Introduction to Clojure

Petri Louhelainen, @plouh
Petteri Parkkila, @pparkkila

Reaktor

Agenda

- Hands on: Clojure basic syntax
  - Lists and Sequences
  - Working with Sequences
- BREAK!
- More hands-on exercises
- Q & A

THE WEB

You're doing it wrong

LISt Processing in Clojure

Lists and Sequences

- Here’s a familiar looking list:

```
(+ 1 2 3)
```

- Now experiment with lists and lists within list

Lists and Sequences

- Here's a familiar looking list:

```
(- 5 (* 2 3))
```

- Like this!

Lists and Sequences

- Creating a sequence of values

```
(list 1 2 3 (+ 1 3))
```

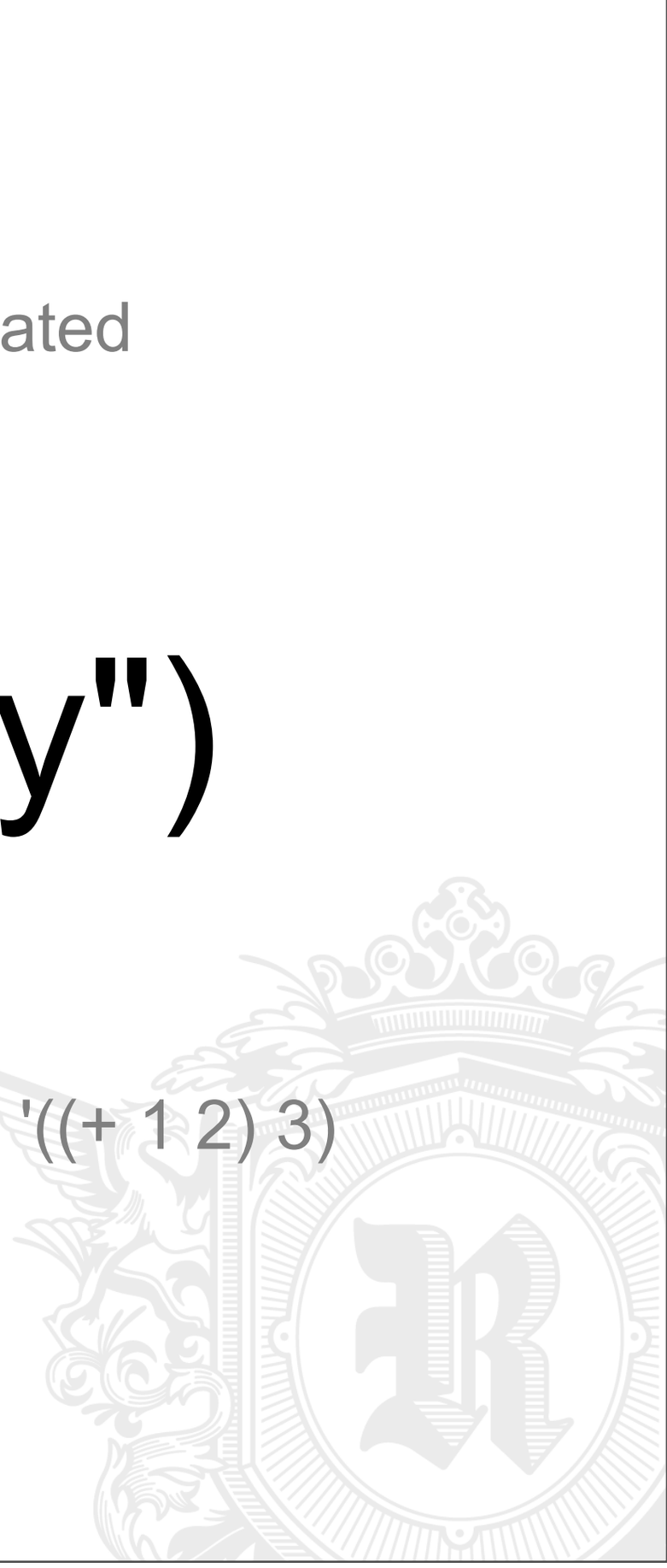
- Try creating some lists of your own

Lists and Sequences

- Lists escaped with single quote are not evaluated

'("hello" "devday")

- Try the difference between (list (+ 1 2) 3) and '((+ 1 2) 3)



