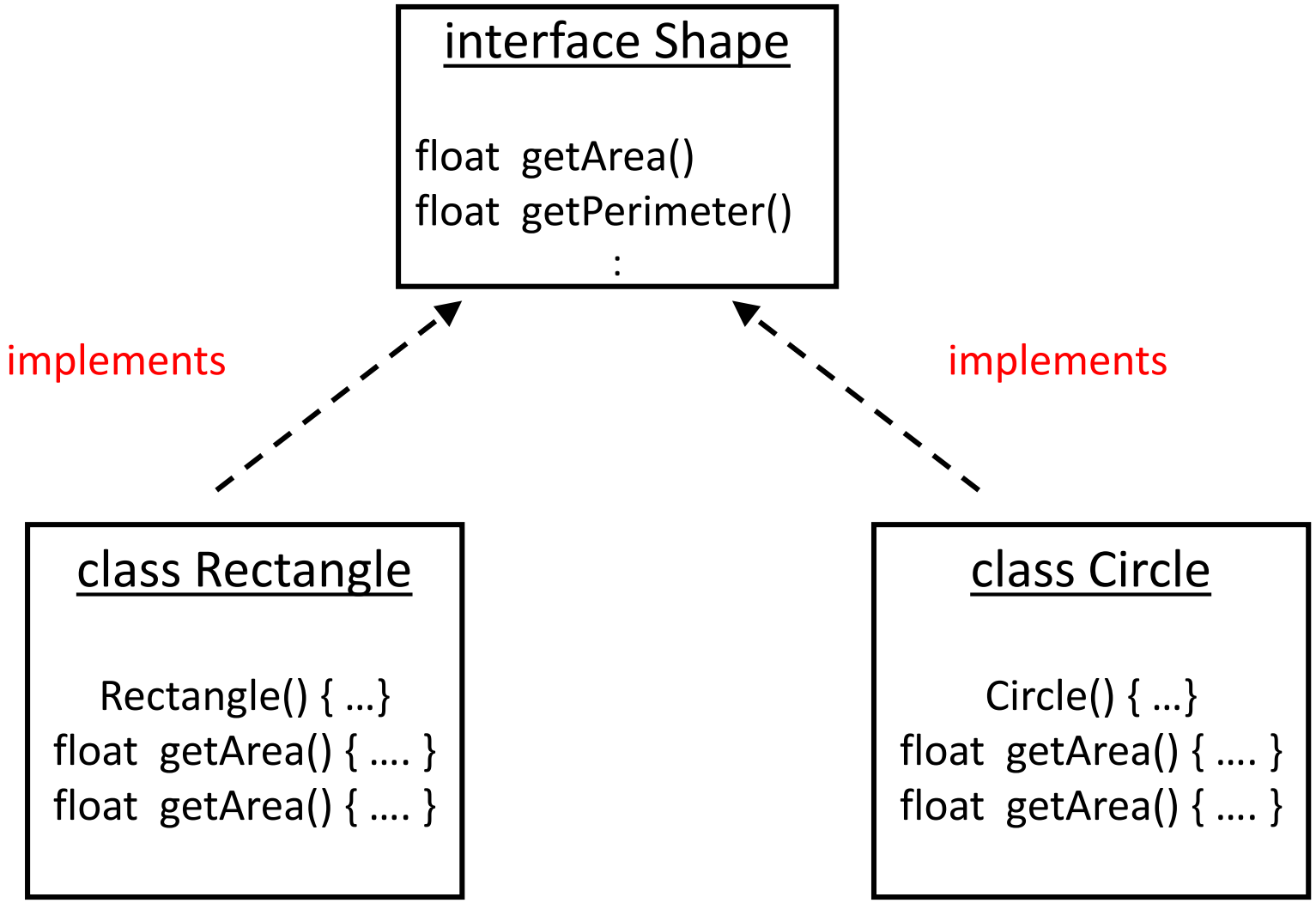
COMP 250

Lecture 31

abstract classes, type conversion

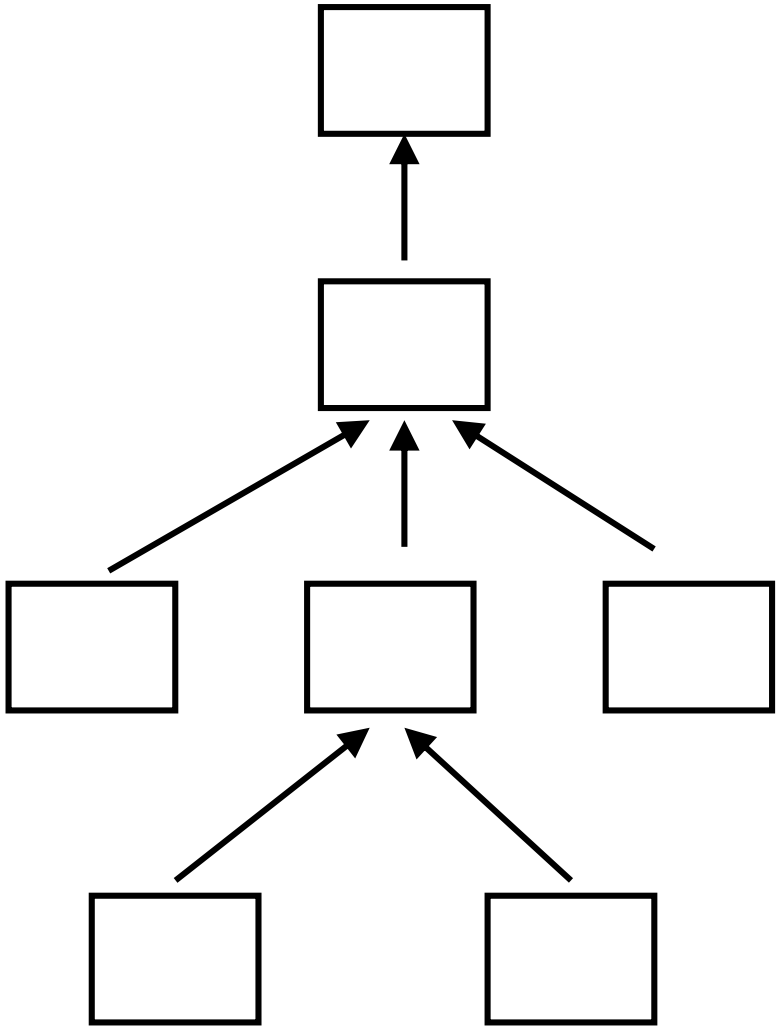
Nov. 23, 2016

RECALL: interfaces

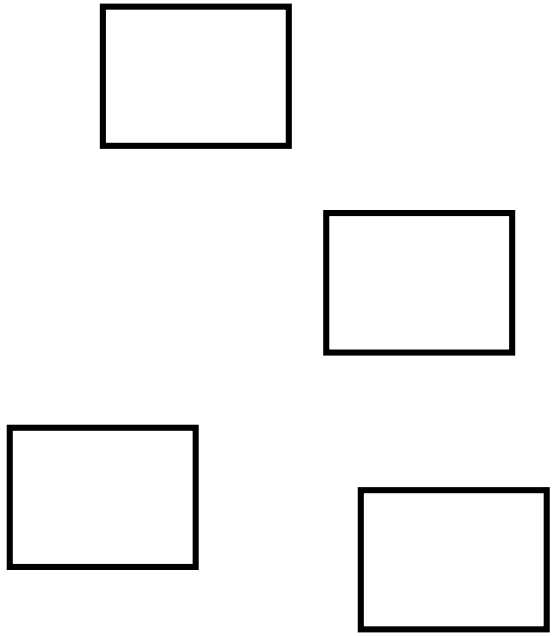


classes

(tree, parent links only)



