VARS VARS
VARS VARS
VARS VARS Vars
VARS VARS
VARS VARS

Gary Fredericks
Chicago Clojure
Feb 25, 2015

Me

Hello, I am Gary Fredericks.

I am: -a software engineer at Groupon
-giving this talk
-@gfredericks_

Does this make me a professional drawer?

Vars

- important because
  - codebase organization
  - dynamic development
  - development tools
- misunderstood because
  - usage is implicit
  - lots of features
  - `(not= "var" "variable")`

Vars are a many-splendored thing
Vars lift us up where we belong
All you need is vars

Roadmap

- What Vars Even Are
- Primary Features and Uses
- Secondary Features and Uses
- Dynamic Vars
- Miscellany

What Vars Even Are

A Box

A var is a box.

A House

A var is a ~~box~~ house.

Some Houses

```
1  (ns user)
2
3  (def highest-number 24)
4
5  (defn commit-heresy
6    []
7    (inc highest-number))
```

Non-Vars

```
;; locals are not vars

(fn [a]
  (let [b (* a a)]
    (- a b b)))


;; java interop does not involve vars

(Double/valueOf Math/PI)

(.toString (java.math.BigInteger. "31" 8) 10)
```

Vars are used:

- Implicitly
- Explicitly
- Discretely

Implicit Var Usage

```
1  (ns user)
2
3  (def highest-number 24)
4
5  (defn commit-heresy
6    []
7    (inc highest-number))
```

Explicit Var Usage

```
1  (add-hook #'leiningen.deploy/deploy
2            (fn [...] ...))
3
4  ;; same as
5
6  (add-hook (var leiningen.deploy/deploy)
7            (fn [...] ...))
```

Discrete Var Usage

```
1  (with-redefs [launch-missiles
2                (constantly :launched)]
3    (run the tests))
```

Okay So

- Vars are created with def (& friends)
- Are used in three different styles

Primary Features & Uses

Primary Features & Uses

Do some deffing and see what happens

def

```
1   user> (def highest-number 24)
2   #'user/highest-number
3
4   user> highest-number
5   24
6
7   user> (- highest-number 7)
8   17
9
10  user> (def highest-number 18)
11  #'user/highest-number
12
13  user> highest-number
14  18
```

#'

```
1  user> #'user/highest-number
2  #'user/highest-number
3
4  user> #'highest-number
5  #'user/highest-number
6
7  user> (var highest-number)
8  #'user/highest-number
9
10 user> (class #'user/highest-number)
11 #<java.lang.Class class clojure.lang.Var>
```

Poking a Var

```
user> @#'highest-number
18

user> (deref #'highest-number)
18
```

Interning

```
1   user> (def my-var-1 (def highest-number 24))
2   #'user/my-var-1
3   user> my-var-1
4   #'user/highest-number
5   user> (def my-var-2 (def highest-number 19))
6   #'user/my-var-2
7   user> my-var-2
8   #'user/highest-number
9   user> (identical? my-var-1 my-var-2)
10  true
11  user> (deref my-var-1)
12  19
```

Var Metadata

```
1   user> (meta #'and)
2   {:added "1.0",
3    :arglists ([] [x] [x & next]),
4    :column 1,
5    :doc "Evaluates exprs one at a time, from left to
6         right. If a form returns logical false (nil or
7         false), and returns that value and doesn't
8         evaluate any of the other expressions,
9         otherwise it returns the value of the last
10        expr. (and) returns true.",
11   :file "clojure/core.clj",
12   :line 801,
13   :macro true,
14   :name and,
15   :ns #<clojure.lang.Namespace@63d300d2 clojure.core>}
```

Primary Features & Uses

Finding Vars

Getting a Var

```
1   user> (clojure.lang.RT/var "clojure.core" "assoc")
2   #'clojure.core/assoc
```

src/jvm/clojure/lang/Namespace.java

```java
package clojure.lang;
public class Namespace extends ... implements ...{
  // ...
  final static ConcurrentHashMap<Symbol, Namespace> namespaces =
    new ConcurrentHashMap<Symbol, Namespace>();
  transient final AtomicReference<IPersistentMap> mappings =
    new AtomicReference<IPersistentMap>();
  // ...
}
```

