Clojure: Enemy of the State*

* Not actually an enemy of the state, or state in general. :)

Alex Miller @puredanger

Roadmap

- **Values vs objects**
- Collections
- Sequences
- Generic data interfaces
- Identity and state

What is a "value"?

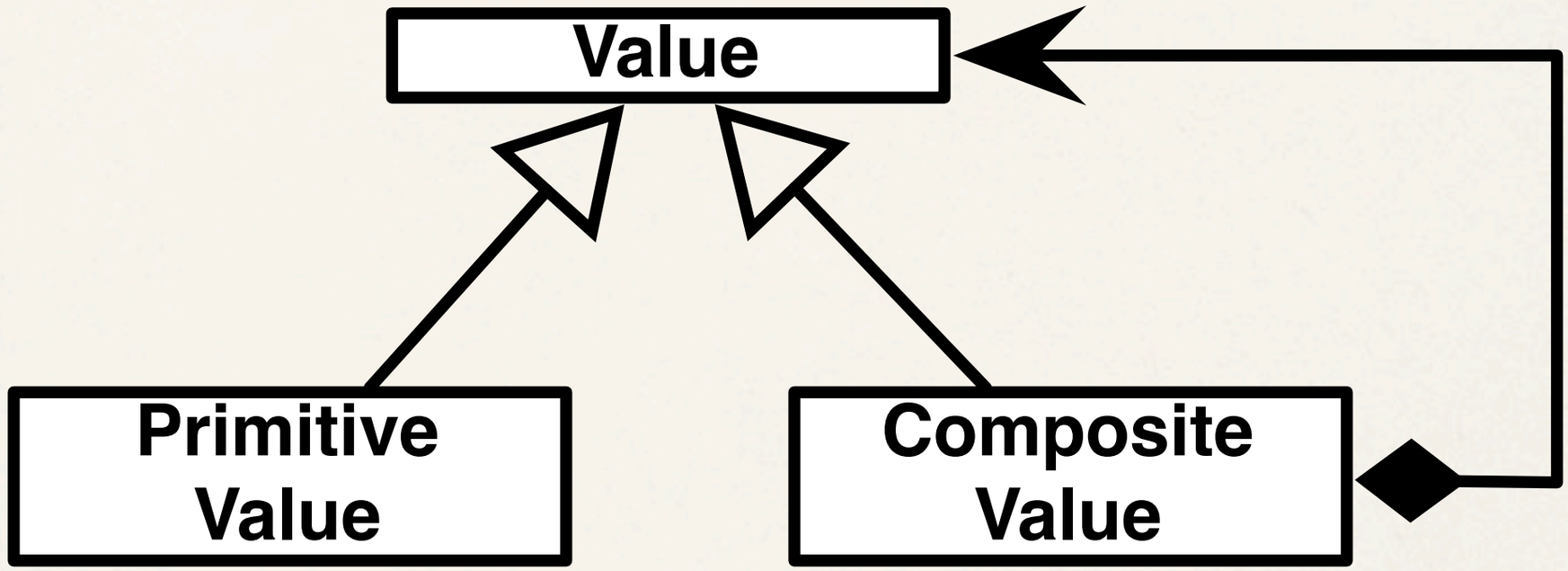
Precise meaning or significance.

Thing or quality having intrinsic worth.

A particular magnitude, number, or amount.

Examples	Properties
20	precise meaning
6.2	immutable
false, true	comparable for equality
a	semantically evident
"abc"	

What about composite values?



Are objects composite values?

An object is a first-class, dynamically dispatched behavior.

First class means that objects **have the same capabilities as other kinds of values**, including being passed to operations or returned as the result of an operation.

Objects can be used anywhere that values can be used.

Selected quotes from William Cook's modern definition of OO:
http://wcook.blogspot.com/2012/07/proposal-for-simplified-modern.html

Have you ever …

- struggled with how to define equality for an object?
- needed to defensively clone or copy an object?
- had issues with multithreaded access to objects?
- had trouble combining thread-safe objects?
- had trouble serializing objects?
- had trouble caching objects?
- had trouble with queuing objects?

Mutable objects ≠ Composite values


CHANGE
AHEAD

Roadmap

- Values vs objects
- **Collections**
- Sequences
- Generic data interfaces
- Identity and state

Clojure

- A Lisp dialect
- Runs on JVM (also: ClojureScript, ClojureCLR)
- Dynamically typed
- Compiled (no interpreter)
- Functional programming

Primitives

- `10`
- `20.3`
- `22/7`
- `false, true`
- `nil`
- `\a`
- `"abc"`

Clojure collections

Vectors

```
[1 2 3]
```

Maps

```
{:first-name "John",
 :last-name "McCarthy"}
```

Lists

```
(1 2 3)
```

Sets

```
#{"larry" "curly" "moe"}
```

Clojure collections — Composite values

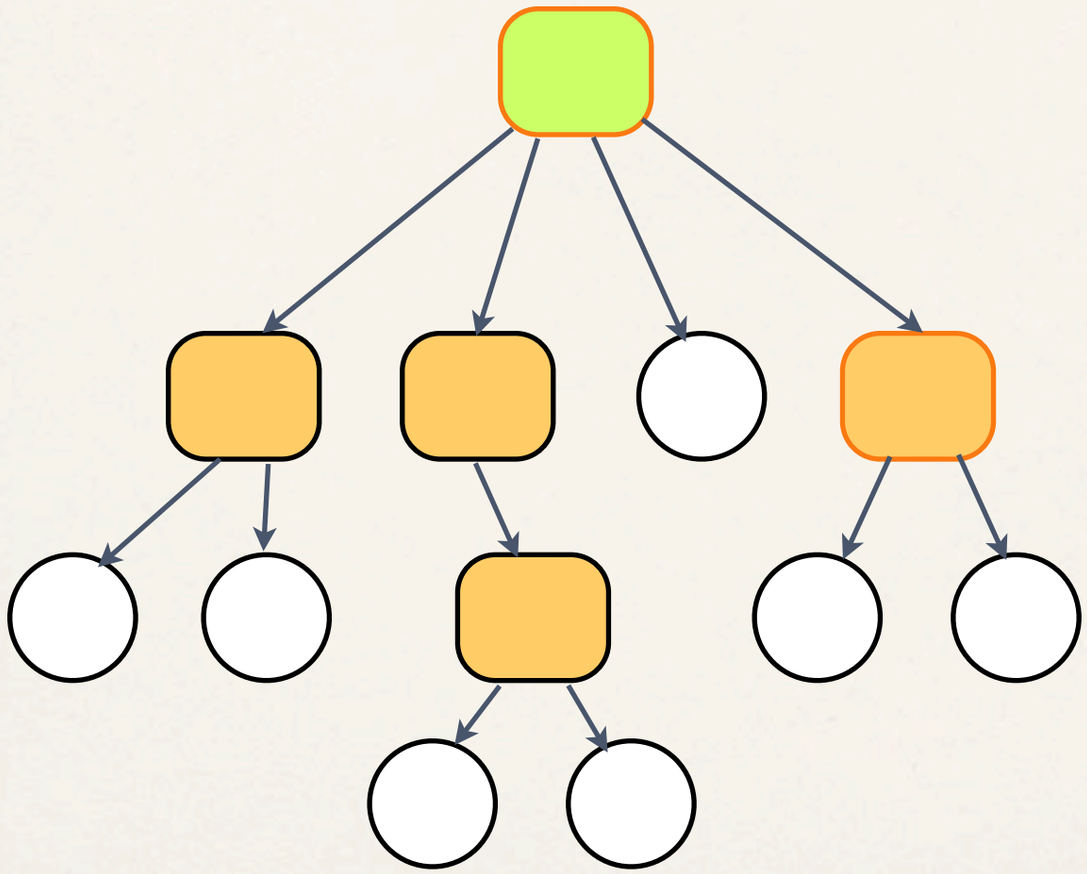
Collection functions

- **All** - `count, conj, seq, into, empty`
- **List** - `list?, first, rest, nth, list, cons, peek, pop`
- **Vector** - `vector?, vector, vec, nth, subvec, replace`
- **Map** - `assoc, dissoc, get, select-keys, contains?, merge, keys, vals, find`
- **Set** - `set?, set, hash-set, disj, contains?, (clojure.set/ join, select, project, union, difference, intersection)`

