Instructors and Salesmen

Written and polished by Andy Gray **last** autumn quarter. Thanks, Andy!

In the first part of today's section, we'll take a closer look at inheritance in some of its more complicated forms. For an example of a extensive class hierarchy, you can see the entire Java hierarchy as well as all the methods for each of the built-in classes at:

http://java.sun.com/products/jdk/1.1/docs/api/packages.html

Documentation on all of the Java built-in classes can be found here, so keep a bookmark handy while you're programming in Java.

The second part of section will cover concurrency and threading in Java.

Inheritance: Instructor Example

(This problem is modified from an old CS107 final exam.) For this problem, you will design Java classes suitable for storing information about university instructors. The goal of the example is to demonstrate arranging classes in a hierarchy for maximum code-sharing. There are three types of instructor: Faculty, Lecturer, and Grad student. At any time, an instructor can best be described by three quantities: number of unread e-mail messages, age, and number of eccentricities. An instructor should be initialized with their age (and by setting them to have no unread mail and no eccentricities).
There are two measures of an instructor's current mood: stress and respect.

stress: an instructor's stress level is the number of unread messages. However, stress is never more than 1000. Grad students are the exception. Their stress is double the number of unread messages and their maximum stress is 2000.

respect: generally an instructor's level of respect in the community is their age minus the number of eccentricities. Respect can never be negative. Faculty are the exception— faculty eccentricities are regarded as "signs of a troubled genius" thereby increasing respect. So for faculty respect goes up with eccentricities. On the other hand, faculty like to think of their personal ivory towers as being as selective as possible. Therefore, a faculty member's level of respect is decreased by the *total number of existing faculty members* . Thus, the computed level of respect for faculty is age *plus* number of eccentricities *minus* the total number of existing faculty.

Everything in an instructor's life is driven by receiving e-mail. Several things happen when an instructor gets new, unread e-mail:

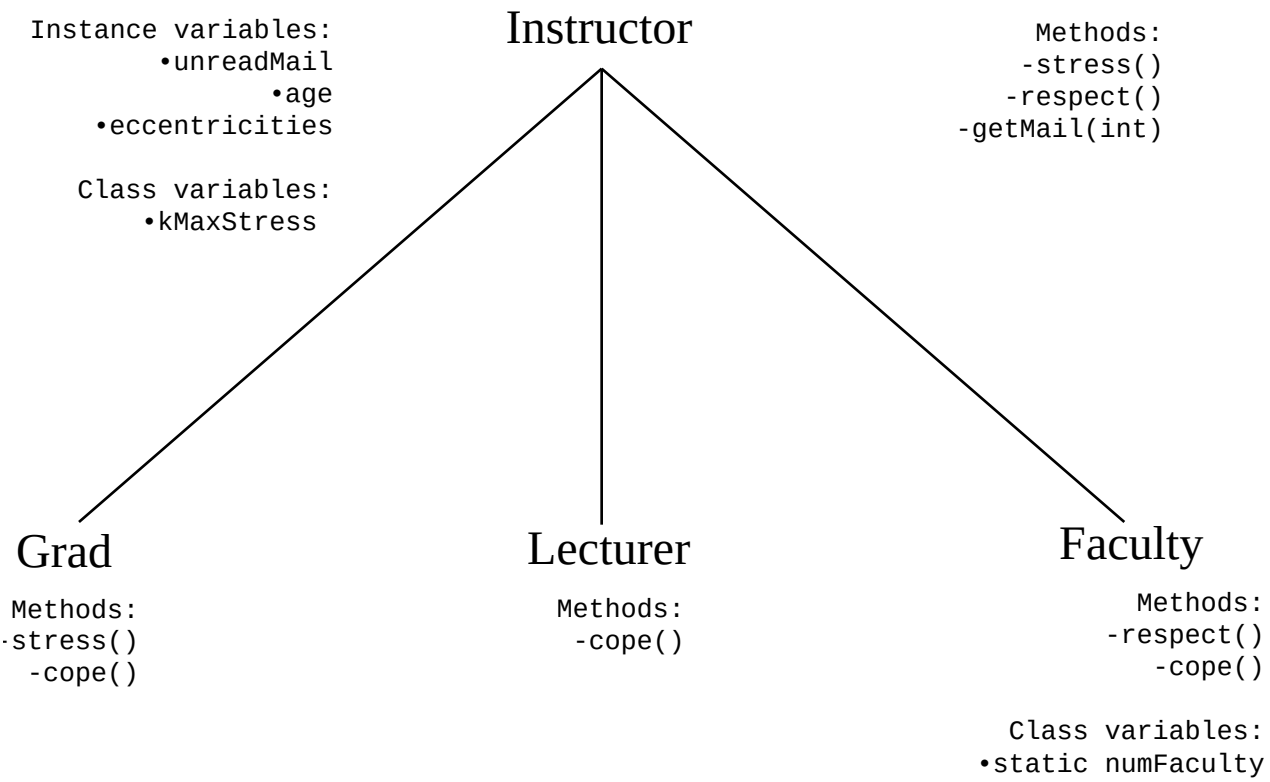
First, the amount of unread mail is increased.

