Clojure

we have a
lot to do...

day 1

topic	format
quick start	talk
labrepl	lab
clojure introduction	talk
names and places	lab
it's all data	lab
functions, values, and abstractions	talk
project euler	lab

day 2

topic	format
compojure	talk
mini-browser	lab
modeling state and time 1	talk
unified update model	lab
modeling state and time 2	talk
zero sum	lab
java interop	talk

day 3

topic	**format**
cellular automata	lab
macros and evaluation	talk
defstrict	lab
oo: records, types, protocols, & multimethods	talk
rock, paper, scissors	lab
the clojure ecosystem	talk

quick start

where are you coming from?

lisp?

java / c# / scala?

ml / haskell?

python / ruby / groovy?

clojure?

multithreaded programming?

data and code

data literals

type	properties	example
list	singly-linked, insert at front	`(1 2 3)`
vector	indexed, insert at rear	`[1 2 3]`
map	key/value	`{:a 100` ` :b 90}`
set	key	`#{:a :b}`

reading code

semantics: fn call arg

```
(println "Hello World")
```

structure: list symbol string

defn semantics

define a fn

fn name

docstring

```clojure
(defn greet
  "Returns a friendly greeting"
  [your-name]
  (str "Hello, " your-name))
```

arguments

fn body

defn structure

symbol symbol

string

```
(defn greet
  "Returns a friendly greeting"
  [your-name]
  (str "Hello, " your-name))
```

vector

list

all forms created equal

form	syntax	example
function	list	`(println "hello")`
operator	list	`(+ 1 2)`
method call	list	`(.trim " hello ")`
import	list	`(require 'mylib)`
metadata	list	`(with-meta obj m)`
control flow	list	`(when valid? (proceed))`
scope	list	`(dosync (alter ...))`

platform interop

java new

java	`new Widget("foo")`
clojure sugar	`(Widget. "red")`

access static members

java	`Math.PI`
clojure sugar	`Math/PI`

access instance members

java	`rnd.nextInt()`
clojure sugar	`(.nextInt rnd)`

chaining access

java	`person.getAddress().getZipCode()`
clojure sugar	`(.. person getAddress getZipCode)`

parenthesis count

java	( ) ( ) ( ) ( )
clojure	( ) ( ) ( )

atomic data types

type	example	java equivalent
string	`"foo"`	String
character	`\f`	Character
regex	`#"fo*"`	Pattern
integer	`42`	long
a.p. integer	42N	BigInteger
double	`3.14159`	double
a.p. double	`3.14159M`	BigDecimal
boolean	`true`	Boolean
nil	`nil`	`null`
symbol	`foo, +`	N/A
keyword	`:foo, ::foo`	N/A

simplicity

simplicity is *absence of complexity*

simplicity is **not**:

simplicity is **not**:

familiarity

simplicity is **not**:

familiarity

a superficial property

simplicity is **not**:

familiarity

a superficial property

easy to do

simplicity is **not**:

familiarity

a superficial property

easy to do

you want simple
tools to tackle
complex problems

example: refactor apache commons isBlank

initial implementation

```java
public class StringUtils {
  public static boolean isBlank(String str) {
    int strLen;
    if (str == null || (strLen = str.length()) == 0) {
      return true;
    }
    for (int i = 0; i < strLen; i++) {
      if ((Character.isWhitespace(str.charAt(i)) == false)) {
        return false;
      }
    }
    return true;
  }
}
```

- type decls

```
public class StringUtils {
  public isBlank(str) {
    if (str == null || (strLen = str.length()) == 0) {
      return true;
    }
    for (i = 0; i < strLen; i++) {
      if ((Character.isWhitespace(str.charAt(i)) == false)) {
        return false;
      }
    }
    return true;
  }
}
```

- class

```
public isBlank(str) {
  if (str == null || (strLen = str.length()) == 0) {
    return true;
  }
  for (i = 0; i < strLen; i++) {
    if ((Character.isWhitespace(str.charAt(i)) == false)) {
      return false;
    }
  }
  return true;
}
```

+ higher-order function

```
public isBlank(str) {
  if (str == null || (strLen = str.length()) == 0) {
    return true;
  }
  every (ch in str) {
    Character.isWhitespace(ch);
  }
  return true;
}
```

- corner cases

```
public isBlank(str) {
  every (ch in str) {
    Character.isWhitespace(ch);
  }
}
```

lispify

```
(defn blank? [s]
  (every? #(Character/isWhitespace %) s))
```

repl
exploration

doc

```
(doc name)
-------------------------
clojure.core/name
([x])
  Returns the name String of a symbol
or keyword.
```

find-doc

```
(find-doc "pmap")
-------------------------
clojure.core/pmap
([f coll] [f coll & colls])
  Like map, except f is applied in parallel. Semi-lazy in that the
  parallel computation stays ahead of the consumption, but doesn't
  realize the entire result unless required. Only useful for
  computationally intensive functions where the time of f dominates
  the coordination overhead.
-------------------------
clojure.core/zipmap
([keys vals])
  Returns a map with the keys mapped to the corresponding vals.
```

source

```
(source odd?)

(defn odd?
  "Returns true if n is odd, throws an exception if
   n is not an integer"
  [n] (not (even? n)))
```

javadoc

```
(javadoc "foo")
```

Overview Package **Class** Use Tree Deprecated Index Help

Java™ Platform Standard Ed. 6

PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

java.lang

Class String

```
java.lang.Object
  └ java.lang.String
```

All Implemented Interfaces:

Serializable, CharSequence, Comparable<String>

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

The `String` class represents character strings. All string literals in Java programs, such as `"abc"`, are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
String str = "abc";
```

dir

```
(dir clojure.contrib.java-utils)

as-file
as-properties
as-str
as-url
file
get-system-property
read-properties
relative-path-string
set-system-properties
with-system-properties
write-properties
```

platform reflection

```
(use 'clojure.reflect)

(->> (reflect "a string")
     :members
     (filter #(= 'int (:return-type %)))
     (map :name))

=> (indexOf lastIndexOf indexOf indexOf
codePointBefore indexOf offsetByCodePoints hashCode
compareTo compareTo lastIndexOf lastIndexOf
codePointAt lastIndexOf lastIndexOf indexOf length
compareToIgnoreCase codePointCount)
```

pprint

```
(use 'clojure.pprint)

(pprint
 (for [rank (range 8 0 -1)]
   (for [file "abcdefgh"]
     (str file rank))))
(("a8" "b8" "c8" "d8" "e8" "f8" "g8" "h8")
 ("a7" "b7" "c7" "d7" "e7" "f7" "g7" "h7")
 ("a6" "b6" "c6" "d6" "e6" "f6" "g6" "h6")
 ("a5" "b5" "c5" "d5" "e5" "f5" "g5" "h5")
 ("a4" "b4" "c4" "d4" "e4" "f4" "g4" "h4")
 ("a3" "b3" "c3" "d3" "e3" "f3" "g3" "h3")
 ("a2" "b2" "c2" "d2" "e2" "f2" "g2" "h2")
 ("a1" "b1" "c1" "d1" "e1" "f1" "g1" "h1"))
```

labrepl organization

directory structure

dir	usage
autodoc	generated docs
classes	compiled classes
config	per-environment config
data	program data
lib	**classpath jars**
log	log files
project.clj	**leiningen config**
public	static resources
script	shell scripts
src	**project source (clj)**
test	**automated tests (clj)**

leiningen: the goodness* of maven wrapped in clojure

*ymmv

labrepl project.clj (elided)

snapshots
bad

```
(defproject labrepl "0.0.2-SNAPSHOT"
  :description "Clojure Labs"
  :dependencies [[org.clojure/clojure
                   "1.3.0-master-SNAPSHOT"]
                  [ring/ring-jetty-adapter
                   "0.3.7"
                   :exclusions
                   [org.clojure/clojure]]]
  :dev-dependencies [[autodoc "0.7.0"]
                     [swank-clojure "1.1.0"]]
  :repositories {"incanter"
                  "http://repo.incanter.org"})
```

exclusions
good

lein tasks

```
>lein
Leiningen is a build tool for Clojure.

Several tasks are available:
  pom
  help
  install
  jar
  test
  deps
  uberjar
  clean
  compile
  swank
  new

Run lein help $TASK for details.
See http://github.com/technomancy/leiningen as well.
```

labrepl script/repl

```
#!/bin/sh
CLASSPATH=src:test:resources:data

for f in lib/*.jar; do
    CLASSPATH=$CLASSPATH:$f
done

java -Xmx1G -cp $CLASSPATH jline.ConsoleRunner
clojure.main -i script/run.clj -r
```

introducing
the labrepl (lab)

Clojure

Introduction

Clojure Objectives

- A Lisp
- Functional
  - emphasis on immutability
- Supporting Concurrency
  - language-level coordination of state
- Designed to be hosted
  - exposes and embraces platform (JVM)
- Adoption (open source, community)

Why pursue Clojure?

- Flexibility
- Interactivity
- Concision
- Exploration
- Power
- Simplicity
- <u>Focus</u> on your problem

Why Lisp?

- Dynamic
- Small core
  - Clojure ~~is~~ was a solo effort
- Elegant syntax
- Core advantage still code-as-data and syntactic abstraction
- Can reduce parens-overload

What about Common Lisp and Scheme?

- Why yet another Lisp?
- Limits to change post standardization
- Core data structures mutable, not extensible
- No concurrency in specs
- Good implementations already exist for JVM (ABCL, Kawa, SISC et al)
- Standard Lisps are their own platforms

Why the JVM?

- VMs, not OSes, are the target platforms of future languages, providing:
  - Type system
    - *Dynamic* enforcement and safety
  - Libraries
    - Huge set of facilities
  - Memory and other resource management
    - GC is platform, not language, facility
  - Bytecode + JIT compilation

Language as platform vs. Language + platform

- Old way - each language defines its own runtime
  - GC, bytecode, type system, libraries etc
- New way (JVM, .Net)
  - Common runtime independent of language
- Platforms are dictated by clients
  - Huge investments in performance, scalability, security, libraries etc.

Java/JVM *is* language + platform

- Not the original story, but other languages for JVM always existed, now embraced by Sun
- JVM has established track record and trust level
  - Now open source
- Interop with other code always required
  - C linkage insufficient these days
  - Ability to call/consume Java is critical
- Clojure is the language, JVM the platform

Clojure is a Lisp

- Dynamic
- Code as data
- Reader
- Small core
- REPL
- Sequences
- Syntactic abstraction (macros)

Atomic Data Types

- Longs - `1234` , BigDecimals `1234567891234567891 23N`
- Doubles `1.234` , BigDecimals `1.234M`
- Ratios - `22/7`
- Strings - `“fred”` , Characters - `\a \b \c`
- Symbols - `fred ethel` , Keywords - `:fred :ethel`
- Booleans - `true false` , Null - `nil`
  - true/false/nil are not symbols
- Regex patterns `#”a*b”`

Symbols

- Simply names, no storage cells
  - have optional prefix using / separator
    - `foo/bar`
- Prefix is called 'namespace' part, but need not designate a namespace
- String components interned for fast equality
  - but `(identical? 'foo 'foo) -> false`
- use of / and . in names subject to special resolution

Keywords

- Simply names, no storage cells, have optional prefix using / separator
  - `:foo/bar`
- Prefix is called 'namespace' part, but need not designate a namespace
- Keywords *are* interned
  - `(identical? :foo :foo) -> true`
- Leading :: causes keyword to be qualified in current namespace, e.g. in user
  - `::foo -> :user/foo`

Data Structures

- Lists - singly linked, grow at front
  - `(1 2 3 4 5), (fred ethel lucy), (list 1 2 3)`
- Vectors - indexed access, grow at end
  - `[1 2 3 4 5], [fred ethel lucy]`
- Maps - key/value associations
  - `{:a 1, :b 2, :c 3}, {1 “ethel” 2 “fred”}`
- Sets `#{fred ethel lucy}`
- Everything Nests

Sequences

```
(drop 2 [1 2 3 4 5]) -> (3 4 5)

(take 9 (cycle [1 2 3 4]))
-> (1 2 3 4 1 2 3 4 1)

(interleave [:a :b :c :d :e] [1 2 3 4 5])
-> (:a 1 :b 2 :c 3 :d 4 :e 5)

(partition 3 [1 2 3 4 5 6 7 8 9])
-> ((1 2 3) (4 5 6) (7 8 9))

(map vector [:a :b :c :d :e] [1 2 3 4 5])
-> ([:a 1] [:b 2] [:c 3] [:d 4] [:e 5])

(apply str (interpose \, "asdf"))
-> "a,s,d,f"

(reduce + (range 100)) -> 4950
```

Maps

```
(def m {:a 1 :b 2 :c 3})

(m :b) -> 2 ;also (:b m)

(keys m) -> (:a :b :c)

(assoc m :d 4 :c 42) -> {:d 4, :a 1, :b 2, :c 42}

(dissoc m :d) -> {:a 1, :b 2, :c 3}

(merge-with + m {:a 2 :b 3}) -> {:a 3, :b 5, :c 3}
```

Sets

```clojure
(use clojure.set)
(def colors #{"red" "green" "blue"})
(def moods #{"happy" "blue"})

(disj colors "red")
-> #{"green" "blue"}

(difference colors moods)
-> #{"green" "red"}

(intersection colors moods)
-> #{"blue"}

(union colors moods)
-> #{"happy" "green" "red" "blue"}
```

bonus: all relational algebra primitives supported for sets-of-maps

Nested Structures

```
(def jdoe {:name "John Doe",
           :address {:zip 27705, ...}})

(get-in jdoe [:address :zip])
-> 27705

(assoc-in jdoe [:address :zip] 27514)
-> {:name "John Doe", :address {:zip 27514}}

(update-in jdoe [:address :zip] inc)
-> {:name "John Doe", :address {:zip 27706}}
```

Clojure is (Primarily) Functional

- Core data structures immutable
- Core library functions have no side effects
- let-bound locals are immutable
- loop/recur functional looping construct

Persistent Data Structures

- Immutable, + old version of the collection is still available after 'changes'
- Collection maintains its performance guarantees for most operations
  - Therefore new versions are not full copies
- All Clojure data structures persistent
  - Hash map and vector both based upon array mapped hash tries (Bagwell)
  - Sorted map is red-black tree

Metadata

- Orthogonal to the logical value of the data
- Symbols and collections support a metadata map
- Does not impact equality semantics, nor seen in operations on the value
- Support for literal metadata in reader

```
(def v [1 2 3])
(def trusted-v (with-meta v {:source :trusted}))

(:source (meta trusted-v)) -> :trusted
(:source (meta v)) -> nil

(= v trusted-v) -> true
```

Syntax

- You’ve just seen it
- Data structures *are* the code
  - Homoiconicity
- No more text-based syntax
- Actually, syntax is in the interpretation of data structures

Expressions

- Everything is an expression
- All data literals represent themselves
  - *Except:*
    - Symbols
      - looks for binding to value, locally, then globally
    - Lists
      - An operation form

Operation forms

- `(op ...)`
- op can be either:
  - one of very few special ops
  - macro
  - expression which yields a function (more generally, something invocable)

Special ops

- Can have non-normal evaluation of arguments
  - `(def name value-expr)`
    - establishes a global variable
  - `(if test-expr then-expr else-expr)`
    - conditional, evaluates only one of then/else
- `fn let loop recur do new . throw try set! quote var`

Special forms

- `(def symbol init?)`
- `(if test then else?)`
- `(do exprs*)`
- `(quote form)`
- `(fn name? [params*] exprs*)`

  `(fn name? ([params*] exprs*)+)`
- `(let [bindings*] exprs*)`
- `(loop [bindings*] exprs*)`
- `(recur exprs*)`
- `(throw expr)`
- `(try expr* catch-clause* finally-clause?)`

nil/false/eos/‘()

	Clojure	CL	Scheme	Java
nil	nil	nil/‘()	-	null
true/false	true/false	-	#t/#f	true/false
Conditional	nil or false/ everything else	nil/non-nil	#f/non-#f	true/false
singleton empty list?	No	‘()	‘()	No
end-of-seq	(seq eos) -> nil	nil	‘()	FALSE
Host null/ true/false	nil/true/false	N/A	N/A	N/A
Library uses concrete types	No	cons/vector	pair	No

Equality

- = is value equality
- identical? is reference equality
  - rarely used in Clojure
- = as per Henry Baker's *egal*
  - Immutable parts compare by value
  - Mutable references by identity
- Generalized equality for collections
  - Partitioned as Sequentials, Maps, Sets

Functions

- First-class values

```
(def five 5)
(def sqr (fn [x] (* x x)))
(sqr five)
25
```

- Maps are functions of their keys

```
(def m {:fred :ethel :ricky :lucy})
(m :fred)
:ethel
```

Function Details

- Multiple distinct bodies in single function object
    - Supports fast 0/1/2...N dispatch with no conditional
- Variable arity with &
- Can refer to self using (fn name [args] ...)
- Closure over enclosing lexical scope

Function Example

```
(defn complement
  "Takes a fn f and returns a fn that takes the
same arguments as f, has the same effects, if
any, and returns the opposite truth value."
  [f]
  (fn
    ([] (not (f)))
    ([x] (not (f x)))
    ([x y] (not (f x y)))
    ([x y & zs] (not (apply f x y zs)))))
```

- Function returns function
- Closure over 'f'
- apply

Pervasive Destructuring

- Abstract structural binding
- In let/loop binding lists, fn parameter lists, and any macro that expands into a let or fn
- Vector binding forms destructure *sequential* things
  - vectors, lists, seqs, strings, arrays, and anything that supports nth
- Map binding forms destructure *associative* things
  - maps, vectors, strings and arrays (the latter three have integer keys)

Why Destructure?

without destructuring,
next-fib-pair is dominated by
code to “pick apart” pair

```
(defn next-fib-pair
  [pair]
  [(second pair) (+ (first pair) (second pair))])

(iterate next-fib-pair [0 1])
-> ([0 1] [1 1] [1 2] [2 3] [3 5] [5 8] [8 13]...)
```

Sequential Destructure

or you can do the same thing with a simple [] ...

```
(defn next-fib-pair
  [[a b]]
  [b (+ a b)])

(iterate next-fib-pair [0 1])
-> ([0 1] [1 1] [1 2] [2 3] [3 5] [5 8] [8 13] ...)
```

Simple Things Inline

which makes next-fib-pair so simple that you will probably inline it away!

```
(defn fibs
  []
  (map first
   (iterate (fn [[a b]] [b (+ a b)]) [0 1])))
```

What About Maps?

same problem as before:
code dominated by
picking apart person

```
(defn format-name
  [person]
  (str/join " " [(:salutation person)
                 (:first-name person)
                 (:last-name person)]))

(format-name
  {:salutation "Mr." :first-name "John" :last-name "Doe"})
-> "Mr. John Doe"
```

Map Destructuring

pick apart name

```
(defn format-name
  [name]
  (let [{salutation :salutation
         first-name :first-name
         last-name :last-name}
        name]
    (str/join " " [salutation first-name last-name])))

(format-name
  {:salutation "Mr." :first-name "John" :last-name "Doe"})
-> "Mr. John Doe"
```

The :keys Option

a common scenario:
parameter names and key names are the same, so say them only once

```
(defn format-name
  [{:keys [salutation first-name last-name]}]
  (str/join " " [salutation first-name last-name]))

(format-name
  {:salutation "Mr." :first-name "John" :last-name "Doe"})
-> "Mr. John Doe"
```

Optional Keyword Args

not a language feature, simply a consequence of variable arity fns plus map destructuring

```
(defn game
  [planet & {:keys [human-players computer-players]}]
  (println "Total players: "
           (+ human-players computer-players)))

(game "Mars" :human-players 1 :computer-players 2)
Total players:  3
```

Destructuring (examples)

```
(let [[a b c & d :as e] [1 2 3 4 5 6 7]]
  [a b c d e])
-> [1 2 3 (4 5 6 7) [1 2 3 4 5 6 7]]

(let [[[x1 y1][x2 y2]] [[1 2] [3 4]]]
  [x1 y1 x2 y2])
-> [1 2 3 4]

(let [{a :a, b :b, c :c, :as m :or {a 2 b 3}} {:a 5 :c 6}]
  [a b c d m])
-> [5 3 6 {:c 6, :a 5}]

(let [{:keys [a b c]} {:a 5 :c 6}]
  [a b c])
-> [5 nil 6]
```

Tangible Runtime

- Incremental (re)definition
- Vars
- Namespaces
- Reader
- Evaluation

Vars

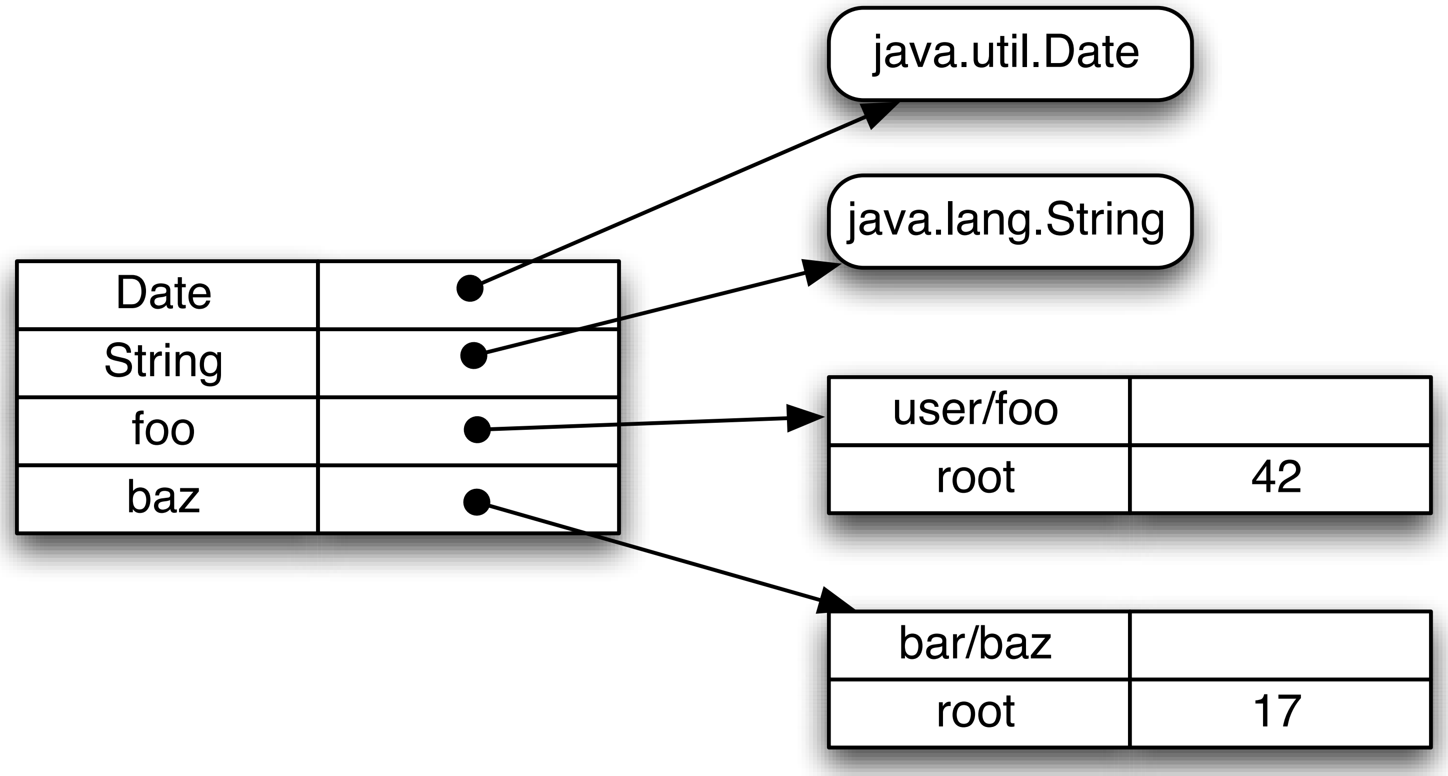
- Similar to CL's special vars
    - dynamic scope, stack discipline
- Shared root binding established by *def*
    - root can be unbound
- Vars established by *def* are interned in namespaces
- Can be *set!* but only if first bound using *binding* (not *let*)
    - Thread-local semantics
- Functions stored in vars, so they too can be dynamically rebound

Namespaces

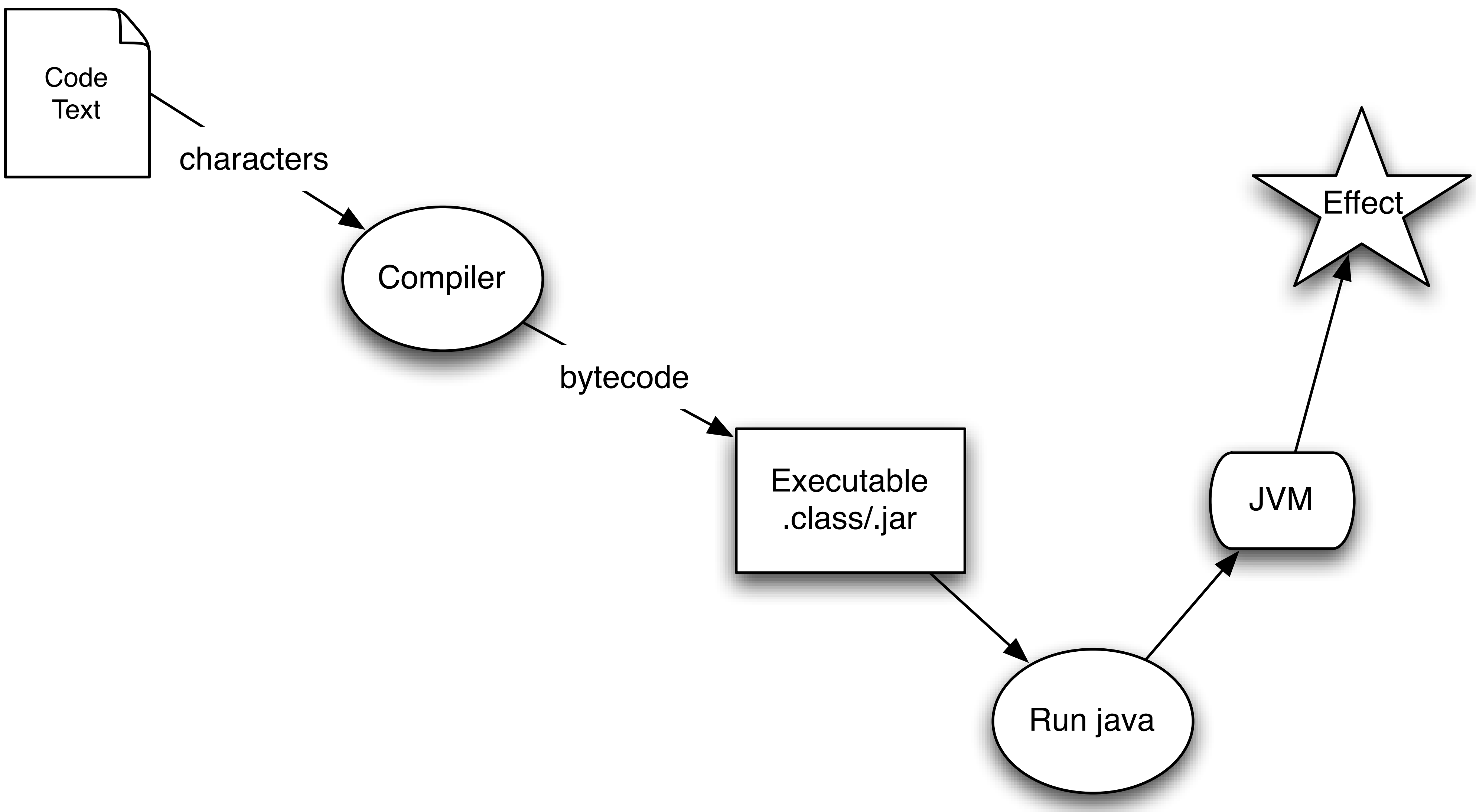
- Uniquely (globally) named by simple (non-qualified) symbol
  - Multi-segment names correspond to Java classpath naming - com.mycompany.ns
- Map of simple symbols to Vars or Classes
- Map of simple symbols to other namespaces (aliases)
- Every simple symbol can have only one meaning per namespace

Namespaces (example)

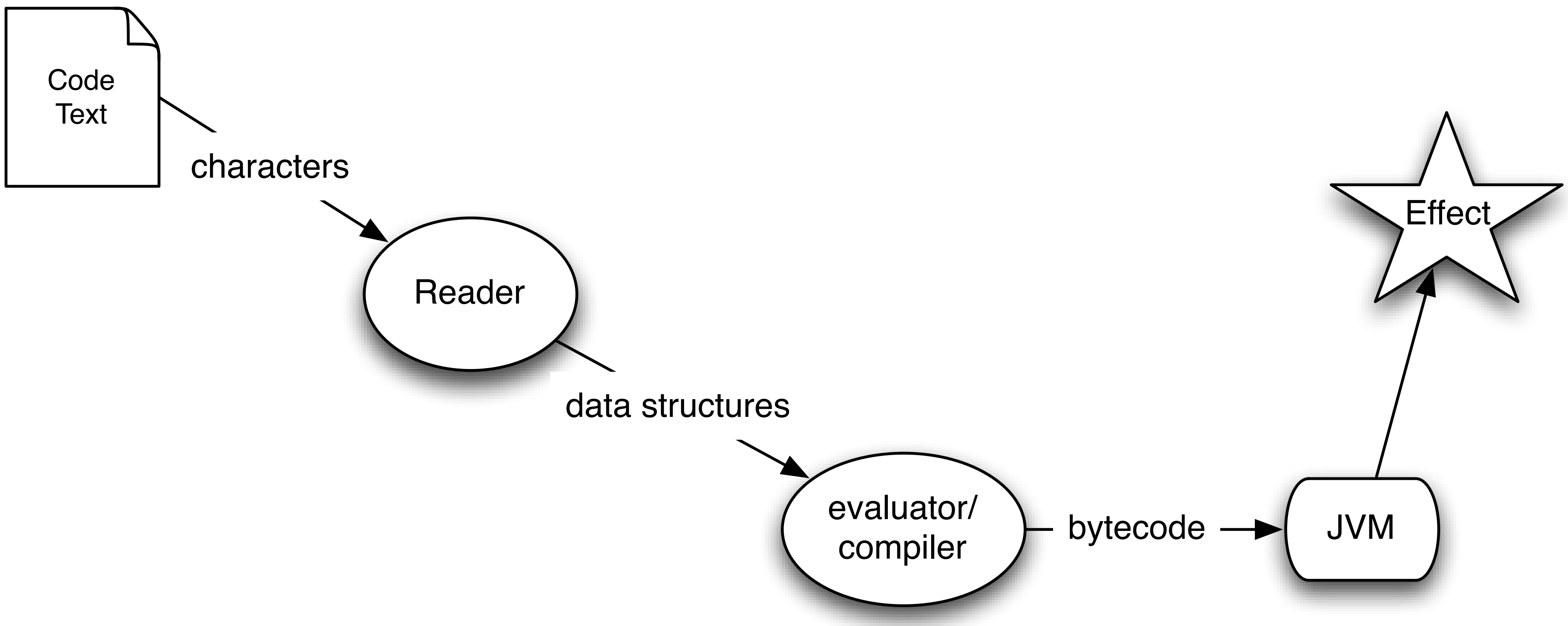
```
user=> (def foo 42)
#'user/foo
user=> (import 'java.util.Date)
user=> (ns bar)

bar=> (def baz 17)
#'bar/baz
bar=> (in-ns 'user)
#<Namespace user>
user=> (refer 'bar)

user=> Date
java.util.Date
user=> baz
17
user=> foo
42
```



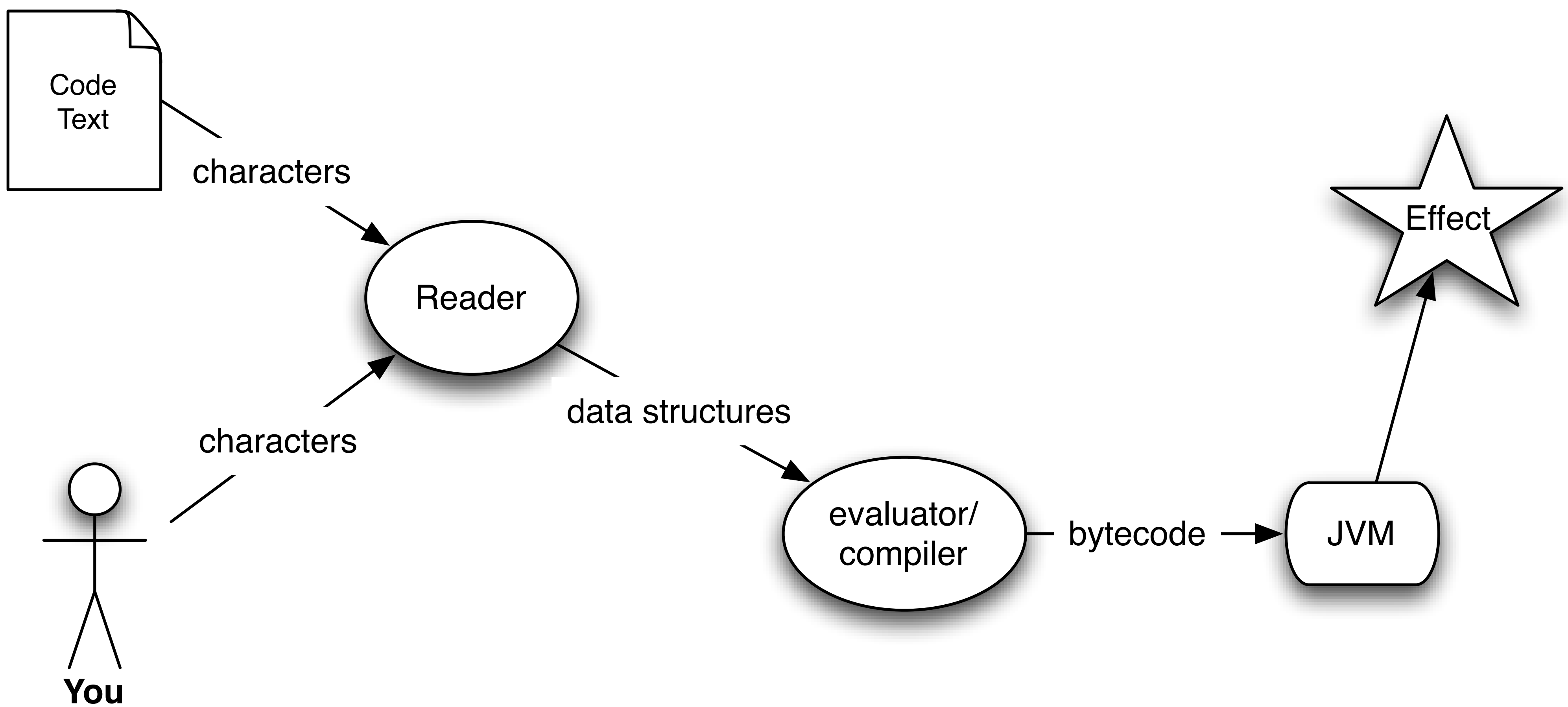
Traditional evaluation



Clojure Evaluation



Interactivity



The Reader

- Turns text into data
- Called when code is loaded from file, or REPL
- Available to programs at runtime
- print/read convenient text-based serialization

