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

deftype - lower level construct

Use Java naming convention
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 then Records are equal

\[ (= \text{(Point. 1 2)} \text{(Point. 1 2)}) \quad \text{true} \]

\[ (= \text{3 3N}) \quad \text{true} \]

\[ (= \text{(Point. 1 2)} \text{(Point. 1N 2N)}) \quad \text{true} \]
Records are like Maps

(let [{:keys [x y]} (Point. 2 3)] 2
  x)

(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

\[(= \text{Point.} \ 1 \ 2) \{:x \ 1 :y \ 2\}\] false

\[(\text{Point.} \ 1 \ 2) :x\] Exception

\[{:x \ 1 :y \ 2} :x\] 1

\[{:x (\text{Point.} \ 1 \ 2)}\] 1

\[.x (\text{Point.} \ 1 \ 2)\] 1

\[(\text{get} (\text{Point.} \ 1 \ 2) :x)\] 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)))
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;
    }
}

Thursday, October 1, 15
State & Identity

Person sarah

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 original

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

---

In addition to all being dereferenceable, all reference types:

- May be decorated with metadata (see [Metadata](#)). Rather than using `with-meta` or `vary-meta`, metadata on reference types may only be changed with `alter-meta!`, which modifies a reference's metadata in-place.

- Can notify functions you specify when the their state changes; these functions are called `watches`, which we discuss in [Watches](#).

- Can enforce constraints on the state they hold, potentially aborting change operations, using `validator` functions (see [Validators](#)).

---

### Classifying Concurrent Operations

In thinking about Clojure's reference types, we'll repeatedly stumble across a couple of key concepts that can be used to characterize concurrent operations. Taken together, they can help us think clearly about how each type is best used.

#### Coordination

A coordinated operation is one where multiple actors must cooperate (or, at a minimum, be properly sequestered so as to not interfere with each other) in order to yield correct results. A classic example is any banking transaction: a process that aims to transfer monies from one account to another must ensure that the credited account not reflect an increased balance prior to the debited account reflecting a decreased balance, and that the transaction fail entirely if the latter has insufficient funds. Along the way, many other processes may provoke similar transactions involving the same accounts. Absent methods to coordinate the changes, some accounts could reflect incorrect balances for some periods, and transactions that should have failed (or should have succeeded) would succeed (or fail) improperly.

In contrast, an uncoordinated operation is one where multiple actors cannot impact each other negatively because their contexts are separated. For example, two different threads of execution can safely write to two different files on disk with no possibility of interfering with each other.

#### Synchronization

Synchronous operations are those where the caller's thread of execution waits or blocks or sleeps until it may have exclusive access to a given context, whereas asynchronous operations are those that can be started or scheduled without blocking the initiating thread of execution.

Just these two concepts (or, four, if you count their duals) are sufficient to fully characterize many (if not most) concurrent operations you might encounter. Given that, it makes sense that Clojure's reference types were designed to implement the semantics necessary to address permutations of these concepts, and that they can be conveniently classified according to the types of operations for which each is suited.

When choosing which reference type(s) to use for a given problem, keep this classification in mind; if you can characterize a particular problem using it, then the most appropriate reference type will be obvious.
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
(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
Figure 4-1. Interaction of conflicting swap! operations on a shared atom

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), 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 swap!'s retry semantics; given this, the function you provide to 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:

```lisp
(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:

```lisp
(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

(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
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 don’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)

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

Private vars
Can be accessed outside of defining
namespace using the full name
(def a
    "Sample doc string"
    10)

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

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

(def max-value 255)                        (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) true
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)
ture
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
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)

    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