CS/ENGRD 2110
SPRING 2017

Lecture 7: Interfaces and Abstract Classes

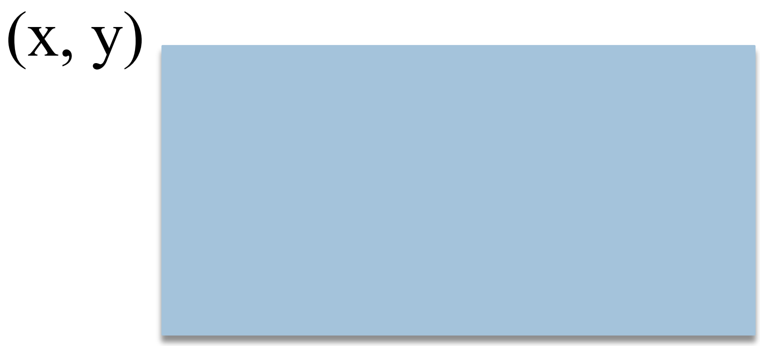
http://courses.cs.cornell.edu/cs2110

Announcements

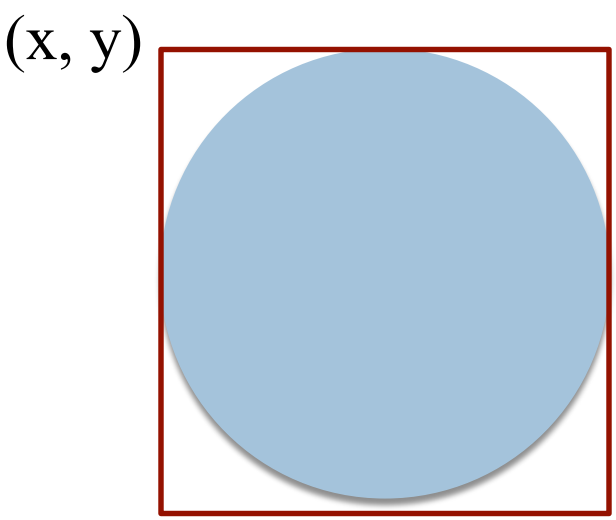
A2 is due tomorrow night (17 February)

Get started on A3 – a method every other day

A Little Geometry!