```
(read-string "(a [b c] [d e f])")
=> (a [b c] [d e f])

(class (first (read-string "(a [b c] [d e f])")))
=> clojure.lang.Symbol

(class (second (read-string "(a [b c] [d e f])")))
=> clojure.lang.PersistentVector
```

Other Reader Syntax

- commas are whitespace
- `;single-line comment`
- `'form => (quote form)`
- `@form => (deref form)`
- `#'x => (var x)`

Anonymous fn Reader Syntax

- `#(...) => (fn [args] (...))`

  `#()` cannot be nested

- %, %1, %2 etc reserved for parameters

```
#(list %1 %2) =>
  (fn [p1__123 p2__124]
    (list p1__123 p2__124))
% => %1
```

Metadata Reader Syntax

- Symbols, Lists, Vectors, Sets and Maps can have metadata - a map associated with the object.
- `^` first reads the metadata map and attaches it to the next form read:
  - `^{:a 1 :b 2} [1 2 3]` yields the vector `[1 2 3]` with a metadata map of `{:a 1 :b 2}`.
- Metadata literal can be a simple symbol or keyword, treated as a single entry map with a key of :tag and a value of the symbol/keyword:
  - `^String x` is the same as `^{:tag String} x`.

Reader Summary

- Reader is side-effect free
- But not context-free
  - syntax-quote and :: do resolution
- Rich data structure set
- No user-defined reader macros

Macros

- Supplied with Clojure, and defined by user
- Argument forms are passed as data to the macro function, which returns a new data structure as a replacement for the macro call
- `(or x y)`
- becomes:

```
(let [or__158 x]
  (if or__158 or__158 y))
```

- Many things that are 'built-in' to other languages are just macros in Clojure

State - You're Doing it Wrong

- Mutable objects are the new spaghetti code
  - Hard to understand, test, reason about
  - Concurrency disaster
  - Terrible as a default architecture
    - (Java/C#/Python/Ruby/Groovy/CLOS...)
- Doing the right thing is very difficult
  - Language support matters!

Concurrency Mechanisms

- Conventional way:
  - Direct references to mutable objects
  - Lock and worry (manual/convention)
- Clojure way:
  - Indirect references to immutable persistent data structures (inspired by SML's `ref`)
  - Concurrency semantics for references
    - Automatic/enforced
    - <u>No locks in user code!</u>

Java Integration

- Clojure strings are Java Strings, numbers are Numbers, collections implement Collection, fns implement Callable and Runnable etc.
- Core abstractions, like seq, are Java interfaces
- Clojure seq library works on Java Iterables, Strings and arrays.
- Implement and extend Java interfaces and classes
- Primitive arithmetic support equals Java's speed.

Why Clojure?

- Expressive, elegant, simple
  - Approachable functional programming
  - Robust, easy-to-use concurrency
- Powerful extensibility, good performance
- Leverage an established, accepted platform
- Focus on your problem
  - Get more done
  - Have more fun

names and places (lab)

it's all data
(lab)

Programming with Functions, Values and Abstractions

Functional Programming

- Immutable data + first-class functions
- Functions produce same output given same input, and are free of side effects
- Could always be done by discipline/convention
- Pure functional languages tend to strongly static types (ML, Haskell)
  - Not for everyone, or every task
- Dynamic functional languages are rarer
  - Clojure, Erlang

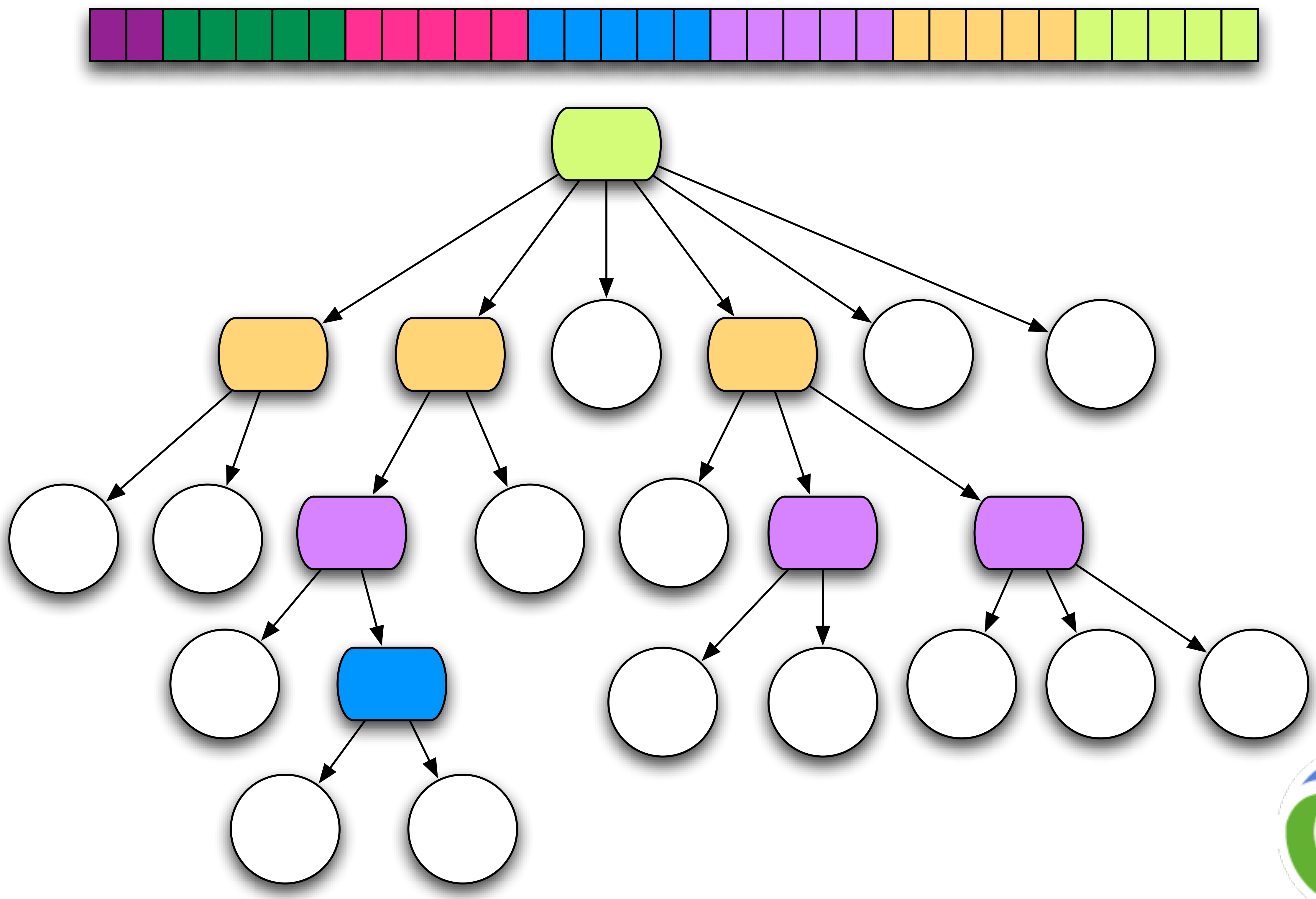
Why Functional Programming?

- Easier to reason about
- Easier to test
- Essential for concurrency (IMO)
  - Java Concurrency in Practice - Goetz
- Additional benefits for purely functional languages (static analysis, proof, program transformation), but not Clojure

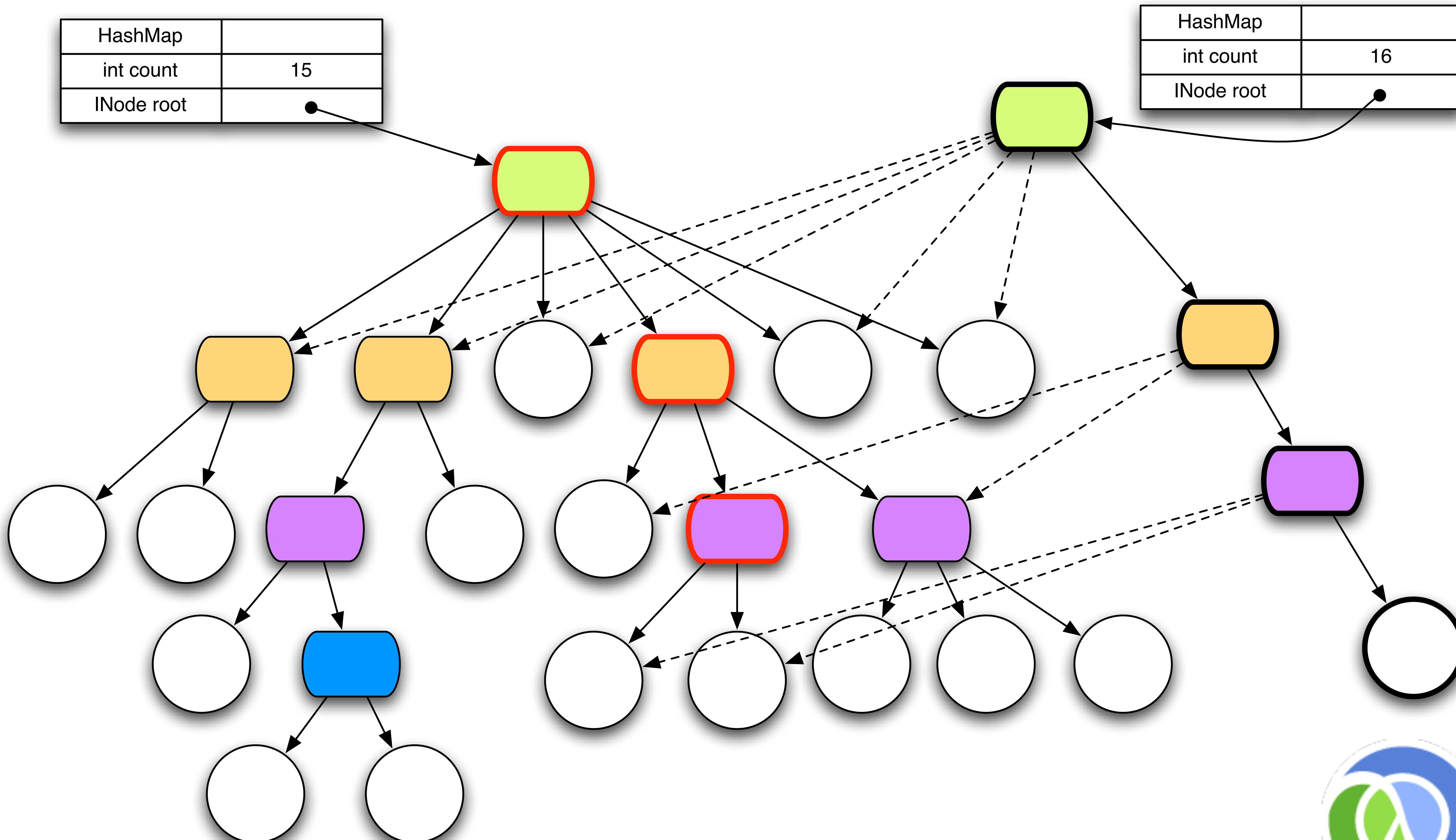
Persistent Data Structures

- Composite values - immutable
- ‘Change’ is merely a function, takes one value and returns another, ‘changed’ value
- Collection maintains its performance guarantees
  - Therefore new versions are not full copies
- Old version of the collection is still available after 'changes', with same performance
- Example - hash map/set and vector based upon array mapped hash tries (Bagwell)

Bit-partitioned hash tries



Path Copying



Structural Sharing

- Key to efficient 'copies' and therefore persistence
- Everything is immutable so no chance of interference
- Thread safe
- Iteration safe

Idioms

- Looping via recursion (recur)
  - prefer higher-order library fns when appropriate
- Iteration via map and list comprehensions (for)
- Accumulation and value building via reduce and into

Recursive Loops

- No mutable locals in Clojure
- No tail recursion optimization in the JVM
- *recur* op does constant-space recursive looping
- Rebinds and jumps to nearest *loop* or function frame

```
(defn zipm [keys vals]
  (loop [m {}
         ks (seq keys)
         vs (seq vals)]
      (if (and ks vs)
        (recur (assoc m (first ks) (first vs))
               (next ks)
               (next vs))
        m)))

(zipm [:a :b :c] [1 2 3])
=> {:a 1, :b 2, :c 3}
```

Loop Alternatives

```
(loop [m {}
       [k & ks :as keys] (seq keys)
       [v & vs :as vals] (seq vals)]
  (if (and keys vals)
    (recur (assoc m k v) ks vs)
    m))

;reduce with adder fn
(reduce (fn [m [k v]] (assoc m k v))
        {} (map vector keys vals))

;apply data constructor fn
(apply hash-map (interleave keys vals))

;map into empty (or not!) structure
(into {} (map vector keys vals))

;get lucky
(zipmap keys vals) ;already in there!
```

Sequence Comprehensions (for)

- Lazy sequence generator/consumer
  - for is not an imperative loop!
- control and binding clauses:
  - :when, :while, :let

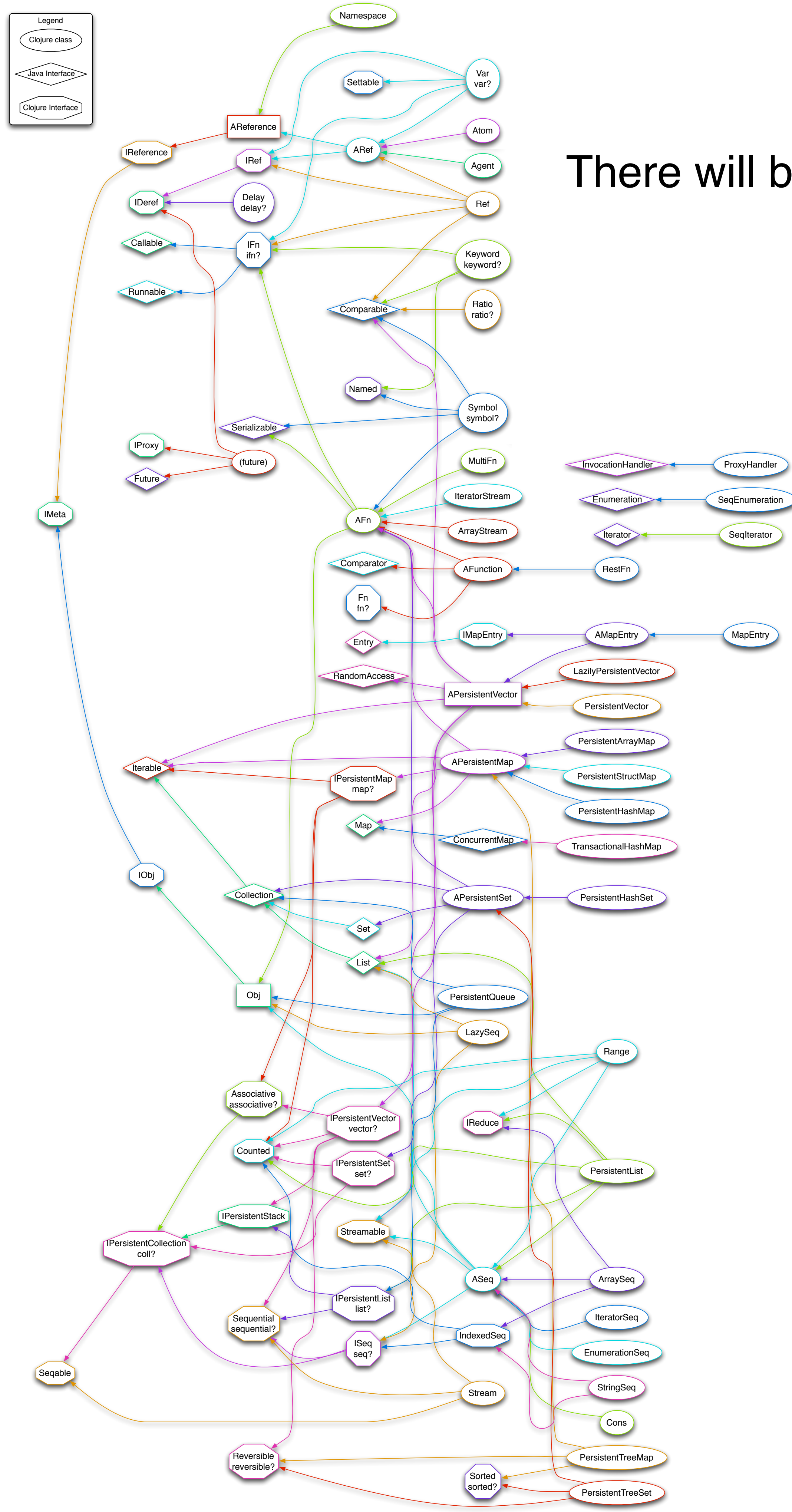
```
(for [x (range 2) y (range 3)] [x y])
=> ([0 0] [0 1] [0 2] [1 0] [1 1] [1 2])

(take 20 (for [x (range 100000000) y (range 1000000)
               :while (< y x)]
           [x y]))
=> ([1 0] [2 0] [2 1] [3 0] [3 1] [3 2] [4 0] [4 1]
[4 2] [4 3] [5 0] [5 1] [5 2] [5 3] [5 4] [6 0] [6 1]
[6 2] [6 3] [6 4])
```

Benefits of Abstraction

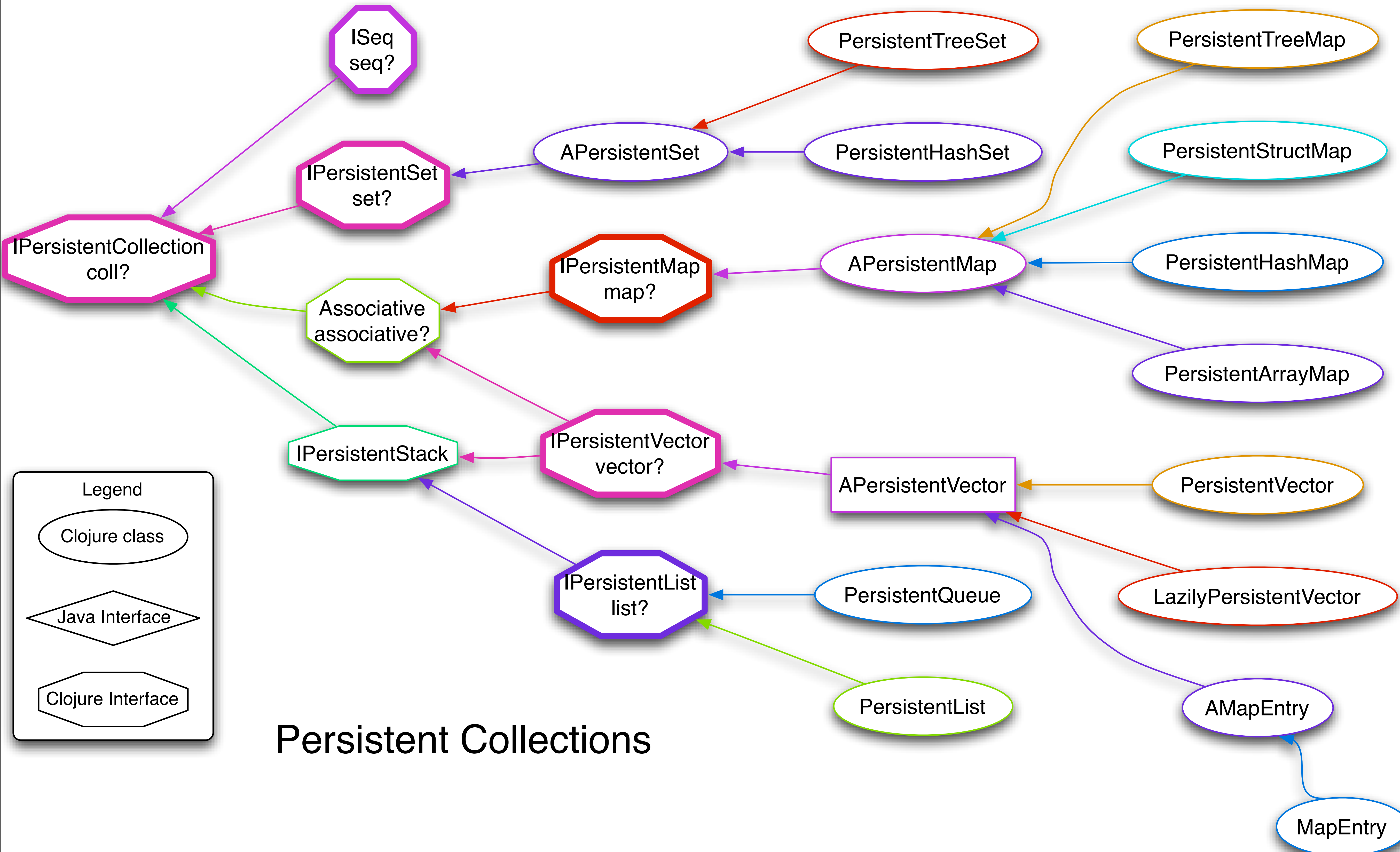
- "It is better to have 100 functions operate on one data structure than to have 10 functions operate on 10 data structures." - Alan J. Perlis
- Better still -100 functions per abstraction
  - E.g. `seq`, implemented for all Clojure collections, all Java collections, Strings, regex matches, files etc.
  - Many library functions defined on seqs

Clojure's Abstractions

- Sequences, replace traditional Lisp lists
  - Seqs on all Clojure collections, all Java collections, Strings, regex matches, files...
  - Can be lazy - like generators
- All Collections
- Functions (call-ability)
  - Maps/vectors/sets are functions
- Many implementations
  - Extensible from Java and Clojure



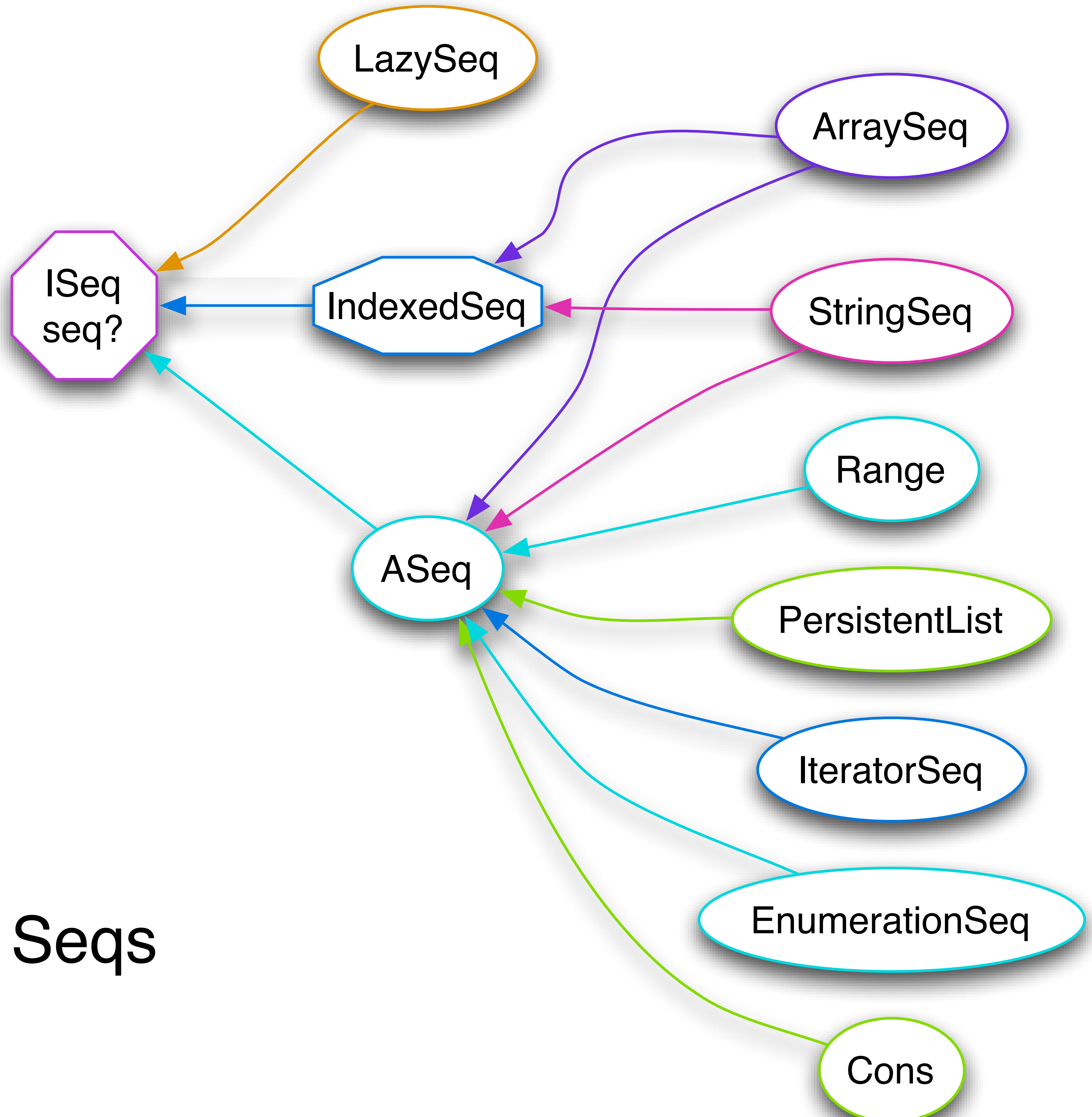
There will be a quiz!



Persistent Collections

Collections

- `conj`, `count`, `seq`
- Lists
  - `cons`, `peek`, `pop`, `list`, `list*`
- Maps
  - `assoc`, `dissoc`, `get`, `contains?`, `find`, `keys`, `vals`
  - `(seq map)` yields map entries with `[key val]`
- Vectors
  - `get`, `nth`, `assoc`, `subvec`, `peek`, `pop`
- Sets
  - `disj`, `get`, `union`, `difference`, `intersection`


LazySeq
ArraySeq
ISeq
seq?
IndexedSeq
StringSeq
Range
ASeq
PersistentList
IteratorSeq
EnumerationSeq
Cons

Seqs

Sequences

- Abstraction of traditional Lisp lists
- `(seq coll)`
  - if collection is non-empty, return seq object on it, else nil
- `(first seq)`
  - returns the first element
- `(rest seq)`
  - returns a sequence of the rest of the elements

Lazy Seqs

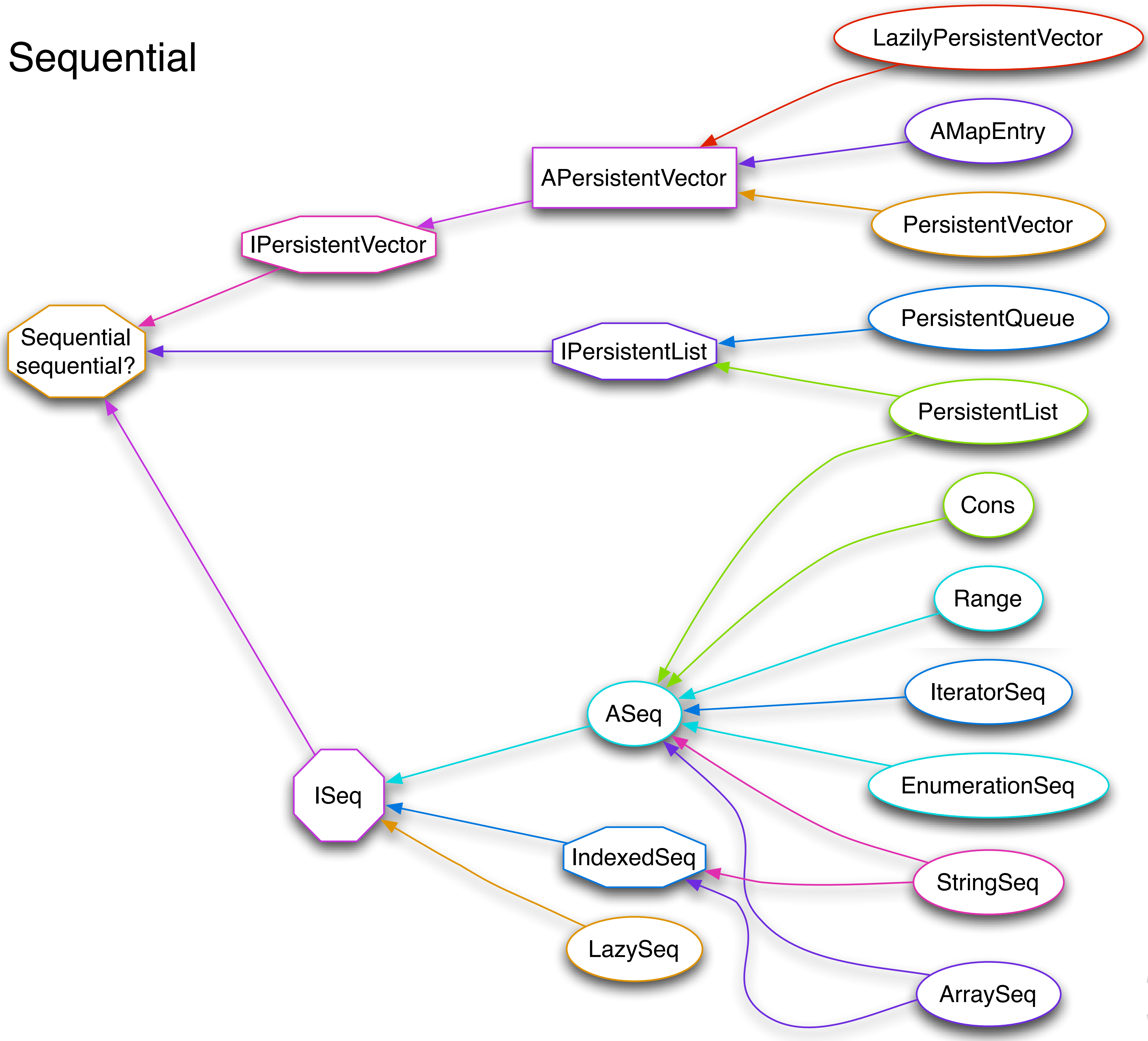
- Not produced until (and as) requested
- Define your own lazy seq-producing functions using the *lazy-seq* macro
- Seqs can be used like generators
- Lazy and concrete seqs interoperate - no separate lazy library

