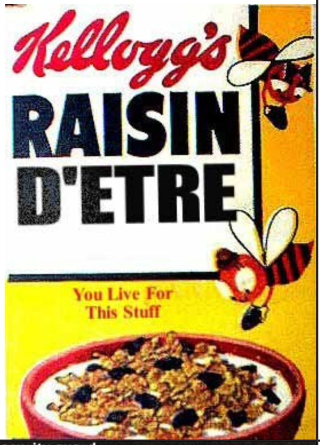
Objectifying and Programming with Objects

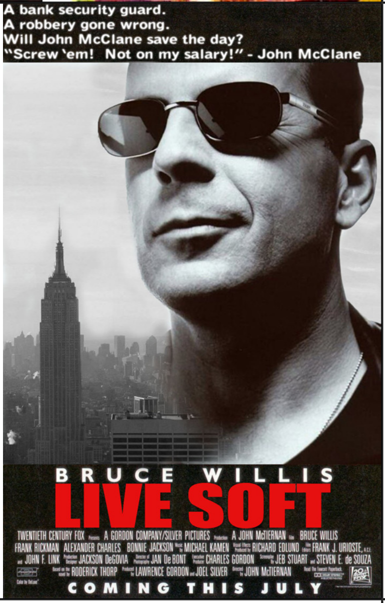


One-Slide Summary

- Real databases, unlike PS5, have many concerns, such as scalability and atomic transactions.
- An **object** packages state and procedures.
- A procedure on an object is called a **method**. We **invoke** a method by sending the object a **message**.
- **Inheritance** allows one object to refine and reuse the behavior of another. This is a good thing.

Outline

- PS5 vs. the Real World
- Problem Sets and PS9
- An Better “Counter”
- Object-Oriented Programming
  - Object = State + Methods
- Inheritance



Interlude: PS5 vs. Wild

How are commercial databases different from what you implemented for PS5?

UVa’s Integrated Systems Project to convert all University information systems to use an Oracle database was originally budgeted for **$58.2 Million** (starting in 1999). Actual cost ended up over $100 Million.

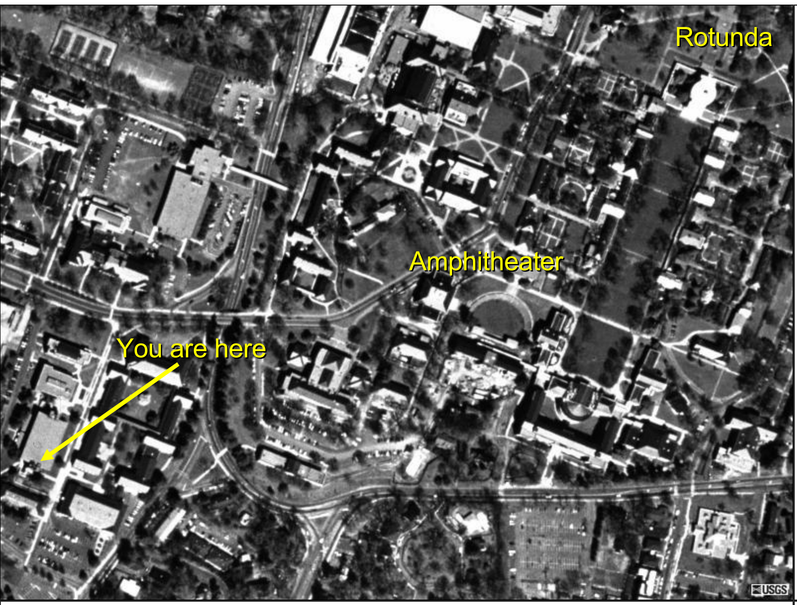
http://www.virginia.edu/isp/

Real Databases

- Atomic Transactions: a transaction may involve many modifications to database tables, but the changes should only happen if the whole transaction happens (e.g., don’t charge the credit card unless the order is sent to the shipping dept)
- Security: limit read/write access to tables, entries and fields
- Storage: need to efficiently store data on disk, provide backup mechanisms
- Scale: to support really big data tables, real databases do lots of clever things

How big are big databases?

- Microsoft TerraServer
  - Claimed biggest in 1998
  - Aerial photos of entire US (1 meter resolution)
  - Let's see an example ...



