Difference between Symbols and Vars in Clojure

I'm always a bit confused about Symbols and Vars in Clojure. For example, is it safe to say that + is a symbol which is used to denote a var, and this var points to a value which is a function that can add numbers?

So what happens, step by step when I just enter "+" in a REPL?

1. The symbol gets qualified to a namespace, in this case clojure.core
2. Then in some symbol table there is the information that + refers to a var
3. When this var is evaluated, the result is a function-value?

clojure

edited Feb 10 '14 at 10:38
Roly
608 6 21

asked Feb 2 '12 at 13:44
Michiel Borkent
14.3k 11 48 86

2 you seem to understand this quite well :) – Arthur Ulfeldt Feb 2 '12 at 20:10

2 Answers

There's a symbol + that you can talk about by quoting it:

```
user=> '+
+
user=> (class '+)
clojure.lang.Symbol
user=> (resolve '+)
#'clojure.core/+
```

So it resolves to #'+, which is a Var:

```
user=> (class #'+)
clojure.lang.Var
```

The Var references the function object:

```
user=> (deref #'+)
#<core$_PLUS_ clojure.core$_PLUS_@55a7b0bf>
user=> @#'+
#<core$_PLUS_ clojure.core$_PLUS_@55a7b0bf>
```

(The @ sign is just shorthand for deref.) Of course the usual way to get to the function is to not quote the symbol:

```
user=> +
#<core$_PLUS_ clojure.core$_PLUS_@55a7b0bf>
```

Note that lexical bindings are a different mechanism, and they can shadow Vars, but you can bypass them by referring to the Var explicitly:

```
user=> (let [+ -] [(+ 1 2) (@#'+ 1 2)])
[-1 3]
```

In that last example the deref can even be left out:

```
user=> (let [+ -] [(+ 1 2) (#'+ 1 2)])
[-1 3]
```

This is because Var implements IFn (the interface for Clojure functions) by calling deref on itself, casting the result to IFn and delegating the function call to that.

The visibility mechanism used when you define private functions with defn- is based on metadata on the symbol. You can bypass it by referring directly to the Var, as above:

```
user=> (ns foo)
nil
foo=> (defn- private-function [] :secret)
#'foo/private-function
foo=> (in-ns 'user)
#<Namespace user>
user=> (foo/private-function)
java.lang.IllegalStateException: var: #'foo/private-function is not public
(NO_SOURCE_FILE:36)
user=> (#'foo/private-function)
:secret
```

edited Feb 2 '12 at 16:43

answered Feb 2 '12 at 16:28

Jouni K. Seppänen
19k 2 42 77

Adding to this excellent answer: You can also qualify symbols to bypass `let` shadowing. `(let [+ -] [(+ 1 1) (clojure.core/+ 1 1)]) => [0 2]` . This shows that symbols don't get qualified before Var resolution. – kotarak Feb 3 '12 at 12:42

@kotarak Good point about qualified symbols not being the same as those bound by let, but I don't get what it has to do with Var resolution. Care to elaborate? – Jouni K. Seppänen Feb 3 '12 at 19:18

@jouni-k-seppanen This relates to the first bullet point in the question. I thought you addressed all the other points very well, but this one went a little uncommented. (Or at least only very implicitly at the beginning.) – kotarak Feb 6 '12 at 6:58

Excellent! Thanks!! – eu-ge-ne Jul 12 '12 at 5:40

See the documentation for namespaces:

> Namespaces are mappings from simple (unqualified) symbols to Vars and/or Classes. Vars can be interned in a namespace, using def or any of its variants, in which case they have a simple symbol for a name and a reference to their containing namespace, and the namespace maps that symbol to the same var. A namespace can also contain mappings from symbols to vars interned in other namespaces by using refer or use, or from symbols to Class objects by using import.

So basically your steps 1 and 2 are unified: the namespaces are the symbol tables.

And regarding step 3: I like the definition for variables that they're combinations of names of values. The symbol is the variable's name, and evaluating it will result in its value.

answered Feb 2 '12 at 15:00

Rörd
5,181 8 18