map #(format "%9.2f" %) col))
```

Apply function to each item, forming new seq

```clojure
user=> (format-doubles '(2.5 3.7 -22.7))
("     2.50" "     3.70" "   -22.70")
```
for: list comprehension

```clojure
user=> (range 0 5)
(0 1 2 3 4)
user=> (for [x (range 0 10) :when (even? x)]
    x)
(0 2 4 6 8)
user=>
```

```clojure
user=> (for [suit [:heart :spade :diamond :club]
    rank [:ace 2 3 4]]
    [suit rank])
[:club :ace] [:club 2] [:club 3] [:club 4])
user=>
```

```clojure
user=> (for [x (range 1 5)
    y (range 0 x)]
    [x y])
[[1 0]
 [2 0] [2 1]
 [3 0] [3 1] [3 2]
 [4 0] [4 1] [4 2] [4 3]]
user=>
```
Laziness is a Virtue

user=> (take 20 (iterate inc 1))
(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20)
user=> (take 20 (map * (iterate inc 1) (iterate inc 1)))
(1 4 9 16 25 36 49 64 81 100 121 144 169 196 225 256 289 324 361 400)
Laziness
Java: Data Encapsulated in Objects

```java
public double averageAge(Collection<Person> persons) {
    double total = 0.0;
    for (Person p : persons)
        total += p.getAge();
    return total / persons.size();
}
```
user=> persons
[{:first-name "Howard", :last-name "Lewis Ship", :age 42}
 {:first-name "Scott", :last-name "Simon", :age 44}
 {:first-name "Molly", :last-name "Newman", :age 29}]
user=> (map :age persons)
(42 44 29)
user=> (apply + (map :age persons))
115
user=>

(defn avg-age
  [coll]
  (/ (apply + (map :age coll)) (count coll)))

(defn avg
  [f coll]
  (/ (apply + (map f coll)) (count coll)))

(avg #(count (:last-name %)) persons)
“Somehow the idea of reusability got attached to object-oriented programming in the 1980s, and no amount of evidence to the contrary seems to be able to shake it free.”

Paul Graham
Clojure Concurrency
Solving Deadlocks: Timeout & Retry
Solving Deadlocks: Coarse Locks
Locks are the Enemy
(def savings (ref 1000.))
(def checking (ref 2000.))
(def mm (ref 7000.))

(defn transfer
  "Transaction to transfer money from one account to another."
  [from to amount]
  (dosync
   (alter from - amount)
   (alter to + amount)))

(transfer checking savings 500.)
(transfer mm checking 300.)
Retries: Transactions Are Speculative

```
(def savings (ref 1000.))
(def checking (ref 2000.))
(def mm (ref 7000.))

(defn transfer
  "Transaction to transfer money from one account to another."
  [from to amount]
  (dosync
   (alter from - amount)
   (alter to + amount)))

(transfer checking savings 500.)
(transfer mm checking 300.)
```
No Blocking
No Locks
Managing Mutation

- **What** can change?
  - Reference types: **atom**, **var**, **agent**, **ref**

- **When** can they change?

- **When** are changes visible to other threads?
Atoms

• Shared, Global

• Changes are atomic, synchronous, & non-blocking

• (swap!): Pass value to function yielding new value

• (reset!): Force new value, regardless of existing value

user=> (def queue (atom []))
#'user/queue
user=> @queue
[]
user=> (swap! queue conj {:parse "http://www.clojure.org/"})
[{:parse "http://www.clojure.org/"}]
user=> @queue
[{:parse "http://www.clojure.org/"}]
user=> (reset! queue [])
[]
user=> @queue
[]
user=>
Vars — Per-Thread Mutables

• **(def)** sets global binding

• **(binding)** to set up a per-thread override

• **(set!)** if per-thread binding

```clojure
user=> (def x 1)
#=(var user/x)
user=> x
1
user=> (defn manipulate-x []
         (binding [x 2]
                 (printf "Local x is %d" x)
                 (set! x 3)
                 (printf "\nLocal x is now %d\n" x)))
#=(var user/manipulate-x)
user=> (.run (Thread. manipulate-x))
Local x is 2
Local x is now 3
nil
user=> x
1
```
(def *tokens*)

(defn add-token
  [token]
  (set! *tokens* (conj *tokens* token)))

(defn tokenize-xml
  [src]
  (binding [*tokens* []]
    (let [factory (SAXParserFactory/newInstance)]
      (.setNamespaceAware factory true)
      (... factory newSAXParser (parse src sax-handler))
      *tokens*)))

Interfacing with Java APIs
Everything's a Var!

Symbol such as x or map
Functions are stored in Vars

