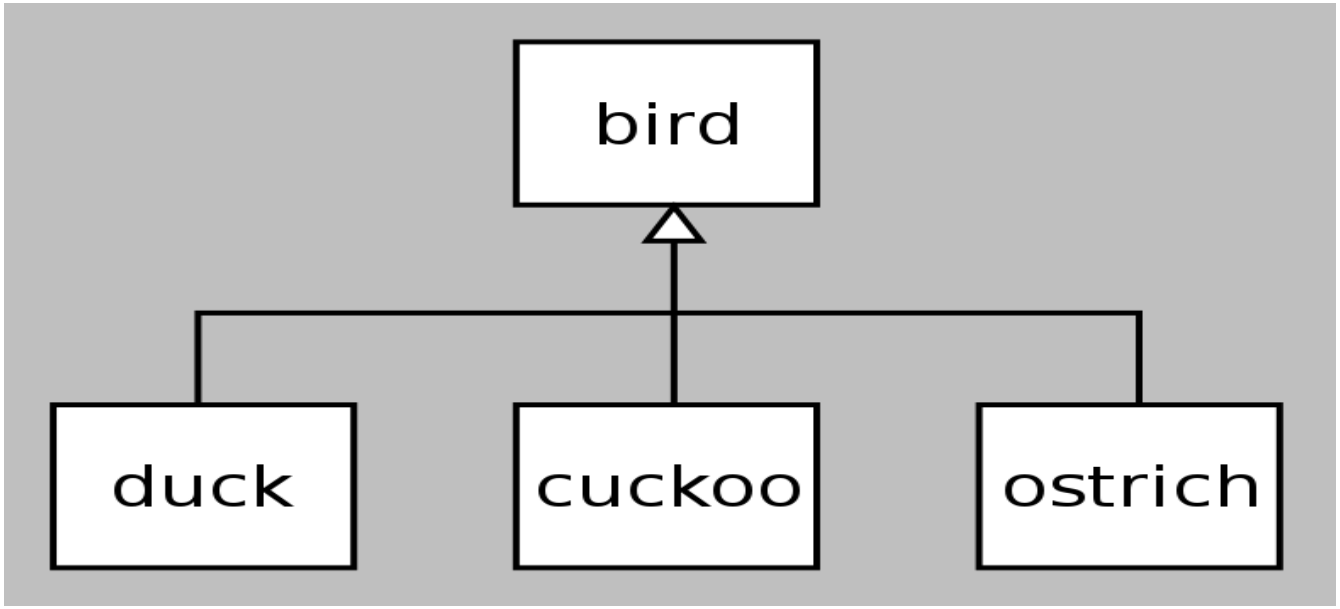
Inheritance and Polymorphism



Computer Science OOD
Boston University

Christine Papadakis-Kanaris

Inheritance:

a vehicle hierarchy

Inheritance allows us to *derive* new classes from existing classes.

sedan

van

jeep

Inheritance:

a vehicle hierarchy

contains all the attributes and behaviors common to all vehicles

vehicle

sedan

van

jeep

Inheritance:

a vehicle hierarchy

Derive new classes each to represent a specific type of vehicles!

vehicle

sedan

van

jeep

Inheritance:

ehicle hierarchy

The derived classes are specialized classes which inherit the properties and methods of the class they are derived from.

vehicle

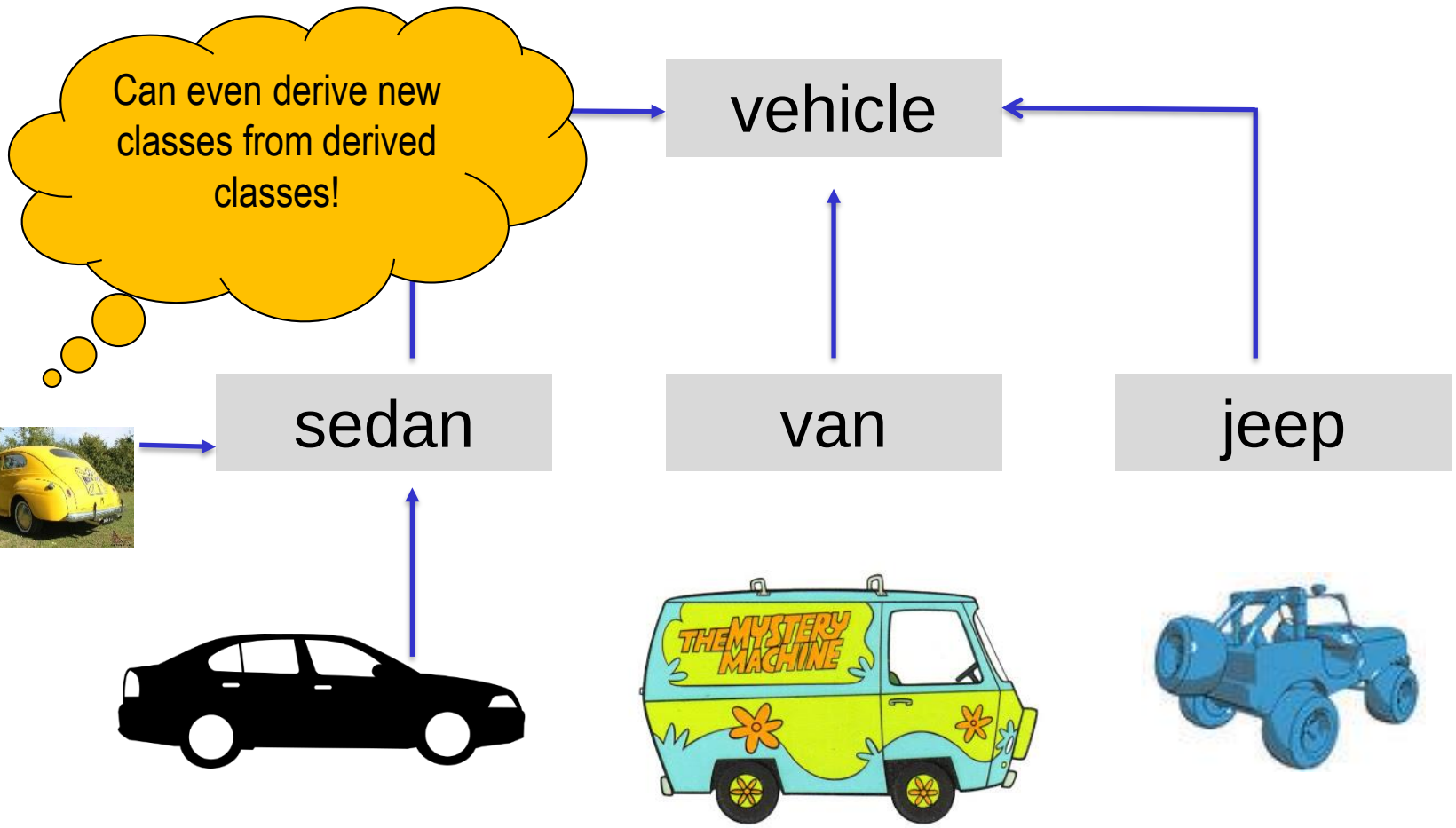
sedan

van

jeep

Inheritance:

a vehicle hierarchy



Inheritance:

a vehicle hierarchy

Inheritance represents a parent .. child heirarchy

vehicle

sedan

van

jeep

Inheritance:

a vehicle hierarchy

Single Inheritance

A new class is derived from only one parent class.

Java only supports single Inheritance!

vehicle

sedan

van

jeep

Inheritance:

a vehicle hierarchy

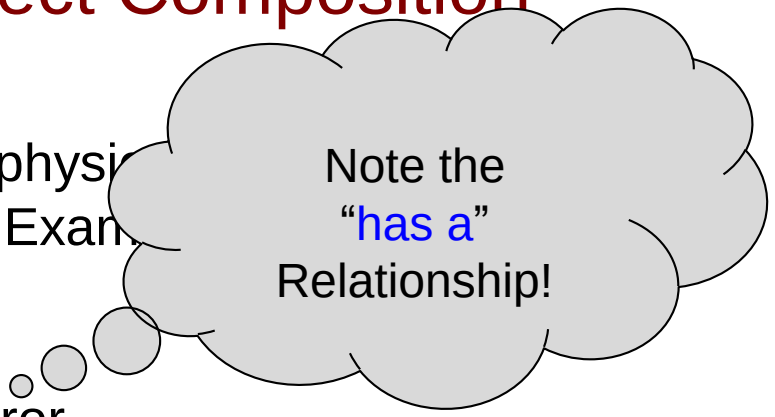
Multiple Inheritance

A new class is derived from more than one parent class.

Java does not support multiple *class* Inheritance!