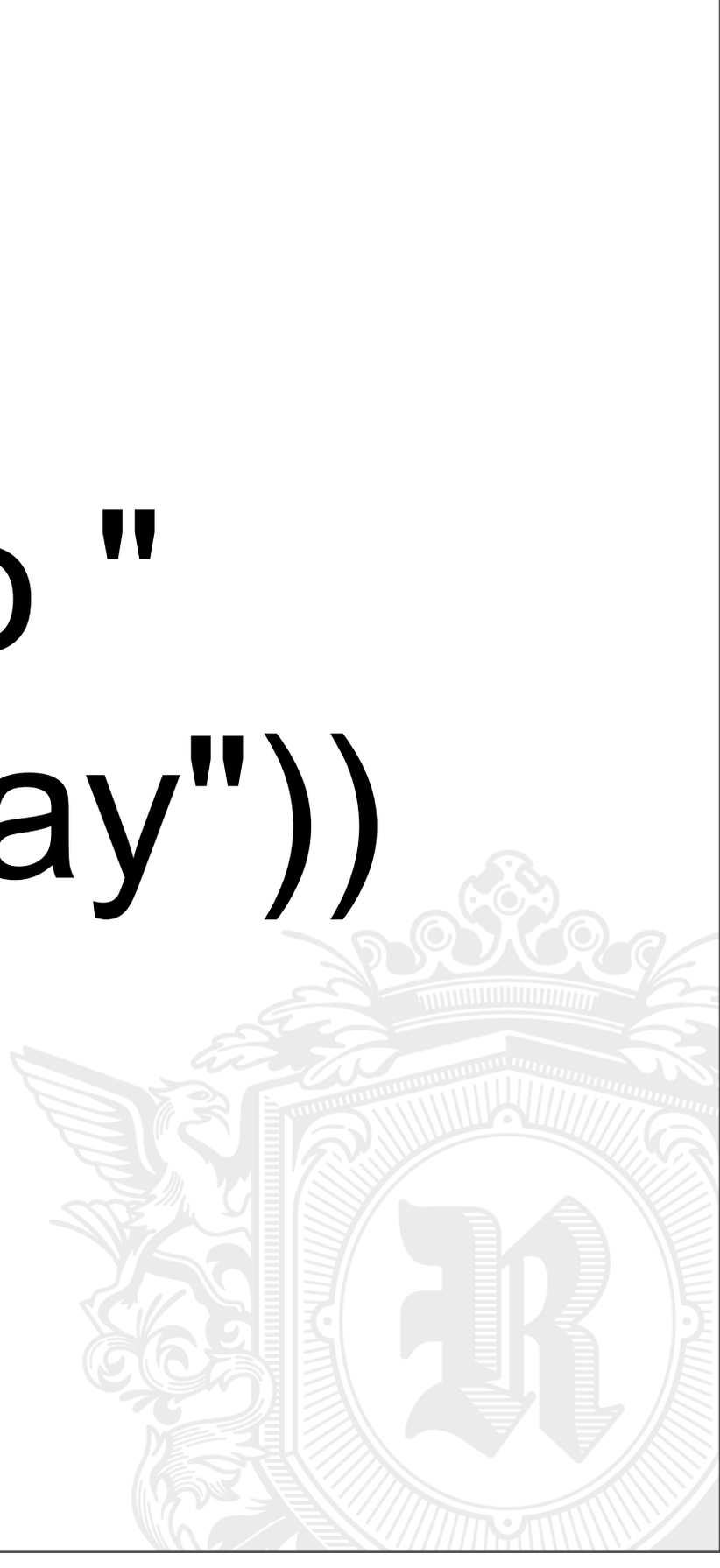
Reaktor

Lists and Sequences

- Creating a vector from a list

```
(vec (list "hello "
           "devday"))
```