```
;the library function take
(defn take [n coll]
  (lazy-seq
   (when (pos? n)
     (when-let [s (seq coll)]
      (cons (first s) (take (dec n) (rest s)))))))
```

Laziness

- Most of the core library functions that produce sequences do so lazily
  - e.g. map, filter etc
  - And thus if they consume sequences, do so lazily as well
- Avoids creating full intermediate results
- Create only as much as you consume
- Work with infinite sequences, datasets larger than memory

Sequential



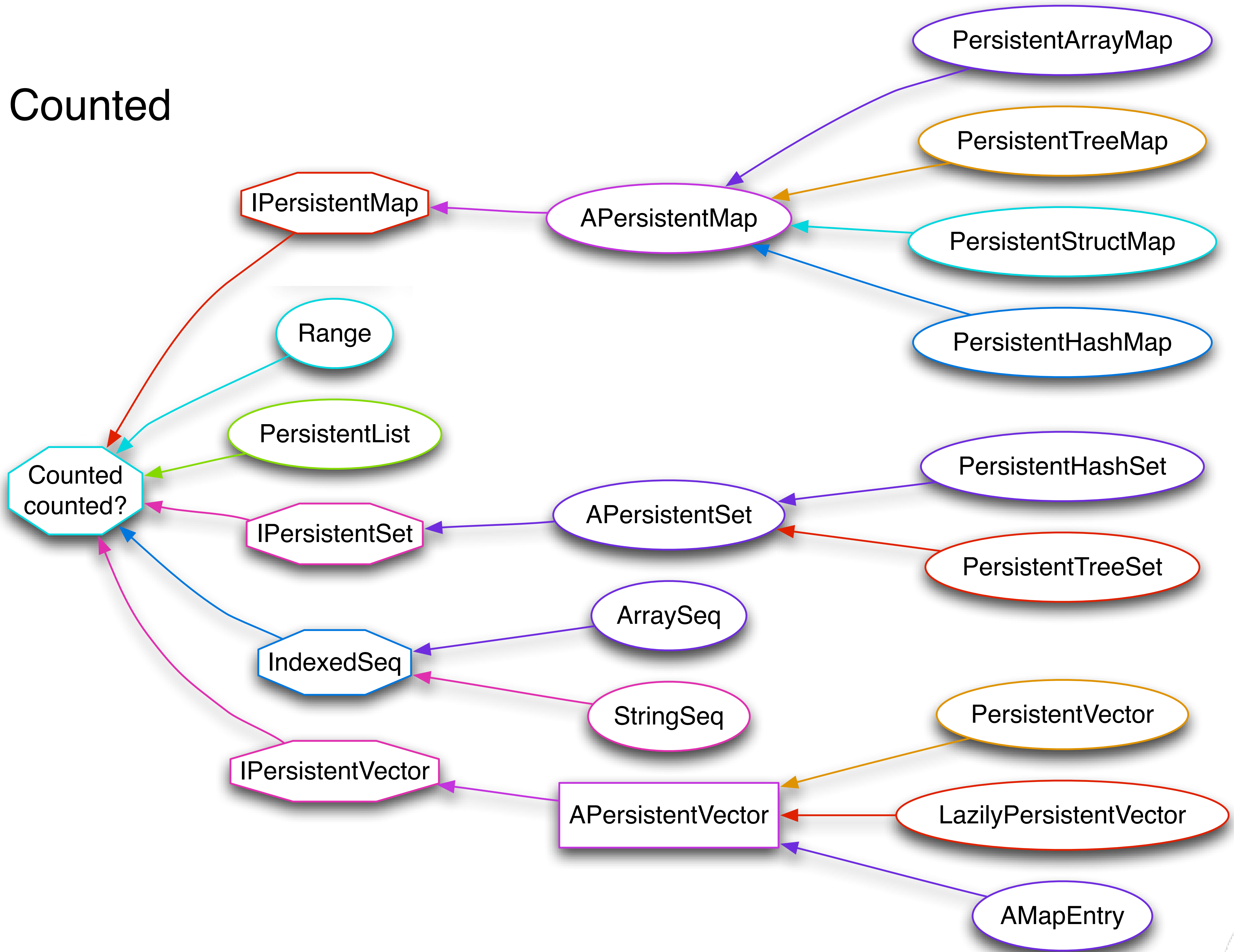
LazilyPersistentVector
AMapEntry
APersistentVector
PersistentVector
IPersistentVector
Sequential
sequential?
IPersistentList
PersistentQueue
PersistentList
Cons
Range
IteratorSeq
ASeq
EnumerationSeq
ISeq
IndexedSeq
StringSeq
LazySeq
ArraySeq

Sequential

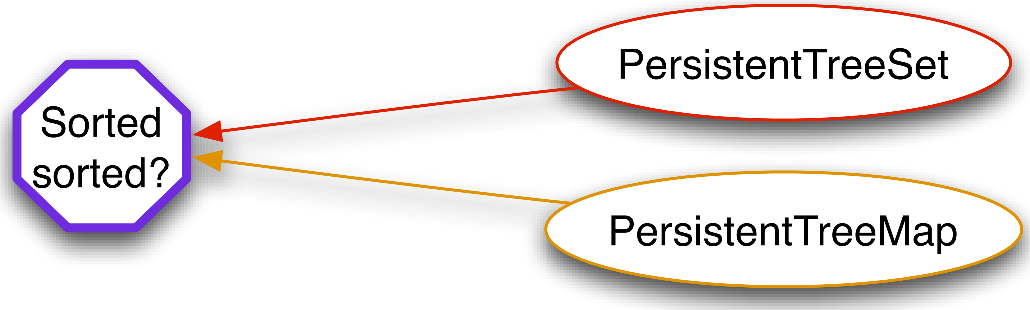
- Example of marker interface
  - No methods of its own
- Used for generalized equality
- Similar generality for maps, sets

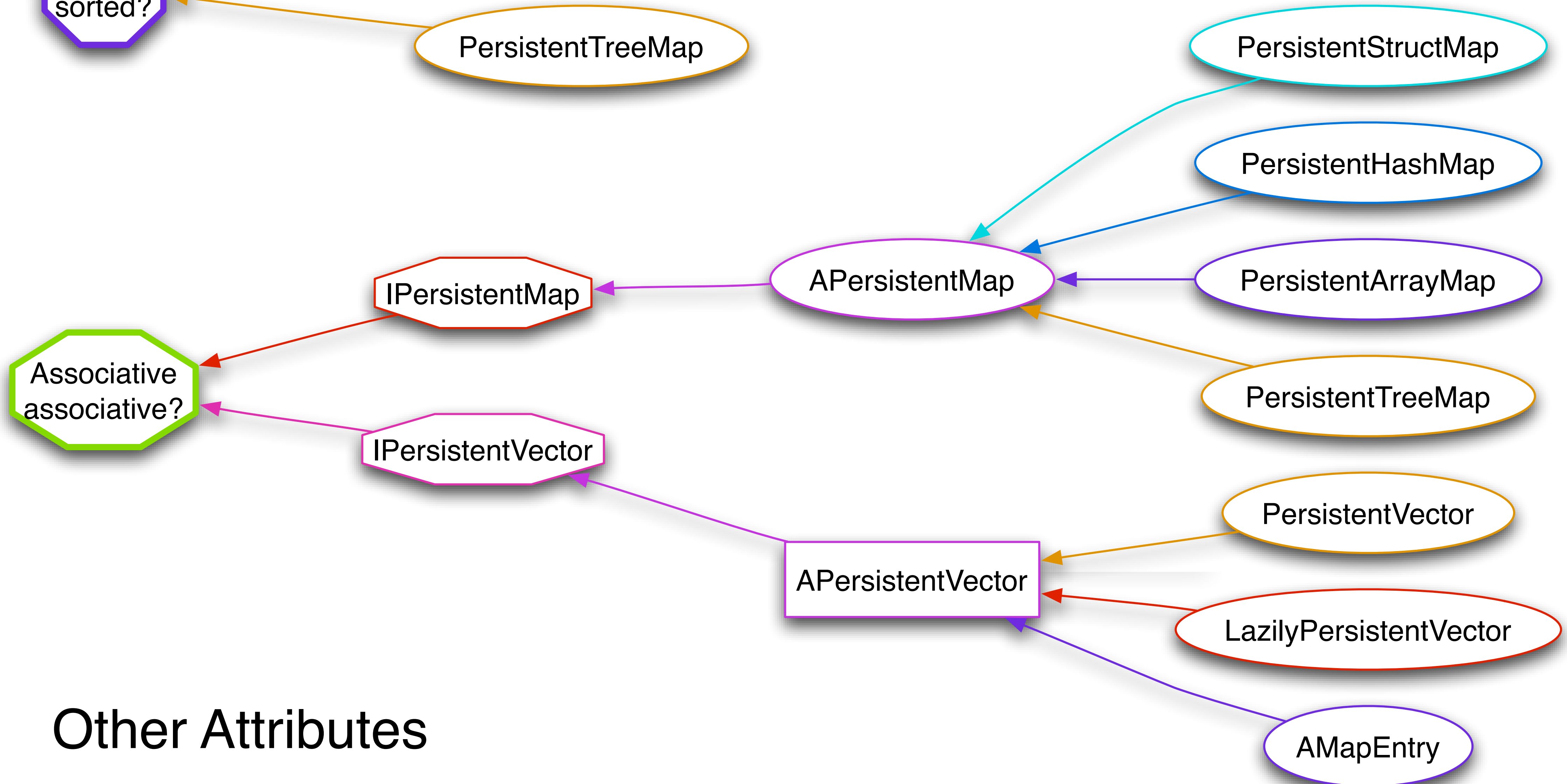
```
(= [1 2 3]
   '(1 2 3)
   (range 1 4)
   (java.util.ArrayList. [1 2 3]))
-> true
```

Counted

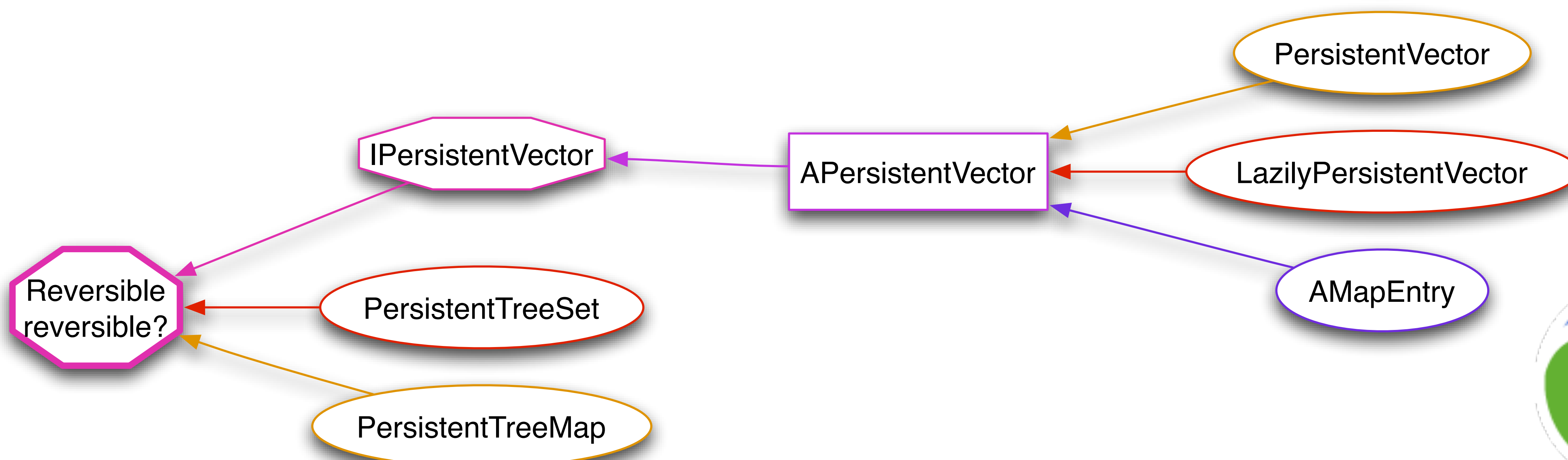


PersistentArrayMap
PersistentTreeMap
IPersistentMap
APersistentMap
PersistentStructMap
PersistentHashMap
Range
PersistentList
Counted
counted?
PersistentHashSet
APersistentSet
IPersistentSet
PersistentTreeSet
ArraySeq
IndexedSeq
StringSeq
PersistentVector
IPersistentVector
APersistentVector
LazilyPersistentVector
AMapEntry





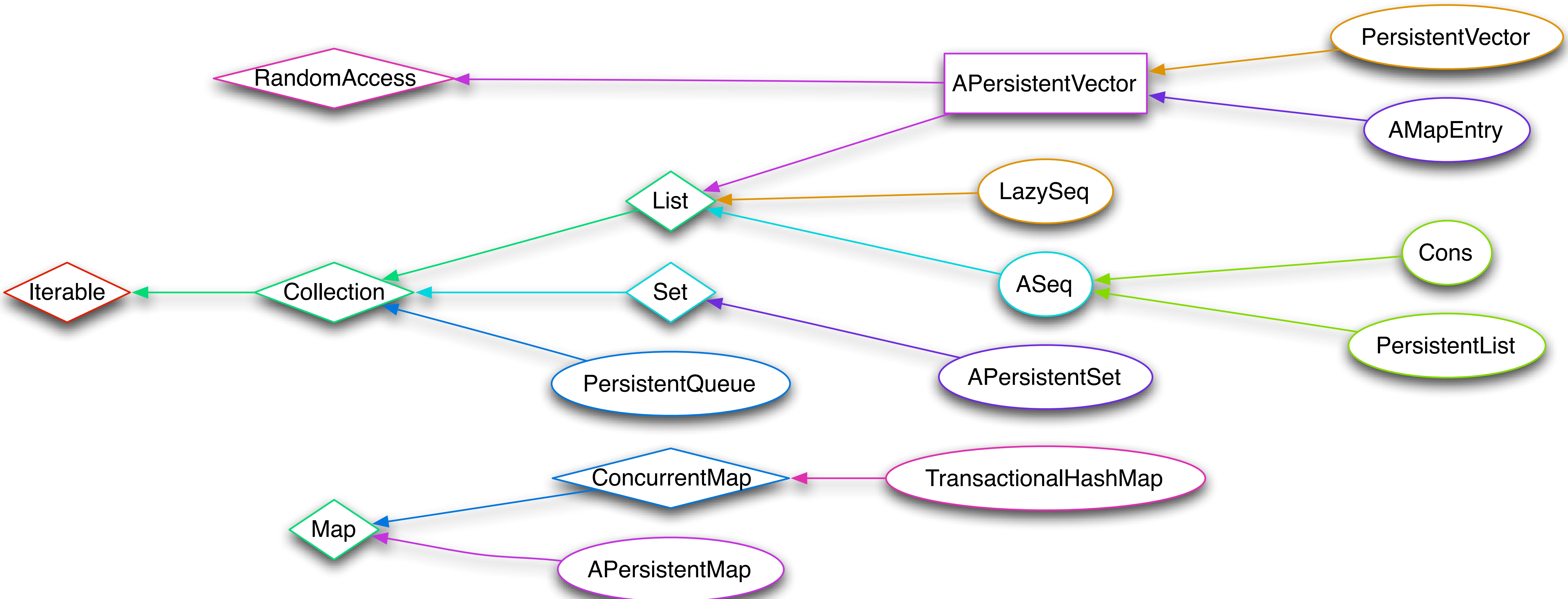
Other Attributes



Other Attributes

- Counted
  - `count` is O(1)
- Sorted
  - `subseq, rsubseq`
- Associative
  - `assoc`, map-style destructuring
- Reversible
  - `rseq`

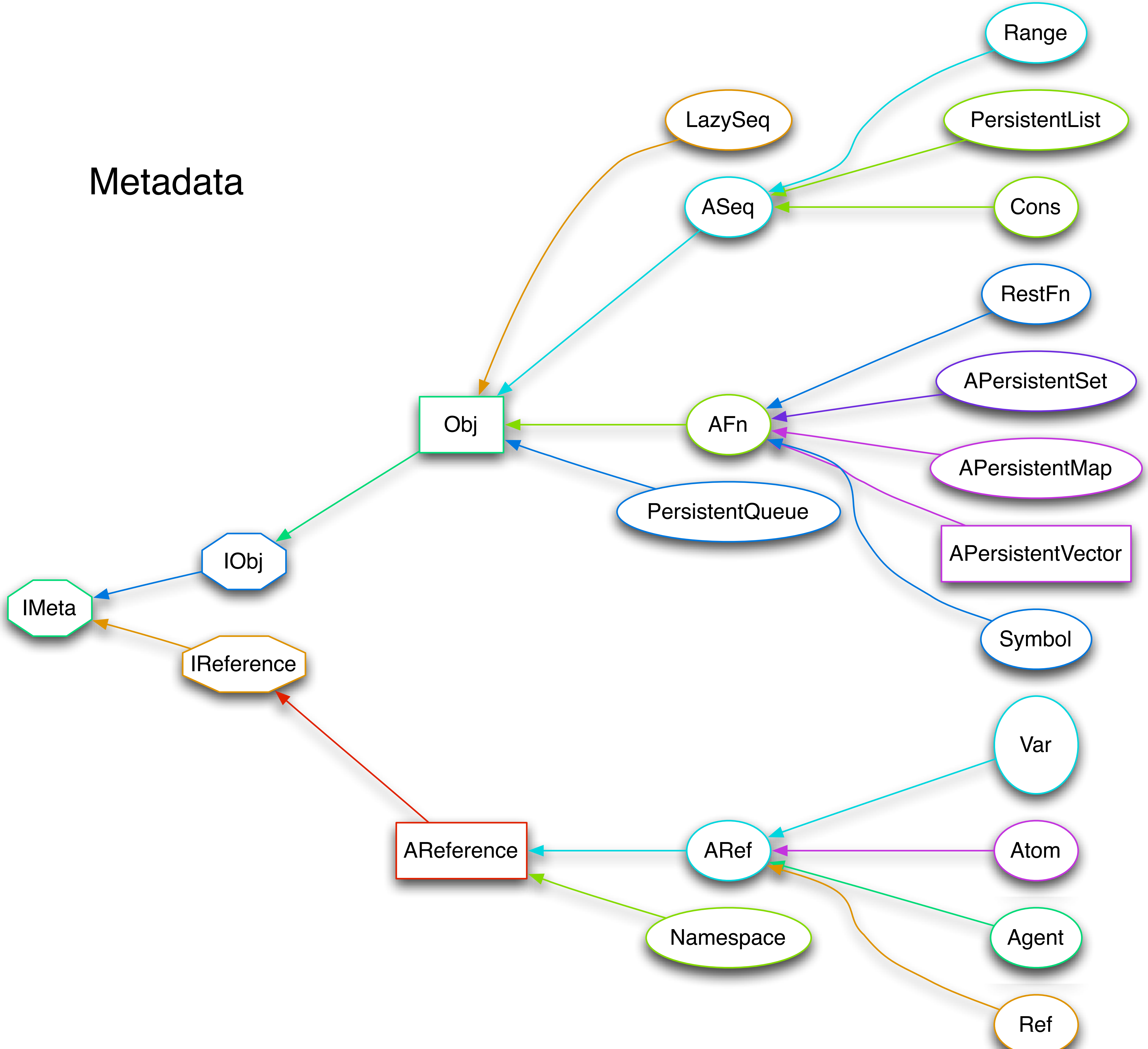
Java Collections



Java Collection Interop

- Clojure collections can be passed to any Java method calling for a standard Java collection
- Supports the entire non-mutable interface of corresponding Java type
- No conversions or copying

Metadata

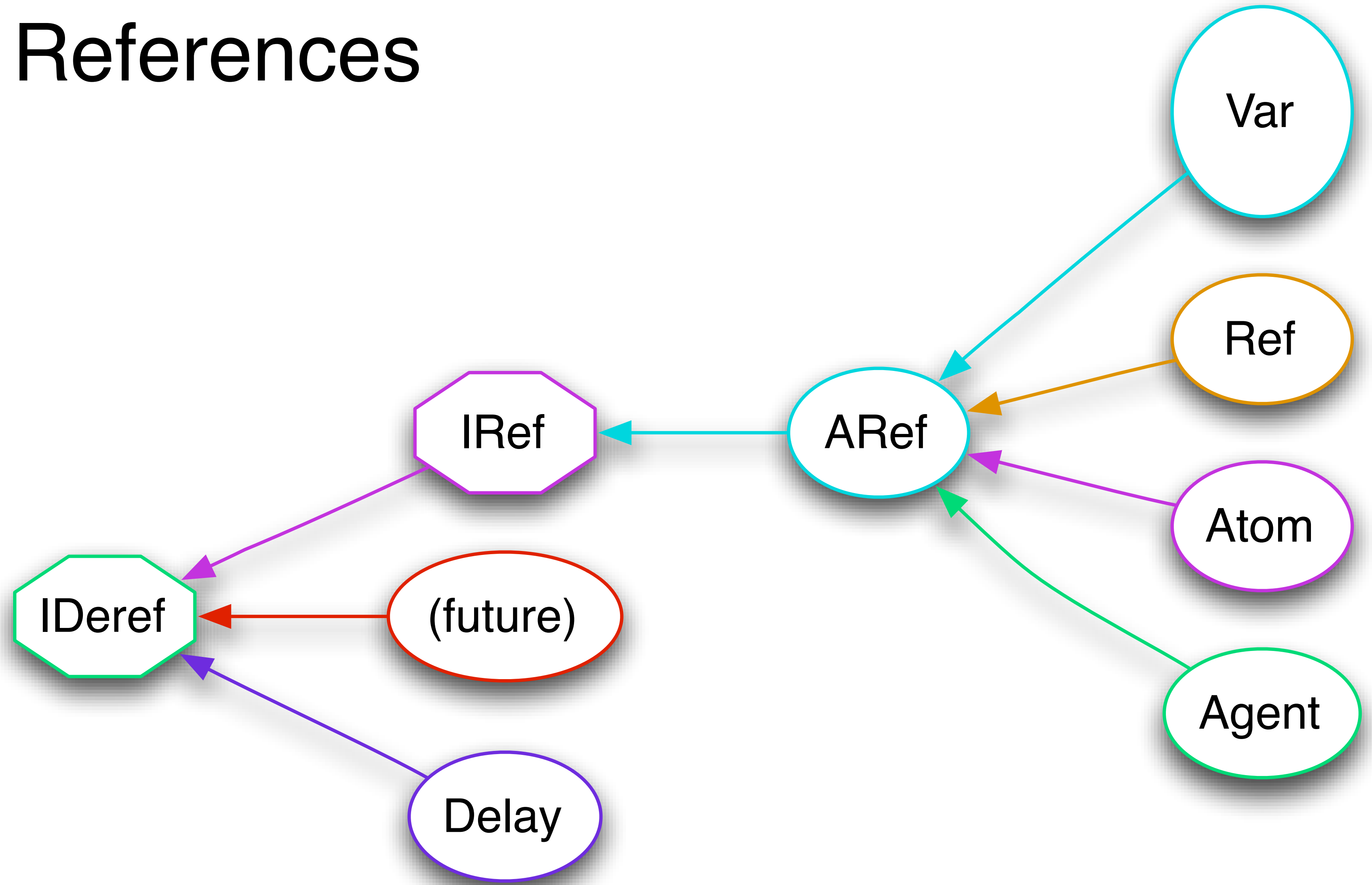


Range
LazySeq
PersistentList
ASeq
Cons
RestFn
APersistentSet
Obj
AFn
APersistentMap
PersistentQueue
APersistentVector
IObj
IMeta
Symbol
IReference
Var
AReference
ARef
Atom
Namespace
Agent
Ref

Metadata

- Orthogonal to the logical value of the data
- Symbols, collections and references support a metadata map
- Does not impact equality semantics, nor seen in operations on the value
- Immutable types have immutable metadata
  - `vary-meta, with-meta`
- Reference types have swappable metadata
  - `alter-meta!, reset-meta!`

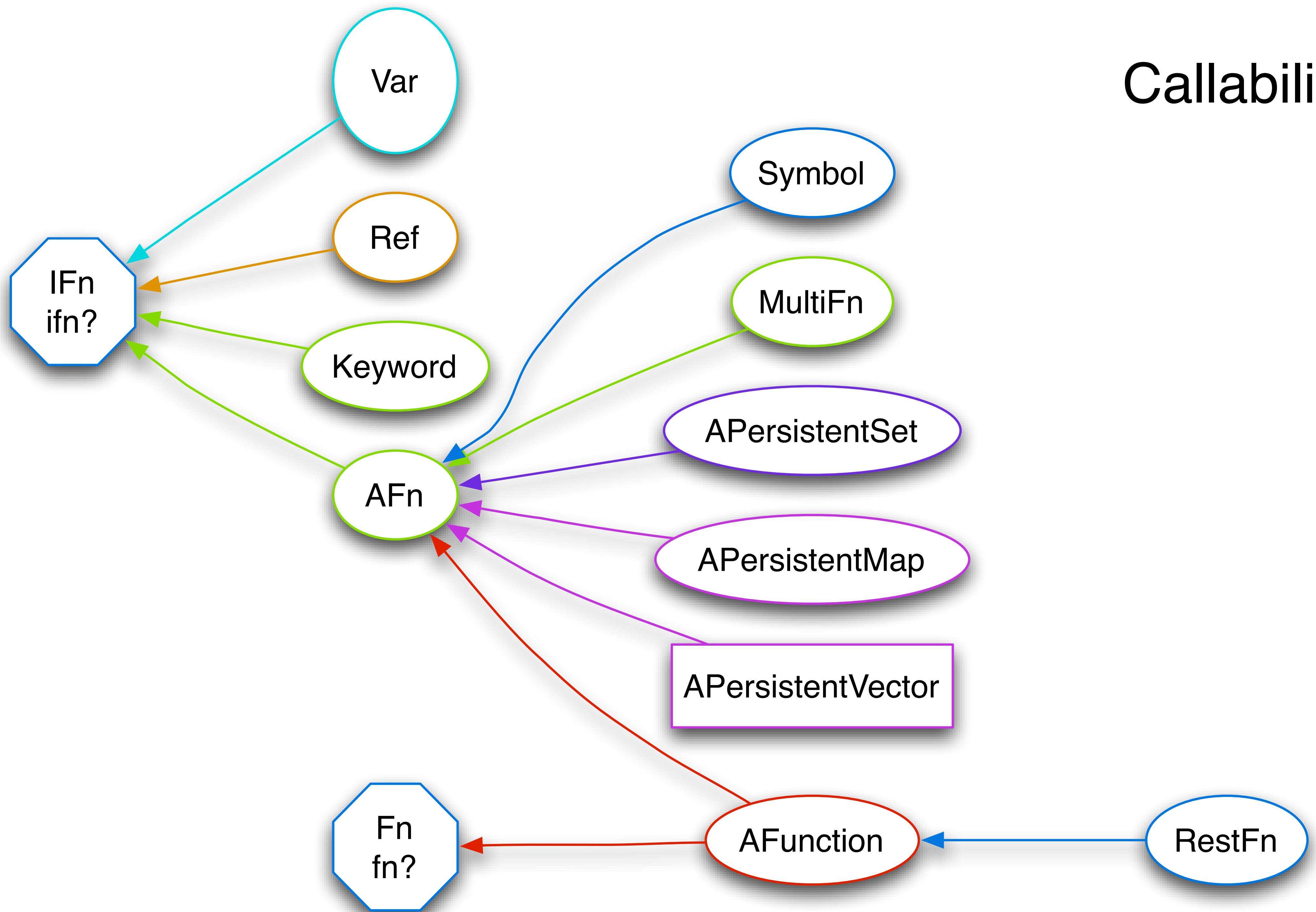
References



References

- All support `deref/@`
- Delays - not-yet-run computation
- Futures - running in a thread pool - deref will block until done
- Others (IRefs)
  - Support watches
  - Support validators

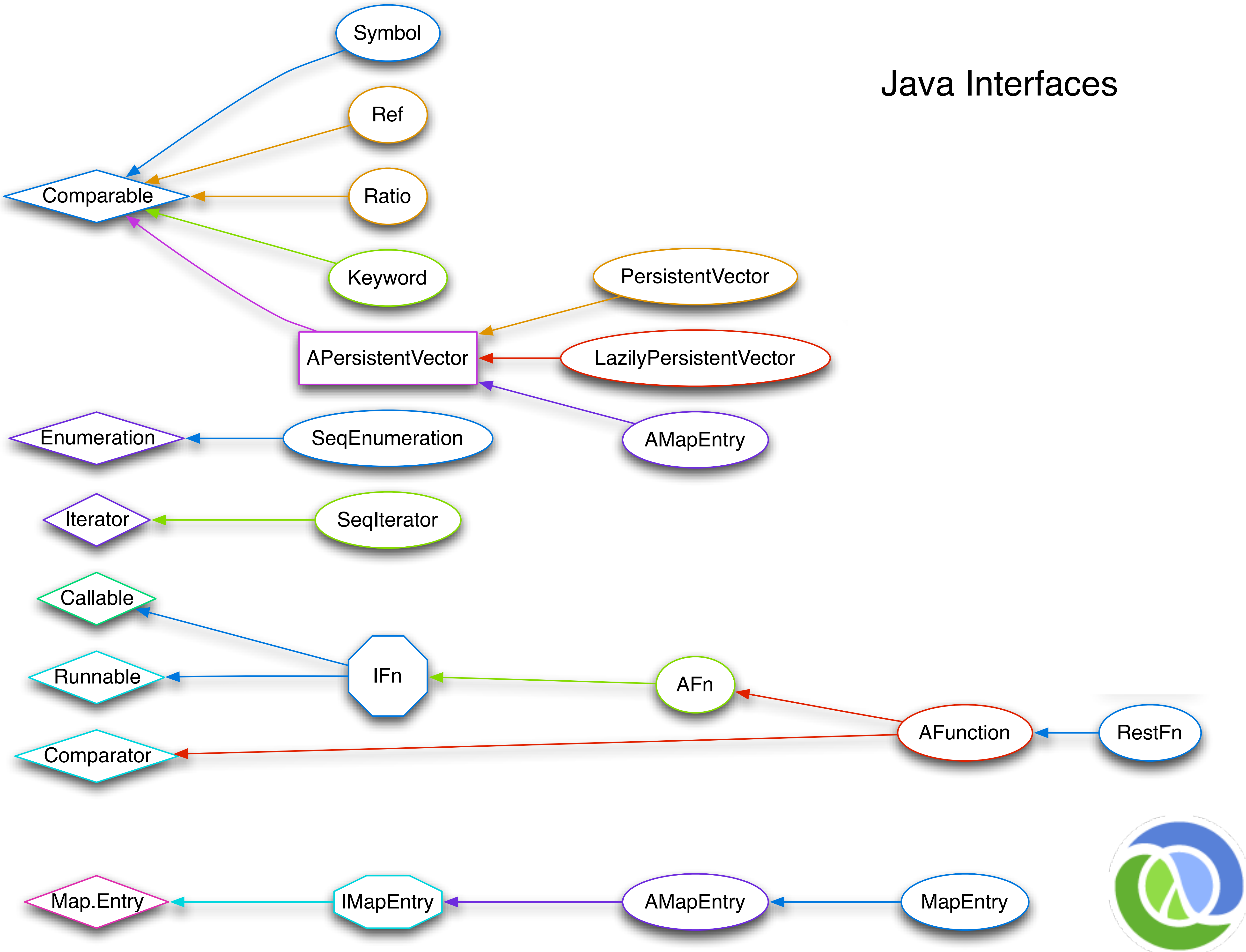
Callability



Callability

- (can-go-here ...)
- (map or-here ...), (filter or-here ...) etc
- Maps are functions of their keys, sets of their members, vectors of their indexes
- Symbols, keywords take an associative arg and look themselves up
- Vars and refs delegate to their (presumed callable) values

Java Interfaces



Summary

- Clojure provides -
  - a rich set of abstractions
  - efficient persistent data structures implementing those abstractions
  - bridges from those abstractions to Java
  - a large library of pure functions built on the abstractions
- thus making functional programming idiomatic and flexible

project euler
(lab)

example:
refactor apache
commons
indexOfAny

indexOfAny behavior

