

Principles of the Object Oriented Paradigm



- Hierarchical nature
- Identifiable Components
- Common Patterns

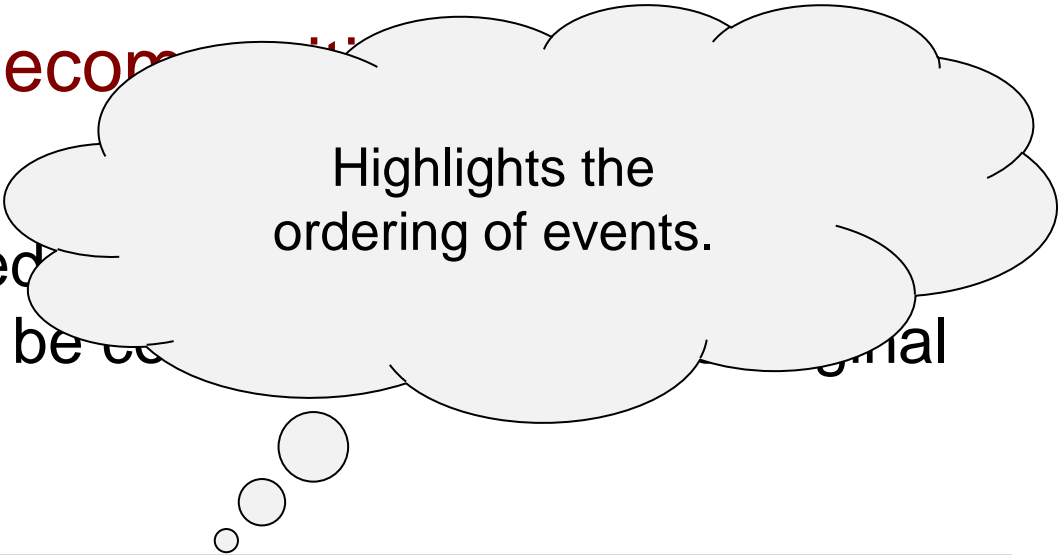
Handle and Model **Complex** Systems

Decomposition

Decomposition is used to break software into *components* that can be combined to solve the original problem.

Algorithmic decomposition breaks down the problem into major steps in the overall process of forming the solution. A top down structured approach to software development.

Decomposition



Highlights the
ordering of events.

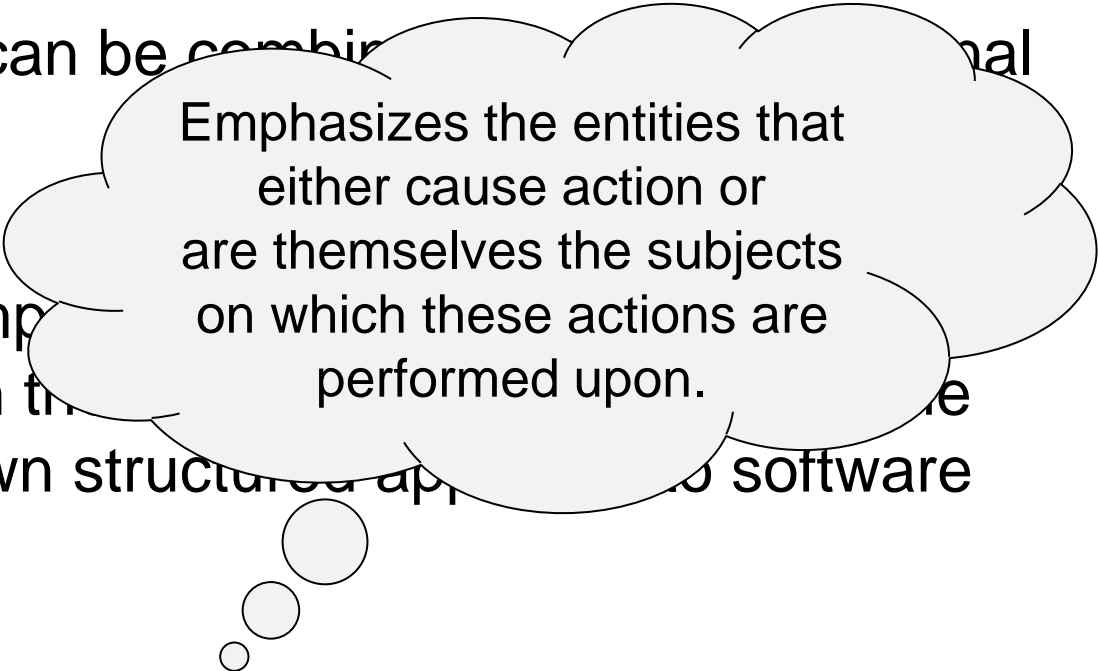
Decomposition is used to break down a problem into *components* that can be solved individually.

Algorithmic decomposition breaks down the problem into major steps in the overall process of forming the solution. A top down structured approach to software development.

Decomposition

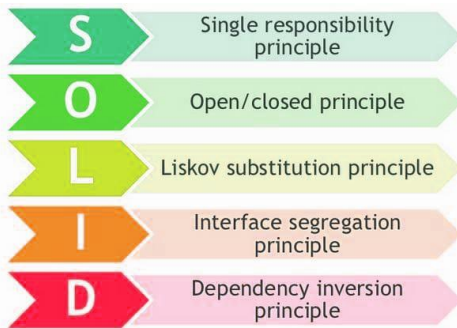
Decomposition is used to break software into *components* that can be combined to solve the original problem.

Algorithmic decomposition breaks down the problem into major steps in the solution. A top down structured approach to software development.



Emphasizes the entities that either cause action or are themselves the subjects on which these actions are performed upon.

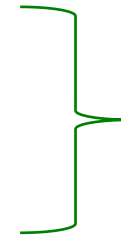
Object decomposition breaks down the problem into identifiable objects. The objects themselves are derived directly from the problem domain.



Decomposition

Single Responsibility Principle

a class should only **have one responsibility**, further defined by Martin as ***‘one reason to change’***



limiting the impact of change

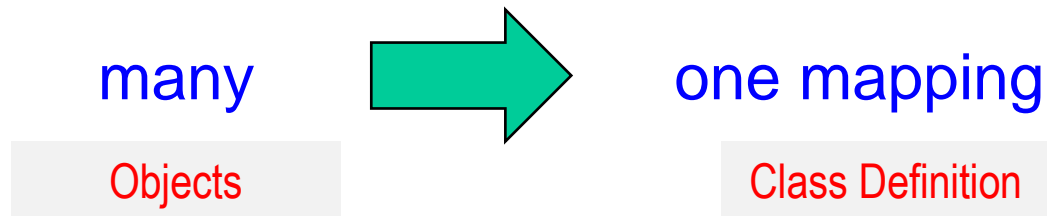
‘gather together those things that change for the same reasons’

Robert Martin

Principle of Abstraction

Purpose of abstraction is to handle the complexity of a software system by hiding unnecessary details from the user or client.

Abstraction assists us the process of decomposition!



Object Decomposition:

Principle of Abstraction

User
Protected?

