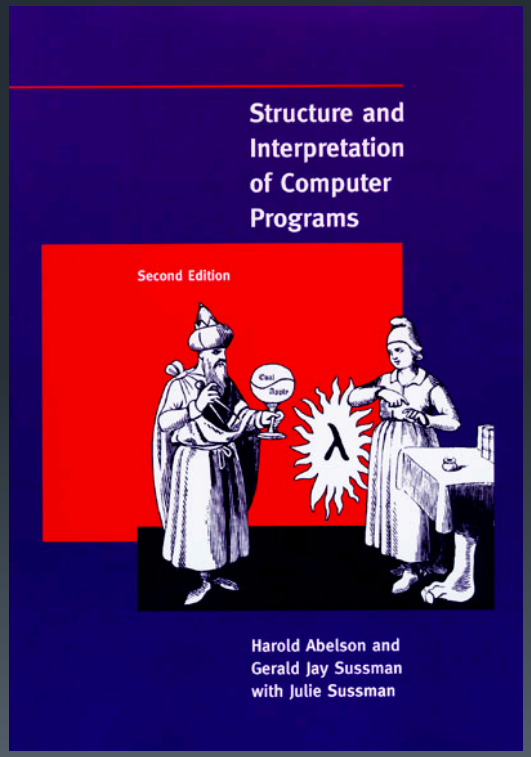
Clojure

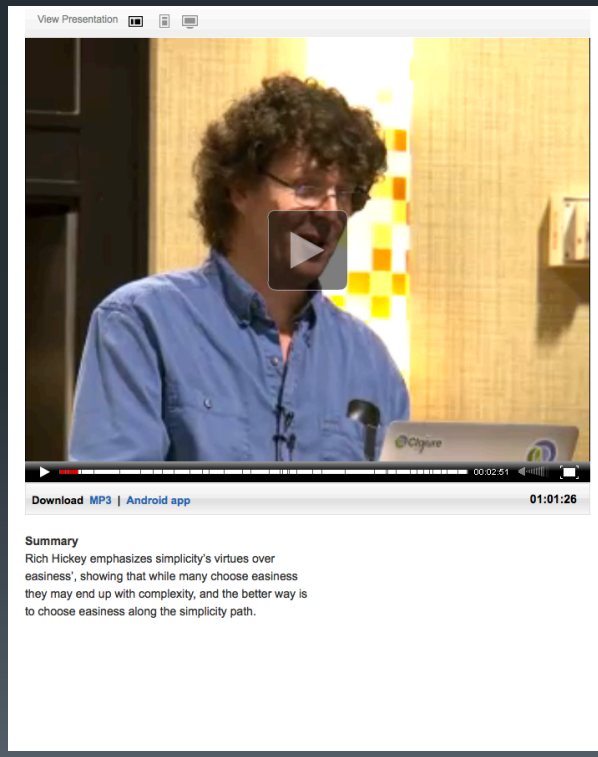
I seldom end up where I wanted to go, but almost always end up where I need to be. – Douglas Adams

First Steps



https://mitpress.mit.edu/sicp/full-text/book/book.html

First Steps



Word Origins

- Simple
  - *sim- plex*
  - one fold/braid
  - vs complex
- Easy
  - *ease < aise < adjacens*
  - lie near
  - vs hard

http://www.infoq.com/presentations/Simple-Made-Easy

What is Clojure?

- Simple: untangled
- Powerful: adequate to complete tasks

Simple

- Lisp
  - Separate reading from evaluation
  - Tiny language syntax
- Functional
  - Pure Functions
  - Immutable Data
  - Sequence
- Polymorphism
  - Separate polymorphism from derivation
- Time Model
  - Separate values, identities, state, and time

Simple

- Lisp
  - Separate reading from evaluation
  - Tiny language syntax
- Functional *mostly*
  - Pure Functions
  - Immutable Data
  - Sequence
- Polymorphism *a la carte*
  - Separate polymorphism from derivation
- Time Model
  - Separate values, identities, state, and time

Power

- Java/Javascript interop
- Lisp
  - Compiler and Macros
- Pragmatic
  - That's why I said *mostly* functional

Why do I Care?

Why am I so excited?

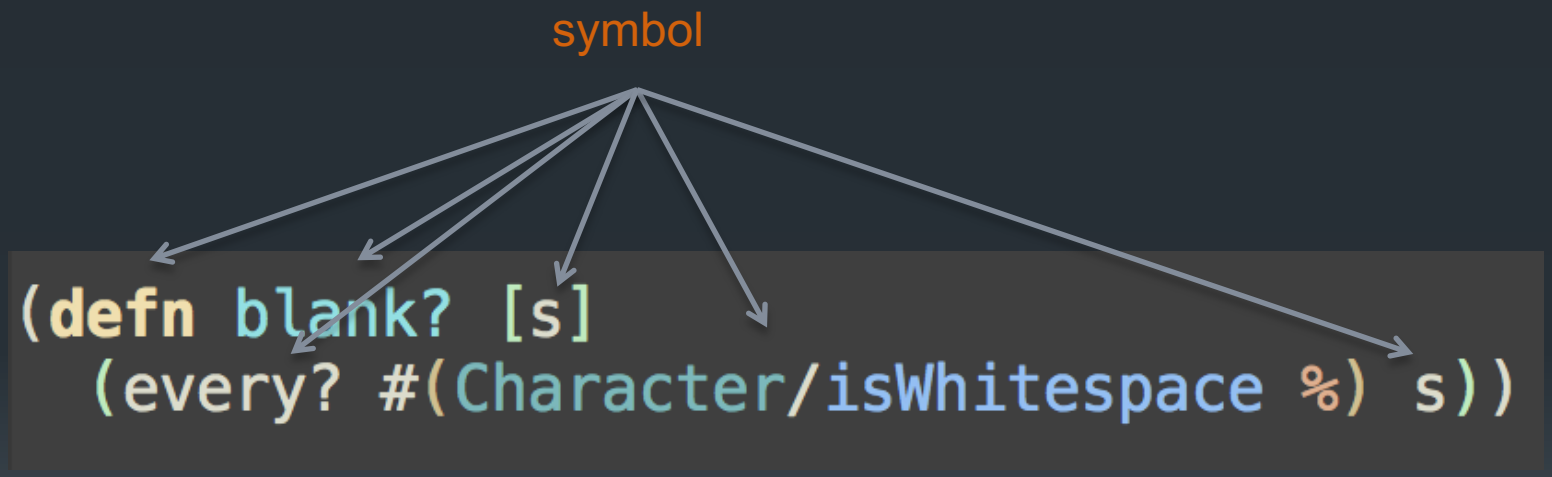
Syntax

- Literals: `\a, "string", 42, :tag, true, false, nil`
- Symbols: `+, defn, foo,`
  - resolve to something
- Composites: `[], {}, #{}`
  - Evaluated values of contained objects
- Lists: ()
  - calls
  - Special form, macro, function

