CS1007: Object Oriented Design and Programming in Java

Lecture #4
Jan 26
Shlomo Hershkop
shlomo@cs.columbia.edu

Outline

- Feedback
- Review
- Homework
- Event Driven programming
- Showing off eclipse
- Graphic Programming I
- Graphic Programming II
- Object Oriented Design Process.
- Intro CRC & UML

- Background reading on graphics (basics and concepts)
- Next class Reading chapter 2-2.5
Announcements

• Please make sure to note: HW due: Feb 12 at 11pm (electronically).

• TA Announcement: Ohan will hold office hours from 4:30 to 6:30 today instead of 11-1 on Friday.

Feedback

• Pace

• Graphic programming background
Exceptions

• Tool to handle error during program run
  – Exception == exceptional event

  – Idea: when an error occurs, a method can create an Object representing the error and hand it to the run time system

  – The runtime system now tries to find someone to handle the particular error, it uses the call stack to find a handler

Exception handlers

• Are defined by your catch expression
• If a specific method doesn’t know how to handle the specific exception, it forwards it up the stack
• Remember: can have multiple catch blocks one after other
  – Exceptions have a hierarchy, they will be evaluated from highest to lowest, so the catch blocks must be in reverse order.
The birth of an exception

- You might use a method which might throw an exception
- You might create a method which creates an exception
- Your code might trigger an exception
InvalidAccountException

```java
public class InvalidAccountException extends Exception {

    public InvalidAccountException (String message) {
        super(message);
    }
}
```

Your method

```java
public boolean checkBalance(int account) throws InvalidAccountException {
    if(account==null || account < 1) {
        throw new InvalidAccountException("Bad Account Number");
    }
    ...
}
```
Chaining Exceptions

```java
try {
    ...
} catch (IOException e) {
    throw new SampleException("Other IOException", e);
}
```

Point

- Can deal with the problem
  - Ask user for help
  - Figure out what should be done
  - Log the error
  - Print a trace to debug
  - Die (ARGHHHHHHH!)
Practical Tips

• In a general sense try, catch blocks impose some overhead to the resulting code
• Although can enclose all your code in some try, catch block its not a good idea
• Need to decide at what point, which errors can occur, and what the appropriate response will be

Homework

• Playing MasterMind
• G = guesses
• X = choices
• N = range of numbers
Programming Models

• Control Flow Programming
  – Program which follows control flow changing course at specific branch points.

• Event Driven Programming
  – Program which is driven by events (signal) and responses in an event loop framework.

• GUI
  – Window system which interacts with users

• AWT
  – Abstract Windows Toolkit

• SWING
  – Updated version of many AWT object with event driven paradigm design
Event and Listeners

- Event Objects
  - Objects which trigger a Listener Object
  - Example: click on a button
- Listener Object
  - Object which react to events
  - Example once clicked do something

- Exception Handling!!

Components and Containers

- Can program GUI using a surface and drawing circles, boxes, etc
- OOD:
  - Components
    - Individual GUI objects
  - Containers
    - Object which can hold components
Simple example

```java
JFrame easyWindow = new JFrame();
easyWindow.setSize(300, 300);
easyWindow.setTitle("This is your first window");
easyWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
easyWindow.setVisible(true);
```

Adding a button

- One component is a button

```java
JButton closeButton = new JButton("Click to close");
easyWindow.add(closeButton);
```
Events

• By default there are a few events associated with each general window (container)
  • Maximize
  • Minimize
  • Close
  • sizing

Can add events to components

closeButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent evt) {
    if (evt.getSource() instanceof JButton) {
      JButton closeButton = (JButton)evt.getSource();
      if (closeButton.getBackground() == Color.BLUE) {
        closeButton.setBackground(Color.YELLOW);
      } else {
        closeButton.setBackground(Color.BLUE);
      }
    }
  }
});
Alternativly can create a class

```java
public class colorListener
    implements ActionListener {

    public void
    actionPerformed(ActionEvent...)
```

Adding a second button

- Can add more using the same code, but then bump into an issue of where do both buttons go??

- Ideas?!?
Layout Managers

- Layout managers are class which handle how things will be set out in the window

Ending a GUI program

- Think of it as being an infinite loop waiting for events
- So if you don’t explicitly end the program……
Other GUI stuff

- Much more components
  - JLabels
  - JRadioButtons
  - Dozens more
- Other Containers
  - JWindow
  - JFrame
  - JPanel

Ahead

- Switch gears from programming java
- Object Oriented Design
Program Design

- Analysis
- Design
- Implementation

Analysis Phase

- Functional Specification
  - Completely defines tasks to be solved
  - Free from internal contradictions
  - Readable both by domain experts and software developers
  - Reviewable by diverse interested parties
  - Testable against reality
Design Phase

• Goals
  – Identify classes
  – Identify behavior of classes
  – Identify relationships among classes

• Artifacts
  – Textual description of classes and key methods
  – Diagrams of class relationships
  – Diagrams of important usage scenarios
  – State diagrams for objects with rich state

Implementation Phase

• Implement and test classes
• Combine classes into program
• Avoid "big bang" integration
• Prototypes can be very useful
Problem 1:

• Design a voicemail system for use in your typical cellphone.

