# *Object Oriented Design and Analysis*
# *CPE 372*

## Lecture 3

## Inheritance

*Dr. Sally E. Goldin*
*Department of Computer Engineering*
*King Mongkut's University of Technology Thonburi*
*Bangkok, Thailand*

Latest update: 25 Jan 2020

# Review – Encapsulation


Expose the minimum amount of information about data format and algorithm implementation

Minimize ***coupling*** between classes

Maximize ***cohesion*** within each class

# What is cohesion?

"*In computer programming, cohesion refers to the degree to which the elements inside a module belong together. ... Cohesion is often contrasted with coupling, a different concept. High cohesion often correlates with loose coupling, and vice versa.*"

https://en.wikipedia.org/wiki/Cohesion_(computer_science)

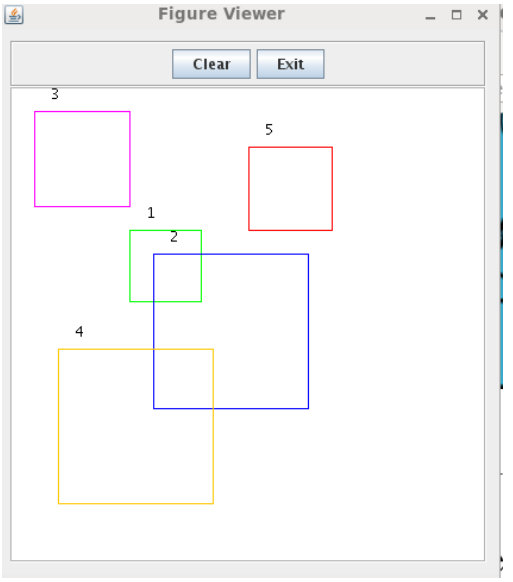
To cohere means to stick together, e.g. "add water to the flour/sugar mixture until the cookie dough coheres."

# Visibility in Java

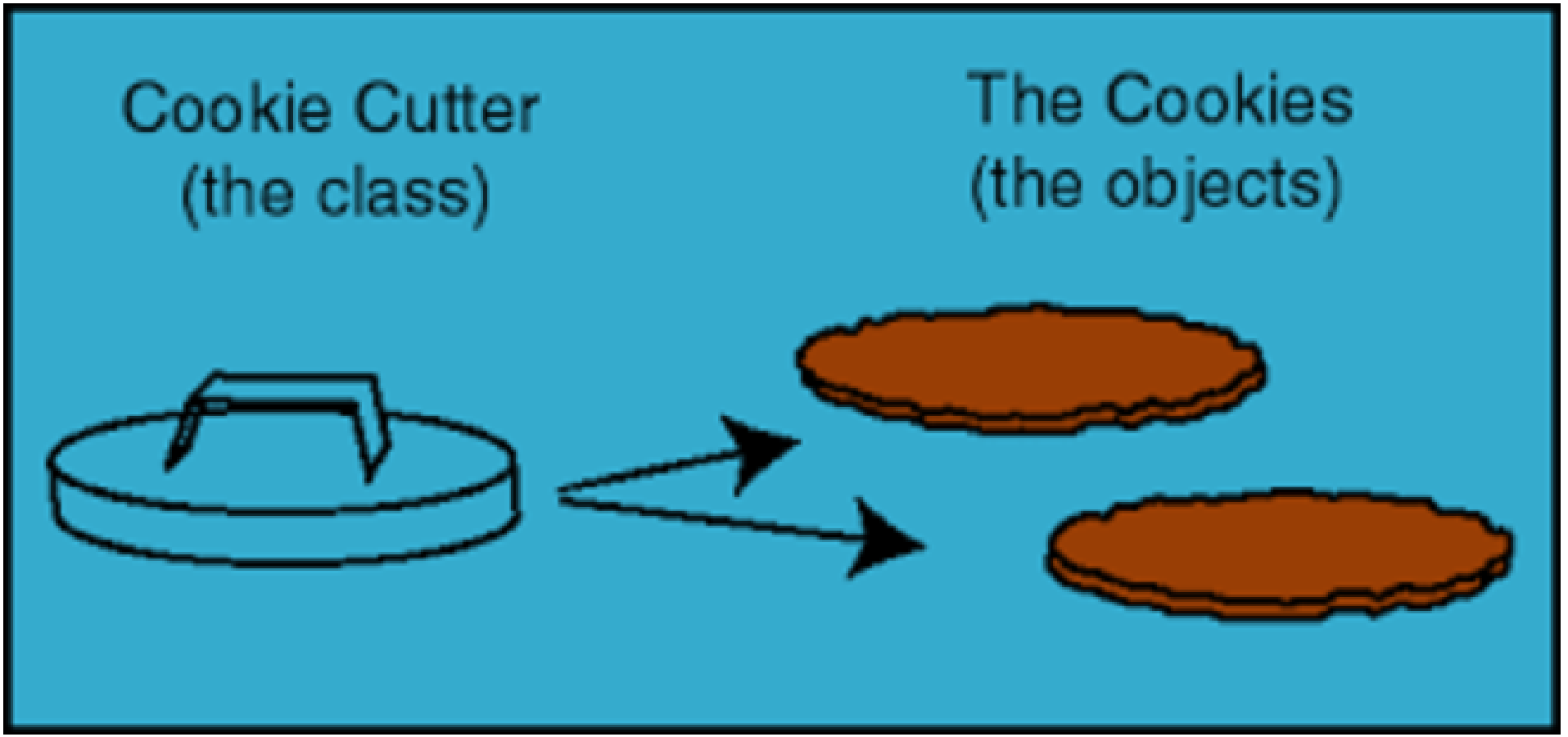
Visibility modifiers restrict who can see and use data and methods: *private*, *public*, *protected* or default

