Clojure-Java Interop

Stuart Halloway
stu@clojure.com
@stuarthalloway

the plan

- syntax
- plumbing
  - Clojure calling Java
  - objects, Clojure style
  - Java calling Clojure
- design principles
  - simplicity
  - direct access
  - bonus example

syntax

atomic data types

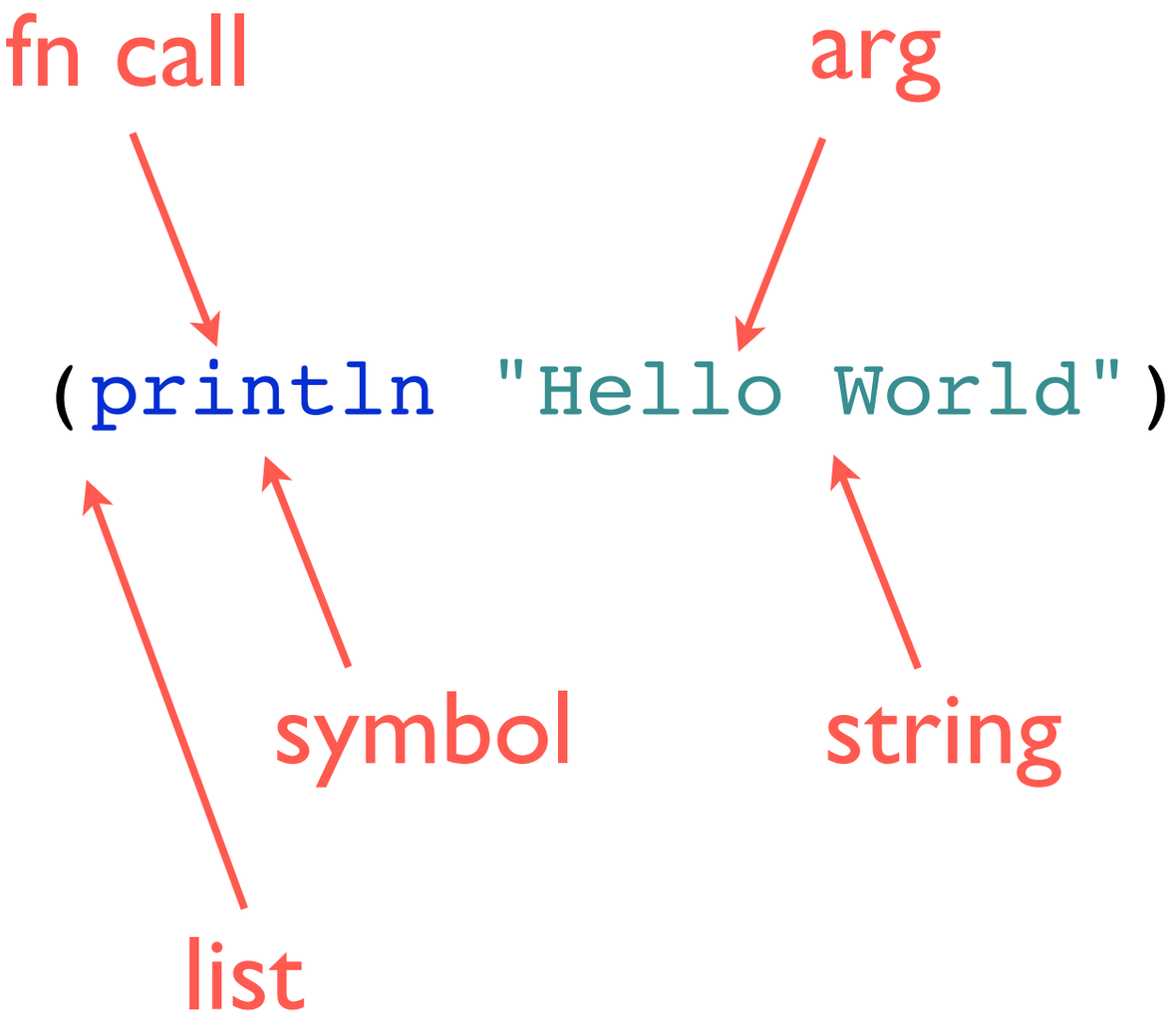
type	example	java equivalent
string	`"foo"`	String
character	`\f`	Character
regex	`#"fo*"`	Pattern
integer	`42`	Long
a.r. integer	`42N`	BigInteger
double	`3.14159`	Double
a.p. double	`3.14159M`	BigDecimal
boolean	`true`	Boolean
nil	`nil`	`null`
ratio	22/7	N/A
symbol	`foo, +`	N/A
keyword	`:foo, ::foo`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

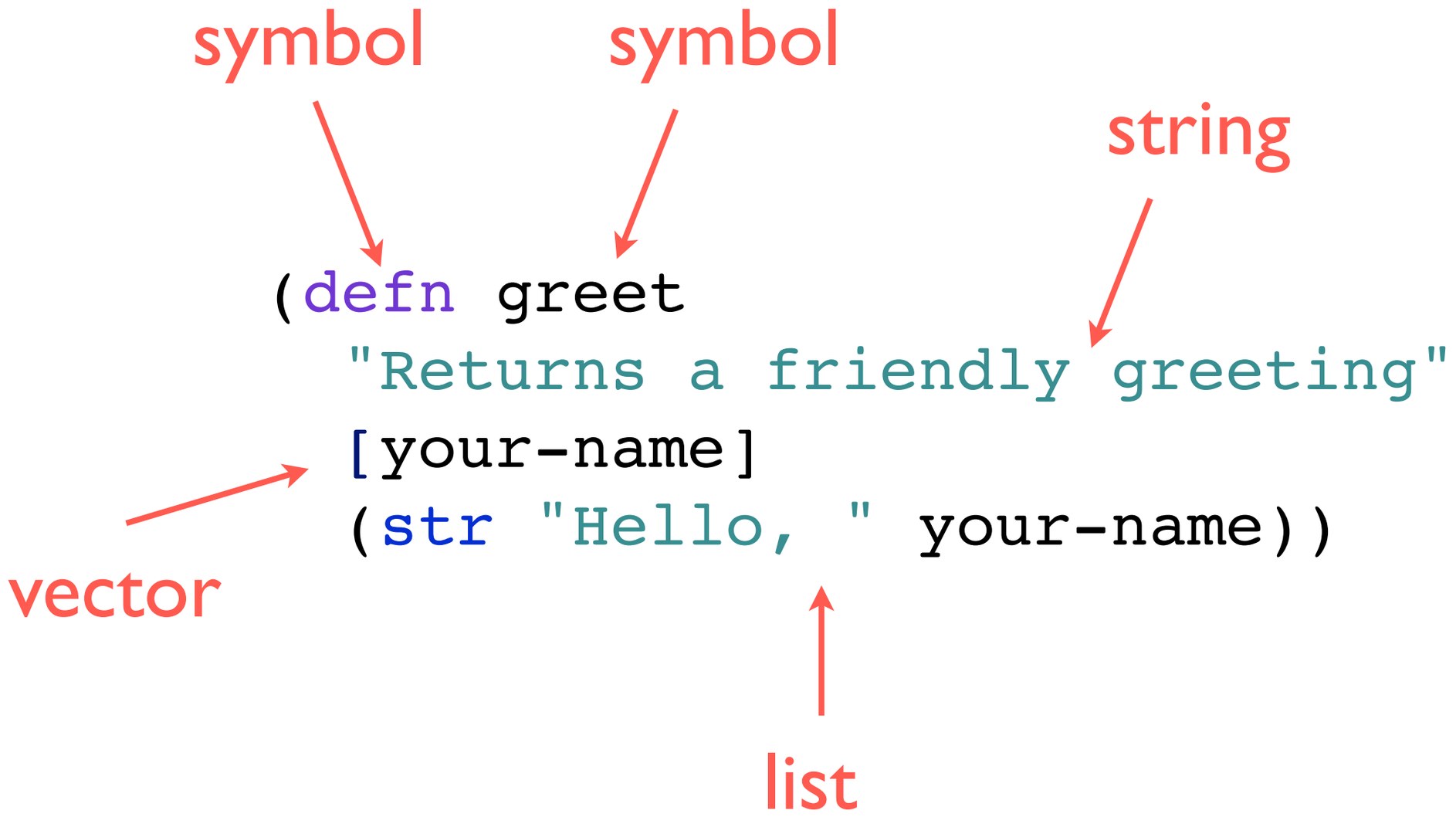
docstring

```
(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))
```

Clojure calling Java

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)`

doto

frame is implicit arg for subsequent forms

```
(doto
 (JFrame. "Foobar")
 (.add (proxy [JPanel] []))
 (.setSize 640 400)
 (.setVisible true))
```

parenthesis count

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

example:
refactor apache commons isBlank

initial implementation

```
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);
  }
}
```

Clojure!

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

objects,

Clojure style

records

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

generic data access

```
(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}
```

from maps...

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

(:lname stu)
=> "Halloway"

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

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

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

data-oriented

keyword access

nested access

update

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)))
```

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

defprotocol

```
(defprotocol Coercions
  "Coerce between various 'resource-namish' things."
  (as-file [x] "Coerce argument to a file.")
  (as-url [x] "Coerce argument to a URL."))
```

named set of generic functions

polymorphic on type of first argument

defines fns in same namespace as protocol

extending inline

```
(defrecord Name [n]
  Coercions
  (as-file [_] (File. n)))

(def n (Name. "johndoe"))

(as-file n)
-> #<File johndoe>
```

not a good design choice in
this case, better approach
follows on next slide...

extend protocol to types...

