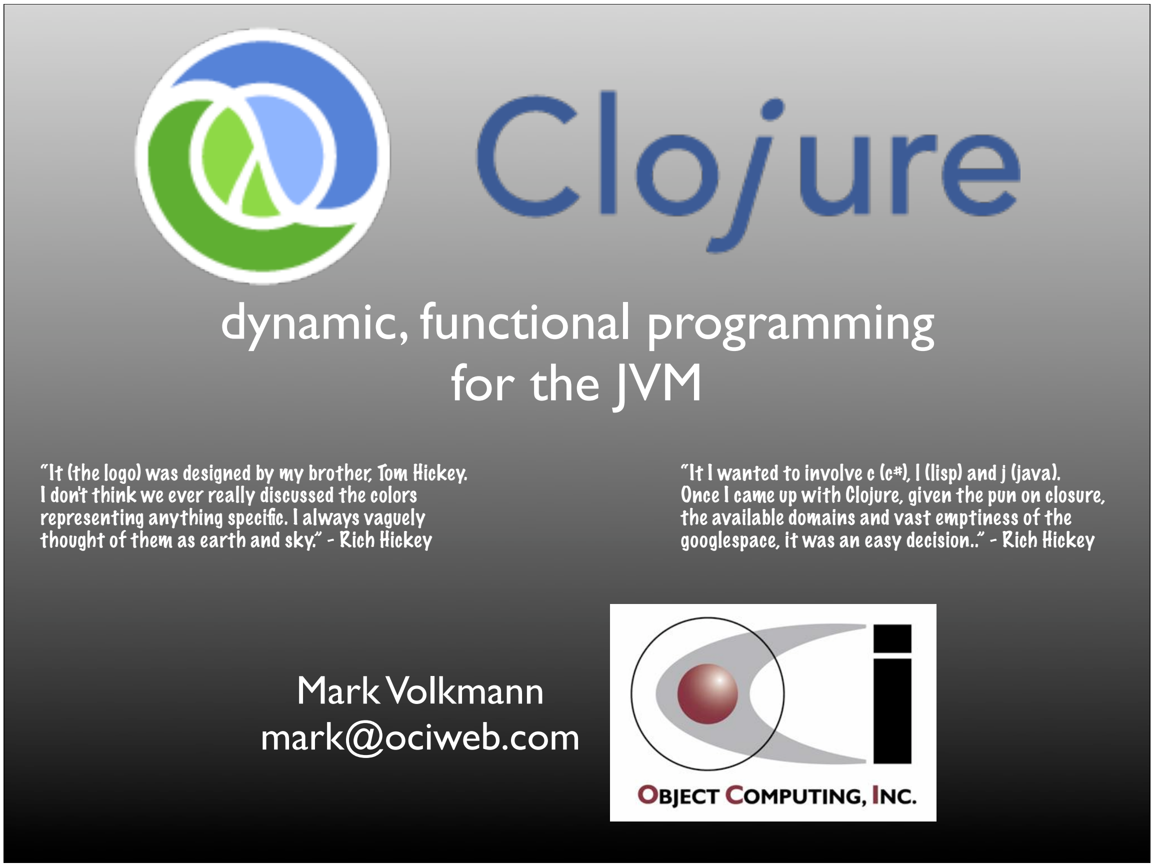


Clojure

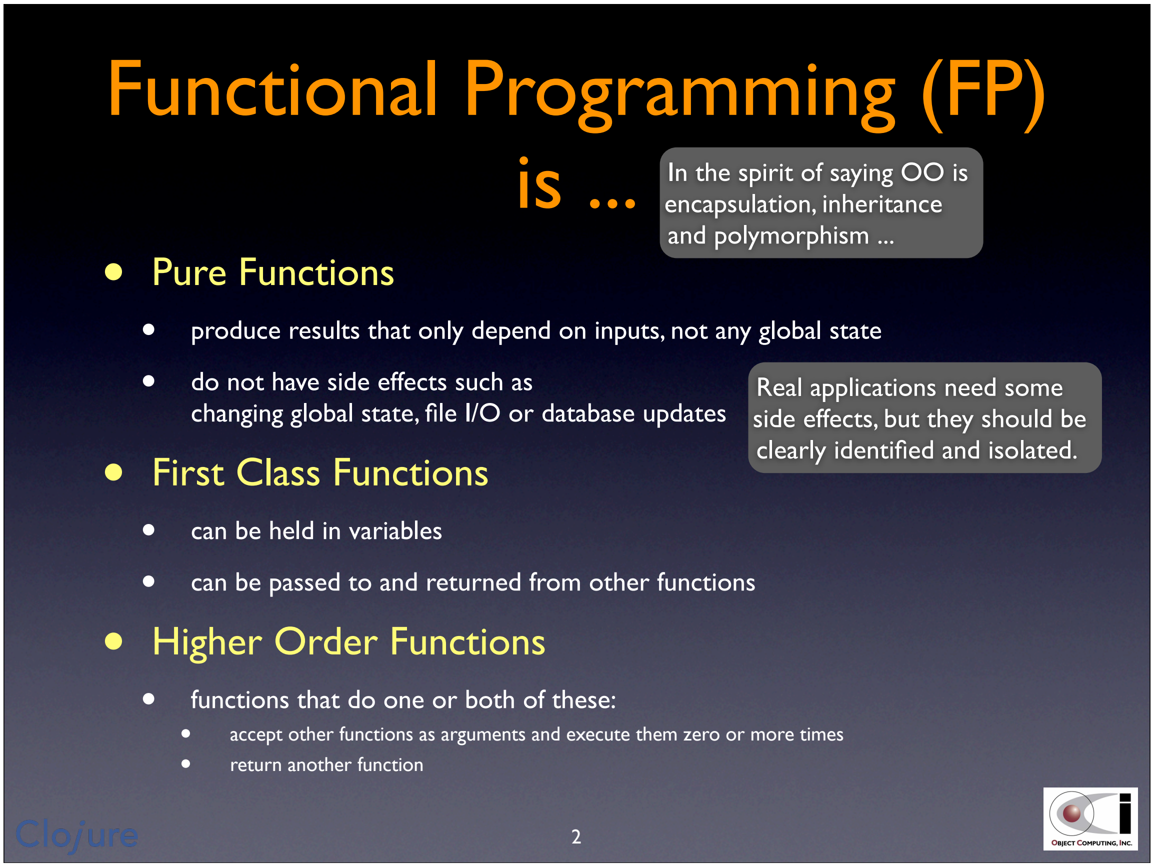
dynamic, functional programming for the JVM

"It (the logo) was designed by my brother, Tom Hickey. I don't think we ever really discussed the colors representing anything specific. I always vaguely thought of them as earth and sky." - Rich Hickey

"It I wanted to involve c (c#), l (lisp) and j (java). Once I came up with Clojure, given the pun on closure, the available domains and vast emptiness of the googlespace, it was an easy decision.." - Rich Hickey

Mark Volkmann
mark@ociweb.com

OBJECT COMPUTING, INC.



Functional Programming (FP) is ...

In the spirit of saying OO is encapsulation, inheritance and polymorphism ...

- Pure Functions
  - produce results that only depend on inputs, not any global state
  - do not have side effects such as changing global state, file I/O or database updates
- First Class Functions
  - can be held in variables
  - can be passed to and returned from other functions
- Higher Order Functions
  - functions that do one or both of these:
    - accept other functions as arguments and execute them zero or more times
    - return another function

Real applications need some side effects, but they should be clearly identified and isolated.

... FP is ...

- Closures
  - special functions that retain access to variables that were in their scope when the closure was created

main use is to pass a block of code to a function

- Partial Application
  - ability to create new functions from existing ones that take fewer arguments
- Currying
  - transforming a function of n arguments into a chain of n one argument functions
- Continuations
  - ability to save execution state and return to it later

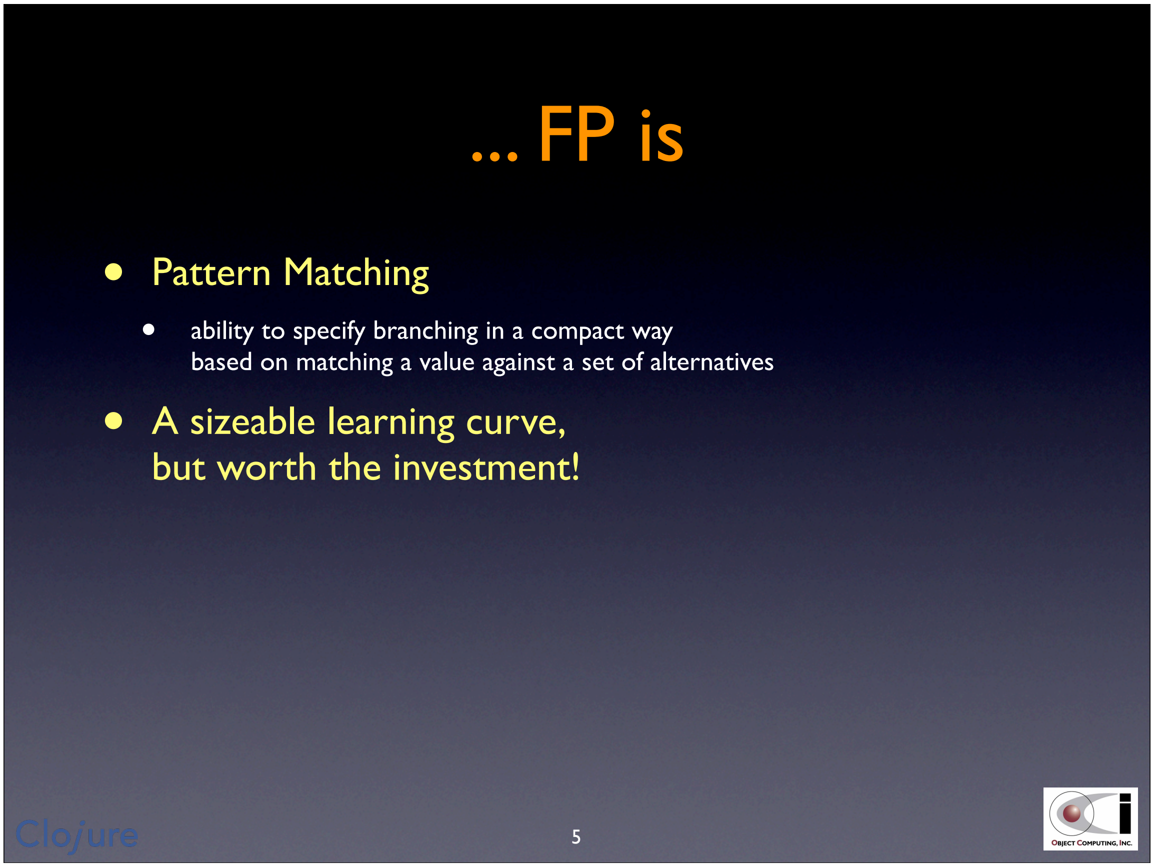
think browser back button

... FP is ...

- Immutable Data
  - after data is created, it can't be changed
  - new versions with modifications are created instead
  - some FP languages make concessions, but immutable is the default

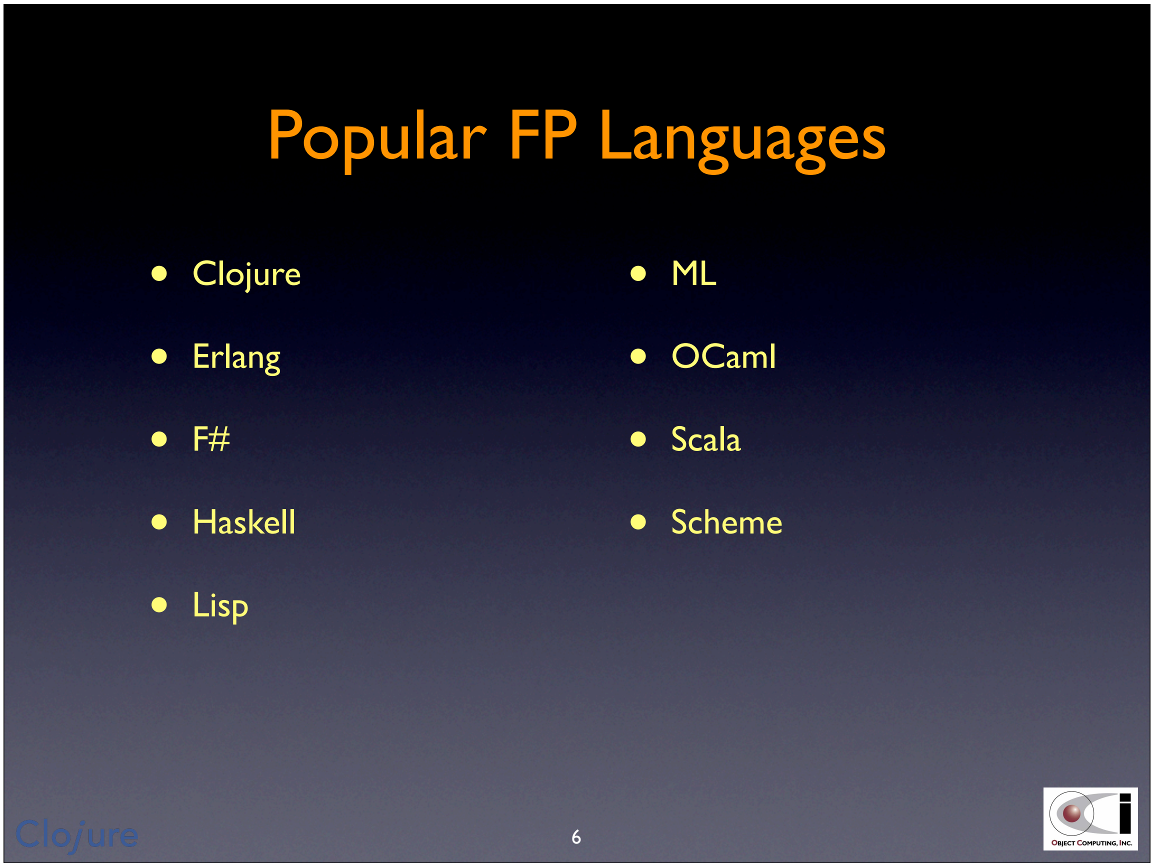
made efficient with persistent data structures

- Lazy Evaluation
  - ability to delay evaluation of functions until their result is needed
  - useful for optimization and implementing infinite data structures
- Monads
  - manage sequences of operations to provide control flow, null handling, exception handling, concurrency and more



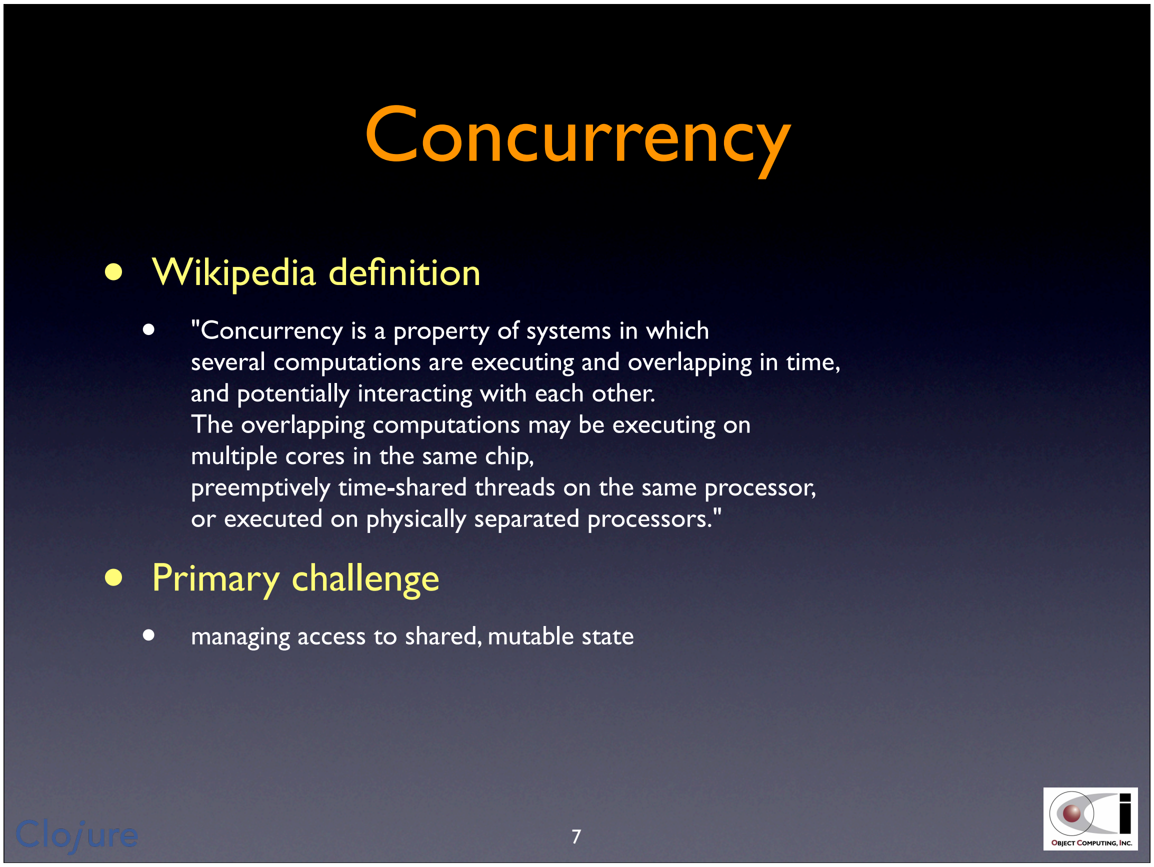
… FP is

- Pattern Matching
  - ability to specify branching in a compact way
    based on matching a value against a set of alternatives
- A sizeable learning curve,
  but worth the investment!



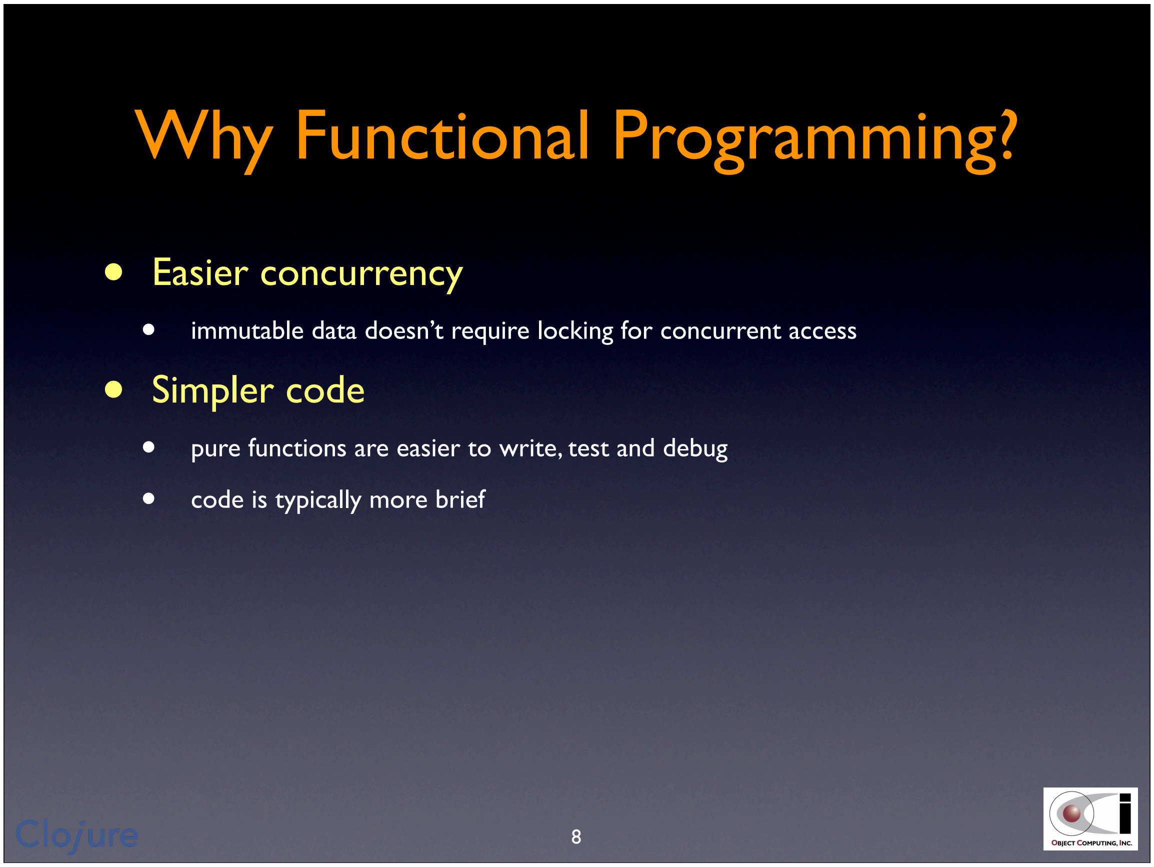
Popular FP Languages

- Clojure
- Erlang
- F#
- Haskell
- Lisp
- ML
- OCaml
- Scala
- Scheme



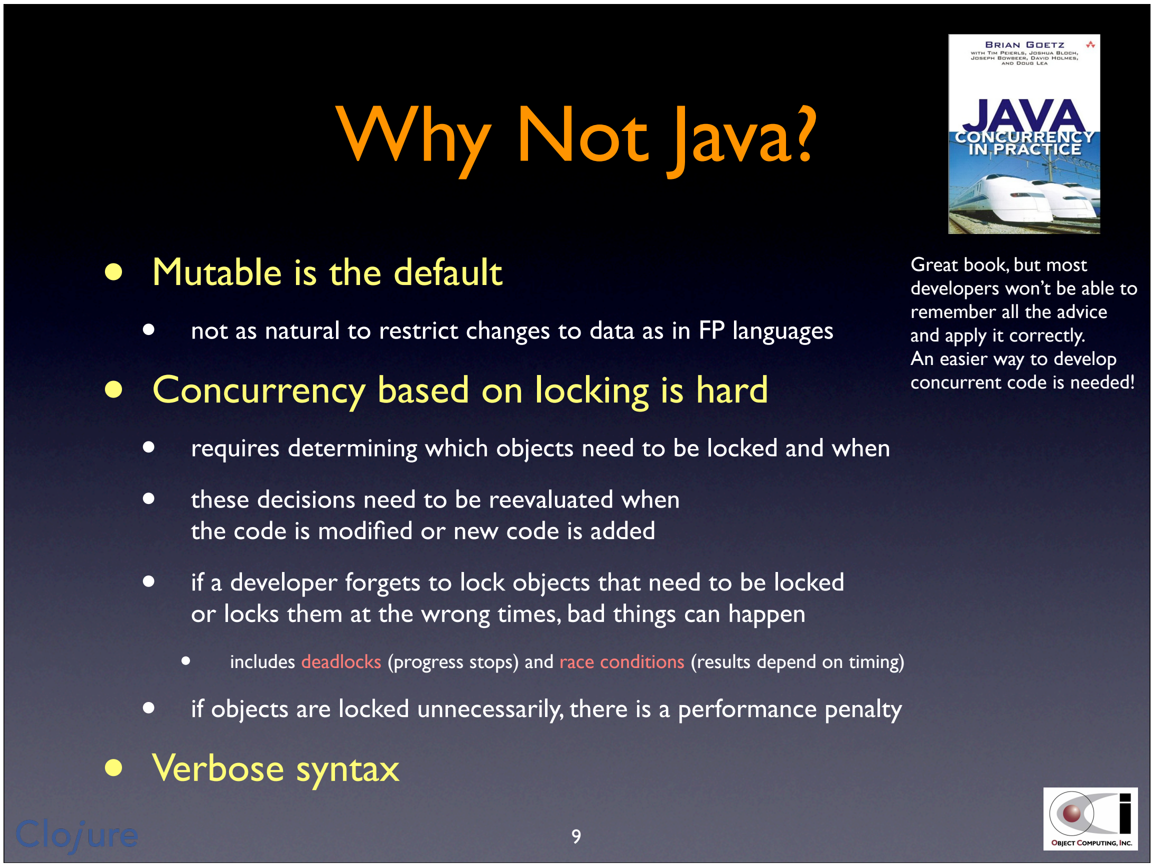
Concurrency

- Wikipedia definition
  - "Concurrency is a property of systems in which several computations are executing and overlapping in time, and potentially interacting with each other. The overlapping computations may be executing on multiple cores in the same chip, preemptively time-shared threads on the same processor, or executed on physically separated processors."
- Primary challenge
  - managing access to shared, mutable state



Why Functional Programming?