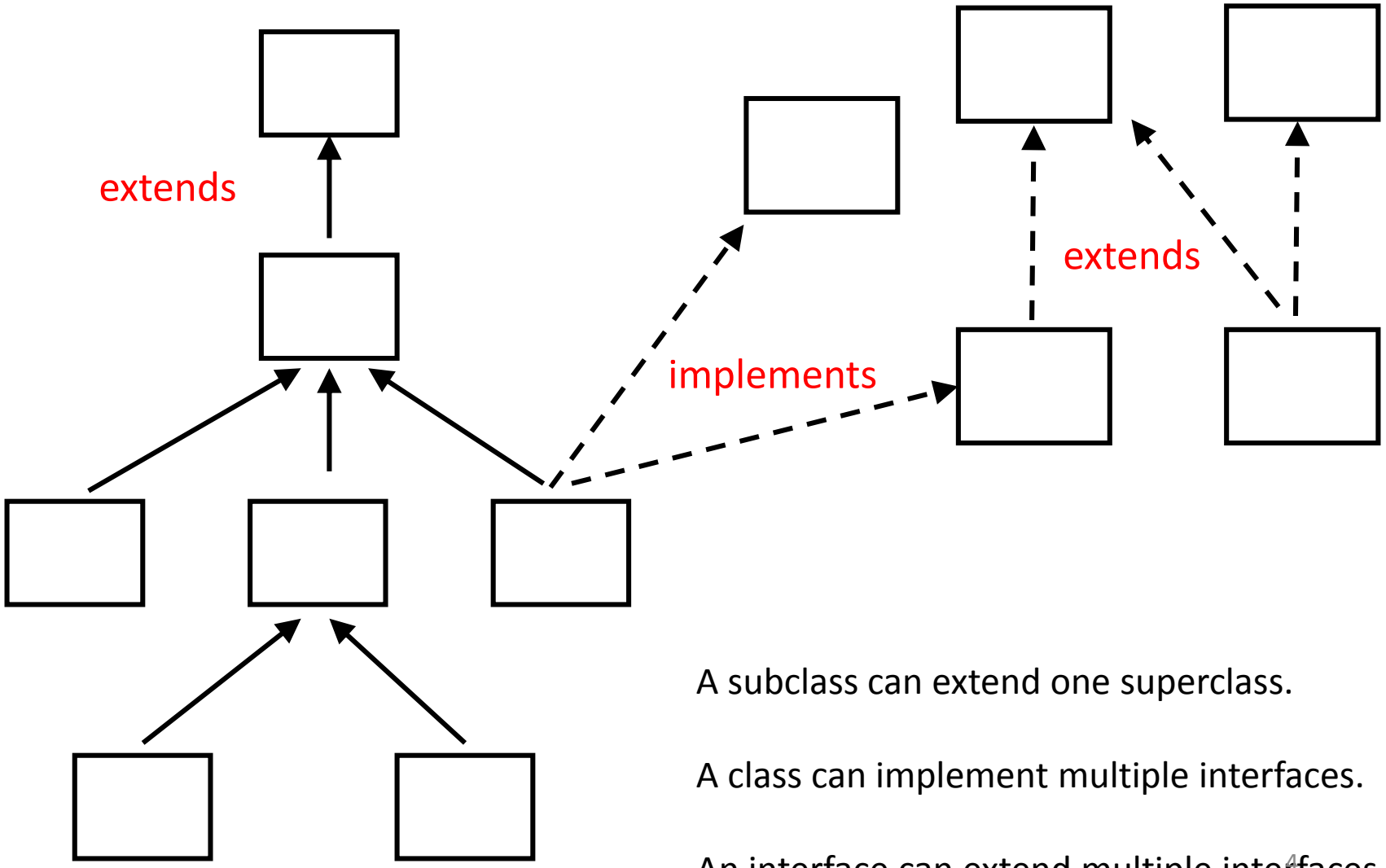
interfaces



classes

(tree, parent links only)

