Clojure for OOP folks

Stefan Tilkov | @stilkov | innoQ

innoQ

Motivation

Syntax

Idioms

OOP Thinking

model domains with classes & interfaces

encapsulate data in objects

prefer specific over generic solutions

explicitly provide for generic access

Namespaces

… just like ~~Java~~ packages

refer: import names

:exclude [], :only [], :rename {...:...}

require: (re-)load libs

:reload, :reload-all, :as

use: require + refer

:exclude [], :only [], :rename {...:...}

ns: create namespace

:require, :refer, :use, :gen-class

Handle var name clashes

Reduce dependencies

Dynamic reloading

Namespace aliases

Convenient REPL usage

Flexible handling in sources

Provide encapsulation

refer: import names

:exclude [], :only [], :rename {...:...}

Handle var name clashes

Reduce dependencies

require: (re-)load libs

:reload, :reload-all, :as, **:refer**

Dynamic reloading

Namespace aliases

Convenient REPL usage

ns: create namespace

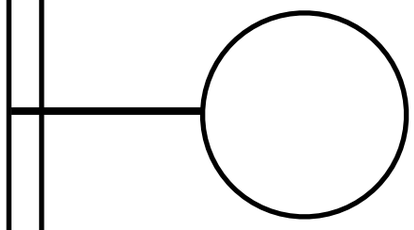
:require, :refer, :use, :gen-class

Flexible handling in sources

Provide encapsulation

```clojure
(ns com.example.some-ns
  "Well-documented ns"
   (:use [com.example.n1 :only [xyz]])
   (:require [com.example.ns2 :as n2]))
```

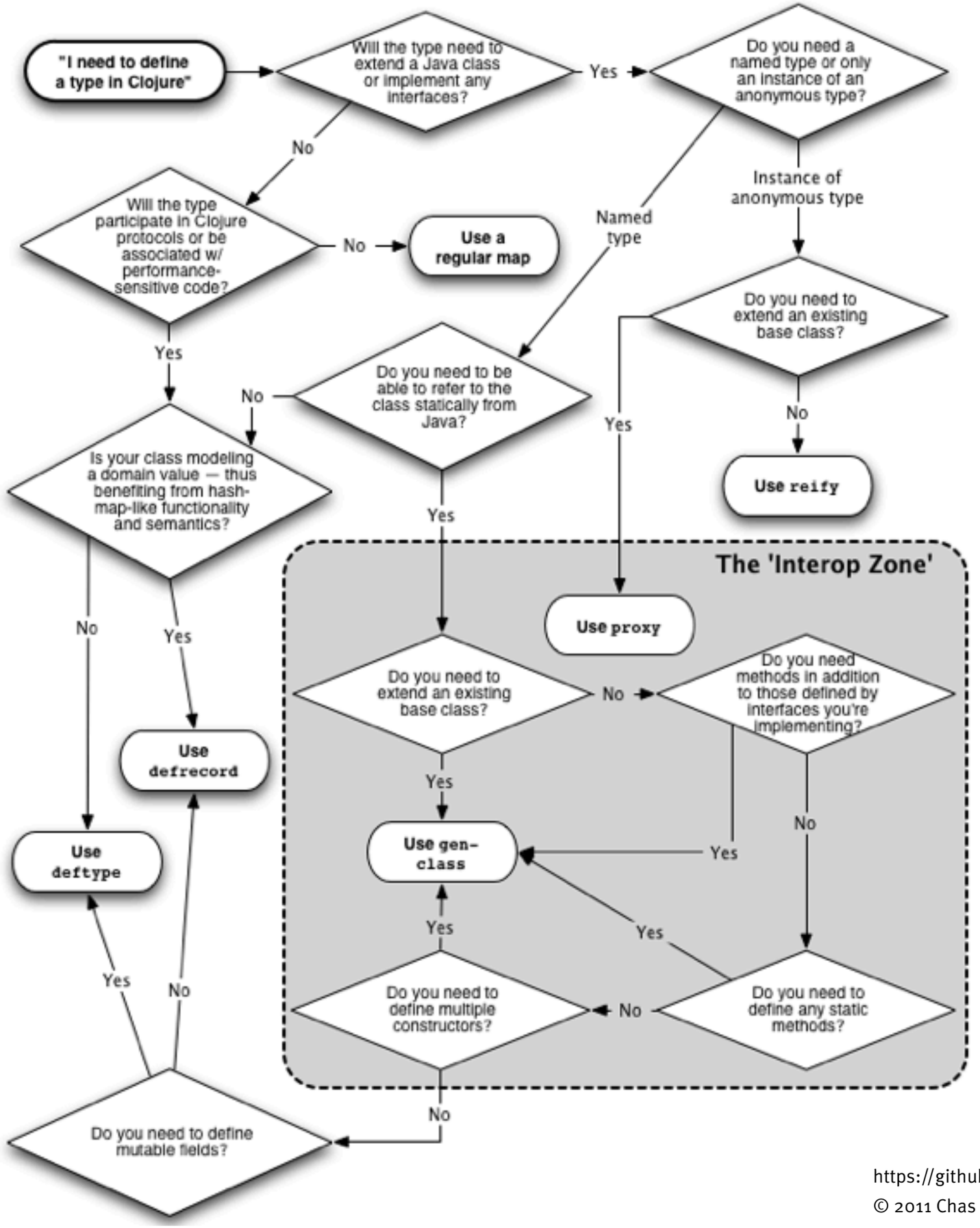
```clojure
(defn ...)
(defmacro ...)
(defmulti ...)
(defmethod ...)
```