```
(as-file "data.txt")
-> java.lang.IllegalArgumentException
```

```
(extend-protocol Coercions
  nil
  (as-file [_] nil)
  (as-url [_] nil)

  String
  (as-file [s] (File. s))
  (as-url [s] (URL. s))
```

```
(as-file "data.txt")
-> #<File data.txt>
```

...or type to protocols

```
(extend-type String
  Coercions
  (as-file [s] (File. s))
  (as-url [s] (URL. s))

  EqualityPartition
  (equality-partition [x] :atom)
```

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

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

Java calling Clojure

compiling

```
(compile 'examples.clojure)
```

interactive

```
(defproject clojure.examples "0.0.1"
  :aot [examples.contacts]
  #_"...more maveny stuff")
```

from a build tool/IDE
(e.g. maven, leiningen, gradle, Eclipse, NetBeans, IDEA, emacs, vim)

Clojure data from Java

```
Address addr = new Address("200 N Mangum",
                           "Durham", "NC", 27701);
Person person = new Person("Stuart", "Halloway",
                           addr);

System.out.println("As person " + person.fname);
```

as objects

```
System.out.println("As map");
for (Iterator it = person.keySet().iterator();
     it.hasNext();) {
    Object key = it.next();
    System.out.println(key + ": " + person.get(key));
}
```

or as
generic data

gen-class

class name

```
(ns examples.tasklist
  (:gen-class
   :extends org.xml.sax.helpers.DefaultHandler
   :init init
   :state state)
```

optional state

optional "constructor"

optional bases

mapping gen-class methods

constructor

```
(defn -init []
  [[] (atom [])])

(defn -startElement
  [this uri local qname atts]
  (when (= qname "target")
    (swap! (.state this)
           conj (.getValue atts "name"))))

(defn -main [& args]
  (doseq [arg args]
    (println (task-list arg))))
```

inherited method

static main

gen-class: for interop *only*

gen-class meets Java in ugly places

prefer defprotocol, defrecord, reify, deftype

previous slides were examples only

xml doesn't have to be that tedious!

xml done better