interfaces



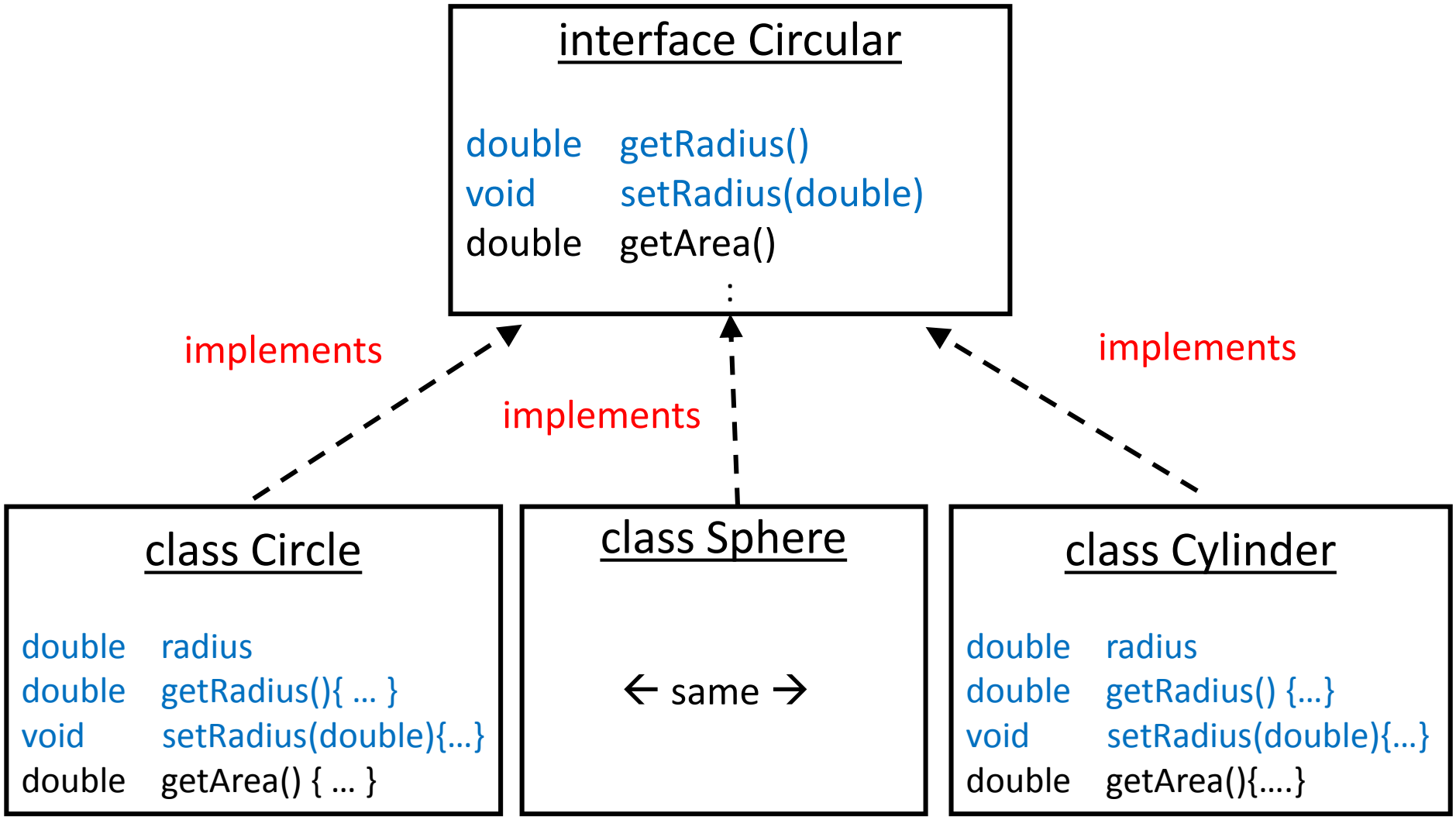
A subclass can extend one superclass.

A class can implement multiple interfaces.

An interface can extend multiple interfaces.

