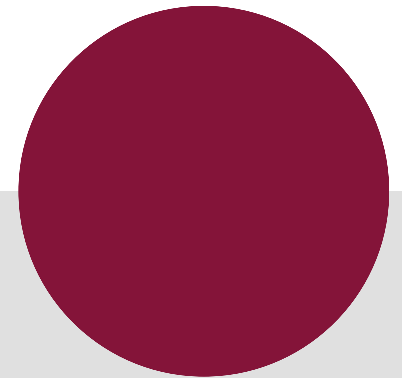
.consulting .solutions .partnership



Clojure by Example

A practical introduction to Clojure on the JVM

Clojure By Example

1 Functional Progamming Concepts 3

2 Clojure Basics 4

3 Clojure Examples 5

4 References 6

Clojure By Example

1 **Functional Progamming Concepts** **3**

2 Clojure Basics 4

3 Clojure Examples 5

4 References 6

.msg

Functional Programming Concepts

- **Functional Programming**
  - uses functions as primary programming construct
  - not imperative
  - based on **Lambda Calculus** by Alonso Church

- **Functional Programming Languages**
  - **Lisp**
    - is the Prototype of functional programming languages
    - 2nd oldest higher programming language (McCarthy, 1959)
  - **Clojure** (Lisp on the JVM)
  - **Haskell**
  - **Scala**
  - **JavaScript**
  - **Java 8**

.msg

Functional Programming Concepts

- **Functions as first class citizens**
  - stored in Variables
  - used as parameters of other functions
  - used as return values of other functions
- **Pure Functions / Referential transparency**
  - functions **without side effects** (e.g. IO)
  - given the same input **always** return the same output
  - easy to reason about
  - easily parallelizable
- **Lambdas are anonymous functions**
  - created as needed

Clojure By Example

1 Functional Progamming Concepts 3

2 Clojure Basics 4

3 Clojure Examples 5

4 References 6

.msg

Clojure Basics – Why Clojure?

A language that doesn't affect the way you think about programming, is not worth knowing.

Alan Perlis

- **Clojure is functional**
  - Makes you think different, if you‘re coming from OO
- **Clojure is running on the JVM**
  - or in the Browser with ClojureScript
- **Clojure embraces the Host Platform**
  - Integrates well with Java on the JVM
  - Works with all the Java Libraries ()
- **Clojure is Open Source**
  - Licensed under EPL 1.0

.msg

Clojure Basics – Why Clojure?

- **Clojure has almost no syntax**
  - Like an AST written out (in list representation)

- **Clojure is homoiconic**
  - Code is data is code is data …
  - Clojure code can transform clojure code as clojure data

