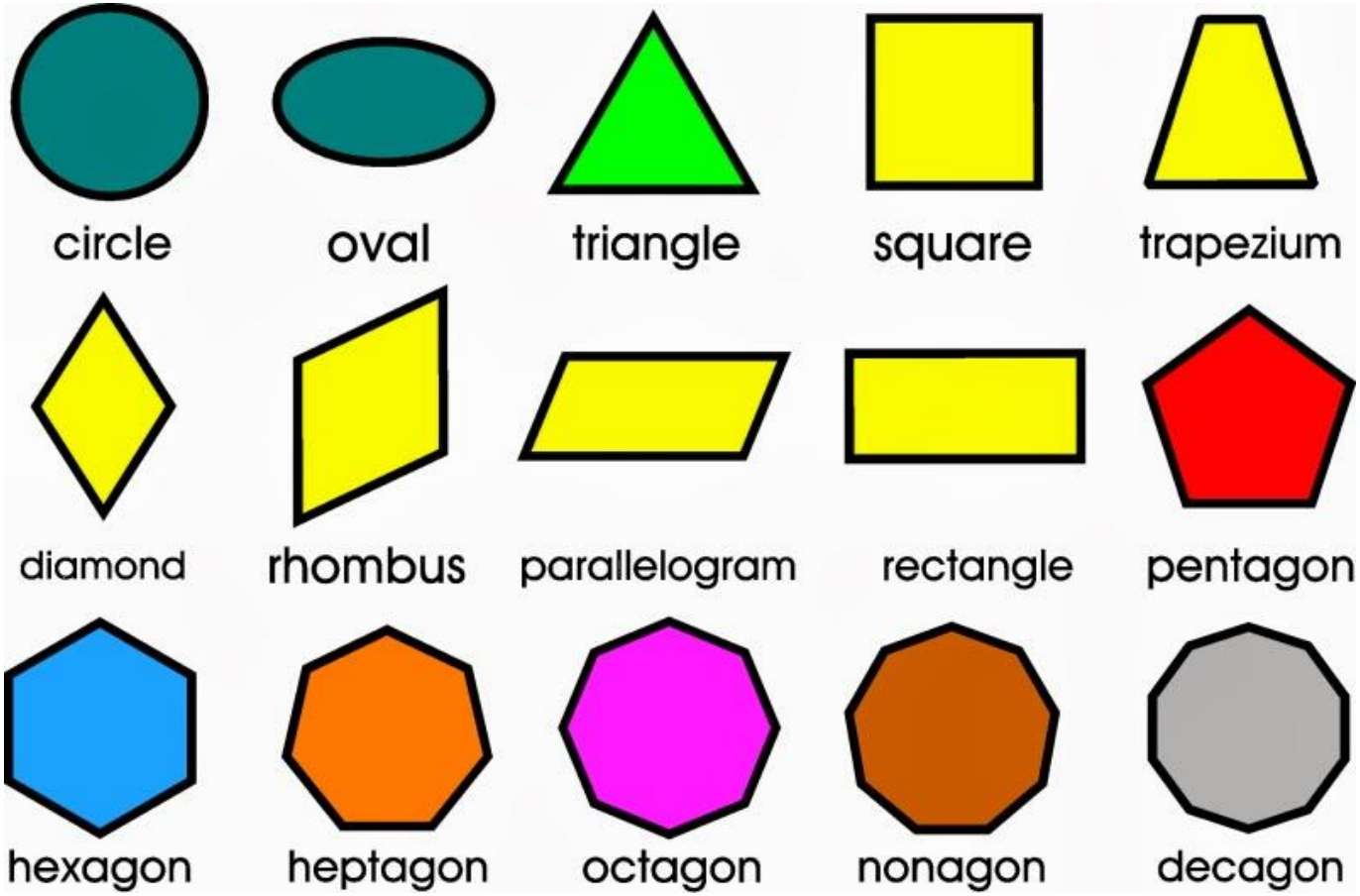
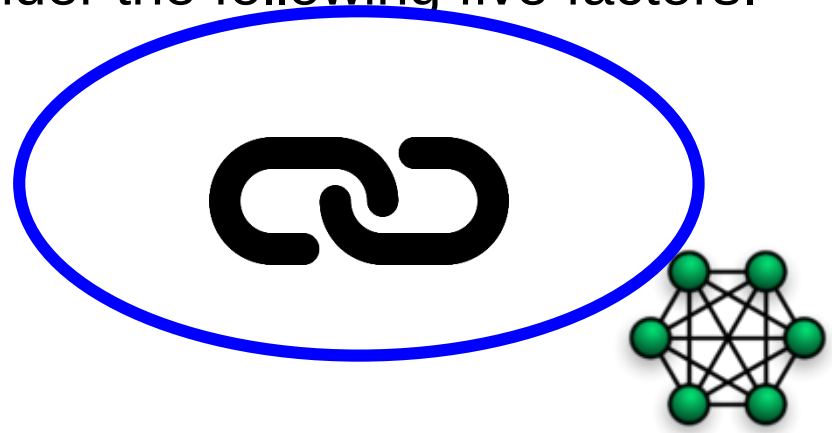
Object Decomposition

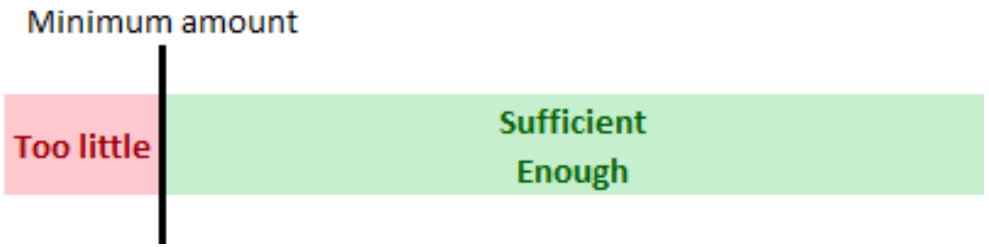


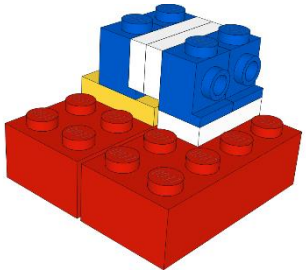
Quality of Abstraction

- How can determine if our class and object structure is well designed? Consider the following five factors.

1. Coupling
2. Cohesion
3. Sufficiency
4. Completeness
5. Primitiveness







Coupling

Strong

Implies a strong connection or dependencies between classes. We may not always want this. To maximize reuse classes should have a weak coupling so that they can be used independent of other classes.

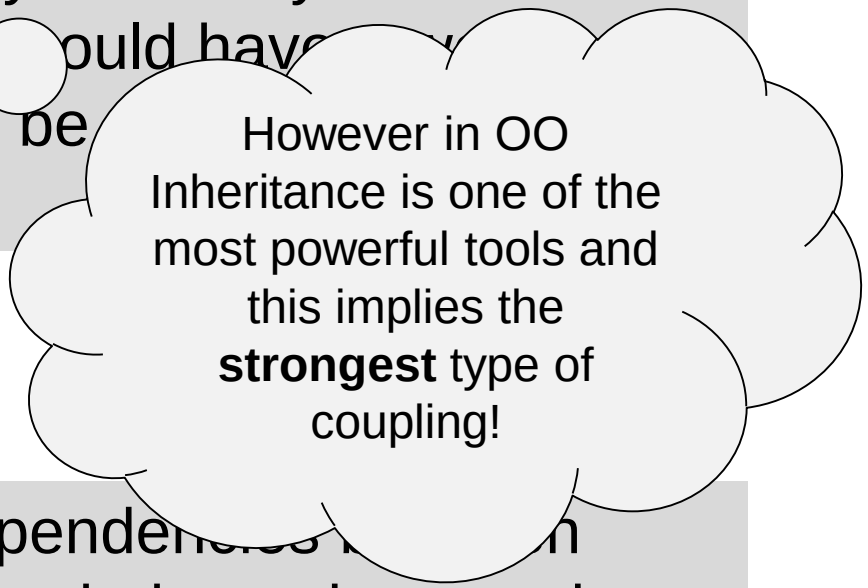
Weak

Implies minimal if any dependencies between classes. Classes which are independent can be used as building blocks to form new programs.

Coupling

Strong

Implies a strong connection or dependencies between classes. We may not always want this. To maximize reuse classes should have weak coupling so that they can be used by other classes.



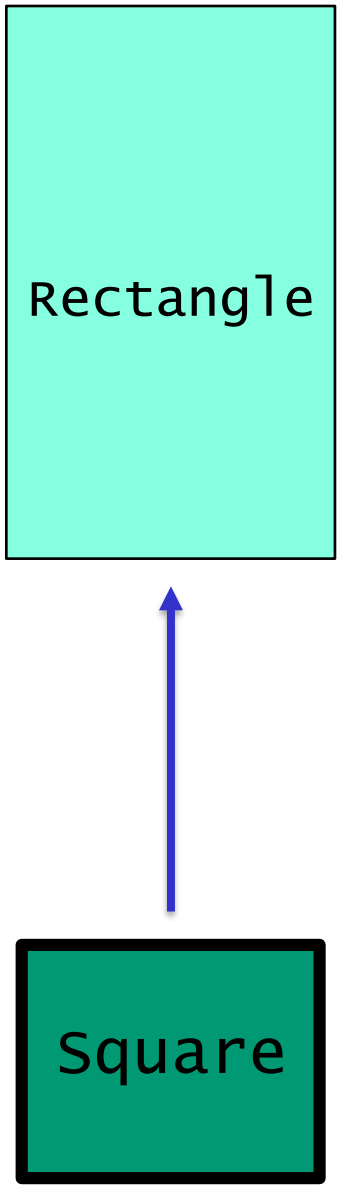
Weak

Implies minimal if any dependencies between classes. Classes which are independent can be used as building blocks to form new programs.

