CSE 143

Lecture 13: Interfaces, Comparable
reading: 9.5 - 9.6, 16.4, 10.2
Related classes

Consider classes for shapes with common features:

- Circle (defined by radius $r$):
  
  area $= \pi r^2$,  
  perimeter $= 2 \pi r$

- Rectangle (defined by width $w$ and height $h$):
  
  area $= w \times h$,  
  perimeter $= 2w + 2h$

- Triangle (defined by side lengths $a$, $b$, and $c$)
  
  area $= \sqrt{s(s-a)(s-b)(s-c)}$
  
  where $s = \frac{1}{2}(a + b + c)$,
  
  perimeter $= a + b + c$

- Every shape has these, but each computes them differently.
Interfaced (9.5)

- **interface**: A list of methods that a class can promise to implement.
  
  - Inheritance gives you an is-a relationship *and* code sharing.
    - A Lawyer can be treated as an Employee and inherits its code.
  
  - Interfaces give you an is-a relationship *without* code sharing.
    - A Rectangle object can be treated as a Shape but inherits no code.
  
  - Analogous to non-programming idea of roles or certifications:
    - "I'm certified as a CPA accountant. This assures you I know how to do taxes, audits, and consulting."
    - "I'm 'certified' as a Shape, because I implement the Shape interface. This assures you I know how to compute my area and perimeter."
Interface syntax

```java
public interface name { 
    public type name(type name, ..., type name);
    public type name(type name, ..., type name);
    ...
    public type name(type name, ..., type name);
}

Example:
public interface Vehicle { 
    public int getSpeed();
    public void setDirection(int direction);
}
```
Shape interface

// Describes features common to all shapes.
public interface Shape {
    public double area();
    public double perimeter();
}

- **Saved as Shape.java**

- **abstract method**: A header without an implementation.
- The actual bodies are not specified, because we want to allow each class to implement the behavior in its own way.
Implementing an interface

public class name implements interface {
    ...
}

- A class can declare that it "implements" an interface.
- The class must contain each method in that interface.

public class Bicycle implements Vehicle {
    ...
}

(Otherwise it will fail to compile.)

Banana.java:1: Banana is not abstract and does not override abstract method area() in Shape
public class Banana implements Shape {
    ^
Interfaces + polymorphism

- Interfaces benefit the *client code* author the most.

- They allow **polymorphism**.
  (the same code can work with different types of objects)

```java
public static void printInfo(Shape s) {
    System.out.println("The shape: " + s);
    System.out.println("area : " + s.area());
    System.out.println("perim: " + s.perimeter());
    System.out.println();
}
...
Circle circ = new Circle(12.0);
Triangle tri = new Triangle(5, 12, 13);
printInfo(circ);
printInfo(tri);
```
Linked vs. array lists

- We have implemented two collection classes:
  - ArrayIntList
    
    | index | 0 | 1 | 2 | 3 |
    |-------|---|---|---|---|
    | value | 42 | -3 | 17 | 9 |
  - LinkedIntList

- They have similar behavior, implemented in different ways. We should be able to treat them the same way in client code.
An IntList interface

// Represents a list of integers.
public interface IntList {
    public void add(int value);
    public void add(int index, int value);
    public int get(int index);
    public int indexOf(int value);
    public boolean isEmpty();
    public void remove(int index);
    public void set(int index, int value);
    public int size();
}

class ArrayIntList implements IntList {
    ...

class LinkedListIntList implements IntList {
    ...
}
public class ListClient {
    public static void main(String[] args) {
        IntList list1 = new ArrayIntList();
        process(list1);
        IntList list2 = new LinkedIntList();
        process(list2);
    }
    public static void process(IntList list) {
        list.add(18);
        list.add(27);
        list.add(93);
        System.out.println(list);
        list.remove(1); // Assuming remove method exists
        System.out.println(list);
    }
}
ADTs as interfaces (11.1)

- **abstract data type (ADT)**: A specification of a collection of data and the operations that can be performed on it.
  - Describes *what* a collection does, not *how* it does it.

- Java's collection framework uses interfaces to describe ADTs:
  - `Collection`, `Deque`, `List`, `Map`, `Queue`, `Set`

- An ADT can be implemented in multiple ways by classes:
  - `ArrayList` and `LinkedList` implement `List`
  - `HashSet` and `TreeSet` implement `Set`
  - `LinkedList`, `ArrayDeque`, etc. implement `Queue`

  - They messed up on `Stack`; there's no `Stack` interface, just a class.
Using ADT interfaces

When using Java's built-in collection classes:

- It is considered good practice to always declare collection variables using the corresponding ADT interface type:

  ```java
  List<String> list = new ArrayList<String>();
  ```