```
StringUtils.indexOfAny(null, *)              = -1
StringUtils.indexOfAny("", *)                = -1
StringUtils.indexOfAny(*, null)              = -1
StringUtils.indexOfAny(*, [])                = -1
StringUtils.indexOfAny("zzabyycdxx",['z','a']) = 0
StringUtils.indexOfAny("zzabyycdxx",['b','y']) = 3
StringUtils.indexOfAny("aba", ['z'])         = -1
```

indexOfAny impl

```
// From Apache Commons Lang, http://commons.apache.org/lang/
public static int indexOfAny(String str, char[] searchChars)
{
  if (isEmpty(str) || ArrayUtils.isEmpty(searchChars)) {
    return -1;
  }
  for (int i = 0; i < str.length(); i++) {
    char ch = str.charAt(i);
    for (int j = 0; j < searchChars.length; j++) {
      if (searchChars[j] == ch) {
        return i;
      }
    }
  }
  return -1;
}
```

simplify corner cases

```
public static int indexOfAny(String str, char[] searchChars)
{
  when (searchChars)
    for (int i = 0; i < str.length(); i++) {
      char ch = str.charAt(i);
      for (int j = 0; j < searchChars.length; j++) {
        if (searchChars[j] == ch) {
          return i;
        }
      }
    }
  }
}
```

- type decls

```
indexOfAny(str, searchChars) {
  when (searchChars)
    for (i = 0; i < str.length(); i++) {
      ch = str.charAt(i);
      for (j = 0; j < searchChars.length; j++) {
        if (searchChars[j] == ch) {
          return i;
        }
      }
    }
  }
}
```

+ when clause

```
indexOfAny(str, searchChars) {
  when (searchChars)
    for (i = 0; i < str.length(); i++) {
      ch = str.charAt(i);
      when searchChars(ch) i;
    }
  }
}
```

+ comprehension

```
indexOfAny(str, searchChars) {
  when (searchChars)
    for ([i, ch] in indexed(str)) {
      when searchChars(ch) i;
    }
  }
}
```

lispify!

```
(defn index-filter [pred coll]
  (when pred
    (for [[idx elt] (indexed coll) :when (pred elt)] idx)))
```

functional

is

simpler

	imperative	**functional**
functions	1	1
classes	1	0
internal exit points	2	0
variables	3	0
branches	4	0
boolean ops	1	0
function calls*	6	3
total	***18***	***4***

functional

is

more general!

reusing index-filter

```
; idxs of heads in stream of coin flips
(index-filter #{:h}
[:t :t :h :t :h :t :t :t :h :h])
-> (2 4 8 9)

; Fibonaccis pass 1000 at n=17
(first
  (index-filter #(> % 1000) (fibo)))
-> 17
```

imperative	**functional**
searches strings	searches *any sequence*
matches characters	matches *any predicate*
returns first match	returns *lazy seq of all matches*

"It is better to have 100 functions operate on one data structure than to have 10 functions operate on 10 data structures."

--Alan J. Perlis

“Better still: have 100 functions per data *abstraction*.”

--Rich Hickey

compojure

http endpoints are functions

```
{request} -> handler -> {response}
```

basic handler

```
(defn hello-world [request]
  (let [{:keys [request-method uri]}
        request]
    {:status  200
     :headers {}
     :body (str "hello, "
                request-method
                " "
                uri)}))
```

request keys

response keys

running embedded

```
(def application (-> lab-routes
                      handlers/with-logging))

(defn -main [& args]
  (run-jetty
    (var application)
    {:port 8080 :join? false})
  (println "Welcome to the labrepl..."))
```

routes

return nil to ignore inputs

```
(defn hello-world [request]
  (let [{:keys [request-method uri]}
        request]
    (when (and (= request-method :get)
               (= uri "/"))
      {:status  200
       :headers {}
       :body    "The index page"})))
```

test for whatever
you care about

a little macro magic later...

```
(defroutes lab-routes
  (GET "/" [] (home))
  (GET "/labs/:name" [name] (render-lab name))
  (route/files "/")
  (route/not-found "Not Found"))
```

route return values

type	effect
integer	set HTTP status code
string	added to response body
seq, file, stream, url	set response body
keyword	:next => nil, or as string
map	smart merge into response map
fn	(fn request response)
vector	update for each value

middleware
wraps handlers

middleware

```
(defn with-header [handler header value]
  (fn [request]
    (let [response (handler request)]
      (assoc-in
       response
       [:headers header]
       value))))
```

call original handler

modify the result

common middleware

with-params

with-cookies

with-multipart

with-session

html
(hiccup)

html elements

clojure
vector

```
(html [:h1 "hi"])
-> "<h1>hi</h1>"
```

html attributes

clojure
map

```
(html [:a
       {:href "http://clojure.org"}
       "Clojure"])
```

```
<a href="http://clojure.org">Clojure</a>
```

id, class shortcuts

id
follows #

class
follows .

```
(html [:h1#title.main "hi"])
```

```
<h1 class="main" id="title">hi</h1>
```

lab home

```
(defn home []
  (layout/home
   [:ul
    (map
     (fn [lab] [:li (make-url lab)])
     all)]))
```

mix clojure literals...

...with fncalls

middleware

simple function wrapping

```
(def full-routes (-> lab-routes with-logging))

(defroutes app
  (routes full-routes static-routes))
```

compose routes

implementation comparison

feature	clojure impl	oo impl
endpoint	function	interfaces, classes
request	map	interfaces, classes
response	map	interfaces, classes
cookies	map	interfaces, classes
session	map	interfaces, classes
routing	functions, macros	interfaces, classes, config, XML
middleware	functions, macros	interfaces, classes, config, XML, AOP

fns are easy to test!

```
(doseq [lab all]
  (let [url (lab-url lab)
        resp (application {:request-method :get
                                    :uri url})]
    (is
     (= {:status 200
         :headers
         {"Content-Type" "text/html"}}
        (select-keys resp
                     [:status :headers])))))
```

mini-browser
(lab)

Modeling State and Time Part 1

Functions

- Function
  - Depends only on its arguments
  - Given the same arguments, always returns the same value
  - Has no effect on the world
  - Has no notion of time

Functional Programming

- Emphasizes functions
  - Tremendous benefits
- But - most programs are not functions
  - Maybe compilers, theorem provers?
    - But - They execute on a machine
    - Observably consume compute resources

Processes

- Include some notion of change over time
- Might have effects on the world
- Might wait for external events
- Might produce different answers at different times (i.e. have state)
- Many real/interesting programs are processes
- This talk is about one way to deal with state and time *in the local context*

State

- Value of an identity at a time
- Sounds like a variable/field?
  - Name that takes on successive 'values'
- Not quite:
  - i = 0
  - i = 42
  - j = i
  - j is 42? - depends

Variables

- Variables (and fields) in traditional languages are predicated on a single thread of control, one timeline
- Not meaningful when composed
- Adding concurrency breaks them badly
    - Non-atomicity (e.g. of longs)
    - volatile, write visibility
    - Composite operations require locks
    - All workarounds for lack of a time model

Time

- When things happen
  - Before/after
  - Later
  - At the same time (concurrency)
  - Now
- Inherently relative

Value

- An immutable magnitude, quantity, number... *or composite thereof*
- 42 - easy to understand as value
- But traditional OO tends to make us think of composites as something other than values
  - Big mistake
    - aDate.setMonth("January") - ugh!
  - Dates, collections etc are all values

Identity

- A logical entity we associate with a series of causally related values (states) over time
- Not a name, but can be named
  - I call my mom 'Mom', but you wouldn't
- Can be composite - the NY Yankees
- Programs that are processes need identity

State

- Value of an identity at a time
- Why not use variables for state?
  - Variable might not refer to a proper value
  - Sets of variables/fields never constitute a proper composite value
  - No state transition management
    - I.e., no time coordination model

Philosophy

- Things don't change in place
- Becomes obvious once you incorporate time as a dimension
  - Place includes time
- The future is a function of the past, and doesn’t change it
- Co-located entities can observe each other without cooperation
- Coordination is desirable in local context

Race-walker foul detector

- Get left foot position
  - off the ground
- Get right foot position
  - off the ground
- Must be a foul, right?

- Snapshots are critical to perception and decision making
- Can't stop the runner/race (locking)
- Not a problem if we can get runner's value
- Similarly don't want to stop sales in order to calculate bonuses or sales report

Coming to Terms with State

Value

- An *immutable* magnitude, quantity, number... or immutable composite thereof

Identity

- A putative entity we associate with a series of causally related values (states) over time

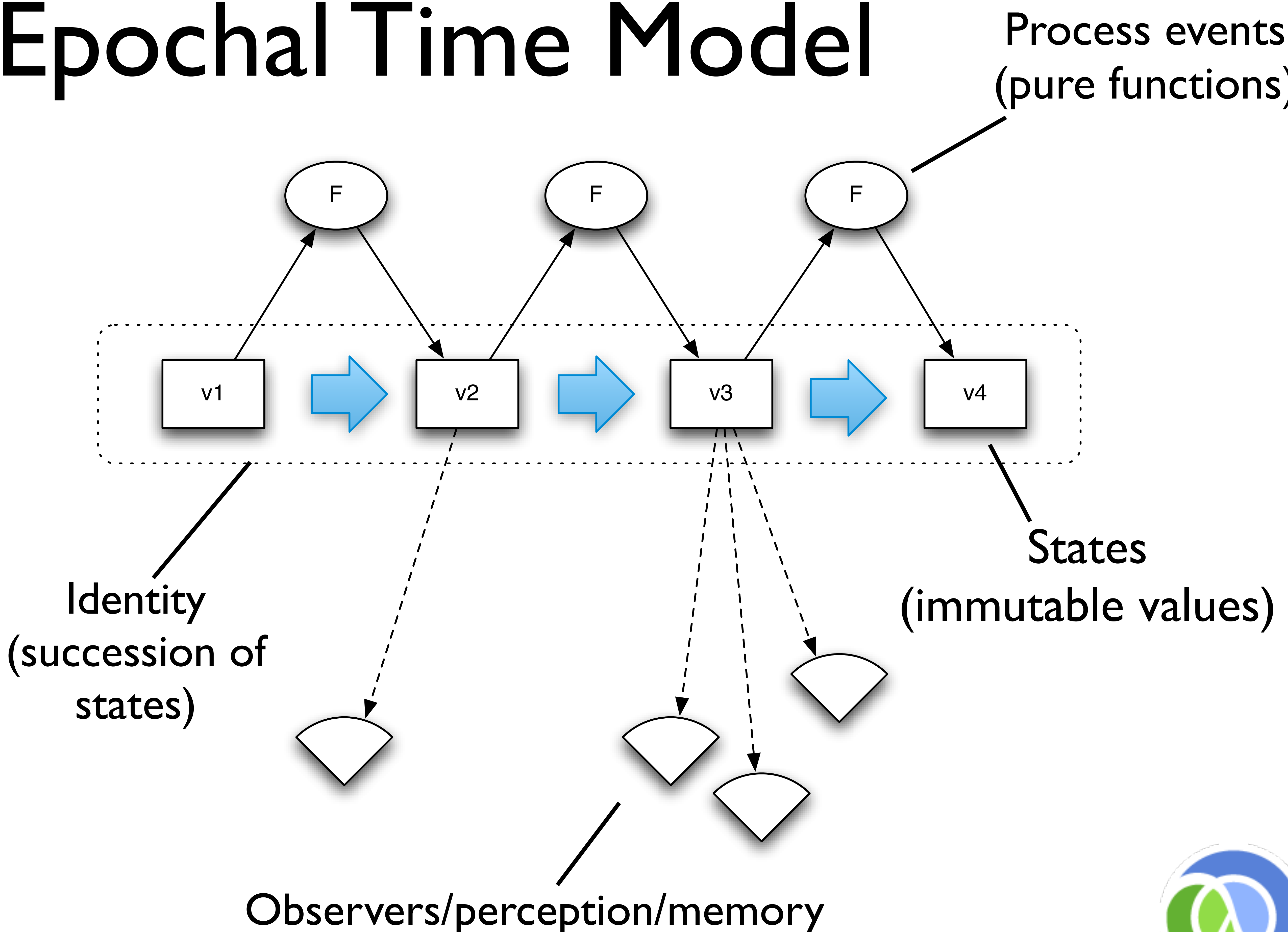
State

- Value of an identity at a moment in time

Time

- Relative before/after ordering of causal values

Epochal Time Model



Concurrency Approach

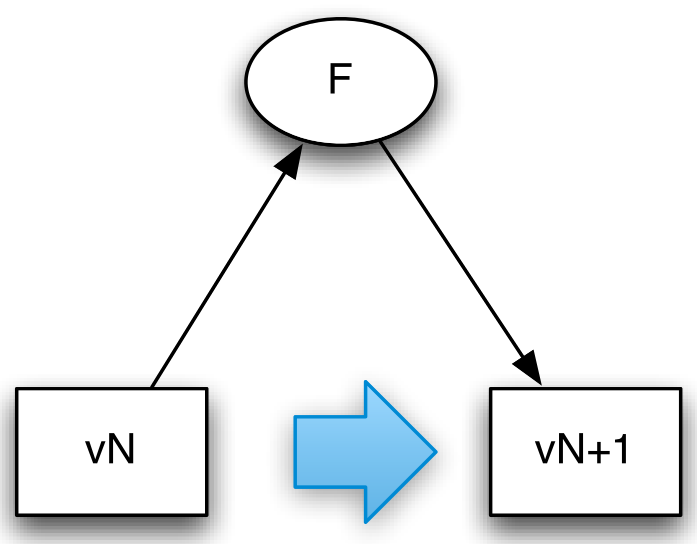
- Programming with values is critical
  - Persistent data structures
- Just need to manage the succession of values (states) of an identity
  - A timeline coordination problem
  - Several semantics possible
- Managed references
  - Variable-like boxes with time coordination semantics

Threads

- All constructs work from any threads - no need to start thread via Clojure
- Can get threads from Clojure with Agents and future
- Easiest async:
  - (future some-expression)
    - will run in thread pool thread
    - returns reference to result, will cache
    - @/deref will block until done

Persistent’s Performance

- Persistent data structures *are* slower in sequential use (especially ‘writing’)
- But - no one can see what happens inside F



- I.e. the ‘birthing process’ of the next value can use our old (and new) performance tricks:
  - Mutation and parallelism
- Parallel map on persistent vector same speed as loop on j.u.ArrayList on quad-core
- Safe ‘transient’ versions of PDS possible, with O(1) conversions between persistent/transient

Time constructs

- Need to ensure atomic state succession
- Need to provide point-in-time value perception
- Multiple timelines possible (and desirable)

- Many implementation strategies with different characteristics/semantics
  - CAS - uncoordinated 1:1
  - Agents - uncoordinated, async. (Like actors, but local and observable)
  - STM - coordinated, arbitrary regions

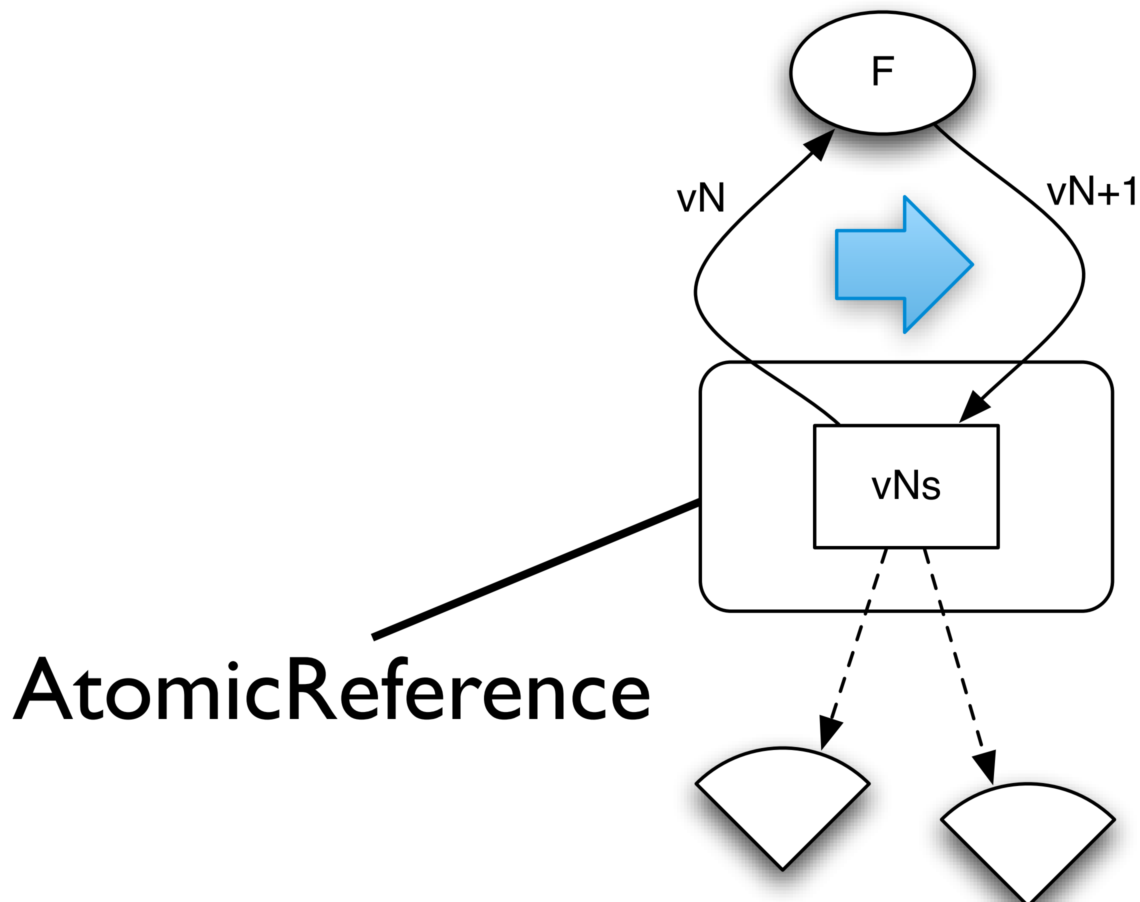
Uniform state transition model

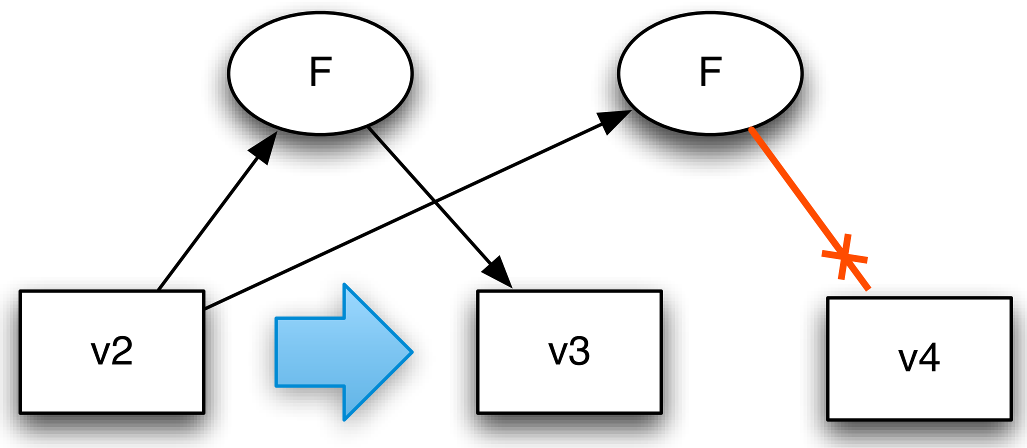
- ('change-state' reference function [args*])
- function will be passed current state of the reference (plus any args)
- Return value of function will be the next state of the reference
- Snapshot of 'current' state always available with deref
- No user locking, no deadlocks

unified update model (lab)

Modeling State and Time Part 2

CAS as Time Construct





`(swap! an-atom f args)`

`(f vN args)` *becomes* vN+1

- can automate spin

- 1:1 timeline/identity
- Atomic state succession
- Point-in-time value perception

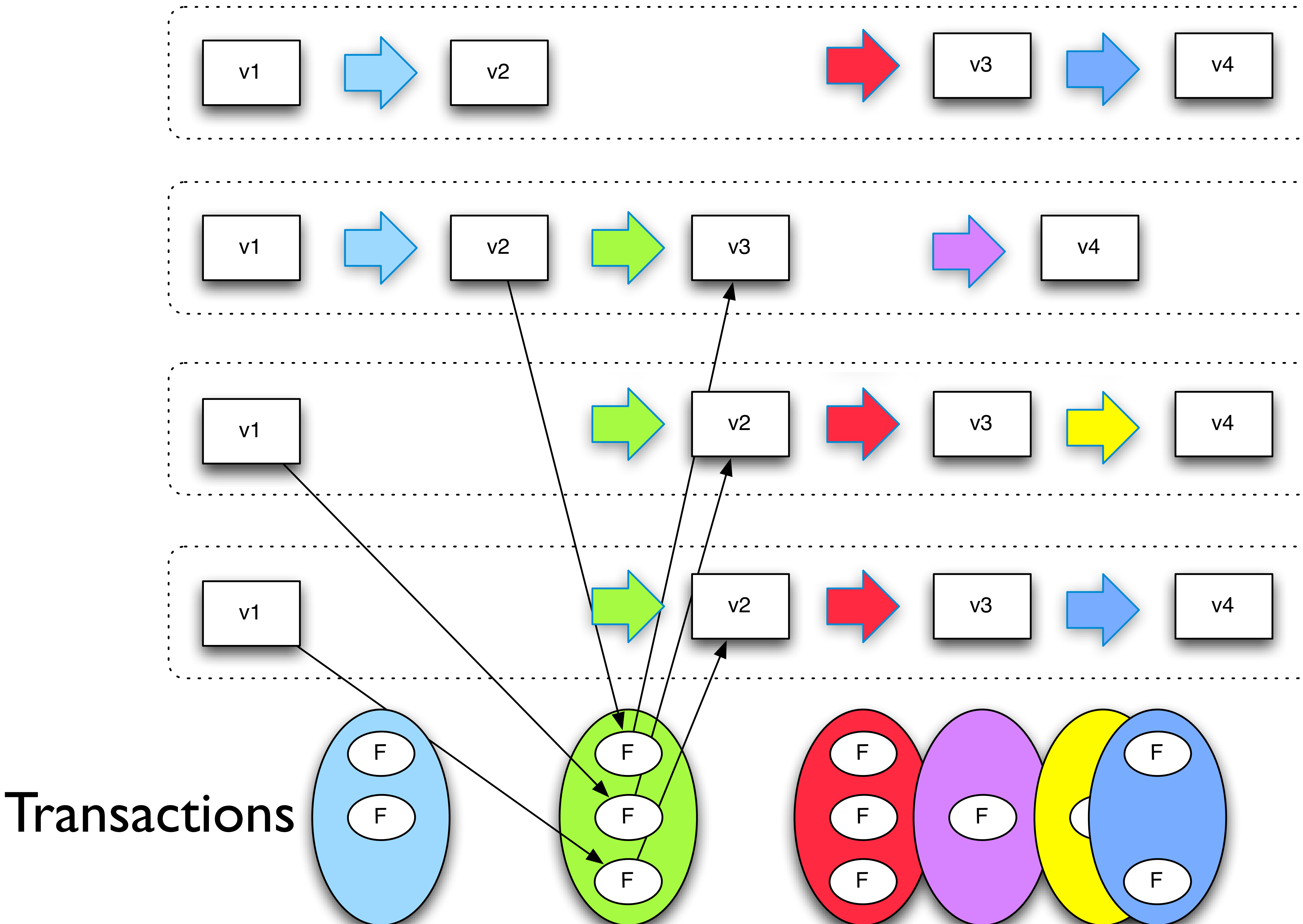
Atoms

- Manage independent state
- State changes through *swap!*, using ordinary function (state=>new-state)
- Change occurs *synchronously* on caller thread
- Models compare-and-set (CAS) spin swap
- Function may be called more than once!
    - Guaranteed atomic transition
    - Must avoid side-effects!

Atoms in Action

```
(def foo (atom {:a "fred" :b "ethel" :c 42 :d 17 :e 6}))

@foo -> {:d 17, :a "fred", :b "ethel", :c 42, :e 6}

(swap! foo assoc :a "lucy")

@foo -> {:d 17, :a "lucy", :b "ethel", :c 42, :e 6}
```

STM as Time Construct



Refs and Transactions

- Software transactional memory system (STM)
- Refs can only be changed within a transaction
- All changes are Atomic and Isolated
  - Every change to Refs made within a transaction occurs or none do
  - No transaction sees the effects of any other transaction while it is running
- Transactions are speculative
  - Will be retried automatically if conflict
  - Must avoid side-effects!

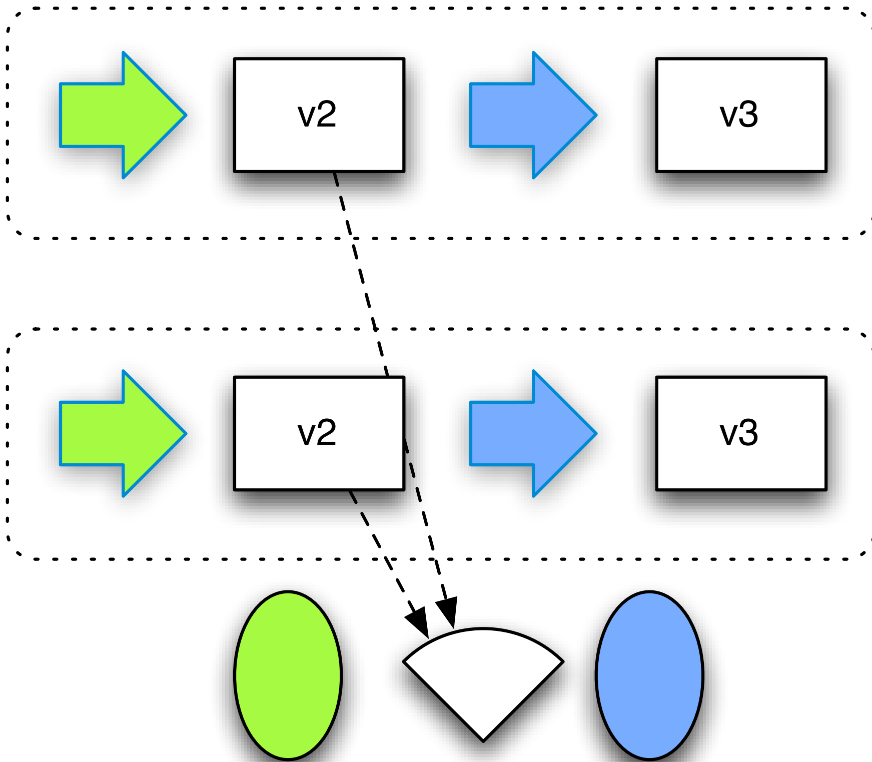
The Clojure STM

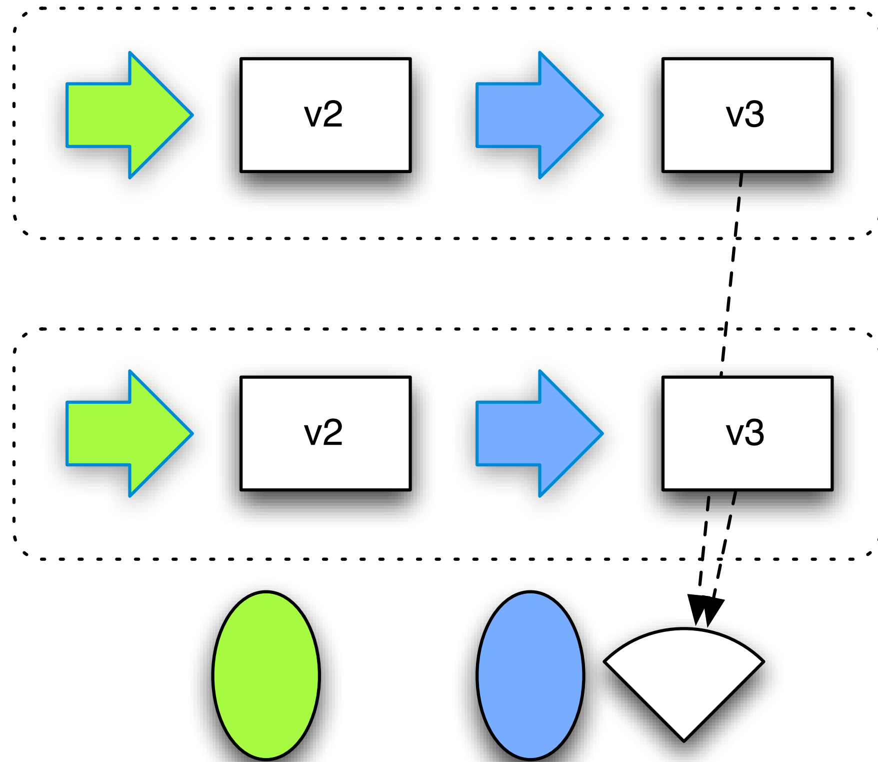
- Surround code with (dosync ...), state changes through *alter/commute*, using ordinary function (state=>new-state)
- Uses Multiversion Concurrency Control (MVCC)
- All reads of Refs will see a consistent snapshot of the 'Ref world' as of the starting point of the transaction, + any changes it has made.
- All changes made to Refs during a transaction will appear to occur at a single point in the timeline.

Multiversion Concurrency Control

- No interference with processes
  - By keeping some history
  - Persistent data structures make history cheap
- Allows observers/readers to have timeline
- Composite snapshots are like visual glimpses, from a point-in-time in the transaction universe
- Free reads are like visual scans that span time

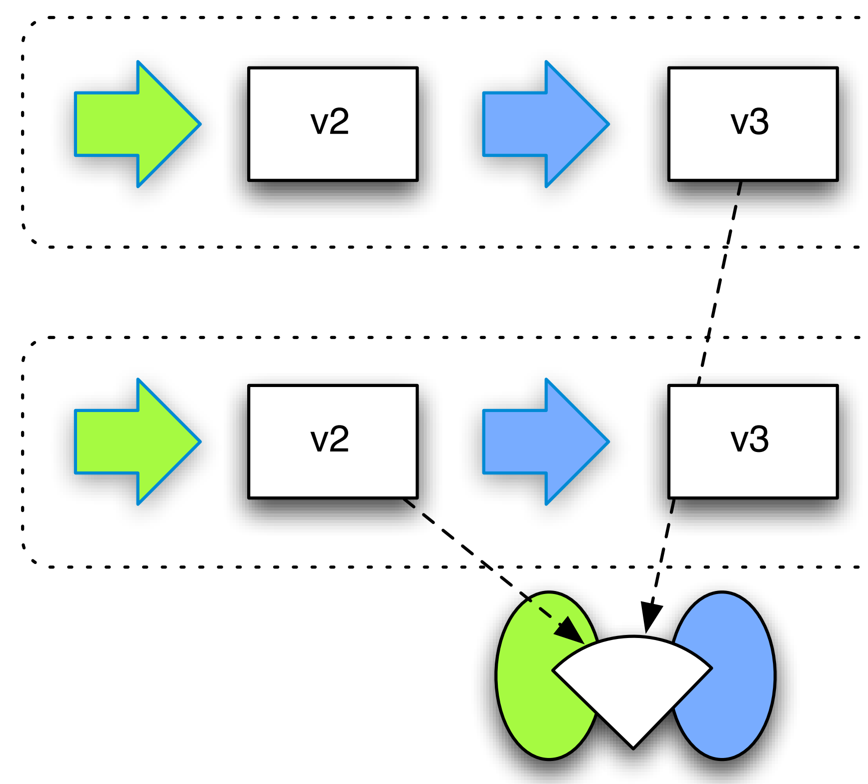
Perception in MVCC STM





Transactional snapshots

Non-transactional scans



Refs in action