- Easier concurrency
  - immutable data doesn’t require locking for concurrent access
- Simpler code
  - pure functions are easier to write, test and debug
  - code is typically more brief

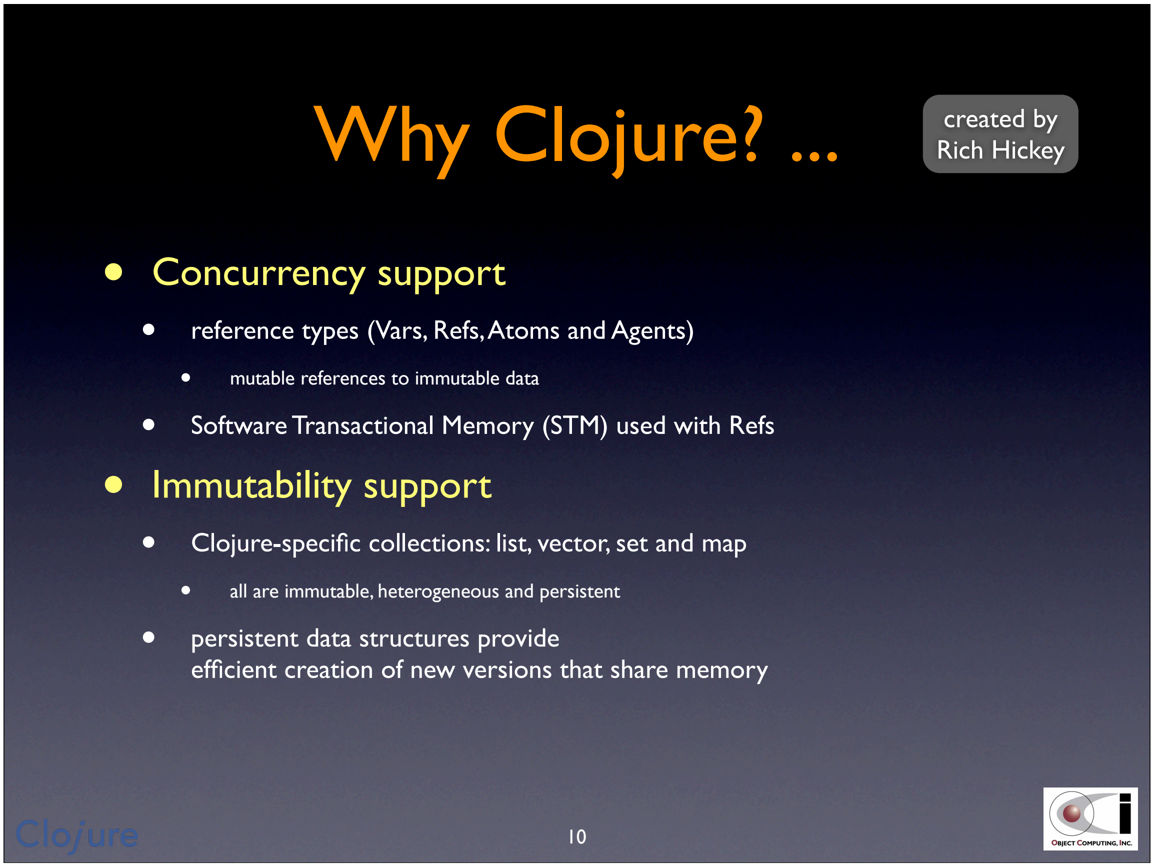

BRIAN GOETZ
WITH TIM PEIERLS, JOSHUA BLOCH, JOSEPH BOWBEER, DAVID HOLMES, AND DOUG LEA
JAVA
CONCURRENCY
IN PRACTICE

Why Not Java?

Great book, but most developers won't be able to remember all the advice and apply it correctly.
An easier way to develop concurrent code is needed!

- Mutable is the default
  - not as natural to restrict changes to data as in FP languages
- Concurrency based on locking is hard
  - requires determining which objects need to be locked and when
  - these decisions need to be reevaluated when the code is modified or new code is added
  - if a developer forgets to lock objects that need to be locked or locks them at the wrong times, bad things can happen
    - includes deadlocks (progress stops) and race conditions (results depend on timing)
  - if objects are locked unnecessarily, there is a performance penalty
- Verbose syntax

Clojure



Why Clojure? ...

created by
Rich Hickey

- Concurrency support
  - reference types (Vars, Refs, Atoms and Agents)
    - mutable references to immutable data
  - Software Transactional Memory (STM) used with Refs
- Immutability support
  - Clojure-specific collections: list, vector, set and map
    - all are immutable, heterogeneous and persistent
  - persistent data structures provide efficient creation of new versions that share memory

Clojure

...Why Clojure? ...

- Sequences
  - a common logical view of data including Java collections, Clojure collections, strings, streams, trees (including directory structures and XML)
  - supports lazy evaluation of sequences
- Runs on JVM (Java 5 or greater)
  - provides portability, stability, performance and security
- Java interoperability
  - can use libraries for capabilities such as I/O, concurrency, database access, GUIs, web frameworks and more

...Why Clojure?

- Dynamically typed
- Minimal, consistent, Lisp-based syntax
  - easy to write code that generates code
  - differs from Lisp in some ways to simplify and support Java interop.
  - all operations are one of
    - special forms (built-in functions known to compiler)
    - functions
    - macros
- Open source with a liberal license

Clojure Processing

- Read-time
  - reads Clojure source code
  - creates a data structure representation of the code
- Compile-time
  - expands macro calls into code
  - compiles data structure representation to Java bytecode
  - can also compile ahead of time (AOT)
- Run-time
  - executes bytecode

Code Comparison

- Java method call

```
methodName(arg1, arg2, arg3);
```

- Clojure function call

```
(function-name arg1 arg2 arg3)
```

This is referred to as a "form".
It uses prefix notation.
This allows what are binary operators in other languages to take any number of arguments.
Other than some syntactic sugar,
EVERYTHING in Clojure looks like this!
This includes function/macro definitions,
function/macro calls, variable bindings
and control structures.

- Java method definition

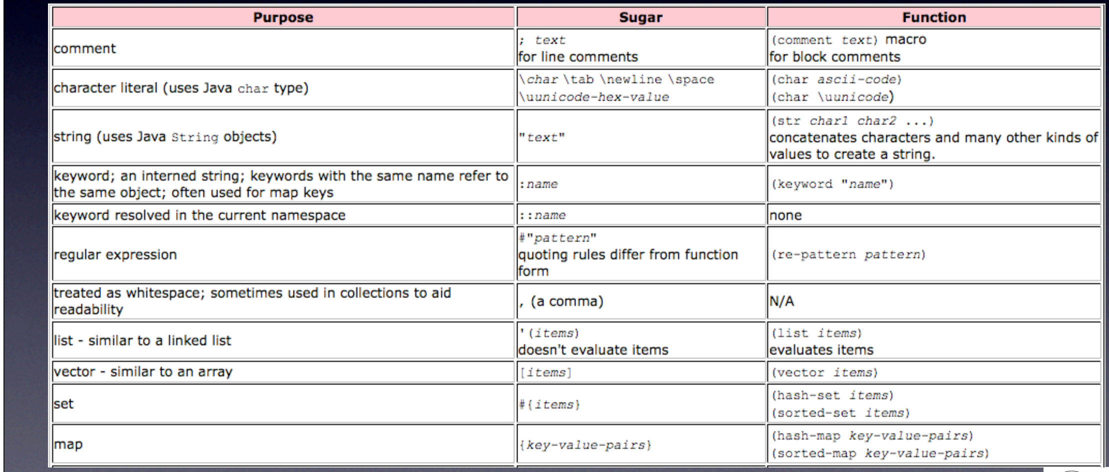
```
public void hello(String name) {
    System.out.println("Hello, " + name);
}
```

- Clojure function definition

```
(defn hello [name]
  (println "Hello," name))
```

Syntactic Sugar

- See http://ociweb.com/mark/clojure/article.html#Syntax



Purpose	Sugar	Function

comment	`; text`
	for line comments
		`(comment text)` macro
		for block comments

character literal (uses Java `char` type)	`\char \tab \newline \space`
	`\uunicode-hex-value`
		`(char ascii-code)`
		`(char \uunicode)`

string (uses Java `String` objects)	`"text"`	`(str char1 char2 ...)` concatenates characters and many other kinds of values to create a string.
keyword; an interned string; keywords with the same name refer to the same object; often used for map keys	`:name`	`(keyword "name")`
keyword resolved in the current namespace	`::name`	none
regular expression	`#"pattern"` quoting rules differ from function form	`(re-pattern pattern)`
treated as whitespace; sometimes used in collections to aid readability	, (a comma)	N/A
list - similar to a linked list	`'(items)` doesn't evaluate items	`(list items)` evaluates items
vector - similar to an array	`[items]`	`(vector items)`

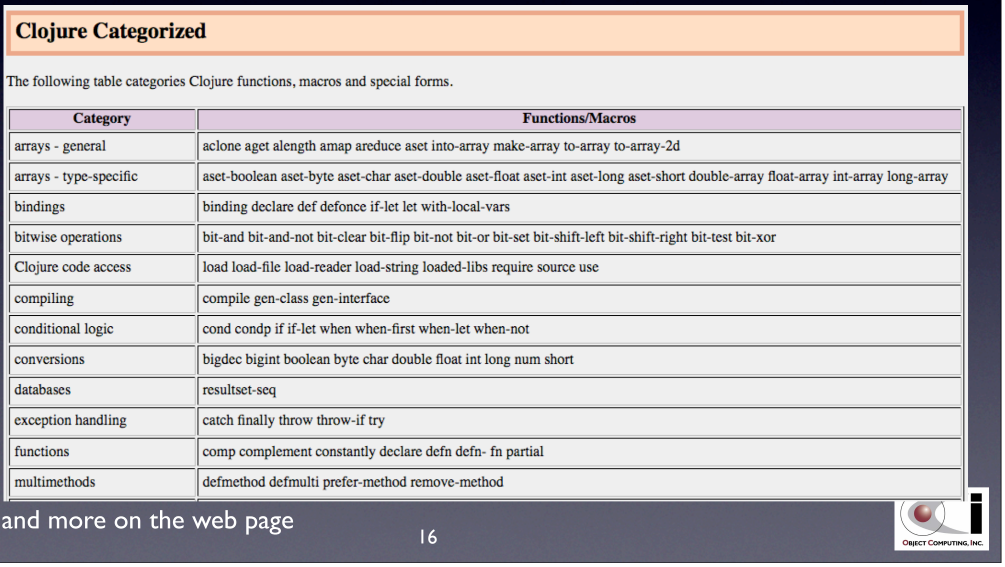
set	`#{items}`	`(hash-set items)`
		`(sorted-set items)`

map	`{key-value-pairs}`	`(hash-map key-value-pairs)`
		`(sorted-map key-value-pairs)`

and more on the web page

Provided Functions/Macros

- See http://ociweb.com/mark/clojure/ClojureCategorized.html

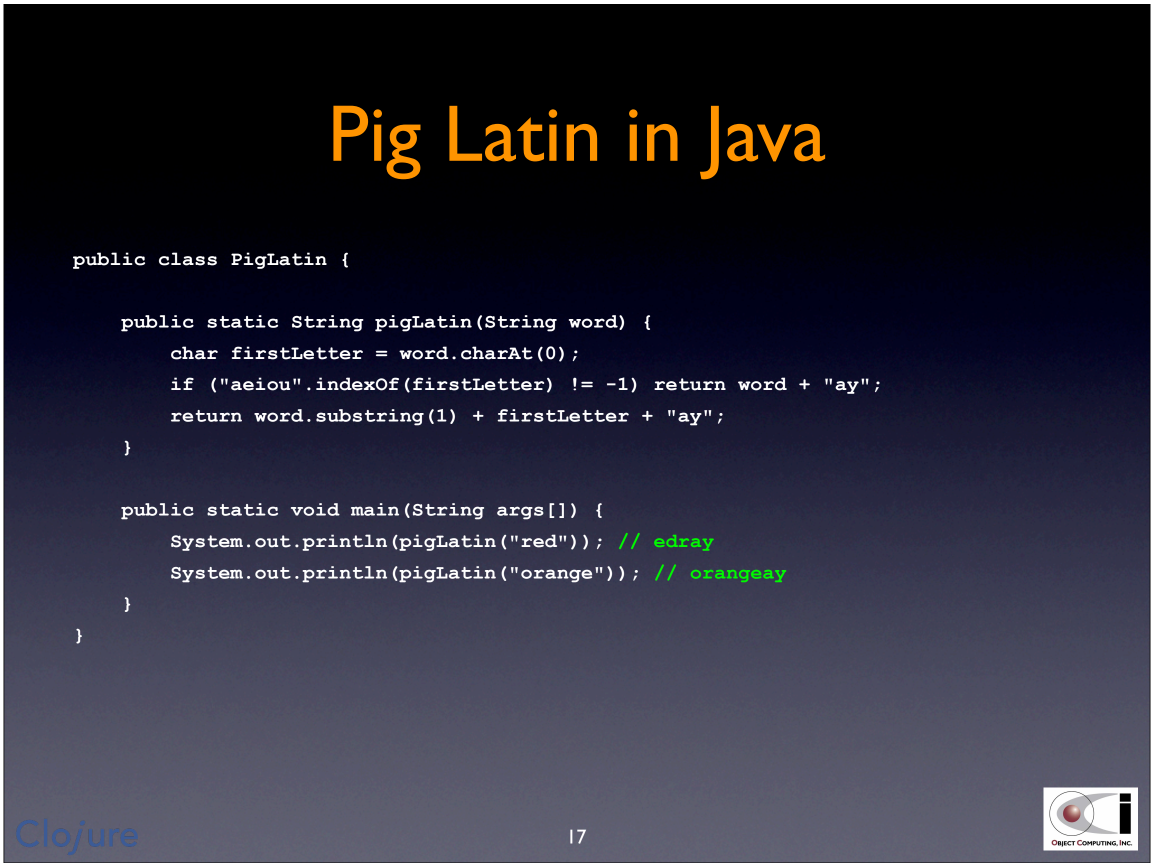


Clojure Categorized

The following table categories Clojure functions, macros and special forms.

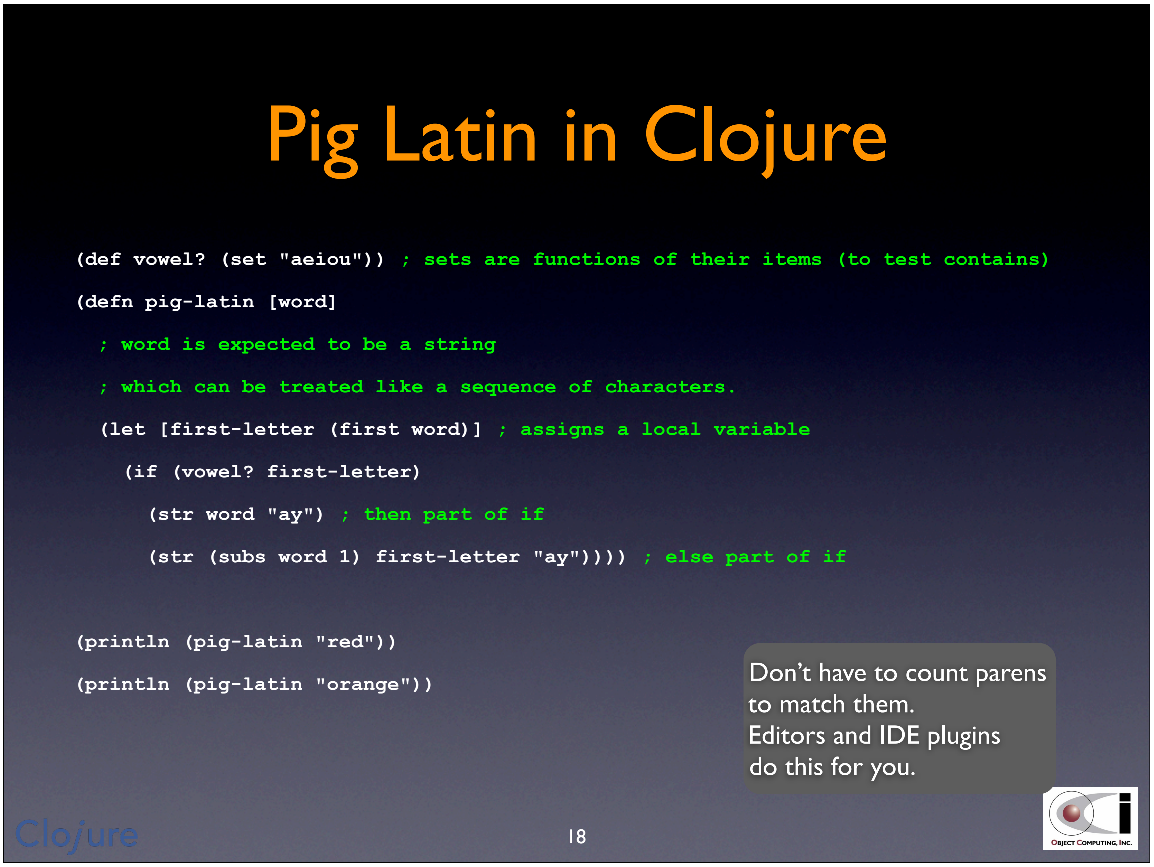
Category	Functions/Macros
arrays - general	aclone aget alength amap areduce aset into-array make-array to-array to-array-2d
arrays - type-specific	aset-boolean aset-byte aset-char aset-double aset-float aset-int aset-long aset-short double-array float-array int-array long-array
bindings	binding declare def defonce if-let let with-local-vars
bitwise operations	bit-and bit-and-not bit-clear bit-flip bit-not bit-or bit-set bit-shift-left bit-shift-right bit-test bit-xor
Clojure code access	load load-file load-reader load-string loaded-libs require source use
compiling	compile gen-class gen-interface
conditional logic	cond condp if if-let when when-first when-let when-not
conversions	bigdec bigint boolean byte char double float int long num short
databases	resultset-seq
exception handling	catch finally throw throw-if try
functions	comp complement constantly declare defn defn- fn partial
multimethods	defmethod defmulti prefer-method remove-method

and more on the web page



Pig Latin in Java

```
public class PigLatin {

    public static String pigLatin(String word) {
        char firstLetter = word.charAt(0);
        if ("aeiou".indexOf(firstLetter) != -1) return word + "ay";
        return word.substring(1) + firstLetter + "ay";
    }

    public static void main(String args[]) {
        System.out.println(pigLatin("red")); // edray
        System.out.println(pigLatin("orange")); // orangeay
    }
}
```



Pig Latin in Clojure

```
(def vowel? (set "aeiou")) ; sets are functions of their items (to test contains)
(defn pig-latin [word]
  ; word is expected to be a string
  ; which can be treated like a sequence of characters.
  (let [first-letter (first word)] ; assigns a local variable
    (if (vowel? first-letter)
      (str word "ay") ; then part of if
      (str (subs word 1) first-letter "ay")))) ; else part of if

(println (pig-latin "red"))
(println (pig-latin "orange"))
```

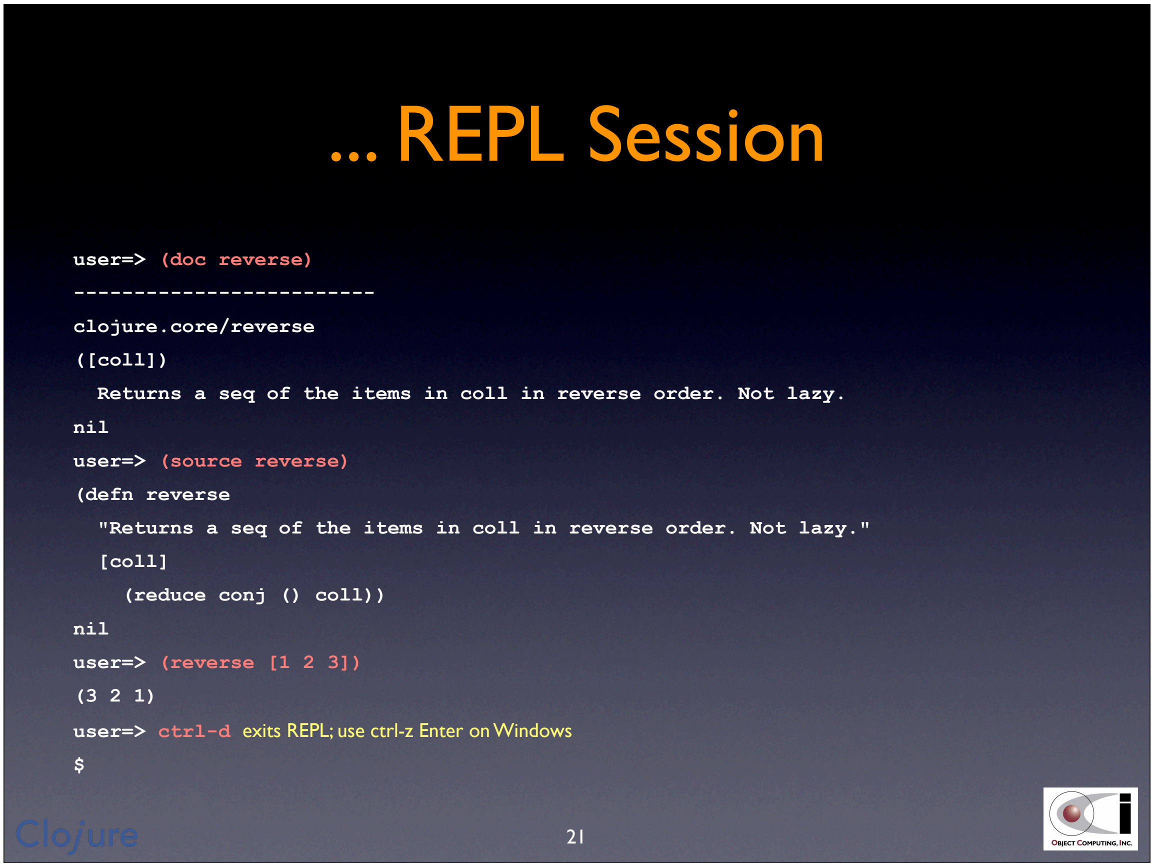
Don't have to count parens
to match them.
Editors and IDE plugins
do this for you.

