COMP 250

Lecture 15

inheritance 3:

interfaces,
abstract classes,
polymorphism

Feb. 9, 2022

Java API

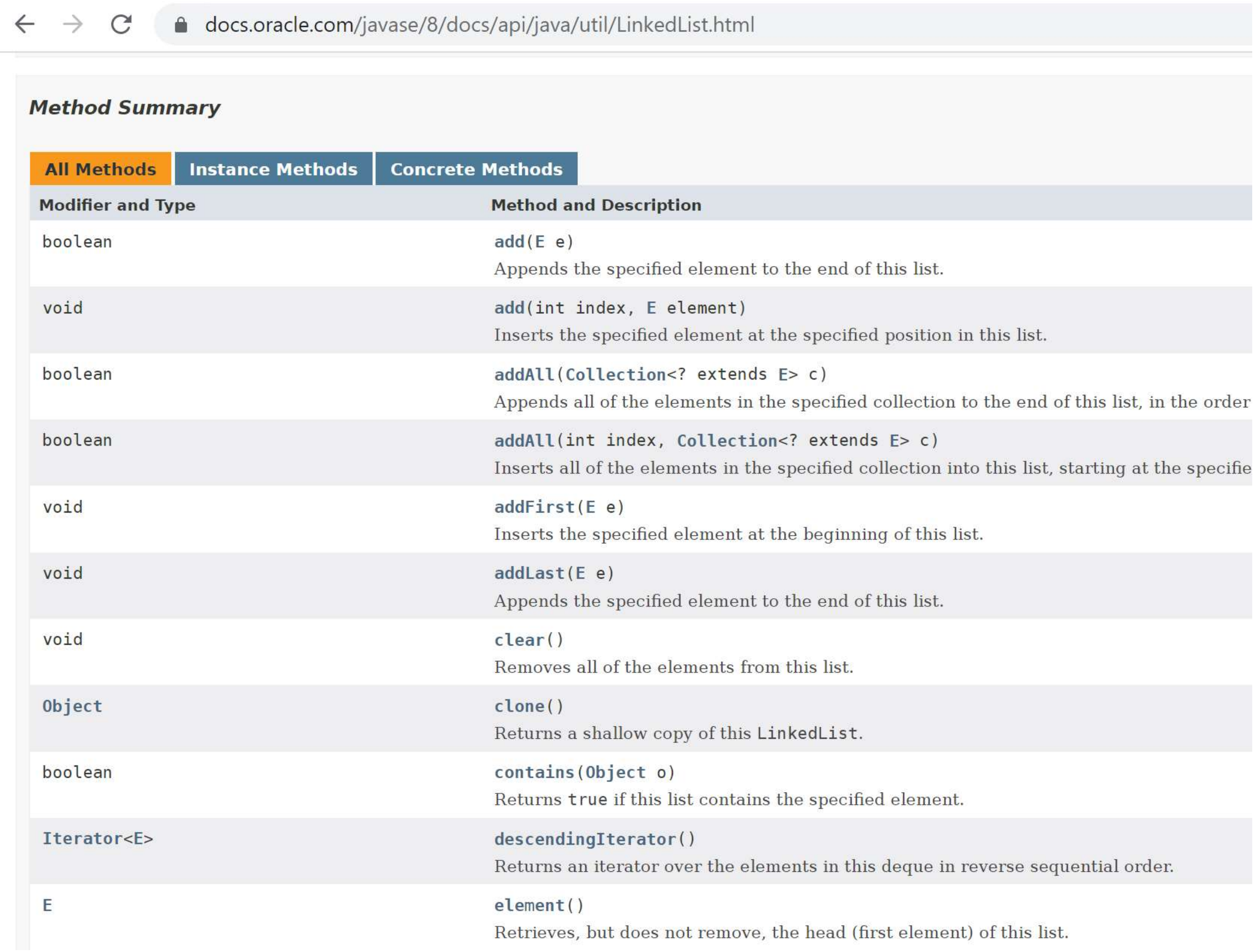
You have heard the word *interface* before:

Java API is the "application programming interface".

It tells you for each class (e.g. LinkedList), what the fields and methods are and what the methods do.

It does *not* tell you how the methods are implemented. An "interface" hides these details from you.

e.g. Java API for LinkedList



Method Summary

Modifier and Type	Method and Description
`boolean`	`add(E e)` Appends the specified element to the end of this list.
`void`	`add(int index, E element)` Inserts the specified element at the specified position in this list.
`boolean`	`addAll(Collection<? extends E> c)` Appends all of the elements in the specified collection to the end of this list, in the order
`boolean`	`addAll(int index, Collection<? extends E> c)` Inserts all of the elements in the specified collection into this list, starting at the specifie
`void`	`addFirst(E e)` Inserts the specified element at the beginning of this list.
`void`	`addLast(E e)` Appends the specified element to the end of this list.
`void`	`clear()` Removes all of the elements from this list.
`Object`	`clone()` Returns a shallow copy of this `LinkedList`.
`boolean`	`contains(Object o)` Returns `true` if this list contains the specified element.
`Iterator<E>`	`descendingIterator()` Returns an iterator over the elements in this deque in reverse sequential order.
`E`	`element()` Retrieves, but does not remove, the head (first element) of this list.

Java `interface`

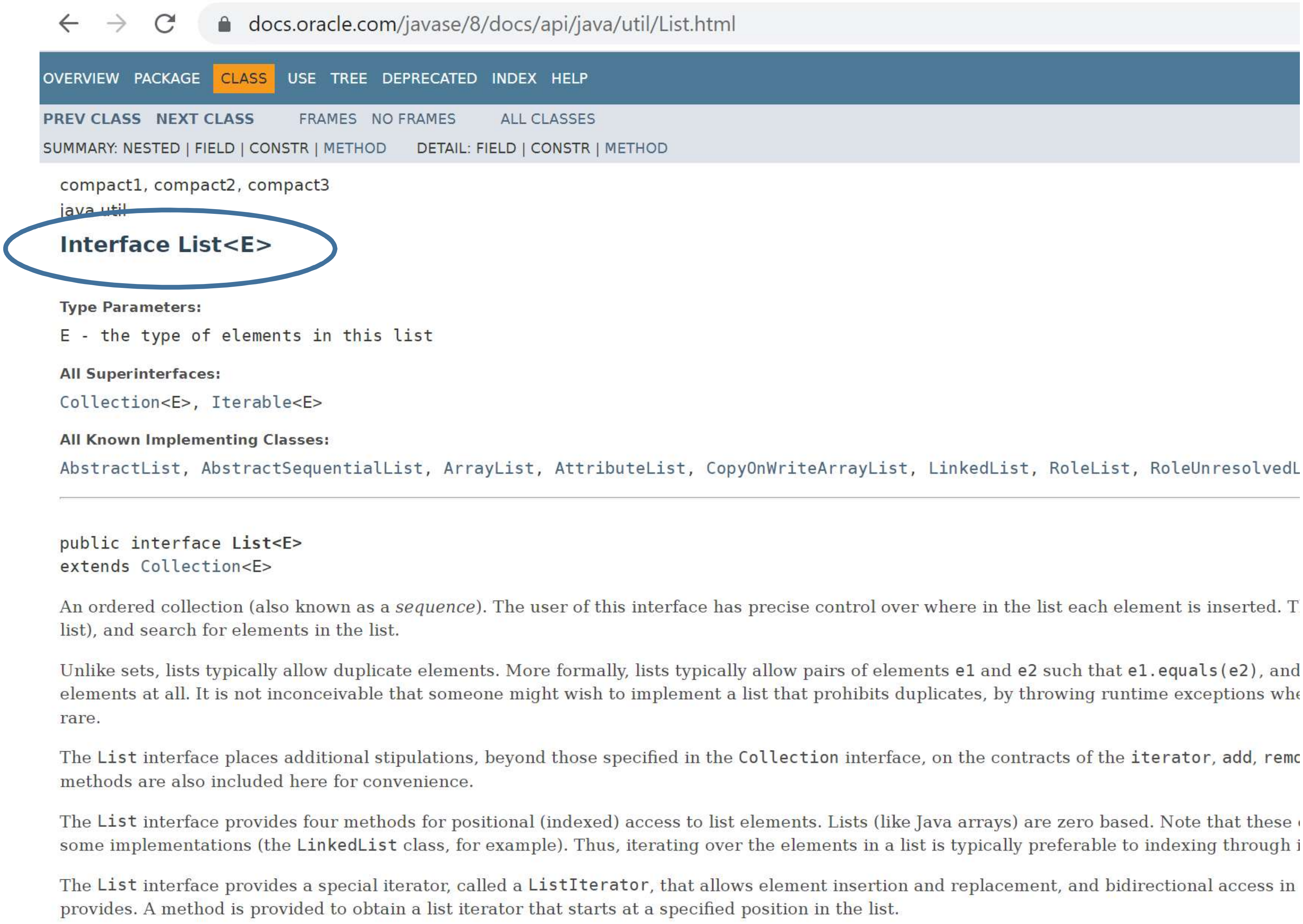
A Java `interface` is something else.