- First item of a vector is not a function

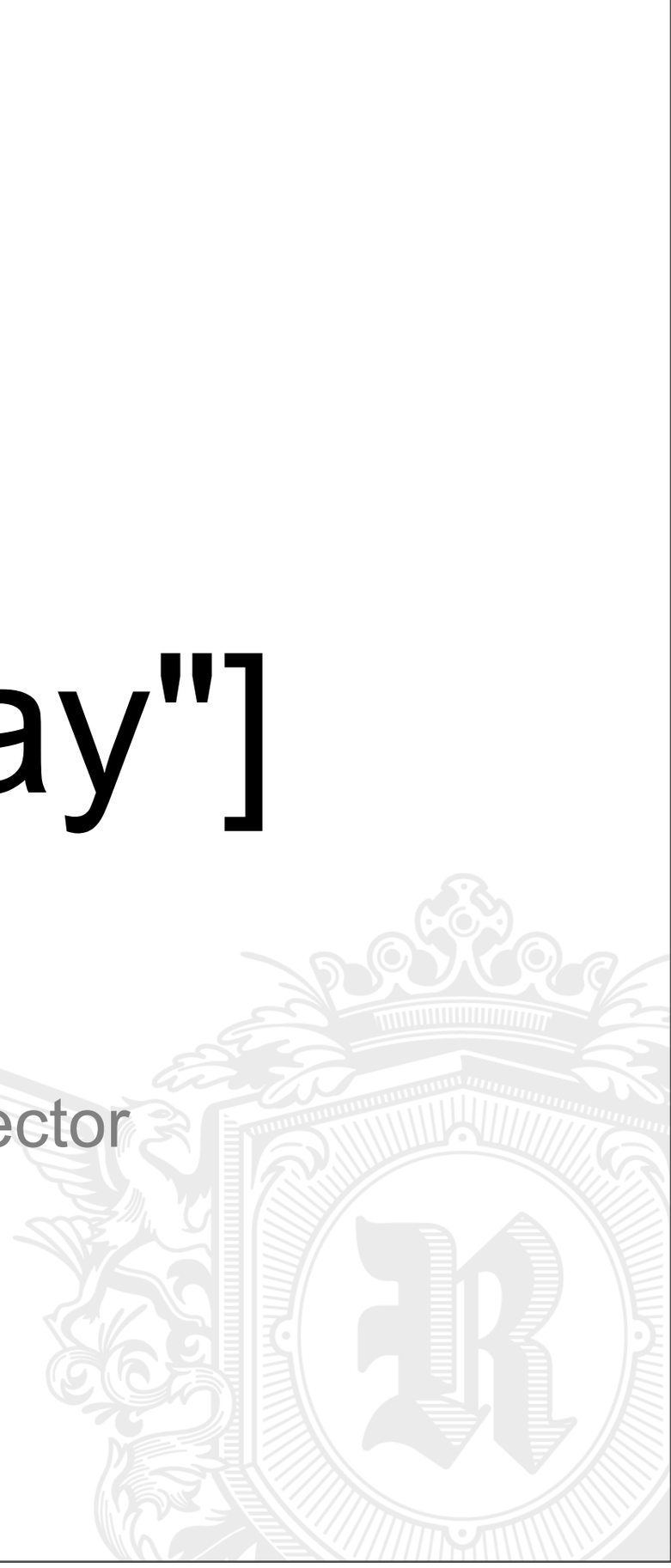


Lists and Sequences

- Easier syntax

```
["hello " "devday"]
```

- *vec* should be used when converting list to vector



Lists and Sequences

- Set is a sequence where one value may occur only once

```
(hash-set 1 2 3 1)
```

- Evaluates to **#{1 2 3}** which is shorthand for hash set

Lists and Sequences

- Map is an associative sequence

```
(hash-map
                    "a" 2 "b" 3)
```

- Evaluates to **{“a” 2 “b” 3}** which is shorthand for map

Lists and Sequences

- Accessing values from map

```
(def m {"a" 1 "b" 2})
(m "a")
```

- Map is also a function that takes key as a parameter and returns associated value

Reaktor

Lists and Sequences

- Keywords can be used for easier access

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

- Keyword is also a function that retrieves corresponding value from a map given as argument