.msg

Clojure Basics - Java/Clojure Syntax Comparison

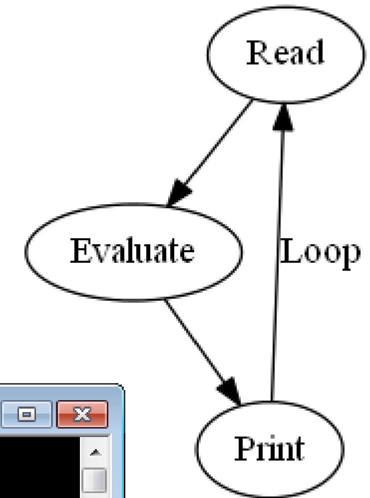
- **Java/Clojure Syntax Comparison**

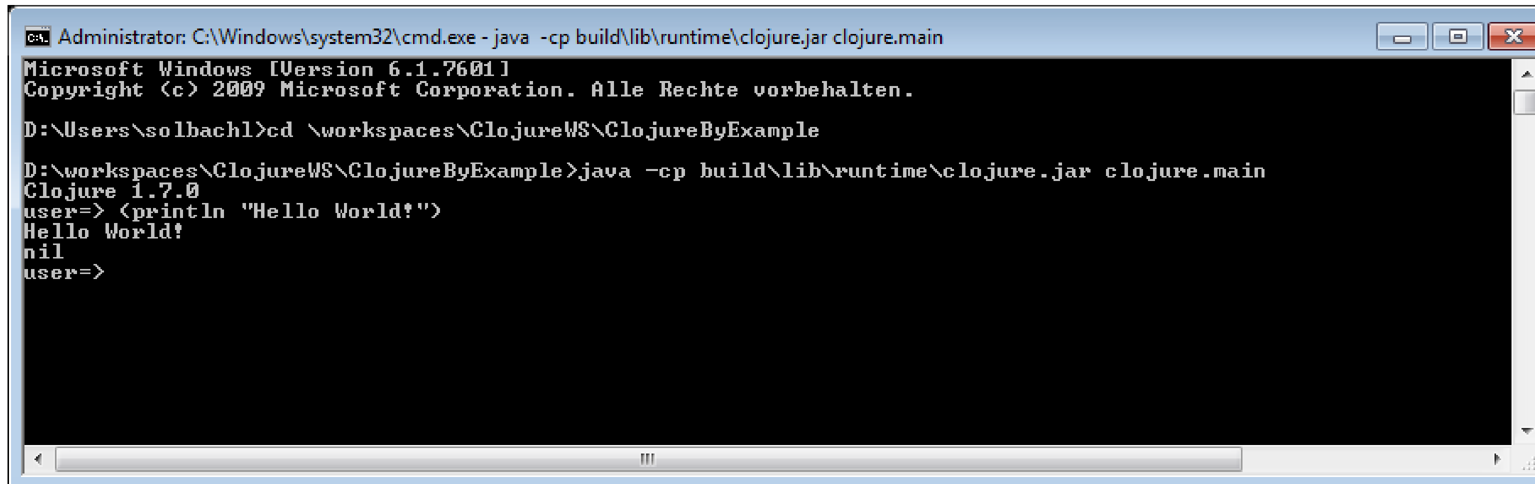
	Operators	Methods/functions
Java	2 + 3 + 4	sb.append("Hello!")
Clojure	(+ 2 3 4)	(append sb "Hello!")

Clojure Basics – The REPL

- Clojure REPL
  - Read-Evaluate-Print-Loop
  - Interactive development
  - Instant Feedback

```
> java -cp clojure-1.7.0.jar clojure.main
```





.msg

Clojure Basics – Clojure Development

- **Eclipse with Counterclockwise**
  - http://eclipse.org/
  - http://doc.ccw-ide.org/
- **Emacs/Slime**
- **VIM**, Lighttable

- **Leiningen** – Clojure Build System
  - Standard Build System for Clojure projects
  - https://github.com/technomancy/leiningen

- **Baumeister** – Multi Language Build System
  - Not for general consumption yet (works on my machine ;-))
  - https://github.com/lsolbach/Baumeister

.msg

Clojure Basics – Symbols and Keywords

- **Symbols**
  - can be defined with def
  - evaluate to the bound value

```
 > (def hello 10)
=> user/hello
 > hello
=> 10
```

- **Keywords**
  - evaluate to themselves
  - can be used as map keys

```
 > :a
=> :a
```

.msg

Clojure Basics – Numeric Literals

- Long

```
> 1
```

- BigInteger

```
> 1M
```

- Double

```
> 1.0
```

- BigDecimal

```
> 1.0N
```

- Rational

```
> 1/3
```

Clojure Basics – String and Character Literals

- String Literals

```
> „Hello World!"
```

- Multiline String Literals

```
> „Hello,
   World!"
```

- Character Literals

```
> \A
```

- Regular Expression Literals

```
> #"(\d*)"
```

.msg

Clojure Basics – Collection Literals

- List Literals

```
> '(1 2 3 4)
```

- Vector Literals

```
> [1 2 3 4]
```

- Set Literals

```
> #{1 2 3 4}
```

- Map Literals

```
> {:a 5 :b 3}
```

.msg

Clojure Basics – Functions

- Basic building block of clojure programs
- Evaluate the arguments before evaluating the body

```
 > (defn avg „Averaging the values of the coll.“
         [coll]
         (/ (reduce + coll) (count coll)))
 > (avg [1 2 3 4 5])
=> 3
```

- Lambdas – Anonymous Functions

```
 > ((fn [coll] (/ (reduce + coll) (count coll))) [1 2 3 4 5])
=> 3
 > (#(/ (reduce + %) (count %))) [1 2 3 4 5])
=> 3
```

.msg

Clojure Basics – Functions

- **Higher order functions**
  - take functions as parameters
  - return functions

- **(map f coll)** returns a collection with ***f*** applied to each element ***e*** of ***coll***

```
(map to-upper [„a“ „b“ „c“]) -> [„A“ „B“ „C“]
```

- **(filter f coll)** returns a collection with all elements ***e*** for which ***f(e)*** is true

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

- **reduce(f, coll)** returns the result of the combination of subsequent elements with ***f***

```
(reduce + [1 2 3 4]) -> 10
```

Clojure Basics – Sequence Abstraction

It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures.