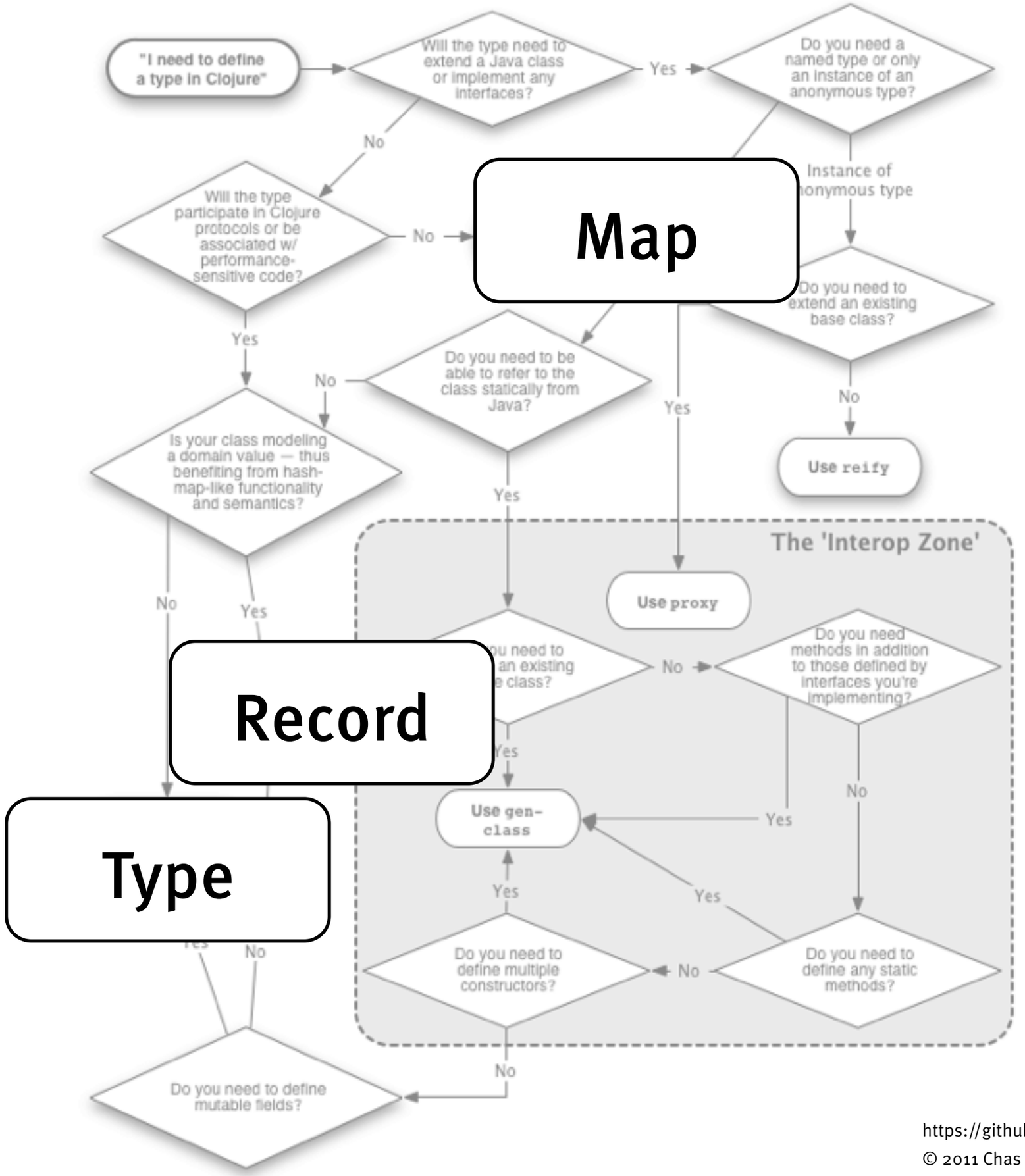
```clojure
(defn- ...)
(def ^:private ...)
(def ^:dynamic ...)
```

