Clojure Protocols

Stuart Halloway
stu@clojure.com
@stuarthalloway

reading code

atomic data types

type	example	java equivalent
string	`"foo"`	String
character	`\f`	Character
regex	`#"fo*"`	Pattern
a. p. integer	`42`	Int/Long/BigInteger
double	`3.14159`	Double
a.p. double	`3.14159M`	BigDecimal
boolean	`true`	Boolean
nil	`nil`	`null`
ratio	22/7	N/A
symbol	`foo, +`	N/A
keyword	`:foo, ::foo`	N/A

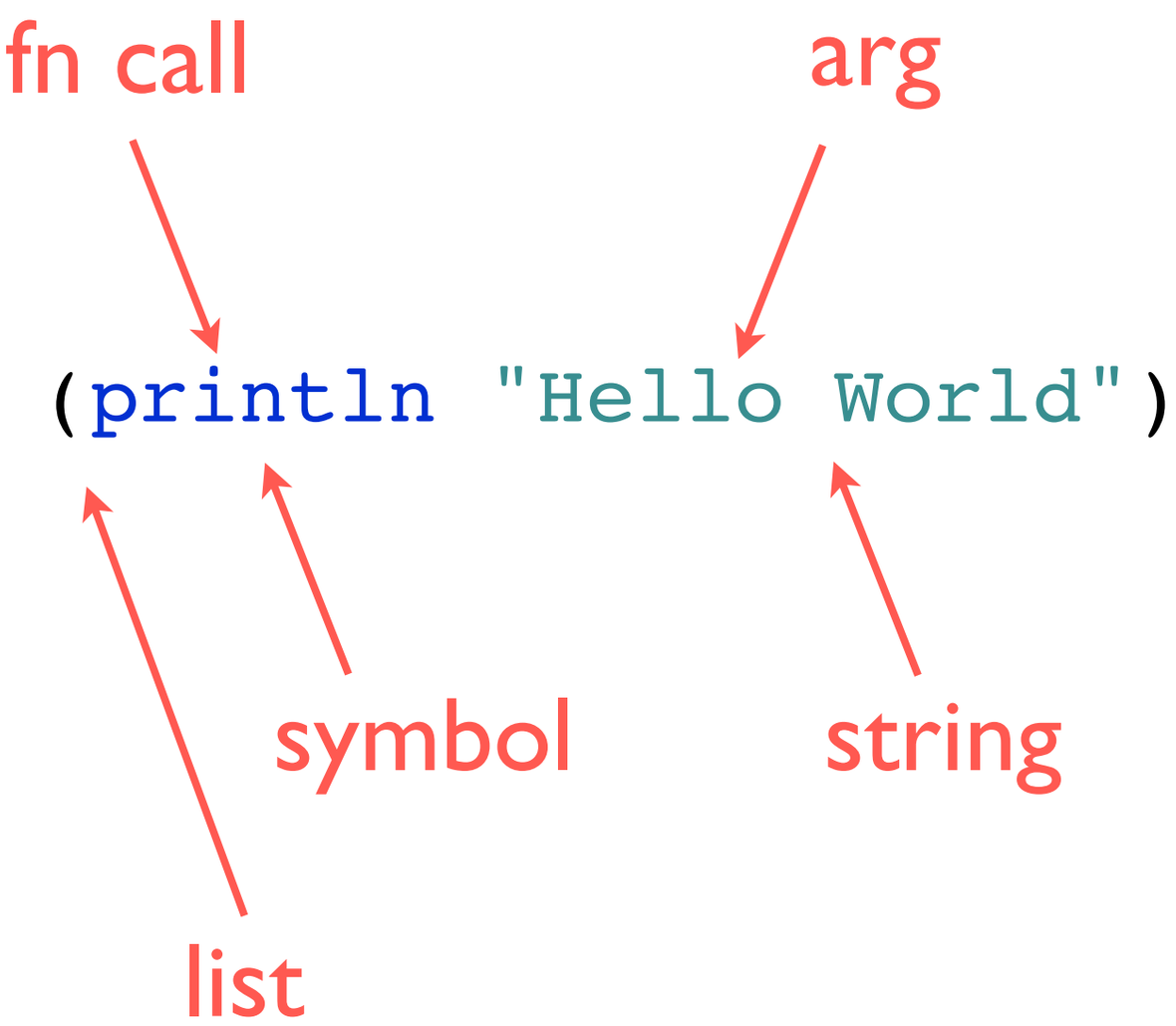
data literals

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}`

function call

semantics:

structure:



function definition

define a fn

fn name

docstring

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

arguments

fn body

it's all data

symbol

symbol

string

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

vector

list

metadata

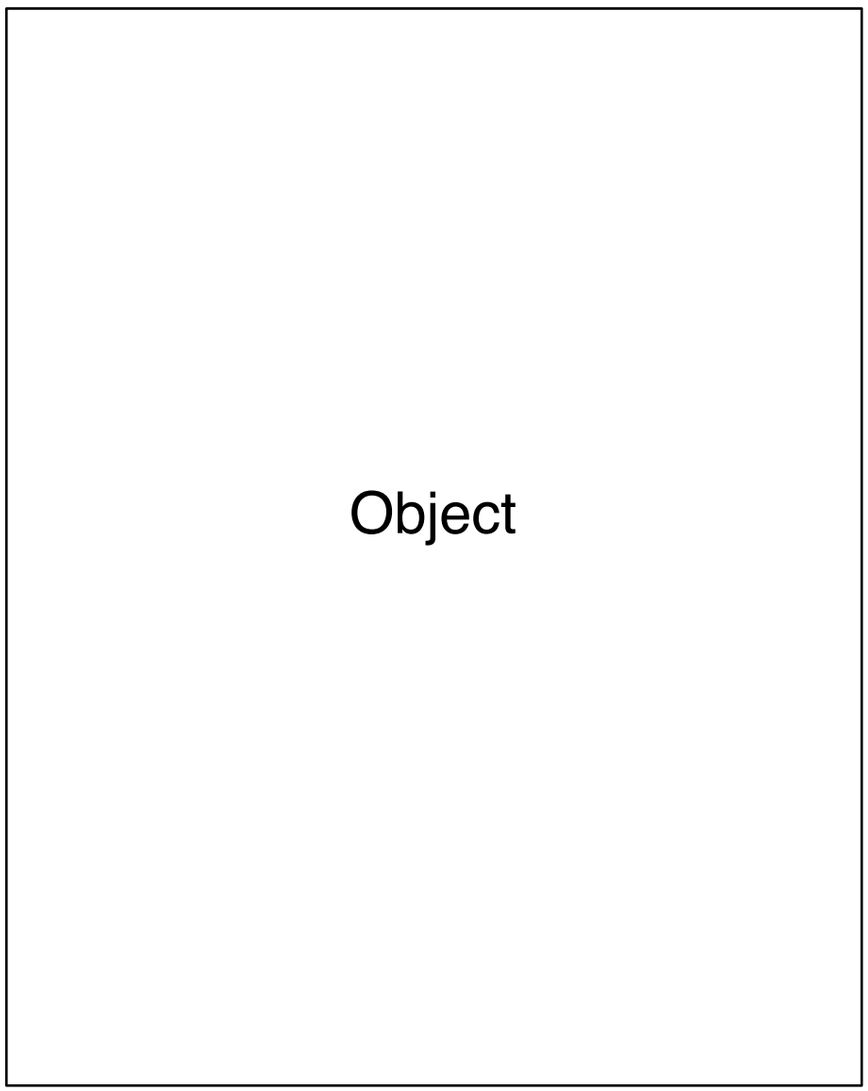
prefix with ^

class name or
arbitrary map

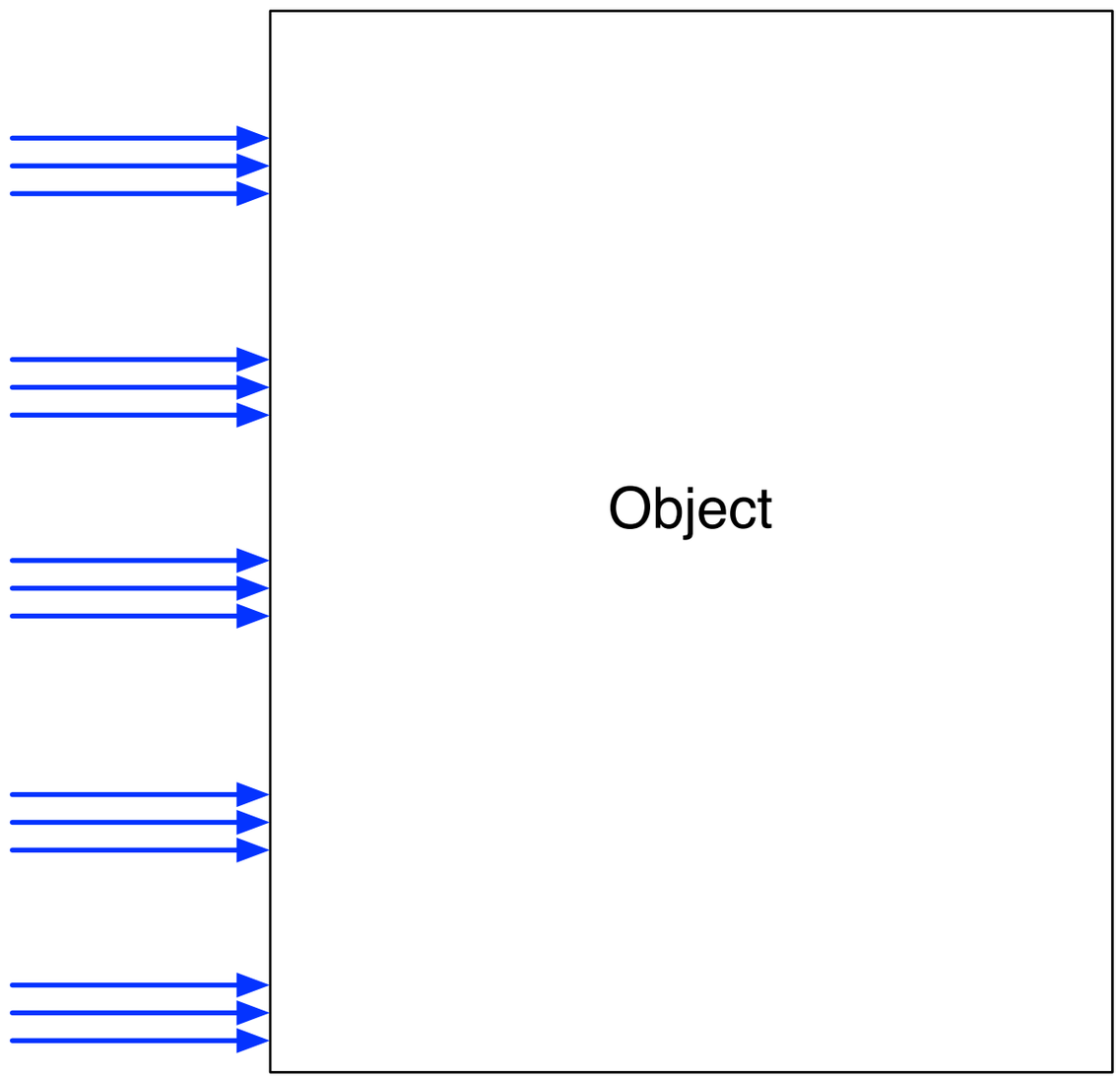
```
(defn ^String greet
  "Returns a friendly greeting"
  [your-name]
  (str "Hello, " your-name))
```

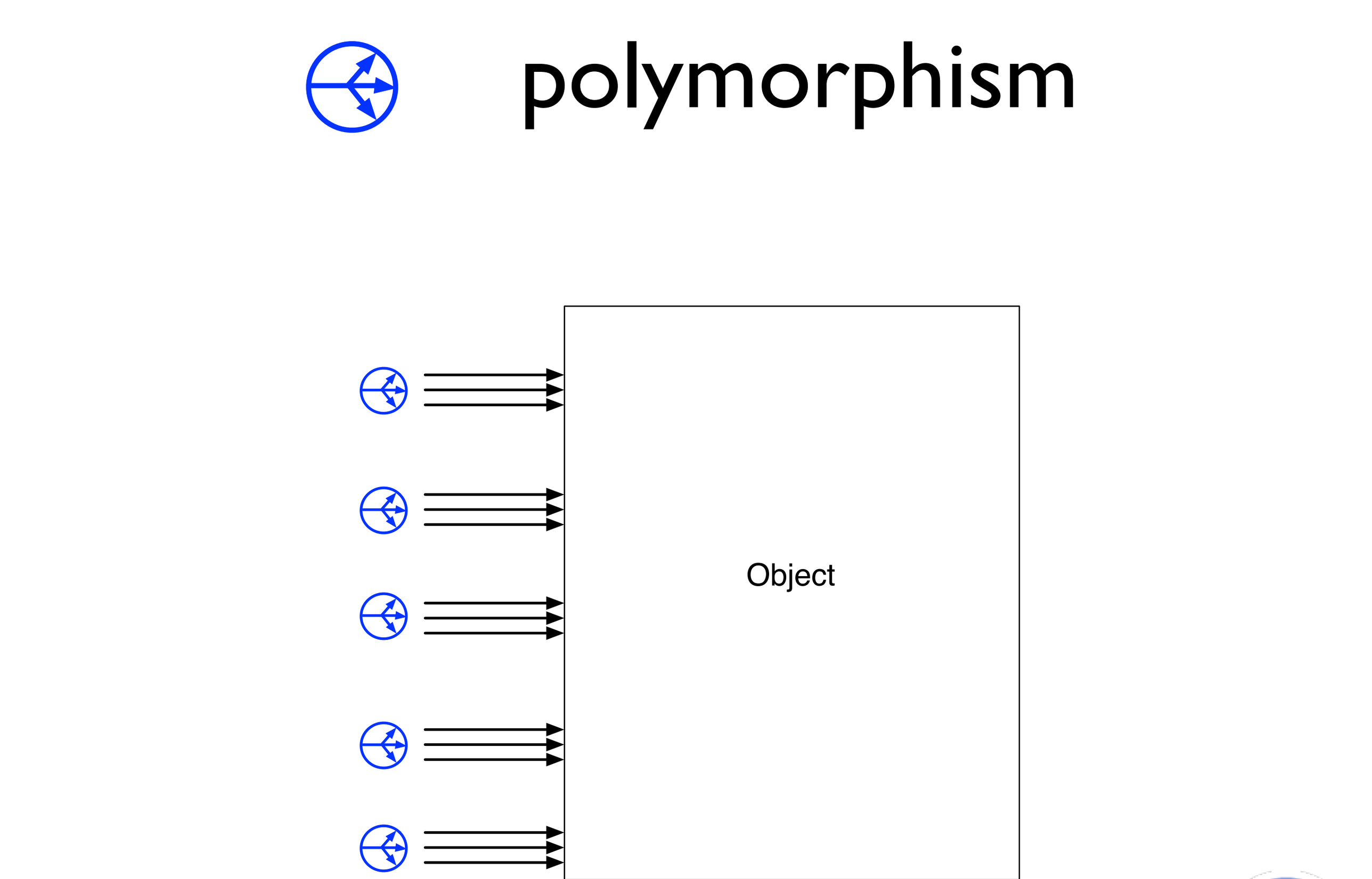
what is OO?

objects provide...



