Announcements

- A2 is due Thursday night (14 February)
- Go back to Lecture 6 & discuss method equals
A Little Geometry!

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!

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

- **Rectangle**
  - `width`
  - `height`
  - `area()`

- **Triangle**
  - `base`
  - `height`
  - `area()`

- **Circle**
  - `radius`
  - `area()`
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!

```java
public abstract class Shape {
    ...
}
```

**Syntactic rule:** if class \( C \) is abstract, the new-expression \( \text{new } C(\ldots) \) 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;
}

Does this work?

Compile-time reference rule says no!

Solutions?
1. Cast down to make the call?
2. Make area a method of Shape?
Approach 2: define area in Shape

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 forget to override area().

**Problem:** a subclass might still forget to override area().
Approach 3: Make area abstract! (Yay!)

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

```java
public abstract class Shape {
    ...
    /** Return the area of this 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:

\[
\text{public abstract } C \{ \ldots \}
\]

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

\[
\text{public abstract int } m(\ldots) ;
\]

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

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

Missing tests for errors! Missing specs!
public abstract class Stack {
    public abstract boolean isEmpty();
    public abstract void push(int k);
    public abstract int pop();
}

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() { ... }
    public void push(int v) { ... }
    public int pop() { ... }
}
/** A class that needs a stack */
public class C {
    Stack st = new ArrayStack(20);
    ...
    public void m() {
        ...
        st.push(5);
    }
    ...
}
/** A class that needs a stack */
public class C {
    LinkedListStack();
    Stack $st=$ new ArrayStack(20);
    ...
    public void m() {
        ...
        st.push(5);
    }
}...
Interfaces

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.

(an interface can have a few other kinds of components, but they are limited. For now, it is easiest to introduce the interface by assuming it can have only public abstract methods and nothing else. Go with that for now!)
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();
}
```

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
Interfaces

```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();
}
```

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

Extend a class:
```java
class StackArray extends Stack {
    ...
}
```

Implement an interface:
```java
class StackArray implements Stack {
    ...
}
```
A start at understanding use of interfaces

Have this class hierarchy:

class Animal { ... }
class Mammal extends Animal { ... }
class Bird extends Animal { ... }
class Human extends Mammal { ... }
class Dog extends Mammal { ... }
class Parrot extends Bird { ... }
A start at understanding use of interfaces

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`
A start at understanding use of interfaces

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

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

(similarly for Parrot)
```
Here’s what an object of class Human looks like

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

class Human extends Mammal implements Speaker {
    public void speak(String w) { // Method implementation
        System.out.println(w); // Print the string
    }
}
```

Usual drawing of object

```
Object
  | Animal
  | Mammal
  | Human
```

Draw it this way

```
Object
  | Animal
  | Mammal
  | Speaker
```

Add interface dimension

```
Object
  | Animal
  | Mammal
  | Human
  | Speaker
```
Here’s what an object of class Human looks like

Human h = new Human();
Object ob = h;
Animal a = (Animal) ob;
Mammal m = h;
Speaker s = h;

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.
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);
}
```
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);
    }
}
```

If c can’t be cast to Shape, a ClassCastException is thrown
Arrays.sort has this method

/** 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:

// Store an array of values in shapes
Shape[] shapes= ...;
...
Arrays.sort(shapes);
What an object of subclasses look like

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

- 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)
Operator instanceof vs getClass

```
<object> instanceof <class-name> is true iff <object> has a partition named <class-name>
```

- h instanceof Object is true
  h.getClass() == Object.class is false

- h instanceof Animal is true
  h.getClass() == Animal.class is false

- h instanceof Cat is true
  h.getClass() == Cat.class is true

- h instanceof PhD is false
**Approach 1: Cast down to make the call**

```java
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;
```

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