Big Databases

- Microsoft TerraServer
  - 3.3 Terabytes (claimed biggest in 1998)
  - 1 Terabyte = 2^{40} Bytes ~ 1 Trillion Bytes
- Google Maps (possibly bigger?)
  - Better color ...
- Wal-Mart
  - 285 Terabytes (2003)

- Stanford Linear Accelerator (BaBar)
  - 500 Terabytes (30 KB per particle collision)

How much work?

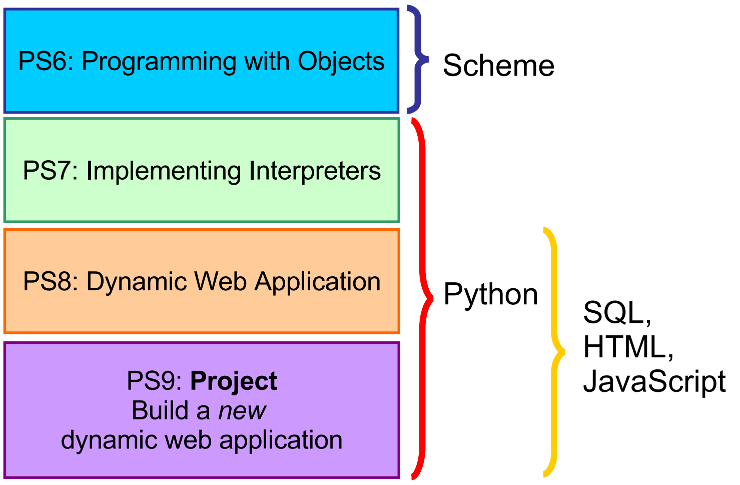
- Suppose we have a huge database.
- table-select is in $\Theta(n)$ where n is the number of entries in the table
  - Would your table-select work for Wal-Mart?
  - If 1M entry table takes 1s, how long would it take Wal-Mart to select from 285TB ~ 2 Trillion Entries?

How much work?

- table-select is in $\Theta(n)$ where n is the number of entries in the table
  - Would your table-select work for Wal-Mart?
  - If 1M entry table takes 1s, how long would it take Wal-Mart to select from 285TB ~ 2 Trillion Entries? 2 000 000s = ~ 23 days

How do expensive databases perform table-select so much faster?
Hint: How did we make sorting faster?

Problem Sets after PS5



PS6: Programming with Objects

PS7: Implementing Interpreters

PS8: Dynamic Web Application

PS9: **Project**
Build a *new*
dynamic web application

Scheme

Python

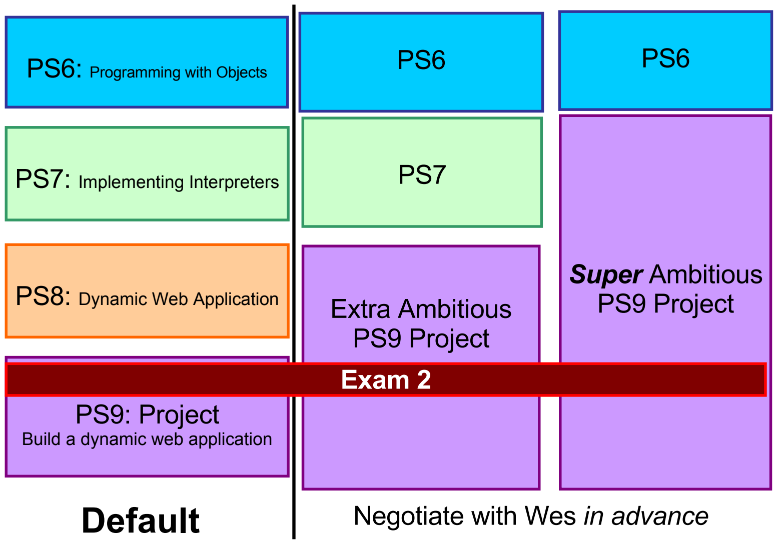
SQL,
HTML,
JavaScript

PS9 Assignment

Problem: Make an interesting dynamic web site.

- Teams of 1-78 students
- Can be anything you want that:
  - Involves interesting computation
  - Follows University's use policies (or on external server)
  - Complies with ADA Section 508 (accessible)

A list of example topics is provided.