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

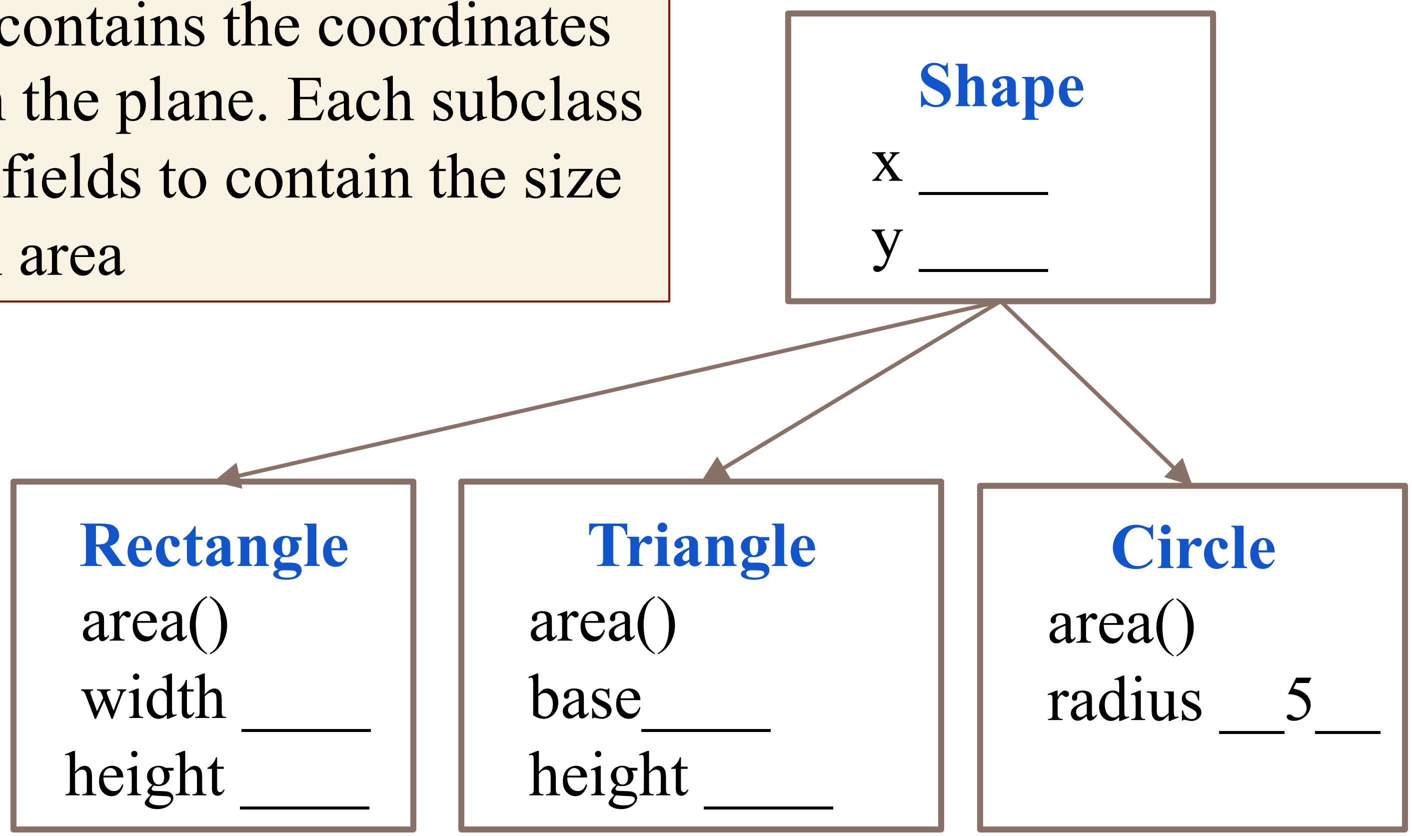


Position of a circle in the plane is given by the upper-left corner of its bounding box

A Little Geometry!

class Shape contains the coordinates of a shape in the plane. Each subclass declares the fields to contain the size and function area

Write variables as lines instead of boxes



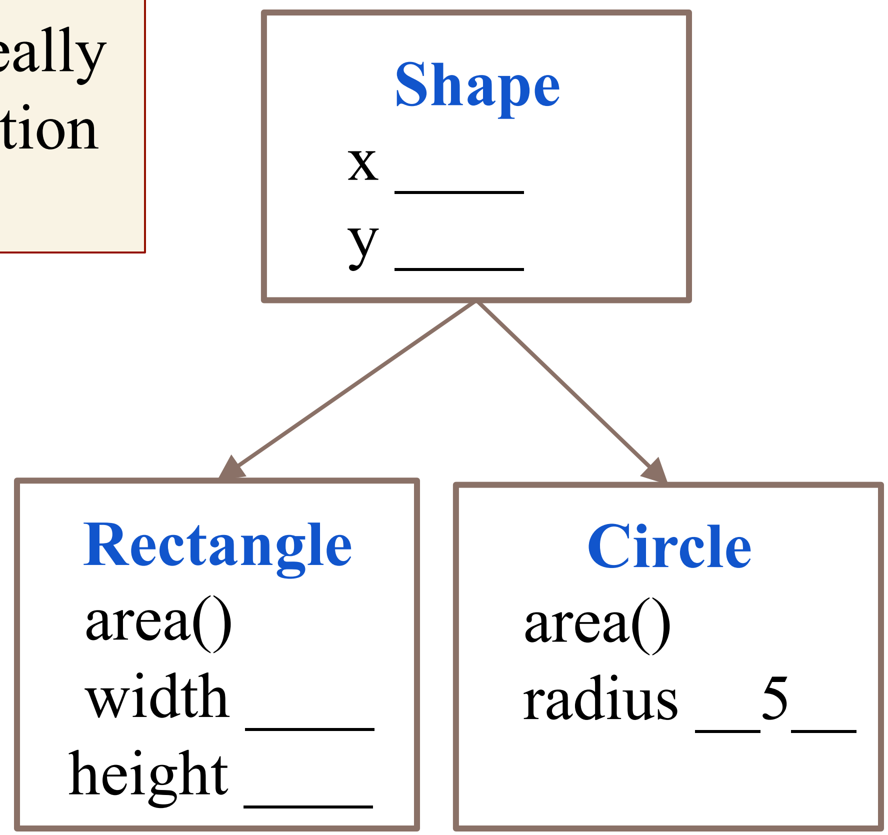
Problem: Don't like creation of Shape objects

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!



Attempt at writing function sumAreas


```
/** 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?

Problems:

1. Use **instanceof** 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:

```
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().

