Vars VARS VARS

VARS VARS VARS VARS

VARS VARS

#'vars vars

Gary Fredericks
CLJ PDX
2018-11-01

Me

Hello, I am
Gary Fredericks

I am: 1) a software engineer at DRW

2) giving this talk

3) @gfredericks_

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

What Vars Even Are

- Examples
- Non-Examples
- Three Usage Styles

Some Vars

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

Some Non-Vars

```clojure
;; 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)
```

Usage Styles

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  (alter-var-root #'my.app/system
2                  (constantly ...))
3
4  ;; same as
5
6  (alter-var-root (var my.app/system)
7                  (constantly ...))
```

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

- Basic Mechanics
- Runtime Hierarchy
- Vars & Functions
- Mutation Mechanics
- Mutation Use Cases

Primary Features & Uses

Basic Mechanics

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
```

#'

```
user> #'user/highest-number
#'user/highest-number

user> #'highest-number
#'user/highest-number

user> (var highest-number)
#'user/highest-number

user> (class #'user/highest-number)
clojure.lang.Var
```

Poking a Var

```
1  user> @#'highest-number
2  18
3
4  user> (deref #'highest-number)
5  18
```

Interning

```
user> (def my-var-1 (def highest-number 24))
#'user/my-var-1

user> my-var-1
#'user/highest-number

user> (def my-var-2 (def highest-number 19))
#'user/my-var-2

user> my-var-2
#'user/highest-number

user> (identical? my-var-1 my-var-2)
true

user> (deref my-var-1)
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 #object[clojure.lang.Namespace
16               0x6821ea29
17               "clojure.core"]}
```

Basic Mechanics

- `def`, `#'`, `deref`
- interning
- metadata

Primary Features & Uses

Runtime Hierarchy

src/jvm/clojure/lang/{Namespace,Var}.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>();
  // ...
}

public class Var extends ... implements ...{
  // ...
  volatile Object root;
  // ...
  public final Symbol sym;
  public final Namespace ns;
  // ...
}
```

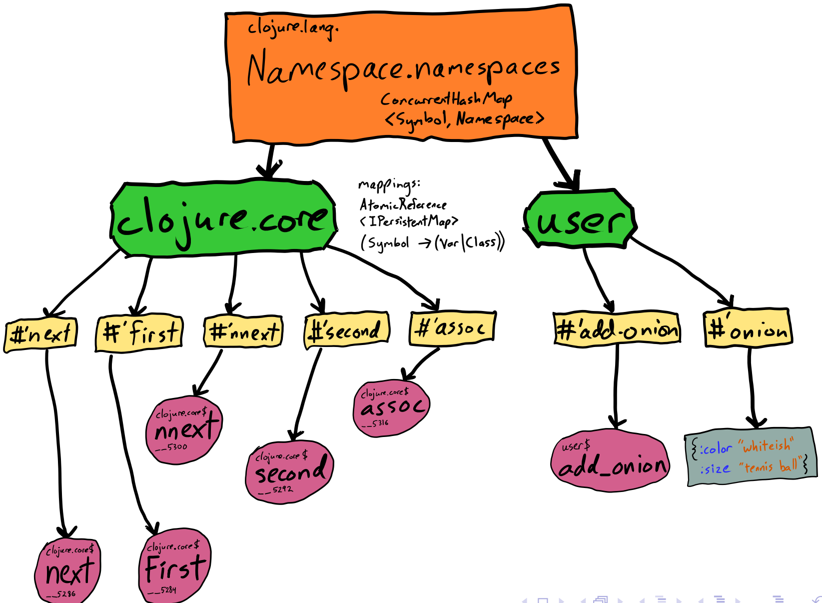
Onion Code

```
(ns user)

(def onion
  {:color "whiteish"
   :size  "tennis ball"})

(defn add-onion
  [m]
  (assoc m :onion onion))
```

Onion Hierarchy



Getting a Var

