



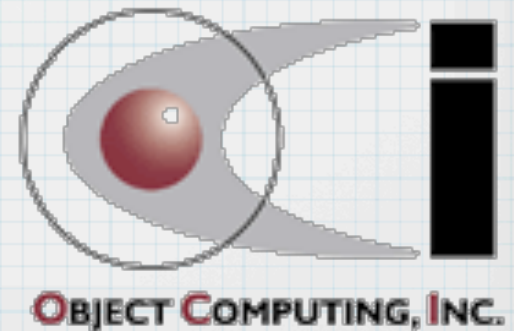
Clojure

pronounced the same as "closure"

"It (the logo) was designed by my brother, Tom Hickey. I don't think we ever really discussed the colors representing anything specific. I always vaguely thought of them as earth and sky." - Rich Hickey

"I wanted to involve c (c#), l (lisp) and j (java). Once I came up with Clojure, given the pun on closure, the available domains and vast emptiness of the googlespace, it was an easy decision.." - Rich Hickey

R. Mark Volkmann
mark@ociweb.com
January 2009



Core Beliefs ...

- * **Functional programming is important**

- * **helps with concurrency issues**

- * emergence of multi-core processors makes this an issue for all kinds of applications
 - * locking is too hard to use correctly
 - * immutable data along with other mechanisms is easier

- * **provides performance optimization**

- * through code reordering by compiler

- * **can make code easier to understand and test**

- * the result of functions with no side-effects only depends on arguments

Side-effects include:
changing values of global variables
and performing any kind of I/O

- * **Java platform (JVM) is the place to be**

- * **portability, stability, performance, security**

- * **access to existing Java libraries**

- * no need to reinvent libraries for file I/O, database access, XML, and so on

... Core Beliefs

- * **Dynamic types and polymorphism are good**

- * see multimethods

- * **Lisp-like syntax is good**

- * code and data have same representation
 - * transformation from Java syntax to Lisp syntax

- * move { in method definitions to beginning
 - * replace { with (and } with)
 - * remove types (can specify type hints)
 - * remove commas from argument lists
 - * remove semicolon statement terminators

- * **less “noise” than Java (4 -> 2 below)**

`myFunction(arg1, arg2); -> (myFunction arg1 arg2)`

```
// Java
public void hello(String name) {
    System.out.println("Hello, " + name);
}

; Clojure
(defn hello [name]
  (println "Hello," name))
```

Side Effects

- * In general, functions should not

- * rely on global data values (only their arguments)
- * modify global data values
- * perform I/O

"using purely functional programming is also not very useful since, if we allow no side effects, our program will do nothing except heat up the CPU" - Simon Peyton Jones

- * Especially for functions invoked in a transaction

- * because they may be invoked more than once if the transaction must be rerun

- * Some benefits

- * makes functions easier to understand and test
- * allows their execution to be reordered and parallelized

- * It's up to you to avoid side effects

- * Clojure doesn't prevent them

Clojure does provide the function `io!` which takes a set of expressions to execute. If it is executed inside a transaction, an `IllegalStateException` is thrown.

Clojure Key Features ...

- * **Functional, inspired by other languages**
 - * Lisp (syntax), Haskell (lazy evaluation), ML, Erlang
- * **Concise**
 - * results in shorter programs which are easier to write and maintain
- * **Lisp syntax**
 - * with enhancements
 - * code is data; can modify the language using the language
- * **Runs on JVM**
 - * popular, efficient platform that is constantly being improved
 - * large number of available libraries
- * **Java interop**
 - * can call Java methods from Clojure code and Clojure code from Java

... Clojure Key Features

- * **Sequences - logical lists**

- * examples include all Clojure and Java collections, streams, directory structures and more

- * **Concurrency without locks**

- * important for multi-threaded, multi-processor applications
- * see refs, agents and atoms

