Time, State, Identity

Time
   Relative moments when an event occurs

State
   Snapshot of entity’s properties at a moment in time

Identity
   Logical entity identified by a common stream of states occurring over time
{:name "Sarah"
 :age 10
 :wears-glasses false}

(def sarah
State & Identity

Different things in Clojure

{:name "Sarah"
 :age 10
 :wears-glasses false}

{:name "Sarah"
 :age 11
 :wears-glasses false}

(def sarah
    ...
State & Identity

Different things in Clojure

{:name "Sarah" :age 10 :wears-glasses false}  
{:name "Sarah" :age 11 :wears-glasses false}  
{:name "Sarah" :age 12 :wears-glasses true}

(def sarah
Java

class Person {
    public String name;
    public int age;
    public boolean wearsGlasses;

    public Person (String name, int age, boolean wearsGlasses) {
        this.name = name;
        this.age = age;
        this.wearsGlasses = wearsGlasses;
    }
}

Thursday, October 30, 14
State & Identity

Person sarah

Sarah
10
false

Complexed in Java
State & Identity

Complexed in Java

Person sarah

Sarah
11
false
State & Identity

Person sarah

Sarah
12
true

Complexed in Java
Memento Pattern

Store an object's internal state, so the object can be restored to this state later without violating encapsulation

State is immutable so when make changes still have originial

Don’t need a pattern to copy old state
Reference Type Basics

var, ref, atom, agent

All are pointers

Can change pointer to point to different data

Dereferencing will never block

Each type as different way of setting/changing its value
Reference Type Basics

var, ref, atom, agent

Each type

Can have meta data

Can have watches (observers)
   Call specified function when value is change

Can have validator
   Enforce constraints on values pointer can point to
### Features of each Type

<table>
<thead>
<tr>
<th></th>
<th>Ref</th>
<th>Agent</th>
<th>Atom</th>
<th>Var</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coordinated</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Asynchronous</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Retriable</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Thread-local</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Synchronous - block until operation completes

Asynchronous - Non blocking, operation can compete on separate thread

Coordinated - Supports transactions

Thread-local - Changes made are local to current thread
### Features of each Type

<table>
<thead>
<tr>
<th></th>
<th>Ref</th>
<th>Agent</th>
<th>Atom</th>
<th>Var</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coordinated</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Asynchronous</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Retriable</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Thread-local</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Synchronous - block until operation completes
Asynchronous - Non blocking, operation can compete on separate thread
Coordinated - Supports transactions
Thread-local - Changes made are local to current thread
Creating & Referencing Each Type

(def ref-example (ref 10))
@ref-example
(deref ref-example)

(def agent-example (agent 10))
@agent-example
(deref agent-example)

(def atom-example (atom 10))
@atom-example
(deref atom-example)

(def var-example 10)
var-example
Creating & Referencing Each Type

(def ref-example (ref 10))
@ref-example
(deref ref-example)

(def agent-example (agent 10))
@agent-example
(deref agent-example)

(def atom-example (atom 10))
@atom-example
(deref atom-example)

(def var-example 10)
var-example

Note the difference
(defn cat-watch
    [key pointer old new]
    (println "Watcher" key pointer old new))

(def cat 4)
(add-watch (var cat) :cat cat-watch)
(def cat 10)
(remove-watch (var cat) :cat)
(def cat 20)
Observer Pattern

One-to-many dependency between objects

When one object changes state, all its dependents are notified and updated automatically

Watches provide same functionality as the Observer pattern
Validator

(def cat 4)

(set-validator! (var cat) #(> 10 %))

(def cat 9)

(def cat 20) ;;exception
Atoms

Changes are
  Synchronous
  Uncoordinated
  Atomic

Synchronous
  Code waits until change done

Uncoordinated
  No transaction support

Atomic
  Threads only see old or new value
  Never see partially changed data
Atoms - Methods for change

swap!
   Applies function to current state for new state

reset!
   Changes state to given value

compare-and-set!
   Changes state to given value only if current value is what you think it is
reset!

(def a (atom 0))

@a 0

(reset! a 5) 5

@a 5
swap!

(def a (atom 0))

@a 0

(swap! a inc) 1

@a 1
swap!

(def sarah (atom {:name "Sarah" :age 10 :wears-glasses? false}))

(swap! sarah update-in [:age] + 3)  
{:name "Sarah", :age 13,  
:wears-glasses? false}

@sarah  
{:name "Sarah", :age 13,  
:wears-glasses? false}
**swap! is Atomic**

```
(swap! sarah (comp #(update-in % [:age] inc)
                  #(assoc % :wears-glasses? true))
```

Compound operation on sarah

What happens if other thread reads sarah during swap!
swap! is Atomic

(swap! sarah (comp #(update-in % [:age] inc)
    #(assoc % :wears-glasses? true)))

Compound operation on sarah

What happens if other thread reads sarah during swap!

