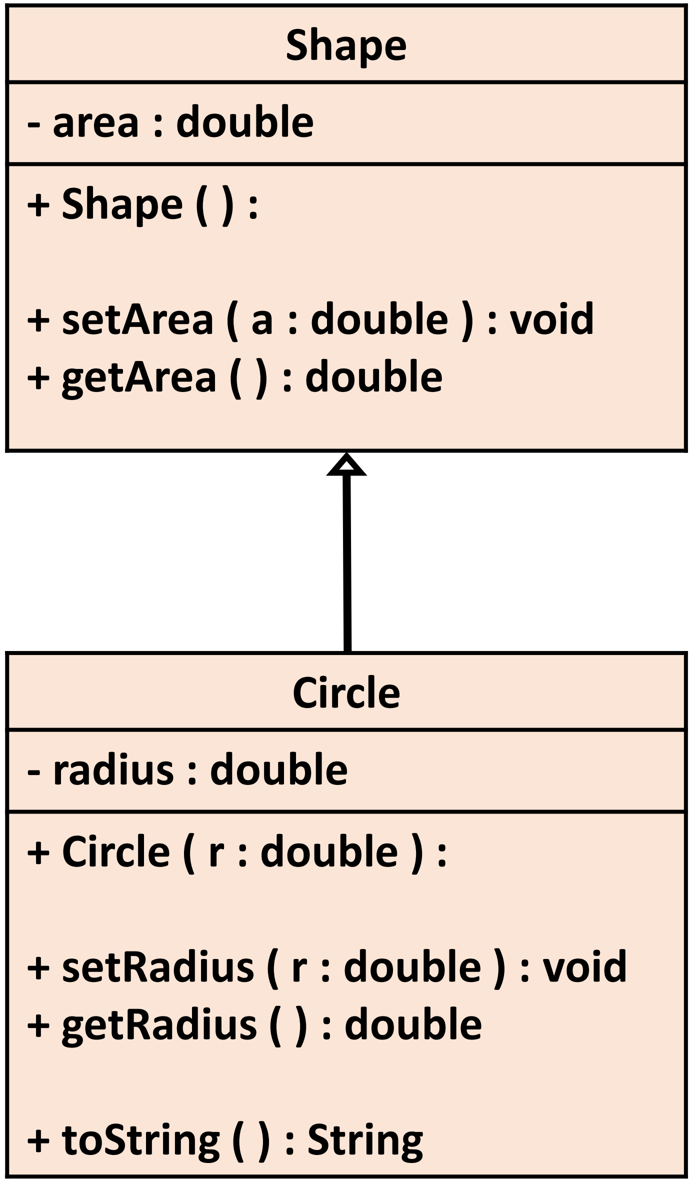
Computer Science II
Handout 7

Inheritance – recap

- Another way for Objects to work together
  - Described as a "*is a*" relationship
  - Subclasses *inherit* non-private members from the superclasses they extend

- Review: which of these are legal?

```
Shape s = new Shape();
Circle c = new Circle(12);

c.setArea(1.0);
s.getArea();
c.area = 10.0;
System.out.println(s);
System.out.println(c);
```



Shape

- area : double

+ Shape ( ) :

+ setArea (a : double) : void
+ getArea ( ) : double

Circle

- radius : double

+ Circle (r : double) :

+ setRadius (r : double) : void
+ getRadius ( ) : double

+ toString ( ) : String

```java
/*
 * Example of aggregation
 */
public class Circle {
        private Shape s;
        private double radius;

        public Circle(double r) {
            s = new Shape();
            setRadius(r);
        }

        public void setRadius(double r) {
            radius = r;
            setArea(Math.PI*radius*radius);
        }
        public double getRadius() {
            return radius;
        }
        public String toString(double r) {
            return “(” + radius + “)”;
        }

        public double getArea() {
            return s.getArea();
        }
        public void setArea(double a) {
            s.setArea(a);
        }
}
```

```java
/*
 * Example of inheritance
 */
public class Circle extends Shape {
        private double radius;


        public Circle(double r) {
            setRadius(r);
        }


        public void setRadius(double r) {
            radius = r;
            setArea(Math.PI*radius*radius);
        }
        public double getRadius() {
            return radius;
        }
        public String toString(double r) {
            return “(” + radius + “)”;
        }

}
```

Inheritance – recap

- Remember the differences between aggregation and inheritance:

Aggregation	Inheritance
Uses “*has a*” relationships	Uses “*is a*” relationships
Uses an Object to access members	Inherits members from a class
Accesses Object members “like a friend”	Accesses class members “like a parent”

Inheritance – overriding

- We have already seen “overridden” constructors

```
public Shape() { }
public Circle() { }
```