```
(defn tasklist
  [resource]
  (-> (read-html resource)
      (select [:target])
      (attr :name)))
```

clojure data

declarative, standards based query
(courtesy cgrand's enlive)

where are we

Clojure can call Java

Java can call Clojure

Clojure makes things better

so what?

simplicity

define

simple

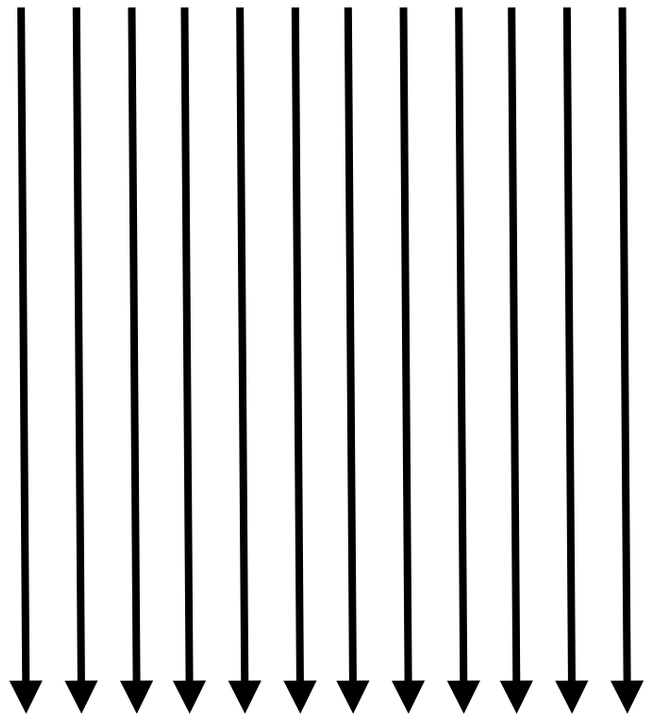
keeping

count?

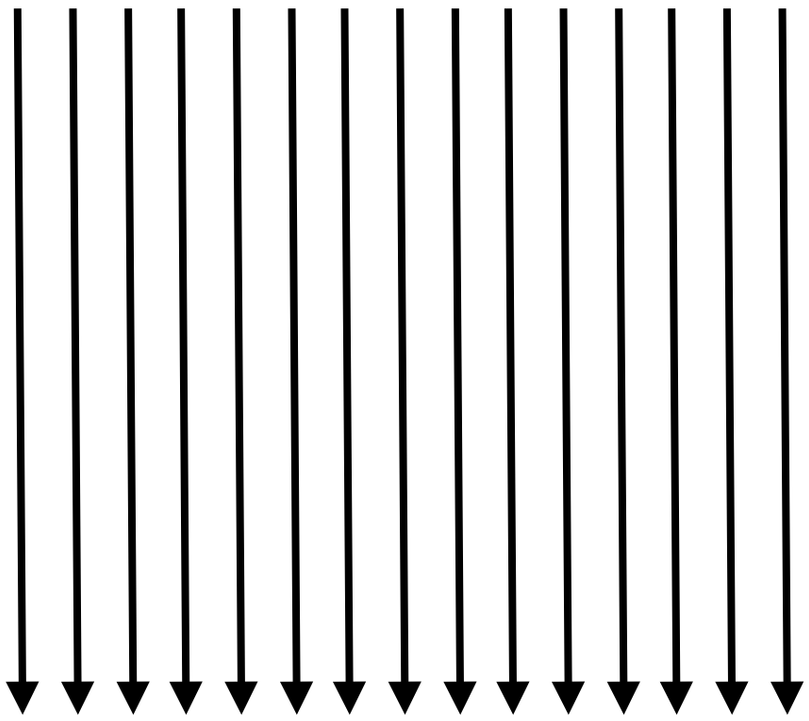
“if I wanted to deploy a simple web app ... how many lines of text do I have to deal with? Most importantly, how many of those are program text, and how many are framework boilerplate? Finally, how many tools do I have to use to get it deployed?”

simple things should be simple thread
Clojure google group

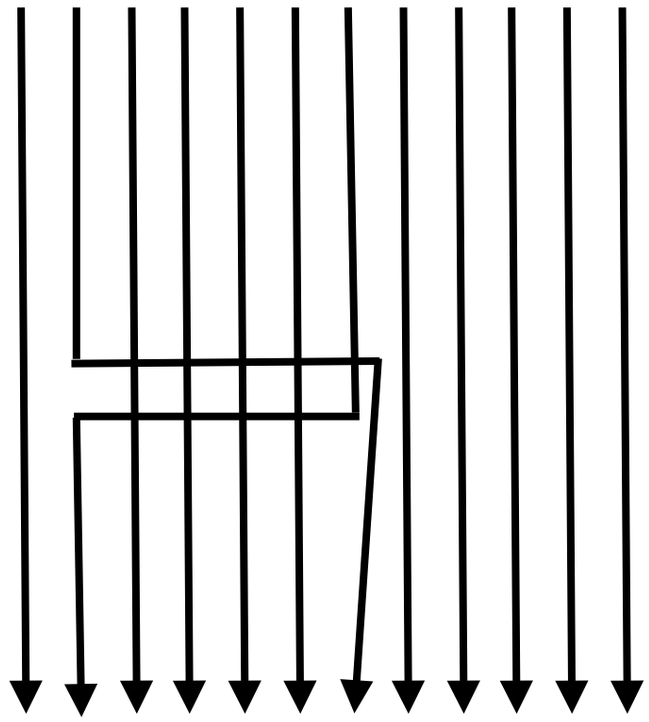
simple



still simple



complex



beginner-friendly?

"The most important gauge of any programming language is how easily a novice programmer can maintain a significant program in any problem domain."

http://www.oreillynet.com/onlamp/blog/2006/03/the_worlds_most_maintainable_p.html

