CS193j, Stanford
Winter, 2001-02

Handout #4
Nick Parlante

Java 1

Today we'll start looking at the basics of OOP and Java syntax.

Procedural vs. OOP

Nouns and Verbs

Nouns -- data
Verbs -- operations

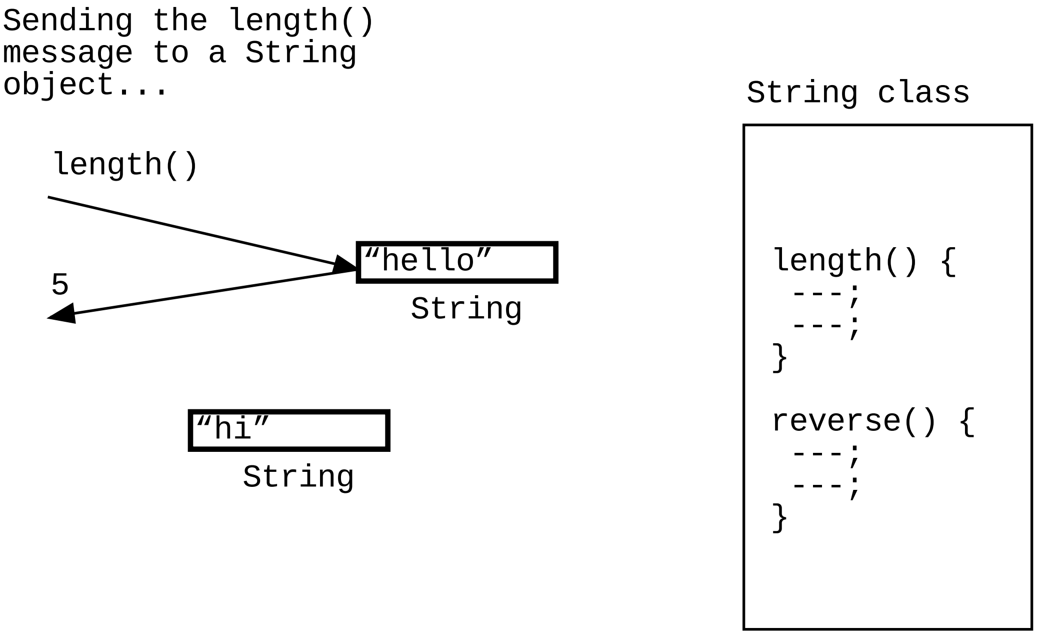
Procedural Structure

C / Pascal / etc. ...
Verb oriented
    decomposition around the verbs -- dividing the big operation into a series of smaller and smaller operations.
Nouns / Verb structure is not formal
    The programmer can group the verbs and nouns together (ADTs), but it's just a convention and the compiler does not especially help out.

OOP Structure

Objects

State
    Stores state, like a regular variable
Class
    Every object belongs to a class that defines its state and behavior.
    An object always remembers its class (in Java).
Instance
    Another, more common word for "object".
Anthropomorphic -- self-contained
    Procedural variables are passive -- they just sit there. A procedure is called and it changes the variable.
    Objects are anthropomorphic-- the object has some state, and the object controls and changes its own state.
String example
    Could have a string object that stores a sequence of characters. The object belongs to the String class. The String class defines how string objects work.


Sending the length() message to a String object...

Class

Exists once -- there is one copy of the class in memory.
Contains definitions for its objects
Storage
    Define the storage that objects of this class will have.
    "instance variables" -- the variables that each object will use for its own storage
Behavior
    Define the behaviors that objects of this class will be able to execute (methods).
String example
    The String class defines the storage structure used by all String objects
    The String class also defines the operations that String objects can perform (methods, below)

Message / Receiver

Sent to an object -- Request -- "getUnits()"
a.getUnits() -- send the "getUnits()" message to the receiver "a"
Receiver
    The object receiving the message.
    obj.units = 15;   NO
    The receiver should operate on its state, not the client
    obj.setUnits(15)   YES
    Sending a message to the receiver object, maps to a method (below), that method is code that actually changes the receiver state.
String example
    The string object might respond to messages like "length()" and "reverse()" which operate on the receiver they are sent to.

Method (code)

A "method" is code that defined in a class

The methods in a class are available to all the objects of that class

The objects of a class can execute all the methods the class defines

String example

So the String class will contain length() and reverse() method code that implements those operations.

Message -> Method resolution

Suppose a message is sent to an object --- string.reverse();

1. The receiver is of some class -- suppose the object a is of the String class
2. Look in that class for a matching reverse() method (code)
3. Execute that code "against" the receiver -- using its memory (instance variables)

In Java this is "dynamic" -- the process uses the true, run-time class of the receiver.

OOP Design - Anthropomorphic

1. Objects responsible for their own state
2. Objects can send messages to each other -- requests
3. the object/message paradigm makes the program more modular internally. Each class deals with its own implementation details, but can be largely independent of the details of the other classes. They just exchange messages.

