Announcements

- Attendance for this week’s recitation is mandatory!
- A2 is due today
- Get started on A3 – a method every other day

A Little Geometry!

Abstract Classes

**Demo 1: Complete this function**

```java
/** 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:

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

```
public class Shape {
    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

```java
public abstract class Shape {
    public abstract double area();
}
```

Abstract class
Means that it can't be instantiated. `new Shape()` illegal.

Solution: Abstract methods

```java
public abstract class Shape {
    public abstract double area();
}
```

Abstract method
Subclass must override.

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.

Problem

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

Interfaces

- Animal
- Mammal
- Bird
- Whistler
- Human
- Dog
- Parrot
No multiple inheritance in Java!

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

Which breathe() should Java run in class Human?

new Human().breathe();

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

public interface Whistler {
    void whistle();
    int MEANING_OF_LIFE = 42;
}
class Human extends Mammal implements Whistler {
}

Multiple interfaces

public interface Singer {
    void singTo(Human h);
}
class Human extends Mammal implements Whistler, Singer {
}

Classes can implement several interfaces. They must implement all the methods in those interfaces they implement.

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

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!
Casting to an interface

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

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);
```

Demo 3: Implement Comparable<T>

```
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);
}
```

Shape implements Comparable<T>

```
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!

```java
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
```
```java
Shape[] shapes= ...; ...
Arrays.sort(shapes);
```
```
During the sorting, when comparing elements, a Shape's compareTo function is used
```

Abstract Classes vs. Interfaces

<table>
<thead>
<tr>
<th>Abstract class represents something</th>
<th>Interface is what something can do</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sharing common code between subclasses</td>
<td>A contract to fulfill</td>
</tr>
<tr>
<td>---</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