Data

How to choose a datatype



https://github.com/cemerick/clojure-type-selection-flowchart/



https://github.com/cemerick/clojure-type-selection-flowchart/

Map

Record

Type

Function

Multimethod

Protocol

Map

Function

Multimethod

Map

Function

PDS

Function

Data structures vs. objects

```
public class Point {
    private final double x;
    private final double y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
}

Point p1 = new Point(3, 4);
```

```
(def p1 [3 4])
```

Data structures vs. objects

```
(def p1 [3 4])
```

Immutable

Reusable

Compatible

Data structures vs. objects

```
import static java.lang.Math.sqrt;

public class Point {
    private final double x;
    private final double y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double distanceTo(Point other) {
        double c1 = other.x - this.x;
        double c2 = other.y - this.y;
        return sqrt(c1 * c1 + c2 * c2);
    }
}
```

Data structures vs. objects

```
(import-static java.lang.Math sqrt)

(defn distance
  [[x1 y1] [x2 y2]]
  (let [c1 (- x2 x1)
        c2 (- y2 y1)]
    (sqrt (+ (* c1 c1) (* c2 c2)))))
```

Data structures vs. objects

```
(defn rand-seq [limit]
  (repeatedly #(rand-int limit)))
```

infinite randoms

```
(take 10 (partition 2 (rand-seq 10)))
```

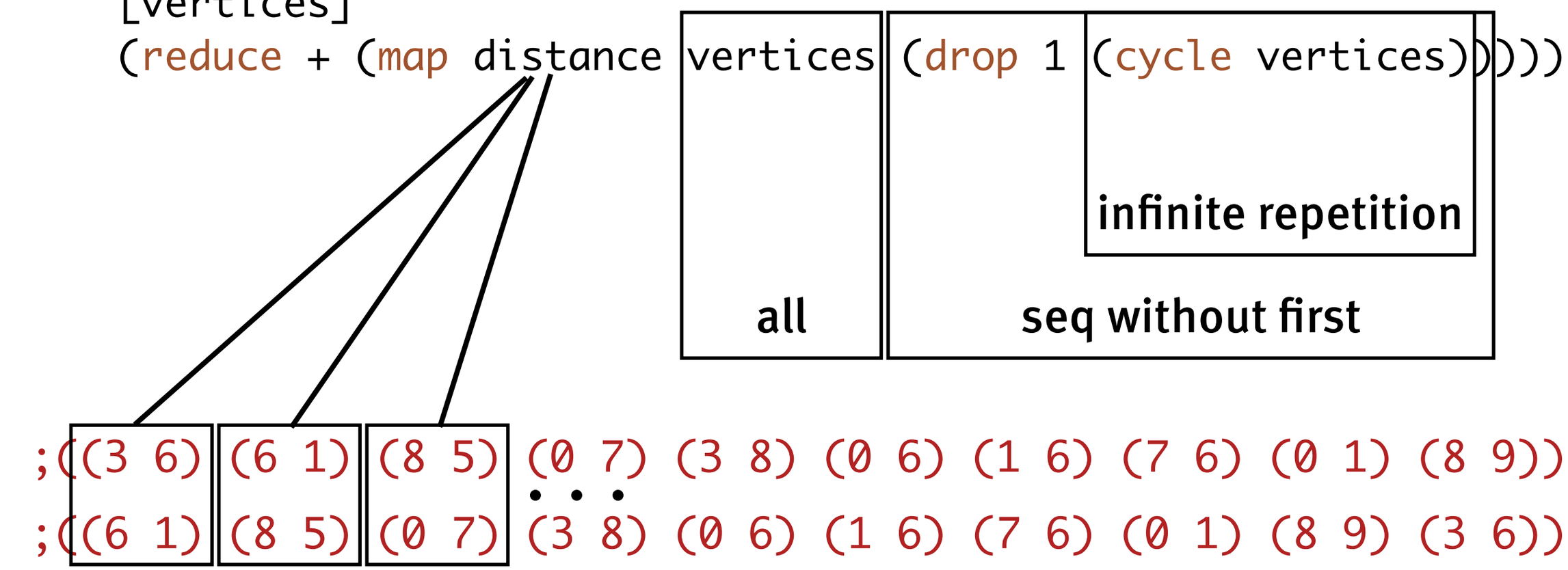
pairs of random ints

10 random points

```
;((3 6) (6 1) (8 5) (0 7) (3 8) (0 6) (1 6) (7 6) (0 1) (8 9))
```

Data structures vs. objects

```
(defn circumference
  [vertices]
  (reduce + (map distance vertices (drop 1 (cycle vertices)))))
```




```
;((3 6) (6 1) (8 5) (0 7) (3 8) (0 6) (1 6) (7 6) (0 1) (8 9))
;((6 1) (8 5) (0 7) (3 8) (0 6) (1 6) (7 6) (0 1) (8 9) (3 6))

;58.06411369758525
```

```
assoc
assoc-in
butlast
concat
conj
cons
count
cycle
