

Records

Dr. Mattox Beckman

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
DEPARTMENT OF COMPUTER SCIENCE



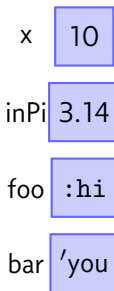
Objectives

- ▶ Explain the difference between a scalar and a reference.
 - ▶ Categorize **CLOJURE** expressions as either scalar or reference.
 - ▶ Draw memory diagrams for scalars and references.
 - ▶ Critique erroneous memory diagrams.
- ▶ Understand the syntax and semantics of **CLOJURE**'s records.
 - ▶ Define record types.
 - ▶ Show how to create an instance of a record.
 - ▶ Draw memory diagrams corresponding to **CLOJURE** code.
 - ▶ Write **CLOJURE** code corresponding to a memory diagram.

Scalars

- ▶ A *scalar* is a value that can fit into a single machine word (currently 32 or 64 bits).
- ▶ Typical scalars: integers, floats
- ▶ Pretend scalars: keywords, symbols
- ▶ Memory diagram: put the value in a box.

```
1 (def x 10)
2 (def inPi 3.14)
3 (def foo :hi)
4 (def bar 'you)
```



Your Turn

- Draw a memory diagram for the following CLOJURE code:

```
1 (def a 12.32)
2 (def b 'x)
```

- Write CLOJURE code that will produce the following diagram.

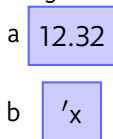
a 

e 

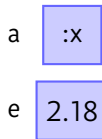
Your Turn

- Draw a memory diagram for the following CLOJURE code:

```
1 (def a 12.32)
2 (def b 'x)
```



- Write CLOJURE code that will produce the following diagram.



```
1 (def a :x)
2 (def e 2.18)
```

Copying

- ▶ When a scalar is copied, the second box has the same address as the first. *The data is not copied.*

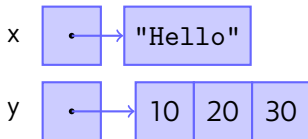
```
1 (def x 10)
2 (def y x)
```

x,y 

References

- ▶ If it's too big to fit in a single word, then we use a *reference* (sometimes called a *pointer*) to handle it.
- ▶ A reference is drawn as an arrow.
- ▶ Two references are equal if their destinations are the same. The source doesn't count.
- ▶ Typical references: strings, vectors, hashmaps, records... any compound type.

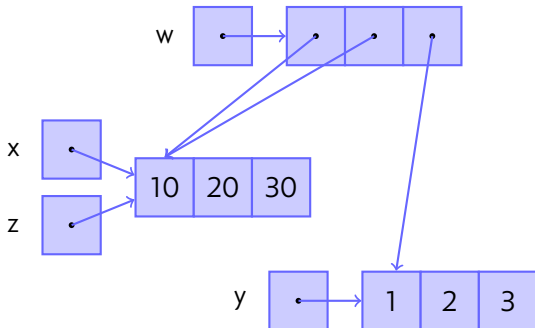
```
1 (def x "Hello")  
2 (def y [10 20 30])
```



Examples

- ▶ When references are copied, we create an arrow with *the same destination*.
- ▶ An arrow can only point to a collection as a whole, never an individual element within the collection.

```
1 (def x [10 20 30])  
2 (def y [1 2 3])  
3 (def z x)  
4 (def w [x z y])
```

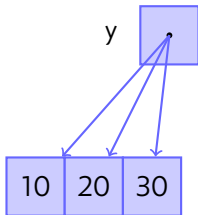


Only one destination!

- ▶ A pointer can only point to one destination at a time!!!

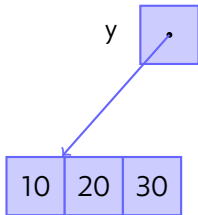
- ▶ **WRONG**

```
1 (def y [10 20 30])
```



- ▶ **RIGHT**

```
1 (def y [10 20 30])
```



Your Turn 2a

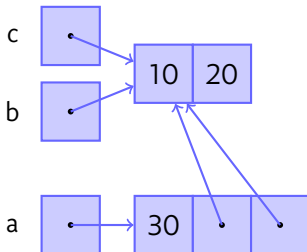
- Draw a memory diagram for the following CLOJURE code:

```
1 (def c [10 20])  
2 (def b c)  
3 (def a [30 b c])
```

Your Turn 2a

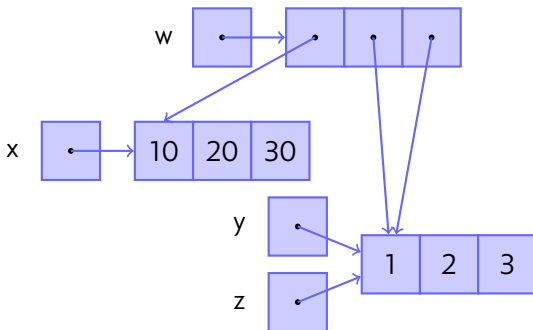
- Draw a memory diagram for the following **CLOJURE** code:

```
1 (def c [10 20])  
2 (def b c)  
3 (def a [30 b c])
```



