Clojure Introduction

An Introduction to the Clojure Programming Language

Rüdiger Gad

http://ruedigergad.com

Outline

Outline

What is it?

- Functional Programming Language
- Hosted on the JVM
- Lisp-dialect
- Dynamically Typed

Why use it?

- Powerful
- Concise Code
- Rapid Development
- Easy Prototyping and Experimenting
- Concurrency Features
- Java Interoperability → Re-use Existing Libs/Code
- JVM → "Platform Independent"
- ...

Don't be afraid. ;)

- Lot's of Brackets
- Entirely different Programming Paradigm
- Might be confusing at first.
- Many Cool Features
- It's worth the effort!

"Installation"

- clojure.org
- Packet Manager
- Download
- Suggested: rlwrap or jline

IDE Integration

- Eclipse: Counterclockwise
  - Update Site: `http://ccw.cgrand.net/updatesite/`
- Vim: `http://dev.clojure.org/display/doc/Getting+Started+with+Vim`
- Emacs: `http://dev.clojure.org/display/doc/Getting+Started+with+Emacs`
- JEdit: `http://dev.clojure.org/display/doc/Getting+Started+with+JEdit`
- ...

Outline

2 Introduction
- Basics
- Data Types

Prepare Yourself

- “You must unlearn what you have learned.”, Yoda
- Try not to think in other languages.
- It is simple.
- Very Few Rules

You will learn about:

- Read Eval Print Loop (REPL)
- Lisp
- Prefix Notation
- Functional Programming
- 1st Class Functions
- Higher-order Functions
- Data Structures
- Homoiconic
- Hands-on Knowledge

Other Cool Stuff (not covered here)

- Macros
- Variable Capture
- Un-/Quoting
- Laziness
- Namespaces
- Software Transactional Memory
- Bindings
- Java Interoperability
- ...

REPL

- Read Eval Print Loop
- Interactive Prompt
- Test and Experiment
- Very Powerful

```
[rc@colin ~]$ clojure
Clojure 1.4.0
user=>
```

Clojure is a Lisp.

- List Processing
- Syntax: (...)
- Special Rule:
  - First Element: "Executable"
  - Rest: "Arguments"
- Example: (println "Hello World")

Basics

- Prefix Notation
  - (+ 1 2)
  - (* 3 4)
- Counterexample
  - Infix Notation
  - 1 + 2
  - 3 * 4
  - Languages: Python, C, ...

Basics (continued)

- nil
  - "NULL", null, ...
  - Simply: nothing
- Comments
  - ";"
  - ; This is a comment.

REPL: Basic Examples

```
user=> ; Comments are started with ";".
user=> (+ 1 2)
3
user=> (+ 1 2 3 4)
10
user=> (println "Hello!") ; Prints to stdout and returns nil.
Hello!
nil
user=>
```

Simple Data Types

- Boolean: true, false
- Numerical: 1, 2.3, ...
  - Arbitrary Size
- Characters: \X
- Strings: "My String"
- Standard Java Classes
- Special
  - Ratio: 1/3

REPL: Basic Data Types

```
user=> true
true
user=> 1
1
user=> 2.3
2.3
user=> 12345678901234567890123456789O
12345678901234567890123456789ON
user=> \X
\X
user=> "My String"
"My String"
user=> 1/3
1/3
```

REPL: Basic Data Types (continued)

```
user=> (type true)
java.lang.Boolean
user=> (type 1)
java.lang.Long
user=> (type 2.3)
java.lang.Double
user=> (type 123456789012345678901234567890)
clojure.lang.BigInt
user=> (type \X)
java.lang.Character
user=> (type "My String")
java.lang.String
user=> (type 1/3)
clojure.lang.Ratio
```

Keywords

- :my-keyword
- Always evaluate to themselves.

```
user=> :my-keyword
:my-keyword
user=> (type :my-keyword)
clojure.lang.Keyword
```

“Complex” Data Types

- Lists: (. . .)
- Vectors: [. . .]
- Sets: #{. . . }
- Maps: {. . . }
- Note!
  - Lists are special.
  - '(. . .)

