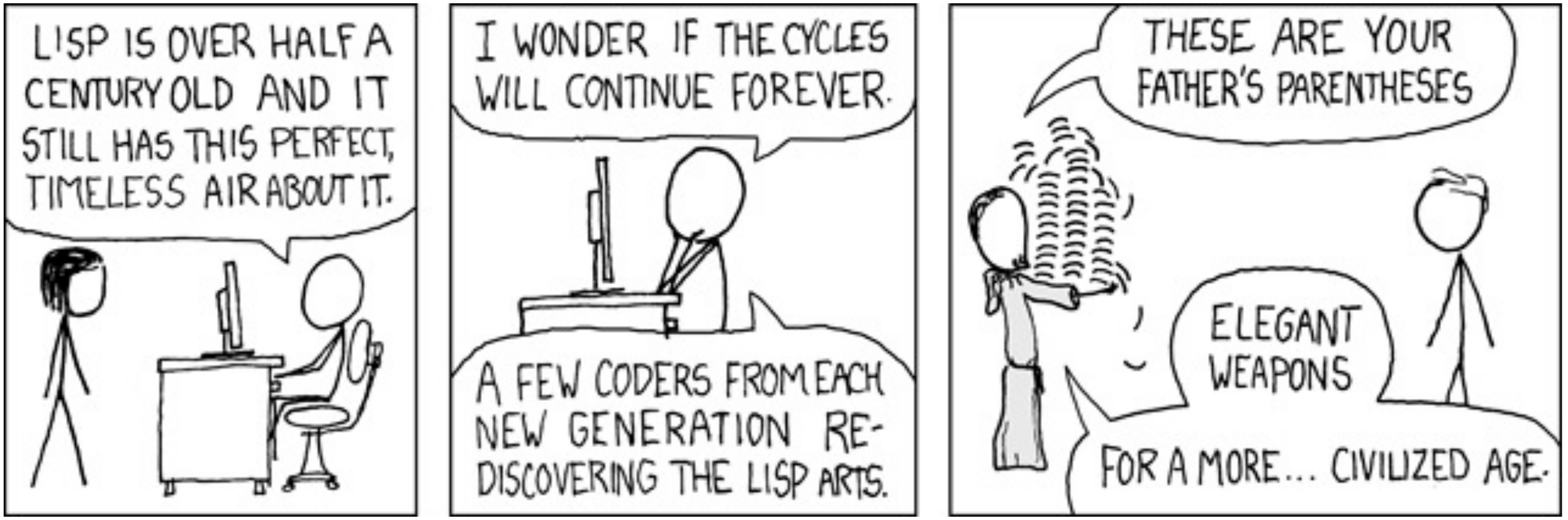


http://xkcd.com/297/

Clojure: Who's afraid of ((()))?

Silvia Schreier, @aivlis_s | Stefan Tilkov, @stilkov | innoQ

© 2012 innoQ Deutschland GmbH

Clojure

A practical Lisp variant for the JVM

Functional programming

Dynamic Typing

Full-featured macro system

Concurrent programming support

Bi-directional Java interop

Immutable persistent data structures

© 2011 innoQ Deutschland GmbH

Data structures

Numbers	`2 3 4 0.234` `3/5 -23989898928200930930902923 21`
Strings	`"Hello" "World"`
Characters	`\a \b \c`
Keywords	`:first :last`
Symbols	`a b c`
Regexps	`#"Ch.*se"`
Lists	`(a b c)` `((:first :last "Str" 3) (a b))`
Vectors	`[2 4 6 9 23]` `[2 4 6 [8 9] [10 11] 9 23]`
Maps	`{:de "Deutschland", :fr "France"}`
Sets	`#{"Bread" "Cheese" "Wine"}`

© 2011 innoQ Deutschland GmbH

Syntax

© 2011 innoQ Deutschland GmbH

“You’ve just seen it”

– Rich Hickey

© 2012 innoQ Deutschland GmbH

Syntax

```clojure
(def my-set #{:a :b :c :c :c}) ;; #{:a :b :c}
(def v [2 4 6 9 23])
(v 0) ;; 2
(v 2) ;; 6

(def people {:pg "Phillip", :st "Stefan"})
(people :st) ;; "Stefan"
(:pg people) ;; "Phillip"
(:xyz people) ;; nil

(+ 2 2) ;; 4
(+ 2 3 5 4) ;; 14
(class (/ 4 3)) ;; clojure.lang.Ratio
(* (/ 4 3) 3) ;; 4

(format "Hello, %s # %d" "world" 1)
```