Using Inheritance

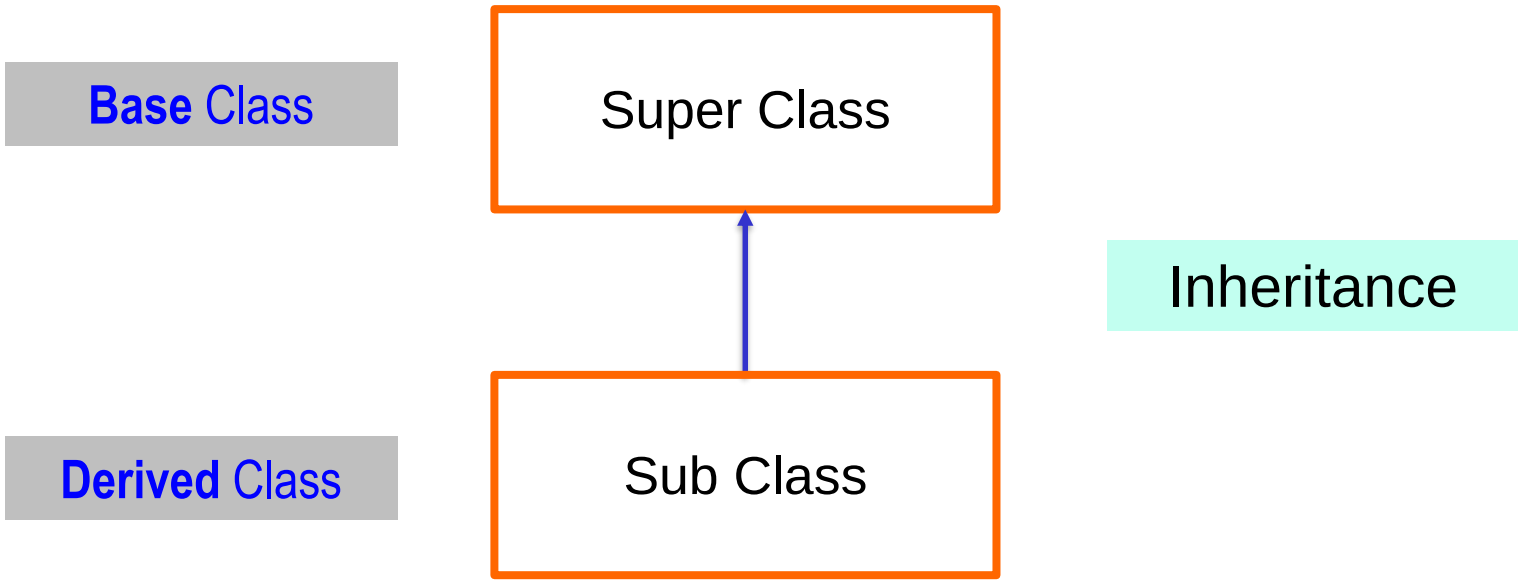
```
public class Rectangle {
    private int width;
    private int height;

    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
}
```



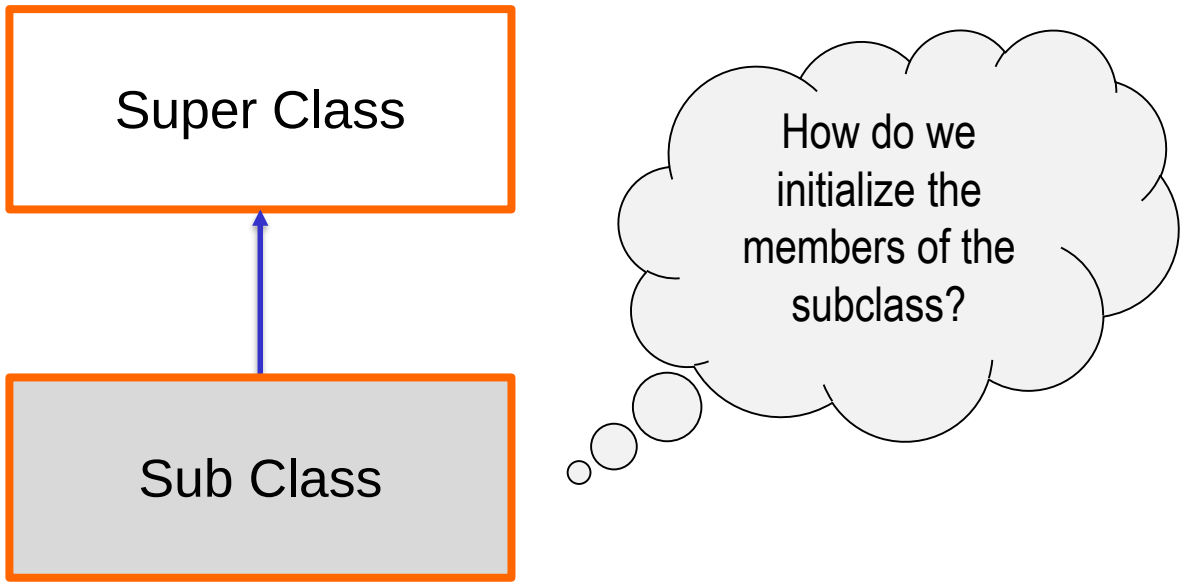
Using Inheritance

- `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

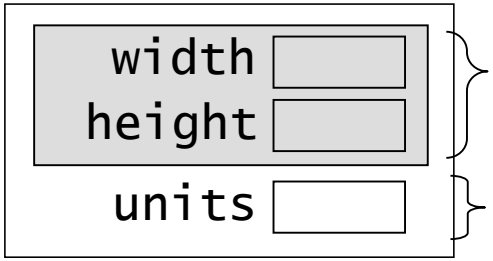
- `Square` *inherits* all of the fields and methods of `Rectangle`.
  - we don't need to redefine them!
- `Square` is a *subclass* of `Rectangle`.
- `Rectangle` is a *superclass* of `Square`.



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

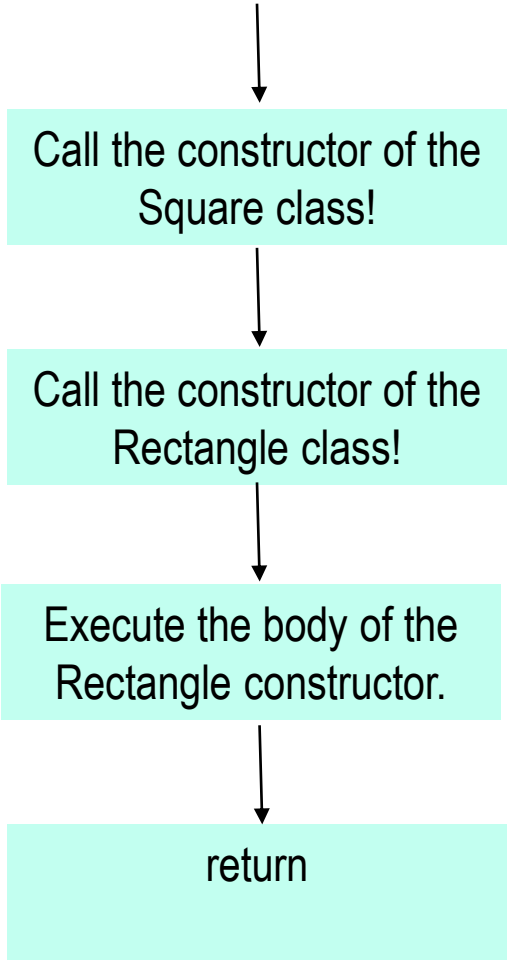
- Change the modifier in the super class from private to protected.

- 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.

