MODULARITY, OBJECTS, AND STATE

OBJECT MODELING

- This semester we've slowly been working through all the portions of a programming language
- We are now at what (unfortunately) is often a *first* step in teaching programming; object models
- I believe that it's wrong to start off with object-oriented programming in CS (especially as taught in High School), and it's because of what we miss along the way

WHY OBJECTS?

- So far we've used procedures and data to design programs
- We've written procedures that take other procedures as parameters
- We have discussed modularity through message dispatch and lookup tables to provide single interfaces to multiple data abstractions
- As we design successively more complex systems, we need different strategies of organization

WHY OBJECTS?

- We build a system where we have computational objects that represent each object in the modeled system
- By using this design mechanism, we will not need to make changes to existing objects in order to extend the overall system to include new modeled objects
- We should be able to localize our feature set and work on only one portion of the system at a time

DEMYSTIFYING OBJECTS

- It’s very tempting to say that we 'understand' objects because we’ve been using them for a significant portion of our programming lifetimes
- Hopefully our discussions about procedures and data show how this isn’t entirely true
- We’re going to work through objects to provide a deeper conceptual understanding of how they work

ASSIGNMENT AND LOCAL STATE

Viewing the world as objects

STATE

- In an object-oriented system, we view the world as populated by independent (and interdependent) objects
- Each object has a *state* that changes over time
- An object is said to have *state* if its behavior is influenced by its history

HISTORICAL PROGRAMMING

- State being based on history means the following:
- If you have a bank account which begins empty, then add $100 to it, then withdraw $25, it has a specific *state* of $75
- We can write procedures that inquire about the *state* of an object and make decisions based upon that *state*.
- Objects have one or more *state variables*

CHANGING STATE

- In order for objects to interact, they have to be able to *assign* values to the states within one another
- This is therefore the principal characteristic of a programming language with objects; the ability for one object to change another object via assignment

SIMPLE BANK ACCOUNT EXAMPLE

```
(deposit 100)
100

(withdraw 25)
75

(withdraw 25)
50

(withdraw 60)
Insufficient funds

(withdraw 15)
35
```

Each time we call the expression (withdraw 25), we get a different result

This is a different world than our previous examples in all sessions, where an expression would always result in the same output, given a specific set of inputs

THIS AIN'T ROCKET SCIENCE, FOLKS

YOU'VE SEEN ALL OF THIS BEFORE

- This is, again, the danger of learning *how to program* before learning about *programming languages*
- The desire to "get results" leaves us having to walk back to the essentials in order to truly understand the tools we use

IMPLEMENTING WITHDRAW

```
(define balance 100)

(define (withdraw amount)
  (if (≥ balance amount)
      (begin (set! balance (- balance amount))
             balance)
      "Insufficient funds"))
```

- Side note: *begin* causes a series of expressions to be evaluated in sequence, and the value of the final expression is returned as the value of the entire *begin* formation

PROBLEMS WITH WITHDRAW

```
(define balance 100)

(define (withdraw amount)
  if ( ⩾ balance amount)
  (begin (set! balance (- balance amount))
         balance)
"Insufficient funds"))
```

This is accessible from any procedure

SAFER WITHDRAW

```
(define withdraw
  (let ((balance 100))
    (lambda (amount)
      (if (≥ balance amount)
          (begin (set! balance
                   (- balance amount))
                 balance)
          "Insufficient funds"))))
```

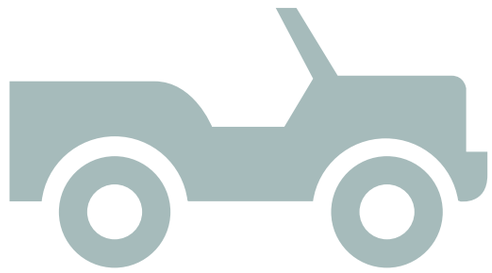
THE BENEFITS OF ASSIGNMENT

- By adding assignment to the programming language, we do introduce some new difficulty
- However, we are no longer able to use the substitution model to evaluate our programs
- Also, we no longer have the ability to predict outputs of any given procedure

THE PITFALLS OF ASSIGNMENT

- In substitution, the notion of a symbol (variable) in our language was a name for a value
- With assignment, that value can now change
- The symbol is now referring to a place where a value can be stored, instead of the actual value