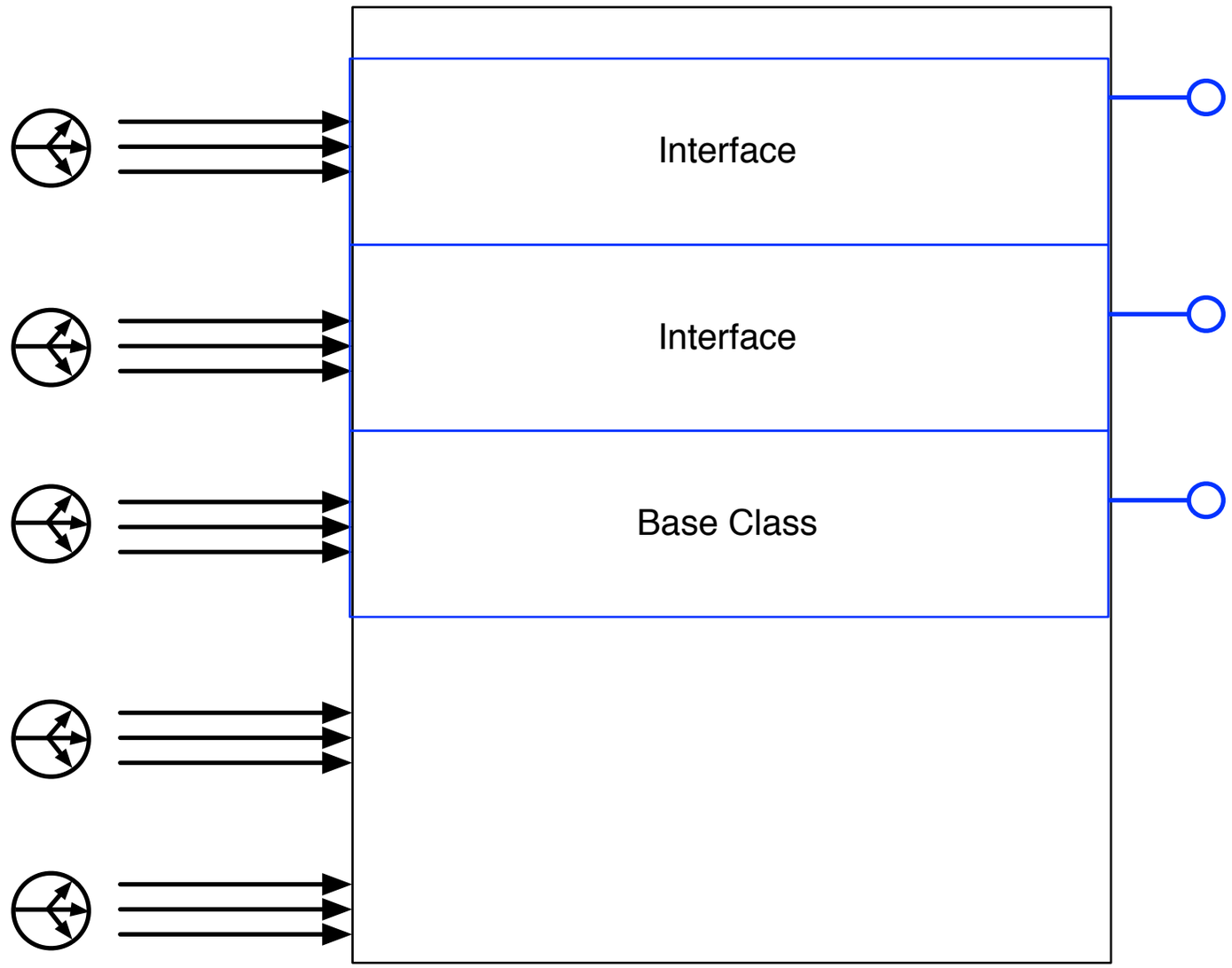
methods



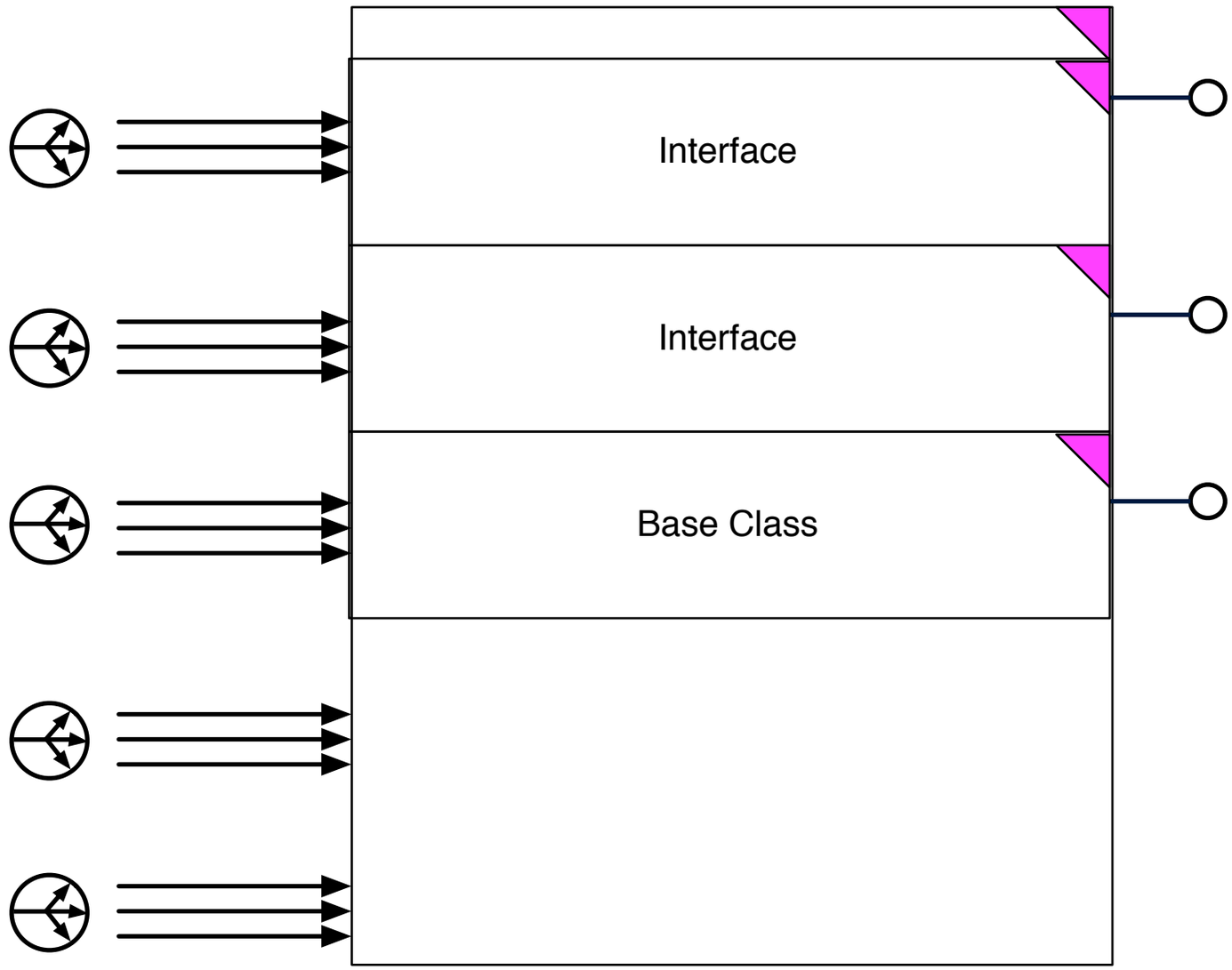


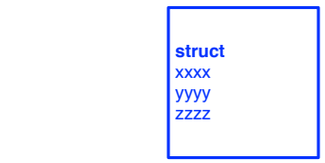
polymorphism

types

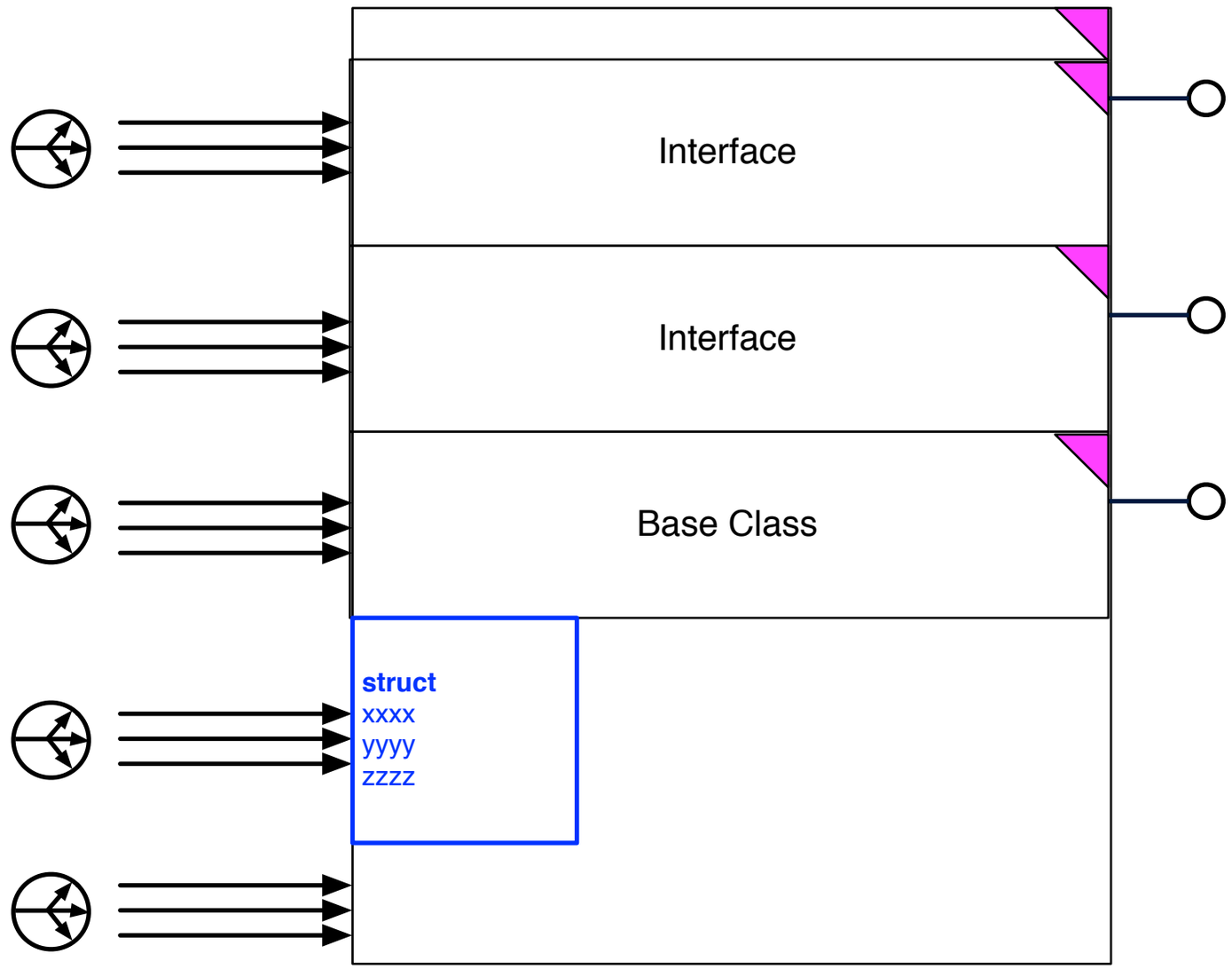


namespaces

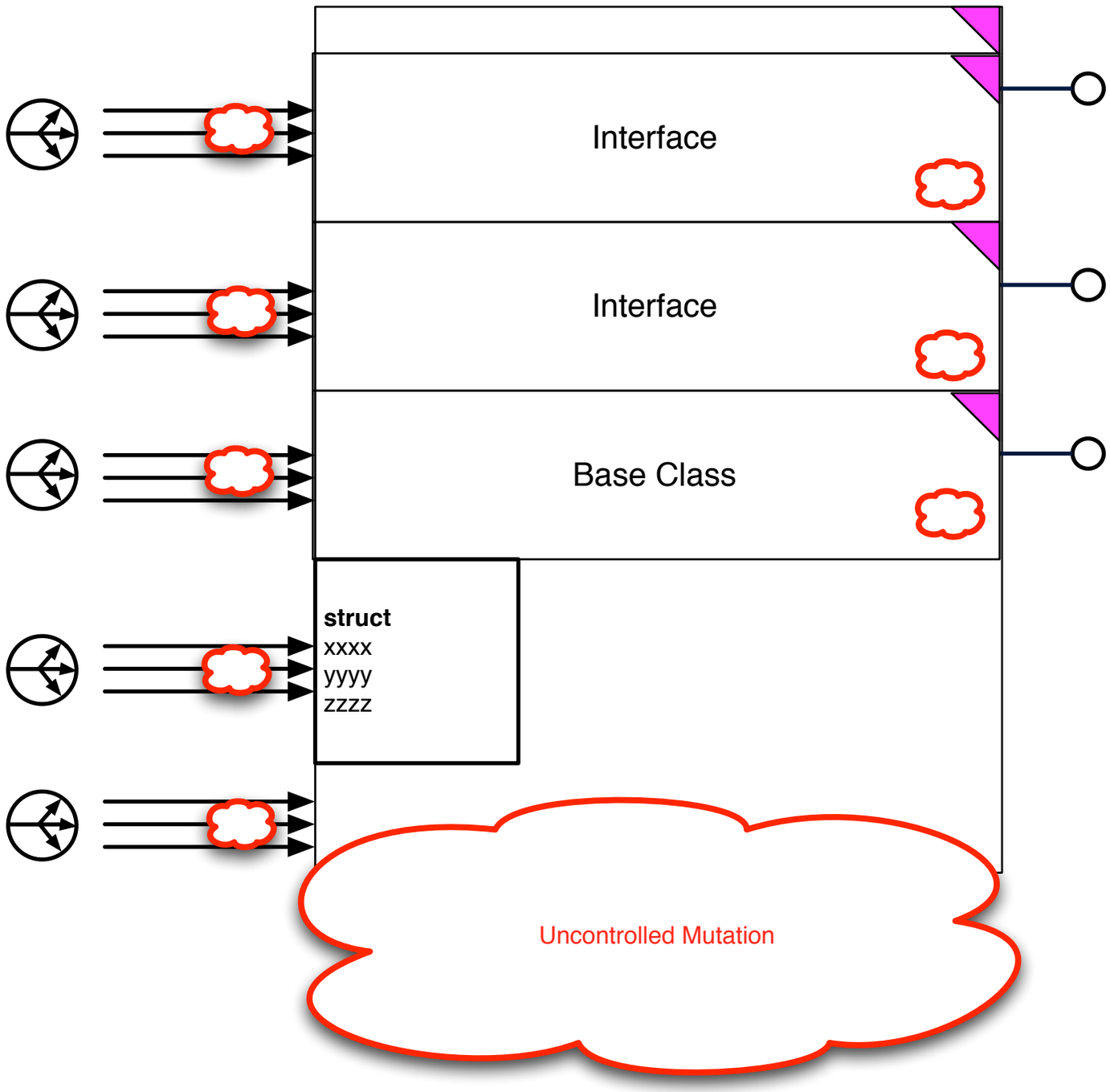



struct
xxxx
yyyy
zzzz

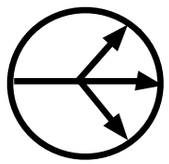
structure


Interface
Interface
Base Class
struct
xxxx
yyyy
zzzz

uncontrolled mutation

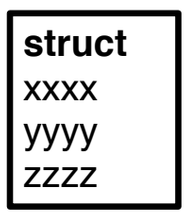


clojure features are a la carte

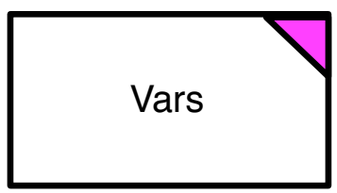


polymorphism

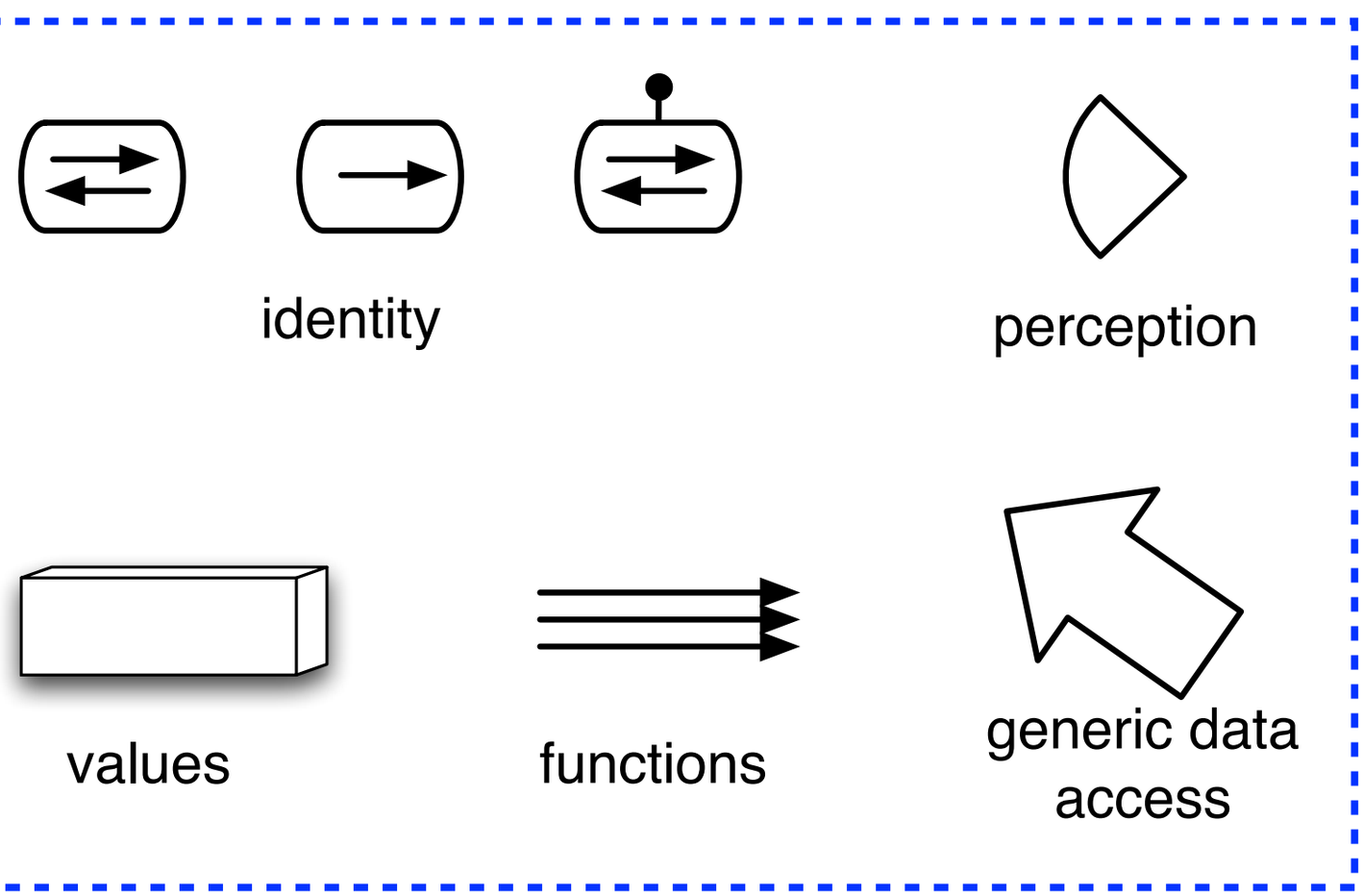
types



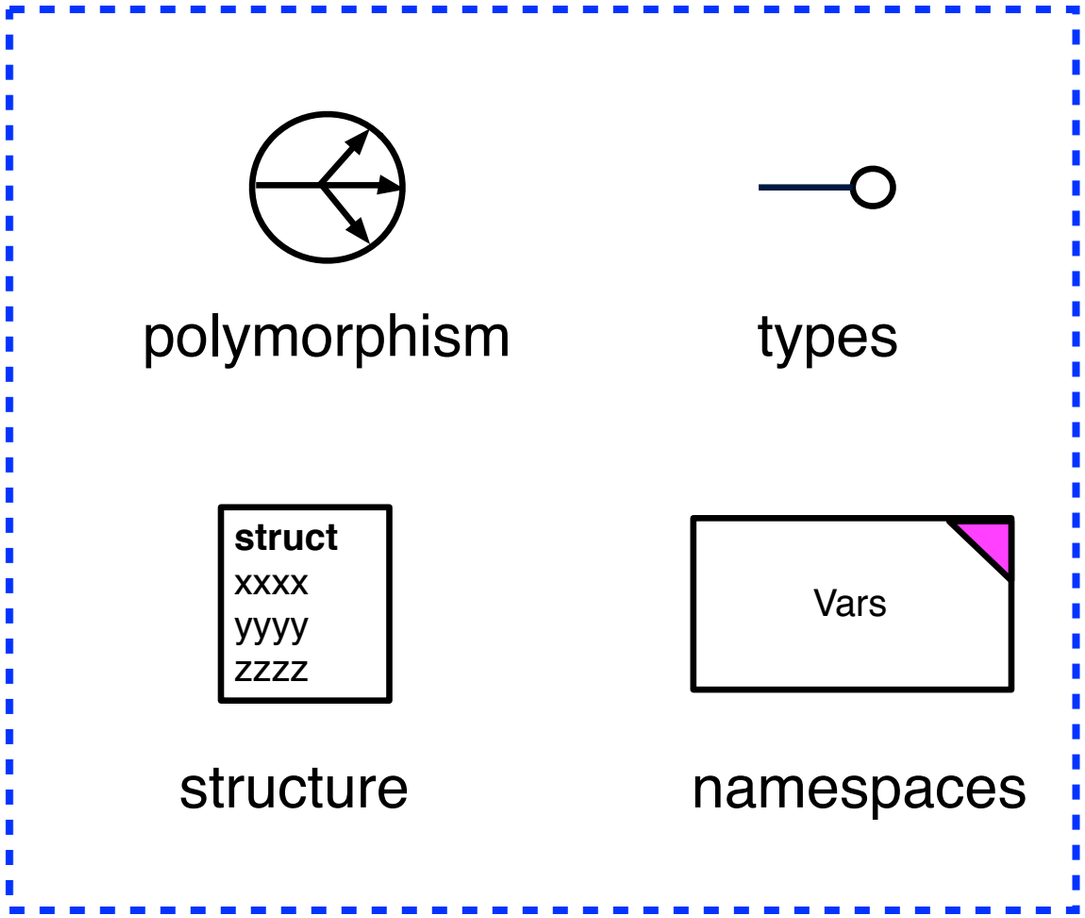
structure



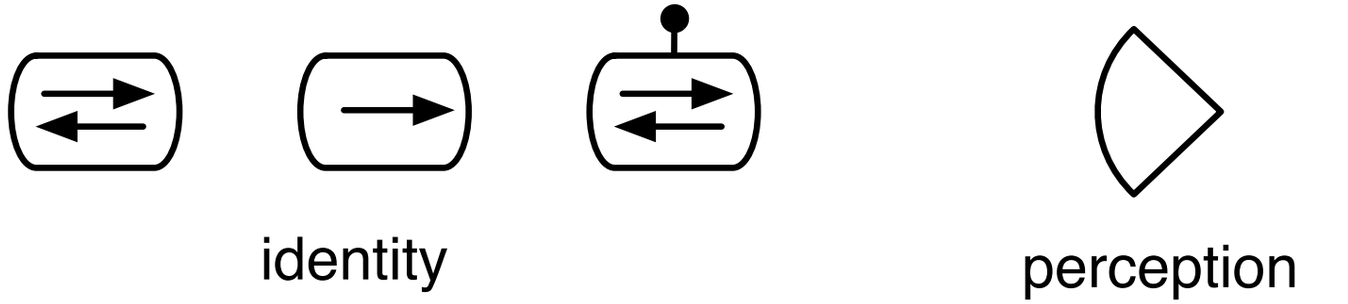
namespaces

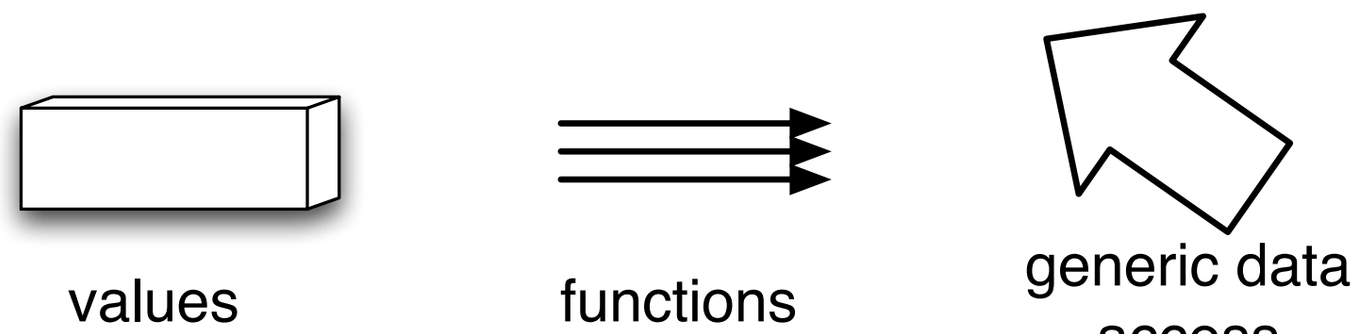


foundation



superstructure

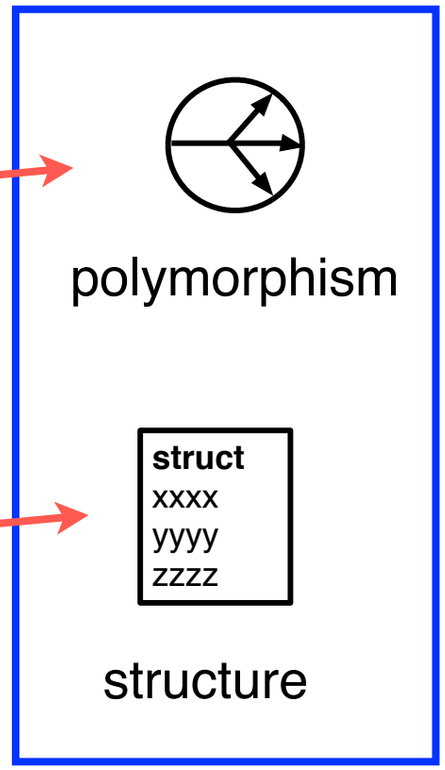


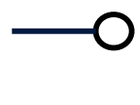


in this talk

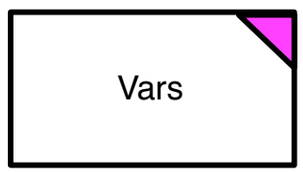
protocols

records

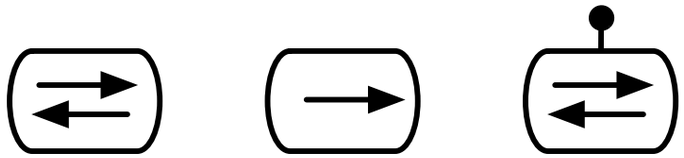




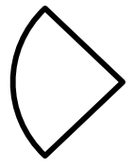
types



namespaces

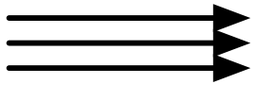


identity

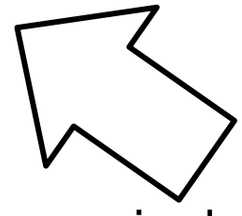


perception

values



functions



generic data access

records

defrecord

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

named type
with slots

defrecord

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

named type with slots

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

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

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

plain ol' class

rasydht*

```
(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...