All The Vars

```
1  user> (all-ns)
2  (#<Namespace clojure.tools.nrepl.misc>
3   #<Namespace clojure.set>
4   #<Namespace clojure.core>
5   #<Namespace clojure.data>
6   #<Namespace user>
7   ...)
8
9  user> (the-ns 'user)
10 #<Namespace user>
```

.getMappings

```
user> (.getMappings (the-ns 'user))
{* #'clojure.core/*,
 *' #'clojure.core/*',
 *1 #'clojure.core/*1,
 *2 #'clojure.core/*2,
 *3 #'clojure.core/*3,
 *agent* #'clojure.core/*agent*,
 ...}

user> (count *1)
711
```

ns-publics

```
1  user> (ns-publics 'user)
2  {highest-number #'user/highest-number}
```

Uses

- Finding documentation
  - `clojure.repl/dir`
  - `clojure.repl/apropos`
  - `clojure.repl/find-doc`
- Finding tests
  - `clojure.test`

Primary Features & Uses

Vars and Functions

Let's use a var

```
(ns user)

(def highest-number 24)

(defn embiggen
  [n]
  (+ n 6))

(defn commit-heresy
  []
  (embiggen highest-number))

(commit-heresy) => 30
```

Vars that Contain Functions

```
1  user> commit-heresy
2  #<user$commit_heresy@3c5fac41 user$commit_heresy@3c5fac41>
3
4  user> (class #'commit-heresy)
5  #<java.lang.Class@18ef659c class clojure.lang.Var>
6
7  user> (class commit-heresy)
8  #<java.lang.Class@3ed7988a class user$commit_heresy>
```

```
1  ;; mostly equivalent
2
3  (defn commit-heresy
4    []
5    (embiggen highest-number))
6
7  (def commit-heresy
8    (fn []
9      (embiggen highest-number)))
```

Let's use a var

```
1   (ns user)
2
3   (def highest-number 24)
4
5   (defn embiggen
6     [n]
7     (+ n 6))
8
9   (defn commit-heresy
10    []
11    (embiggen highest-number))
```

commit-heresy in Java

```
1  import clojure.lang.*;
2  public final class user$commit_heresy extends AFunction {
3    public static final Var embiggenVar;
4    public static final Var highestNumberVar;
5
6    static {
7      embiggenVar = RT.var("user", "embiggen");
8      highestNumberVar = RT.var("user", "highest-number");
9    }
10
11   public user$commit_heresy(){super();}
12
13   public Object invoke(){
14     IFn embiggen = (IFn) embiggenVar.getRawRoot();
15     Object highestNumber = highestNumberVar.getRawRoot();
16     return embiggen.invoke(highestNumber);
17   }
18 }
```

Primary Features & Uses

Mutation - How?

Def again

```
1   user> (def highest-number 24)
2   #'user/highest-number
3
4   user> (defn commit-heresy [] (inc highest-number))
5   #'user/commit-heresy
6
7   user> (commit-heresy)
8   25
9
10  user> (def highest-number 18)
11  #'user/highest-number
12
13  user> (commit-heresy)
14  19
```

src/jvm/clojure/lang/Var.java

```java
package clojure.lang;
public final class Var extends ARef implements ... {
  // ...
  volatile Object root;
  // ...
  final public Object getRawRoot(){
    return root;
  }
  // ...
  synchronized public Object alterRoot
      (IFn fn, ISeq args) {
    // abridged
    Object newRoot = fn.applyTo(RT.cons(root, args));
    this.root = newRoot;
    return newRoot;
  }
}
```

On running code

```
user> (def temperature 19)
#'user/temperature
user> (def observations (atom {}))
#'user/observations
user> (future (dotimes [n 10000000]
                (swap! observations
                       update
                       temperature
                       (fnil inc 0))))
#<Future@2751f26b pending>
user> (def temperature 20)
#'user/temperature
user> @observations
{19 9129874, 20 870126}
```

alter-var-root

```
1  (def highest-number 24)
2
3  (defn commit-heresy [] (inc highest-number))
4
5  (alter-var-root #'highest-number + 100)
6  ;; => 124
7
8  (commit-heresy) ;; => 125
```

alter-meta!

```
1   (def highest-number 24)
2
3   (alter-meta! #'highest-number
4                assoc :good? true)
5
6   (meta #'highest-number)
7
8   =>
9   {:column 1,
10   :file "/tmp/form-init5058489039916962905.clj",
11   :good? true,
12   :line 1,
13   :name highest-number,
14   :ns #<clojure.lang.Namespace@3fbebf70 user>}
```

Primary Features & Uses

Mutation - Why?