REPL: "Complexer" Data Types

```
user=> '(1 2 3)
(1 2 3)
user=> (type '(1 2 3))
clojure.lang.PersistentList
user=> ["a" "b" "c"]
["a" "b" "c"]
user=> (type ["a" "b" "c"])
clojure.lang.PersistentVector
user=> #{1 2 3}
#{1 2 3}
user=> (type #{1 2 3})
clojure.lang.PersistentHashSet
user=> {"a" 1 "b" 2}
{"a" 1, "b" 2}
user=> (type {"a" 1 "b" 2})
clojure.lang.PersistentArrayMap
```

Get Help

- REPL
  - (doc println)
  - (find-doc "foo")
- Cheat Sheet: `http://clojure.org/cheatsheet`
- API Docs: `http://clojure.github.com/clojure/api-index.html`
- Books: Clojure in Action, The Joy of Clojure
- `https://github.com/ruedigergad/clojure-by-example`

REPL: Getting Help

```
user=> (doc println)
-------------------------
clojure.core/println
([& more])
  Same as print followed by (newline)
nil
user=> (doc print)
-------------------------
clojure.core/print
([& more])
  Prints the object(s) to the output stream that is the current va
  of *out*.  print and println produce output for human consumptio
nil
user=>
```

REPL: Getting Help (continued)

```
user=> (find-doc "produce output")
-------------------------
clojure.core/print
([& more])
  Prints the object(s) to the output stream that is the current va
  of *out*.  print and println produce output for human consumptio
nil
```

Wrap-up

- Data Types
  - Simple
  - Complex
- Get Help
- Prefix Notation
- Very Few Rules
- One Important Rule:
  - Lists are special.

Outline

We will look into . . .

- Some Built-in Functions
- Storing Things
- Define own functions.
- 1st Class Functions
- Higher-order Functions
- Homoiconic
- Lexical Scope

Built-in Functions

- Mathematical: +, -, *, /, ...
- Modify Data Structures
- Data Structures and Keywords as Functions
- Experiment on your own.
- Use "(type ...)" for your experiments.

REPL: Some Mathematical Functions

```
user=> (+ 1 2)
3
user=> (+ 1 2 3 4 5)
15
user=> (/ 10 2)
5
user=> (/ 1 4)
1/4
user=> (* (/ 1 4) 2)
1/2
```

REPL: Functions on Data Structures

```
user=> (first [1 2 3])
1
user=> (last [1 2 3])
3
user=> (nth [1 2 3] 1)
2
user=> (nth '(1 2 3) 1)
2
user=> (conj [1 2] 3)
[1 2 3]
user=> (conj '(1 2) 3)
(3 1 2)
user=> (conj #{"a" "b"} "c")
#{"a" "b" "c"}
user=> (get {"a" 1 "b" 2} "a")
1
```

REPL: Data Structures and Keywords as Functions

```
user=> ([1 2 3] 1)
2
user=> (:a {:a "a" :b "b"})
"a"
user=> ({:a "a" :b "b"} :a)
"a"
user=> ({"a" 1 "b" 2} "a")
1
user=> (#{"a" "b" "c"} "x")
nil
user=> (#{"a" "b" "c"} "b")
"b"
```

"Store" Things

- Symbols
  - Used to reference things.
  - Begin with non-numerical characters.
- Vars
  - Name: Symbol
  - "Binding" to "Something"
  - Evaluate to the bound "something".
- Example: (def my-var 1234)

REPL: Vars

```
user=> my-var
CompilerException java.lang.RuntimeException: Unable to resolve sy
user=> (def my-var 1234)
#'user/my-var
user=> my-var
1234
user=> (type my-var)
java.lang.Long
user=> (def another-var "foo")
#'user/another-var
user=> another-var
"foo"
```

REPL: Vars (continued)

```
user=> (def my-var "string")
#'user/my-var
user=> my-var
"string"
user=> (def my-var another-var)
#'user/my-var
user=> my-var
"foo"
```

Functions

- Created with the `fn` Special Form.
- Syntax: (fn [args] body)
- Example: (fn [] (println “Hello Function”))
- Note: `args` is a Vector
- Remember: Clojure is homoiconic.

```
user=> (fn [] (println "Hello Function"))
#<user$eval160$fn__161 user$eval160$fn__161@6c0089ab>
```

Functions (continued)

- What to do with that?
- How to give it a name?
- How to “call” functions?
- Solution → “Store” in a Var
- (def my-function (fn [] (println “Hello Function”)))
- “Call”: (my-function)
- Recall: Lists are special.

REPL: Basic Function

```
user=> (def my-function (fn [] (println "Hello Function")))
#'user/my-function
user=> (my-function)
Hello Function
nil
```

Functions Shortcut

- (def foo (fn [] ...)) all the time?
- Something "more Clever"?
- Shortcut → (defn foo [] ...)
- Keep in mind: A function is "stored" in a var.
- Sidenote: defn is a macro.
- Clojure has lots of clever "shortcuts".
- Custom "Shortcuts" Possible

REPL: Basic Function

```
user=> (defn my-function [] (println "Hello Function"))
#'user/my-function
user=> (my-function)
Hello Function
nil
```

Functions, 1st-class

- Functions: 1st-class Members
- Q: 1st-class?
- A: Like any other Member
  $\rightarrow$
  - Storable
  - Returnable
  - Pass as Argument
- Powerful Feature

Anonymous Functions

- (fn [] body)
- Anonymous $\rightarrow$ no Name
- “Call” this?
- Recall: Lists are special.
- $\rightarrow$ ((fn [] body))
- Shortcut: #(body)
- Strange?
- Powerful!

