CS1007: Object Oriented Design and Programming in Java

Lecture #8

Feb 9

Shlomo Hershkop
shlomo@cs.columbia.edu

1

Outline

- Review and some basics
- Enforcing Encapsulation
- Unit Testing
- Sorting Algorithms
- Polymorphism
- Interfaces
- Basic graphics
- Layout managers
- Anonymous Classes

2

Announcement

- If you are having trouble with the HW…please see me
- I will be releasing the next hw over the weekend
- Please do not be embarrassed to ask basic programming questions….its the only way to learn

3

Word on packages

- If you create a package called “mastermind” all classes will sit in a mastermind directory
- mastermind/Guess.java
- To manually compile:
- Directory containing mastermind directory
  - javac mastermind/*.java
  - java mastermind.GuessMainGame ← class with main in it.

4

Exceptions

- Lot of confusion on using/setting/programming exceptions
- Have you stopped by oh?
- Have you done background reading in a java book?

5

From last Time

- Basic principle of OOD
  - Encapsulation
  - Using accessors and mutators to limit outside view of internal implimentation

6

- Immutable object – are those which can't be changed once set…That is its state is guaranteed to stay identical over its lifetime
  - Simple to use and understand
  - Great building blocks

7

final keyword

- final modifier has different meanings depending on context
  - Variables
    - Can only set value once
  - Parameters
    - Can not change value
  - Methods
    - No one can redefine this class
  - Classes
    - Can't inherit from this class

8

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

9

- Want to create a simple class:
- Represent an employee
  - Name
  - Salary
  - Hire date

What would base class look like?

10

- Which methods would we add?

11

Some code:

```
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;
}
```

12

Setting data

- How would set hire date be written?

13

Danger of sharing

- Pitfall:

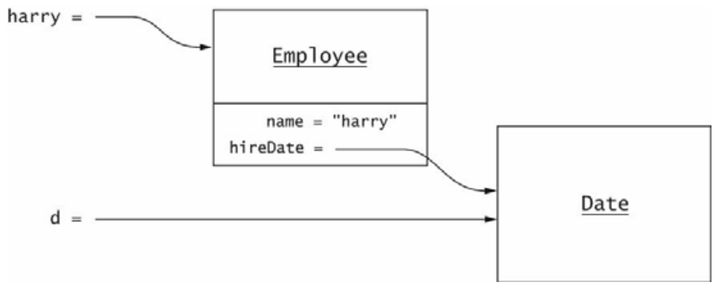
```
Employee harry = . . .;
Date d = harry.getHireDate();
d.setTime(t); // changes Harry's state!!!
```

- Remedy: Use clone

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

14

Visually



15

Separating your 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?

16

- 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

17

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

18

Background

- SimpleDateFormat API
  - Allows you to move from text->date and date->text using patterns
  - "EEE, MMM d, ''yy" == Wed, Jul 4, '01
  - "h:mm a" == 12:08 PM

19

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

20

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");
```

- WHY???
- Rule of thumb: Minimize side effects beyond implicit parameter

21

Idea:

- For real object oriented programming, should be a minimum of objects floating in memory.
  - Using just methods to change objects
  - More responsibilities per object, but cleaner overall design

22

- Example: Mail system in chapter 2

```
Mailbox currentMailbox =
  mailSystem.findMailbox(...);
```

- Breaks encapsulation!!
- Suppose future version of MailSystem uses a database
- Then it no longer has mailbox objects
- Common in larger systems
- Karl Lieberherr: Law of Demeter

23

- The law: A method should only use objects that are
  - instance fields of its class
  - parameters
  - objects that it constructs with new
- Shouldn't use an object that is returned from a method call
- Remedy in mail system: Delegate mailbox methods to mail system

```
mailSystem.getCurrentMessage(int mailboxNumber);
mailSystem.addMessage(int mailboxNumber, Message msg);
. . .
```

- Rule of thumb, not a mathematical law
- Design of what not to do….

24

Emphasis

- Some of the design choices come with experience
- No “One size fits all solution”
- Solution to balance decisions:
  - Documentation
  - Redesign

25

Designing projects

- Will now talk about what goes into designing a set of classes which work together
- Remember
  - In general you will both give and be given only class files.
  - Along with the documentations (API) it is the only to know
    - What
    - How
    - Why

26

Quality of Class Interface

- Customers: Programmers using the class
  - Cohesion
  - Completeness
  - Convenience
  - Clarity
  - Consistency
- Engineering activity: make tradeoffs

27

Cohesion

- Class describes a single abstraction
- Methods should be related to the single abstraction
- Bad example:

```
public class Mailbox
{
   public addMessage(Message aMessage) { ... }
   public Message getCurrentMessage() { ... }
   public Message removeCurrentMessage() { ... }
   public void processCommand(String command) { ...
  }
   ...
}
```

28

Completeness

- Support operations that are well-defined on abstraction
- Potentially bad example: Date

```
Date start = new Date();
// do some work
Date end = new Date();
```

- How many milliseconds have elapsed?
- No such operation in Date class
- Does it fall outside the responsibility?
- After all, we have before, after, getTime

Convenience

- A good interface makes all tasks possible . . . and common tasks simple
- Bad example: Reading from System.in before Java 5.0

```
BufferedReader in = new BufferedReader(new
InputStreamReader(System.in));
```

- Why doesn't System.in have a readLine method?
- After all, System.out has println.
- Scanner class fixes inconvenience

- What is wrong with hard to read code??

31

- At least no one will be able to fix it (aka job security)
- Problem: they might hire you to fix it
- Discourages reuse and redesign

32

Be Clear

- Confused programmers write buggy code
- Bad example: Removing elements from LinkedList
- Reminder: Standard linked list class

```
LinkedList countries = new LinkedList();
countries.add("A");
countries.add("B");
countries.add("C");
```

- Iterate through list:

```
ListIterator iterator = countries.listIterator();
while (iterator.hasNext())
   System.out.println(iterator.next());
```

33

- Iterator between elements
- Like blinking caret in word processor
- add adds to the left of iterator (like word processor):
- Add X before B:

```
ListIterator iterator =
  countries.listIterator(); // |ABC
iterator.next(); // A|BC
iterator.add("France"); // AX|BC
```

34

Interesting

- To remove first two elements, you can't just "backspace"
- remove does not remove element to the left of iterator
- From API documentation:
```
Removes from the list the last element
that was returned by next or previous.
This call can only be made once per call
to next or previous. It can be made only
if add has not been called after the last
call to next or previous.
```
- Huh?

35

Be Consistent

- Related features of a class should have matching
  - names
  - parameters
  - return values
  - behavior
- Bad example:

```
new GregorianCalendar(year, month - 1, day)
```

- Why is month 0-based?

36

Consistency

- Bad example: String class

```
s.equals(t) vs. s.equalsIgnoreCase(t)
```

- But

```
boolean regionMatches(int toffset,
   String other, int ooffset, int len)
boolean regionMatches(boolean ignoreCase, int
  toffset,
   String other, int ooffset, int len)
```

- Why not regionMatchesIgnoreCase?
- Very common problem in student code

37

Programming by Contract

- Spell out responsibilities
  - of caller
  - of implementer
- Increase reliability
- Increase efficiency

38

Preconditions

- Caller attempts to remove message from empty MessageQueue
- What should happen?
- MessageQueue can declare this as an error
- MessageQueue can tolerate call and return dummy value
- What is better?

39

Make exception .. exceptional

- Excessive error checking is costly
- Returning dummy values can complicate testing
- Contract metaphor
  - Service provider must specify preconditions
  - If precondition is fulfilled, service provider must work correctly
  - Otherwise, service provider can do anything
- When precondition fails, service provider may
  - throw exception
  - return false answer
  - corrupt data

40

Preconditions

```
/**
   Remove message at head
   @return the message at the head
   @precondition size() > 0
*/
Message remove()
{
   return elements.remove(0);
}
```

- What happens if precondition not fulfilled?
- IndexOutOfBoundsException
- Other implementation may have different behavior

41

Arrays and queues

- If you have a queue
- What does it mean to remove a message?

42

- What does it mean to remove a message if the queue is empty?

43

Next Time

- Read chapter 3 in the book
- Wrap up hw1
- Check out hw2 (Sunday)

44