Good solution:

In abstract class Shape, to require all subclasses to override function area, make it abstract:

```
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 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:

```
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)

Type: set of values together with operations on them

Suppose we want to define type Stack (of ints). It's 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 subclasses of Stack

```
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 lots of tests for errors! Missing specs!

11

Example of subclasses of Stack

```
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 LinkeListStack() {}

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

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

    public int pop() { …}
}
```

Missing lots of tests for errors!
Missing specs!

12

Flexibility!

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

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

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

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

Choose an array implementation, max of 20 values

Store the ptr in a variable of type Stack!

Use only methods available in abstract class Stack

13

Flexibility!

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

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

```
public class ArrayStack 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!)

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

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();
}
```

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

Extend a class

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

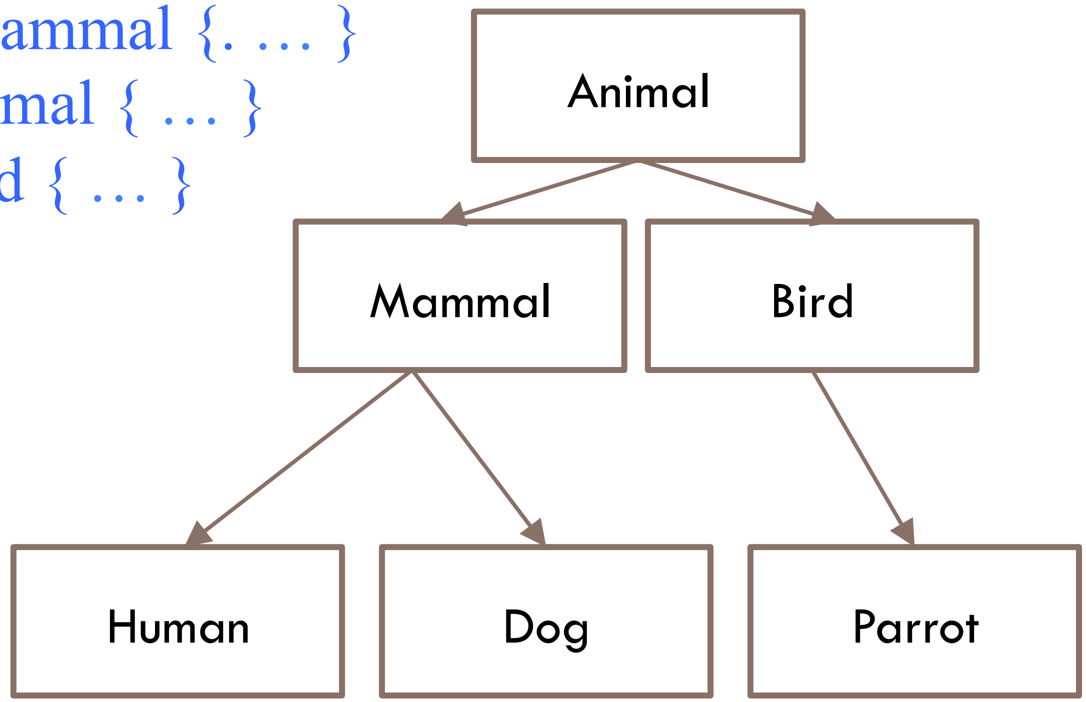
Implement an interface