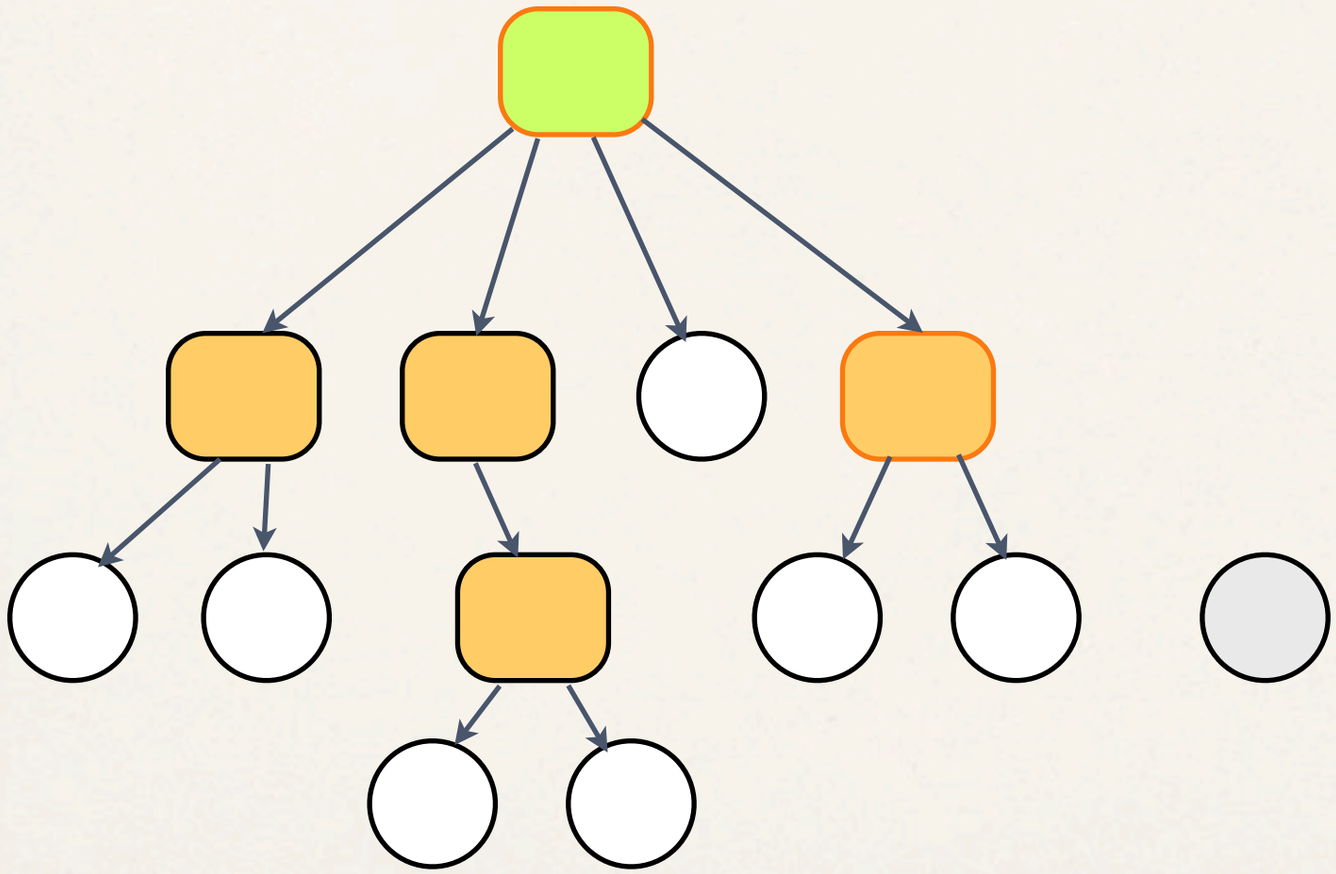
To the demo...

Isn't this all horribly inefficient?

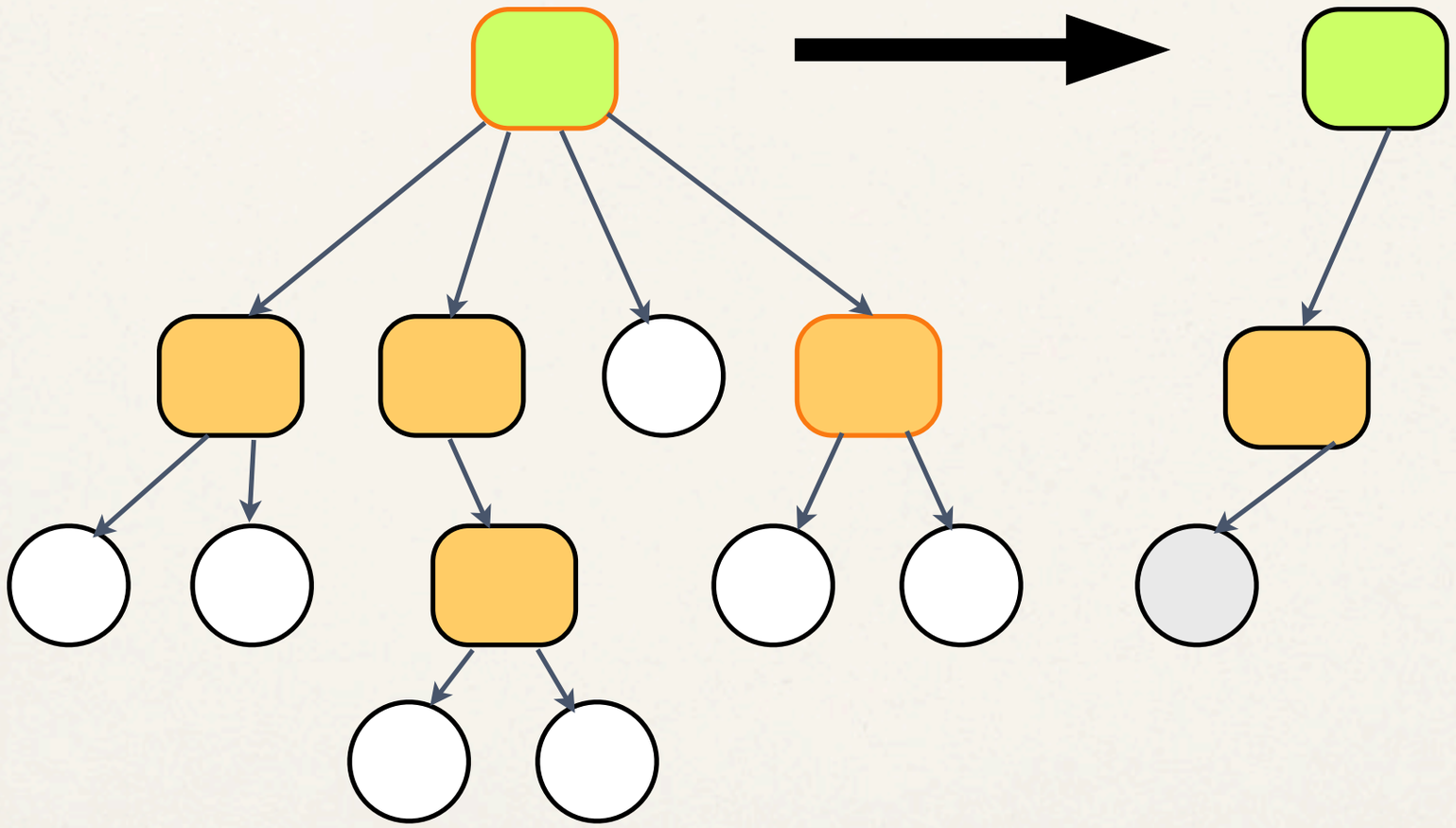
Isn't this all horribly inefficient?



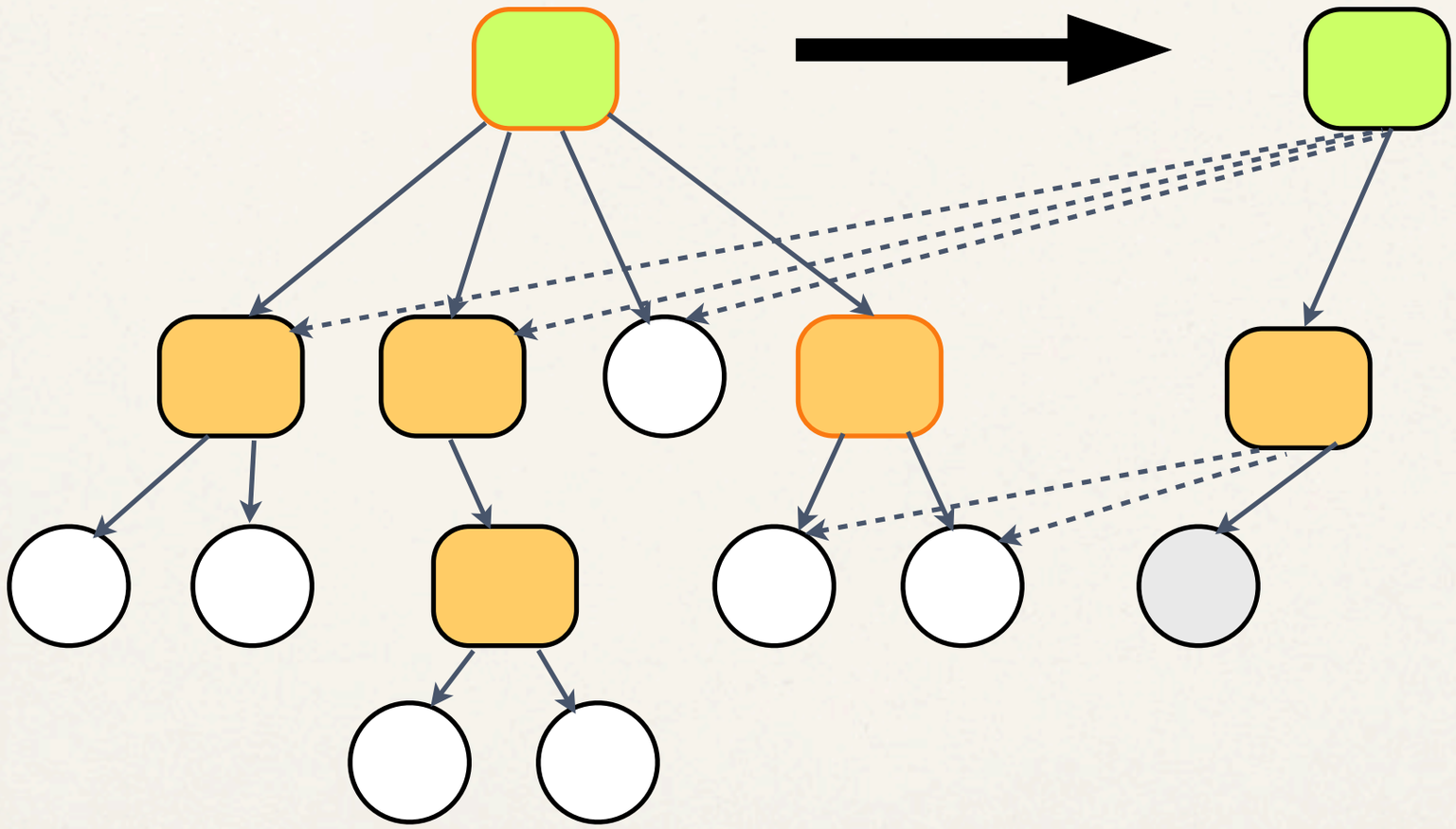
Isn't this all horribly inefficient?



