Announcements

CS 187- now two discs on Wed, 9:05, and 10:10
Program 7 due next Wednesday
OWL 10 due this Wednesday (tomorrow)
4/16 - embedded 11 due
4/23 - embedded 12 due
4/28 - embedded 13 due
Final - May 11, 130, Totman
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

• An 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”

```java
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);
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);
```

Tricky: It’s binary - compares two objects, the **calling** object, and the **parameter** object

Note: the parameter is completely general: it’s of type **Object**. This is something we’ll have to deal with.
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 ordering: lexicographic (alphabetic)

So: "boy" < "girl"
Informally, “donkey” < “monkey”

Suppose d = “donkey”; m = “monkey”; z = “zebra”;  

Then:

    d.compareTo(m) returns a value < 0
    m.compareTo(d) returns a value > 0
    d.compareTo(d)
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)
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) -> negative value
b.compareTo(c) -> negative value
c.compareTo(a) -> positive value
c.compareTo(c) -> zero
Notice that if
\[
a \text{compareTo}(b) < 0
\]
Then
\[
b \text{compareTo}(a) > 0
\]

String a1 = “dog”; String a2 = “cat”; 

\[
a1 \text{compareTo}(a2) > 0 \text{ (“dog” later than “cat”)}
\]
\[
a2 \text{compareTo}(a1) < 0 \text{ (“cat” earlier than “dog”)}
\]
public class Infant {

--->

public class Infant implements Comparable{

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;
    }
    ...
    public int compareTo(Object other){ // by age
        int b = ((Infant)other).age; // cast req!
        int a = this.age;
        return(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 String class
Why is Comparable valuable?

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

```java
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.*;

public 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:

   public class Infant implements Comparable{
       ...
       public int compareTo(Object other){...}
   }

And then, if kids is an array of Infants:

   java.util.Arrays.sort(kids);

Does the trick
Rethinking the cell model for variables

```java
int num = 5;

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

Scoring s;
```

```
5

num

1224

kid

3376

s

s

3376

1224
```
Rethinking the cell model for variables

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.
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;
    }
}
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();
    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