CS 121 - Intro to Java - Lecture 21

Announcements

Ch 10 OWL hwk due 11/23
Program 6 due 11/29 - see website!
Exam released tonight
A note on extra credit problems
They can increase your overall grade
They are fine training for exams
If you plan to do 187 / 242...
How to prepare for the exam

• Go over OWL problems, ch 5-9 (ask about, if you have problems!) big topics: arrays, inheritance, while loops

• Do program six - it relies very heavily on inheritance

• Do the extra credit problems - any issues that turn up with them could be an issue for the exam (also - they are worth something!)

• Study with others

• SI tonight, office hours this afternoon

• Get your sleep
Program #6 - write a decoder for external files...

ZIL NBCM JLIALUG SIOL DIV CM NI CGUACHY NBUN
NBYLY ULY NYRN ZCFYM NBUN BUPY VYYH YHWIXYX
OMCHA U WUYMUL WCJBYL, U FCHYUL MBCZN WIXY, UHX
SIOL DIV CM NI XYWCJBYL NBYG. QY'PY UNNUWBYX NQI
YRUGJFYM VYFIQ, WIXY1.NRN UHX WIXY2.NRN. SIOL
MBIOFX WIMMNLOWN U NQI WFUMM MIFONCIH, QCNB
WFUMMYM XYWIXXYLCPYL UHX XYWIXYL. NBY XLCPYL
MBIOFX UWWYJN U WIXYX ZCFY (HUGY) UM CHJON – U
MNLCHA – UHX MBIOFX NBYH JLCHN NBY XYWIXXYX ZCFY
IH NBY MWLYYH. SIO GUS QILE QCNB U JULNHYL IH
NBCM UMMCAHGYHN. CZ SIO XI QILE QCNB MIGYIHY, VY
MOLY NI CHWFOXY HUGYM ZIL VINB JULNCWCJUHMN CH U
WIGGYHN UN NBY NIJ IZ YUWB IZ SIOL MOVGCMNNYX
ZCFY.
Interfaces in Java
The concept of an interface:

• A mechanism for specification

• We’ve seen them before: Java API

• Encapsulation: working on a “need-to-know” basis

• An interface is a kind of contract

• Big principles -> reuse; hygiene
Classic example of an interface at work

• A text editor’s copy & paste feature
• You almost surely only understand its functionality - that is, its interface
• Its implementation is opaque
• You don’t need to know how it’s implemented
• You would rather not know how it’s implemented
• An implementor may alter the implementation - and you would never know
Here is the “Scoring” interface - it’s just two methods -- and not even that: the methods involved are “disembodied”

```java
public interface Scoring{

    public double getScore();

    public void setScore(double newScore);
}
```
Here is the “Scoring” interface - it’s just two methods -- and not even that: the methods involved are “disembodied”:

```
public interface Scoring{

    public double getScore();

    public void setScore(double newScore);
}
```

Note the syntax.
Which classes might implement Scoring?

Baseball players - runs scored
Golfers - money earned or shots taken
Employees - days arriving on time
Union members - seniority
public class CookieSeller implements Scoring {

    private String name;
    private double boxesSold;

    public CookieSeller(String n, double sold) {
        name = n;
        boxesSold = sold;
    }

    public String getName() { return name; }

    public double getBoxesSold() { return boxesSold; }
}
public void setName(String newName){
    name = newName; }

public void setBoxesSold(double sold){
    boxesSold = sold; }

public double getScore(){  // req by interface
    return boxesSold; }

public void setScore(double sold){  // also req
    boxesSold = sold; }
}
An interface can also involve constants, and in fact may involve only constants...
public interface Directions{
    final int NORTH = 0;
    final int EAST = 1;
    final int SOUTH = 2;
    final int WEST = 3;
}

class BigTrip implements Directions {

    if (myDir() == NORTH) setDir(EAST);
}
There are standard interfaces, in the Java libraries

These provide a general framework/specification for some significant activity
Three familiar ones that are framed by an interface

Edit interface

highlight()
copyToClipboard()
cutToClipboard()
markCursor()
paste()
MouseListener interface

public void mouseClicked(MouseEvent e);
public void mouseEntered(MouseEvent e);
public void mouseExited(MouseEvent e);
public void mousePressed(MouseEvent e);
public void mouseReleased(MouseEvent e);
Comparable Interface

calculateTo(Object other)
A practical problem:

An array of Infants - kids - at a daycare center

Put them in alphabetical order, by name

Or

Put them in order by age..

This is easy, using the library interface Comparable
Extremely important library interface: **Comparable** - it’s intended to model the “natural” ordering of elements in a class

A single method: `compareTo`

```java
public int compareTo(Object other);
```
public int compareTo(Object other);

Tricky:

It’s binary - compares two objects, the calling object, and the single parameter object;

Return type isn’t boolean - it’s int.