Getting Started

- See http://ociweb.com/mark/clojure/article.html#GettingStarted
- Download latest code from Subversion
- Build using Ant
- Create a clj script
  - starts a read-eval-print loop (REPL)
    - standard tool used by Lisp dialects to experiment with code
  - runs Clojure source files (.clj extension)
  - adds frequently used JARs to classpath
  - adds editing features using rlwrap or JLine
  - adds use of a startup script for other customizations

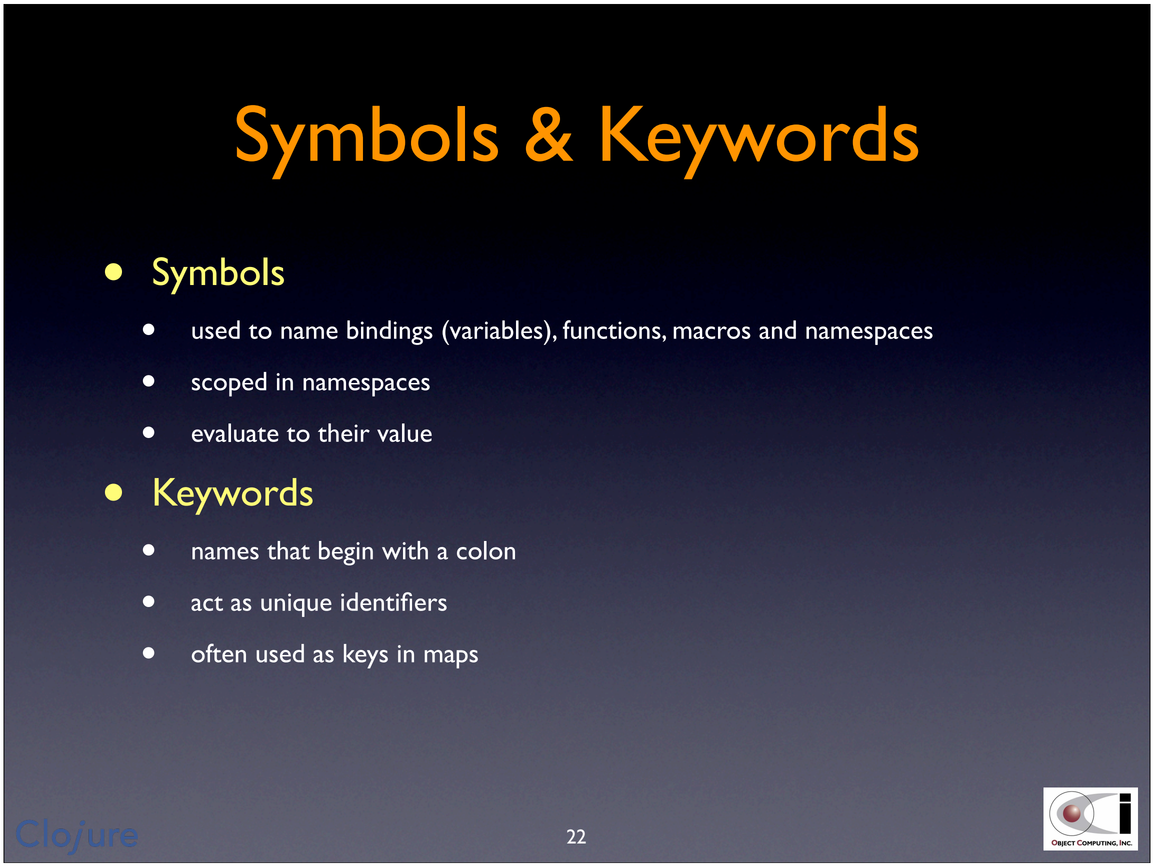
REPL Session ...

```
$ clj  starts REPL session
user=>  default namespace is “user”
user=> (def x 2)  defines a Var named x with a value of 2
#'user/x  string representation of the Var
user=> (* x 3 4)  multiplies x by 3 and 4
24  result
user=> (load-file "src/demo.clj")  loads given source file making its definitions available
user=> (find-doc "reverse")  prints documentation for all functions that contain the given string
                             in their name or documentation string
```



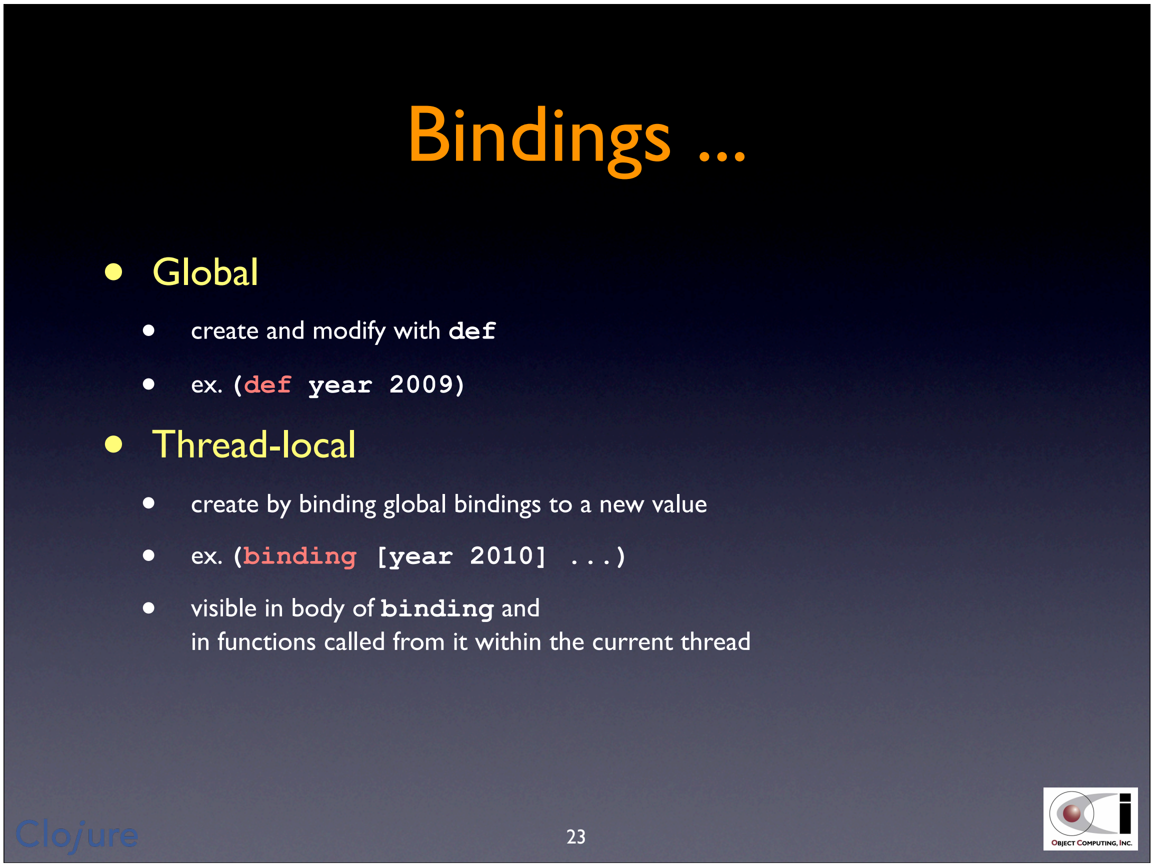
… REPL Session

```
user=> (doc reverse)
-------------------------
clojure.core/reverse
([coll])
  Returns a seq of the items in coll in reverse order. Not lazy.
nil
user=> (source reverse)
(defn reverse
  "Returns a seq of the items in coll in reverse order. Not lazy."
  [coll]
    (reduce conj () coll))
nil
user=> (reverse [1 2 3])
(3 2 1)
user=> ctrl-d exits REPL; use ctrl-z Enter on Windows
$
```



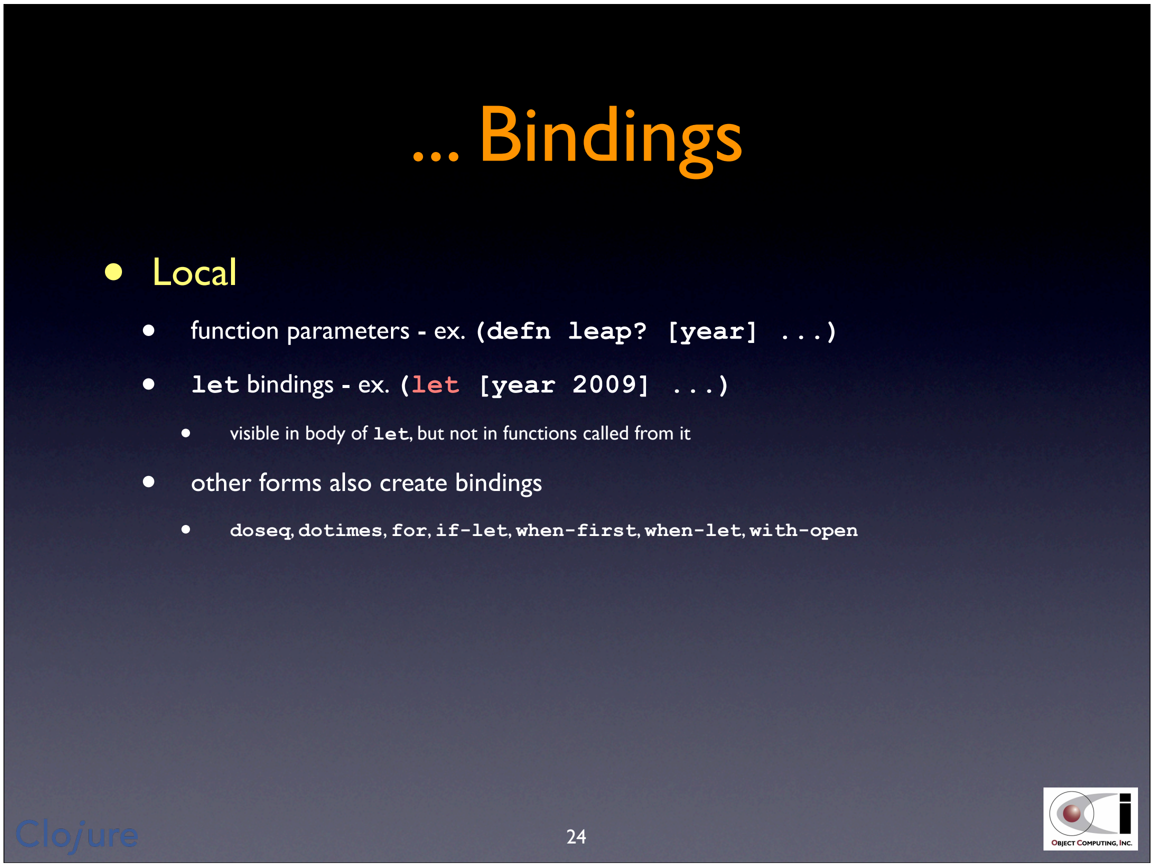
Symbols & Keywords

- Symbols
  - used to name bindings (variables), functions, macros and namespaces
  - scoped in namespaces
  - evaluate to their value
- Keywords
  - names that begin with a colon
  - act as unique identifiers
  - often used as keys in maps



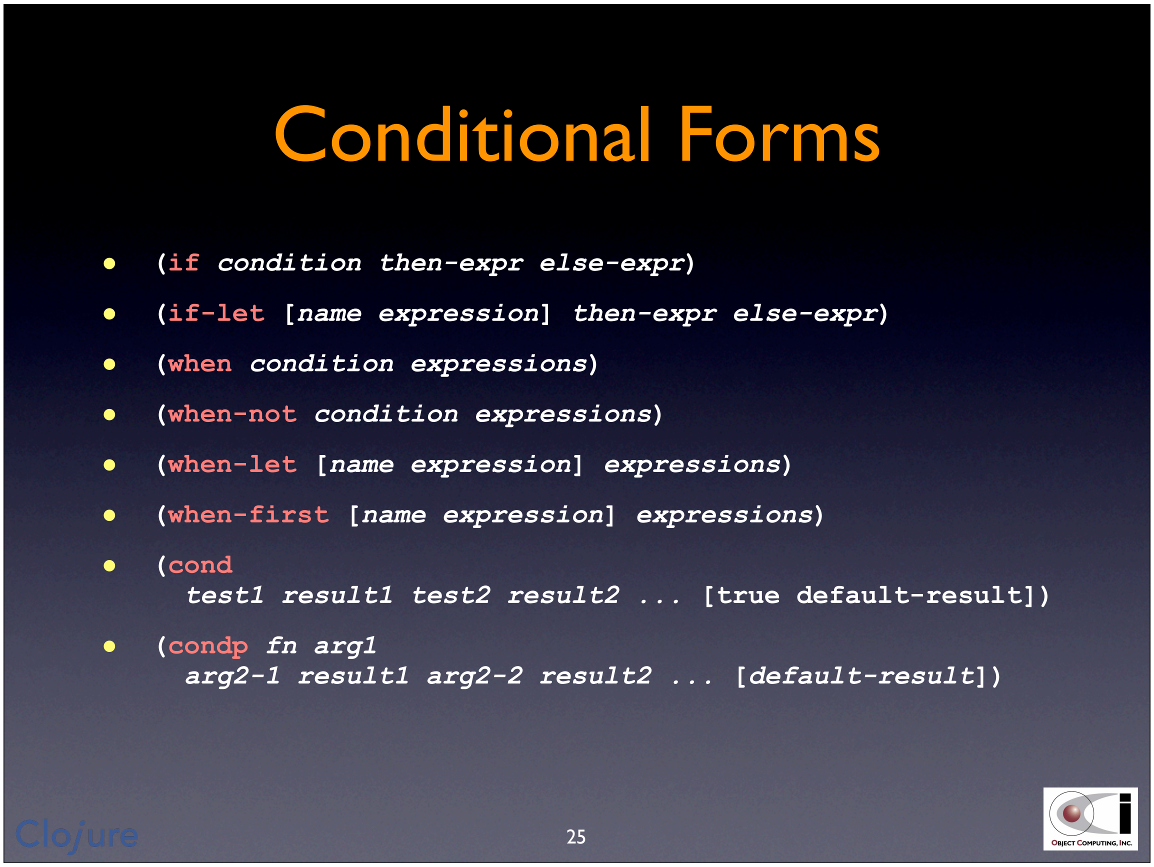
Bindings ...

- Global
  - create and modify with `def`
  - ex. `(def year 2009)`
- Thread-local
  - create by binding global bindings to a new value
  - ex. `(binding [year 2010] ...)`
  - visible in body of `binding` and
    in functions called from it within the current thread



... Bindings

- Local
  - function parameters - ex. `(defn leap? [year] ...)`
  - `let` bindings - ex. `(let [year 2009] ...)`
    - visible in body of `let`, but not in functions called from it
  - other forms also create bindings
    - `doseq`, `dotimes`, `for`, `if-let`, `when-first`, `when-let`, `with-open`

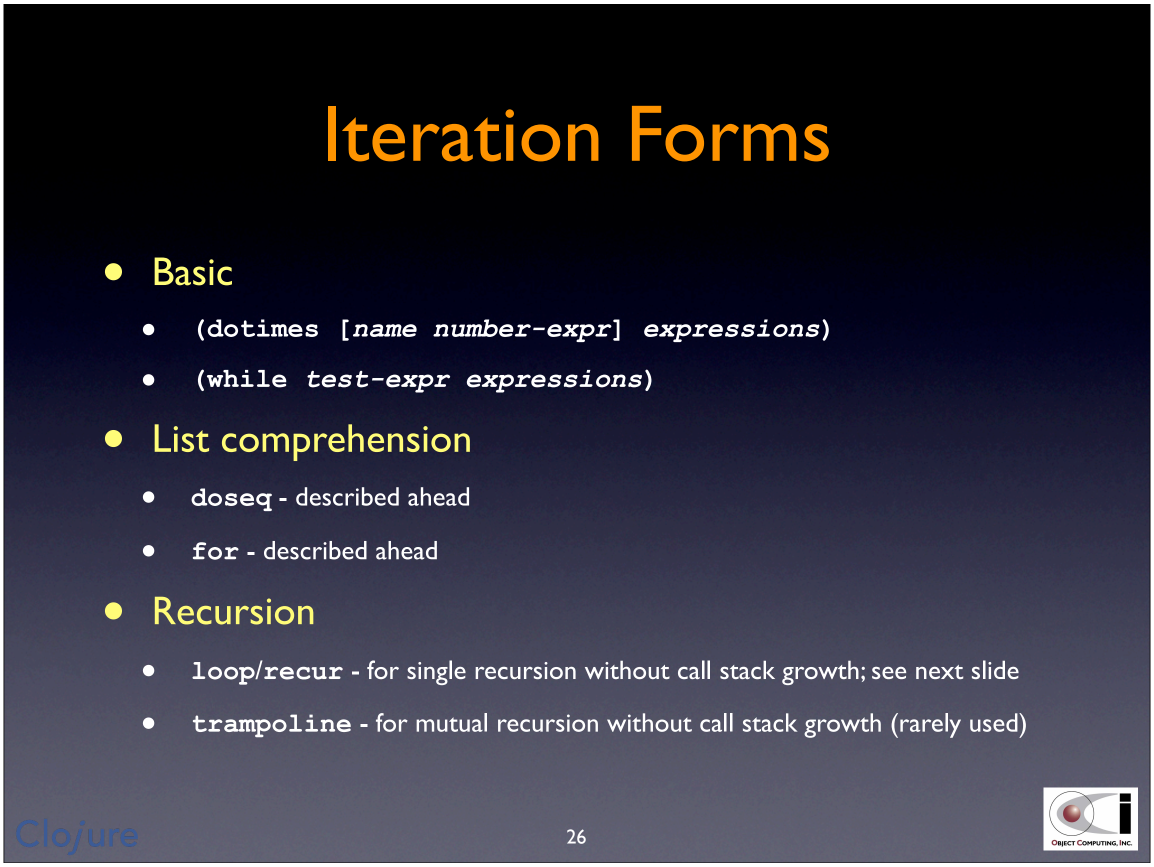


Conditional Forms

- `(if condition then-expr else-expr)`
- `(if-let [name expression] then-expr else-expr)`
- `(when condition expressions)`
- `(when-not condition expressions)`
- `(when-let [name expression] expressions)`
- `(when-first [name expression] expressions)`

-
  ```
  (cond
    test1 result1 test2 result2 ... [true default-result])
  ```

-
  ```
  (condp fn arg1
    arg2-1 result1 arg2-2 result2 ... [default-result])
  ```



Iteration Forms

- Basic
  - `(dotimes [name number-expr] expressions)`
  - `(while test-expr expressions)`
- List comprehension
  - `doseq` - described ahead
  - `for` - described ahead
- Recursion
  - `loop/recur` - for single recursion without call stack growth; see next slide
  - `trampoline` - for mutual recursion without call stack growth (rarely used)

loop / recur ...

- Converts what would otherwise be a tail recursive call to a loop that doesn't consume stack space
- `loop`
  - creates initial bindings and establishes a recursion point
- `recur`
  - returns to the recursion point with new bindings
- containing function definition
  - also establishes a recursion point
  - can use **`recur`** without **`loop`** to recur to beginning of function

... loop / recur

- Example

```
(loop [m 2 n 0]
  (println m n)
  (when-not (zero? m)
    (recur (dec m) (inc n))))
```

- Output

```
2 0
1 1
0 2
```

List Comprehension ...