```
(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}})

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

data-oriented

keyword 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

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...

```
(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

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

nested update

...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)))

(: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)))

(: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 ...} ...}
```

object-oriented

...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)))

(: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 ...} ...}
```

object-oriented

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)))

(: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 ...} ...}
```

object-oriented

still data-oriented:
everything works as before

type is there when you care

protocols

defprotocol

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

defprotocol

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

named set of generic functions

defprotocol

```
(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

defprotocol

```
(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

defines fns in same namespace as protocol

defprotocol

```
(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

defines fns in same namespace as protocol

extending a protocol

extending a protocol

inline

extending a protocol

inline

extend protocol to multiple types

extending a protocol

inline

extend protocol to multiple types

extend type to multiple protocols

extending a protocol

inline

extend protocol to multiple types

extend type to multiple protocols

build directly from fns and maps

extending a protocol

inline

extend protocol to multiple types

extend type to multiple protocols

build directly from fns and maps

extension happens in the protocol fns,
not in the types

extending a protocol

inline

extend protocol to multiple types

extend type to multiple protocols

build directly from fns and maps

extension happens in the protocol fns,
not in the types

extending inline

```
(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"
```

extend type to protocol(s)

```
(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"
```

extending protocol to type(s)

```
;; elided from clojure.java.io
(extend-protocol Coercions
  String
  (as-file [s] (File. s))
  (as-url [s] (URL. s))

  File
  (as-file [f] f)
  (as-url [f] (.toURL f))

  URI
  (as-url [u] (.toURL u))
  (as-file [u] (as-file (as-url u))))
```

roll-your-own

```
;; elided from clojure.java.io
(extend InputStream
  IOFactory
  (assoc default-streams-impl
    :make-input-stream
    (fn [x opts] (BufferedInputStream. x))
    :make-reader
    inputstream->reader))

(extend Reader
  IOFactory
  (assoc default-streams-impl
    :make-reader
    (fn [x opts] (BufferedReader. x))))
```

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 unnamed type

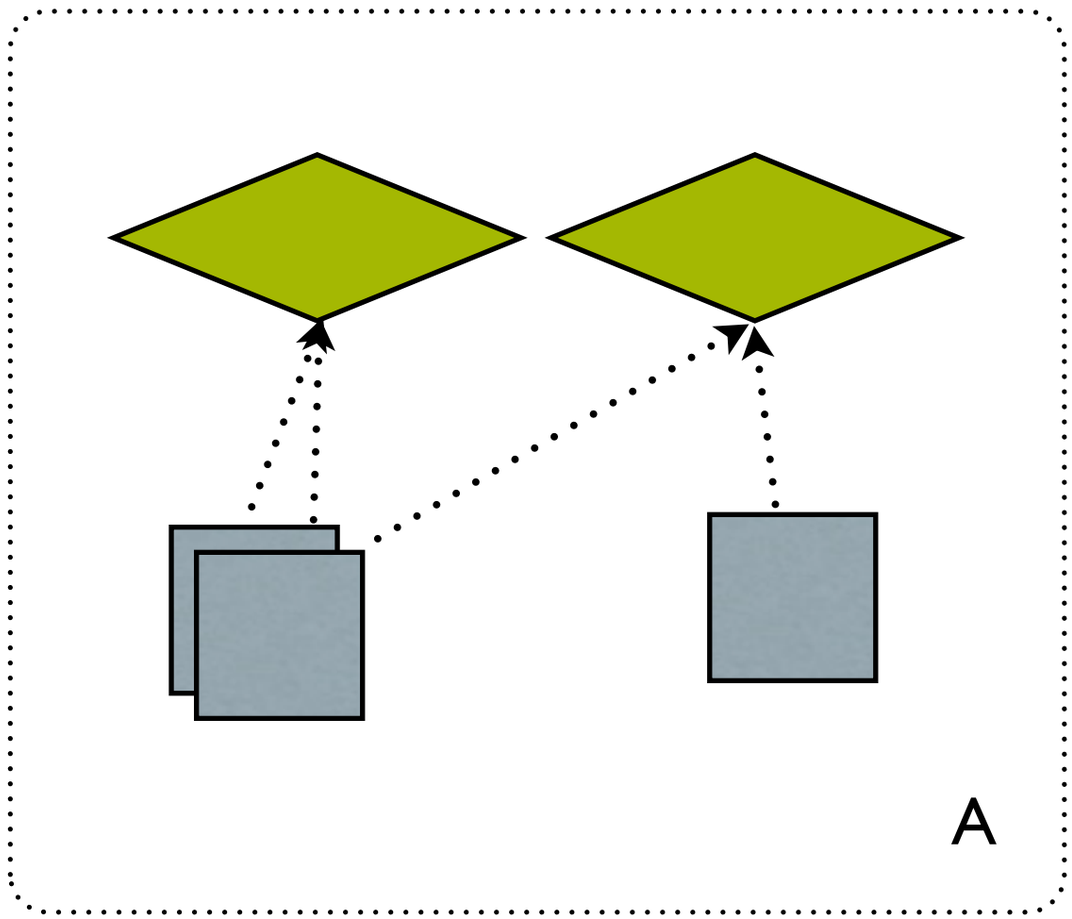
implement 0 or more protocols or interfaces

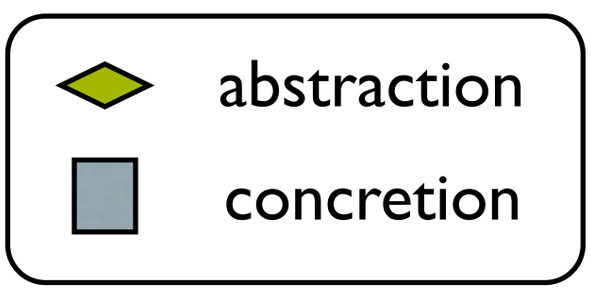
```
(let [x 42
      r (reify AProtocol
          (bar [this b] "reify bar")
          (baz [this ] (str "reify baz " x)))]
  (baz r))