"Simplicity: Should be fairly simple to read, learn and understand. Python is a good example. PHP is a great example (at a simpler class of problems). C++ and Scala not good examples."

http://blog.dhananjaynene.com/2010/08/my-ideal-programming-language/

http://www.aues21.dsl.pipex.com/orchestralinstruments.htm

familiar

syntax?

"Constructs that are natural to humans not mathematics : This is actually a sub point to Simplicity. The constructs should be consistent with the normal average non mathematically trained brains."

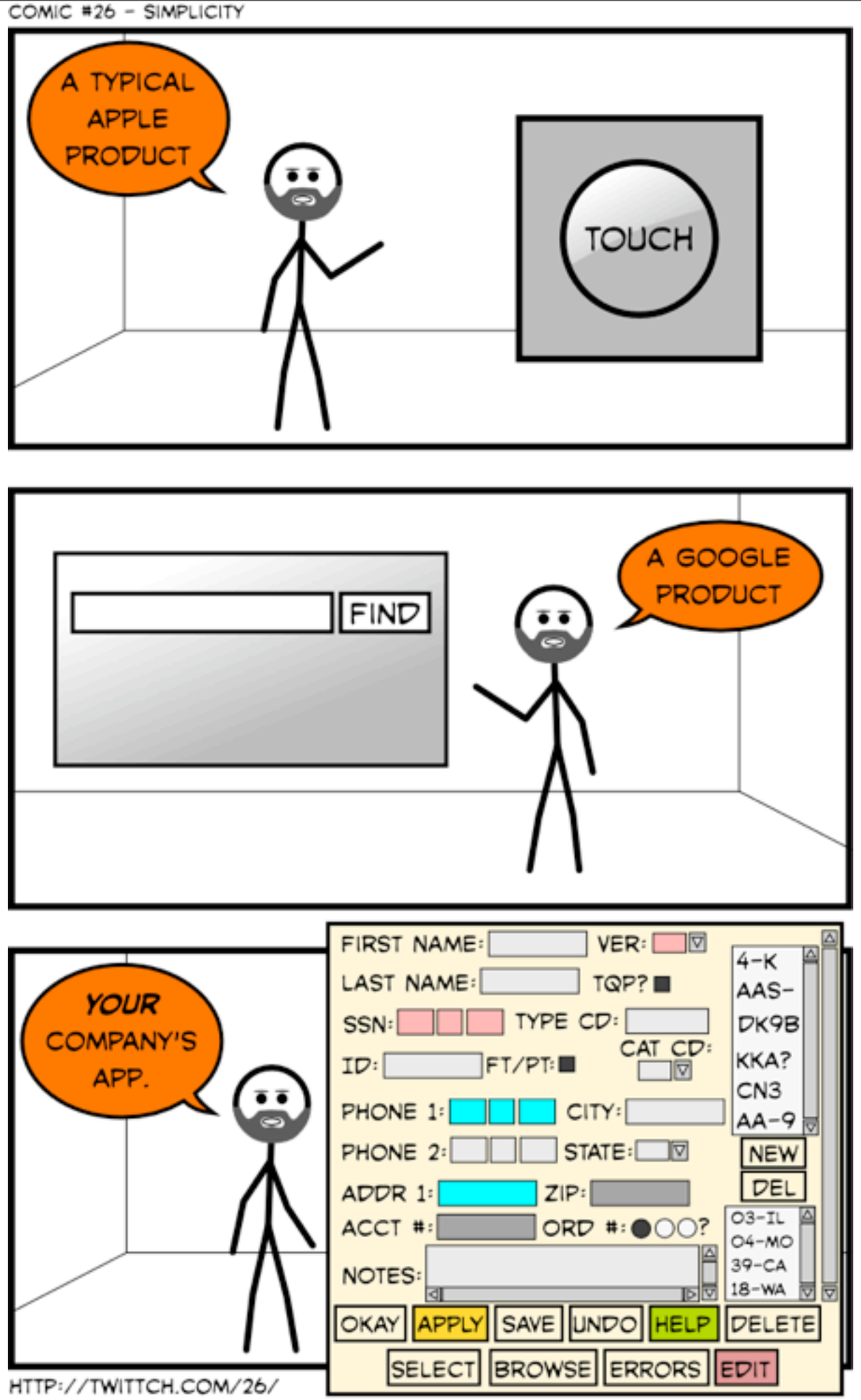
http://blog.dhananjaynene.com/2010/08/my-ideal-programming-language/

minimal?

"If the semantic model of a language is clear - if its rules are simple and precise - the language will be more productive to use. Less is more."

http://java.sun.com/developer/technicalArticles/JavaLP/jpl_proposals.html

uncomplicated?



http://twittch.com/26/

valuable, but not simple

convenient

newb-friendly

easy

familiar syntax

minimal

