Clojure

01 - Vorbesprechung

Beliankou - Beyer - Naumann

Wintersemester 2016

Warum Clojure?

- ☐ *Clojure* ist eine funktionale Programmiersprache und funktionale Programme ist oft klarer strukturiert, einfacher zu schreiben und zu verstehen und zu warten als ein einer objektorientierten/ imperativen Programmiersprache geschrieben Programme.

- ☐ *Clojure* ist eine moderne LISP-Variante, die einerseits die Flexibilität von LISP mit eine Reihe moderner Eigenschaften (z.B. *software transactional memory system*) verbindet.

- ☐ *Clojure* läuft auf der JVM und ermöglicht einen direkten Zugriff auf alle wichtigen JAVA-Bibliotheken, -Klassen, -Methoden etc.

- ☐ *Clojure* unterstützt parallele Programmierung: Es enthält ein *software transactional memory* mit geeigneten Referenztypen.

Warum Clojure?

JAVA (Apache Commons)

```
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;
    }
}
```

Clojure

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

Arbeitsumgebung einrichten

1. Leiningen (http://leiningen.org)

   - [ ] Software-Paket zum Anlegen und Verwalten von Clojure-Projekten

   - [ ] Installation:
     1 Download the lein script (or on Windows lein.bat)
     2 Place it on your $PATH where your shell can find it (eg. ~/bin)
     3 Set it to be executable (chmod a+x ~/bin/lein)
     4 Run it (lein) and it will download the self-install package

   - [ ] Erzeugung eines Projektes:
     $ lein new [*Projekttyp*] [*Projektname*]

Arbeitsumgebung einrichten

1. Leiningen (http://leiningen.org)

   - *$ lein new app project_1*
     Dadurch wird ein Verzeichnis project_1 mit folgenden Dateien bzw. Verzeichnissen angelegt:

```
-rw-r--r--  1 Sven  staff  11219 19 Sep 15:50 LICENSE
-rw-r--r--  1 Sven  staff    469 19 Sep 15:50 README.md
drwxr-xr-x  3 Sven  staff    102 19 Sep 15:50 doc
-rw-r--r--  1 Sven  staff    365 19 Sep 15:50 project.clj
-rw-r--r--  1 Sven  staff   1344 19 Sep 15:51 project_1.iml
drwxr-xr-x  2 Sven  staff     68 19 Sep 15:50 resources
drwxr-xr-x  3 Sven  staff    102 19 Sep 15:50 src
drwxr-xr-x  3 Sven  staff    102 19 Sep 15:50 target
drwxr-xr-x  3 Sven  staff    102 19 Sep 15:50 test
```

   - Dateien mit Programmdaten sollten im Verzeichnis src/project_1 abgelegt werden, das zunächst nur eine Datei mit dem Namen core.clj enthält:

Arbeitsumgebung einrichten

```
(ns project-1.core
  (:gen-class))

(defn -main
  "I don't do a whole lot ... yet."
  [& args]
  (println "Hello, World!"))
```

Arbeitsumgebung einrichten

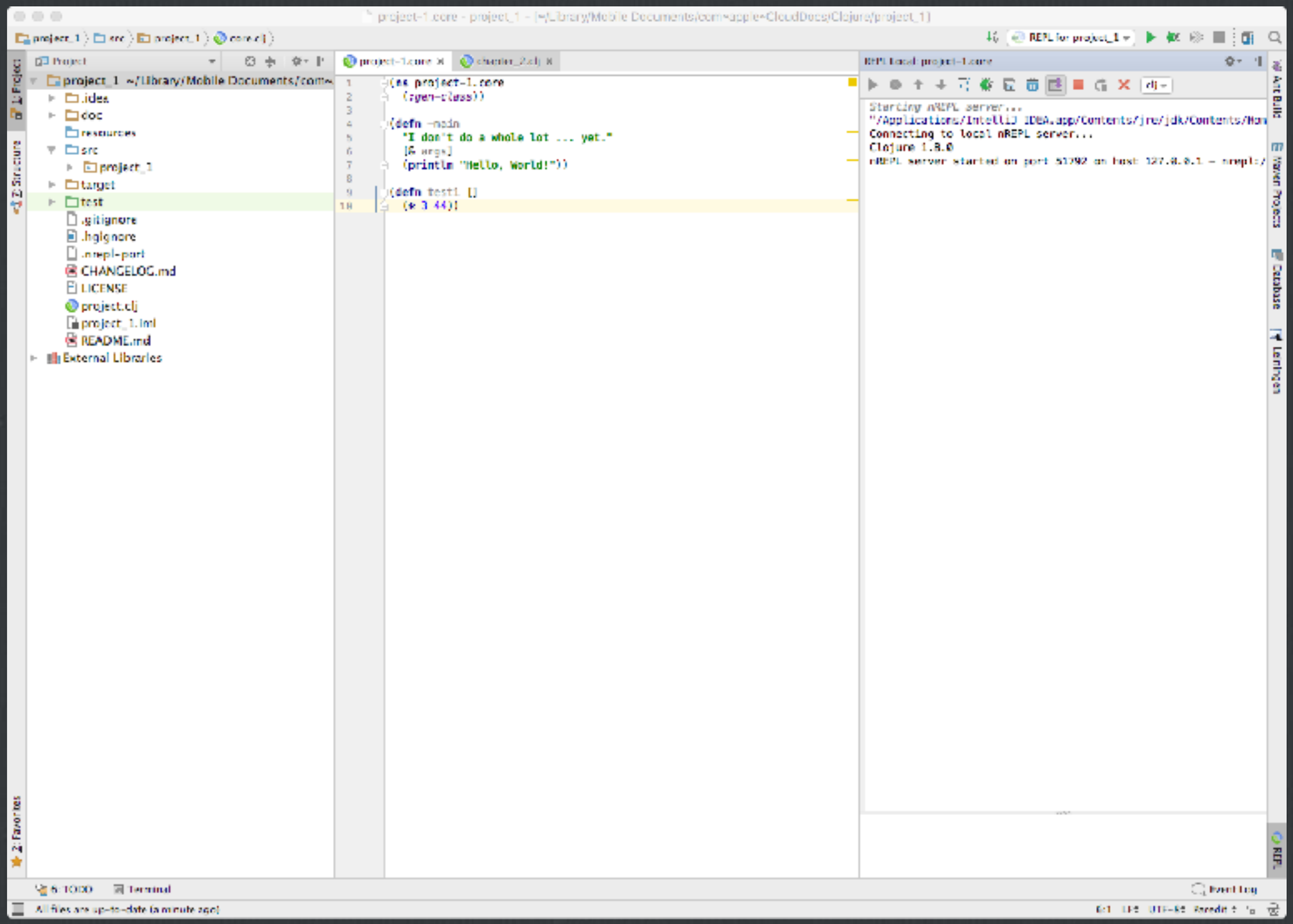
2. IntelliJ IDEA (https://www.jetbrains.com)