Isn't this all horribly inefficient?



Isn't this all horribly inefficient?



Roadmap

- Values vs objects
- Collections
- **Sequences**
- Generic data interfaces
- Identity and state

Sequences

- Immutable view into a collection (not a stateful cursor)
- **Also** immutable composite values (logically a list)
- `first, rest` - the first and rest of the logical list
- `seq` - returns either a sequence or nil (indicating no more)
- Often lazy (maybe infinite!) but you can usually consider this an implementation detail

What things produce sequences?

- All Clojure collections - `list, set, vector, map`
- `strings` - sequence of characters
- `arrays` - sequence of values
- `file-seq` - files in a directory
- `line-seq` - lines in a file
- `resultset-seq` - `ResultSet` rows from a database
- `xml-seq` - tags in an XML doc
- `tree-seq` - a tree of collections
- `re-seq` - regular expression matches
- `iterator-seq` - Java Iterators
- `enumeration-seq` - Java Enumerations

What can you do with them?

- distinct, filter, remove, for, keep, keep-indexed
- cons, concat, lazy-cat, mapcat, cycle, interleave, interpose
- rest, next, fnext, nnext, drop, drop-while, nthnext, for
- flatten, reverse, sort, sort-by, shuffle
- split-at, split-with, partition, partition-all, partition-by
- map, pmap, mapcat, replace, reductions, map-indexed, seque
- first, ffirst, nfirst, second, nth, when-first, last, rand-nth
- zipmap, into, set, vec, into-array, to-array, to-array-2d, frequencies, group-by, apply
- not-empty, some, seq?, every?, not-every?, not-any?, empty?
- doseq, dorun, doall, realized?
- vals, keys, rseq, subseq, rsubseq, lazy-seq, repeatedly, iterate
- repeat, range

The Grand Abstraction

- All Clojure collections
- `strings` - sequence of characters
- `arrays` - sequence of values
- `file-seq` - files in a directory
- `line-seq` - lines in a file
- `resultset-seq` - `ResultSet` rows from a DB
- `xml-seq` - tags in an XML doc
- `tree-seq` - a tree of collections
- `re-seq` - regular expression matches
- `iterator-seq` - Java Iterators
- `enumeration-seq` - Java Enumerations

The Grand Abstraction

Data

- All Clojure collections
- `strings` - sequence of characters
- `arrays` - sequence of values
- `file-seq` - files in a directory
- `line-seq` - lines in a file
- `resultset-seq` - `ResultSet` rows from a DB
- `xml-seq` - tags in an XML doc
- `tree-seq` - a tree of collections
- `re-seq` - regular expression matches
- `iterator-seq` - Java Iterators
- `enumeration-seq` - Java Enumerations

The Grand Abstraction

Data

- All Clojure collections
- `strings` - sequence of characters
- `arrays` - sequence of values
- `file-seq` - files in a directory
- `line-seq` - lines in a file
- `resultset-seq` - `ResultSet` rows from a DB
- `xml-seq` - tags in an XML doc
- `tree-seq` - a tree of collections
- `re-seq` - regular expression matches
- `iterator-seq` - Java Iterators
- `enumeration-seq` - Java Enumerations

- `distinct, filter, remove, for, keep, keep-indexed`
- `cons, concat, lazy-cat, mapcat, cycle, interleave, interpose`
- `rest, next, fnext, nnext, drop, drop-while, nthnext, for`
- `flatten, reverse, sort, sort-by, shuffle`
- `split-at, split-with, partition, partition-all, partition-by`
- `map, pmap, mapcat, replace, reductions, map-indexed, seque`
- `first, ffirst, nfirst, second, nth, when-first, last, rand-nth`
- `zipmap, into, set, vec, into-array, to-array, to-array-2d, frequencies, group-by, apply`
- `not-empty, some, seq?, every?, not-every?, not-any?, empty?`
- `doseq, dorun, doall, realized?`
- `vals, keys, rseq, subseq, rsubseq, lazy-seq, repeatedly, iterate`
- `repeat, range`

The Grand Abstraction

Data

- All Clojure collections
- `strings` - sequence of characters
- `arrays` - sequence of values
- `file-seq` - files in a directory
- `line-seq` - lines in a file
- `resultset-seq` - `ResultSet` rows from a DB
- `xml-seq` - tags in an XML doc
- `tree-seq` - a tree of collections
- `re-seq` - regular expression matches
- `iterator-seq` - Java Iterators
- `enumeration-seq` - Java Enumerations

Code

- `distinct, filter, remove, for, keep, keep-indexed`
- `cons, concat, lazy-cat, mapcat, cycle, interleave, interpose`
- `rest, next, fnext, nnext, drop, drop-while, nthnext, for`
- `flatten, reverse, sort, sort-by, shuffle`
- `split-at, split-with, partition, partition-all, partition-by`
- `map, pmap, mapcat, replace, reductions, map-indexed, seque`
- `first, ffirst, nfirst, second, nth, when-first, last, rand-nth`
- `zipmap, into, set, vec, into-array, to-array, to-array-2d, frequencies, group-by, apply`
- `not-empty, some, seq?, every?, not-every?, not-any?, empty?`
- `doseq, dorun, doall, realized?`
- `vals, keys, rseq, subseq, rsubseq, lazy-seq, repeatedly, iterate`
- `repeat, range`

The Grand Abstraction

Data

- All Clojure collections
- `strings` - sequence of characters
- `arrays` - sequence of values
- `file-seq` - files in a directory
- `line-seq` - lines in a file
- `resultset-seq` - `ResultSet` rows from a DB
- `xml-seq` - tags in an XML doc
- `tree-seq` - a tree of collections
- `re-seq` - regular expression matches
- `iterator-seq` - Java Iterators
- `enumeration-seq` - Java Enumerations

Sequence

Code

- `distinct, filter, remove, for, keep, keep-indexed`
- `cons, concat, lazy-cat, mapcat, cycle, interleave, interpose`
- `rest, next, fnext, nnext, drop, drop-while, nthnext, for`
- `flatten, reverse, sort, sort-by, shuffle`
- `split-at, split-with, partition, partition-all, partition-by`
- `map, pmap, mapcat, replace, reductions, map-indexed, seque`
- `first, ffirst, nfirst, second, nth, when-first, last, rand-nth`
- `zipmap, into, set, vec, into-array, to-array, to-array-2d, frequencies, group-by, apply`
- `not-empty, some, seq?, every?, not-every?, not-any?, empty?`
- `doseq, dorun, doall, realized?`
- `vals, keys, rseq, subseq, rsubseq, lazy-seq, repeatedly, iterate`
- `repeat, range`

Line counts

```
(defn line-count [file]
  (count (line-seq (reader file))))

(defn file? [file]
  (.isFile file))

(defn file-counts [dir]
  (map line-count
       (filter file?
               (file-seq (file dir)))))


(reduce + (file-counts "."))
```