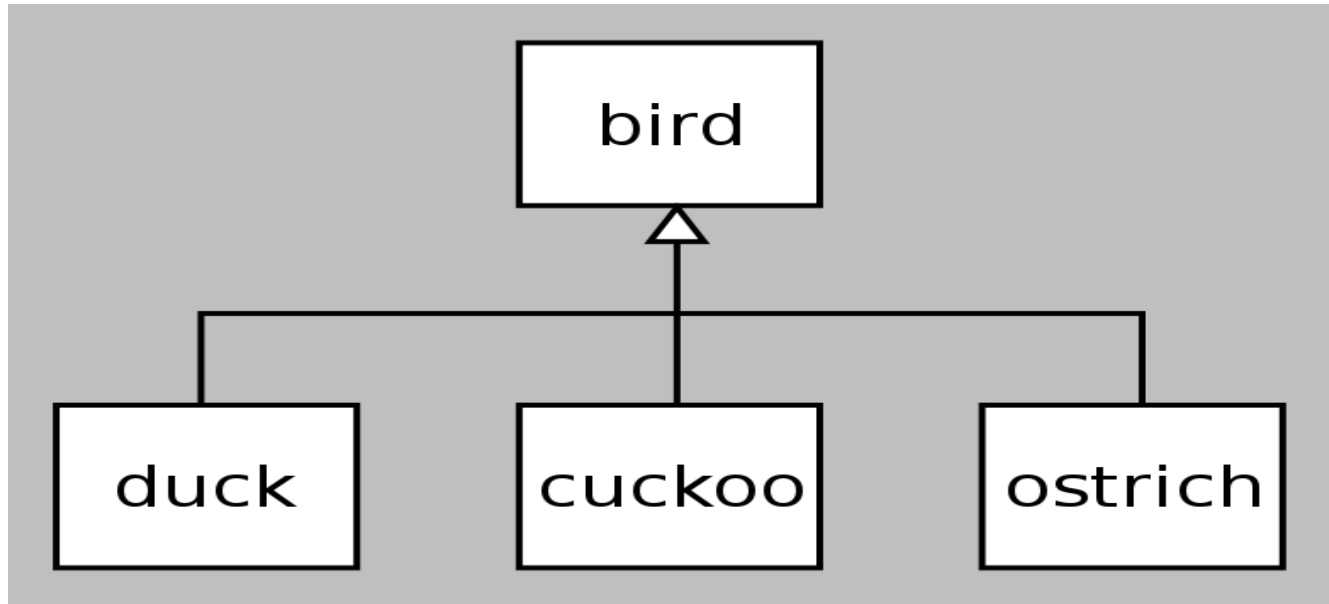
Scalable?

Are the
components
re-useable?

Is it
extendible?



Inheritance and Polymorphism



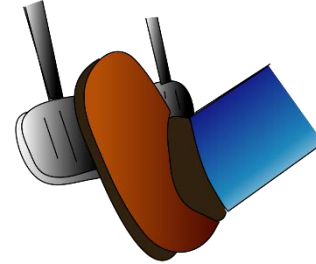
Computer Science OOD
Boston University

Christine Papadakis-Kanaris

Inheritance: *a vehicle hierarchy*



shutterstock · 151949822



shutterstock · 313264631

shutterstock · 133042568

sedan

van

jeep



Inheritance:

a vehicle hierarchy

switch on

turn

signal

accelerate

break

reverse

sedan

van

jeep



Inheritance:

a vehicle hierarchy

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

sedan



van



jeep



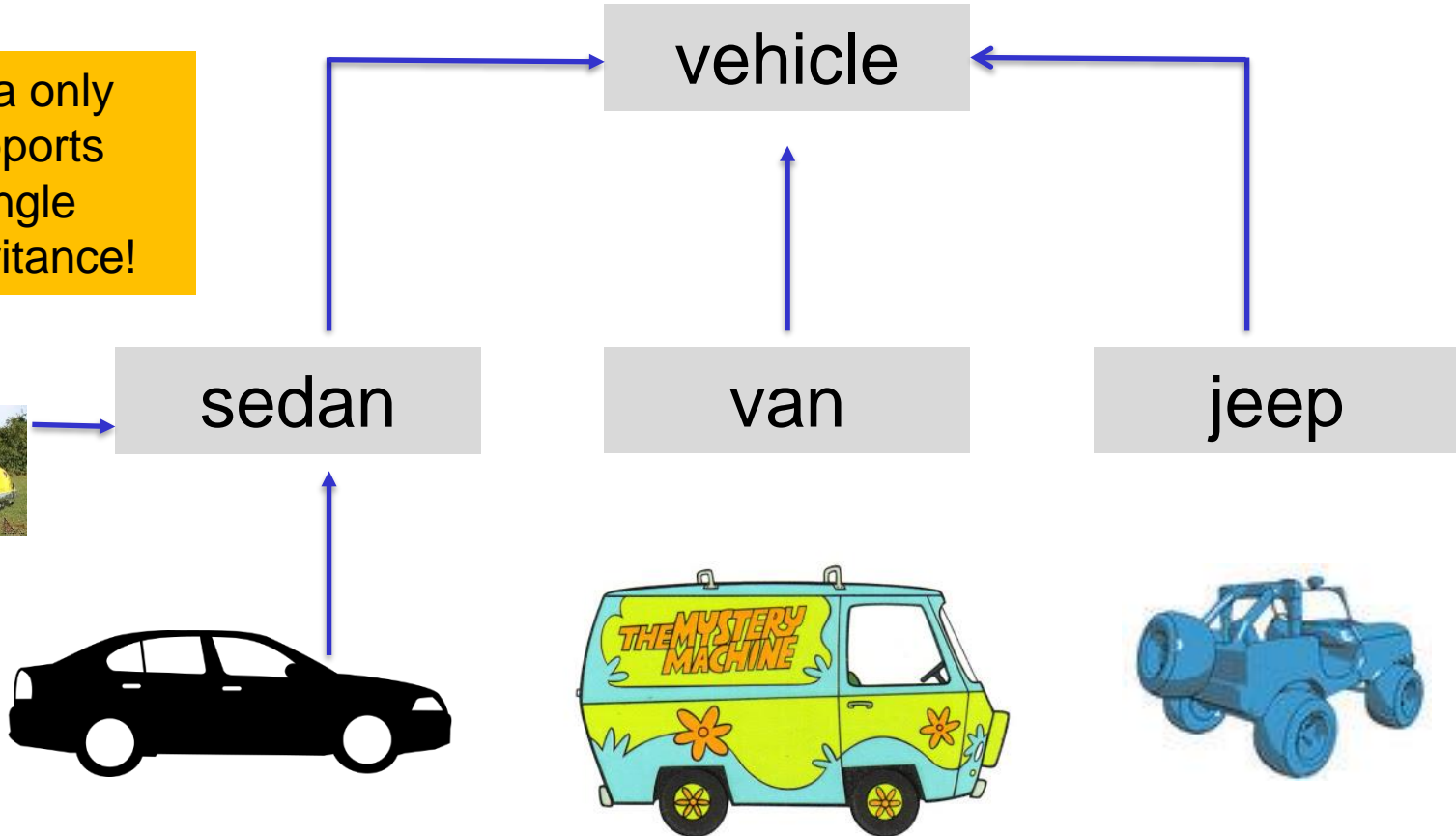
Inheritance:

a vehicle hierarchy

Single Inheritance

A new class is derived from only one parent class.

Java only supports single Inheritance!