```
(def foo (ref {:a "fred" :b "ethel" :c 42 :d 17 :e 6}))

@foo -> {:d 17, :a "fred", :b "ethel", :c 42, :e 6}

(assoc @foo :a "lucy")
-> {:d 17, :a "lucy", :b "ethel", :c 42, :e 6}

@foo -> {:d 17, :a "fred", :b "ethel", :c 42, :e 6}

(alter foo  assoc :a "lucy")
-> IllegalStateException: No transaction running

(dosync (alter foo  assoc :a "lucy"))
@foo -> {:d 17, :a "lucy", :b "ethel", :c 42, :e 6}
```

Implementation - STM

- Not a lock-free spinning optimistic design
- Uses locks, latches to avoid churn
- Deadlock detection + barging
- One timestamp CAS is only global resource
- No read tracking
- Coarse-grained orientation
  - Refs + persistent data structures
- Readers don't impede writers/readers, writers don't impede readers, supports commute

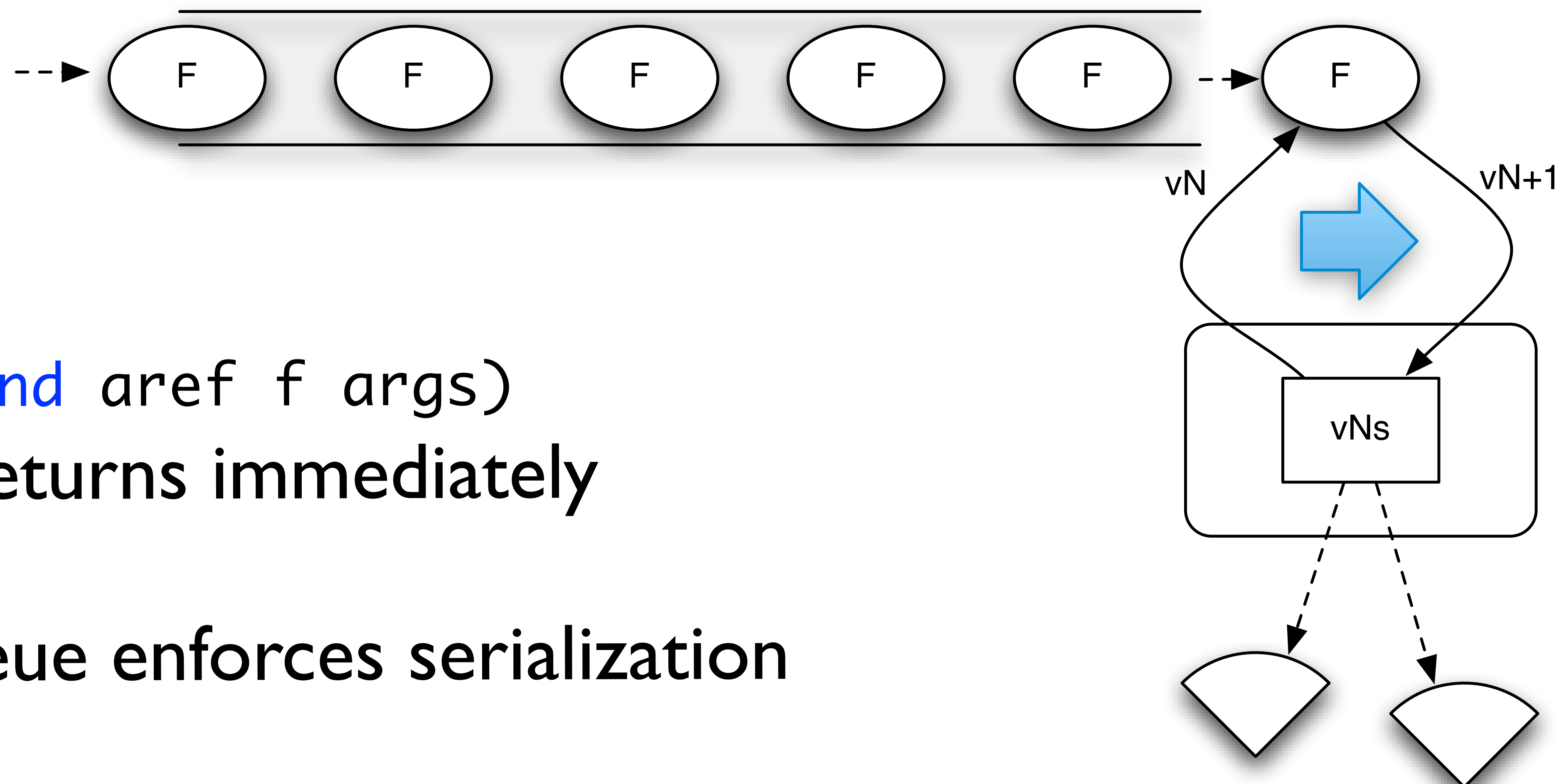
STM - commute

- Often a transaction will need to update a jobs-done counter or add its result to a map
- If done with `alter`, update is a read-modify-write, so if multiple transactions contend, one wins, one retries
- If transactions don't care about resulting value, and operation is commutative, can instead use `commute`
- Both transactions will succeed without retry
- Always just an optimization

STM - ensure

- MVCC is subject to *write-skew*
  - Where validity of transaction depends on stability of value unchanged by it
  - e.g. one of two accounts can go negative but not both
- Simply reading does not preclude modification by another transaction
- Can use ensure for values that are read but must remain stable
- More efficient than dummy write

Agents as Time Construct



`(send aref f args)`
returns immediately

queue enforces serialization

`(f vN args)` becomes vN+1

happens asynchronously in thread pool thread

- 1:1 timeline/identity
- Atomic state succession
- Point-in-time value perception

Agents

- Manage independent state
- State changes through actions, which are ordinary functions (state=>new-state)
- Actions are dispatched using *send* or *send-off*, which return immediately
- Actions occur *asynchronously* on thread-pool threads
- Only one action per agent happens at a time

Agents

- Agent state always accessible, via deref/@, but may not reflect all actions
- Any dispatches made during an action are held until *after* the state of the agent has changed
- Agents coordinate with transactions - any dispatches made during a transaction are held until it commits
- Agents are not Actors (Erlang/Scala)

Agents in Action

```
(def foo (agent {:a "fred" :b "ethel" :c 42 :d 17 :e 6}))

@foo -> {:d 17, :a "fred", :b "ethel", :c 42, :e 6}

(send foo assoc :a "lucy")

@foo -> {:d 17, :a "fred", :b "ethel", :c 42, :e 6}

... time passes ...

@foo -> {:d 17, :a "lucy", :b "ethel", :c 42, :e 6}
```

Uniform state transition

```
;refs
(dosync
  (alter foo  assoc :a "lucy"))

;agents
(send foo assoc :a "lucy")

;atoms
(swap! foo assoc :a "lucy")
```

Experiences - Concurrency Model

- Programming with values is more robust
- Approachable - like variables, with semantics
- Uniform state transition model works
  - Can easily move from atoms to agents or STM
- Leverages locality
  - High performance
  - Direct reads

Summary

- Immutable values, a feature of the functional parts of our programs, are a critical component of the parts that deal with time
- Persistent data structures provide efficient immutable composite values
- Once you accept immutability, you can separate time management, and swap in various concurrency semantics
- Managed references provide easy to use and understand time coordination

zero sum
(lab)

java interop

clojure calling java

java new

java	`new Widget("foo")`
clojure	`(new Widget "foo")`
clojure sugar	`(Widget. "red")`

access static members

java	`Math.PI`
clojure	`(. Math PI)`
clojure sugar	`Math/PI`

access instance members

java	`rnd.nextInt()`
clojure	`(. rnd nextInt)`
clojure sugar	`(.nextInt rnd)`

chaining access

java	`person.getAddress().getZipCode()`
clojure	`(. (. person getAddress) getZipCode)`
clojure sugar	`(.. person getAddress getZipCode)`

atomic data types

type	example	java equivalent
string	`"foo"`	String
character	`\f`	Character
regex	`#"fo*"`	Pattern
integer	`42`	long
a.p. integer	`42N`	BigInteger
double	`3.14159`	double
a.p. double	`3.14159M`	BigDecimal
boolean	`true`	Boolean
nil	`nil`	`null`
symbol	`foo, +`	N/A
keyword	`:foo, ::foo`	N/A

doto

```
(import '[javax.swing JFrame JPanel])
-> javax.swing.JPanel

(doto
 (JFrame. "Foobar")
 (.add (proxy [JPanel] []))
 (.setSize 640 400)
 (.setVisible true))
-> #<JFrame javax.swing.JFrame>
```

implicit arg for
all subsequent forms

type info

```
(instance? Comparable "foobar")
-> true

(class "foobar")
-> java.lang.String

(bases java.io.BufferedReader)
-> (java.io.Reader)

(supers java.io.BufferedReader)
-> #{java.io.Closeable java.io.Reader
java.lang.Object java.lang.Readable}
```

immediate bases

all supers

arrays

```
(def nums (make-array Integer/TYPE 10))
-> #'user/nums

(aset nums 4 1000)
-> 1000

(aget nums 4)
-> 1000

(seq nums)
-> (0 0 0 0 1000 0 0 0 0 0)
```

seq -> array

```
(def nums (range 3))
-> #'user/nums

(class (to-array nums))
-> [Ljava.lang.Object;

(class (into-array nums))
-> [Ljava.lang.Integer;

(class (into-array Comparable nums))
-> [Ljava.lang.Comparable;
```

Object

inferred from first

explicit

unboxed math (1.2)

```
(time
 (loop [i 0]
   (if (< i million)
     (recur (inc i))
     i)))
"Elapsed time: 42.473 msecs"

(time
 (let
  [million (int million)]
  (loop [i (int 0)]
    (if (< i million)
      (recur (inc i))
      i))))
"Elapsed time: 3.468 msecs"
```

let idiom:
same name, different type

coercion ops
for primitives/arrays

unboxed math (1.3+)

```
(defn fib [n]
  (if (<= n 1)
    1
    (+ (fib (dec n)) (fib (- n 2)))))

(time (fib 38))
"Elapsed time: 3565.579 msecs"

;; hint arg and return
(defn fib ^long [^long n]
  (if (<= n 1)
    1
    (+ (fib (dec n)) (fib (- n 2)))))

(time (fib 38))
"Elapsed time: 395.365 msecs"
```

hints flow,
nothing needed
after method sig

java calling clojure

implement interface

base class, interfaces

base class cons args

```
(proxy [java.util.Comparator] []
  (compare [o1 o2]
    (- (count o1) (count o2))))
```

method bodies

prefer reify

interface

```
(reify java.util.Comparator
       (compare [_ o1 o2]
                (- (count o1) (count o2))))
```

method bodies

add more interfaces here

don't need a method?
skip it!

```
(reify FloorWax DesertTopping)
```

implements two interfaces, but
all methods will throw
AbstractMethodError

add
type hints
to improve
performance

type metadata example

```
(defn capitalize
  "Upcase the first character of a string,
   lowercase the rest."
  [s]
  (if (.isEmpty s)
    s
    (let [up (.. s
                 (substring 0 1)
                 (toUpperCase))
          down (.. s
                   (substring 1)
                   (toLowerCase))]
      (.concat up down))))
```

warn-on-reflection

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

(require :reload 'demo.capitalize)
Reflection warning, demo/capitalize.clj:6 -
  reference to field isEmpty can't be resolved.
Reflection warning, demo/capitalize.clj:8 -
  call to substring can't be resolved.
Reflection warning, demo/capitalize.clj:8 -
  call to toUpperCase can't be resolved.
Reflection warning, demo/capitalize.clj:11 -
  call to substring can't be resolved.
Reflection warning, demo/capitalize.clj:11 -
  call to toLowerCase can't be resolved.
Reflection warning, demo/capitalize.clj:14 -
  call to concat can't be resolved.
-> nil
```

add type metadata

```
(defn capitalize
  "Upcase the first character of a string,
   lowercase the rest."
  [^String s]
  (if (.isEmpty s)
    s
    (let [up (.. s
                 (substring 0 1)
                 (toUpperCase))
          down (.. s
                   (substring 1)
                   (toLowerCase))]
      (.concat up down))))
```

s is known to be a String

no more warnings

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

(require :reload 'demo.capitalize)
-> nil
```

more idiomatic

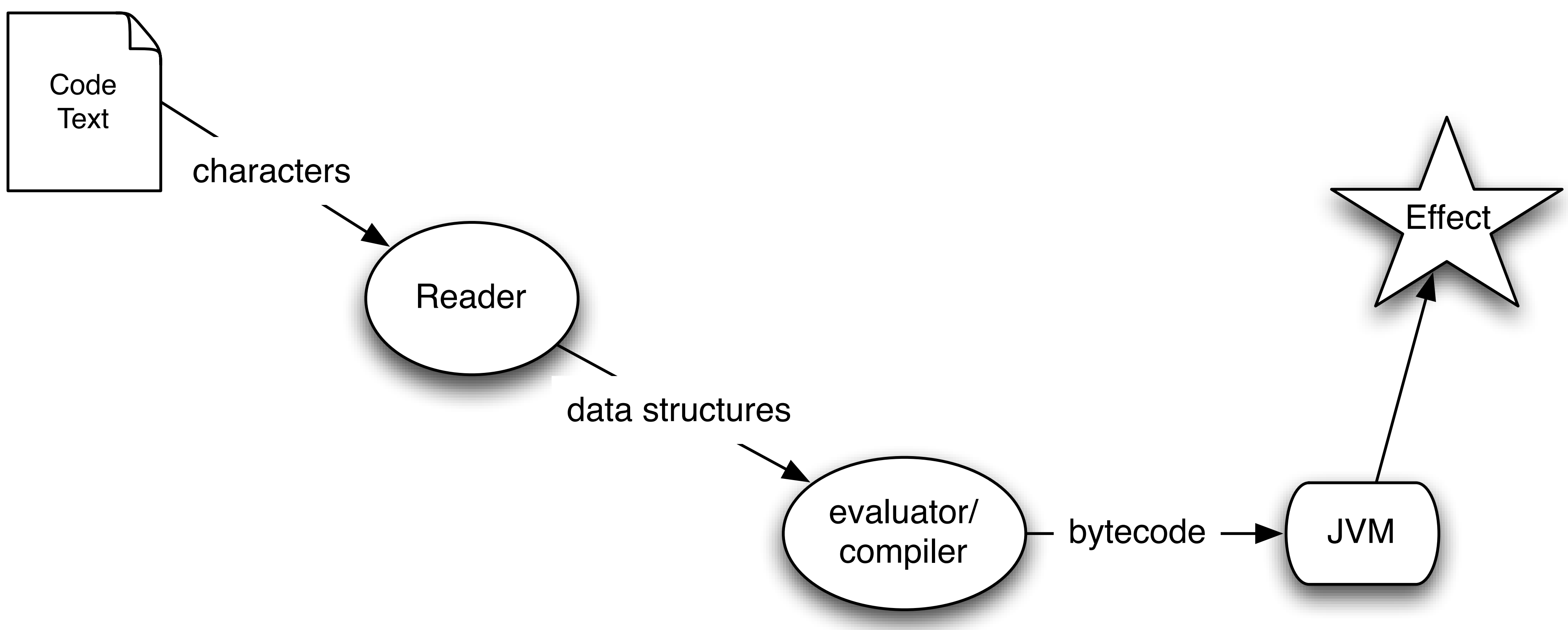
```clojure
(defn capitalize
  "Upcase the first character of a string,
   lowercase the rest."
  [^String s]
  (if (.isEmpty s)
    s
    (.concat
     (.toUpperCase (subs s 0 1))
     (.toLowerCase (subs s 1)))))
```

still non-reflective,
if subs is non-reflective

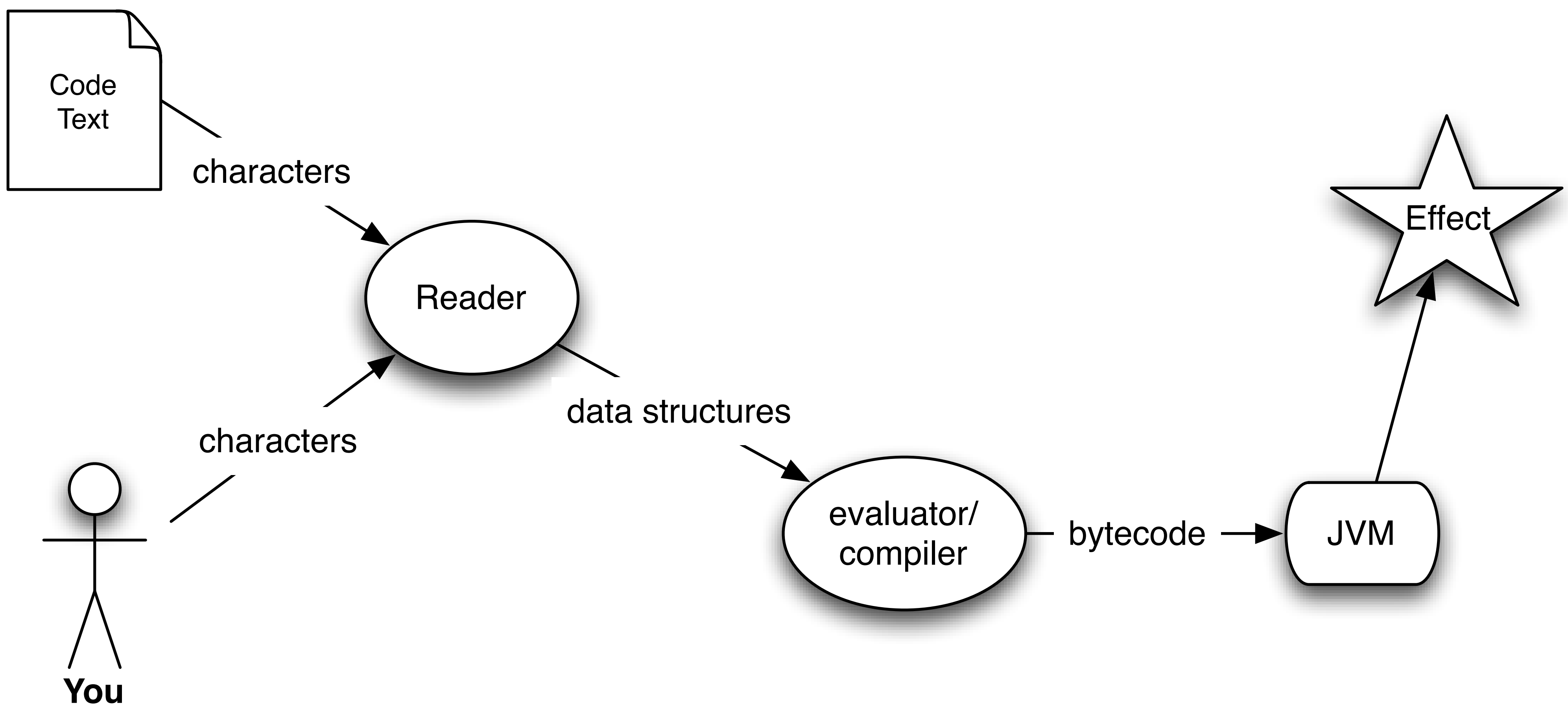
cellular automata (lab)

Macros and Evaluation

Clojure Evaluation



Interactivity



Evaluation

- Strings, numbers, characters, true, false, nil and keywords evaluate to themselves.
- If symbol has prefix, and prefix names namespace, refers to global var
    - else prefix must name Class, refers to static member
- else if symbol has package-qualified structure ([name.]+name), resolves to Class name
- Unqualified symbol refers to special form, else nearest enclosing lexical binding, else mapping of that name in namespace

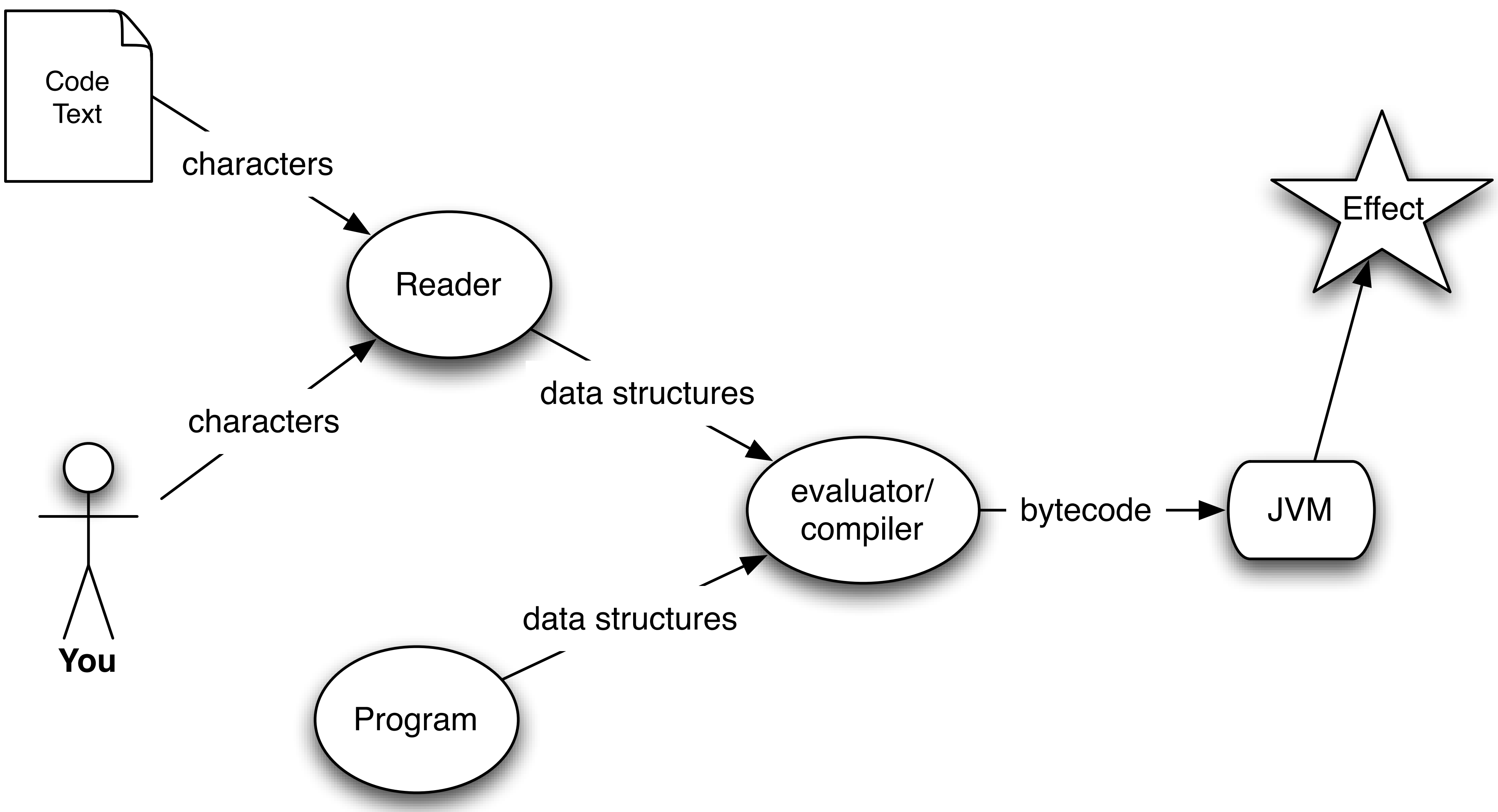
List Evaluation

- () evaluates to empty list - ()
- First item in non-empty list is ‘operator’
  - if operator is symbol naming special form, list is that special form
  - else if operator is symbol naming global var marked as macro, is macro ‘call’, unevaluated rest of list passed as args to fn in that var
  - .instanceMember, Class/staticMember, Classname. - special macroexpansions
- else evaluated, cast to IFn, and invoked

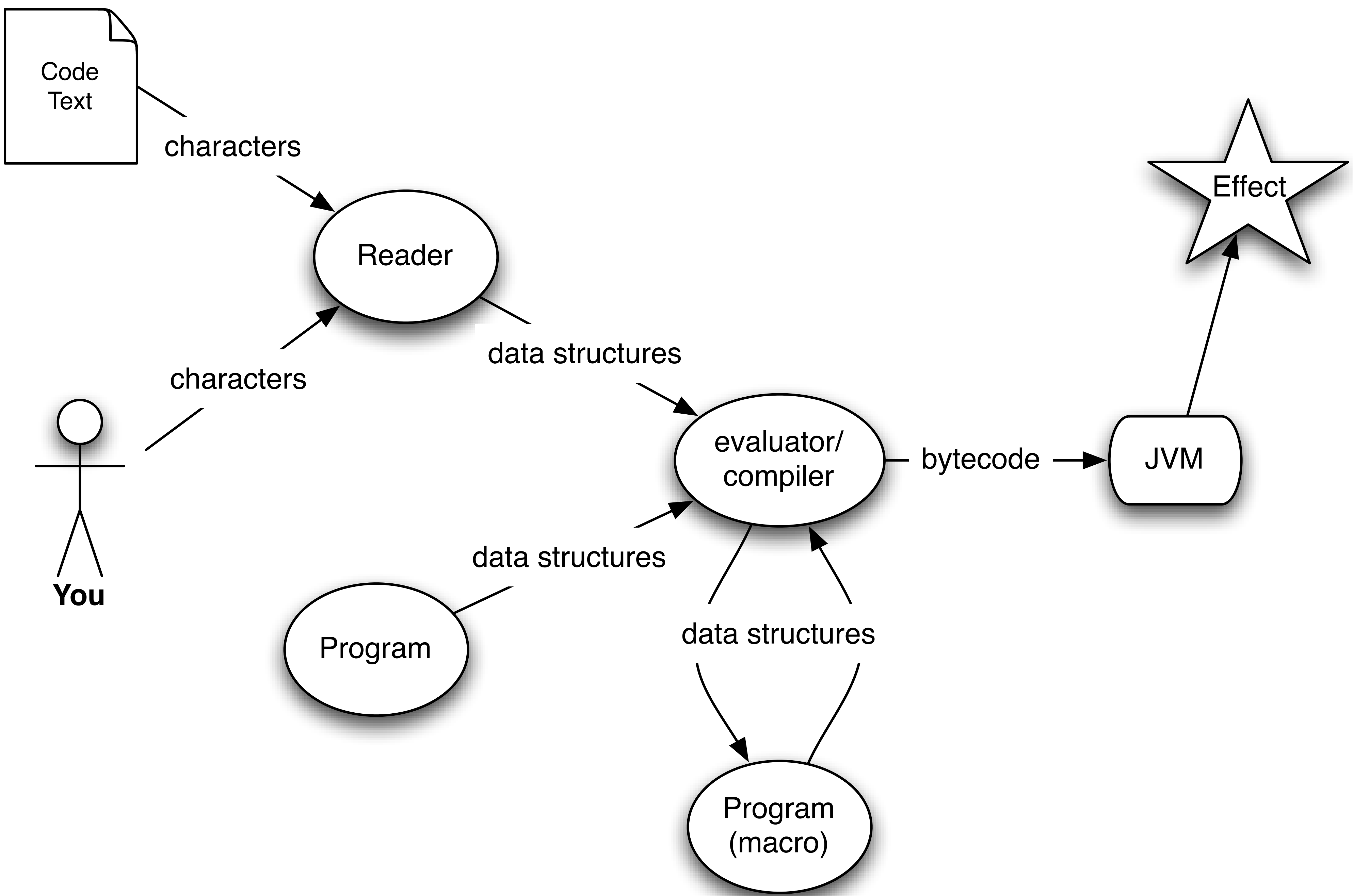
Data Structure Evaluation

- Vectors, Sets and Maps yield vectors and (hash) sets/maps whose contents are the evaluated values of the objects they contain.
- The same is true of metadata maps. If the vector, set or map has metadata, the evaluated metadata map will become the metadata of the resulting value.
  - ```
    (let [x 1 y 2]
      ^{:x x} [x y 3])
    => ^{:x 1} [1 2 3]
    ```
- Everything else evaluates to itself
    ```

Programs writing Programs



Syntactic Abstraction



What is a Macro?