Inheritance vs. Object Composition

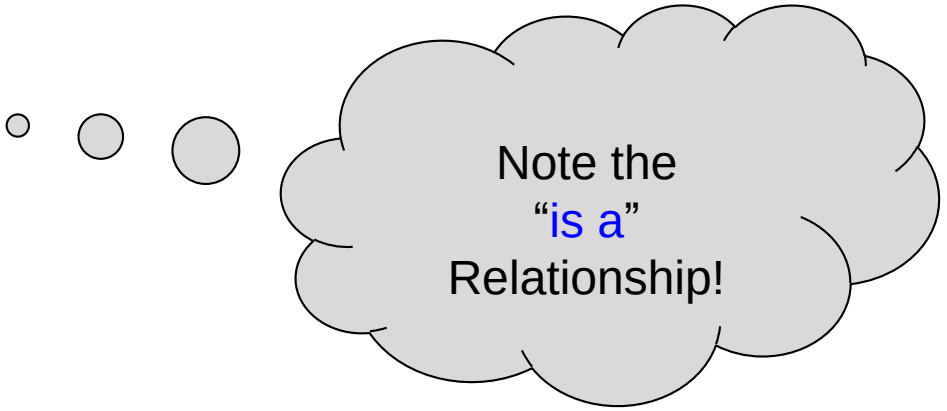
- Object composition refers to the physi… make-up or *compose* the object. Exam…
  - a vehicle has tires
  - a vehicle has a rear view mirror
  - a vehicle has a break pedal, etc.



Inheritance vs. Object Composition

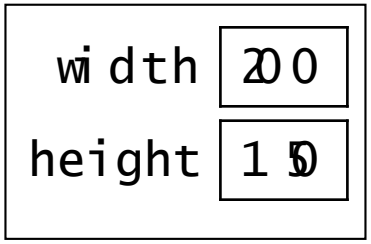
- Object composition refers to the physical entities that make-up or *compose* the object. Example:
  - a vehicle has tires
  - a vehicle has a rear view mirror
  - a vehicle has a break pedal, etc.

- Inheritance represents a hierarchical relationship. Example:
  - a sedan is a vehicle
  - a van is a vehicle
  - a jeep is a vehicle



Recall: A Class for `Rectangle` Objects

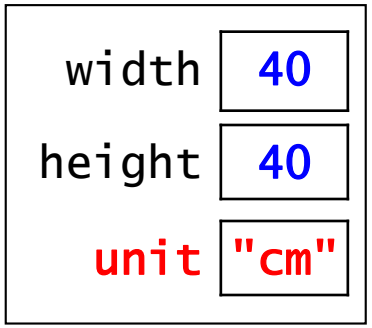
- Every `Rectangle` object has two fields:
  - `width`
  - `height`



- It also has methods inside it:
  - `grow()`
  - `area()`
  - `toString()`
  - etc.

Squares *are* Special Rectangles!

- A square also has a width and a height.
  - but the two values must be the same
- Assume that we also want `Square` objects to have a field for the unit of measurement.



- `Square` objects should mostly behave like `Rectangle` objects:

```
Rectangle r = new Rectangle(20, 30);
int area1 = r.area();

Square sq = new Square(40, "cm");
int area2 = sq.area();    // same computation
```

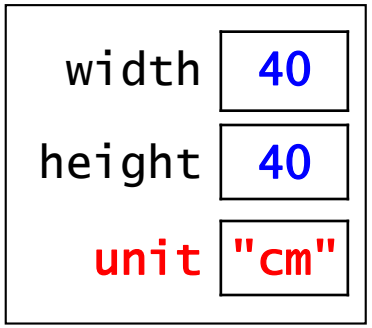
- But there may be differences as well:

```
System.out.println(r);
```
⇨ output:
```
20 x 30
```

```
System.out.println(sq);
```
⇨ output:
```
square with 40-cm sides
```

Squares *are* Special Rectangles!

- A square also has a width and a height.
  - but the two values must be the same



- Assume that we also want `Square` objects to have a field for the unit of measurement.

- `Square` objects should mostly behave like `Rectangle` objects:

```
Rectangle r = new Rectangle(20, 30);
int area1 = r.area();

Square sq = new Square(40, "cm");
int area2 = sq.area();
```

- But there may be differences as well:

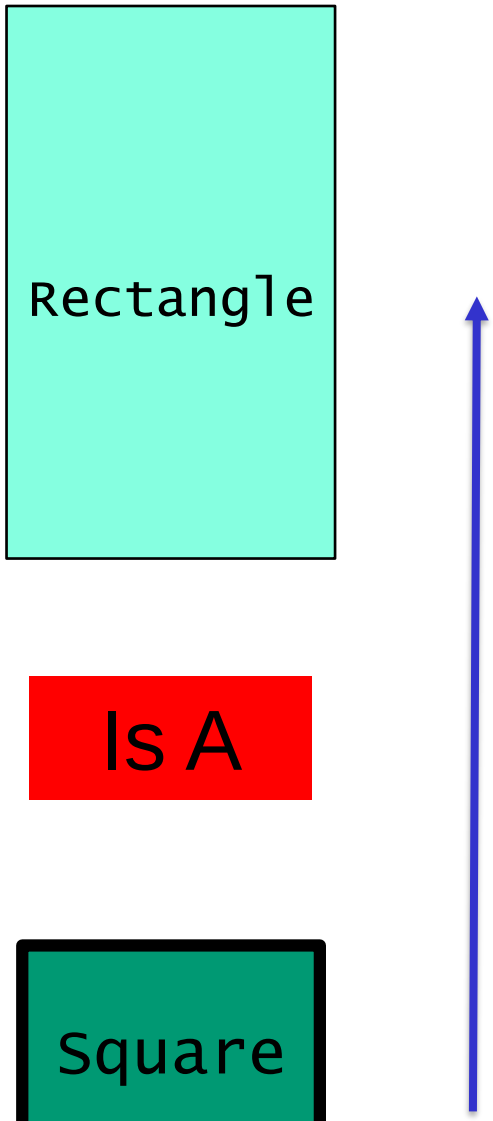
```
System.out.println(r);
```
⇨ output:
20 x 30

```
System.out.println(sq);
```
⇨ output:
square with 40-cm sides

Using Inheritance

```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        setWidth(w);
        setHeight(h);
    }
    ...  // other methods
    public int area() {
        return width * height;
    }
}
```

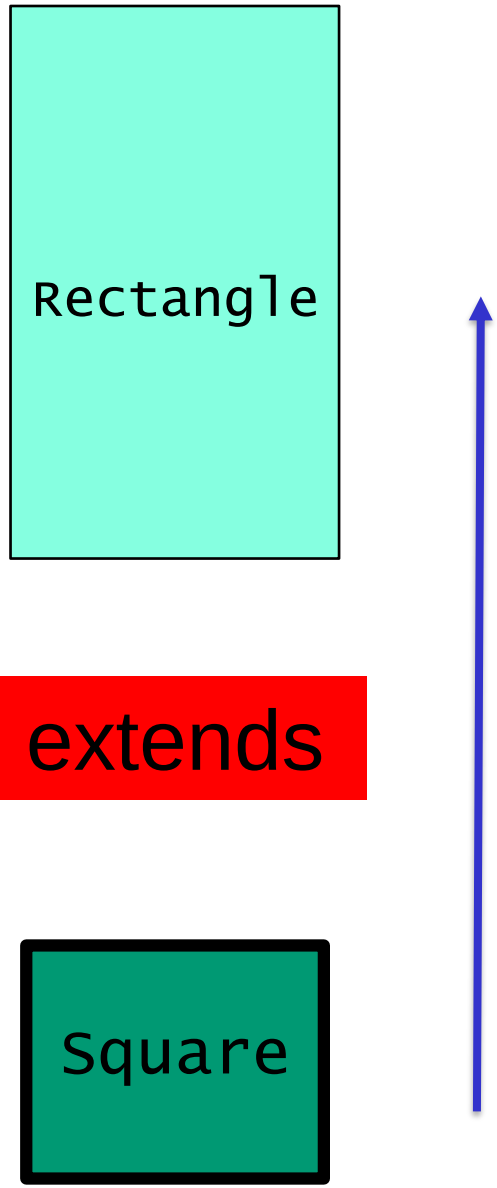
```
public class Square {
    int width, height;
    String unit;

    public Square(int side, String unit) {
        width = height = side;
        this.unit = unit;
    }
    public int area() {
        return width * height;
    }
    ...
```



Using Inheritance

```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        setWidth(w);
        setHeight(h);
    }
    ...  // other methods
    public int area() {
        return width * height;
    }
}
```

```
public class Square extends Rectangle {
    String unit;

    public Square(int side, String unit) {
        // initialize data members


    }

    // inherits other methods
}
```

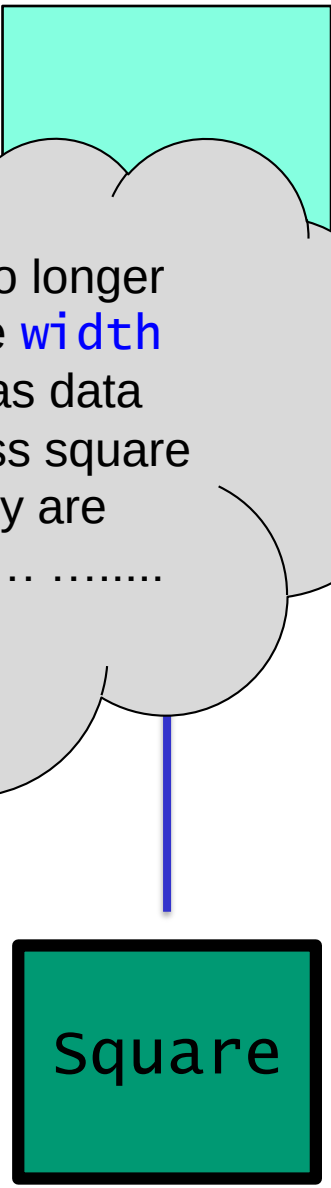


Using Inheritance

```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        setWidth(w);
        setHeight(h);
    }
    ...  // other methods
    public int area() {
        return width * height;
    }
}
```

```
public class Square extends Rectangle {
    String unit;

    public Square(int side, String unit) {
        // initialize data members


    }

    // inherits other methods
}
```

Note that we no longer have to include `width` and `height` as data members of class square because they are inherited from … ……..

