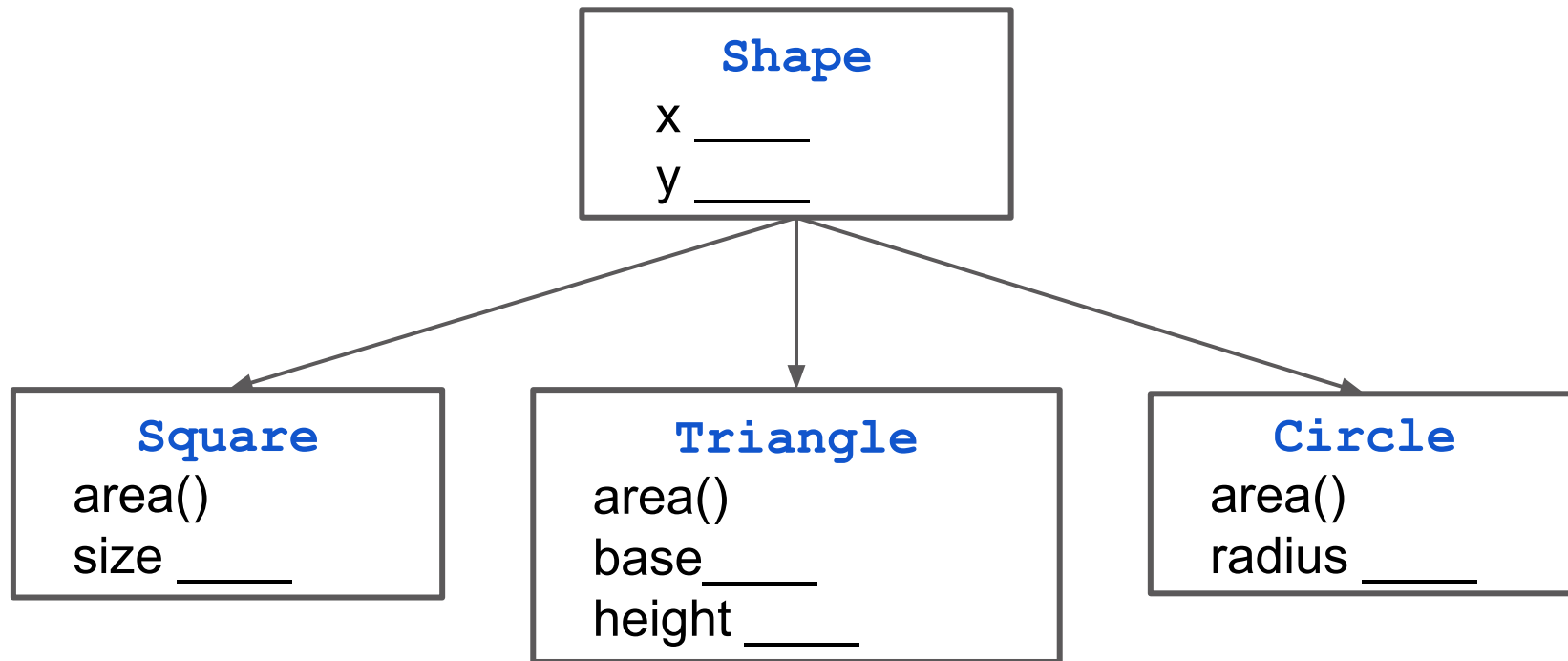# Recitation 4

Abstract classes, Interfaces

A Little More Geometry!

Demo 1: **Complete this function**

```
/** Return the sum of the areas of
   * the shapes in s */
static double sumAreas(Shape[] s) { }
```

1. Operator **instanceof** and casting are required
2. Adding new `Shape` subclasses breaks `sumAreas`

A Partial Solution:

Add method area to class Shape:

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



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

Problems not solved

1. What is a Shape that isn't a Circle, Square, Triangle, etc? What is *only* a shape, nothing more specific?
    a. `Shape s = ` **`new`** ` Shape(...);` Should be disallowed

2. What if a subclass doesn't override area()?
    a. Can't force the subclass to override it!
    b. Incorrect value returned or exception thrown.