The parameter is completely general: it’s of type Object;
Reminder

Object is the ultimate class

Every class is either:

• Derived from Object directly
• Indirectly derived from Object (e.g., Chance, from Random, from Object)
String implements Comparable

Implemented lexicographic (alphabetic) order

So: “boy” < “girl”

And thus if names is an array of Strings, then
java.util.Arrays.sort(names);
sorts them - that is, puts them in alphabetical order
String d = "dog";
String c = "cat";
String z = "zebra";

int j = d.compareTo(c);
Informally, “donkey” < “monkey”

Suppose \( d = \) “donkey”; \( m = \) “monkey”; \( z = \) “zebra”;

Then:

\[
\begin{align*}
\text{d.compareTo(m)} & \text{ returns a value < 0} \\
\text{m.compareTo(d)} & \text{ returns a value > 0} \\
\text{d.compareTo(d)} & \text{ returns value 0}
\end{align*}
\]
The “meaning” of `compareTo`:
- `a, b` are of some type (they’re cars, or strings, or tennis balls, or whatever)
- `a.compareTo(b) < 0` means: `a` comes before `b` in natural ordering
- `a.compareTo(b) == 0` means: `a`, `b`, equal in natural ordering
- `a.compareTo(b) > 0` means: `a` comes after `b` in natural ordering.

Example: String implements `Comparable`; natural ordering - lexicographic
- `a = "cow"`; `b = "snake"`; `c = "walrus"`;
- `a.compareTo(b) ->`  
- `b.compareTo(c) ->`  
- `c.compareTo(a) ->`  
- `c.compareTo(c) ->`
The “meaning” of compareTo:
a,b are of some type (they’re cars, or strings, or
tennis balls, or whatever)

\[\text{a}.\text{compareTo}(\text{b}) < 0 \text{ means: a comes before b in}
\text{natural ordering}\]

\[\text{a}.\text{compareTo}(\text{b}) == 0 \text{ means: a, b, equal in natural}
\text{ordering}\]

\[\text{a}.\text{compareTo}(\text{b}) > 0 \text{ means: a comes after b in}
\text{natural ordering.}\]

Example: String implements Comparable; natural
ordering - lexicographic

a = “cow”; b = “snake”; c = “walrus”;

\[\text{a}.\text{compareTo}(\text{b}) \rightarrow \text{negative value}\]

\[\text{b}.\text{compareTo}(\text{c}) \rightarrow \text{negative value}\]

\[\text{c}.\text{compareTo}(\text{a}) \rightarrow \text{positive value}\]

\[\text{c}.\text{compareTo}(\text{c}) \rightarrow \text{zero}\]
public class Infant {

To do this, we need to tack on a definition for
compareTo (that realizes the version of
less-than we’re interested in..)
public class Infant implements Comparable{
    private String name;
    private int age; // in months
    public Infant(String who, int months){
        name = who;
        age = months;
    }

    // by age
    public int compareTo(Object other){
        int b = ((Infant)other).age; // cast req!
        int a = this.age;
        return(a-b); // will be negative if a < b
    }
}
// now name, alphabetical, is natural ordering:

    public int compareTo(Object other) {
        String a = this.name;
        String b = ((Infant) other).name;
        return (a.compareTo(b));
    }

In other words: since a, b are Strings, we “hand off” the ordering decision to the String class
Why is Comparable valuable?

It’s pervasive - we’re always comparing things - with Comparable we can build functionality around it:

```
Arrays.sort(words);    // Arrays: a class in java.util
```

words: an array of Strings;

Arrays: library class, array manipulation functionality
sort: put things in order
This combo works for any array for which the object type implements Comparable.

Note: there’s a version of sort, Arrays.sort(A, j, k), that sorts array A from j through k.
import java.util.*;

class StringTest {
    public static void main(String[] args) {
        String[] words = {
            "now", "is", "the", "time", "to",
            "go"};

        for (int j = 0; j < words.length; j++)
            System.out.print(words[j] + " ");
        System.out.println();
        Arrays.sort(words);
        System.out.println("*and now in sorted order*");
        for (int j = 0; j < words.length; j++)
            System.out.print(words[j] + " ");
        System.out.println();
    }
}
Program output:

now is the time to go
*and now in sorted order*
go is now the time to
This solves our daycare center problem:

1) Reimplement Infant:
   ```java
   public class Infant implements Comparable{
       ...
       public int compareTo(Object other){...}
   }
   ```

   And then, if kids is an array of Infants:
   ```java
   Arrays.sort(kids);
   ```

   Does the trick
Given our Scoring interface, can we make sense of a declaration like this one:

```java
Scoring s;
```

So here `s` is a variable that references a Scoring object (?) (Remember: there are no Scoring objects - Scoring is an interface)

