SUMMARY: abstract classes and interfaces

Make a class abstract so instances of it cannot be created. Make a method abstract so it must be overridden.

An interface is like an abstract class whose methods are all abstract and whose fields are all public constants. This allows multiple inheritance without ambiguity. An interface has a different syntax and a different way of using it.

References to text and to JavaSummary.pptx:
Abstract class: C.27, slides 42-44
Abstract method: C.27, slide 44
Interface declaration: D.11-D.13, D.28, slide 60
Implementing interfaces: D.14-D.15, slide 60
Casting with interfaces: none, slide 61
Interface Comparable: D.20, slide 62

abstract classes and interfaces

Every shape has a position (x, y) in the plane, so use a superclass Shape to hold the point. Subclass has necessary fields to describe a shape.

Teach using the problem of using objects to represent shapes in the plane.

Motivating abstract classes

b[1].area() is illegal, even though each Subclass object has function area()

Don’t want to cast down! Instead, define area() in Shape

Motivating abstract classes

area() in class Shape doesn’t return useful value

public double area() { return 0.0; }

Problem: How to force subclasses to override area?

Problem: How to ban creation of Shape objects

Every subclass has a different area() function

We are dealing with shapes that have areas: Circles, Rectangles, Triangles, Polyhedrons, Squares, etc.

Therefore, each subclass has a (different) function area(), which returns its area.
Abstract class and method solves both problems

Abstract class. Means can’t create object of Shape: new Shape(...) syntactically illegal

public abstract class Shape {
	...
	public abstract double area();
	...
}

Body is replaced by ;

Abstract method. Means it must be overridden in any subclass

Place abstract method only in abstract class.

Can extend only one class

public class C extends C1, C2 {
	...
}

Use abstract classes? Seems OK, because method bodies not given!

But Java does not allow this, because abstract classes can have non-abstract methods

Instead, Java has a construct, the interface, which is like an abstract class but has more restrictions.

Interfaces

An interface is a fully abstract class with a slightly different syntax.

An interface can contain type signatures for methods, just like abstract methods in abstract classes, but they have to be public.

An interface can contain fields, but they have to be public, static, and final and they have to contain an initializer. So they are really just constants

Interface declaration and use of an interface

public class C implements C1, C2 {
	...
}

Methods declared in interface are automatically public, abstract

Use of public, abstract is optional

Use ; not { ... }

Field declared in interface automatically public, static, final

Must have initialization

Use of public, static, final optional

Eclipse: Create new interface? Create new class, change keyword class to interface

Casting with interfaces

class B extends A implements C1, C2 { ... }

b = new B();

What does object b look like?

Object b has 5 perspectives. Can cast to any one of them at any time. Examples:

(C2) b
(32)
(32)
(Object) b
(C1) b

You’ll see such casting later

Add C1, C2 as new dimensions:
Same rules apply to classes and interface

class B extends A implements C1, C2 { … } 
interface C1 { … } 
interface C2 { … } 
class A { … }

$B = \text{new } B();$
$C2 c = b;$

$\text{c.m(...)} \text{ calls overriding m declared in B}$

$\text{c.m(...)} \text{ syntactically legal only if m declared in C2}$

Look at: interface java.lang.Comparable

```java
/** Comparable requires method compareTo */
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 cannot be cast to the class of this object. */
    int compareTo(Object c);
}
```

Classes that implement Comparable:
- Boolean
- Byte
- Double
- Integer
- String
- BigDecimal
- BigInteger
- Calendar
- Time
- Timestamp

In class java.util.Arrays:

```java
public static void sort (Comparable[] a) {…}
```

Which class should implement Comparable?

First idea: all the subclasses
Circle, Rect, …

Doesn’t work! Each element of b has static type Shape, and compareTo isn’t available in Shape partition

Use this. Shape must implement Comparable

```java
b = … [Shape];
```

Shape should implement Comparable

```java
Shape[] b = …
```

In class java.util.Arrays:

```java
public static void sort (Comparable[] a) {…}
```

Cast from Shape[] to Comparable[] happens automatically

```java
Shape[] b = …
```
Class Shape implements Comparable

```java
public abstract class Shape implements Comparable {
    /** If c is not a Shape, throw a CastClase exception.
     * Otherwise, return neg number, 0, or pos number
     * depending on whether this shape has smaller area than c,
     * same area, or greater area */
    public @Override int compareTo(Object c) {
        return area() - ((Shape) c).area();
    }
    ...
}
```

We take advantage of the fact that we don’t have to return -1, 0, or 1! Simpler code.

Java Library static methods:

```java
Arrays.sort(Comparable[] a)
```

Class Arrays has many other useful static methods.

**Beauty of interfaces:**

`Arrays.sort` sorts an array or list for any class C, as long as C implements interface Comparable — and thus implements `compareTo` to say which of two elements is bigger.