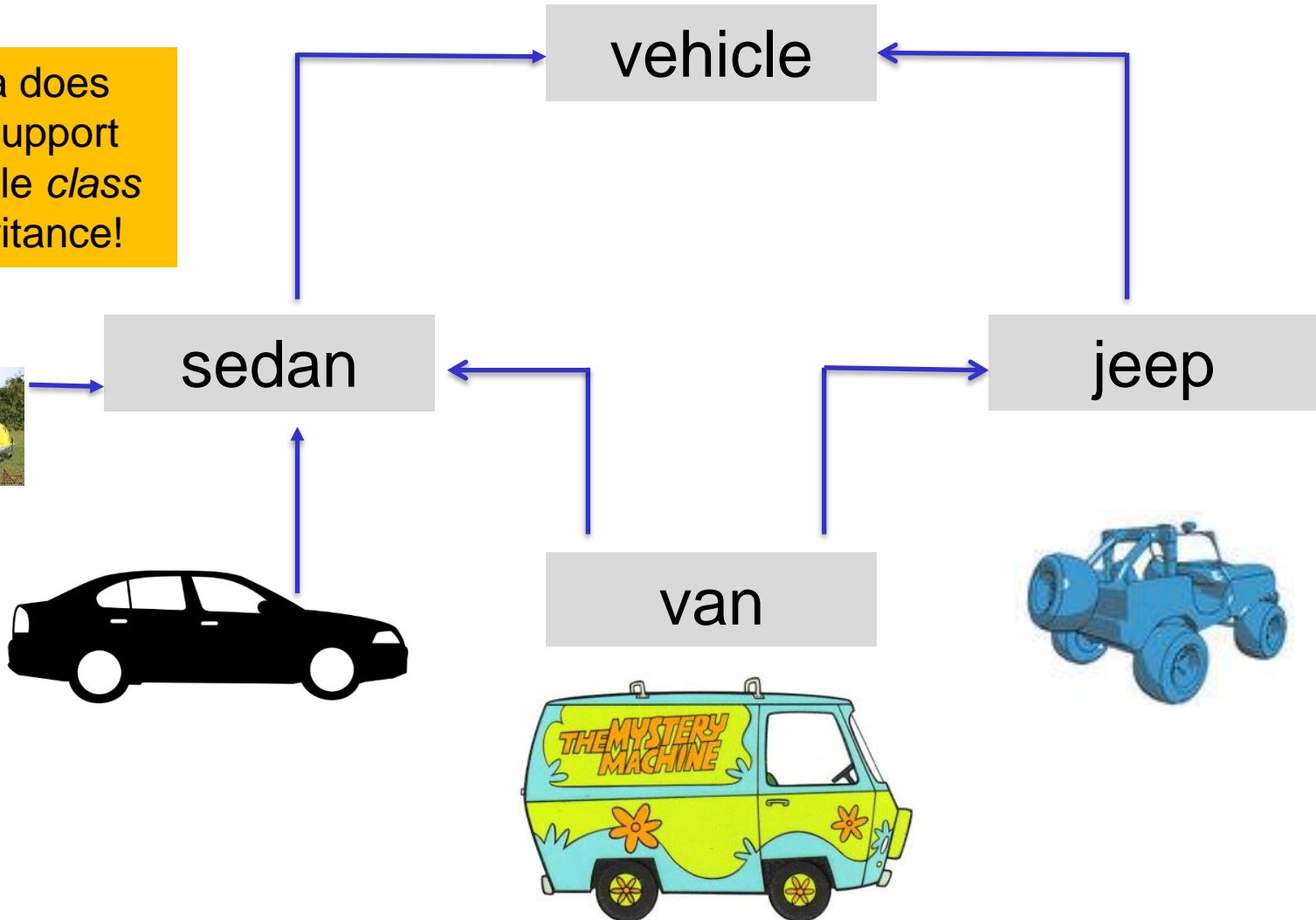
Inheritance:

a vehicle hierarchy

Multiple Inheritance

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

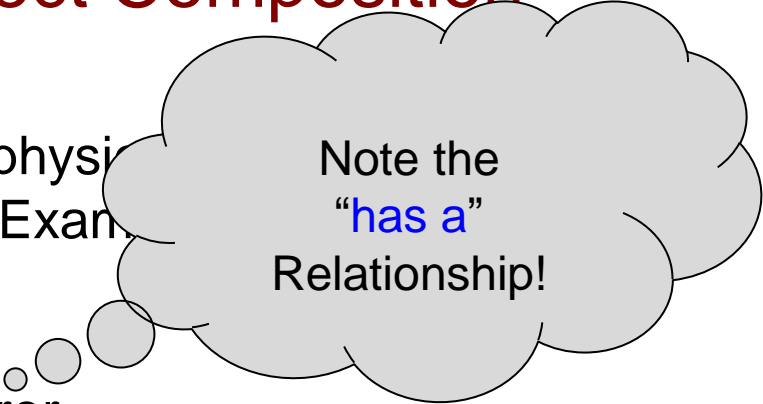
Java does not support multiple *class* Inheritance!



Inheritance vs. Object Composition

- Object composition refers to the physical 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.



Note the
“has a”
Relationship!

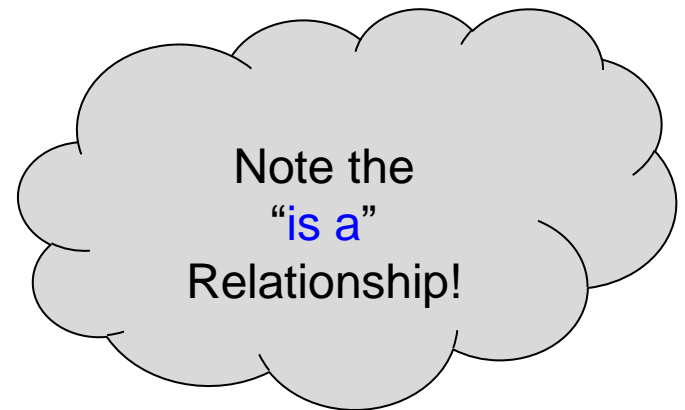
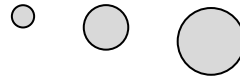
Inheritance vs. Object Composition

- 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



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.

| | |
|--------|------|
| width | 40 |
| height | 40 |
| unit | "cm" |

- 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

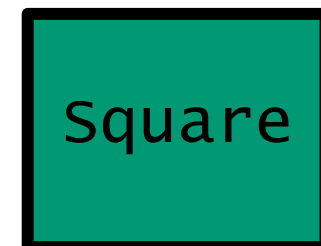
Using Inheritance

```
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;  
    }  
    ...  
}
```



Is A



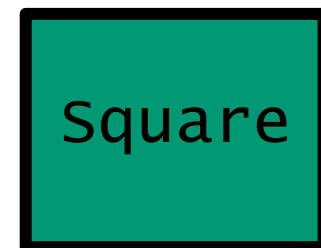
Using Inheritance

```
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;  
    }  
    ...  
}
```



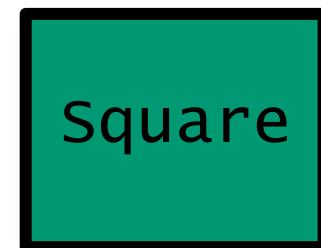
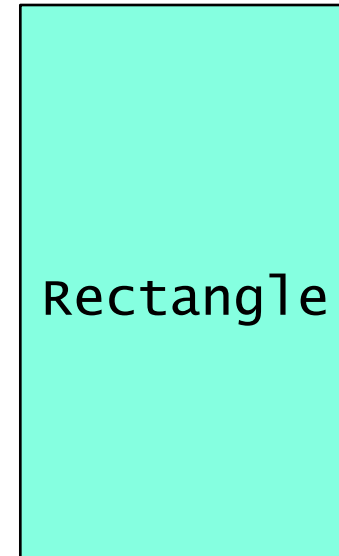
extends



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  
}
```

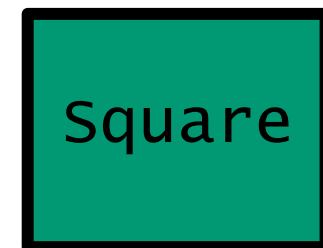
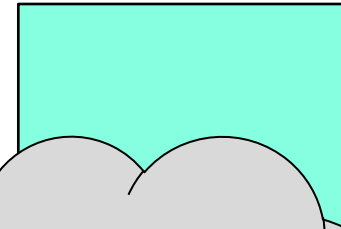


Using Inheritance