```
1  user> (resolve 'assoc)
2  #'clojure.core/assoc
3
4  user> (ns-resolve 'clojure.core 'assoc)
5  #'clojure.core/assoc
6
7  ;; creates namespace and/or var if they don't exist
8  user> (clojure.lang.RT/var "clojure.core" "assoc")
9  #'clojure.core/assoc
```

All The Vars

```
1  user> (all-ns)
2  (#object[clojure.lang.Namespace 0x79145d5a
3           "clojure.core.server"]
4   #object[clojure.lang.Namespace 0x4c4d27c8
5           "com.gfredericks.user"]
6   #object[clojure.lang.Namespace 0x338494fa
7           "clojure.core.protocols"]
8   #object[clojure.lang.Namespace 0x5af3a0f
9           "clojure.core"]
10  #object[clojure.lang.Namespace 0x24b6b8f6
11          "user"]
12  ...)
13
14 user> (the-ns 'user)
15 #object[clojure.lang.Namespace 0x24b6b8f6
16         "user"]
```

.getMappings

```
1  user> (.getMappings (the-ns 'user))
2  {*        #'clojure.core/*,
3   *'       #'clojure.core/*',
4   *1       #'clojure.core/*1,
5   *2       #'clojure.core/*2,
6   *3       #'clojure.core/*3,
7   *agent* #'clojure.core/*agent*,
8   ...}
9
10 user> (count *1)
11 765
```

ns-publics

```
1  user> (ns-publics 'user)
2  {add-onion #'user/add-onion
3   onion     #'user/onion}
```

Uses

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

Runtime Hierarchy

- Runtime hierarchy of namespaces & vars
- API for interrogating the hierarchy
- Uses for interrogation

Primary Features & Uses

Vars and Functions

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))
12
13  (commit-heresy) => 30
```

Vars that Contain Functions

```
1  user> commit-heresy
2  #object[user$commit_heresy
3          0x33e01298
4          "user$commit_heresy@33e01298"]
5
6  user> (class #'commit-heresy)
7  clojure.lang.Var
8
9  user> (class commit-heresy)
10 user$commit_heresy
```

```
;; mostly equivalent

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

(def commit-heresy
  (fn []
    (embiggen highest-number)))
```

Let's use a var

```clojure
(ns user)

(def highest-number 24)

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

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

```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 }
```

Vars and Functions

- Vars contain function objects
- Those function objects contain references to any vars they use

Primary Features & Uses

Mutation Mechanics

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

```
1  package clojure.lang;
2  public final class Var extends ARef implements ... {
3    // ...
4    volatile Object root;
5    // ...
6    final public Object getRawRoot(){
7      return root;
8    }
9    // ...
10   synchronized public Object alterRoot
11       (IFn fn, ISeq args) {
12     // abridged
13     Object newRoot = fn.applyTo(RT.cons(root, args));
14     this.root = newRoot;
15     return newRoot;
16   }
17 }
```

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))))
#object[clojure.core$future_call$reify__8334
        0x56f6d40b
        {:status :pending, :val nil}]
user> (def temperature 20)
#'user/temperature
user> @observations
{19 9129874, 20 870126}
```

def in a Function

```clojure
;; don't do this
(defn set-highest-number
  [n]
  (def highest-number n))

(set-highest-number 24)

highest-number ;; => 24
```

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!

```clojure
(def highest-number 24)

(alter-meta! #'highest-number
             assoc :good? true)

(meta #'highest-number)

=>
{:column 7,
 :file "NO_SOURCE_PATH",
 :good? true,
 :line 1,
 :name highest-number,
 :ns #object[clojure.lang.Namespace
             0x7e3060d8
             "user"]}
```

Mutation Mechanics

- repeat `def`
- mutation visible to running code
- `alter-var-root`
- `alter-meta!`

Primary Features & Uses

Mutation Use Cases

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.*

You can reevaluate a single function, or a whole file, and in many cases [1][2][3][4][5][6]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 with direct linking

