Java 3

Time Example

Suppose you have a Time class that represents a time, such as "10:53 am"
(There's a simple Time.java among the starter files for hw1)
(See also the Date java library class)

Operations

What operations might the Time class expose?
Theme: expose things for the convenience of the client. Hide implementation details.

- -getHours() / getMinutes() / isAM() -- standard accessors
- -setMinutes() / setHours() / setAM() -- these may need to "renormalize" the data from the client, e.g. minutes to the range 0..59, to maintain the internal correctness of the time object and its assumptions. This is an advantage of making the client go through setter methods -- the object can control its state.
- -isBefore(Time) -- compare to another time: is the receiver before or equal to
- -shift(int hours, int minutes) -- shift the receiver by the given hours and minutes. (internally handles messy wrap-around logic for hours and minutes, midnight)

Implementation vs. Interface

Could implement as: int hours, int minutes, boolean am / pm
Could implement as: int minutesSinceMidnight

- Give the illusion to the client of hours / minutes, but do internal logic just in terms of minutes

The abstraction should be optimized to be convenient and understandable to the client. The implementation should be optimized for easy implementation.
The hours / minutes representation may be most convenient for the client, but it is not an easy represenation for computations.

OOP Part 2 — Inheritance

Arrange several related classes in a way to avoid duplication / promote code re-use. (See the OOP Concepts handout)
OOP Hierarchy
Superclass / Subclass
Inheritance
Overriding
ISA -- the subclass ISA instance of the superclass -- it has **all** the properties that instances of the superclass are supposed to have (and it has some additional properties as well)

Inheritance Warning

Inheritance is a neat and appealing technology.
However, it is only usable in somewhat rare circumstances -- where you have several very similar classes.
It is a common error for beginning OOP programmers to try to use inheritance for everything.
Modularity may be less flashy, but it is incredibly common. Inheritance is rare, but where it works, nothing else will do.

Horse / Zebra Style

Suppose you have a hierarhcy of animals, except the zebra was omitted and you have been asked to add it in.
No: define the zebra from scratch
Yes: locate the Horse class. Introduce Zebra as a subclass of Horse
    Zebra inherits 90% of its behavior (no coding required)
    In the Zebra class, define the few things that are features of Zebras but not Horses

Grad Variation

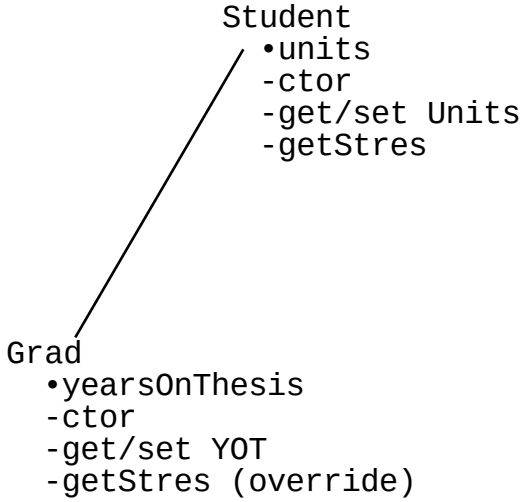
Suppose we want to add a Grad class based on the Student class -- Grad students are like students, but with two differences...
    * Years on thesis -- a grad has a count of the number of years worked on thesis
    - getStress() is different -- Grads are more stressed. Their stress is (2* the Student stress) + yearsOnThesis

Student Inheritance

Student defined by int units
Grad is everything that a Student is + the idea of yearsOnThesis (yot)
"isa" relationship with its superclass -- Grad isa Student
Subclass has **all** the properties of its superclass + a few
Grad overrides getStress() with a specialized version
Grad (subclass) has more properties / is more constrained / more specific
Student (superclass) has fewer properties / is less constrained. / more general

Student/Grad Design Diagram

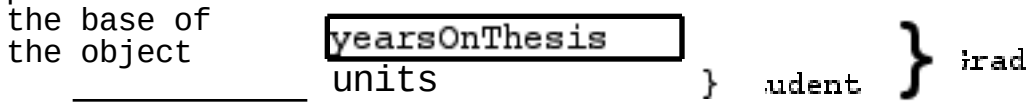
The following is an excellent sort of diagram to make early in the design to think about the division of responsibility between a Superclass and its Subclass.
('•' = instance variable, '-' = method)



Student/Grad Memory Layout

Implementation detail: the ivars of the subclass are layered on top of the ivars of the superclass

Result:: if you have a pointer to the base of an instance of the superclass (Grad), you can treat it as if it were a superclass object (Student) and it just works since the objects look the same from the bottom up.



Simple Inheritance Example

```
Student s = new Student(10);
Grad g = new Grad(10);
s.getStress();          // goes to Student.getStress()
g.getUnits();           // goes to Student.getUnits() -- INHERITANCE
g.getStress();          // goes to Grad.getStress() --OVERRIDING
```

Never Forget Class

In Java, no matter what code is being executed, the receiver is the same receiver (even if the code is in a different class) and the receiver never forgets its class.

EG even getUnits() (Student class) executing on a Grad, remembers that the receiver is a Grad

Semantics of "Student s;"

NO: "s points to a Student object"
YES: "s points to an object that responds to all the messages that Students respond to"
YES: "s points to a Student, or a subclass of Student"

Simple Substitution Example

Subclass can be used in a context which call for the superclass
This works because of the ISA property -- Grad ISA Student
Therefore, a Grad type pointer may be stored in a Student type variable

```
Student s = new Student(10);
Grad g = new Grad(10);
s = g;   // ok -- subclass may be used in place of superclass
// what operations are allowed on s?
```

Compile Time

Because of the substitution rule, the compile time and run time type systems diverge.
The compile time type system is more loose -- not knowing the exact class of the receiver.
e.g., with the following the compiler, knows that "s" points to a Student object or a Grad object

```
void foo(Student s) {
  // s points to Student or Grad -- don't know for sure
```

The compile time type system is used for error checking
Code is only allowed if the compiler can determine with 100% confidence that the receiver does respond to the given message.