=> "reify baz 42"
```

closes over environment like fn

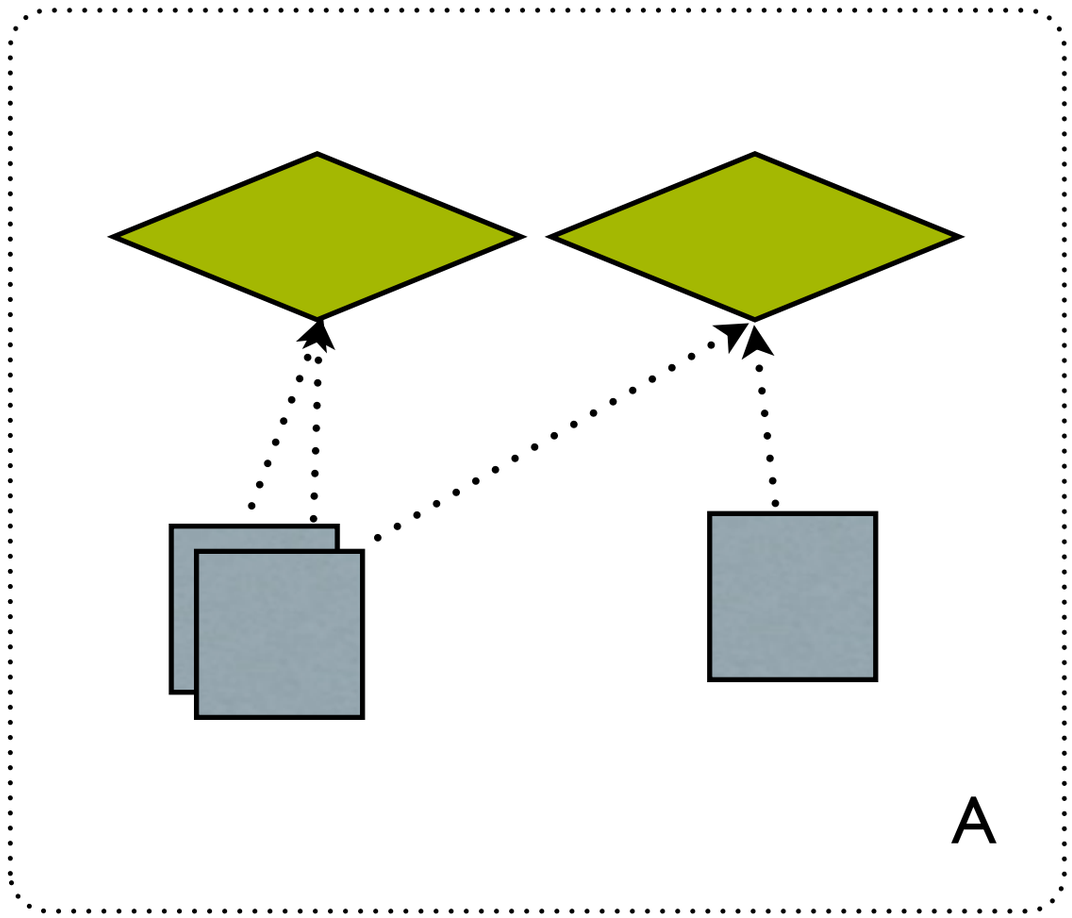
the expression problem

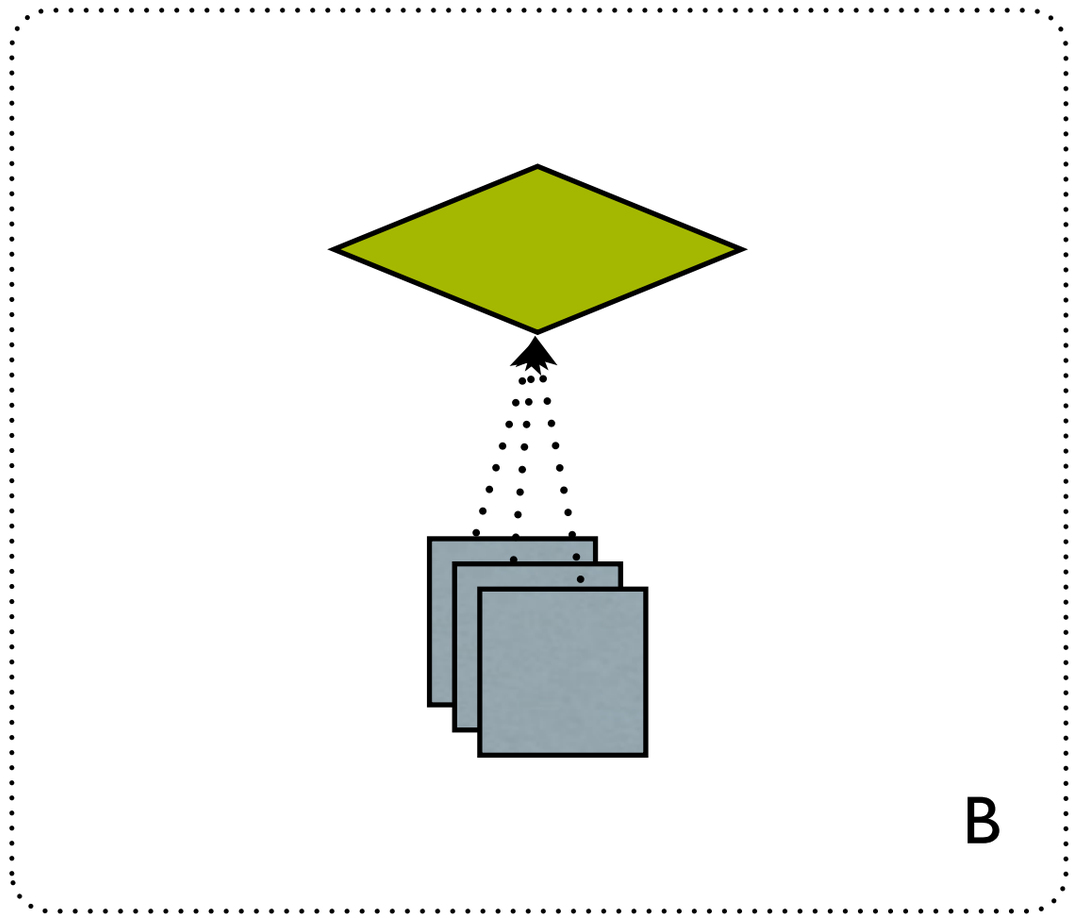
the expression problem

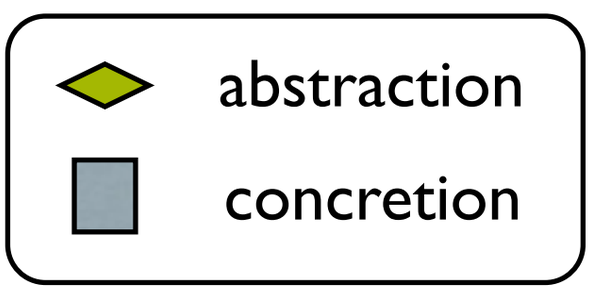




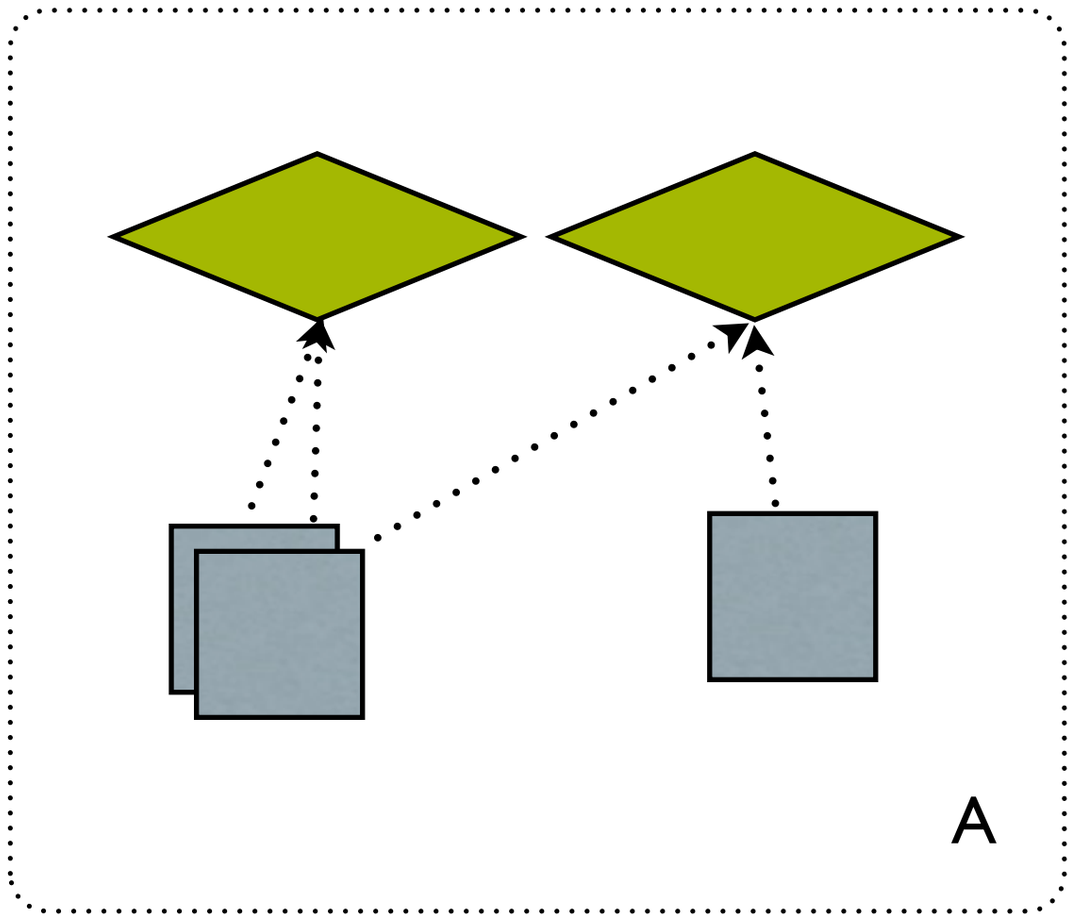
the expression problem

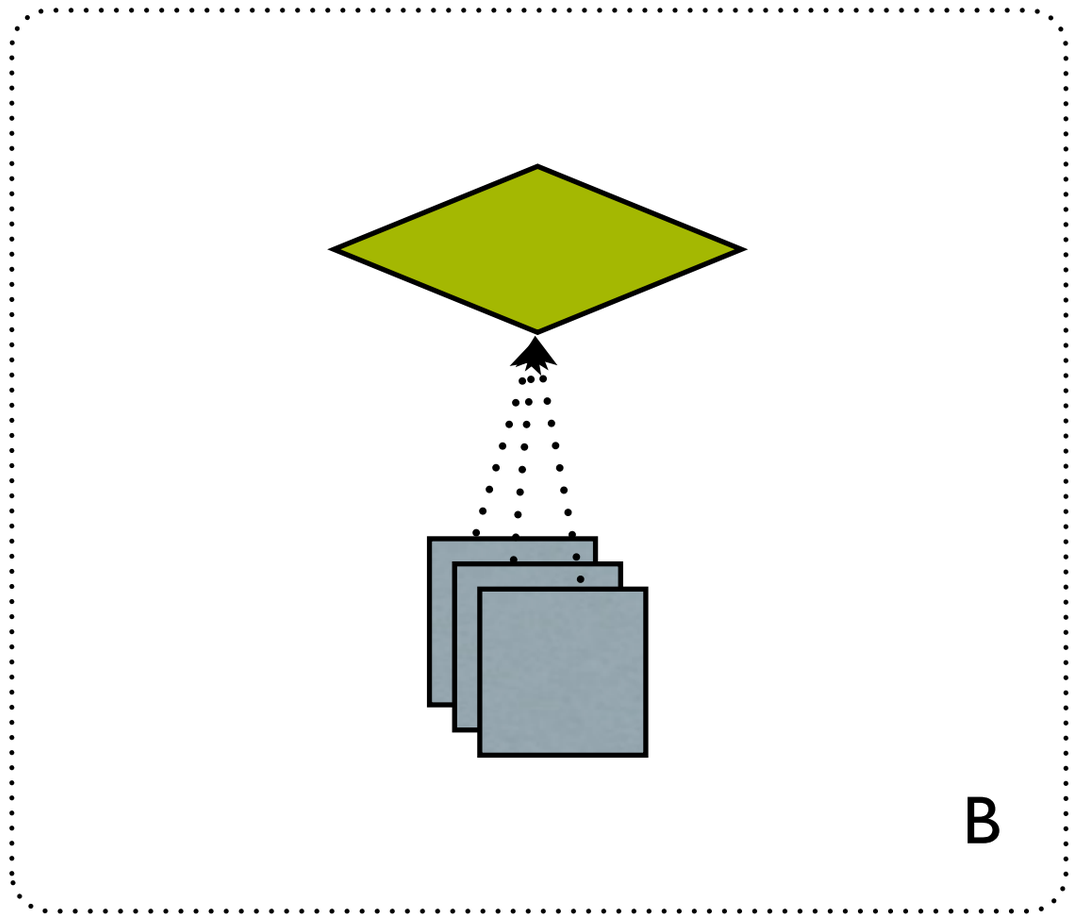


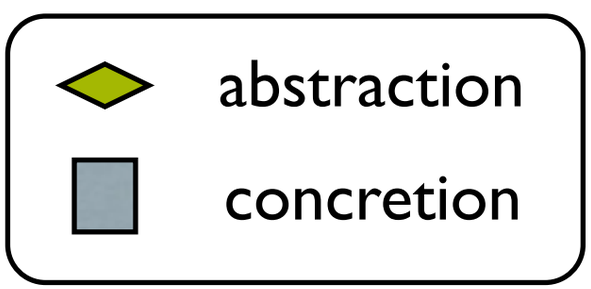




the expression problem

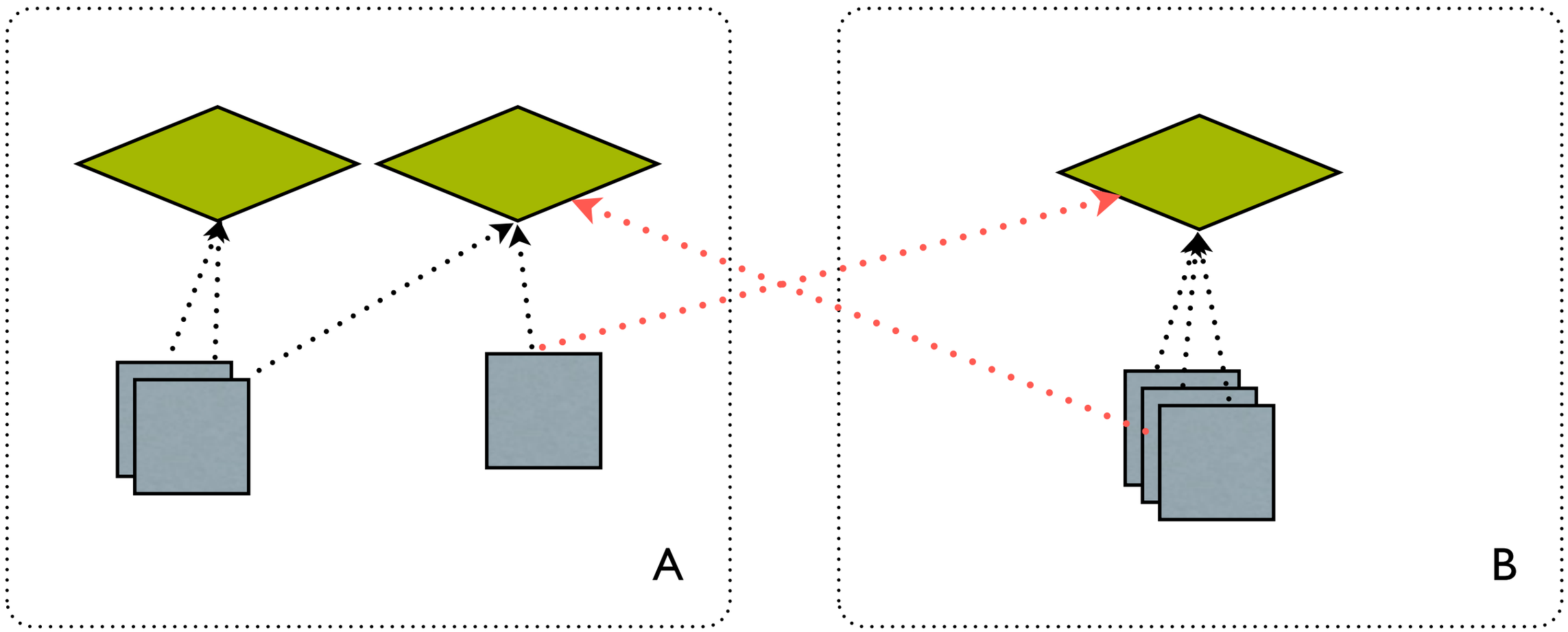


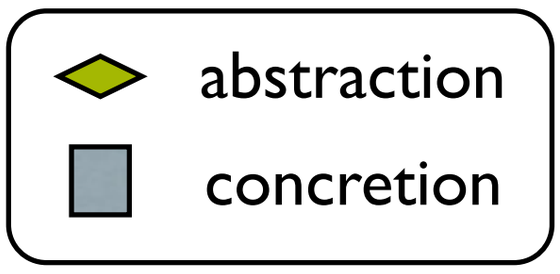




A should be able to work with B's abstractions, and vice versa, **without modification of the original code**

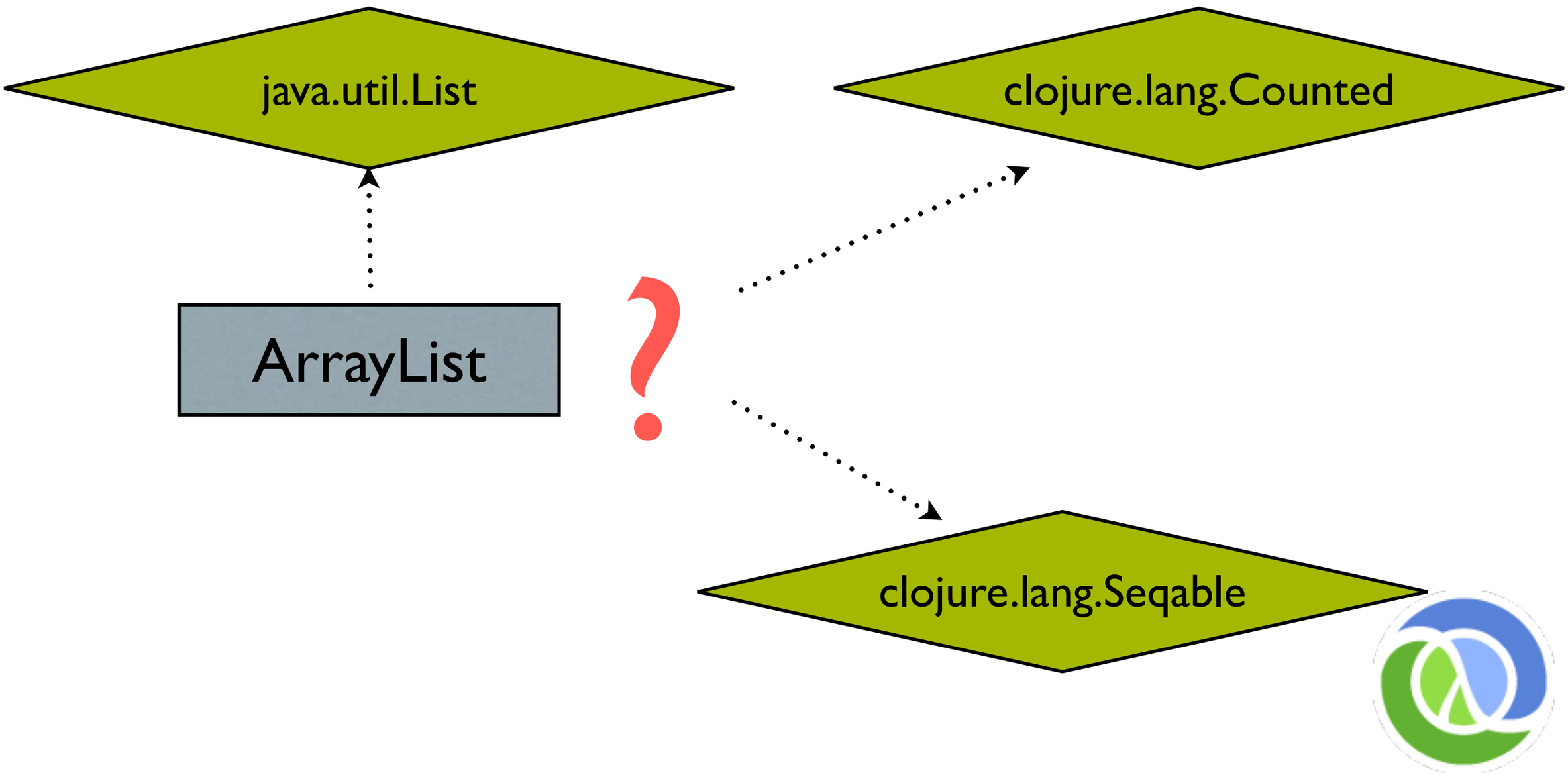
is this really a problem?



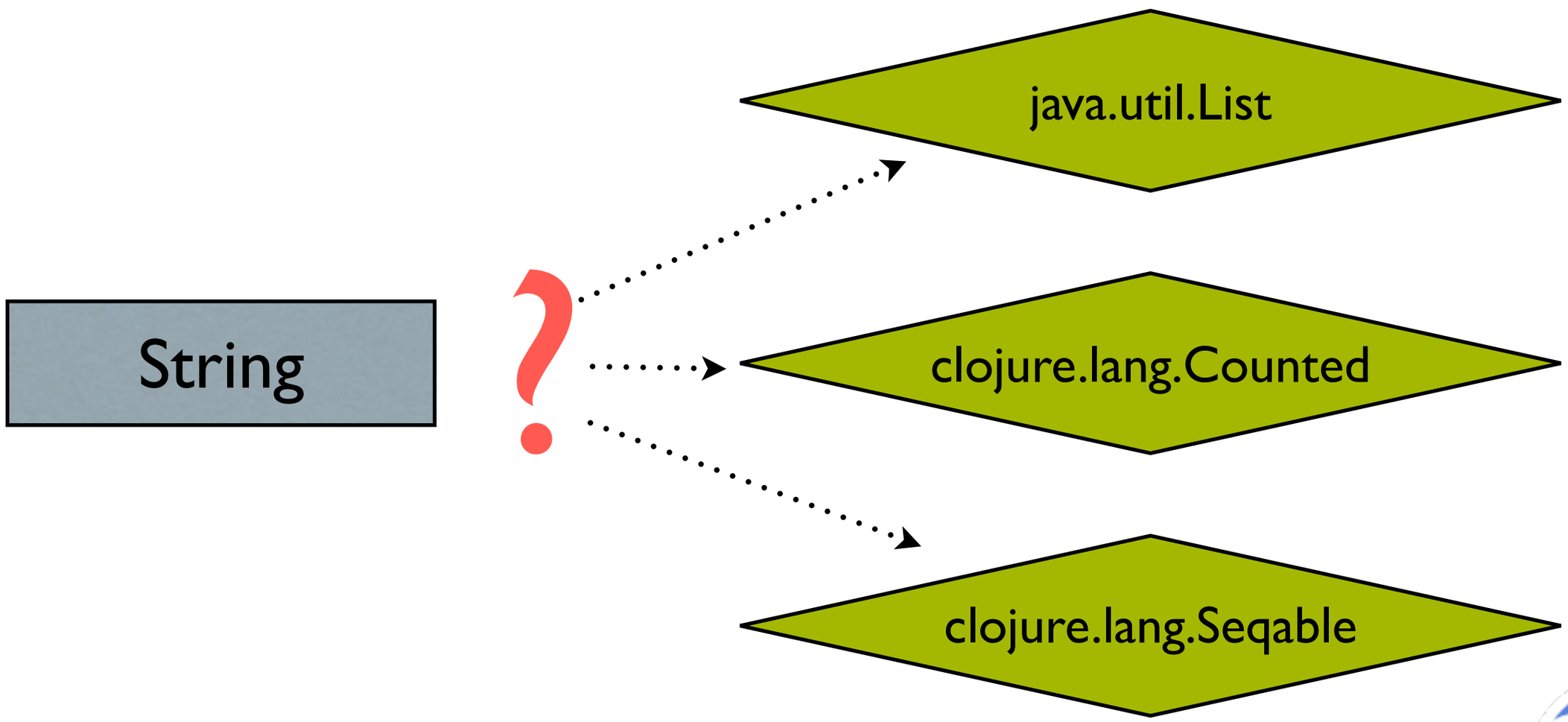


just use interfaces
for abstraction (??)

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

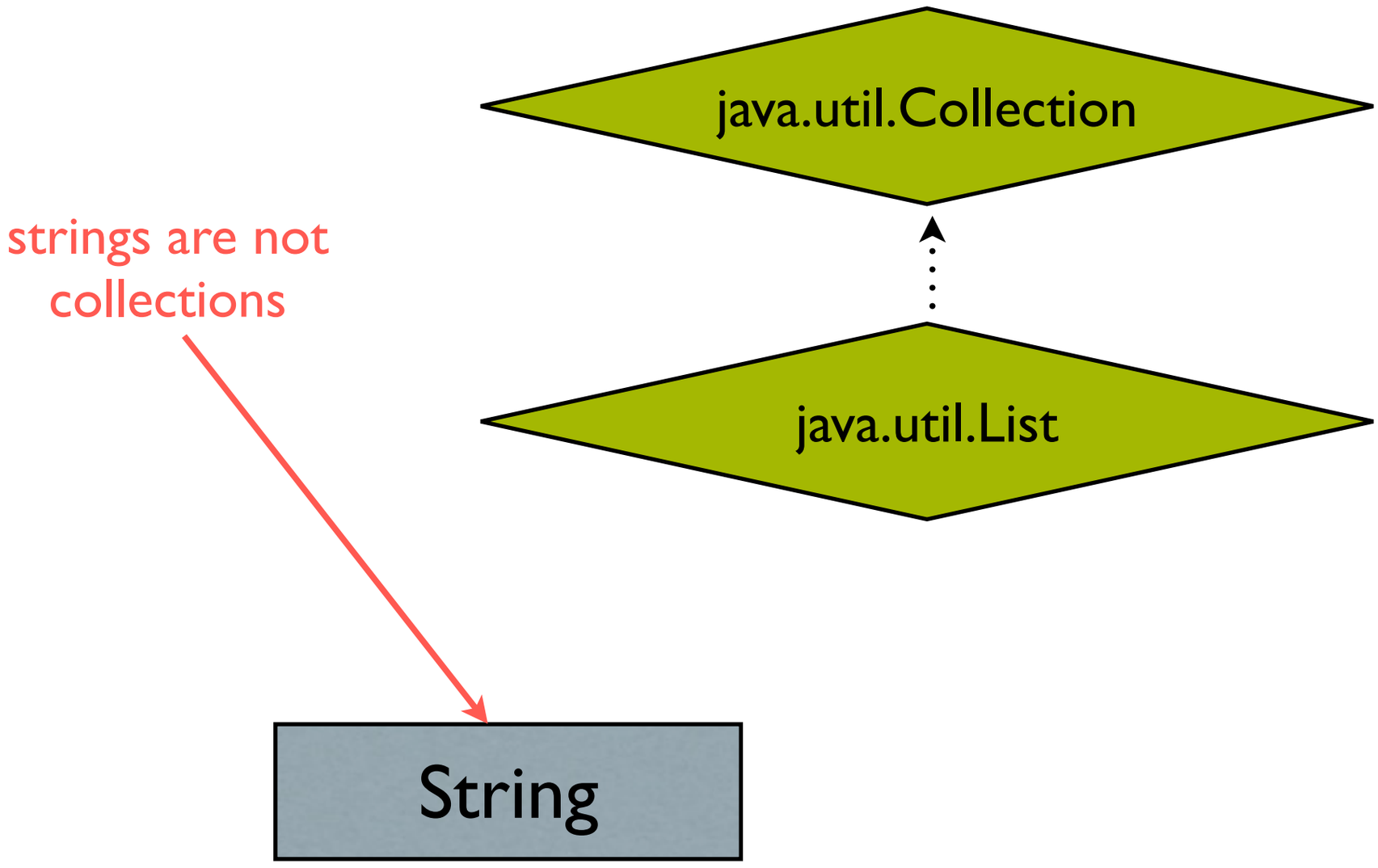