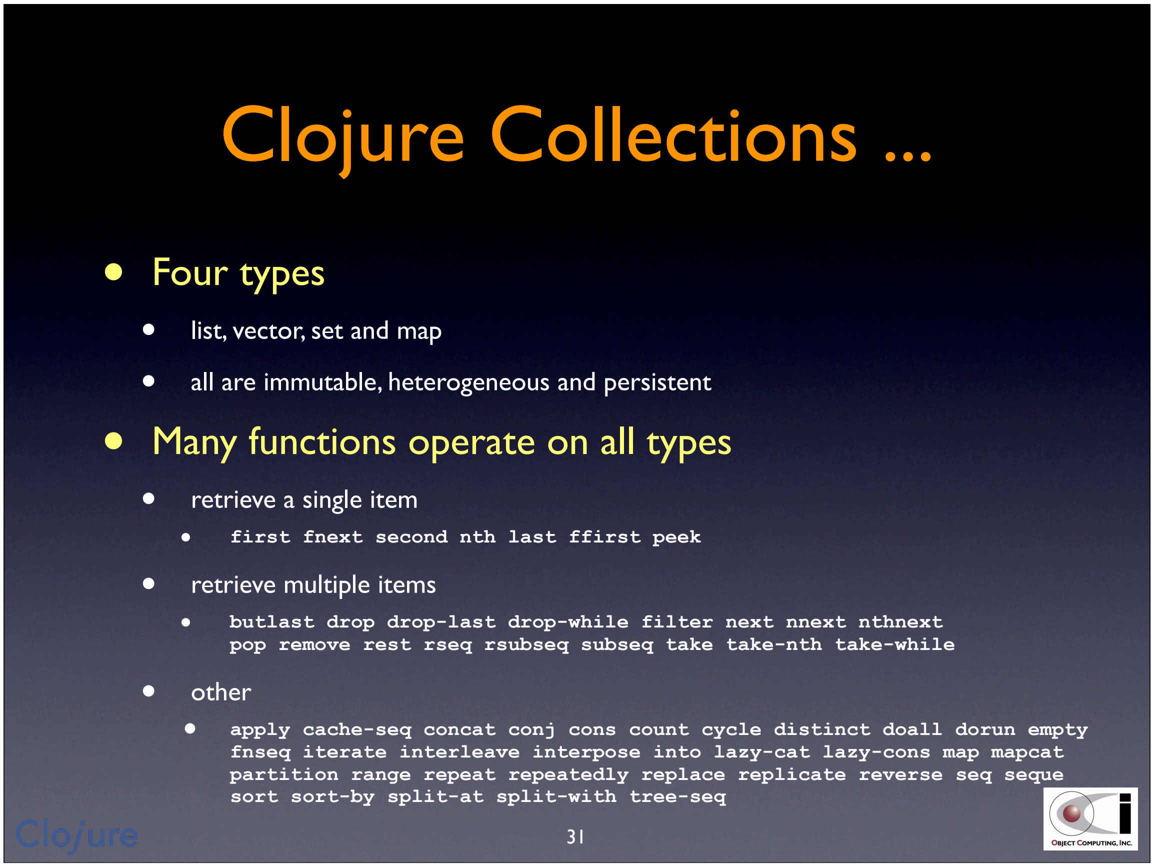
- Iterates through one or more sequences
- Two types
  - `for` returns a lazy sequence of results
  - `doseq` forces evaluation for side effects; returns nil
- Specify filtering with `:when` and `:while`
  - using `:when` causes only items that meet a condition to be processed
  - using `:while` causes iteration to stop
    when an item is reached that fails the condition

List Comprehension

```
(def cols "ABCD")
(def rows (range 1 4)) ; purposely larger than needed to demonstrate :while

(doseq [col cols :when (not= col \B)
        row rows :while (< row 3)]
  (println (str col row)))

(dorun ; forces evaluation of lazy sequence returned by for
  (for [col cols :when (not= col \B)
        row rows :while (< row 3)]
    (println (str col row))))
```

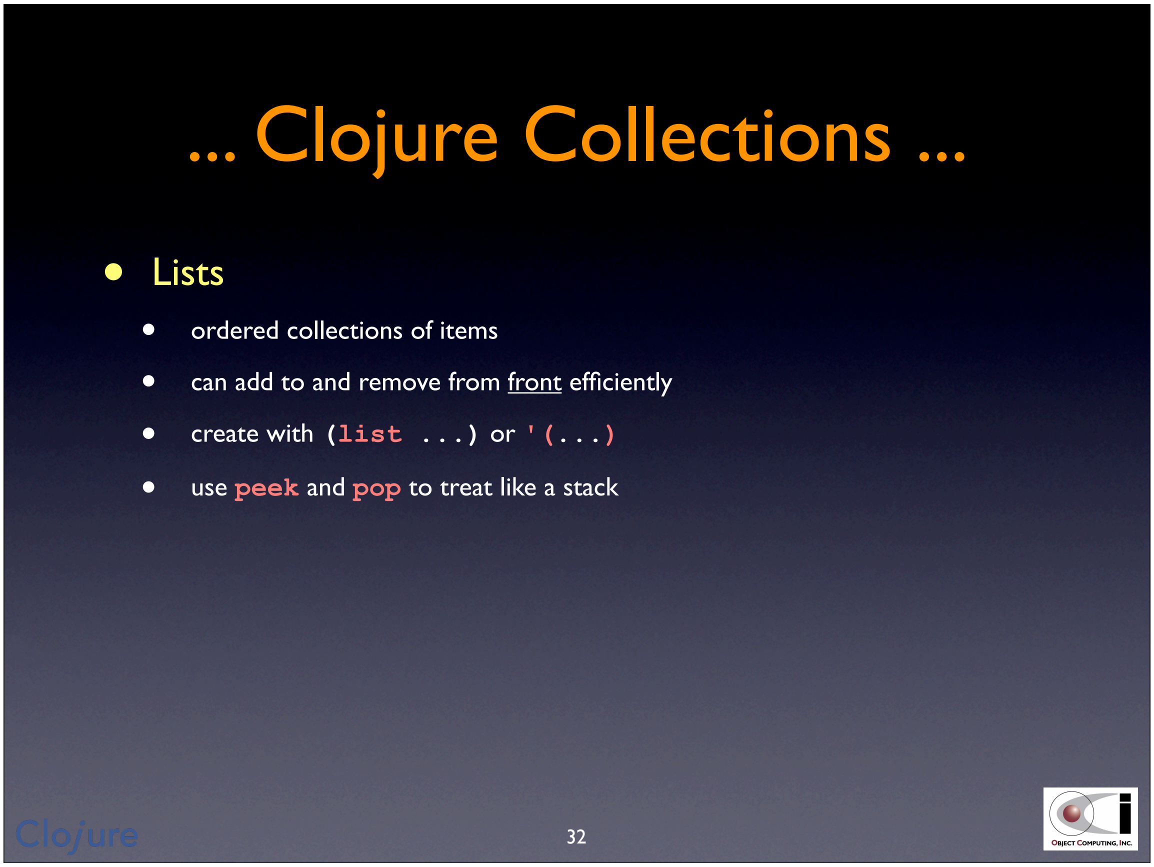
Output from both:

```
A1
A2
C1
C2
D1
D2
```



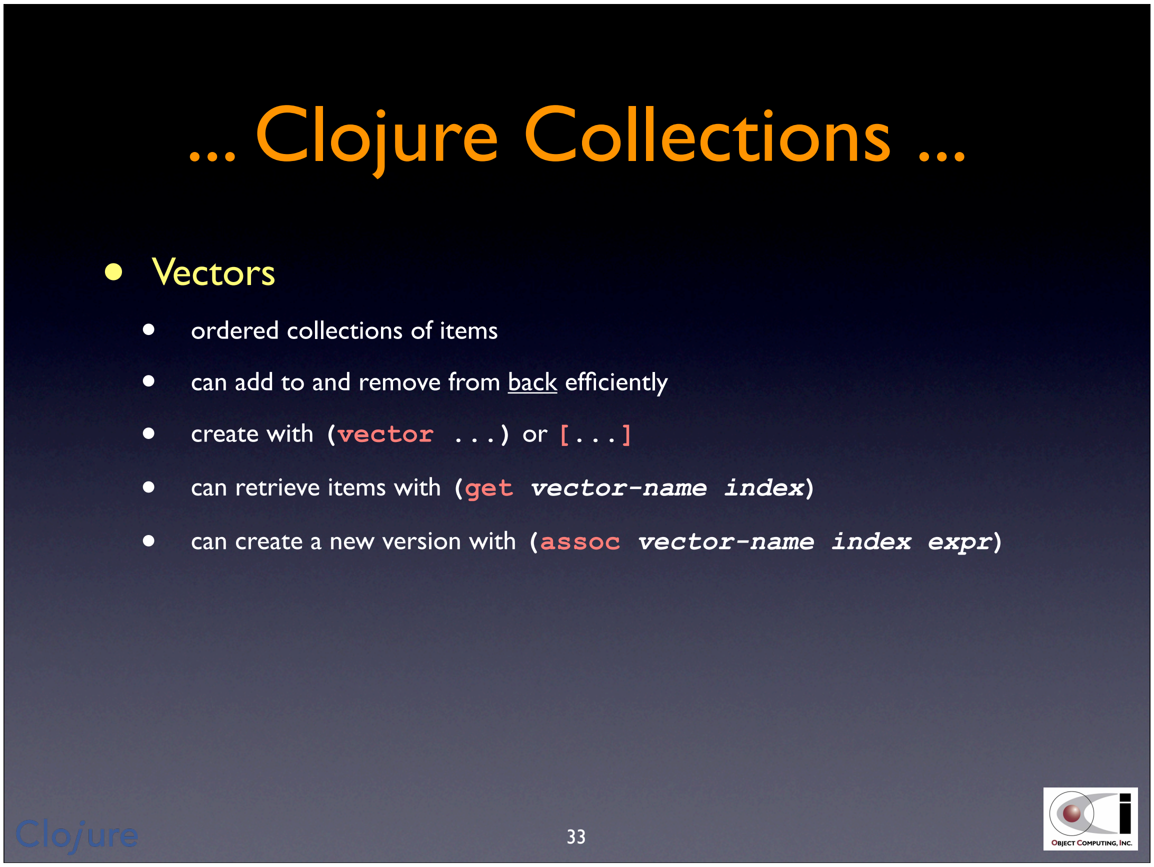
Clojure Collections ...

- Four types
  - list, vector, set and map
  - all are immutable, heterogeneous and persistent
- Many functions operate on all types
  - retrieve a single item
    - `first fnext second nth last ffirst peek`
  - retrieve multiple items
    - `butlast drop drop-last drop-while filter next nnext nthnext pop remove rest rseq rsubseq subseq take take-nth take-while`
  - other
    - `apply cache-seq concat conj cons count cycle distinct doall dorun empty fnseq iterate interleave interpose into lazy-cat lazy-cons map mapcat partition range repeat repeatedly replace replicate reverse seq seque sort sort-by split-at split-with tree-seq`



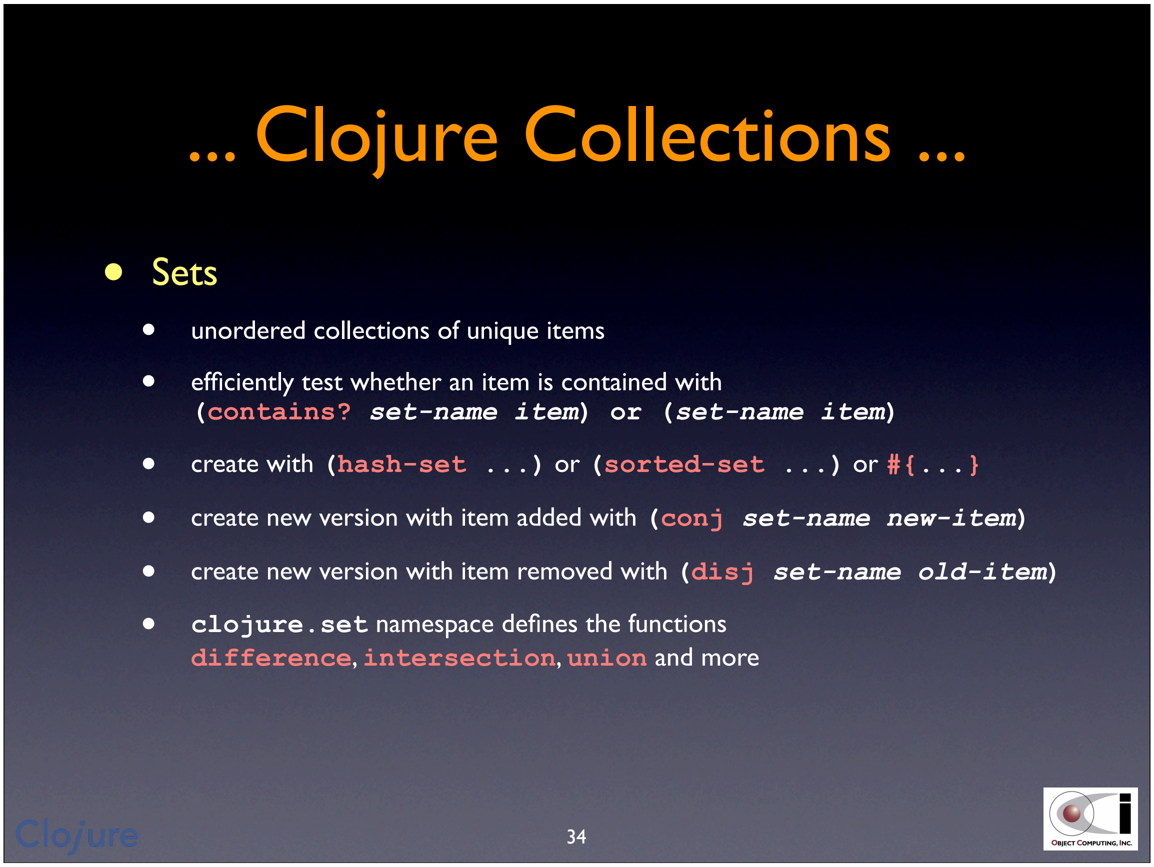
... Clojure Collections ...

- Lists
  - ordered collections of items
  - can add to and remove from front efficiently
  - create with `(list ...)` or `'(...)`
  - use `peek` and `pop` to treat like a stack



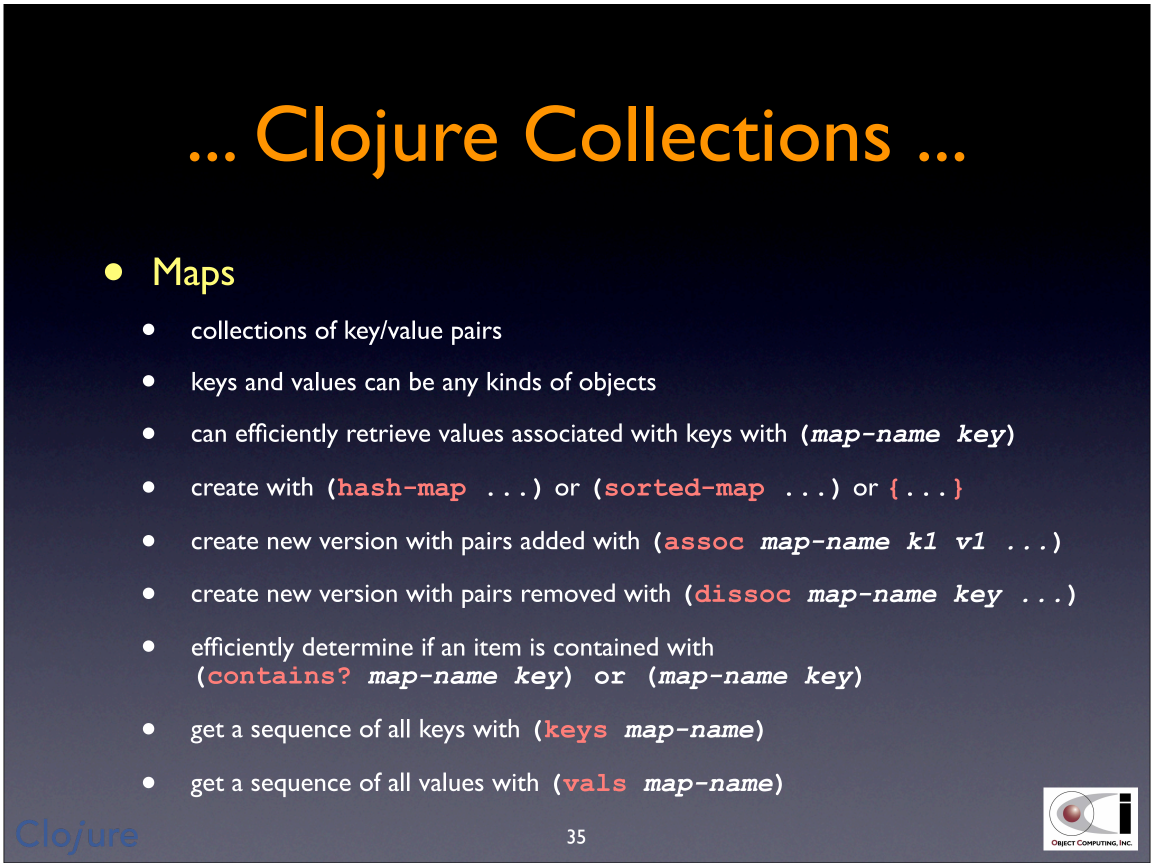
... Clojure Collections ...

- Vectors
  - ordered collections of items
  - can add to and remove from <u>back</u> efficiently
  - create with `(vector ...)` or `[...]`
  - can retrieve items with `(get vector-name index)`
  - can create a new version with `(assoc vector-name index expr)`



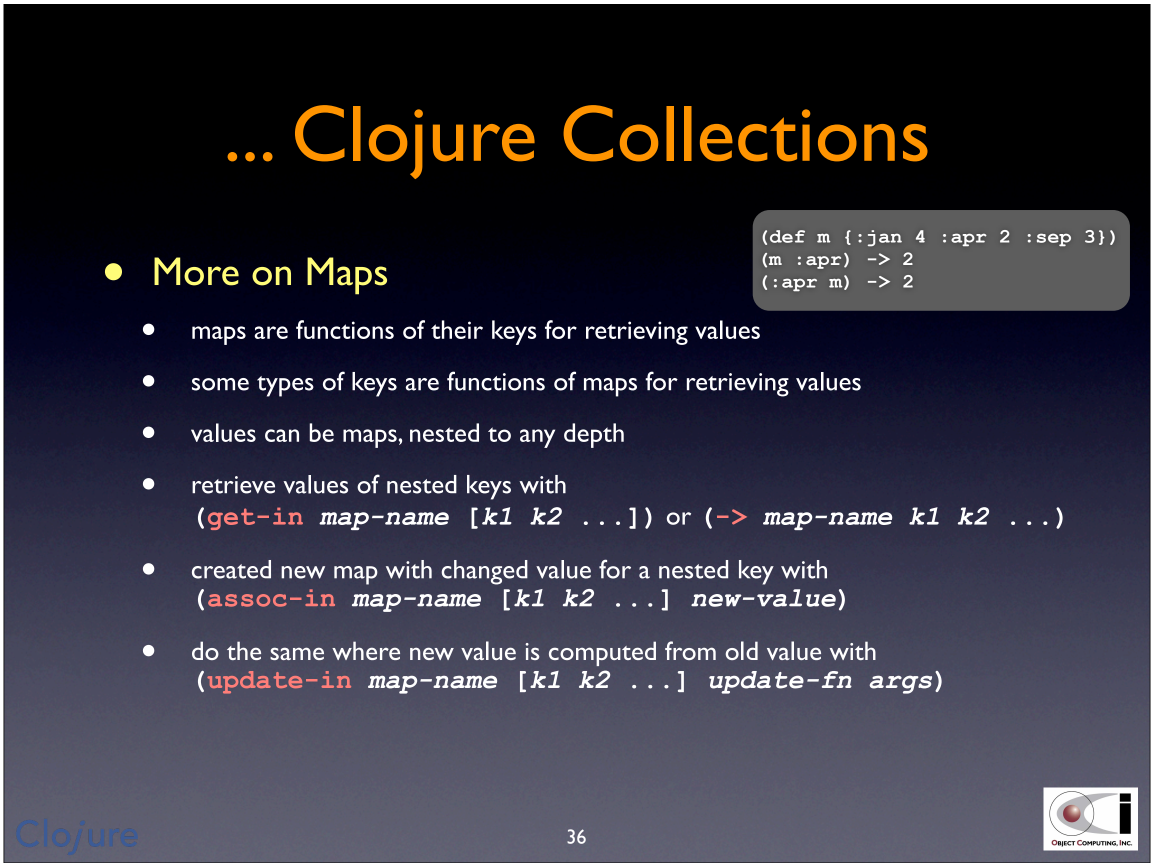
... Clojure Collections ...