© 2011 innoQ Deutschland GmbH

Functions

```
(fn [x] (format "The value is %s\n" x))
;; user$eval__1706$fn__1707@390b755d

((fn [x] (format "The value is %s\n" x)) "Hello")
;; "The value is Hello"
```

```
(def testfn (fn [x] (format "The value is %s\n" x)))
(testfn "Hello")

(defn testfn [x] (format "The value is %s\n" x))
(testfn "Hello")
```

© 2011 innoQ Deutschland GmbH

Lots of other cool stuff

- Persistent data structures
- Sequences
- Support for concurrent programming
- Destructuring
- List comprehensions
- Metadata
- Optional type information
- Multimethods
- Pre & Post Conditions
- Records/Protocols
- Extensive core and contrib libraries
- ...

© 2011 innoQ Deutschland GmbH

Syntax

Idioms

© 2012 innoQ Deutschland GmbH

OOP Thinking

model domains with classes & interfaces

encapsulate data in objects

prefer specific over generic solutions

explicitly provide for generic access

© 2012 innoQ Deutschland GmbH

Example

```
(ns sample.grep
  "A simple complete Clojure program."
  (:use [clojure.contrib.io :only [read-lines]])
  (:gen-class))

(defn numbered-lines [lines]
  (map vector (iterate inc 0) lines))

(defn grep-in-file [pattern file]
  {file (filter #(re-find pattern (second %)) (numbered-lines (read-lines file)))})

(defn grep-in-files [pattern files]
  (apply merge (map #(grep-in-file pattern %) files)))

(defn print-matches [matches]
  (doseq [[fname submatches] matches, [line-no, match] submatches]
    (println (str fname ":" line-no ":" match))))

(defn -main [pattern & files]
  (if (or (nil? pattern) (empty? files))
    (println "Usage: grep <pattern> <file...>")
    (do
      (println (format "grep started with pattern %s and file(s) %s"
                       pattern (apply str (interpose ", " files))))
      (print-matches (grep-in-files (re-pattern pattern) files))
      (println "Done."))))
```

© 2011 innoQ Deutschland GmbH

Data

© 2012 innoQ Deutschland GmbH

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])
```

© 2012 innoQ Deutschland GmbH

Data structures vs. objects

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

Immutable

Reusable

Compatible

© 2012 innoQ Deutschland GmbH

```
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
```

© 2012 innoQ Deutschland GmbH

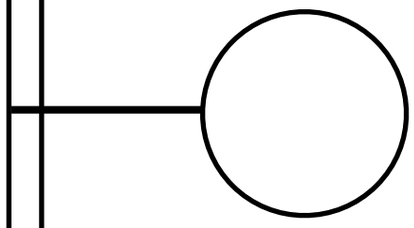
Maps