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

[5]or when you're redefining a macro

[6]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

```clojure
(require '[robert.hooke :refer [add-hook]])

(defn save-things
  [things]
  (do some things))

(add-hook #'save-things
          :logging
          (fn [orig things]
            (log/info "Start!")
            (orig things)
            (log/info "Done!")))
```

Improved development experience

- thalia – changes docstrings to more detailed and beginner-friendly versions
- clojure.spec – instrumentation redefines functions to check arguments

Mutation Use Cases

- GIVMRoT
- Code reloading
- `with-redefs`
- robert.hooke
- thalia, clojure.spec instrumentation

Primary Features & Uses

Finally

Okay So

- Basic Mechanics
- Runtime Hierarchy
- Vars & Functions
- Mutation Mechanics
- Mutation Use Cases

Secondary Features & Uses

Secondary Features & Uses

- Compile-Time Features
- `Var implements IFn`
- Reference Features

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

```
1  (defn <
2    "Returns non-nil if nums are in monotonically
3    increasing order, otherwise false."
4    {:inline (fn [x y]
5               `(. clojure.lang.Numbers (lt ~x ~y)))
6     :inline-arities #{2}
7     :added "1.0"}
8    ([x] true)
9    ([x y] (. clojure.lang.Numbers (lt x y)))
10   ([x y & more]
11      (if (< x y)
12        (if (next more)
13          (recur y (first more) (next more))
14          (< y (first more)))
15        false)))
```

Direct Linking

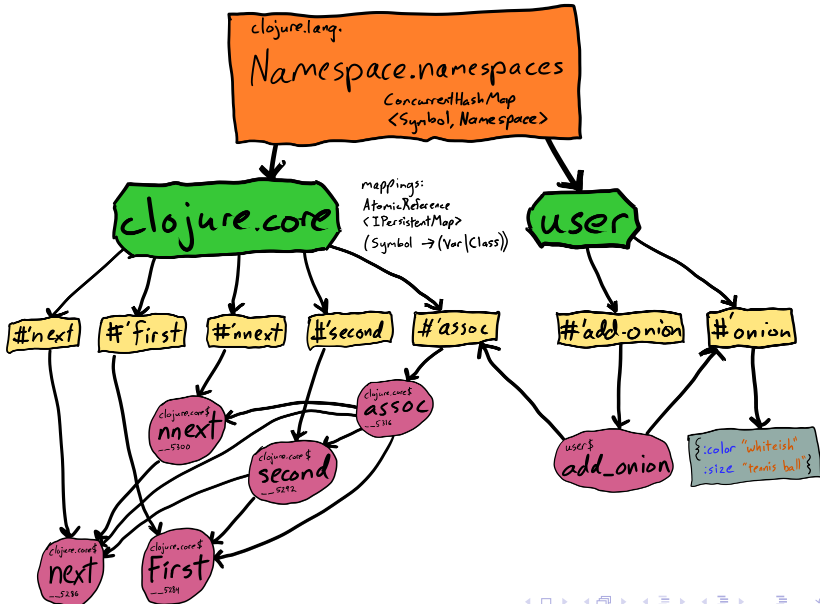
Compiling with direct-linking bypasses vars for direct function calls.

`^:redef` metadata opts out of this behavior.

Direct Linking

```
1  ;; (clojure.repl/source rand-int)
2  (defn rand-int
3    "Returns a random integer between
4     0 (inclusive) and n (exclusive)."
5    {:added "1.0", :static true}
6    [n] (int (rand n)))
7
8  (defn my-rand-int
9    [n] (int (rand n)))
10
11 (with-redefs [rand (constantly 0)]
12   {'rand-int    (rand-int 1000000)
13    'my-rand-int (my-rand-int 1000000)})
14 ;; =>
15 {clojure.core/rand-int 713186
16  user/my-rand-int      0}
```

Hierarchy w/ Direct Linking



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

```
;; mostly the same as
;; (def is-this-number-odd?)
(declare is-this-number-odd?)

(defn is-this-number-even?
  [n]
  (or (zero? n) (is-this-number-odd? (dec n))))

(defn is-this-number-odd?
  [n]
  (is-this-number-even? (dec n)))
```

