CS1007: Object Oriented Design and Programming in Java

Lecture #6
Sept 22

Shlomo Hershkop
shlomo@cs.columbia.edu

Outline

- Feedback
- Some Theory
- Encapsulation
- Inheritance
- Interface
- Class design

- Reading
  - Chapter 3-3.4

Feedback

- UML design requirements on the HWs?
- How many diagrams /use cases necessary?
- Java inheritance
- Javadoc
- Extend class
- User input
- Running the example
  - javac *.java
  - java MailSystemTester

Announcements

- Lab components.
  - Hands on assignments
  - Thursdays (every other)
  - Will need CS account
    - www.cs.columbia.edu/crf/accounts

Abstraction

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

Example

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

Encapsulation

- Hide implementation details
- Data access always done through methods
- 2 levels of protection
  - State can not be changed directly from outside
  - Implementation can change without affecting users

- So how would the ATM machine object be described from an outside point of view?

Example 2

- Date class in standard Library
- Many programs manipulate dates such as "Saturday, February 3, 2001"
- Date class:

```
Date now = new Date();
   // constructs current date/time
System.out.println(now.toString());
  // prints date such as
  // Sat Feb 03 16:34:10 PST 2001
```

Example 2

- Representing the date.
- Date class encapsulates point in time
- 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

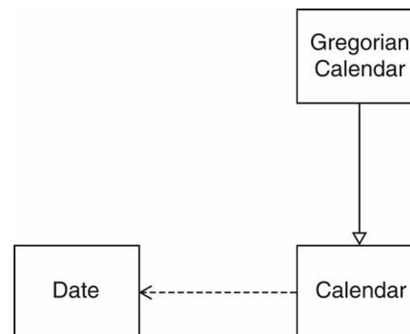
Think in OO

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

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

Java Date Handling



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?

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
```

- Clearer when written with "overloaded operators":

```
(d + n) - d == n
d1 + (d2 - d1) == d2
```

- 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: }
```

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

- Mutator: Changes object state
- Accessor: Reads object state without changing it
- 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

Sharing Mutable References

- References to immutable objects can be freely shared
- Don't share mutable references

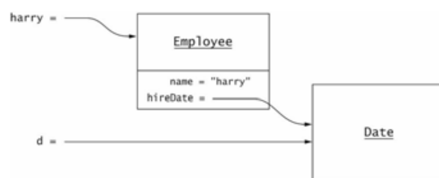
```
class Employee
{
   . . .
   public String getName() { return name; }
   public double getSalary() { return salary; }
   public Date getHireDate() { return hireDate; }
   private String name;
   private double salary;
   private Date hireDate;
}
```

- Pitfall:

```
Employee harry = . . .;
Date d = harry.getHireDate();
d.setTime(t); // changes Harry's state!!!
```

- Remedy: Use clone

```
public Date getHireDate()
{
   return (Date)hireDate.clone();
}
```



Final Instance Fields

- Good idea to mark immutable instance fields as final

```
private final int day;
```

- final object reference can still refer to mutating object

```
private final ArrayList elements;
```

- `elements` can't refer to another array list
- The contents of the array list can change

Separating Accessors and Mutators

- If we call a method to access an object, we don't expect the object to mutate
- Rule of thumb:
  Mutators should return void
- Example of violation:

```
Scanner in = . . .;
String s = in.next();
```

- Yields current token and advances iteration
- What if I want to read the current token again?

- Better interface:

```
String getCurrent();
void next();
```

- Even more convenient:

```
String getCurrent();
String next(); // returns current
```

- Refine rule of thumb:
  Mutators can return a convenience value, provided there is also an accessor to get the same value

Side Effect

- Side effect of a method: any observable state change
- Mutator: changes implicit parameter
- Other side effects: change to
  - explicit parameter
  - static object
- Avoid these side effects--they confuse users
- Good example, no side effect beyond implicit parameter

```
a.addAll(b)
```

mutates a but not b

Side Effects II

- Date formatting (basic):

```
SimpleDateFormat formatter = . . .;
String dateString = "January 11, 2012";
Date d = formatter.parse(dateString);
```

- Advanced:

```
FieldPosition position = . . .;
Date d = formatter.parse(dateString, position);
```

- Side effect: updates position parameter
- Design could be better: add position to formatter state

III

- Avoid modifying static objects
- Example: System.out
- Don't print error messages to System.out:

```
if (newMessages.isFull())
   System.out.println("Sorry--no space");
```

- Your classes may need to run in an environment without System.out
- Rule of thumb: Minimize side effects beyond implicit parameter

Next Time

- Do homework assignment
- Read chapter 3-3.5