Java interfaces & polymorphism

- Comparable interface
- defining an interface
- implementing an interface
- generic 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

```java
ArrayList<String> words = new ArrayList<String>();
words.add("foo");
words.add("bar");
words.add("koo");
words.add("baz");
words.add("biz");
Collections.sort(words);
System.out.println(words);
```

- `\[\text{bar, baz, biz, boo, foo}\]`

```java
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

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
    ...

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 >
Implementing an interface

the String and Integer class definitions specify that they are Comparable

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

```java
public class String implements Comparable<String> {
    ...  
    public int compareTo(String other) {
        // code that returns either -1, 0, or 1
    }
    ...  
}
```

```java
public class Integer implements Comparable<Integer> {
    ...  
    public int compareTo(Integer other) {
        // code that returns either -1, 0, or 1
    }
    ...  
}
```

Implementing an interface

user-defined classes can similarly implement an interface

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

```java
public class Name implements Comparable<Name> {
    private String firstName;
    private String lastName;
    public Name(String first, String last) {
        this.firstName = first;
        this.lastName = last;
    }
    public int compareTo(Name other) {
        int lastTest = this.lastName.compareTo(other.lastName);
        if (lastTest != 0) {
            return lastTest;
        } else {
            return this.firstName.compareTo(other.firstName);
        }
    }
    public String toString() {
        return firstName + " " + lastName;
    }
    ...  
}
```
Generic methods

methods can take parameters that are specified by an interface

```java
public static <T extends Comparable<T>> String which(T c1, T 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

ArrayList<Name> names = new ArrayList<Name>();

Collections.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);

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

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

  \[ \text{GPA} = \frac{\text{total grade points over all classes}}{\text{total number of hours}} \]
Grade interface

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

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

```java
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;
        } else {
            return 2.0 * this.courseHours;
        }
    }
}
```

```java
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;
        }
    }
}
```

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

```java
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

```java
double pts1 = csc221.gradePoints(); // CALLS LetterGrade METHOD
double pts2 = mth245.gradePoints(); // CALLS PassFailGrade METHOD
```
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
```

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));
```
List interface

ArrayList implements the List interface

```java
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

```java
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 (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());
    }
}
```
**Collections class**

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

```java
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>> T max(List<T> list);
static <T extends Comparable<? super T>> 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`

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

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