

# Clojure

A new Lisp

with excellent concurrency support  
and tight Java integration

# Why Lisp?

Simple – easy to learn

Expressive – say a lot with a little

Flexible – becomes what you want

# Why Clojure?

Advantages of Lisp, plus...

Works with your existing Java code

Uses your favorite Java libraries

Makes concurrency easy

# Simple Data

- \h
- 74
- true
- :first-name
- 7/8
- -5.4
- nil

# Compound Data

- "house"
- [1 2 3 4]
- (true false)
- {:first-name "Eric"  
:last-name "Lavigne"}
- #{"house" "office"  
"museum"}

# Expressions

- 74
- (+ 1 1)
- (str "The date is " (new Date))
- (println (str "The date is " (new Date)))
- (def x 7)
- (< 7 8)
- (if (< x 8) "less than 8" "at least 8")

# Evaluation and the REPL

- `java -cp clojure.jar clojure.lang.Repl`
- Evaluate one expression at a time
- Good for experimental programming

`(+ 1 1)`  $\Rightarrow$  `2`

`(def x 6)`  $\Rightarrow$  `#'user/x`

`(if (< x 10) "less" "more")`  $\Rightarrow$  `"less"`

# Functions

```
(def add-six  
  (fn [x] (+ 6 x)))
```

```
(def hello  
  (fn [name1 name2]  
    (str "Hello " name1 " and " name2 ".")))
```

```
(add-six 4)      => 10
```

```
(hello "Jack" "Jill") => "Hello Jack and Jill."
```

# Advantages of Functions

- Break code into smaller pieces
  - Easier to understand
  - Tests more precisely identify errors
- Re-use rather than repeat
  - Less code, easier to read and write
  - Improvements can be made in one place

# Higher-Order Functions

Functions can be used as input to other functions.

```
(map add-six [0 4 10]) => (6 10 16)
```

```
(def sum  
  (fn [x] (reduce + 0 x)))
```

```
(def product  
  (fn [x] (reduce * 1 x)))
```

# Macros

Tired of patterns like this?

```
(def add-six (fn [x] (+ x 6)))
```

The `defn` macro in the standard library allows you to write this:

```
(defn add-six [x] (+ x 6))
```

Patterns are a sign that it's time for a function or macro. Don't repeat yourself :-)

# Macros for Web Development

```
(ns hello
  (:require [compojure.http.servlet :as servlet]
            [compojure.http.routes :as routes]
            [compojure.server.jetty :as jetty]))
(servlet/defservlet hello-servlet
  (routes/ANY "/" "Hello.))
(jetty/defserver hello-server
  {:port 80}
  "/" hello-servlet)
(jetty/start hello-server)
```

# Using Java Libraries

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

`(.getName String)`      =>      `"java.lang.String"`

`(System/getProperty`  
    `"java.vm.version")`      =>      `"1.6.0_07-b06-57"`

`Math/PI`      =>      `3.14159...`

`(import 'org.joda.time.Duration)`

`(new Duration`  
    `(.longValue 5000))`      =>      `#<Duration PT5S>`

# Extending Classes and Interfaces

```
button.addActionListener(  
    new ActionListener() {  
        public void actionPerformed(ActionEvent evt) {  
            System.out.println("Button Pressed");  
        }  
    }  
);
```

```
(.addActionListener button  
 (proxy [ActionListener] []  
 (actionPerformed [evt]  
 (println "Button Pressed"))))
```

# Creating Java Libraries

- Using Clojure to create Java classes, compiled to \*.class files
- Recent feature (December 2008) so expect additional improvements soon

# How It Will Be Used

```
package aotexample;

public class TestMultipliers {

    public static void main(String[] args) {

        System.out.println("Result of Java version is "
            + new JavaMultiplier(3).multiply(2));

        System.out.println("Result of Clojure version is "
            + new ClojureMultiplier(new Integer(3)).multiply(2));

    }

}
```

# Java Version

```
package aotexample;

import java.io.Serializable;

public class JavaMultiplier implements Serializable {
    private Integer a;

    public JavaMultiplier(Integer a) {
        this.a = a;
    }

    public Integer multiply(Integer b) {
        return a * b;
    }
}
```

# Clojure Version

```
(ns aotexample.ClojureMultiplier
  (:import (java.io Serializable)))
(:gen-class
  :implements [java.io.Serializable]
  :methods [[multiply [Integer] Integer]]
  :state a
  :init init-multiplier
  :constructors { [Integer] [] })
(defn -init-multiplier [a]
  [] a)
(defn -multiply [this b]
  (* (.a this) b))
```

# Compilation

```
java -cp src:classes:lib/clojure.jar clojure.lang.Script  
src/aotexample/compilemultiplier.clj
```

```
javac -cp src:classes:lib/clojure.jar -d classes  
./src/aotexample/JavaMultiplier.java
```

```
javac -cp src:classes:lib/clojure.jar -d classes  
./src/aotexample/TestMultipliers.java
```

```
java -cp src:classes:lib/clojure.jar aotexample.TestMultipliers
```

```
src/aotexample/compilemultiplier.clj:
```

```
(compile 'aotexample.ClojureMultiplier)
```

# Transactional Memory

- Place any shared state inside a ref
- Change refs only within a transaction
- Read refs with or without transaction
  - deref vs ensure
- Optimistic locking and automatic retrying
- Only effects of transaction should be altering refs and returning value. Anything else could be overdone due to automatic retrying.

# Transaction Example

```
(defn transfer [from to amount]
  (dosync
    (when (> (ensure from) amount)
      (alter from (fn [x] (- x amount)))
      (alter to (fn [x] (+ x amount))))))
(def account1 (ref 2000))
(def account2 (ref 1000))
(transfer account1 account2 1000)
(dosync [(ensure account1) (ensure account2)])
```

# Agents

- Each agent has a state.
- You can read the agent's state, or add a watcher that responds to state changes.
- You can asynchronously send a function to the agent that will change the state.
- You can add a validator function to block state changes.

# How to Get Started

- [http://clojure.org/getting\\_started](http://clojure.org/getting_started)
- <http://groups.google.com/group/clojure>
- Programming Clojure (Pragmatic Programmers)
- <http://ericlavigne.wordpress.com>
- Ask Questions