Constructor Chaining

Square r = new **Square(10, “cm”);**



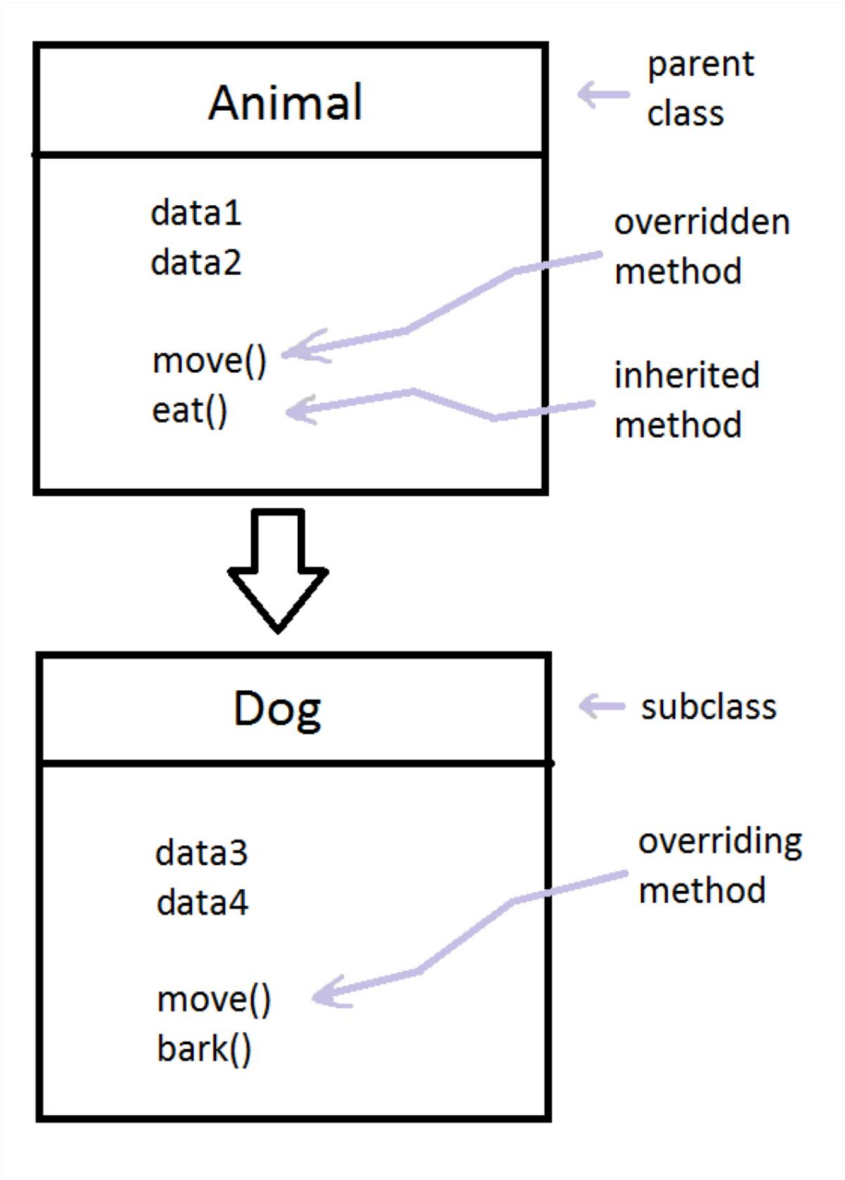
Constructor Chaining

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);
```

An Inherited Method:

toString()

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

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

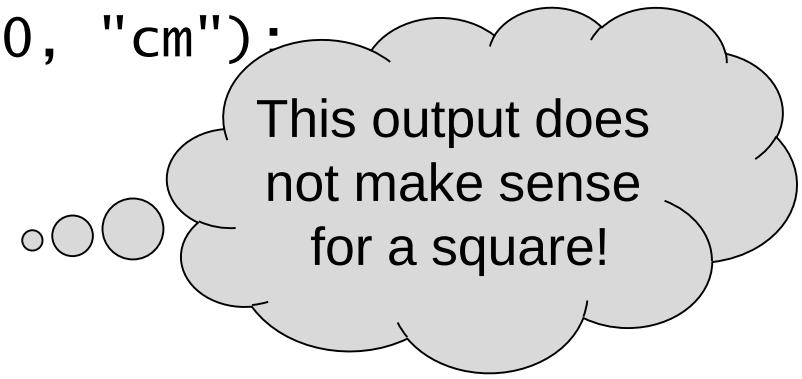
- The `Square` class inherits it from `Rectangle`.

- 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);
```

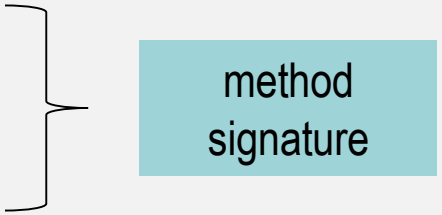
output:

```
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

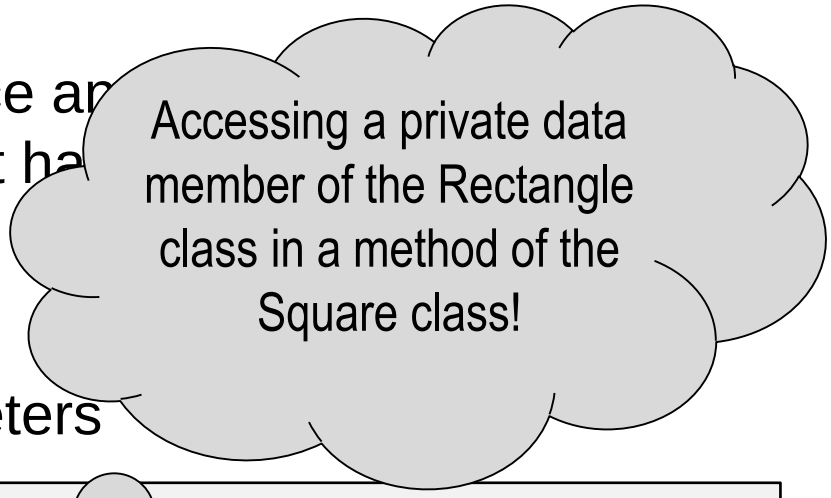
- A subclass can *override* / replace an inherited method with its own version, which must have the same:
  - return type
  - name
  - number and types of parameters

- 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
  with its own version, which must ha
  - return type
  - name
  - number and types of parameters

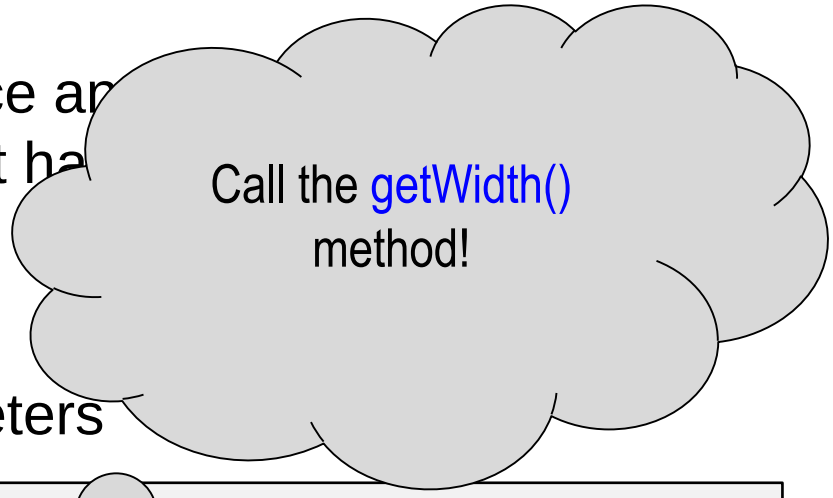


- Example: our `Square` class can define its own `toString()`:

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

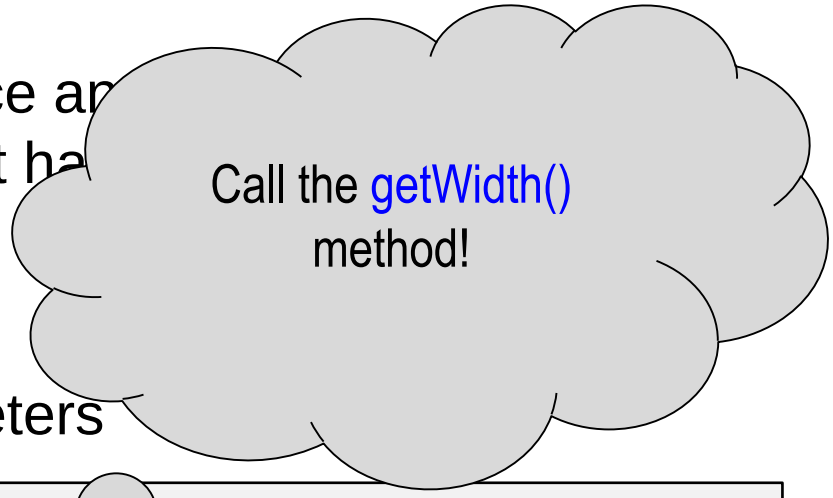


- Example: our `Square` class can define its own `toString()`:

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

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

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

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,
    if (dw != dh) {
        throw new Illega
    }
    super.setWidth(this.getWidth() + dw);
    super.setHeight(this.getHeight() + dh);
}
```

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

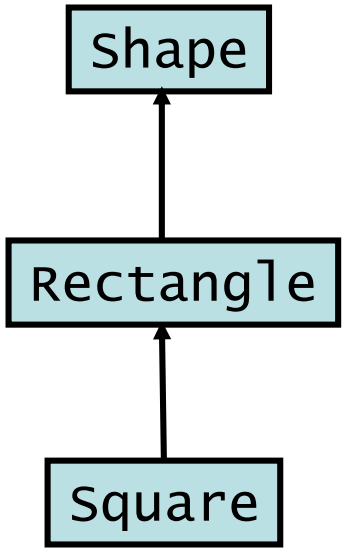
Accessing Methods from the Superclass

- 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*:

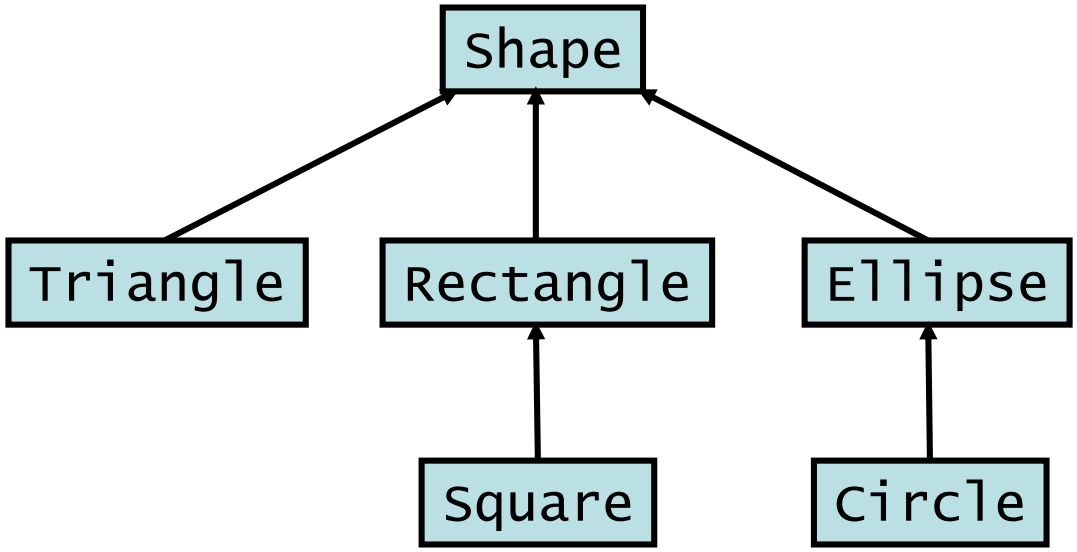



```
public class Shape {
    // fields and methods
    // common to all shapes
    ...
}
```

```
public class Rectangle
extends Shape {

    ...

}
```

