CS 12b
Advanced Programming Techniques

ArrayLists
Wrapper classes
Wrapper classes

✓ ArrayLists only contain objects, and primitive values are not objects.
  • e.g. ArrayList<int> is not legal

✓ If you want to store primitives in an ArrayList, you must declare it using a "wrapper" class as its type.

<table>
<thead>
<tr>
<th>Primitive type</th>
<th>Wrapper class</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Integer</td>
</tr>
<tr>
<td>double</td>
<td>Double</td>
</tr>
<tr>
<td>char</td>
<td>Character</td>
</tr>
<tr>
<td>boolean</td>
<td>Boolean</td>
</tr>
</tbody>
</table>

• example:
  ArrayList<Integer> list = new ArrayList<Integer>();
Wrapper example

The following list stores int values:

```java
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(13);
list.add(47);
list.add(15);
list.add(9);
int sum = 0;
for (int n : list) {
    sum += n;
}
System.out.println("list = "+list);
System.out.println("sum = "+sum);
```

Output:

```plaintext
list = [13, 47, 15, 9]
sum = 84
```

Though you must say Integer when declaring the list, you can refer to the elements as type int afterward.
Wrapper example

✓ The following list stores int values:

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(13);
list.add(47);
list.add(15);
list.add(9);
int sum = 0;
for (int n : list) {
    sum += n;
}
System.out.println("list = "+list);
System.out.println("sum = "+sum);
```

✓ Output:

```
list = [13, 47, 15, 9]
sum = 84
```

✓ Though you must say Integer when declaring the list, you can refer to the elements as type int afterward.
Boxing/Unboxing

✓ Java automatically converts between the two using techniques known as **boxing** and **unboxing**.

✓ **boxing**: automatic conversion from primitive data to a wrapper object of the appropriate type
  ▪ an int boxed to an Integer
  ▪ e.g., list.add(13);

✓ **unboxing**: automatic conversion from a wrapper object to its corresponding primitive data
  ▪ an Integer unboxed to yield an int
  ▪ e.g., int sum = list.get(0) + list.get(1);
The Comparable interface
Natural ordering

✓ Many types have a notion of a natural ordering that describes whether one value of that type is "less than" or "greater than" another:
  ▪ int, double: numeric value
  ▪ String: lexical (alphabetical) order

✓ Not all types have a natural ordering:
  ▪ Point: How would they be ordered? By y? By x? Distance from origin?
  ▪ GroceryList: What makes one list "less than" another?
Uses of natural ordering

✓ **An ArrayList of orderable values can be sorted using the Collections.sort method:**

```java
ArrayList<String> words = new ArrayList<String>();
words.add("four");
words.add("score");
words.add("and");
words.add("seven");
words.add("years");
words.add("ago");