Unbound

```
user> (def just-a-def)
#'user/just-a-def

user> just-a-def
#object[clojure.lang.Var$Unbound
        0x58359ebd
        "Unbound: #'user/just-a-def"]

user> (just-a-def)
;; Evaluation error (IllegalStateException) at
;; clojure.lang.Var$Unbound.throwArity (Var.java:45).
;; 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 "NO_SOURCE_PATH",
10   :line 1,
11   :macro true,
12   :name comment,
13   :ns #object[clojure.lang.Namespace 0x338494fa
14               "user"]}
15
16  (.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))
```

Compile-Time Features

- `^:private`
- `^:const`
- `^:inline`
- Direct Linking
- `defonce`
- `declare`
- `Unbound`
- `.isMacro`, `.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 ...))

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

Multimethods

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

A Var is an IFn

- Vars proxy to the functions they contain
- Using functions at compile-time
- Multimethods

Secondary Features & Uses

Reference Features

watchers

```
(def highest-number 24)

(add-watch #'highest-number :k prn)

(alter-var-root #'highest-number inc)
=> 25
;; Prints:
;;    :k #'user/highest-number 24 25
```

potemkin/import-vars

```
1  (ns my.lib.internal)
2
3  (defn multiplicatify
4    "Like * but with more
5    indirection and fewer arities."
6    [a b]
7    (* a b))
8
9  (ns my.lib
10   (:require [potemkin :refer [import-vars]]))
11
12 (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

```
(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

highest-number
;; => 24
```

Reference Features

- watchers (`potemkin/import-vars`)
- validators

Secondary Features & Uses

Finally

Okay So

- Compile-Time Features
- `Var implements IFn`
- Reference Features

Dynamic Vars

Dynamic Vars

- Mechanics
- Usage Styles
- Implementation

Dynamic Vars

Mechanics

^:dynamic

```clojure
(def ^:dynamic *highest-number* 24)

(defn commit-heresy
  []
  (inc *highest-number*))

(commit-heresy) => 25

(:dynamic (meta #'*highest-number*)) => true
(.isDynamic #'*highest-number*) => true
```

binding

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

set!

```
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

```
1  (with-out-str
2    (println "captured!"))
3
4  (with-out-str
5    @(future
6       (println "also captured!")))
7
8  (with-out-str
9    (doto (Thread. #(println "not captured"))
10     (.start)
11     (.join)))
```

Mechanics

- `^:dynamic`
- `binding`
- `set!`
- `bound-fn`, futures, agents

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

Usage Styles

- User-bound vars (`*out*`, etc.)
- Context-bound read-only vars (`*1`, etc.)
- Context-bound `set!` able vars (`*warn-on-reflection*`, etc.)

Dynamic Vars

Implementation

Implementation

- `Frame`, `TBox`
- Illustrated example
- Compilation of dynamic vs regular var use

Var.java: Frame

```
1  public final class Var ... {
2    static class Frame{
3      final static Frame TOP =
4        new Frame(PersistentHashMap.EMPTY, null);
5      Associative bindings;
6      Frame prev;
7
8      public Frame(Associative bindings, Frame prev){
9        this.bindings = bindings;
10       this.prev = prev;
11     }
12   }
13
14   static final ThreadLocal<Frame>dvals=new ThreadLocal<Frame>(){
15     protected Frame initialValue(){
16       return Frame.TOP;
17     }
18   };
19 }
```

```java
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 }
```

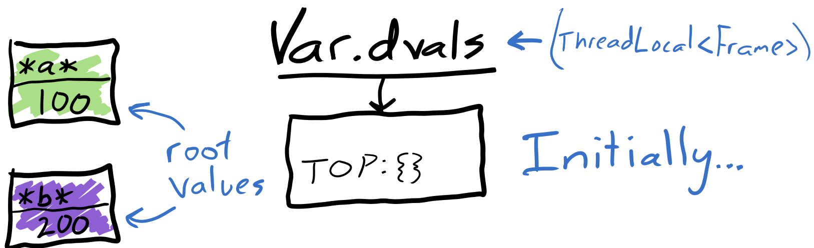
Var.java: .deref

```
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 }
```