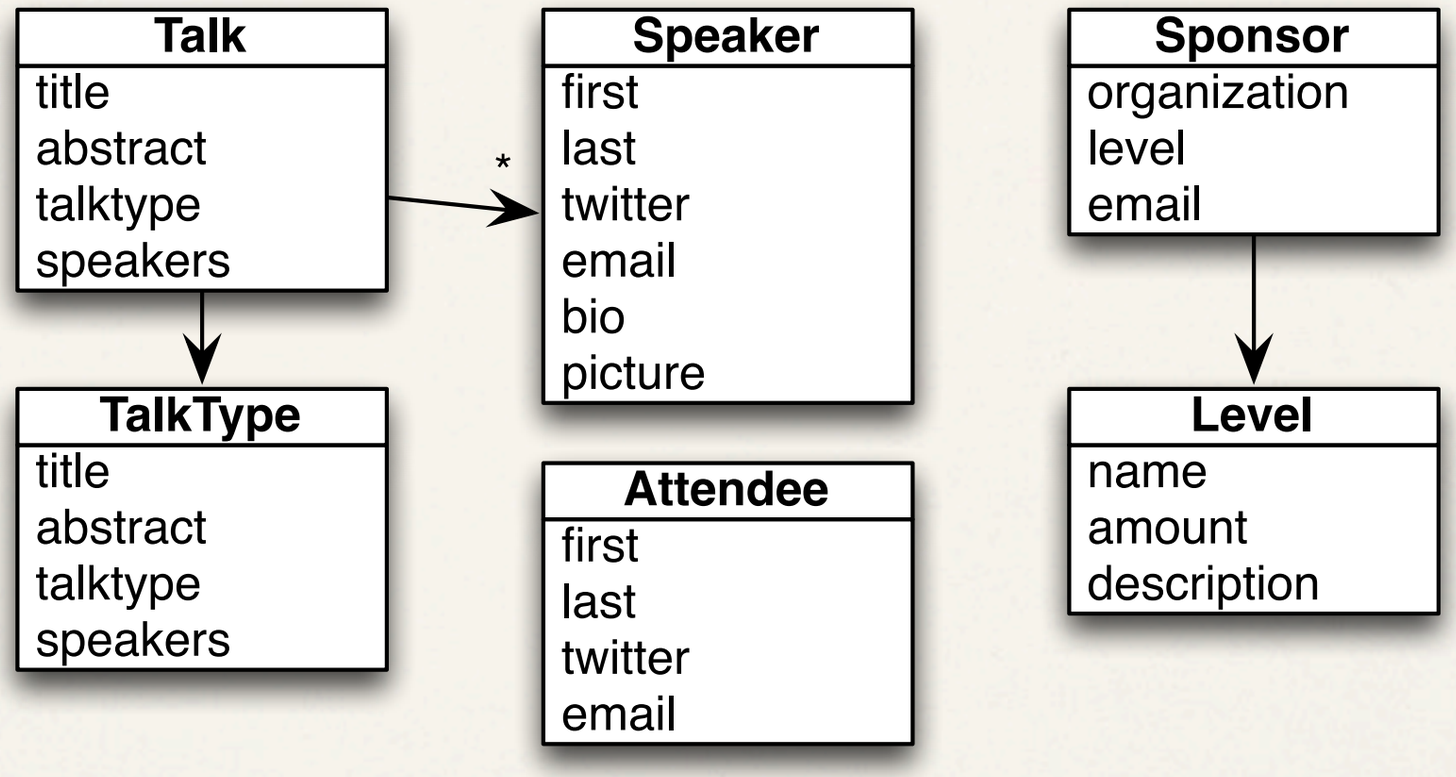
Line counts

```
(defn line-count [file]
  (count (line-seq (reader file))))

(defn file? [file]
  (.isFile file))

(defn file-counts [dir]
  (->> dir          ;; start with directory string
       file         ;; convert to java.io.File
       file-seq     ;; get sequence of files in dir
       (filter file?)       ;; get only the files
       (map line-count)))  ;; line-count each

(reduce + (file-counts "."))
```

Roadmap

- Values vs objects
- Collections
- Sequences
- **Generic data interfaces**
- Identity and state

Representing entities



Custom data interface

```
public class Speaker {
	private String first;
	private String last;
	private String twitterID;

	public Speaker(String first, String last, String twitterID) {
		super();
		this.first = first;
		this.last = last;
		this.twitterID = twitterID;
	}

	public String getFirst() {
		return first;
	}
	public void setFirst(String first) {
		this.first = first;
	}
	public String getLast() {
		return last;
	}
	public void setLast(String last) {
		this.last = last;
	}
	public String getTwitterID() {
		return twitterID;
	}
	public void setTwitterID(String twitterID) {
		this.twitterID = twitterID;
	}

	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result + ((first == null) ? 0 :
first.hashCode());
		result = prime * result + ((last == null) ? 0 :
last.hashCode());
		result = prime * result
				+ ((twitterID == null) ? 0 :
twitterID.hashCode());
		return result;
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Speaker other = (Speaker) obj;
		if (first == null) {
			if (other.first != null)
				return false;
		} else if (!first.equals(other.first))
			return false;
		if (last == null) {
			if (other.last != null)
				return false;
		} else if (!last.equals(other.last))
			return false;
		if (twitterID == null) {
			if (other.twitterID != null)
				return false;
		} else if (!twitterID.equals(other.twitterID))
			return false;
		return true;
	}

	@Override
	public String toString() {
		return "Speaker [first=" + first + ", last=" + last + ",
twitterID="
				+ twitterID + "]";
	}

}
```

Custom data interface

Fields

```
public class Speaker {
	private String first;
	private String last;
	private String twitterID;

	public Speaker(String first, String last, String twitterID) {
		super();
		this.first = first;
		this.last = last;
		this.twitterID = twitterID;
	}

	public String getFirst() {
		return first;
	}
	public void setFirst(String first) {
		this.first = first;
	}
	public String getLast() {
		return last;
	}
	public void setLast(String last) {
		this.last = last;
	}
	public String getTwitterID() {
		return twitterID;
	}
	public void setTwitterID(String twitterID) {
		this.twitterID = twitterID;
	}

	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result + ((first == null) ? 0 :
first.hashCode());
		result = prime * result + ((last == null) ? 0 :
last.hashCode());
		result = prime * result
				+ ((twitterID == null) ? 0 :
twitterID.hashCode());
		return result;
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Speaker other = (Speaker) obj;
		if (first == null) {
			if (other.first != null)
				return false;
		} else if (!first.equals(other.first))
			return false;
		if (last == null) {
			if (other.last != null)
				return false;
		} else if (!last.equals(other.last))
			return false;
		if (twitterID == null) {
			if (other.twitterID != null)
				return false;
		} else if (!twitterID.equals(other.twitterID))
			return false;
		return true;
	}

	@Override
	public String toString() {
		return "Speaker [first=" + first + ", last=" + last + ",
twitterID="
				+ twitterID + "]";
	}

}
```

Custom data interface

```
public class Speaker {
	private String first;
	private String last;
	private String twitterID;
```

Fields

```
	public Speaker(String first, String last, String twitterID) {
		super();
		this.first = first;
		this.last = last;
		this.twitterID = twitterID;
	}
```

Constructor

```
	public String getFirst() {
		return first;
	}
	public void setFirst(String first) {
		this.first = first;
	}
	public String getLast() {
		return last;
	}
	public void setLast(String last) {
		this.last = last;
	}
	public String getTwitterID() {
		return twitterID;
	}
	public void setTwitterID(String twitterID) {
		this.twitterID = twitterID;
	}

	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result + ((first == null) ? 0 :
first.hashCode());
		result = prime * result + ((last == null) ? 0 :
last.hashCode());
		result = prime * result
				+ ((twitterID == null) ? 0 :
twitterID.hashCode());
		return result;
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Speaker other = (Speaker) obj;
		if (first == null) {
			if (other.first != null)
				return false;
		} else if (!first.equals(other.first))
			return false;
		if (last == null) {
			if (other.last != null)
				return false;
		} else if (!last.equals(other.last))
			return false;
		if (twitterID == null) {
			if (other.twitterID != null)
				return false;
		} else if (!twitterID.equals(other.twitterID))
			return false;
		return true;
	}

	@Override
	public String toString() {
		return "Speaker [first=" + first + ", last=" + last + ",
twitterID="
				+ twitterID + "]";
	}

}
```

Custom data interface

Fields

```
public class Speaker {
	private String first;
	private String last;
	private String twitterID;
```

Constructor

```
	public Speaker(String first, String last, String twitterID) {
		super();
		this.first = first;
		this.last = last;
		this.twitterID = twitterID;
	}
```

Getters, setters

```
	public String getFirst() {
		return first;
	}
	public void setFirst(String first) {
		this.first = first;
	}
	public String getLast() {
		return last;
	}
	public void setLast(String last) {
		this.last = last;
	}
	public String getTwitterID() {
		return twitterID;
	}
	public void setTwitterID(String twitterID) {
		this.twitterID = twitterID;
	}
```

```
	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result + ((first == null) ? 0 :
first.hashCode());
		result = prime * result + ((last == null) ? 0 :
last.hashCode());
		result = prime * result
				+ ((twitterID == null) ? 0 :
twitterID.hashCode());
		return result;
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Speaker other = (Speaker) obj;
		if (first == null) {
			if (other.first != null)
				return false;
		} else if (!first.equals(other.first))
			return false;
		if (last == null) {
			if (other.last != null)
				return false;
		} else if (!last.equals(other.last))
			return false;
		if (twitterID == null) {
			if (other.twitterID != null)
				return false;
		} else if (!twitterID.equals(other.twitterID))
			return false;
		return true;
	}

	@Override
	public String toString() {
		return "Speaker [first=" + first + ", last=" + last + ",
twitterID="
				+ twitterID + "]";
	}

}
```

Custom data interface

Fields

```
public class Speaker {
	private String first;
	private String last;
	private String twitterID;
```

Constructor

```
	public Speaker(String first, String last, String twitterID) {
		super();
		this.first = first;
		this.last = last;
		this.twitterID = twitterID;
	}
```

