



Modelos de Linguagem de Programação

Unidade 10 – Tipos simples de dados e Imutabilidade em Clojure

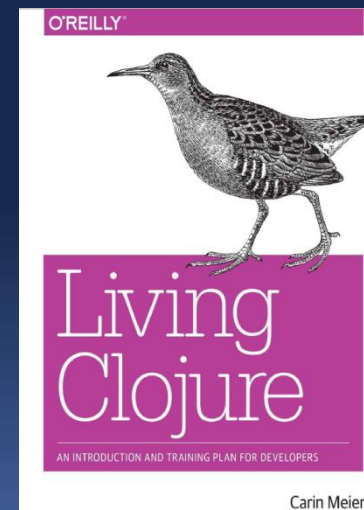
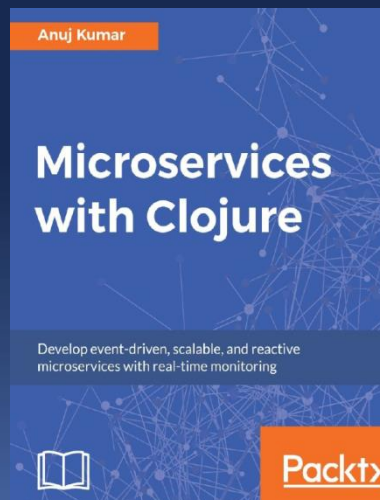
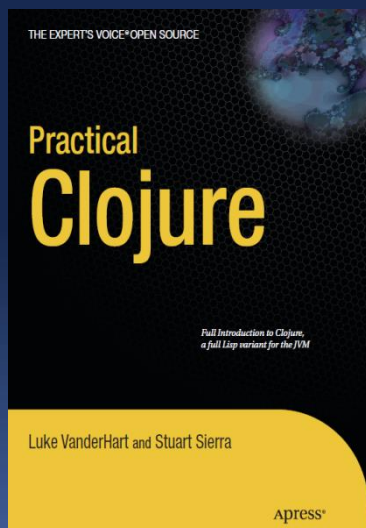
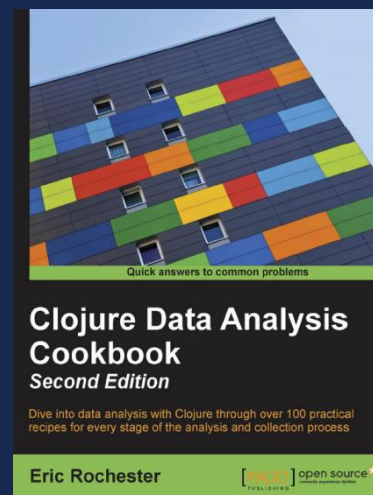
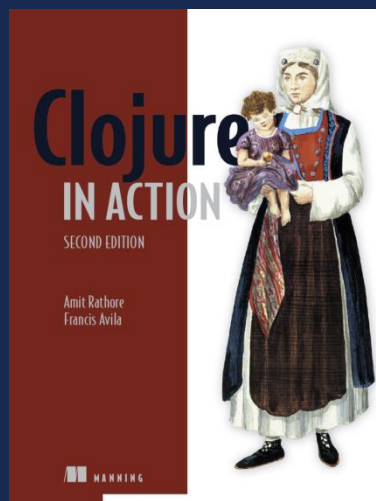


Prof. Aparecido V. de Freitas
Doutor em Engenharia
da Computação pela EPUSP
aparecido.freitas@prof.uscs.edu.br
aparecidovfreitas@gmail.com





Bibliografia





Introdução

- ✓ Na linguagem **Clojure** se trabalha quase que exclusivamente com tipos de dados imutáveis;
- ✓ Esses tipos de dados trazem **legibilidade no código fonte** e evitam conflitos decorrentes de compartilhamento de dados;
- ✓ **Clojure** trabalha dessa forma, com estruturas de dados persistentes. Ou seja, novos dados são produzidos a partir dos dados existentes. Versões prévias dos dados são mantidas;
- ✓ Clojure trabalha com os seguintes tipos básicos de dados: **Strings, Números, Booleans, Keywords** e **nil** (todos eles imutáveis).