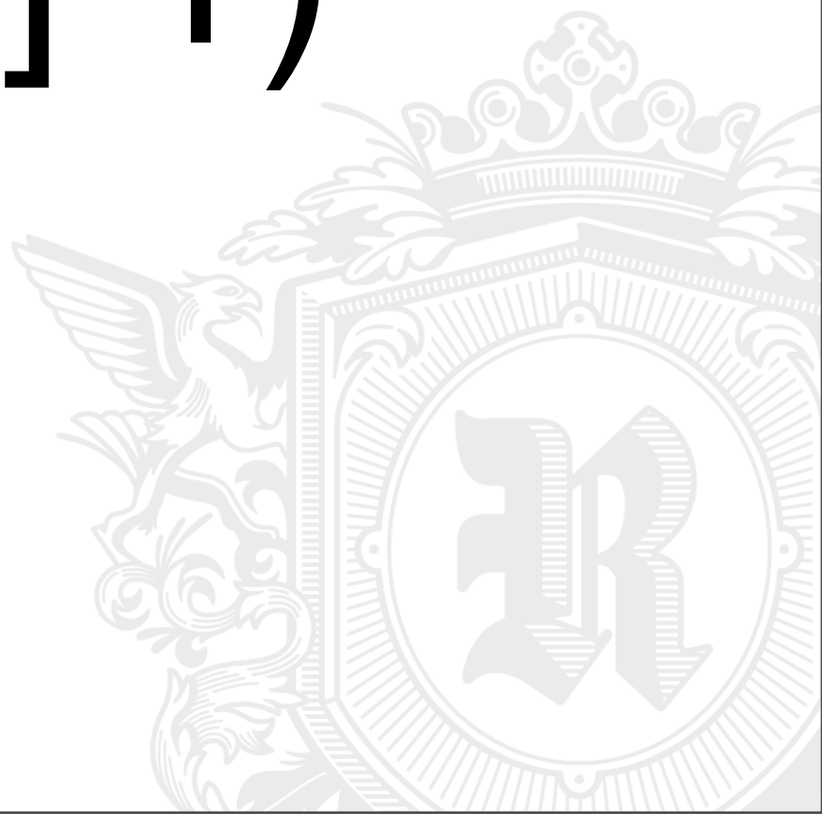
Reaktor

Lists and Sequences

- Accessing first or n’th item of a sequence

```
(first [1 2 3 4])
(nth ["a" "b" "c"] 1)
```

- Try also **second**, **last**, **rest** and **reverse**



Lists and Sequences

- Retrieve all the map keys as a sequence

```
(keys {:who "pete"
       :greet "hello"})
```

- **vals** does the same for values

Write an expression that returns the last item from a sequence without using ***last***

Reaktor

Write an expression that returns the next to last item from a sequence

Reaktor

Working with Sequences

Reaktor

Working with Sequences

- Defining anonymous functions

```
(fn [x] (* x x))
```

- Doesn't really do much?
  - Try it as a first value to a list with one number

Reaktor

Working with Sequences

- Defining anonymous functions

```
((fn [x] (* x x)) 4)
```

- Can be used wherever a function is required

Reaktor

Working with Sequences

- Defining named functions

```
(def sqr
  (fn [x] (* x x)))
```

- Now you can use it like this: **(sqr 2)**

Working with Sequences

- Shorter syntax for defining functions

```
(defn sqr [x] (* x x))
```

- You can still call it like this: **(sqr 2)**

Reaktor

Working with Sequences

- Map applies function to every value in the sequence

```
(map sqr [1 2 3 4])
```

- Map returns a new sequence, in this case **(1 4 9 16)**

Working with Sequences

- Filter returns sequence that satisfies the condition

```
(filter even?
        [1 2 3 4])
```

- Functions used as test conditions usually end with ? like **even?** or **odd?**
- **remove** is the opposite of **filter**

Reaktor

Working with Sequences

- Reduce a value from function applied over sequence

```
(reduce min 3
        [1 2 5 4])
```

- second parameter is ***memo***
- Function takes two parameters, ***memo*** and item from list, producing the next ***memo***

Reaktor

Lists and Sequences

- Applying a function to a sequence

```
(apply min
       [1 4 3 2 5])
```

- Apply evaluates the function with list of parameters like **(str "hello " "devday")**

Working with Sequences

- Conditional processing

```
(if (even? 2)
    "even"
    "that's odd")
```

- Experiment with different values for test. How **false** and **nil** work differently to everything else?
- **when** omits the else branch

Reaktor

Working with Sequences

- Testing for multiple conditions

```
(defn pos-or-neg[x]
(cond (> 0 x) "positive"
      (< 0 x) "negative"
      :else "neither"))
```

Working with Sequences

- Create a sequence of integers

```
(range 10)
```

- Without arguments it will create an endless sequence

Reaktor

Working with Sequences

- Grouping and slicing lists

```
(partition 5 3
           (range 10))
```

- Test also partitioning without second parameter

Reaktor

When given 500 character length string of digits, find the maximum value when multiplying five consecutive digits