Solution: Abstract classes

Abstract class
Can't be instantiated.
(**new** Shape() illegal)

```java
public abstract class Shape {

    public double area() {
        return 0;
    }

}
```

Solution: Abstract methods

```java
public abstract class Shape {

    public abstract double area();

}
```

Abstract method
Subclass must override.

- Can have implemented methods, too

- Place abstract method only in abstract class.

- Semicolon instead of body.

Demo 2: A better solution

We modify class Shape to be abstract and make `area()` an abstract method.

- Abstract class prevents instantiation of class Shape
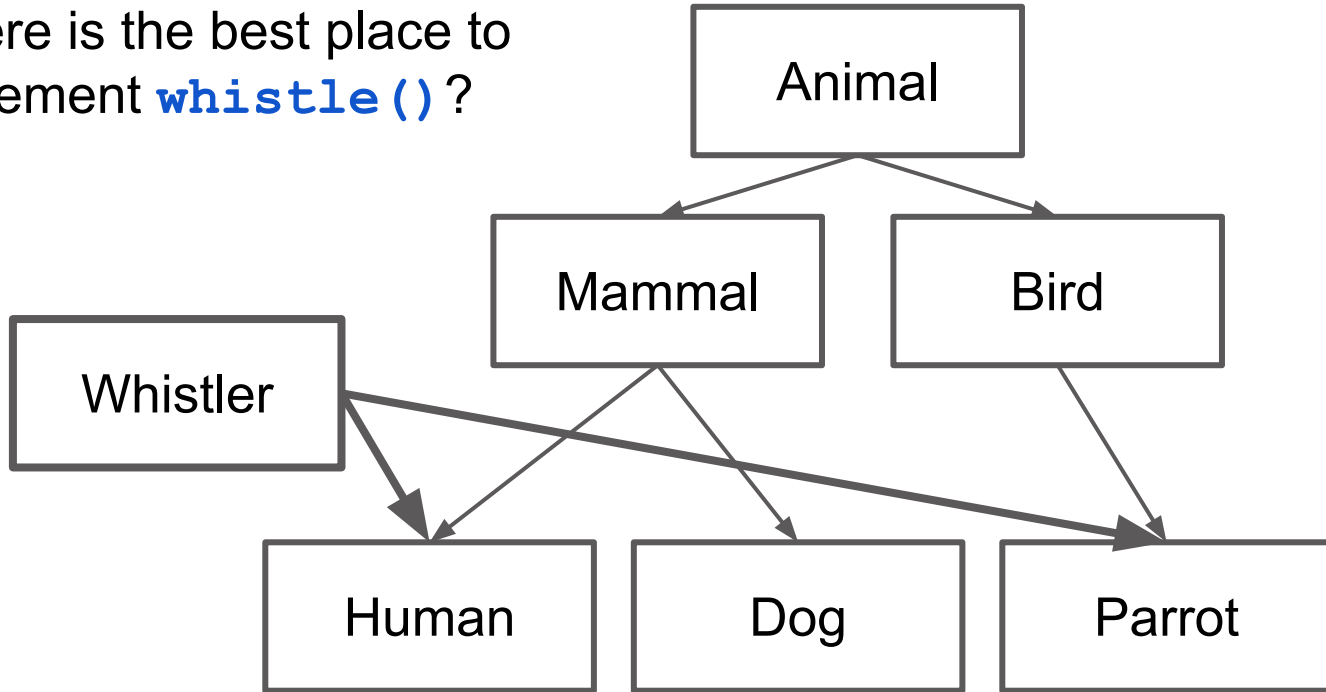- Abstract method forces all subclasses to override `area()`

Abstract Classes, Abstract Methods

1. **Cannot instantiate an object of an abstract class.**

   (Cannot use new-expression)

2. **A subclass must override abstract methods.**

Interfaces

Problem

Where is the best place to implement `whistle()`?