```
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  
}
```

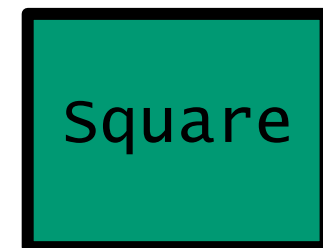
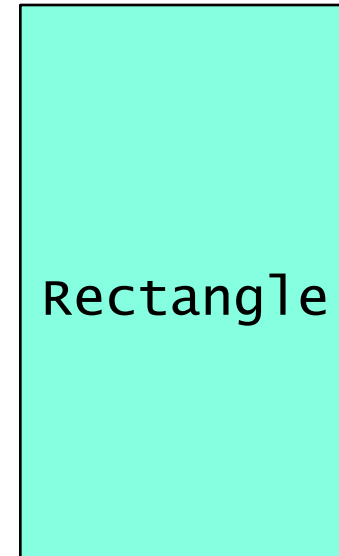
Note that we no longer have to include width and height as data members of class square because they are inherited from ... **class Rectangle!**

Square

Using Inheritance

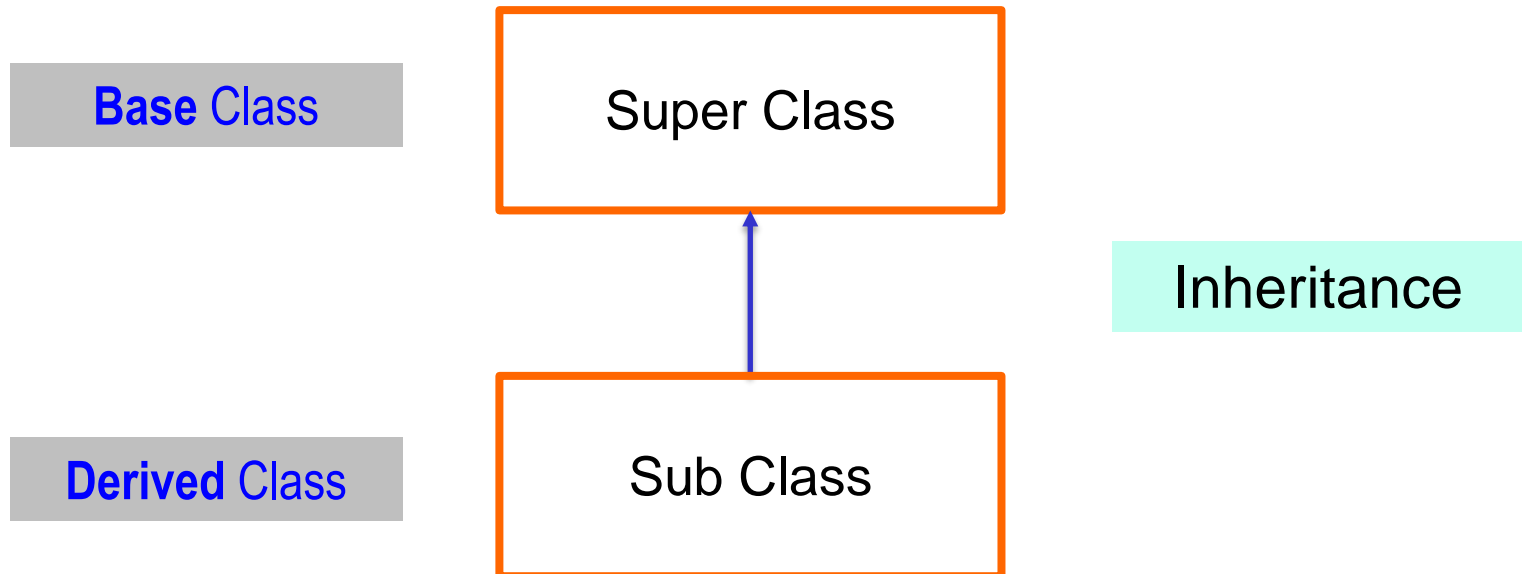
```
public class Rectangle {  
    private int width;  
    private int height;  
  
    public Rectangle(int w, int h) {  
        setWidth(w);  
        setHeight(h);  
    }  
    ... // other methods  
    public int area() {  
        return width * height;  
    }  
}
```

```
public class Square extends Rectangle {  
    String unit;  
  
    public Square(int side, String unit) {  
        // initialize data members  
    }  
  
    // inherits all visible methods  
}
```



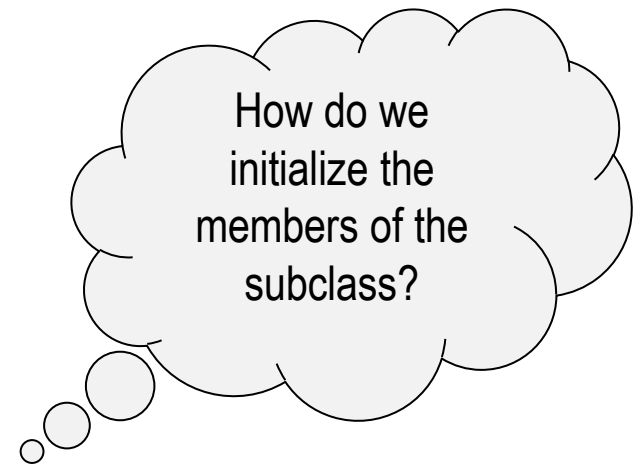
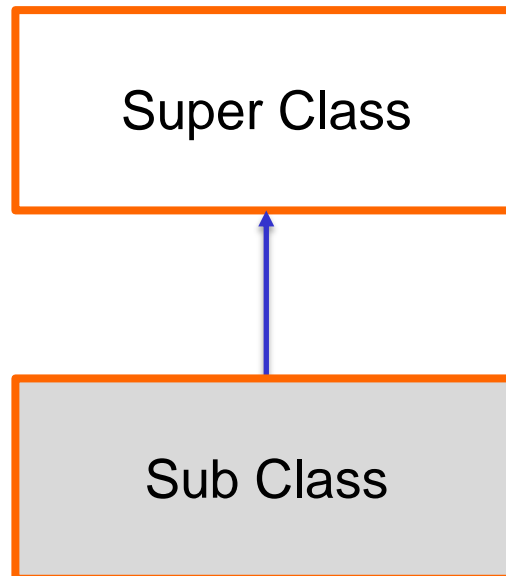
Using Inheritance

- 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.



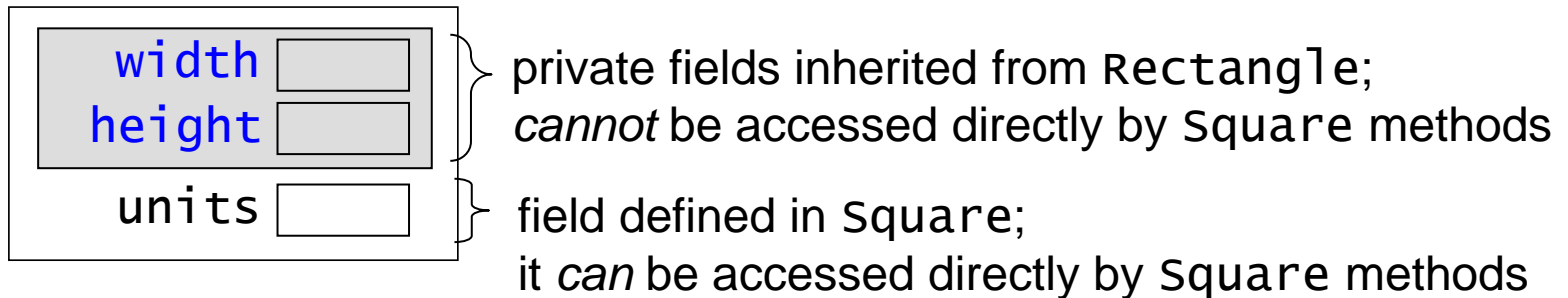
Using Inheritance

- 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:



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

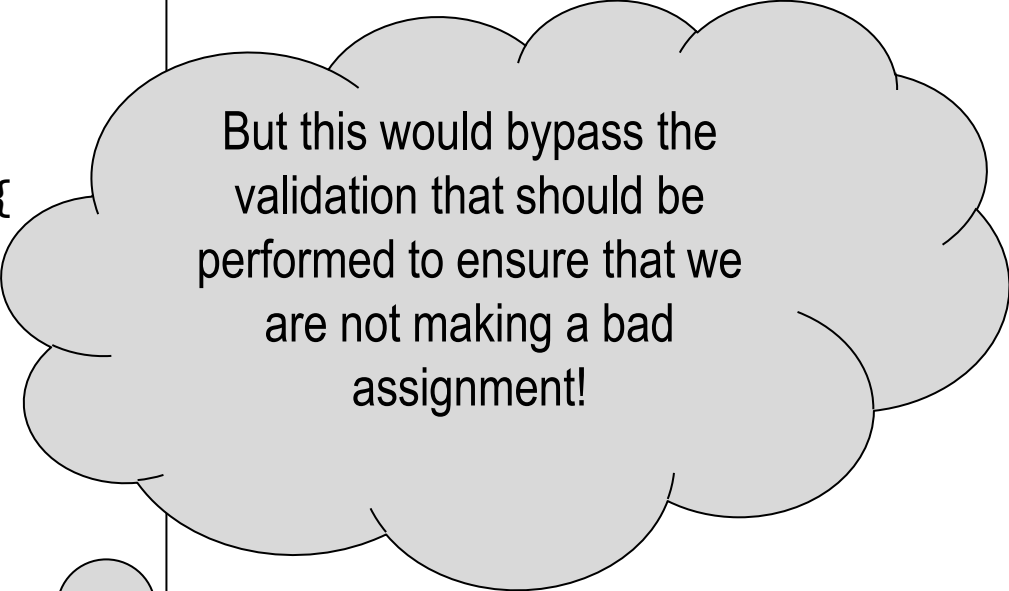
- Change the modifier in the super class from private to protected.
- The protected modifier allows the fields to remain private within the class they are defined in but allows them to be accessible to all subclasses.
- But for the most part it is more prudent to use the **public accessor** and **mutator** 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  
}
```