// show list before and after sorting
System.out.println("before sort, words = " + words);
Collections.sort(words);
System.out.println("after sort, words = " + words);
```

✓ **Output:**

before sort, words = [four, score, and, seven, years, ago]
after sort, words = [ago, and, four, score, seven, years]
Comparable interface

The natural ordering of a class is specified through the `compareTo` method of the `Comparable` interface:

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

- **Classes such as** `String` **and** `Integer` **implement** `Comparable`.
- **`compareTo`** **returns an integer that is** `< 0`, `> 0`, **or** `0`:

<table>
<thead>
<tr>
<th>Relationship</th>
<th>Primitive comparison</th>
<th>Object comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>less than</td>
<td>if (x &lt; y) {</td>
<td>if (x.compareTo(y) &lt; 0)</td>
</tr>
<tr>
<td>less than or equal</td>
<td>if (x &lt;= y) {</td>
<td>if (x.compareTo(y) &lt;= 0)</td>
</tr>
<tr>
<td>equal</td>
<td>if (x == y) {</td>
<td>if (x.compareTo(y) == 0)</td>
</tr>
<tr>
<td>not equal</td>
<td>if (x != y) {</td>
<td>if (x.compareTo(y) != 0)</td>
</tr>
<tr>
<td>greater than</td>
<td>if (x &gt; y) {</td>
<td>if (x.compareTo(y) &gt; 0)</td>
</tr>
<tr>
<td>greater or equal</td>
<td>if (x &gt;= y) {</td>
<td>if (x.compareTo(y) &gt;= 0)</td>
</tr>
</tbody>
</table>
How to compare objects

✔ For any class that implements the Comparable interface
  ▪ You can compare to objects with the compareTo method
    String s1 = "hello";
    String s2 = "world";
    if (s1.compareTo(s2)<0) {
        System.out.println("S1 less than S2");
    }
  ▪ You cannot use relational operators. The next is illegal:
    String s1 = "hello";
    String s2 = "world";
    if (s1 < s2) {
        System.out.println("S1 less than S2");
    }
Implementing Comparable

✓ You can define a natural ordering for your own class by making it implement the Comparable interface.
  - Comparable is a generic interface, Comparable<T>
  - When implementing it, you must write your class's name in <> after the word Comparable.

- Example:
  public class Point implements Comparable<Point>

- You must also write a method compareTo that compares the current object (the implicit parameter) to a given other object.

- Example:
  public int compareTo(Point p) {
      ...
  }
Comparable implementation

// The CalendarDate class stores information about a single calendar date (month and day but no year).
public class CalendarDate implements Comparable<CalendarDate> {
    private int month;
    private int day;

    public CalendarDate(int month, int day) {
        this.month = month;
        this.day = day;
    }

    public int compareTo(CalendarDate other) {
        if (this.month != other.month) {
            return this.month - other.month;
        } else {
        }
    }
}
Comparable implementation

// The CalendarDate class stores information about a single calendar date
// (month and day but no year).
public class CalendarDate implements Comparable<CalendarDate> {
    private int month;
    private int day;

    public CalendarDate(int month, int day) {
        this.month = month;
        this.day = day;
    }

    public int compareTo(CalendarDate other) {
        if (this.month != other.month) {
            return this.month - other.month;
        } else {

            // Compares this calendar date to another date.
            // Dates are compared by month and then by day.
        }
    }
}
Example Client Program

// Short program that creates a list of the birthdays of the first 5
// US Presidents and that puts them into sorted order.

import java.util.*;

public class CalendarDateTest {
    public static void main(String[] args) {
        ArrayList<CalendarDate> dates = new ArrayList<CalendarDate>();
        dates.add(new CalendarDate(2, 22));
        dates.add(new CalendarDate(10, 30));
        dates.add(new CalendarDate(4, 13));
        dates.add(new CalendarDate(3, 16));
        dates.add(new CalendarDate(4, 28));

        System.out.println("birthdays before sorting = " + dates);
        Collections.sort(dates);
        System.out.println("birthdays after sorting = " + dates);
    }
}
// Short program that creates a list of the birthdays of the first 5
// US Presidents and that puts them into sorted order.

import java.util.*;

class CalendarDateTest {
    public static void main(String[] args) {
        ArrayList<CalendarDate> dates = new ArrayList<CalendarDate>();
        dates.add(new CalendarDate(2, 22));
        dates.add(new CalendarDate(10, 30));
        dates.add(new CalendarDate(4, 13));
        dates.add(new CalendarDate(3, 16));
        dates.add(new CalendarDate(4, 28));

        System.out.println("birthdays before sorting = " + dates);
        Collections.sort(dates);
        System.out.println("birthdays after sorting = " + dates);
    }
}

since CalendarDate implements the Comparable we can use the Collections.sort method
Example Client Program

// Short program that creates a list of the birthdays of the first 5
// US Presidents and that puts them into sorted order.

import java.util.*;

public class CalendarDateTest {
    public static void main(String[] args) {
        ArrayList<CalendarDate> dates = new ArrayList<CalendarDate>();
        dates.add(new CalendarDate(2, 22));