- Sets
  - unordered collections of unique items
  - efficiently test whether an item is contained with `(contains? set-name item) or (set-name item)`
  - create with `(hash-set ...)` or `(sorted-set ...)` or `#{...}`
  - create new version with item added with `(conj set-name new-item)`
  - create new version with item removed with `(disj set-name old-item)`
  - `clojure.set` namespace defines the functions `difference`, `intersection`, `union` and more



… Clojure Collections …

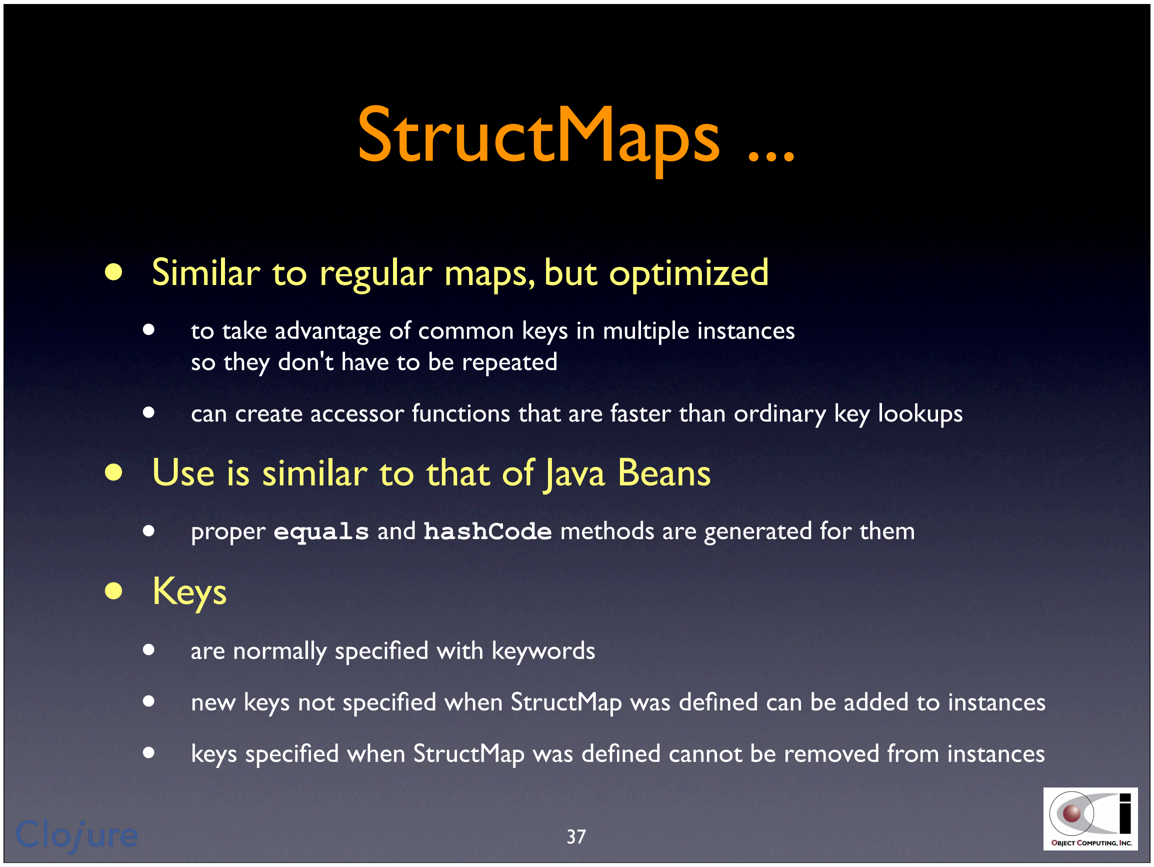
- Maps
  - collections of key/value pairs
  - keys and values can be any kinds of objects
  - can efficiently retrieve values associated with keys with `(map-name key)`
  - create with `(hash-map ...)` or `(sorted-map ...)` or `{...}`
  - create new version with pairs added with `(assoc map-name k1 v1 ...)`
  - create new version with pairs removed with `(dissoc map-name key ...)`
  - efficiently determine if an item is contained with `(contains? map-name key) or (map-name key)`
  - get a sequence of all keys with `(keys map-name)`
  - get a sequence of all values with `(vals map-name)`



… Clojure Collections

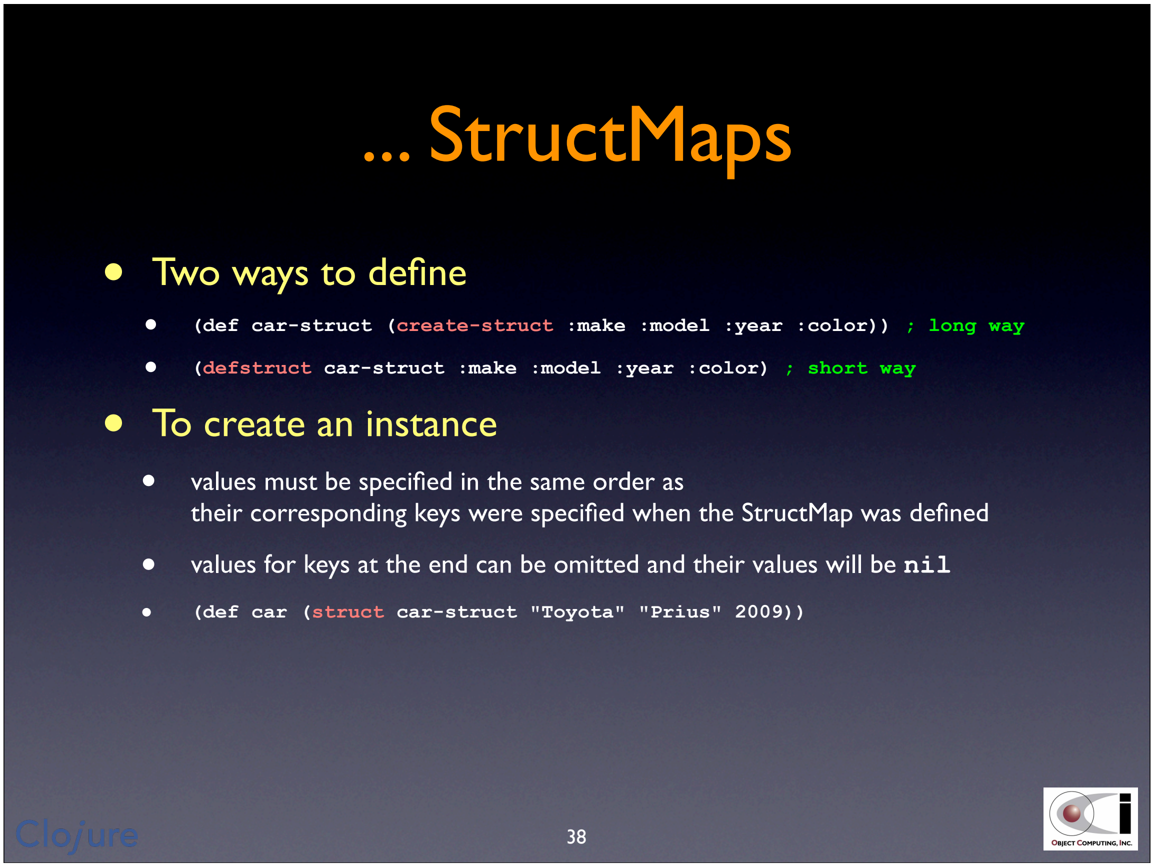
```
(def m {:jan 4 :apr 2 :sep 3})
(m :apr) -> 2
(:apr m) -> 2
```

- More on Maps
  - maps are functions of their keys for retrieving values
  - some types of keys are functions of maps for retrieving values
  - values can be maps, nested to any depth
  - retrieve values of nested keys with `(get-in map-name [k1 k2 ...])` or `(-> map-name k1 k2 ...)`
  - created new map with changed value for a nested key with `(assoc-in map-name [k1 k2 ...] new-value)`
  - do the same where new value is computed from old value with `(update-in map-name [k1 k2 ...] update-fn args)`



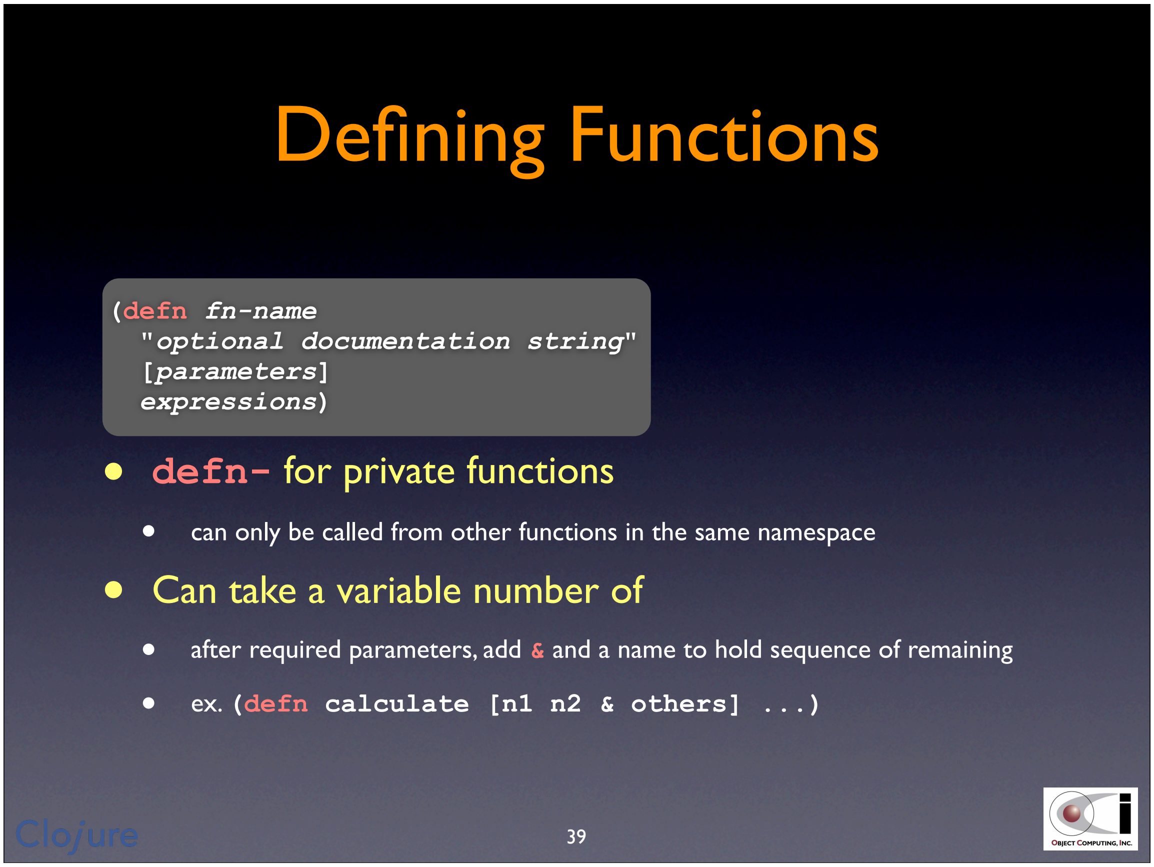
StructMaps ...

- Similar to regular maps, but optimized
  - to take advantage of common keys in multiple instances
    so they don't have to be repeated
  - can create accessor functions that are faster than ordinary key lookups
- Use is similar to that of Java Beans
  - proper **`equals`** and **`hashCode`** methods are generated for them
- Keys
  - are normally specified with keywords
  - new keys not specified when StructMap was defined can be added to instances
  - keys specified when StructMap was defined cannot be removed from instances



... StructMaps

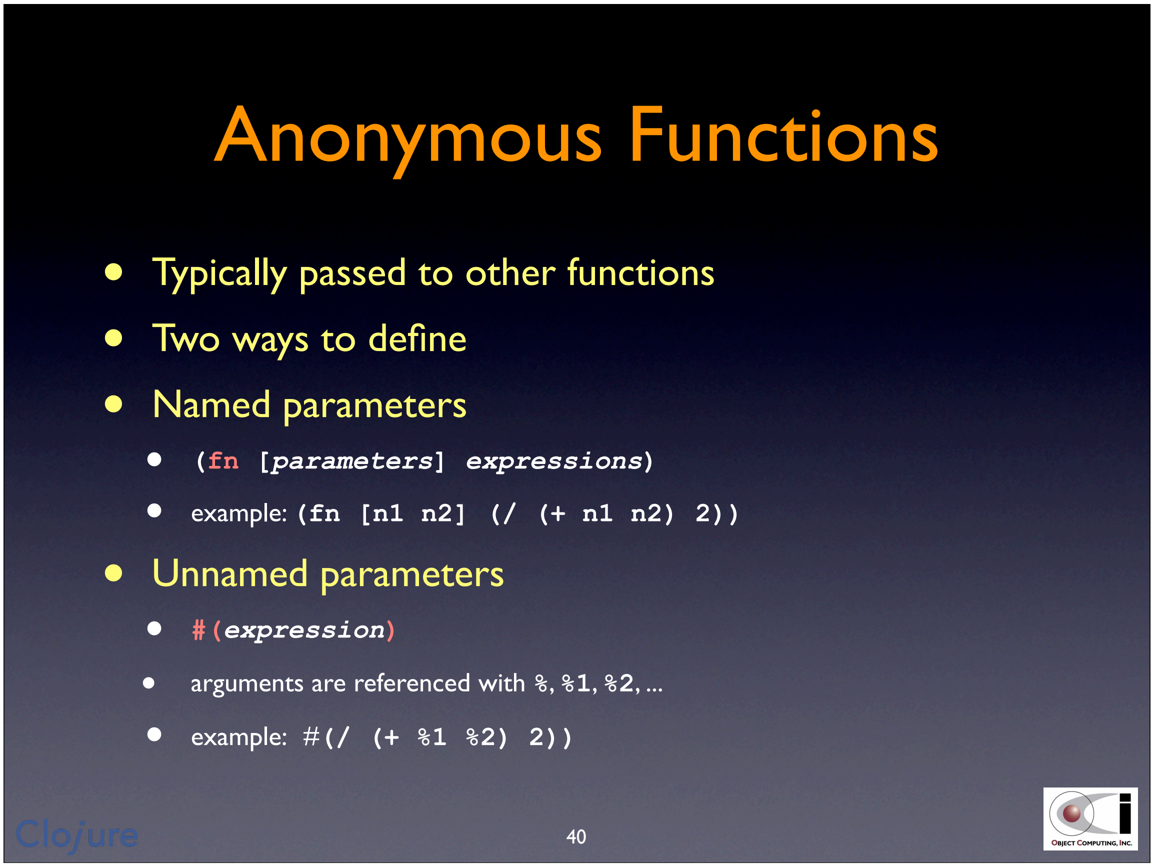
- Two ways to define
  - `(def car-struct (create-struct :make :model :year :color)) ; long way`
  - `(defstruct car-struct :make :model :year :color) ; short way`
- To create an instance
  - values must be specified in the same order as
    their corresponding keys were specified when the StructMap was defined
  - values for keys at the end can be omitted and their values will be **`nil`**
  - `(def car (struct car-struct "Toyota" "Prius" 2009))`



Defining Functions

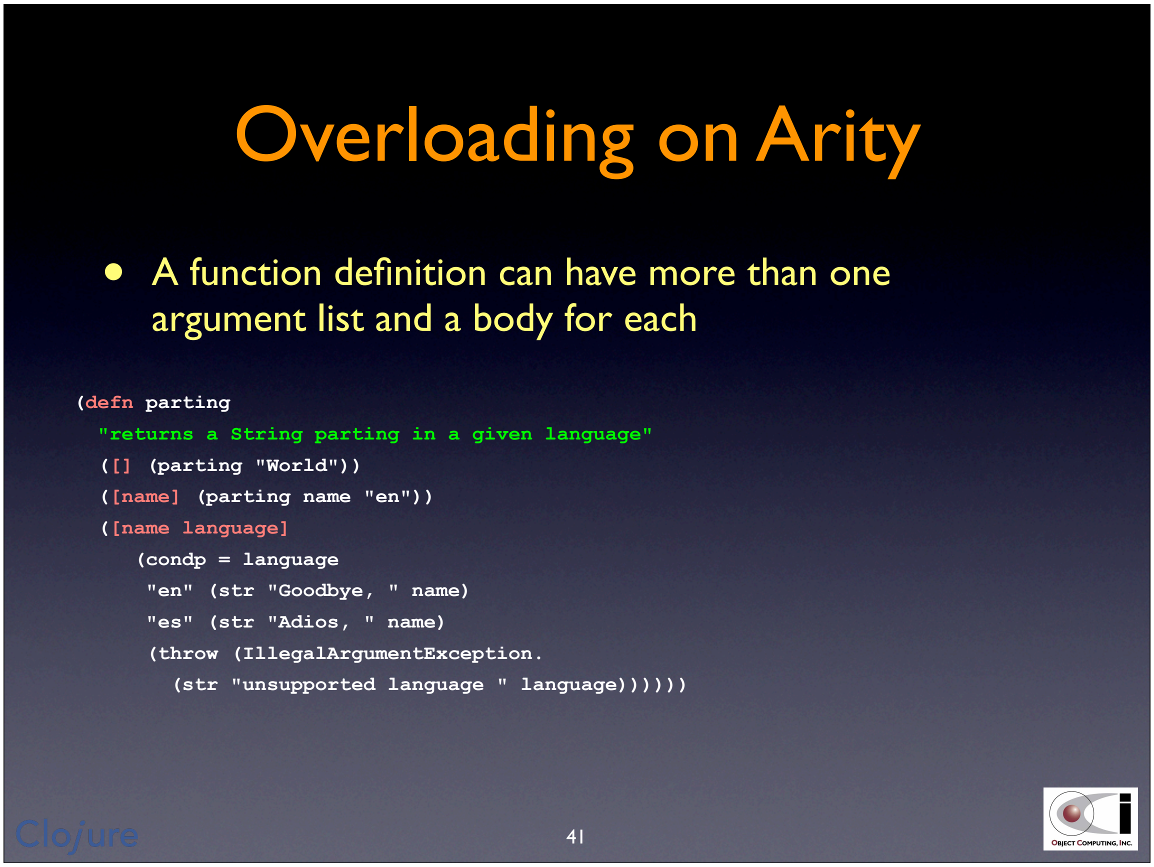
```
(defn fn-name
  "optional documentation string"
  [parameters]
  expressions)
```

- `defn-` for private functions
  - can only be called from other functions in the same namespace
- Can take a variable number of
  - after required parameters, add `&` and a name to hold sequence of remaining
  - ex. `(defn calculate [n1 n2 & others] ...)`



Anonymous Functions

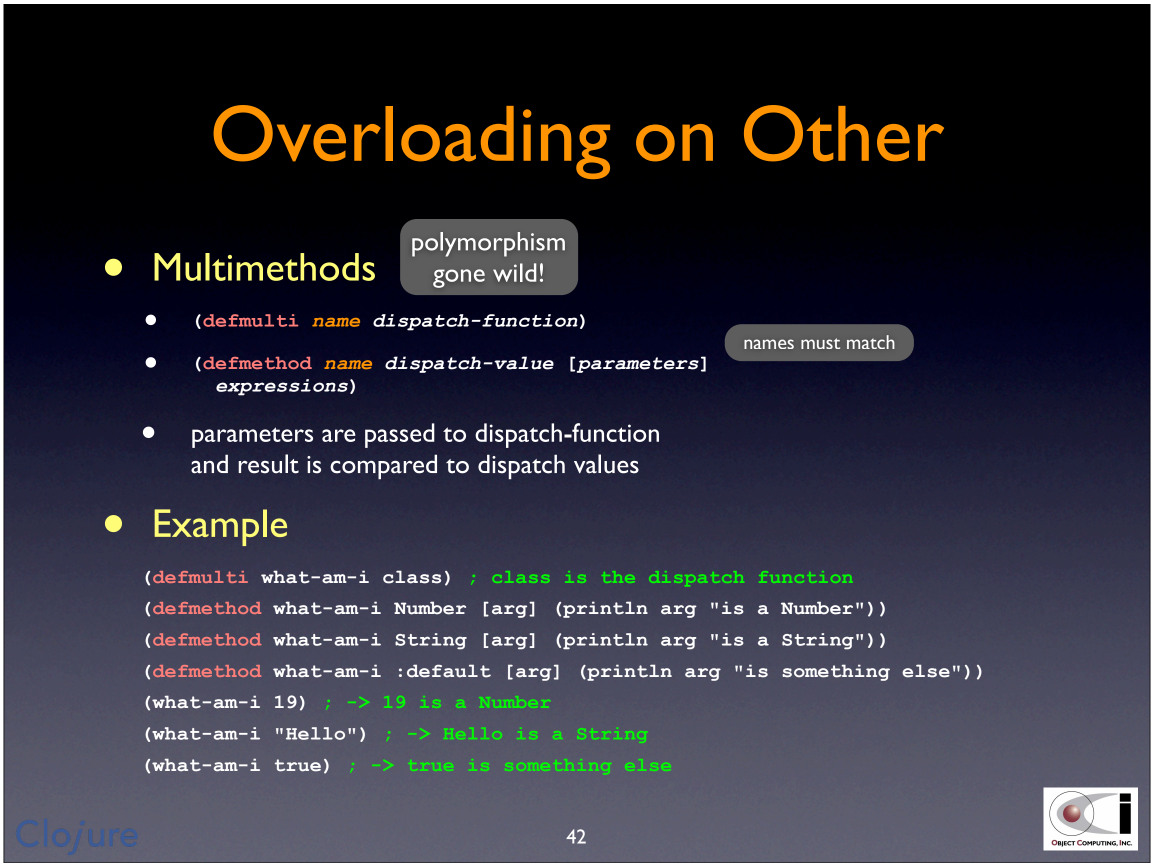
- Typically passed to other functions
- Two ways to define
- Named parameters
  - `(fn [parameters] expressions)`
  - example: `(fn [n1 n2] (/ (+ n1 n2) 2))`
- Unnamed parameters
  - `#(expression)`
  - arguments are referenced with `%`, `%1`, `%2`, ...
  - example: `#(/ (+ %1 %2) 2))`



Overloading on Arity

- A function definition can have more than one argument list and a body for each

```
(defn parting
  "returns a String parting in a given language"
  ([] (parting "World"))
  ([name] (parting name "en"))
  ([name language]
     (condp = language
      "en" (str "Goodbye, " name)
      "es" (str "Adios, " name)
      (throw (IllegalArgumentException.
        (str "unsupported language " language))))))
```



Overloading on Other

- Multimethods

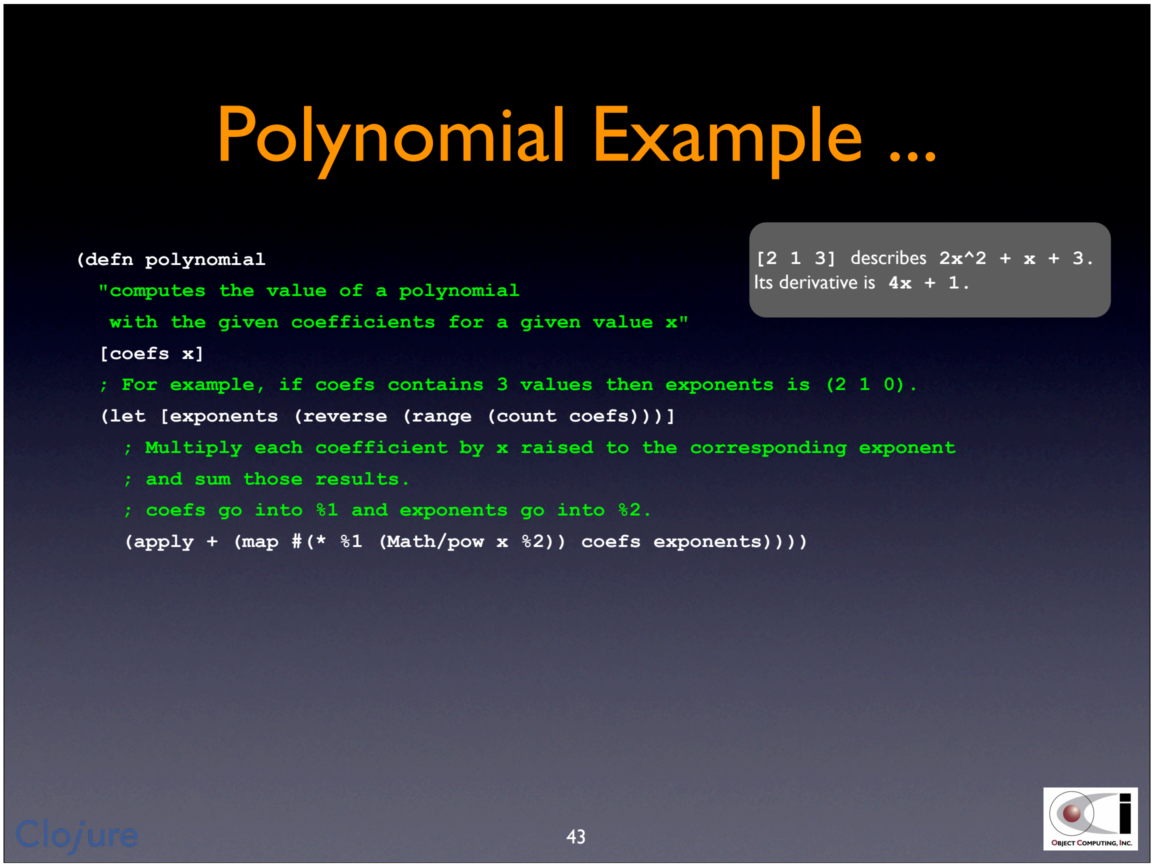
  polymorphism gone wild!