But this would bypass the validation that should be performed to ensure that we are not making a bad assignment!

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;  
    }  
}
```

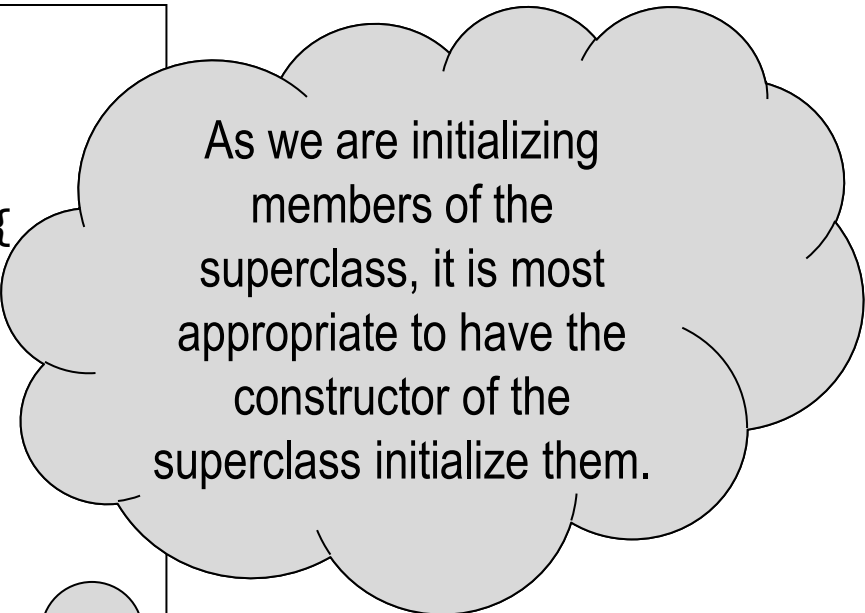
We could invoke public *mutator* methods of the Rectangle class, but ... We are already doing this in the Rectangle constructor!

```
public class Square extends Rectangle {  
    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;  
    }  
}
```



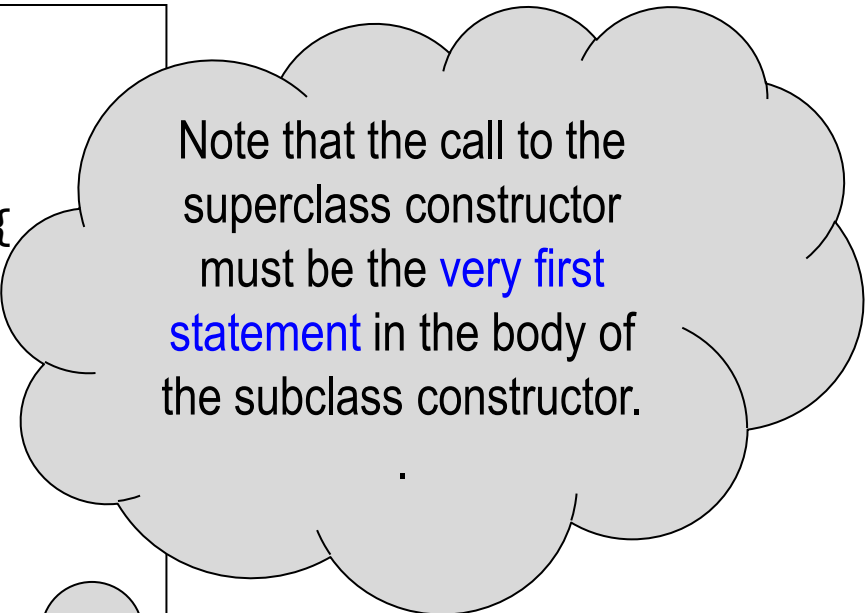
As we are initializing members of the superclass, it is most appropriate to have the constructor of the superclass initialize them.

```
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;  
    }  
}
```



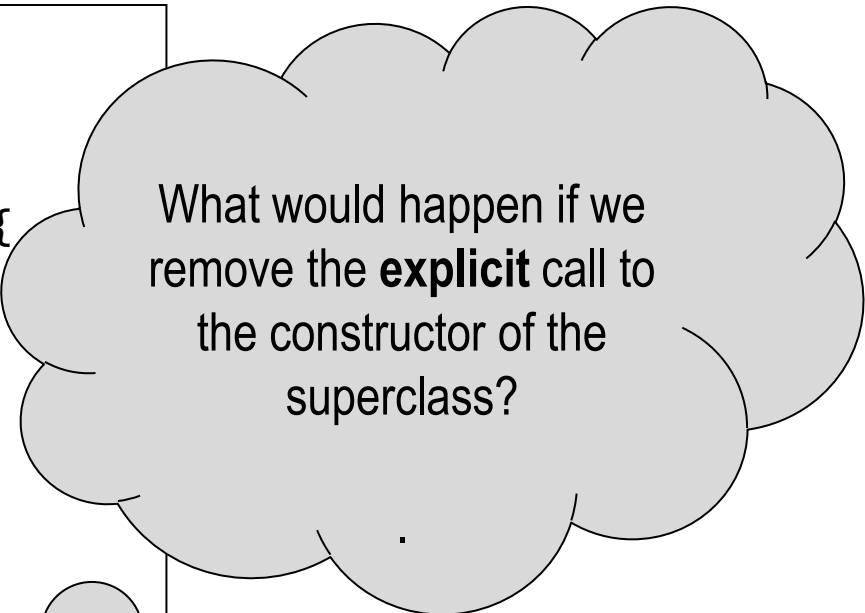
Note that the call to the superclass constructor must be the **very first statement** in the body of the subclass constructor.