difference
dissoc
distinct
distinct?
drop-last
empty
empty?
every?
filter
first
flatten
group-by
interleave
interpose
intersection
into
join
lazy-cat
mapcat
merge
merge-with
not-any?
not-empty?
not-every?
nth
partition
partition-all
partition-by
peek
pop
popy
project
remove
replace
rest
rseq
select
select-keys
shuffle
some
split-at
split-with
subvec
take
take-last
take-nth
take-while
union
update-in
```

Maps

```
(def projects #{{:id "1",
                 :kind :time-material,
                 :description "Consulting for BigCo",
                 :budget 25000,
                 :team [:joe, :chuck, :james]}
                {:id "2",
                 :kind :fixed-price,
                 :description "Development for Startup",
                 :budget 100000,
                 :team [:john, :chuck, :james, :bill]}
                {:id "3",
                 :kind :fixed-price,
                 :description "Clojure Training",
                 :budget 3000,
                 :team [:joe, :john]}})
```

Map access

```
(defn all-members
  [projects]
  (reduce conj #{} (flatten (map :team projects))))
```

seq of vectors

seq of members with duplicates

set of all team members

```
(all-members projects)
;#{:chuck :joe :james :john :bill}
```

Map access & coupling

```
(defn all-members
  [projects]
  (reduce conj #{} (flatten (map :team projects))))
```

```
#{{:id "2",
   :kind :fixed-price,
   :description "Development for Startup",
   :budget 100000,
   :team [:john, :chuck, :james, :bill]}}
```

Map access & coupling

```
(defn all-members
  [projects]
  (reduce conj #{} (flatten (map :team projects))))
```

```
#{{:id "2",
   :kind :fixed-price,
   :description "Development for Startup",
   :budget 100000,
   :team [:john, :chuck, :james, :bill]}}
```

(json-str)