VALUE TYPES VS REFERENCE TYPES



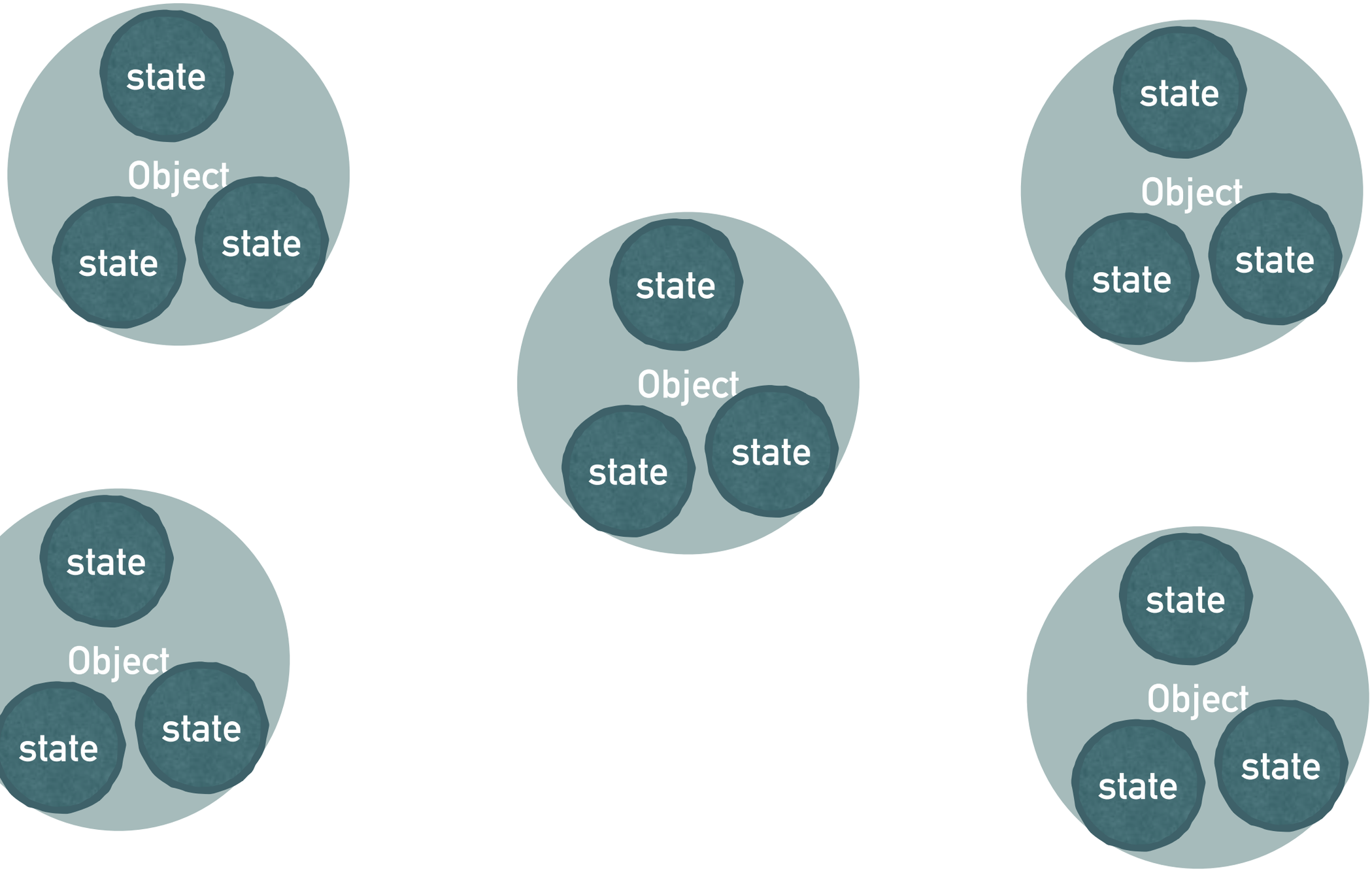
(set-speedometer 20)

(assign-speedometer bicycle)

(set-speedometer 100)

(assign-speedometer jeep)

AN OBJECT SYSTEM



OBJECT SYSTEMS

- An object system is at its most powerful when objects have *loose coupling* and *tight cohesion*
  - Cohesion: the degree to which all of the information about an object is (essentially) contained within that single object, and not within other objects outside itself
  - Coupling: the degree to which independent objects within a system are dependent upon each other in order to complete their work
  - Loose: not implemented extensively
  - Tight: implemented extensively

THE COUNTER-STATE

- Sometimes it's helpful to define a set of rules by what they are *not*
- If one had *tight coupling*, then each entity directly interacts with other entities in very specific ways, and knows a lot about the other objects
  - Example: a system of grading might link directly into the financial aid system to determine continued eligibility
- If one had *loose cohesion*, the information about an entity might be spread across multiple systems of objects
  - Example: a system might have records of a student's birthdate in one location, and grade records in another, and addresses in a third

- It's easy enough to see that these aren't ideal states, and yet we're discussing them. Why?
- These are very real states for systems to get into
  - Most programs *evolve* and change over the years by accretion
    - Sub-optimal decisions get made because programmers are people, and people make poor choices
- Being able to *identify* these problems and put a *name* to them is the first step in correcting them

TIGHT COHESION AND LOOSE COUPLING

- Under tight coupling and loose cohesion, all relevant state about an object is contained within that object
- Under loose cohesion, no object knows too much about other objects and how they work

HOMEWORK

HOMEWORK 11

- Proof of state (and the hazards thereof)

- Define a procedure f where:
  - (+ (f 0) (f 1))
  - returns 0 if the arguments are evaluated left-to right, but 1 if the arguments are evaluated right-to-left

- Use the simplest possible implementation— do not make this the most complex version possible