CS/ENGRD 2110
SPRING 2019

Lecture 7: Interfaces and Abstract Classes
http://courses.cs.cornell.edu/cs2110

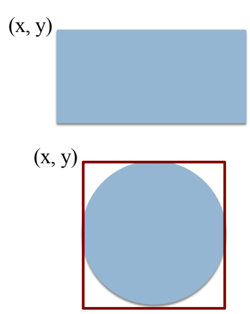
1

Announcements

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

2

A Little Geometry!



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

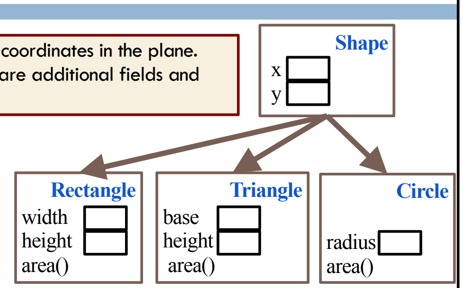
Position of a circle in the plane is given by the upper-left corner of its bounding box

3

A Little Geometry!

Abstract Classes

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

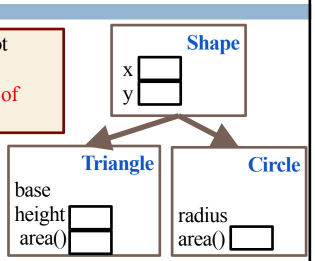


4

What is "only" a **Shape**, anyway??

Abstract Classes

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

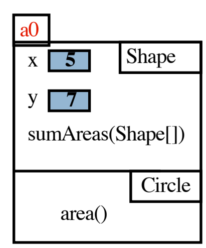
Abstract Classes

```
/** 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?



6

Approach 2: define area in Shape

Add method area to class Shape:

```
public double area() {
    return 0;
}
```

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

Use this instead?

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

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:

```
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:

public abstract C { …}

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

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

public abstract int m(…) ;

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

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

Naturally, need specifications

Example of Stack subclass: ArrayStack

```
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]; }
}
```

Missing tests for errors!
Missing specs!

Example of Stack subclass: LinkedListStack

```
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() { … }
}
```

Flexibility!

public abstract class Stack { … }

public class ArrayStack extends Stack { … }

public class LinkedListStack extends Stack { … }

```
/** A class that needs a stack */
public class C {
    Stack st= new ArrayStack(20);
    …
    public void m() {

        …
    }   st.push(5);
        …
}
```

Store the ptr in a variable of type Stack!

Choose an array implementation, max of 20 values

Use only methods available in abstract class Stack

13

Flexibility!

public abstract class Stack { … }

public class ArrayStack extends Stack { … }

public class LinkedListStack extends Stack { … }

```
/** A class that needs a stack */
public class C {        LinkedListStack();
    Stack st= new ArrayStack(20);
    …
    public void m() {

        …
    }   st.push(5);
        …
}
```

Want to use a linked list instead of an array? Just change the new-expression!

14

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!)

15

Interfaces

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

```
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

```
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

16

Interfaces

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

```
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:

```
class StackArray extends Stack {
  …
}
```

Implement an interface:

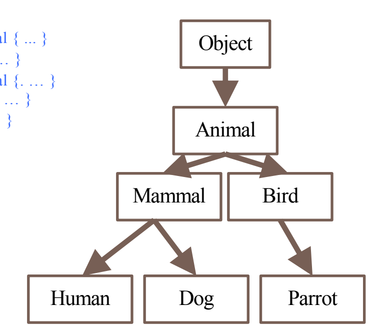
```
class StackArray implements Stack {
  …
}
```

17

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 { … }
```



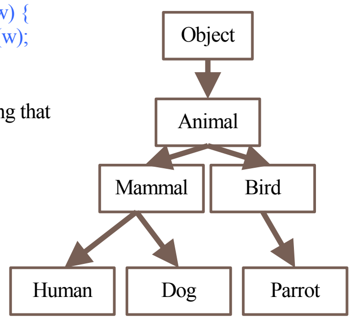
18

A start at understanding use of interfaces

Humans and Parrots can speak. Other Animals cannot.

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

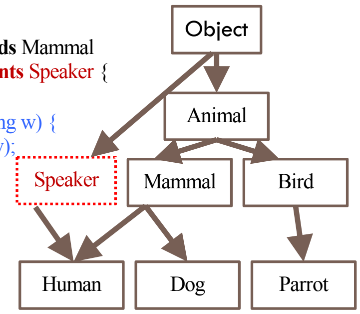
We need a way of indicating that classes Human and Parrot have this method speak



A start at understanding use of interfaces

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

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

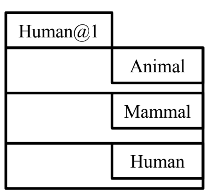
(similarly for Parrot)



Here's what an object of class Human looks like

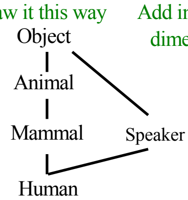
```
public interface Speaker {void speak(String w); }

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

Usual drawing of object



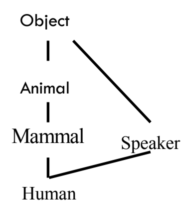
Draw it this way

Add interface dimension



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 **int**s: 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

```
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

```
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:

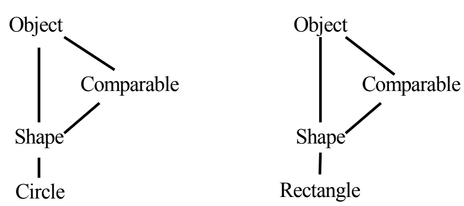
```
// Store an array of values in shapes
Shape[] shapes= ...;
...
Arrays.sort(shapes);
```

26

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!



27

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)

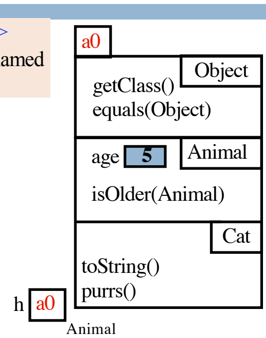
28

Operator instanceof vs getClass

29

<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

Abstract Classes

```
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;
```

or ?

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

30