CS1007: Object Oriented Design and Programming in Java

Lecture #7

Feb 7

Shlomo Hershkop
shlomo@cs.columbia.edu

Outline

- How to design classes….correctly
- Design considerations
- Testing
- Putting all together
- Code
- Other considerations and issues
- Work responsibilty

Announcements

- Homework 1 due Feb 12
  - If you are having problems….OH
- Midterm date 2/28:
  - Will post review notes and study tips
  - Will be open book
  - No computers

  - Reading: Chapter 3-3.5

From last Time

- Encapsulation allows us to divide objects into logical parts and only present specific views of the object to outside manipulators
- Division of work
  - Accessors methods
    - Read a value
  - Mutators methods
    - Change a value (state) of object

Abstraction

- Process of picking out common features of an object
- Focus on essentials
- Eliminate details
- Information hiding

Example

- ATM Machine
- What is an abstract idea of an ATM ?

Encapsulation

- Hide implementation details
- Data access always done through methods
- Accessors and Mutators

- 2 levels of protection
  - State can not be changed directly from outside
  - Implementation can change without affecting users

- So how would a credit card object be described from an outside point of view?

Class design

- When designing a class need to be aware
  - What will the class represent
  - What processing is it going to be doing
  - What are the relationships to other classes
- Remember:
  - There is more than one way to represent an idea
  - Don’t be afraid of going back and changing something

Changing designs

- Changes can be painful
  - Introduce new bugs
  - Domino effect, small change can affect many classes
  - Break working program
  - New docs
- Or can be easy
  - If follow object oriented approach

Goal

- The goal of a well designed class
- **Reusability** – but all that hard work to work
- **Reliability** – if you find a bug can easily isolate it
- **Encapsulation** – can always come back and upgrade without changing anything else

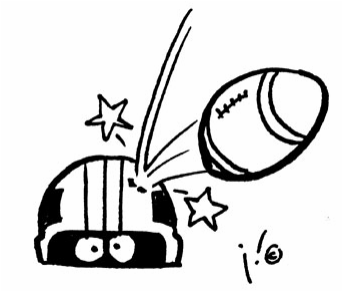
Example

- Want to store a bunch of emails, without using a database
- Class will take messages to store
- What accessor operations would we want to support?

Example2

- Ok we want to measure something which can't be seen or sensed
- It just happens
- Would like to differentiate between 2 of this stuff
- Decide to use an arbitrary system to tell one thing from another

- We aren't talking about the results of the SuperBowl



Measuring Time

- Date class in standard Library (very useful)

```
Date now = new Date();
   // constructs current date/time
System.out.println(now.toString());
  // prints date such as
  // Tue Feb 07 11:34:10 EST 2006
```

- Need a class to represent the date.
- Date class encapsulates point in time measured in milliseconds
- What is the best way?

Date class methods

boolean after(Date other)	Tests if this date is after the specified date
boolean before(Date other)	Tests if this date is before the specified date
int compareTo(Date other)	Tells which date came before the other
long getTime()	Returns milliseconds since the epoch (1970-01-01 00:00:00 GMT)
void setTime(long n)	Sets the date to the given number of milliseconds since the epoch

Some deprecated methods

int getDay() Deprecated. As of JDK version 1.1, replaced by Calendar.get(Calendar.DAY_OF_WEEK).

int getHours()

int getMinutes()

int getMonth()

int getSeconds()

Deprecated. As of JDK version 1.1, replaced by Calendar.get(Calendar.SECOND).