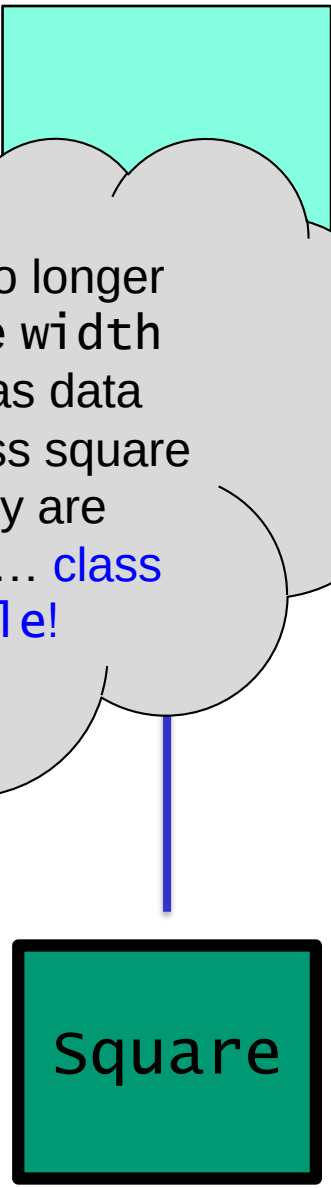


Using Inheritance

```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        setWidth(w);
        setHeight(h);
    }
    ...  // other methods
    public int area() {
        return width * height;
    }
}
```

```
public class Square extends Rectangle {
    String unit;

    public Square(int side, String unit) {
        // initialize data members


    }

    // inherits other methods
}
```

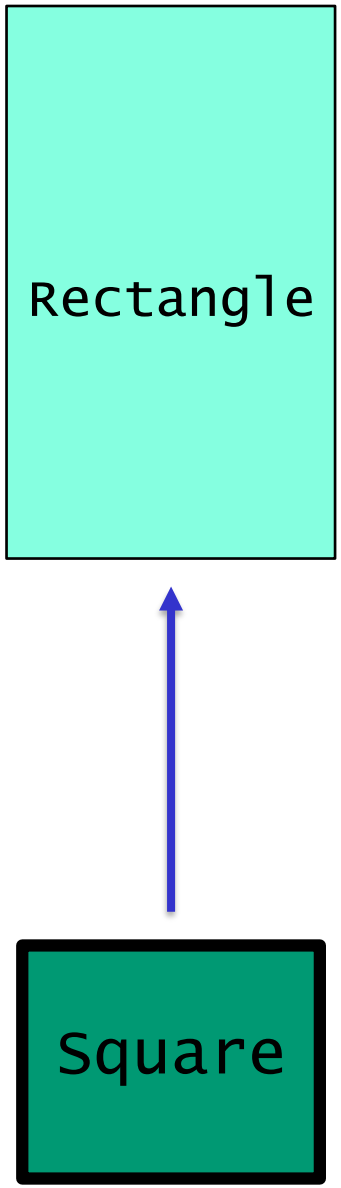
Note that we no longer have to include `width` and `height` as data members of class square because they are inherited from … class `Rectangle`!



Using Inheritance

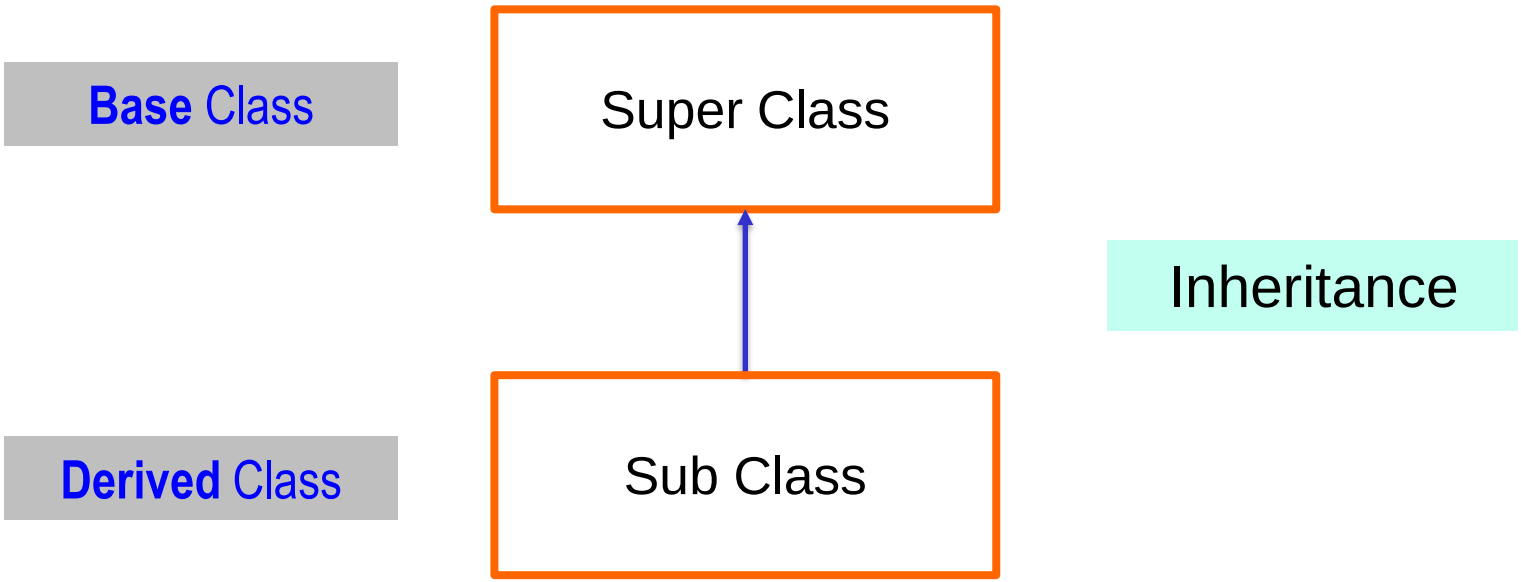
```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        setWidth(w);
        setHeight(h);
    }
    ...  // other methods
    public int area() {
        return width * height;
    }
}
```

```
public class Square extends Rectangle {
    String unit;

    public Square(int side, String unit) {
        // initialize data members


    }

    // inherits other methods
}
```



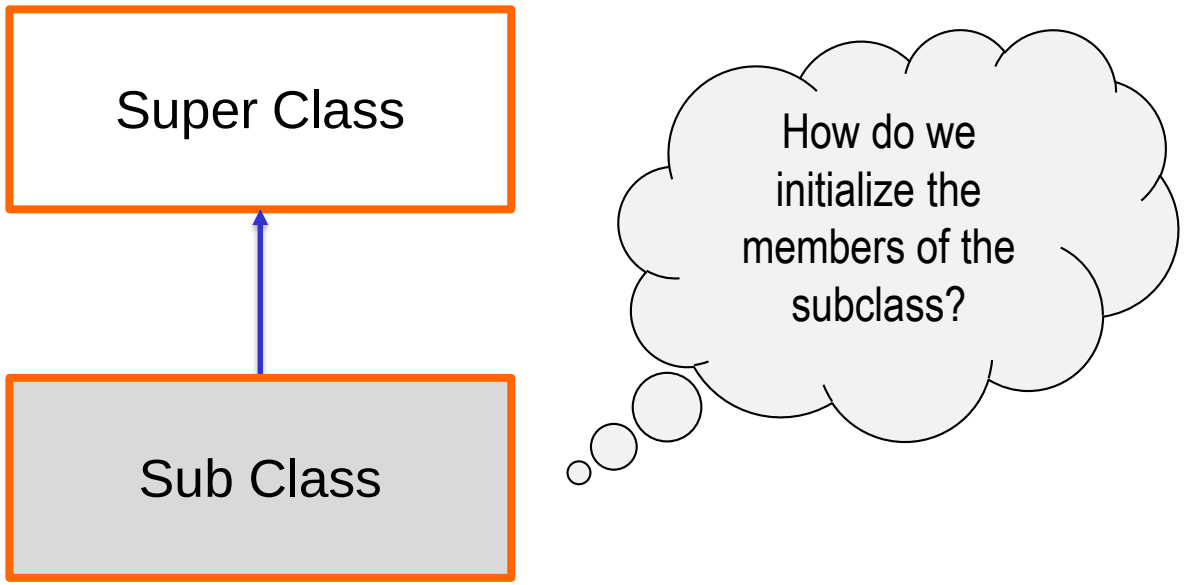
Using Inheritance

- `Square` *inherits* all of the fields and methods of `Rectangle`.
  - we don't need to redefine them!
- `Square` is a *subclass* of `Rectangle`.
- `Rectangle` is a *superclass* of `Square`.