Getters, setters

```
	public String getFirst() {
		return first;
	}
	public void setFirst(String first) {
		this.first = first;
	}
	public String getLast() {
		return last;
	}
	public void setLast(String last) {
		this.last = last;
	}
	public String getTwitterID() {
		return twitterID;
	}
	public void setTwitterID(String twitterID) {
		this.twitterID = twitterID;
	}
```

hashCode

```
	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result + ((first == null) ? 0 :
first.hashCode());
		result = prime * result + ((last == null) ? 0 :
last.hashCode());
		result = prime * result
				+ ((twitterID == null) ? 0 :
twitterID.hashCode());
		return result;
	}
```

```
	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Speaker other = (Speaker) obj;
		if (first == null) {
			if (other.first != null)
				return false;
		} else if (!first.equals(other.first))
			return false;
		if (last == null) {
			if (other.last != null)
				return false;
		} else if (!last.equals(other.last))
			return false;
		if (twitterID == null) {
			if (other.twitterID != null)
				return false;
		} else if (!twitterID.equals(other.twitterID))
			return false;
		return true;
	}

	@Override
	public String toString() {
		return "Speaker [first=" + first + ", last=" + last + ",
twitterID="
				+ twitterID + "]";
	}

}
```

Custom data interface

Fields

```
public class Speaker {
	private String first;
	private String last;
	private String twitterID;
```

Constructor

```
	public Speaker(String first, String last, String twitterID) {
		super();
		this.first = first;
		this.last = last;
		this.twitterID = twitterID;
	}
```

Getters, setters

```
	public String getFirst() {
		return first;
	}
	public void setFirst(String first) {
		this.first = first;
	}
	public String getLast() {
		return last;
	}
	public void setLast(String last) {
		this.last = last;
	}
	public String getTwitterID() {
		return twitterID;
	}
	public void setTwitterID(String twitterID) {
		this.twitterID = twitterID;
	}
```

hashCode

```
	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result + ((first == null) ? 0 :
first.hashCode());
		result = prime * result + ((last == null) ? 0 :
last.hashCode());
		result = prime * result
				+ ((twitterID == null) ? 0 :
twitterID.hashCode());
		return result;
	}
```

equals

```
	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Speaker other = (Speaker) obj;
		if (first == null) {
			if (other.first != null)
				return false;
		} else if (!first.equals(other.first))
			return false;
		if (last == null) {
			if (other.last != null)
				return false;
		} else if (!last.equals(other.last))
			return false;
		if (twitterID == null) {
			if (other.twitterID != null)
				return false;
		} else if (!twitterID.equals(other.twitterID))
			return false;
		return true;
	}
```

```
	@Override
	public String toString() {
		return "Speaker [first=" + first + ", last=" + last + ",
twitterID="
				+ twitterID + "]";
	}

}
```

Custom data interface

Fields

```
public class Speaker {
	private String first;
	private String last;
	private String twitterID;
```

Constructor

```
	public Speaker(String first, String last, String twitterID) {
		super();
		this.first = first;
		this.last = last;
		this.twitterID = twitterID;
	}
```

Getters, setters

```
	public String getFirst() {
		return first;
	}
	public void setFirst(String first) {
		this.first = first;
	}
	public String getLast() {
		return last;
	}
	public void setLast(String last) {
		this.last = last;
	}
	public String getTwitterID() {
		return twitterID;
	}
	public void setTwitterID(String twitterID) {
		this.twitterID = twitterID;
	}
```

hashCode

```
	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result + ((first == null) ? 0 :
first.hashCode());
		result = prime * result + ((last == null) ? 0 :
last.hashCode());
		result = prime * result
				+ ((twitterID == null) ? 0 :
twitterID.hashCode());
		return result;
	}
```

equals

```
	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Speaker other = (Speaker) obj;
		if (first == null) {
			if (other.first != null)
				return false;
		} else if (!first.equals(other.first))
			return false;
		if (last == null) {
			if (other.last != null)
				return false;
		} else if (!last.equals(other.last))
			return false;
		if (twitterID == null) {
			if (other.twitterID != null)
				return false;
		} else if (!twitterID.equals(other.twitterID))
			return false;
		return true;
	}
```

toString

```
	@Override
	public String toString() {
		return "Speaker [first=" + first + ", last=" + last + ",
twitterID="
				+ twitterID + "]";
	}

}
```

Map representation

```
(def alex
  {:first "Alex"
   :last "Miller"
   :twitter "puredanger"
   :email "alex@puredanger.com"
   :bio "I like nachos."
   :picture "alex.jpg"})
```

Speaker
first
last
twitter
email
bio
picture

Where's the type?

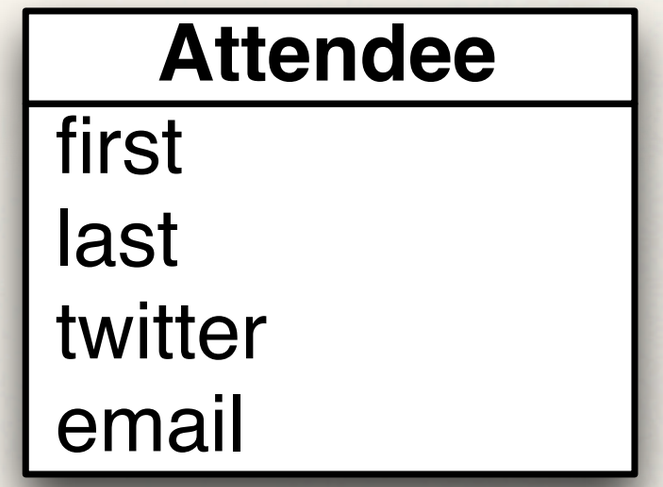
Map representation

```
(def alex
  {:first "Alex"
   :last "Miller"
   :twitter "puredanger"
   :email "alex@puredanger.com"
   :bio "I like nachos."
   :picture "alex.jpg"
   :type "Speaker"})
```

Speaker
first
last
twitter
email
bio
picture

More entities

```
(def attendee
  {:type "Attendee"
   :first "Peter"
   :last "Gibbons"
   :email "peter@initech.com"})

(def sponsor
  {:type "Sponsor"
   :organization "Initech"
   :level :gold
   :email "marketing@initech.com"})
```



Extracting an interface in OO

```
public interface Emailable {
  public String getEmail();
}
```

```
public class Speaker implements Emailable {
  private String email;
  public String getEmail() { return email; }
}
```

```
...
```

Preparing an email blast

```
(def entities (concat attendees speakers sponsors))

(map :email entities)
```

Universal getters and setters

```
;; define entity
(def peter {:first "peter"
            :last "gibbons"
            :email "peter@initech.com"})

;; associate a new key/value pair in entity
(assoc peter :twitter "tps")

;; get field in entity
(get peter :first)
(:first peter)
```

Access to all fields or all values

```
;; Get all fields in an entity
(keys speaker)

;; Does speaker have a twitter or github id?
(defn has-twitter? [speaker]
  (or (contains? speaker :twitter)
      (contains? speaker :github)))

;; Get all values in an entity
(vals speaker)
```

Traversable entities

```
;; Reach into nested entities
(get-in track [:speaker :first])

;; Update inside nested entities
(update-in track [:speaker :last] str/upper-case)

;; Add new key/value inside nested entities
(assoc-in track [:speaker] {:first "..."})
```

Walk

```
;; Helper function to prepend an @ to twitter id
(defn prefix-at [entity]
  (if (and (associative? entity)
           (contains? entity :twitter))
      (str "@" (:twitter entity))
      entity))

;; Walk through a tree depth-first, post-order
(postwalk prefix-at conference)
```

Zippers

- Allow you to traverse and modify a tree as a functional data structure

- Go read about it!

Records

- Often useful to switch behavior based on type

```
(defrecord Speaker [first last twitter email])
```

```
(def alex
  (->Speaker "Alex" "Miller" "puredanger"
             "alex@puredanger.com"))

(class alex)  ;; user.Speaker
```

Cards demo

```
(defrecord Card [suit value])

(def suits [:heart :spade :diamond :club])
(def values (map keyword
                 (concat (map str (range 2 11))
                         ["J" "Q" "K" "A"])))
(def full-deck (for [suit suits,
                     value values]
                 (->Card suit value)))

(defn cut [deck]
  (concat (drop 26 deck) (take 26 deck)))
```

Polymorphism

- Dynamically choosing code to run
- Conditional logic - if, cond, condp, case, etc
- Multimethods - switch based on arbitrary function
- Protocols - switch based on type

Multimethods

Arbitrary dispatch function

```
(defmulti area :type)

(defmethod area :circle [{radius :radius}]
  (* 3.14159 radius radius))

(defmethod area :square [{side :side}]
  (* side side))

(area {:type :square :side 5})
(area {:type :circle :radius 5})
```

Protocols

```
(defprotocol Shape
  (area [shape]))

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

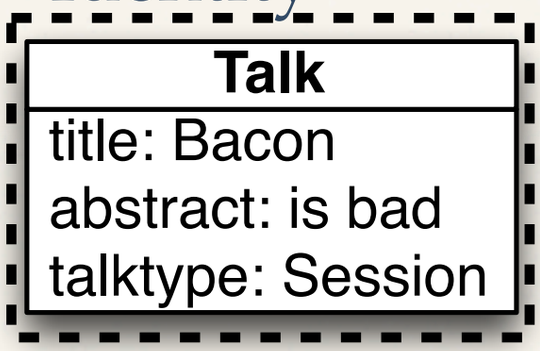
```
(extend-protocol Shape
  Circle
  (area [{radius :radius}]
    (* 3.14159 radius radius))
  Square
  (area [{side :side}]
    (* side side)))
```