   - ☐ JAVA-IDE der Firma JetBrains, die durch Verwendung geeigneter Plugins für diverse andere Programmiersprachen (*Ruby*, *Clojure*, etc.) genutzt werden kann.
   - ☐ Einmalige Registrierung ermöglicht kostenlose Nutzung aller von dieser Firma angebotenen Produkte (https://www.jetbrains.com/shop/eform/students).

3. Cursive (https://cursive-ide.com)

   - ☐ Als Plugin für IntelliJ IDEA verfügbar.
   - ☐ (Kostenlose) Registrierung erforderlich.
   - ☐ Installation beschrieben auf: https://cursive-ide.com/userguide/

Arbeitsumgebung einrichten



Ressourcen

1. Bücher

   - [ ] Stuart Halloway, Aaron Bedarf (2012). Programming Clojure. 2. ed., The Pragmatic Programmers.
   - [ ] Michael Bolin (2010). Closure: The Definite Guide. O'Reilly
   - [ ] Daniel Higginbotham (2015). Clojure for the Brave and True. No Starch Press

2. Webseiten/Archive

   - [ ] http://clojure.org/
   - [ ] https://www.reddit.com/r/Clojure/

Read-Eval-Print-Loop

Zwar werden Clojure-Programme kompiliert, aber es steht auch ein Interpreter in Form einer Read-Eval-Print-Loop (kurz: REPL) zur Verfügung, der zum interaktiven Entwickeln und Testen von Funktionen und Programmen genutzt werden kann:

```
(+ 3 4)
=> 7

(println "hello world")
=> hello world

(defn hello [name] (str "Hello, " name))
=> #'project-1.core/hello

(hello "Karen")
=> "Hello, Karen"
```

Read-Eval-Print-Loop

Es gibt eine Reihe spezieller Variablen, die die Nutzung der REPL vereinfachen: So werden z.B. die Variablen *1, *2 und *3 an die Werte der letzten drei Formen gebunden

```
*1
=> "Hello, Karen"
```

Mit den Funktionen doc und find-doc kann man sich die Dokumentation von Funktionen und Variablen anzeigen lassen:

```
(doc println)
clojure.core/println
([& more])
  Same as print followed by (newline)
=> nil
```

Read-Eval-Print-Loop

```
(doc hello)
project-1.core/hello
([name])
  nil
=> nil
```

Um selbstdefinierten Funktionen eine aussagekräftige Beschreibung zuzuordnen, kann man hinter der Parameterliste eine Dokumentationszeichenkette einfügen:

```
(defn hello "Diese Funktion gibt einen String der Form 'Hello NAME' aus.“ [name]
    (str "Hello, " name))

(doc hello)
project-1.core/hello
([name])
  Diese Funktion gibt einen String der Form 'Hello NAME' aus.
=> nil
```

Bibliotheken

- Clojure Funktionen sind in Bibliotheken organisiert und mit jeder Bibliothek ist ein eigener Namensraum assoziiert (vergleichbar den *packages* in JAVA).
- Eine Bibliothek kann geladen werden mit Hilfe von require:
  (require quoted-namespace-symbol)
  user=> *(require 'clojure.java.io)*
  -> nil
- Funktionen aus einer mit require geladenen Bibliothek müssen unter Verwendung des Namensraumpräfixes aufgerufen werden:
  user=> *(require 'examples.introduction)*
  -> nil
  (take 10 examples.introduction/fibs)
  -> (0 1 1 2 3 5 8 13 21 34)
- Durch Verwendung von refer ist es möglich, diese Funktionsnamen in den aktuellen Namensraum abzubilden und die Funktionen dann ohne Präfix aufzurufen:

Bibliotheken

```
(refer quoted-namespace-symbol)
user=> (refer 'examples.introduction)
-> nil
(take 10 fibs)
-> (0 1 1 2 3 5 8 13 21 34)
```

- Statt in zwei Schritten eine Bibliothek zunächst zu laden und anschließend die in ihr verwendeten Bezeichner in den aktuellen Namensraum zu importieren, kann man mit use beide Schritte kombinieren:

```
(use quoted-namespace-symbol)
user=> (use 'examples.introduction)
-> nil
(take 10 fibs)
-> (0 1 1 2 3 5 8 13 21 34)
```