Second, 90% of the time, there will be no change in the number of eccentricities. 10% of the time the number of eccentricities will randomly go up or down by one.

Third, the new unread mail may cause the instructor's stress factor to become larger than their respect factor which makes the instructor unhappy. In this case, instructors have different coping mechanisms. Faculty react by gaining 10 eccentricities. Lecturers react by accidentally deleting half their unread mail. Grad students react by reading all of their unread mail. The resulting mental anguish causes the Grad student's eccentricities to go up or down by one randomly. The coping mechanisms may or may not bring the stress and respect factors back in line. The coping mechanism is activated at most one time for each batch of incoming mail.

The big picture: why inheritance?

This problem is particularly suitable because it involves a set of classes that are similar to each other in most respects but differ in a few crucial aspects of their behavior. Below is the sort of design drawing you might make to help think about your solution:



Inheritance diagrams such as the one above are the bread-and-butter of object-oriented design. The class at the top has a list of all its methods and variables. The three classes below it are subclasses of the Instructor class, as indicated by the connecting lines. By convention, the class on the top is the superclass (i.e., the base class) and the one below is the subclass (the specialized class). Note that the three subclasses each only lists a few methods and variables. By default, we assume that each subclass has exactly the same methods and variables as its superclass, and only explicitly indicate methods and variables when they are exclusive to the subclass.

These diagrams are useful because they clearly indicate the code sharing within a class hierarchy without the clutter and hassle of actual code. All common data have been factored up to the Instructor superclass, and each subclass is only listed with the methods and variables that are specific to that class. As a rule, your code should match your hierarchy diagram—each subclass should only need to implement the methods and variables listed on the diagram. Everything else is handled by inheritance.

Those in tune with the object-oriented mindset have probably noticed that the hierarchy diagram does seem to have some repetition. All three subclasses include the method cope(), which should immediately cause the object-oriented programmer to consider moving that method upward into the Instructor superclass. However, each of the three subclasses has a different behavior for cope(), so each subclass will have to implement that method separately. Moreover, there is no default behavior for cope(); we only know what each of the specific types of Instructor does to cope. Even if we wanted to implement some sort of default method Instructor.cope(), we really don't have any idea of how to do so.

This leads us to the observation that there really is no such thing as an Instructor. We have Lecturer's, Grad's, and Faculty's, but these are all types of Instructors; there is no just-a-plain-Instructor. We need the Instructor class to properly design our class hierarchy, but it doesn't make sense to have an Instructor object. This is a classic example of an *abstract* class: a class which exists only to be a superclass to other objects.