Getting Clojure

* Prebuilt

- * download from <http://clojure.org>

* From source

- * `svn co http://clojure.googlecode.com/svn/trunk/ clojure-read-only`
- * `cd clojure-read-only`
- * `ant clean jar`

* clojure-contrib from source

- * the “standard library”
- * not well-documented yet; see examples in source
- * `svn co http://clojure-contrib.googlecode.com/svn/trunk/ \`
`clojure-contrib-read-only`
- * `cd clojure-contrib-read-only`
- * `ant clean jar`

REPL

- * **Read Eval Print Loop**

- * an interactive shell for experimenting with Clojure code
- * like Ruby's irb
- * the "reader" reads program text and produces data structures (mostly lists)
- * these are evaluated to obtain results that are printed

- * **To start, run clj script** - see next slide

- * **To load code from a file,** `(load-file "file-path")`

- * **Special variables**

- * result of last three evaluations are saved in `*1`, `*2` and `*3`
- * last exception is saved in `*e`
 - * to see stack trace `(.printStackTrace *e)`

- * **To exit, press ctrl-d or ctrl-c**

Running Clojure Code

- * Create a script like this named `clj` - why isn't this supplied?
- * see http://en.wikibooks.org/wiki/Clojure_Programming/Getting_Started

```
#!/bin/bash
# Runs Clojure on a script file or interactively using a REPL.
CLOJURE_JAR=$CLOJURE_DIR/clojure-read-only/clojure.jar
CONTRIB_JAR=$CLOJURE_DIR/clojure-contrib-read-only/clojure-contrib.jar
BREAK_CHARS="(){}[],^%$#@\"'\";: ' | \\"
CP=$CLOJURE_JAR:$CONTRIB_JAR:$JLINE_JAR

# If there are no command-line arguments ...
if [ -z "$1" ]; then
    rlwrap --remember -c -b $BREAK_CHARS -f $HOME/.clj_completions \
    java -cp $CP clojure.main --init ~/user.clj --repl
else
    java -cp $CP clojure.lang.Script $1 -- $*
fi
```

rlwrap supports tab completion, paren matching, command recall across sessions, and vi or emacs keystrokes. See <http://utopia.knoware.nl/~hlub/uck/rlwrap/>. Another option is JLine.

For more options, run
`java -jar ./clojure/trunk/clojure.jar -help`

Hello World!

- * `hello.clj`

```
(println "Hello World!")
```

- * To run

```
$ clj hello.clj
```

- * To get documentation on any function, even ones you wrote

```
$ clj
```

```
user=> (doc function-name)
```

Alternative

Add `#!/usr/bin/env clj` as first line of `.clj` files.

- * To get documentation on all functions whose name or documentation match a given regex

```
$ clj
```

```
user=> (find-doc "regex-string")
```

The `doc` function generates a description of the allowed arguments from the code and outputs the function doc-string if one was provided.

For example, to find all the predicate functions, `(find-doc "\\?$")`

Invoking From Java - Option 1

- * From a Java application,
read a text file containing Clojure code
and invoke specific functions it defines

```
import clojure.lang.RT;  
import clojure.lang.Var;  
...  
// path must be in CLASSPATH  
RT.loadResourceScript("path/name.clj");  
Var function = RT.var("namespace", "function-name");  
function.invoke("arg1", "arg2", ...);
```

Invoking From Java - Option 2

- * Compile Clojure code to bytecode and use it from a Java application just like any other Java code

- * easy if your Clojure code implements an existing Java interface

```
(ns namespace
  (:gen-class :implements [java-interface]))

(defn -function-defined-in-interface [this arg1 arg2 ...]
  ...)
```

- * **Note**

- * **defn** names for functions defined in the interface begin with "-"
- * every method takes an extra, first "this" argument
- * to generate the .class file, use
(compile namespace) or **clojure.lang.Compile**

Books

- * Only one now ... “Programming Clojure”