Idiomatic Mutation

Gary's Idiomatic Var Mutation Rule of Thumb (GIVMRoT):

> *It is not idiomatic for an application to mutate a var as part of its normal operation (after fully starting up). Other mutations are only okay if they are super useful.*

def in a Function

```
1  ;; don't do this
2  (defn set-highest-number
3    [n]
4    (def highest-number n))
5
6  (set-highest-number 24)
7
8  highest-number ;; => 24
```

Code Reloading

You can reevaluate a single function, or a whole file, and in many cases [1][2][3][4][5]this will do exactly what you expect.

[1] maybe not with type-generation facilities like `defprotocol`, `deftype`, `defrecord`, etc.

[2] or with multimethods

[3] or values with `^:const` or functions with `^:inline`

[4] or when you're redefining a macro

[5] is AOT related to this? who even knows?

with-redefs

```
1  (def highest-number 24)
2
3  (defn commit-heresy [] (inc highest-number))
4
5  (with-redefs [highest-number 900]
6    (commit-heresy))
7  ;; => 901
8
9  (commit-heresy) ;; => 25
```

robert.hooke

```
1  (require '[robert.hooke :refer [add-hook]])
2
3  (defn save-things
4    [things]
5    (do some things))
6
7  (add-hook #'save-things
8            :logging
9            (fn [orig things]
10             (log/info "Start!")
11             (orig things)
12             (log/info "Done!")))
```

Improved development experience

- thalia – changes docstrings to more detailed and beginner-friendly versions
- dynalint – redefines core functions to check for bad arguments

Primary Features & Uses

Finally

Okay So

- Implicit use means always dereferencing
- Var updates: root & metadata
- GIVMRoT

Secondary Features & Uses

Secondary Features & Uses

Compile-Time Features

^:private

```clojure
(def ^:private highest-number 24)

(defn ^:private encrypt
  [x]
  (bit-xor x highest-number))

(defn- encrypt
  [x]
  (bit-xor x highest-number))

(-> #'encrypt meta :private) => true
```

^:const

```
1  (def ^:const TAU (* 2 Math/PI))
```

^:inline

```
(defn <
  "Returns non-nil if nums are in monotonically
  increasing order, otherwise false."
  {:inline (fn [x y]
             `(. clojure.lang.Numbers (lt ~x ~y)))
   :inline-arities #{2}
   :added "1.0"}
  ([x] true)
  ([x y] (. clojure.lang.Numbers (lt x y)))
  ([x y & more]
     (if (< x y)
       (if (next more)
         (recur y (first more) (next more))
         (< y (first more)))
       false)))
```

defonce

```
1  user> (defonce highest-number 24)
2  #'user/highest-number
3
4  user> (defonce highest-number 25)
5  nil
6
7  user> highest-number
8  24
```

declare

```
1  (declare is-this-number-odd?)
2
3  (defn is-this-number-even?
4    [n]
5    (or (zero? n) (is-this-number-odd? (dec n))))
6
7  (defn is-this-number-odd?
8    [n]
9    (is-this-number-even? (dec n)))
```

declare

Mostly the same as:

```
1  (def is-this-number-odd?)
2
3  (defn is-this-number-even?
4    [n]
5    (or (zero? n) (is-this-number-odd? (dec n))))
6
7  (defn is-this-number-odd?
8    [n]
9    (is-this-number-even? (dec n)))
```

Unbound

```
1   user> (def just-a-def)
2   #'user/just-a-def
3
4   user> just-a-def
5   #<clojure.lang.Var$Unbound@2be3c053 Unbound:
6     #'user/just-a-def>
7
8   user> (just-a-def)
9   ;; java.lang.IllegalStateException:
10  ;; Attempting to call unbound fn: #'user/just-a-def
```

.isMacro

```
1   (defmacro comment
2     [& args]
3     nil)
4
5   (meta #'comment)
6   =>
7   {:arglists ([& args]),
8    :column 1,
9    :file "/tmp/form-init4027054685560776019.clj",
10   :line 1,
11   :macro true,
12   :name comment,
13   :ns #<clojure.lang.Namespace user>}
14
15  (.isMacro #'comment) => true
```