A is hard to change

we don't control A

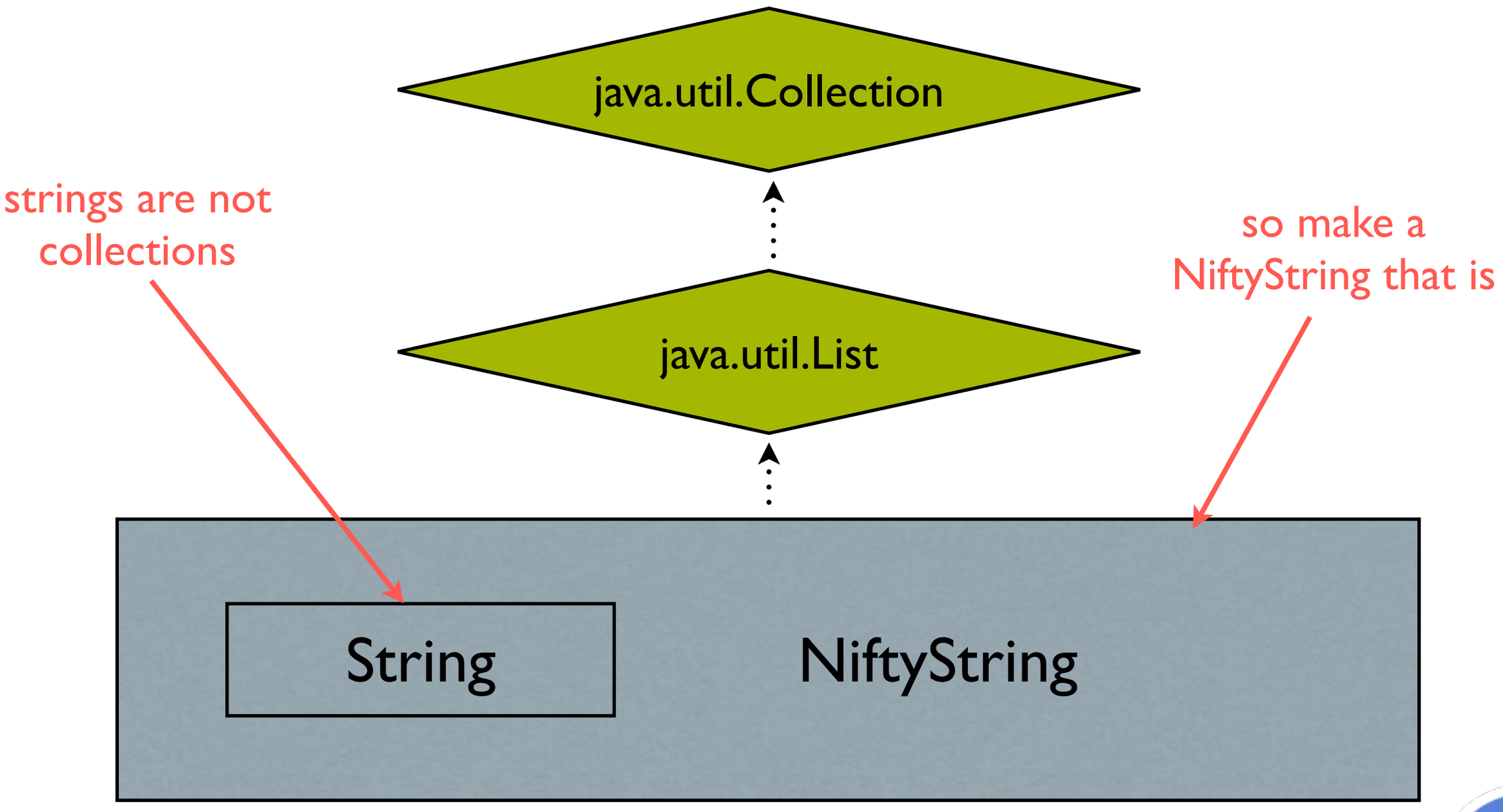
happens even **within** *a single lib*

some approaches to the expression problem

1. wrappers


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

1. 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

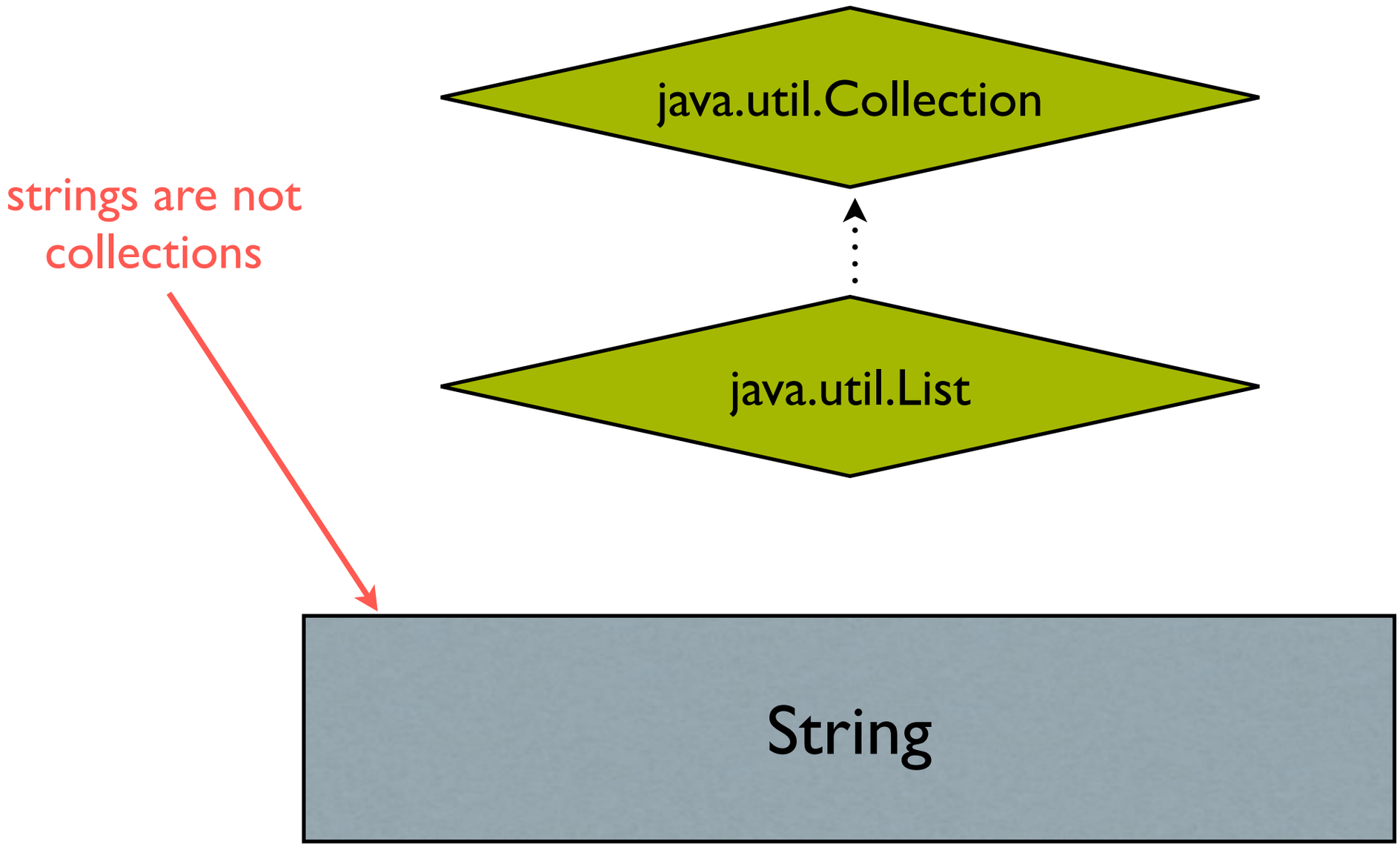
ruin equality

cause nonlocal defects

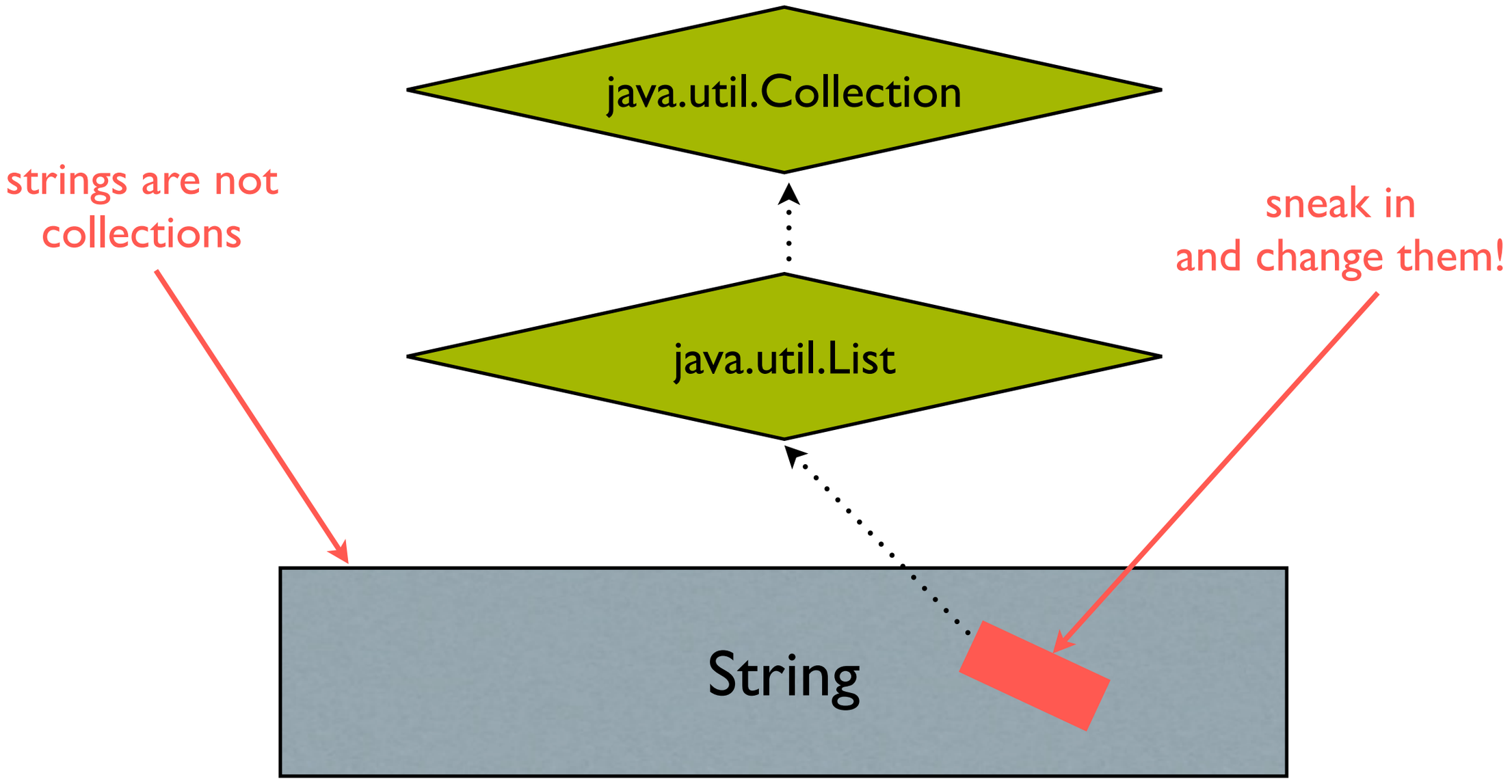
don't compose: AB + AC != ABC

have bad names

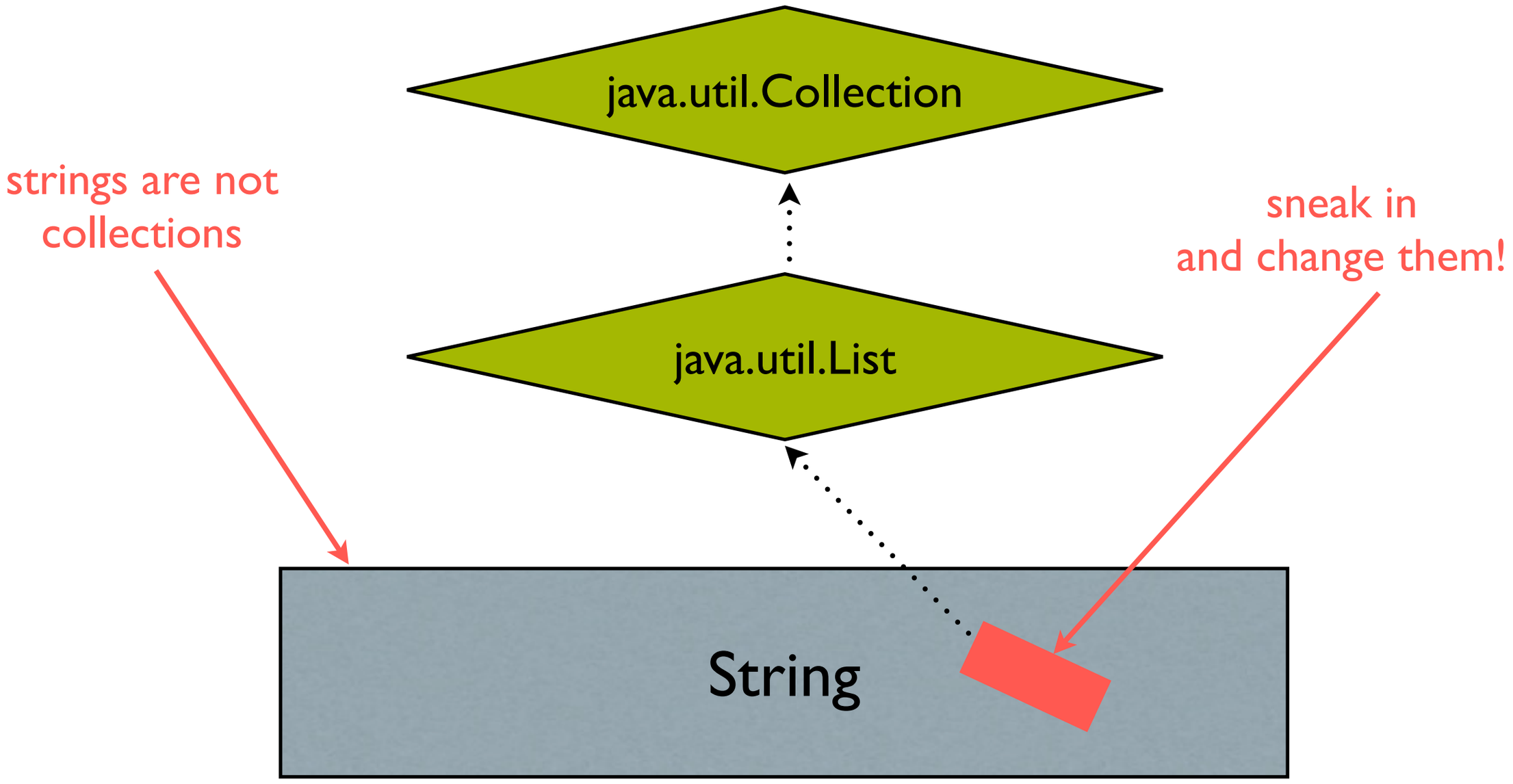
2. monkey patching



2. monkey patching



2. 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

3. generic functions (CLOS)

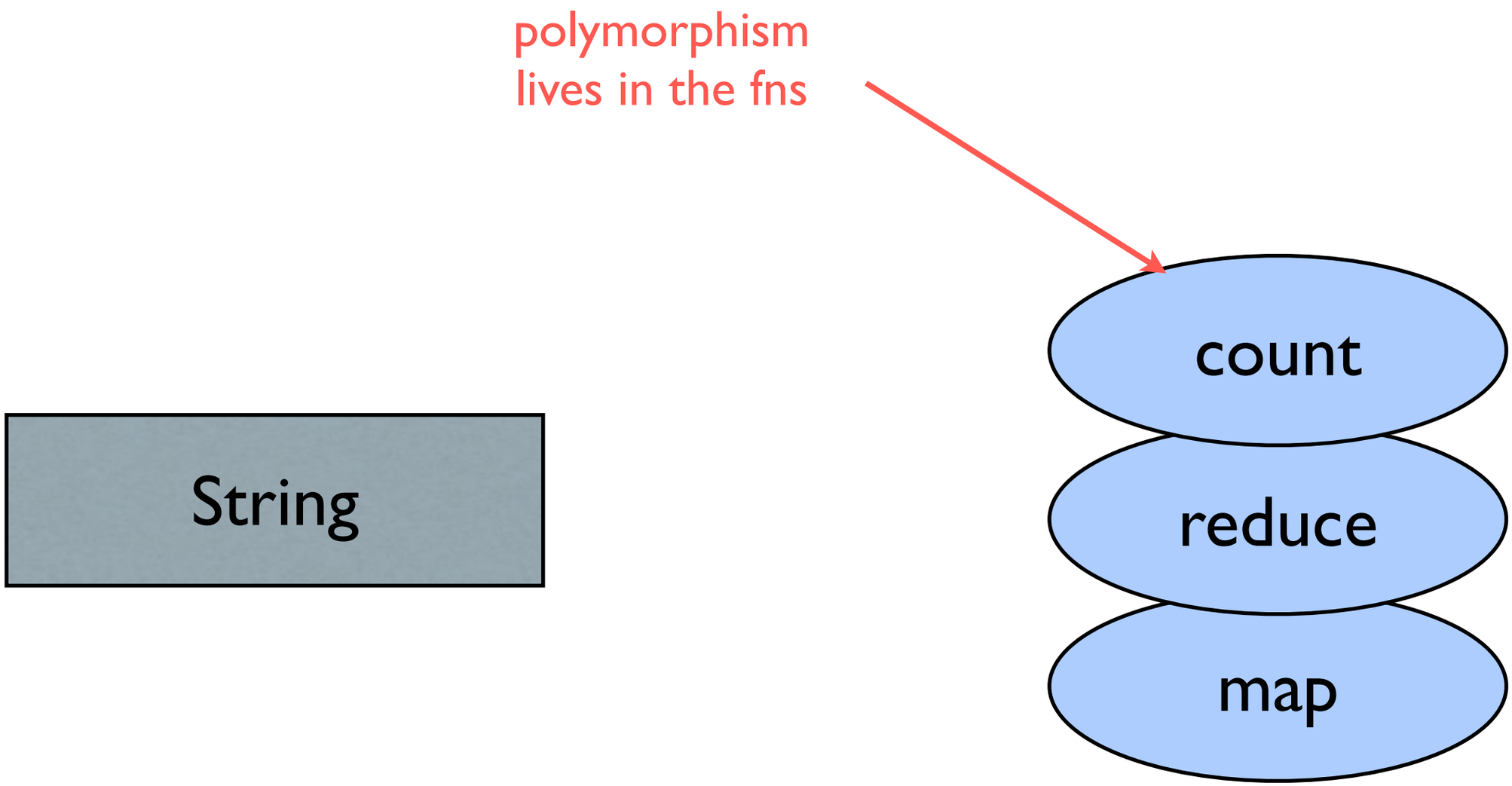
String

count

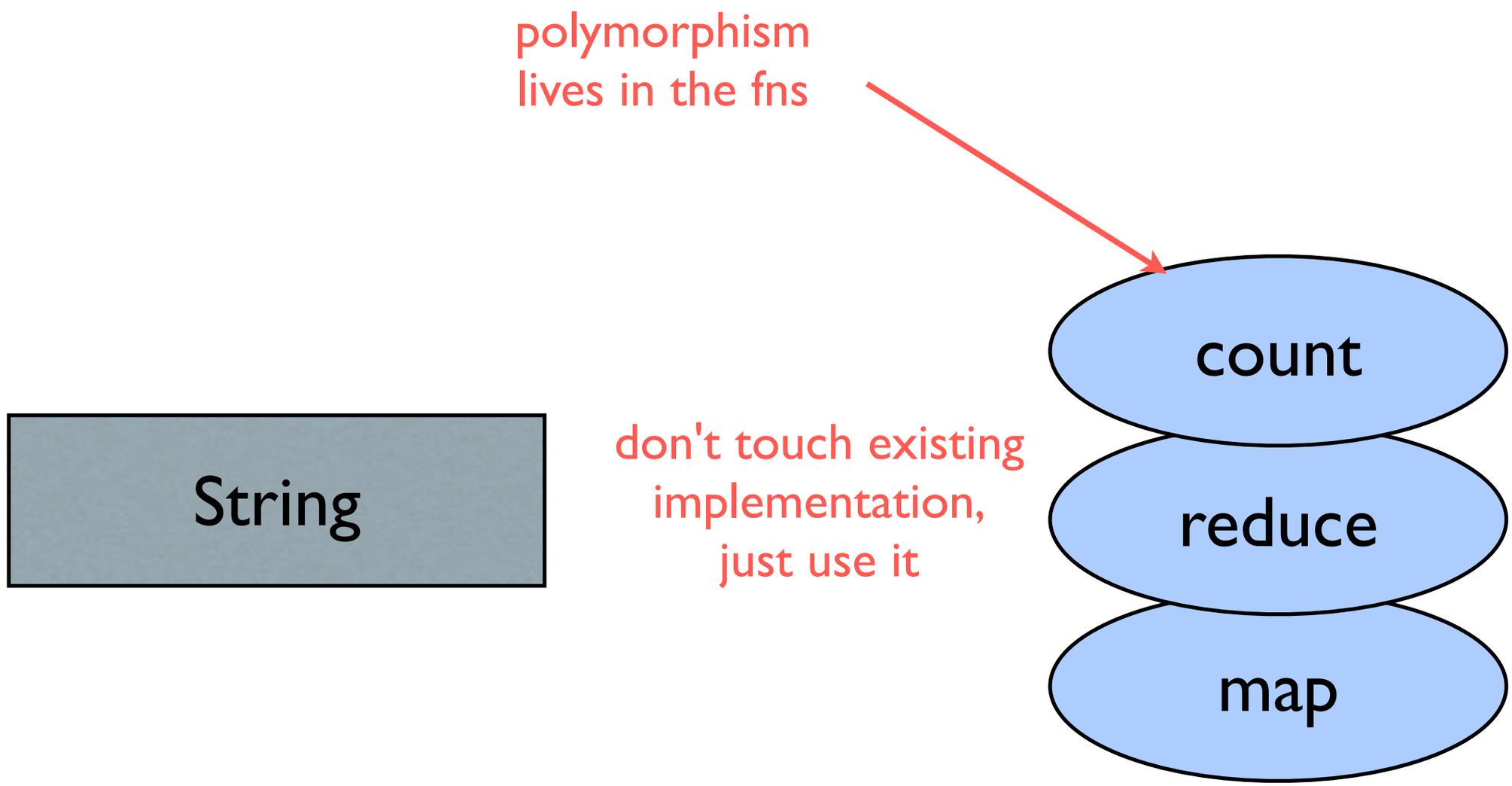
reduce

map

3. generic functions (CLOS)



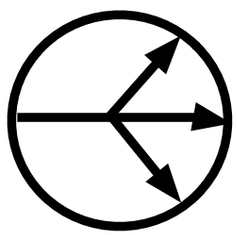