Inheritance Hierarchy

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




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

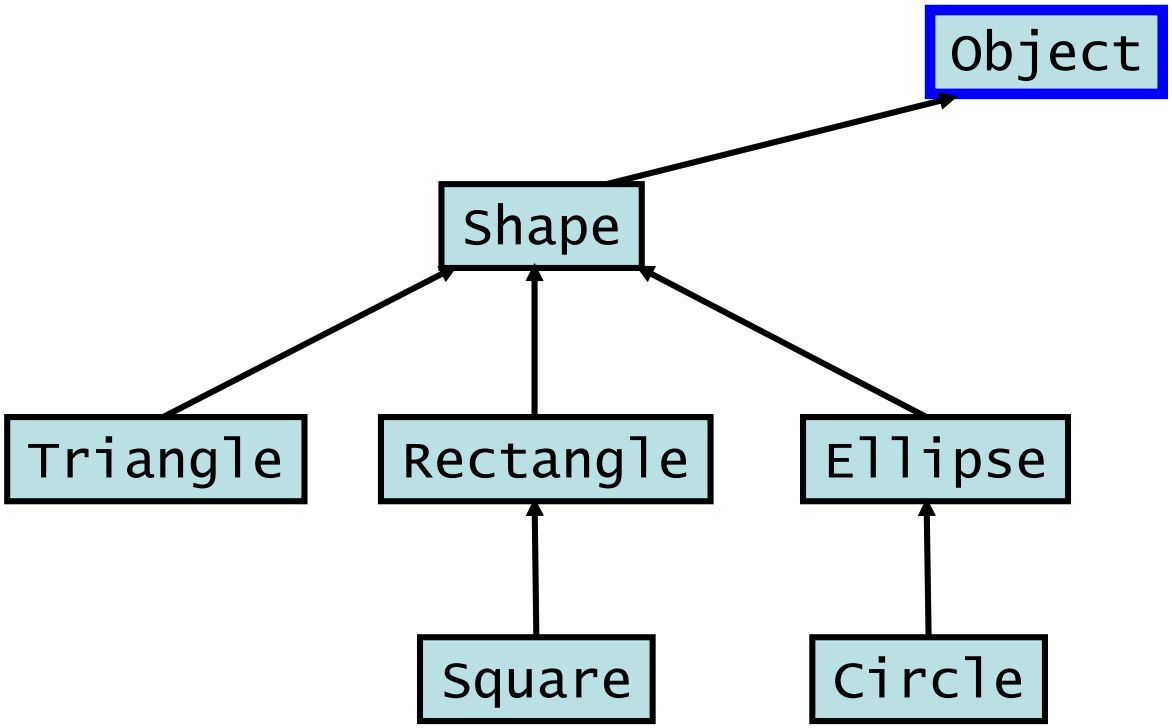
```
public class Rectangle
extends Shape {

    ...

}
```

- 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`.

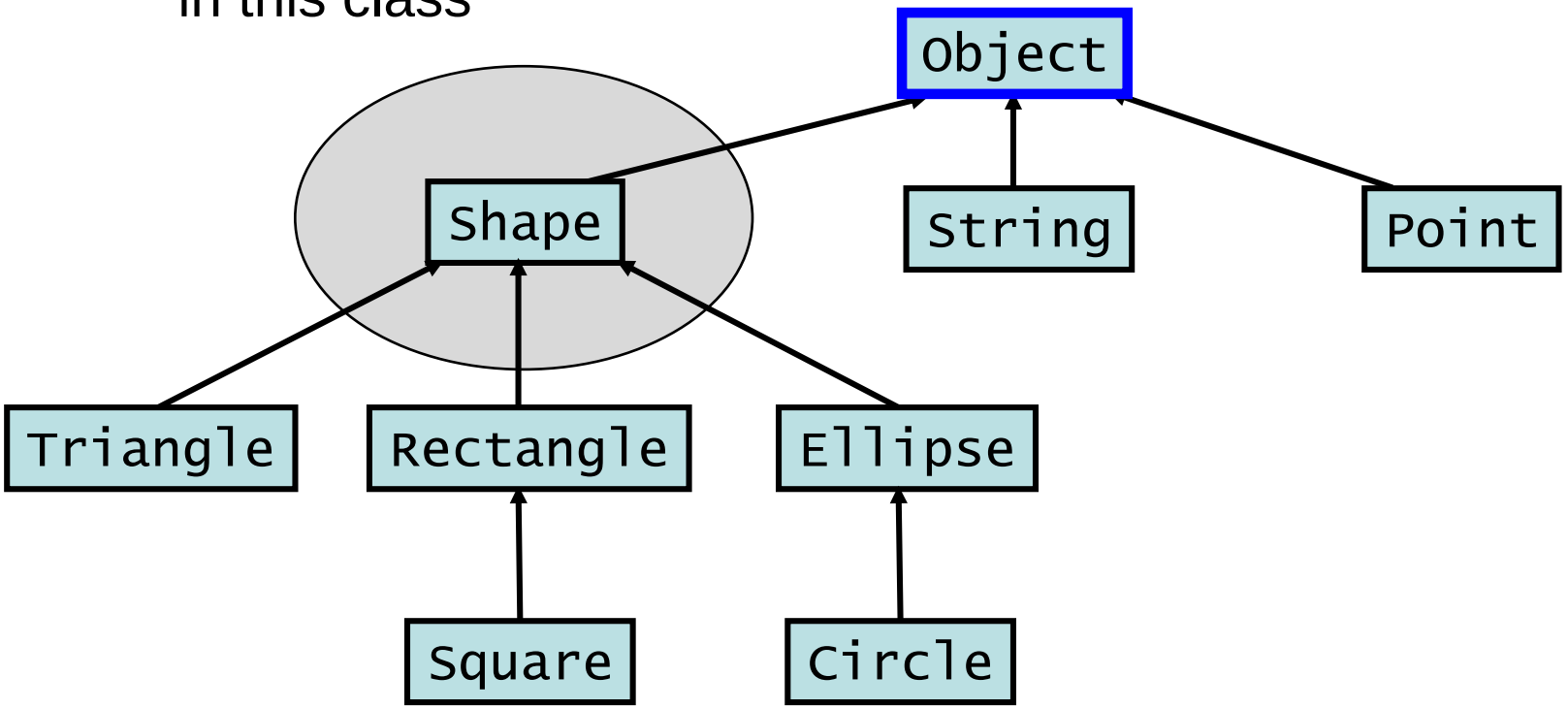
```
public class Shape {
    // fields and methods
    // common to all shapes
    ...
}
```



