CSC 222: Object-Oriented Programming

Fall 2017

Java interfaces & polymorphism

- Comparable interface
- defining & implementing an interface
- generic interfaces, classes & methods
- polymorphism
- List interface, Collections.sort

Collections utilities

Java provides many useful routines for manipulating collections such as ArrayLists

- `Collections` is a utility class (contains only static methods)
- e.g., `Collections.sort(anylist)` will sort an ArrayList of objects

```
ArrayList<String> words =
    new ArrayList<String>();

words.add("foo");
words.add("bar");
words.add("boo");
words.add("baz");
words.add("biz");

Collections.sort(words);

System.out.println(words);

------------------------------

→ [bar, baz, biz, boo, foo]
```

```
ArrayList<Integer> nums =
    new ArrayList<Integer>();

nums.add(5);
nums.add(3);
nums.add(12);
nums.add(4);


Collections.sort(nums);

System.out.println(nums);

------------------------------

→ [3, 4, 5, 12]
```

how can this one method work for ArrayLists of different types?

Interfaces

in the real world, an interface is a description of how an object can be used

- e.g., USB interface
  DVD interface
  headphone interface
  Phillips-head screw interface

interfaces allow for the development of general-purpose devices

- e.g., as long as electronic device follows USB specs, can be connected to laptop
  - as long as player follows DVD specs, can play movie
  - …

Java allows a developer to use and define software interfaces

- an interface defines a required set of methods
- any class that "implements" that interface must provide those methods exactly
- e.g., `java.util.Comparable`

3

Comparable interface

```
public interface Comparable<T> {
    int compareTo(T other);
}
```

- any class `T` that implements the `Comparable<T>` interface must provide a `compareTo` method, that takes an object of class `T`, compares, and returns an int

```
val1.compareTo(val2)
```
returns a negative number if val1 < val2
returns 0 if val1 == val2
returns a positive number if val1 > val2

- `String` implements the `Comparable<String>` interface:

```
String str1 = "foo";  String str2 = "bar";
str1.compareTo(str2)
str2.compareTo(str1)
```
returns a positive int (since "foo" > "bar")
returns a negative int (since "bar" < "foo")

- `Integer` implements the `Comparable<Integer>` interface

```
Integer num1 = 12;  Integer num2 = 23;
num1.compareTo(num2)
num1.compareTo(num1)
```
returns a negative int (since 12 < 23)
returns 0 (since 12 == 12)

4

Implementing an interface

the String and Integer class definitions specify that they are Comparable

- "implements Comparable<T>" appears in the header for the class

```
public class String implements Comparable<String> {
    . . .

    public int compareTo(String other) {
        // code that returns either neg, 0, or pos int
    }

    . . .
}
```

```
public class Integer implements Comparable<Integer> {
    . . .

    public int compareTo(Integer other) {
        // code that returns either neg, 0, or pos int
    }

    . . .
}
```

5

Implementing an interface

user-defined classes can similarly implement an interface

- must add "implements XXX" to header
- must provide the required methods

```
public class UFOsighting implements Comparable<UFOsighting> {
  private String date;
  private String state;
  private String city;

  .
  .
  .

  public int compareTo(UFOsighting other) {
    return this.date.compareTo(other.date);
  }
}
```

- via the compareTo method, you determine how objects of that class are sorted
- could sort by date
- could sort by city+state

```
public class UFOsighting implements Comparable<UFOsighting> {
  private String date;
  private String state;
  private String city;

  .
  .
  .

  public int compareTo(UFOsighting other) {
    String thisLoc = this.city + ", " + this.state;
    String otherLoc = other.city + ", " + other.state;
    return thisLoc.compareTo(otherLoc);
  }
}
```

6

Sorting sightings

`Collections.sort` is a static method that takes a list of `Comparable` objects, so can now sort `UFOsightings`