General rule: **Use the most restrictive visibility that will allow your system to work correctly**

Exercise 2 involved reducing coupling and increasing cohesion.



# Classes and Instances



# Instance versus Static Data & Methods

Instance data

```
xcoord[], ycoord[], oneside
drawColor, figureNumber
```

Instance methods

```
move(), draw(), calcPerimeter(), calcArea()
```

Class (static) data

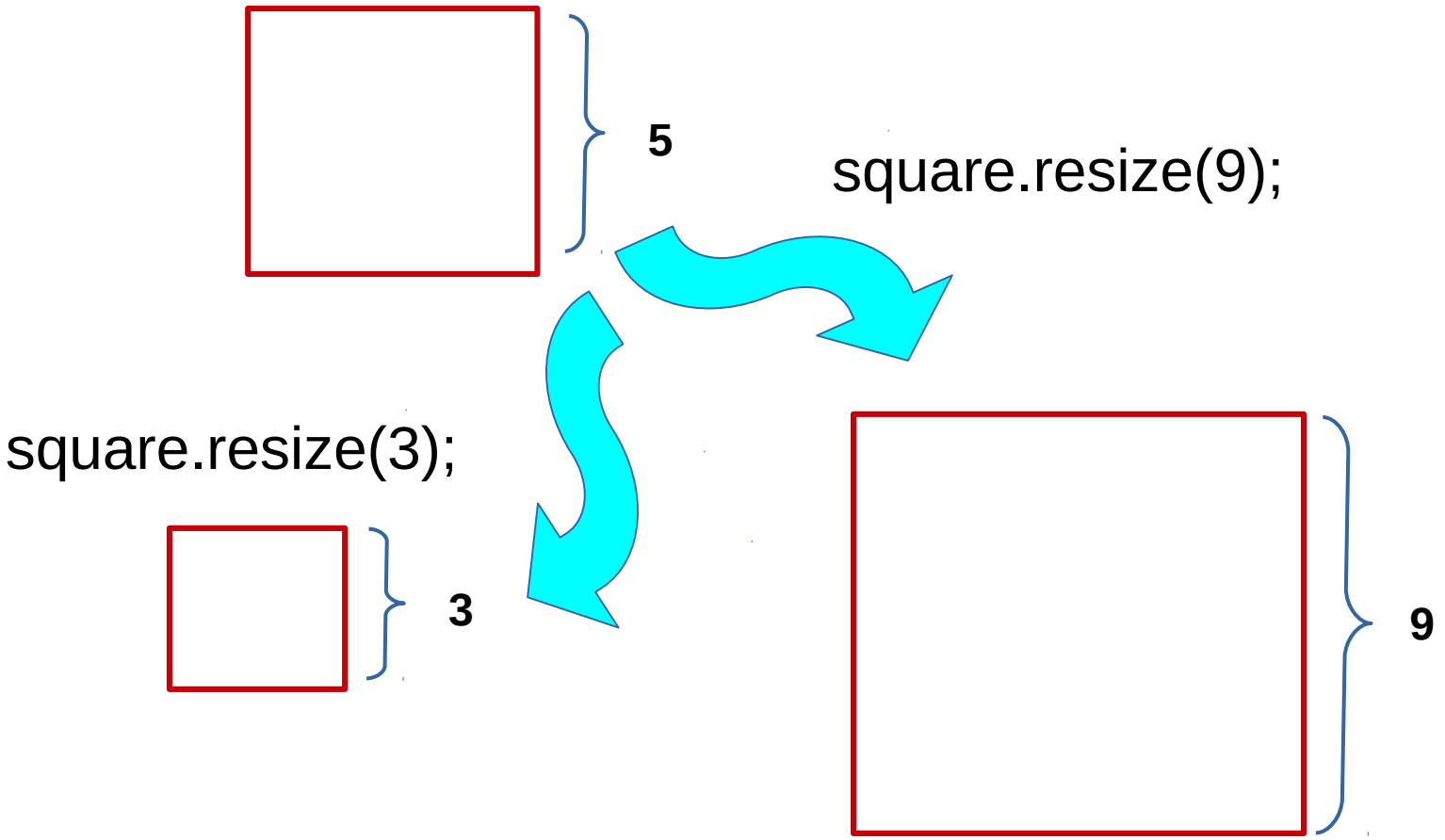
```
counter, allFigures, colors
```

Class (static) methods

```
drawAll()
```

# New class: AdjustableSquare

Like *Square* but can be made bigger or smaller