```
public class Square extends Rectangle {  
    String unit;  
  
    public Square(int side, String unit) {  
        super(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;  
    }  
}
```



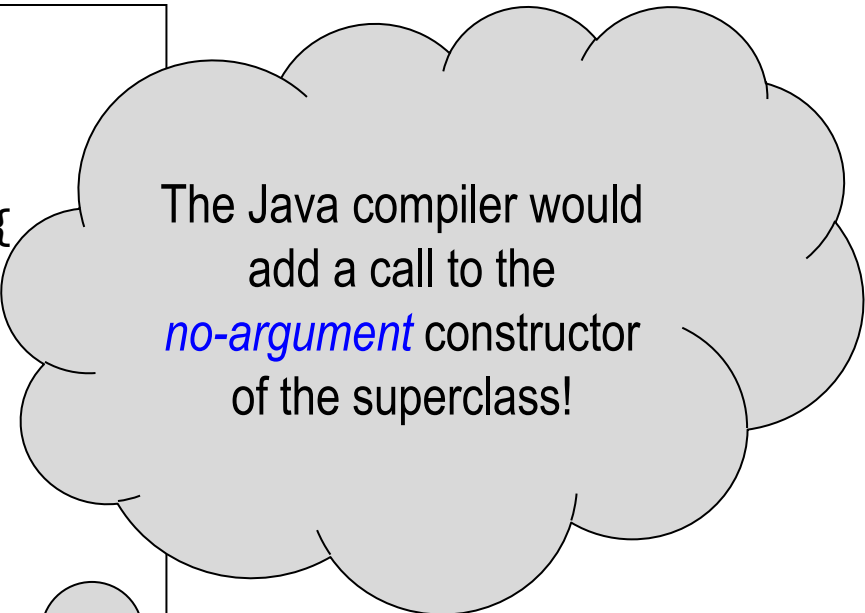
What would happen if we remove the **explicit** call to the constructor of the superclass?

```
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;  
    }  
}
```



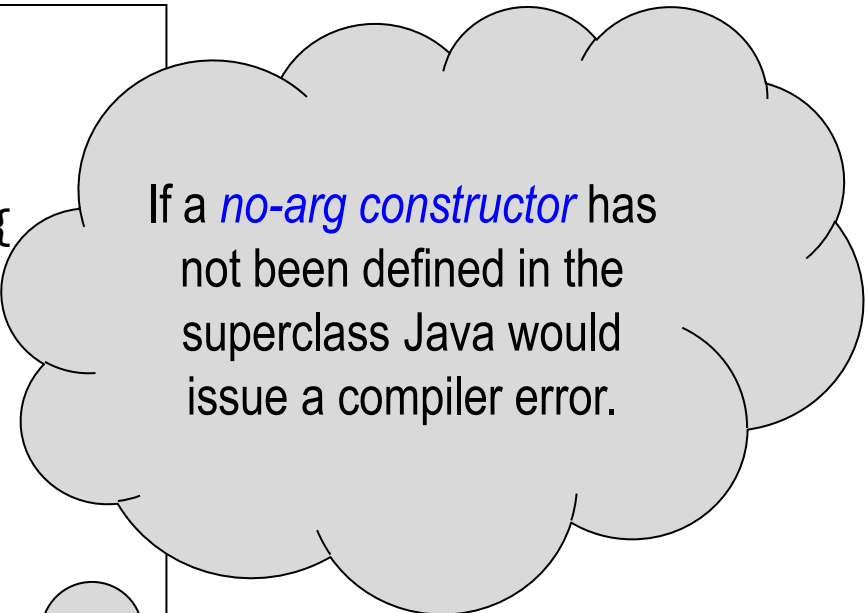
The Java compiler would
add a call to the
no-argument constructor
of the superclass!

```
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

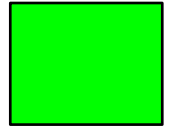
```
public class Rectangle {  
    private width;  
    private height;  
  
    public Rectangle(int w, int h) {  
        setwidth(w);  
        setHeight(h);  
    }  
    ... // other methods  
    public int area() {  
        return width * height;  
    }  
}
```



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



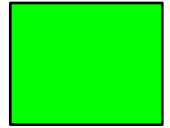
If a class does not define *any* constructors the Java compiler will create a **no-argument** constructor for our class. This constructor will be used if we create an object without passing any arguments.

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



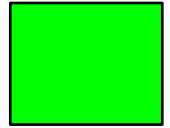
Unlike the methods of a class, constructors of a superclass **are not inherited** by the subclass.

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”);

Call the constructor of the
Square class!

Call the constructor of the
Rectangle class!

Execute the body of the
Rectangle constructor.

return



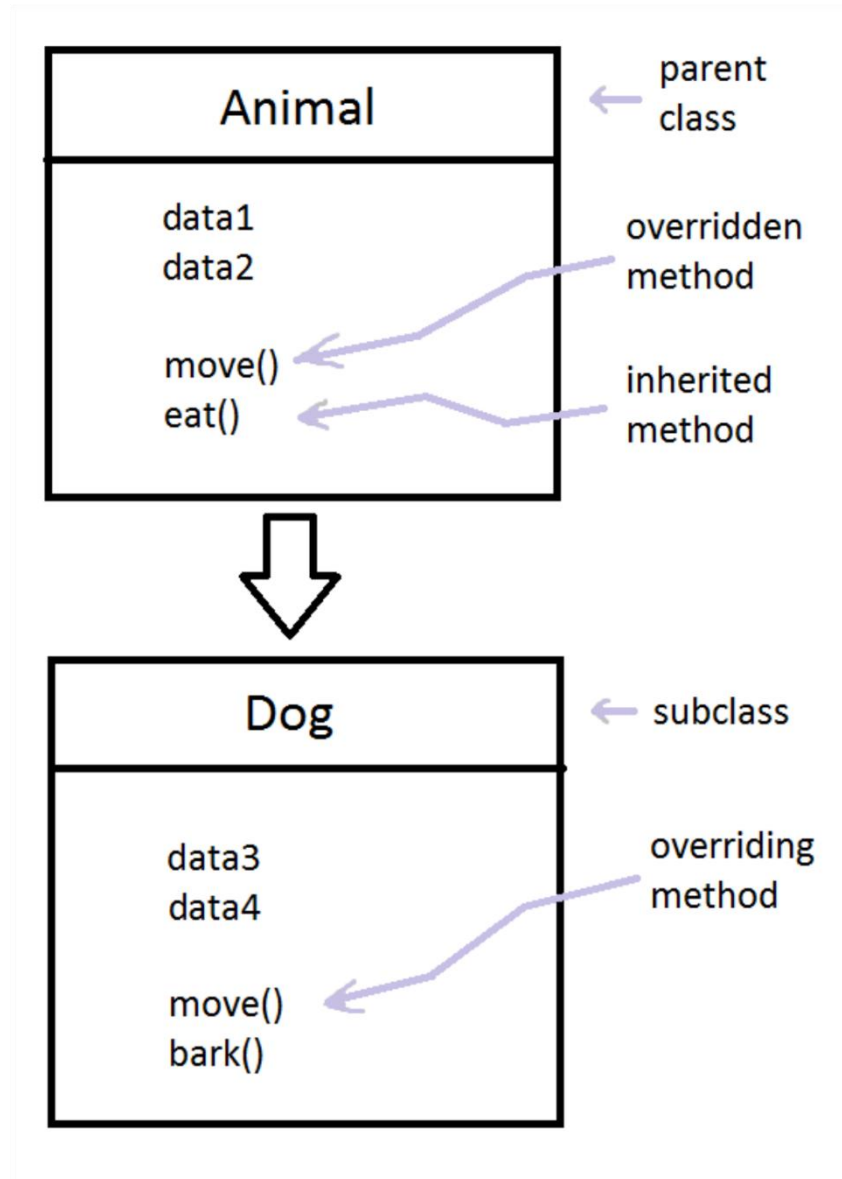
Constructor Chaining

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()

