CSE 143
Lecture 14

Interfaces; Abstract Data Types (ADTs)

reading: 9.5, 11.1; 16.4

slides adapted from Marty Stepp and Hélène Martin
http://www.cs.washington.edu/143/
Consider classes for shapes with common features:

- **Circle** (defined by radius $r$):
  \[
  \text{area} = \pi r^2, \quad \text{perimeter} = 2 \pi r
  \]

- **Rectangle** (defined by width $w$ and height $h$):
  \[
  \text{area} = w h, \quad \text{perimeter} = 2w + 2h
  \]

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

  - Every shape has these, but each computes them differently.
**Interfaces (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."
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);
}
// 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 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);
```
We have implemented two collection classes:

- **ArrayIntList**

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>42</td>
<td>-3</td>
<td>17</td>
<td>9</td>
</tr>
</tbody>
</table>

- **LinkedIntList**

  - They have similar behavior, implemented in different ways. We should be able to treat them the same way in client code.
// 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();
}

public class ArrayIntList implements IntList {
    ...

public 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);
        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) {
      ...
  }
  ```
Iterators

reading: 11.1; 15.3; 16.5
Examining sets and maps

- elements of Java Sets and Maps can't be accessed by index
  - must use a "foreach" loop:

    ```java
    Set<Integer> scores = new HashSet<Integer>();
    for (int score : scores) {
        System.out.println("The score is " + score);
    }
    ```

- Problem: foreach is read-only; cannot modify set while looping

  ```java
  for (int score : scores) {
      if (score < 60) {
          // throws a ConcurrentModificationException
          scores.remove(score);
      }
  }
  ```
Iterators (11.1)

- **iterator**: An object that allows a client to traverse the elements of any collection.
  - Remembers a position, and lets you:
    - get the element at that position
    - advance to the next position
    - remove the element at that position
### Iterator methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hasNext()</td>
<td>Returns true if there are more elements to examine</td>
</tr>
<tr>
<td>next()</td>
<td>Returns the next element from the collection (throws a</td>
</tr>
<tr>
<td></td>
<td>NoSuchElementException if there are none left to examine)</td>
</tr>
<tr>
<td>remove()</td>
<td>Removes the last value returned by next() (throws an</td>
</tr>
<tr>
<td></td>
<td>IllegalStateException if you haven't called next() yet)</td>
</tr>
</tbody>
</table>

- **Iterator interface in java.util**
  - Every collection has an iterator() method that returns an iterator over its elements

```java
Set<String> set = new HashSet<String>();
...
Iterator<String> itr = set.iterator();
...```
Set<Integer> scores = new TreeSet<Integer>();
scores.add(94);
scores.add(38); // Jenny
scores.add(87);
scores.add(43); // Marty
scores.add(72);
...

Iterator<Integer> itr = scores.iterator();
while (itr.hasNext()) {
    int score = itr.next();
    System.out.println("The score is " + score);

    // eliminate any failing grades
    if (score < 60) {
        itr.remove();
    }
}
System.out.println(scores); // [72, 87, 94]
A surprising example

• What's bad about this code?

```java
List<Integer> list = new LinkedList<Integer>();
...
(add lots of elements)...
for (int i = 0; i < list.size(); i++) {
    System.out.println(list.get(i));
}
```
Iterators and linked lists

• Iterators are particularly useful with linked lists.
  – The previous code is $O(N^2)$ because each call on `get` must start from the beginning of the list and walk to index $i$.
  – Using an iterator, the same code is $O(N)$. The iterator remembers its position and doesn't start over each time.
### ListIterator

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add(value)</code></td>
<td>inserts an element just after the iterator's position</td>
</tr>
<tr>
<td><code>hasPrevious()</code></td>
<td>true if there are more elements <em>before</em> the iterator</td>
</tr>
<tr>
<td><code>nextIndex()</code></td>
<td>the index of the element that would be returned the next time <code>next</code> is called on the iterator</td>
</tr>
<tr>
<td><code>previousIndex()</code></td>
<td>the index of the element that would be returned the next time <code>previous</code> is called on the iterator</td>
</tr>
<tr>
<td><code>previous()</code></td>
<td>returns the element before the iterator (throws a <code>NoSuchElementException</code> if there are none)</td>
</tr>
<tr>
<td><code>set(value)</code></td>
<td>replaces the element last returned by <code>next</code> or <code>previous</code> with the given value</td>
</tr>
</tbody>
</table>

```java
ListIterator<String> li = myList.listIterator();
```

- **lists have a more powerful ListIterator with more methods**
  - can iterate forwards or backwards
  - can add/set element values (efficient for linked lists)