        dates.add(new CalendarDate(10, 30));
        dates.add(new CalendarDate(4, 13));
        dates.add(new CalendarDate(3, 16));
        dates.add(new CalendarDate(4, 28));

        System.out.println("birthdays before sorting = " + dates);
        Collections.sort(dates);
        System.out.println("birthdays after sorting = " + dates);
    }
}

OUTPUT:
birthdays before sorting = [2/22, 10/30, 4/13, 3/16, 4/28]
birthdays after sorting = [2/22, 3/16, 4/13, 4/28, 10/30]
CASE STUDY

Vocabulary Comparison
Problem Description

✓ Develop a program that read two text files and compares their vocabulary
  ▪ Determine the set of words used in each file and compute their overlap (i.e., words that appear in both)

✓ Steps
  1. Version 1: read two files and report the unique words in each. Test with small files
  2. Version 2: Compute overlap between two files. Test with small files
  3. Version 3: Read from large files and analyze the results
Version 1: Compute Vocabulary
Version 1: Compute Vocabulary

✓ Input files: test1.txt, test2.txt
Version 1: Compute Vocabulary

✓ Input files: test1.txt, test2.txt
Version 1: Compute Vocabulary

✓ Input files: test1.txt, test2.txt

✓ How do we read from files?
  Scanner input1 = new Scanner(new File("test1.txt"));
  Scanner input2 = new Scanner(new File("test2.txt"));
Version 1: Compute Vocabulary

✓ Input files: test1.txt, test2.txt

✓ How do we read from files?
   Scanner input1 = new Scanner(new File("test1.txt"));
   Scanner input2 = new Scanner(new File("test2.txt"));

✓ Compute unique vocabulary of each test. How?
   - Read from Scanner every word and add it in the ArrayList<String>
   - Same operation for both files: we need a method
   - ArrayList<String> getWords(Scanner input)

   ArrayList<String> list1 = getWords(input1);
   ArrayList<String> list2 = getWords(input2);
Version 1: Compute Vocabulary
Version 1: Compute Vocabulary

✓ Print the two vocabularies

    System.out.println("list1 = " + list1);
    System.out.println("list2 = " + list2);
Version 1: Compute Vocabulary

✓ Print the two vocabularies
  System.out.println("list1 = " + list1);
  System.out.println("list2 = " + list2);

✓ Implement the getWords method. What should it do?
  ▪ read all the words from the scanner
  ▪ build the list that contains these words
  ▪ eliminate any duplicates

ArrayList<String> words = new ArrayList<String>();
while (input.hasNext()) {
    String next = input.next().toLowerCase();
    if (!words.contains(next)) {
        words.add(next);
    }
}
public class Vocabulary1A {
    public static void main(String[] args) throws FileNotFoundException {
        Scanner input1 = new Scanner(new File("test1.txt"));
        Scanner input2 = new Scanner(new File("test2.txt"));

        ArrayList<String> list1 = getWords(input1);
        ArrayList<String> list2 = getWords(input2);

        System.out.println("list1 = " + list1);
        System.out.println("list2 = " + list2);
    }

    public static ArrayList<String> getWords(Scanner input) {
        ArrayList<String> words = new ArrayList<String>();
        while (input.hasNext()) {
            words.add(input.next());
        }
        return words;
    }
}
✓ **Test 1:**
The wheels on the bus go round and round round and round and round round and round. The wheels on the bus go round and round all through the town.

✓ **Test 2:**
The wipers on the bus go Swish, swish, swish, Swish, swish, swish, Swish, swish, swish. The wipers on the bus go Swish, swish, swish, all through the town.

✓ **Output:**
list1 = [the, wheels, on, bus, go, round, and, round.., all, through, town.]
list2 = [the, wipers, on, bus, go, swish,, swish., all, through, town.]
Let’s add some improvements

✓ Issue 1: program does not ignore differences in punctuation
  ▪ “round” and “round.” are two different words
  ▪ we will solve that later (version 3)

✓ Approach will not work fast for large files
  ▪ contains is increasingly expensive as the list grows
  ▪ for each new word not in the list -> we have to examine all words in the list
  ▪ for each word in the list -> we have to examine half of the words (in average) in the list

✓ We will use another faster approach
  ▪ it requires more memory
New faster approach
New faster approach

✅ We will NOT try to eliminate duplicates as we read words
  - we eliminate the expensive `contains` method calls
New faster approach

✓ We will NOT try to eliminate duplicates as we read words
  ▪ we eliminate the expensive `contains` method calls
✓ We will sort the word list
  ▪ duplicates will appear next to each other
New faster approach

✓ We will NOT try to eliminate duplicates as we read words
  ▪ we eliminate the expensive \texttt{contains} method calls
✓ We will sort the word list
  ▪ duplicates will appear next to each other
✓ We can easily eliminate duplicates by parsing the sorted word list only once
New faster approach

✓ We will NOT try to eliminate duplicates as we read words
  - we eliminate the expensive \textit{contains} method calls

✓ We will sort the word list
  - duplicates will appear next to each other

✓ We can easily eliminate duplicates by parsing the sorted word list only once

✓ Issue: we will use more memory
  - Typical trade-off (memory-speed) in computing
Let's rewrite the `getWords` method.

- We read words, put them in a list and sort the list.