# Implementing AdjustableSquare

We could just add a `resize()` method to ***Square***

*But suppose we want some squares to not be adjustable*

We could copy Square.java to AdjustableSquare.java, then add the new method

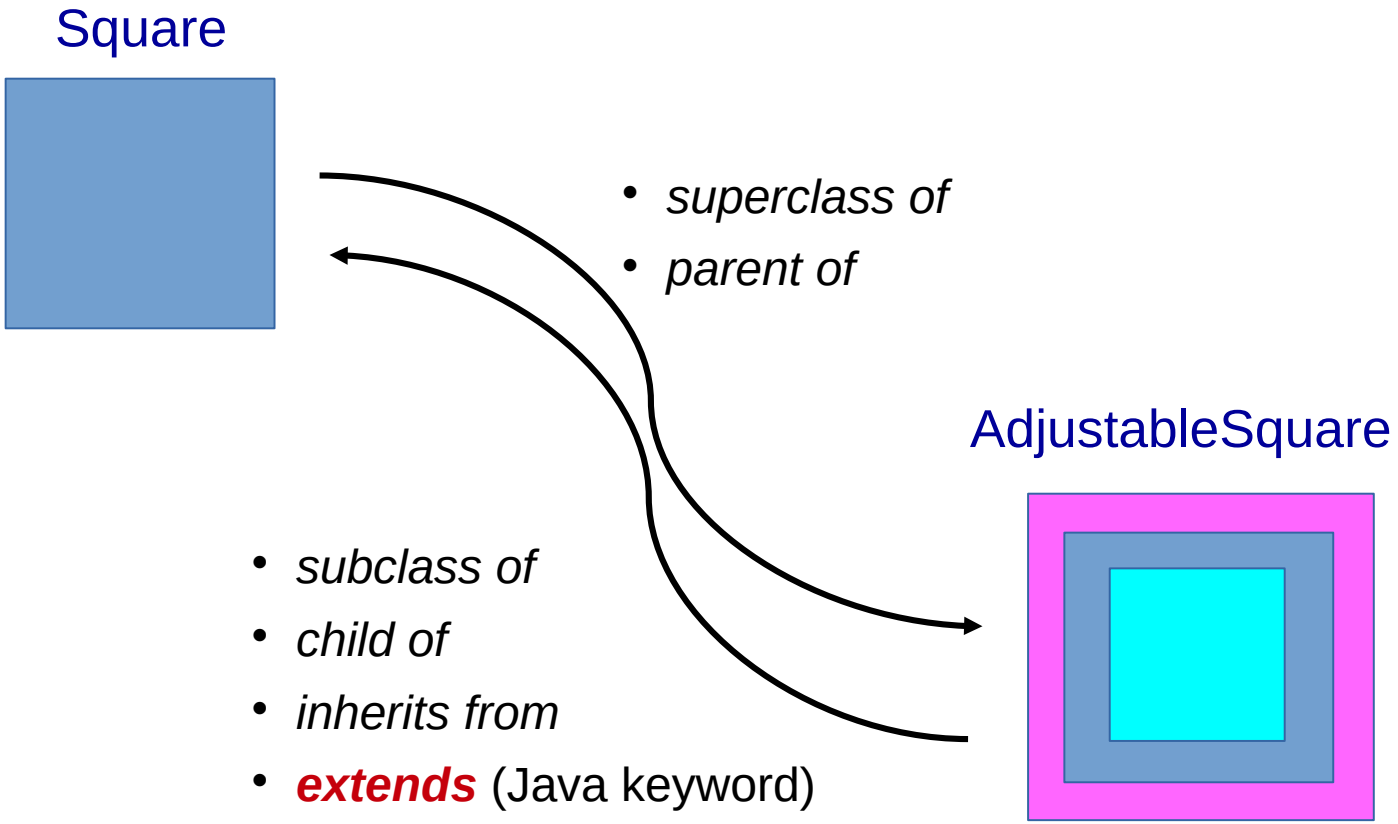
*But this duplicates a lot of code*

*If we change one class, need to change in the other, too*

Is there a better strategy?

# Subclasses

OOD solution: make *AdjustableSquare* a **subclass** of *Square*



# Inheritance

*AdjustableSquare* automatically has all the methods and data of *Square*