  - (defmulti ***name*** ***dispatch-function***)
  - (defmethod ***name*** ***dispatch-value*** [***parameters***] ***expressions***)

    names must match

  - parameters are passed to dispatch-function and result is compared to dispatch values
- Example

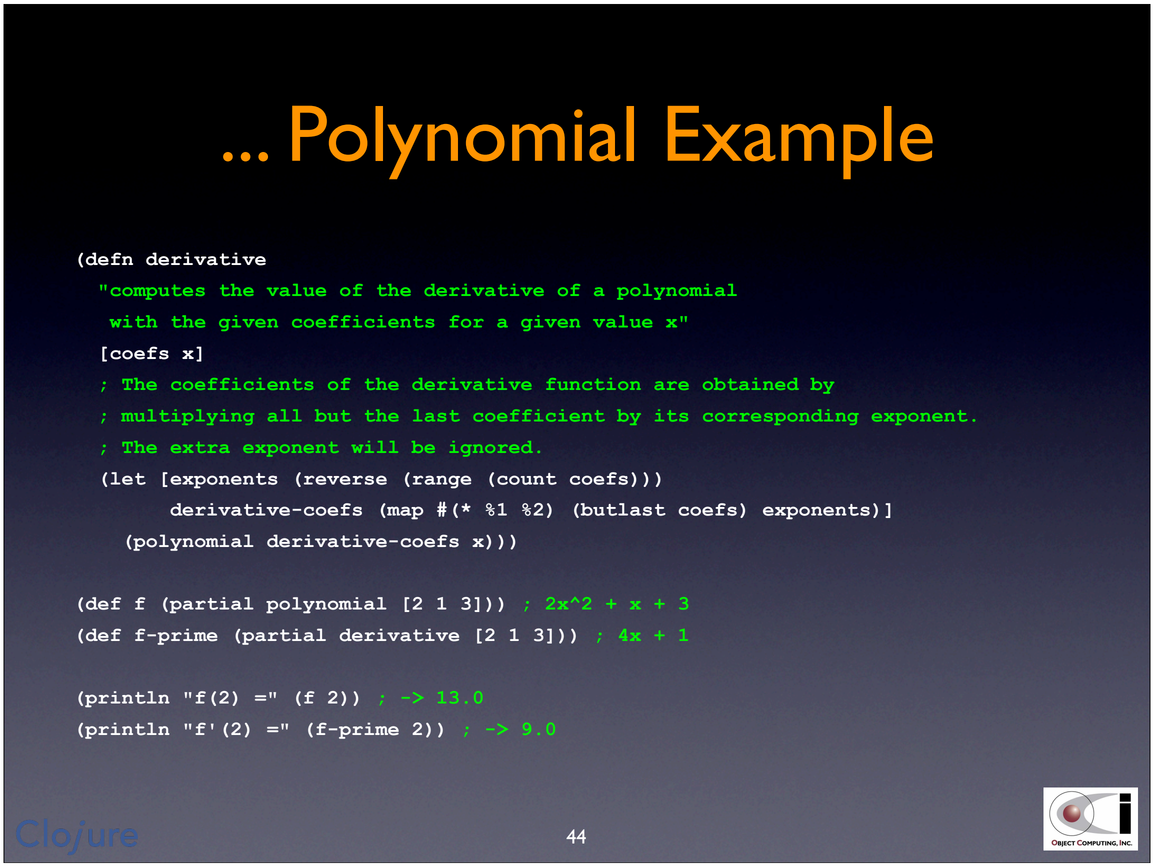
```
(defmulti what-am-i class) ; class is the dispatch function
(defmethod what-am-i Number [arg] (println arg "is a Number"))
(defmethod what-am-i String [arg] (println arg "is a String"))
(defmethod what-am-i :default [arg] (println arg "is something else"))
(what-am-i 19) ; -> 19 is a Number
(what-am-i "Hello") ; -> Hello is a String
(what-am-i true) ; -> true is something else
```



Polynomial Example ...

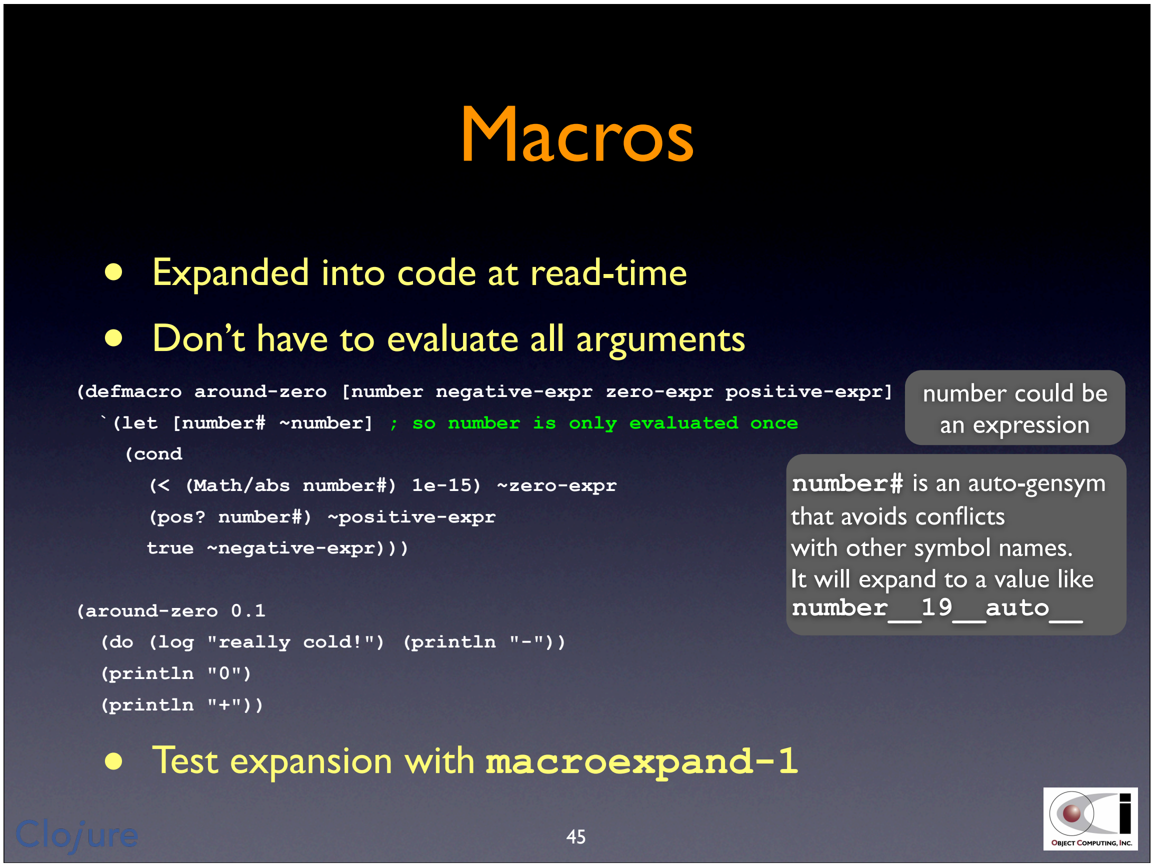
```
(defn polynomial
  "computes the value of a polynomial
   with the given coefficients for a given value x"
  [coefs x]
  ; For example, if coefs contains 3 values then exponents is (2 1 0).
  (let [exponents (reverse (range (count coefs)))]
    ; Multiply each coefficient by x raised to the corresponding exponent
    ; and sum those results.
    ; coefs go into %1 and exponents go into %2.
    (apply + (map #(* %1 (Math/pow x %2)) coefs exponents))))
```

`[2 1 3]` describes `2x^2 + x + 3`.
Its derivative is `4x + 1`.



... Polynomial Example

```
(defn derivative
  "computes the value of the derivative of a polynomial
   with the given coefficients for a given value x"
  [coefs x]
  ; The coefficients of the derivative function are obtained by
  ; multiplying all but the last coefficient by its corresponding exponent.
  ; The extra exponent will be ignored.
  (let [exponents (reverse (range (count coefs)))
        derivative-coefs (map #(* %1 %2) (butlast coefs) exponents)]
    (polynomial derivative-coefs x)))

(def f (partial polynomial [2 1 3])) ; 2x^2 + x + 3
(def f-prime (partial derivative [2 1 3])) ; 4x + 1

(println "f(2) =" (f 2)) ; -> 13.0
(println "f'(2) =" (f-prime 2)) ; -> 9.0
```



Macros

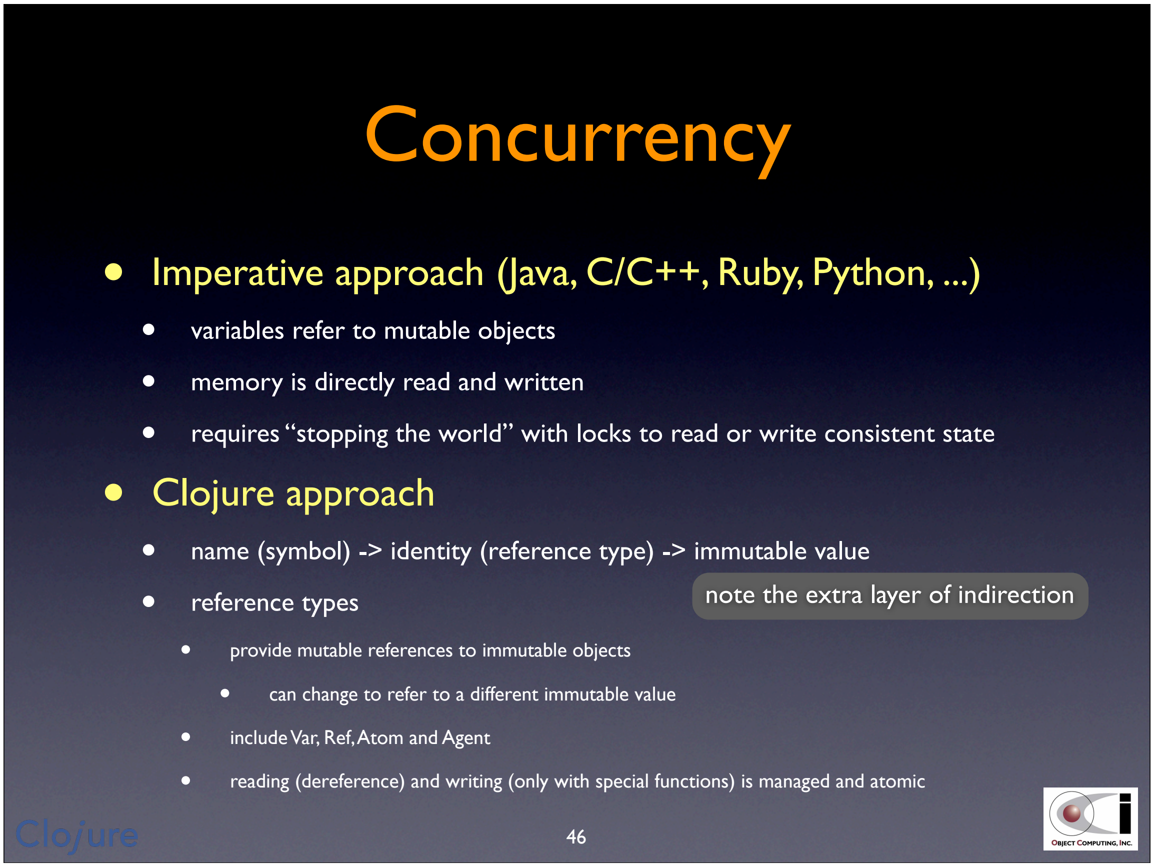
- Expanded into code at read-time
- Don't have to evaluate all arguments

```
(defmacro around-zero [number negative-expr zero-expr positive-expr]
  `(let [number# ~number] ; so number is only evaluated once
    (cond
      (< (Math/abs number#) 1e-15) ~zero-expr
      (pos? number#) ~positive-expr
      true ~negative-expr)))

(around-zero 0.1
  (do (log "really cold!") (println "-"))
  (println "0")
  (println "+"))
```

number could be an expression

`number#` is an auto-gensym that avoids conflicts with other symbol names. It will expand to a value like **`number__19__auto__`**

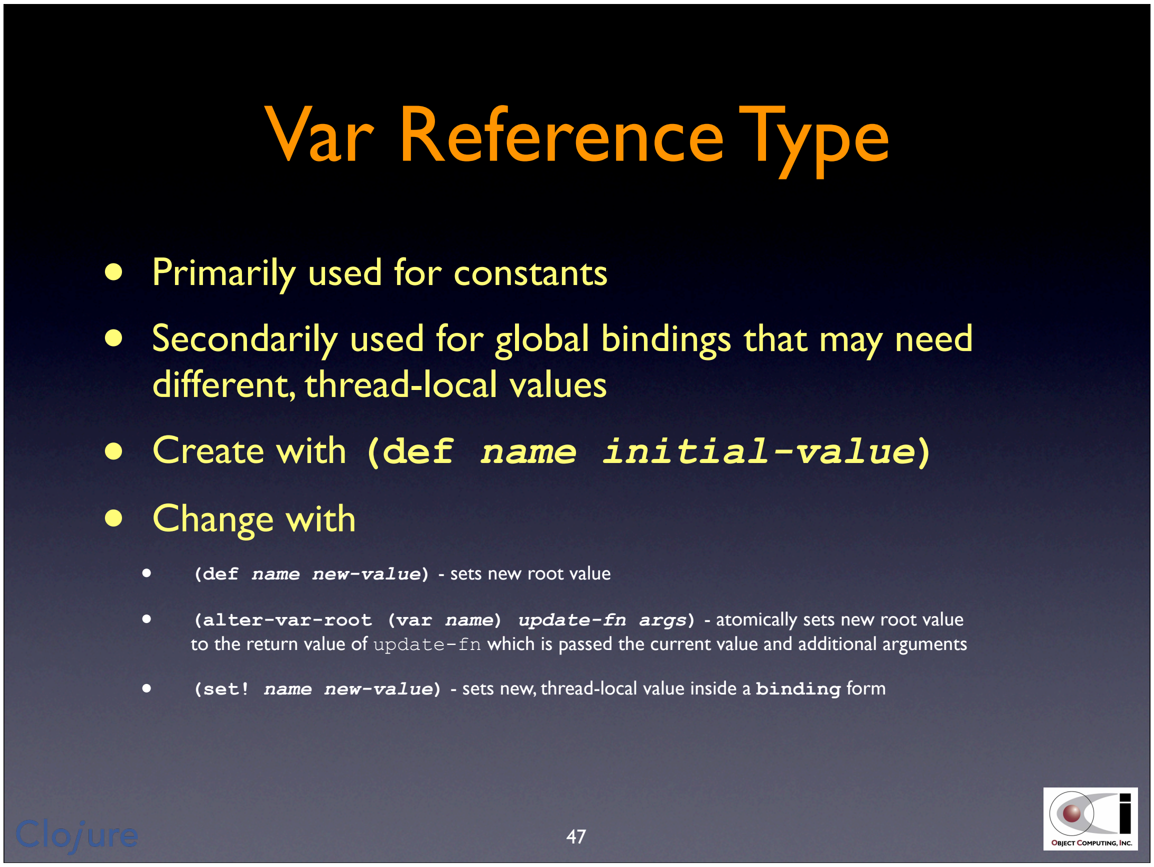
- Test expansion with **`macroexpand-1`**



Concurrency

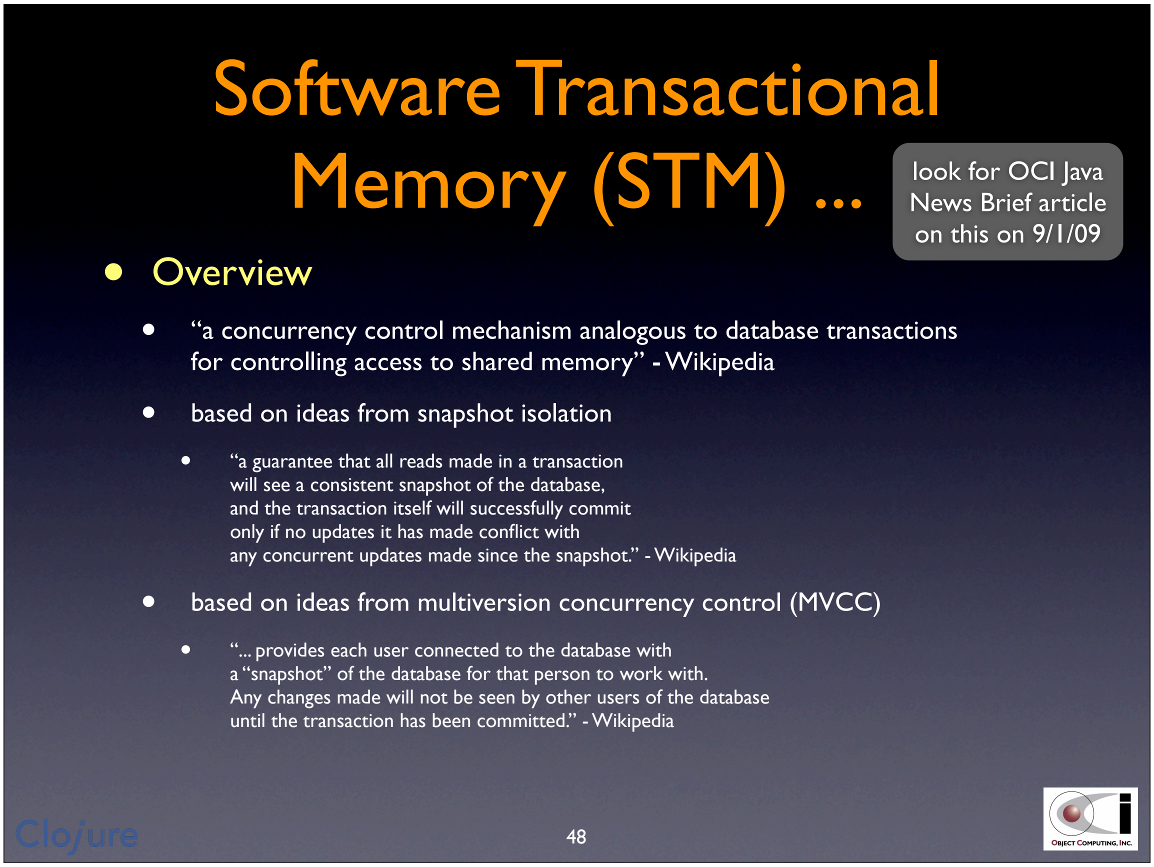
- Imperative approach (Java, C/C++, Ruby, Python, ...)
  - variables refer to mutable objects
  - memory is directly read and written
  - requires "stopping the world" with locks to read or write consistent state
- Clojure approach
  - name (symbol) -> identity (reference type) -> immutable value
  - reference types
    - provide mutable references to immutable objects
      - can change to refer to a different immutable value
    - include Var, Ref, Atom and Agent
    - reading (dereference) and writing (only with special functions) is managed and atomic

note the extra layer of indirection



Var Reference Type

- Primarily used for constants
- Secondarily used for global bindings that may need different, thread-local values
- Create with `(def name initial-value)`
- Change with
  - `(def name new-value)` - sets new root value
  - `(alter-var-root (var name) update-fn args)` - atomically sets new root value to the return value of `update-fn` which is passed the current value and additional arguments
  - `(set! name new-value)` - sets new, thread-local value inside a **`binding`** form



Software Transactional Memory (STM) ...

look for OCI Java News Brief article on this on 9/1/09

- Overview
  - “a concurrency control mechanism analogous to database transactions for controlling access to shared memory” - Wikipedia
  - based on ideas from snapshot isolation
    - “a guarantee that all reads made in a transaction will see a consistent snapshot of the database, and the transaction itself will successfully commit only if no updates it has made conflict with any concurrent updates made since the snapshot.” - Wikipedia
  - based on ideas from multiversion concurrency control (MVCC)
    - “... provides each user connected to the database with a “snapshot” of the database for that person to work with. Any changes made will not be seen by other users of the database until the transaction has been committed.” - Wikipedia

… STM …

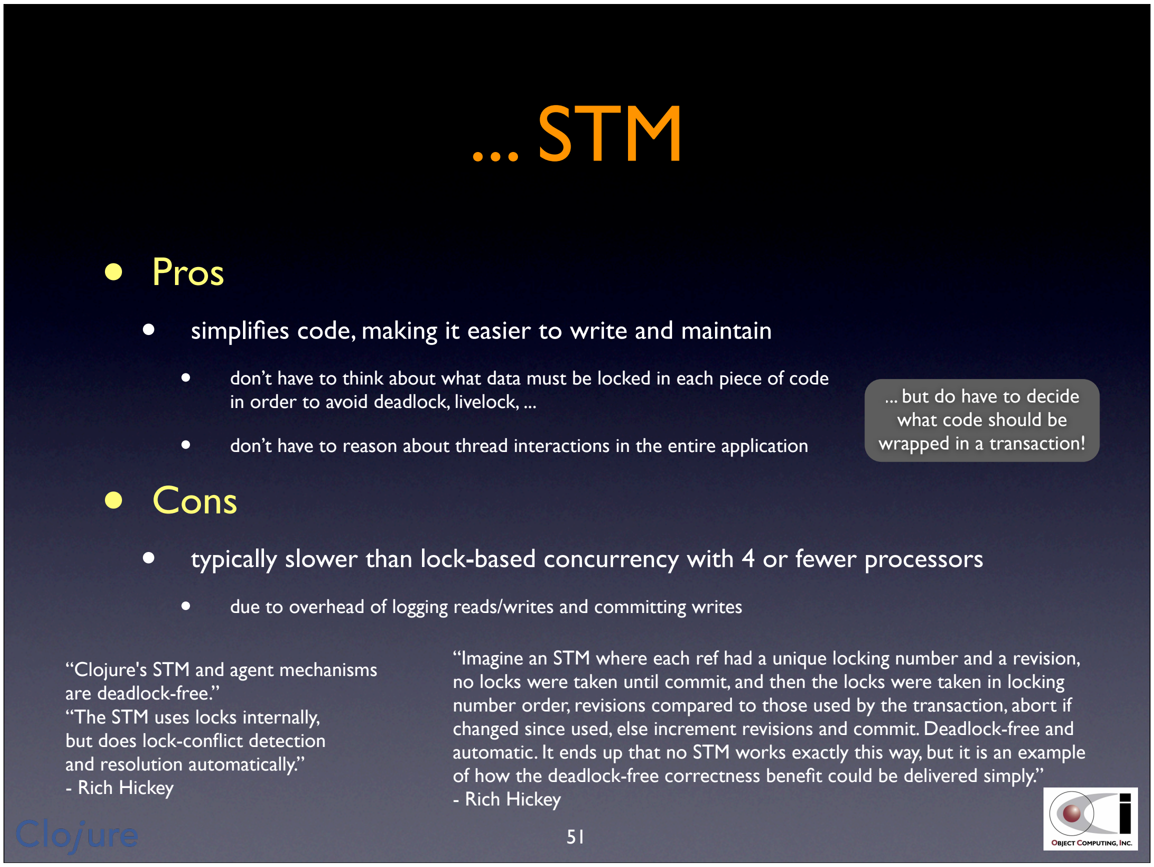
Database transactions have the ACID properties.
Atomic
Consistent
Isolated
Durable

- STM transactions
  - blocks of code that read and/or write shared memory (Refs in Clojure)
  - inside a transaction, Refs have a private, in-transaction value (makes them isolated)
    - intermediate states are not visible to other transactions
  - all changes to Refs made inside a transaction are either committed or rolled back so when the transaction ends, the Ref values are in a consistent state (makes them consistent)
  - all changes to Refs appear to occur at a single instant when the transaction commits (makes them atomic)
  - changes to Refs are lost if the application crashes (makes them not durable)

… STM …

- Optimistic
  - threads don't have to wait for access to shared resources
  - provides increased concurrency, especially for Ref reads
- Rollbacks
  - triggered by exceptions
  - triggered if another transaction commits changes to memory that was read or written in this transaction
  - all writes are discarded and the transaction is retried from the beginning until it succeeds
    - so shouldn't do anything in a transaction that can't be undone, such as I/O
    - one way to handle is to log the desired I/O and perform it once after the transaction completes

Another way to handle this is to perform the I/O in an "action" that is sent to an Agent inside the transaction. It will only be executed once during the commit.