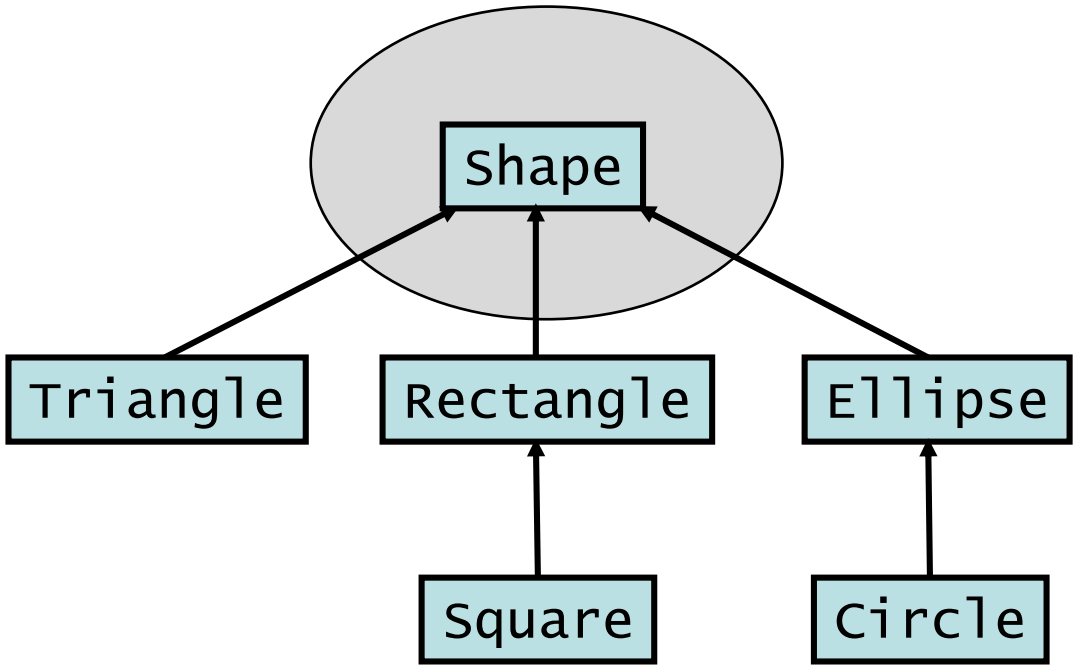
The `Object` Class

- 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

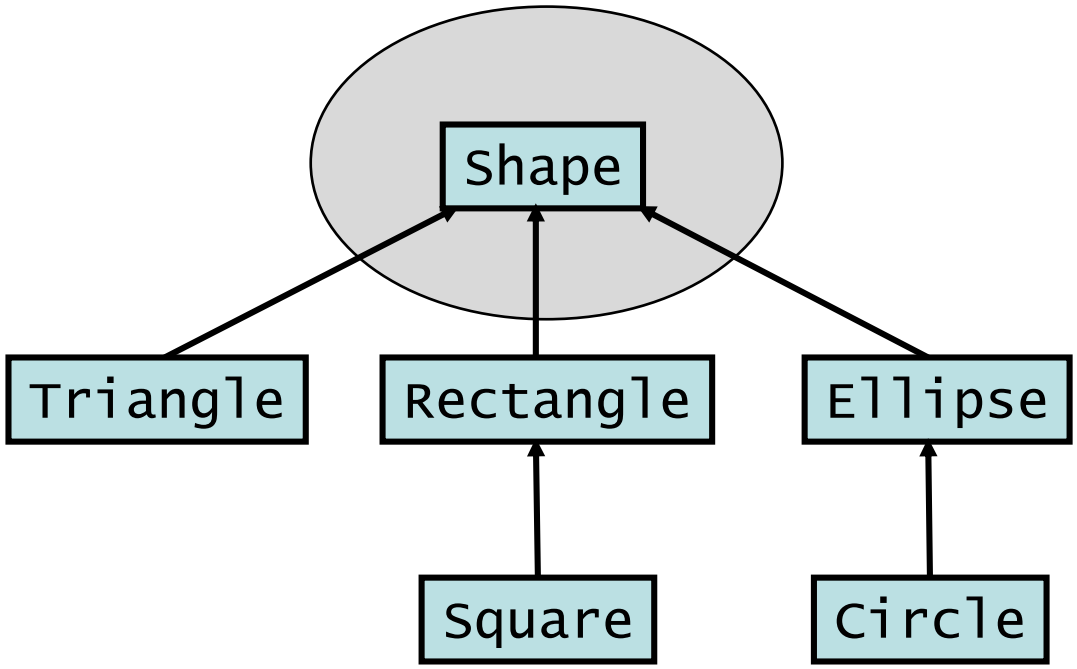



```
public class Shape {
    // fields and methods
    // common to all shapes
    ...
}
```

```
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!

Abstract Classes

```
public class Shape {
    // members common to all shapes
    String shapeName;
    Point p;
    Color c;

    // 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 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 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 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 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 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 );
    }
    public double area() {
        ?
    }
}
```

Abstract Classes

```
public 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();



}
```

Abstract Classes

```
public 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();
    abstract public double perimeter(); ?


}
```

Abstract Classes

```
public 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();
    abstract public void draw();


}
```

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 m
    }
    Shape( String name ) {
        this();                 // in
        shapeName = name;
    }
    // methods common to all shape
    public String toString() {
        return( shapeName );
    }
    abstract public double area();
    abstract public void draw();


}
```

Once we declare a method of a class to be abstract, we must make the class an abstract class!

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.

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), and must be overridden in a subclass.

- 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.

Concrete Classes

```
public class Rectangle extends Shape {
    // members common to all Rectangles
    private int width;
    private int height;

    // constructors
    Rectangle() {
        // assign default values to data members
    }
    Rectangle(int width, int height ) {
        super("Rectangle");
        setWidth(width);
        setHeight(height);
    }
    // methods common to all shapes
    public String toString() {
        return( super.toString() + width + "x" + height );
    }
    public double area() {
       return(width * height)'

    public void draw() { .. }

}
```

Concrete Classes

```
public class Rectangle extends Shape {
    // members common to all Rectangles
    private int width;
    private int height;

    // constructors
    Rectangle() {
        // assign default values to data members
    }
    Rectangle(int width, int height ) {
        super("Rectangle");
        setWidth(width);
        setHeight(height);
    }
    // methods common to all shapes
    public String toString() {
        return( super.toString() + width + "x" + height );
    }
    public double area() {
       return(width * height)'

    public void draw() { .. }

}
```

Concrete Classes

```
public class Rectangle extends Shape {
    // members common to all Rectangles
    private int width;
    private int height;

    // constructors
    Rectangle() {
        // assign default values to data members
    }
    Rectangle(int width, int height ) {
        super("Rectangle");
        setWidth(width);
        setHeight(height);
    }
    // methods common to all shapes
    public String toString() {
        return( super.toString() + " " + width + "x" + height );
    }
    public double area() {
       return(width * height)'

    public void draw() { .. }

}
```

Concrete Classes

```
public class Rectangle extends Shape {
    // members common to all Rectangles
    private int width;
    private int height;

    // constructors
    Rectangle() {
        // assign default values to data members
    }
    Rectangle(int width, int height ) {
        super("Rectangle");
        setWidth(width);
        setHeight(height);
    }
    // methods common to all shapes
    public String toString() {
        return( super.toString() + width + "x" + height );
    }
    public double area() {
        return(width * height)'

    public void draw() { .. }

}
```

Declared as **concrete** in the Abstract Super Class, and must be implemented in the concrete class.

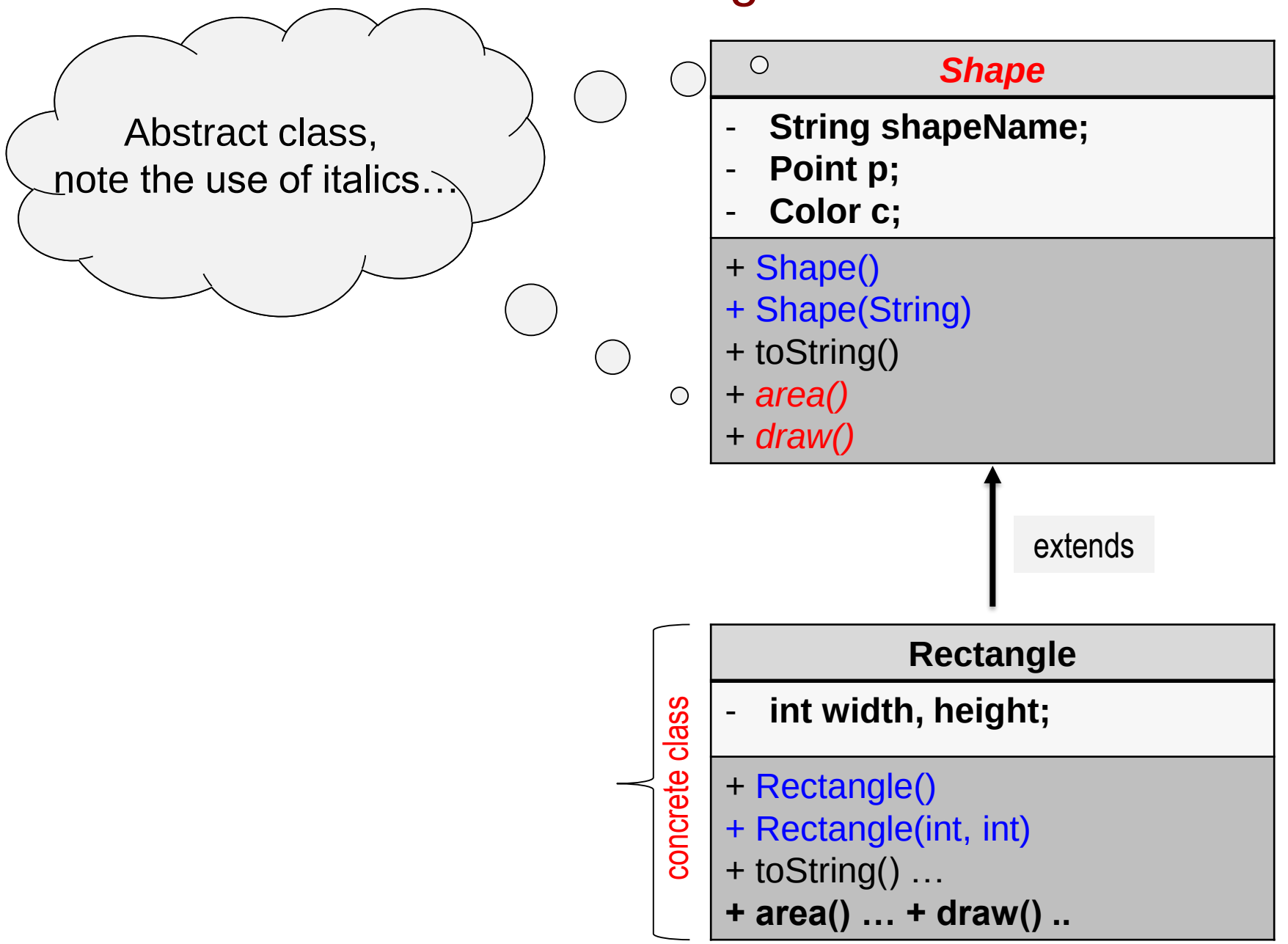
Concrete Classes

```
public class Rectangle extends Shape {
    // members common to all Rectangles
    private int width;
    private int height;

    // constructors
    Rectangle() {
        // assign default value
    }
    Rectangle(int width, int
        super("Rectangle");
        setWidth(width);
        setHeight(height);
    }
    // methods common to all shap
    public String toString()
        return( super.toS ng() + width + "x" + height );
    }
    public double area() {
        return(width * height)'

    public void draw() { .. }

}
```