```clojure
user=> (defn say-hello [] (println "Hello"))
#'user/say-hello
user=> (say-hello)
Hello
nil
user=> (binding [say-hello #(println "Goodbye") (say-hello))
Goodbye
nil
user=> (say-hello)
Hello
nil
user=>
```
Agents — Single Thread Writes

```
user=> (def savings (agent 1000.))  
#=(var user/savings)
user=> (def checking (agent 2000.))  
#=(var user/checking)
user=> @savings  
1000
user=> @checking  
2000
user=> (send savings - 300.)  
#<clojure.lang.Agent@c3233b>
user=> (send checking + 300.)  
#<clojure.lang.Agent@67e5a7>
user=> @savings  
700
user=> @checking  
2300
```

- Asynchronous
- Single threaded
- Non-transactional
Refs — Software Transactional Memory

(def savings (ref 1000.))
(def checking (ref 2000.))
(def mm (ref 7000.))

(defn transfer
  "Transaction to transfer money from one account to another."
  [from to amount]
  (dosync
   (alter from - amount)
   (alter to + amount)))

(user=> @checking
2000
(user=> @savings
1000
(user=> (transfer checking savings 500.)
1500
(user=> @checking
1500
(user=> @savings
1500
(user=> (ref-set savings 2000.)
java.lang.IllegalStateException: No transaction running (NO_SOURCE_FILE:0)
Concurrency Notes

• All reference types can have a validator function

• All refs can have a watcher: an agent notified of changes

• (send) inside (dosync) waits until successful completion

• No guarantees when calling Java objects
“The key to performance is elegance, not battalions of special cases.”

Jon Bentley and Doug McIlroy
Wrap Up
Clojure

• 1.0 release: May 4 2009

• Simple, regular syntax

• Improves on Lisp: vectors, maps, sets

• Fully integrates with Java

• Impressive functional & concurrency support

• Many features not covered here

http://www.clojure.org

Java Technical Insight of the Month

Clojure - Functional Programming for the JVM

by

R. Mark Volkmann, Partner
Object Computing, Inc. (OCI)

Contents

<table>
<thead>
<tr>
<th>Introduction</th>
<th>Conditional Processing</th>
<th>Reference Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional Programming</td>
<td>Iteration</td>
<td>Compiling</td>
</tr>
<tr>
<td>Clojure Overview</td>
<td>Recursion</td>
<td>Automated Testing</td>
</tr>
<tr>
<td>Getting Started</td>
<td>Predicates</td>
<td>Editors and IDEs</td>
</tr>
<tr>
<td>Clojure Syntax</td>
<td>Sequences</td>
<td>Desktop Applications</td>
</tr>
<tr>
<td>REPL</td>
<td>Input/Output</td>
<td>Web Applications</td>
</tr>
<tr>
<td>Bindings</td>
<td>Destructuring</td>
<td>Databases</td>
</tr>
<tr>
<td>Collections</td>
<td>Namespaces</td>
<td>Libraries</td>
</tr>
<tr>
<td>StructMaps</td>
<td>Metadata</td>
<td>Conclusion</td>
</tr>
<tr>
<td>Defining Functions</td>
<td>Macros</td>
<td>References</td>
</tr>
<tr>
<td>Java Interoperability</td>
<td>Concurrency</td>
<td></td>
</tr>
</tbody>
</table>

Introduction

The goal of this article is to provide a fairly comprehensive introduction to the Clojure programming language. A large number of features are covered, each in a fairly brief manner. Feel free to skip sections if you are already familiar with the topic.
Object Oriented
Functional
Picture Credits

© 2007 Jon Fife
http://flickr.com/photos/good-karma/577632972/

© 2008 Marcin Wichary
http://flickr.com/photos/mwichary/2222776430/

© 2008 Daniel Chan
http://flickr.com/photos/chanchan222/2847443980/

© 2007 James Manners
http://flickr.com/photos/jmanners/443421045/

© 2008 Woodley Wonderworks
http://flickr.com/photos/wwwworks/2222523486/

© 2005 Jack Keene
http://www.flickr.com/photos/whatknot/3118124/

© 2007 Alan Chia
http://flickr.com/photos/seven13avenue/2080281038/

© 2007 A. Lipson 2003
http://www.andrewlipson.com/escher/relativity.html