```
(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]}})
```

© 2012 innoQ Deutschland GmbH

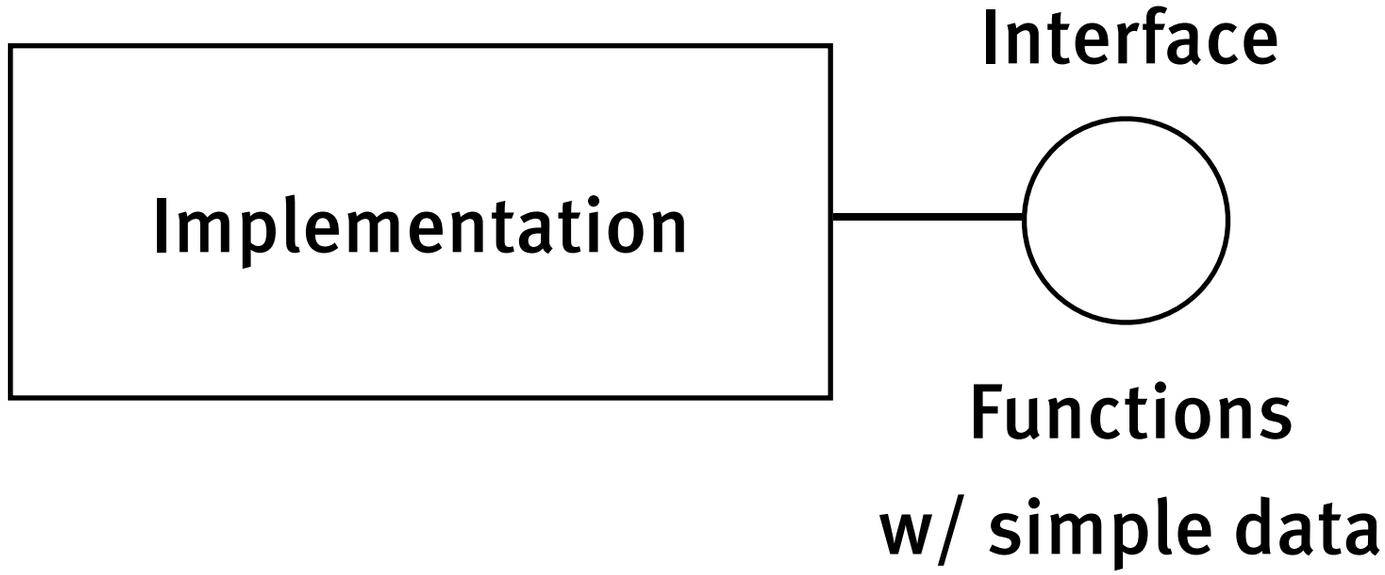
```clojure
(ns com.example.some-ns
  "Well-documented ns"
   (:use [com.example.n1 :only [xyz]])
   (:require [com.example.ns2 :as n2]))
```

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



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

© 2012 innoQ Deutschland GmbH



© 2012 innoQ Deutschland GmbH

And in real life?

```
package jax14.java;

public class IdentityService {

  public static IdentityService getInstance() {...}
  public Long authenticate(String email, String password) {...}
  public User getUserDetails(long userId) {...}
  public User updateUserDetails(User user) {...}

}
```

© 2012 innoQ Deutschland GmbH

Hello Java, Clojure is calling

```
(ns jax14.identity
  (:import [jax14.java IdentityService]))

(def id-service (IdentityService/getInstance))

(defn user-details [id]
  (.getUserDetails id-service id))

=> (user-details 1234)
#<User jax14.java.User@34a32e6f>

=> (.getName (user-details 1234))
"Ada Lovelace"

(defn auth [email password]
  (if-let [id (.authenticate id-service email password)]
    (user-details id)))

=> (auth "ada@lovelace.com" "password")
#<User jax14.java.User@5bce3f54>
```

© 2012 innoQ Deutschland GmbH

Beans and Maps

```
(defn user-details [id]
  (bean (.getUserDetails id-service id)))

=> (user-details 1234)
{:userId 1234,
 :name "Ada Lovelace",
 :email "ada@lovelace.com",
 :department #<Department jax14.java.Department@1772ed62>,
 :class jax14.java.User
}
```

© 2012 innoQ Deutschland GmbH

Beans and Maps

```
(ns jax14.identity
  (:require [clojure.java.data :as data])
  (:import [jax14.java IdentityService]))

(defn user-details [id]
  (data/from-java (.getUserDetails id-service id)))
```

```
=> (user-details 1234)
{:userId 1234,
 :name "Ada Lovelace",
 :email "ada@lovelace.com",
 :department {:name "IT",
              :departmentId 111}}
```

© 2012 innoQ Deutschland GmbH

Creating Objects

```
=> (new User 1234 "ada@lovelace.com")
#<User jax14.java.User@4018bf71>

=> (User. 1234 "ada@lovelace.com")
#<User jax14.java.User@7ed2df72>
```

© 2012 innoQ Deutschland GmbH

Creating Objects

```
(defn update-user [user]
  (let [dep (:department user)
        java-dep (Department.)
        java-user (User.)]
    (doto java-dep
      (.setName (:name dep))
      (.setDepartmentId (:departmentId dep)))
    (doto java-user
      (.setName (:name user))
      (.setEmail (:email user))
      (.setUserId (:userId user))
      (.setDepartment java-dep))
    (.updateUserDetails id-service java-user)))
```

© 2012 innoQ Deutschland GmbH

Creating Objects

```
=> (data/to-java User {:userId 1234
                       :email "ada@lovelace.com"})
#<User jax14.java.User@a8199db>