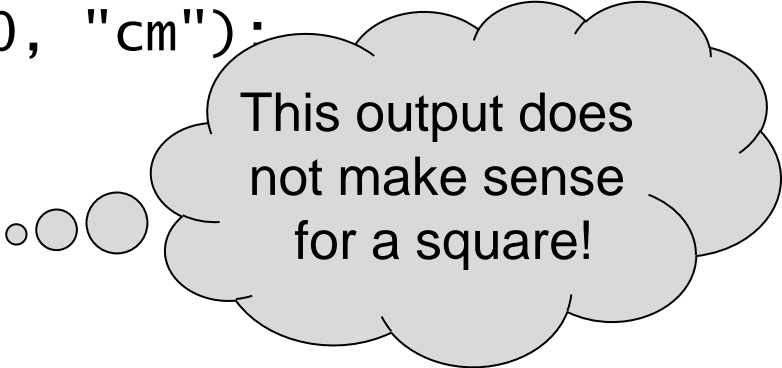
- The Rectangle class has this toString() method:

```
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



This output does not make sense for a square!

Overriding an Inherited Method

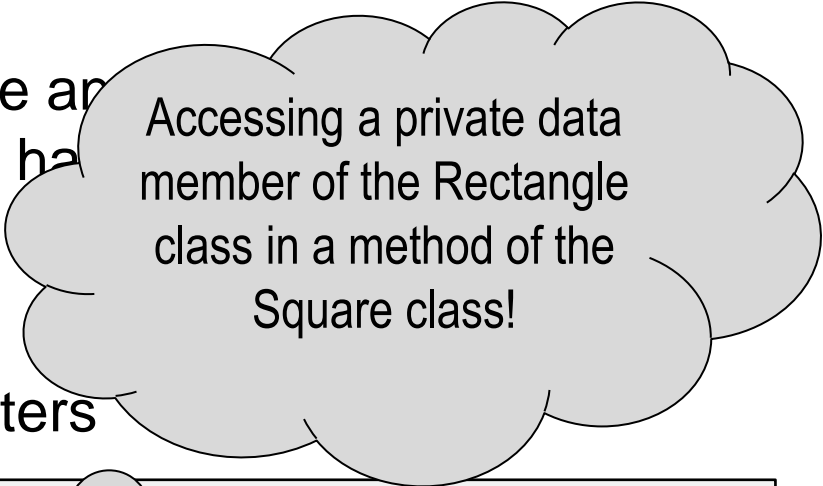
- A subclass can *override* / replace an inherited method with its own version, which must have the same:
 - return type
 - name
 - number and types of parameters
- Example: our Square class can define its own toString():

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



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

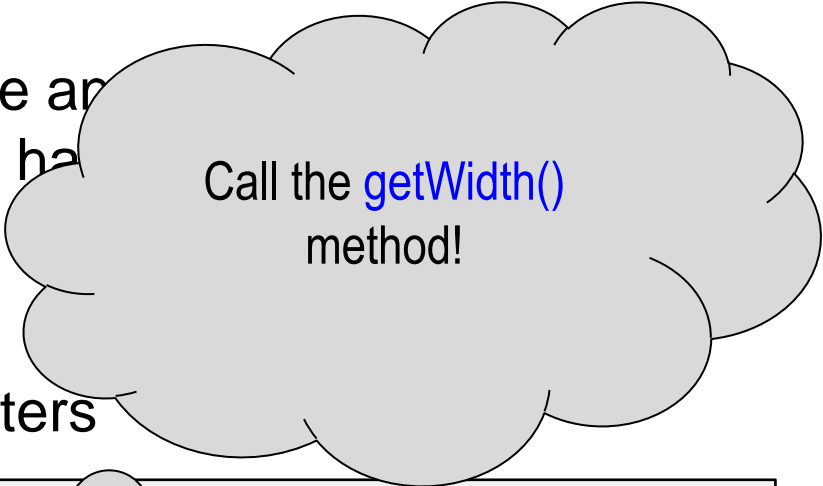
- Example: our Square class can define its own toString():

```
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 inherited method with its own version, which must have the same

- return type
- name
- number and types of parameters



Call the `getWidth()` method!

- 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 += this.getWidth() + "-";  
    s += this.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.
- Static methods are inherited, but cannot be overridden (sort of). If a static method of the superclass is redefined in the subclass, the superclass method is *hidden* but can be invoked using the name of the **superclass** (i.e. **superClass.methodName()**).

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 solution: use `super` to access the inherited version of the method – the one we are overriding:

```
// Square version
public void setWidth(int w) {
    super.setWidth(w); // call the Rectangle version
    super.setHeight(w);
}
```

- Only use `super` if you want to call a method from the superclass *that has been overridden*.
- If the method has *not* been overridden, use this as usual.

Accessing Methods from the Superclass

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

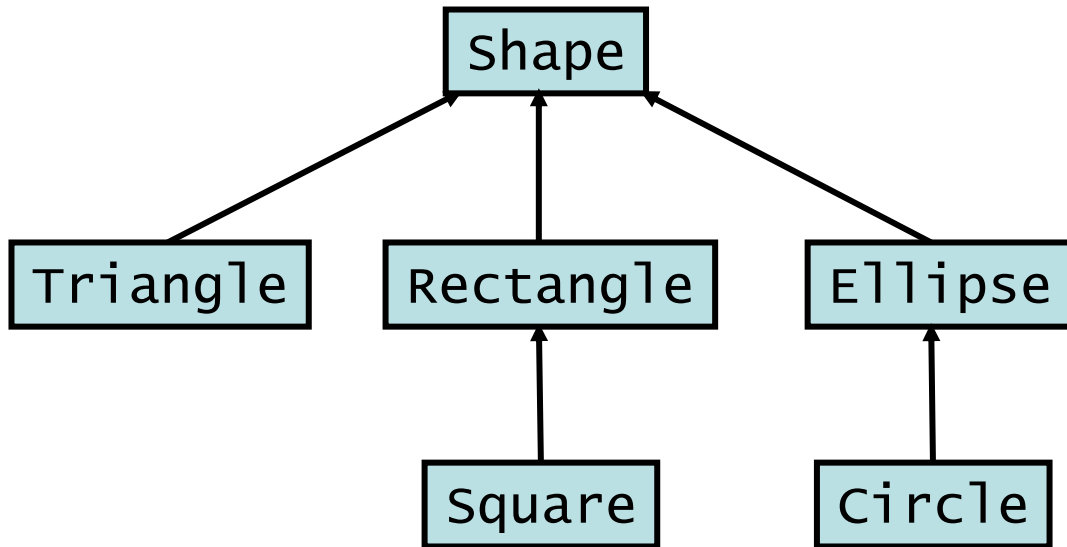
```
// Square versions
public void setwidth(int w) {
    super.setwidth(w);
    super.setHeight(w);
}

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(getwidth()+dw);
    super.setHeight(getHeight()+dh);
}
```