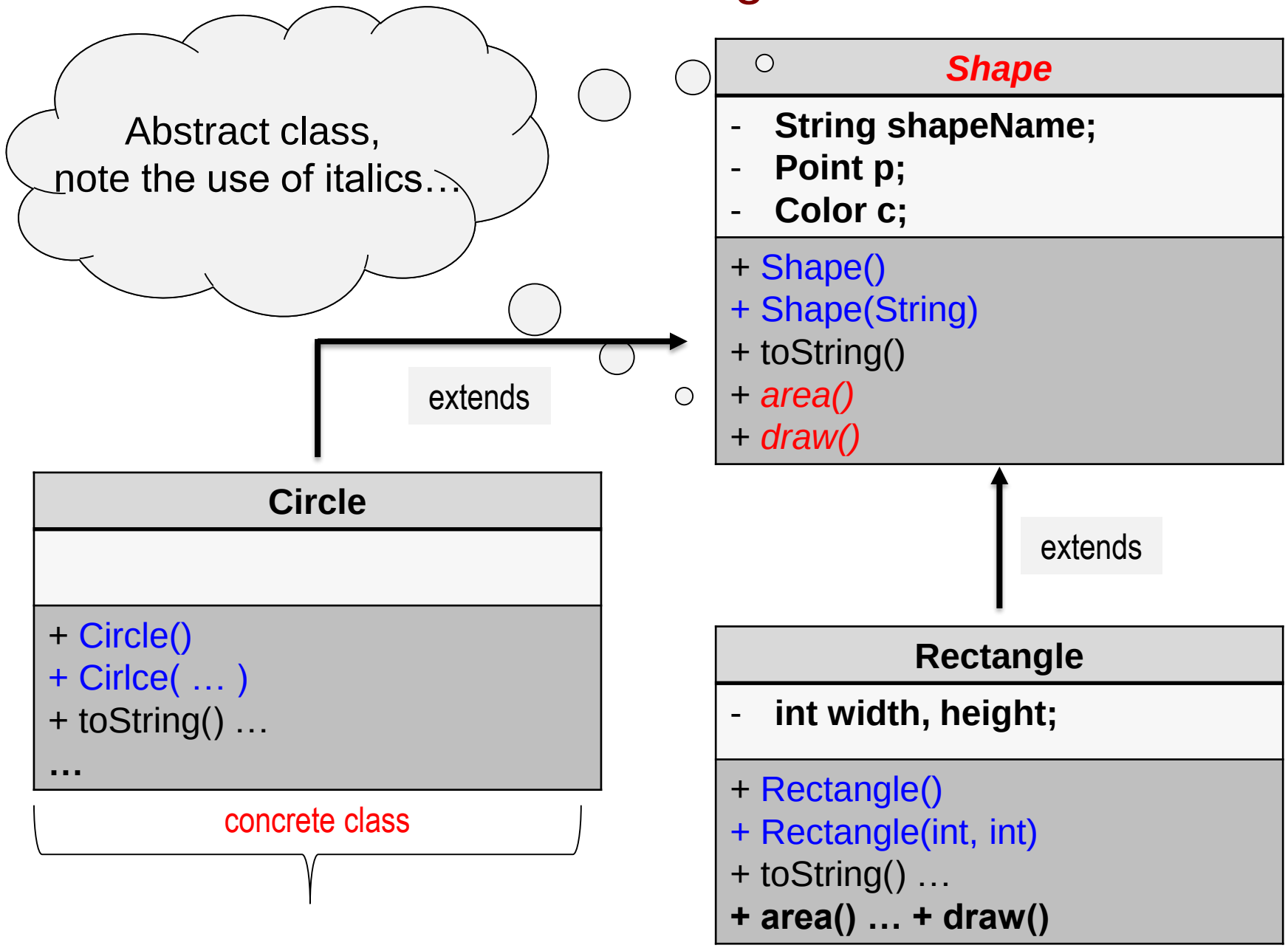
Declaring them abstract in the Super class, ensures that all **concrete** classes of Shape have these methods implemented!

Declared as **concrete** in the Abstract Super Class, and must be implemented in the concrete class.

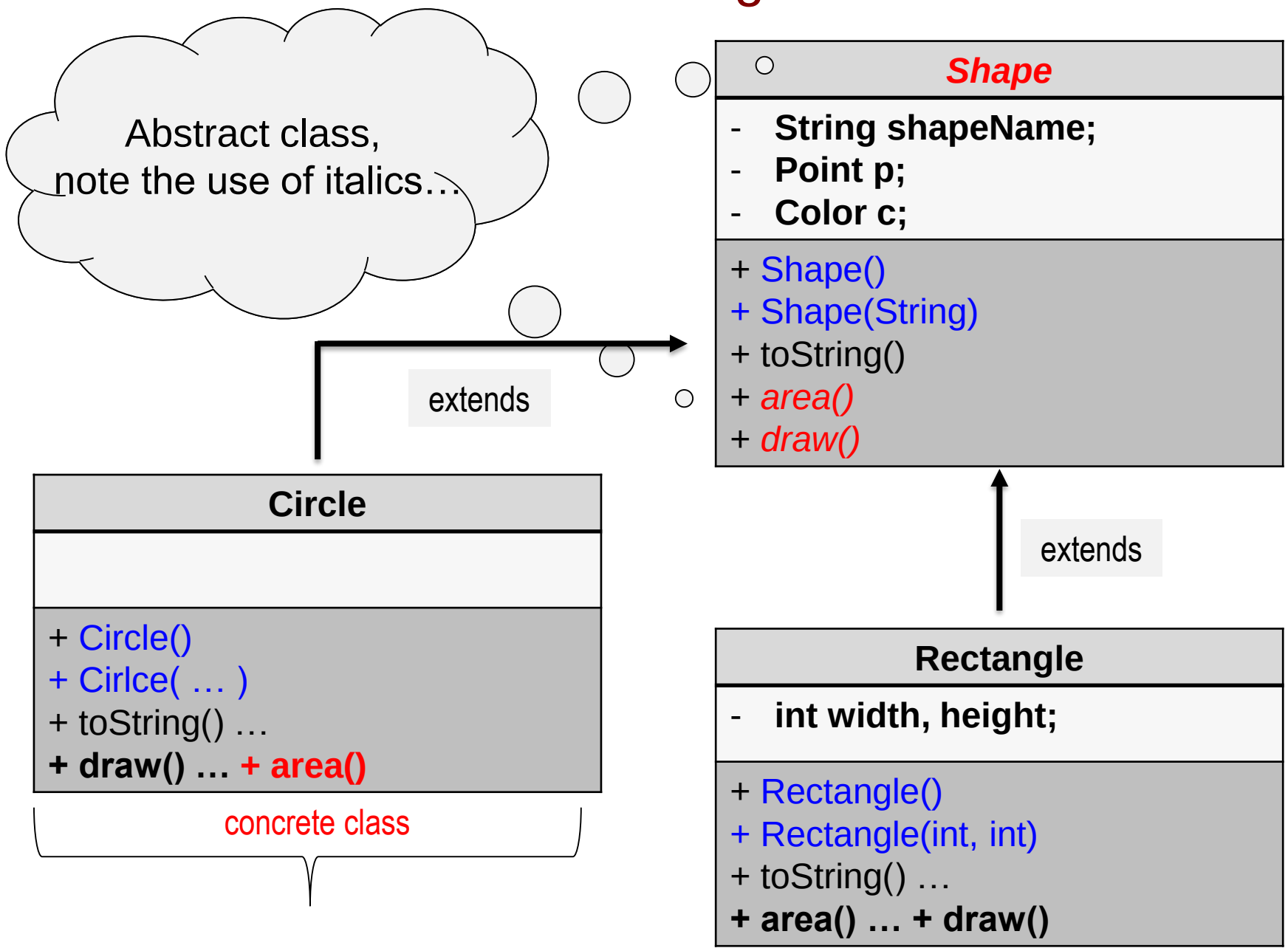
UML Diagram


Abstract class,
note the use of italics…
Shape
- String shapeName;
- Point p;
- Color c;
+ Shape()
+ Shape(String)
+ toString()
+ area()
+ draw()
extends
concrete class
Rectangle
- int width, height;
+ Rectangle()
+ Rectangle(int, int)
+ toString() …
+ area() … + draw() ..