```
(area (->Square 5))
(area (->Circle 5))
```

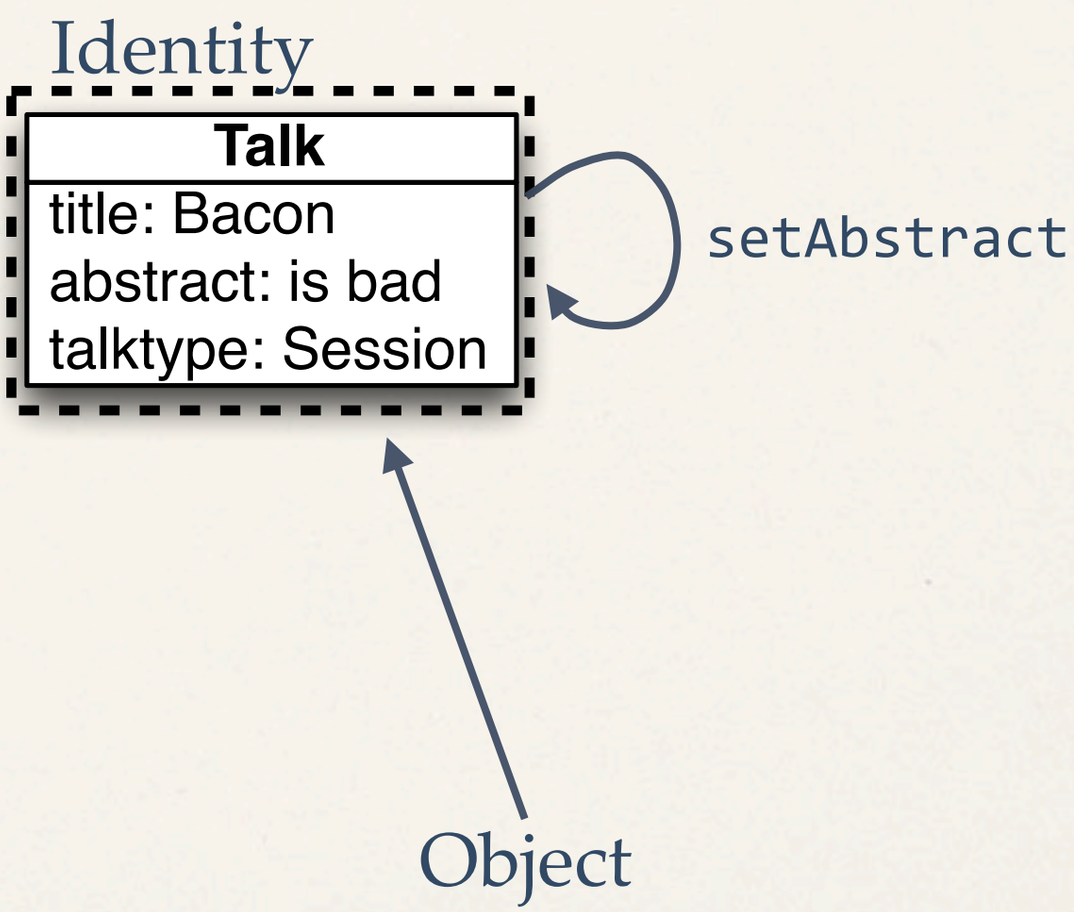
Roadmap

- Values vs objects
- Collections
- Sequences
- Generic data interfaces
- **Identity and state**

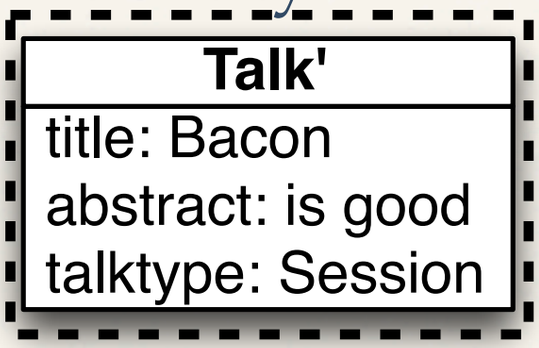
Identity



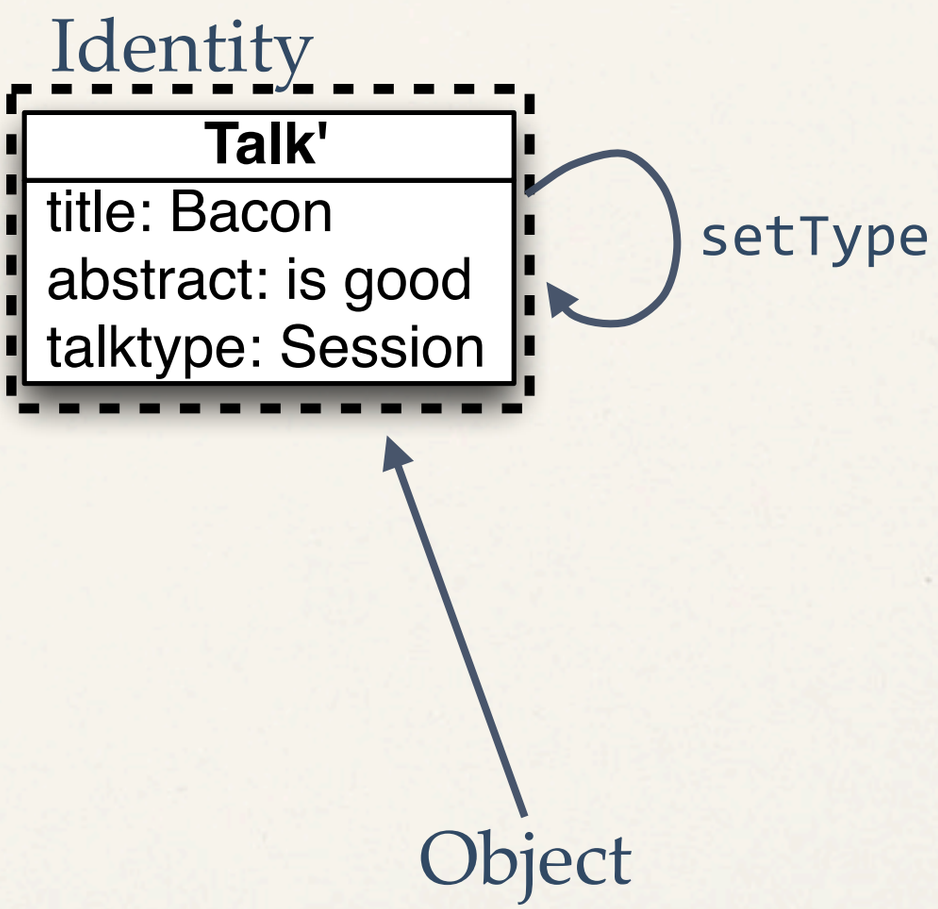
Object

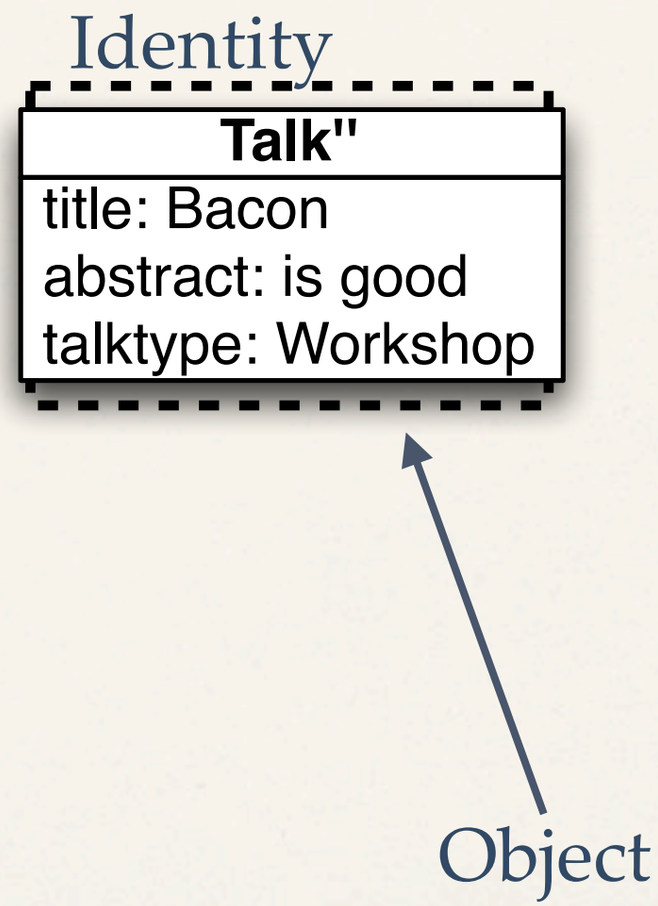

Identity
Talk
title: Bacon
abstract: is bad
talktype: Session
setAbstract
Object

Identity



Object

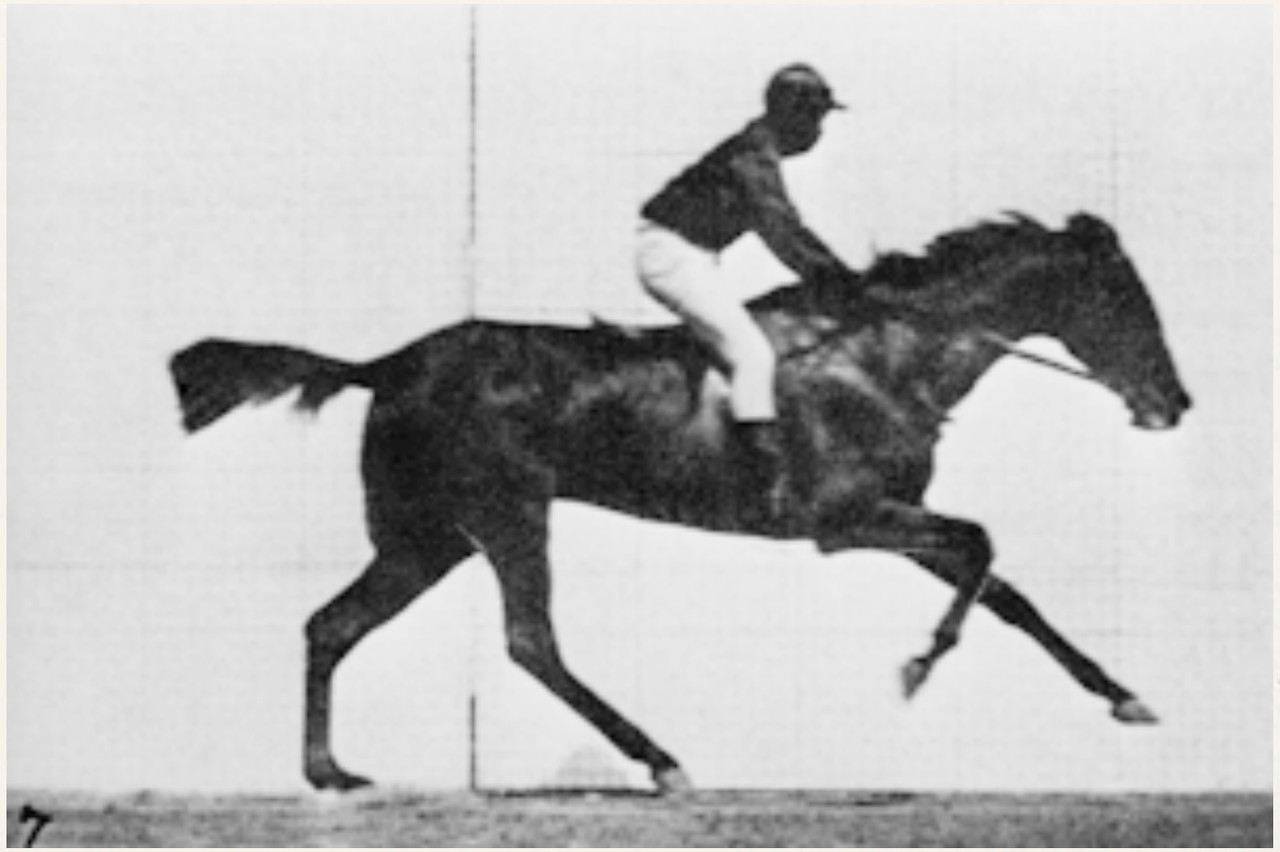

Identity
Talk'
title: Bacon
abstract: is good
talktype: Session
setType
Object


Identity
Talk"
title: Bacon
abstract: is good
talktype: Workshop
Object

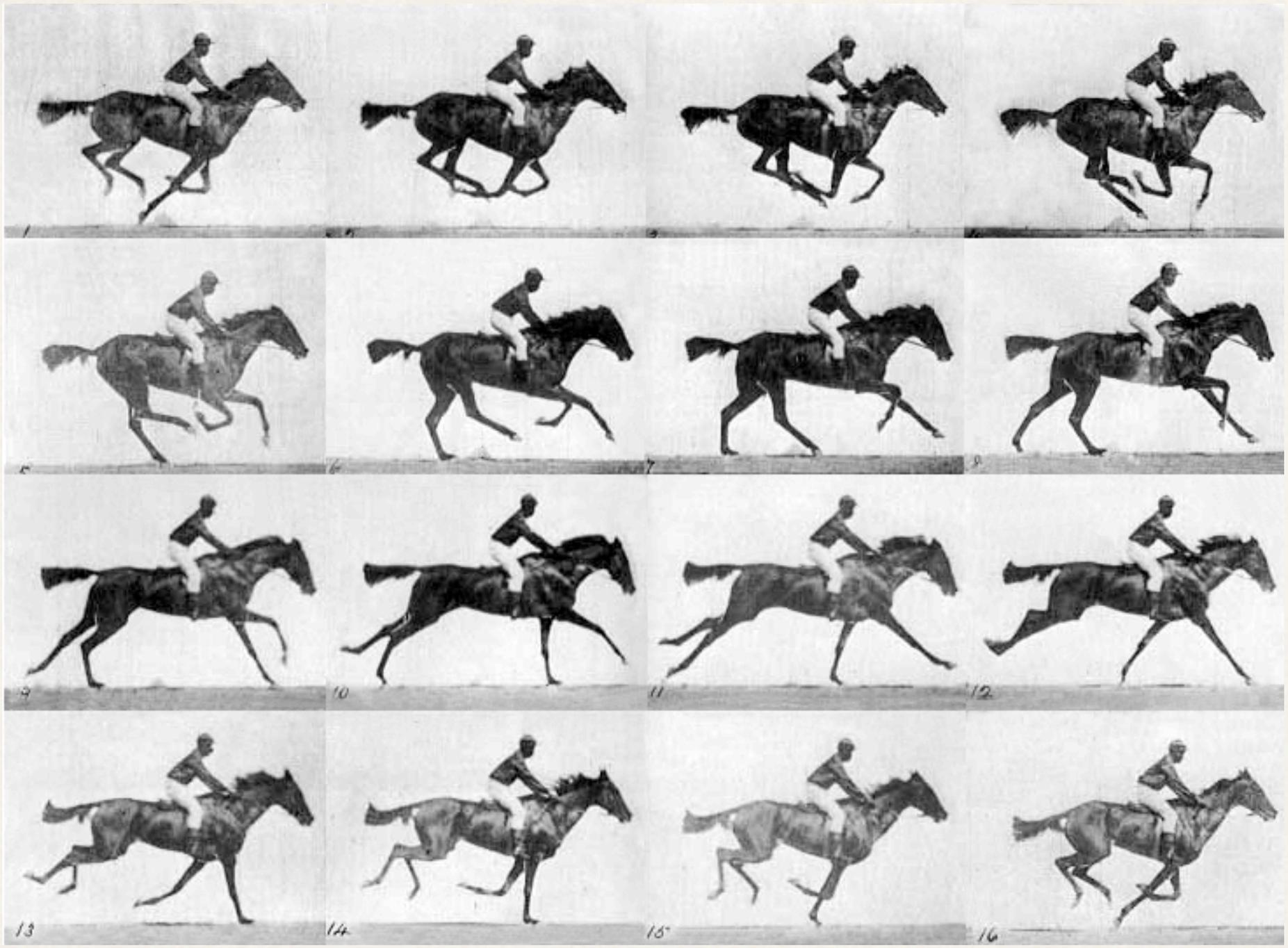
With objects, we must be concerned with how a computational object can change and yet maintain its identity. This will force us to **abandon our old substitution model of computation** in favor of a more mechanistic but less theoretically tractable environment model of computation. The difficulties of dealing with objects, change, and identity are **a