PS6: Programming with Objects

PS7: Implementing Interpreters

PS8: Dynamic Web Application

PS9: Project
Build a dynamic web application

Default

PS6

PS7

Extra Ambitious
PS9 Project

PS6

Super Ambitious
PS9 Project

Exam 2

Negotiate with Wes *in advance*

Liberal Arts Trivia: Biology

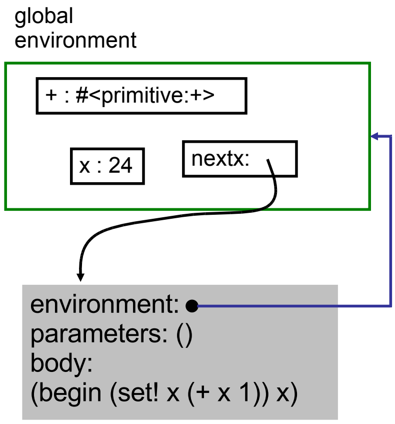
- This egg-laying, venomous (from a calcaneus spur found on the hind limb), beaver-tailed, otter-footed mammal is perhaps best known for its “nose”, which follows the style of the Anatidae family of birds. It is native to eastern Australia and Tasmania, and occurs on the Australian 20 cent coin.

Liberal Arts Trivia: Art History

- Name the Spanish surrealist artist who painted *The Persistence of Memory* (oil on canvas, 1931).

Recall from before: nextx

```
(define x 0)
(define (nextx)
  (set! x (+ x 1))
  x)
> (nextx)
1
> (set! x 23)
> (next x)
24
```



global environment

+ : #<primitive:+>

x : 24

nextx:

environment: ●
parameters: ()
body:
(begin (set! x (+ x 1)) x)

J. Random Luser can come along and change our counter! This is bad.

A Better Counter

- The place that keeps track of the count should be *part of the counter*, not part of the global environment
  - Can have more than one counter
  - Counter state is ***encapsulated***: can only be modified by counter procedure
- Can we do this?

Recall: Application Rule 2:

1. Construct a new environment, whose parent is the environment to which the environment pointer of the applied procedure points.
2. Create a place in that frame for each parameter containing the value of the corresponding operand expression.
3. Evaluate the body in the new environment. Result is the value of the application.

A Better Counter

```
(define (make-counter)
  ((lambda (count)
     (lambda ()
       (set! count (+ 1 count))
       count))
   0))
```

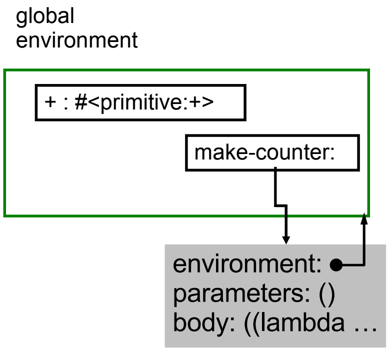
Very slick! We "make our own" zero to start off the counter (purple text).

Sweeter Version

```
(define (make-counter)
  (let ((count 0))
    (lambda ()
      (set! count (+ 1 count))
      count)))
```

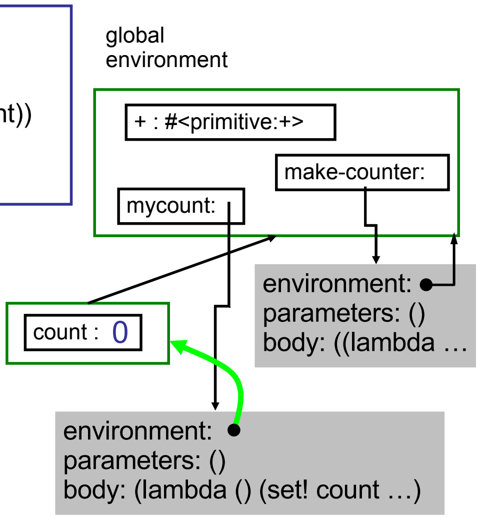
This is easier to read (syntactic sugar), but means the same thing. The place for count is created because of the **let** on this slide and the application on the previous slide mean the same thing.

```
(define (make-counter)
  ((lambda (count)
     (lambda ()
       (set! count (+ 1 count))
       count))
   0))
```



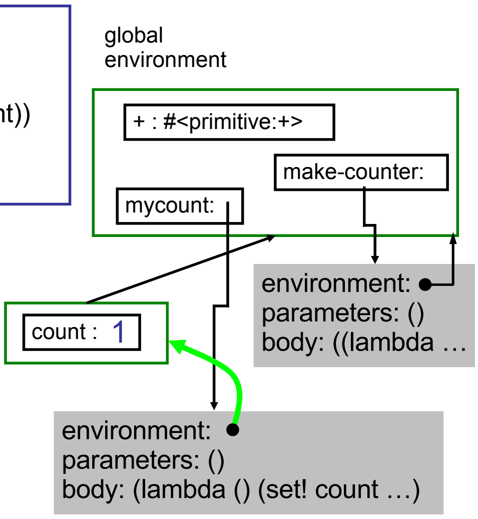
```
(define (make-counter)
  ((lambda (count)
     (lambda ()
       (set! count (+ 1 count))
       count))
   0))
```

```
> (define mycount
    (make-counter))
```