Using Inheritance

- `Square` *inherits* all of the fields and methods of `Rectangle`.
  - we don't need to redefine them!
- `Square` is a *subclass* of `Rectangle`.
- `Rectangle` is a *superclass* of `Square`.



Encapsulation and Inheritance

- A subclass has direct access to the *public* fields and methods of a superclass.
  - it ***cannot*** access its *private* fields and methods

- Example: we can think of a `Square` object as follows:

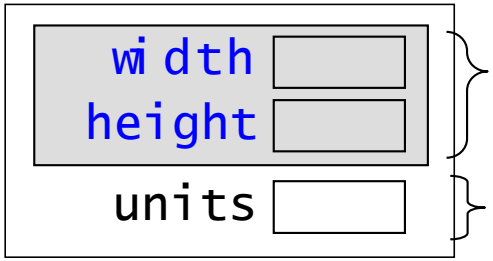
private fields inherited from `Rectangle`;
cannot be accessed directly by `Square` methods

field defined in `Square`;
it *can* be accessed directly by `Square` methods

Encapsulation and Inheritance

- A subclass has direct access to the *public* fields and methods of a superclass.
  - it ***cannot*** access its *private* fields and methods

- Example: we can think of a `Square` object as follows:



private fields inherited from `Rectangle`;
cannot be accessed directly by `Square` methods

field defined in `Square`;
it *can* be accessed directly by `Square` methods

Using Inheritance

```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        setWidth(w);
        setHeight(h);
    }
    ...  // other methods
    public int area() {
        return width * height;
    }
}
```

As `width` and `height` are *private* data member of the superclass Rectangle, we cannot directly access them here!

```
public class Square extends Rectangle {
    String unit;

    public Square(int side, String unit) {
        width = height = side;



    }

    // inherits other methods
}
```

Using Inheritance

```
public class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        setWidth(w);
        setHeight(h);
    }
    ...  // other methods
    public int area() {
        return width * height;
    }
}
```

As `width` and `height` are *private* data member of the superclass Rectangle, we cannot directly access them here!

```
public class Square extends Rectangle {
    String unit;

    public Square(int side, String unit) {
        this.width = this.height = side;


    }

    // inherits other methods
}
```

Encapsulation and Inheritance

- Change the modifier in the super class from private to protected.

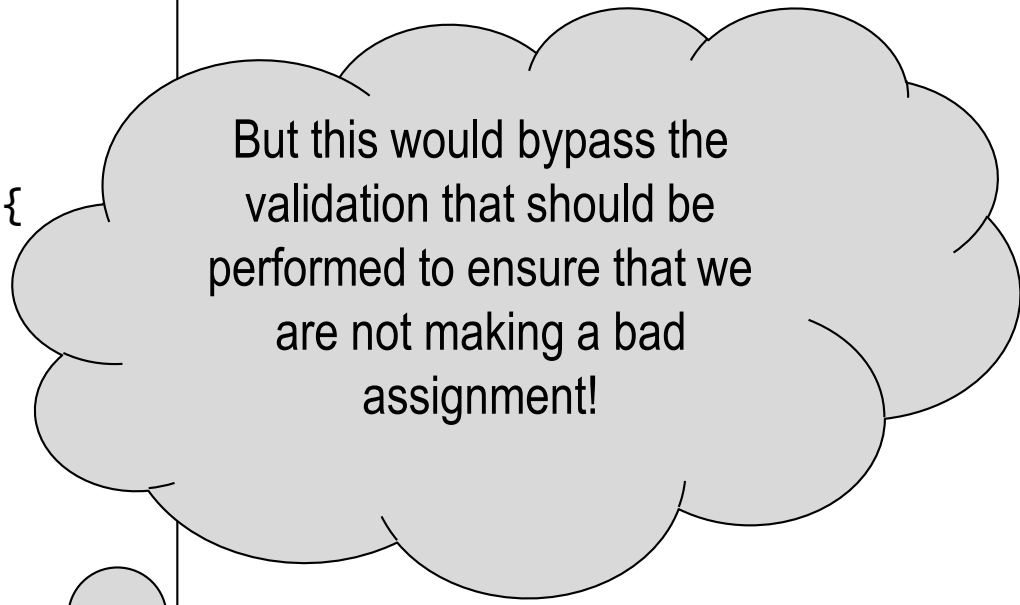
- The protected modifier allows the fields to remain private within the class they are defined in but allows them to be accessible to all subclasses.

- But for the most part it is more prudent to use the public accessor and mutator methods of the super class – even within the subclass.

Using Inheritance:

option #1

```
public class Rectangle {
    protected width;
    protected height;

    public Rectangle(int w, int h) {
        setWidth(w);
        setHeight(h);
    }
    ...  // other methods
    public int area() {
        return width * height;
    }
}
```

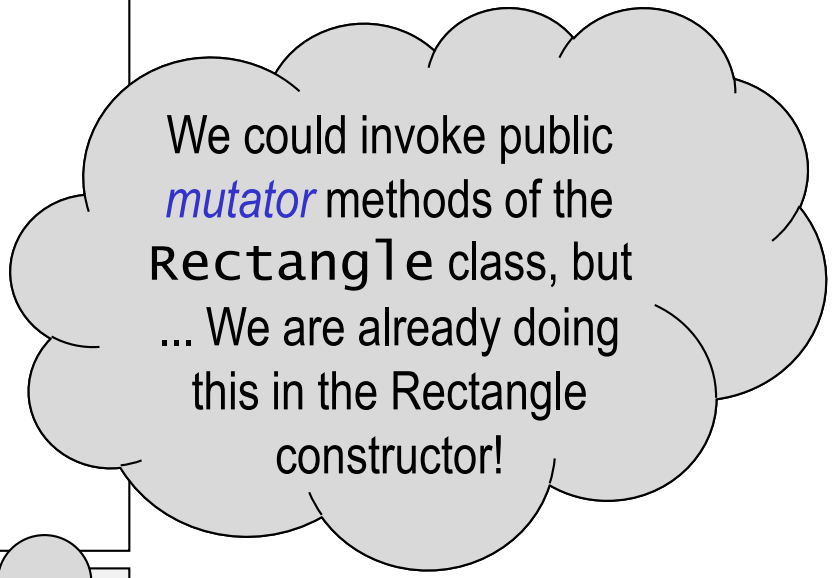



```
public class Square extends Rectangle {
    String unit;

    public Square(int side, String unit) {
        width = height = side;
        this.unit = unit;

    }

    // inherits other methods
}
```

Using Inheritance:

option #1

```
public class Rectangle {
    protected width;
    protected height;

    public Rectangle(int w, int h) {
        setWidth(w);
        setHeight(h);
    }
    ...  // other methods
    public int area() {
        return width * height;
    }
}
```




```
public class Square extends Rectangl
    String unit;

    public Square(int side, String unit) {
        setWidth(side);
        setHeight(side);
        this.unit = unit;

    }
    // inherits other methods
}
```

Using Inheritance:

option #3

```
public class Rectangle {
    private width;
    private height;

    public Rectangle(int w, int h) {
        setWidth(w);
        setHeight(h);
    }
    ...  // other methods
    public int area() {
        return width * height;
    }
}
```

As we are declaring members in the superclass, it is most appropriate to have the constructor of the superclass initialize them.

```
public class Square extends Rectangle {
    String unit;

    public Square(int side, String unit) {
        super(side, side);

        this.unit = unit;
    }

    // inherits other methods
}
```

Using Inheritance:

option #3

```
public class Rectangle {
    private width;
    private height;

    public Rectangle(int w, int h) {
        setWidth(w);
        setHeight(h);
    }
    ...  // other methods
    public int area() {
        return width * height;
    }
}
```

```
public class Square extends Rectangle {
    String unit;

    public Square(int side, String unit) {
        super(side);

        this.unit = unit;
    }

    // inherits other methods
}
```

As we are declaring members in the superclass, it is most appropriate to have the constructor of the superclass initialize them.

Using Inheritance:

option #3

```
public class Rectangle {
    private width;
    private height;

    public Rectangle(int w, int h) {
        setWidth(w);
        setHeight(h);
    }
    ...  // other methods
    public int area() {
        return width * height;
    }
}
```

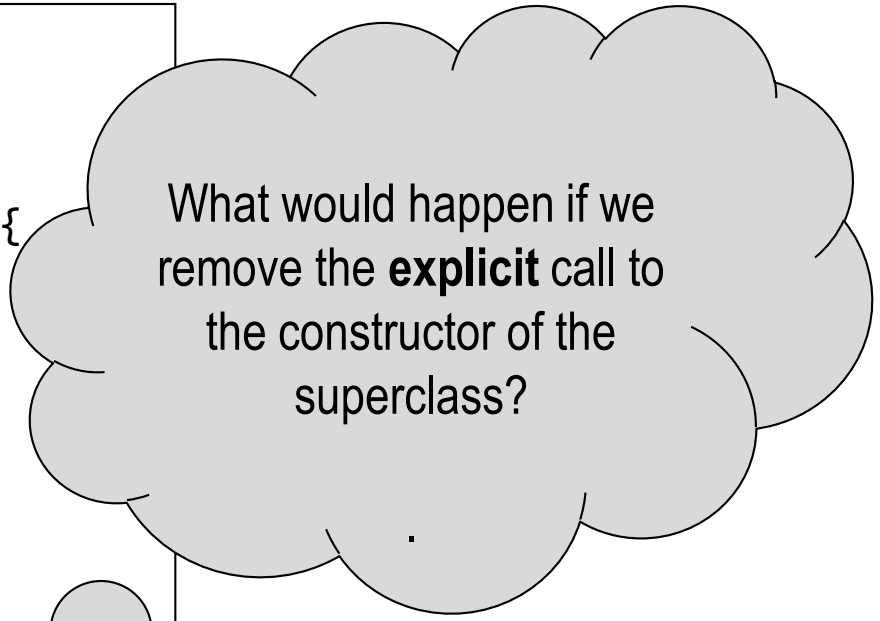
```
public class Square extends Rectangle {
    String unit;

    public Square(int side, String unit) {
        super(side);

        this.unit = unit;
    }

    // inherits other methods
}
```

Note that the call to the superclass constructor must be the very first statement in the body of the subclass constructor.