- Methods that accept a collection as a parameter should also declare the parameter using the ADT interface type:

  ```java
  public void stutter(List<String> list) {
      ...
  }
  ```
The Comparable Interface

reading: 10.2
# Collections class

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>binarySearch(list, value)</code></td>
<td>returns the index of the given value in a sorted list (&lt; 0 if not found)</td>
</tr>
<tr>
<td><code>copy(listTo, listFrom)</code></td>
<td>copies <code>listFrom</code>'s elements to <code>listTo</code></td>
</tr>
<tr>
<td><code>emptyList()</code>, <code>emptyMap()</code>, <code>emptySet()</code></td>
<td>returns a read-only collection of the given type that has no elements</td>
</tr>
<tr>
<td><code>fill(list, value)</code></td>
<td>sets every element in the list to have the given value</td>
</tr>
<tr>
<td><code>max(collection)</code>, <code>min(collection)</code></td>
<td>returns largest/smallest element</td>
</tr>
<tr>
<td><code>replaceAll(list, old, new)</code></td>
<td>replaces an element value with another</td>
</tr>
<tr>
<td><code>reverse(list)</code></td>
<td>reverses the order of a list's elements</td>
</tr>
<tr>
<td><code>shuffle(list)</code></td>
<td>arranges elements into a random order</td>
</tr>
<tr>
<td><code>sort(list)</code></td>
<td>arranges elements into ascending order</td>
</tr>
</tbody>
</table>
Ordering and objects

- Can we sort an array of Strings?
  - Operators like < and > do not work with String objects.
  - But we do think of strings as having an alphabetical ordering.

- **natural ordering**: Rules governing the relative placement of all values of a given type.

- **comparison function**: Code that, when given two values A and B of a given type, decides their relative ordering:
  - A < B,  A == B,  A > B
The `compareTo` method (10.2)

- The standard way for a Java class to define a comparison function for its objects is to define a `compareTo` method.

  - Example: in the `String` class, there is a method:
    ```java
    public int compareTo(String other)
    ```

- A call of `A.compareTo(B)` will return:
  - a value < 0 if `A` comes "before" `B` in the ordering,
  - a value > 0 if `A` comes "after" `B` in the ordering,
  - or 0 if `A` and `B` are considered "equal" in the ordering.
Using `compareTo`

- `compareTo` can be used as a test in an `if` statement.

```java
String a = "alice";
String b = "bob";
if (a.compareTo(b) < 0) { // true
    ...
}
```

<table>
<thead>
<tr>
<th>Primitives</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (a &lt; b) { ...</td>
<td>if (a.compareTo(b) &lt; 0) { ...</td>
</tr>
<tr>
<td>if (a &lt;= b) { ...</td>
<td>if (a.compareTo(b) &lt;= 0) { ...</td>
</tr>
<tr>
<td>if (a == b) { ...</td>
<td>if (a.compareTo(b) == 0) { ...</td>
</tr>
<tr>
<td>if (a != b) { ...</td>
<td>if (a.compareTo(b) != 0) { ...</td>
</tr>
<tr>
<td>if (a &gt;= b) { ...</td>
<td>if (a.compareTo(b) &gt;= 0) { ...</td>
</tr>
<tr>
<td>if (a &gt; b) { ...</td>
<td>if (a.compareTo(b) &gt; 0) { ...</td>
</tr>
</tbody>
</table>
**compareTo and collections**

- You can use an array or list of strings with Java's included `binarySearch` method because it calls `compareTo` internally.

  ```java
  String[] a = {"al", "bob", "cari", "dan", "mike"};
  int index = Arrays.binarySearch(a, "dan"); // 3
  ```

- Java's `TreeSet`/`Map` use `compareTo` internally for ordering.

- A call to your `compareTo` method should return:
  - a value < 0 if this object is "before" the other object,
  - a value > 0 if this object is "after" the other object,
  - or 0 if this object is "equal" to the other.
public interface Comparable<E> {
    public int compareTo(E other);
}

- A class can implement the Comparable interface to define a natural ordering function for its objects.

- A call to your compareTo method should return:
  
a value < 0 if this object is "before" the other object,
  a value > 0 if this object is "after" the other object,
  or 0 if this object is "equal" to the other.

- If you want multiple orderings, use a Comparator instead (see Ch. 13.1)
Comparable template

public class name implements Comparable<name> {

    ...

    public int compareTo(name other) {
        ...
    }
}
compareTo tricks

- **delegation trick** - If your object's fields are comparable (such as strings), use their `compareTo` results to help you:

```java
// sort by employee name, e.g. "Jim" < "Susan"
public int compareTo(Employee other) {
    return name.compareTo(other.getName());
}
```

- **toString trick** - If your object's `toString` representation is related to the ordering, use that to help you:

```java
// sort by date, e.g. "09/19" > "04/01"
public int compareTo(Date other) {
    return toString().compareTo(other.toString());
}
```
**compareTo tricks**

- *subtraction trick* - Subtracting related values produces the right result for what you want `compareTo` to return:

  ```java
  // sort by x and break ties by y
  public int compareTo(Point other) {
    if (x != other.x) {
      return x - other.x; // different x
    } else {
      return y - other.y; // same x; compare y
    }
  }
  ```

- The idea:
  - if `x > other.x`, then `x - other.x > 0`
  - if `x < other.x`, then `x - other.x < 0`
  - if `x == other.x`, then `x - other.x == 0`

- **NOTE**: This trick doesn't work for doubles (but see `Math.signum`)