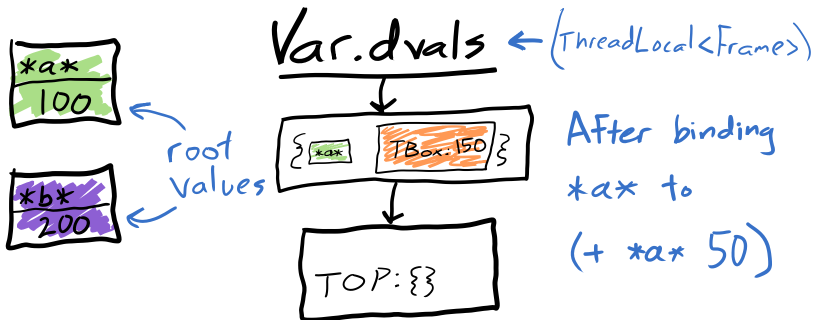
Example

```
1  (def ^:dynamic *a* 100)
2  (def ^:dynamic *b* 200)
3
4  (binding [*a* (+ *a* 50)]
5    (binding [*b* (+ *b* 60)]
6      (set! *a* (+ 4 *a*)))
7    [*a* *b*])
8  => [154 200]
```

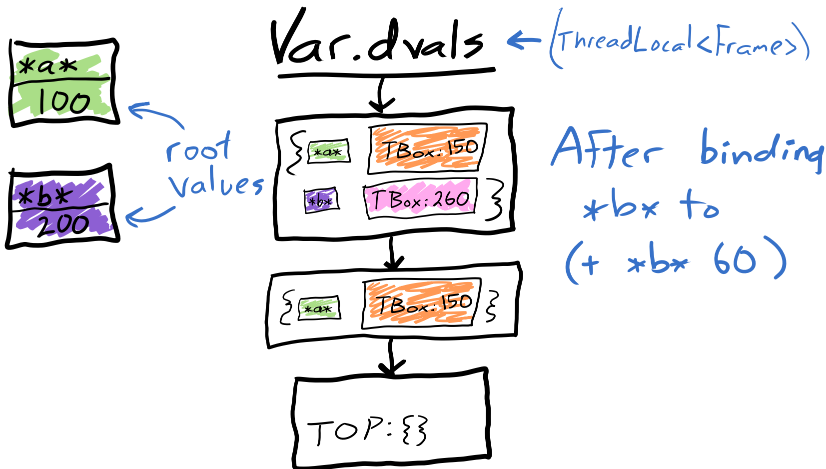
Step 0



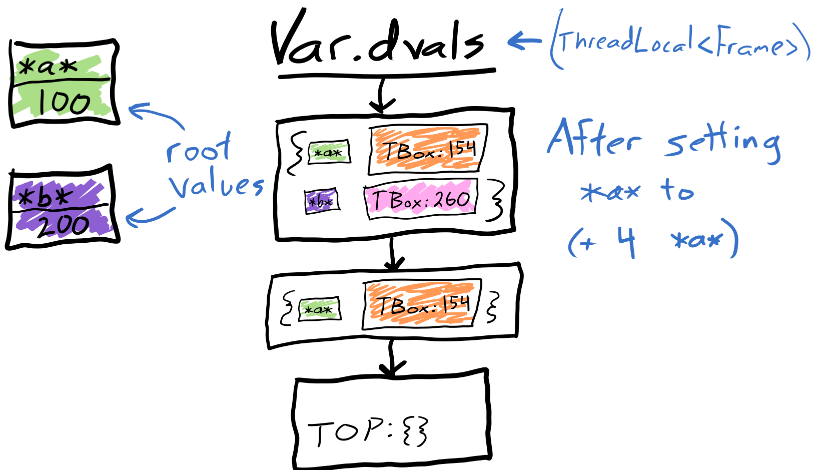
Step 1



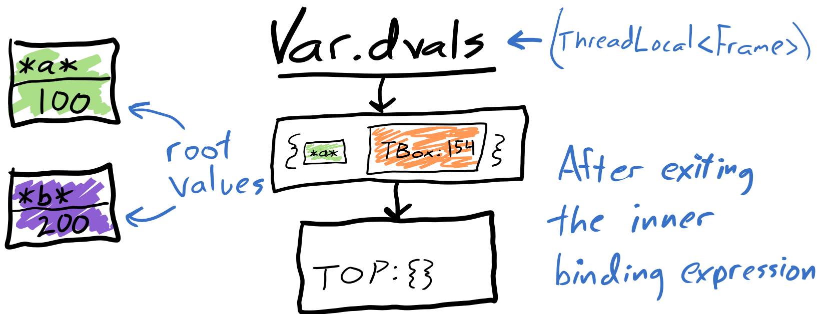
Step 2



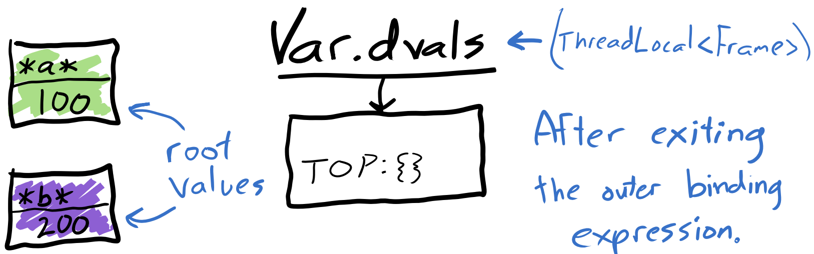
Step 3



Step 4



Step 5



Various Var Usages – Clojure

```
1  (def ^:const a 12)
2  (def b 24)
3  (def ^:dynamic *c* 36)
4
5  (defn add-and-stringificate
6    [d]
7    (str (+ a b *c* d)))
```

Various Var Usages – Java

```java
import clojure.lang.*;
public final class user$add_and_stringificate extends AFunction{
  public static final Var strVar, bVar, cVar;
  static {
    strVar = RT.var("clojure.core","str");
    bVar   = RT.var("user","b");
    cVar   = RT.var("user","*c*");
  }

  public Object invoke(Object d){
    IFn str      = (IFn)strVar.getRawRoot();
    long a       = 12;
    Object b     = bVar.getRawRoot();
    Number tmp1 = Numbers.add(a, b);
    Object c     = cVar.get();
    Number tmp2 = Numbers.add(tmp1, c);
    Number tmp3 = Numbers.add(tmp2, d);
    return str.invoke(tmp3);
  }
}
```

Various Var Usages – Java w/ Direct Linking

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

Implementation

- `Frame`, `TBox`
- Illustrated example
- Compilation of dynamic vs regular var use

Dynamic Vars

Finally

Okay So

- Mechanics
- Usage Styles
- Implementation

Miscellany

(doc with-local-vars)

```
-------------------------
clojure.core/with-local-vars
([name-vals-vec & body])
Macro
  varbinding=> symbol init-expr

  Executes the exprs in a context in which the
  symbols are bound to vars with per-thread bindings
  to the init-exprs. The symbols refer to the var
  objects themselves, and must be accessed with
  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   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}
```

```
1  (def the-server
2    (-> (read-config)
3        (create-server-component)
4        (component/start)))
```

dot-slash-2

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!!
```

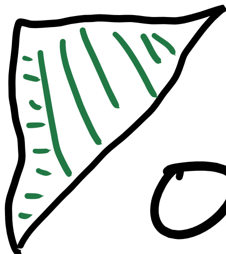
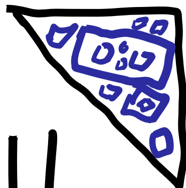
bg

```
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

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

Okay well that's it.