   It gets the old value
swap! is Atomic

(swap! sarah (comp #(update-in % [:age] inc)
    #(assoc % :wears-glasses? true)))

What happens if other thread modifies sarah during swap!
swap! is Atomic

(swap! sarah (comp #(update-in % [:age] inc)
    #(assoc % :wears-glasses? true)))

What happens if other thread modifies sarah during swap!

It retries until it can read the new value

Then modifies sarah
If the value of atom \( a \) changes between the time when function \( g \) is invoked and the time when it returns a new value for \( a \) (\( a_1 \) and \( a_2 \), respectively), \( \text{swap!} \) will discard that new value and reevaluate the call with the latest available state of \( a \). This will continue until the return value of \( g \) can be set on \( a \) as the immediate successor of the state of \( a \) with which it was invoked.

There is no way to constrain \( \text{swap!} \)'s retry semantics; given this, the function you provide to \( \text{swap!} \) must be pure, or things will surely go awry in hard-to-predict ways.

Being a synchronous reference type, functions that change atom values do not return until they have completed:

```clojure
(def x (atom 2000))
;; #'user/x
(swap! x #(Thread/sleep %))
;; nil
```

This expression takes at least two seconds to return.

A "bare" compare-and-set! operation is also provided for use with atoms, if you already think you know what the value of the atom being modified is; it returns true only if the atom's value was changed:

```clojure
(compare-and-set! xs :wrong "new value")
;; false
(compare-and-set! xs @xs "new value")
;; true
@xs
;; "new value"
```
Recall - Future

Computes body on another thread

Use @ or deref to get answer

@, deref blocks until computation is done
Recall - Future

Computes body on another thread

Use @ or deref to get answer

@, deref blocks until computation is done

(def long-calculation (future (apply + (range 1e8))))
@long-calculation
Macro from Text

(wait-futures n f1 f2 ... fk)

Runs each function in n different futures
Macro from Text

(wait-futures n f1 f2 ... fk)

Runs each function in n different futures

(wait-futures
    3
    (println "Hi Mom")
    (println "Hi Dad"))
Macro from Text

(wait-futures n f1 f2 ... fk)

Runs each function in n different futures

(wait-futures
  3
  (println "Hi Mom")
  (println "Hi Dad"))

Console

Hi Mom
Hi Dad
Hi Mom
Hi Dad
Hi Dad
Hi Mom
Showing the Retries

(def xs (atom [1 2 3]))

(wait-futures 2
  (swap! xs (fn [v]
              (Thread/sleep 400)
              (println "trying 4")
              (conj v 4)))
  (swap! xs (fn [v]
              (Thread/sleep 500)
              (println "trying 5")
              (conj v 5))))

@xs
(def xs (atom [1 2 3]))

(wait-futures 2
  (swap! xs (fn [v]
    (Thread/sleep 400)
    (println "trying 4")
    (conj v 4)))
  (swap! xs (fn [v]
    (Thread/sleep 500)
    (println "trying 5")
    (conj v 5))))

@xs

Console

trying 4
trying 4
trying 5
trying 5
trying 4
trying 5
trying 5
trying 5 trying 5
trying 5
Showing the Retries

(def xs (atom [1 2 3]))

(wait-futures 2
  (swap! xs (fn [v]
              (Thread/sleep 400)
              (println "trying 4")
              (conj v 4)))
  (swap! xs (fn [v]
              (Thread/sleep 500)
              (println "trying 5")
              (conj v 5))))

@xs [1 2 3 4 4 5 5]
compare-and-set!

(compare-and-set! atom oldval newval)
Only changes the atom to newval if the value of atom is oldval

Used when you don’t want to change the atom after another thread does
(defn running-sum
  [n]
  (let [sum (atom n)]
    (fn [x]
      (swap! sum + x)
      @sum))))
Identity local to method

(defn running-sum [n]
  (let [sum (atom n)]
    (fn [x]
      (swap! sum + x)
      @sum)))

(def bill (running-sum 10))
(bill 5)
(bill 12.5)
@sum
Identity local to method

(defn running-sum [n]
  (let [sum (atom n)]
    (fn [x]
      (swap! sum + x)
      @sum)))

(def bill (running-sum 10))

(bill 5) 15
(bill 12.5) 27.5
@sum
Exception
Var

Private
Docstrings
Constants
Dynamic Scope
Private Var

(def ^:private life 42)
(def ^{:private true} life 42)