uncomplicated

hopelessly

subjective?

"I'm coming to the conclusion that one man's simple is another man's complex. Fowler evidently finds it easier to read many small methods than a few larger ones; I find the opposite."

http://reprog.wordpress.com/2010/03/28/what-is-simplicity-in-programming/

don’t give up

simple has an objective definition

simplicity informs design

simplicity is fundamental

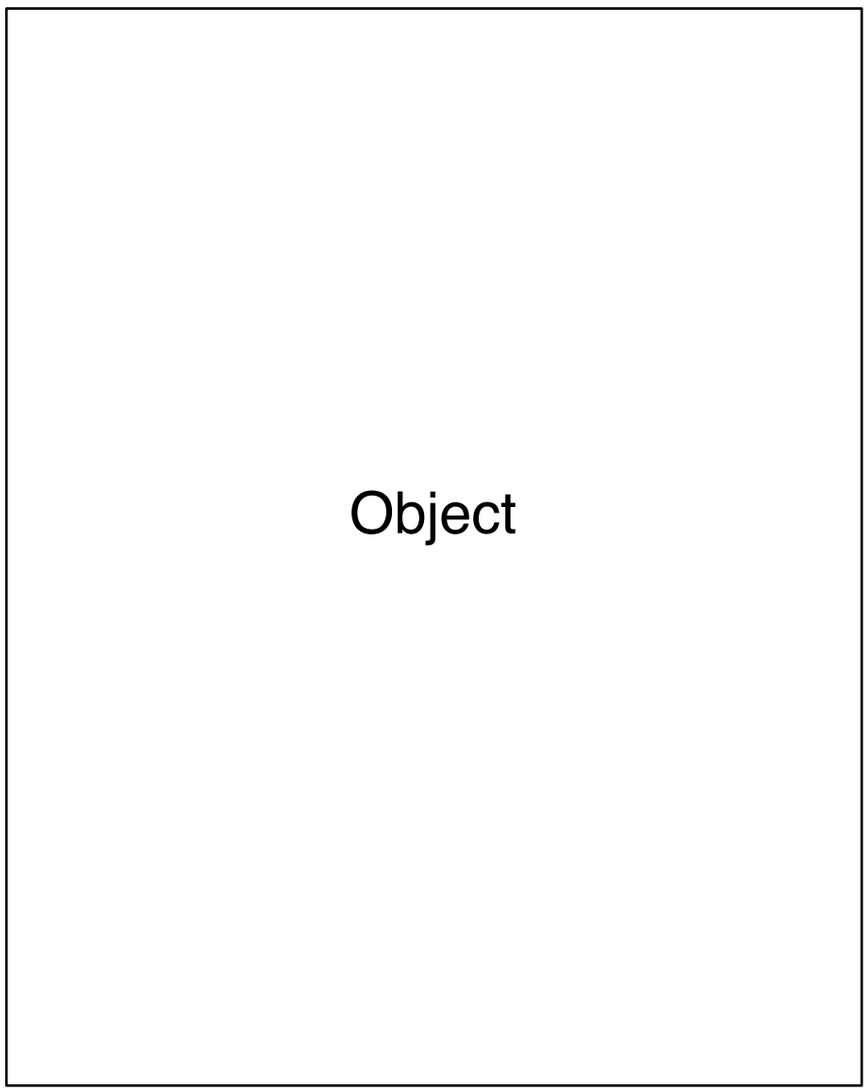
simple:

not compound

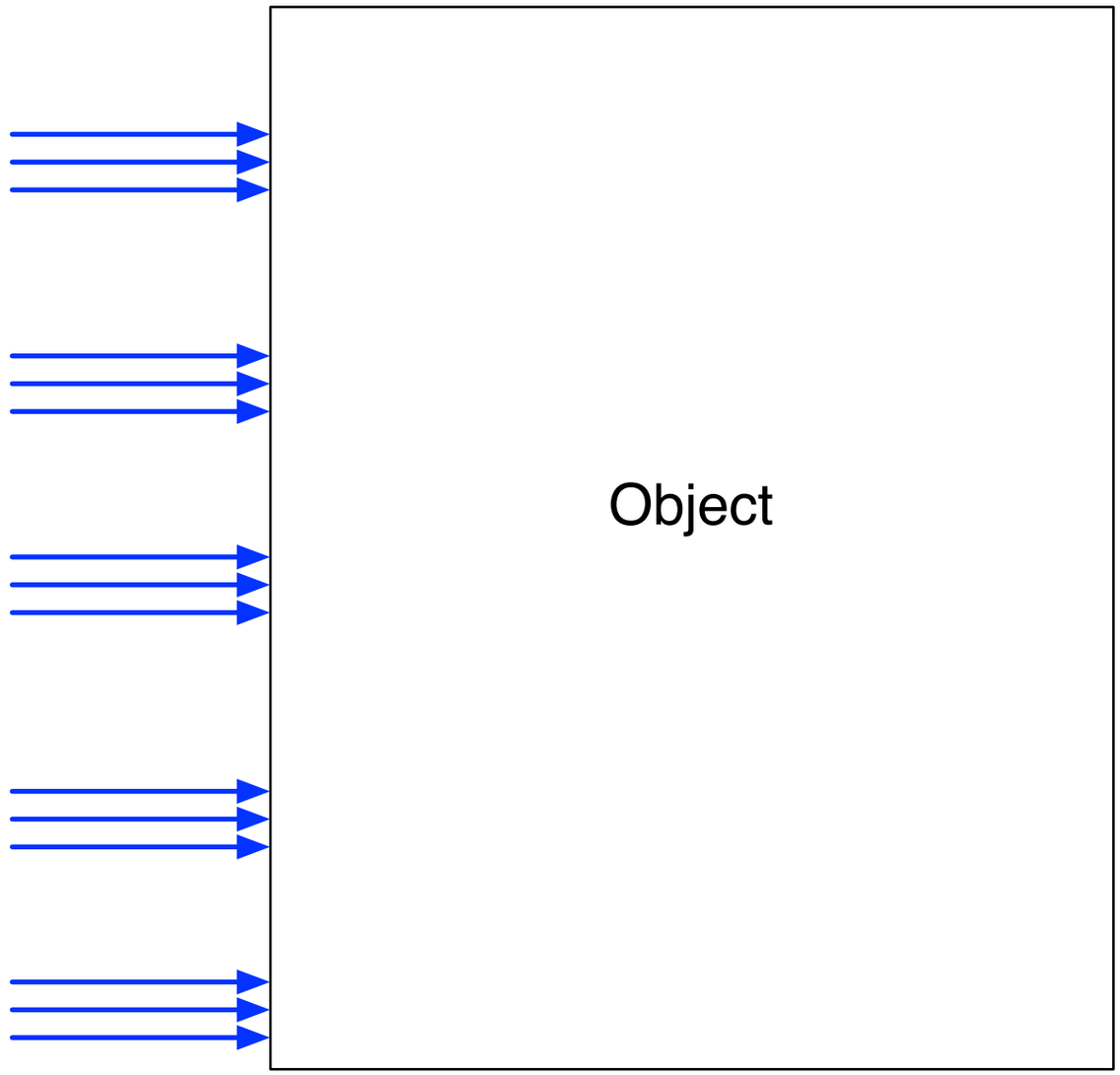
Java delivers

compounds

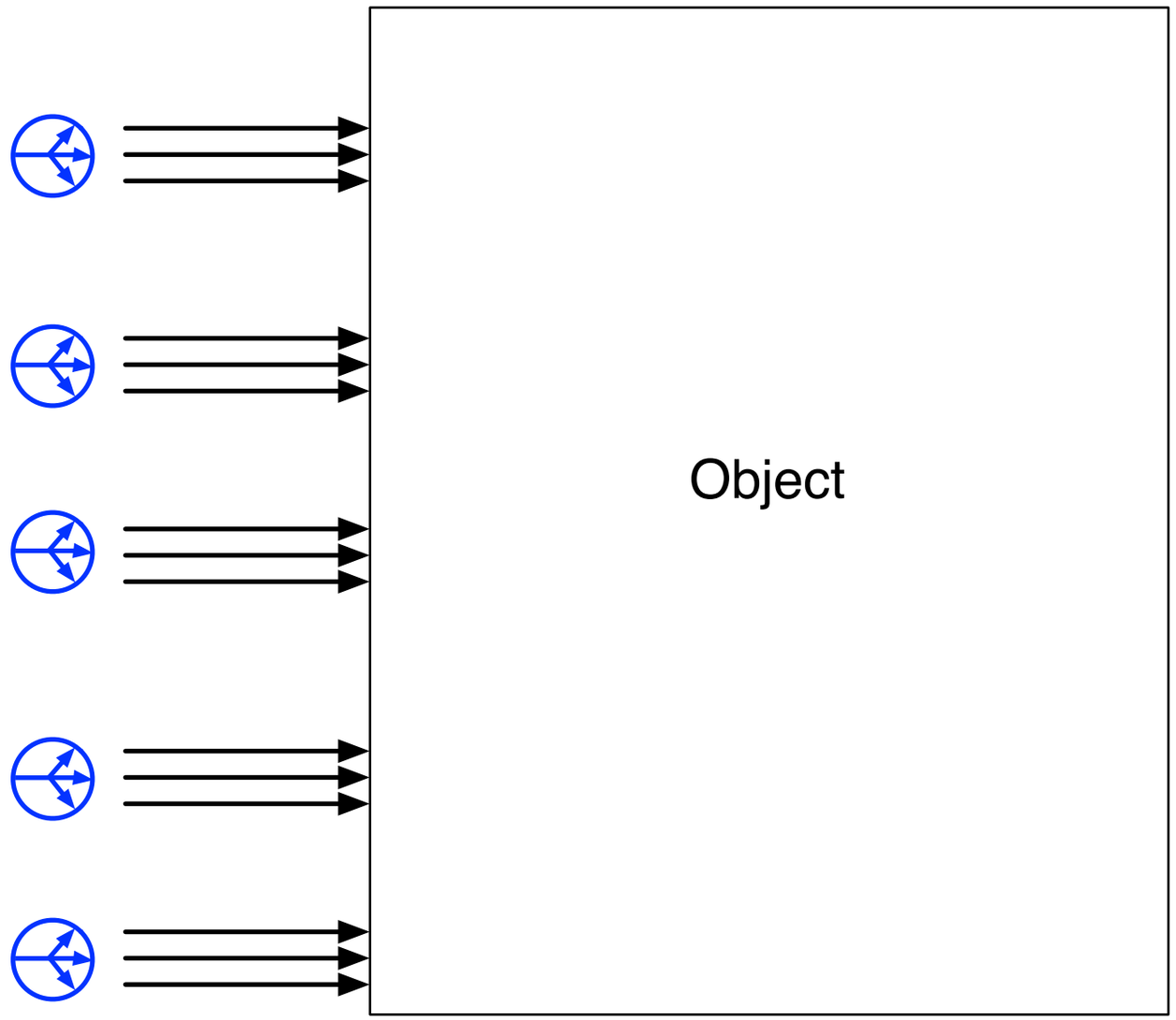
objects provide...

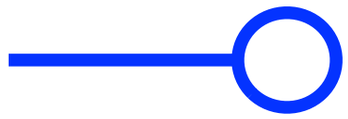


methods

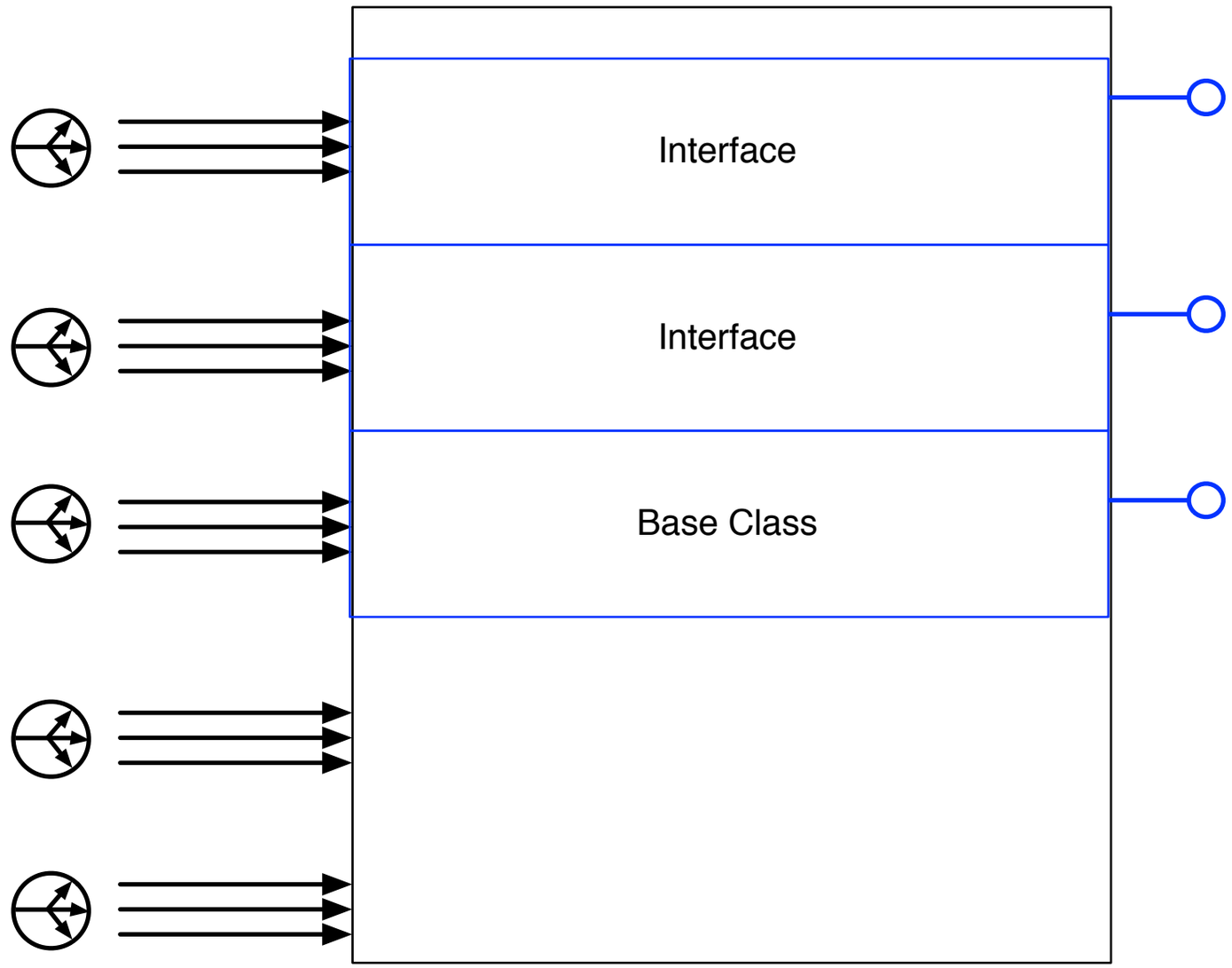


polymorphism

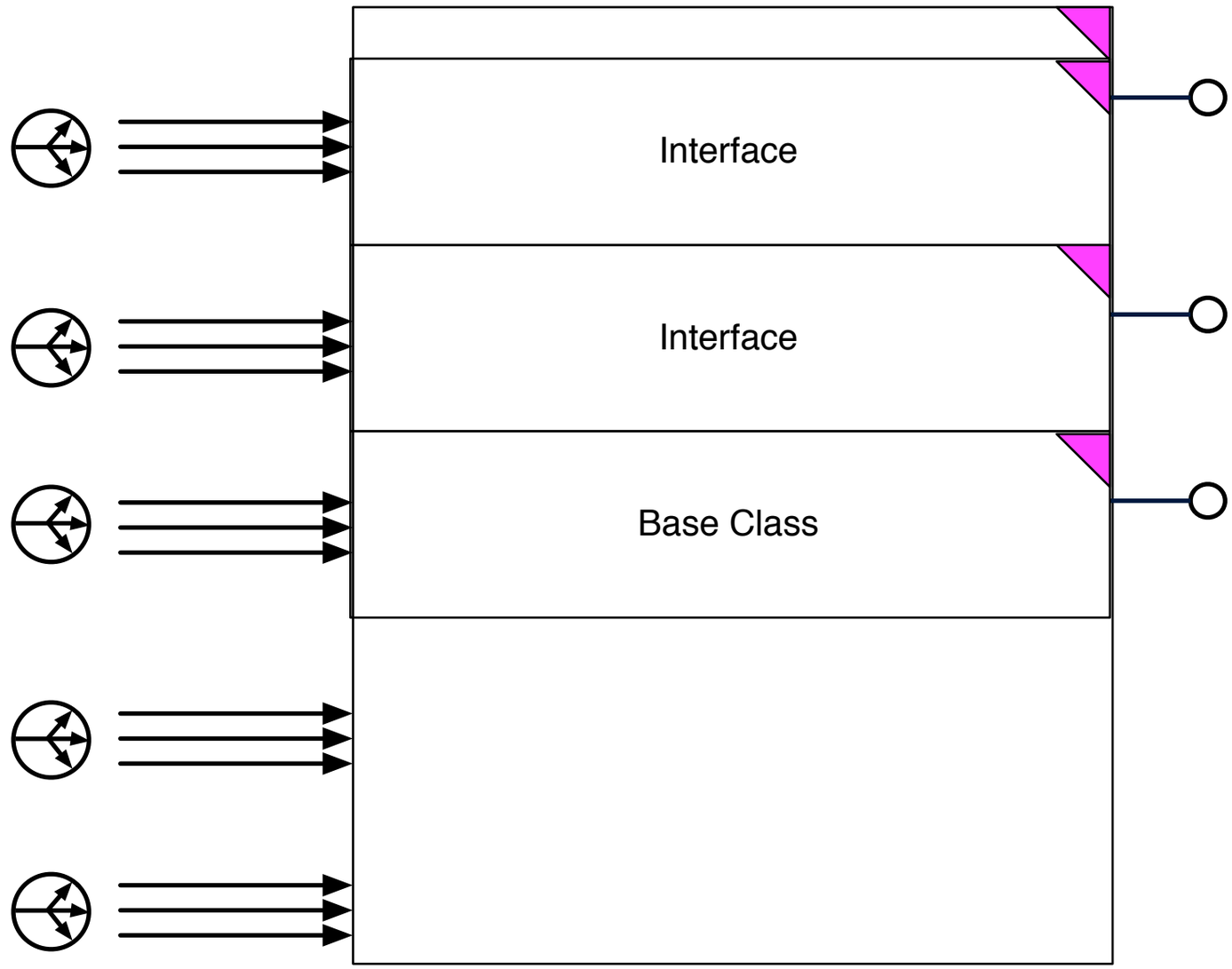


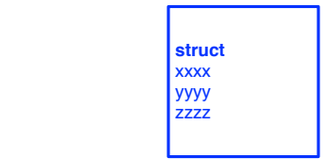


types

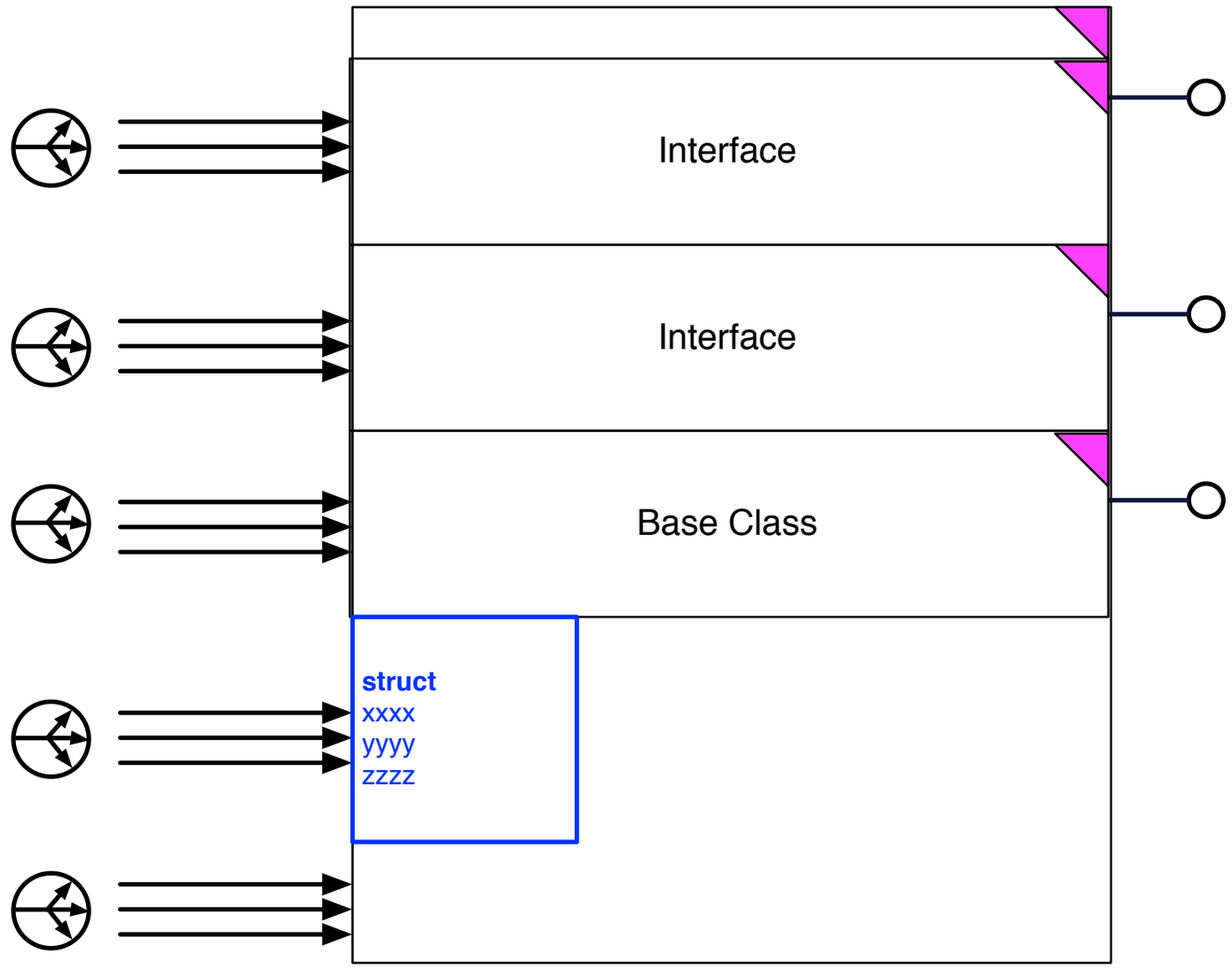


namespaces

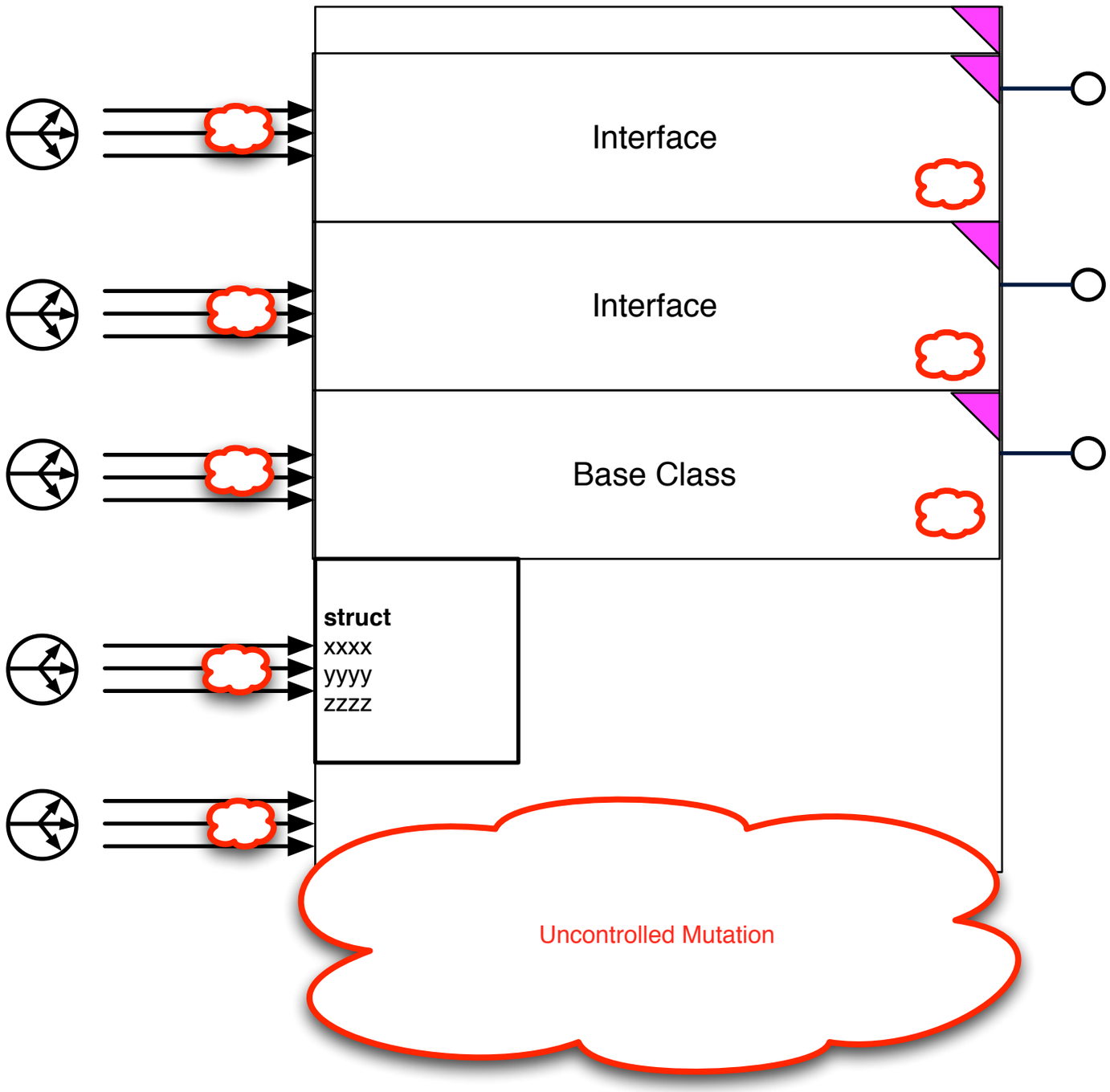



struct
xxxx
yyyy
zzzz

structure


Interface
Interface
Base Class
struct
xxxx
yyyy
zzzz

uncontrolled mutation



pojo is an