Using Inheritance:

option #3

```
public class Rectangle {
    private width;
    private height;

    public Rectangle(int w, int h) {
        setWidth(w);
        setHeight(h);
    }
    ...  // other methods
    public int area() {
        return width * height;
    }
}
```

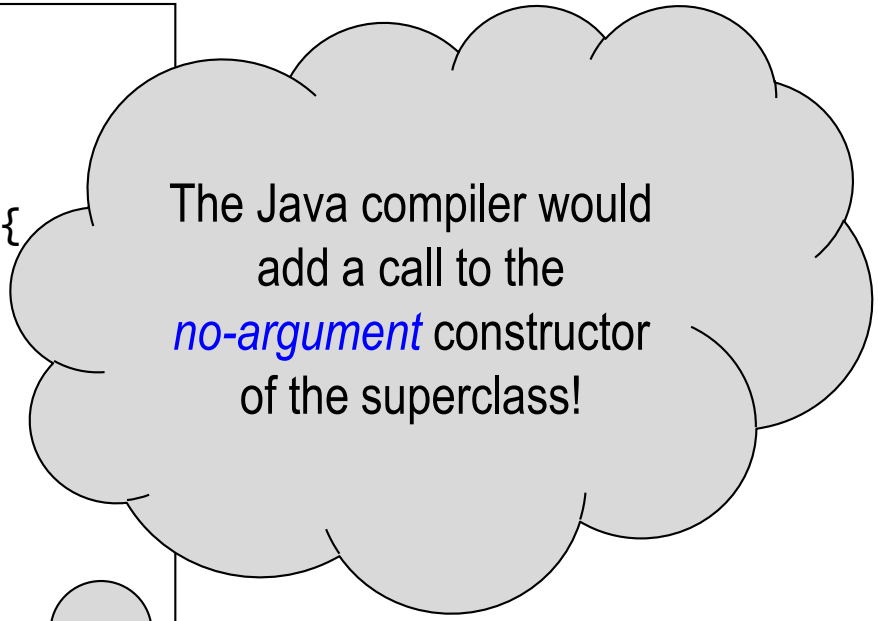



```
public class Square extends Rectangle {
    String unit;

    public Square(int side, String unit) {


        this.unit = unit;
    }

    // inherits other methods
}
```

Using Inheritance:

option #3

```
public class Rectangle {
    private width;
    private height;

    public Rectangle(int w, int h) {
        setWidth(w);
        setHeight(h);
    }
    ...  // other methods
    public int area() {
        return width * height;
    }
}
```




```
public class Square extends Rectangle {
    String unit;

    public Square(int side, String unit) {
        // no-arg constructor
        super();
        this.unit = unit;
    }

    // inherits other methods
}
```

Using Inheritance:

option #3

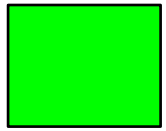
```
public class Rectangle {
    private width;
    private height;

    public Rectangle(int w, int h) {
        setWidth(w);
        setHeight(h);
    }
    ...  // other methods
    public int area() {
        return width * height;
    }
}
```

If a *no-arg constructor* has not been defined in the superclass Java would issue a compiler error.

```
public class Square extends Rectangle {
    String unit;

    public Square(int side, String unit) {
        // no-arg constructor
        super();
        this.unit = unit;
    }

    // inherits other methods
}
```

A note about Constructors





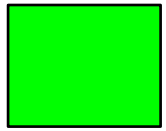
If a class does not define *any* constructors the Java compiler will create a no-argument constructor for our class. This constructor will be used if we create an object without passing any arguments.

Rectangle r = new Rectangle();

However once we define any constructor, then it is up to the class to define a no-argument constructor should we want to allow objects to be created with just default values.

Constructor Chaining





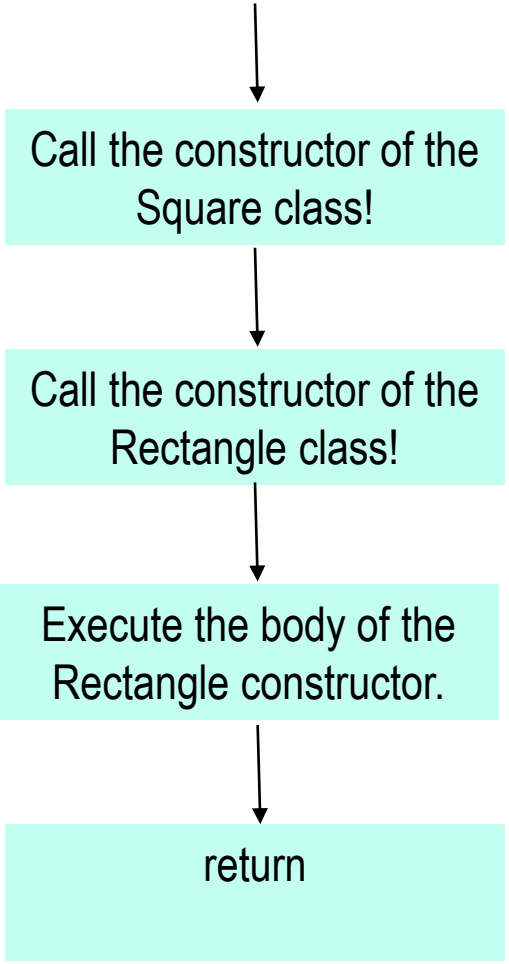
Unlike the methods of a class, constructors of a superclass are not inherited by the subclass.

They can only be invoked from the constructors of the subclass using the keyword super.

Constructing an instance of a class invokes the constructors of all the super classes along the inheritance chain.

Constructor Chaining

Square r = new **Square(10, “cm”);**



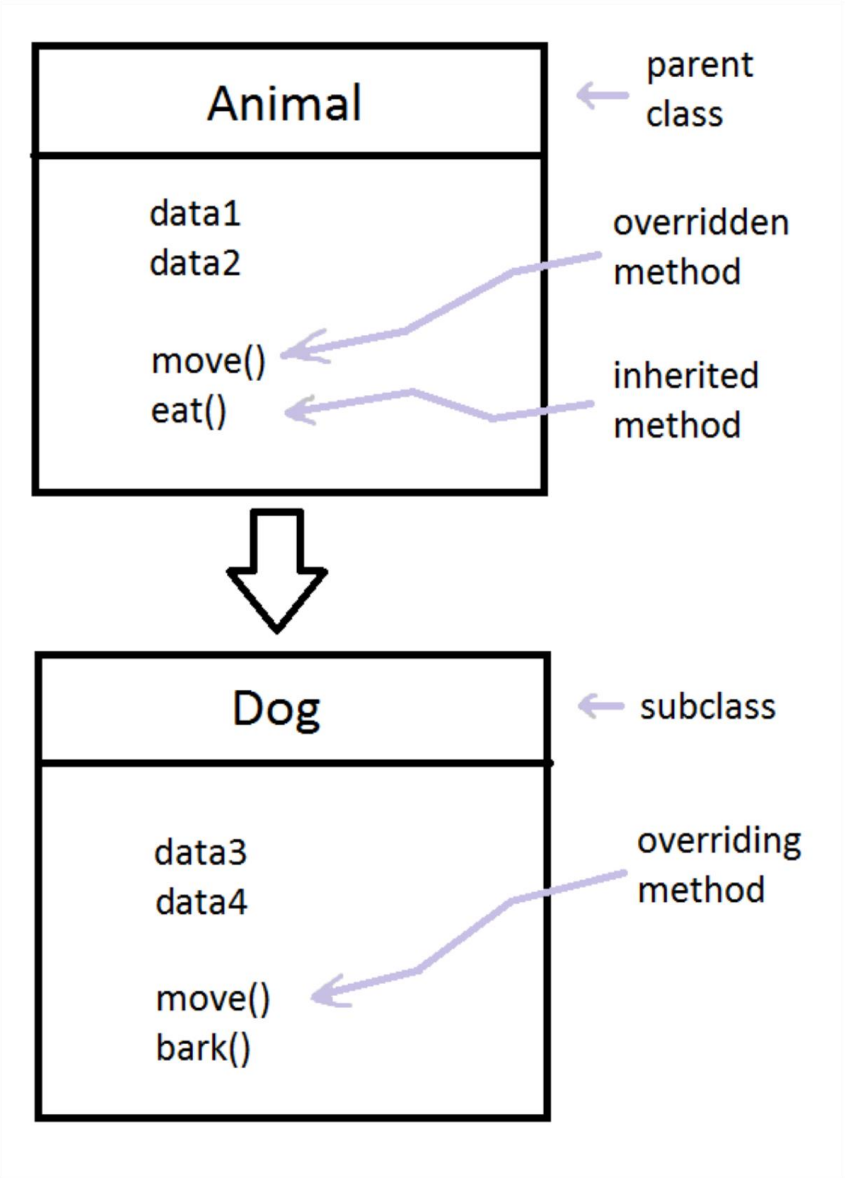
Constructor Chaining

Square r = new **Square(10, “cm”);**

Call the constructor of the Square class!