• How would the requirements look like?
• What would be a typical session?
• What modules are involved?

Identifying Classes in design

• Rule of thumb: Look for nouns in problem description

• Mailbox
• Message
• User
• Passcode
• Extension
• Menu
When defining classes

• Focus on concepts, not implementation

• ??? stores messages
  – Lets say a messageQueue

• Don't worry yet how the queue is implemented

Categories

• Tangible Things
• Agents
• Events and Transactions
• Users and Roles
• Systems
• System interfaces and devices
• Foundational Classes
Identifying Responsibilities

• Rule of thumb: Look for verbs in problem description

• Behavior of MessageQueue:
  • Add message to tail
  • Remove message from head
  • Test whether queue is empty

OO Design

• OO Principle: Every operation is the responsibility of a single class
• Example:
  – Add message to mailbox
• Who is responsible:
  – Message or Mailbox?
Relationship

• Dependency ("uses")
• Aggregation ("has")
• Inheritance ("is")

Dependancy

• C depends on D: Method of C manipulates objects of D
Example: Mailbox depends on Message
• If C doesn't use D, then C can be developed without knowing about D
Java definitions

- When class X extends Y
  - X is a subclass
  - Y is a superclass
- When interface A extends Interface B
  - A is a subinterface
  - B is a superinterface
- When G implements interface B
  - G is an implementation of B
  - B is an interface of class G

Independent operations

- Minimize dependency:
  - reduce having to relay on anything set in stone

- Example: Replace
  void print() // prints to System.out
  with

  String getText() // can print anywhere

- Removes dependence on System, PrintStream
Aggregation

- Object of a class contains objects of another class
- Example: MessageQueue aggregates Messages
- Example: Mailbox aggregates MessageQueue
- Implemented through instance fields

Relationships

- 1 : 1 or 1 : 0...1 relationship:

```java
public class Mailbox
{
    . . .
    private Greeting myGreeting;
}
```

- 1 : n relationship:

```java
public class MessageQueue
{
    . . .
    private ArrayList<Message> elements;
}
```
Inheritance

• More general class = superclass
• More specialized class = subclass
• Subclass supports all method interfaces of superclass (but implementations may differ)
• Subclass may have added methods, added state
• Subclass inherits from superclass
• Example:
  – ForwardedMessage inherits from Message
  – Greeting does not inherit from Message (Can't store greetings in mailbox)

Use Cases

• Analysis technique
• Each use case focuses on a specific scenario
• Use case = sequence of actions
• Action = interaction between actor and computer system
• Each action yields a result
• Each result has a value to one of the actors
• Use variations for exceptional situations
Use case: Leave a Message

1. Caller dials main number of voice mail system
2. System speaks prompt
   • Enter mailbox number followed by #
3. User types extension number
4. System speaks
   • You have reached mailbox xxxx. Please leave a message now
5. Caller speaks message
6. Caller hangs up
7. System places message in mailbox

Variations

• user enters invalid extension number
  – What do you do?
  – Who does it?
• What if user hangs up instead of using message?
• How many attempts at password?
CRC Cards

- CRC = Classes, Responsibilities, Collaborators

- Use an index card for each class
- Class name on top of card
- Responsibilities on left
- Collaborators on right

CRC

- Responsibilities should be high level
- 1 - 3 responsibilities per card
- Collaborators are for the class, not for each responsibility
Example

- Use case: "Leave a message"
- Caller connects to voice mail system
- Caller dials extension number
- "Someone" must locate mailbox
- Neither Mailbox nor Message can do this
- New class: MailSystem
- Responsibility: manage mailboxes

UML

- UML = Unified Modeling Language
- Many diagram types
- We'll use three types:
  - Class Diagrams
  - Sequence Diagrams
  - State Diagrams
UML

• Why do we model?
  – Provide structure for problem solving
  – Experiment to explore multiple solutions
  – Furnish abstractions to manage complexity
  – Decrease development costs
  – Manage the risk of mistakes

• Graphical Approach
  – Picture is worth 1000 words

UML Building Blocks

• model elements (classes, interfaces, components, use cases, etc.)
• relationships (associations, generalization, dependencies, etc.)
• diagrams (class diagrams, use case diagrams, interaction diagrams, etc.)

• Simple building blocks are used to create large, complex structures
  – elements, bonds and molecules in chemistry
  – components, connectors and circuit boards in hardware
Next time

• Read
• Make sure sketch out the homework and have a rough outline of what you need to do
• Download Violet (UML designer) and try to play with it.