OOP Part 1 -- Encapsulation

Objects "protect" their own state from direct access by other objects. Other objects can send requests, but only the receiver actually changes its own state. This allows more reliable software -- once a class is correct and debugged, putting it in a new context should not create new bugs in the class.

Abstraction vs. Implementation

This is the old Abstract Data Type (ADT) style of separating the abstraction from the implementation, but re-cast as messages (abstraction) vs. methods (implementation)

OOP Design Process

Think about the objects that make up an application

Think about the behaviors or capabilities those objects should have

Endow the objects with those abilities as methods

Co-operation

Objects sent each other messages to co-operate

But each method operates on its own receiver

Tidy style

Experience shows that having each object operate on its own state is a pretty intuitive and modular way to organize things

the tutorial.

Student Java Example

Student Example

For this example, we'll look at a simple Student class. Each student object has an integer number of units and responds to messages like getUnits() and getStress(). The stress of a student is defined to be units * 10.

Java Client Side

First we'll look at Java code as the client of a class -- creating objects and sending them messages

Allocate objects with "new" -- calls constructor

Objects are always accessed through pointers -- shallow, pointer semantics

Send messages -- methods execute against the receiver

Can access public, but not private/protected from client side

Object Pointers

Student x;

Declares a pointer "x" to a Student object, but does not allocate the object yet.

new Student()

The "new" operator allocates a new object in the heap and returns a pointer to it.

Constructors

Classes define "constructors" that initialize objects at the time new is called.

The word "constructor" can be abbreviated as "ctor"

The constructor uses the same name as the class. e.g. the constructor for the "Student" class uses the name "Student"

There can be multiple constructors. They are distinguished by having different arguments -- this is called "overloading".

e.g. The Student class defines one ctor that takes an int argument, and one ctor that takes no arguments.

The ctor that takes no arguments is called the "default ctor", and it is used in some cases by default when no other ctor is specified.

Message send

Send messages to an object..

```
    a.getUnits();
    b.getStress();
```

Finds the matching method in the class of the receiver, executes that method against the receiver and returns.

Student Client Side Code

```
// Make two students
Student a = new Student(12);  // new 12 unit student
Student b = new Student();    // new 15 unit student

// They respond to getUnits() and getStress()
System.out.println("a units:" + a.getUnits() +
   " stress:" + a.getStress());

System.out.println("b units:" + b.getUnits() +
   " stress:" + b.getStress());


a.dropClass(3);   // a drops a class


System.out.println("a units:" + a.getUnits() +
   " stress:" + a.getStress());


// Now "b" points to the same object as "a"
b = a;
b.setUnits(10);


// So the "a" units have been changed
System.out.println("a units:" + a.getUnits() +
   " stress:" + a.getStress());


// NOTE: public vs. private
// A statement like "b.units = 10;" will not compile in a client
// of the Student class when units is declared protected or private

/*
 OUTPUT...
   a units:12 stress:120
   b units:15 stress:150
   a units:9 stress:90
   a units:10 stress:100
*/
```

Student Implementation Side

Now we'll look at the definition of the Student class -- in the file Student.java

Instance Variables

```
   protected int units;
```

"ivars"

Defines the variables that each object of this class will have

protected / private = not accessible to client code (will not compile)

public = accessible to client code

Constructor (ctor)

```
   public Student(int initUnits) {
    units = initUnits;
   }
```

New objects are set to all 0's first, then the ctor (if any) is run to further initialize the object.

Can have multiple ctors, distinguished by different arguments (overloading)

Ctor with no args is known as the "default ctor"

If a class has constructors, the compiler will insist that one of them is invoked when new is called.

If a class has no ctors, new objects will just have the default "all 0's" state. A class that is at all complex should have a ctor.

Bug control

- Make it easier for the client to do the right thing since objects are always put into an initialized state when created.

Every ivar goes in Ctor

- Every time you add an instance variable to a class, go add the line to the ctor that inits that variable.
- Or you can give an initial value to the ivar right where it is declared, like this... int units = 0; -- there is not currently agreement about which ivar init style is better.

Method

```
   public int getStress() {
    return(units * 10);
   }
```

Code stored in class

This code will execute against instances of the class

Message-Method Lookup

- Message sent to a receiver
- Receiver looks in its class for matching method
- That method executes against the receiver

Receiver Relative (Method, Ctor)

The code runs "on" or "against" the receiving object

Any ivar read / write operations happen to the receiver

Method code is written with a "receiver relative" style

e.g. "units" instance variables automatically that of the receiver

Self message send

- "setUnits(units - drop);" -- easy to send a message keeping the same receiver

"this" -- receiver

"this" in a method

- "this" is a pointer to the receiver
- Don't write "this.units", write: "units"
- Don't write "this.setUnits(5)", write "setUnits(5);"

ivar vs. local var

- Usually, just refer to the ivar by name directly. Sometimes you have a local var with the same name as the ivar, in which case the expression this.ivar refers to the ivar. Having a local var with the same name as an ivar is a stylistically questionable, but it can be handy sometimes. Some people prefer to give ivars a name always starting with "m" -- mUnits, etc.