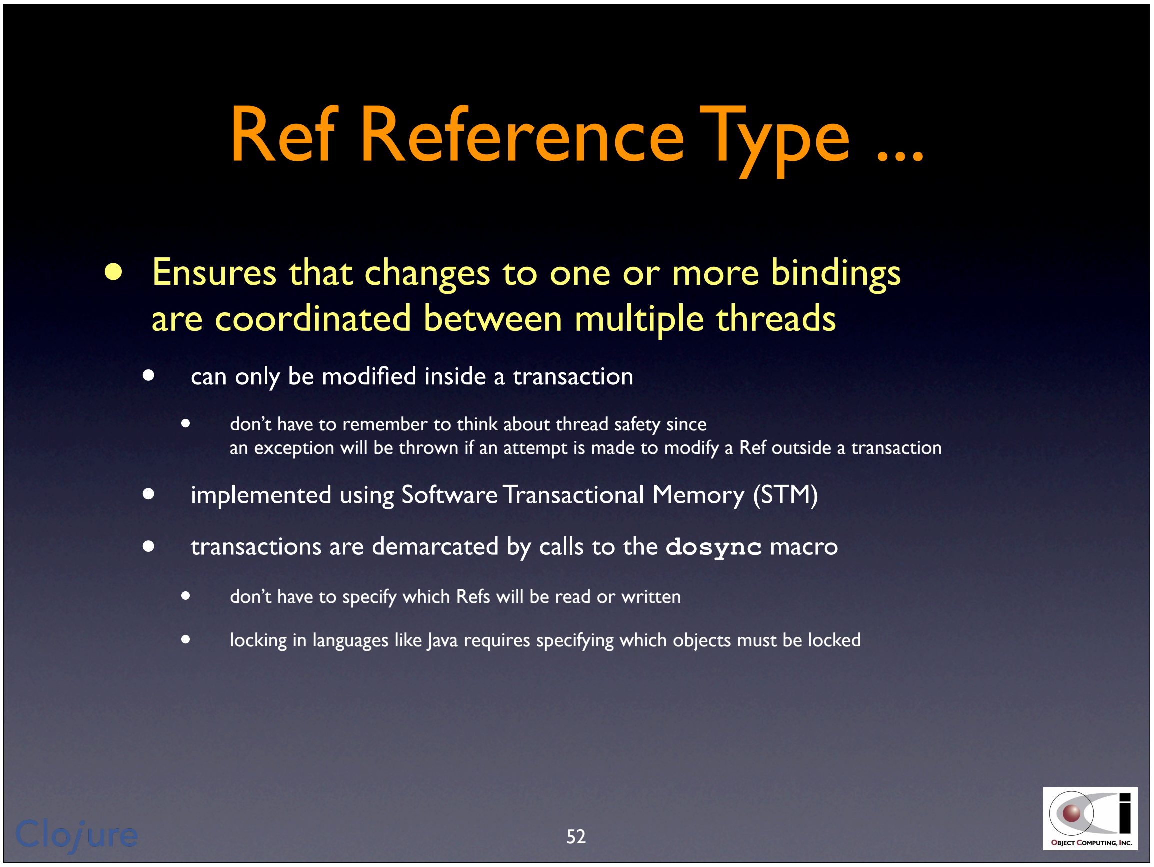
… STM

- Pros
  - simplifies code, making it easier to write and maintain
    - don’t have to think about what data must be locked in each piece of code in order to avoid deadlock, livelock, …
    - don’t have to reason about thread interactions in the entire application
- Cons
  - typically slower than lock-based concurrency with 4 or fewer processors
    - due to overhead of logging reads/writes and committing writes

… but do have to decide what code should be wrapped in a transaction!

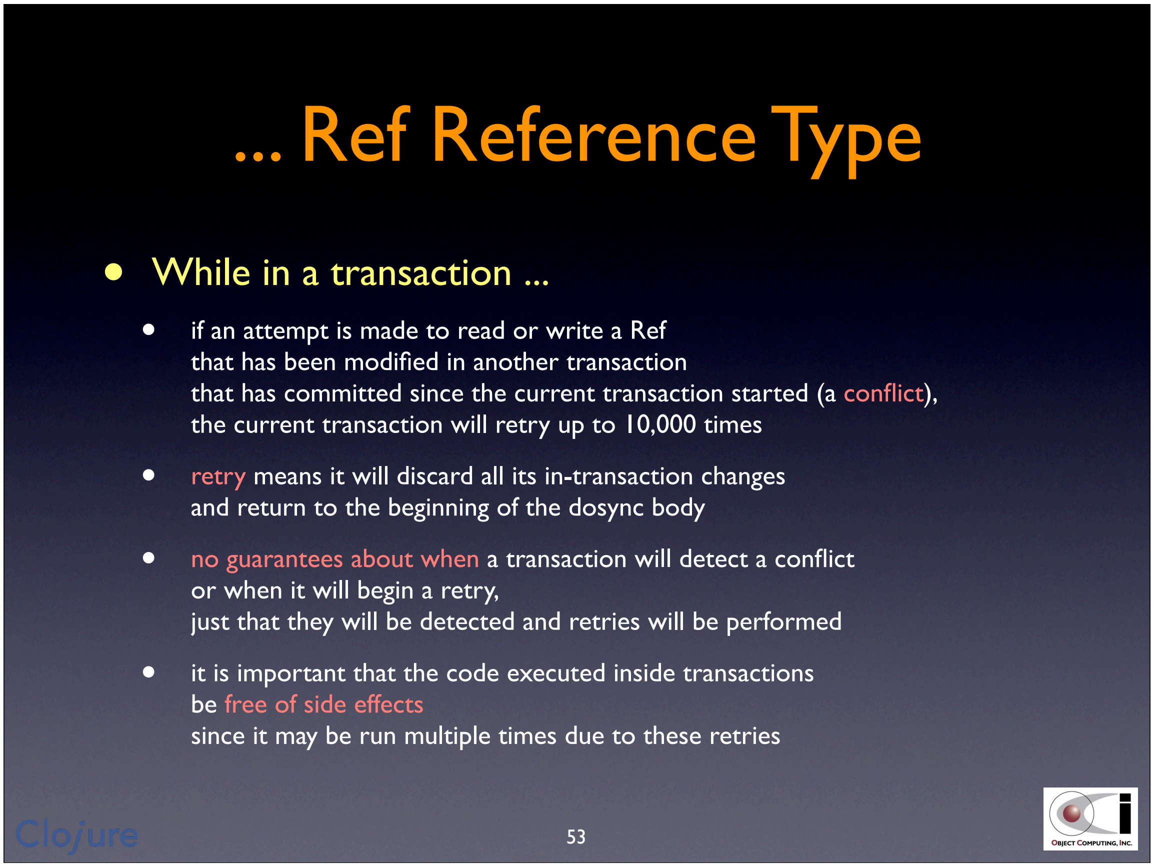
“Clojure's STM and agent mechanisms are deadlock-free.”
“The STM uses locks internally, but does lock-conflict detection and resolution automatically.”
- Rich Hickey

“Imagine an STM where each ref had a unique locking number and a revision, no locks were taken until commit, and then the locks were taken in locking number order, revisions compared to those used by the transaction, abort if changed since used, else increment revisions and commit. Deadlock-free and automatic. It ends up that no STM works exactly this way, but it is an example of how the deadlock-free correctness benefit could be delivered simply.”
- Rich Hickey



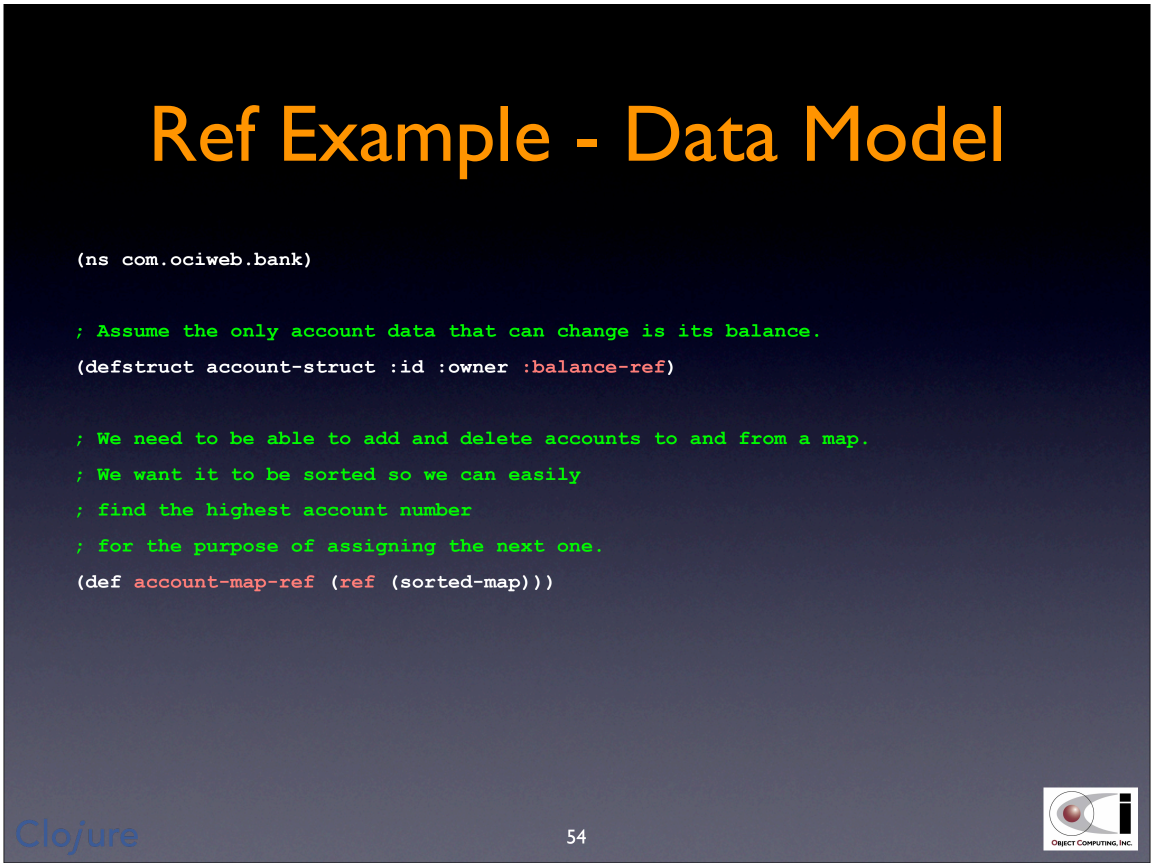
Ref Reference Type …

- Ensures that changes to one or more bindings are coordinated between multiple threads
  - can only be modified inside a transaction
    - don’t have to remember to think about thread safety since an exception will be thrown if an attempt is made to modify a Ref outside a transaction
  - implemented using Software Transactional Memory (STM)
  - transactions are demarcated by calls to the `dosync` macro
    - don’t have to specify which Refs will be read or written
    - locking in languages like Java requires specifying which objects must be locked



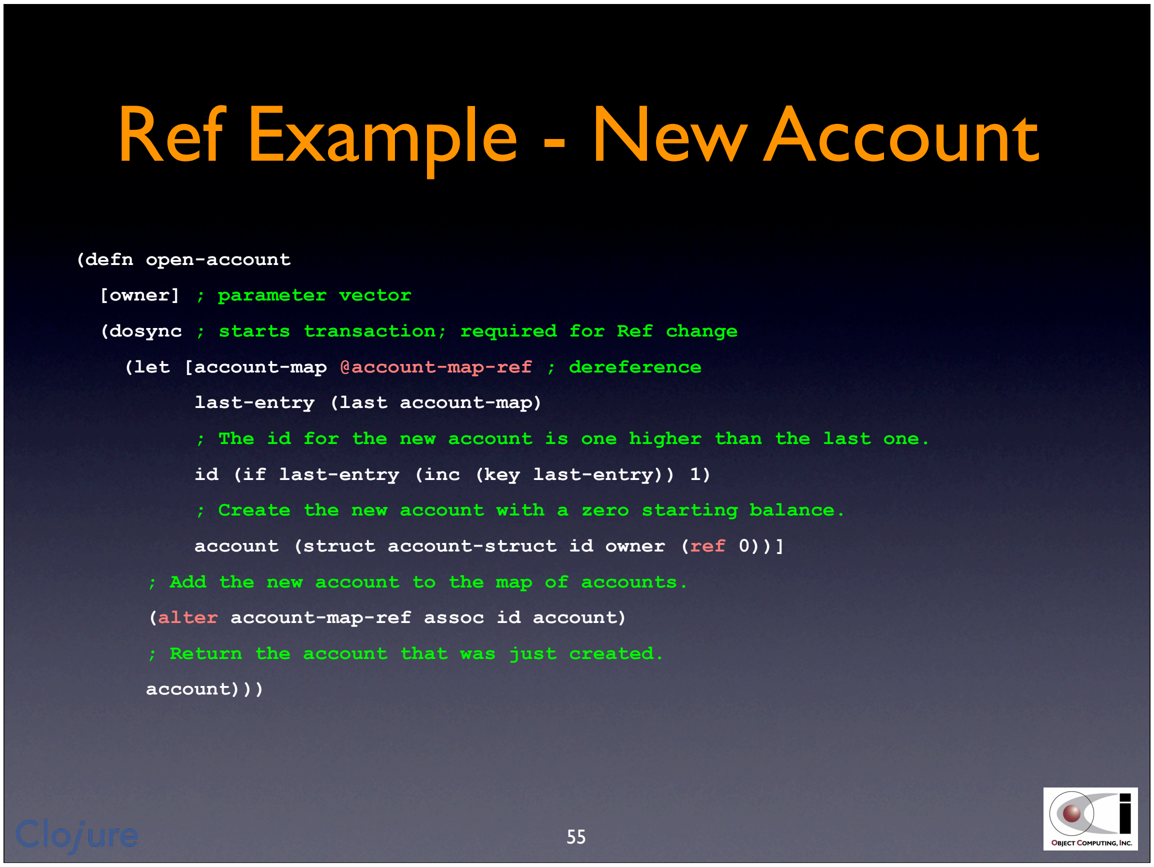
… Ref Reference Type

- While in a transaction …
  - if an attempt is made to read or write a Ref
    that has been modified in another transaction
    that has committed since the current transaction started (a conflict),
    the current transaction will retry up to 10,000 times
  - retry means it will discard all its in-transaction changes
    and return to the beginning of the dosync body
  - no guarantees about when a transaction will detect a conflict
    or when it will begin a retry,
    just that they will be detected and retries will be performed
  - it is important that the code executed inside transactions
    be free of side effects
    since it may be run multiple times due to these retries



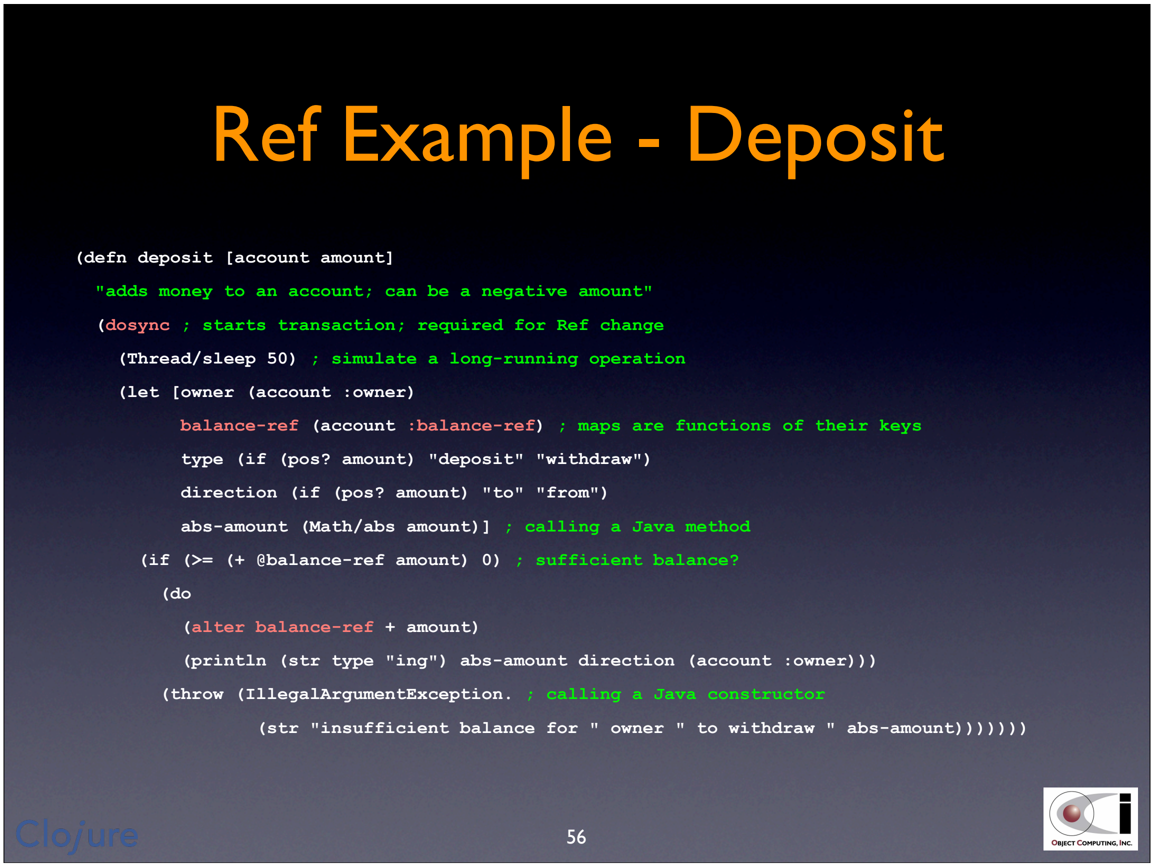
Ref Example - Data Model

```
(ns com.ociweb.bank)

; Assume the only account data that can change is its balance.
(defstruct account-struct :id :owner :balance-ref)

; We need to be able to add and delete accounts to and from a map.
; We want it to be sorted so we can easily
; find the highest account number
; for the purpose of assigning the next one.
(def account-map-ref (ref (sorted-map)))
```



Ref Example - New Account

```
(defn open-account
  [owner] ; parameter vector
  (dosync ; starts transaction; required for Ref change
    (let [account-map @account-map-ref ; dereference
          last-entry (last account-map)
          ; The id for the new account is one higher than the last one.
          id (if last-entry (inc (key last-entry)) 1)
          ; Create the new account with a zero starting balance.
          account (struct account-struct id owner (ref 0))]
      ; Add the new account to the map of accounts.
      (alter account-map-ref assoc id account)
      ; Return the account that was just created.
      account)))
```