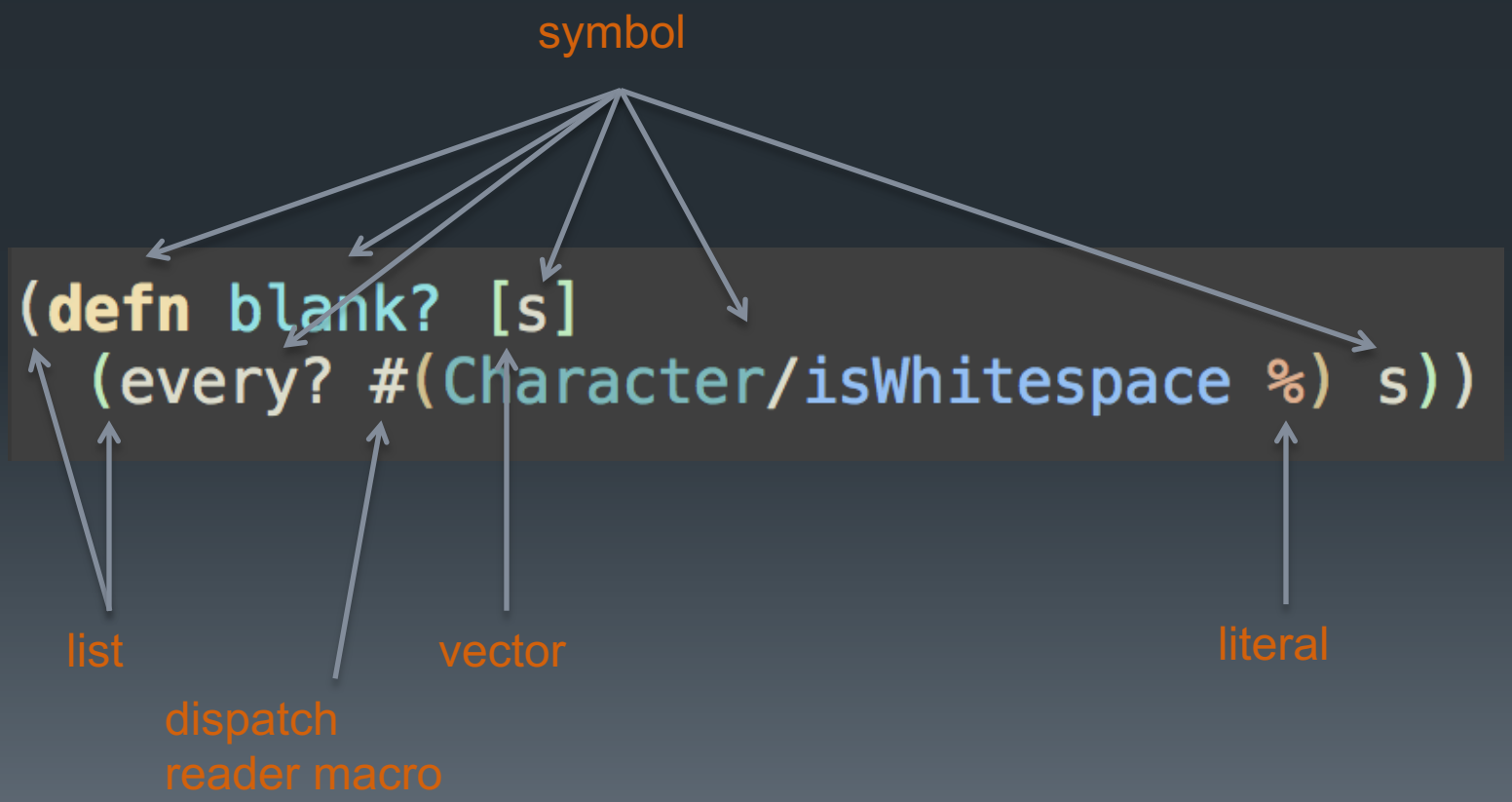
Syntax

```
(defn blank? [s]
  (every? #(Character/isWhitespace %) s))
```

Syntax



Syntax


symbol
(defn blank? [s]
(every? #(Character/isWhitespace %) s))
list
dispatch
reader macro
vector
literal

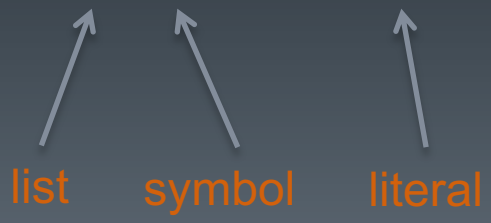
Syntax

```
(defn blank? [s]
  (every? #(Character/isWhitespace %) s))
```

Syntax

```
(defn blank? [s]
  (every? #(Character/isWhitespace %) s))
```

```
(blank? "Adam") => false
```



Syntax

```
(defn blank? [s]
  (every? #(Character/isWhitespace %) s))
```

```
(blank? "Adam") => false
(blank? "    ") => true
```

Syntax

```
(defn blank? [s]
  (every? #(Character/isWhitespace %) s))
```

```
(blank? "Adam") => false
(blank? "    ") => true
(blank? "") => true
```

Syntax

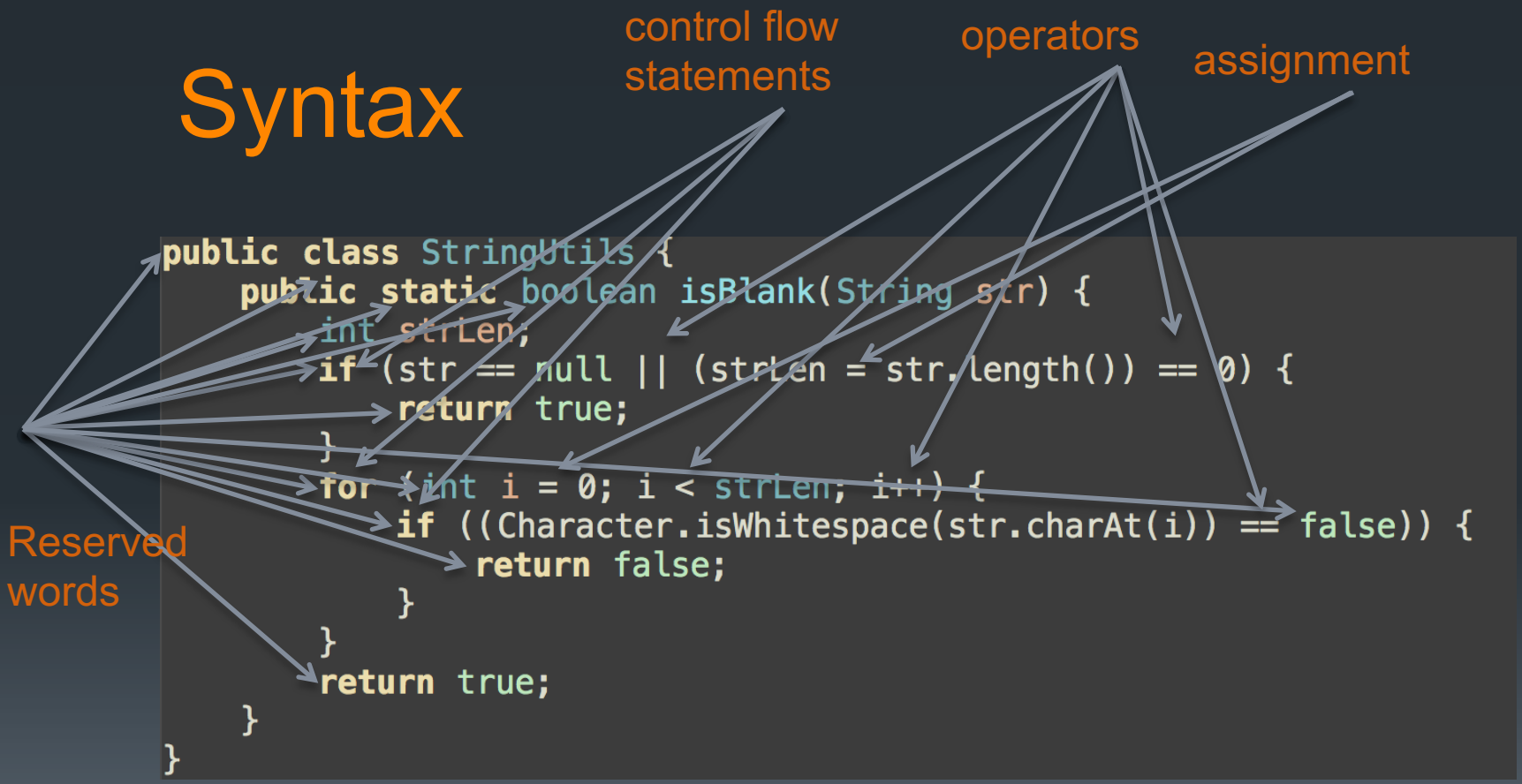
```
(defn blank? [s]
  (every? #(Character/isWhitespace %) s))
```

```
(blank? "Adam") => false
(blank? "    ") => true
(blank? "") => true
(blank? [\space \space \space]) => true
```