- * Stuart Halloway, Pragmatic Programmers

- * Website

- * <http://pragprog.com/titles/shcloj/programming-clojure>

- * has example code, errata and a forum

- * Running example code

```
$ clj
```

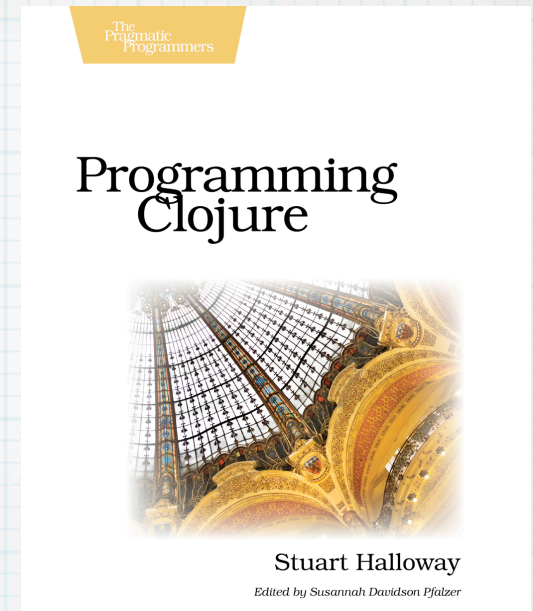
```
user=> (require 'examples.introduction)
```

```
user=> (take 10 examples.introduction/fibs)
```

- * **take** returns a lazy sequence of the first *n* items in a collection

- * namespaces are separated from names by a /

- * **examples.introduction** is the namespace of the **fibs** function



Processing Order

* Read-time

- * reader macros and “normal” macros are converted to non-macro forms
- * macros only evaluate their arguments if and when directed to do so

* Compile-time

- * forms, including function calls, are compiled to Java bytecode
- * not interpreted

* Run-time

- * Java bytecode is executed
- * functions evaluate all their arguments before running

Forms

- * The Clojure “reader” looks for forms in program text and creates data structures from them
- * Supported forms include

- * **no value** - `nil`; treated as false in boolean contexts; same as Java’s null
- * **boolean** - `true` or `false`
- * **character** - `\char`, `\newline`, `\space`, `\tab` - uses `java.lang.Character`
- * **number** - integer, decimal or ratio; automatically uses `BigInteger` when needed
- * **keyword** - name that begins with `:`; like Java interned Strings

- * **symbol** - names things like variables and functions

- * composed of letters, digits (not first), `+` `-` `/` `|` `?` `.` `_`

- * **string** - `"char*"` - uses `java.lang.String`
- * **list** - `'( item* )` - allows duplicates; not indexable
- * **vector** - `[ item* ]` - similar to a list, but indexable like an array (w/ `get`)
- * **set** - `#{ item* }` - like list, but no duplicates
- * **map** - `{ item-pair* }` - each item-pair is a key and value separated by a space

Clojure collections

All forms except symbols and lists are literals, i.e. they evaluate to themselves.

`6/9` is a ratio that will be represented by `2/3`.
Math with ratios maintains precision.

`'foo` evaluates to the symbol
`foo` evaluates to the value

lists are evaluated literally only if they are quoted

empty lists `()` evaluate to themselves

keywords and symbols have a name and an optional namespace

Clojure Collections

* Include

See diagram showing relationships at
<http://tinyurl.com/clojure-classes>

- * **lists** - `'(items)` or `(list items)`
 - * a singly linked list
 - * without the quote it is evaluated as a function call
- * **vectors** - `[items]` or `(vector items)`
 - * a dynamic array; can be treated as a map with integer index keys
 - * often used in place of lists to avoid need to quote to avoid being evaluated as a function call
- * **sets** - `#{items}` or `(hash-set items)`
 - * items must be unique
- * **maps** - `#{pairs}` or `(hash-map pairs)`
 - * associative array of key/value pairs
- * **sorted-set** - `(sorted-set items)`
- * **sorted-map** - `(sorted-map pairs)`
and `(sorted-map-by comparator pairs)`
- * and more that are used internally

... Clojure Collections ...