(def defmacro

```
1  (def defmacro
2    (fn [&form &env
3         name & args]
4      (let [prefix            ...
5            fdecl             ...
6            fdecl             ...
7            add-implicit-args ...
8            add-args          ...
9            fdecl             ...
10           decl              ...]
11       (list 'do
12             (cons `defn decl)
13             (list '. (list 'var name) '(setMacro))
14             (list 'var name)))))
15
16 (. (var defmacro) (setMacro))
```

Secondary Features & Uses

A Var is an IFn

A Var is an IFn

```
1  user> (#'commit-heresy)
2  25
3  user> (#'clojure.core/>1? 7)
4  => true
```

(def f (HOF g ...))

```clojure
(defn foo
  [x]
  (bar "heyo" x))

;; vs

(def foo (partial bar "heyo"))

;; w.r.t.

(with-redefs [bar (constantly :stubbed)]
  (foo x))
```

(def f (HOF g ...))

```
1  (def foo (partial #'bar "heyo"))
```

Multimethods

- Can't change dispatch function
- But dispatch function can be a var

Secondary Features & Uses

Reference Features

watchers

```
1  (def highest-number 24)
2
3  (add-watch #'highest-number :k #(prn %&))
4
5  (alter-var-root #'highest-number inc)
6  => 25
7  ;; Prints:
8  ;;   (:k #'user/highest-number 24 25)
```

potemkin/import-vars

```clojure
(ns my.lib.internal)

(defn multiplicatify
  "Like * but with more
  indirection and fewer arities."
  [a b]
  (* a b))

(ns my.lib
  (:require [potemkin :refer [import-vars]]))

(import-vars [my.lib.internal multiplicatify])
```

potemkin/import-vars

```
1  (defn link-vars
2    "Makes sure that all changes to
3    ‘src‘ are reflected in ‘dst‘."
4    [src dst]
5    (add-watch src dst
6               (fn [_ src old new]
7                 (alter-var-root dst (constantly @src))
8                 (alter-meta! dst
9                              merge
10                             (dissoc (meta src)
11                                     :name)))))
```

Validators

```clojure
(def highest-number 24)

(set-validator! #'highest-number
                (fn [n]
                  (and (number? n)
                       ;; numbers less than ten
                       ;; aren't very high
                       (>= n 10))))

(alter-var-root #'highest-number (constantly -15))
;; java.lang.IllegalStateException:
;; Invalid reference state
```

Secondary Features & Uses

Finally

Okay So

- Lots of compiler options
- Vars are functions
- Watchers and validators

Dynamic Vars

Dynamic Vars

Mechanics

^:dynamic

```
1   (def ^:dynamic *highest-number* 24)
2
3   (defn commit-heresy
4     []
5     (inc *highest-number*))
6
7   (commit-heresy) => 25
8
9   (:dynamic (meta #'*highest-number*)) => true
10  (.isDynamic #'*highest-number*) => true
```

binding

```
1  user> (binding [*highest-number* 17] (commit-heresy))
2  18
3
4  user> (commit-heresy)
5  25
```

```
 1  (defn double-the-highest-number!
 2    []
 3    (set! *highest-number* (* 2 *highest-number*)))
 4
 5  (double-the-highest-number!)
 6  ;; java.lang.IllegalStateException:
 7  ;; Can't change/establish root binding of:
 8  ;; *highest-number* with set
 9
10  (binding [*highest-number* 17]
11    (let [x1 (commit-heresy)]
12      (double-the-highest-number!)
13      (let [x2 (commit-heresy)]
14        [x1 x2])))
15  => [18 35]
```

Crossing Threads

- Futures and agents intentionally copy thread-local bindings from your thread before executing tasks
- bound-fn can be used in other concurrency contexts when you need to share bindings
  - compare at a repl:

```
1  (.start (Thread. (fn [] (println "raw thread"))))
2
3  (future (println "future"))
```

Dynamic Vars

Usage Styles

User-Bound Dynamic Vars

- Normally are not dynamically-bound
- `*out*` (compare to `System/out`)
  - `*in*`, `*err*`
- `*read-eval*`
- Connections in some IO libs