Example: Circular

Circle Sphere Cylinder

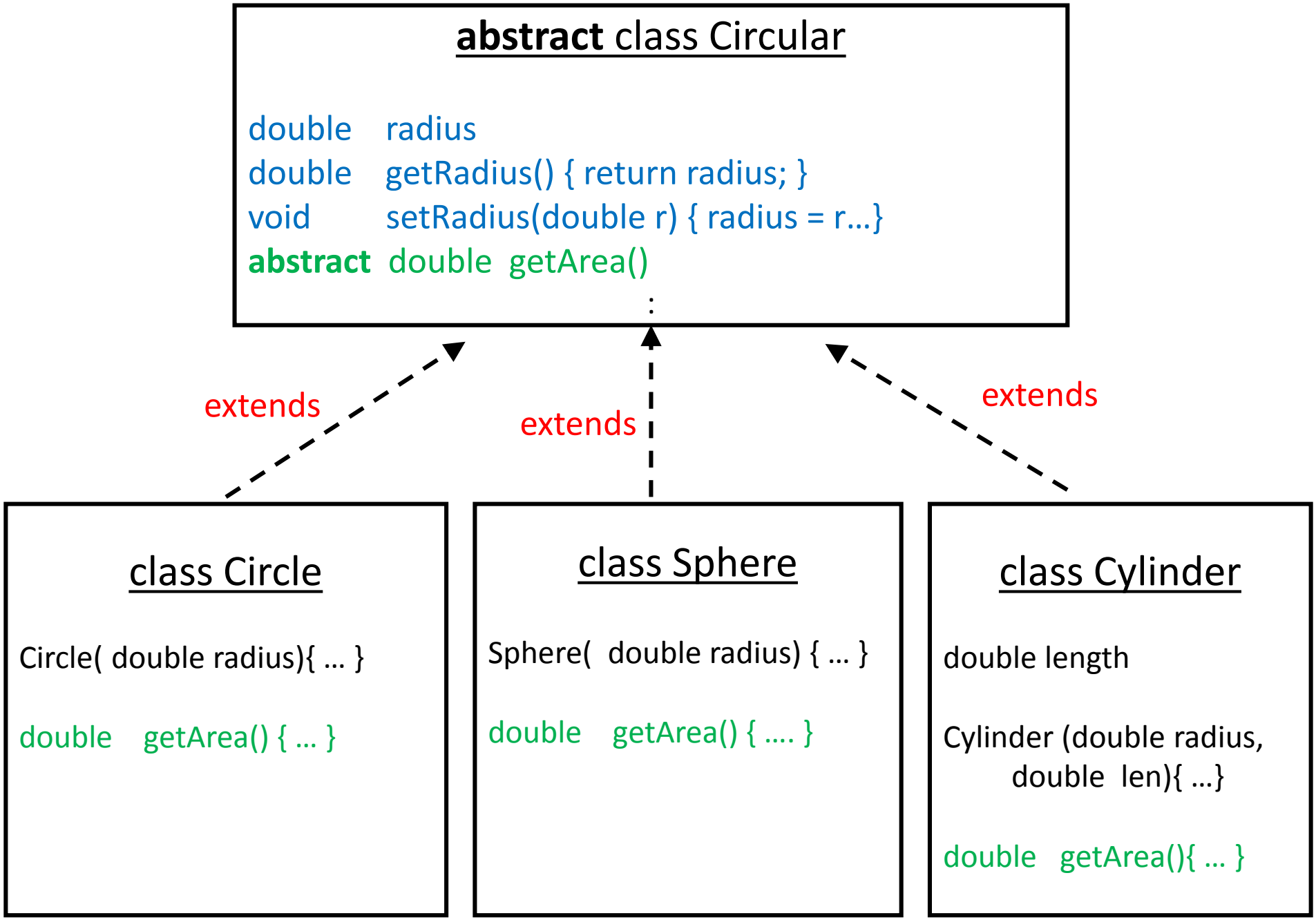


Can we avoid repeating these method definitions?

Abstract Class

- Like a class, it can have fields and methods with bodies

- Like an interface, it can have methods with only signatures.