The details:

```
// File: Instructor.java
// Description: This file implements the class Instructor in figure above.
////////////   Instructor Definitions  ///////////////

import java.lang.*;
import java.util.*;
// Class Instructor
// Description: Abstract class for all types of Instructors
// Here Instructor is subclassed (by default) from Object.
// This means it can be easily used with other class in the
// java.utils package including Vector, Hashtable, etc.

public abstract class Instructor {
```

```
private final int kMaxStress = 1000;
protected int unreadMail;
protected int age;
protected int eccentricities;
```

```
   // Construct with an age and init other to 0.
   protected Instructor(int anAge)
   {

      age = anAge;
      unreadMail = 0;
      eccentricities = 0;
   }
   public int stress()
   {
      if (unreadMail <= kMaxStress)
         return (unreadMail);
      else
         return (kMaxStress);
   }

   public int respect()
   {
      int stress = age - eccentricities;
      if (stress >= 0)
         return (stress);
      else
         return (0);
   }

   public void getMail(int numMessages)
   {
      unreadMail += numMessages;

      // 10% chance of eccentricity change
      if (Math.random() <= 0.10)
         randomEccentric();

      if (stress() > respect())
         cope();  // this is the key polymorphic line
   }

   // Helper which randomly change the eccentricities up or down
   // by 1. Does not allow negative number of eccentricities.
   protected void randomEccentric()
   {
      int delta;

      if (Math.random() < 0.5)
         delta = 1;
      else
         delta = -1;

      eccentricities += delta;

      if (eccentricities < 0)
         eccentricities = 0;
   }

   // helper method
   // This indicates to the compiler that Instructor
   // is an abstract super class.
   abstract protected void cope();
}
```

```
// File: GradStudent.java
// Description: This file implements the class GradStudent in figure above.
////////////   GradStudent Definitions  ///////////////

public class GradStudent extends Instructor {

   public GradStudent(int anAge)
   {
      super(anAge); //just use super class constructor
   }

   public int stress()
   {
      return (2 * super.stress());
   }

   protected void cope()
   {
      unreadMail = 0;            // read all mail
      randomEccentric();
   }
}

// File: Lecturer.java
// Description: This file implements the class hierachy in figure above.
////////////   Lecturer Definitions  ///////////////

public class Lecturer extends Instructor {

   public Lecturer(int anAge)
   {
      super(anAge);
   }

   protected void cope()
   {
      unreadMail = unreadMail / 2;     // delete half of unread mail
   }
}

// File: Faculty.java
// Description: This file implements the class Faculty in figure above.
////////////   Faculty Definitions  ///////////////

public class Faculty extends Instructor
{
   private static int numFaculty = 0;  // initialize class static

   public Faculty(int anAge)
   {
      super(anAge);
      numFaculty++;        // add to static class count
   }

   public int respect()
   {
      return (age + eccentricities - numFaculty);
   }
```

```
    protected void cope()
    {
       eccentricities += 10;
    }

    protected void finalize() throws Throwable
    {
       // this is supposed to update the numFaculty when the object is
       // destructed.  Unfortunately, this only happens when it's
       // garbage collected.  Garbage collection can be forced to happen
       // by calling System.gc(); the problem is: we don't know when to call
       // it.

       numFaculty--;
    }
}
```

Concurrency in Java

Concurrency in Java is relatively simple. Most of the messy details are hidden away in clean, high-level abstractions, so we don't have to worry about them. For details, see handout #55 ("Concurrency in Java") or the information on the `Thread` class in the JDK 1.1 documentation (located at the URL provided on the first page of this handout, as part of the package `java.lang.*`).

You can think of the `Thread` class as the object-oriented Java version of the thread package we used earlier this year. For each type of thread we want to run, we create a specialized subclass of `Thread`, making sure to override the `run` method. We can then just message the `start` method of those subclasses to set them running.

For synchronization and protection of shared data, Java provides the `synchronized` keyword. If a method is tagged as `synchronized`, that method is automatically protected, meaning that only one thread is allowed to execute a `synchronized` method for a given object. Again, see handout #55 for further details.

