Clojure Basic Training - Module 4

Vars, bindings and namespaces

Clojure is dynamic

- You can connect and alter running programs
- But Clojure is not interpreted!
- Clojure is compiled (to JVM bytecode)
- “lein repl :connect host:port” lets you connect to any running nREPL out there!

What are Vars?

```
user=> (def x)
#'user/x
user=> (type #'user/x)
clojure.lang.Var
user=> (type user/x)
clojure.lang.Var$Unbound
```

- An indirection mechanism to refer to a value
- A Var is “bound” to a value or “unbound”
- A Var can be “re-bound” to another value
- Can be created with (def)

Root binding

```
user=> (def x 1)
user=> (type (var x))
clojure.lang.Var
user=> (type x)
java.lang.Long
user=> x
1
```

- A var can be given a root value at creation time
- The root value can be changed anytime
- You rarely use unbound vars :)

Thread-bound vars

```
user=> (def ^:dynamic *earmuff*)
#'user/*earmuff*
user=> *earmuff*
#<Unbound Unbound: #'user/y>
user=> (binding [y 1] y)
1
```

- Each thread can be bound a different value
- The root value is only visible outside the “binding”
- Mainly used to propagate overridden configs

Var-Value resolution

- Cascading rules:
  - Per-thread binding if any, otherwise
  - Root binding if any, otherwise
  - Unbound

Namespaces

- Once a Var is created it goes into “storage”
- Everything in Clojure must be defined in storage
- This storage mechanism is called a **namespace**
- The REPL puts you automatically in “user>”
- *.clj filenames implicitly declare a namespace

Using namespaces

The REPL shows you the current namespace

```
user=> (create-ns 'foo)
#<Namespace foo>
user=> (intern 'foo 'bar)
#'foo/bar
user=> (in-ns 'foo)
#<Namespace foo>
foo=>
```

Brand new namespace created

Bar var was created in namespace foo

Moved to namespace “foo”

- Namespaces can be created and searched
- Var are always declared in a namespace
- The namespace can be implicit or not
- We can set a different default namespace

Namespaces and files

Folder nesting gives ns name

```
$> foo/bar.clj
(ns foo.bar)
(def baz)
#'foo.bar/baz
```

Form declaration ends up in corresponding ns

- Clojure programs can be split across different files
- Forms in a file are attached to a default ns
- Default ns depends on file name and nested folders
- But (ns) declaration at the top is mandatory!

Including other namespaces

```
(ns foo.bar (:require [clojure.string :as s]))
(s/blank? "")
true
```

- If you want to use “clojure.string/blank?”
- No need to constantly repeat “clojure.string/“
- You can import a namespace (qualified or not)

Other useful :require

```
1. (ns my-ns (:require [clojure.string :refer [blank?]]))
2. (ns my-ns (:require [clojure.string :refer [blank?] :as s]))
3. (ns my-ns (:require [clojure.string :refer :all]))
4. (ns my-ns (:refer-clojure :exclude [print]))
```

1. (blank?) is the only available without ns
2. same as 1, but clojure.string available as “s/”
3. all definitions in clojure.string are available (danger!)
4. I want to define my own print

References

- Daniel Solano Gomez, “How Clojure Works”, Clojure/West 2014

- “Clojure Programming”, O’Reilly