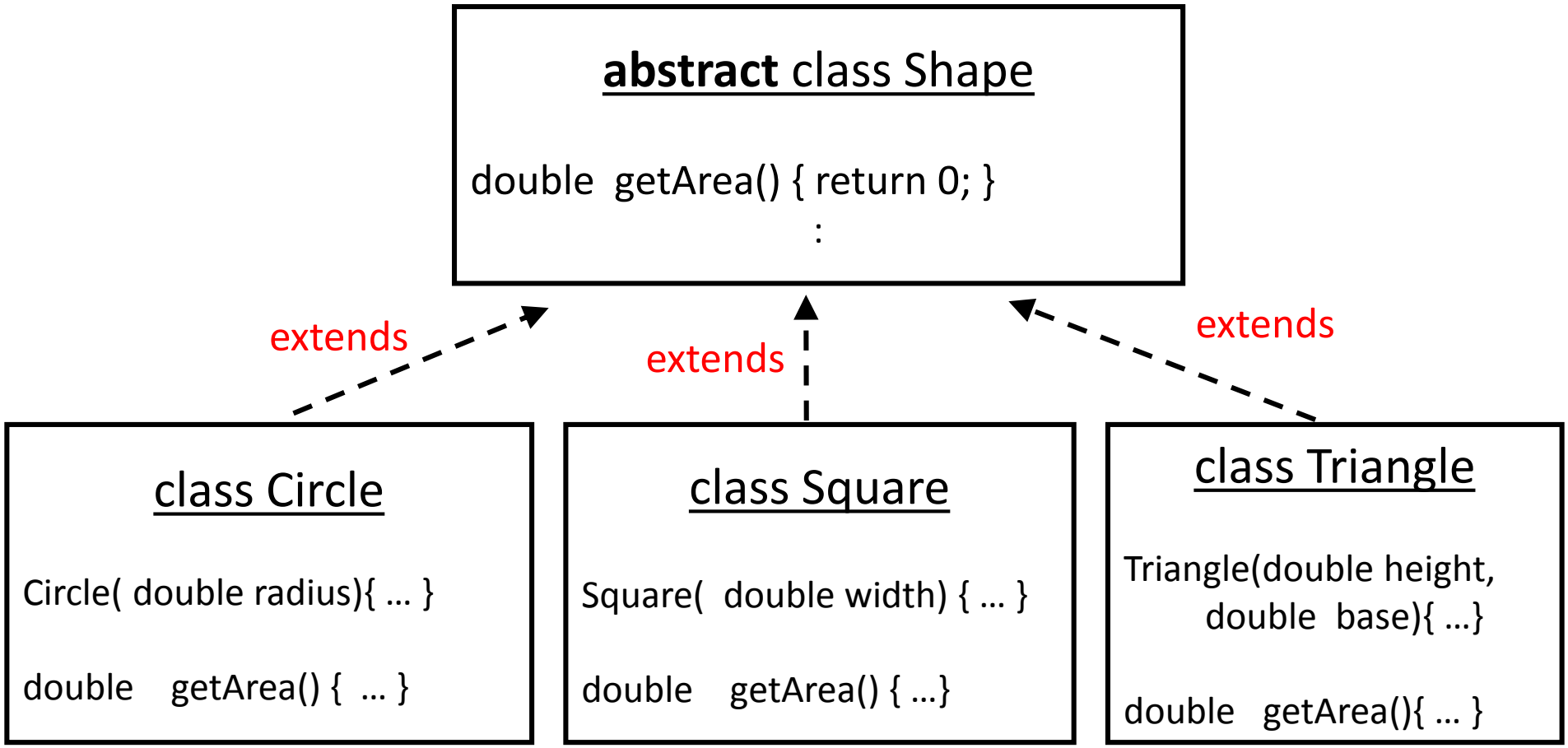

abstract class Circular
double radius
double getRadius() { return radius; }
void setRadius(double r) { radius = r…}
abstract double getArea()
extends
extends
extends
class Circle
Circle(double radius){ … }
double getArea() { … }
class Sphere
Sphere(double radius) { … }
double getArea() { …. }
class Cylinder
double length
Cylinder (double radius,
double len){ …}
double getArea(){ … }

```
abstract class Circular {

        double radius;                  //  field