Execute the remaining body of the Square constructor.

Overriding *Inherited* Methods



An Inherited Method:

toString()

- The `Rectangle` class has this `toString()` method:

```
public String toString() {
    return this.width + " x " this.height;
}
```

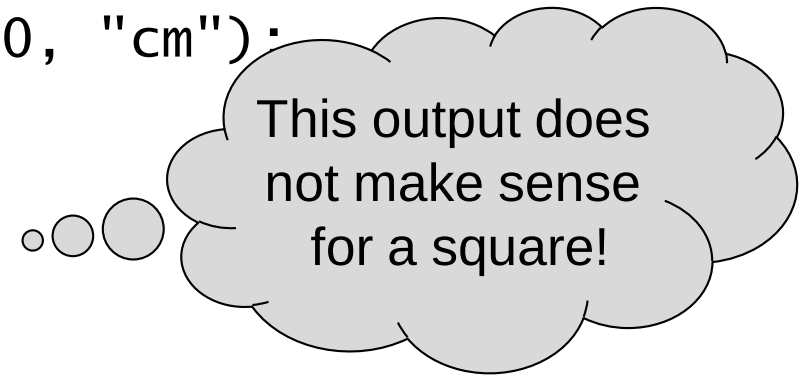
- The `Square` class inherits it from `Rectangle`.

- Thus, unless we take special steps, this method will be called when we print a `Square` object:

```
Square sq = new Square(40, "cm");
System.out.println(sq);
```

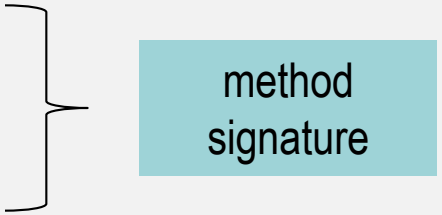
output:

```
40 x 40
```



Overriding an Inherited Method

- A subclass can *override* / replace an inherited method with its own version, which must have the same:
  - return type
  - name
  - number and types of parameters



Overriding an Inherited Method

- A subclass can *override* / replace an
  with its own version, which must ha
  - return type
  - name
  - number and types of parameters

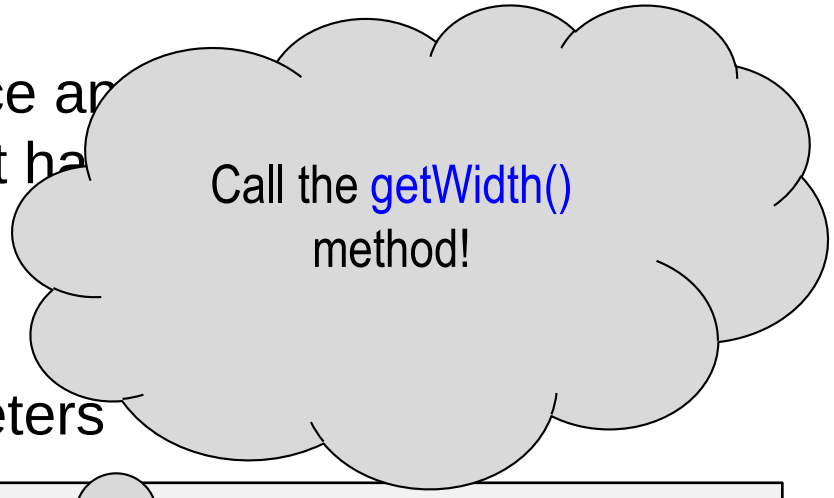
Accessing a private data member of the Rectangle class in a method of the Square class!

- Example: our `Square` class can define its own `toString()`:

```
public String toString() {
    String s = "square with ";
    s += this.width + "-";
    s += this.unit + " sides";
    return s;
}
```

Overriding an Inherited Method

- A subclass can *override* / replace an
  with its own version, which must ha
  - return type
  - name
  - number and types of parameters



- Example: our `Square` class can define its own `toString()`:

```
public String toString() {
    String s = "square with ";
    s += this.getWidth() + "-";
    s += this.unit + " sides";
    return s;
}
```

Overriding an Inherited Method

- A subclass can *override* / replace an inherited method with its own version, which must have the same:
  - return type
  - name
  - number and types of parameters

- Example: our `Square` class can define its own `toString()`:

```
public String toString() {
    String s = "square with ";
    s += getWidth() + "-";
    s += unit + " sides";
    return s;
}
```

Overriding an Inherited Method

- A subclass can *override* / replace an inherited method with its own version, which must have the same:
  - return type
  - name
  - number and types of parameters

- Example: our `Square` class can define its own `toString()`:

```
public String toString() {
    String s = "square with ";
    s += getWidth() + "-";
    s += unit + " sides";
    return s;
}
```

- Printing a `Square` will now call this method, not the inherited one:

```
Square sq = new Square(40, "cm");
System.out.println(sq);
```

```
square with 40-cm sides
```

Overriding Inherited Methods

- A subclass can override **any** method that is accessible to an instance of the subclass.
- Methods that are declared private in the superclass are not accessible to an instance of the subclass and cannot be overridden in the subclass.
- If a private method of the subclass has the same signature as a private method of the superclass, they are completely independent of one another.
- To prevent a method from being overridden in the subclass the method can be defined to be final in the superclass.

Overriding Inherited Methods:

an example

- The `Rectangle` class has the following mutator method:

```
public void setWidth(int w) {
    if (w <= 0) {
        throw new IllegalArgumentException();
    }
    this.width = w;
}
```

- The `Square` class inherits it. Why should we override it?
  to prevent a `Square`'s dimensions from becoming unequal

- One option: have the `Square` version change width *and* height.

Which of these works?

A.

```
// Square version, which overrides
// the version inherited from Rectangle
public void setWidth(int w) {    // no!
    this.width = w;    // can't directly access private
    this.height = w;   // fields from the superclass!
}
```

B.

```
// Square version, which overrides
// the version inherited from Rectangle
public void setWidth(int w) {    // no!
    this.setWidth(w);  // a recursive call!
    this.setHeight(w);
}
```

C. either version would work

D. neither version would work

Accessing Methods from the Superclass

- The Square class should override *all* of the inherited **mutator methods**:

```
// Square versions
public void setWidth(int w) {
    super.setWidth(w);
    super.setHeight(w);
}

public void setHeight(int h) {
    super.setWidth(h);
    super.setHeight(h);
}
```

Accessing Methods from the Superclass

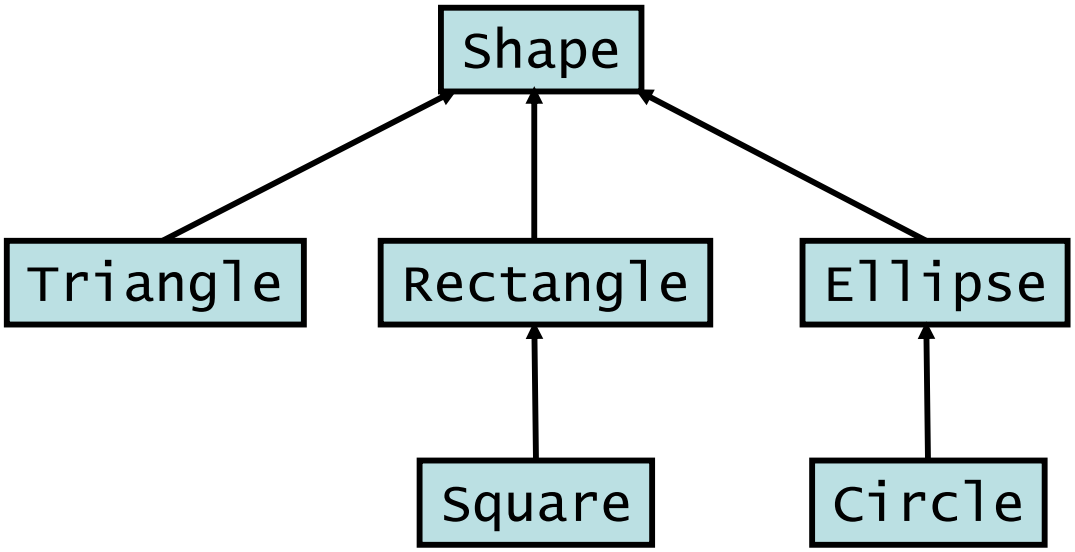
- The Square class should override *all* of the inherited **mutator methods**:

```
// Square versions
public void setWidth(int w) {
    super.setWidth(w);
    super.setHeight(w);
}

public void setHeight(int h) {
    super.setWidth(h);
    super.setHeight(h);
}

public void grow(int dw, int dh) {
    if (dw != dh) {
        throw new IllegalArgumentException();
    }
    super.setWidth(this.getWidth() + dw);
    super.setHeight(this.getHeight() + dh);
}
```

getWidth() and getHeight() are not overridden, so we use this.