Alan Perlis

- Sequences are a core data structure
- Every Collection in Clojure implements ISeq
- There **are** about 100 functions working on sequences in clojure.core alone

Clojure Basics – Sequence Abstraction

- Returning the first element of the sequence

```
 > (first [1 2 3 4])
=> 1
```

- Returning the sequence without the first element

```
 > (rest [1 2 3 4])
=> [2 3 4]
```

- Adding an element at the head of the sequence

```
 > (cons 0 [1 2 3 4])
=> [0 1 2 3 4]
```

- Creating a sequence from a collection

```
 > (seq [1 2 3 4 ])
```

.msg

Clojure Basics – Persistent Data

- Clojure data structures are immutable
- Functions manipulating data structures return new data structures

```
 > (def m {:a 1 :b 2})
 > (assoc m :c 4)
=> {:a 1 :b 2 :c 4} ; result is a new map
 > m
=> {:a 1 :b 2} ; m is unchanged
```

- Performant implementation of the data structures with structural sharing
  - Implemented with B-Trees

.msg

Clojure Basics – Reference Types

- Use reference types for mutable state
- Software Transactional Memory (**STM**) provides „In Memory Transactions“

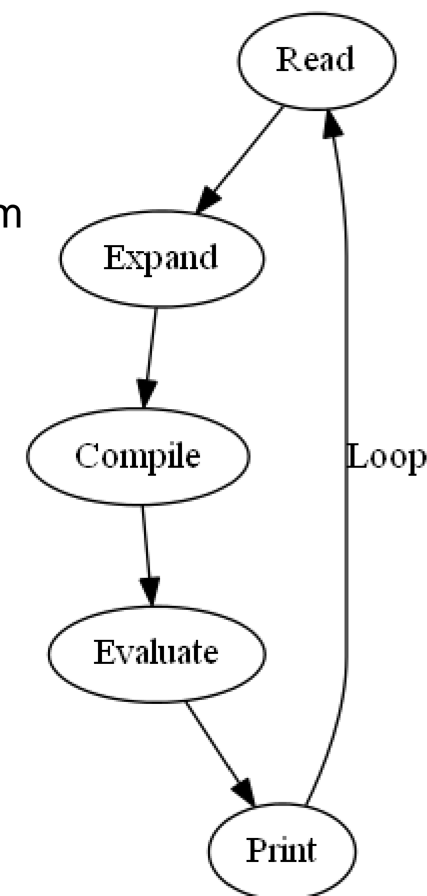
	Ref	Agent	Atom	Var
Coordinated	x			
Asynchronous		x		
Retriable	x		x	
Thread-local				x

.msg

Clojure Basics – Clojure Macros

- Clojure is **homoiconic**
- **Code is Data is Code** …
  - Clojure code can transform clojure code as clojure data
- Macros provide an extremly powerful **compile time templating** mechanism
- Macros do **not** evaluate the parameters on expansion
  - unlike functions, which evaluate the arguments before evaluation
- Will be expanded by the reader in the **Expand** phase

```
 > (defmacro unless [expr form]
         `(if ~expr nil ~form))
 > (macroexpand-1 '(unless false (println „should be printed“)))
=> (if false nil (println „should be printed“))
```



Clojure By Example

1 Functional Progamming Concepts 3

2 Clojure Basics 4

3 Clojure Examples 5

4 References 6

Clojure By Example

1 Functional Progamming Concepts 3

2 Clojure Basics 4

3 Clojure Examples 5

4 **References** **6**

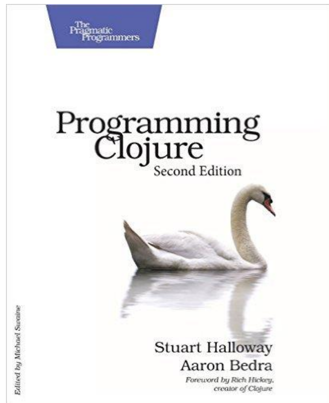
.msg

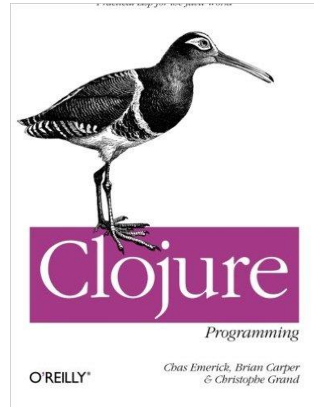
Clojure References

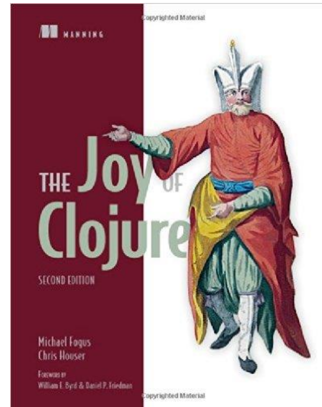
- Clojure Home Page
  - http://clojure.org
- Clojure Project on GitHub
  - https://github.com/clojure
- Clojure IRC Channel
  - irc://irc.freenode.net/#clojure