- A small program with a single entry point that is passed some code as data and returns some other data for use as code
- Defined via defmacro, its name is specially flagged:
  - Called by the compiler *at compile-time* and passed contained code data structures
  - Compiler continues, using result of the macro call in its place

How to write a Macro

- Think about what form you want to write
- Think about what form you want it to become
- Write the macro as a function that takes the former and returns the latter
- Common mistake:
    - Thinking of a macro as an ordinary function with special runtime evaluation properties - it's not!

Macro Example: when

```clojure
;we want to write:
(when foo
  (dothis)
  (dothat))

;and have it become:
(if foo
  (do
    (dothis)
    (dothat)))

;from core.clj
(defmacro when
  [test & body]
  (list 'if test (cons 'do body)))

;use macroexpand to try it
(macroexpand-1 '(when foo (dothis) (dothat)))
=> (if foo (do (dothis) (dothat)))
```

syntax-quote (`)

- Manually constructing code forms using list and cons can be tedious and obscure
- syntax-quote allows us to write a template that looks like the expansion
- Does the quoting and list construction for us

```
(defmacro when
  [test & body]
  (list 'if test (cons 'do body)))

;same thing with syntax-quote
(defmacro when2
  [test & body]
  `(if ~test (do ~@body)))
```

Syntax-quote - `

- For all forms other than Symbols, Lists, Vectors, Sets and Maps, `` `x `` is the same as `'x`.
- For unqualified Symbols, syntax-quote *resolves* the symbol in the current context, yielding a qualified *symbol* (`namespace/name` or `fully.qualified.Classname`)
  - If symbol not currently mapped, resolves to `current-ns/name`
- Auto-gensyms - symbols ending in `#` become uniquely named, unqualified symbols
  - All uses within same ` level resolve to same symbol

syntax-quote Resolution

- If symbol has prefix (foo/bar), or has package-qualified structure ([name.]+name), resolves to itself
- Unqualified symbol resolves to special form if it names one
  - else mapping of that name in namespace if there is one
  - else current-ns/name

Syntax-quote (cont.)

- For Lists/Vectors/Sets/Maps, syntax-quote establishes a template of the corresponding data structure.
  - unqualified forms behave as if recursively syntax-quoted
  - exempt elements from recursive quoting by qualifying with unquote (~) or unquote-splicing (~@)
    - replaces with value or sequence of values respectively

Syntax-quote (examples)

```
;in fresh user namespace
'foo -> foo
`foo -> user/foo
(= 'foo `foo) -> false

(def x 5)
(def lst '(a b c))

`(list x ~x lst ~@lst 7 8 :nine ten# ten#)
=> (clojure.core/list user/x 5 user/lst a b c 7
    8 :nine ten__172__auto__ ten__172__auto__)

`#{x ~x y z}
=> #{5 user/z user/x user/y}

`{x ~@lst}
=> {b c, user/x a}
```

Macro Example: or

```
;we want to write:
(or a b c)

;and have it become:
(let [or__42 a]
  (if or__42
    or__42
    (or b c))) ;note ‘recursion’

;from core.clj
(defmacro or
  ([] nil)
  ([x] x)
  ([x & next]
    `(let [or# ~x]
       (if or# or# (or ~@next)))))

;why not just (if ~x ...) above?
```

Macros Encapsulate ‘Patterns’

```
(defmacro locking
  "Executes exprs in an implicit do, while holding the
monitor of x.
  Will release the monitor of x in all circumstances."
  [x & body]
  `(let [lockee# ~x]
     (try
      (monitor-enter lockee#)
      ~@body
      (finally
       (monitor-exit lockee#)))))
```

Preserving Form Metadata

```
(defmacro ->
  "Threads the expr through the forms. Inserts x as the
second item in the first form, making a list of it if it is
not a list already. If there are more forms, inserts the
first form as the second item in second form, etc."
  ([x] x)
  ([x form]
    (if (seq? form)
        (with-meta `(~(first form) ~x ~@(next form))
                   (meta form))
        (list form x)))
  ([x form & more] `(-> (-> ~x ~form) ~@more)))
```

Macros are Powerful

- Use them only to do things functions can't do
  - Transform, generate or rearrange code
  - Expand into code which would evaluate less than a function call would
    - e.g. or can't be a function
- Macros should be pure functions of code to code
  - Remember they run at compile time on code data structures

defstrict (lab)

OO:
Records, Types, Protocols, & Multimethods

typical OO

typical OO

encapsulation

typical OO

encapsulation

polymorphism

typical OO

encapsulation

polymorphism

inheritance

typical OO

encapsulation

polymorphism

inheritance

objects are mutable

typical OO

encapsulation

polymorphism

inheritance

objects are mutable

polymorphism baked into objects

typical OO

encapsulation

polymorphism

inheritance

objects are mutable

polymorphism baked into objects

interfaces optional

typical OO

encapsulation

polymorphism

inheritance

objects are mutable

polymorphism baked into objects

interfaces optional

implementation inheritance

oo step 1:
hide every new kind of data in a private mini-language
??!!

Clojure's answer

Clojure’s answer

public fields

Clojure's answer

public fields

polymorphism through protocols

Clojure's answer

public fields

polymorphism through protocols

objects are immutable

Clojure's answer

public fields

polymorphism through protocols

objects are immutable

polymorphism separate from objects

Clojure's answer

public fields

polymorphism through protocols

objects are immutable

polymorphism separate from objects

interfaces are mandatory

Clojure's answer

public fields

polymorphism through protocols

objects are immutable

polymorphism separate from objects

interfaces are mandatory

no implementation inheritance

Clojure's answer

public fields

polymorphism through protocols

objects are immutable

polymorphism separate from objects

interfaces are mandatory

no implementation inheritance

it’s your data!

(it can be objects if you like, but it never stops being accessible)

defrecord

```
(defrecord Foo [a b c])
-> user.Foo
```

named type
with slots

defrecord

```
(defrecord Foo [a b c])
-> user.Foo

(def f (Foo. 1 2 3))
-> #'user/f
```

named type
with slots

positional
constructor

defrecord

```
(defrecord Foo [a b c])
-> user.Foo
```

named type with slots

```
(def f (Foo. 1 2 3))
-> #'user/f
```

positional constructor

```
(:b f)
-> 2
```

keyword access

defrecord

```
(defrecord Foo [a b c])
-> user.Foo
```

named type
with slots

```
(def f (Foo. 1 2 3))
-> #'user/f
```

positional
constructor

```
(:b f)
-> 2
```

keyword access

```
(class f)
-> user.Foo
```

plain ol' class

defrecord

```
(defrecord Foo [a b c])
-> user.Foo

(def f (Foo. 1 2 3))
-> #'user/f

(:b f)
-> 2

(class f)
-> user.Foo

(supers (class f))
-> #{clojure.lang.IObj clojure.lang.IKeywordLookup java.util.Map
 clojure.lang.IPersistentMap clojure.lang.IMeta java.lang.Object
 java.lang.Iterable clojure.lang.ILookup clojure.lang.Seqable
 clojure.lang.Counted clojure.lang.IPersistentCollection
 clojure.lang.Associative}
```

named type with slots

positional constructor

keyword access

plain ol' class

rasydht*

defrecord

```
(defrecord Foo [a b c])
-> user.Foo
```

named type
with slots

```
(def f (Foo. 1 2 3))
-> #'user/f
```

positional
constructor

```
(:b f)
-> 2
```

keyword access

rasydht*

```
(class f)
-> user.Foo
```

plain ol' class

```
(supers (class f))
-> #{clojure.lang.IObj clojure.lang.IKeywordLookup java.util.Map
 clojure.lang.IPersistentMap clojure.lang.IMeta java.lang.Object
 java.lang.Iterable clojure.lang.ILookup clojure.lang.Seqable
 clojure.lang.Counted clojure.lang.IPersistentCollection
 clojure.lang.Associative}
```

***Rich abstracts so you don't have to**

from maps...

```clojure
(def stu {:fname "Stu"
          :lname "Halloway"
          :address {:street "200 N Mangum"
                    :city "Durham"
                    :state "NC"
                    :zip 27701}})
```

data-oriented

from maps...

```
(def stu {:fname "Stu"
          :lname "Halloway"
          :address {:street "200 N Mangum"
                    :city "Durham"
                    :state "NC"
                    :zip 27701}})
```

data-oriented

```
(:lname stu)
=> "Halloway"
```

keyword access

from maps...

```clojure
(def stu {:fname "Stu"
          :lname "Halloway"
          :address {:street "200 N Mangum"
                    :city "Durham"
                    :state "NC"
                    :zip 27701}})
```

data-oriented

```clojure
(:lname stu)
=> "Halloway"
```

keyword access

```clojure
(-> stu :address :city)
=> "Durham"
```

nested access

from maps...

```
(def stu {:fname "Stu"
          :lname "Halloway"
          :address {:street "200 N Mangum"
                    :city "Durham"
                    :state "NC"
                    :zip 27701}})
```

data-oriented

```
(:lname stu)
=> "Halloway"
```

keyword access

```
(-> stu :address :city)
=> "Durham"
```

nested access

```
(assoc stu :fname "Stuart")
=> {:fname "Stuart", :lname "Halloway",
    :address ...}
```

update

from maps...

```clojure
(def stu {:fname "Stu"
          :lname "Halloway"
          :address {:street "200 N Mangum"
                    :city "Durham"
                    :state "NC"
                    :zip 27701}})
```

data-oriented

```clojure
(:lname stu)
=> "Halloway"
```

keyword access

```clojure
(-> stu :address :city)
=> "Durham"
```

nested access

```clojure
(assoc stu :fname "Stuart")
=> {:fname "Stuart", :lname "Halloway",
     :address ...}
```

update

nested update

```clojure
(update-in stu [:address :zip] inc)
=> {:address {:street "200 N Mangum",
              :zip 27702 ...} ...}
```

...to records!

```clojure
(defrecord Person [fname lname address])
(defrecord Address [street city state zip])
(def stu (Person. "Stu" "Halloway"
                  (Address. "200 N Mangum"
                            "Durham"
                            "NC"
                            27701)))

(:lname stu)
=> "Halloway"

(-> stu :address :city)
=> "Durham"

(assoc stu :fname "Stuart")
=> :user.Person{:fname "Stuart", :lname"Halloway",
                :address ...}

(update-in stu [:address :zip] inc)
=> :user.Person{:address {:street "200 N Mangum",
                          :zip 27702 ...} ...}
```

...to records!

object-oriented

```
(defrecord Person [fname lname address])
(defrecord Address [street city state zip])
(def stu (Person. "Stu" "Halloway"
                  (Address. "200 N Mangum"
                            "Durham"
                            "NC"
                            27701)))

(:lname stu)
=> "Halloway"

(-> stu :address :city)
=> "Durham"

(assoc stu :fname "Stuart")
=> :user.Person{:fname "Stuart", :lname"Halloway",
                :address ...}

(update-in stu [:address :zip] inc)
=> :user.Person{:address {:street "200 N Mangum",
                          :zip 27702 ...} ...}
```

...to records!

```
(defrecord Person [fname lname address])
(defrecord Address [street city state zip])
(def stu (Person. "Stu" "Halloway"
                  (Address. "200 N Mangum"
                            "Durham"
                            "NC"
                            27701)))
```

object-oriented

```
(:lname stu)
=> "Halloway"

(-> stu :address :city)
=> "Durham"

(assoc stu :fname "Stuart")
=> :user.Person{:fname "Stuart", :lname"Halloway",
                :address ...}

(update-in stu [:address :zip] inc)
=> :user.Person{:address {:street "200 N Mangum",
                          :zip 27702 ...} ...}
```

still data-oriented: everything works as before

...to records!

```
(defrecord Person [fname lname address])
(defrecord Address [street city state zip])
(def stu (Person. "Stu" "Halloway"
                  (Address. "200 N Mangum"
                            "Durham"
                            "NC"
                            27701)))
```

object-oriented

```
(:lname stu)
=> "Halloway"
```

still data-oriented: everything works as before

```
(-> stu :address :city)
=> "Durham"
```

type is there when you care

```
(assoc stu :fname "Stuart")
=> :user.Person{:fname "Stuart", :lname"Halloway",
                :address ...}

(update-in stu [:address :zip] inc)
=> :user.Person{:address {:street "200 N Mangum",
                          :zip 27702 ...} ...}
```

protocols

```clojure
(defprotocol AProtocol
  "A doc string for AProtocol abstraction"
  (bar [a b] "bar docs")
  (baz [a] "baz docs"))
```

protocols

```clojure
(defprotocol AProtocol
  "A doc string for AProtocol abstraction"
  (bar [a b] "bar docs")
  (baz [a] "baz docs"))
```

named set of generic functions

protocols

```
(defprotocol AProtocol
  "A doc string for AProtocol abstraction"
  (bar [a b] "bar docs")
  (baz [a] "baz docs"))
```

named set of generic functions

polymorphic on type of first argument

protocols

```
(defprotocol AProtocol
  "A doc string for AProtocol abstraction"
  (bar [a b] "bar docs")
  (baz [a] "baz docs"))
```

named set of generic functions

polymorphic on type of first argument

no implementation

protocols

```
(defprotocol AProtocol
  "A doc string for AProtocol abstraction"
  (bar [a b] "bar docs")
  (baz [a] "baz docs"))
```

named set of generic functions

polymorphic on type of first argument

no implementation

define fns in same namespace as protocol

protocols

```
(defprotocol AProtocol
  "A doc string for AProtocol abstraction"
  (bar [a b] "bar docs")
  (baz [a] "baz docs"))
```

named set of generic functions

polymorphic on type of first argument

no implementation

define fns in same namespace as protocol

implement protocols in-line

```
(deftype Bar [a b c]
  AProtocol
  (bar [this b] "Bar bar")
  (baz [this] (str "Bar baz " c)))

(def b (Bar. 5 6 7))

(baz b)

=> "Bar baz 7"
```

extending a protocol

```
(baz "a")

java.lang.IllegalArgumentException:
No implementation of method: :baz
of protocol: #'user/AProtocol
found for class: java.lang.String

(extend-type String
  AProtocol
  (bar [s s2] (str s s2))
  (baz [s] (str "baz " s)))

(baz "a")

=> "baz a"
```

extension options

extension options

extend to classes/interfaces: extend-type

extension options

extend to classes/interfaces: extend-type

extend to nil

extension options

extend to classes/interfaces: extend-type

extend to nil

extend multiple protocols: extend-type

extension options

extend to classes/interfaces: extend-type

extend to nil

extend multiple protocols: extend-type

extend to multiple types: extend-protocol

extension options

extend to classes/interfaces: extend-type

extend to nil

extend multiple protocols: extend-type

extend to multiple types: extend-protocol

at bottom, arbitrary fn maps: extend

extension options

extend to classes/interfaces: extend-type

extend to nil

extend multiple protocols: extend-type

extend to multiple types: extend-protocol

at bottom, arbitrary fn maps: extend

reify

reify

code-gen happens in fn (lambda)

reify

code-gen happens in fn (lambda)

are objects closures, or vice versa?

reify

code-gen happens in fn (lambda)

are objects closures, or vice versa?

objects more primitive

reify

code-gen happens in fn (lambda)

are objects closures, or vice versa?

objects more primitive

make fn code-gen available to objects, too!

reify

code-gen happens in fn (lambda)

are objects closures, or vice versa?

objects more primitive

make fn code-gen available to objects, too!

reify

```
(let [x 42
      r (reify AProtocol
          (bar [this b] "reify bar")
          (baz [this ] (str "reify baz " x)))]
  (baz r))

=> "reify baz 42"
```

reify

instantiate an
unnamed type

```
(let [x 42
      r (reify AProtocol
          (bar [this b] "reify bar")
          (baz [this ] (str "reify baz " x)))]
  (baz r))

=> "reify baz 42"
```

reify

instantiate an unnamed type

implement 0 or more protocols or interfaces

```
(let [x 42
      r (reify AProtocol
          (bar [this b] "reify bar")
          (baz [this ] (str "reify baz " x)))]
  (baz r))

=> "reify baz 42"
```

reify

instantiate an unamed type

implement 0 or more protocols or interfaces

```clojure
(let [x 42
      r (reify AProtocol
          (bar [this b] "reify bar")
          (baz [this ] (str "reify baz " x)))]
  (baz r))

=> "reify baz 42"
```

closes over environment like fn

defrecord:
because data
belongs in maps

programming constructs are not like domain data

use **defrecord** for domain information

use **defrecord** for domain information

use **deftype** for programming constructs

deftype

```
(deftype Bar [a b c])
-> user.Bar
```

still a named
type with slots

deftype

```
(deftype Bar [a b c])
-> user.Bar
```

still a named type with slots

```
(def o (Bar. 1 2 3))
-> #'user/o
```

constructor, check

deftype

```
(deftype Bar [a b c])
-> user.Bar
```

still a named type with slots

```
(def o (Bar. 1 2 3))
-> #'user/o
```

constructor, check

```
(.b o)
-> 2
```

direct field access only

deftype

```
(deftype Bar [a b c])
-> user.Bar
```

still a named type with slots

```
(def o (Bar. 1 2 3))
-> #'user/o
```

constructor, check

```
(.b o)
-> 2
```

direct field access only

```
(class o)
-> user.Bar
```

still a plain ol' class

deftype

```
(deftype Bar [a b c])
-> user.Bar
```

still a named type with slots

```
(def o (Bar. 1 2 3))
-> #'user/o
```

constructor, check

```
(.b o)
-> 2
```

direct field access only

```
(class o)
-> user.Bar
```

still a plain ol' class

```
(supers (class o))
-> #{java.lang.Object}
```

yoyo*

deftype

```
(deftype Bar [a b c])
-> user.Bar
```

still a named type with slots

```
(def o (Bar. 1 2 3))
-> #'user/o
```

constructor, check

```
(.b o)
-> 2
```

direct field access only

```
(class o)
-> user.Bar
```

still a plain ol' class

```
(supers (class o))
-> #{java.lang.Object}
```

yoyo*

***you're on your own**

the other constructor

```
(def f (Foo. 1 2 3 {:meta 1} {:extra 4}))
-> #'user/f
```

metadata

```
(meta f)
-> {:meta 1}
```

extra k/v pairs

```
(into {} f)
-> {:a 1, :b 2, :c 3, :extra 4}
```

details

details

type fields can be primitives

details

type fields can be primitives

value-based equality and hash

details

type fields can be primitives

value-based equality and hash

in-line methods defs can inline

details

type fields can be primitives

value-based equality and hash

in-line methods defs can inline

keyword field lookups can inline

details

type fields can be primitives

value-based equality and hash

in-line methods defs can inline

keyword field lookups can inline

protocols make interfaces (interop only)

details

type fields can be primitives

value-based equality and hash

in-line methods defs can inline

keyword field lookups can inline

protocols make interfaces (interop only)

add java annotations (interop only)

details

type fields can be primitives

value-based equality and hash

in-line methods defs can inline

keyword field lookups can inline

protocols make interfaces (interop only)

add java annotations (interop only)

deftype fields can be mutable (experts only)

details

type fields can be primitives

value-based equality and hash

in-line methods defs can inline

keyword field lookups can inline

protocols make interfaces (interop only)

add java annotations (interop only)

deftype fields can be mutable (experts only)

example: rock/paper/scissors

http://rubyquiz.com/quiz16.html

a player

```
(defprotocol Player
  (choose [p])
  (update-strategy [p me you]))
```

a player

pick :rock, :paper,
or :scissors

```
(defprotocol Player
  (choose [p])
  (update-strategy [p me you]))
```

a player

pick :rock, :paper,
or :scissors

```
(defprotocol Player
  (choose [p])
  (update-strategy [p me you]))
```

return an updated
Player based on what
you and I did

stubborn player

```
(defrecord Stubborn [choice]
  Player
  (choose [_] choice)
  (update-strategy [this _ _] this))
```

stubborn player

initialize with choice

```
(defrecord Stubborn [choice]
  Player
  (choose [_] choice)
  (update-strategy [this _ _] this))
```

stubborn player

initialize with choice

```
(defrecord Stubborn [choice]
  Player
  (choose [_] choice)
  (update-strategy [this _ _] this))
```

play the choice

stubborn player

initialize with choice

```
(defrecord Stubborn [choice]
  Player
  (choose [_] choice)
  (update-strategy [this _ _] this))
```

play the choice

never change

mean player

```clojure
(defrecord Mean [last-winner]
  Player
  (choose [_]
   (if last-winner
     last-winner
     (random-choice)))
  (update-strategy [_ me you]
    (Mean. (when (iwon? me you) me))))
```

mean player

last thing that
worked for me

```
(defrecord Mean [last-winner]
  Player
  (choose [_]
   (if last-winner
     last-winner
     (random-choice)))
  (update-strategy [_ me you]
    (Mean. (when (iwon? me you) me))))
```

mean player

last thing that
worked for me

```
(defrecord Mean [last-winner]
  Player
  (choose [_]
   (if last-winner
     last-winner
     (random-choice)))
  (update-strategy [_ me you]
    (Mean. (when (iwon? me you) me))))
```

play last winner
or random

mean player

last thing that
worked for me

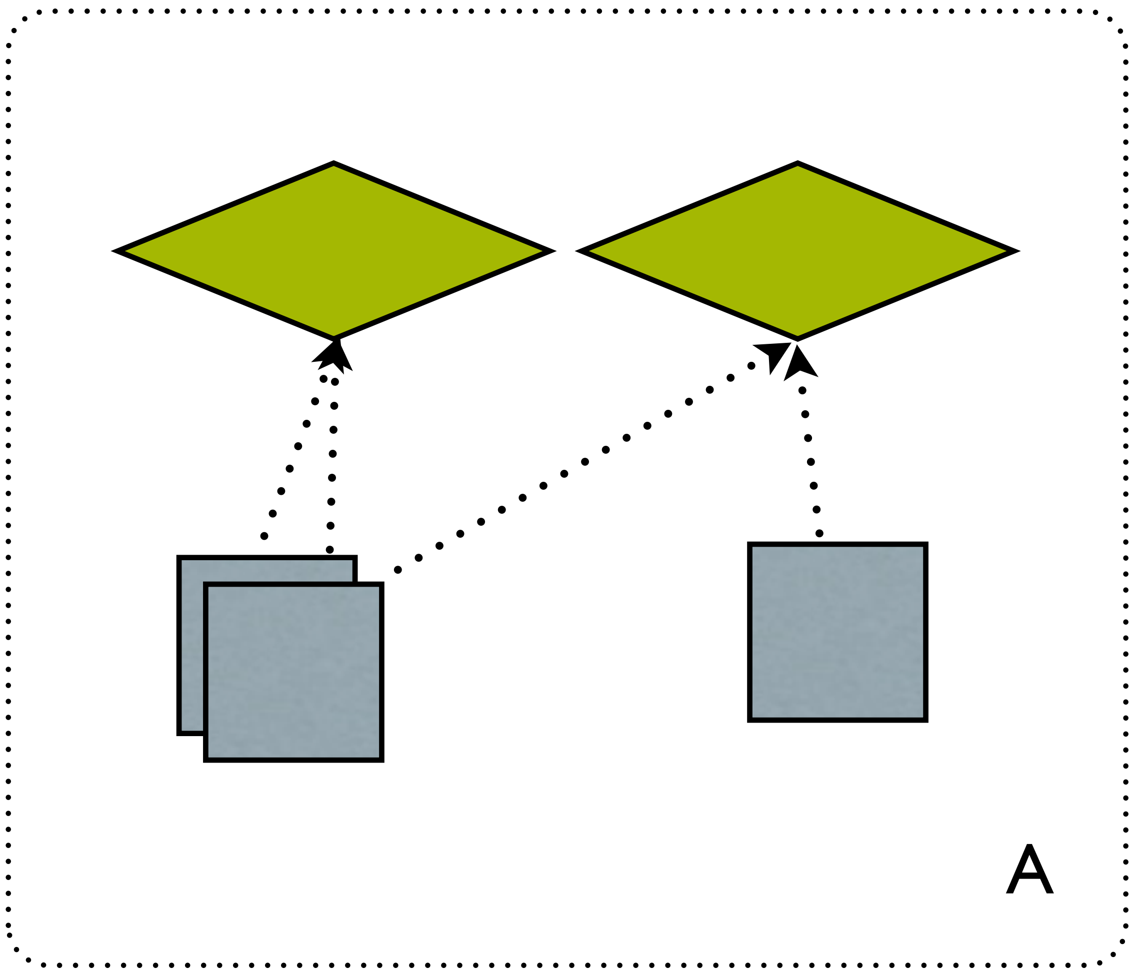
```
(defrecord Mean [last-winner]
  Player
  (choose [_]
   (if last-winner
     last-winner
     (random-choice)))
  (update-strategy [_ me you]
    (Mean. (when (iwon? me you) me))))
```