- In class **`Circle`**, the latter constructor is called instead of the former

- When using inheritance, we can also override other methods
  - Just like *overloading*, this relies on method signatures

- *Overriding* is implementing a method in the *subclass* that has the same method signature as a method in the *superclass*

Inheritance – overriding

```java
public class Circle extends Shape {
    private double radius;

    public Circle(double r) {
        radius = r;
    }

    // Overrides superclass method!
    public double getArea() {
        return Math.PI*radius*radius;
    }

    // Other methods ...
}
```

Inheritance – overriding

- When calling a method that has been overridden, the method in the *subclass* is called

```
public class Demo {
    public static void main(String[] args) {
        Shape s = new Shape();
        Circle c = new Circle();

        s.getArea(); // Calls method in class Shape
        c.getArea(); // Calls method in class Circle
    }
}
```

Inheritance – super keyword

- When overriding methods, the **super** keyword allows direct access to matching method in the superclass

```
public double getArea() {
    return super.getArea();
}
```

- When implementing a constructor, the super keyword allows access to a superclass constructor

```
public Circle() {
    super();
}
```

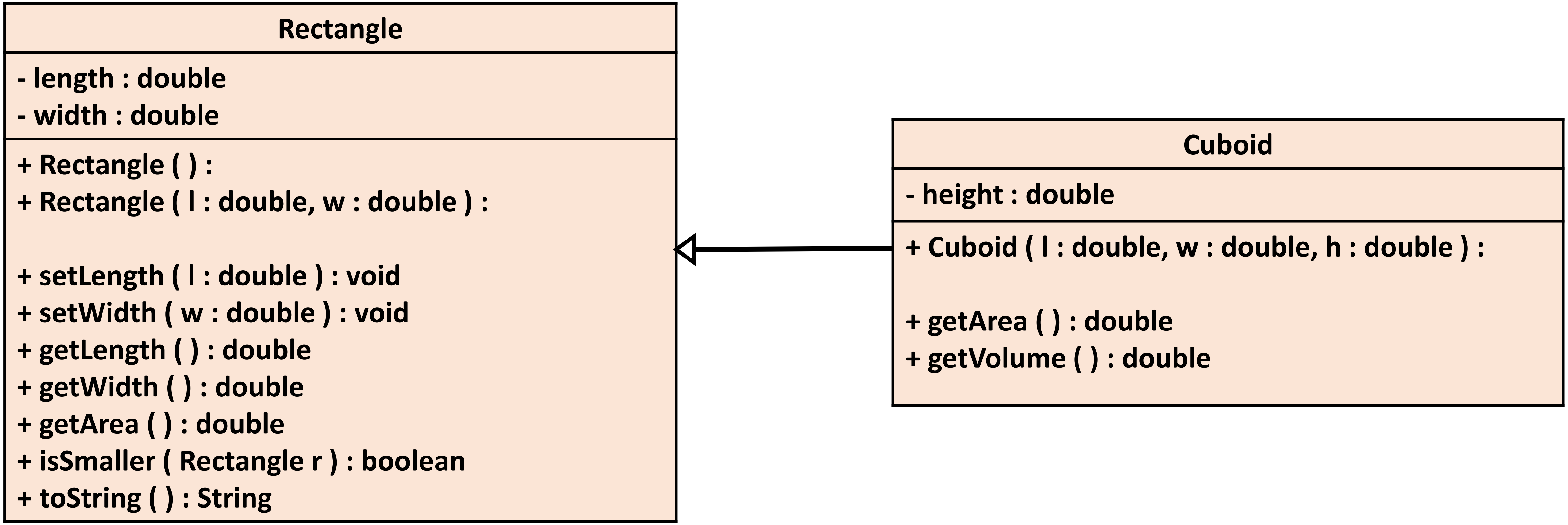
Inheritance – overriding example

- Use the class Rectangle as a superclass and implement a subclass Cuboid that has
  - An instance variable for height
  - A constructor to set the cuboid’s length, width, and height
  - Get methods for calculating the surface area and volume of the cuboid

```java
public class Rectangle {

    private double width;
    private double length;

    public Rectangle() { }

    public Rectangle(double l, double w)  {
        this.width = w;
        this.length = l;
    }

    public void setWidth(double w) {
        width = w;
    }

    public void setLength(double l) {
        length = l;
    }

    public double getLength() {
        return length;
    }

    public double getWidth() {
        return width;
    }

    public double getArea() {
        return length * width;
    }

    public boolean isSmaller(Rectangle rect) {
        return (getArea() < rect.getArea());
    }

    public String toString() {
        String ts = "[ ";
        ts += length + " x ";
        ts += width + " ]";
        return ts;
    }
}
```

Inheritance – overriding example

UML diagram of class Cuboid extending Rectangle (Cuboid *is a* Rectangle)