fundamental consequence of the need to grapple with time in our computational models**. These difficulties become even greater when we allow the possibility of concurrent execution of programs.

Structure and Interpretation of Computer Programs
- Abelson, Sussman, and Sussman

Edward Muybridge

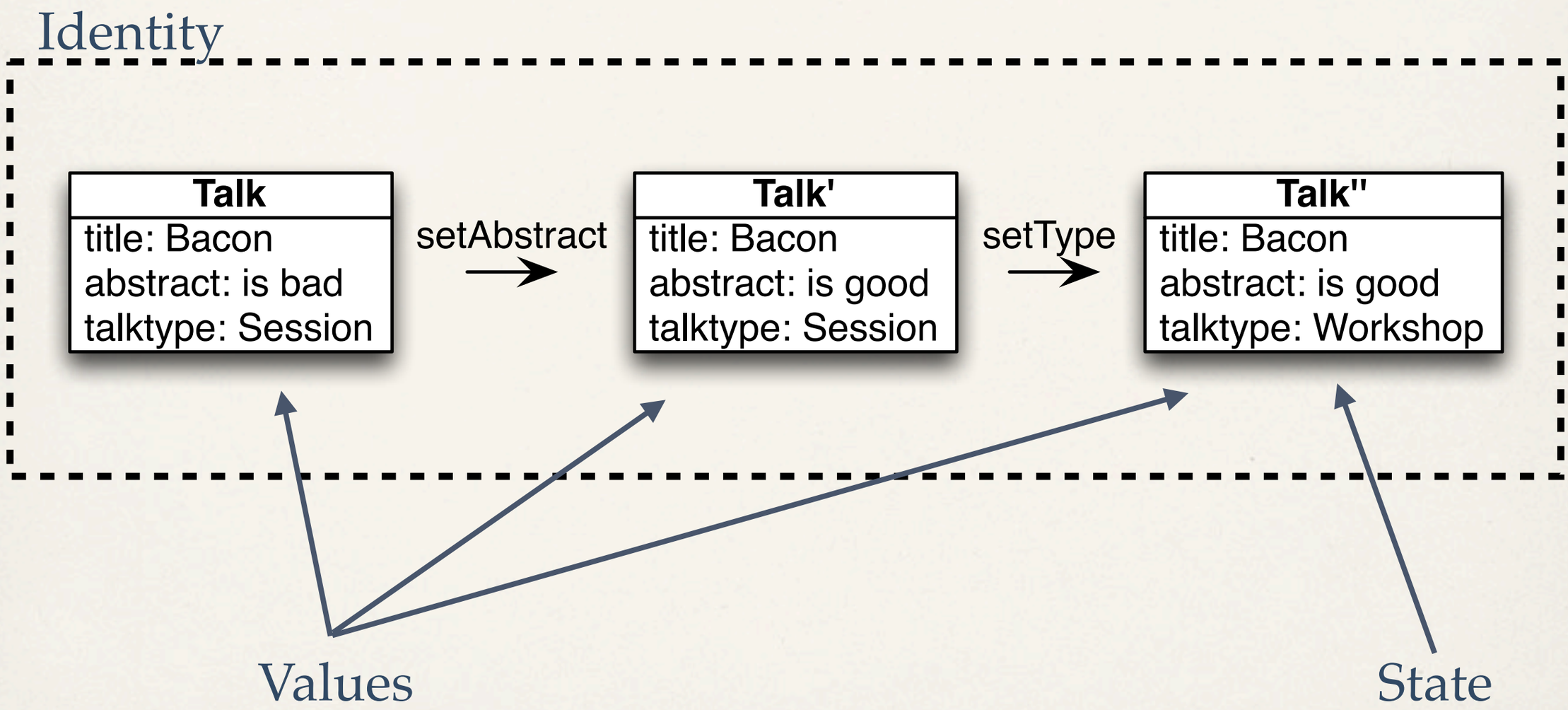


The Horse in Motion (1878)



The Horse in Motion (1878)

State model


Identity
Talk
title: Bacon
abstract: is bad
talktype: Session
setAbstract
Talk'
title: Bacon
abstract: is good
talktype: Session
setType
Talk''
title: Bacon
abstract: is good
talktype: Workshop
Values
State

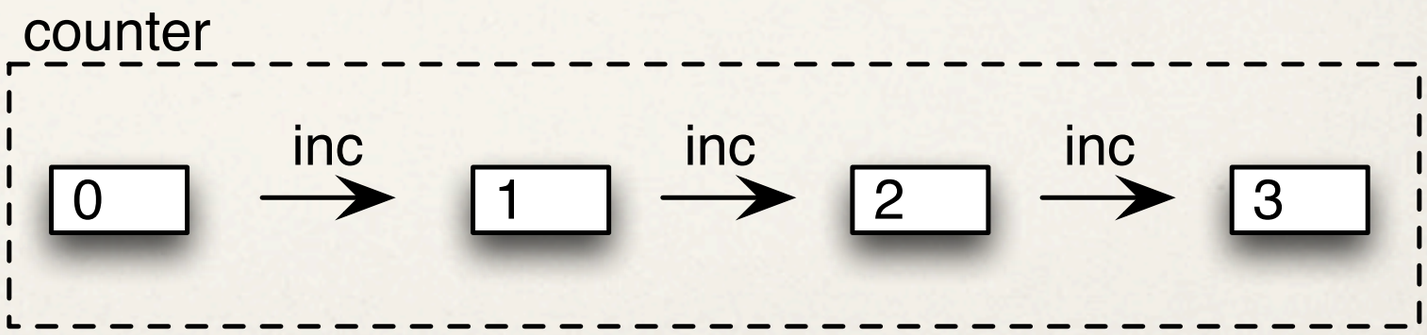
Representing state

- Vars - mutable storage with per thread-bindings (def)
- Atoms - single timeline coordination
- Refs - multi timeline coordination
- Agents - asynchronous single timeline coordination

Atoms

- Atoms are **uncoordinated synchronized** state

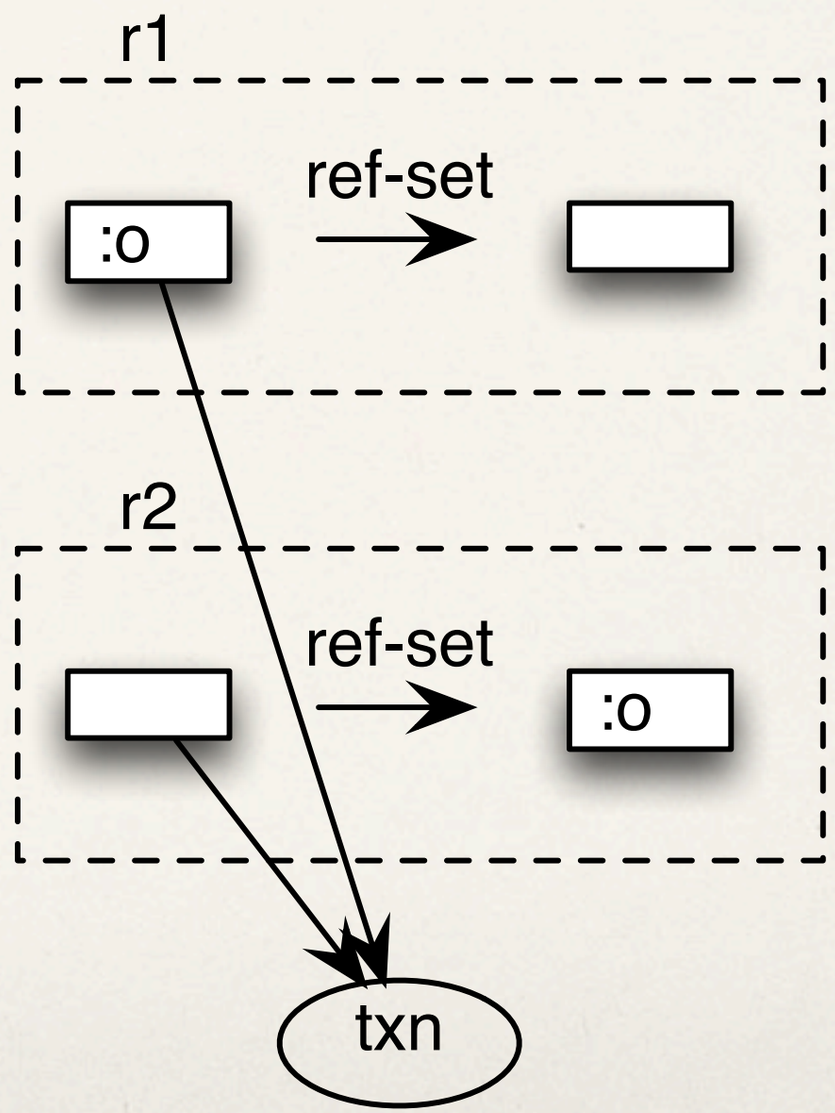
```
;; Create a counter atom initialized to 0
(def counter (atom 0))

