CSE 143
Lecture 12 (A)

Interfaces

reading: 9.5, 11.1

slides created by Marty Stepp
http://www.cs.washington.edu/143/
Related classes

- Consider the task of writing classes to represent 2D shapes such as Circle, Rectangle, and Triangle.

- Certain operations are common to all shapes:
  - perimeter: distance around the outside of the shape
  - area: amount of 2D space occupied by the shape

  - Every shape has these, but each computes them differently.
Shape area and perimeter

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

• Rectangle (as defined by width $w$ and height $h$):
  - area $= w h$
  - perimeter $= 2w + 2h$

• Triangle (as 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 $= a + b + c$
Common behavior

- Suppose we have 3 classes Circle, Rectangle, Triangle.
  - Each has the methods *perimeter* and *area*.

- We'd like our client code to be able to treat different kinds of shapes in the same way:
  - Write a method that prints any shape's area and perimeter.
  - Create an array to hold a mixture of the various shape objects.
  - Write a method that could return a rectangle, a circle, a triangle, or any other kind of shape.
  - Make a `DrawingPanel` display many shapes on screen.
**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);
}
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 promises to contain each method in that interface.
    (Otherwise it will fail to compile.)

  – Example:
    public class Bicycle implements Vehicle {
        ...
    }

public class Banana implements Shape {
    // haha, no methods! pwned
}

• If we write a class that claims to be a Shape but doesn't implement area and perimeter methods, it will not 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);
```
Linked vs. array lists

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

- 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();
}

public class ArrayIntList implements IntList {
    ...
}

public class LinkedIntList implements IntList {
    ...
}
public class ListClient {
    public static void main(String[] args) {
        ArrayIntList list1 = new ArrayIntList();
        list1.add(18);
        list1.add(27);
        list1.add(93);
        System.out.println(list1);
        list1.remove(1);
        System.out.println(list1);
        LinkedIntList list2 = new LinkedIntList();
        list2.add(18);
        list2.add(27);
        list2.add(93);
        System.out.println(list2);
        list2.remove(1);
        System.out.println(list2);
    }
}
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);
    }
}
**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) {
      ...
  }
  ```
Why use ADTs?

- Why would we want more than one kind of list, queue, etc.?
  - **Answer:** Each implementation is more efficient at certain tasks.
    - **ArrayList** is faster for adding/removing at the end;
      **LinkedList** is faster for adding/removing at the front/middle.
    - **HashSet** can search a huge data set for a value in short time;
      **TreeSet** is slower but keeps the set of data in a sorted order.
    - You choose the optimal implementation for your task, and if the rest of your code is written to use the ADT interfaces, it will work.