- * All are immutable
- * All are heterogenous
 - * can hold a variety of types
- * All are “persistent”
 - * “support efficient creation of modified versions by utilizing structural sharing”
 - * works because they are immutable
 - * not related to persistent storage
 - * see http://en.wikipedia.org/wiki/Persistent_data_structure

Vectors

* To create

- * `[ :a 2 "three" ]`
- * `(vector :a 2 "three")`
- * `(vec another-collection)`

* To access elements

- * indexes are zero-based
- * `(def my-vector [2 5 7])`
- * `(get my-vector 1) -> 5` - returns nil if index is out of bounds
- * `(nth my-vector 1) -> 5` - can throw `IndexOutOfBoundsException`
- * `(my-vector 1) -> 5` - vectors are a function of their indexes
- * integers are not functions of vectors - can't use `(1 my-vector)`

Sets

- * To create

- * `#{} - an empty set`
- * `#{:a 2 "three"}`
- * `(set :a 2 "three")`

- * To put an empty set into a variable,

`(def mySet (ref #{}))`

- * To add a value to the set,

`(dosync (commute mySet conj value))`

- * `dosync` evaluates its argument in an STM
- * yikes that's verbose!

- * To dereference the set from the variable

`(deref mySet)`

Maps ...

* To create

- * `{ key1 value1 key2 value2 ... }`
- * often keywords are used for keys because comparing them is fast
 - * can get the name of a keyword as a string - `(name :foo) -> "foo"`
- * `(def my-map {:a 1 :b 2})`

* To get the value of a key, returning nil if not found

- * `(get map key)` or `(get map key not-found-value)`
- * maps are functions of keys and keys are functions of maps
- * `(map key)` or `(key map)`
- * `(my-map :b)` or `(:b my-map)`

* To get an entry, returning nil if not found

- * `(find map key)`
- * `(find my-map :b) -> <:b 2>` (printed form of an entry)

use **key** and **val** functions
to get pieces of an entry

... Maps

- * To determine if a key is present
 - * `(contains? map key)`
- * To get a new map with entries added
 - * `(assoc map key value key value ...)`
 - * `(assoc my-map :c 3 :d 4)`
- * To get a new map with entries removed
 - * `(dissoc map keys)`
 - * `(dissoc my-map :a :c)`
- * To get all the keys or all the values as a sequence
 - * `(keys map)`
 - * `(values map)`

Sequences ...

- * **Logical list of things; view on a collection**
 - * not a data structure
 - * not a copy of the collection
- * **Immutable**
- * **Supported by classes that implement the `clojure.lang.ISeq` interface**
 - * extends `clojure.lang.IPersistentCollection`
- * **Many types can be treated as sequences**
 - * Clojure and Java collections, strings, regex matches, streams, XML, directory structures, SQL results
 - * most functions that operate on “seq-able” things begin by calling `seq` on their argument
 - * when treating a map as a sequence, each key/value pair is a vector containing the key and value

... Sequences

* Operations supported for all sequences

- * get first - `(first seq)` instead of Lisp `car`
- * get rest - `(rest seq)` instead of Lisp `cdr`
 - * returns a new sequences with the first item removed or `nil` (not an empty sequence; logically false) if the sequence only contains one item
 - * nice because `nil` is logically false whereas an empty list is not
- * “construct” new sequence with one item added to front
 - * `(cons item seq)` same as Lisp `cons`
 - * can usually use `conj` instead
- * “conjoin” items to a sequence to create a new sequence
 - * where the items are added depends on the collection type
 - * `(conj seq items)`
- * get size - `(count seq)`
- * create a new, empty collection of the same type - `(empty seq)`
- * many sequence function eliminate the need to write loops

The Lisp `car` function stands for “contents of the address register”.
The Lisp `cdr` function stands for “contents of the decrement register”.