Run Time

In Java, the run time type system is exact -- the receiver knows exactly what class it is.
The run time type system is used to resolve message sends (i.e. "message/method resolution").

Substitution Code

```
Student s = new Student(10);
Grad g = new Grad(10);
s = g;   // ok
s.getStress();  // ok -- goes to Grad.getStress() (overriding)
s.getUnits();   // ok -- goes to Student.getUnits (inheritance)
s.getYearsOnThesis();         // NO -- does not compile (s is compile time type Student)
```

Inheritance Client Code

The compiler only allows message sends that are guaranteed to work, and the compiler uses the compile-time types of variables in its estimations. The programmer can put in casts to edit the compile-time types of expressions.

At run-time, the message-method resolution usees the run-time type of the receiver, not the compile-time type -- this is a feature. Languages that use the compile-time type for message resolution are too inflexible.

```
Student s = new Student(10);
Grad g = new Grad(15, 2);
Student x = null;

System.out.println("s " + s.getStress());
System.out.println("g " + g.getStress());

// Note how g responds to everthing s responds to
// with a combination of inheritance and overriding...
g.dropClass(3);
System.out.println("g " + g.getStress());

/*
 OUTPUT...
   s 100
   g 302
   g 242
*/

// s.getYearsOnThesis();   // NO does not compile
g.getYearsOnThesis();   // ok

// Substitution rule -- subclass may play the role of superclass
x = g;   // ok (this is allowed with no cast)

// At runtime, this goes to Grad.getStress()
// Point: message/method resolution uses the RT class of the receiver,
// not the CT class in the source code.
// This is essentially the objects-know-their-class rule at work.
x.getStress();

// g = x;   // NO -- does not compile,
// substitution does not work that direction

// x.getYearsOnThesis();   // NO, does not compile

((Grad)x).getYearsOnThesis(); // work around with cast
// Ok, so long as x really does point to a Grad
```

Insomnia Example

Suppose we have an insomnia() method in the Student class that returns true of the receiver is very stressed.

Question: how does this work if we send the insomnia() message to a Grad object?

```
public boolean insomnia() {
 return(getStress() > 100);
// Pops DOWN to Grad.getStress()
// if the receiver is a Grad
}
```

Client code...

```
Student s = new Student(...);
Grad g = new Grad(....);
s.insomnia();          // does the right thing
g.insomnia();          // does the right thing
```

g.insomnia() Series

Where does the code flow go when sending insomnia() to a Grad object

1. Student.insomnia()
2. Grad.getStress() // pop-down
3. Student.getStress() // the super.getStress() call in Grad.getStress

"Pop-Down" Rule

The receiver knows its class

The flow of control jumps around different classes

No matter where the code is executing, the receiver knows its class and does message->method mapping correctly for each message send.

e.g. Receiver is the subclass (Grad), executing a method up in the superclass (Student), a message send that Grad overrides will "pop-down" to the Grad definition (getStress())

Grad.java

```
// Grad.java

/*
 Grad is a subclass of Student.
 Grad adds the state of yearsOnThesis.

 Grad overrides getStress() to provide a Grad specific version.
*/
public class Grad extends Student {

   private int yearsOnThesis;


   public Grad(int units, int yearsOnThesis) {
      // NOTE "super" must be first if used --
      // chains up to the superclass constructor
      super(units);

      this.yearsOnThesis = yearsOnThesis;
   }


   /*
    Grad stress is based on twice the Student stress
    plus an additional factor for the yearsOnThesis.

    NOTE: avoid code repetition between subclass/superclass
    at all costs -- that's why we use Student.getStress()
    for the core of our computation.
   */
   public int getStress() {
      // NOTE "super" still invokes message/method resolution
      // but it starts the search one class higher up
      // (there is no super.super)
      int student = super.getStress();

      return(student*2 + yearsOnThesis);
   }

   // Standard accessors
   public void setYearsOnThesis(int yearsOnThesis) {
      this.yearsOnThesis = yearsOnThesis;
   }

   public int getYearsOnThesis() {
      return(yearsOnThesis);
   }
```

```
/*
 Things to notice...

 -The ctor takes both Student and Grad state -- the Student state is passed up
 to the Student ctor by the first "super" line in the Grad ctor.

 -getStress() is a classic override. Note that it does not _repeat_ the code
  from Student.getStress(). It calls it using super, and fixes the result.
  The whole point of inheritance is to avoid code repetition.

 -Grad responds to every message that a Student responds to -- either
 a) inherited such as getUnits()
 b) overriden such as getStress()

 -Grad also esponds to things that Students do not,
 such as getYearsOnThesis().
*/
```

Inheritance / Notification Style

Here is an illustration of how all this inheritance stuff is actually used...
Suppose there is a Car class with go(), stop(), and turn() methods
Suppose there is an existing system of code that sends "notifications" to the car over time: go(), stop(), go(), ...
You want to create your own car, but that turns differently...
    Subclass off Car
    Override the turn() method with your own definition
The existing system / car relationship with the methods go(), stop(), etc. continues to work
A turn() notification will pop down to use your turn(), and then pop back up and continue using the standard car code

Notification Style Results

Use this style as a way of integrating your code with library code -- subclass off a library class, 90% inherit the standard behavior, and 10% override a few key methods.
e.g. Servlets -- inherit the standard HTTP Servlet behavior and define custom behavior in a few key method overrides.