Object Oriented Programming and Design in Java

Session 4
Instructor: Bert Huang
Announcements

• ACM competition
• Homework 1 officially out
  • due Feb. 17\textsuperscript{th} 11 AM
HACK!

Competition

Yit

Prizes: Xbox and Hundreds of Dollars
Register Online: tinyurl.com/acm-hacker

Date: Thursday February 4th
Tech Talk by Prof. Steve Bellovin

Time: 6:00 PM
Location: CS Lounge (Mudd 403)

FREE PIZZA!!
Homework 1

• Battleship against computer via text interface
• Start ASAP
• Use O.H. and email to bounce design ideas off the TAs and me
• Academic honesty
• Have fun!
Review

• Turning ideas into a program
• Use cases
• identifying classes and responsibilities
• UML diagrams: class diagram, sequence diagram, state diagram
• Example: todo list manager
• Reading voice mail example
Today’s Plan

• Review example from end of last class
• Designing classes
  • encapsulation
  • accessors/mutators
• programming by contract
Class Diagram

**TodoFileManager**
- savedFile : File
- loadFile()
- saveFile()

**TodoItem**
- name : String
- date : Date
- getName()
- getDate()

**TodoPrompt**
- executeCommand(String s)
- displayList()

**TodoList**
- list<TodoItem> : ArrayList
- sort()
- addItem()
- deleteItem()
- getItem(int index)
Class Diagram

```
Todoltem
- name : String
- date : Date
- getName()
- getDate()

TodoList
- list<Todoltem> : ArrayList
- sort()
- addItem()
- deletemItem()
- getItem(int index)

TodoFileManager
- savedFile : File
- loadFile()
- saveFile()

TodoPrompt
- executeCommand(String s)
- displayList()
```
State Diagram

- no list loaded
- list loaded, prompt user to view change list (add or delete)
- user exits, save list
Ideas to Programs

Analysis

Design

Implementation

(common sense)

(object-oriented)

(actual programming)

Today’s material
Designing Classes

• Even simple classes have various design decisions:
  • How much error checking?
  • How much power should the user have?
  • How far “under the hood” can the user see?
Why Encapsulation?

```
int data
String name
OtherClass thing
```

```
MyClass

void doSomething()
int getSomething()
```

The rest of your program...

No encapsulation
Why Encapsulation?

MyClass

int data
String name
OtherClass thing

void doSomething()
int getSomething()

/* interface methods */

The rest of your program...

Encapsulation
Why Encapsulation?

- Easier changes to implementation
- Control of inputs and outputs
- Less old code to have to maintain when updating
- When changes are made, easier to find what code is affected
Good Interfaces

- **Cohesion** - represent only one concept

- **Completeness** - does everything you’d expect

- **Convenience** - some syntactic sugar,
  
  BufferedReader(new InputStreamReader(System.in))

- **Clarity** - behavior of class should be easy to explain accurately

- **Consistency** - naming conventions, etc
Accessors vs. Mutators

- Methods to handle data members
- **Accessors** for reading
- **Mutators** for writing/modifying
- Keep them separate
Side Effects

• Avoid methods with side effects
• Calling accessors repeatedly should yield same result
  • counterexample: Scanner.nextLine()
• Mutators should change things in an obvious way
Programming by Contract

- Another formalism to help organization
- All methods and classes have “contracts” detailing responsibilities
- Contracts expressed as **preconditions**, **postconditions**, and **invariants**
Preconditions

• Condition that must be true before method is called
  • e.g., indices must be in range, objects must not be null
• Limits responsibilities of your method
Assertions

• You can check preconditions before executing on bad input using assertions

• Java includes assertions via
  assert (boolean) : “explanation”;

• When assertions enabled, program exits and displays explanation

• java -enableassertions MyProgram
Postconditions

- Conditions guaranteed to be true after method runs
- e.g., after calling sort(), ToDoList elements are sorted by due date
- Useful when in addition to @return tags
- I.e., usually involves mutators or side effects
Invariants

- General properties of any member of a class that are always true
  - e.g., ToDoList is always sorted
- *Implementation invariants* are useful when building the class
- *Interface invariants* are useful when using the class
Exceptions

• What happens when the contract is breached? Crash?

• Exceptions are ideal for when contracts can be breached

• javadoc:
  @throws IndexOutOfBoundsException

  throw new IndexOutOfBoundsException("Accessed " + i + " when size = " + A.length());
Law of Demeter

- A method should only use
  - Instance fields of its class
  - Parameters
  - Objects that it constructs with new
- Think of your programs as growing
ToDoList.addItem()

- addItem(String name, Date date)
- @precondition ArrayList is initialized
- @postcondition new item is in list
- @postcondition list is sorted
- assert list != null : “list wasn’t init’d”;

```
TodoList

| list<Todoltem> : ArrayList |
| sort() |
| addItem() |
| deleteItem() |
| getItem(int index) |
```
ToDoList.deleteItem()

- deleteItem(String item_name)
- @precondition list has element named item_name
- @postcondition item no longer in list
- @postcondition list is sorted
ToDoList.getItem()

- getItem(int index)
- @precondition 0 ≤ index < list.size()
- @postcondition list is sorted
- @throws IndexOutOfBoundsException
- (This design is flawed.)
Reading

- Horstmann Ch. 3