Strings

- ✓ **Strings** são sequências de caracteres representando texto;
- ✓ **Strings** sempre são criados com aspas duplas ("");
- ✓ **Strings** não podem ser alterados; eles são imutáveis;
- ✓ Qualquer função que os usa, transforma-os num outro valor.



```
Clojure
Clojure 1.10.2-master-SNAPSHOT
user=>
user=>
user=>
user=>
user=> "Eu sou um String"
"Eu sou um String"
user=>
user=>
user=>
user=> "Eu sou imutável..."
"Eu sou imutável..."
user=> _
```





Strings são imutáveis

- ✓ A função `clojure.string/replace` retornou um novo `String`;
- ✓ Mesmo após a aplicação desta função, o valor original do símbolo texto-string foi mantido.



```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (def texto-string "Eu sou imutável")
#'user/texto-string
user=>
user=>
user=> texto-string
"Eu sou imutável"
user=>
user=> (clojure.string/replace texto-string "Eu sou" "Você é")
"Você é imutável"
user=>
user=>
user=>
user=> texto-string
"Eu sou imutável"
user=>
user=>
user=>
user=>
user=>
user=>
user=>
```





Strings são imutáveis

- ✓ Embora string seja uma unidade de representação de texto, na verdade é composto por um conjunto de caracteres;
- ✓ Na implementação **Java JVM** um string é do tipo **java.lang.String** e é implementado por coleções de caracteres de **java.lang.Character**;
- ✓ **first** retornou o primeiro elemento da coleção, o qual é um caractere;
- ✓ Literais caracteres são representados por ****;
- ✓ Em REPL, ***1** representa o último valor retornado;

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (first "USCS - Universidade Mun. de São Caetano do Sul")
\U
user=>
user=>
user=> (type *1)
java.lang.Character
user=>
user=>
user=>
user=>
```





Strings

- ✓ No **namespace core** há diversas funções para manuseio de strings;
- ✓ Mais funções para tratamento de strings podem ser encontradas no **namespace clojure.string**.

```
Clojure
user=>
user=>
user=>
user=>
user=> (dir clojure.string)
blank?
capitalize
ends-with?
escape
includes?
index-of
join
last-index-of
lower-case
re-quote-replacement
replace
replace-first
reverse
split
split-lines
starts-with?
trim
trim-newline
triml
trimr
upper-case
nil
user=>
```





Apenas lembrando....

- ✓ O exemplo abaixo mostra como usamos uma função de um **namespace** específico:

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=> (clojure.string/includes? "Universidade" "versi")
true
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
```





Números Racionais

- ✓ Em **Clojure**, números expressos por razões inexatas, são do tipo "Ratio"

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=> 5/4
5/4
user=>
user=>
user=> (/ 4 5)
4/5
user=>
user=>
user=>
user=>
user=> (/ 4 4 )
1
user=>
user=>
user=>
user=> 4/4
1
user=>
user=>
user=>
```





Números Decimais

- ✓ Em **Clojure**, números decimais são números de precisão dupla ("double").

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> 1.2
1.2
user=>
user=>
user=>
user=> (/ 3 4.0)
0.75
user=>
user=>
user=> (* 1.0 2)
2.0
user=>
user=>
user=>
user=>
user=>
```





Função type

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=> (type 1)
java.lang.Long
user=>
user=>
user=>
user=> (type 1.2)
java.lang.Double
user=>
user=>
user=> (type 3/4)
clojure.lang.Ratio
user=>
user=>
user=>
```





Função type

```
Clojure 1.10.2-master-SNAPSHOT
user=>
user=>
user=>
user=> (type "Hello World...")
java.lang.String
user=>
user=> (type 1.2)
java.lang.Double
user=>
user=> (type nil)
nil
user=>
user=> (type true)
java.lang.Boolean
user=>
user=> (type 4/4)
java.lang.Long
user=>
user=> (type 3/4)
clojure.lang.Ratio
user=>
user=> _
```





Lendo um valor constante da library Math

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (Math/PI)
3.141592653589793
user=>
user=>
user=>
user=>
user=>
user=>
```