Lists `conjoin` at front.
Vectors `conjoin` at end.
Maps `conjoin` key/value entries or whole other maps.

Lazy Sequences

- * Most sequences are lazy

- * items are only evaluated when requested
- * allows processing of sequences that are larger than the available memory
- * can force evaluation of all items with **doall** function

- * Examples of creating a lazy sequence

```
(defn f [x] (/ (* x x) 2.0))  
(take 5 (map f (iterate inc 0)))
```

Result

(0.0 0.5 2.0 4.5 8.0)

can also use **lazy-cons**,
lazy-cat and proxies to
implement lazy sequences

```
(defn next-value [x]  
  (println "next-value: x =" x) ; so we know if invoked  
  (+ x x 1))  
(let [start-value 2  
      my-sequence (iterate next-value start-value)]  
  (doseq [x (take 3 my-sequence)] (println x)))
```

Output

```
2  
next-value: x = 2  
5  
next-value: x = 5  
11
```


StructMaps ...

- * Immutable maps used in place of Java Beans
- * Optimized
 - * each instance shares a common set of keys, so doesn't need to repeat them
 - * can add entries with new keys not defined for the struct
- * To define
 - * use keywords for keys (start with a colon)
 - * long way - `(def name (create-struct key+))`
 - * short way - `(defstruct name key+)`
 - * `defstruct` is a macro which can be changed if needed, for example, to add logging of instance creation
 - * proper `hashCode` and `equals` methods are generated
 - * example
 - * `(defstruct car-struct :make :model :year :color)`

... StructMaps ...

- * To create an instance

- * `(struct name value+)`

- * where the order of the values matches the order of the keys

- * example

- * `(def car (struct car-struct "Toyota", "Prius", 2009, Color/YELLOW))`

```
(import ' (java.awt Color))
```

- * To access fields

- * structs are maps

- * `(println (car :year) (car :model)) ; outputs 2009 Prius`

... StructMaps

// Java way

```
public class Car {
    private String make;
    private String model;
    private int year;

    public Car(String make, String model, int year) {
        this.make = make;
        this.model = model;
        this.year = year;
    }

    public String getMake() { return make; }
    public String getModel() { return model; }
    public String getYear() { return year; }
}

Car c = new Car("BMW", "Z3", 2001);
System.out.println("The year is " + c.getYear());
```

; Clojure way

```
; Define a struct (actually a StructMap) for cars.
(defstruct car :make :model :year)

; Optionally define an accessor function.
(def year (accessor car :year))

(let
  ; Create a struct instance.
  [c (struct car "BMW" "Z3" 2001)]

  (println "The year is" (c :year))

  ; Same using the accessor function.
  (println "The year is" (year c))
)
```

Destructuring

- * Functions can take a collection and extract parts of it in the argument list

- * supported by `defn`, `fn`, `let` and `loop`

- * With lists

underscore means don't care about corresponding item

```
(defn add-2nd-and-3rd [[_ p2 p3]] (+ p2 p3))  
(add-2nd-and-3rd [3 4 5 6]) -> 9
```

- * With maps

```
(defstruct car :make :model :year :color)  
(defn print-color-and-model [{c :color m :model}]  
  (println c m))  
(def my-car (struct car "BMW" "Z3" 2001 "yellow"))  
(print-color-and-model my-car)
```


Defining Functions

- * Example - need a different example from book

```
(defn greet [name]
  (println (str "Hello " name)))
```

- * str converts a list of arguments to strings and concatenates them
- * puts the function "greet" into the default namespace "user"
 - * full name is user/greet

- * Can also have a different body for each arity

```
(defn my-function
  ([] (prn "no args"))
  ([x] (prn "one arg"))
  ([x y] (prn "two args")))
```