Announcements

A2 is due tomorrow night (17 February)
Get started on A3 – a method every other day.

Time to do A1:
356/416 groups gave the time.
Mean: 5 hours, 22 minutes.
Median: 5 hours exactly.

A Little Geometry!

Abstract Classes
Shape
x __ y ____
Triangle
area() base____ height__
Circle
area() radius __5__

PROBLEM
Since an object of Shape is not really a shape, don’t want to allow creation of objects of class Shape!

Solution
public abstract class Shape {
    ... 
}

Syntactic rule: if a class C is abstract, the new-expression new C(...) cannot be used!
**Return sum of areas of shapes in s */  
public static double sumAreas(Shape[] s) {  
    double sum= 0;  
    for (int k= 0; k < s.length; k= k+1)  
        sum= sum + s[k].area();  
    return sum;  
}  

Problems:  
1. Use instanceof (not getClass()) to figure out which subclass s[k] is and cast down so that function area() can be called.  
Adding new Shape subclass requires modifying sumAreas

---

A Partial Solution:  
Add method area to class Shape:  
```java  
public double area() {  
    return 0;  
}  
```

Use this instead?  
```java  
public double area() {  
    throw new RuntimeException("area not overridden");  
}  
```

Problem: a subclass might still forget to override area().

---

Good solution:  
In abstract class Shape, to require all subclasses to override function area, make it abstract:  
```java  
public abstract class Shape {  
    public abstract double area();  
}  
```

Syntax:  
If a method has keyword abstract in its declaration, use a semicolon instead of a method body

---

Abstract Summary  
1. To make it impossible to create an instance of a class C, make C abstract:  
```java  
public abstract C {  
    ...  
}  
```

Syntax:  
the program cannot be compiled if it contains a new-expression new C(...) and C is abstract.

2. In an abstract class, to require each subclass to override method m(...), make m abstract:  
```java  
public abstract int m(...);  
```

Syntax:  
the program cannot be compiled if a subclass of an abstract class does not override an abstract method.

---

Example of subclasses of Stack  
```java  
public abstract class Stack {  
    public abstract boolean isEmpty();  
    public abstract void push(int k);  
    public abstract int pop();  
}  
```

public class ArrayStack extends Stack{  
private int n;  // stack elements are in private int[] b; // b[0..n-1]. b[0] is bottom  
/** Constructor: An empty stack of max size s. */  
public ArrayStack(int s) {b= new int[s];}  
public boolean isEmpty() {return n == 0;}  
public void push(int v) {b[n]= v; n= n+1;}  
public int pop() {n= n-1; return b[n];}  
```
Example of subclasses of Stack

```java
public abstract class Stack {
    public abstract boolean isEmpty();
    public abstract void push(int k);
    public abstract int pop();
}
```

```java
public class LinkedListStack extends Stack {
    private int n; // number of elements in stack
    private Node first; // top node on stack

    /** Constructor: An empty stack */
    public LinkedListStack() {} 

    public boolean isEmpty() {
        return n == 0;
    }

    public void push(int v) { prepend v to list }

    public int pop() {
        ...
    }
}
```

```
public class ArrayStack extends Stack {
    ...
}
```

Flexibility!

```java
public abstract class Stack { … }
```  

```java
public class LinkedListStack extends Stack { … }
```  

```java
public class ArrayStack extends Stack { … }
```  

Interfaces

An interface is like an abstract class all of whose components are public abstract methods. Just have a different syntax

```java
public abstract class Stack {
    public abstract boolean isEmpty();
    public abstract void push(int k);
    public abstract int pop();
}
```

```java
public interface Stack {
    boolean isEmpty();
    void push(int k);
    int pop();
}
```

Since methods have to be public and abstract, we can leave off those keywords.