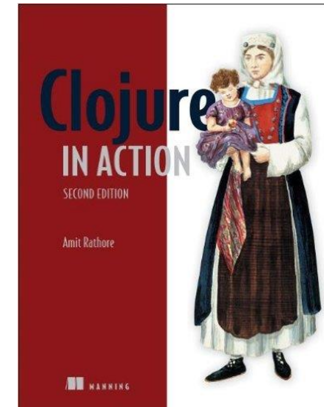
.msg

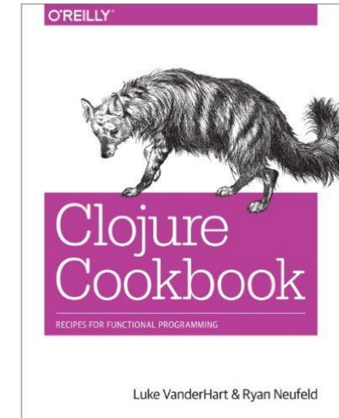
Clojure Literature Selection

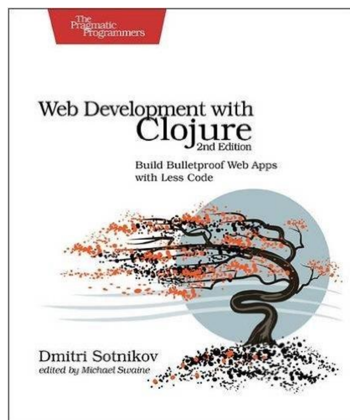


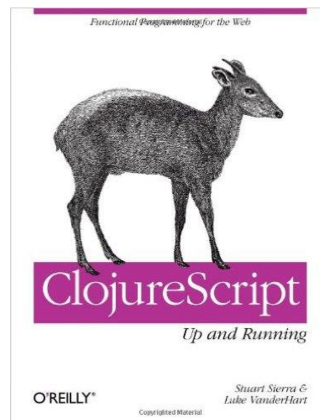


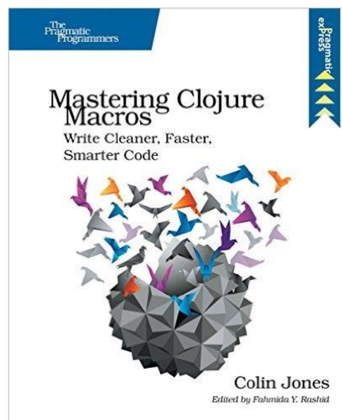


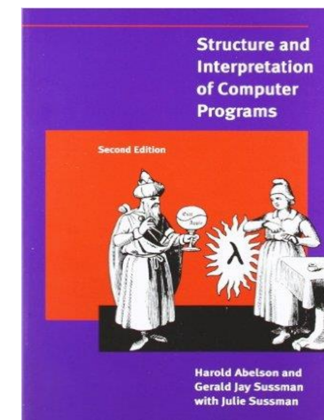


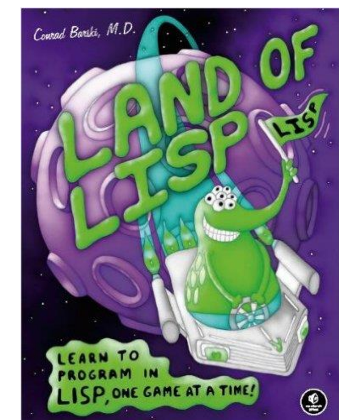










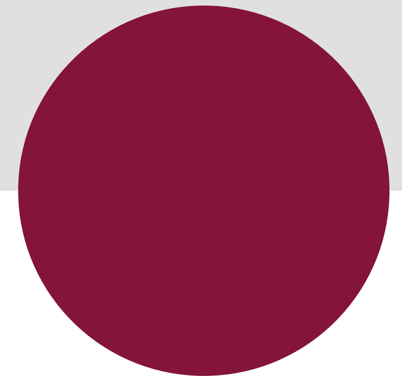


Ludger Solbach
Senior IT Consultant

+49 (0) 711 / 94958 681
Ludger.solbach@msg-systems.com

msg systems ag (Headquarters)
Robert-Buerkle-Str. 1, 85737 Ismaning
Germany

www.msg-systems.com



.msg .consulting .solutions .partnership