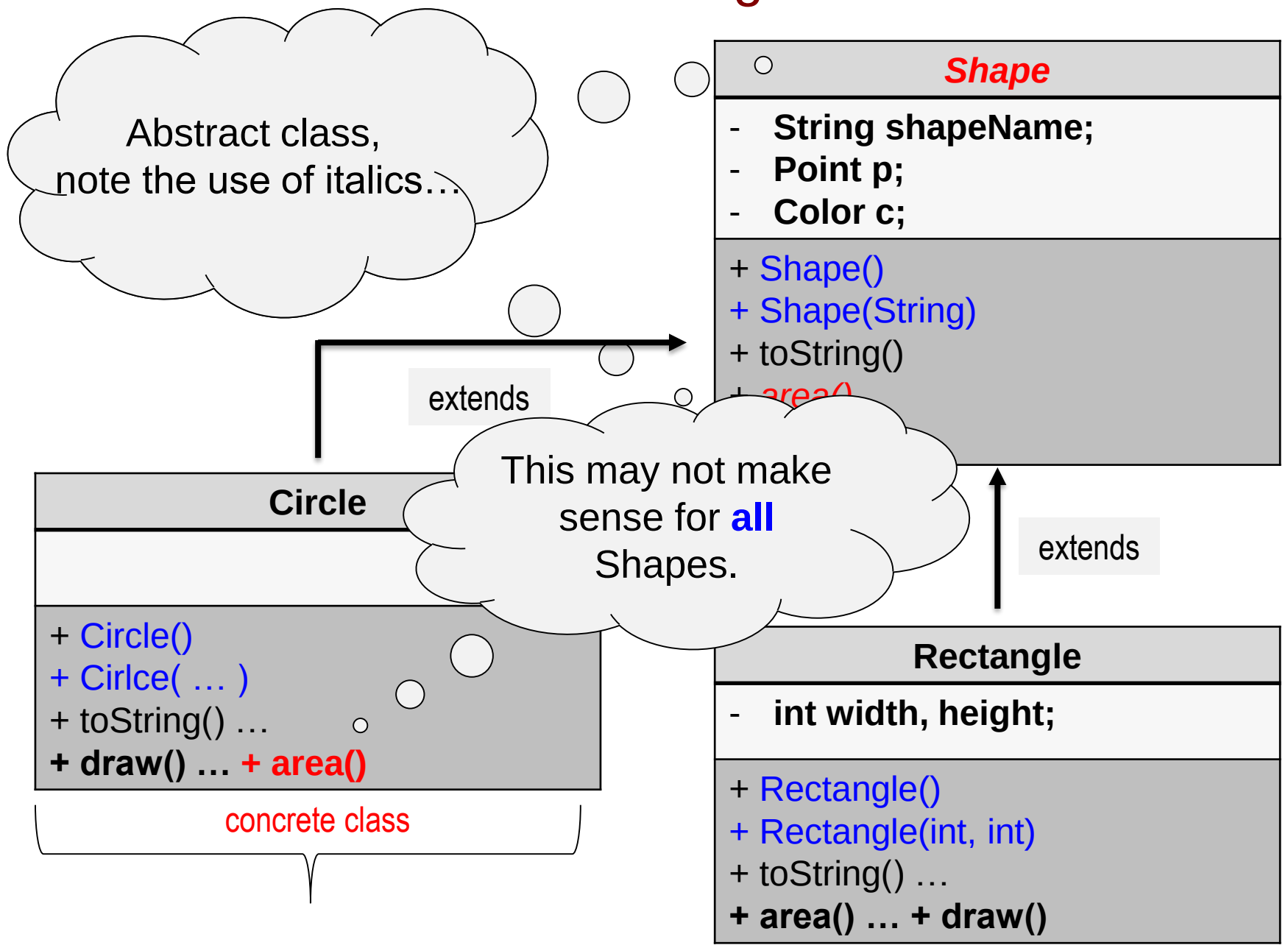
UML Diagram


Abstract class,
note the use of italics…
Shape
- String shapeName;
- Point p;
- Color c;
+ Shape()
+ Shape(String)
+ toString()
+ area()
+ draw()
extends
Circle
+ Circle()
+ Cirlce(…)
+ toString() …
…
concrete class
extends
Rectangle
- int width, height;
+ Rectangle()
+ Rectangle(int, int)
+ toString() …
+ area() … + draw()

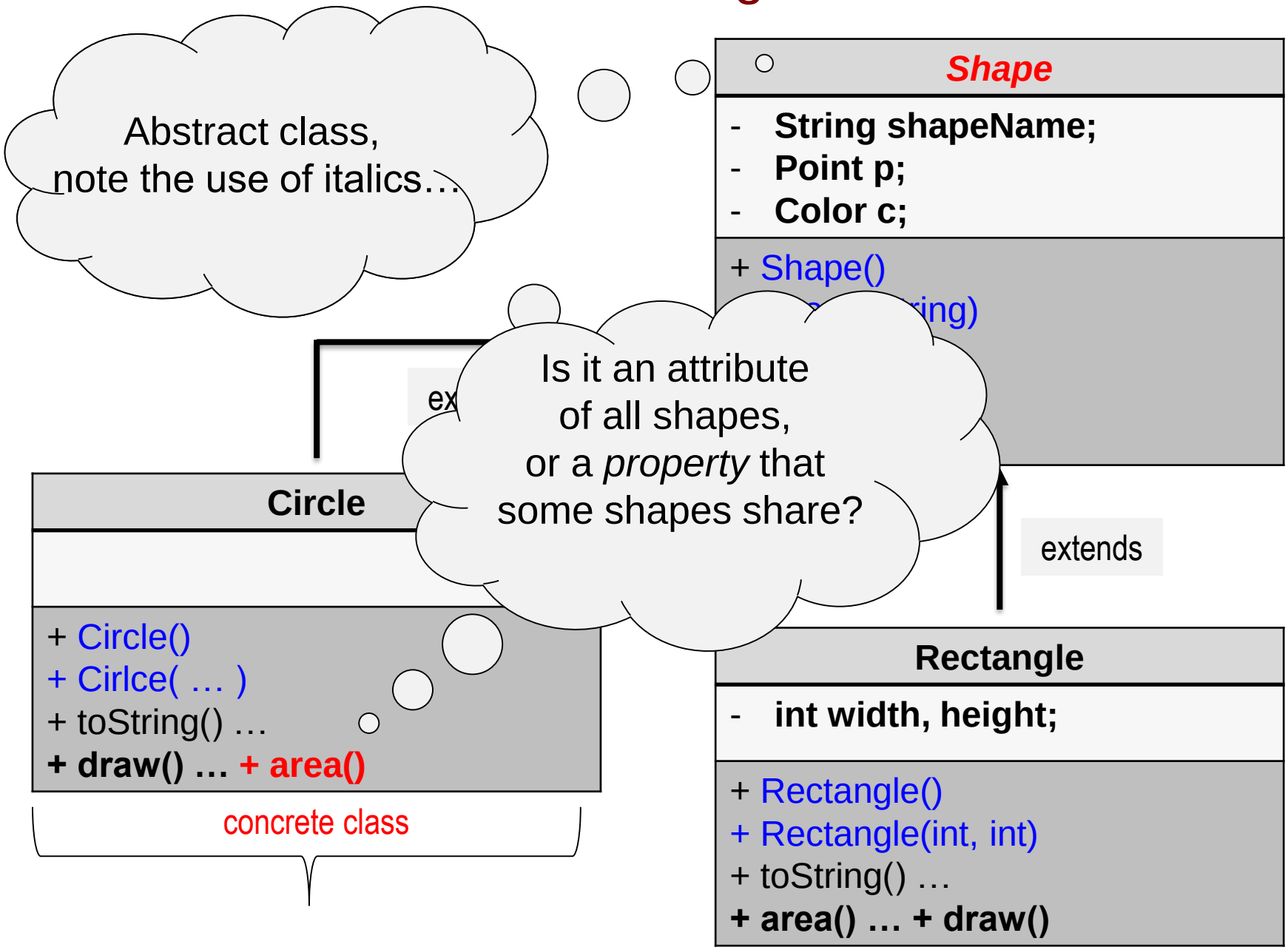
UML Diagram


Abstract class,
note the use of italics…
Shape
- String shapeName;
- Point p;
- Color c;
+ Shape()
+ Shape(String)
+ toString()
+ area()
+ draw()
extends
Circle
+ Circle()
+ Cirlce(…)
+ toString() …
+ draw() … + area()
concrete class
extends
Rectangle
- int width, height;
+ Rectangle()
+ Rectangle(int, int)
+ toString() …
+ area() … + draw()

UML Diagram


Abstract class,
note the use of italics…
Shape
- String shapeName;
- Point p;
- Color c;
+ Shape()
+ Shape(String)
+ toString()
+ area()
extends
This may not make sense for all Shapes.
Circle
+ Circle()
+ Cirlce(…)
+ toString() …
+ draw() … + area()
concrete class
extends
Rectangle
- int width, height;
+ Rectangle()
+ Rectangle(int, int)
+ toString() …
+ area() … + draw()

UML Diagram


Abstract class,
note the use of italics…
Shape
- String shapeName;
- Point p;
- Color c;
+ Shape()
ring)
Is it an attribute
of all shapes,
or a property that
some shapes share?
ex
extends
Circle
+ Circle()
+ Cirlce(…)
+ toString() …
+ draw() … + area()
concrete class
Rectangle
- int width, height;
+ Rectangle()
+ Rectangle(int, int)
+ toString() …
+ area() … + draw()

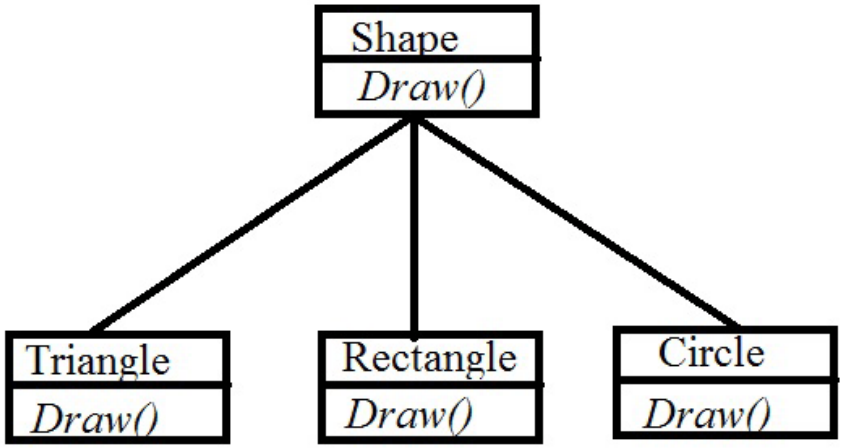
Sometimes we don’t want a class to be extended...

- 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!