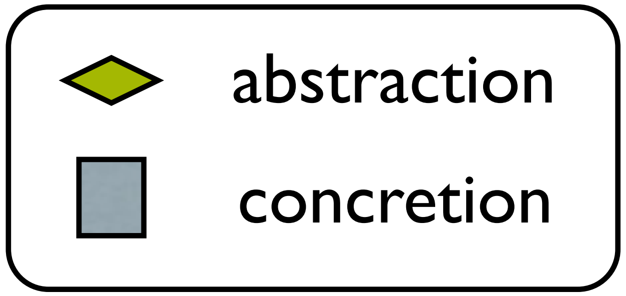
play last winner
or random

remember
how/if I won

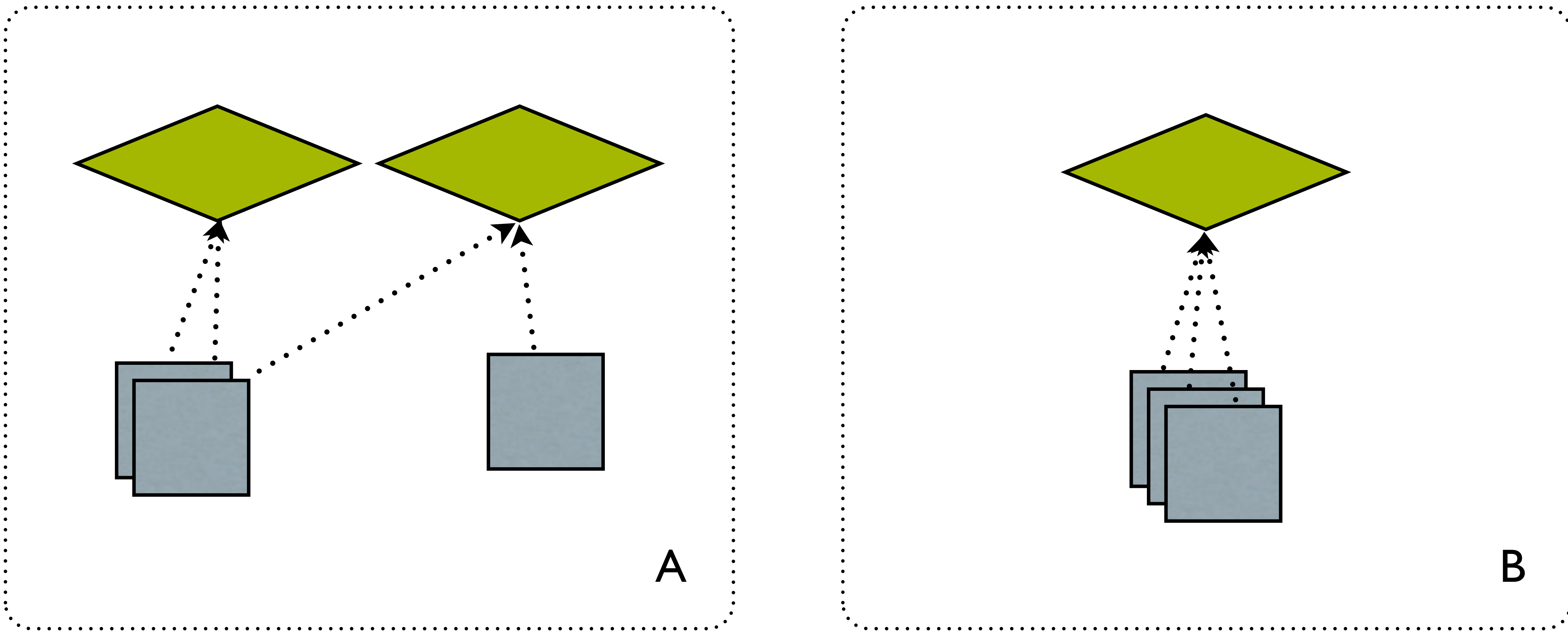
the expression problem

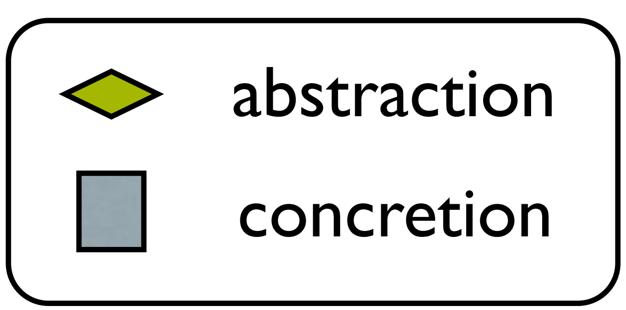
the expression problem



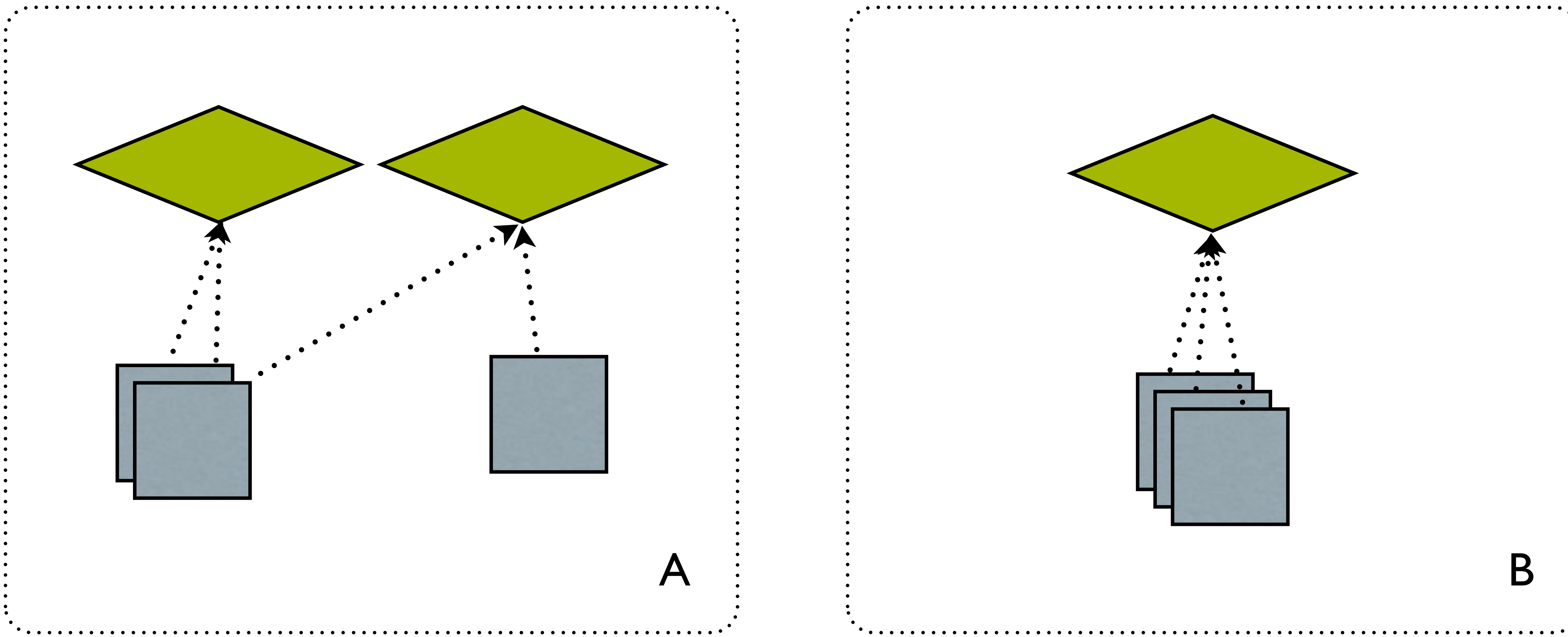


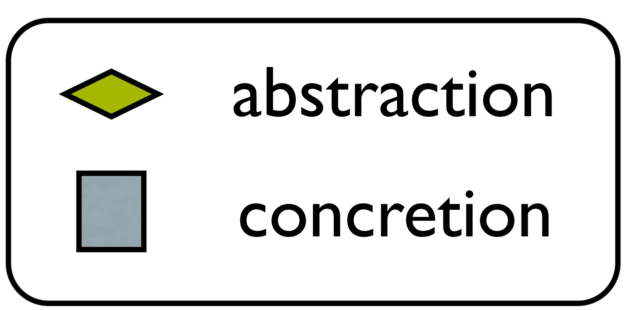
the expression problem





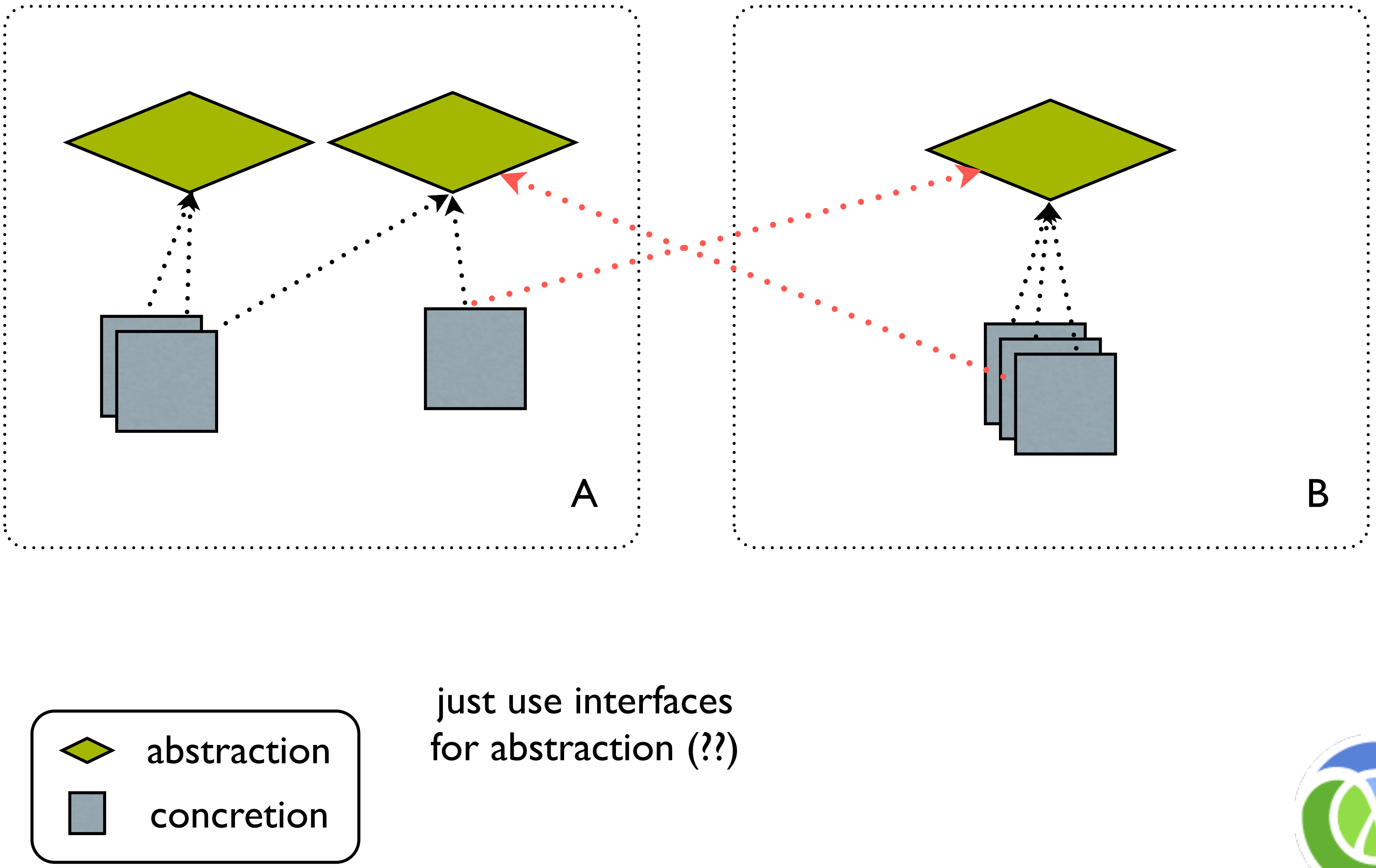
the expression problem





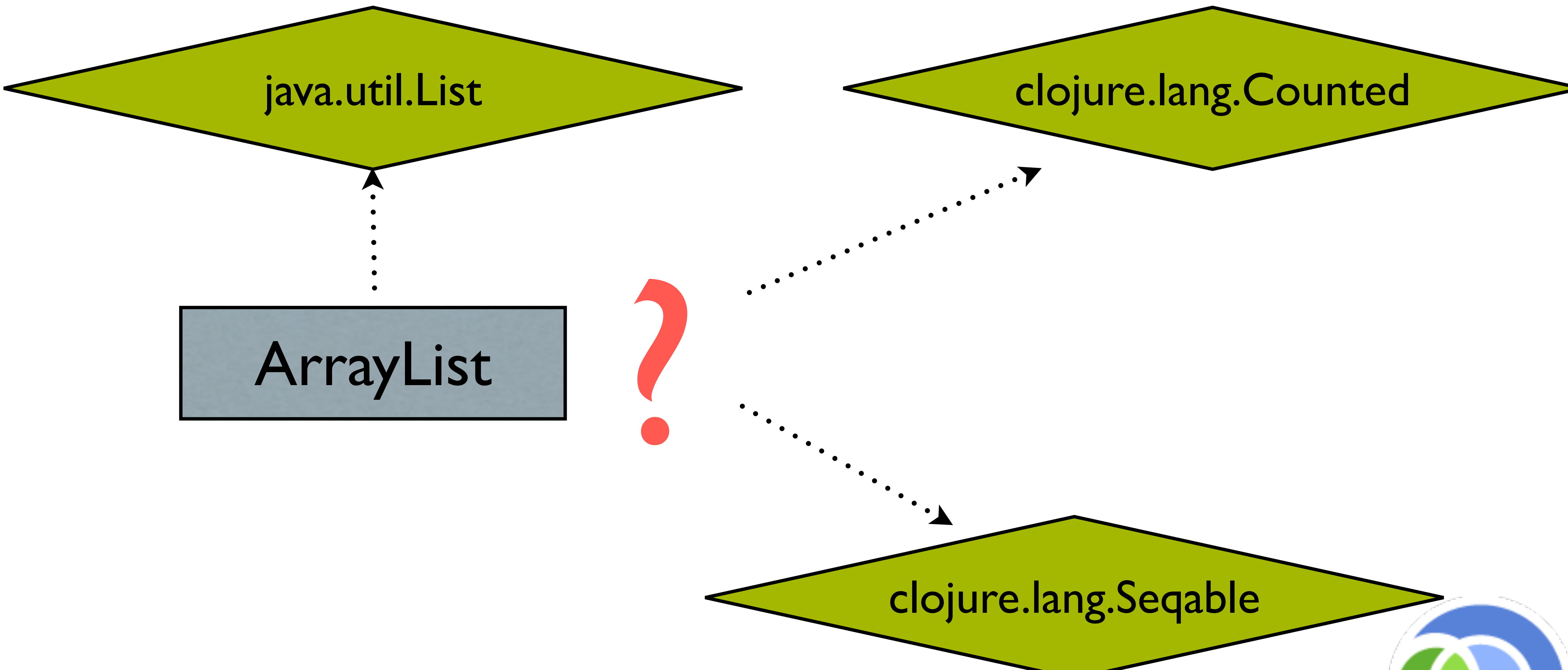
A should be able to work with B's abstractions, and vice versa, **without modification of the original code**

is this really a problem?

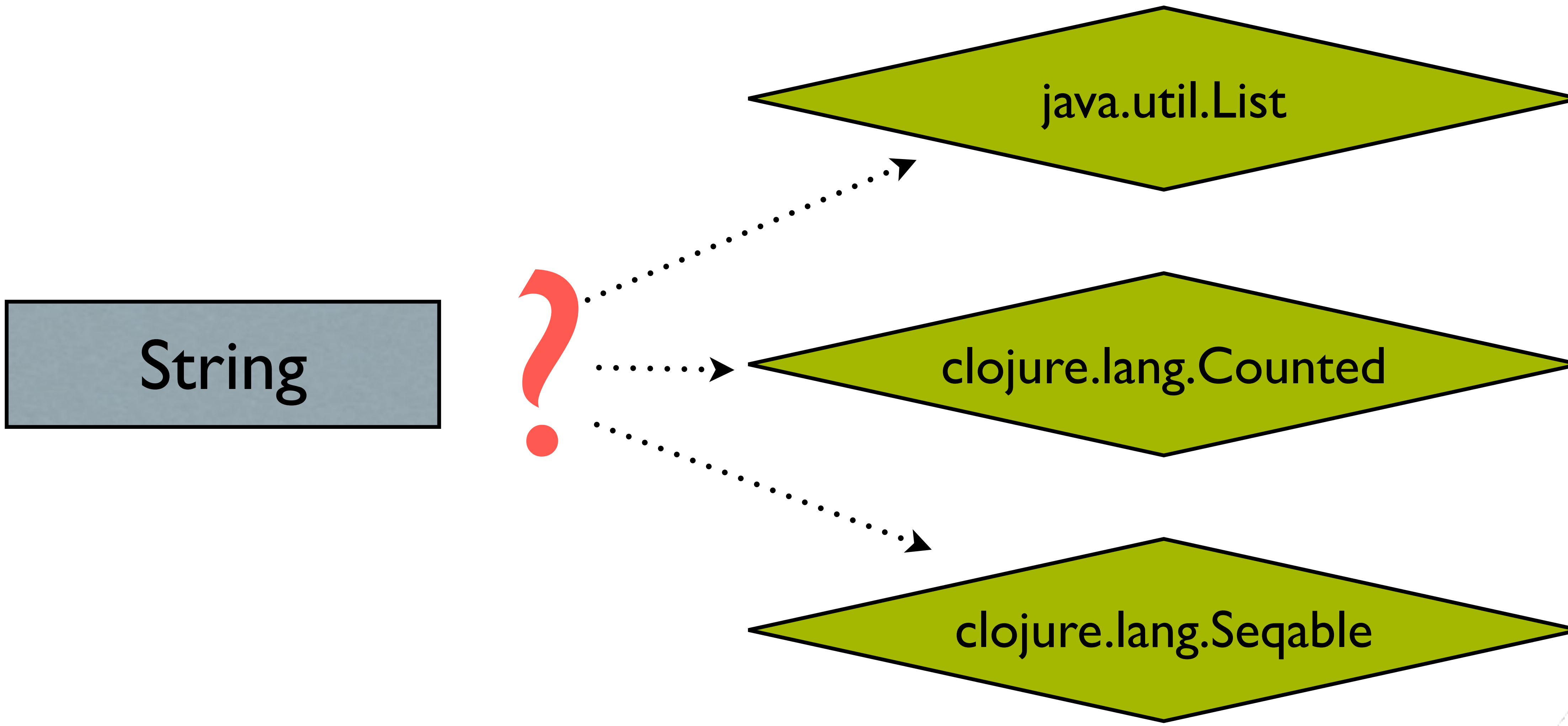


just use interfaces
for abstraction (??)

example: arraylist vs. the abstractions



example: string vs. the abstractions



A can't inherit from B

A can't inherit from B

B is newer than A

A can't inherit from B

B is newer than A

A is hard to change

A can't inherit from B

B is newer than A

A is hard to change

we don't control A

A can't inherit from B

B is newer than A

A is hard to change

we don't control A

happens even **within** *a single lib*

A can't inherit from B

B is newer than A

A is hard to change

we don't control A

happens even **within** *a single lib*

some approaches to the expression problem

1. roll-your-own

1. roll-your-own

if/then instanceof? logic

1. roll-your-own

if/then instanceof? logic

closed

1. roll-your-own

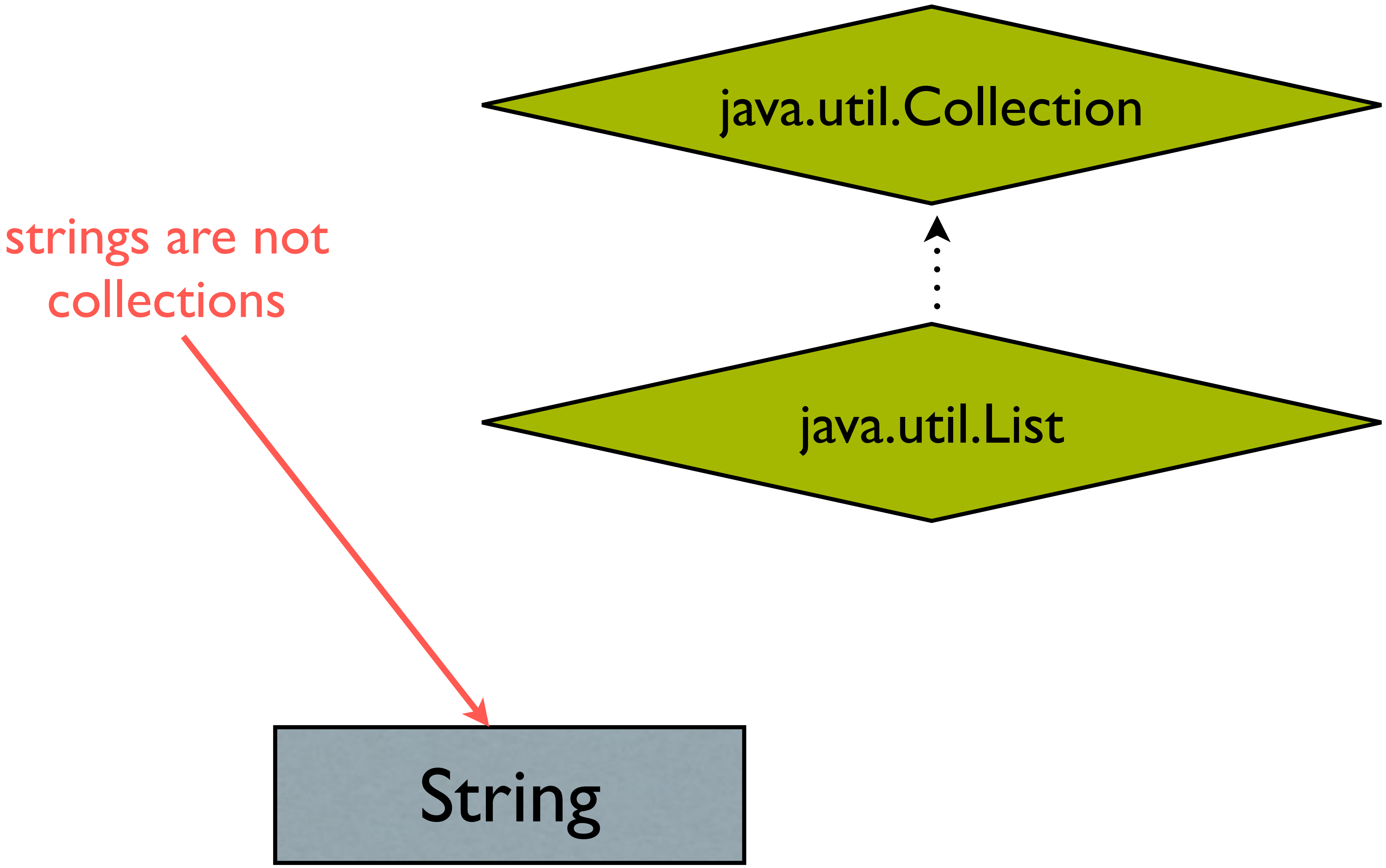
if/then instanceof? logic

closed

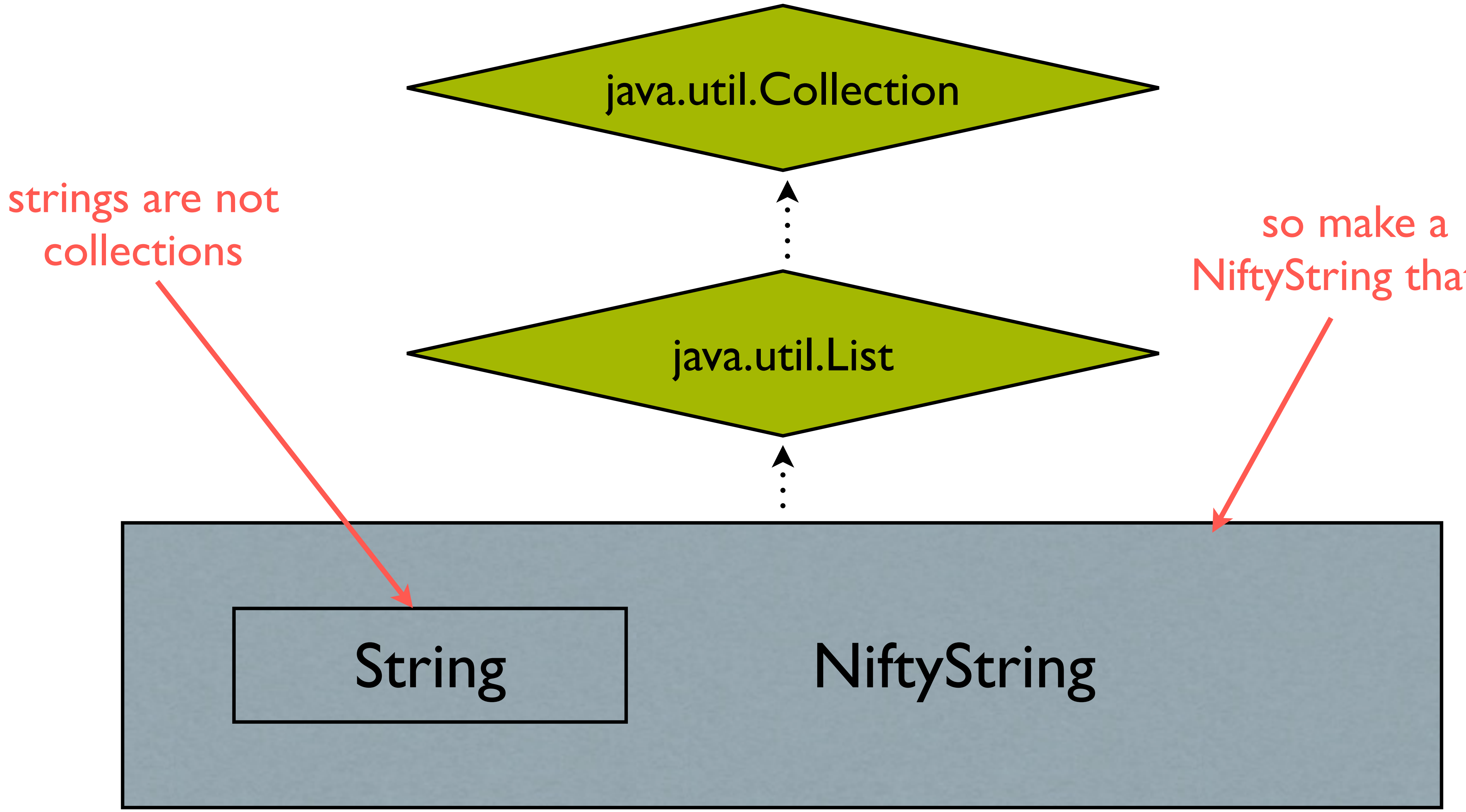
a closed world

```
static ISeq seqFrom(Object coll){
  if(coll instanceof Seqable)
    return ((Seqable) coll).seq();
  else if(coll == null)
    return null;
  else if(coll instanceof Iterable)
    return IteratorSeq.create(((Iterable) coll).iterator());
  else if(coll.getClass().isArray())
    return ArraySeq.createFromObject(coll);
  else if(coll instanceof CharSequence)
    return StringSeq.create((CharSequence) coll);
  else if(coll instanceof Map)
    return seq(((Map) coll).entrySet());
  else {
    Class c = coll.getClass();
    Class sc = c.getSuperclass();
    throw new IllegalArgumentException(
      "Don't know how to create ISeq from: " + c.getName());
  }
}
```

2. wrappers


java.util.Collection
java.util.List
strings are not collections
String

2. wrappers



java.util.Collection
java.util.List
strings are not collections
so make a NiftyString that is
String
NiftyString

wrappers = complexity

wrappers = complexity

ruin identity

wrappers = complexity

ruin identity

ruin equality

wrappers = complexity

ruin identity

ruin equality

cause nonlocal defects

wrappers = complexity

ruin identity

ruin equality

cause nonlocal defects

don't compose: AB + AC != ABC

wrappers = complexity

ruin identity

ruin equality

cause nonlocal defects

don't compose: AB + AC != ABC

have bad names

wrappers = complexity

ruin identity

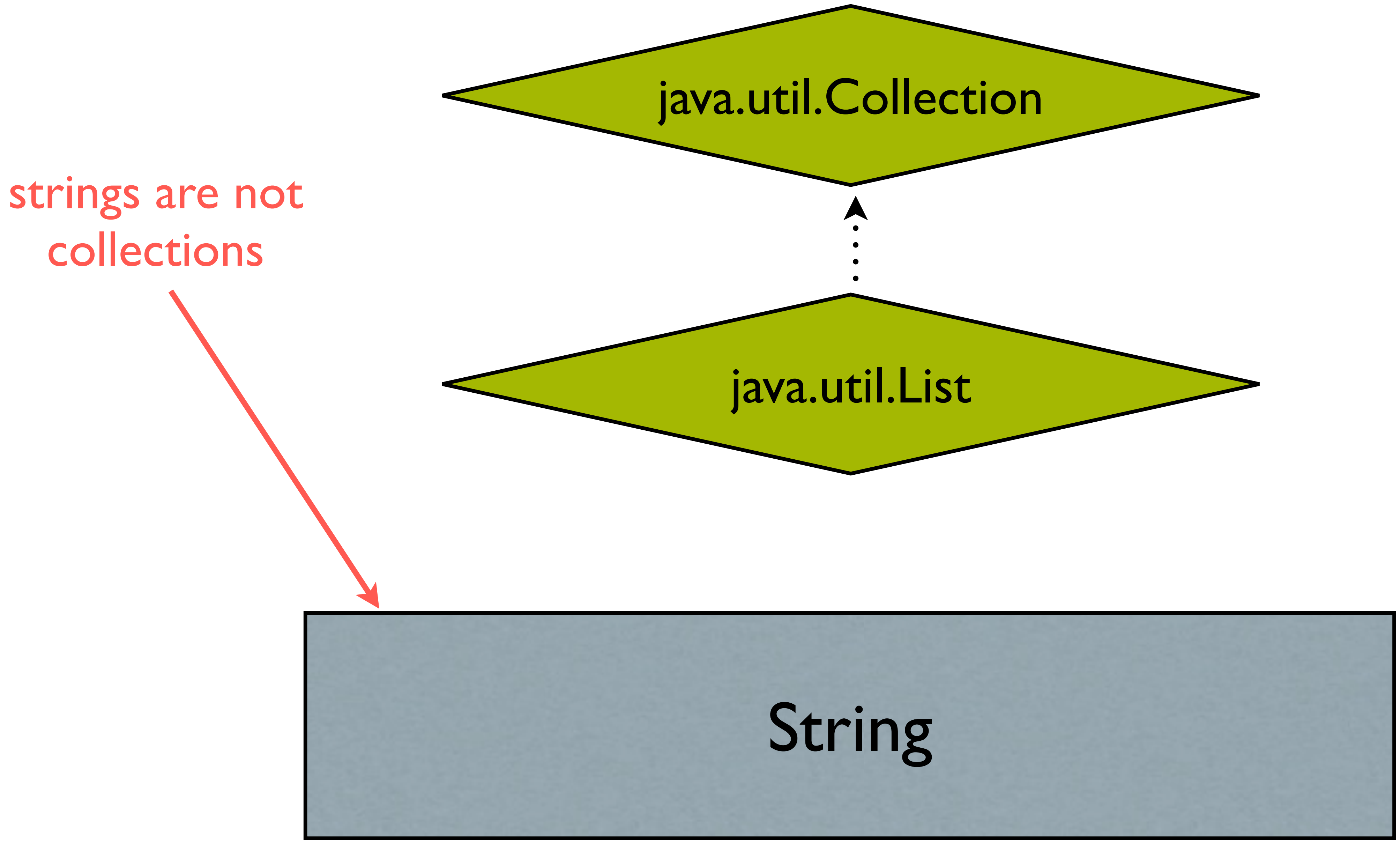
ruin equality

cause nonlocal defects

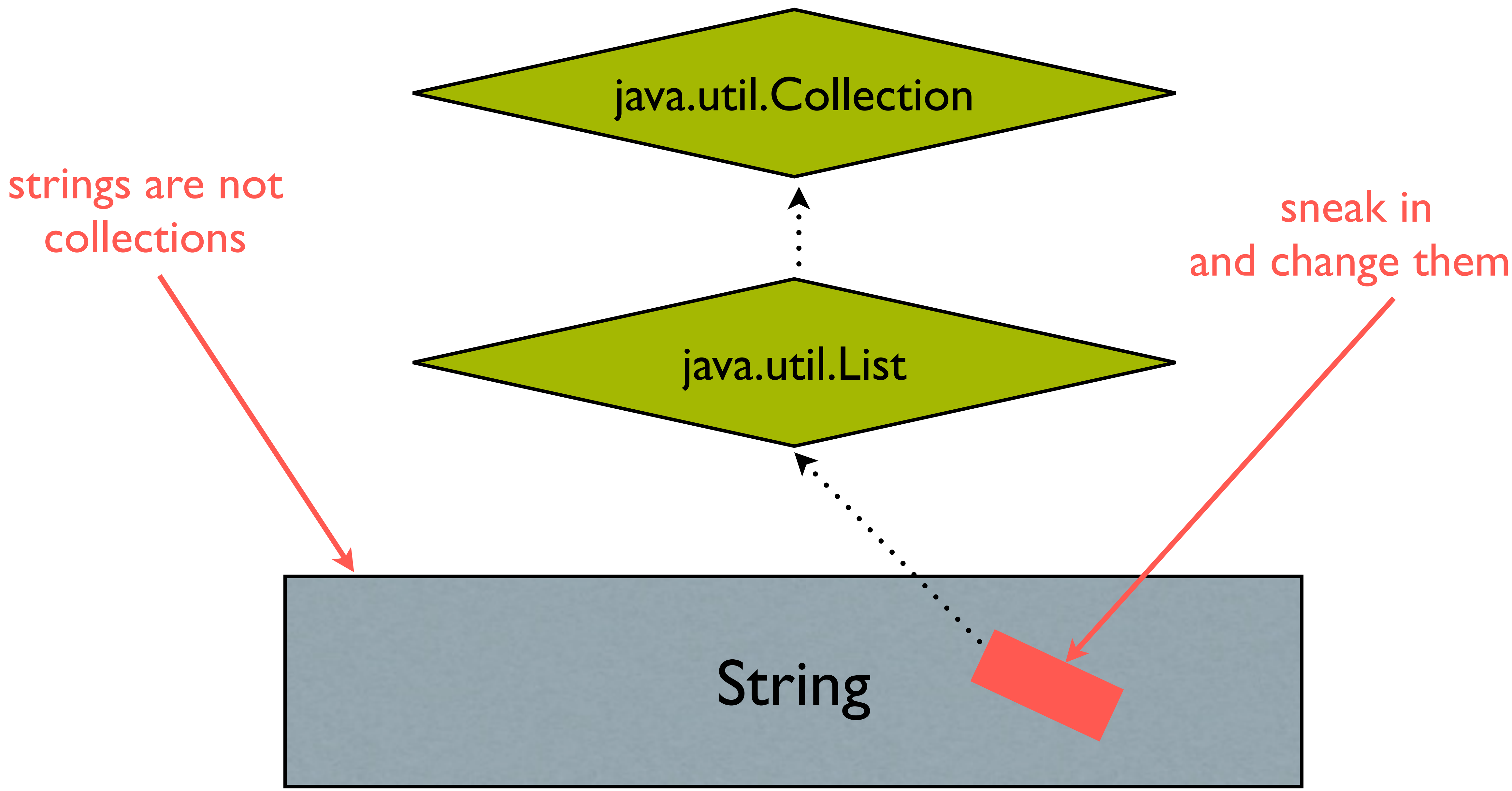
don't compose: AB + AC != ABC

have bad names

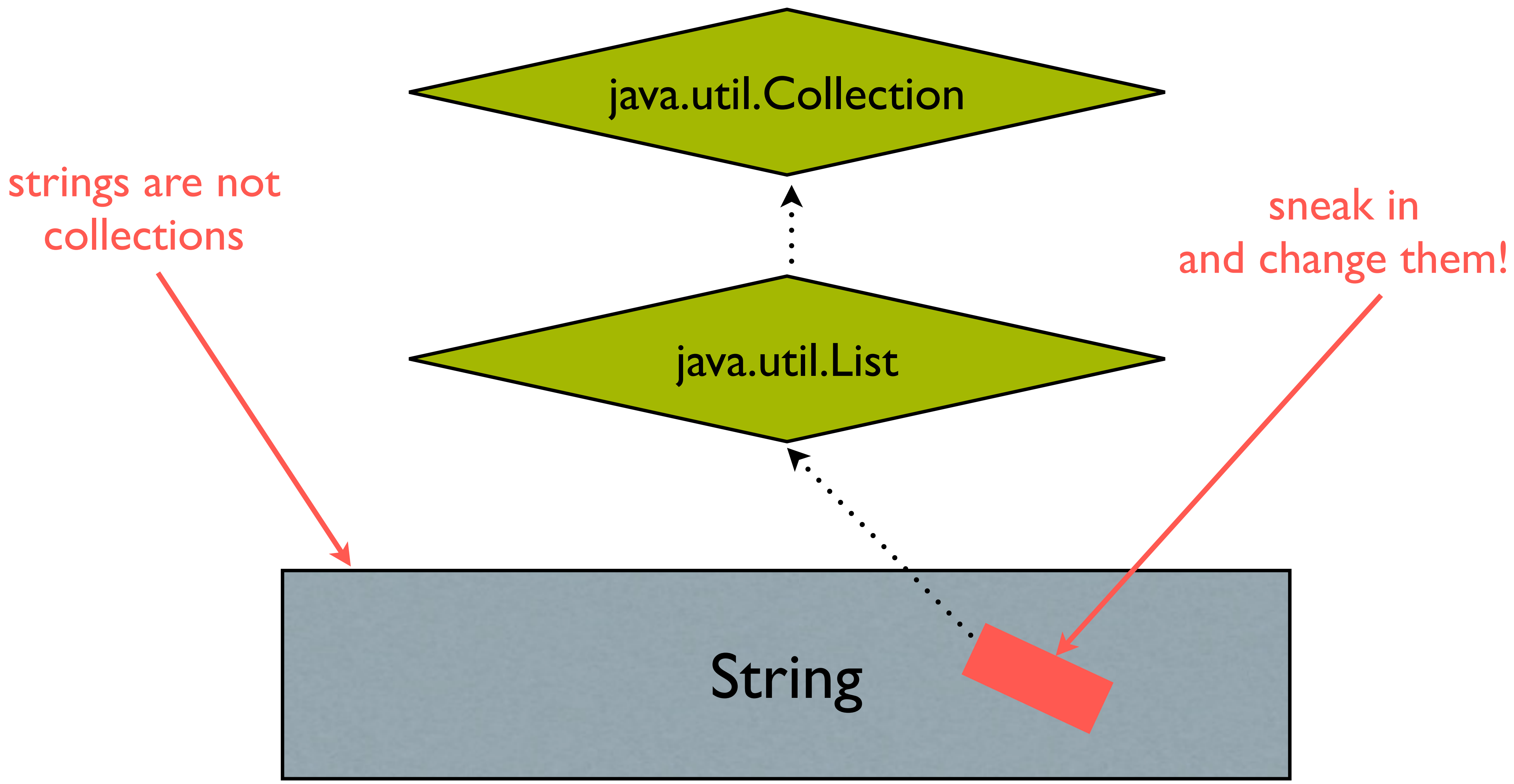
3. monkey patching



3. monkey patching



3. monkey patching



common in e.g. ruby
not possible in java

monkey patching = complexity

monkey patching = complexity

preserves identity (mostly)