Inheritance Hierarchy

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



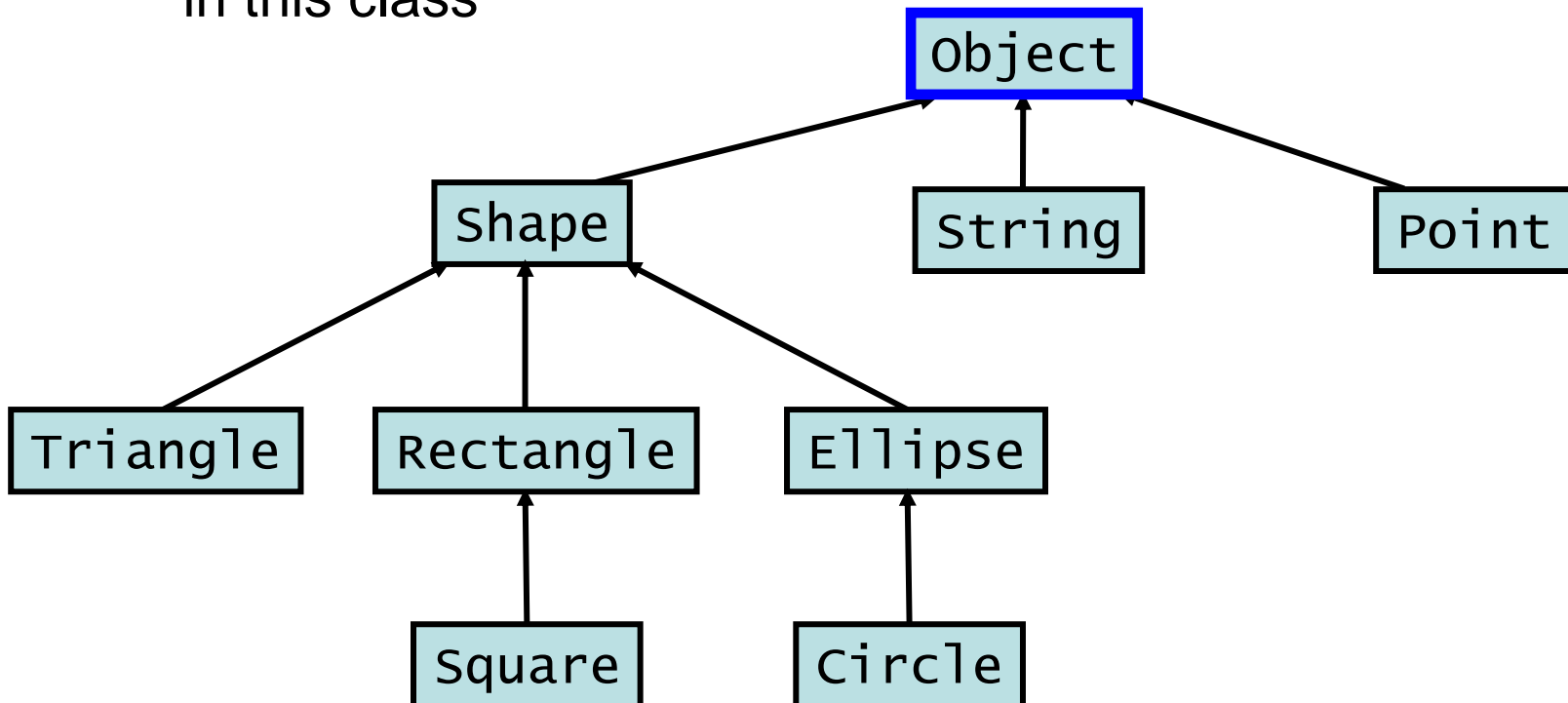
```
public class Shape {  
    // fields and methods  
    // common to all shapes  
    ...  
}
```

```
public class Rectangle  
extends Shape {  
    ...  
}
```

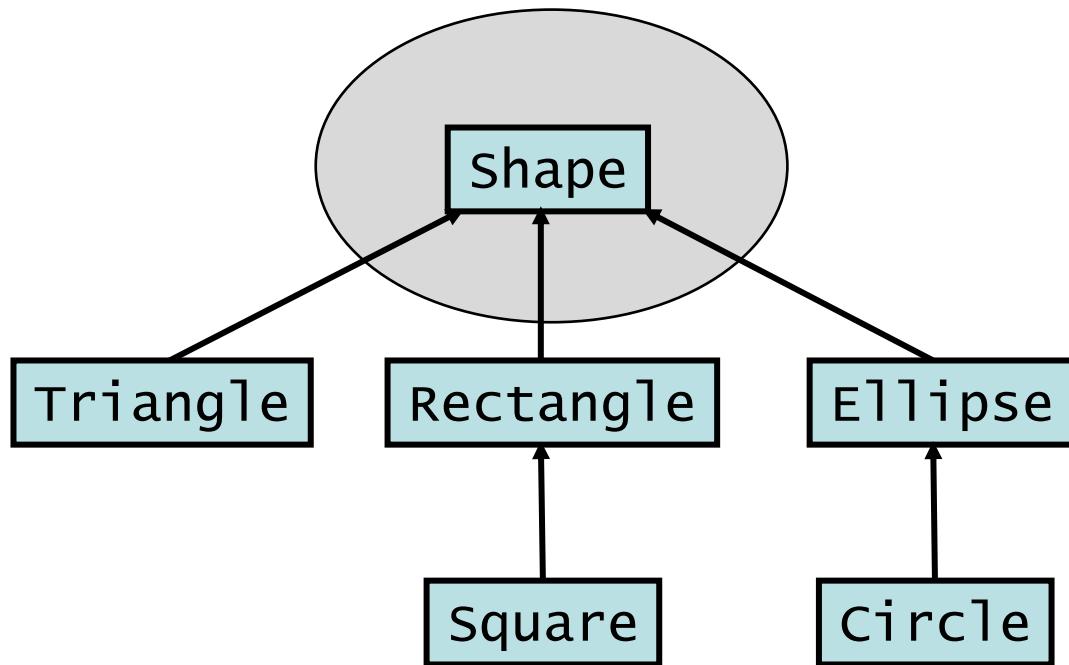
- A class in Java inherits *directly* from at most one class.
- However, a class can inherit *indirectly* from a class higher up in the hierarchy.
 - example: Square inherits indirectly from Shape

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

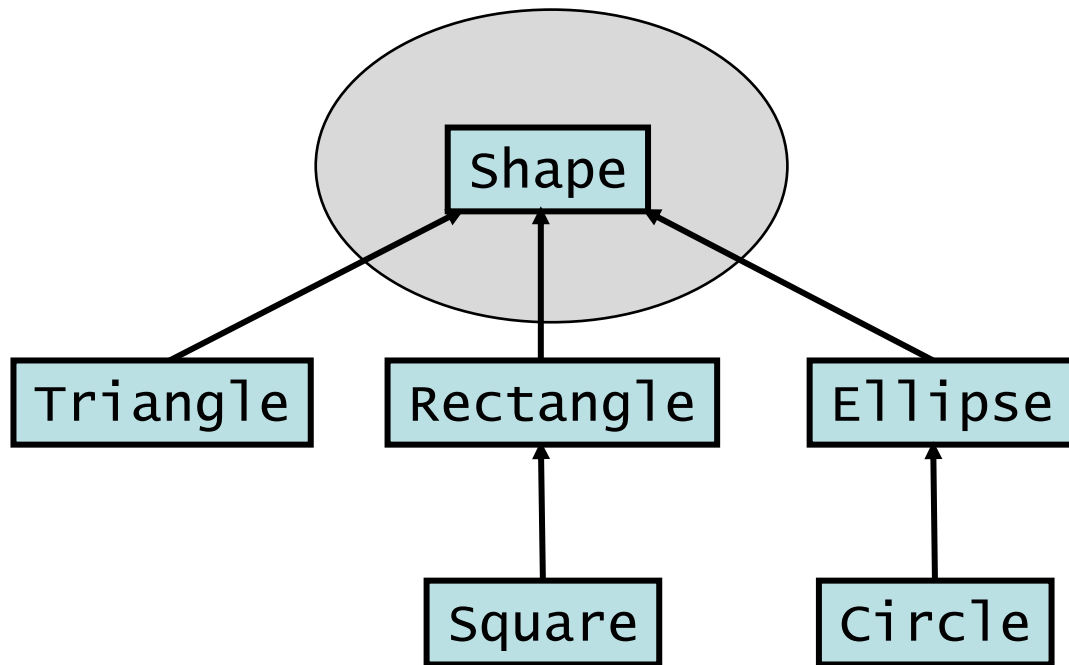


```
public class Shape {  
    // fields and methods  
    // common to all shapes  
    ...  
}
```

```
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!

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? **Our shape class should then only be an abstraction by which we only use to create other classes!**