`interface` is a reserved word in the Java language.

A Java is interface like a class, but the methods have no bodies.

Example: List interface

```
interface  List<T>  {
    void        add(T);
    void        add(int,  T);
    T            remove(int);
    boolean   isEmpty();
    T            get( int );
    int          size();
           :
}
```


docs.oracle.com/javase/8/docs/api/java/util/List.html
OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP
PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

compact1, compact2, compact3

java.util

Interface List<E>

Type Parameters:

`E - the type of elements in this list`

All Superinterfaces:

`Collection<E>, Iterable<E>`

All Known Implementing Classes:

`AbstractList, AbstractSequentialList, ArrayList, AttributeList, CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedL`

```
public interface List<E>
extends Collection<E>
```

An ordered collection (also known as a *sequence*). The user of this interface has precise control over where in the list each element is inserted. T list), and search for elements in the list.

Unlike sets, lists typically allow duplicate elements. More formally, lists typically allow pairs of elements `e1` and `e2` such that `e1.equals(e2)`, and elements at all. It is not inconceivable that someone might wish to implement a list that prohibits duplicates, by throwing runtime exceptions whe rare.

The `List` interface places additional stipulations, beyond those specified in the `Collection` interface, on the contracts of the `iterator`, `add`, `remo` methods are also included here for convenience.

The `List` interface provides four methods for positional (indexed) access to list elements. Lists (like Java arrays) are zero based. Note that these some implementations (the `LinkedList` class, for example). Thus, iterating over the elements in a list is typically preferable to indexing through i

The `List` interface provides a special iterator, called a `ListIterator`, that allows element insertion and replacement, and bidirectional access in provides. A method is provided to obtain a list iterator that starts at a specified position in the list.

```
class  ArrayList<T>   implements  List<T> {

      void          add(T)          { .... }
      void          add(int,  T)   {  ....  }
      T             remove(int) {  ....  }
      boolean    isEmpty()     {  ....  }
      T             get( int )      {  ....  }
      int           size()          {  ....  }
      void          ensureCapacity(int) { ... }
      void          trimToSize( ) { ... }
               :
}
```

Here I have not listed parameter names, but yes they need to be there too.

Each of the List methods is implemented in ArrayList<T>.
Other methods are also implemented.

```
class  LinkedList<T>   implements  List<T> {

      void        add(T)          { .... }
      void        add(int,  T)   {  ....  }
      T            remove(int) {  ....  }
      boolean    isEmpty()     {  ....  }
      T            get( int )      {  ....  }
      int           size()          {  ....  }
      void        addFirst(T)  {  .... }
      void        addLast(T)  {  .... }
               :
}
```

Here I have not listed parameter names, but yes they need to be there too.

Each of the List methods is implemented in LinkedList<T>.

Other methods are also implemented.

How are Java interface’s used ?

```
List<String>       list;

list  =  new  ArrayList<String>();
list.add( "hello“ );
            :
list  =  new  LinkedList<String>();
list.add( “goodbye” );
```

How are Java interface’s used ?

```
void   someFlexibleListMethod(   List<String>   list ){
       :
   list.add("hello");
       :
   list.remove( 3 );
}
```

someFlexibleListMethod() can be called with either a LinkedList<String> or an ArrayList<String> argument.

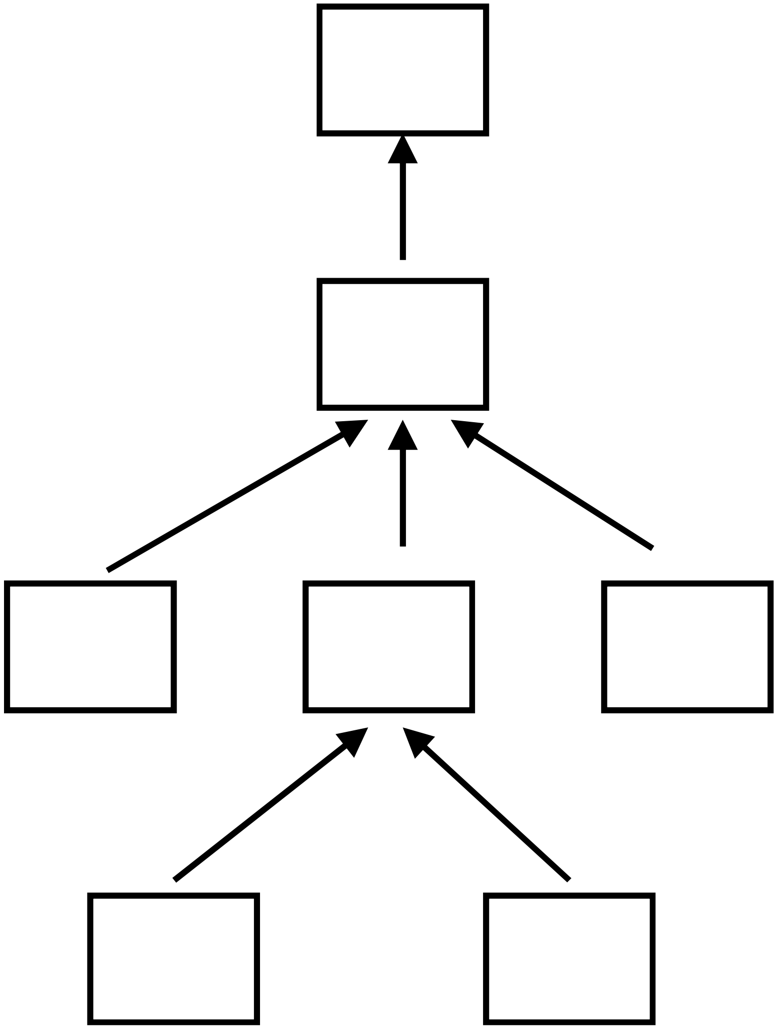
How are Java interface’s used ?

```
void   someFlexibleListMethod(   List<String>   list ){
      :
   list.add("hello");
      :
   list.remove( 3 );

   list.addFirst(  “goodbye” );     ✗
}
```

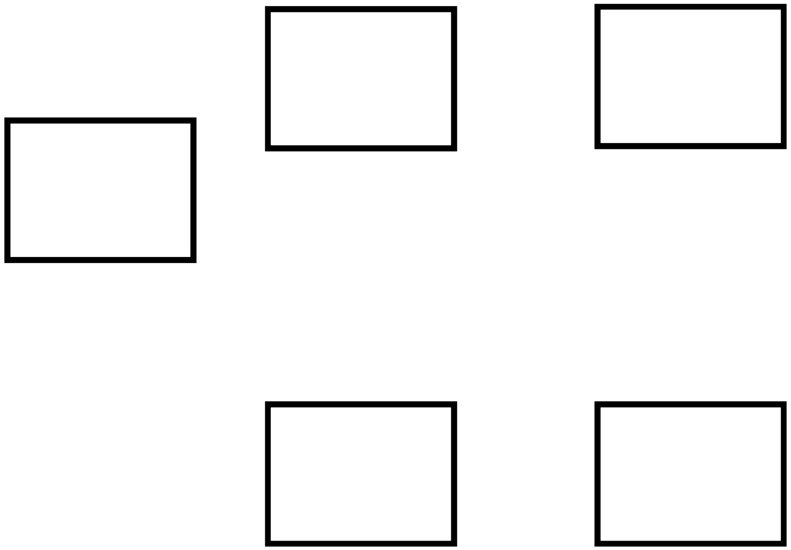
The list interface does not have an addFirst() method.