Date Class

- Deprecated methods were re-thought
- Date class methods supply total ordering on Date objects
- Convert to scalar time measure
- Note that before/after not strictly necessary
- (Presumably introduced for convenience)
- "I'll see you on 996,321,998,346." doesn't really work

Think in OO

- Is Date the correct idea?
- What are the limitations?
- i.e. what are the advantages and disadvantages of Date class

Ideas

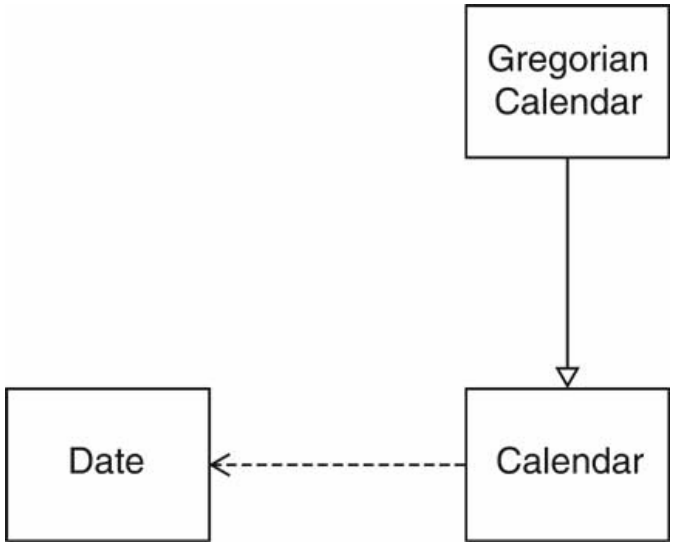
- Although would like to represent a point in time, usually time is associated with other measurements

- Month
- Year

The GregorianCalendar Class

- The Date class doesn't measure months, weekdays, etc.
- That's the job of a calendar
- A calendar assigns a name to a point in time
- Many calendars in use:
  - Gregorian
  - Contemporary: Hebrew, Arabic, Chinese
  - Historical: French Revolutionary, Mayan

Relationships



Next step

- Lets design a new class to represent a day

- Today is Tuesday

```
Day today = new Day();
Today.add(1); //should give us wednesday
```

Designing a Day Class

- Use the standard library classes, not this class, in your own programs
- Day encapsulates a day in a fixed location
- No time, no time zone
- Use Gregorian calendar

Goal of Day Class

- Answer questions such as

- How many days are there between now and the end of the year?
- What day is 100 days from now?
- How many days till my birthday (I've always wanted a _____)

Using what we learned

- What would the CRC card look like?

CRC Card

Day
relate calendar days to day counts

Design Phase

- daysFrom computes number of days between two days:

```
int n = today.daysFrom(birthday);
```

- addDays computes a day that is some days away from a given day:

```
Day later = today.addDays(999);
```

- Mathematical relationship:

```
d.addDays(n).daysFrom(d) == n
d1.addDays(d2.daysFrom(d1)) == d2
```

Lets digress

- There is some confusion in many programming languages between
- = and ==

- Bad choice

- Assignment
- Equality

Overloading operators

- Java doesn't have this (yet)
- Some languages (c++) allow you to redefine the common operators so that you can create a class and say

Class X = new Class(..

Class Y = new Class(…

Class Z = X + Y

What to do

- Create methods
  - Add
  - Multiply
  - getCopy
  - Etc

Class X = new Class(..

Class Y = new Class(…

Class Z = X.getCopy().add(Y)

- Constructor Date(int year, int month, int date)
- getYear, getMonth, getDate acccesors

Implementation

- Straightforward implementation:

```
private int year
private int month
private int date
```

- addDays/daysBetween tedious to implement
  - April, June, September, November have 30 days
  - February has 28 days, except in leap years it has 29 days
  - All other months have 31 days
  - Leap years are divisible by 4, except after 1582, years divisible by 100 but not 400 are not leap years
  - There is no year 0; year 1 is preceded by year -1
  - In the switchover to the Gregorian calendar, ten days were dropped: October 15, 1582 is preceded by October 4

Day Code

```
 public Day(int aYear, int aMonth, int aDate)
{
    year = aYear;
    month = aMonth;
    date = aDate;
}

private int year;
private int month;
private int date;

private static final int[] DAYS_PER_MONTH
= { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

private static final int GREGORIAN_START_YEAR = 1582;
private static final int GREGORIAN_START_MONTH = 10;
private static final int GREGORIAN_START_DAY = 15;
private static final int JULIAN_END_DAY = 4;

private static final int JANUARY = 1;
private static final int FEBRUARY = 2;
private static final int DECEMBER = 12;
```