Context-Bound Read-Only Dynamic Vars

- Repl vars: `*1`, `*2`, `*3`, `*e`
- robert.bruce
- `*agent*`
- `*ns*` (when compiling)

Context-Bound `set!` able Dynamic Vars

- Mostly compiler flags
- `*warn-on-reflection*`
- `*unchecked-math*`
- `*assert*`

Dynamic Vars in clojure.core

```
 1  #'*1                              #'*fn-loader*
 2  #'*2                              #'*in*
 3  #'*3                              #'*math-context*
 4  #'*agent*                         #'*ns*
 5  #'*allow-unresolved-vars*         #'*out*
 6  #'*assert*                        #'*print-dup*
 7  #'*clojure-version*               #'*print-length*
 8  #'*command-line-args*             #'*print-level*
 9  #'*compile-files*                 #'*print-meta*
10  #'*compile-path*                  #'*print-readably*
11  #'*compiler-options*              #'*read-eval*
12  #'*data-readers*                  #'*source-path*
13  #'*default-data-reader-fn*        #'*unchecked-math*
14  #'*e                              #'*use-context-classloader*
15  #'*err*                           #'*verbose-defrecords*
16  #'*file*                          #'*warn-on-reflection*
17  #'*flush-on-newline*              #'pr
```

Dynamic Vars

Implementation

Var.java: Frame

```
public final class Var ... {
  static class Frame{
    final static Frame TOP =
      new Frame(PersistentHashMap.EMPTY, null);
    Associative bindings;
    Frame prev;

    public Frame(Associative bindings, Frame prev){
      this.bindings = bindings;
      this.prev = prev;
    }
  }

  static final ThreadLocal<Frame>dvals=new ThreadLocal<Frame>(){
    protected Frame initialValue(){
      return Frame.TOP;
    }
  };
}
```

Var.java: TBox

```
1   public final class Var ... {
2     static class TBox{
3
4       volatile Object val;
5       final Thread thread;
6
7       public TBox(Thread t, Object val){
8         this.thread = t;
9         this.val = val;
10      }
11    }
12  }
```

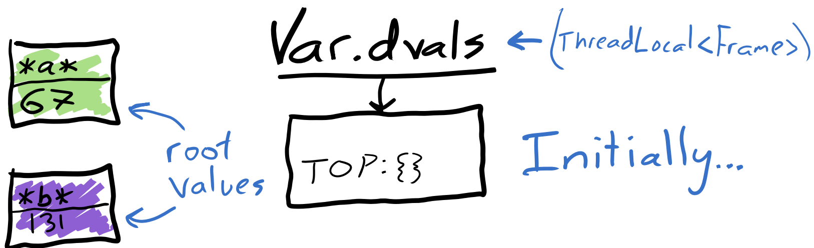
Var.java: .deref

```
1   public final class Var ... {
2     final public Object deref(){
3       TBox b = getThreadBinding();
4       if(b != null)
5         return b.val;
6       return root;
7     }
8     public final TBox getThreadBinding(){
9       if(threadBound.get())
10        {
11          IMapEntry e = dvals.get().bindings.entryAt(this);
12          if(e != null)
13            return (TBox) e.val();
14        }
15      return null;
16    }
17  }
```

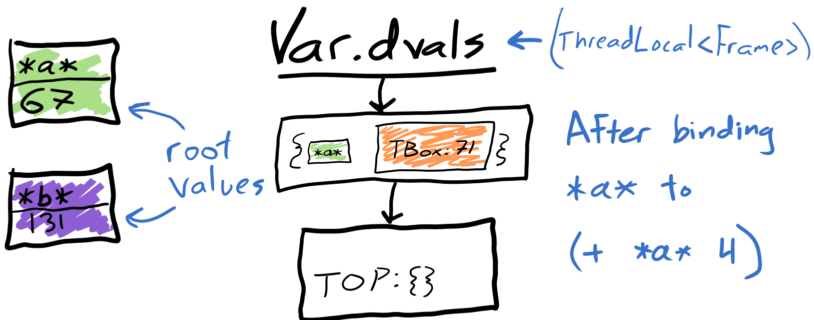
Example

```
1  (def ^:dynamic *a* 67)
2  (def ^:dynamic *b* 131)
3
4  (binding [*a* (+ *a* 4)]
5    (binding [*b* (+ *b* 6)]
6      (set! *a* (+ 2 *a*)))
7    [*a* *b*])
8  => [73 131]
```