(defn- foo [] "foo")
(def ^:private (fn [] "foo")

Private vars
Can be accessed outside of defining namespace using the full name
Docstrings

(def a
  "Sample doc string"
  10)

(defn b
  "Another doc string"
  [b]
  (inc b))
Docstrings

(def a
  "Sample doc string"
  10)

(defn b
  "Another doc string"
  [b]
  (inc b))

(def b
  "Another doc string"
  (fn [b]
   (inc b)))
(def max-value 255)

(defn valid-value? [v] (<= v max-value))

(valid-value? 270)    false

(def max-value 511)

max-value    511

(valid-value? 270)    true
Constants

(def max-value 255)  (def ^:const max-value 255)

(defn valid-value? 
[v]
(<= v max-value))

(valid-value? 270) false (valid-value? 270) false

(def max-value 511) (def max-value 511)

max-value 511 max-value 511

(valid-value? 270) true (valid-value? 270) false
Dynamic Scoping

(def ^:dynamic *max-value* 255)

(defn valid-value? [v]
  (<= v *max-value*))

(valid-value? 270)        false

(binding [*max-value* 511]
  (valid-value? 270))     true

(valid-value? 270)        false
Dynamic Scoping - Works across Threads

(binding [*max-value* 500]
 (println (valid-value 299)) true
 @(future (valid-value 299))) true
Dynamic Scoping - Need ^:dynamic

(def *max-value* 255)

(defn valid-value? [v]
  (<= v *max-value*))

(valid-value? 270) false

(binding [*max-value* 511]
  (valid-value? 270)) Exception

(valid-value? 270)
Dynamic Scoping - const wins

(def ^:dynamic ^const *max-value* 255)

(defn valid-value?
  [v]
  (<= v *max-value*))

(valid-value? 270)  false

(binding [*max-value* 511]
  (valid-value? 270))  false

(valid-value? 270)  false
Sample uses

In repl (not in light table)

*print-length* - var use in print to determine how many items in a collection to print out

(set! *print-length* 3)
(iterate inc 0)   (0 1 2 ...)
(set! *print-length* 10)
(iterate inc 0)   (0 1 2 3 4 5 6 7 8 9...)

Default settings that don’t change very often
*warn-on-reflection*

user=> (def i 23)
#'user/i
user=> (.toString i)
"23"
user=> (set! *warn-on-reflection* true)
true
user=> (.toString i)
Reflection warning, NO_SOURCE_PATH:1:1 - reference to field toString can't be resolved.
"23"

user=> (def ^Long i 23)
#'user/i
user=> (.toString i)
"23"
What is Going On?

Java is statically typed
Clojure compiles to Java
Clojure infers the types of data
If can not infer uses Java's reflection
Reflection is slow
*warn-on-reflection* used to find out when reflection is used
Add type hints to avoid reflection
Type Hints Example

(defn ^Float sum-square
  [...floats xs]
  (let [...floats squares (map #(* % %) xs)]
    (reduce + squares)))
**alter-var-root**  
(alter-var-root a-var f & args)  
Changes the root value of a-var by applying f to a-var and binding a-var to the result

```
(defn foo 
  [n] 
  (inc n))

(alter-var-root 
  (var foo) 
  (fn [f] 
    #((do (println "fooing" %)) 
      (f %)))))

(foo 2)
```
Aspect-Oriented Programming

Separation of cross-cutting concerns

Before, after, around methods

Logging
cross-cuts all classes/methods you want to log
alter-var-root

Allows us to implement AOP

Show execution of program

Coverage tool

Profile tool
Ref

Coordinated reference type

Multiple values can be changed

Changes are atomic

No Race conditions

No deadlocks

No manual locks, monitors etc
Software Transactional Memory

Ref changes are done in a transaction

No changes are visible outside transaction until transaction is completed

Exceptions abort the transaction

If
   Transaction A and B modify one or more of the same refs
   Transaction A starts before B, but ends between B’s start and end
Then
   Transaction B will retry with the new values of the refs
Starting a Transaction

(dosync form1 form2 ... formN)
Altering a ref

(alter ref fun & args)

Applies the fun to the ref to get new value

(ref-set ref val)

Sets the ref to val
Example

(def sam-account (ref 10))
(def pete-account (ref 20))
(set-validator! sam-account #(<= 0 %))
(set-validator! pete-account #(<= 0 %))
(defn sam-pay-pete [amount]
  (dosync
   (alter pete-account + amount)
   (alter sam-account - amount)))
(sam-pay-pete 8)
@sam-account   2
@pete-account   28
(sam-pay-pete 8) Exception
@sam-account   2
@pete-account   28