Use add(0, “goodbye”) instead.

classes

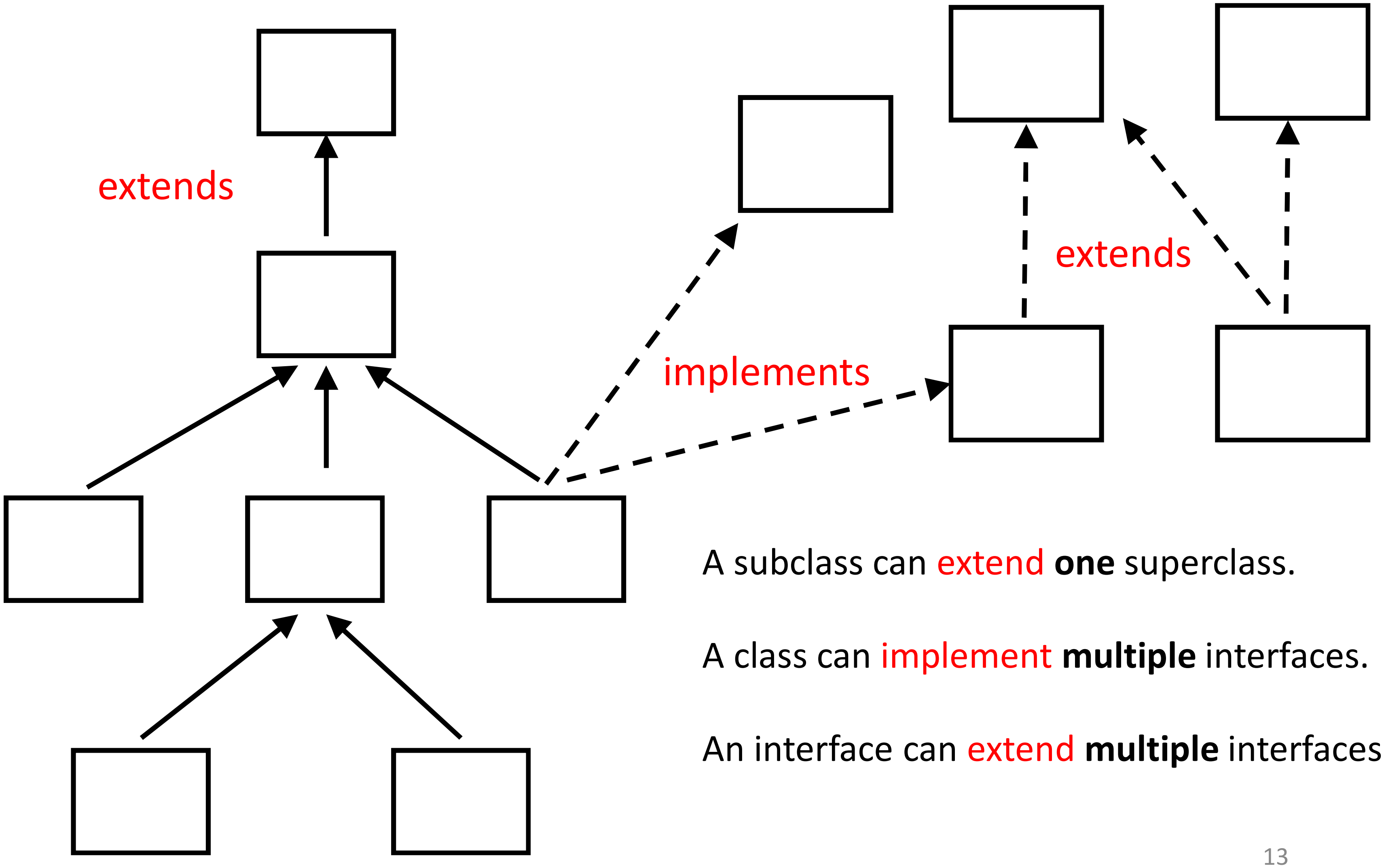


interfaces



classes

interfaces



extends

implements

extends

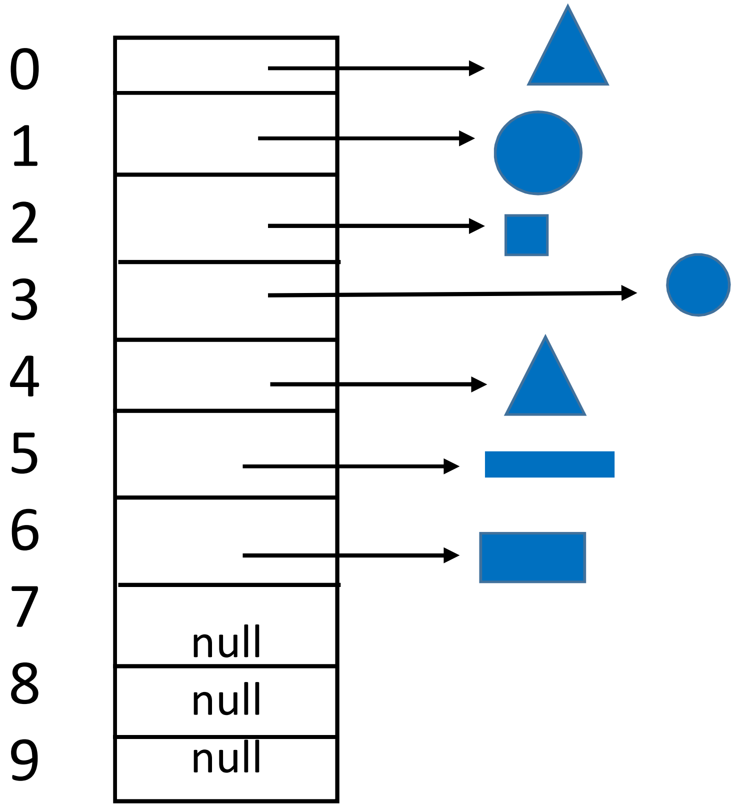
A subclass can extend **one** superclass.

A class can implement **multiple** interfaces.

An interface can extend **multiple** interfaces.

