Announcements

- A2 is due Sunday night (16 September)
- Prelim 1 is two weeks away! **GO TO THE WEBPAGE:**
- Study Guide
- When you take the exam (where comes later)
  - Half of you will take the exam at 5:30!
  - The other half at 7:30!
- Whom to contact about conflicts
  READ CAREFULLY & THOROUGHLY

Where are we?

You watched 15 minutes of videos in preparation for this lecture. You learned about abstract classes and interfaces.

**Reason for asking you to watch it:**

You now have seen the technical details and have a basic understanding. We can now give examples and discuss things at greater depth than if you had not watched the videos.

Operator `instanceof` and function `getClass`

```java
s instanceof Rect
s.getClass()
```

Where are we?

Position of a rectangle in the plane is given by its upper-left corner

Position of a circle in the plane is given by the upper-left corner of its bounding box
A Little Geometry!

Abstract Classes

Shape contains coordinates in the plane. Subclasses declare additional fields and the method area.

Rectangle
- width
- height
- area()

Triangle
- base
- height
- area()

Circle
- radius
- area()

Shape
- x
- y

What is “only” a Shape, anyway??

Notice: An object of Shape is not really a shape. → Don’t want to allow creation of objects of class Shape!

Make the class abstract!

public abstract class Shape {
    /*
    * Syntactic rule: if class C is abstract, the new-expression new C(...) cannot be used!
    */
}

Writing sumAreas in class Shape

/** 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;
}

Compile-time reference rule says no!

Solutions?
1. Use instanceof and cast down to make the call?
   (next slide)
2. Make area a method of Shape?

Approach 1: Cast down to make the call

double sum= 0;
for (int k= 0; k < s.length; k= k+1) {
    if (sh[k] instanceof Circle)
        sum= sum + ((Circle) sh[k]).area();
    else if (sh[k] instanceof Rectangle)
        sum= sum + ((Rectangle) sh[k]).area();
}
return sum;

Does this work?

1. Code is ugly
2. Code doesn’t age well

Approach 2: define area in Shape

Add method area to class Shape:

public double area() { return 0; }

Use this instead?

public double area() {
    throw new RuntimeException("area not overridden");
}

Approach 3: Make area abstract! (Yay!)

In abstract class Shape, an abstract function area is required of all subclasses:

public abstract class Shape {
    /*
    * Problem: a subclass might forget to override area().
    */
    public abstract double area();
}

Syntax:
If a method has keyword abstract in its declaration, use a semicolon instead of a method body.
1. To make it impossible to create an instance of a class C, make C abstract:

```java
public abstract C {...}
```

2. In an abstract class, to require each subclass to override method m(...), make m abstract:

```java
public abstract int m(...);
```

---

**Abstract class used to “define” a type (abstract data type, or ADT)**

**Type:** set of values together with operations on them

**Define type Stack (of ints). Its operations are:**

- `isEmpty()` -- return true iff the stack is empty
- `push(k)` -- push integer k onto the Stack
- `pop()` -- pop the top stack element

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

**Naturally, need specifications**

```java
/** A class that needs a stack */
public class C {
  Stack st; // new ArrayStack(20);
  public void m() {
    st.push(5);
    st.push(5);
    st.push(5);
    st.push(5);
    st.push(5);
    st.push(5);
    st.push(5);
    st.push(5);
    st.push(5);
  }
}
```

**Example of Stack subclass:**

**ArrayStack**

```java
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 Stack subclass:**

**LinkedListStack**

```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() { return first.value; first = first.next; }
}
```

**Flexibility!**

```java
public abstract class Stack {...}
public class ArrayStack extends Stack {...}
public class LinkedListStack extends Stack {...}
```

Choose an array implementation, max of 20 values

```java
/** A class that needs a stack */
public class C {
  LinkedListStack st; // new ArrayStack(20);
  public void m() {
    st.push(5);
    st.push(5);
    st.push(5);
    st.push(5);
    st.push(5);
    st.push(5);
    st.push(5);
    st.push(5);
    st.push(5);
  }
}
```

Want to use a linked list instead of an array? Just change the new-expression!
An interface is like an abstract class all of whose components are public abstract methods. Just have a different syntax.

We don’t tell you immediately WHY Java has this feature, this construct. First let us define the interface and see how it is used. The why will become clear as more and more examples are shown.

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

Here is an abstract class. Contains only public abstract methods.

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

Here is how we declare it as an interface.

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

Methods must be public and abstract, so we can leave off those keywords.

```java
public interface Speaker {
    void speak(String w);
}
```

Humans and Parrots can speak. Other Animals cannot.

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

We need a way of indicating that classes Human and Parrot have this method speak.
Here’s what an object of class Human looks like

```java
public interface Speaker {
    void speak(String w);
}

public class Human extends Mammal implements Speaker {
    public void speak(String w) {
        System.out.println(w);
    }
}
```

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

We implement Comparable in class Shape

```java
public abstract class Shape implements Comparable {
    /** Return area of this shape */
    public abstract double area();

    /** See previous slide*/
    public int compareTo(Object c) {
        Shape s = (Shape) c;
        double diff = area() - s.area();
        return diff == 0 ? 0 : (diff < 0 ? -1 : 1);
    }
}
```

Interface Comparable

Package java.lang contains this interface

```java
public interface Comparable {
    /** = 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(Object c);
}
```

Arrays.sort has this method

```java
/** Sort array b. Elements of b must implement interface Comparable. 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);
```
What an object of subclasses look like

```java
public abstract class Shape implements Comparable {
    ...
}
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!

### Abstract Classes vs. Interfaces

<table>
<thead>
<tr>
<th>Abstract Classes</th>
<th>Interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Abstract class represents something</td>
<td>- Interface is what something can do. Defines an “abstract data type”</td>
</tr>
<tr>
<td>- Share common code between subclasses</td>
<td>- A contract to fulfill</td>
</tr>
<tr>
<td>- Software engineering purpose</td>
<td>- Software engineering purpose</td>
</tr>
</tbody>
</table>

**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)