No multiple inheritance in Java!

```
class Whistler {
    void breathe() { … }
}
class Animal {
    void breathe() { … }
}
class Human extends Animal, Whistler {
}
```

```
new Human().breathe();
```
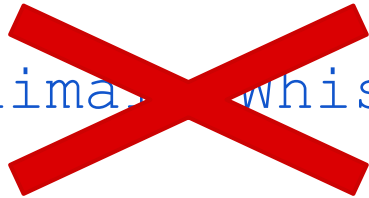
Which breathe() should java run in class Human?

Why not make it fully abstract?

```
class abstract Whistler {
    abstract void breathe();
}
class abstract Animal {
    abstract void breathe();
}
class Human extends Animal Whistler {
}
```

Java doesn't allow this, even though it would work. Instead, Java has another construct for this purpose, the interface

Solution: Interfaces

```java
public interface Whistler {
    void whistle();
    int MEANING_OF_LIFE= 42;
}
```

- methods are automatically `public` and `abstract`

- fields are automatically `public`, `static`, and `final` (i.e. constants)

```java
class Human extends Mammal implements Whistler {
}
```
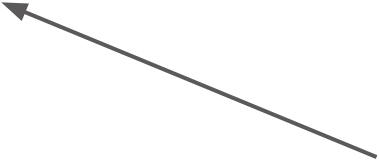
Must implement all methods in the implemented interfaces

Multiple interfaces

```java
public interface Singer {

    void singTo(Human h);

}
```

Classes can implement several interfaces! They must implement all the methods in those interfaces they implement.
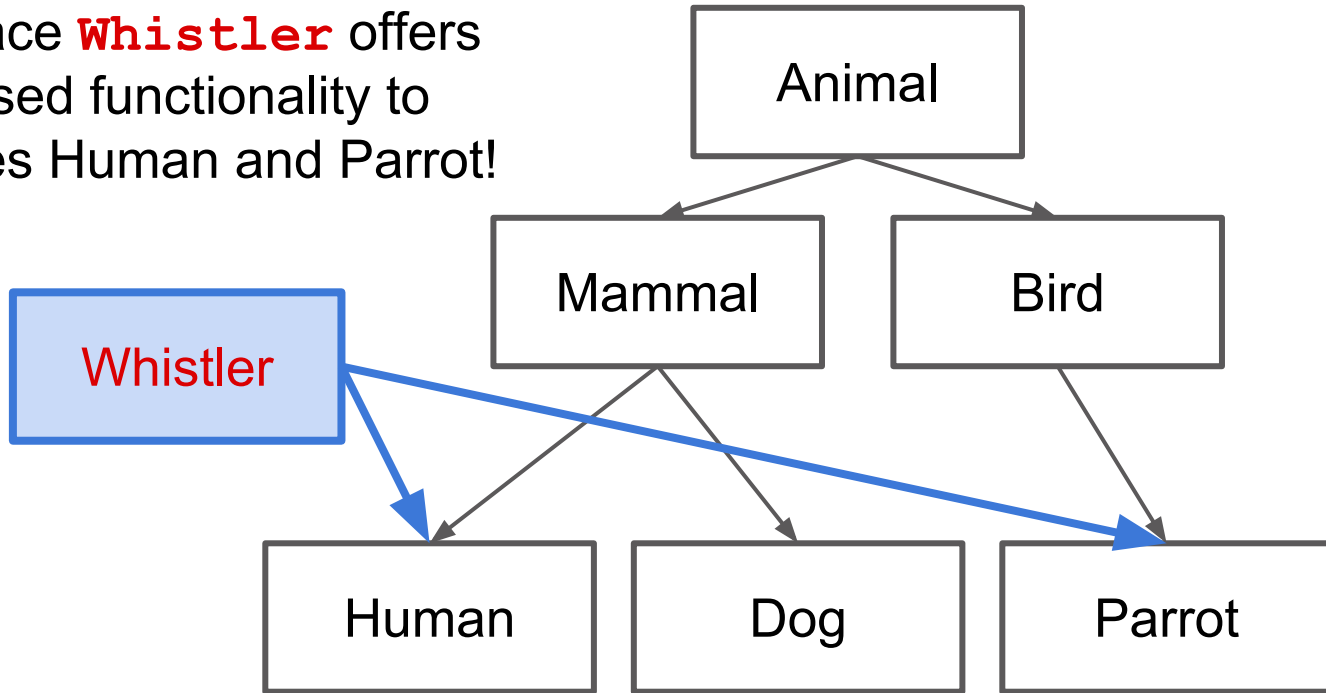
```java
class Human extends Mammal implements Whistler, Singer {

}
```

