

CSC 222: Computer Programming II

Spring 2005

Java interfaces & polymorphism

- Comparable interface
- defining an interface
- implementing an interface
- polymorphism
- action listeners, timer events
- inner classes

1

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?

2

Interfaces

Java libraries make extensive use of interfaces

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
...

3

Java interfaces

Java allows a developer to define software interfaces

- an interface defines a required set of methods
- any class that "implements" that interface must provide those methods exactly

e.g., the following interface is defined in `java.util.Comparable`

```
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`
- `String` implements the `Comparable<String>` interface:
`str1.compareTo(str2)` returns -1 if `str1 < str2`, 0 if =, 1 if >
- `Integer` implements the `Comparable<Integer>` interface
`num1.compareTo(num2)` returns -1 if `num1 < num2`, 0 if =, 1 if >

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 -1, 0, or 1
    }

    . . .
}
```

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

    public int compareTo(Integer other)
    {
        // code that returns either -1, 0, or 1
    }

    . . .
}
```

5

Implementing an interface

user-defined classes can similarly implement an interface

- must add "implements `XXX`" to header
- must provide the required methods (here, `compareTo`)

```
public class Name implements Comparable<Name>
{
    private String firstName;
    private String lastName;

    public Name(String first, String last)
    {
        firstName = first;
        lastName = last;
    }

    public int compareTo(Name other)
    {
        int lastTest = lastName.compareTo(other.lastName);
        if (lastTest != 0) {
            return lastTest;
        }
        else {
            return firstName.compareTo(other.firstName);
        }
    }

    public String toString()
    {
        return firstName + " " + lastName;
    }

    . . .
}
```

6

Generic methods

methods can take parameters that are specified by an interface

```
public String which(Comparable c1, Comparable c2)
{
    int result = c1.compareTo(c2);
    if (result < 0) {
        return "LESS THAN";
    }
    else if (result > 0) {
        return "GREATER THAN";
    }
    else {
        return "EQUAL TO";
    }
}
```

can call this method with 2 objects whose type implements the Comparable interface

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

```
ArrayList<Name> names = new ArrayList<Name>();
names.add(new Name("Joe", "Smith"));
names.add(new Name("Jane", "Doe"));
names.add(new Name("Chris", "Doe"));

Collections.sort(names);

System.out.println(names);
```

7

Interfaces for code reuse

interfaces are used to express the commonality between classes

- e.g., 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

$$\text{GPA} = (\text{total grade points over all classes}) / (\text{total number of hours})$$

8

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) {
        courseGrade = g;
        courseHours = hrs;
    }

    public int hours() {
        return courseHours;
    }

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

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

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

    public int hours() {
        return courseHours;
    }

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

9

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
```

10

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>(); // OK, SINCE CONTENTS
                                                    // ARE NOT CREATED YET

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

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

11

Interface restrictions

interestingly enough, interface generalization does not apply to lists

```
ArrayList<Grade> classes = new ArrayList<LetterGrade>(); // ILLEGAL
```

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

- e.g., suppose LetterGrade class had additional method, getLetterGrade

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

String g1 = csc221.getLetterGrade(); // ILLEGAL - Grade INTERFACE DOES
                                     // NOT SPECIFY getLetterGrade

String g2 = ((LetterGrade)csc221).getLetterGrade()
           // HOWEVER, CAN CAST BACK TO
           // ORIGINAL CLASS, THEN CALL
           // IF CAST TO WRONG CLASS, AN
           // EXCEPTION IS THROWN
```

12

Interfaces & event handling

interfaces are used extensively in handling Java events such as button clicks, text entries, mouse movements, etc.

```
public interface ActionListener
{
    void actionPerformed(ActionEvent event);
}
```

you can define classes that implements the ActionListener class

- specify the code to be executed when some event occurs

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

class Timeout implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        System.out.println("Sorry, time's up.");
        System.exit(0);
    }
}
```

13

Timer class

a simple event-driven class is Timer, defined in the Java Swing library

- construct by specifying a time duration (msec) & action listener object
- start and stop methods control the timer
- when reach time duration, an event is triggered and the action listener reacts

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.Timer;
import java.util.Scanner;

public class TimeDemo {
    private static final int TIME_LIMIT = 10000;

    public static void main(String[] args) {
        Timer t = new Timer(TIME_LIMIT, new Timeout());

        System.out.println("What is the capital of Nebraska? ");
        t.start();

        Scanner input = new Scanner(System.in);
        String userEntry = input.next();
        t.stop();

        if (userEntry.equals("Lincoln")) {
            System.out.println("That is correct!");
        }
        else {
            System.out.println("No, it's Lincoln.");
        }
    }
}
```

14

Inner classes

when a class is specific to a particular method, can declare the class inside the method

- Timeout class needed for this main method, not generally applicable
- define it inside main → local to that method
- saves you from having to create separate class files

USE ONLY WHEN THE INNER CLASS HAS NO POSSIBLE USE OUTSIDE THE METHOD

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.Timer;
import java.util.Scanner;

public class TimeDemo {
    private static final int TIME_LIMIT = 10000;

    public static void main(String[] args) {
        class Timeout implements ActionListener {
            public void actionPerformed(ActionEvent event) {
                System.out.println("Sorry, time's up.");
                System.exit(0);
            }
        }

        Timer t = new Timer(TIME_LIMIT, new Timeout());

        System.out.println("What is the capital of Nebraska? ");
        t.start();

        Scanner input = new Scanner(System.in);
        String userEntry = input.next();
        t.stop();

        if (userEntry.equals("Lincoln")) {
            System.out.println("That is correct!");
        }
        else {
            System.out.println("No, it's Lincoln.");
        }
    }
}
```