Chamando uma função da library Math

```
Clojure
user=>
user=> (Math/random)
0.4212198501940272
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (Math/sqrt 25)
5.0
user=>
user=>
user=>
user=> (Math/round 5.4)
5
user=>
user=>
user=>
user=>
```





Valores Booleanos

- ✓ Em **Clojure**, valores boolean são implementados com o tipo `java.lang.Boolean`;
- ✓ Esse tipo de dados pode assumir os valores `true` ou `false` e suas representações literais são `true` e `false` (caracteres minúsculos).

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> (type true)
java.lang.Boolean
user=>
user=>
user=>
user=> (type false)
java.lang.Boolean
user=>
user=>
user=>
```






Símbolos



- ✓ Em **Clojure**, símbolos são identificadores que se referem à alguma coisa. Já os utilizamos na criação de bindings (ligações) e funções.
- ✓ Quando usamos **def**, o primeiro argumento é um símbolo o qual estará associado a algum valor.



```
Clojure
user=>
user=>
user=>
user=> (def foo "bar")
#'user/foo
user=>
user=>
user=>
user=> foo
"bar"
user=>
user=>
user=> (defn add-2 [x] (+ x 2) )
#'user/add-2
user=>
user=>
user=> add-2
#object[user$add_2 0x640f11a1 "user$add_2@640f11a1"]
user=>
user=>
user=>
```





Keywords



- ✓ Em Clojure, **keywords** correspondem à constantes especiais string;
- ✓ São definidas simplesmente anexando-se **:** no início da palavra que será tratada como keyword;
- ✓ São tipicamente usadas como chaves em um mapeamento associativo do tipo chave-valor.

```
Clojure
user=>
user=>
user=>
user=>
user=> :a
:a
user=>
user=>
user=>
user=> :b
:b
user=>
user=>
user=> :k
:k
user=>
user=>
```





Coleções



- ✓ Na programação funcional, e mais especificamente em Clojure, trabalha-se com poucos tipos de dados;
- ✓ Collections são tipos de dados que podem conter outros tipos de dados e descrevem a forma pela qual esses itens de dados se relacionam mutuamente;
- ✓ As quatro principais estruturas de dados para Collections são: Maps, Sets, Vectors e Lists

"Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. Data structures, not algorithms, are central to programming." – Rob Pike's Rule #5 of programming.

“Os dados dominam. Se você definiu a estrutura de dados correta e as organizou convenientemente, os algoritmos serão quase sempre evidentes e naturais. Regra #5 de Programação de Rob Pike”.





Maps

- ✓ Um Map é uma coleção de pares key-value;
- ✓ Clojure provê - de forma persistente e imutável - o usual HashMap mas também um SortedMap;
- ✓ HashMaps são chamados "Hash" porque eles criam um hash da chave e a mapeiam para um dado valor;
- ✓ Consultas, bem como outras operações comuns (insert e delete) são eficientes;
- ✓ Pode-se criar um HashMap com a notação literal { }.





Maps

- ✓ Segue um exemplo de um Map com três pares key-value:

```
Clojure
user=>
user=> { :artist "David Bow" :song "The Man" :year 1970 }
{:artist "David Bow", :song "The Man", :year 1970}
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
```

- ✓ A definição de um Map é feita com espaços entre um key-value e outro;
- ✓ Porém Clojure avalia o Map retornando os pares separados por vírgula.





Maps

- ✓ Porém, pode-se definir um Map separando-se também os pares entre vírgulas.

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=> {:a 1 , :b 2 , :c 3 , :d 4 }
{:a 1, :b 2, :c 3, :d 4}
user=>
user=>
user=>
user=>
```





Maps

- ✓ Os pares key-value podem ter tipos diferentes;

```
Clojure
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=>
user=> { :a 1 , :b "Hello" , :c true }
{:a 1, :b "Hello", :c true}
user=>
user=>
user=>
user=>
user=>
```





Maps ***** (Página 53)

- ✓ Maps podem ser definidos de forma aninhada;