- if compareTo uses date, then showByState will show all sightings from a state in chronological order
- if it uses city+state, then will show sightings in alphabetical order

```
public class UFOlookup {
  private ArrayList<UFOsighting> sightings;

  public UFOlookup(String filename) {
    this.sightings = new ArrayList<UFOsighting>();
    try {
      Scanner infile = new Scanner(new File(filename));

      while (infile.hasNextLine()) {
        String date = infile.next();
        String state = infile.next();
        String city = infile.nextLine().trim();
        UFOsighting sight = new UFOsighting(date, state, city);
        this.sightings.add(sight);
      }
      infile.close();

      Collections.sort(this.sightings);
    }
    catch (java.io.FileNotFoundException e) {
      System.out.println("No such file: " + filename);
    }
  }


  public void showByState(String state) {
    int count = 0;
    for (UFOsighting sight : this.sightings) {
      if (sight.getState().equals(state)) {
        System.out.println(sight);
        count++;
      }
    }
    System.out.println();
    System.out.println("# of sightings = " + count);
  }

  . . .
}
```

7

HW5: Extending CityStats & CityLookup

make CityStats Comparable by

- adding to class header: `implements Comparable<CityStats>`
- adding a `compareTo` method that returns
  - ✓ a negative int if the COLI of this object is < the COLI of the parameter
  - ✓ 0 if their COLI's are the same
  - ✓ a positive int if the COLI of this object is > the COLI of the parameter

modify CityLookup to take advantage

- sort the list of `CityStats` in the constructor (so ordered by COLI)
- add a `lookupByRank` method, which takes a rank and returns info on the city with that rank (positive rank means low→high, negative rank means high→low)

```
cities.lookupByRank(1)   → "1) Harlingen, TX: 82.9"
cities.lookupByRank(-1)  → "334) New York (Manhattan), NY: 215.4"
```

- add `showLowest` and `showHighest` methods, which show the specified number of lowest or highest cities (by COLI)

```
cities.showLowest(3)
1) Harlingen, TX: 82.9
2) Pryor Creek, OK: 84.5
3) McAllen, TX: 85.2
```

```
cities.showHighest(5)
334) New York (Manhattan), NY: 215.4
333) New York (Brooklyn), NY: 180.7
332) Honolulu, HI: 165.4
331) San Francisco, CA: 163.5
330) New York (Queens), NY: 158.2
```

8

Generics

note that the Comparable interface is generic

```
public interface Comparable<T> {
    int compareTo(T other);
}
```

- T is a generic parameter that must be instantiated

```
public class UFOsighting implements Comparable<UFOsighting> {
    . . .
}

public class CityStats implements Comparable<CityStats> {
    . . .
}
```

- when the generic is instantiated, the specified value replaces the parameter in the code (e.g., UFOsighting → T or CityStats → T)

9

Generic classes

similarly, classes can be generic

```
public class ArrayList<T> {
    private T[] items;

    . . .

    public T get(int index) throws NoSuchElementException {
        if (index < 0 || index >= items.length) {
            throw new NoSuchElementException();
        }
        return items[index];
    }
}
```

- again, when you instantiate the generic, the specified type replaces the parameter
  `ArrayList<CityStats> cities;` → CityStats replaces T everywhere

10

Generic methods

methods can also be generic, e.g., Collections.reverse

```
public class Collections {
    . . .

    public static <T> void reverse(List<T> arblist) {
        for (int i = 0; i < arblist.size()/2; i++) {
            T temp = arblist.get(i);
            arblist.set(i, arblist.get(arblist.size()-i-1);
            arblist.set(arblist.size()-i-1, temp);
        }
    }

    . . .
}
```

- List is an interface that specifies basic operations on lists of items
- ArrayList implements the List interface (MORE LATER)
  ➔ can call Collections.reverse on any ArrayList (regardless of type stored)

11

Collections.sort

finally, Collections.sort combines generics and interfaces

```
public class Collections {
  . . .

  public static <T extends Comparable<? super T>> void sort(List<T> arblist) {
    // IMPLEMENTS MERGE SORT
}
```

- <T extends Comparable<? super T>> effectively says that the type stored in the List must either implement Comparable<T> or else be derived from a class that does so (MORE LATER)

```
ArrayList<String> words;
…
Collections.sort(words);
```
→ OK, since String implements Comparable<String>

```
ArrayList<Die> dice;
...
Collections.sort(dice);
```
→ ILLEGAL, since Die is not Comparable

12

Interfaces for code reuse

in general, interfaces are used to express the commonality between classes

- e.g., all Comparable classes have a compareTo method for comparing two objects

another example: suppose a school has two different types of course grades

LetterGrades:
A → 4.0 grade points per hour
B+ → 3.5 grade points per hour
B → 3.0 grade points per hour
C+ → 2.5 grade points per hour
. . .