;; Swap the atom to a new value
(defn id [] (swap! counter inc))

(id)  ;; 1
(id)  ;; 2
(id)  ;; 3
```



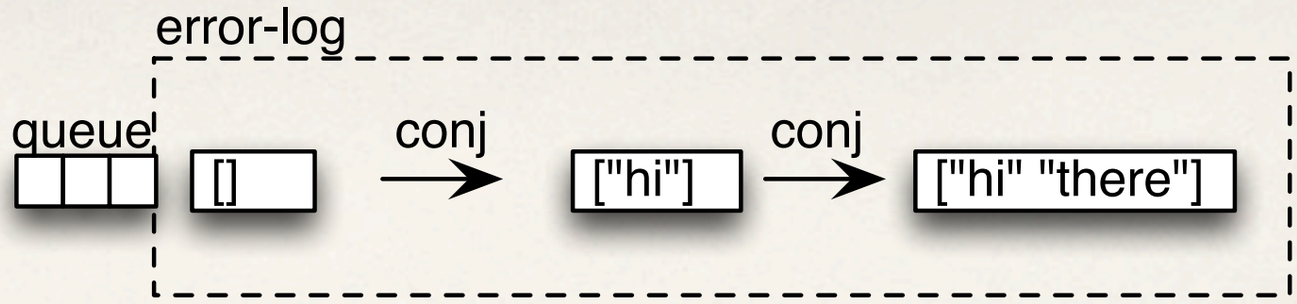
References

- References are for **coordinated synchronous** state (using STM)

```
;; Create references
(def r1 (ref [:o]))
(def r2 (ref []))

(defn yoyo []
  (dosync
   (let [v1 @r1
         v2 @r2]
     (ref-set r1 v2)
     (ref-set r2 v1))
   [@r1 @r2]))
(yoyo)
```



Agents



- Agents are for uncoordinated **asynchronous** state

```
;; Create agent around a vector
(def error-log (agent []))

;; Log a message by sending a function to the agent
(defn log [msg] (send-off error-log conj msg))

(log "hi")
(log "there")
;; Deref to observe the agent
@error-log
```

To sum up...

- Objects are not composite values
- Collections - are immutable composite values
- Sequences - unifies composite values with FP
- Generic data interface - powerful tools for manipulating entities
- Identity and state - reference types separate identity from value

References

- More on values and state:
  - http://www.infoq.com/presentations/Are-We-There-Yet-Rich-Hickey
  - http://www.infoq.com/presentations/Value-Values
- Find me here:
  - @puredanger
  - alex@puredanger.com
  - http://tech.puredanger.com