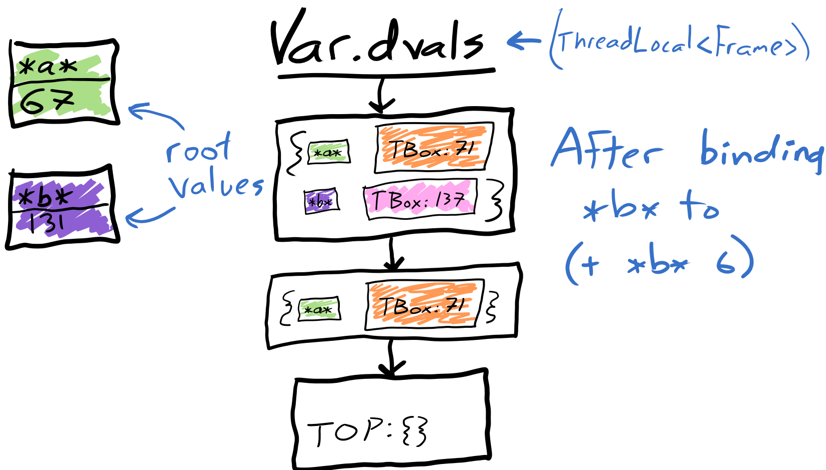
Step 0



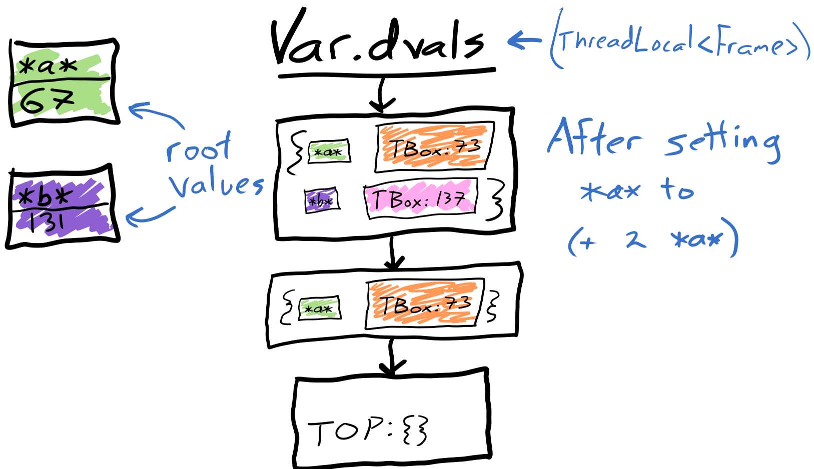
Step 1



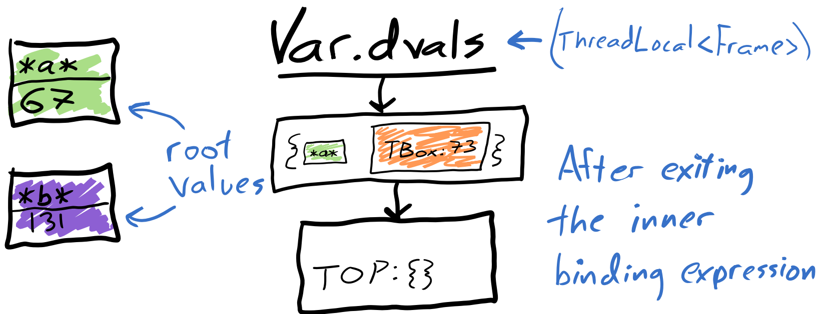
Step 2



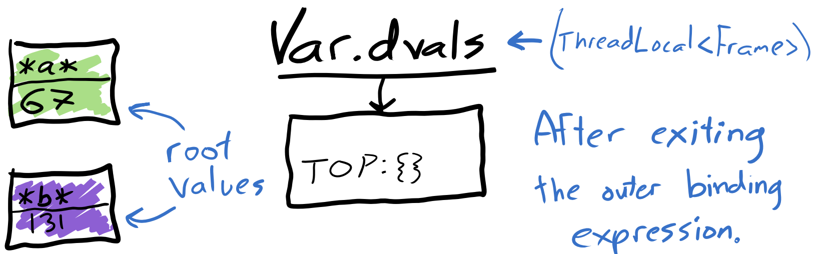
Step 3



Step 4



Step 5



Various Var Usages – Clojure

```clojure
(def ^:const a 12)
(def b 24)
(def ^:dynamic *c* 36)

(defn add-and-stringificate
  [d]
  (str (+ a b *c* d)))
```