We say that the subclass *inherits* (สืบทอด) the methods and members from the superclass

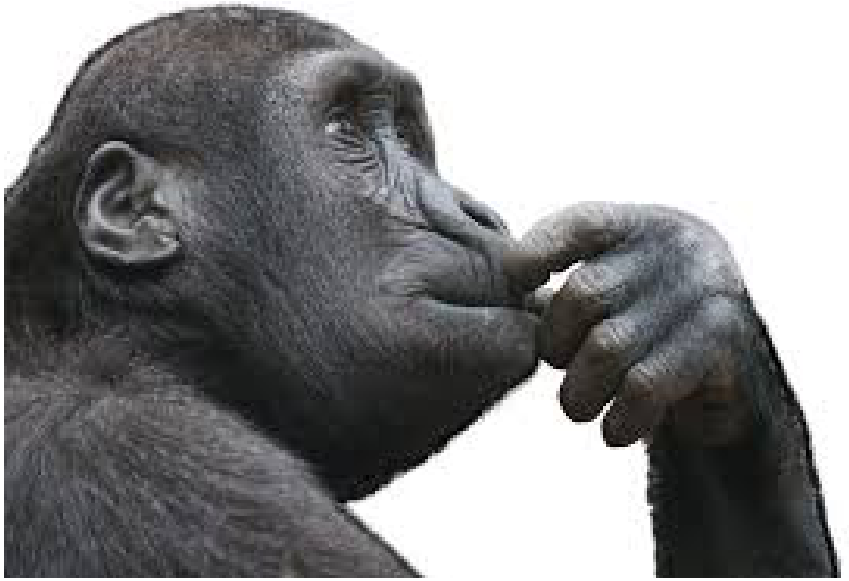
All we need to add are the methods (or data items) that are special for *AdjustableSquare*

**We can create AdjustableSquareTesterGraphics.java and use AdjustableSquare with almost no code changes.**

# Implementing `resize()`

We want to implement the `resize()` method that is the main difference between Square and AdjustableSquare

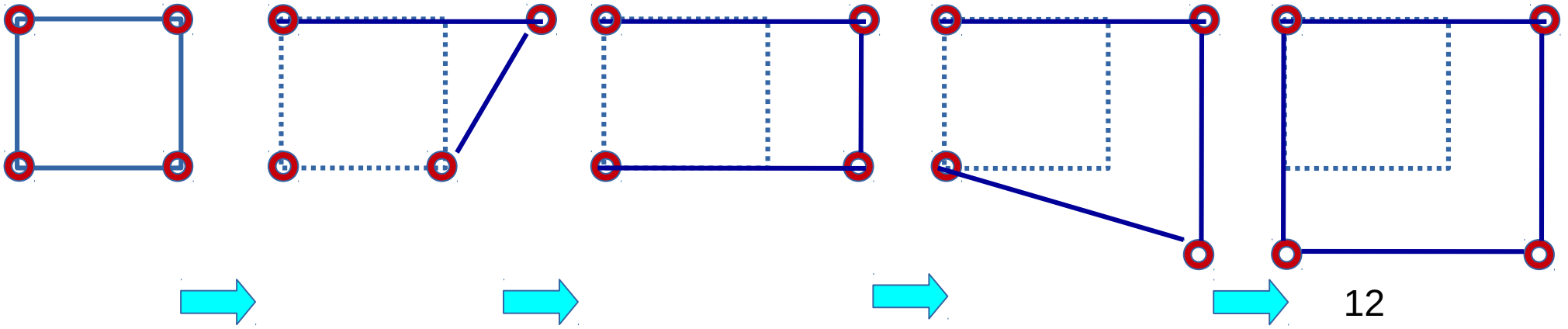
How should we do this? What is the logic?