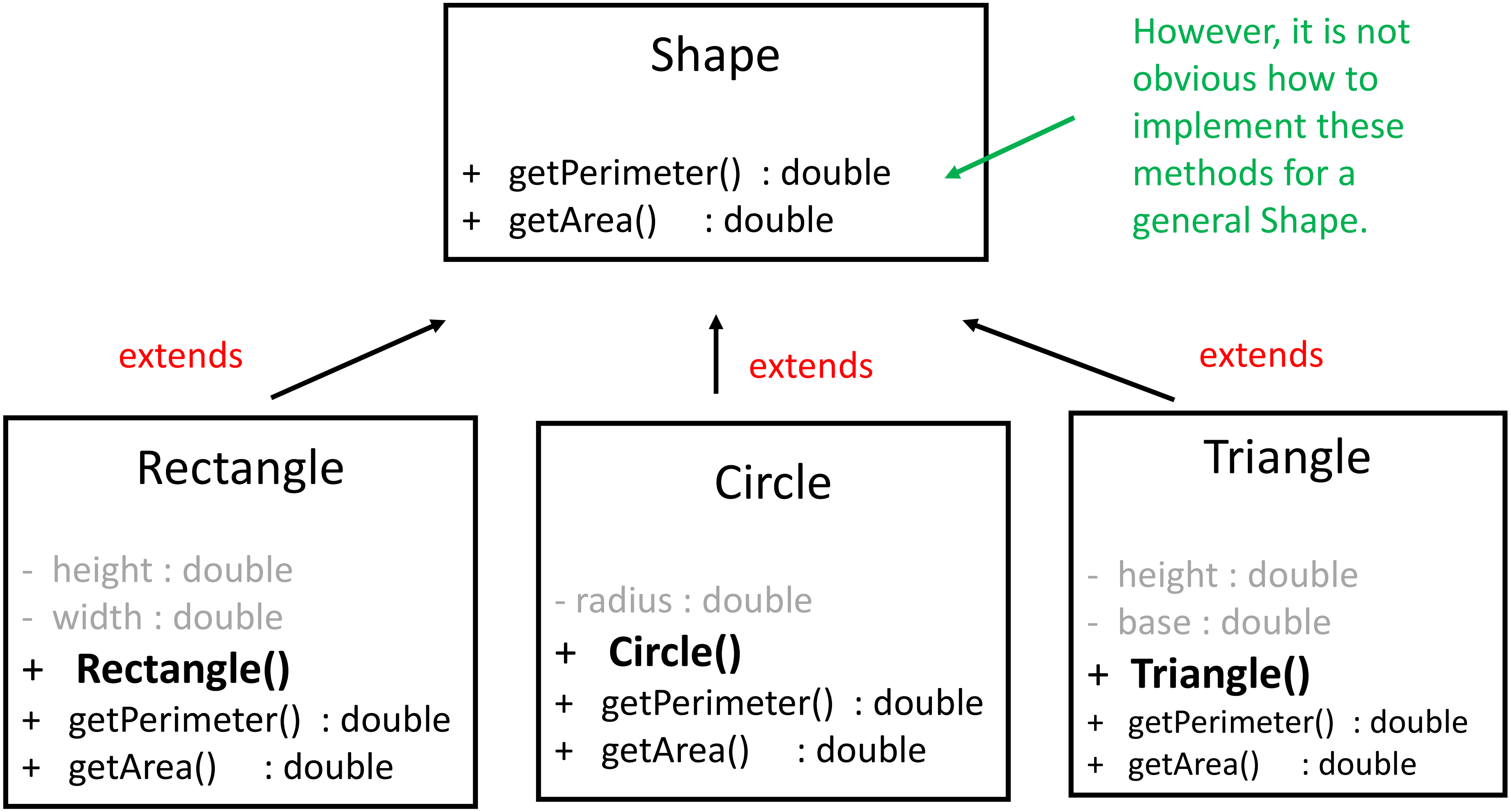
Example

Recall lecture 8 :
ArrayList<Shape>



Q: How should we design classes so that we can have different types of shapes e.g. Rectangle, Triangle, Circle, ...?

A: *Without using interfaces*, we might try this:



Shape

+ getPerimeter() : double
+ getArea() : double

However, it is not obvious how to implement these methods for a general Shape.

extends

extends

extends

Rectangle

- height : double
- width : double
+ **Rectangle()**
+ getPerimeter() : double
+ getArea() : double

Circle

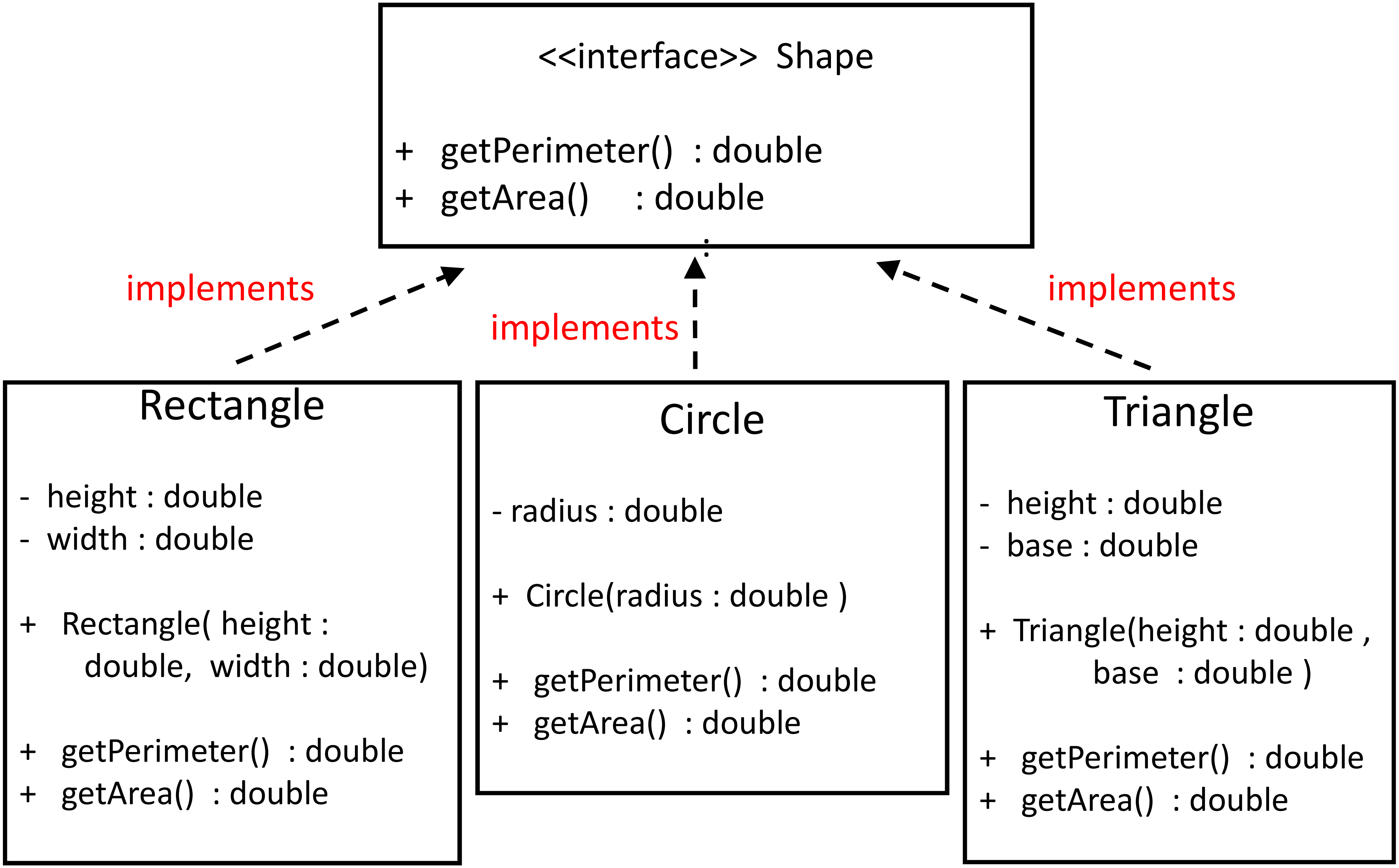
- radius : double
+ **Circle()**
+ getPerimeter() : double
+ getArea() : double

Triangle

- height : double
- base : double
+ **Triangle()**
+ getPerimeter() : double
+ getArea() : double

We ignore color of the shape...

Instead we could use an interface for Shape.



<<interface>> Shape

+ getPerimeter() : double
+ getArea() : double

implements

implements

implements

Rectangle

- height : double
- width : double

+ Rectangle(height : double, width : double)

+ getPerimeter() : double
+ getArea() : double

Circle

- radius : double

+ Circle(radius : double)

+ getPerimeter() : double
+ getArea() : double

Triangle

- height : double
- base : double

+ Triangle(height : double , base : double)

+ getPerimeter() : double
+ getArea() : double

```
interface  Shape {
      double  getPerimeter();
      double  getArea();
}
```

Don’t provide implementation.
No curly brackets.

```
class   Rectangle  implements  Shape{
      double  height;
      double  width;
      Rectangle(double height, double width){...}
      double  getPerimeter() { ...};
      double  getArea() { ... } ;
}
```

See next two
slides for details.

```
class   Circle implements  Shape{
      double  radius;
      Circle (double height, double width){...}
      double  getPerimeter() { ...};
      double  getArea() { ... } ;
}

etc...   Triangle
```

```
class   Rectangle  implements  Shape{

      double  height,  width;

      Rectangle(  double h,  double w ){
               height = h;   weight = w;
      }

      double   getArea(){   return height * width;   }

      double   getPerimeter(){   return 2*(height +
                                       width);  }


}
```

```
class   Circle  implements  Shape{

      double  radius;

      Circle(  double r ){ radius = r; }

      double getArea(){   return   MATH.PI * radius
                                             * radius; }

      double getPerimeter(){ return 2*MATH.PI * radius }

}
```

….. similarly for Triangle

How are Java interface’s used ?