Reaktor

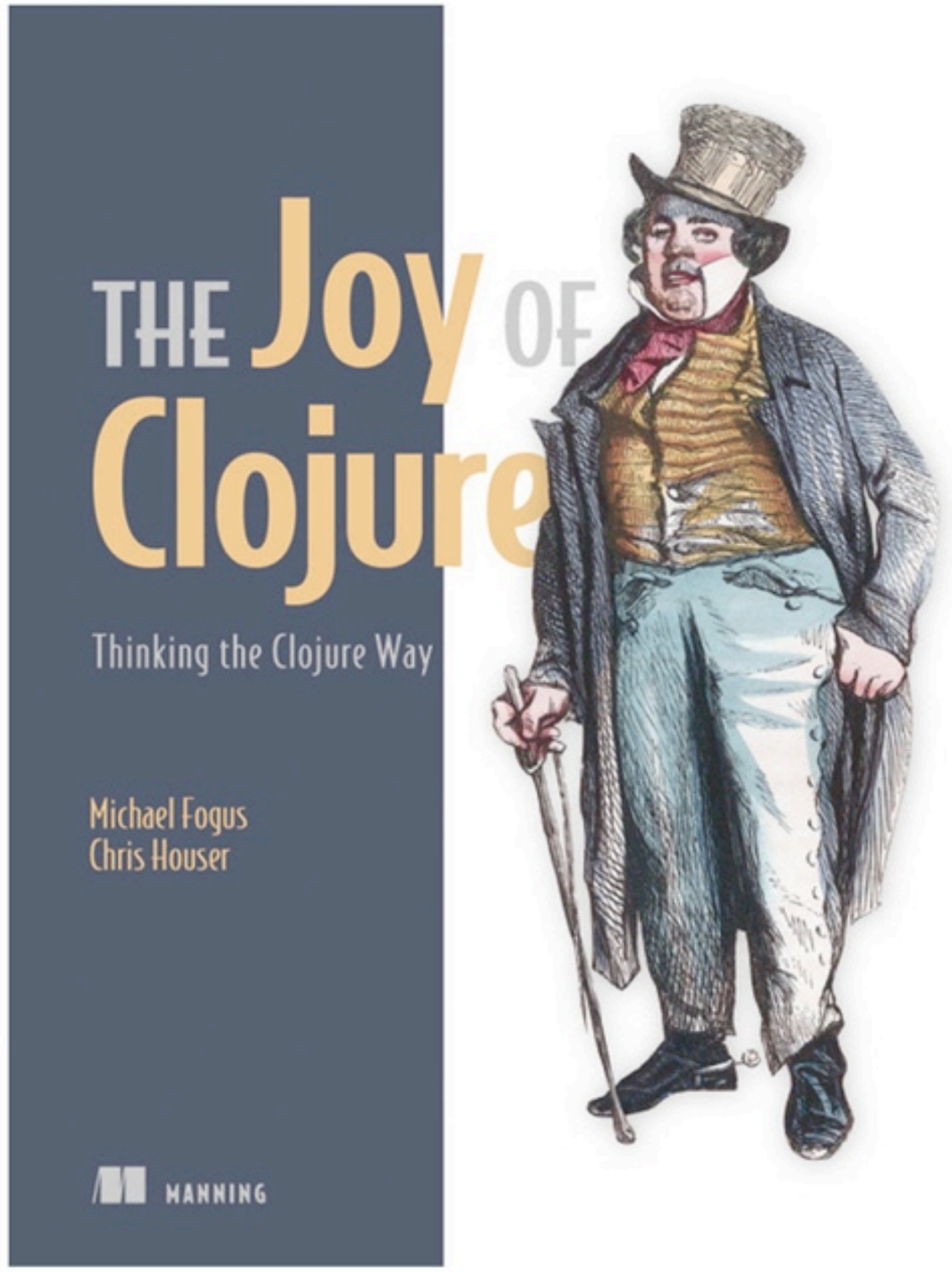
Questions?

Why Clojure (and not Scala)?

- Clojure maps are perfect match for RESTFUL web services (e.g. JSON APIs)
- Easier to write decent functional programs
- Doesn’t try to make things easy to Java developers
- Maintains Java interoperability

Reaktor

Further reading



Reaktor

Thank

You!

Petri Louhelainen
@plouh, petri.louhelainen@reaktor.fi
Petteri Parkkila
@pparkkila, petteri.parkkila@reaktor.fi

Reaktor