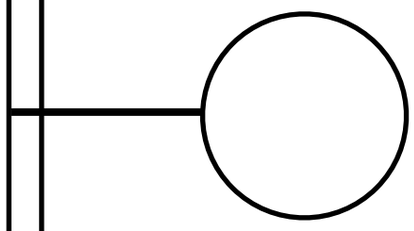
```
[{:kind "fixed-price",
  :team ["john" "chuck" "james" "bill"],
  :budget 100000,
  :id "2",
  :description "Development for Startup"}
{:kind "fixed-price",
  :team ["joe" "john"],
  :budget 3000,
  :id "3",
  :description "Clojure Training"}
{:kind "time-material",
  :team ["joe" "chuck" "james"],
  :budget 25000,
  :id "1",
  :description "Consulting for BigCo"}]
```

```
[{"kind":"fixed-price",
  "team":["john", "chuck", "james",
"bill"],
  "budget":100000,
  "id":"2",
  "description":"Development for Startup"},
{"kind":"fixed-price",
  "team":["joe", "john"],
  "budget":3000,
  "id":"3",
  "description":"Clojure Training"},
{"kind":"time-material",
  "team":["joe", "chuck", "james"],
  "budget":25000,
  "id":"1",
  "description":"Consulting for BigCo"}]
```

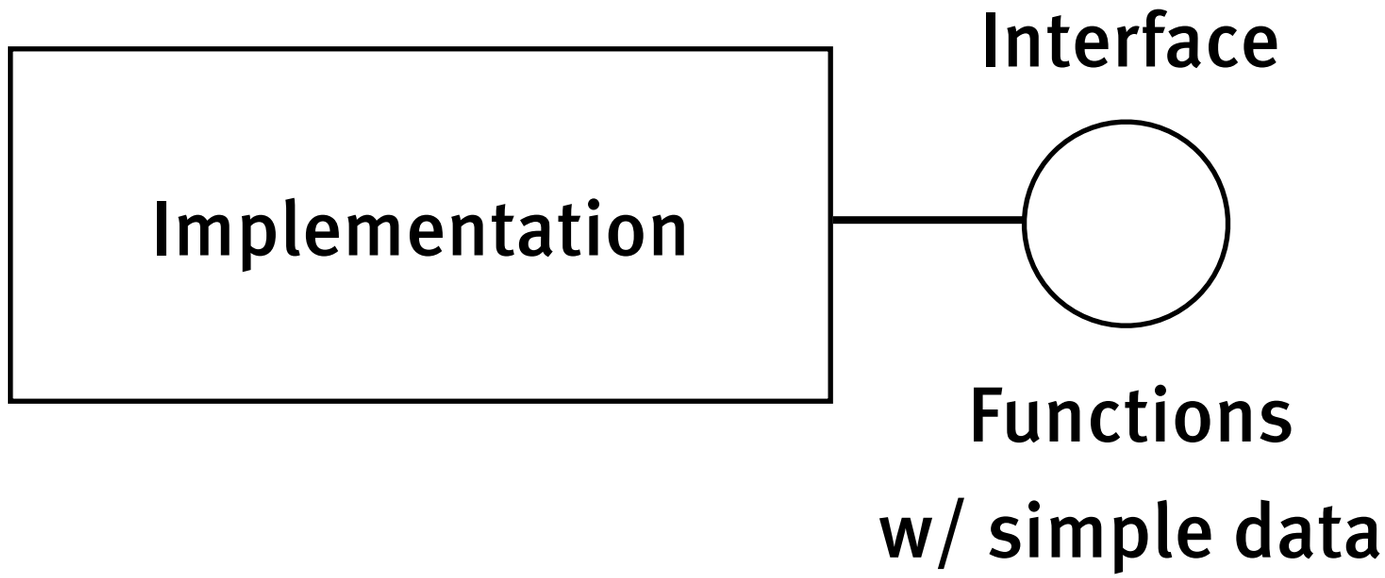
(read-json)

```clojure
(ns com.example.some-ns
  "Well-documented ns"
   (:use [com.example.n1 :only [xyz]])
   (:require [com.example.ns2 :as n2]))
```

```clojure
(defn ...)
(defmacro ...)
(defmulti ...)
(defmethod ...)
```