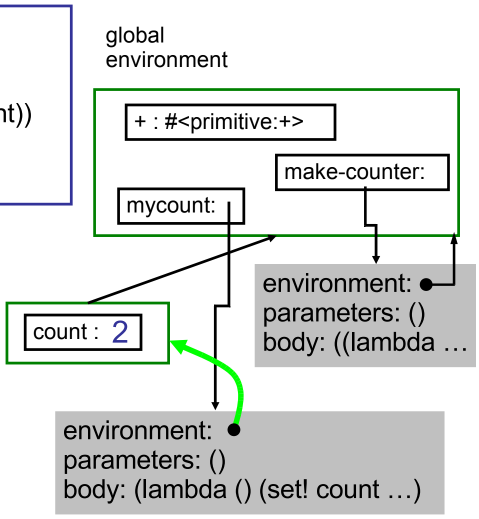
```
(define (make-counter)
  ((lambda (count)
     (lambda ()
       (set! count (+ 1 count))
       count))
   0))
```

```
> (define mycount
    (make-counter))
> (mycount)
1
```



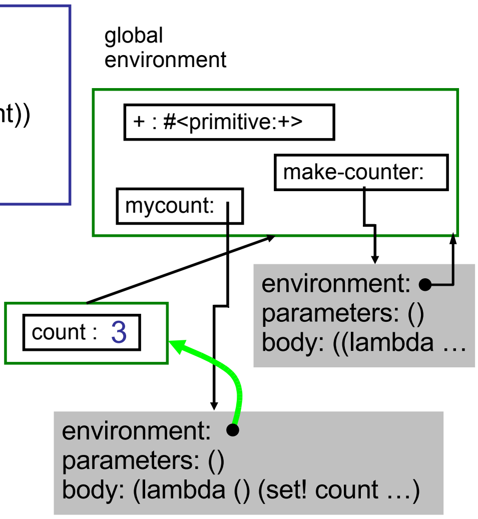
```
(define (make-counter)
  ((lambda (count)
     (lambda ()
       (set! count (+ 1 count))
       count))
   0))
```

```
> (define mycount
    (make-counter))
> (mycount)
1
> (mycount)
2
```



```
(define (make-counter)
   ((lambda (count)
      (lambda ()
        (set! count (+ 1 count))
        count))
      0))
```

```
> (define mycount
     (make-counter))
> (mycount)
1
> (mycount)
2
> (mycount)
3
```



global environment

+ : #<primitive:+>

make-counter:

mycount:

environment:
parameters: ()
body: ((lambda …

count : 3

environment:
parameters: ()
body: (lambda () (set! count …)

An Even Better Counter

```
(define (make-counter)
  (let ((count 0))
    (lambda (message)
      (cond ((eq? message 'reset!)
                  (set! count 0))
               ((eq? message 'next!)
                  (set! count (+ 1 count)))
               ((eq? message 'current) count)
               (else
                  (error "Unrecognized message"))))))
```

In **object-oriented programming**, state is encapsulated with methods that operate on that state. Methods are invoked by sending **messages**.

In Scheme, the **single quote** (as in 'current) just means "I am making up a symbol or a message name." See the textbook for more info.

Using Counter

```
> (define oocounter (make-counter))
> (oocounter 'next)
> (oocounter 'next)
> (oocounter 'next)
> (oocounter 'how-many)
3
> (oocounter 'reset)
> (oocounter 'how-many)
0
```

Objects

An ***object*** packages:

- **state** ("instance variables")
- **procedures** for manipulating and observing that state ("methods")

Why is this useful?

