CS 696 Functional Programming and Design
Fall Semester, 2015
Doc 11 Records, Protocols, References
Oct 1, 2015

Records

Defining Clojure Types

```
(defrecord Point [x y])

(deftype Point [x y])
```

Both

- Compile to Java class with final fields
- Accessing & updating fields faster than Clojure maps
- Use Java naming convention

deftype - lower level construct

Creating & Accessing

(defrecord Point [x y])	
(def a (Point. 2 3))	
(.x a)	2
(:x a)	2
(:z a 0)	0

Creating with Types

```
(defrecord NamedPoint [^String name ^long x ^long y])

(def b (NamedPoint. "Small" 2 4))

(:x b)

(NamedPoint/getBasis)                [name x y]
```

This avoid the autoboxing of the values

Records

Support value semantics

Act like maps

Metadata support

Reader support

Value Semantics

Immutable

If fields of two records are equal than Records are equal

(= (Point. 1 2) (Point. 1 2))	true
(= 3 3N)	true
(= (Point. 1 2) (Point. 1N 2N))	true

Records are like Maps

(let [{:keys [x y]} (Point. 2 3)] x)	2
(assoc (Point. 1 2) :z 5)	#user.Point{:x 1, :y 2, :z 5}
(dissoc (Point. 1 2) :x)	{:y 2}
(seq (Point. 1 2))	([:x 1] [:y 2])
(into {} (Point. 3 4))	{:x 3, :y 4}

assoc returns a Point
dissoc returns a map

But Records are not Maps

(= (Point. 1 2) {:x 1 :y 2})	false
((Point. 1 2) :x)	Exception
({:x 1 :y 2} :x)	1
(:x (Point. 1 2))	1
(.x (Point. 1 2))	1
(get (Point. 1 2) :x)	1

Records are not Defined in Namespaces

Records are Java Classes

Not included when import/require Clojure namespace

Have to require the Record

Namespace record is declared in is part of the full name of the Record

Auxiliary Constructor

(Point. 1 2 {:foo :bar} {:z 3})

metadata

More fields

Constructors & Factory Functions

Text recommends you provide functions to create records

Functions can be used by higher order functions

Makes it easier to change record definition

Built in Factory Methods

->RecordType	positional
map->RecordType	from a map

```
(->Point 2 3)

(map->Point {:y 2 :x 1})
```

Records verses Maps

Performance

- Records define Java class
- Faster access to values
- Operations with data can be faster

Documentation

- Records specify what fields they must contain

Some Clojure Performance

```
(def i 5)

(def s "12")

(.toString s)
(.toString i)
```

No type information for i or s

So how to select correct toString method at runtime?

Use Java reflection - which is slow

warn-on-reflection

```
(def i 5)
(def s "12")
=> i
=> s

(set! *warn-on-reflection* true)
=> true

(.toString s)
Reflection warning, /private/var/folders/br/q_fcsjqc8xj9qn0059bctj3h0000gr/T/form-init8847540080428279079.clj:1:1 - reference to field toString can't be resolved.
=> "12"
```

warn-on-reflection

```
(def i 5)
(def s "12")
=> i
=> s

(set! *warn-on-reflection* true)
=> true

(.toString ^String s)
=> "12"
(.toString ^Long i)
=> "5"
```

Protocols

Protocols

Like Java interfaces

Contains one or more methods

Each method can have multiple arities

Each method has at least one argument

Single dispatch on first argument

```
(defprotocol ProtocolName
   "documentation"
   (a-method [this arg1 arg2] "method docstring")
   (another-method [x] [x arg] "docstring"))
```

Protocols

```
(defprotocol Shape
  (area [s] )
  (perimeter [s]))

(defrecord Rectangle [length width]
  Shape
  (area [this] (* length width))
  (perimeter [this] (+ (* 2 length)
                       (* 2 width))))

(defrecord Circle [radius]
  Shape
  (area [this] (* (Math/PI) radius radius))
  (perimeter [this] (* 2 (Math/PI) radius)))
```

```
(area (Circle. 2))
(area (Rectangle. 2 3))
```

Extending Existing Types

```
(defprotocol FIFO
  (fifo-push [fifo value])
  (fifo-pop [fifo])
  (fifo-peek [fifo]))

(extend-type clojure.lang.IPersistentVector
  FIFO
  (fifo-push [vector value]
    (conj vector value))
  (fifo-pop [vector]
    (pop vector))
  (fifo-peek [vector]
    (last vector)))
```

```
(fifo-pop [1 2 3 4])
(fifo-peek [1 2 3])
```

Extending Existing Types

```
(extend-type clojure.lang.PersistentList
  FIFO
  (fifo-push [seq value]
    (conj seq value))
  (fifo-pop [seq]
    (pop seq))
  (fifo-peek [seq]
    (first seq)))
```

```
(fifo-push '(1 2 3) 4)
```

