Immutability: obstacle or tool?

Discuss based on:
- Programming experience in 251 and previously
- Readings about language implementation and GC

• Efficiency?
• Reliability?
• Ease of making/avoiding mistakes?
• Clarity?
• ...

• Try for at least 3 pros and 3 cons; OK to disagree.

“In a world where bindings and values are immutable…”

- Have you noticed?

- Patterns for accumulating results
  (when your Java brain says "x++", etc.):
  - Build result recursively
  - Create fresh copy
  - "Thread state through" in the style of fold!
    - Small function "does one step"
    - HOF passes result on to the next step.

Cannot tell if you copy

```
(define (sort-pair p)
  (if (< (car p) (cdr p))
    p
    (cons (cdr p) (car p)))))
```

```
(define (sort-pair p)
  (if (< (car p) (cdr p))
    (cons (car p) (cdr p))
    (cons (cdr p) (car p)))))
```

**Without mutation**, these two implementations are indistinguishable
- Change at any time without introducing bugs outside.

Motivating examples/slides adapted from Dan Grossman

Suppose we had mutation...

```
(define x (mcons 3 4))
(define y (sort-pair x))

; mutate car of x to hold 5
(set-mcar! x 5)
(define z (modr y))
```

- What is z?
  - Depends on sort-pair implementation
    - Document and be very careful.
    - Changing implementation requires changing uses

This code is close to (but not quite) working Racket...
An even better example

```
(define (append xs ys)
  (if (null? xs)
      ys
      (cons (car xs) (append (cdr xs) ys)))))
(define x (list 2 4))
(define y (list 5 3 0))
(define z (append x y))
```

or

```
x → 2 4
y → 5 3 0
z → 2 4 5 3 0
```

Java security nightmare (really happened)

```
class ProtectedResource {
  private Resource theResource = ...;
  private String[] allowedUsers = ...;
  public String[] getAllowedUsers() {
    return allowedUsers;
  }
  public String currentUser() { ... }
  public void useTheResource() {
    for (int i=0; i < allowedUsers.length; i++) {
      if (currentUser().equals(allowedUsers[i])) {
        ... // access allowed: use it
        return;
      }
    }
    throw new IllegalAccessException();
  }
}
```

Mutant users!
The problem:

```
p.getAllowedUsers()[0] = p.currentUser();
p.useTheResource();
```

The fix:

```
public String[] getAllowedUsers() {
  _ return a copy of allowedUsers _
}
```

Could this happen without mutability?

A biasing on aliasing

**Immutability**

- Aliasing choices do not affect correctness, just performance.
- Other code can't break your code.
- Changing your choice can't break other code.
- Document what, not how.
- Start with safety, optimize for performance.

**Mutability**

- Aliasing choices do affect correctness and performance.
- Other code can break your code, depending on your choice.
- Changing your choice can break other code.
- Document what and how.
- Start with performance (maybe), optimize for safety.
**A broader PL theme:**

- Limiting how programs can be expressed can be very useful!

- *Not* limiting what computable functions can be implemented, just how.

- Less is more (reliable)?