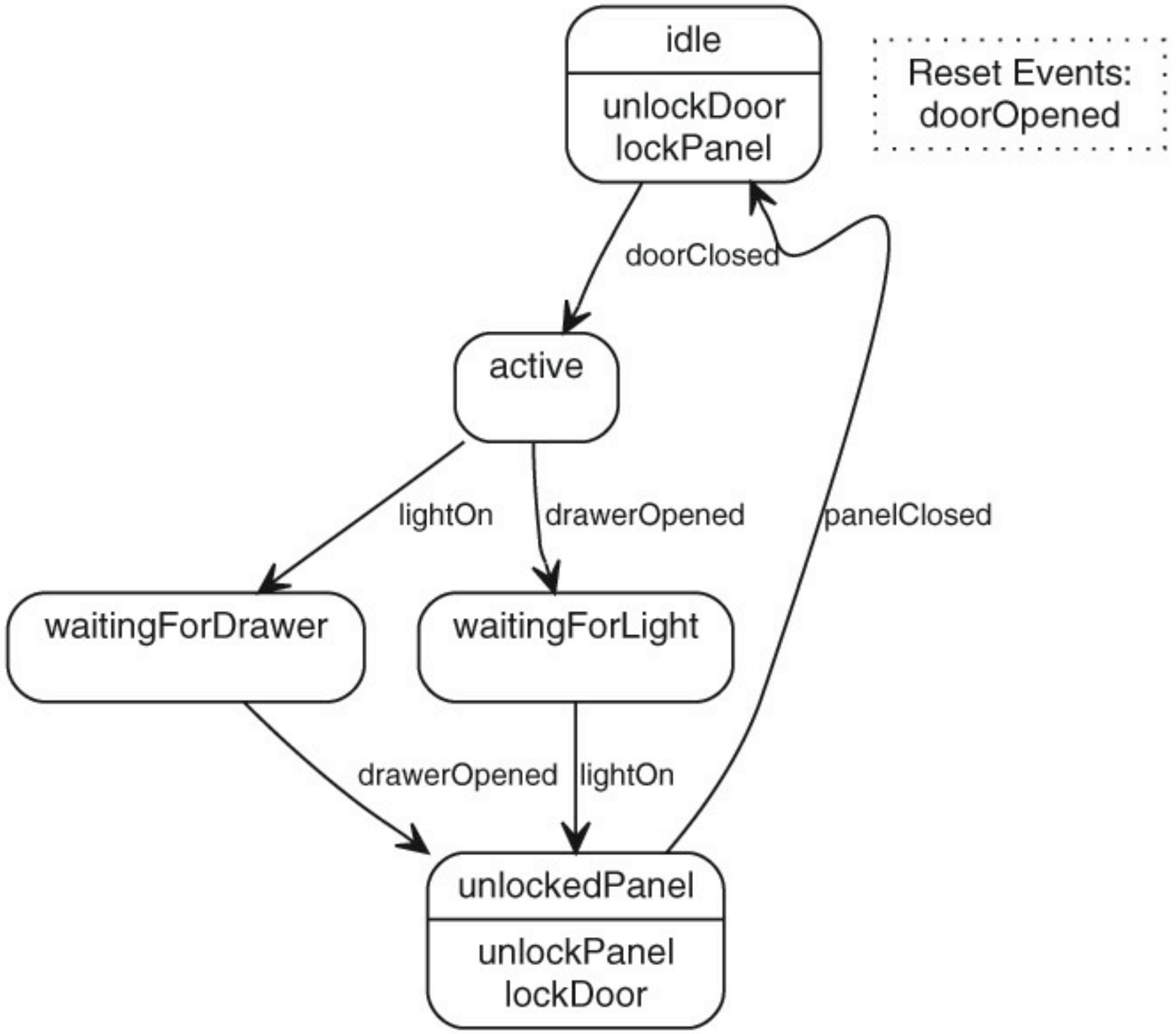
```clojure
(defn- ...)
(def ^:private ...)
```



Closures

```clojure
(defn make-id
  [prefix id]
  (join "-" [prefix (Long/toString id 16)]))

(defn id-generator
  ([prefix]
     (id-generator prefix 0))
  ([prefix v]
     (let [cnt (atom v)]
       (fn [] (make-id prefix (swap! cnt inc))))))

(def prj-id (id-generator "prj"))

(prj-id)
;; "prj-1"
(prj-id)
;; "prj-2"
(prj-id)
;; "prj-3"

(defn make-project [map]
  (assoc map :id (prj-id)))
```