```
public interface Stack {
    boolean isEmpty();
    void push(int k);
    int pop();
}
```

Extend a class

```java
public class StackArray extends Stack {
    ...
}
```

Implement an interface

```java
public class StackArray implements Stack {
    ...
}
```
A start at understanding use of interfaces

Have this class hierarchy:

```java
class Animal {
    ...
}
class Mammal extends Animal {
    ...
}
class Bird extends Animal {
    ...
}
class Human extends Mammal {
    ...
}
class Dog extends Mammal {
    ...
}
class Parrot extends Bird {
    ...
}
```

Humans and Parrots can whistle. Other Animals cannot. “listenTo” is given as a whistling method:

```java
public void listenTo(String w) {
    System.out.println(w);
}
```

We need a way of indicating that classes Human and Parrot have this method listenTo.

```java
public interface Whistle {
    void listenTo(String w);
}
```

```java
public class Human extends Mammal implements Whistle {
    ...
    public void listenTo(String w) {
        System.out.println(w);
    }
}
```

Here's what an object of class Human looks like:

```java
public interface Whistle {
    void listenTo(String w);
}
public class Human extends Mammal implements Whistle {
    ...
    public void listenTo(String w) {
        System.out.println(w);
    }
}
```

Usual drawing of object:

```
Animal
   |   |
 Mammal   Whistle
   |   |
 Human   Dog  Parrot
```

A dimension for each class that is extended and interface that is implemented.

Draw it this way:

```
Animal
   |   |
 Mammal   Whistle
   |   |
 Human   Dog  Parrot
```

Add interface dimension:

```
Animal
   |   |
 Mammal   Whistle
   |   |
 Human   Dog  Parrot
```

h, ob, a, m, and w all point to the same object.

The object can be (and is) cast to any “partition” in it: h, ob, a, m, and w.

Upward casts: can be implicit; inserted by Java

Downward casts: must be explicit.
A real use of interface: sorting

Consider an array of Shapes: want to sort by increasing area
Consider an array of ints: want to sort them in increasing order
Consider an array of Dates: want to put in chronological order
We don’t want to write three different sorting procedures!
The sorting procedure should be the same in all cases. What differs is how elements of the array are compared.
So, write ONE sort procedure, tell it the function to be used to compare elements. To do that, we will use an interface.

Real example: Comparable<T>

We implement Comparable<T> in class Shape

```java
public abstract class Shape implements Comparable<Shape> {
    // ... 
    /** Return the area of this shape */
    public abstract double area();
    /** Return negative number, 0, or a positive number depending on whether this are is <, =, or > c's area */
    public int compareTo(Shape c) {
        double diff = area() - c.area();
        return diff == 0 ? 0 : (diff < 0 ? -1 : 1); // compareTo() method
    }
}
```

What an object of subclasses look like

```java
public abstract class Shape implements Comparable<Shape> { ... }
public class Circle extends Shape { ... }
public class Rectangle extends Shape { ... }
```

When sort procedure is comparing elements of a Shape array, each element is a Shape. Sort procedure views it from Comparable perspective!

```
<table>
<thead>
<tr>
<th>Object</th>
<th>Comparable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shape</td>
<td></td>
</tr>
<tr>
<td>Circle</td>
<td></td>
</tr>
</tbody>
</table>
```

Interface Comparable<T>

Package java.lang contains this interface

```java
public interface Comparable<T> {
    /** = a negative integer if this object < c,
    = 0 if this object = c,
    = a positive integer if this object > c.
    Throw a ClassCastException if c can't be cast to the class of this object. */
    int compareTo(T c);
}
```

Arrays.sort has this method.

```java
/** Sort array b. Elements of b must implement interface Comparable<T>. Its method compareTo is used to determine ordering of elements of b. */
Arrays.sort(Object[] b)
```

Shape implements Comparable, so we can write

```java
// Store an array of values in shapes
Shape[] shapes= ...; ...
Arrays.sort(shapes);
```

Abstract Classes vs. Interfaces

- Abstract class represents something
- Share common code between subclasses
- Interface is what something can do. Defines an “abstract data type”
- A contract to fulfill
- Software engineering purpose

Similarities:
- Can’t instantiate
- Must implement abstract methods
- Later we’ll use interfaces to define “abstract data types”
  - (e.g. List, Set, Stack, Queue, etc)