Example:

```
Shape  s  =   new  Rectangle( 30, 40 );

       s  =   new   Circle(  2.5  );

       s  =   new   Triangle(  4.5,  6.3  );
```

COMP 250

Lecture 15

inheritance 3:

interfaces,

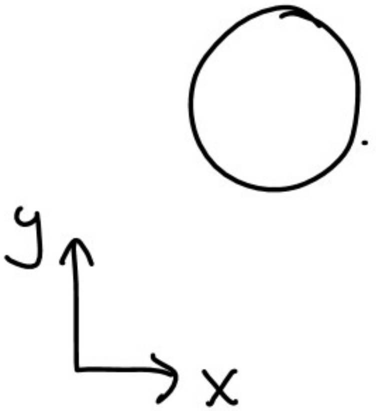
abstract classes,

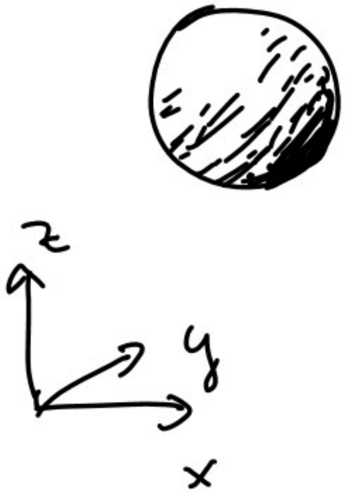
polymorphism

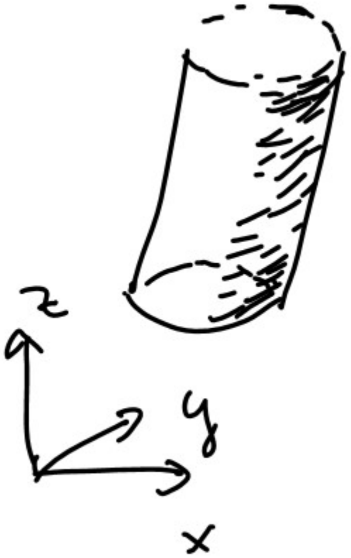
Feb. 9, 2022

Abstract Classes

Motivating Example: Circular



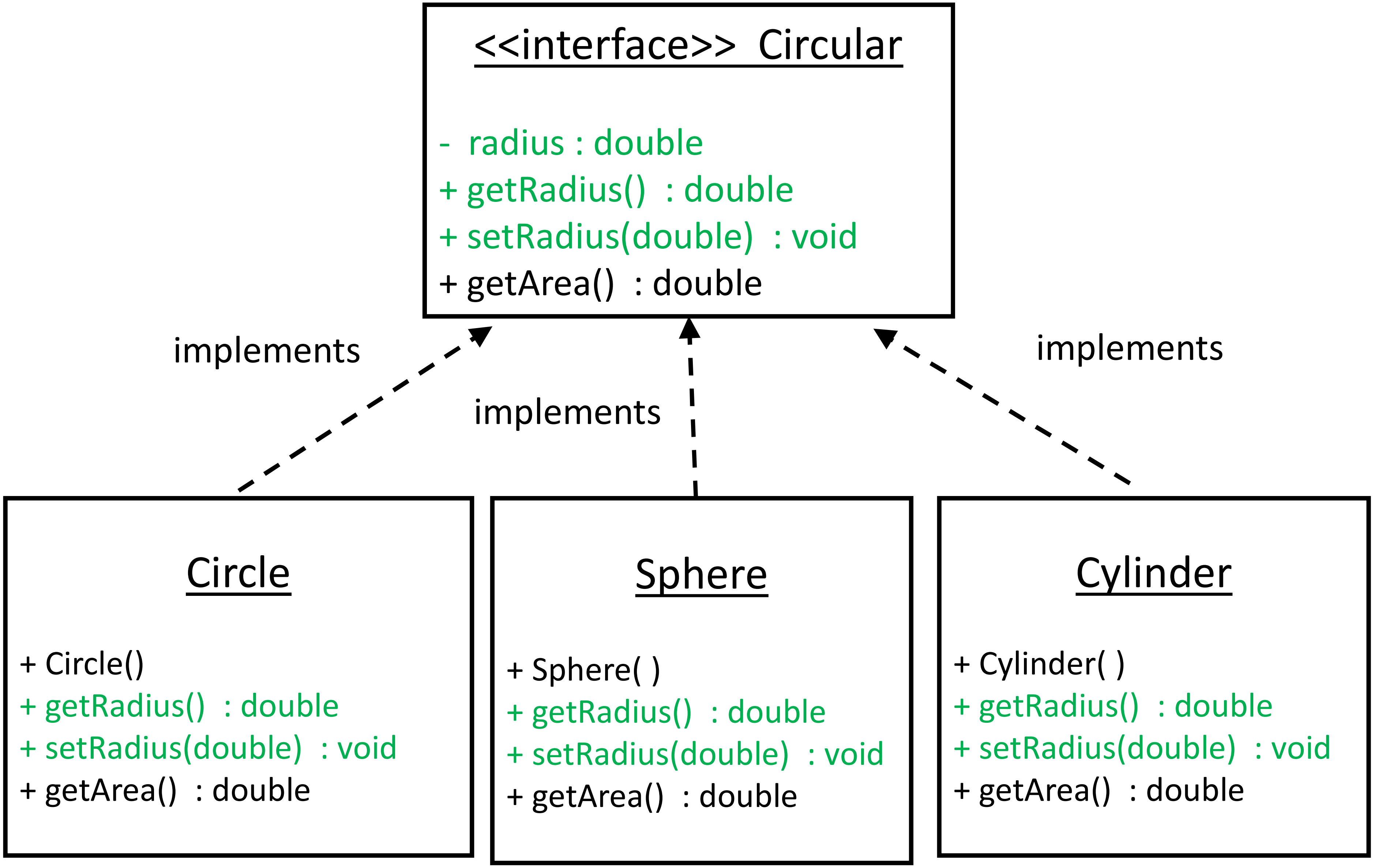




Circle Sphere Cylinder

These objects all have a radius and an area.

Using an interface here would create some redundancies.



<<interface>> Circular

- radius : double
+ getRadius() : double
+ setRadius(double) : void
+ getArea() : double

implements

implements

implements

Circle

+ Circle()
+ getRadius() : double
+ setRadius(double) : void
+ getArea() : double

Sphere

+ Sphere( )
+ getRadius() : double
+ setRadius(double) : void
+ getArea() : double