Meet Miss Grant



http://www.informit.com/articles/article.aspx?p=1592379

```clojure
(defn unlock-door [] (println "Unlocking door"))
(defn lock-door [] (println "Locking door"))
(defn unlock-panel [] (println "Unlocking panel"))
(defn lock-panel [] (println "Locking panel"))


(def fsm
  (make-fsm :idle :doorOpened
            {:idle                [[unlock-door lock-panel]
                                   {:doorClosed :active}]
             :active              [[] {:drawerOpened :waitingForLight
                                       :lightOn :waitingForDrawer}]
             :waitingForLight     [[] {:lightOn :unlockedPanel}]
             :waitingForDrawer    [[] {:drawerOpened :unlockedPanel}]
             :unlockedPanel       [[unlock-panel lock-door]
                                   {:panelClosed :idle}]}))
```

```clojure
(defn make-fsm
  "creates an fsm with initial state s0, a reset event, and a map of transitions.
  [state-transitions] must be a map of state->[[f1 f2 ...] {e0->s0, e1->s2, ...}]"
  [s0 reset-event state-transitions ]
  (let [s (atom s0)]
    (fn [evt]
      (if (= evt reset-event)
        (do
          (println  "Reset event, returning to " s0)
          (swap! s (fn [_] s0)))
        (let [[actions transitions] (state-transitions @s)]
          (if-let [new-state (transitions evt)]
            (do
              (println  "Event" evt "causes transition from" @s "to" new-state)
              (doseq [f actions] (f))
              (swap! s (fn [_] new-state)))
            (println "Unexpected/unhandled event" evt "in state" @s)))))))
```

```clojure
(def fsm
  (make-fsm :idle :doorOpened
            {:idle                [[unlock-door lock-panel]
                                   {:doorClosed :active}]
             :active              [[] {:drawerOpened :waitingForLight
                                       :lightOn :waitingForDrawer}]
             :waitingForLight     [[] {:lightOn :unlockedPanel}]
             :waitingForDrawer    [[] {:drawerOpened :unlockedPanel}]
             :unlockedPanel       [[unlock-panel lock-door]
                                   {:panelClosed :idle}]}))

(dorun (map fsm [:doorClosed :lightOn :drawOpened :panelClosed]))

;; Event :doorClosed causes transition from :idle to :active
;; Unlocking door
;; Locking panel
;; Event :lightOn causes transition from :active to :waitingForDrawer
;; Event :drawerOpened causes transition from :waitingForDrawer to :unlockedPanel
;; Event :panelClosed causes transition from :unlockedPanel to :idle
;; Unlocking panel
;; Locking door
;; Reset event, returning to  :idle
```

Map

Function

Multimethod

Method problems

“Global” state

Coarse-grained re-use

Simple-minded dispatch

Methods vs. Multimethods

	Methods	Multimethods
Dispatch	Type	customizable
# of args	1	arbitrary
Hierarchy	based on type inheritance	customizable

Multimethods

```
(def projects #{{:id "1",
                 :kind :time-material,
                 :description "Consulting for BigCo",
                 :budget 25000,
                 :team [:joe, :chuck, :james]}
                {:id "2",
                 :kind :fixed-price,
                 :description "Development for Startup",
                 :budget 100000,
                 :team [:john, :chuck, :james, :bill]}
                {:id "3",
                 :kind :fixed-price,
                 :description "Clojure Training",
                 :budget 3000,
                 :team [:joe, :john]}})
```

Multimethods

```
(defmulti expected-revenue :kind)

(defmethod expected-revenue :default [p]
  (:budget p))

(defmethod expected-revenue :fixed-price [p]
  (* 0.8 (:budget p)))

(defn total-expected-revenue
  [projects]
  (reduce + (map expected-revenue projects)))
```

Multimethods

```
(defn make-rectangle
  [[p1 p2 p3 p4 :as vertices]]
  (let [a (distance p1 p2)
        b (distance p2 p3)]
    (assert (= a (distance p3 p4)))
    (assert (= b (distance p4 p1)))
    {:kind :rectangle, :vertices vertices, :a a, :b b}))

(defn make-circle
  [center r]
  {:kind :circle, :center center, :r r})

(defmulti area :kind)

(defmethod area :rectangle
  [{:keys [a b]}]
  (* a b))

(defmethod area :circle
  [{:keys [r]}]
  (* PI (pow r 2)))
```