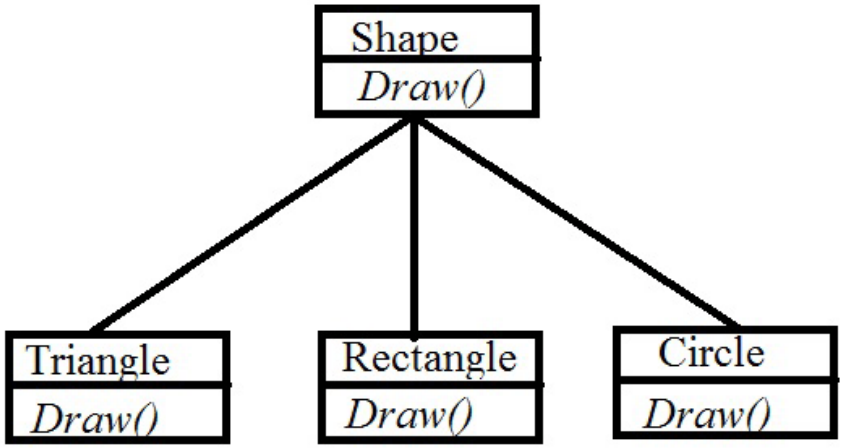
Polymorphism



Principle of Polymorphism

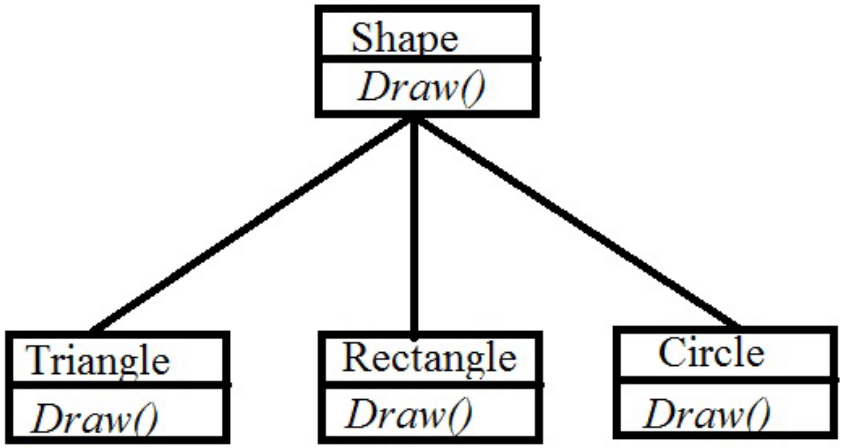
Polymorphism

- Recall that an instance of a subclass is an instance of the superclass!



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
- dynamic polymorphism

Polymorphism

There are two types of Polymorphism:

- static polymorphism → method overloading
- dynamic polymorphism

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

The signature of the method allows the compiler to identify which method is to be called and to bind the call with that method at compile time.

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.

The signature of the method allows the compiler to identify which method is to be called and to bind the call with that method at compile time.

Dynamic Polymorphism is what allows subclasses to override methods written in the superclass. Dynamic or run-time polymorphism binds the call to the method during run-time.

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!

Polymorphism and Collections of Objects

- Polymorphism is useful when we have a collection of objects of different but related types.

- Example:
  - let's say that you need a collection of different shapes:
  - we can store all of them in an array of type `Shape`:

```
Shape[] myShapes = new Shape[5];
myShapes[0] = new Rectangle(20, 30);
myShapes[1] = new Square(50, "cm");
myShapes[2] = new Triangle(10, 8);
myShapes[3] = new Circle(10);
myShapes[4] = new Rectangle(50, 100);
```

Processing a Collection of Objects

- We can print out a description of each shape as follows:

```
Shape[] myShapes = new Shape[5];
myShapes[0] = new Rectangle(20, 30);
myShapes[1] = new Square(50, "cm");
myShapes[2] = new Triangle(10, 8);
myShapes[3] = new Circle(10);
myShapes[4] = new Rectangle(50, 100);

for (int i = 0; i < myShapes.length; i++) {
    System.out.println(myShapes[i]);
}
```

- For each element of the array, the appropriate `toString()` method is called!
  - `myShapes[0]`: the `Rectangle` version of `toString()` is called
  - `myShapes[1]`: the `Square` version of `toString()` is called
  - etc.

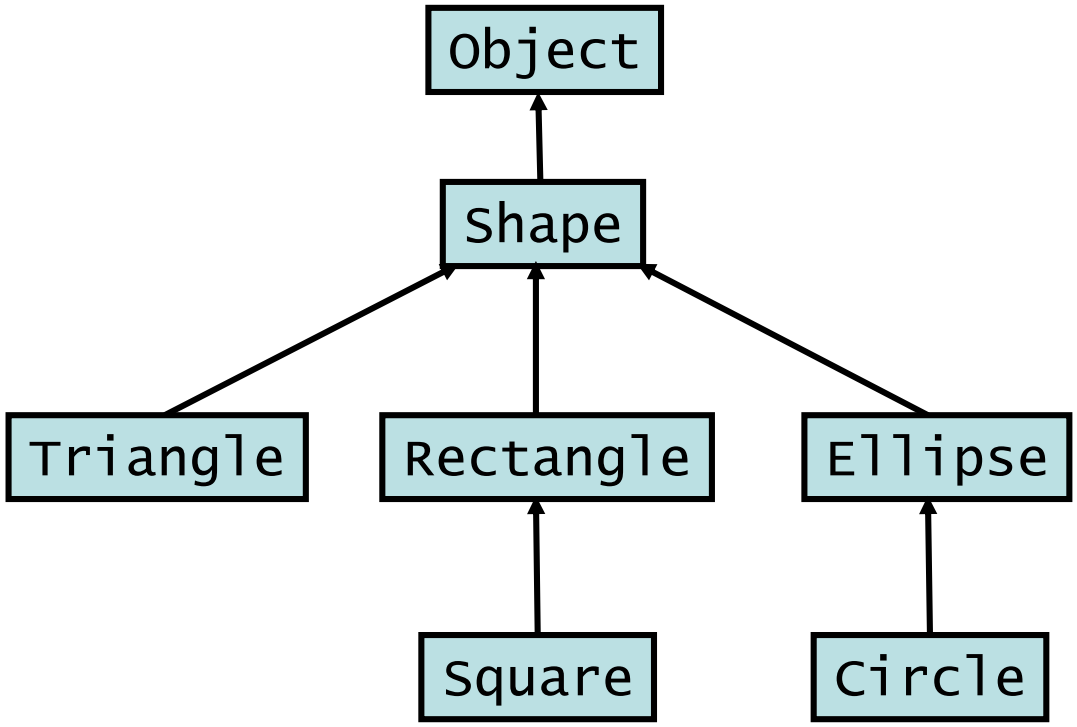
Processing a Collection of Objects

- We can print out a description of each shape as follows:

```
Shape[] myShapes = new Shape[5];
myShapes[0] = new Rectangle(20, 30);
myShapes[1] = new Square(50, "cm");
myShapes[2] = new Triangle(10, 8);
myShapes[3] = new Circle(10);
myShapes[4] = new Rectangle(50, 100);

for (int i = 0; i < myShapes.length; i++) {
    System.out.println(myShapes[i]);
}
```

- For each element of the array, the appropriate `toString()` method is called!
  - `myShapes[0]`: the `Rectangle` version of `toString()` is called
  - `myShapes[1]`: the `Square` version of `toString()` is called
  - etc.

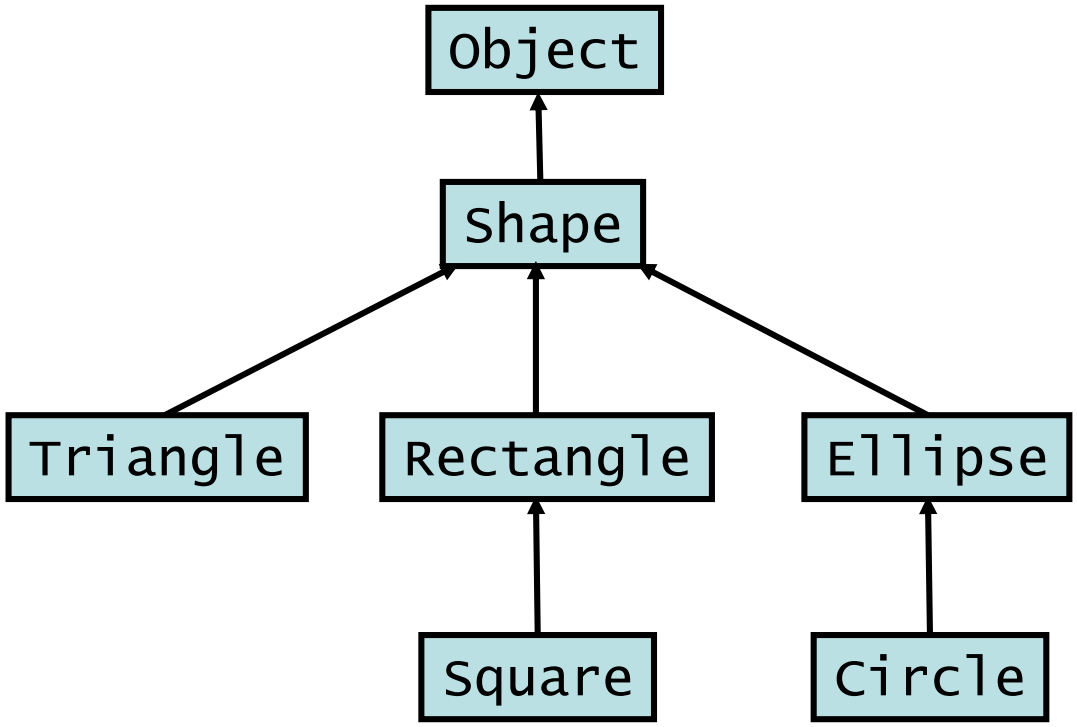
Practice with Polymorphism



- 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

Program to an Interface,

not to an Implementation