Receiver / Noun Style

- You think a little differently about your code -- code is grouped around the noun it operates on

"private"

Implementation visibility

- Essentially, only code that is implementing the class can access the method or ivar.

Ivars

Methods

Ctors

"Sibling Access"

- Private does not prevent one object in the class access the state of **another** object in the class. Such access is slightly less desirable OOP style, but the private keyword does not guard against it.

"public"

Visible to all

"Official" class interface

aka the interface the class "exposes" for other classes to use.

Public = supported

- public features will not be removed in a future rev
- Other classes can depend on these features
- Sun deprecates some public features, so new code won't be written with them, but it very rarely removes a formerly public feature
- private things can be removed from an implementation at will

"protected"

Similar to "private" but allows access to subclasses

Student.java Code Example

```
// Student.java
/*
 Demonstrates the most basic features of a class.

 A student is defined by their current number of units.
 There are standard get/set accessors for units.

 The student responds to getStress() to report
 their current stress level which is a function
 of their units.

 NOTE A well documented class should include an introductory
 comment like this. Don't get into all the details -- just
 introduce the landscape.
*/
public class Student extends Object {
   // NOTE this is an "instance variable" named "units"
   // Every Student object will have its own units variable.
   // "protected" and "private" mean that clients do not get access
   protected int units;


   /* NOTE
    "public static final" declares a public readable constant that
    is associated with the class -- it's full name is Student.MAX_UNITS.
    It's a convention to put constants like that in upper case.
   */
   public static final int MAX_UNITS = 20;
   public static final int DEFAULT_UNITS = 15;


   // Constructor for a new student
   public Student(int initUnits) {
      units = initUnits;
      // NOTE this is example of "Receiver Relative" coding --
      // "units" refers to the ivar of the receiver.
      // OOP code is written relative to an implicitly present receiver.
   }

   // Constructor that that uses a default value of 15 units
   // instead of taking an argument.
   public Student() {
      units = DEFAULT_UNITS;
   }


   // Standard accessors for units
   public int getUnits() {
      return(units);
   }
```

```
public void setUnits(int units) {
   if ((units < 0) || (units > MAX_UNITS)) {
      return;
      // Could use a number of strategies here: throw an
      // exception, print to stderr, return false
   }
   this.units = units;
   // NOTE: "this" trick to allow param and ivar to use same name
}


/*
 Stress is units *10.

 NOTE another example of "Receiver Relative" coding
*/
public int getStress() {
   return(units*10);
}


/*
 Tries to drop the given number of units.
 Does not drop if would go below 9 units.
 Returns true if the drop succeeds.
*/
public boolean dropClass(int drop) {
   if (units-drop >= 9) {
      setUnits(units - drop);    // NOTE send self a message
      return(true);
   }
   return(false);
}

/*
 Here's a static test function with some simple
 client-of-Student code.
 NOTE Invoking "java Student" from the command line runs this.
 It's handy to put test/demo/sample client code in the main() of a class.
*/
public static void main(String[] args) {
   // Make two students
   Student a = new Student(12);  // new 12 unit student
   Student b = new Student();    // new 15 unit student

   // They respond to getUnits() and getStress()
   System.out.println("a units:" + a.getUnits() +
      " stress:" + a.getStress());

   System.out.println("b units:" + b.getUnits() +
      " stress:" + b.getStress());


   a.dropClass(3);   // a drops a class


   System.out.println("a units:" + a.getUnits() +
      " stress:" + a.getStress());
      // Now "b" points to the same object as "a"
      b = a;
      b.setUnits(10);


      // So the "a" units have been changed
      System.out.println("a units:" + a.getUnits() +
         " stress:" + a.getStress());


      // NOTE: public vs. private
      // A statement like "b.units = 10;" will not compile in a client
      // of the Student class when units is declared protected or private

      /*
       OUTPUT...
         a units:12 stress:120
         b units:15 stress:150
         a units:9 stress:90
         a units:10 stress:100
      */
   }
}

/*
 Things to notice...

 -Demonstrates the Object-lifecycle -- clients create the object (must go
 through constructor), then send it messages. Hard for the client to mess
 up the state of the object. Note how setUnits() can maintain the internal
 correctness of the object.

 -The implementation code can refer to instance variables like "units"
 by name. It automatically binds to the ivar of the receiver.

 -"units" is declared protected. Thereore, a client cannot write something like
 "a.units++;". The client must go through public messages like setUnits().
 This promotes a less fragile design. The client may access things declared
 "public".

 -State vs. Computation -- notice that the client can't really tell if stress is
 stored or computed. It just appears to be a property that Students have. Whether
 it is stored or computed is just a detail. This is a nice separation between
 the abstraction exposed by client and how it is actually implemented.
*/
```