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

        double getRadius(){             //  implemented methods
                return radius;
        }

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

        abstract double  getArea();    //  abstract method
}
```

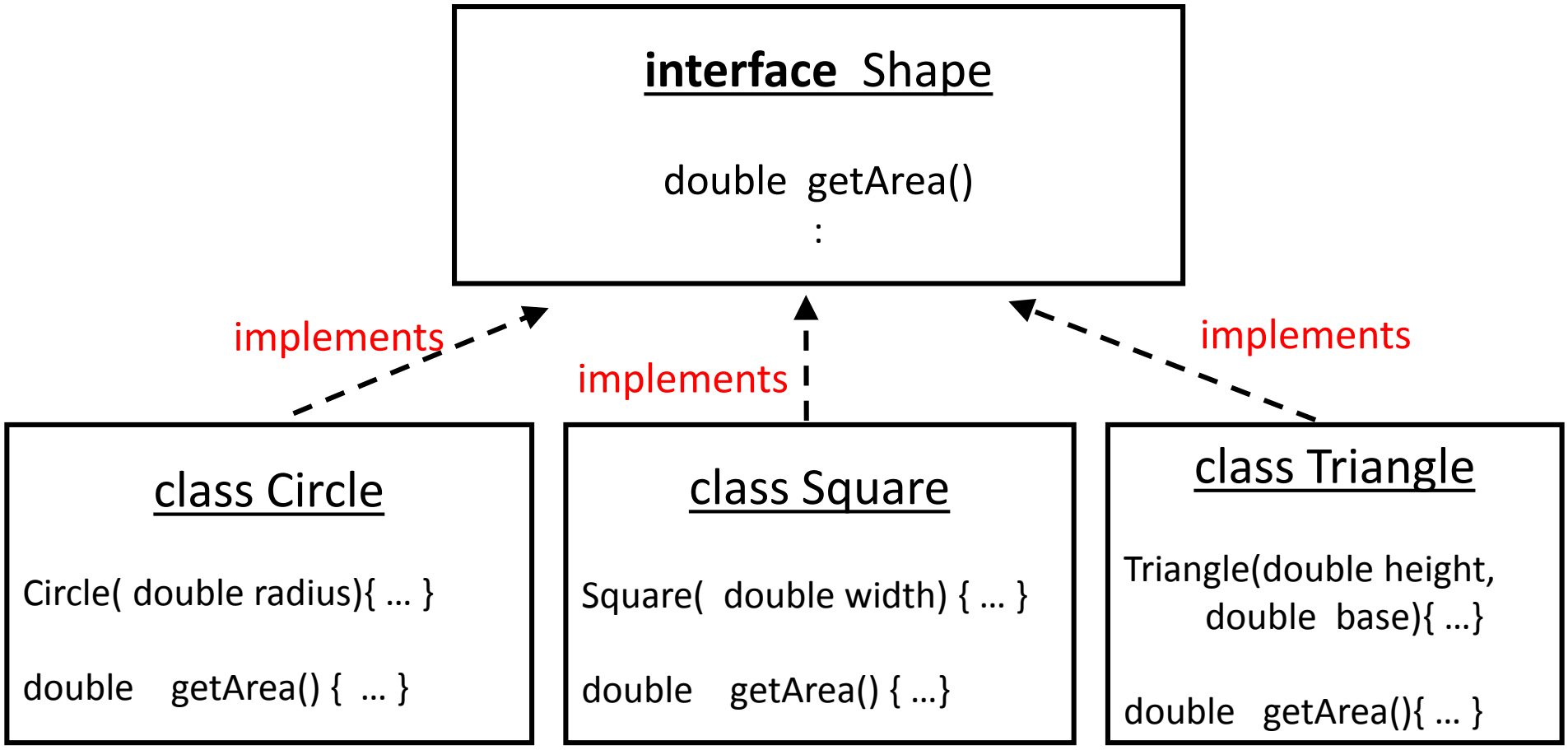
```
class Circle extends Circular{

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

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