Syntax

```
public class StringUtils {
    public static boolean isBlank(String str) {
        int strLen;
        if (str == null || (strLen = str.length()) == 0) {
            return true;
        }
        for (int i = 0; i < strLen; i++) {
            if ((Character.isWhitespace(str.charAt(i)) == false)) {
                return false;
            }
        }
        return true;
    }
}
```

Syntax


control flow statements
operators
assignment
Reserved words
public class StringUtils {
public static boolean isBlank(String str) {
int strLen;
if (str == null || (strLen = str.length()) == 0) {
return true;
}
for (int i = 0; i < strLen; i++) {
if ((Character.isWhitespace(str.charAt(i)) == false)) {
return false;
}
}
return true;
}
}

Syntax

- Simple
  - few structures that always do the same thing
- Powerful
  - begets semantics

Reader

- Forms: Symbols, Literals, Lists, Vectors, Maps, Sets
- Macros: ‘ \ ; @ ^ # ` ~

Evaluation

- Compiler – *no interpreter*
  - Special Forms
  - Macros
  - Everything else: expression -> value
- Literal
- Symbol
- Vector, Maps, Sets
- List: call to special form, macros, functions

Symbol

- begin with a non-numeric character and can contain alphanumeric characters and *, +, !, -, _, ', and ?
  - eg: `name, my-function, isPrime?`
- resolved
  - special form
  - Java Class
  - local binding
  - namespace binding
  - error

Functional

- Pure Functions
- Persistent Data Structures
- … my precious
- Recursion

Pure Functions

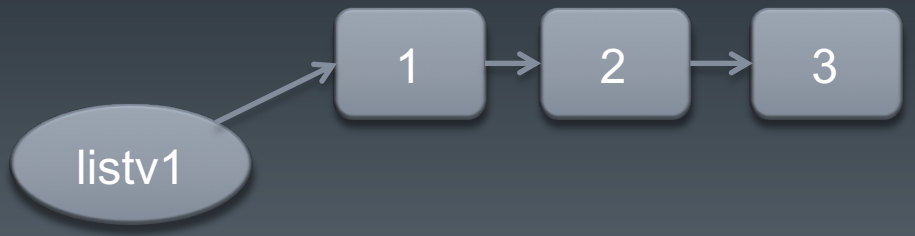
```
(defn blank? [s]
  (every? #(Character/isWhitespace %) s))
```

Persistent Data Structures

- [], {}, #{}, ()
- behave as though a complete copy is made each time they're modified