oxymoron

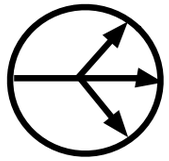
Java* style:
compound, complex, complicated

features are delivered as compounds

compounds contain more than you need

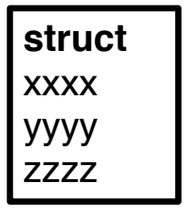
snowballs from language into libs

Clojure delivers simples & composites

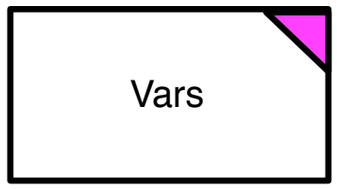


polymorphism

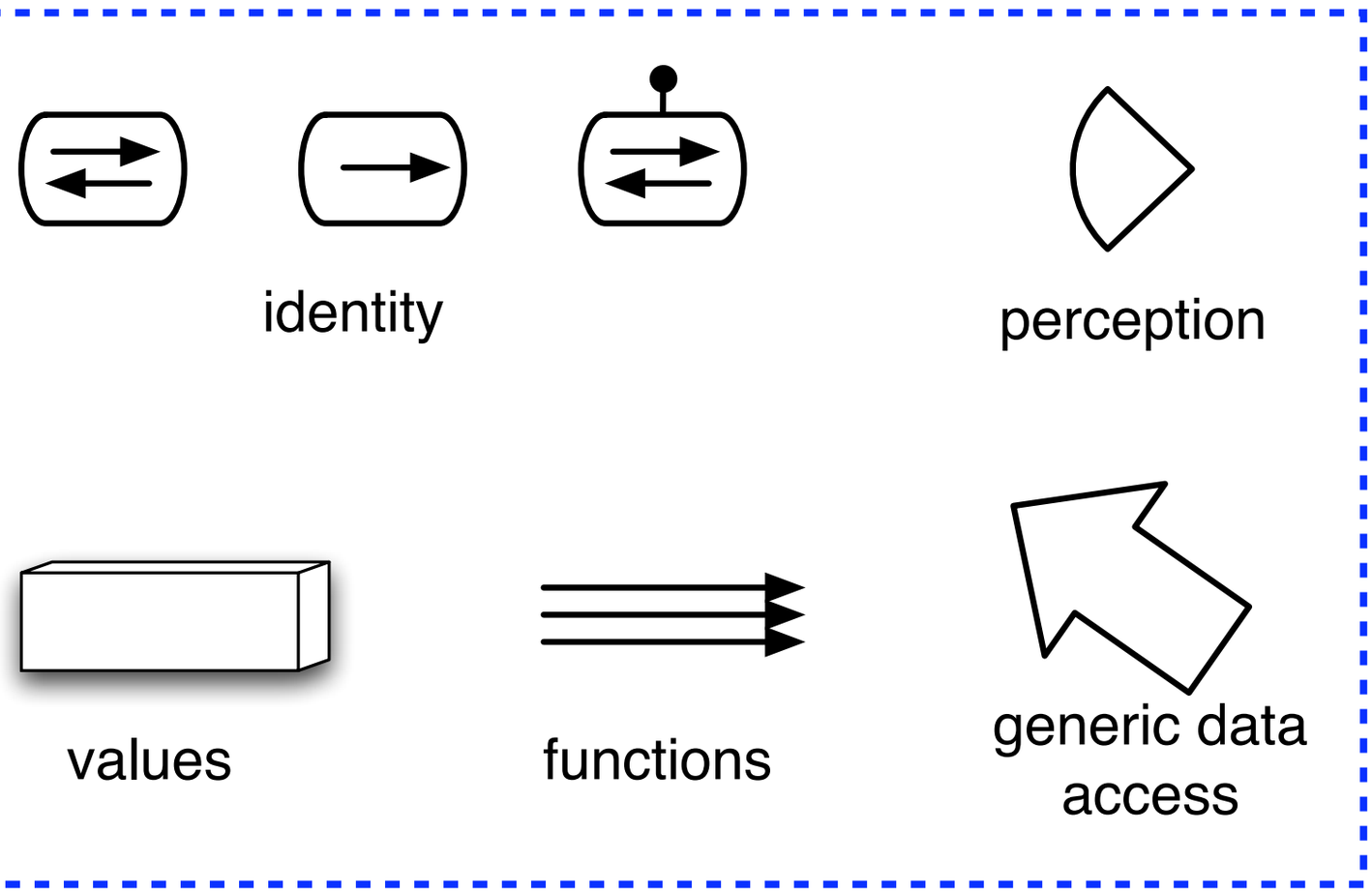
types



structure



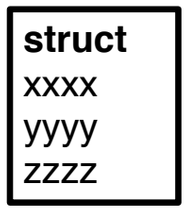
namespaces



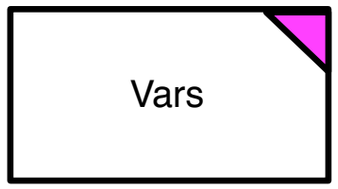
foundation

polymorphism

types

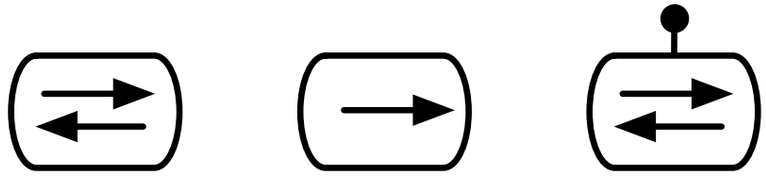


structure

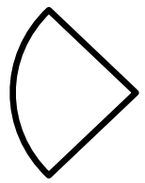


namespaces

superstructure

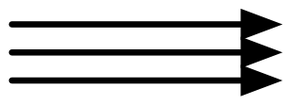


identity

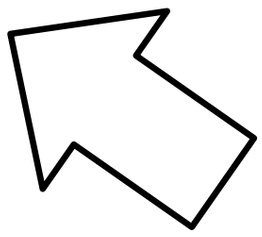


perception

values



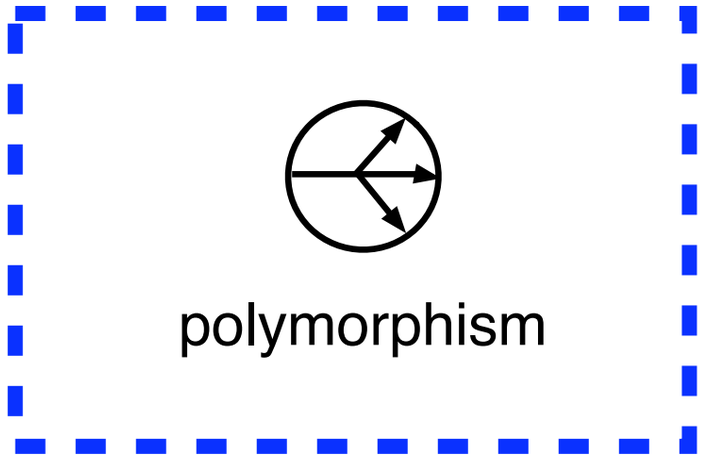
functions

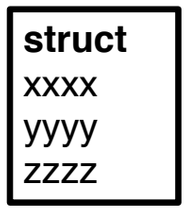


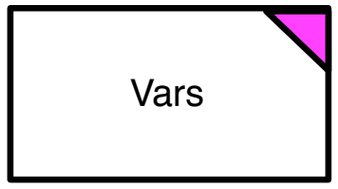
generic data access

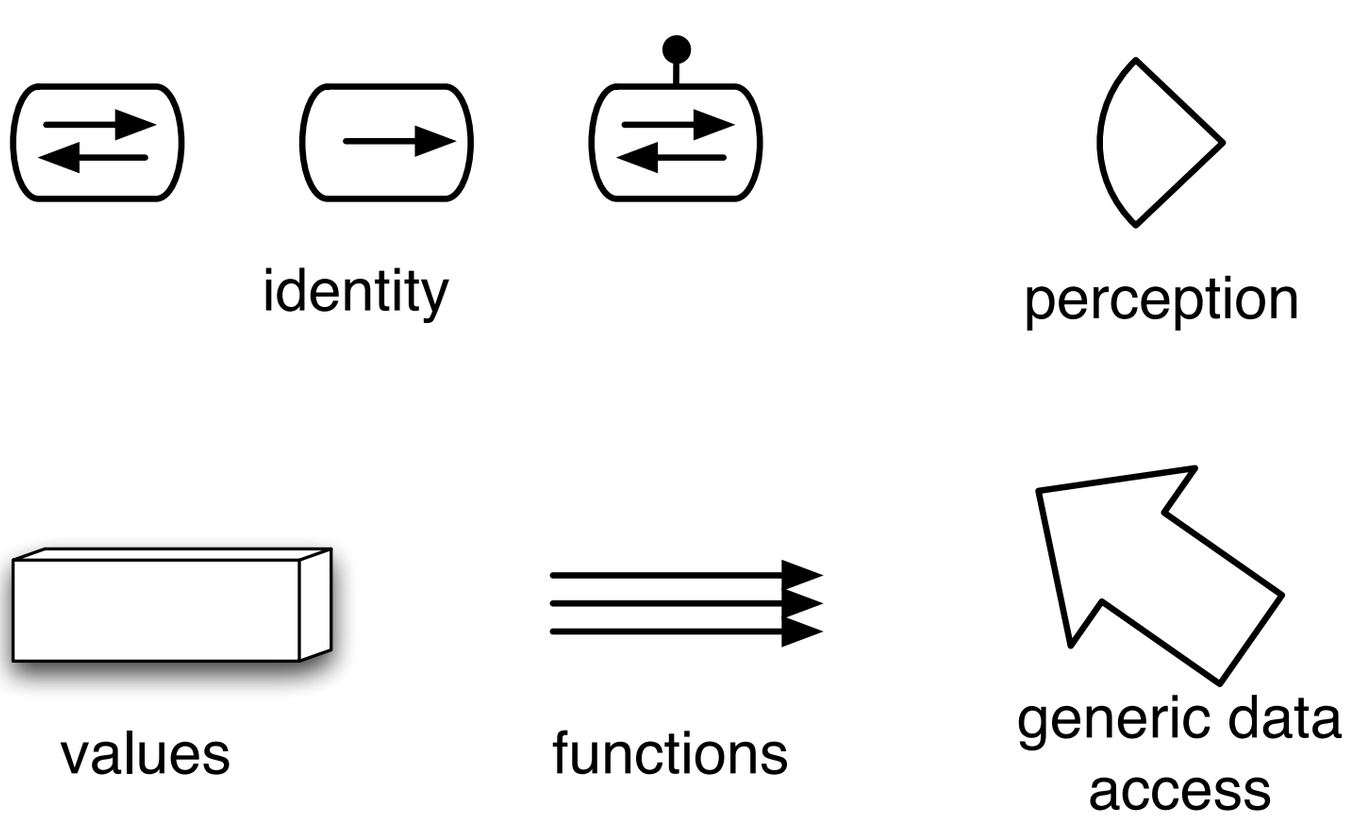
examples

polymorphism a la carte









Coercions