PassFailGrades:
pass → 4.0 grade points per hour
fail → 0.0 grade points per hour

- for either type, the rules for calculating GPA are the same

  GPA = (total grade points over all classes) / (total number of hours)

13

Grade interface

can define an interface to identify the behaviors common to all grades

```
public interface Grade {
  int hours();                // returns # of hours for the course
  double gradePoints();       // returns number of grade points earned
}
```

```
class LetterGrade implements Grade {
   private int courseHours;
   private String courseGrade;

   public LetterGrade(String g, int hrs) {
      this.courseGrade = g;
      this.courseHours = hrs;
   }

   public int hours() {
      return this.courseHours;
   }

   public double gradePoints() {
      if (this.courseGrade.equals("A")) {
         return 4.0*this.courseHours;
      }
      else if (this.courseGrade.equals("B+")){
         return 3.5*this.courseHours;
      }
      . . .
   }
}
```

```
class PassFailGrade implements Grade {
   private int courseHours;
   private boolean coursePass;

   public PassFailGrade(boolean g, int hrs) {
      this.coursePass = g;
      this.courseHours = hrs;
   }

   public int hours() {
      return this.courseHours;
   }

   public double gradePoints() {
      if (this.coursePass) {
         return 4.0*this.courseHours;
      }
      else {
         return 0.0;
      }
   }

}
```

14

Polymorphism

an interface type encompasses all implementing class types

- can declare variable of type Grade, assign it a LetterGrade or PassFailGrade
- but, can't create an object of interface type

```
Grade csc221 = new LetterGrade("A", 3);          // LEGAL

Grade mth245 = new PassFailGrade(true, 4);       // LEGAL

Grade his101 = new Grade();                      // ILLEGAL
```

polymorphism: behavior can vary depending on the actual type of an object

- LetterGrade and PassFailGrade provide the same methods
- the underlying state and method implementations are different for each
- when a method is called on an object, the appropriate version is executed

```
double pts1 = csc221.gradePoints();              // CALLS LetterGrade METHOD

double pts2 = mth245.gradePoints();              // CALLS PassFailGrade METHOD
```

15

Grade interface

to implement an interface, must provide all the required methods

- can certainly add other methods as well

```
class LetterGrade implements Grade {
   private int courseHours;
   private String courseGrade;

   public LetterGrade(String g, int hrs) {
      this.courseGrade = g;
      this.courseHours = hrs;
   }

   public int hours() {
      return this.courseHours;
   }

   public double gradePoints() {
      if (this.courseGrade.equals("A")) {
         return 4.0*this.courseHours;
      }
      else if (this.courseGrade.equals("B+")){
         return 3.5*this.courseHours;
      }
      . . .
   }

   public String getLetter() {
      return this.courseGrade;
   }
}
```

```
class PassFailGrade implements Grade {
   private int courseHours;
   private boolean coursePass;

   public PassFailGrade(boolean g, int hrs) {
      this.coursePass = g;
      this.courseHours = hrs;
   }

   public int hours() {
      return this.courseHours;
   }

   public double gradePoints() {
      if (this.coursePass) {
         return 4.0*this.courseHours;
      }
      else {
         return 0.0;
      }
   }

   public String getPassFail() {
      return this.coursePass;
   }
}
```

16

Interface restrictions

if you assign an object to an interface type, can only call methods defined by the interface

```
LetterGrade csc221 = new LetterGrade("B", 3);

String g1 = csc221.getLetter();       // LEGAL: assigns g1 = "B"


Grade csc222 = new LetterGrade("A", 3);

String g2 = csc222.getLetter();       // ILLEGAL - Grade INTERFACE DOES
                                      //   NOT SPECIFY getLetterGrade

String g3 = ((LetterGrade)csc222).getLetter()
                                      // HOWEVER, CAN CAST BACK TO
                                      //   ORIGINAL CLASS, THEN CALL
                                      // IF CAST TO WRONG CLASS, AN
                                      //   EXCEPTION IS THROWN
```

consider Collections.sort:

- since the parameter is specified as Comparable<T>, the only method that can be called on the parameter in Collections.sort is compareTo

17

Polymorphism (cont.)