3. generic functions (CLOS)



polymorphism a la carte

values

types



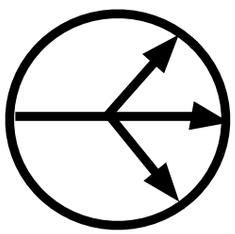
polymorphism

polymorphism a la carte

polymorphism in the fns, not the types

values

types



polymorphism

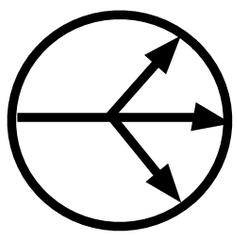
polymorphism a la carte

polymorphism in the fns, not the types

no "isa" requirement

values

types



polymorphism

polymorphism a la carte

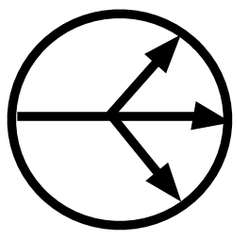
polymorphism in the fns, not the types

no "isa" requirement

no **type intrusion** necessary

values

types



polymorphism

polymorphism a la carte

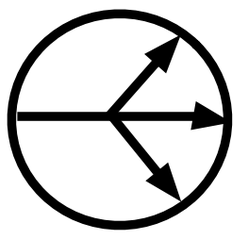
polymorphism in the fns, not the types

no "isa" requirement

no **type intrusion** necessary

values

types



polymorphism

protocols = generic functions

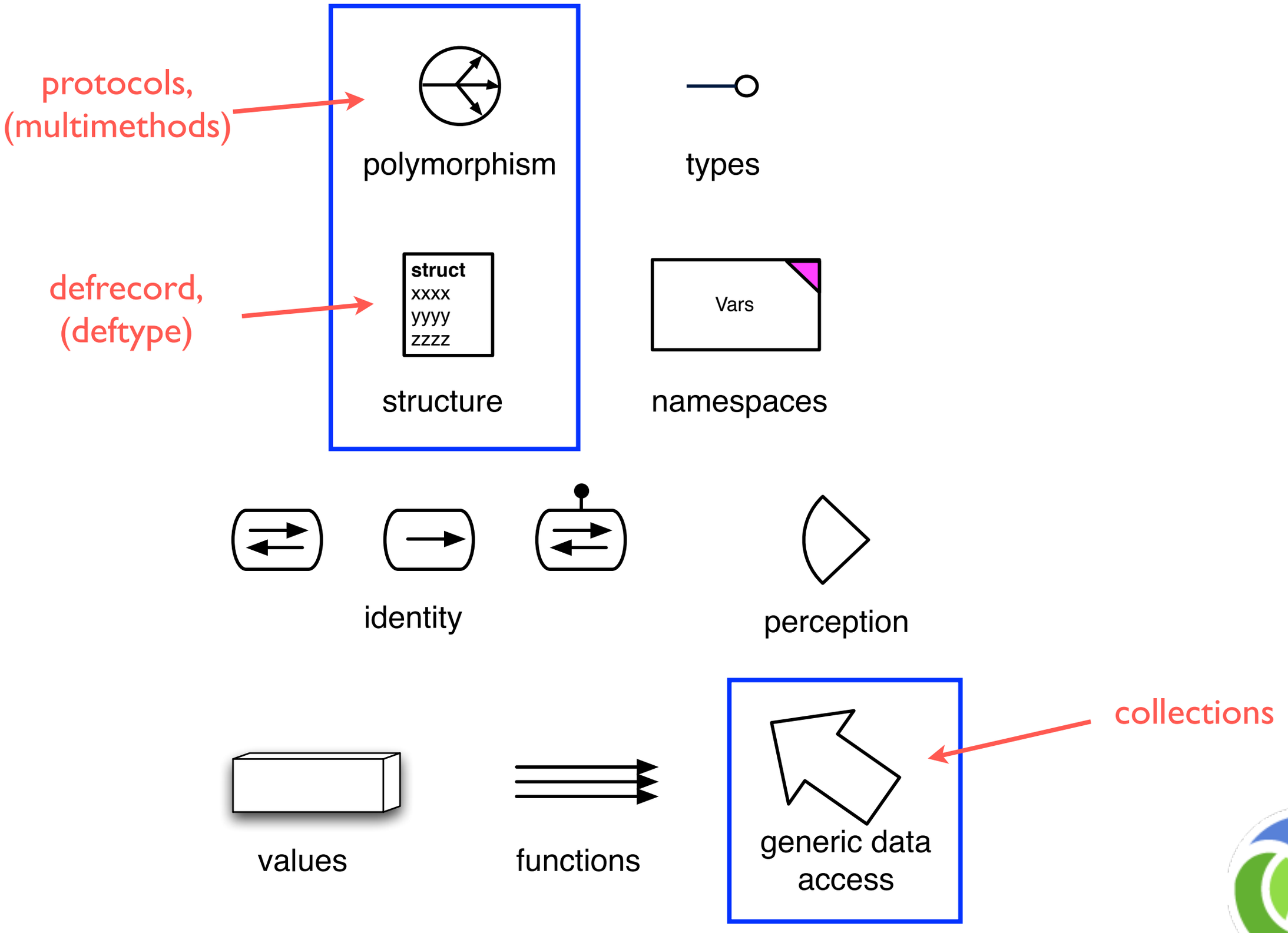
- arbitrary dispatch

+ speed

+ grouping

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

where are we?


protocols,
(multimethods)
polymorphism
types
defrecord,
(deftype)
struct
xxxx
yyyy
zzzz
Vars
structure
namespaces
identity
perception
values
functions
generic data
access
collections

for more

information

community

main Clojure site

http://clojure.org/

google group

http://groups.google.com/group/clojure

Clojure/core team

http://clojure.com

The conj

http://clojure-conj.org/

free resources

labrepl

http://github.com/relevance/labrepl

screencasts

http://clojure.blip.tv/

full disclojure screencasts

http://vimeo.com/channels/fulldisclojure

mark volkmann's Clojure article

http://java.ociweb.com/mark/clojure/article.html

thanks!

http://clojure.org

extra

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))
```

