Building Java Programs

Chapter 5+5: ArrayList

Chapter outline

- ArrayList
  - basic operations
  - searching for elements
  - wrapper classes

- Comparable interface
  - natural ordering and compareTo
  - implementing Comparable

Lists

- list: an ordered sequence of elements, each accessible by a 0-based index
  - one of the most basic collections of data

These are the slides of the textbook, modified/corrected where necessary.
The chapters have been re-enumerated according to CMPE160.
The ArrayList class

- Class ArrayList\(<E>\) implements the notion of a list using a partially-filled array
- when you want to use ArrayList, remember to import java.util.*;

```java
ArrayList\(<E>\)
```

ArrayList features

- think of it as an auto-resizing array that can hold any type of object, with many convenient methods
- maintains most of the benefits of arrays, such as fast random access
- frees us from some tedious operations on arrays, such as sliding elements and resizing

```java
ArrayList\(<E>\).toString()
```

Generic classes

- **generic class**: A type in Java that is written to accept another type as part of itself.
  - Generic ("parameterized") classes were added to Java to improve the type safety of Java's collections.
    - A parameterized type has one or more other types' names written between < and >.
- ArrayList\(<E>\) is a generic class.
  - The \(<E>\) is a placeholder in which you write the type of elements you want to store in the ArrayList.

```java
ArrayList\(<E>\) words = new ArrayList\(<E>\)();
```

ArrayList vs. array

- array
  ```java
  String[] names = new String[5];
  names[0] = "Jennifer";
  String name = names[0];
  ```
- ArrayList
  ```java
  ArrayList\(<E>\) namesList = new ArrayList\(<E>\)();
  namesList.add("Jennifer");
  String name = namesList.get(0);  ```
Adding elements

- Elements are added dynamically to the list:
  ```java
  ArrayList<String> list = new ArrayList<String>();
  System.out.println("list = " + list);
  list.add("Tool");
  System.out.println("list = " + list);
  list.add("Phish");
  System.out.println("list = " + list);
  list.add("Pink Floyd");
  System.out.println("list = " + list);
  ```

- Output:
  ```java
  list = []
  list = [Tool]
  list = [Tool, Phish]
  list = [Tool, Phish, Pink Floyd]
  ```

Removing elements

- Elements can also be removed by index:
  ```java
  System.out.println("before remove list = " + list);
  list.remove(0);
  System.out.println("after remove list = " + list);
  ```

- Output:
  ```java
  before remove list = [Tool, U2, Phish, Pink Floyd]
  after remove list = [U2, Pink Floyd]
  ```

  - Notice that as each element is removed, the others shift downward in position to fill the hole.
  - Therefore, the second `remove` gets rid of Phish, not U2.

```
index 0 1
value U2 Pink Floyd
```

Searching for elements

- You can search the list for particular elements:
  ```java
  if (list.contains("Phish")) {
    int index = list.indexOf("Phish");
    System.out.println(index + " " + list.get(index));
  }
  if (list.contains("Madonna")) {
    System.out.println("Madonna is in the list");
  } else {
    System.out.println("Madonna is not found.");
  }
  ```

- Output:
  ```java
  2 Phish
  Madonna is not found.
  ```

- `contains` tells you whether an element is in the list or not, and `indexOf` tells you at which index you can find it.

ArrayList methods

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add(value)</code></td>
<td>adds the given value to the end of the list</td>
</tr>
<tr>
<td><code>add(index, value)</code></td>
<td>inserts the given value before the given index</td>
</tr>
<tr>
<td><code>clear()</code></td>
<td>removes all elements</td>
</tr>
<tr>
<td><code>contains(value)</code></td>
<td>returns true if the given element is in the list</td>
</tr>
<tr>
<td><code>get(index)</code></td>
<td>returns the value at the given index</td>
</tr>
<tr>
<td><code>indexOf(value)</code></td>
<td>returns the first index at which the given element appears in the list (or -1 if not found)</td>
</tr>
<tr>
<td><code>lastIndexOf(value)</code></td>
<td>returns the last index at which the given element appears in the list (or -1 if not found)</td>
</tr>
<tr>
<td><code>remove(index)</code></td>
<td>removes value at given index, sliding others back</td>
</tr>
<tr>
<td><code>size()</code></td>
<td>returns the number of elements in the list</td>
</tr>
</tbody>
</table>
ArrayList and for loop

- Recall the enhanced for loop syntax from Chapter 7:
  ```java
  for (<type> <name> : <collection>) {
     <statement(s>);
  }
  ```

- This syntax can be used to examine an ArrayList:
  ```java
  int sum = 0;
  for (String s : list) {
     sum += s.length();
  }
  System.out.println("Total of lengths = " + sum);
  ```

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:
  ```java
  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:
  ```java
  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.
  - Java automatically converts between the two using techniques known as boxing and unboxing.
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?
- ArrayList: 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:

```
Relationship    Primitive comparison           Object comparison
less than        if (x < y) {                if (x.compareTo(y) < 0) {
less than or equal if (x <= y) {           if (x.compareTo(y) <= 0) {
equal           if (x == y) {            if (x.compareTo(y) == 0) {
not equal       if (x != y) {          if (x.compareTo(y) != 0) {
n greater than   if (x > y) {           if (x.compareTo(y) > 0) {
greater or equal if (x >= y) {         if (x.compareTo(y) >= 0) {
```

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:**

```java
public class Point implements Comparable<Point> {
    public int compareTo(Point p) {
        return ...;
    }
}
```

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

**Example:**

```java
public int compareTo(Point p) {
    return ...;
}
```

Copyright 2006 by Pearson Education
Comparable implementation

The following `CalendarDate` class implements `Comparable`:

```java
public class CalendarDate implements Comparable<CalendarDate> {
    private int month; private int day;
    public CalendarDate(int month, int day) {
        this.month = month;
        this.day = day;
    }
    // Compares two dates by month and then by day.
    public int compareTo(CalendarDate other) {
        if (month != other.month) {
            return month - other.month;
        } else {
            return day - other.day;
        }
    }
    public int getMonth() {
        return month;
    }
    public int getDay() {
        return day;
    }
    public String toString() {
        return month + "/" + day;
    }
}
```

Example: Reverse file

- Write a program to reverse the lines of a file and also to reverse the order of the words in each line of the file.
- Use `ArrayLists` to help you.
- Solution available on the course web site.