monkey patching = complexity

preserves identity (mostly)

ruins namespacing

monkey patching = complexity

preserves identity (mostly)

ruins namespacing

causes nonlocal defects

monkey patching = complexity

preserves identity (mostly)

ruins namespacing

causes nonlocal defects

forbidden in some languages

monkey patching = complexity

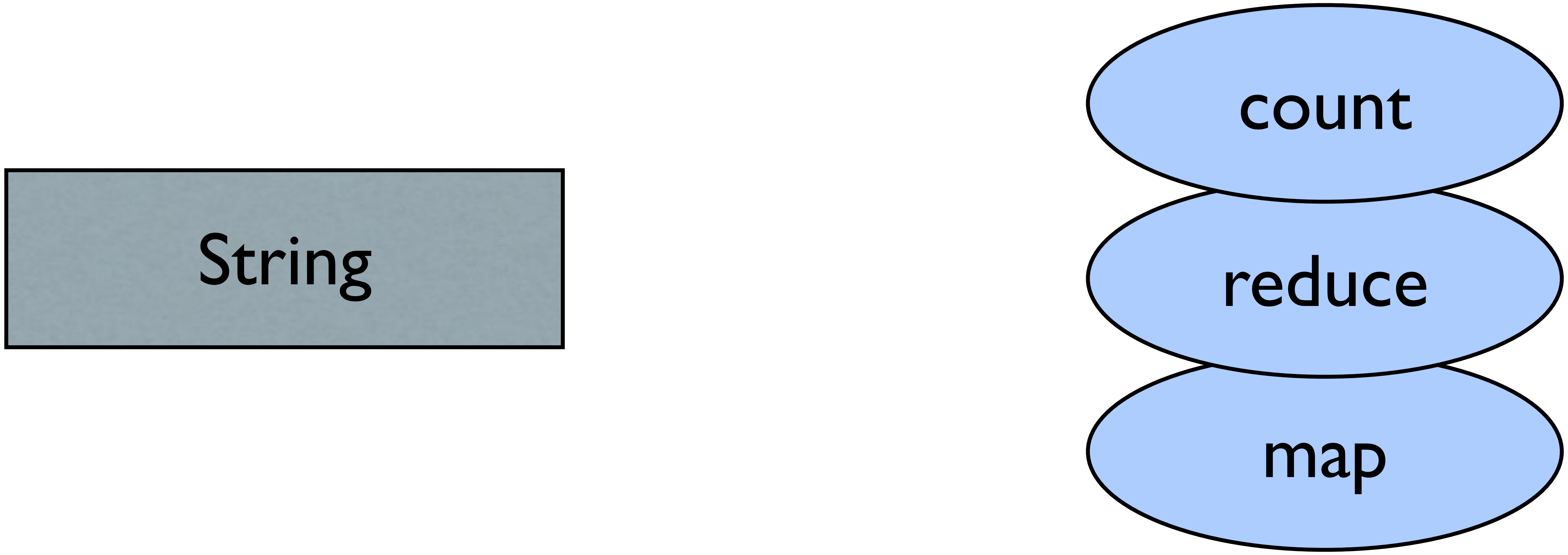
preserves identity (mostly)

ruins namespacing

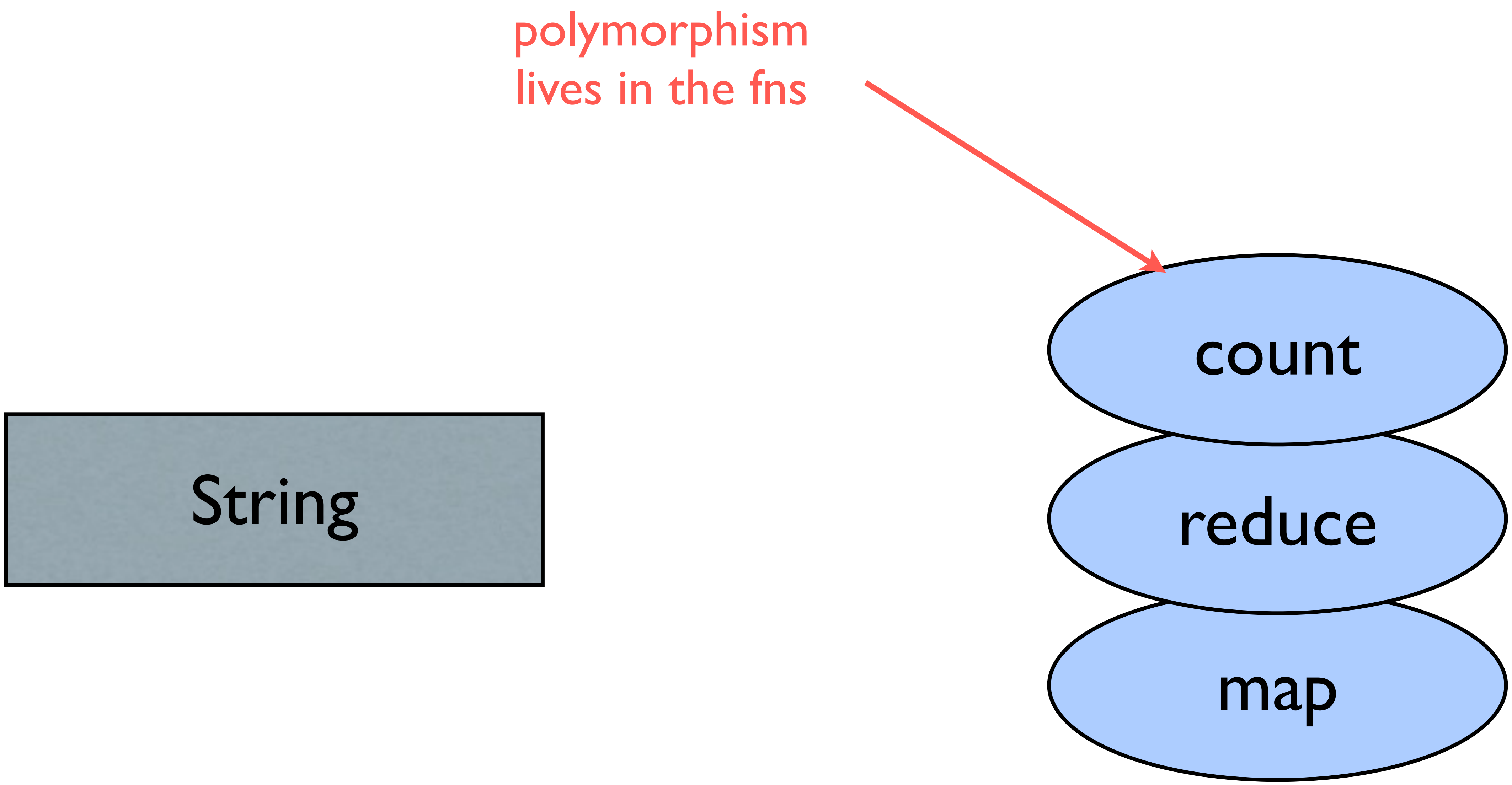
causes nonlocal defects

forbidden in some languages

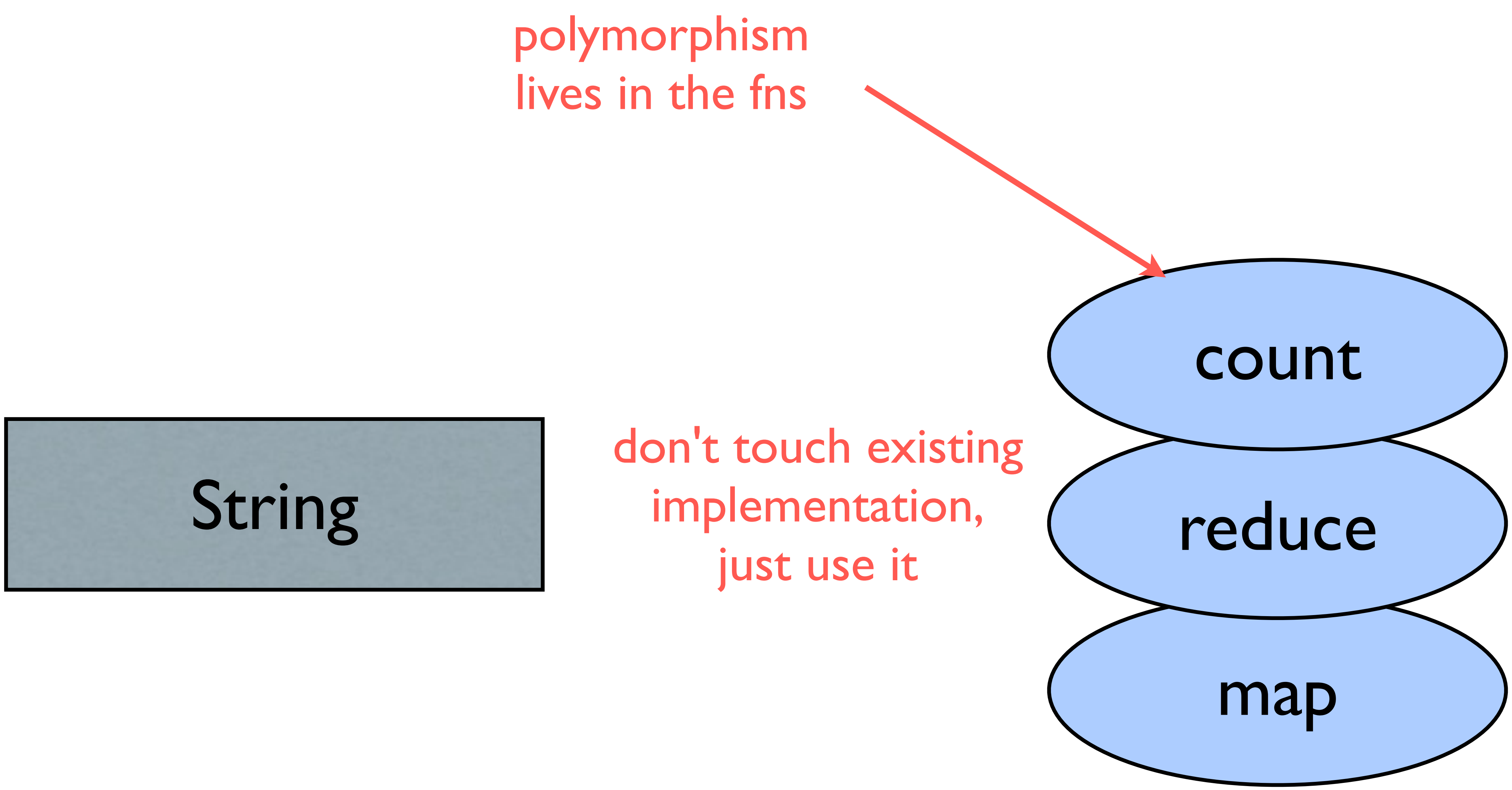
4. generic functions (CLOS)



4. generic functions (CLOS)



4. generic functions (CLOS)



generic functions

generic functions

decouple polymorphism and types

generic functions

decouple polymorphism and types

polymorphism in the fns, not the types

generic functions

decouple polymorphism and types

polymorphism in the fns, not the types

no "isa" requirement

generic functions

decouple polymorphism and types

polymorphism in the fns, not the types

no "isa" requirement

no **type intrusion** necessary

generic functions

decouple polymorphism and types

polymorphism in the fns, not the types

no "isa" requirement

no **type intrusion** necessary

protocols = generic functions
- arbitrary dispatch
+ speed
+ grouping

(and still powerful enough to solve the expression problem!)

a non-trivial example: speeding up reduce

a little reduce

```
(reduce + [1 2 3 4])
-> 10
```

a little reduce

apply this fn...

```
(reduce + [1 2 3 4])
-> 10
```

a little reduce

apply this fn...

```
(reduce + [1 2 3 4])
-> 10
```

pairwise down this collection

a little reduce

apply this fn...

```
(reduce + [1 2 3 4])
-> 10
```

pairwise down this collection

```
(reduce
 #(assoc %1 %2 (inc (%1 %2 0)))
 {}
 "hello")
-> {\o 1, \l 2, \e 1, \h 1}
```

a little reduce

apply this fn...

```
(reduce + [1 2 3 4])
-> 10
```

pairwise down this collection

```
(reduce
 #(assoc %1 %2 (inc (%1 %2 0)))
 {}
 "hello")
-> {\o 1, \l 2, \e 1, \h 1}
```

optional initial value

simple reduce

everything worth reducing is seqable

```
(defn reduce
  [f val coll]
  (let [s (seq coll)]
    (if s
      (recur f (f val (first s)) (next s))
      val)))
```

InternalReduce

```
(defprotocol InternalReduce
  "Protocol for concrete seq types that can reduce
   themselves faster than first/next recursion.
   Called by clojure.core/reduce."
  (internal-reduce [seq f start]))
```

InternalReduce

protocol name

```
(defprotocol InternalReduce
  "Protocol for concrete seq types that can reduce
   themselves faster than first/next recursion.
   Called by clojure.core/reduce."
  (internal-reduce [seq f start]))
```

InternalReduce

protocol name

```
(defprotocol InternalReduce
  "Protocol for concrete seq types that can reduce
   themselves faster than first/next recursion.
   Called by clojure.core/reduce."
  (internal-reduce [seq f start]))
```

docstring

InternalReduce

protocol name

```
(defprotocol InternalReduce
  "Protocol for concrete seq types that can reduce
   themselves faster than first/next recursion.
   Called by clojure.core/reduce."
  (internal-reduce [seq f start]))
```

docstring

method sigs

extending to a type

```
(extend-protocol InternalReduce
  nil
  (internal-reduce
   [s f val]
   val))
```

extending to a type

```
(extend-protocol InternalReduce
  nil
  (internal-reduce
   [s f val]
   val))
```

type (or nil)

extending to a type

```
(extend-protocol InternalReduce
  nil
  (internal-reduce
   [s f val]
   val))
```

type (or nil)

implementation

another extension

```
clojure.lang.StringSeq
(internal-reduce
 [str-seq f val]
 (let [s (.s str-seq)]
   (loop [i (.i str-seq)
          val val]
     (if (< i (.length s))
       (recur (inc i) (f val (.charAt s i)))
       val))))
```

another extension

```
clojure.lang.StringSeq
(internal-reduce
 [str-seq f val]
 (let [s (.s str-seq)]
   (loop [i (.i str-seq)
          val val]
     (if (< i (.length s))
       (recur (inc i) (f val (.charAt s i)))
       val))))
```

internal knowledge
of String

another extension

could be any type,
owned by me or not

```
clojure.lang.StringSeq
(internal-reduce
 [str-seq f val]
 (let [s (.s str-seq)]
   (loop [i (.i str-seq)
          val val]
     (if (< i (.length s))
       (recur (inc i) (f val (.charAt s i)))
       val))))
```

internal knowledge
of String

changes in your code

changes in your code

none!!

changes in your code

none!!

internal-reduce lives under reduce

changes in your code

- none!!
- internal-reduce lives under reduce
- "Rich abstracts so you don't have to"

changes in your code

none!!

internal-reduce lives under reduce

"Rich abstracts so you don't have to"

protocol mythbusting

protocol mythbusting

there is no nil class

protocol mythbusting

- there is no nil class
- nor any wrapper class

protocol mythbusting

there is no nil class

nor any wrapper class

nor monkey-patching the code of others

protocol mythbusting

there is no nil class

nor any wrapper class

nor monkey-patching the code of others

internal-reduce is **just a fn**

protocol mythbusting

there is no nil class

nor any wrapper class

nor monkey-patching the code of others

internal-reduce is **just a fn**

it lives in a namespace

protocol mythbusting

there is no nil class

nor any wrapper class

nor monkey-patching the code of others

internal-reduce is **just a fn**

it lives in a namespace

name can mean something else in a different ns

protocol mythbusting

there is no nil class

nor any wrapper class

nor monkey-patching the code of others

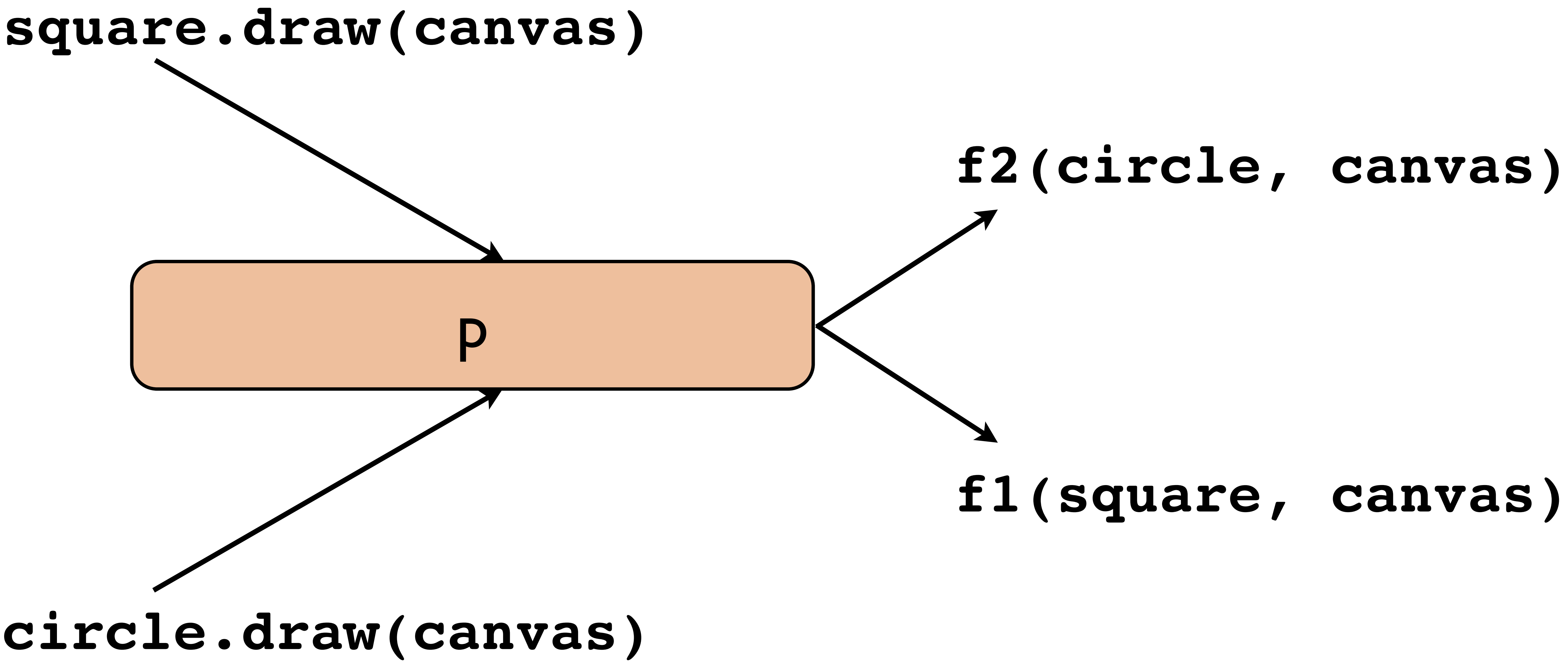
internal-reduce is **just a fn**

it lives in a namespace

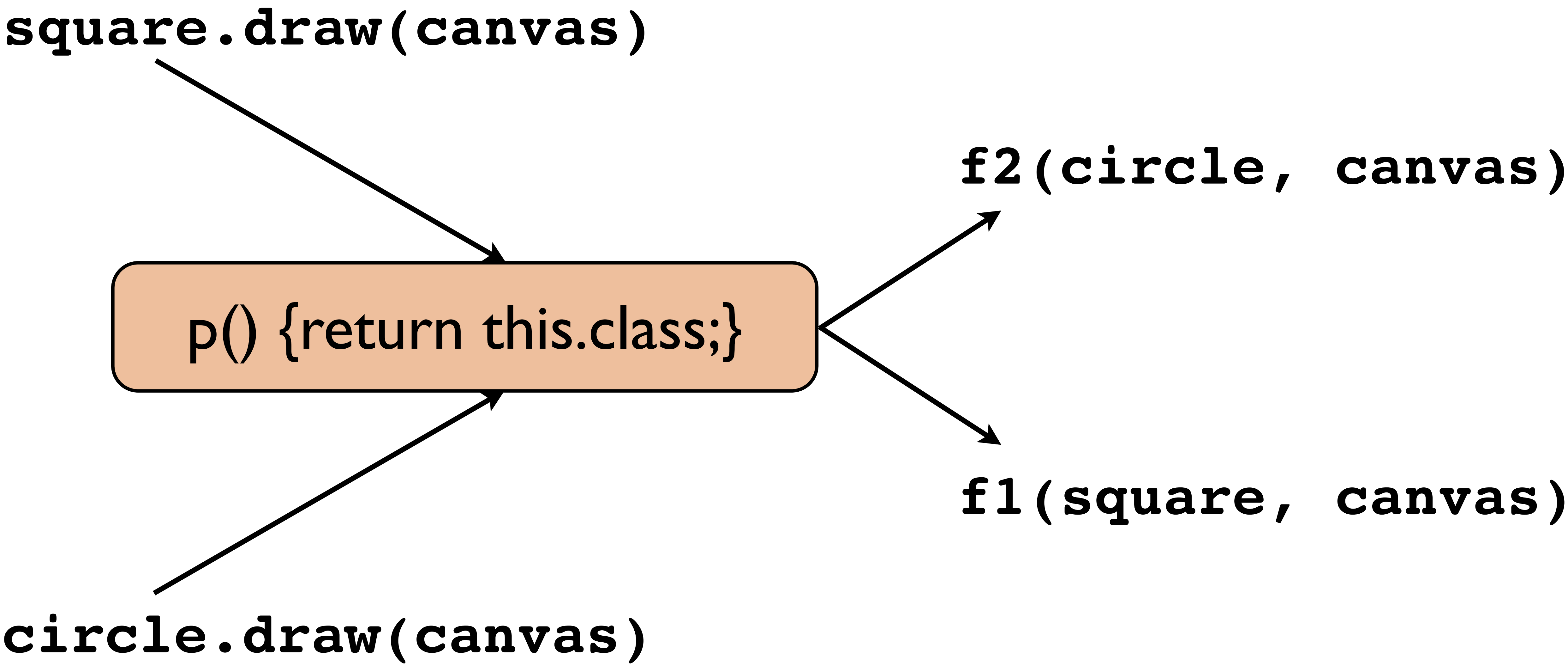
name can mean something else in a different ns

multimethods

polymorphism



p is just a function



clojure multimethods

```
(defmulti blank? class)
```

dispatch by
class of first arg

clojure multimethods

```
(defmulti blank? class)
```

dispatch by
class of first arg

no impl yet!

```
(blank? "blah")
-> No method in multimethod 'blank?'
   for dispatch value: class java.lang.String"
```

clojure multimethods

```
(defmulti blank? class)
```

dispatch by
class of first arg

no impl yet!

```
(blank? "blah")
-> No method in multimethod 'blank?'
   for dispatch value: class java.lang.String"
```

```
(defmethod blank? String [s]
 (every? #(Character/isWhitespace %)) s))
```

add impls
anytime

clojure multimethods

```
(defmulti blank? class)
```

dispatch by
class of first arg

no impl yet!

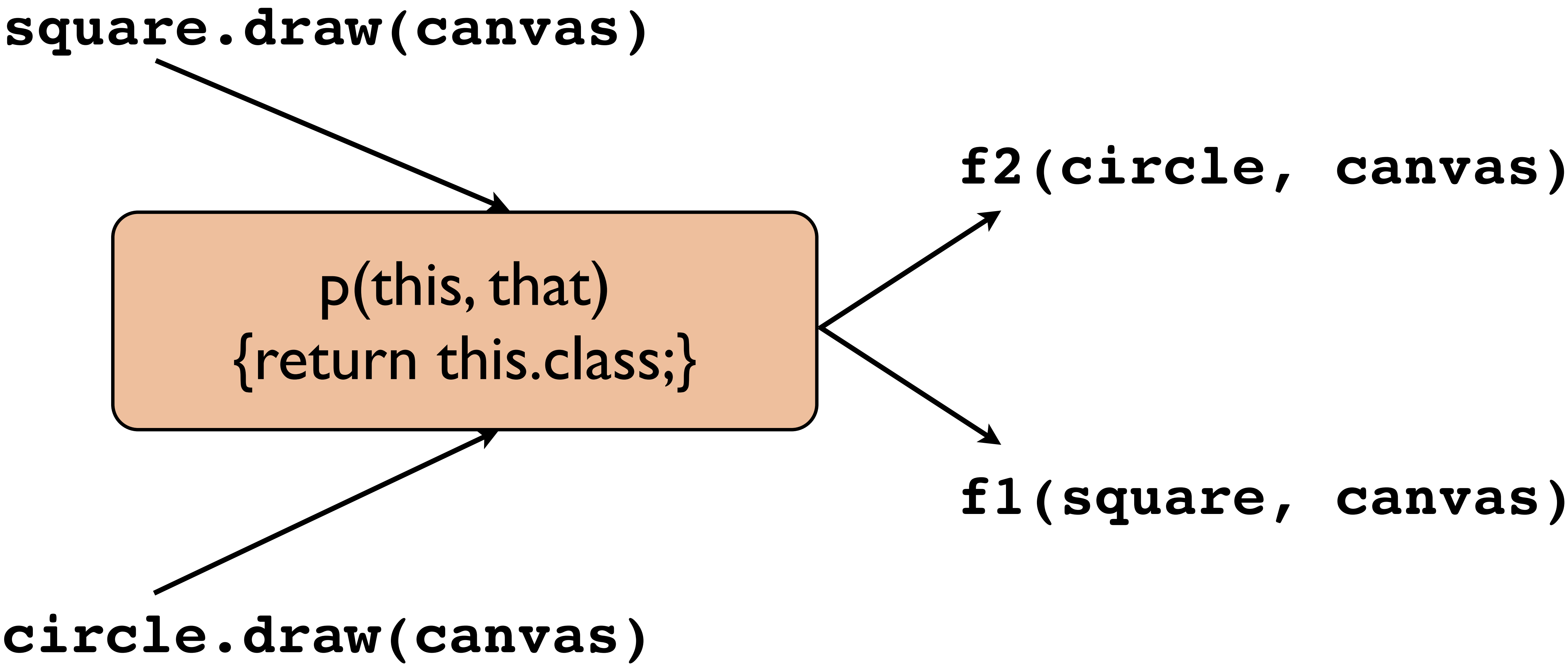
```
(blank? "blah")
-> No method in multimethod 'blank?'
   for dispatch value: class java.lang.String"
```

```
(defmethod blank? String [s]
 (every? #(Character/isWhitespace %)) s))
```

add impls
anytime

```
(blank? "blah")
-> false
```

this isn't special



check all args

```
(fn [this, that]
  [(class this)
   (class that)])
```

```
(fn [square, canvas])

(fn [circle, canvas])

(fn [square, surface])

(fn [circle, surface])
```

check arg twice

```
(fn [this, that]
  [(class this)
   (opaque? this)
   (class that)])
```

fn1
fn2
fn3
fn4
fn5
fn6
fn7
fn8

example: coerce

define a
multimethod

```
(defmulti coerce
  (fn [dest-class src-inst]
    [dest-class (class src-inst)]))
```

based on
dest (a class)

and src
(an inst)

method impls

dispatch value to match

```
(defmethod coerce
  [java.io.File String]
  [_ str]
  (java.io.File. str))

(defmethod coerce
  [Boolean/TYPE String] [_ str]
  (contains?
   #{"on" "yes" "true"}
   (.toLowerCase str)))
```

args

body

defaults

```
(defmethod coerce
  :default
  [dest-cls obj]
  (cast dest-cls obj))
```

class inheritance

```
(defmulti whatami? class)

(defmethod whatami? java.util.Collection
  [_] "a collection")

(whatami? (java.util.LinkedList.))
-> "a collection"

(defmethod whatami? java.util.List
  [_] "a list")

(whatami? (java.util.LinkedList.))
-> "a list"
```

add methods anytime

most derived type wins

name inheritance

```clojure
(defmulti interest-rate :type)
(defmethod interest-rate ::account
  [_] 0M)
(defmethod interest-rate ::savings
  [_] 0.02)
```

double colon (::) is shorthand for resolving keyword into the current namespace, e.g. ::savings == :my.current.ns/savings

deriving names

derived name

base name

```
(derive ::checking ::account)
(derive ::savings ::acount)

(interest-rate {:type ::checking})
-> 0M
```

there is no ::checking method, so select method for base name ::account

multimethods lfu

function	notes
`prefer-method`	resolve conflicts
`methods`	reflect on {dispatch, meth} pairs
`get-method`	reflect by dispatch
`remove-method`	remove by dispatch
`prefers`	reflect over preferences

multimethod elegance

multimethod elegance

solve the expression problem

multimethod elegance

solve the expression problem

no wrappers

multimethod elegance

solve the expression problem

no wrappers

non-intrusive

multimethod elegance

solve the expression problem

no wrappers

non-intrusive

open (add more at any time)

multimethod elegance

solve the expression problem

no wrappers

non-intrusive

open (add more at any time)

namespaces work fine

multimethod elegance

solve the expression problem

no wrappers

non-intrusive

open (add more at any time)

namespaces work fine

summary

summary

rethink of traditional oo

summary

rethink of traditional oo

records: concretion done right

summary

rethink of traditional oo

records: concretion done right

protocols: abstraction done right

summary

rethink of traditional oo

records: concretion done right

protocols: abstraction done right

the record/type split

summary

rethink of traditional oo

records: concretion done right

protocols: abstraction done right

the record/type split

solving the expression problem

summary

rethink of traditional oo

records: concretion done right

protocols: abstraction done right

the record/type split

solving the expression problem

multimethods

summary

rethink of traditional oo

records: concretion done right

protocols: abstraction done right

the record/type split

solving the expression problem

multimethods

rock/paper/scissors (lab)

thanks for participating!

http://clojure.org