55

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

55

mean player

```
(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

play last winner or random

```
(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

remember how/if I won

deftype

57

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

60

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)

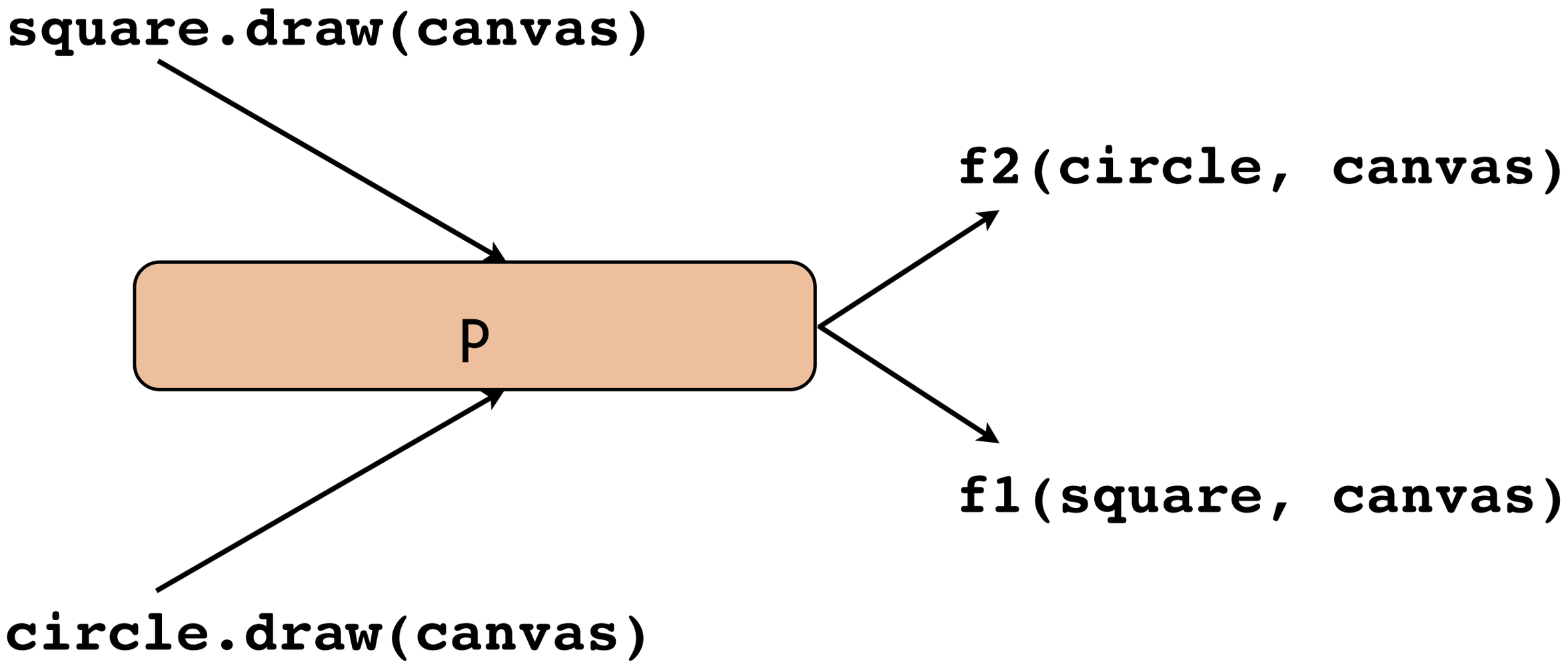
add java annotations (interop only)

deftype fields can be mutable (experts only)

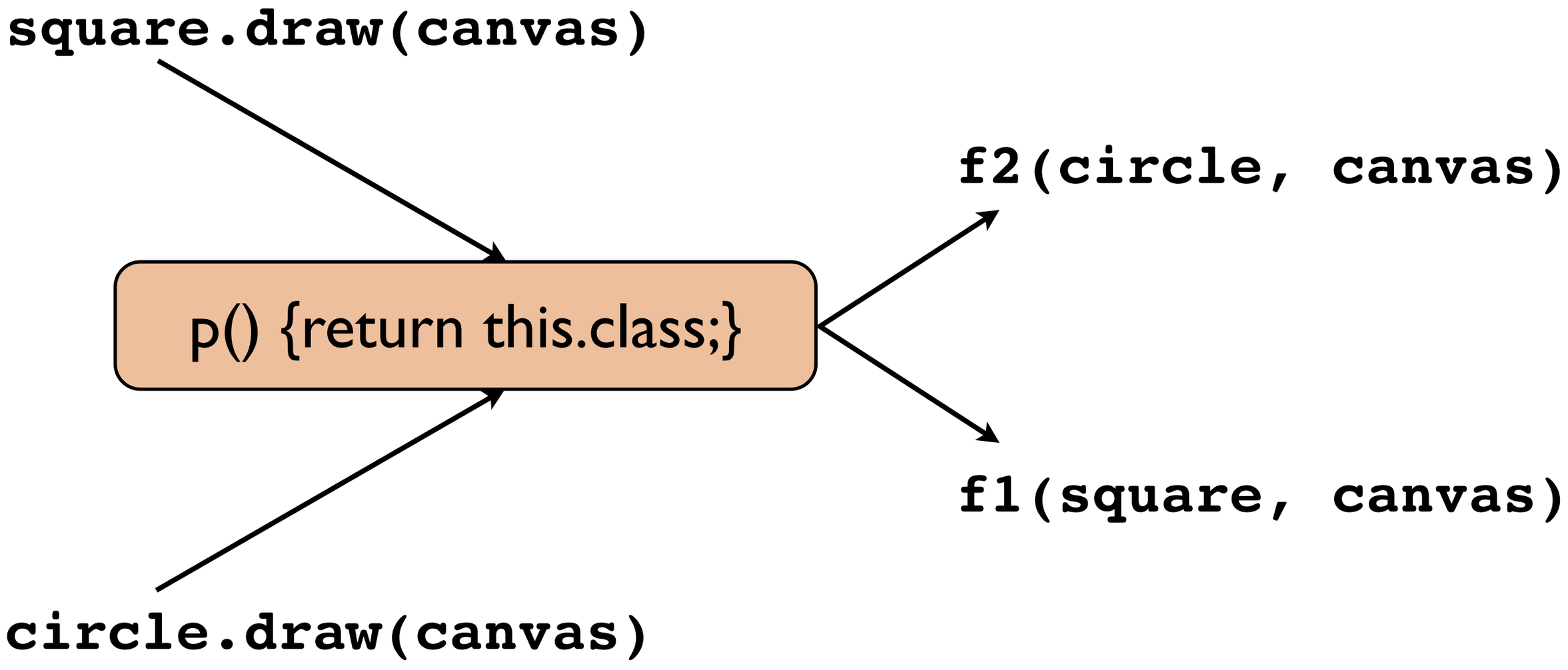
multimethods

63

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!

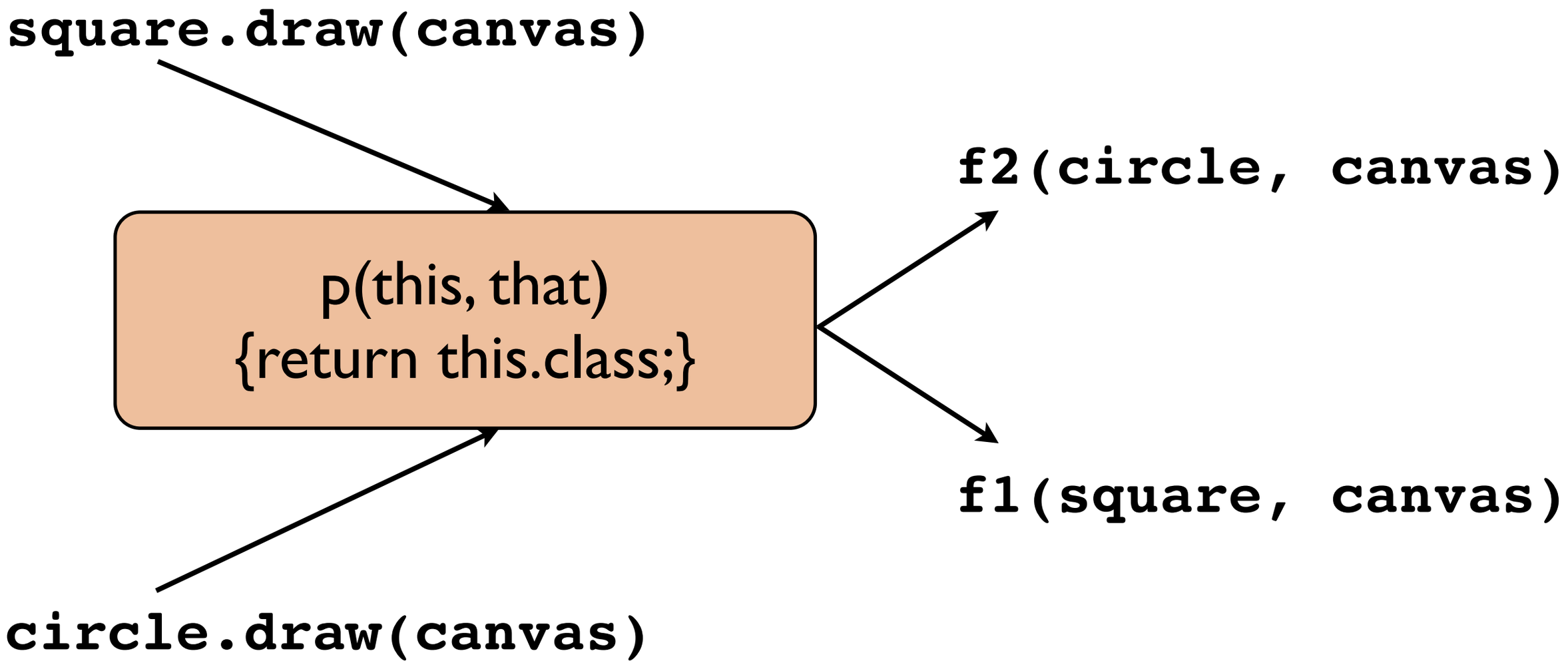
```
(blank? "blah")
-> No method in multimethod 'blank?'
     for dispatch value: class java.lang.String"
```

```
(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"
```

add methods anytime

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

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

most derived type wins

name inheritance

```
(defmulti interest-rate :type)
(defmethod interest-rate ::account
  [_] 0.00)
(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)

77

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