```
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;
        }
}
```

MY BAD Example: Assignment 4



It should have been:



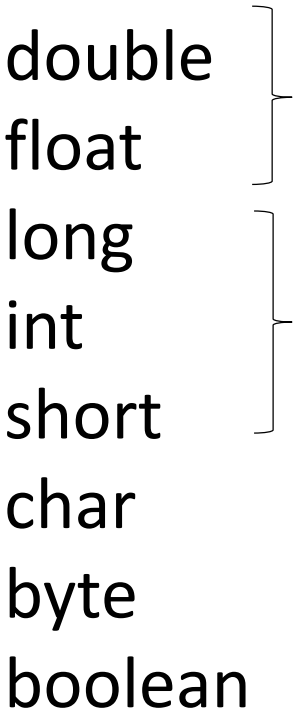
COMP 250

Lecture 31

abstract classes, type conversion

Nov. 23, 2016

Primitive Type Conversion



In COMP 273, you will learn exactly how these number representations are related to each other.

But you should have some intuitive ideas….

https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

Primitive Type Conversion

wider

Wider usually (but not always) means more bytes.

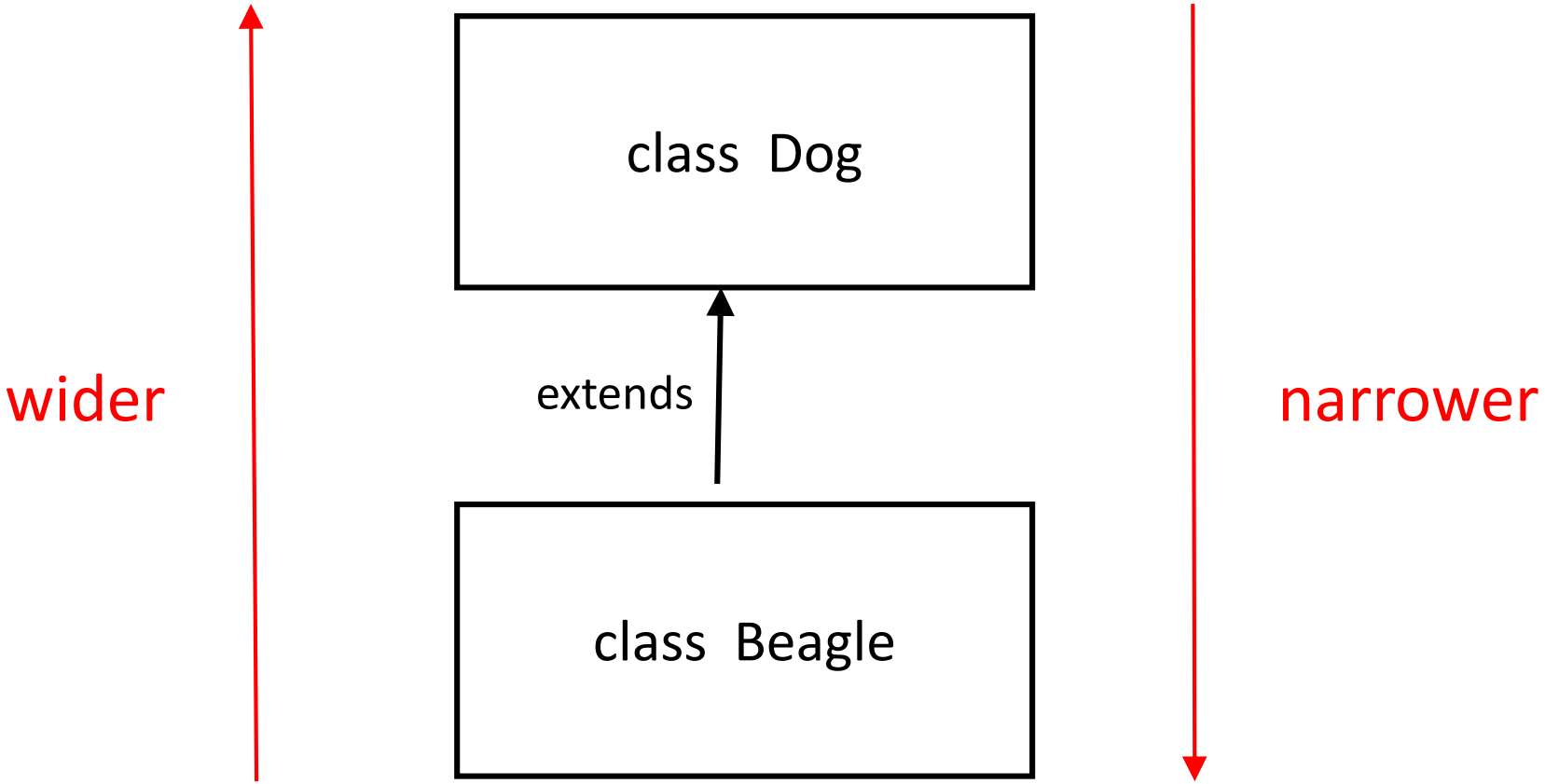
	number of bytes
double	8
float	4
long	8
int	4
short	2
char	2
byte	1
boolean	1

narrower

Examples

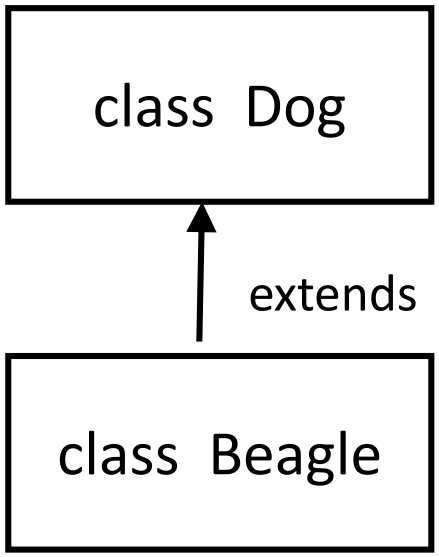
```
int     i = 3;
double  d = 4.2;
        d = i;                  // widening