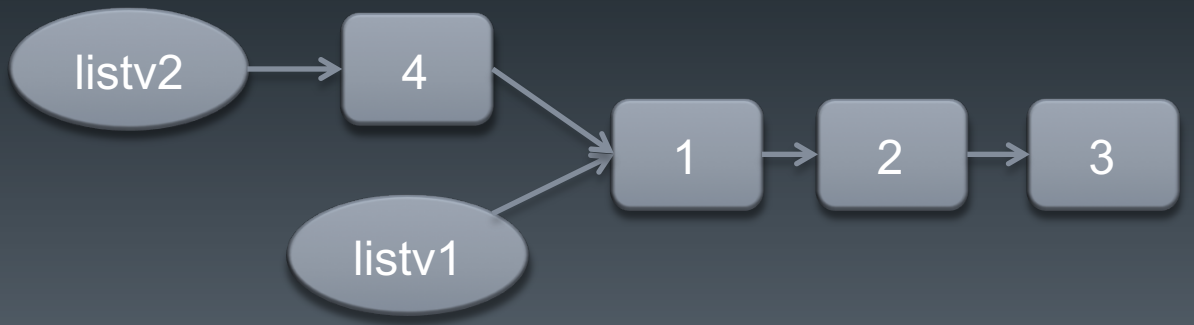
Persistent Data Structures

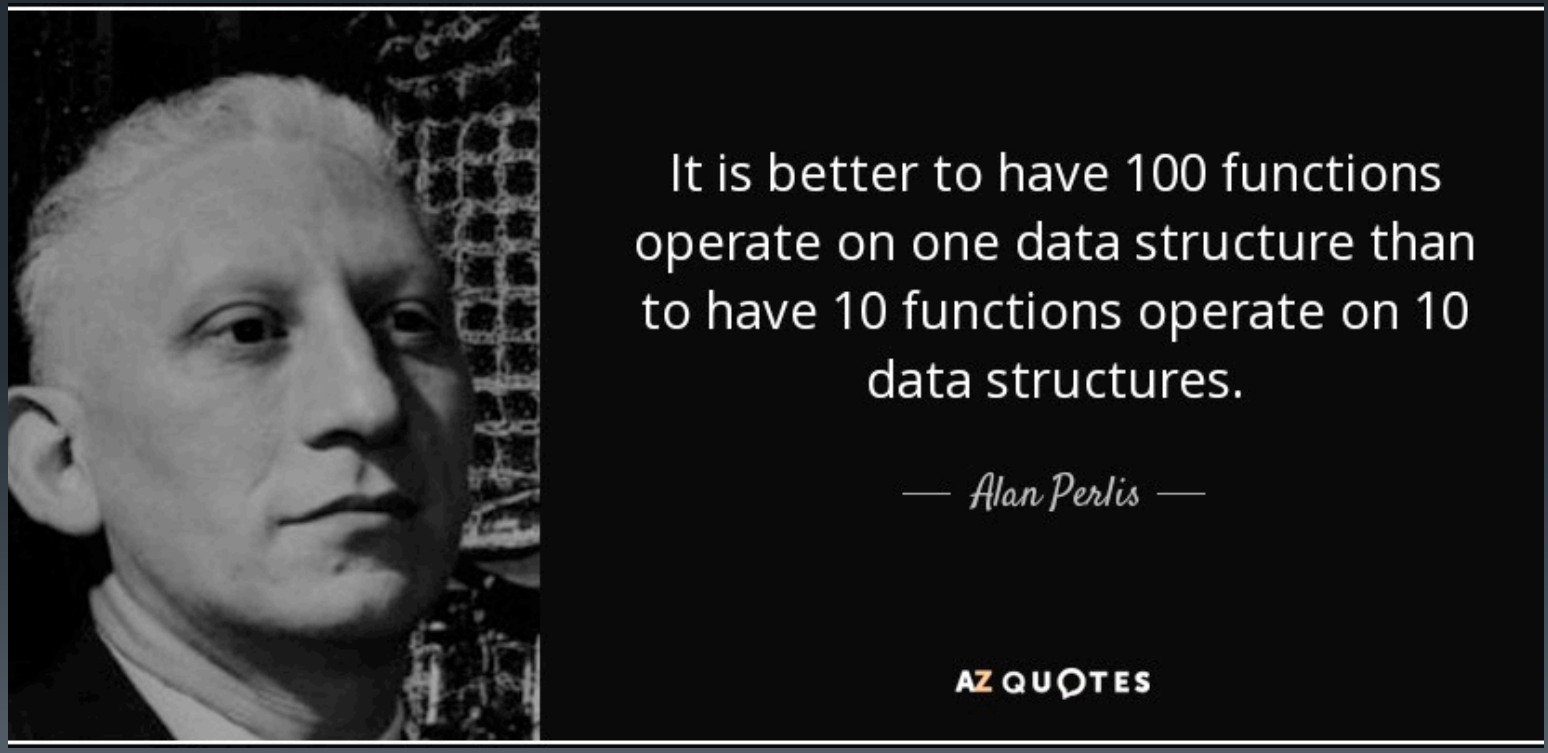
```
user> (def listv1 '(1 2 3))
#'user/listv1
user> listv1
(1 2 3)
```



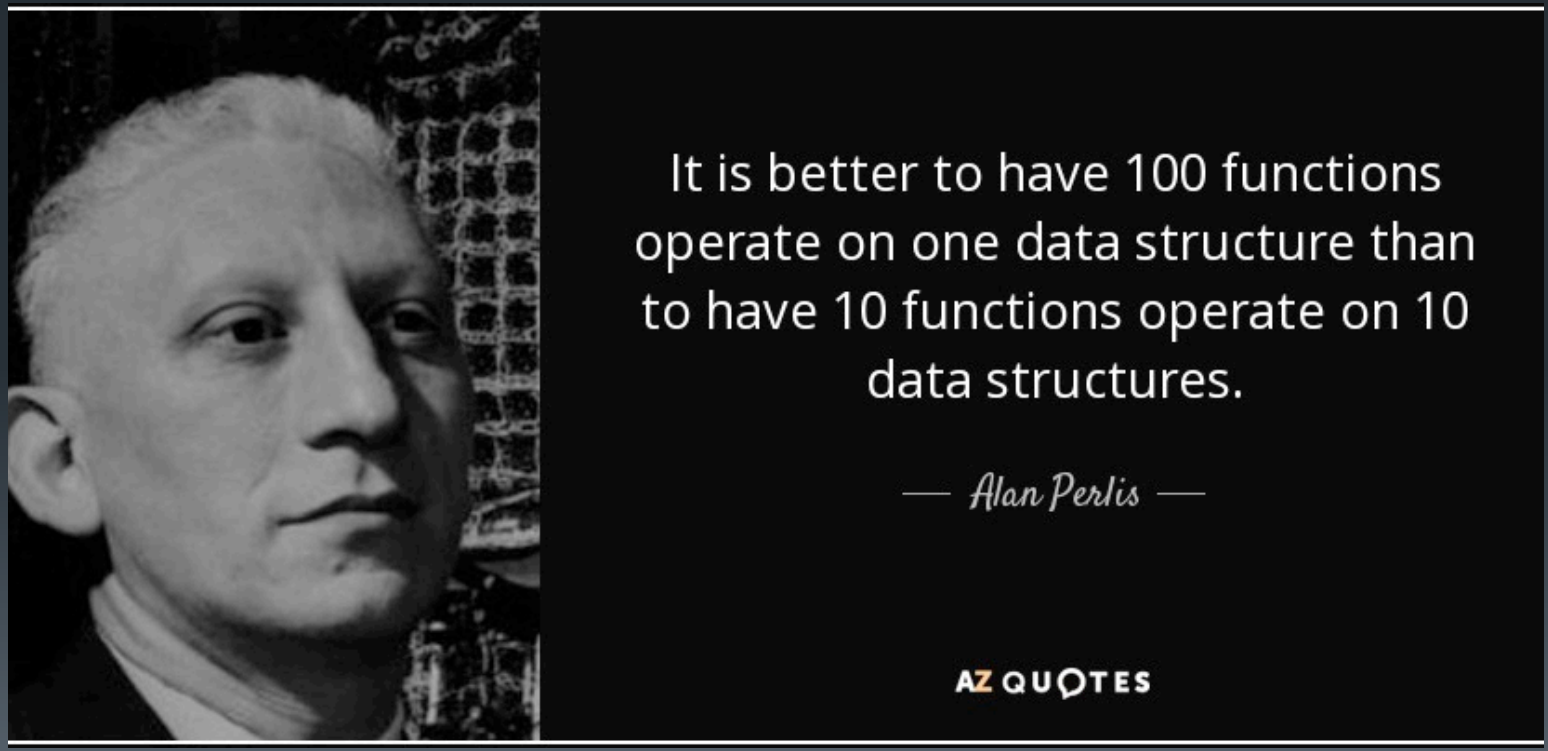
Persistent Data Structures

```
user> (def listv1 '(1 2 3))
#'user/listv1
user> listv1
(1 2 3)
user> (def listv2 (cons 4 listv1))
#'user/listv2
user> listv2
(4 1 2 3)
```




It is better to have 100 functions operate on one data structure than to have 10 functions operate on 10 data structures.
— Alan Perlis —
AZ QUOTES

Sequence


It is better to have 100 functions operate on one data structure than to have 10 functions operate on 10 data structures.
— Alan Perlis —
AZ QUOTES

Sequence

```
(first [1 2 3])=> 1
(rest [1 2 3])=> (2 3)
(cons 0 [1 2 3])=> (0 1 2 3)
```