using polymorphism, can define a method that will work on any list of grades

```
public double GPA(ArrayList<Grade> grades) {
   double pointSum = 0.0;
   int hourSum = 0;
   for (int i = 0; i < grades.size(); i++) {
      Grade nextGrade = grades.get(i);
      pointSum += nextGrade.gradePoints();
      hourSum += nextGrade.hours();
   }
   return pointSum/hourSum;
}

-------------------------------------------------------------

Grade csc221 = new LetterGrade("A", 3);
Grade mth245 = new LetterGrade("B+", 4);
Grade his101 = new PassFailGrade(true, 1);

ArrayList<Grade> classes = new ArrayList<Grade>();

classes.add(csc221);
classes.add(mth245);
classes.add(his101);

System.out.println("GPA = " + GPA(classes) );
```

18

Gradebook

likewise, you could create a class with an ArrayList of Grades

- if addGrade is called with "pass" or "fail", it adds a PassFailGrade
- otherwise, a LetterGrade is added
- since both are Grades, it is fine to mix them in the ArrayList

```
public class GradeBook {
  private ArrayList<Grade> grades;

  public GradeBook() {
      this.grades = new ArrayList<Grade>();
  }

  public void addGrade(String g, int h) {
    if (g.equalsIgnoreCase("pass")) {
        this.grades.add(new PassFailGrade(true, h));
    }
    else if (g.equalsIgnoreCase("fail")) {
        this.grades.add(new PassFailGrade(false, h));
    }
    else {
        this.grades.add(new LetterGrade(g, h));
    }
  }

  public double GPA() {
     double pointSum = 0.0;
     int hourSum = 0;
     for (Grade nextGrade : this.grades) {
        pointSum += nextGrade.gradePoints();
        hourSum += nextGrade.hours();
     }
     return pointSum/hourSum;
  }
}
```


List interface

ArrayList implements the List interface

```
public interface List<T> {
    boolean add(T obj);
    boolean add(int index, T obj);
    void clear();
    boolean contains(Object obj);
    T get(int index);
    T remove(int index);
    boolean remove(T obj)
    T set(int index, T obj);
    int size();
    . . .
}
```

other types of Lists are possible, with different performance tradeoffs

- e.g., `LinkedList` stores items in a linked structure (more in CSC 321)
  advantage: can add/remove from either end in O(1) time
  disadvantage: get operation is O(N)

  if you knew you were only going to be adding at end and no searching was required, then a LinkedList would be a better choice

Example: Dictionary

can use the List interface to write a more generic Dictionary

the field is declared to be of type `List`

- if choose to instantiate with an `ArrayList`, its methods will be called
- if choose to instantiate with a `LinkedList`, its methods will be called

```
import java.util.List;
import java.util.ArrayList;
import java.util.Scanner;
import java.io.File;

public class Dictionary {
    private List<String> words;

    public Dictionary() {
        this.words = new ArrayList<String>();
    }

    public Dictionary(String filename) {
        this();

        try {
            Scanner infile = new Scanner(new File(filename));
            while (infile.hasNext()) {
                String nextWord = infile.next();
                this.words.add(nextWord.toLowerCase());
            }
        }
        catch (java.io.FileNotFoundException e) {
            System.out.println("FILE NOT FOUND");
        }
    }

    public void add(String newWord) {
        this.words.add(newWord.toLowerCase());
    }

    public void remove(String oldWord) {
        this.words.remove(oldWord.toLowerCase());
    }

    public boolean contains(String testWord) {
        return this.words.contains(testWord.toLowerCase());
    }
}
```

21

Collections class

`java.util.Collections` provides a variety of static methods on Lists

```
static <T extends Comparable<? super T>> void sort(List<T> list);
static <T extends Comparable<? super T>> int binarySearch(List<T> list, T key);
static <T extends Comparable<? super T>>  max(List<T> list);
static <T extends Comparable<? super T>>  min(List<T> list);
static <T> void reverse(List<T> list);
static <T> void shuffle(List<T> list);
```

since the `List` interface is specified, can make use of polymorphism

- these methods can be called on both `ArrayLists` and `LinkedLists`

```
ArrayList<String> words = new ArrayList<String>();
…
Collections.sort(words);

LinkedList<Integer> nums = new LinkedList<Integer>();
…
Collections.sort(nums);
```

22