Cylinder

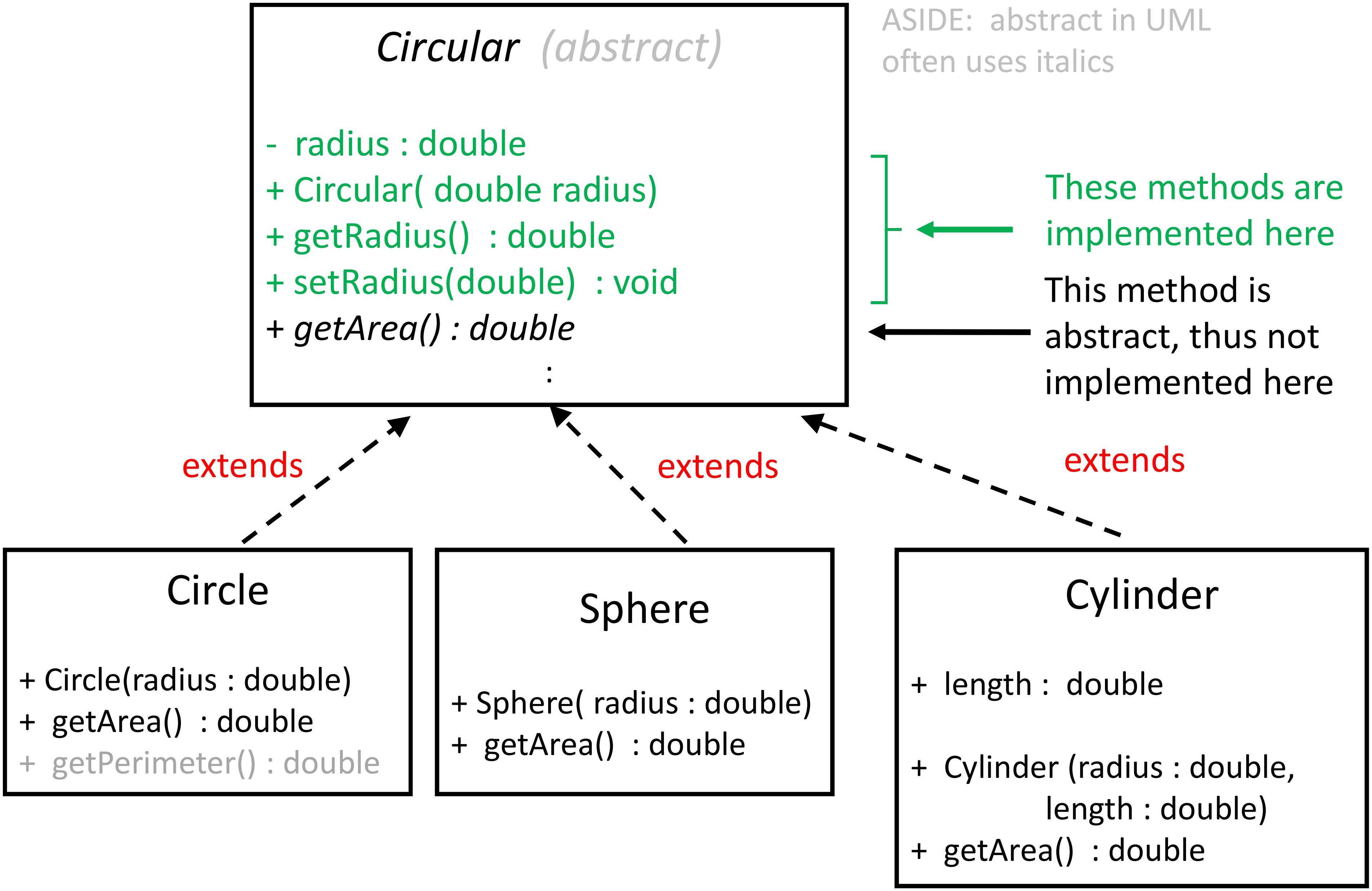
+ Cylinder( )
+ getRadius() : double
+ setRadius(double) : void
+ getArea() : double

Abstract Class

- `abstract` is a reserved word in Java

- An abstract class is a hybrid between an interface and a class
  - Like an interface, it can have methods without bodies.
  - Like a class, it can have fields and methods with bodies.

- An abstract class cannot be instantiated. But it has constructor(s) which are called by the sub-classes.



Circular (*abstract*)

- radius : double
+ Circular(double radius)
+ getRadius() : double
+ setRadius(double) : void
+ *getArea() : double*
:

ASIDE: abstract in UML often uses italics

These methods are implemented here

This method is abstract, thus not implemented here

extends

extends

extends

Circle

+ Circle(radius : double)
+ getArea() : double
+ getPerimeter() : double

Sphere

+ Sphere(radius : double)
+ getArea() : double

Cylinder

+ length : double

+ Cylinder (radius : double,
length : double)
+ getArea() : double

The getArea() method is implemented separately for each subclass.

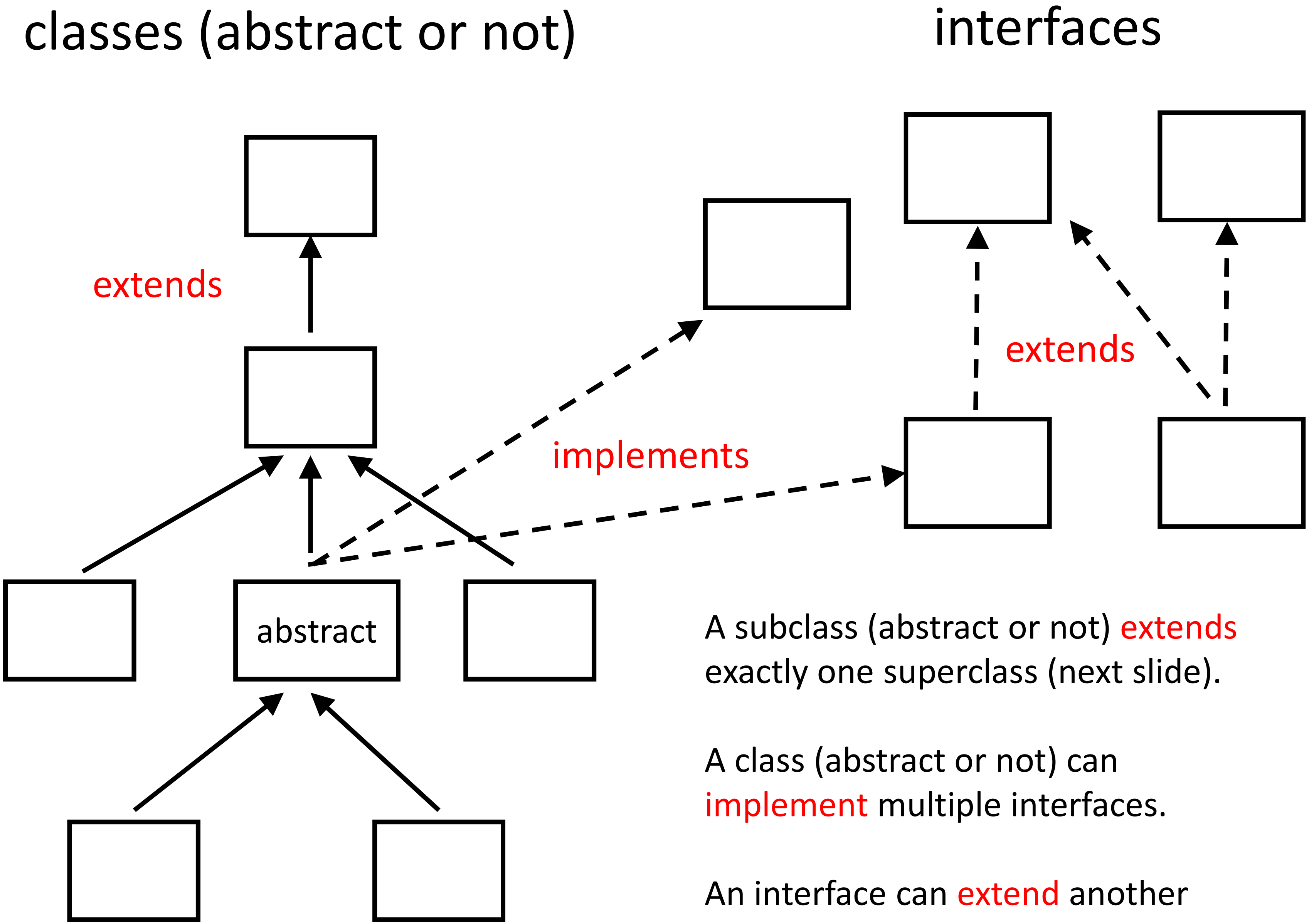
```
abstract class Circular {

        double radius;

        Circular(double radius){            //  constructor
                this.radius = radius;
        }

        double getRadius(){
                return radius;
        }

        void setRadius(double r){
                this.radius = r;
        }

        abstract double  getArea();
}
```

This method is abstract, thus not implemented here

```
class Circle extends Circular{

    Circle(double radius){           // constructor
            super(radius);          //   initialize superclass field
    }

    double getArea(){
         double r = this.getRadius();
         return  Math.PI * r*r;
     }

     double getPerimeter(){   return 2*MATH.PI * this.getRadius();  }
}
```

This method is not part of Circular abstract class.
This method would make no sense in the Sphere class.

```
class Cylinder extends Circular{

        double height;

        Cylinder(double radius, double h){                // constructor
                super(radius);
                this.height = h;
        }

        double getArea(){
                double r = this.getRadius();
                return  2 * Math.PI * radius * height;
        }
}
```