Problem-Solving Strategies

- PS1-PS4: **Functional Programming**
  - Focused on **procedures**
  - Break a problem into procedures that can be combined to solve it
- PS5: **Imperative Programming**
  - Focused on **data**
  - Design data for representing a problem and procedures for updating that data

Problem-Solving Strategies

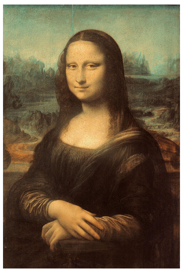
- PS6: **Object-Oriented Programming**
  - Focused on **objects**: package procedures and state
  - Model a problem by dividing it into objects
  - Lots of problems in real (and imaginary) worlds can be thought of this way

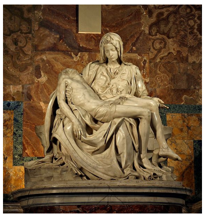
Liberal Arts Trivia: Art History and American Literature

- Give the Renaissance master (or Ninja Turtle) associated with each work of art:

(a) Tomb of Antipope John XXIII

(b) Mona Lisa





(c) Pieta

(d) Transfiguration

Liberal Arts Trivia: Cooking

- This Japanese delicacy is vinegared rice, usually topped with other ingredients, including fish. The dish as we know it today was invented as a fast food by Hanaya Yohei at the end of the Edo period (19th century) in Tokyo: it could be eaten on the road side or in a theatre using fingers or chopsticks. The basic idea can be traced back to 4th century BCE China as a preservative: the fermentation of the rice prevents the fish from spoiling.

Counter Object

```
(define (make-counter)
  (let ((count 0))          Instance variable
    (lambda (message)
      (cond ((eq? message ’reset!)        Methods
              (set! count 0))
             ((eq? message ’next!)
              (set! count (+ 1 count)))
             ((eq? message ’current) count)
             (else
               (error "Unrecognized message"))))))
```

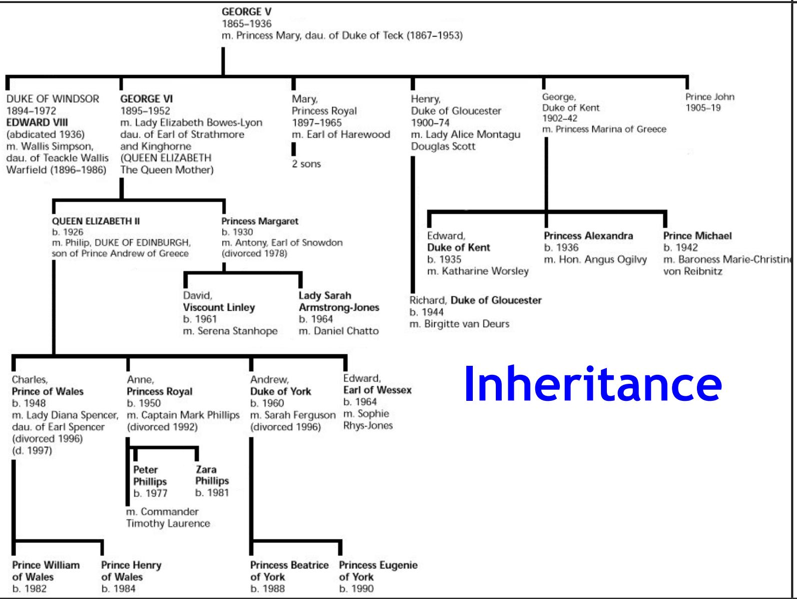
Defining **ask**

(ask *Object Method*)

```
> (define oocounter (make-counter))
> (ask oocounter 'current)
0
> (ask oocounter 'next)
> (ask oocounter 'current)
1
```

```
(define (ask object message)
   (object message))
```

Inheritance



The Truth About Dogs and Dogs

- There are many types of dogs out there, but most of them behave similarly.

```
(define make-dog
  (lambda (message)
    (cond ((eq? message 'speak) “woof”)
              ((eq? message 'solve-mystery)
              “<dogs cannot solve mysteries!>”)
)))
```

make-scooby

```
(define make-scooby
  (lambda (superclass) ;; normal dog
    (lambda (message)
      (cond
       ((eq? message 'solve-mystery)
        “Scooby solves the mystery!”)
       ((eq? message 'snack)
        “Scooby snacks!”))
       (else (ask superclass message)
))))
```

Can You Solve The Mystery?

```
(define make-dog
 (lambda (message)
  (cond ((eq? message 'speak)
   “woof”)
  ((eq? message 'solve-mystery)
  “<dogs cannot solve mysteries!>”)
)))
```

```
> (define lassie (make-dog))
> (define scooby-doo (make-scooby
 (make-dog))
> (ask lassie 'speak)
???
> (ask lassie 'solve-mystery)
???
> (ask lassie 'snack)
???
```

```
(define make-scooby
 (lambda (superclass)
  (lambda (message)
   (cond
    ((eq? message 'solve-mystery)
     “Scooby solves the mystery!”)
    ((eq? message 'snack)
     “Scooby snacks!”))
    (else (ask superclass message)
))))
```