REPL: Anonymous Function

```
user=> (fn [] (println "My fn"))
#<user$eval9$fn__10 user$eval9$fn__10@1776323e>
user=> ((fn [] (println "My fn")))
My fn
nil
user=> #(println "My 2nd fn")
#<user$eval17$fn__18 user$eval17$fn__18@2430c6ca>
user=> (#(println "My 2nd fn"))
My 2nd fn
nil
user=>
```

Passing Arguments

- (fn [**arguments**] body)
- Arguments: Vector []
- Example: (defn hello [name] (println "Hello" name "!"))
- Usage: (hello "Bob")

REPL: Passing Arguments

```
user=> (defn hello [name] (println "Hello" name "!"))
#'user/hello
user=> (hello "Bob")
Hello Bob !
nil
user=>
```

Multi-arity Functions

- Arity → # of Arguments
- Example
  - (+ 1)
  - (+ 1 2)
  - (+ 1 2 3)
- Syntax:

```
(defn foo
      ([] zero args)
      ([x] one arg)
      ([x y] two args)
      ...)
```

REPL: Multi-arity Functions

```
user=> (defn hello
         ([name] (println "Hi" name))
         ([name surname] (println "Hello Mr." name surname))
         ([n sn prof]
           (hello n sn)
           (println "How is the" prof "business?")))
#'user/hello
user=> (hello "Bob")
Hi Bob
nil
user=> (hello "Robert" "Terwilliger")
Hello Mr. Robert Terwilliger
nil
user=> (hello "Robert" "Terwilliger" "TV")
Hello Mr. Robert Terwilliger
How is the TV business?
nil
```

Variable Length Arguments

- (+ 1 2 3 ... 99 100) ← Possible?
- Special “Trick”
- (defn foo [a & b] ...)
  - a → “Normal Argument”
  - & b → List with Remaining Arguments
- Experiment:
  (defn my-fn [a & b] (println a b))

REPL: Variable Length Arguments Experiment

```
user=> (defn my-fn [a & b] (println a b))
#'user/my-fn
user=> (my-fn :x)
:x nil
nil
user=> (my-fn :x :y)
:x (:y)
nil
user=> (my-fn :x :y :z)
:x (:y :z)
nil
user=> (my-fn :x :y :z :foo :bar :baz)
:x (:y :z :foo :bar :baz)
nil
user=> ; More later!
```

Arguments & Anonymous Functions

- (fn [a b] ...)
  - "Same procedure as every year."
- #()?
  - Special Convention
  - % and %1 %2 ...
  - #(println %)
  - #(println %1 %2)

REPL: Arguments & Anonymous Functions

```
user=> ((fn [a b] (println a b)) "foo" "bar")
foo bar
nil
user=> (#(println %) "foo")
foo
nil
user=> (#(println %1 %2) "foo" "bar")
foo bar
nil
user=>
```

Higher-order Functions

- Functions that return functions.
- Functions that accept functions as arguments.
- Remember: Functions are first-class!
- Example
  - Multiply each element in a vector.
  - E. g.: [1 2 3 4 5 6 7 8 9 10]
  - `map` Function
  - Plus (in our case) Anonymous Function for Multiplication
  - (map #(* 2 %) [1 2 3 ...])

REPL: Higher-order Functions

```
user=> (map #(* 2 %) [1 2 3 4 5 6 7 8 9 10])
(2 4 6 8 10 12 14 16 18 20)
user=> (map #(+ 1 %) [1 2 3 4 5 6 7 8 9 10])
(2 3 4 5 6 7 8 9 10 11)
user=> (defn times-two [x] (* 2 x))
#'user/times-two
user=> (map times-two [1 2 3 4 5 6 7 8 9 10])
(2 4 6 8 10 12 14 16 18 20)
user=> (map #(* %1 %2) [1 2 3 4 5] [10 20 30 40 50])
(10 40 90 160 250)
user=>
```

Higher-order Functions (continued)

- Sum up all elements of a vector.
- “Good Old Gauß”
- `reduce` Function
- First Argument: Function with Two Arguments
- Second Argument: Sequence
- Pass result of fn and element from sequence to next fn call.
-

REPL: Higher-order Functions (continued)

```
user=> (reduce #(+ %1 %2) [1 2 3 4 5 6 7 8 9 10])
55
user=> (range 1 11)
(1 2 3 4 5 6 7 8 9 10)
user=> (reduce #(+ %1 %2) (range 1 101))
5050
user=>
```

Variable Length Arguments

- (+ 1 2 3 . . . 99 100) ← Possible?
- Special “Trick”
- (defn foo [a & b] . . .)
- Recall: “& b” → List with remaining Arguments
- Problem: How to process the list?
- Solution: Higher-order Functions!