Rectangle

- length : double
- width : double

+ Rectangle ( ) :
+ Rectangle (l : double, w : double) :

+ setLength (l : double) : void
+ setWidth (w : double) : void
+ getLength ( ) : double
+ getWidth ( ) : double
+ getArea ( ) : double
+ isSmaller (Rectangle r) : boolean
+ toString ( ) : String

Cuboid

- height : double

+ Cuboid (l : double, w : double, h : double) :

+ getArea ( ) : double
+ getVolume ( ) : double

```java
public class Cuboid extends Rectangle {
        private double height;

        public Cuboid(double l, double w, double h) {



        }

        public double getHeight() {
                return height;
        }

        public double getArea() {
                double area



                return area;
        }

        public double getVolume() {
                return super.getArea() * height;  // Calls superclass method
        }
}
```

```java
import java.util.Scanner;
public class CuboidDemo {
    public static void main(String[] args) {
        Scanner kb = new Scanner(System.in);
        System.out.print(“Enter length, width, height: ”);
        double r = kb.nextDouble();

        Cuboid c = new Cuboid(kb.nextDouble(), kb.nextDouble(), kb.nextDouble());

        System.out.println(“Length: ” + c.getLength());
        System.out.println(“Width: ” + c.getWidth());
        System.out.println(“Height: ” + c.getHeight());

        System.out.println(“Area: ” + c.getArea());
        System.out.println(“Volume: ” + c.getVolume());



    }
}
```

```
> Enter length, width, height: 2 3 5
  Length: 2.0
  Width: 3.0
  Height: 5.0
  Area: 62.0
  Volume: 310.0
```

Inheritance – overriding example

- Overriding methods lets us customize functionality in more specialized subclasses
  - Using the `super` keyword still lets us access the superclass method that was overridden

- The super keyword also gives access to superclass constructors

Inheritance – constructors

- Constructors have specific behaviour when using inheritance
  - What is the output when a new Object of class Circle is created?

```
public class Shape {
    public Shape() {
        System.out.println(“Superclass constructor called”);
    }
}

public class Circle extends Shape {
    public Circle() {
        System.out.println(“Subclass constructor called”);
    }
}
```

Inheritance – constructors

```
public class Demo {
    public static void main(String[] args) {
        Circle c = new Circle();
    }
}
```

```
>
```

Inheritance – constructors

- The no-arg superclass constructor is always called *by default* before the rest of any subclass constructor executes

- If the no-arg superclass constructor does not exist, then one of two things can happen:
  1. You can specify another superclass constructor call *on the first line* of the subclass constructor, or
  2. There will be an error at run-time

Inheritance – constructors

- We can mimic the existing (default) behaviour of Java:

```
public class Circle extends Shape {
    public Circle() {
        super();  // Requires constructor Shape()
        System.out.println(“Subclass constructor called”);
    }
}
```

Inheritance – constructors

- We can also force Java to call a different superclass constructor, indicated by the parameter list

```
public class Circle extends Shape {
    public Circle() {
        super(2.5);  // Requires constructor Shape(double)
        System.out.println(“Subclass constructor called”);
    }
}
```

Inheritance – constructors

- In our class Cuboid, we could have used the super keyword to call the appropriate superclass constructor directly

```
public class Cuboid extends Rectangle {
     private double height;

     public Cuboid(double l, double w, double h) {
          super(l, w);
          height = h;
     }

     // etc ...
```

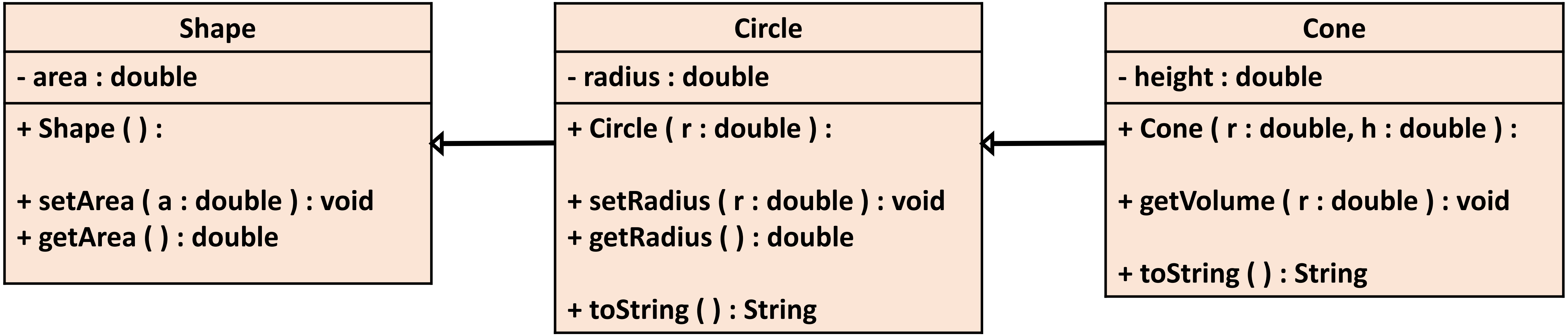
Inheritance – chaining together subclasses