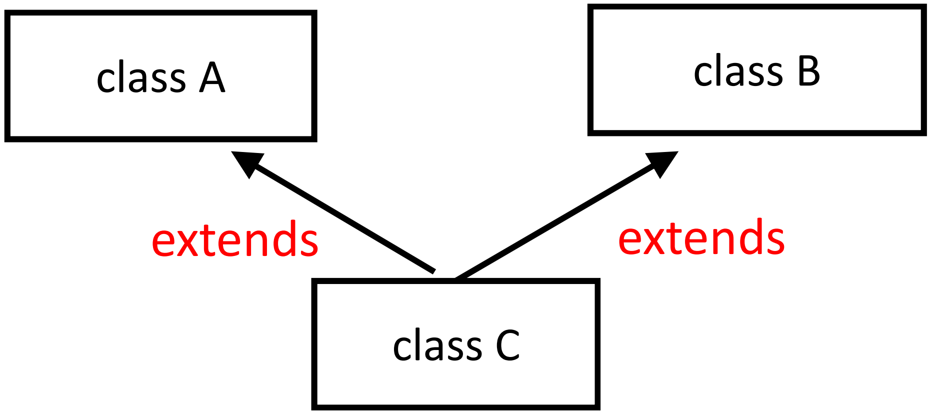


A subclass (abstract or not) extends exactly one superclass (next slide).

A class (abstract or not) can implement multiple interfaces.

An interface can extend another interface.

A class (abstract or not) cannot extend more than one class (abstract or not).



Not allowed! ✗

Why not ?

A problem could occur if two superclasses have two methods with the same signature, but different implementations.
Which would be inherited by the subclass?

COMP 250

Lecture 15

inheritance 3:

interfaces,
abstract classes,
polymorphism

Feb. 9, 2022

Compile time: Reference variables have a *declared type*:

```
C       varC ;       //    C is a class
A       varA ;       //    A  is an abstract class
I       varI ;       //    I   is an interface
```

Run time: Reference variables *reference objects*.

varC can reference any object of class **C** or subclass of **C**, etc.

e.g. Cat class or subclass SiameseCat

varA can reference any object whose class extends **A**, etc.

varI can reference any object whose class implements **I**.

Polymorphism (runtime behavior)

“poly” = multiple, “morph” = form

When you write **variable.method()**, the method that is called at runtime depends on the *referenced object’s class*, not on the variable’s declared type.

Let’s consider some examples.

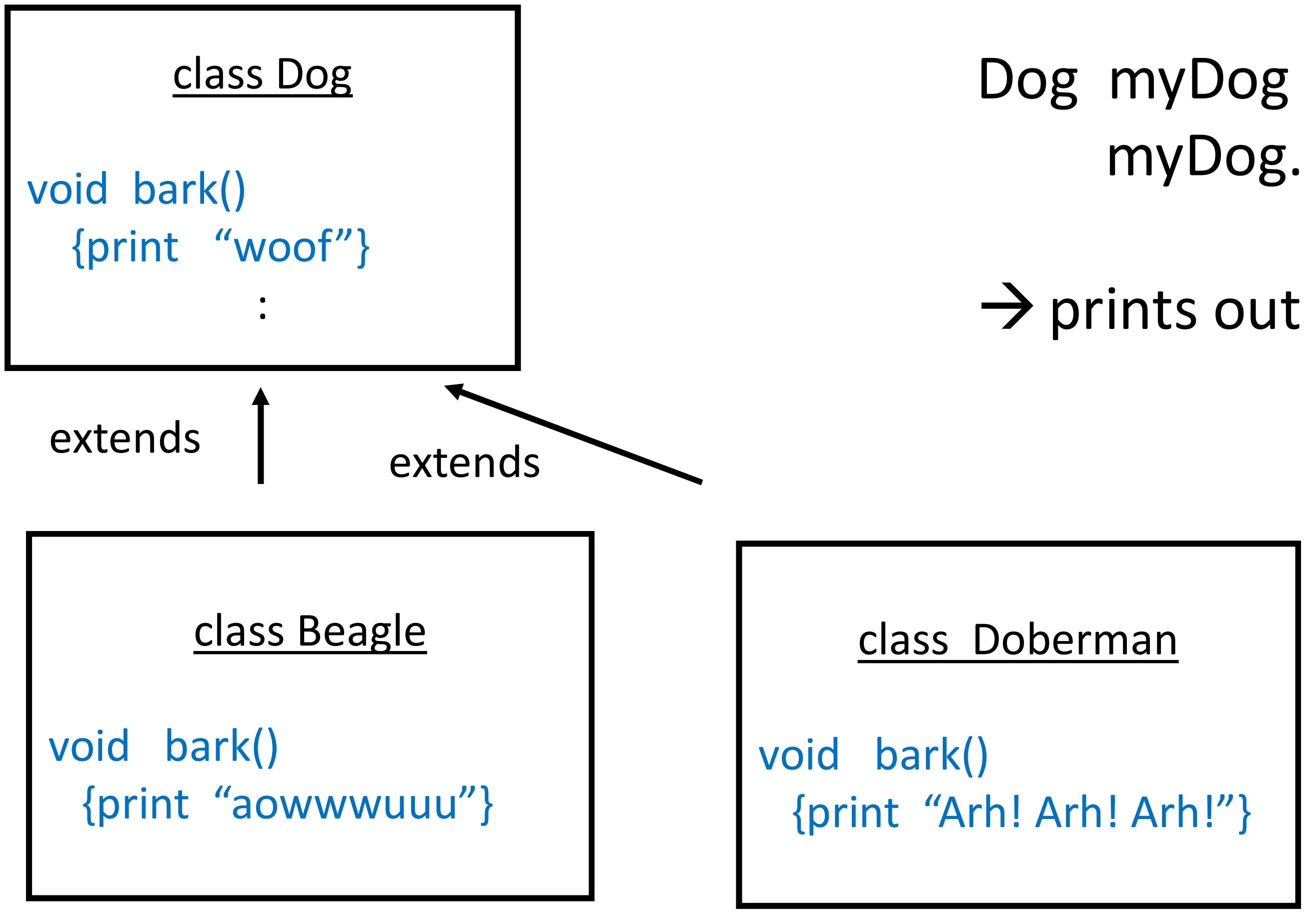
Example with Animal Classes

```
boolean  b;
Animal    pet;

//   .....

if ( b )
     pet = new Cat();
else
     pet = new Dog();

System.out.print( pet );
```

Q: Which **`toString()`** method gets called ?

A: It depends on the object that **`pet`** is referencing.