REPL: Variable Length Arguments Experiment

```
user=> (defn add
         ([] 0)
         ([x] x)
         ([x y] (+ x y))
         ([x y & more] (reduce add (add x y) more)))
#'user/add
user=> (add)
0
user=> (add 1)
1
user=> (add 1 2)
3
user=> (add 1 2 3)
6
user=> (add 1 2 3 4 5 6)
21
user=>  ; Same solution as in Clojures built-in + etc.
```

Locals

- (fn [x] ...)
- x is a local.
- Immutable
- Once defined it cannot be changed!
- Other way to create locals
  - `let`
  - (let [y 1] (println y))

REPL: Locals

```
user=> (let [x 1] (println x))
1
nil
user=>
```

Closures (Lexical Scope)

- "Close over locals"
- (def my-fn (let [x 1] (fn [] (println x)))
- More practical example
  - Higher-order Function
  - Return a dynamically created function.
  - E. g.: Create an "add-x" function.

REPL: Closures

```
user=> (def my-fn (let [x 1] (fn [] (println x))))
#'user/my-fn
user=> (my-fn)
1
nil

user=> (defn create-add [x] #(+ x %))
#'user/create-add
user=> (def plus-two (create-add 2))
#'user/plus-two
user=> (plus-two 1)
3
user=> (def plus-ten (create-add 10))
#'user/plus-ten
user=> (plus-ten 5)
15
user=>
```

Outline

4 Hands-on
- Automating Clojure Projects
- Test-driven Development
- Running the Application

Leiningen

Overview

- "Leiningen is for automating Clojure projects without setting your hair on fire."
- `https://github.com/technomancy/leiningen`

Installation

- Get the "lein" shell script.
- Put the script on your $PATH, e. g., in /usr/local/bin
- Run: lein self-install.

What can Leiningen do for you?

- Automation
- Building
- Running
- Running Tests
- Resolve and download dependencies.
- ...

Example Project

- Command Line Calculator
- "foo-calc"
- Example
  - java -jar foo-calc.jar $2 + 3 \rightarrow 5$

Example Project (continued)

Approach

- Test-driven Development
- Iterative Approach
- Continuous Refactoring

Prerequisites

- REPL for Experiments, Prototyping
- IDE/Editor for Coding
- Leiningen for Running Tests, Building, etc.

Lets get started...

- Create a directory for your projects.
- cd into that directory.
- Run: lein new foo-calc
- cd into the newly created project skeleton.
- Explore the project skeleton.

Leiningen: New Project

```
[rc@colin examples]$ lein new foo-calc
Created new project in: /home/rc/repositories/research/rcgWiP/arch
Look over project.clj and start coding in foo_calc/core.clj
[rc@colin examples]$ cd foo-calc/
[rc@colin foo-calc]$ find
.
./README
./.gitignore
./project.clj
./test
./test/foo_calc
./test/foo_calc/test
./test/foo_calc/test/core.clj
./src
./src/foo_calc
./src/foo_calc/core.clj
[rc@colin foo-calc]$
```

Leiningen: New Project (continued)

```
[rc@colin foo-calc]$ lein test
Copying 1 file to /home/rc/repositories/research/rcgWiP/archive/cl

Testing foo-calc.test.core

FAIL in (replace-me) (core.clj:6)
No tests have been written.
expected: false
  actual: false

Ran 1 tests containing 1 assertions.
1 failures, 0 errors.
[rc@colin foo-calc]$
```

Define Tests

- `deftest`
  - Define Test
- `is`
  - Test

```
(deftest failing-test
         (is (= 1 2)))
(deftest succeeding-test
         (is (= 1 1)))
```

Leiningen: Running Tests

```
[rc@colin foo-calc]$ lein test

Testing foo-calc.test.core

FAIL in (failing-test) (core.clj:6)
expected: (= 1 2)
  actual: (not (= 1 2))

Ran 2 tests containing 2 assertions.
1 failures, 0 errors.
[rc@colin foo-calc]$
```

Start Coding

- Start with a simple test.
- $2 + 3 = 5$
- Implement function stub to make code compile.
- Run test to assure the test fails.
- Implement functionality.
- Experiment on the REPL.
- Run test to see if the implementation works correctly.