Sequence

- All Clojure collections
- All Java collections
- Java arrays and strings
- Regular expression matches
- Directory structures
- I/O streams
- XML trees
- …

Sequences

Creating a Lazy Seq

From collection	`seq vals keys rseq subseq rsubseq`
From producer fn	`lazy-seq repeatedly iterate`
From constant	`repeat range`
From other	`file-seq line-seq resultset-seq re-seq tree-seq xml-seq iterator-seq enumeration-seq`
From seq	`keep keep-indexed`

Seq in, Seq out

Get shorter	`distinct filter remove take-nth for`
Get longer	`cons conj concat lazy-cat mapcat cycle interleave interpose`
Tail-items	`rest nthrest next fnext nnext drop drop-while take-last for`
Head-items	`take take-while butlast drop-last for`
'Change'	`conj concat distinct flatten group-by partition partition-all partition-by split-at split-with filter remove replace shuffle`
Rearrange	`reverse sort sort-by compare`
Process items	`map pmap map-indexed mapcat for replace seque`

Using a Seq

Extract item	`first second last rest next ffirst nfirst fnext nnext nth nthnext rand-nth when-first max-key min-key`
Construct coll	`zipmap into reduce reductions set vec into-array to-array-2d (1.4) mapv filterv`
Pass to fn	`apply`
Search	`some filter`
Force evaluation	`doseq dorun doall`
Check for forced	`realized?`

Recursion

```
(defn cnt [xs]
  (loop [ys  xs
         acc 0]
    (if (first ys)
      (recur (rest ys) (inc acc))
      acc)))
```

Laziness!

- Laziness: elements calculated as needed

- So what?
  - You can postpone expensive computations that may not in fact be needed.
  - You can work with huge data sets that do not fit into memory.
  - You can delay I/O until it is absolutely needed.

Functional

- Pure Functions
- Persistent Data Structures

Functional

- Pure Functions *mostly*
- Immutable Data Structures *mostly*

Pure Functions *mostly*

```
(defn blank? [s]
  (every? #(Character/isWhitespace %) s))
```

Pure Functions *mostly*

```
(defn blank? [s]
  (every? #(do (println "Checking [" % "]: ")
               (Character/isWhitespace %))    s))
```

Immutable *mostly*

```
user> (def v [1 2 3])
#'user/v
user> (conj v 4)
[1 2 3 4]
user> v
[1 2 3]
```

Immutable *mostly*

```
user> (def tv (transient [1 2 3]))
#'user/tv
user> (conj! tv 4)
#object[clojure.lang.PersistentVector$TransientVector
user> (persistent! tv)
[1 2 3 4]
```

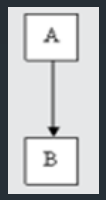
Functional

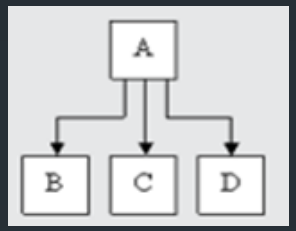
- Simple
  - Functional Programming
  - Immutable Data Structures
  - Sequence
  - Helpful Recursion Constructs
- Powerful
  - Sequence
  - Laziness
  - *mostly*

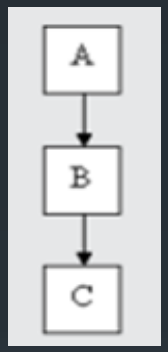
Polymorphism

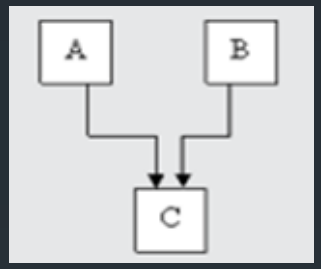
Polymorphism

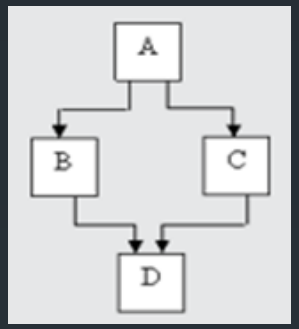
Polymorphism *inheritance*





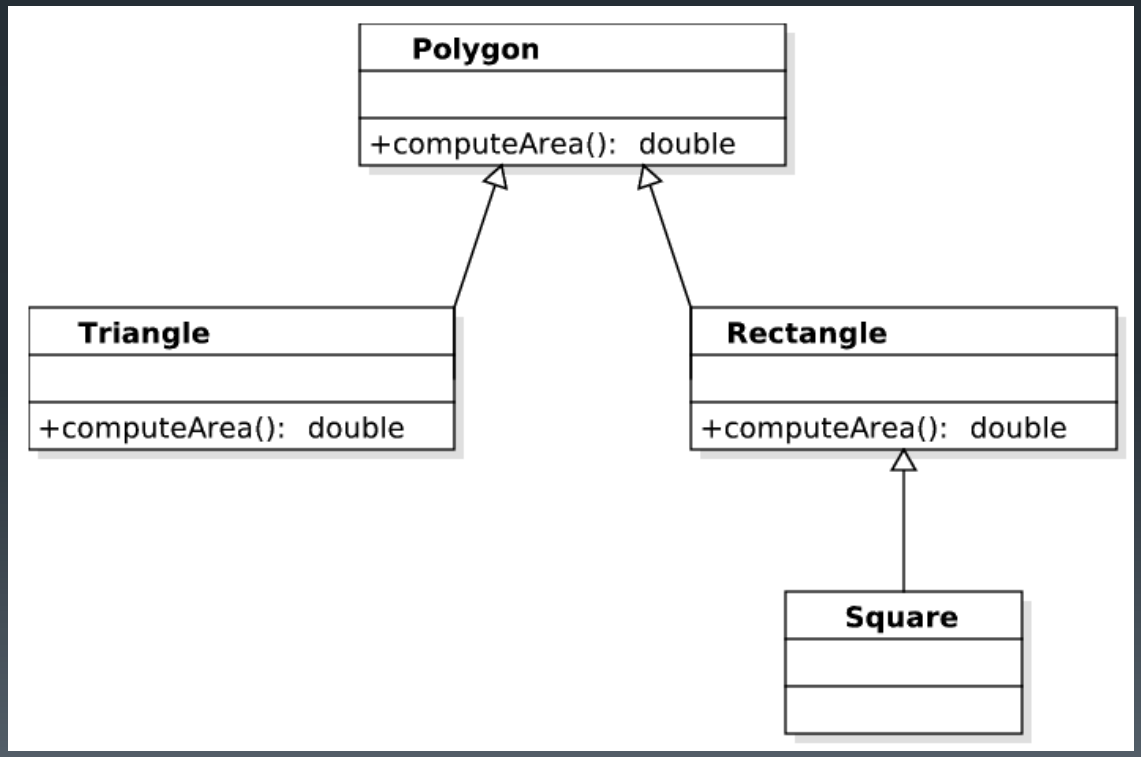






http://eduladder.com/viewnotes.php?notesid=49

Polymorphism *inheritance*



Polymorphism *inheritance*

The biggest point of confusion and contention seems to be composition versus inheritance, often summarized in the mantra "favor composition over inheritance". Let's talk about that.