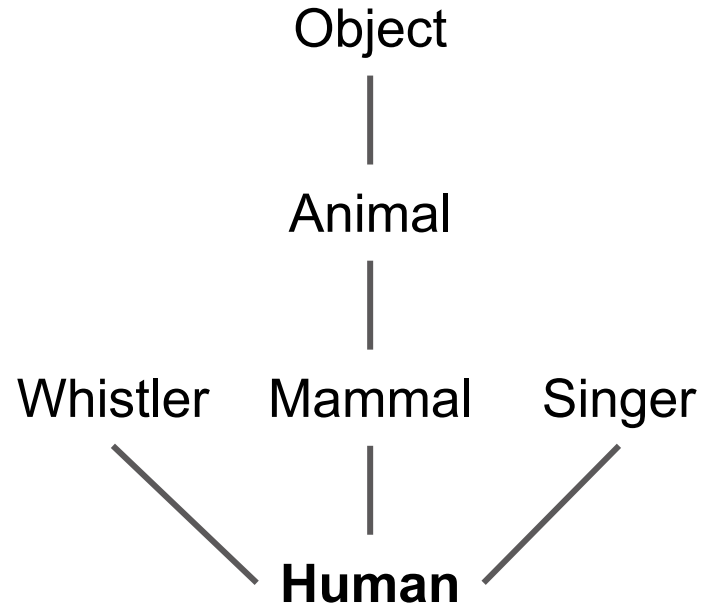Must implement `singTo(Human h)` and `whistle()`

Solution: Interfaces

Interface **Whistler** offers promised functionality to classes Human and Parrot!

Casting to an interface

```
Human h  = new Human();
Object o = (Object) h;
Animal a = (Animal) h;
Mammal m = (Mammal) h;


Singer s = (Singer) h;
Whistler w = (Whistler) h;

All point to the same memory address!
```

Object

Animal

Whistler   Mammal   Singer
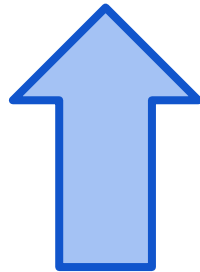
Human

Casting to an interface

```
Human h  = new Human();
Object o = h;
Animal a = h;
Mammal m = h;
Singer s = h;
Whistler w = h;
```