This can only mean: `s` references an object whose class implements the Scoring interface.
Rethinking the cell model for variables

```java
int num = 5;

Infant kid = new Infant("Sy",8);

Scoring s;
```

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>num</td>
<td>5</td>
</tr>
<tr>
<td>kid</td>
<td>1224</td>
</tr>
<tr>
<td>s</td>
<td>3376</td>
</tr>
</tbody>
</table>
Rethinking the cell model for variables

```java
int num = 5;
Infant kid = new Infant("Sy",8);

Scoring s;
```

Holds any object whose class implements Scoring.
However:

A variable that references a Scoring object can only use methods & constants from the Scoring interface!
Scoring c = new CookieSeller("Dana", 122);

Scoring g = new Golfer("Lulu", 77);

System.out.println(c.getScore()); // ok
System.out.println(g.getScore()); // ok

System.out.println(c.getBoxesSold()); // no way!

System.out.println(((CookieSeller)c).getBoxesSold()); // this IS ok!
This generalized view of variables underlies concept of **polymorphism** - “many forms”

Thus a scoring variable \( s \) can hold a:

- golfing object
- CookieSeller object
- etc.
With the Scoring interface in mind, we could create a class called Scorefns (analogous to the Arrays class in java.util).

It would provide a set of static methods that add functionality for objects that satisfy the Scoring interface:
public class Scorefns {
    //has methods that exploit Scoring interface
    public static int scoreMax(Scoring[] theArray) {
        // returns array position of highest score ele
        int highPos = 0;
        for (int j = 1; j < theArray.length; j++) {
            if (theArray[j].getScore() > theArray[highPos].getScore()) {
                highPos = j;
            }
        }
        return highPos;
    }
}
public static double scoreAvg(Scoring[] theArray){

    // returns avg score
    int scores = 0;
    for(Scoring s : theArray) {
        scores = scores + s.getScore();
    }
    return ((double)scores / theArray.length);
}

Polymorphism: another manifestation, with inheritance

Look at this interaction, in the DrJava interactions pane:

> Object o = new Object();
> o.toString()
"java.lang.Object@55641ee0"
Now suppose the Infant class includes a toString method

```java
public class Infant{

... blah blah

    public String toString(){return name + age;}
}
```
Now:

> Object x = new Infant("jill", 9);
> x.toString()

- What happens?

- Issue: which version of toString gets used
Object x = new Infant("jill", 9);
x.toString()

What happens?

Issue: which version of toString gets used

Answer:

"jill9"

Method used is determined by the object, not by the reference
Another notion: an **abstract class**

At one extreme -> full-blown, concrete classes
Other extreme -> interfaces: everything disembodied

**In the middle**: abstract classes: classes that

1) Cannot be instantiated

2) Generally have at least one method marked “abstract”

3) You make “real” classes out of them by extending them, providing bodies for the abstract methods.
One way to think of an abstract class:

it’s a fancy sound system, all ready to go, except that the component that provides the sound isn’t there, although the wire to the sound component (an iPod?) is ready to be plugged in.

You build a “concrete” system by extending what you have - you add a sound source.
import java.util.Scanner;
import java.io.*;

public abstract class LineReader{

  String fileName; // external file name
  Scanner scan; // for reading from file

  public LineReader(String f) throws IOException{
    fileName = f;
    scan = new Scanner(new FileReader(fileName));
  }
}
public void readLines(){
    while(scan.hasNext()){
        processLine(scan.nextLine());
    }
    scan.close();
}

public abstract void processLine(String line);
import java.io.*;
import java.util.*;

public class NewEcho extends LineReader {

   public NewEcho(String f) throws IOException {
      super(f);
   }

   public void processLine(String line){
      System.out.println(line);
   }
}
import java.util.*;
import java.io.*;
public class LineDriver{
public static void main(String[] args){
try{
    Scanner scan = new Scanner(System.in);
    System.out.println("Enter name of text file");
    String fileName = scan.next();
    NewEcho r = new NewEcho(fileName);
    r.readLine();
}
catch(Exception e){e.printStackTrace();}
}
public abstract class JobTimer {

public abstract void doJob();
    // does some job, to be named later in a subclass

public void runJob() {
    //garbage collector to make more memory available
    System.gc();
    long s1 = System.currentTimeMillis();
    doJob();
    doJob();
    long s2 = System.currentTimeMillis();
    long runTime = (s2 - s1);
    System.out.println("running time in milliseconds: " + runTime);
}
}
public class AddTimer extends JobTimer {
    // how many times operation performed
    private long numOperations = 10000000;

    // implementation of the abstract method

    public void doJob(){
        long k = 0;
        int result = 0, operand = 12345;
        while(k < numOperations){
            result = operand + operand;
            k++;
        }
    }
}
public static void main(String[] args) {
    AddTimer a = new AddTimer();
    a.runJob();
}

running time in milliseconds: 186