Inheritance should only be used when:

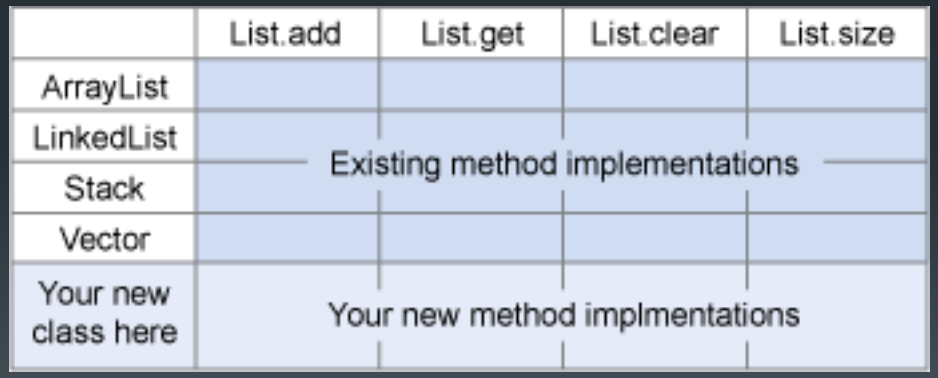
1. Both classes are in the same logical domain
2. The subclass is a proper subtype of the superclass
3. The superclass's implementation is necessary or appropriate for the subclass
4. The enhancements made by the subclass are primarily additive.

There are times when all of these things converge:

- Higher-level domain modeling
- Frameworks and framework extensions
- Differential programming

https://www.thoughtworks.com/insights/blog/composition-vs-inheritance-how-choose

Polymorphism *expression*



http://www.ibm.com/developerworks/library/j-clojure-protocols/

Polymorphism *expression*

	conj	nth	empty	count	
list					Your new function here
vector		Existing implementations			
map					
set					

http://www.ibm.com/developerworks/library/j-clojure-protocols/

Protocols and Records

- Behavior

```
(defprotocol Areable
  (area [this]))
```

- Type

```
(defrecord Square [side])
```

Protocols and Records

- Behavior

```
(defprotocol Areable
  (area [this]))
```

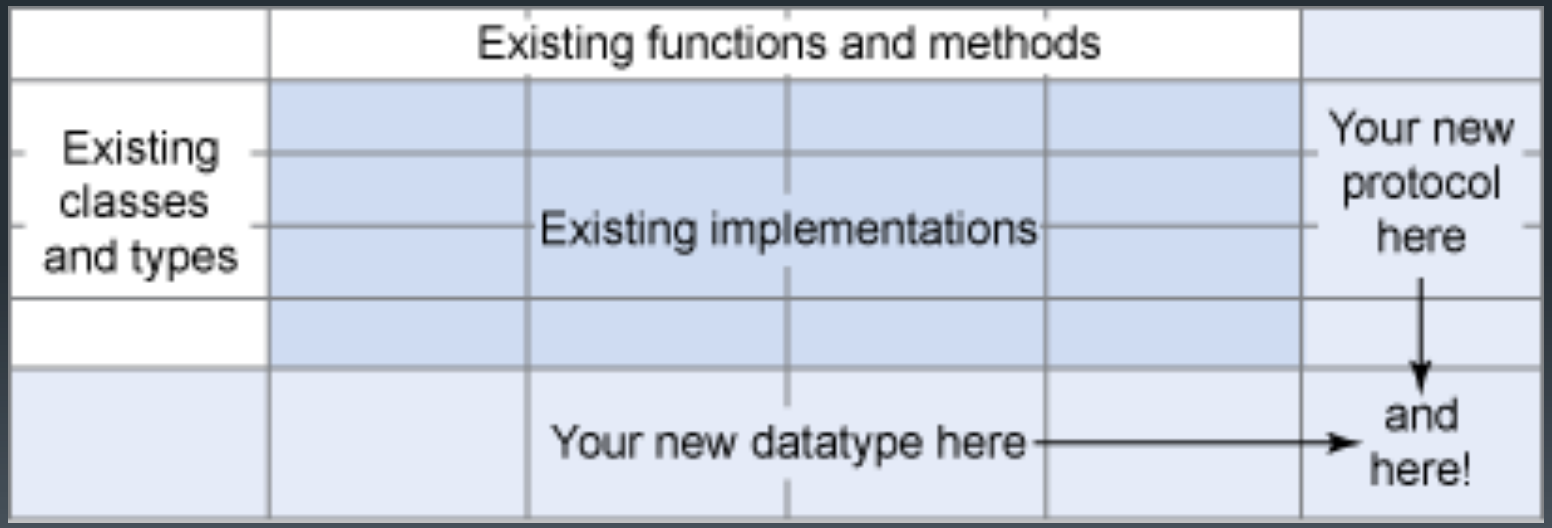
- Type

```
(defrecord Square [side])
```

- Polymorphism *a la carte*

```
(extend Square
  Areable
  {:area (fn [this] (* (:side this) (:side this)))})
```

Protocols and Records



http://www.ibm.com/developerworks/library/j-clojure-protocols/

Polymorphism

- Simple
  - Detangled from type
- Powerful
  - Extend existing types without Monkey Patching

Concurrency vs Parallelism

- Concurrency:
  - *composition of independently executing processes*
  - *dealing with lots of things happening at once*
  - *structure*
- Parallelism:
  - *more than 1 thing happening at the same time*
  - *doing lots of things at once*
  - *execution*

Concurrency … so far

- Pure Functions
  - Decouple execution order from desired result
- Immutable Data
  - Can be shared without worrying

Concurrency

- Concurrency is a way to structure a program by breaking it into pieces that can execute independently
- Concurrency requires communication to coordinate independent executions
- Up to now everything has been immutable
- Well … you can communicate immutable things without worrying so the model has been inherently concurrent
- But … let's model some things that ***change***

http://www.infoq.com/presentations/An-Introduction-to-Clojure-Time-Model

Concurrency … the beginning

- Future:
  - blocking cached computation on separate thread
  - start immediately
- Promise:
  - blocking cached computation
  - computation thread must deliver

Concurrency … the beginning

- Future:

```
user> (def f (future (+ 1 2)))
#'user/f
user> (time @f)
"Elapsed time: 0.022891 msecs"
3
user> (def long-f (future (do (Thread/sleep 10000) (+ 1 2))))
#'user/long-f
user> (time @long-f)
"Elapsed time: 6114.87105 msecs"
3
```

Concurrency … the beginning

- Promise:

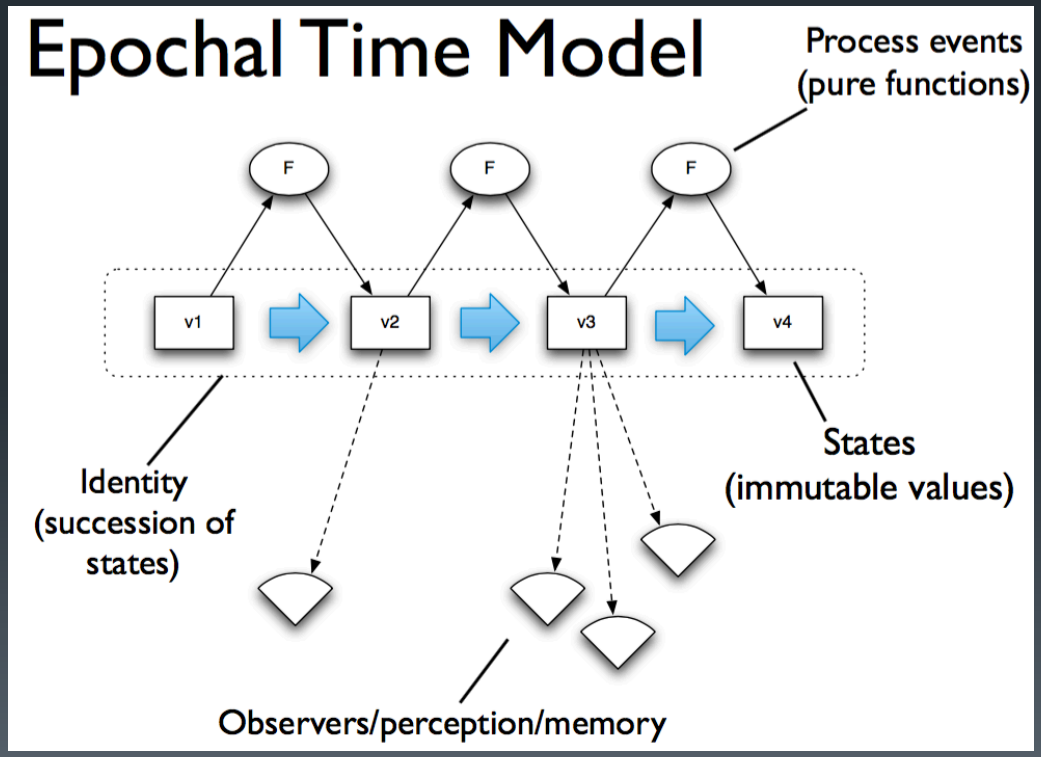
```
user> (def p (promise))
#'user/p
user> (future (do (Thread/sleep 10000) (deliver p 10)))
#object[clojure.core$future_call$reify__6736 0x3cc98100 {:status :pending, :val nil}]
user> (time @p)
"Elapsed time: 6757.060297 msecs"
10
```

Model of Time

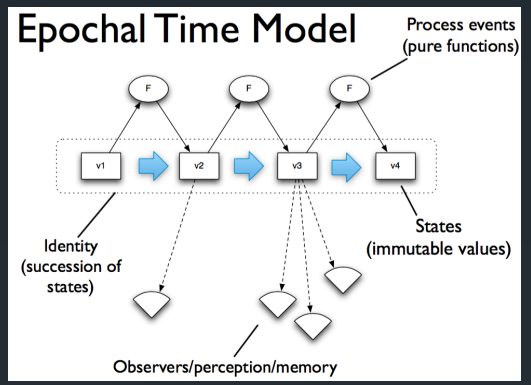
- Value: is immutable and semantically transparent, a fact
- Time: concept that we overlay on causal events
- Identity: concept we lay on a series of *values* over *time*
- State: the *value* of an *identity* at a point in *time*

Key Insight: *Separate* *Identity* and *State*

Model of Time

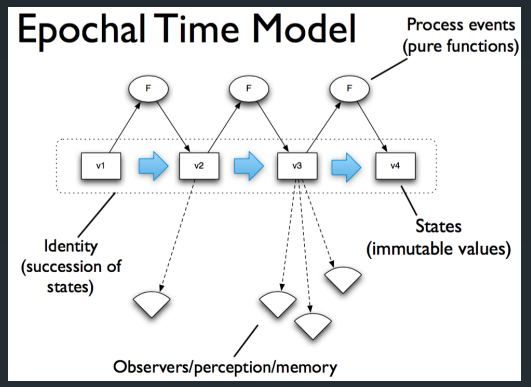


http://www.infoq.com/presentations/An-Introduction-to-Clojure-Time-Model



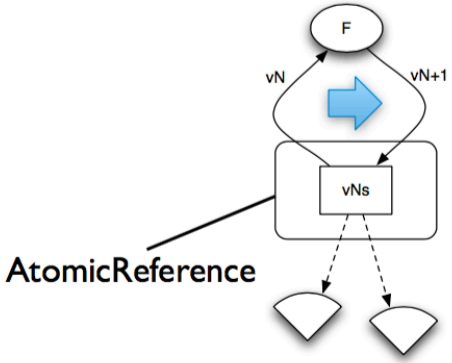
Identity Types

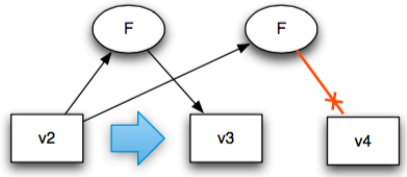
- Atoms enable independent, synchronous changes to single values
- Agents enable independent, asynchronous changes to single values
- Refs enable coordinated, synchronous changes to multiple values



Identity Types *atom*

cas as time construct



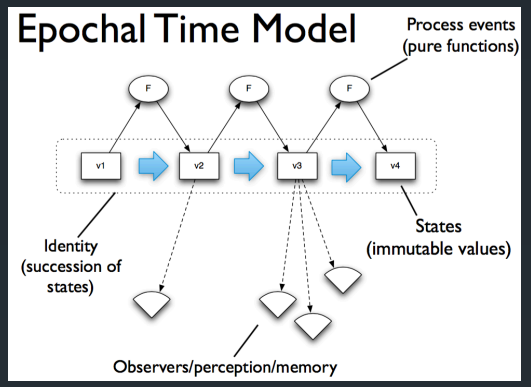


`(swap! an-atom f args)`

`(f vN args)` *becomes* vN+1

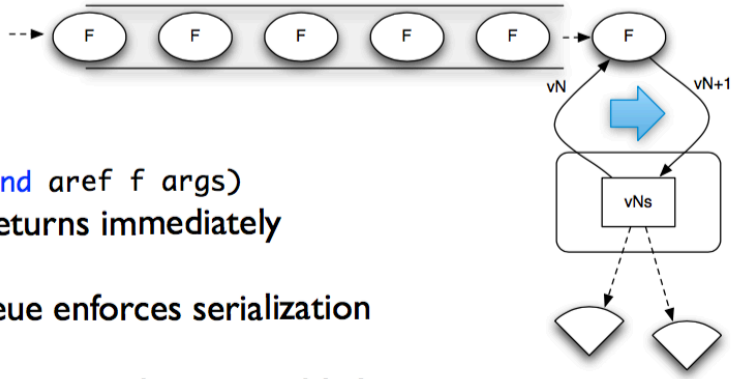
- can automate spin

- 1:1 timeline/identity
- Atomic state succession
- Point-in-time value perception