```
(defprotocol Coercions
  "Coerce between various 'resource-namish' things."
  (as-file [x] "Coerce argument to a file.")
  (as-url [x] "Coerce argument to a URL."))
```

implementation contract
!= caller API

```
(extend-protocol Coercions
  nil
  (as-file [_] nil)
  (as-url [_] nil)

  String
  (as-file [s] (File. s))
  (as-url [s] (URL. s))
```

extend to
existing types

extension options revisited

inline

extend types to protocol

extend protocol to types

build directly from fns and maps

implementation

reuse

extend

```
(extend BufferedInputStream
  IOFactory
  (assoc default-streams-impl
    :make-input-stream (fn [x opts] x)
    :make-reader inputstream->reader))
```

no special mixin API: use generic collection fns

plain ol' keywords and fns

mere maps

```
;; elided from clojure.java.io
(def default-streams-impl
  {:make-reader (fn [x opts] ...)
   :make-writer (fn [x opts] ...)
   :make-input-stream (fn [x opts] ...)
   :make-output-stream (fn [x opts] ...)})
```


make a map
of names -> fns...

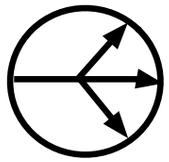
simple syntax: singular *interpretation*

(verb & args)

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

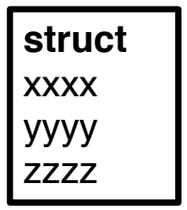
[bindings]

form	example
fn	`(fn [x] (* x 2))`
named fn	`(defn hi [s] (str "hi, " s)`
lexical bind	`(let [[x y] (range) ...)`
dynamic bind	`(binding [answer 42] ...)`
comprehension	`(for [x (range) ...] ...)`
side effects	`(doseq [{m :msg} warnings]...)`

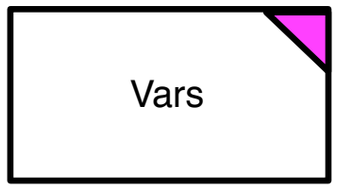


polymorphism

types

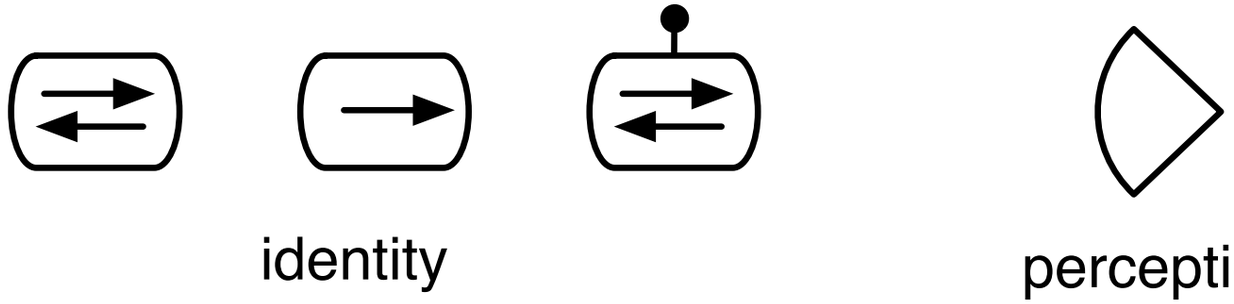


structure



namespaces

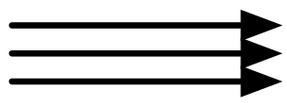
generic data access



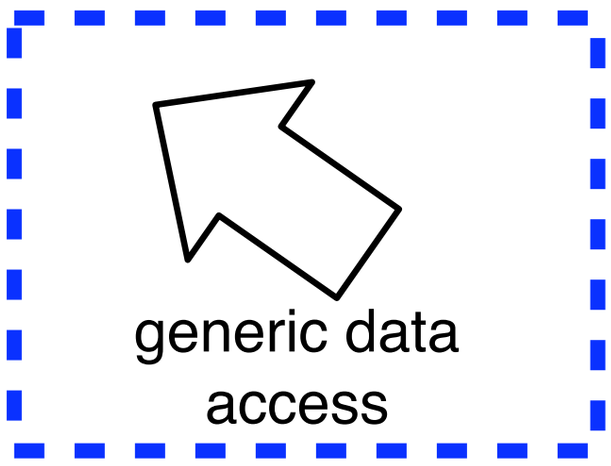
identity

perception

values



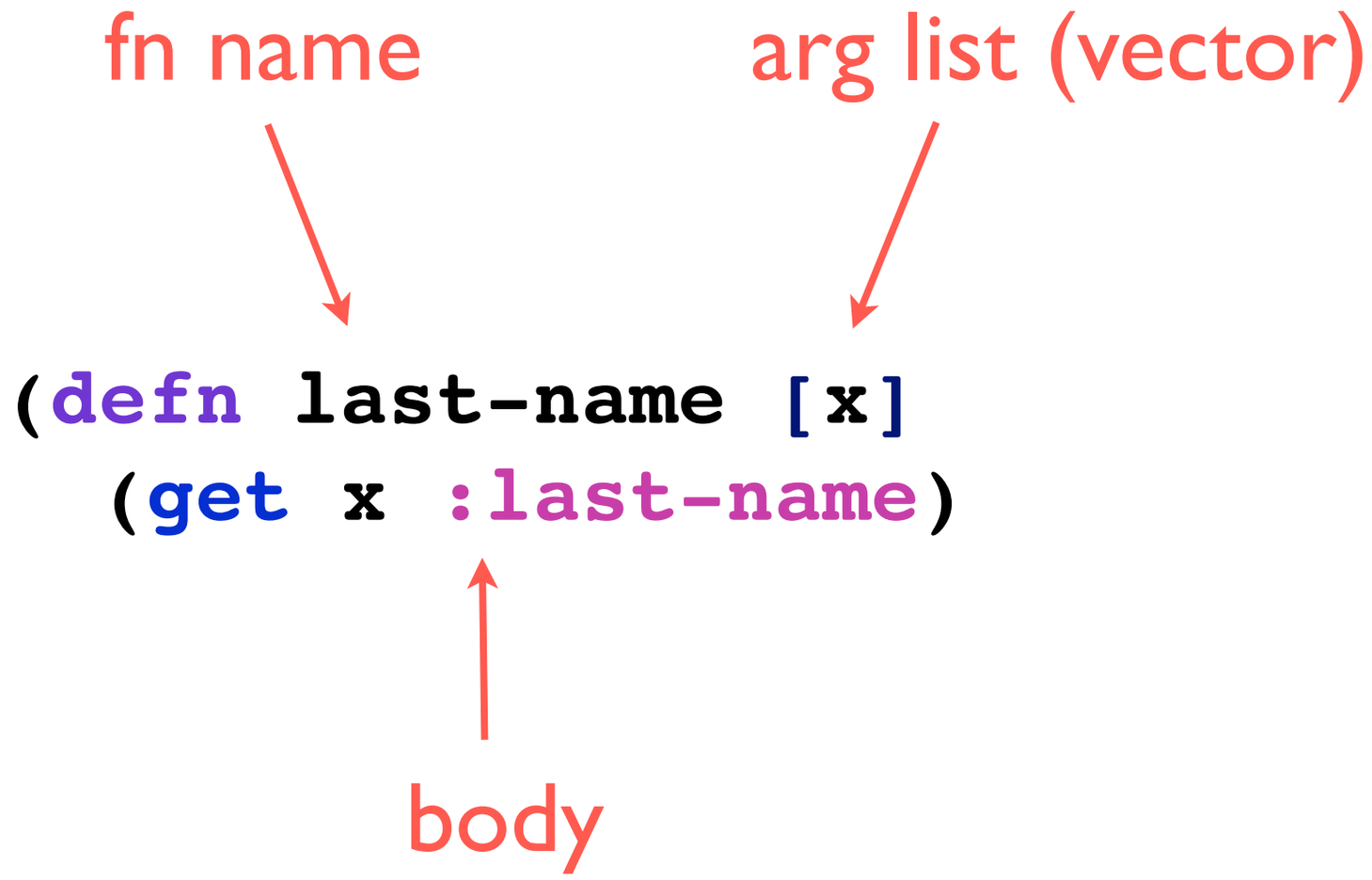
functions



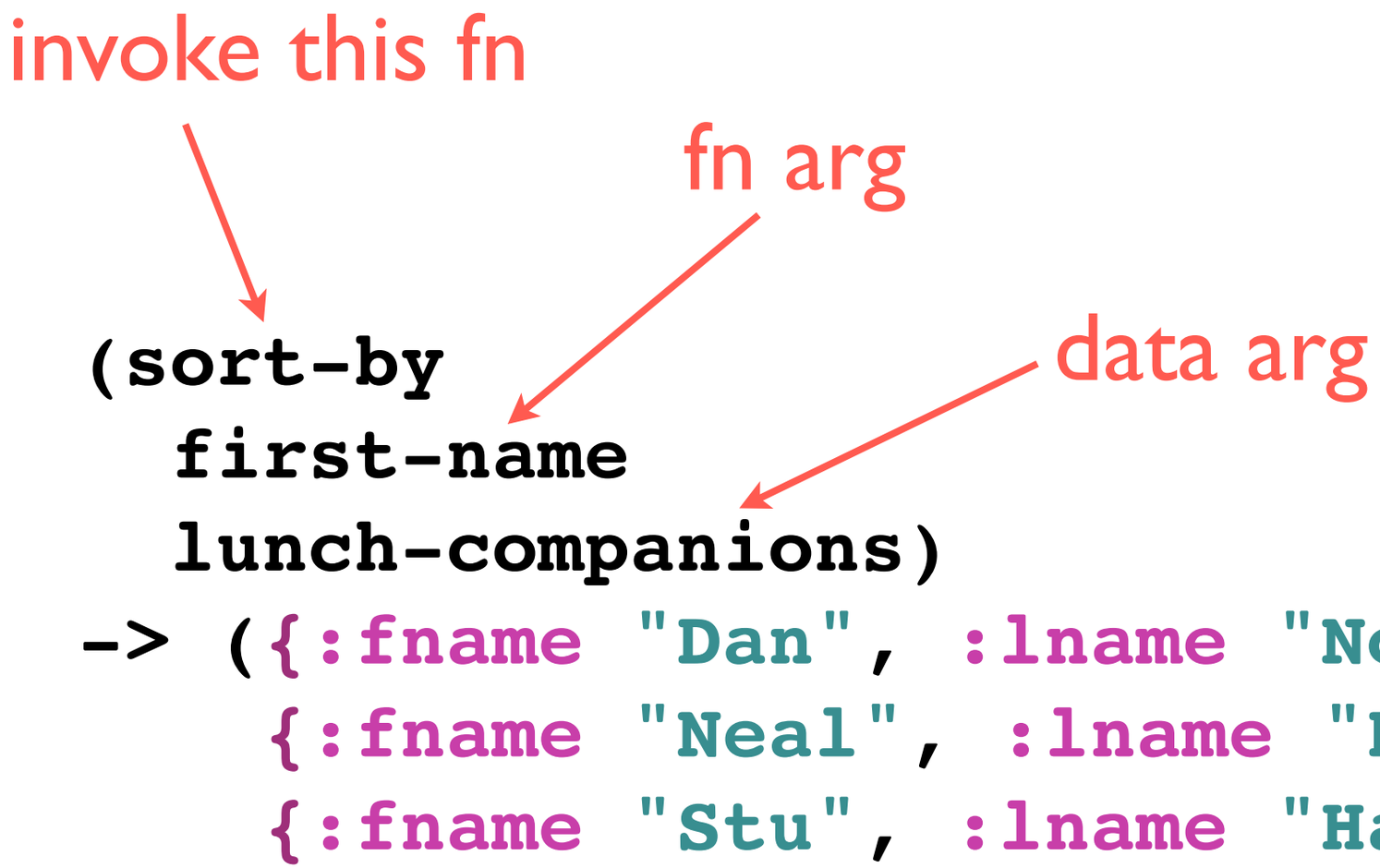
some data

```
lunch-companions
-> ({:fname "Neal", :lname "Ford"}
    {:fname "Stu", :lname "Halloway"}
    {:fname "Dan", :lname "North"})
```

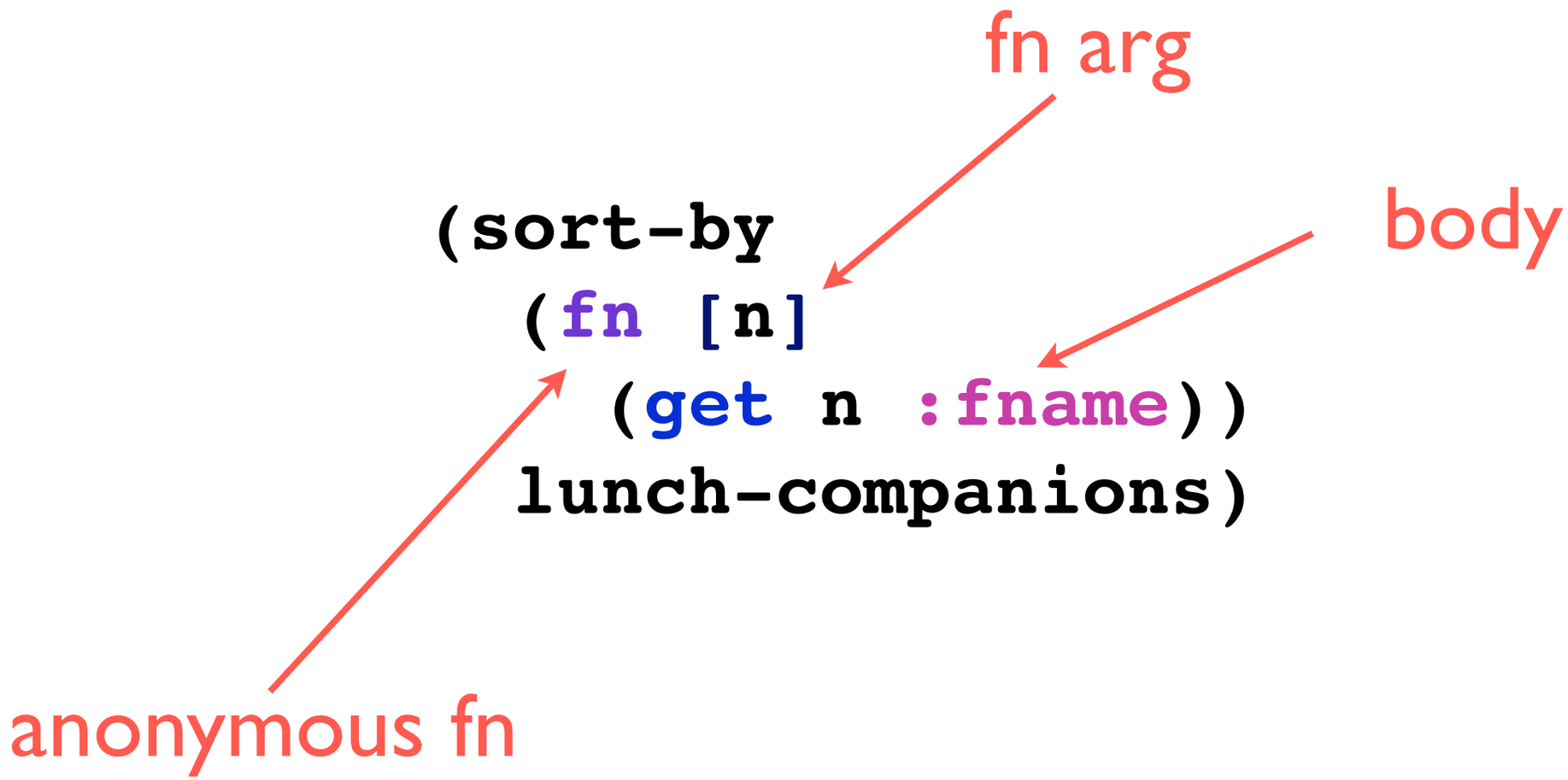
“getter” function



pass fn to fn


invoke this fn
fn arg
data arg
(sort-by
first-name
lunch-companions)
-> ({:fname "Dan", :lname "North"}
{:fname "Neal", :lname "Ford"}
{:fname "Stu", :lname "Halloway"})

anonymous fn



anonymous #()

fn arg