Inheritance Hierarchy

- Inheritance leads classes to be organized in a *hierarchy*:




```
public class Shape {
    // fields and methods
    // common to all shapes
    ...
}
```

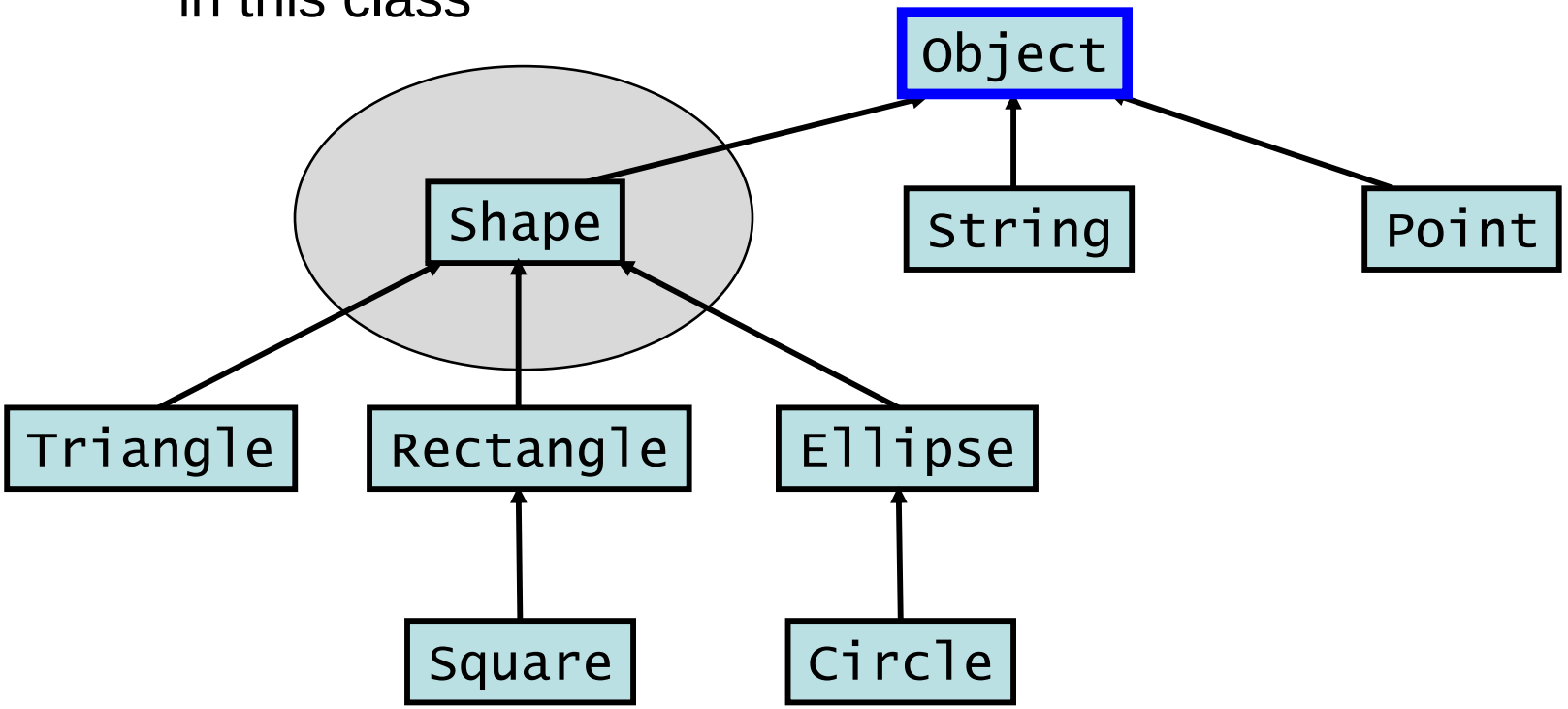
```
public class Rectangle
extends Shape {

    ...

}
```

- A class in Java inherits *directly* from at most one class.
- However, a class can inherit *indirectly* from a class higher up in the hierarchy.
  - example: `Square` inherits indirectly from `Shape`

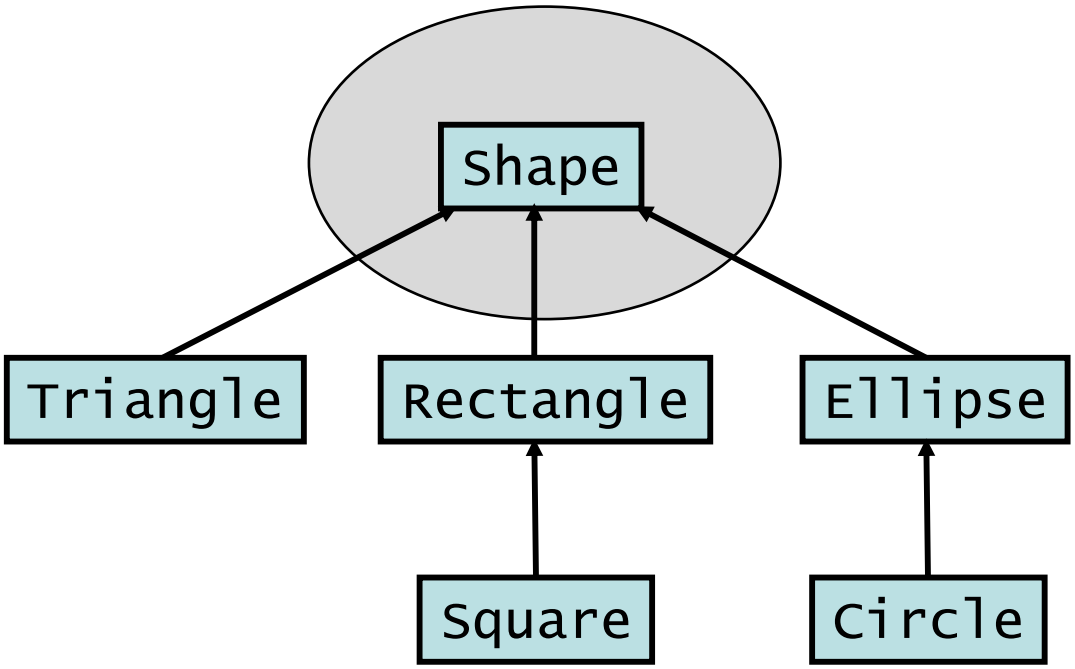
The `Object` Class

- If a class doesn't explicitly extend another class, it implicitly extends a special class called `Object`.

- Thus, the `Object` class is at the top of the class hierarchy.
  - *all* classes are subclasses of this class
  - the default `toString()` and `equals()` methods are defined in this class



Inheritance Hierarchy




```
public class Shape {
    // fields and methods
    // common to all shapes
    ...
}
```

```
public class Rectangle
extends Shape {

    ...

}
```

- What is a shape?
- Does it even make sense to create an object of class Shape? No, Shape is just an *abstraction* by which we identify different types of shapes!

Abstract Classes

```
public abstract class Shape {
    // members common to all shapes
    String shapeName;          // name of the shape
    Point p;                   // some x, y coordinates
    Color c;                   // color

    // constructors
    Shape() {
        // assign default values to name, point, and color
    }
    Shape( String name ) {
        this();                // initialize default values
        shapeName = name;
    }

    // methods common to all shapes
    public String toString() {
        return( shapeName );
    }



}
```

Abstract Classes

```
public abstract class Shape {
    // members common to all shapes
    String shapeName;          // name of the shape
    Point p;                   // some x, y coordinates
    Color c;                   // color

    // constructors
    Shape() {
        // assign default values to name, point, and color
    }
    Shape( String name ) {
        this();                // initialize default values
        shapeName = name;
    }

    // methods common to all shapes
    public String toString() {
        return( shapeName );
    }

    abstract public double area();

}
```

Properties of Abstract Classes

- An *abstract class* is a class that is *declared* to be abstract—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be sub-classed.

- An *abstract method* is a method that is declared without an implementation (without braces, and followed by a semicolon).

- If a class includes abstract methods, then the class itself *must* be declared abstract.

- When an abstract class is sub-classed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, then the subclass must also be declared abstract.