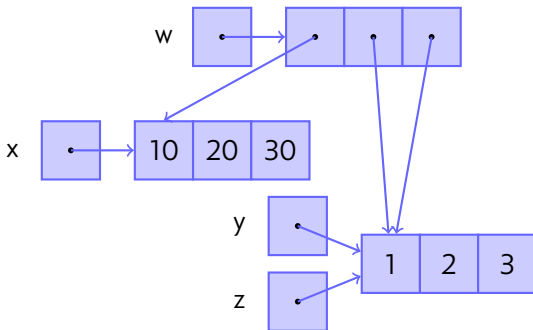
Your Turn 2b

- Write **CLOSURE** code that will produce the following diagram.



Your Turn 2b

- Write **CLOJURE** code that will produce the following diagram.



```
1 (def x [10 20 30])
2 (def y [1 2 3])
3 (def z y)
4 (def w [x y y])
```

Defining a Record

```
1 (defrecord name [fields*] specs*)
```

- ▶ The name can be any legal **CLOJURE** name. We usually capitalize the first letter.
- ▶ The fields are usually lower case, as other **CLOJURE** variables.
- ▶ We will discuss specs in a future lecture.
- ▶ This creates an actual Java class!

Example

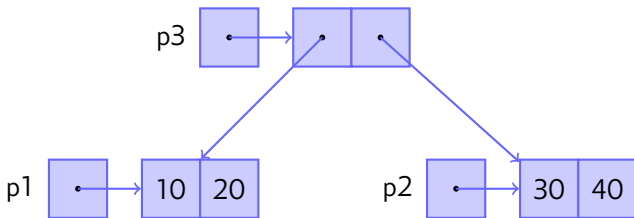
- ▶ Create an instance using the name followed by a dot.
- ▶ Keywords become field lookup functions, like in hash maps.

```
1 user> (defrecord Pair [x y])
2 user.Pair
3 user> (def p1 (Pair. 10 20))
4 #'user/p1
5 user> p1
6 #user.Pair{:x 10, :y 20}
7 user> (:x p1)
8 10
```



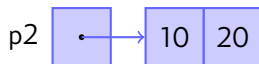
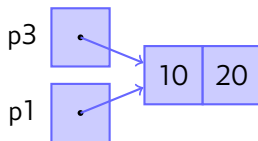
scalars vs references

```
1 user> (def p1 (Pair. 10 20))  
2 user> (def p2 (Pair. 30 40))  
3 user> (def p3 (Pair. p1 p2))
```



Equality

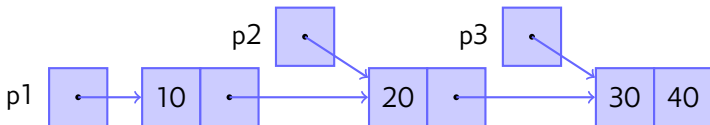
```
1 user> (def p1 (Pair. 10 20))
2 user> (def p2 (Pair. 10 20))
3 user> (def p3 p1)
```



```
1 user> (= p1 p2)
2 true
3 user> (= p1 p3)
4 true
5 user> (identical? p1 p2)
6 false
7 user> (identical? p1 p3)
8 true
```

the `->` macro

```
1 user> (def p3 (Pair. 30 40))
2 user> (def p2 (Pair. 20 p3))
3 user> (def p1 (Pair. 10 p2))
```



```
1 user> (:x (:y (:y p1)))
2 30
3 user> (-> p1 :y :y :x)
4 30
```

Examples

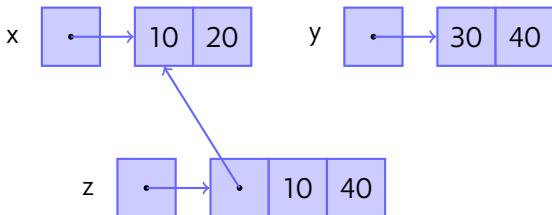
- What do you think this code will do?

```
1 (defrecord Double [x y])  
2 (defrecord Triple [a b c])  
3 (def x (Double. 10 20))  
4 (def y (Double. 30 40))  
5 (def z (Triple. x (:x x) (:y y)))
```

Examples

- What do you think this code will do?

```
1 (defrecord Double [x y])  
2 (defrecord Triple [a b c])  
3 (def x (Double. 10 20))  
4 (def y (Double. 30 40))  
5 (def z (Triple. x (:x x) (:y y)))
```



Examples

- What do you think this code will do?

```
1 (defrecord Double [x y])  
2 (defrecord Triple [a b c])  
3 (def x (Double. 10 20))  
4 (def y (Double. x 40))  
5 (def z (Triple. x (:x y) y))
```

Examples

- What do you think this code will do?

```
1 (defrecord Double [x y])  
2 (defrecord Triple [a b c])  
3 (def x (Double. 10 20))  
4 (def y (Double. x 40))  
5 (def z (Triple. x (:x y) y))
```