Concurrency example: TicketSellers

Here we are going to implement the classic multiple-ticket-sellers-accessing-a-single-ticket-database-concurrently problem. Don't worry - this is cake compared to sweethand. We remember how to do this in C, but it becomes even easier in Java.

We have the following classes:

SalesmanApplication
Salesman
TicketDatabase

```
public class SalesmanApplication {

   public final static int NUM_TICKETS = 10000; // #define in Java
   public final static int NUM_SALESMAN = 2;

   public static void main(String args[]) {

      TicketDatabase db = new TicketDatabase(NUM_TICKETS);

      For (int i=0; i < NUM_SALESMAN; i++) {
         Salesman randomPerson = Salesman("Salesman"+Integer.toString(i),db);
         randomPerson.start();
      }

      try {
         System.in.read(); // prevent console window from going away
      } catch (java.io.IOException e) {}
   }
}

public class TicketDatabase extends Object
{
   private int numTicketsRemaining;

   public TicketDatabase(int startingCount)
   {
      numTicketsRemaining = startingCount;
   }

   public boolean sellOneTicket()
   {
      if(numTicketsRemaining > 0)    // this function will return false if
      {                              // there are no tickets left to sell
         numTicketsRemaining--;
         return true;
      }

      return false;
   }
}

public class Salesman extends Thread
{
   private TicketDatabase db;
   private int numTicketsISold;
   private String name;

   public Salesman(String name, TicketDatabase db)
   {
      super(name);
      this.db = db;
      numTicketsISold = 0;
   }
```

```
    public void run()
    {
       while(db.sellOneTicket()) // sellOneTicket will return true if there
       {           // were tickets to sell and false otherwise
          iSoldOneTicket();
       }

       System.out.println(getName()
                + " sold "
                + Integer.toString(numTicketsISold)
                + " tickets.");
          // Crazy little printf to print the name and num tickets sold
    }

    public void pause(int numMilliseconds) // need this for later
    {
       try{ sleep(numMilliseconds); }
       catch (InterruptedException e) {}
    }

    public void iSoldOneTicket()
    {
       numTicketsISold++;
    }
}
```

Now what will happen when we run the application? How many tickets will we sell? How do we fix the problem?

Answer

We need to have a critical section in `sellOneTicket`. This is done in Java by declaring the method as `synchronized`. The new function definition is:

```
    public synchronized boolean sellOneTicket()
    {
       if(numTicketsRemaining > 0)     // this function will return false if
       {                // there are no tickets left to sell
          numTicketsRemaining--;
          return true;
       }
       return false;
    }
```

Note that nothing else in the function changed. By declaring the function `synchronized`, only one thread can be in the function at a time. This solves our selling too many tickets problem because no thread can enter the `sellOneTicket` function until another leaves. If we had another function in `TicketDatabase` that we also wanted to `synchronize`, we could -- but only one thread can be in any of the `synchronized` functions at a time. All the `synchronized` functions of a class are protected by the same binary semaphore.

More inheritance

Now if we wanted to make special sales people such as Jerry and Andy who do all the behavior of any other salesperson with one addition—they each print out a nice little message each time they sell a ticket - how would we do this?

```
public class JerrySalesman extends Salesman
{
      public JerrySalesman (TicketDatabase db) {
         super("Jerry", db);
      }

      public void iSoldOneTicket()
      {
         super.iSoldOneTicket();
         System.out.println("Yeah baby, I sold a ticket!");
      }
}

public class AndySalesman extends Salesman
{
      public AndySalesman (TicketDatabase db) {
         super("Andy",db);
      }

      public void iSoldOneTicket()
      {
         super.iSoldOneTicket();
         System.out.println("I am a ticket selling Machine!");
      }
}
```

Since the original `Salesman` had an `iSoldOneTicket` function, we simply had to override this method. Note that the immediate parent of `JerrySalesman` and `AndySalesman` are not `Thread`, but they inherit all that behavior through the `Salesman` class. In turn, `Thread` inherits behavior from `Object` like all other Java classes.