Identity Types *agent*

agents as time construct



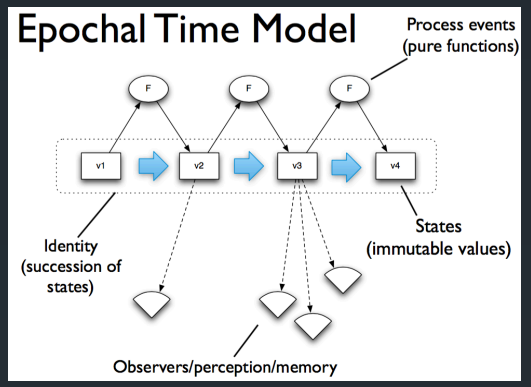

```
(send aref f args)
```
returns immediately

queue enforces serialization

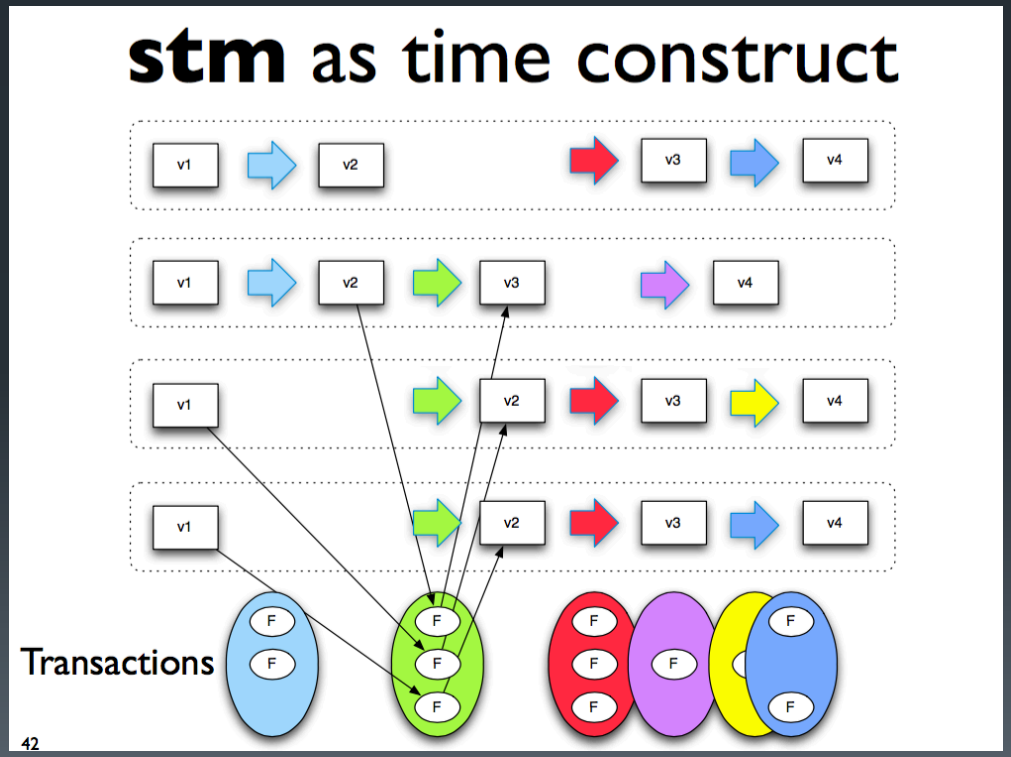
`(f vN args)` becomes vN+1

happens asynchronously in thread pool thread

- 1:1 timeline/identity
- Atomic state succession
- Point-in-time value perception



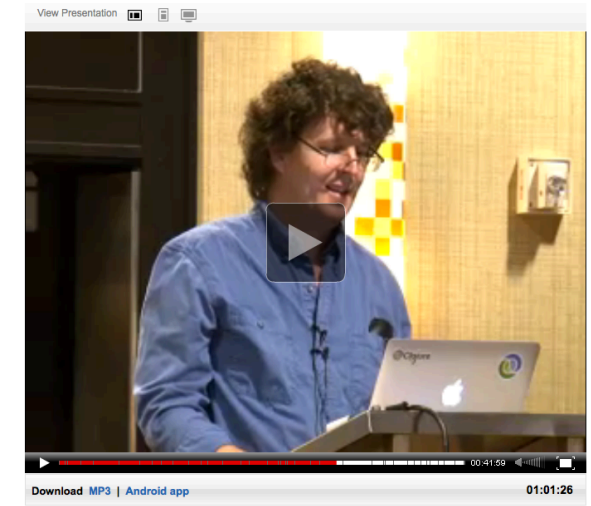
Identity Types *ref*



Concurrency

- Simple
  - Functional Programming Helps
    - Pure Functions and Immutable Data
  - Good primitives
  - Intuitive Model
- Powerful
  - Controlled Mutation

Complexity

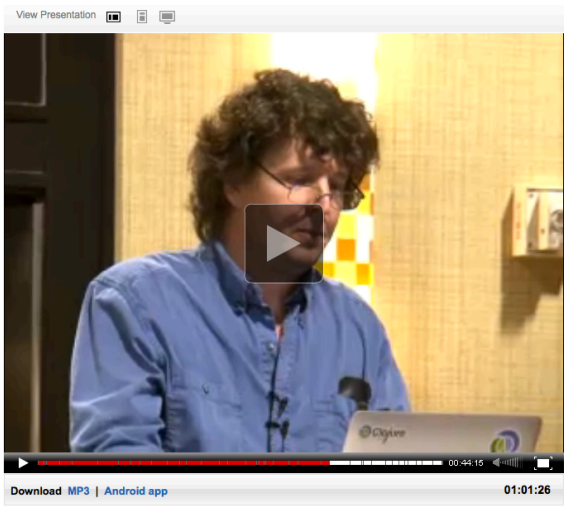


The Complexity Toolkit

Construct	Complects
State	Everything that touches it
Objects	State, identity, value
Methods	Function and state, namespaces
Syntax	Meaning, order
Inheritance	Types
Switch/matching	Multiple who/what pairs
var(iable)s	Value, time
Imperative loops, fold	what/how
Actors	what/who
ORM	OMG
Conditionals	Why, rest of program

http://www.infoq.com/presentations/Simple-Made-Easy

Simplicity



The Simplicity Toolkit

Construct	Get it via...
Values	final, persistent collections
Functions	a.k.a. stateless methods
Namespaces	language support
Data	Maps, arrays, sets, XML, JSON etc
Polymorphism a la carte	Protocols, type classes
Managed refs	Clojure/Haskell refs
Set functions	Libraries
Queues	Libraries
Declarative data manipulation	SQL/LINQ/Datalog
Rules	Libraries, Prolog
Consistency	Transactions, values

http://www.infoq.com/presentations/Simple-Made-Easy

References

- http://www.infoq.com/presentations/Simple-Made-Easy
- https://mitpress.mit.edu/sicp/full-text/book/book.html
- http://www.infoq.com/presentations/Value-Values
- https://pragprog.com/book/pb7con/seven-concurrency-models-in-seven-weeks
- http://www.joyofclojure.com/
- https://pragprog.com/book/shcloj2/programming-clojure