# Pseudocode

resize(newsize):

set *oneside* to be *newsize*
set upper right X to be upper left X plus newsize
set lower right X to be upper left X plus newsize
set lower left Y to be upper left Y plus newsize
set lower right Y to be upper left Y plus newsize



# Let’s try it!


What
Went
Wrong?

# Remember *protected* visibility?

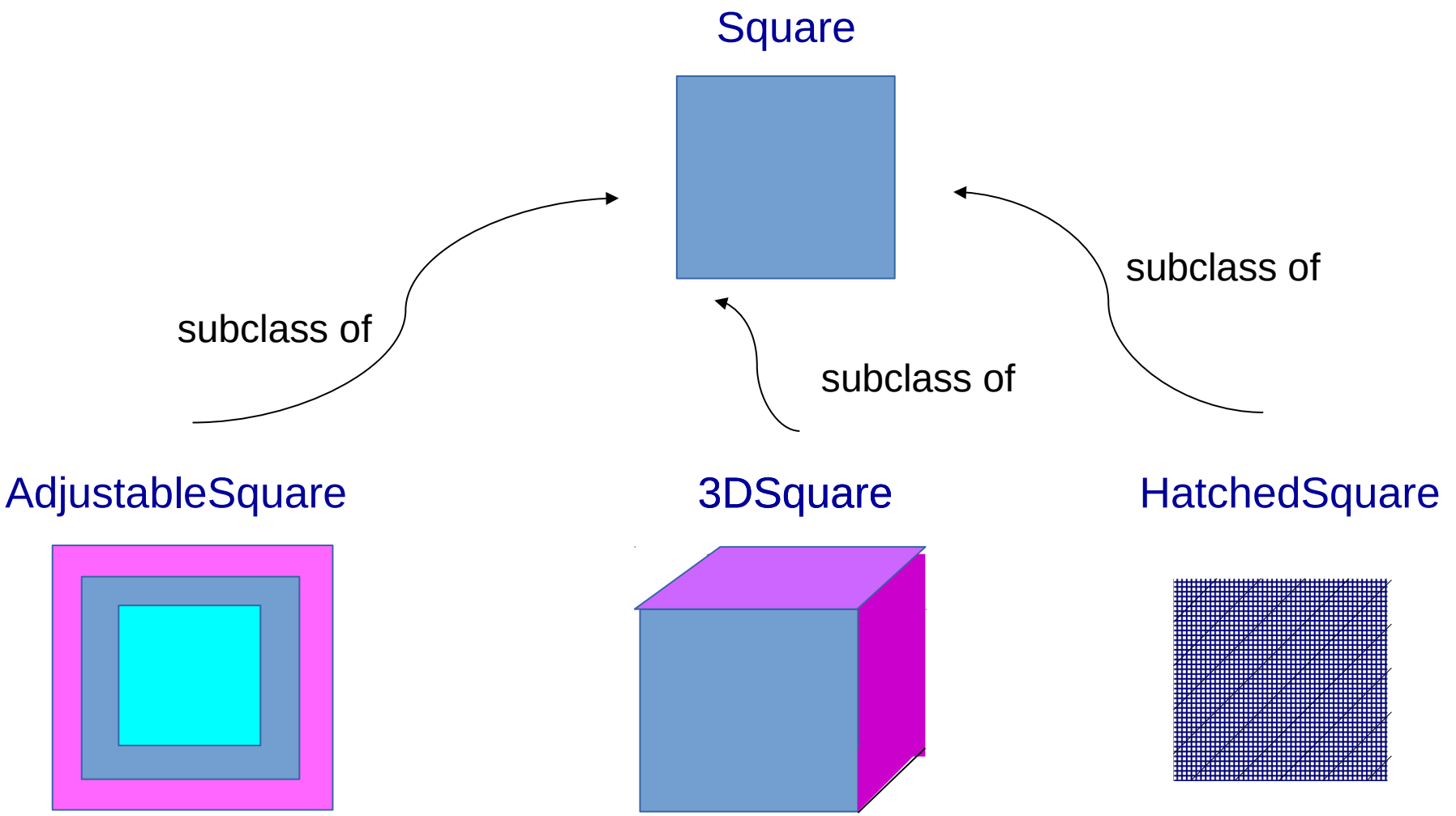
This situation shows why we need ***protected*** visibility