```
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 { … }

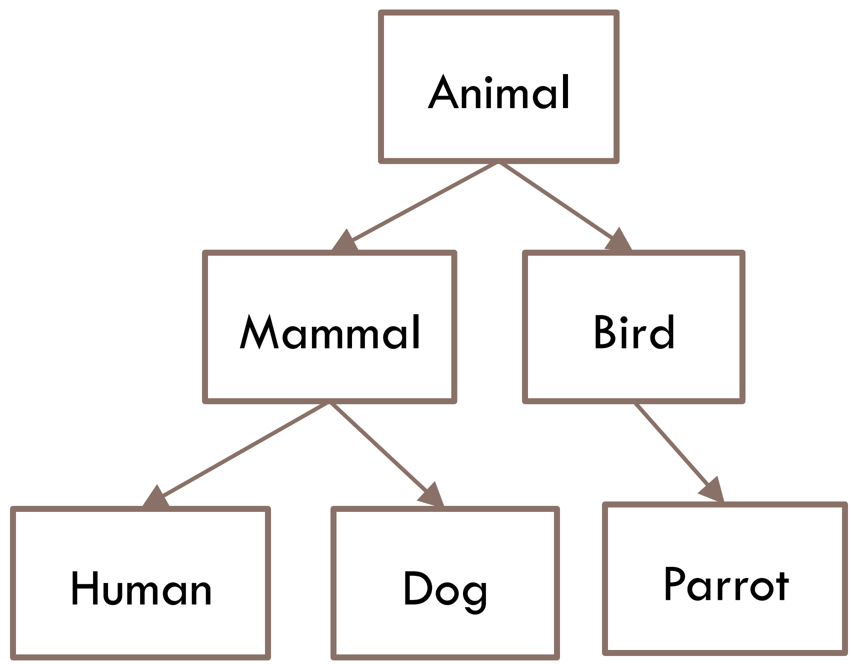


A start at understanding use of interfaces

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

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

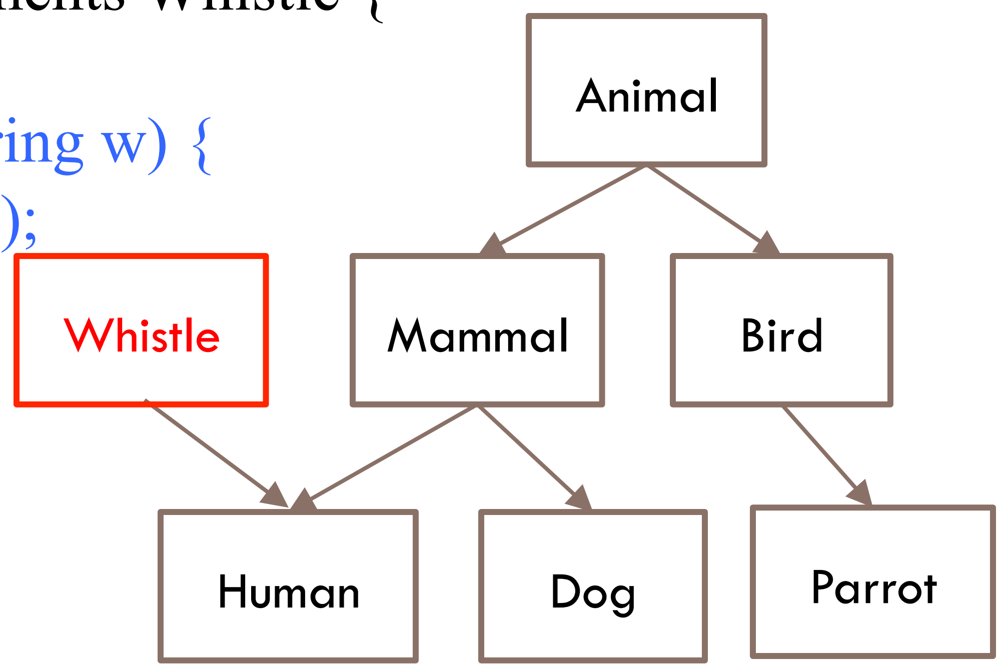
We need a way of indicating that classes Human and Parrot have this method listenTo



A start at understanding use of interfaces

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

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

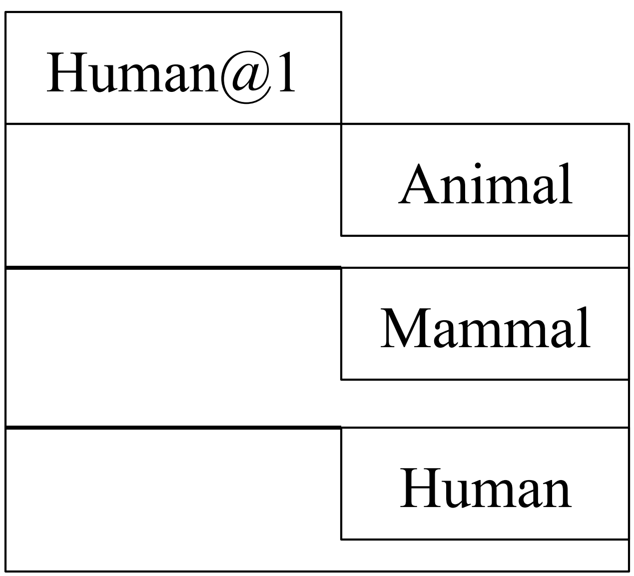
(similarly for Parrot)



Here's what an object of class Human looks like

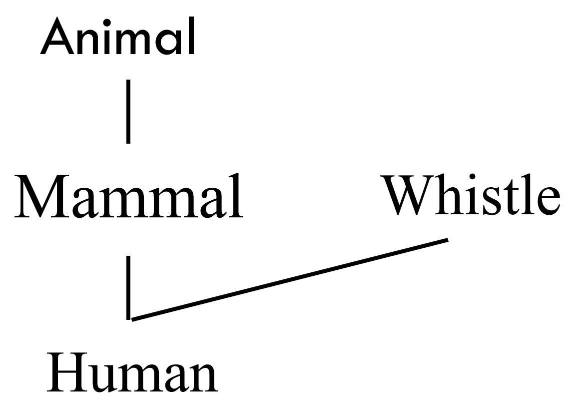
```
public interface Whistle { void listenTo(String w) ; }

public class Human extends Mammal implements Whistle {
    …
    public void listenTo(String w) { …}
}
```