Multimethods

```
(defmulti circumference :kind :default :polygon)

(defmethod circumference :polygon
  [{:keys [vertices]}]
  (reduce + (map distance vertices (drop 1 (cycle vertices)))))
```

```
(defmethod circumference :rectangle
  [{:keys [a b]}]
  (* 2 (+ a b)))
```

Multimethods

```
(defmulti draw-shape
  (fn [shape canvas] [(:kind shape) (:type canvas)]))

(defmethod draw-shape :default
  [shape canvas]
  (str "Drawing " (:kind shape) " on " (:type canvas)))
```

```
(defmethod draw-shape [:circle :print-canvas]
  [shape canvas]
  "Printing a circle")

(defmethod draw-shape [:rectangle :display-canvas]
  [shape canvas]
  "Showing a rectangle")
```

defrecord, deftype

Map

Record

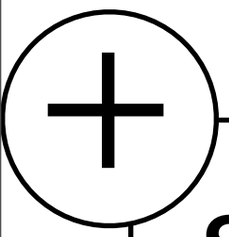
Type

Function

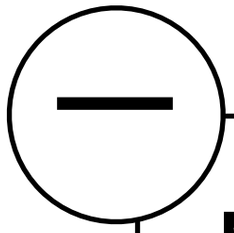
Multimethod

Protocol

defrecord

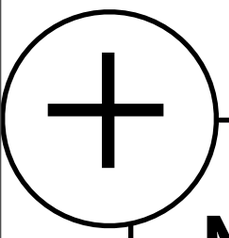


- Supports map access
- Flexible & extensible
- Convenience functions
- Better performance
- Platform integration
- Protocol support

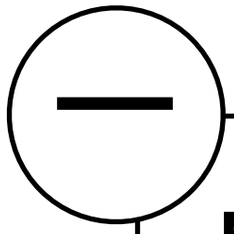


- No structural sharing
- Code overhead

deftype



- No generic overhead
- Convenience functions
- Best performance
- Platform integration
- Protocol support



- No structural sharing
- No map access
- Static & fixed
- Code overhead

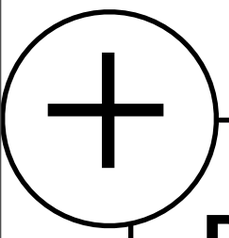
Protocols

```
(defprotocol Shape
  (area [shape])
  (circumference [shape]))

(defrecord Rectangle [vertices]
  Shape
  (area [shape] ...)
  (circumference [shape] ...))

(defrecord Circle [center r]
  Shape
  (area [shape] ...)
  (circumference [shape] ...))
```

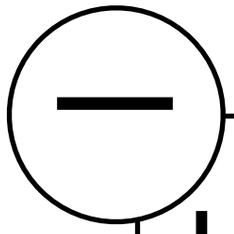
Protocols

```
(defprotocol ShapeStorage
  (read-from [storage])
  (write-to [storage shape]))

(extend-protocol ShapeStorage
  XmlStorage
    (read-from [storage] ...)
    (write-to [storage shape] ...)
  CouchDB
    (read-from [storage] ...)
    (write-to [storage shape] ...))

(extend-protocol ShapeStorage
  String
    (read-from [storage] ...)
    (write-to [storage shape] ...))
```

Protocols



- Performance
- Grouping
- Platform integration



- Limited dispatch
  (single arg, type-based)

Summary

Roadmap Recommendation

1 Namespaces, Functions, Persistent Data Structures

2 Multimethods

3 defrecord
defprotocol

4 deftype

Thanks!

Q&A

innoQ Deutschland GmbH

Krischerstr. 100
40789 Monheim am Rhein
Germany
Phone: +49 2173 3366-0
http://www.innoq.com

innoQ Schweiz GmbH

Gewerbestr. 11
CH-6330 Cham
Switzerland
Phone: +41 41 743 0116
info@innoq.com

Stefan Tilkov
stefan.tilkov@innoq.com
@stilkov