        d  =  5.3 * i;          // widening     (by "promotion")
        i  =  (int)   d;        // narrowing    (by casting)
float   f  =  (float) d;        // narrowing    (by casting)

char   c = 'g';
int    index = c;               // widening
c = (char) index;               // narrowing
```

For narrowing conversions, you get a compiler error if you don't cast.



Heads up! Although the subclass is narrower, it has more bytes than the superclass.

```
Dog     myDog   =   new Beagle();

                // upcast, widening
```



This is similar to:

```
double  myDouble   =   3;     //    from int to double.
```

```
Dog       myDog = new Beagle();       // Upcasting.


Poodle    myPoodle  =   myDog;        // Compiler error.

          // implicit downcast Dog to Poodle not allowed.


myDog.show()                  //  Compiler error.

                          //   Poodle has show() method,
                          //   but Dog  does not.
```

```
Dog       myDog = new Beagle();       // Upcasting.


Poodle   myPoodle  =  (Poodle)  myDog;

          //   allowed by compiler

myPoodle.show()        //   allowed by compiler
                       //   Runtime error:   Dog object
                       //   does not have show()  method

((Poodle)  myDog).show()

   //   allowed by compiler,  but will generate runtime
   //   error if  actual object doesn't have a show method.
```

How to avoid such runtime errors?

```
if   (myDog  instanceOf  Poodle){

      ( (Poodle) myDog ).show();
}


if   (myPoodle  instanceOf  Poodle){

       myPoodle.show();
}
```