(defn update-user [user]
  (->> user
  (data/to-java User)
  (.updateUserDetails id-service)
  (data/from-java)))
```

© 2012 innoQ Deutschland GmbH

Creating Objects

```
=> (def user (auth "ada@lovelace.com" "password"))
#'user/user


=> (def modified-user
     (assoc user :name "Augusta Ada Byron King"))
#'user/modified-user
```

```
=> (update-user modified-user)
{:userId 1234,
 :name "Augusta Ada Byron King",
 :email "ada@lovelace.com",
 :department {:name "IT",
              :departmentId 111}}
```

© 2012 innoQ Deutschland GmbH

What else?

Exception handling

Helper functions for array conversion

Implement interfaces on the fly

Extend classes

Functions implement Runnable, Callable, Comparable

© 2012 innoQ Deutschland GmbH

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]}})
```

© 2012 innoQ Deutschland GmbH

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}
```

© 2012 innoQ Deutschland GmbH

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]}}
```

© 2012 innoQ Deutschland GmbH

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]}}
```

© 2012 innoQ Deutschland GmbH

(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)

© 2012 innoQ Deutschland GmbH

JSON Processing

```
{"projects":
  [{"kind":"fixed-price",
    "team":["john", "chuck"],
    "budget":100000,
    "id":"2",
    "description":"Development for Startup"},
   {"kind":"fixed-price",
    "team":["john"],
    "id":"3",
    "description":"Clojure Training"}],
"users":
 [{"email": "john@innoq.com", "name": "john"},
  {"email": "chuck@innoq.com", "name": "chuck"},
  {"email": "bill@innoq.com", "name": "bill"}]}
```

```
["john@innoq.com", "chuck@innoq.com"]
```

© 2012 innoQ Deutschland GmbH

JSON Processing

```
(ns jax14-json.project
  (:require [clojure.data.json :as json])

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

(defn users-to-email [names users]
  (map :email (filter #(contains? names (:name %)) users)))

(defn teamEmails [json]
  (let [info (json/read-str json :key-fn keyword)
        projects (:projects info)
        all-users (:users info)
        members (all-members projects)]
    (users-to-email members all-users)))
```

© 2012 innoQ Deutschland GmbH

Clojure Java API

```
import clojure.java.api.Clojure;
import clojure.lang.IFn;
import java.util.List;

IFn require = Clojure.var("clojure.core", "require");
require.invoke(Clojure.read("jax14-json.project"));
IFn teamEmails = Clojure.var("jax14-json.project", "teamEmails");

Object result = teamEmails.invoke("my json");
System.out.println(result.getClass());
for (String email : (List<String>) result) {
  System.out.println(email);
}
```

© 2012 innoQ Deutschland GmbH

Ahead of time

```
(ns jax14-json.project
  (:require [clojure.data.json :as json])
  (:gen-class
    :name jax14_json.project.JsonTransformer
    :prefix ""
   :methods
     [^{:static true}
      [teamEmails [String] java.util.List]]))
```

© 2012 innoQ Deutschland GmbH

Ahead of time

```
import java.util.List;
import jax14_json.project.JsonTransformer;

List<String> result =  JsonTransformer.teamEmails(json);
System.out.println(result);
for(String email : result) {
  System.out.println(email);
}
```

© 2012 innoQ Deutschland GmbH

Multimethods

© 2011 innoQ Deutschland GmbH

Methods vs. Multimethods

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

© 2012 innoQ Deutschland GmbH

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]}})
```

© 2012 innoQ Deutschland GmbH

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)))
```

© 2012 innoQ Deutschland GmbH

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)))
```

© 2012 innoQ Deutschland GmbH

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)))
```

© 2012 innoQ Deutschland GmbH

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")
```

© 2012 innoQ Deutschland GmbH

Summary

© 2012 innoQ Deutschland GmbH

Functional programming is different (for a reason)

© 2012 innoQ Deutschland GmbH

Embrace data structures

© 2012 innoQ Deutschland GmbH

There’s no good reason not to use Clojure

© 2012 innoQ Deutschland GmbH

Thanks!

Q&A

Silvia Schreier

silvia.schreier@innoq.com

@aivlis_s

Stefan Tilkov

stefan.tilkov@innoq.com

@stilkov

innoQ

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

© 2012 innoQ Deutschland GmbH