Usual drawing of object



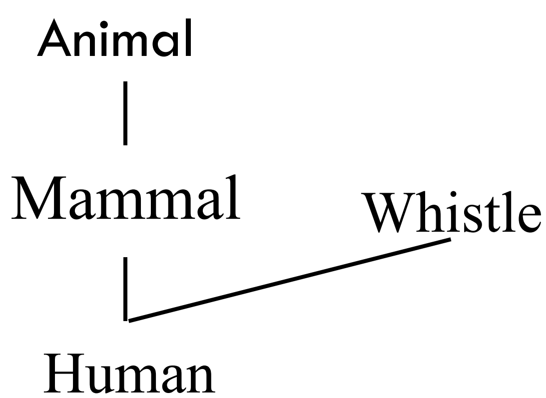
Draw it this way

Add interface dimension



Here's what an object of class Human looks like

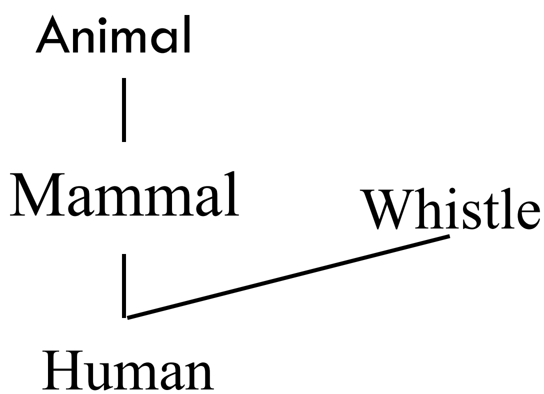
```
public interface Whistle { void listenTo(String w) ; }

public class Human extends Mammal implements Whistle {
    …
    public void listenTo(String w) { …}
}
```



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

Here's what an object of class Human looks like

```
Human h= new Human();
Object ob= h;
Animal a= (Animal) ob;
Mammal m=  h;
Whistle w= 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<T>

Package java.lang contains this interface

```
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);
}
```

Real example: Comparable<T>

We implement Comparable<T> in class Shape

```
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);

}
```

Arrays.sort has this method.

```
/** 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

```
    // Store an array of values in shapes
    Shape[] shapes= ...;  ...

  Arrays.sort(shapes);
```

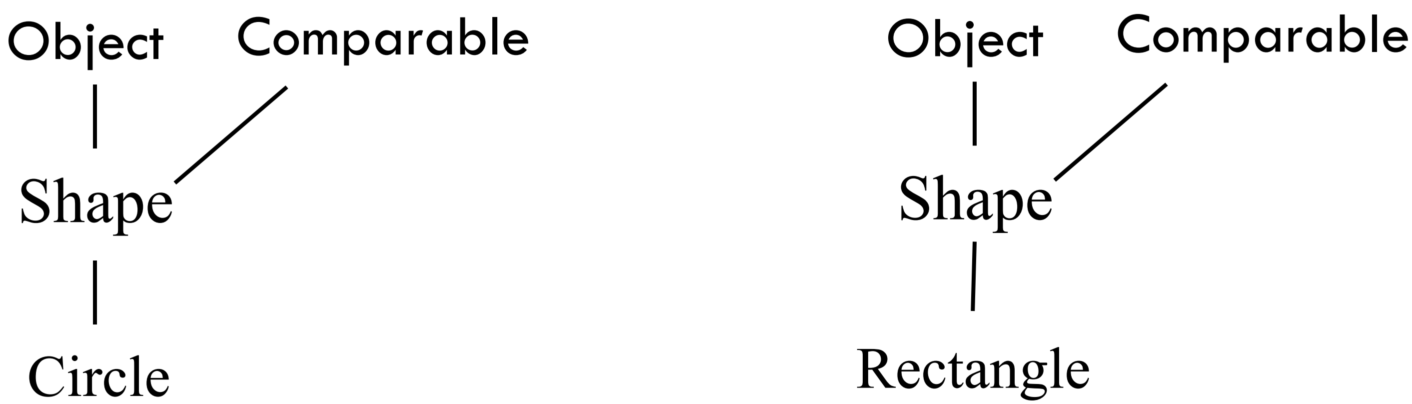
What an object of subclasses look like

public abstract class Shape implements Comparable<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!



Abstract Classes vs. Interfaces

- Abstract class represents something
- Shar3 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)