```
(sort-by
  #(get %  :fname)
  lunch-companions)
```

anonymous fn

maps are functions

map is fn!

```
(sort-by
   #(% :fname)
lunch-companions)
```

keywords are functions

keyword
is fn!

```
(sort-by
  #(:fname %)
  lunch-companions)
```

beautiful

```
(sort-by :fname lunch-companions)
```

direct access

to the **jvm**

living on the jvm

performance

power

reach

jvm tradeoffs

primitive operations

collection conformance

absence of **tco**

oo interop (genclass, proxy)

alternate numeric ops

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)
```

converting to arrays

```
(def nums (range 3))
(defn member-type [arr]
  (-> arr class (.getComponentType)))

(-> (to-array nums) member-type)
-> java.lang.Object

(-> (into-array nums) member-type)
-> java.lang.Integer

(-> (into-array Comparable nums) member-type)
-> java.lang.Comparable
```

always Object

inferred from first member

explicit

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 only where needed, inference handles the rest

```
;; 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"
```

explicit **tco**

```
(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}
```

loops are rare

```
(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)
```

annotations

plain ol'
Clojure metadata

```
(deftype ^{Deprecated true}
  Foo
  [^{Retention RetentionPolicy/RUNTIME} a b])
```

with keys and values
as required by annotation

numeric options

Arbitrary
Fast
Limited
No
Not **A**pplicable
Blow
Very Fast
Yes
* (caller choose one)

		safety?	speed	canonical?	precision	contagion?
java	default	N	VF	N	L	N
java	big	Y	S	N/A	A	N/A
clojure	default	Y	F*	Y	L	Y*
clojure	big	Y	S	N/A	A	N/A
clojure	unchecked	N	VF	Y	L	N

example:

reflection

sure wish we could...

```
(reflect "Foo")
(reflect [1 2 3])
(reflect FileInputStream)
```

history: repl-utils/show

```
(show "foo" #"last")
===  public final java.lang.String  ===
[56] lastIndexOf : int (String)
[57] lastIndexOf : int (String,int)
[58] lastIndexOf : int (int)
[59] lastIndexOf : int (int,int)
```

java reflection is compound

query

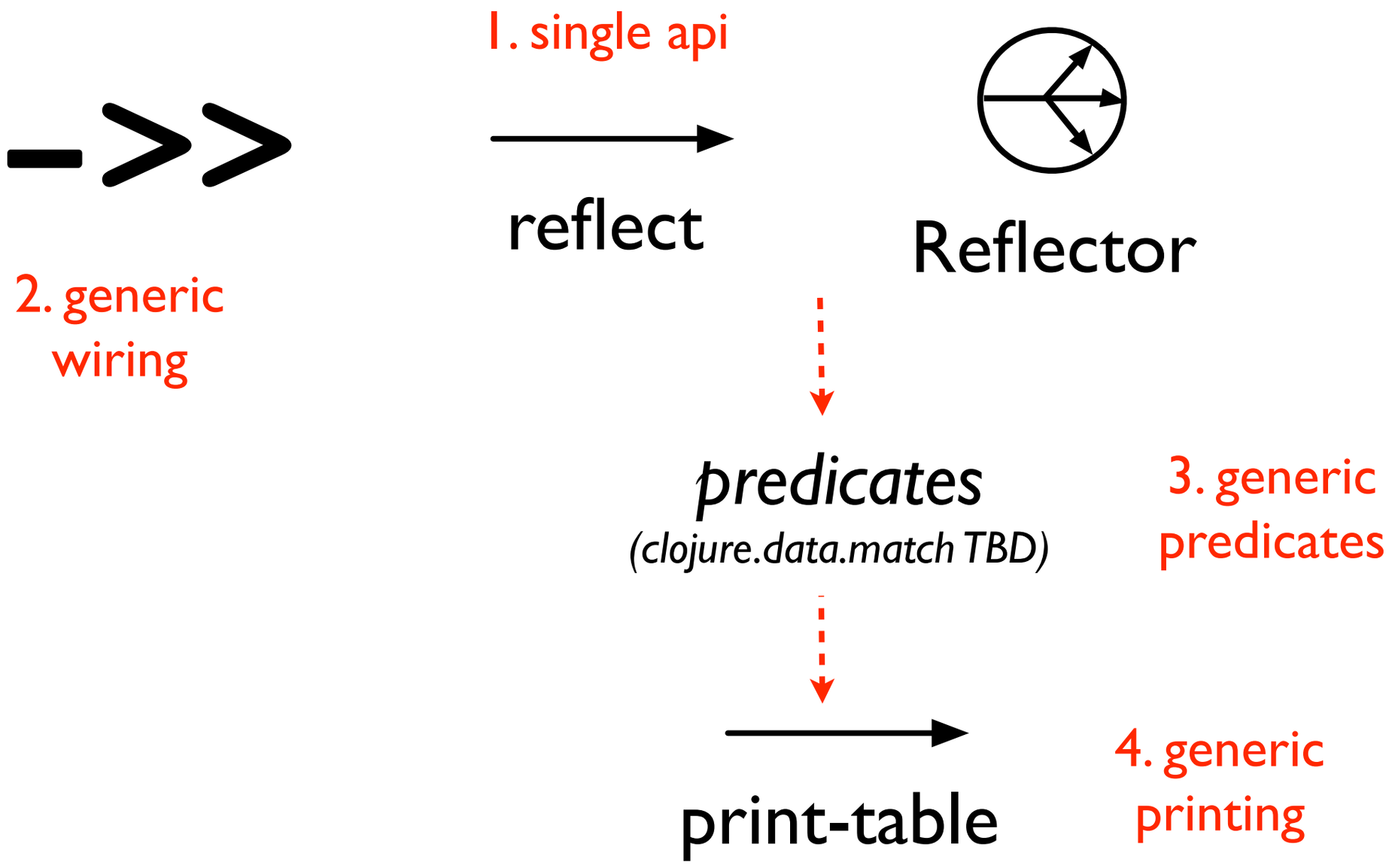
describe

invoke

class in hand

class *in context*

approach


->>
1. single api
reflect
Reflector
2. generic
wiring
predicates
(clojure.data.match TBD)
3. generic
predicates
print-table
4. generic
printing

usage

```
(->> (reflect java.lang.String)
     :members
     (filter #(.startsWith (str (:name %)) "last"))
     (print-table))
```

```
=================================================================================================================
:name       | :return-type | :declaring-class | :parameter-types                    | :exception-types | :flags
=================================================================================================================
lastIndexOf | int          | java.lang.String | [java.lang.String]                  | []               | #{:public}
lastIndexOf | int          | java.lang.String | [char<> int int char<> int int int] | []               | #{:static}
lastIndexOf | int          | java.lang.String | [int]                               | []               | #{:public}
lastIndexOf | int          | java.lang.String | [java.lang.String int]              | []               | #{:public}
lastIndexOf | int          | java.lang.String | [int int]                           | []               | #{:public}
=================================================================================================================
```

modeling classfiles, poorly

```
// from java.lang.reflect.Modifier
public static final int PUBLIC           = 0x00000001;
public static final int PRIVATE          = 0x00000002;
public static final int PROTECTED        = 0x00000004;
public static final int STATIC           = 0x00000008;
public static final int FINAL            = 0x00000010;
public static final int SYNCHRONIZED     = 0x00000020;
public static final int VOLATILE         = 0x00000040;
public static final int TRANSIENT        = 0x00000080;
```

problems

classes are poor containers

no data literals to use instead

modifiers, types add no value

data doesn't capture the model

knowledge is data

```
(defn- access-flag
  [[name flag & contexts]]
  {:name name
   :flag flag
   :contexts (set (map keyword contexts))})

(def flag-descriptors
  (vec
   (map access-flag
        [[:public 0x0001 :class :field :method]
         [:private 0x002 :class :field :method]
         [:protected 0x0004  :class :field :method]
         [:static 0x0008  :field :method]
         [:final 0x0010  :class :field :method]
         [:synchronized 0x0020  :method]
         [:volatile 0x0040  :field]
         [:bridge 0x0040  :method]])))
```

irrelevant destractions

field

method

constructor

member

modifier

relevant abstractions

```
(defprotocol Reflector
  "Protocol for reflection implementers."
  (do-reflect [reflector typeref]))

(defprotocol TypeReference
  "A TypeReference can be unambiguously
   converted to a type name on the host
   platform."
  (typename [o]))
```

abstractions != **api**

```
(defn reflect
  [obj & options]
  (apply type-reflect
   (if (class? obj) obj (class obj)) options))
```

information ~ maps

```
(->> (reflect java.lang.String)
     :members
     (filter #(.startsWith (str (:name %)) "last"))
     pprint)

({:name lastIndexOf,
  :return-type int,
  :declaring-class java.lang.String,
  :parameter-types [java.lang.String],
  :exception-types [],
  :flags #{:public}}
 {:name lastIndexOf,
  :return-type int,
  :declaring-class java.lang.String,
  :parameter-types [char<> int int char<> int int int],
  :exception-types [],
  :flags #{:static}}
  ...)
```

Clojure fits if you value

simplicity and *directness*

over

convenience and *familiarity*

thanks!

http://clojure.org