Ref Example - Deposit

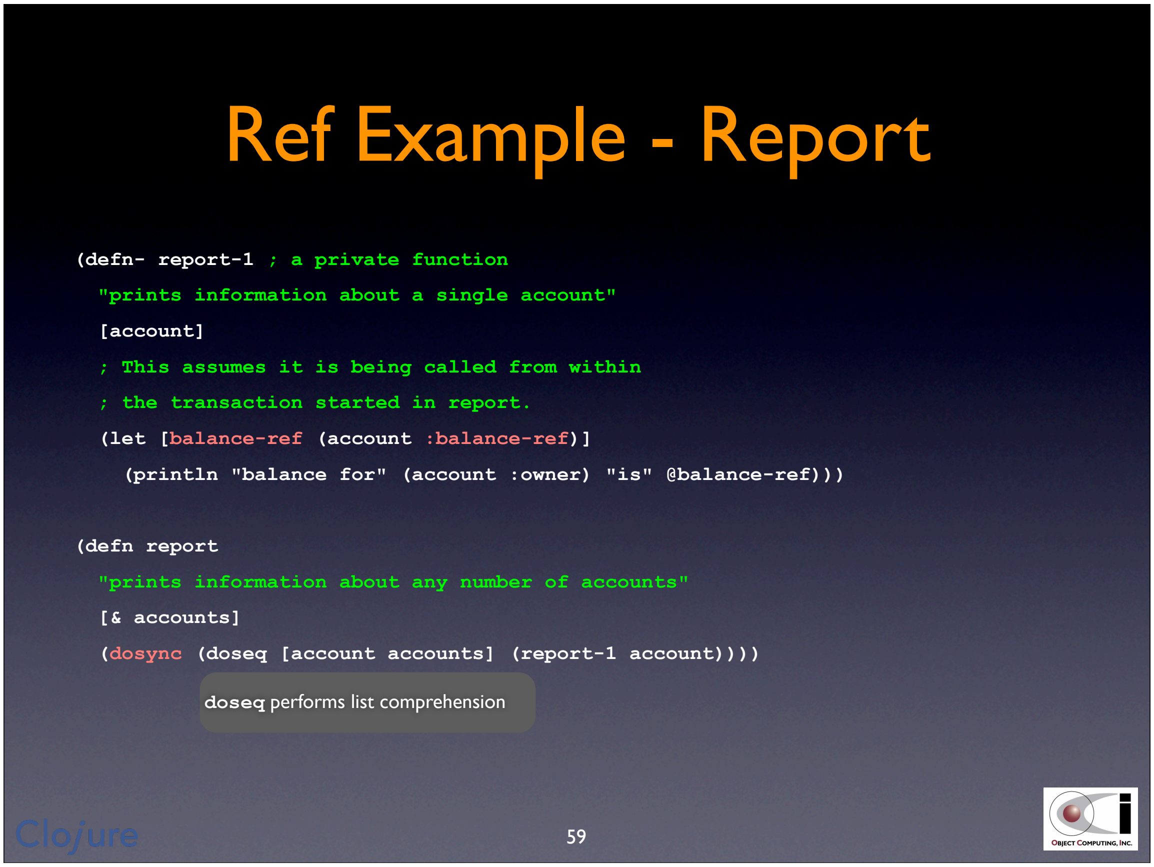
```
(defn deposit [account amount]
  "adds money to an account; can be a negative amount"
  (dosync ; starts transaction; required for Ref change
    (Thread/sleep 50) ; simulate a long-running operation
    (let [owner (account :owner)
          balance-ref (account :balance-ref) ; maps are functions of their keys
          type (if (pos? amount) "deposit" "withdraw")
          direction (if (pos? amount) "to" "from")
          abs-amount (Math/abs amount)] ; calling a Java method
      (if (>= (+ @balance-ref amount) 0) ; sufficient balance?
        (do
          (alter balance-ref + amount)
          (println (str type "ing") abs-amount direction (account :owner)))
        (throw (IllegalArgumentException. ; calling a Java constructor
                 (str "insufficient balance for " owner " to withdraw " abs-amount))))))
```

Ref Example - Withdrawal

```
(defn withdraw
  "removes money from an account"
  [account amount]
  ; A withdrawal is like a negative deposit.
  (deposit account (- amount)))
```

Ref Example - Transfer

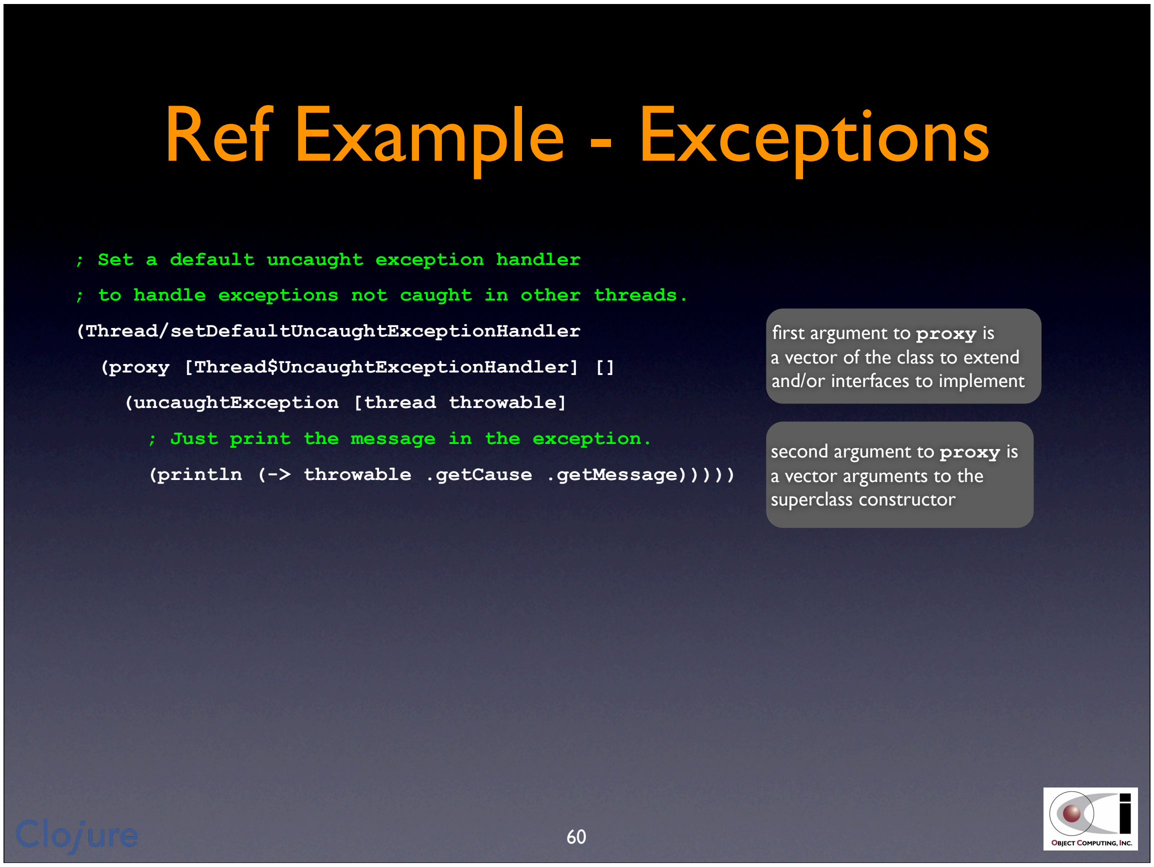
```
(defn transfer [from-account to-account amount]
  (dosync ; composing transactions from withdraw and deposit
    (println "transferring" amount
             "from" (from-account :owner)
             "to" (to-account :owner))
    (withdraw from-account amount)
    (deposit to-account amount)))
```



Ref Example - Report

```
(defn- report-1 ; a private function
  "prints information about a single account"
  [account]
  ; This assumes it is being called from within
  ; the transaction started in report.
  (let [balance-ref (account :balance-ref)]
    (println "balance for" (account :owner) "is" @balance-ref)))

(defn report
  "prints information about any number of accounts"
  [& accounts]
  (dosync (doseq [account accounts] (report-1 account))))
```

`doseq` performs list comprehension

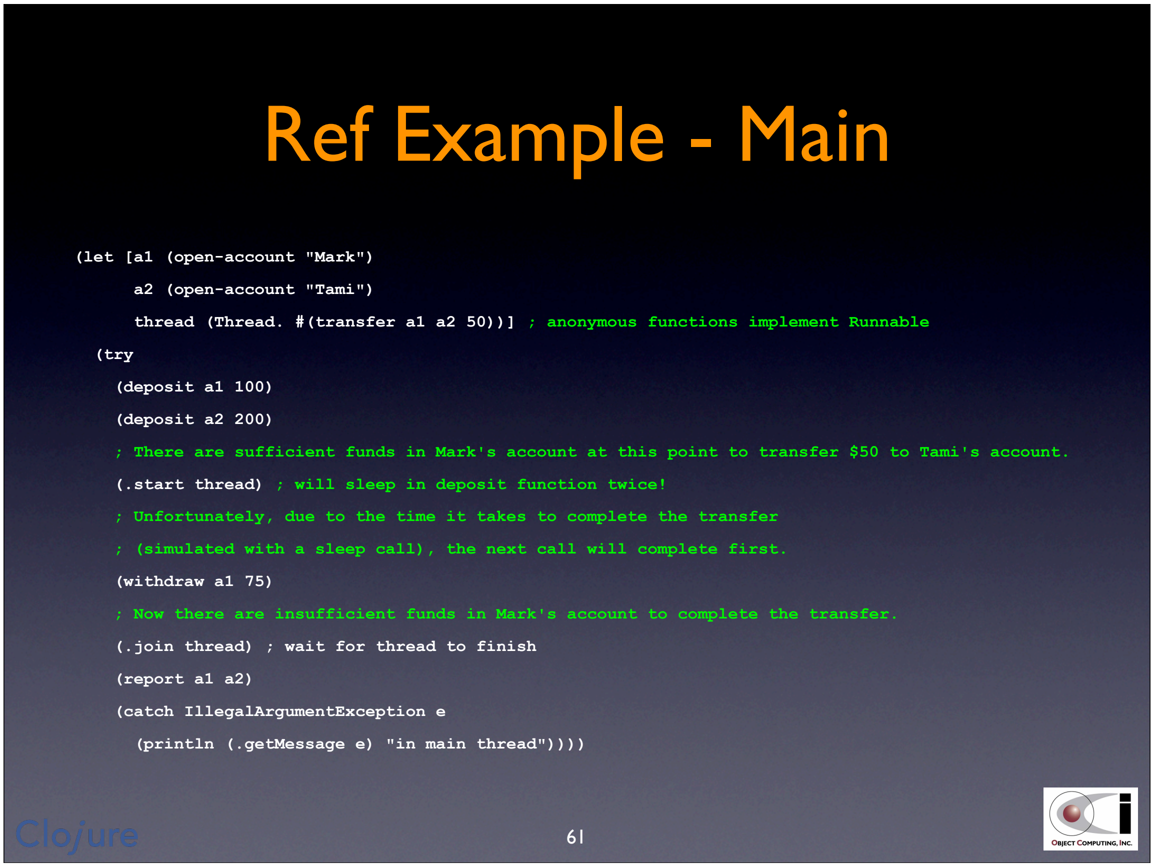


Ref Example - Exceptions

```
; Set a default uncaught exception handler
; to handle exceptions not caught in other threads.
(Thread/setDefaultUncaughtExceptionHandler
  (proxy [Thread$UncaughtExceptionHandler] []
    (uncaughtException [thread throwable]
      ; Just print the message in the exception.
      (println (-> throwable .getCause .getMessage)))))
```

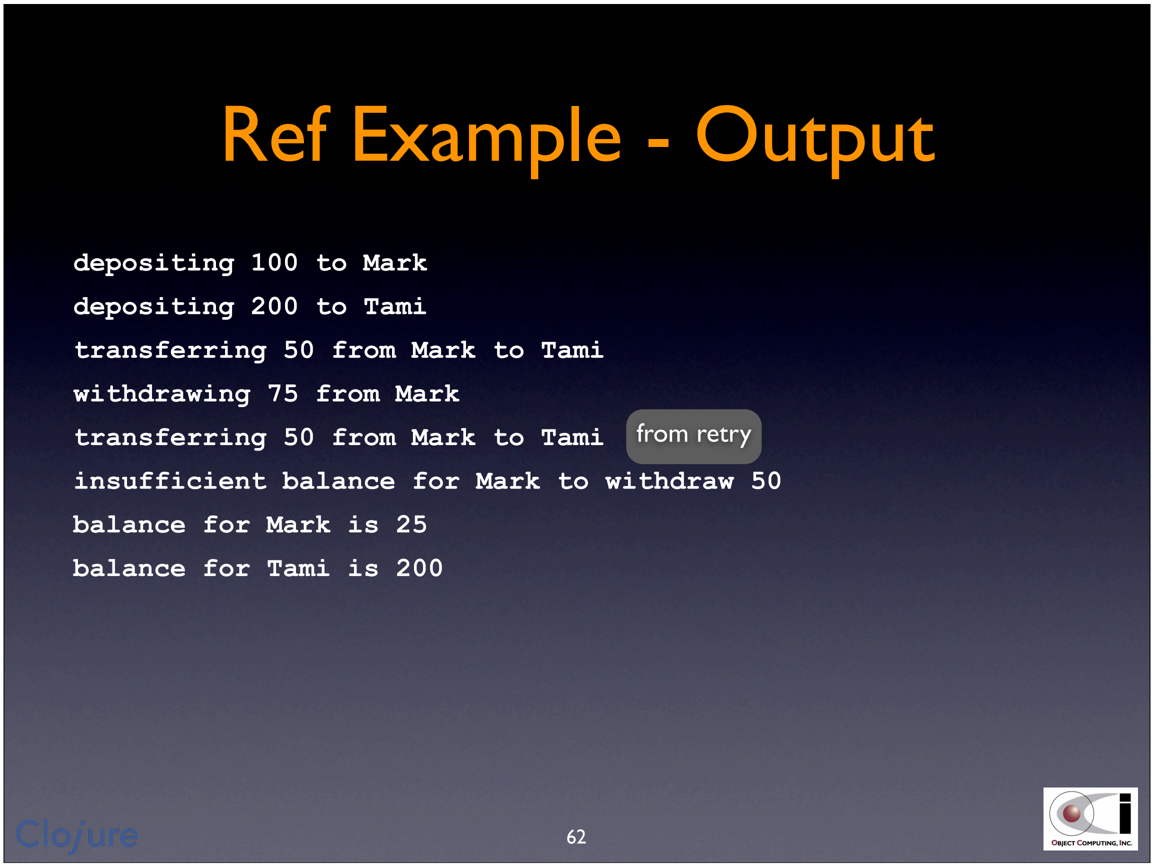
first argument to **proxy** is
a vector of the class to extend
and/or interfaces to implement

second argument to **proxy** is
a vector arguments to the
superclass constructor



Ref Example - Main

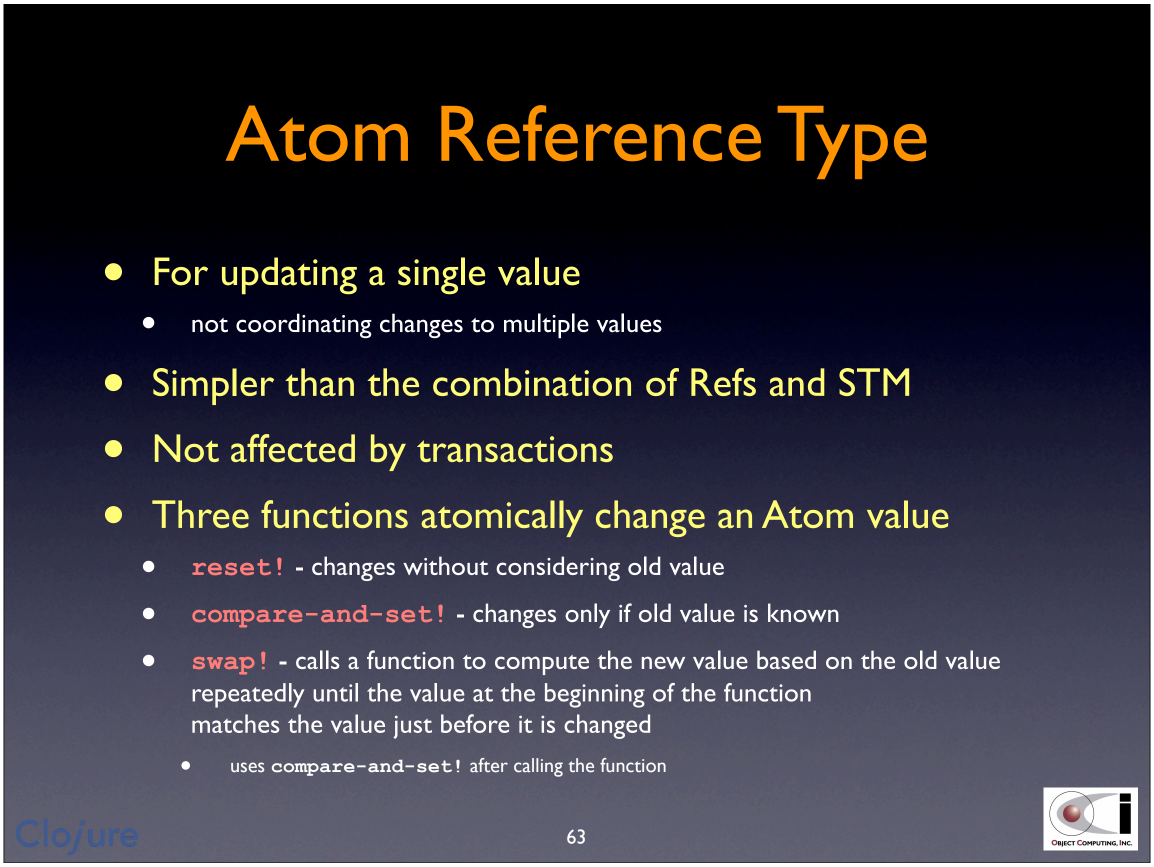
```
(let [a1 (open-account "Mark")
      a2 (open-account "Tami")
      thread (Thread. #(transfer a1 a2 50))] ; anonymous functions implement Runnable
  (try
    (deposit a1 100)
    (deposit a2 200)
    ; There are sufficient funds in Mark's account at this point to transfer $50 to Tami's account.
    (.start thread) ; will sleep in deposit function twice!
    ; Unfortunately, due to the time it takes to complete the transfer
    ; (simulated with a sleep call), the next call will complete first.
    (withdraw a1 75)
    ; Now there are insufficient funds in Mark's account to complete the transfer.
    (.join thread) ; wait for thread to finish
    (report a1 a2)
    (catch IllegalArgumentException e
      (println (.getMessage e) "in main thread"))))
```



Ref Example - Output

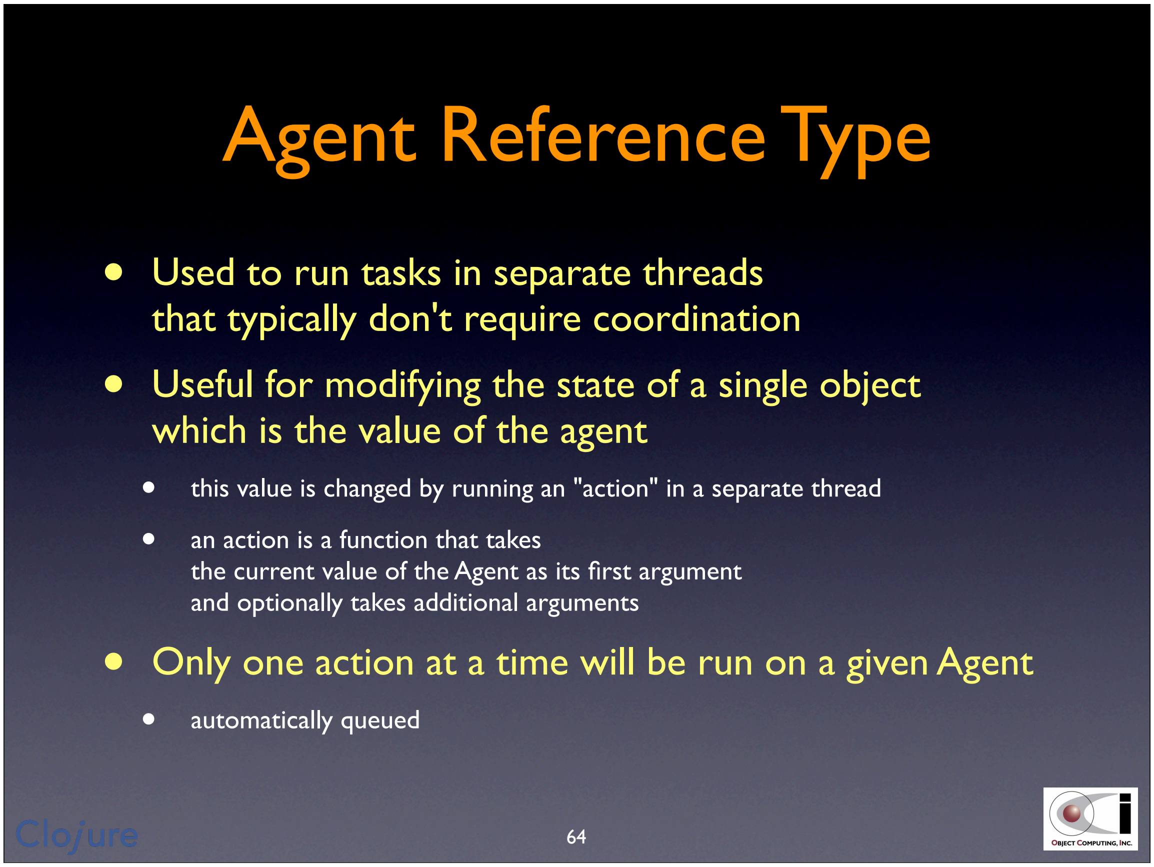
```
depositing 100 to Mark
depositing 200 to Tami
transferring 50 from Mark to Tami
withdrawing 75 from Mark
transferring 50 from Mark to Tami
insufficient balance for Mark to withdraw 50
balance for Mark is 25
balance for Tami is 200
```

from retry



Atom Reference Type

- For updating a single value
  - not coordinating changes to multiple values
- Simpler than the combination of Refs and STM
- Not affected by transactions
- Three functions atomically change an Atom value
  - `reset!` - changes without considering old value
  - `compare-and-set!` - changes only if old value is known
  - `swap!` - calls a function to compute the new value based on the old value
    repeatedly until the value at the beginning of the function
    matches the value just before it is changed
    - uses `compare-and-set!` after calling the function



Agent Reference Type

- Used to run tasks in separate threads
  that typically don't require coordination
- Useful for modifying the state of a single object
  which is the value of the agent
  - this value is changed by running an "action" in a separate thread
  - an action is a function that takes
    the current value of the Agent as its first argument
    and optionally takes additional arguments
- Only one action at a time will be run on a given Agent
  - automatically queued

Editors and IDEs

- Emacs
  - clojure-mode and swank-clojure, both at http://github.com/jochu. swank-clojure
  - uses the Superior Lisp Interaction Mode for Emacs (Slime)
    described at http://common-lisp.net/project/slime/
- Vim
  - VimClojure http://kotka.de/projects/clojure/vimclojure.html and
    Gorilla at http://kotka.de/projects/clojure/gorilla.html
- NetBeans - enclojure at http://enclojure.org/
- IDEA - "La Clojure" at http://plugins.intellij.net/plugin/?id=4050
- Eclipse - clojure-dev at http://code.google.com/p/clojure-dev/

Lambda Lounge

- A St. Louis user group that focuses on
  functional and dynamic programming languages
- http://lambdalounge.org/
- Meets in Appistry office
  - near intersection of Olive and Lindbergh
- First Thursday of each month
  - 6 p.m. to around 8 p.m.
- Google Group - lambda-lounge

Resources

- http://ociweb.com/mark/clojure/ contains
  - a link to a long Clojure article I wrote
  - a link to page that categorizes all built-in Clojure special forms, functions and macros
  - many other Clojure-related links