Various Var Usages – Java

```
1   import clojure.lang.*;
2   public final class user$add_and_stringificate extends AFunction{
3     public static final Var str, b, c;
4     static {
5       strVar = RT.var("clojure.core","str");
6       bVar = RT.var("user","b");
7       cVar = RT.var("user","*c*");
8     }
9
10    public Object invoke(Object d){
11      IFn str = (IFn)strVar.getRawRoot();
12      long a = 12;
13      Object b = bVar.getRawRoot();
14      Number tmp1 = Numbers.add(a, b);
15      Object c = cVar.get();
16      Number tmp2 = Numbers.add(tmp1, c);
17      Number tmp3 = Numbers.add(tmp2, d);
18      return str.invoke(tmp3);
19    }
20  }
```

Dynamic Vars

Finally

Okay So

- Thread-local values
- Can be updated in two different ways
- Lots of use cases & use styles

Miscellany

(doc with-local-vars)

```
1  -------------------------
2  clojure.core/with-local-vars
3  ([name-vals-vec & body])
4  Macro
5    varbinding=> symbol init-expr
6
7    Executes the exprs in a context in which the
8    symbols are bound to vars with per-thread bindings
9    to the init-exprs. The symbols refer to the var
10   objects themselves, and must be accessed with
11   var-get and var-set
```

what?

Well try it

```
1   user> (with-local-vars [v 24] v)
2   #<Var: --unnamed-->
3
4   user> (with-local-vars [v 24] (class v))
5   #<java.lang.Class@5677da01 class clojure.lang.Var>
6
7   user> (with-local-vars [v 24] (deref v))
8   24
9
10  user> (with-local-vars [v 24] (var-get v))
11  24
12
13  user> (with-local-vars [v 24] (meta v))
14  {:name nil, :ns nil}
```

```clojure
(def the-server
  (-> (read-config)
      (create-server-component)
      (component/start)))
```

lein-shorthand

Lets you configure an alias namespace with a tiny name like `.` or `&`.

```
1  (./dir .)
2  ;; add-dep
3  ;; add-hook
4  ;; apropos
5  ;; bg
6  ;; bg-deref
7  ;; break!
8  ;; dir
9  ;; doc
10 ;; mexpand-all
11 ;; pp
12 ;; pst
13 ;; remove-hook
14 ;; source
15 ;; unbreak!
16 ;; unbreak!!
```

bg

```
1  user> (./bg (Thread/sleep 500) (inc 24))
2  #<bg0 has been running for 0.000 seconds>
3  ;; Starting background task bg0
4  ;; #<DONE: bg0 ran for 0.501 seconds>
5  user> bg0
6  25
```

This is the beginning of the end

Review

- Vars are Clojure glue
- Vars are reference types
- Dynamic vars

Okay thank you for listening
do we have time for questions
if so I can do that or regardless
you can ask me later or on twitter or
something also thank you to Joe Hirn and
and to Groupon and Sean Massa and for that matter to
my wife who has to singlehandedly convince both of the children to not
only eat their dinner but also prepare for and actually begin to sleep. Speaking of which,
back when I used to spend more time sitting in the dark waiting for children to fall asleep than I've had to lately, I decided to try to pass the time by practicing mentally
factoring arbitrary three-digit numbers, and there are some interesting tricks for testing divisibility like let's say you're trying to determine if 401 is divisible by 17. Well obviously
you should be able to add or subtract arbitrary multiples of 17 without changing the answer so the trick is picking the right multiple to reach a multiple of 10 and it turns out that there's only four possibilities you have to
worry about depend on the last digit of the number to be factored (1,3,7,9). In this case we have 401 and we'll want to subtract 17x3=51 which gets us to 350 from whence we can immediately factor out the 10 to get 35 which is obviously not a multiple of 17 and so neither is 401.