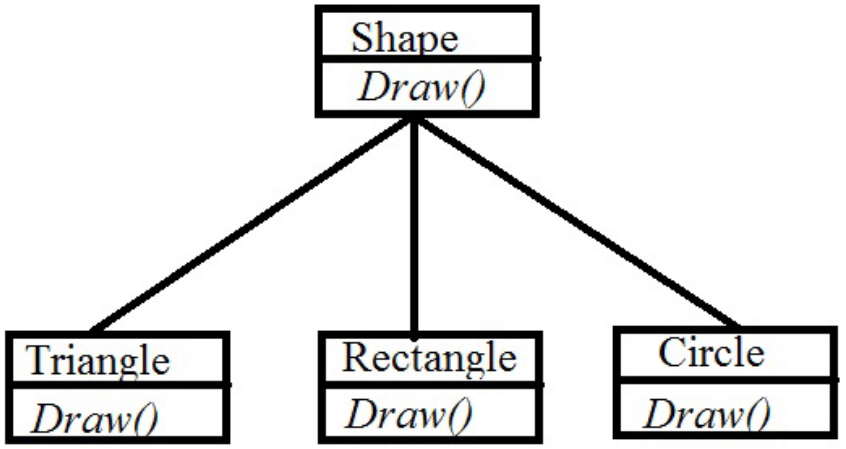
Sometimes we don’t want a class to be extended...

- To prevent a class from being extended, qualify the class name with the final modifier. Example:

```
public final class Circle {



}
```

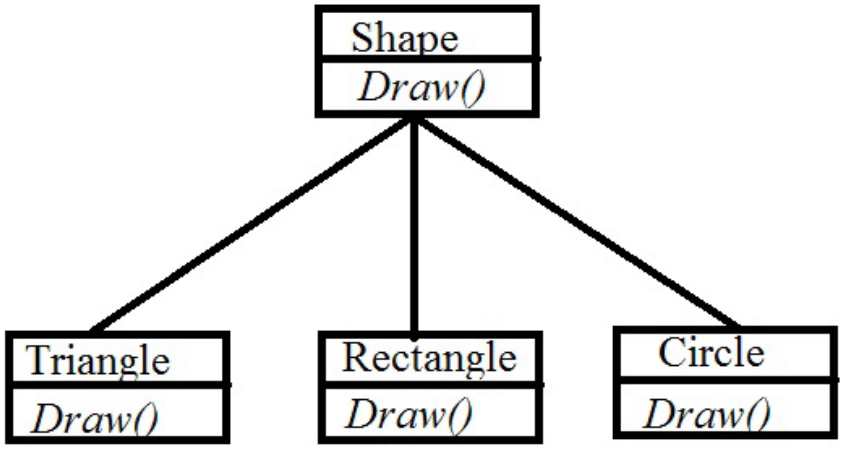
- Will not allow class Circle to be extended!

Polymorphism



Polymorphism

- Recall that an instance of a subclass is an instance of the superclass!
- Polymorphism is the ability to reference instances of a subclass from references of the superclass.



Polymorphism

There are two types of Polymorphism:

- static polymorphism → method overloading!
- dynamic polymorphism → method overriding!

Static Polymorphism is what allows us to implement multiple methods using the same name, but having different signatures.

Dynamic Polymorphism is what allows subclasses to override methods written in the superclass.

Polymorphism

- We've been using reference variables like this:

```
Rectangle r1 = new Rectangle(20, 30);
```

  - variable `r` is declared to be of type `Rectangle`
  - it holds a reference to a `Rectangle` object

- In addition, a reference variable of type T can hold a reference to an object from a *subclass* of T:

```
Rectangle r1 = new Square(50, "cm");
```

  - this works because `Square` is a subclass of `Rectangle`
  - a square *is* a rectangle!

Polymorphism

- We've been using reference variables like this:

```
Rectangle r1 = new Rectangle(20, 30);
```

  - variable `r` is declared to be of type `Rectangle`
  - it holds a reference to a `Rectangle` object

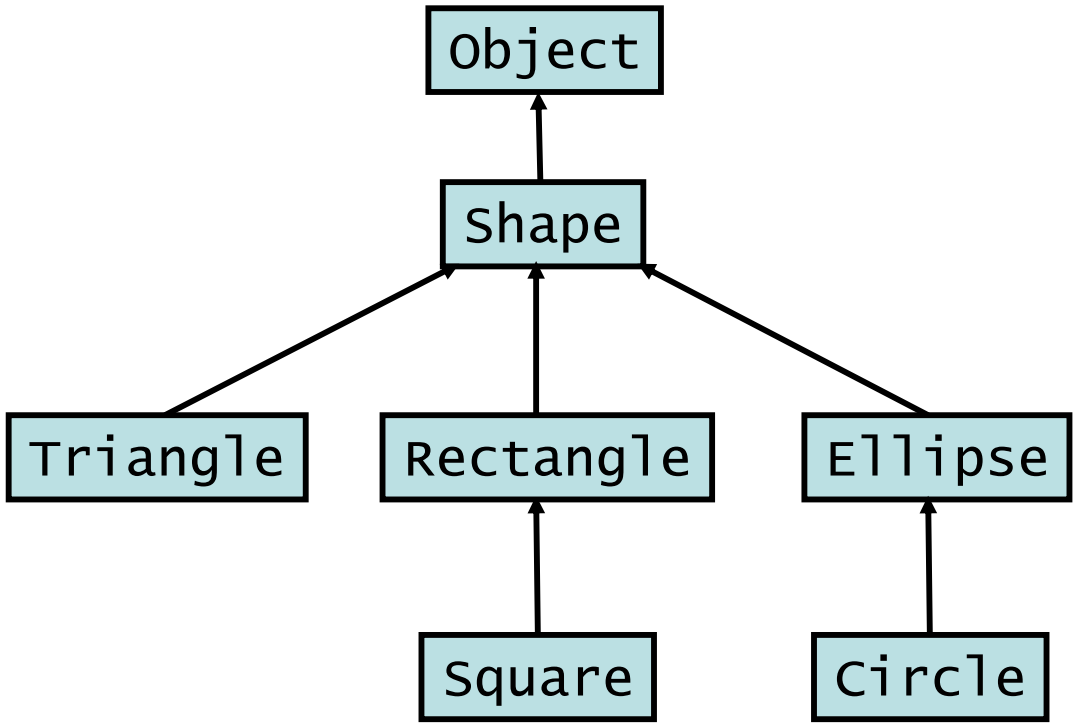
- In addition, a reference variable of type T can hold a reference to an object from a *subclass* of T:

```
Rectangle r1 = new Square(50, "cm");
```

  - this works because `Square` is a subclass of `Rectangle`
  - a square *is* a rectangle!

- The name for this feature of Java is *polymorphism*.
  - from the Greek for “many forms”
  - the same code can be used with objects of different types!

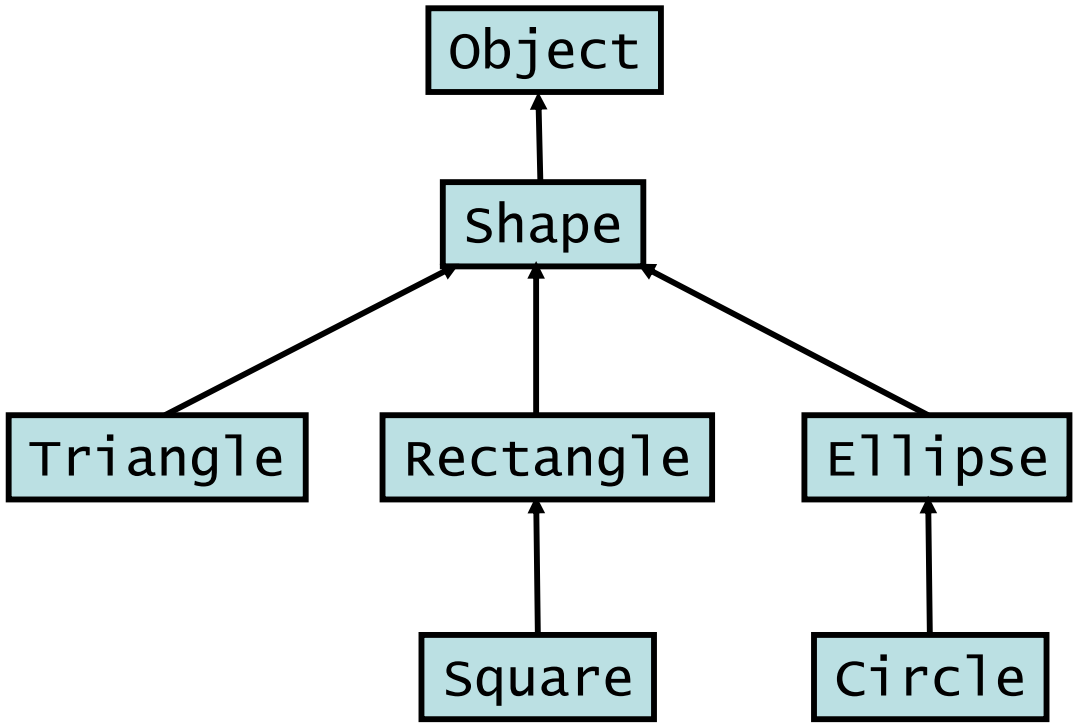
Practice with Polymorphism



- Which of these assignments would be allowed?

```
Shape s1 = new Triangle(10, 8);        // allowed
Square sq = new Rectangle(20, 30);     // not allowed
Rectangle r1 = new Circle(15);         // not allowed
Object o = new Circle(15);             // allowed
Shape s = new Shape();                 // not allowed
```

Which of these would be allowed?



A. `Circle c = new Shape(5);`

B. **`Shape s2 = new Square(8, "inch");`**

C. both would be allowed

D. neither would be allowed