```java
ArrayList<String> words = new ArrayList<String>();
while (input.hasNext()) {
    String next = input.next().toLowerCase();
    words.add(next);
}
Collections.sort(words);
```
Algorithm for eliminating duplicates
Algorithm for eliminating duplicates

Duplicates are now next to each other
Algorithm for eliminating duplicates

✓ Duplicates are now next to each other
✓ One possible algorithm:
Algorithm for eliminating duplicates

✓ Duplicates are now next to each other
✓ One possible algorithm:

```plaintext
for (each i) {
    if (value at i equals value at i+1) {
        remove value at i+1
    }
}
```
Algorithm for eliminating duplicates

✓ Duplicates are now next to each other
✓ One possible algorithm:

```java
for (each i) {
    if (value at i equals value at i+1) {
        remove value at i+1
    }
}
```

✓ Issue: slow because remove will shift values
Algorithm for eliminating duplicates

✓ Duplicates are now next to each other
✓ One possible algorithm:

```java
for (each i) {
    if (value at i equals value at i+1) {
        remove value at i+1
    }
}
```

✓ Issue: slow because remove will shift values
✓ Solution: build a new list that contains only unique words
Solution for fast duplicate elimination
Solution for fast duplicate elimination

✓ Any ideas?
Solution for fast duplicate elimination

✓ Any ideas?
✓ When we see a word that is not equal to the previous one we now we hit a new word
Solution for fast duplicate elimination

✓ Any ideas?
✓ When we see a word that is not equal to the previous one we now we hit a new word
✓ Algorithm

construct a new empty list
add first word to the new list
for (each i) { // i is the index in the old list
    if (value at i does not equal value at i-1) {
        add value at i in new list.
    }
}
Code for our pseudocode

```java
ArrayList<String> result = new ArrayList<String>();
result.add(words.get(0));
for (int i = 1; i < words.size(); i++) {
    if (!words.get(i).equals(words.get(i - 1))) {
        result.add(words.get(i));
    }
}
```
Code for our pseudocode

```java
ArrayList<String> result = new ArrayList<String>();
result.add(words.get(0));
for (int i = 1; i < words.size(); i++) {
    if (!words.get(i).equals(words.get(i - 1))) {
        result.add(words.get(i));
    }
}

✓ Any problems with this code?
```
Code for our pseudocode

```java
ArrayList<String> result = new ArrayList<String>();
result.add(words.get(0));
for (int i = 1; i < words.size(); i++) {
    if (!words.get(i).equals(words.get(i - 1))) {
        result.add(words.get(i));
    }
}
```

✓ Any problems with this code?
✓ What if the input file is empty?
  - You will get an null pointer exception at line 2