- It is acceptable (and sometimes preferable) to have multiple levels of inheritance

- Return to the class **`Shape`**, that stores only an area and has appropriate get/set methods
- We extended this to class **`Circle`**
- Extend this again to class **`Cone`**

Inheritance – chaining together subclasses

- Use our existing code to implement this UML diagram
  - Conic volume = ${}^{h\pi r^2}/_{3}$



Shape

- area : double

+ Shape ( ) :

+ setArea (a : double) : void
+ getArea ( ) : double

Circle

- radius : double

+ Circle (r : double) :

+ setRadius (r : double) : void
+ getRadius ( ) : double

+ toString ( ) : String

Cone

- height : double

+ Cone (r : double, h : double) :

+ getVolume (r : double) : void

+ toString ( ) : String

```
public class Cone

        public Cone(double r, double h) {



        }

        public double getVolume() {



        }


        public String toString() {
                String ts = “Base radius: ” + getRadius() + “\n”;
                ts += “Volume: ” + getVolume();
                return ts;
        }
}
```

```java
import java.util.Scanner;
public class ConeDemo {
    public static void main(String[] args) {
        Scanner kb = new Scanner(System.in);
        System.out.print(“Enter radius and height: ”);

        Cone c = new Cone(kb.nextDouble(), kb.nextDouble());

        System.out.println(c);


    }
}
```

```
> Enter radius and height: 3 10
  Base radius: 3.0
  Volume: 94.2477796076938
```

Abstract classes

- Abstract classes allow for giving an *outline* for a class that will never be instantiated

- For example: to track three major subject in a school, we could create a Subject class that defines the similar characteristics
    - English, Math, and Science could all be classes that inherit from Subject
    - Yet, Subject itself never needs to be instantiated; it doesn’t make sense

- Subject can therefore be written as an *abstract class*

Abstract classes

- Abstract classes are written to be extended
  - Never instantiated

```
public abstract class Subject {
```

- Abstract classes are a mixture of regular methods and *abstract methods*
  - These are methods that *must* be overridden in any subclass
  - They have no method bodies, only headers that specify their parameters and return type

```
public abstract int getEnrolmentCount();
```

Abstract classes

- Methods can be defined “as usual” inside an abstract class
- Abstract methods can also be defined, which have no bodies
- Constructors can be defined so they can be used by subclasses

- Inheritance for the subclass then works normally, except abstract methods *must* be overridden

Abstract classes – example

- Create a Student abstract class that holds general student information: name, ID, and startYear
  - Include a constructor that takes in all three values as parameters and initializes the instance variables
  - Include a toString method that returns the name and ID
  - Include an abstract method called getRemainingHours that has no parameters and returns an int

```java
public abstract class Student {
      private String name;
      private int id;
      private int year;

      public Student(String n, int i, int y) {




      }

      public String toString() {

      }




}
```

Abstract classes – example

- Follow this with a CSStudent class that inherits from Student
    - Include a total number of hours needed to graduate from each of math, computer science, and general courses
    - Include instance variables to hold the current hours in each of the three areas for this Student
    - Include set/get methods for all instance variables
    - Include a constructor that initializes all instance variable values
    - Override getRemainingHours that returns the total credit hours still needed to graduate

```
public class CSStudent extends Student {
    private                 MATH_HOURS = 20;
    private                 CS_HOURS = 40;
    private                 GEN_HOURS = 60;

    private int mathHours;
    private int csHours;
    private int genHours;

    public CSStudent(String n, int id, int y)  {

    }

    public int getRemainingHours() {




    }

    public void setMathHours(int h) {
        mathHours = h;
    }

    public void setCSHours(int h) {
        csHours = h;
    }

    public void setGenHours(int h) {
        genHours = h;
    }
```

```

    public int getMathHours() {
        return mathHours;
    }

    public int getCSHours() {
        return csHours;
    }

    public int getGenHours() {
        return genHours;
    }
}
```

```java
public class StudentDemo {
    public static void main(String[] args) {
        CSStudent c = new CSStudent(“Jeremy”, 1001, 2015);

        c.setMathHours(12);
        c.setCSHours(20);
        c.setGenHours(40);
        System.out.println(c + “ has ” + c.getRemainingHours() + “h left”);


    }
}
```

```
>
```