Automatic up-cast

Forced down-cast

Object
|
Animal
|
Whistler   Mammal   Singer
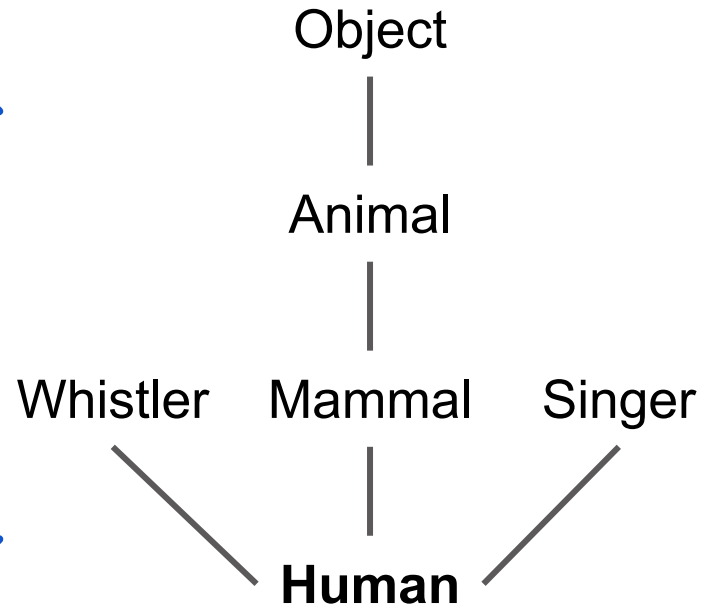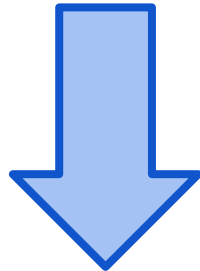
Human
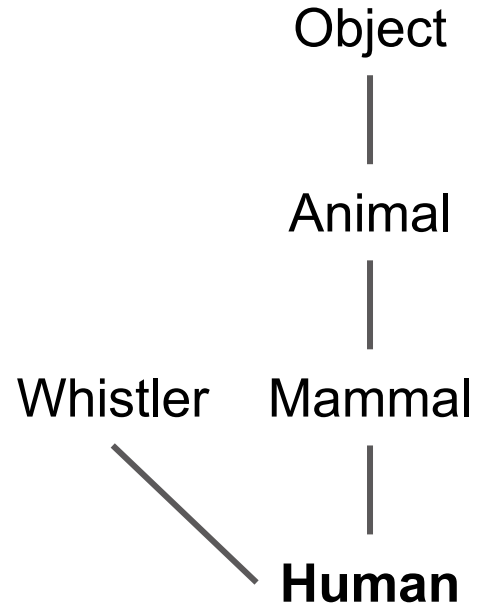
Casting up to an interface automatically

```
class Human … implements Whistler {
    void listenTo(Whistler w) {...}
}

Human h = new Human(...);

Human h1= new Human(...);

h.listenTo(h1);
```

Arg h1 of the call has type Human. Its value is being stored in w, which is of type Whistler. Java does an upward cast automatically. It costs no time; it is just a matter of perception.

Object

|

Animal

|

Whistler   Mammal

|

Human

Demo 3: Implement `Comparable<T>`

Implement interface Comparable in class Shape:

```
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 cannot
            be cast to the class of this object. */

    int compareTo(T c);

}
```

Shape implements `Comparable<T>`

```java
public class Shape implements Comparable<Shape> {
    ...
    /** … */
    public int compareTo(Shape s) {
        double diff= area() - s.area();
        return (diff == 0 ? 0 : (diff < 0 ? -1 : +1));
    }
}
```

Beauty of interfaces

`Arrays.sort` sorts an array of *any* class C, as long as C implements interface `Comparable<T>` without needing to know any implementation details of the class.
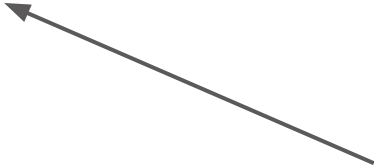
Classes that implement Comparable:

```
Boolean      Byte         Double       Integer
String       BigDecimal   BigInteger   Calendar
Time         Timestamp    and 100 others
```

String sorting

Arrays.sort(Object[] b) sorts an array of *any* class C, as long as C implements interface **Comparable<T>**.

String implements Comparable, so you can write
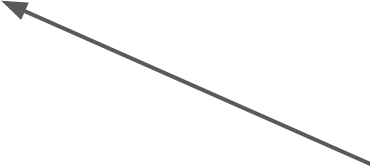
```
String[] strings= ...;  ...
Arrays.sort(strings);
```

During the sorting, when comparing elements, a String's compareTo function is used

And Shape sorting, too!

`Arrays.sort(Object[] b)` sorts an array of *any* class C, as long as C implements interface **`Comparable<T>`**.

Shape implements Comparable, so you can write

```
Shape[] shapes= ...;   ...
Arrays.sort(shapes);
```

During the sorting, when comparing elements, a Shape's compareTo function is used

Abstract Classes vs. Interfaces

- Abstract class represents something
- Sharing common code between subclasses

- Interface is what something can do
- 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)