Can You Solve The Mystery?

```
(define make-dog
 (lambda (message)
  (cond ((eq? message 'speak)
   “woof”)
  ((eq? message 'solve-mystery)
  “<dogs cannot solve mysteries!>”)
)))
```

```
> (define lassie (make-dog))
> (define scooby-doo (make-scooby
 (make-dog))
> (ask lassie 'speak)
“woof”
> (ask lassie 'solve-mystery)
“<dogs cannot solve mystery!>”
> (ask lassie 'snack)
;; nothing
```

```
(define make-scooby
 (lambda (superclass)
  (lambda (message)
   (cond
    ((eq? message 'solve-mystery)
     “Scooby solves the mystery!”)
    ((eq? message 'snack)
     “Scooby snacks!”))
    (else (ask superclass message)
))))
```

Can You Solve The Mystery?

```
(define make-dog
 (lambda (message)
  (cond ((eq? message 'speak)
   “woof”)
  ((eq? message 'solve-mystery)
  “<dogs cannot solve mysteries!>”)
)))
```

```
> (define lassie (make-dog))
> (define scooby-doo (make-scooby
 (make-dog))
> (ask scooby-doo 'speak)
???
> (ask scooby-doo 'solve-mystery)
???
> (ask scooby-doo 'snack)
???
```

```
(define make-scooby
 (lambda (superclass)
  (lambda (message)
   (cond
    ((eq? message 'solve-mystery)
     “Scooby solves the mystery!”)
    ((eq? message 'snack)
     “Scooby snacks!”))
    (else (ask superclass message)
))))
```

Can You Solve The Mystery?

```
(define make-dog
 (lambda (message)
  (cond ((eq? message 'speak)
   “woof”)
  ((eq? message 'solve-mystery)
  “<dogs cannot solve mysteries!>”)
)))
```

```
> (define lassie (make-dog))
> (define scooby-doo (make-scooby
 (make-dog))
> (ask scooby-doo 'speak)
“woof”
> (ask scooby-doo 'solve-mystery)
“Scooby solves the mystery!”
> (ask scooby-doo 'snack)
“Scooby snacks!”
```

```
(define make-scooby
 (lambda (superclass)
  (lambda (message)
   (cond
    ((eq? message 'solve-mystery)
     “Scooby solves the mystery!”)
    ((eq? message 'snack)
     “Scooby snacks!”))
    (else (ask superclass message)
))))
```

You're a Mystery Machine!

```
(define make-dog
 (lambda (message)
  (cond ((eq? message 'speak)
   “woof”)
  ((eq? message 'solve-mystery)
  “<dogs cannot solve mysteries!>”)
)))
```

```
> (define lassie (make-dog))
> (define scooby-doo (make-scooby
 (make-dog))
> (ask scooby-doo 'speak)          Inherit behavior ('speak)
“woof”
> (ask scooby-doo 'solve-mystery)Inherit and Override behavior ('solve-mystery)
“Scooby solves the mystery!”
> (ask scooby-doo 'snack)          New behavior ('snack)
“Scooby snacks!”
```

```
(define make-scooby
 (lambda (superclass)
  (lambda (message)
   (cond
    ((eq? message 'solve-mystery)
     “Scooby solves the mystery!”)
    ((eq? message 'snack)
     “Scooby snacks!”))
    (else (ask superclass message)
))))
```

Object-Oriented Terminology

- An **object** is an entity that packages state and procedures.
- The state variables that are part of an object are called **instance variables**.
- The procedures that are part of an object are called **methods**.
- We **invoke** (call) a method by sending the object a **message**.
- A **constructor** is a procedure that creates new objects (e.g., make-*dog*).

Charge

- Start PS6 early
  - You can turn in PS5 up to Friday (popular demand), but the clock is ticking for PS6!
  - PS6 is challenging
  - Opportunity for creativity
- Start thinking about PS9 Project ideas
  - If you want to do an “extra ambitious” project convince me your idea is worthy before March 26 (ps7 and 8)/April 4 (ps8)
  - Discuss ideas and look for partners **on the forum**

Homework

- PS 5 due *Friday*
  - Extension granted.
- PS 6 due Monday March 23rd
- Read GEB Chapters 2-4 and 6-9