Day Code

```
private Day nextDay()
112:    {
113:       int y = year;
114:       int m = month;
115:       int d = date;
116:
117:       if (y == GREGORIAN_START_YEAR
118:             && m == GREGORIAN_START_MONTH
119:             && d == JULIAN_END_DAY)
120:          d = GREGORIAN_START_DAY;
121:       else if (d < daysPerMonth(y, m))
122:          d++;
123:       else
124:       {
125:          d = 1;
126:          m++;
127:          if (m > DECEMBER)
128:          {
129:             m = JANUARY;
130:             y++;
131:             if (y == 0) y++;
132:          }
133:       }
134:       return new Day(y, m, d);
135:    }
```

```
private static int daysPerMonth(int y, int m)
{
    int days = DAYS_PER_MONTH[m - 1];
        if (m == FEBRUARY && isLeapYear(y))
          days++;
        return days;
}

 private static boolean isLeapYear(int y)
{
if (y % 4 != 0) return false;
if (y < GREGORIAN_START_YEAR) return true;
return (y % 100 != 0) || (y % 400 == 0);
}
```

Tester

```
01: public class DayTester
02: {
03:    public static void main(String[] args)
04:    {
05:       Day today = new Day(2001, 2, 3);
  //February 3, 2001
06:       Day later = today.addDays(999);
07:       System.out.println(later.getYear()
08:             + "-" + later.getMonth()
09:             + "-" + later.getDate());
10:       System.out.println(later.daysFrom(today));
  // Prints 999
11:    }
12: }
```

Notice

- Private helper methods
- Notice all the work to increment a day

Another idea

- For greater efficiency, use Julian day number
- Used in astronomy
- Number of days since Jan. 1, 4713 BCE
- May 23, 1968 = Julian Day 2,440,000
- Greatly simplifies date arithmetic

Code

```
public Day(int aYear, int aMonth, int aDate)
{
julian = toJulian(aYear, aMonth, aDate);
}

private int julian;
```

Code

```
private static int toJulian(int year, int month, int date)
{
   int jy = year;
   if (year < 0) jy++;
   int jm = month;
   if (month > 2) jm++;
   else{
        jy--;
        jm += 13;
   }
   int jul = (int) (java.lang.Math.floor(365.25 * jy)
   + java.lang.Math.floor(30.6001 * jm) + date + 1720995.0);
   int IGREG = 15 + 31 * (10 + 12 * 1582);
        // Gregorian Calendar adopted Oct. 15, 1582
   if (date + 31 * (month + 12 * year) >= IGREG)
   // Change over to Gregorian calendar
   {
   int ja = (int) (0.01 * jy);
   jul += 2 - ja + (int) (0.25 * ja);
   }
   return jul;
}
```

Any other ideas?

Why should you encapsulate?

- Even a simple class can benefit from different implementations
- Users are unaware of implementation
- Public instance variables would have blocked improvement
  - Can't just use text editor to replace all
    d.year
    with
    d.getYear()
  - How about
    d.year++?
  - d = new Day(d.getDay(), d.getMonth(), d.getYear() + 1)
  - Ugh--that gets really inefficient in Julian representation
- Don't use public fields, even for "simple" classes

Accessors and Mutators

- Day class has no mutators!
- Class without mutators is immutable
- String is immutable
- Date and GregorianCalendar are mutable

Don't Supply a Mutator for every Accessor

- Day has getYear, getMonth, getDate accessors
- Day does not have setYear, setMonth,setDate mutators
- These mutators would not work well
  - Example:

```
Day deadline = new Day(2001, 1, 31);
deadline.setMonth(2); // ERROR
deadline.setDate(28);
```

  - Maybe we should call setDate first?

```
Day deadline = new Day(2001, 2, 28);
deadline.setDate(31); // ERROR
deadline.setMonth(3);
```

- GregorianCalendar implements confusing rollover.
  - Silently gets the wrong result instead of error.
- Immutability is useful

Next Time

- Understand the 3 Day implementations covered in class.
- Do reading for chapter 3