References

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

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

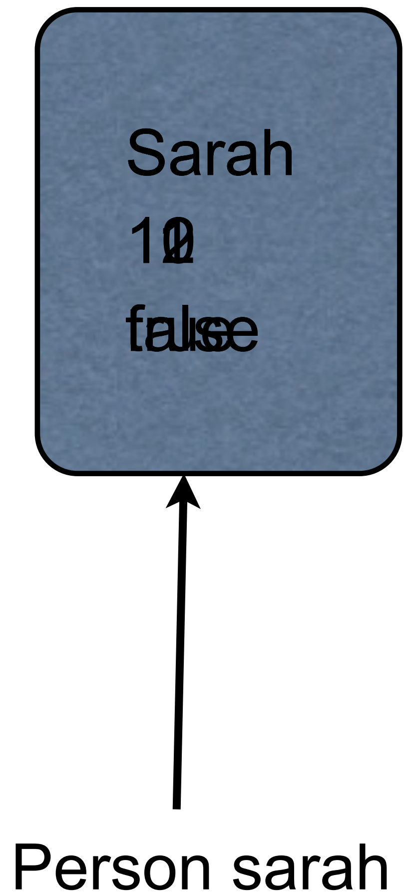
Java

```
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;
    }
}
```

State & Identity

Complexted in Java



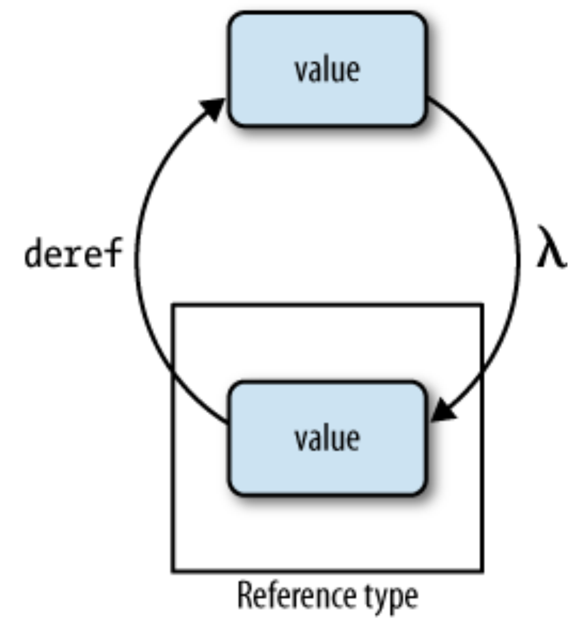
Memento Pattern

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 orginial

Don’t need a pattern to copy old state

Reference Type Basics



var, ref, atom, agent

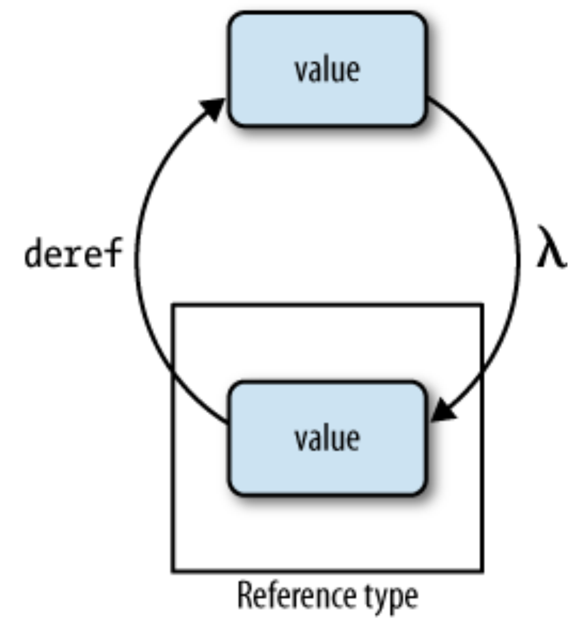
All are pointers

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)

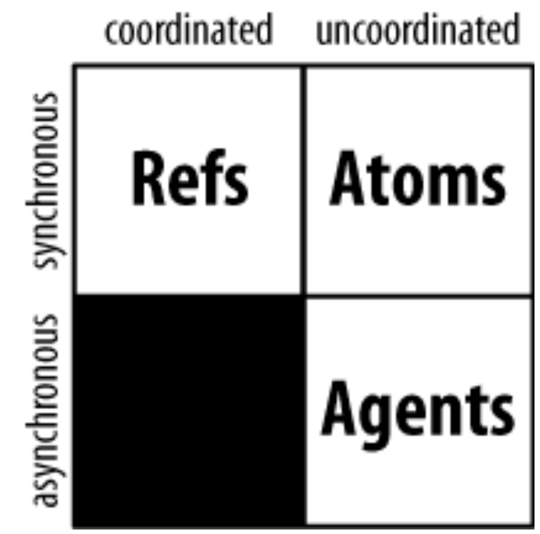
Call specified function when value is change

Can have validator

Enforce constraints on values pointer can point to

Features of each Type

	Ref	Agent	Atom	Var
Coordinated	X			
Asynchronous		X		
Retriable	X		X	
Thread-local				X