Leiningen: Test & Stub

```
; Test
(deftest simple-add-test
         (is (= 5 (foo-calculate "2 + 3"))))
; Stub
(defn foo-calculate [input])
; Test Output
[rc@colin foo-calc]$ lein test

Testing foo-calc.test.core

FAIL in (simple-add-test) (core.clj:11)
expected: (= 5 (foo-calculate "2 + 3"))
  actual: (not (= 5 nil))

Ran 1 tests containing 1 assertions.
1 failures, 0 errors.
```

Very First Implementation (Needs a lot of improvement!)

```
(ns foo-calc.core
; We need the split function from the clojure.string namespace.
  (:use [clojure.string :only (split)]))

(defn foo-calculate [input]
  (let [input-vector (split input #" ")
        operand-1 (read-string (input-vector 0))
        operation (input-vector 1)
        operand-2 (read-string (input-vector 2))]
    (if (= operation "+")
      (+ operand-1 operand-2))))
; This implementation works but is pretty ugly and complicated.
```

Possible Improvements

```
(defn foo-calculate [input]
  (let [[op1 operation op2] (map read-string (split input #" "))]
    ((resolve operation) op1 op2)))
; Dangerous as "operation" might do something bad!

; Better to check if we do something that is allowed.
(defn foo-calculate [input]
  (let [[op1 operation op2] (map read-string (split input #" "))
        allowed-operations #{'+}]
    (if (allowed-operations operation)
      ((resolve operation) op1 op2))))
; Looks better but may be still improved.
```

Next Steps

- More Tests
- More Functionality
- Increasing Complexity
- $\rightarrow$ Iterative Test-driven Development
- Try yourself.

Run the application.

We want to ...

- Run the application.
- java -jar foo-calc.jar 3 + 4

We need ...

- "main Method"
- Runnable Jar

Add a main method.

- “main Method”

```
(defn -main [& args]
  (println (foo-calculate (first args))))
```

- Additionally needed:

```
(ns foo-calc.core
  (:use [clojure.string :only (split)])
; In order to be able to call the main method
; we need to add genclass here.
  (:gen-class))
```

Tell Leiningen about the “main method”.

- Leiningen needs to now the namespace the “main method” is in.

```
(defproject foo-calc "1.0.0-SNAPSHOT"
  :description "FIXME: write description"
  :dependencies [[org.clojure/clojure "1.3.0"]]
; Here we tell Leiningen where to find the "main method".
  :main foo-calc.core)
```

Run via Leiningen.

- lein run foo
- Interactive prompt may be problematic.
- $\rightarrow$ Run from command line.

```
[rc@colin foo-calc]$ lein run "1 + 2"
3
[rc@colin foo-calc]$
```

Create a jar.

- “Normal”
- “Stand-alone”
- For Convenience → Stand-alone

```
[rc@colin foo-calc]$ lein uberjar
Copying 1 file to /home/rc/repositories/research/rcgWiP/archive/cl
Compiling foo-calc.core
Compilation succeeded.
Created /home/rc/repositories/research/rcgWiP/archive/clojure/exam
Including foo-calc-1.0.0-SNAPSHOT.jar
Including clojure-1.3.0.jar
Created /home/rc/repositories/research/rcgWiP/archive/clojure/exam
[rc@colin foo-calc]$
```

Run the jar.

- Like any Java Command Line Application
- java -jar ...

```
[rc@colin foo-calc]$ java -jar \
> foo-calc-1.0.0-SNAPSHOT-standalone.jar "3 + 4"
7
[rc@colin foo-calc]$
```

Outline

5 End

Summary

- Clojure Basics
  - Data Types
    - Simple
    - Complex
  - Vars
  - Functions
- 1st Class Functions
- Higher-order Function
- Homoiconic
- Lexical Scope & Closures

Summary (continued)

- REPL
- Lisp
- Test-driven Development
- Leiningen
- Automating Projects
- Run Command Line Applications
- Create Deployable Jars

Take-away Message

- Don't be scared.
- Very Few, Simple Rules
- Brackets are not harmful. ;)
- Clojure → "A Cool Language"
- Quick Development
- Powerful Features
- A Nice Ecosystem
- Worth the Effort!

More to come.

- Macros
- Variable Capture
- Un-/Quoting
- Laziness
- Namespaces
- Software Transactional Memory
- Bindings
- Java Interoperability
- ...

End

Thank you for your attention!

End

Thank you for your attention!

Questions?