Earlier example with Dog Class



class Dog

```
void  bark()
   {print   “woof”}
                :
```

extends

extends

class Beagle

```
void   bark()
  {print  “aowwwuuu”}
```

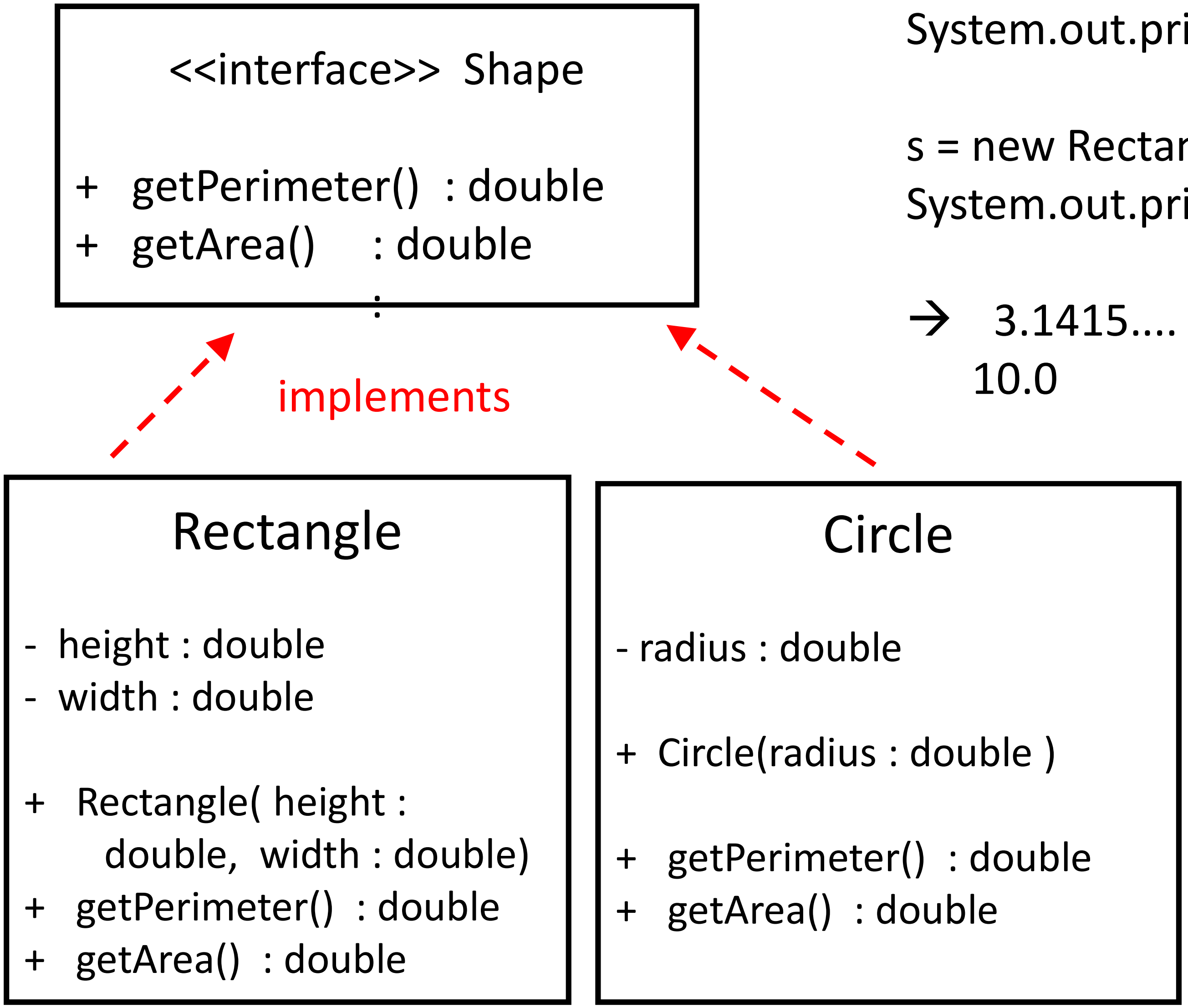
class Doberman

```
void   bark()
  {print  “Arh! Arh! Arh!”}
```

```
Dog  myDog = new Beagle();
        myDog.bark();
```

→ prints out “aowwwuuu”

Example with interface Shape



<<interface>> Shape

+ getPerimeter() : double
+ getArea() : double
:

implements

Rectangle

- height : double
- width : double

+ Rectangle(height : double, width : double)
+ getPerimeter() : double
+ getArea() : double

Circle

- radius : double

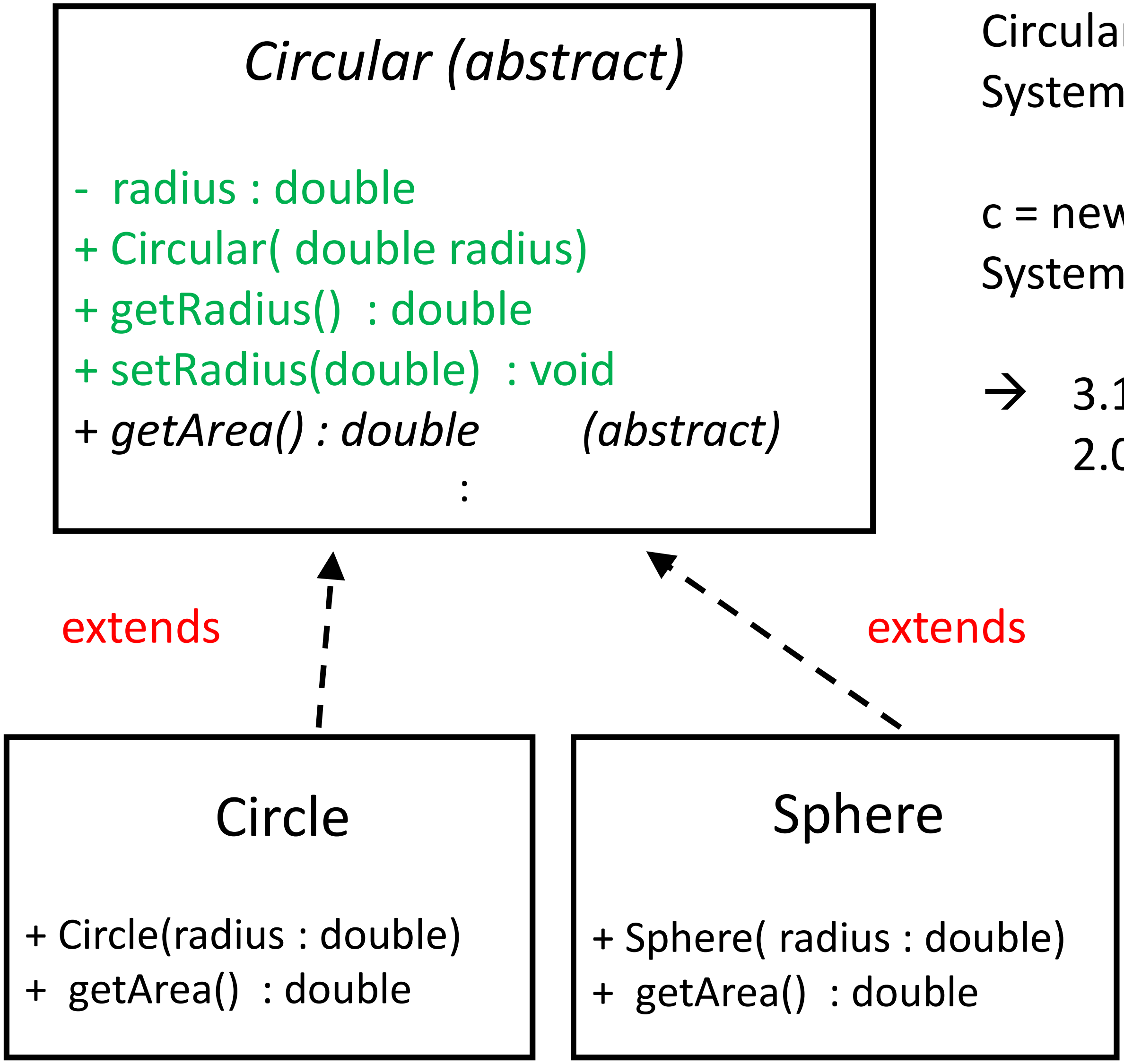
+ Circle(radius : double)

+ getPerimeter() : double
+ getArea() : double

```
Shape  s = new Circle( 1.0 );
System.out.println( s.getArea() );

s = new Rectangle( 2.0, 3.0 );
System.out.println ( s.getPerimeter() );
```

```
→  3.1415....
   10.0
```

Example with abstract class Circular



Circular (abstract)

- radius : double
+ Circular(double radius)
+ getRadius() : double
+ setRadius(double) : void
+ *getArea() : double* *(abstract)*
:

extends

extends

Circle

+ Circle(radius : double)
+ getArea() : double

Sphere

+ Sphere(radius : double)
+ getArea() : double

```
Circular  c = new Circle( 1.0 );
System.out.println( c.getArea() );

c = new Sphere( 2.0 );
System.out.println ( c.getRadius() );
```

```
→   3.1415....
    2.0
```

Coming up...

Lectures

Fri. Feb. 11

Inheritance 4 :

examples of interfaces:

comparable, iterable

Other

Thursday Feb 10 zoom tutorial

(Liam, Ricky, Kavosh)

: SLinkedList + debug mode

Assignment 1

- due on Friday, Feb. 11

Quiz 2 also on Friday, Feb. 11