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
```

Note the difference

Watches

```
(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)
```

Output in Console

```
Watcher :cat #'user/cat 4 10
```

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!

It gets the old value

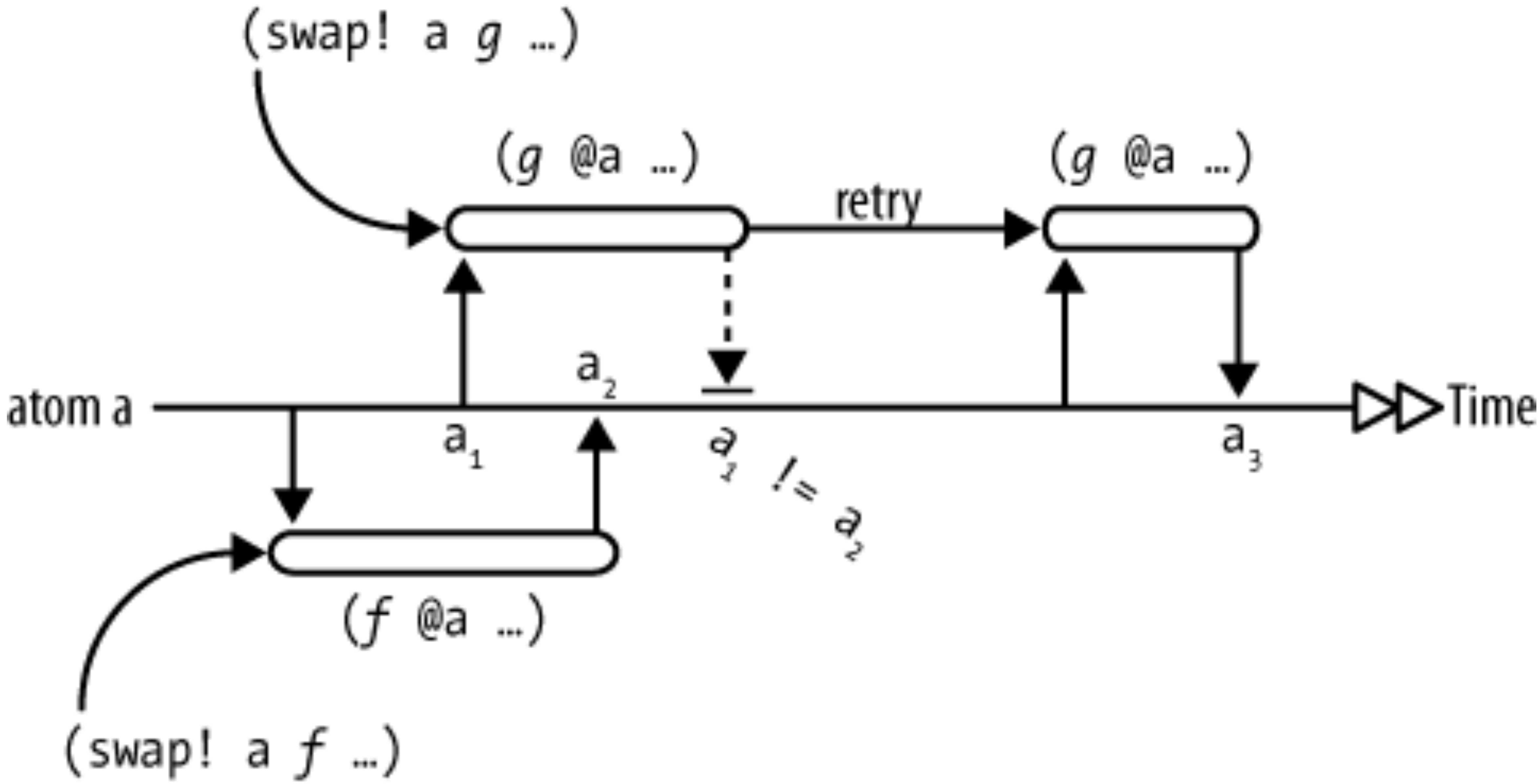
swap! is Atomic

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

What happens if other thread modifies sarah during swap!

It retries until it can read the new value

Then modifies sarah



(swap! a g ...)
(g @a ...)
retry
(g @a ...)
atom a
a2
a1
a1 != a2
a3
Time
(f @a ...)
(swap! a f ...)

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

```
(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                         [1 2 3 4 4 5 5]
```

Console

```
trying 4
trying 4
trying 5
trying 5
trying 4
trying 5
trying 5
trying 5 trying 5
trying 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 do't want to change the atom after another thread does

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
```

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

```
(defn- foo [] “foo”)

(def ^:private (fn [] “foo”)
```

Docstrings

```
(def a
    "Sample doc string"
    10)
```

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

```
(def b
    "Another doc string"
    (fn [b]
    (inc b)))
```

Constants

```
(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
```

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

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

(valid-value? 270)                false

(def max-value 511)

max-value                     511

(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 relection

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)
```

Console
fooing 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 out side 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)

Applys 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