*AdjustableSquare* needs to modify some of the member data items that are declared in its parent, *Square*

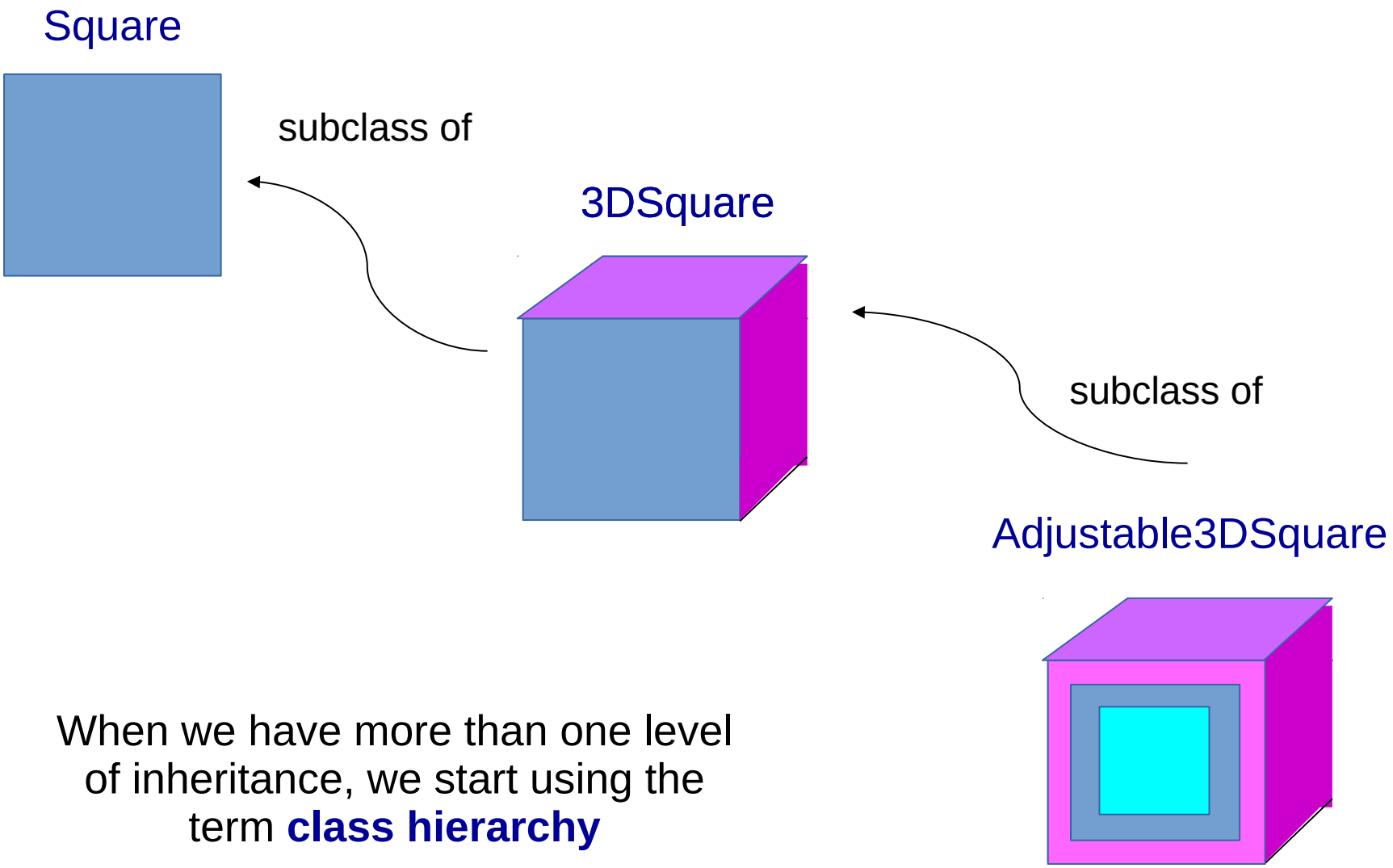
However, those members are declared as ***private***

If we make them protected, subclasses can access them, but no other classes can!

# One class can have many subclasses



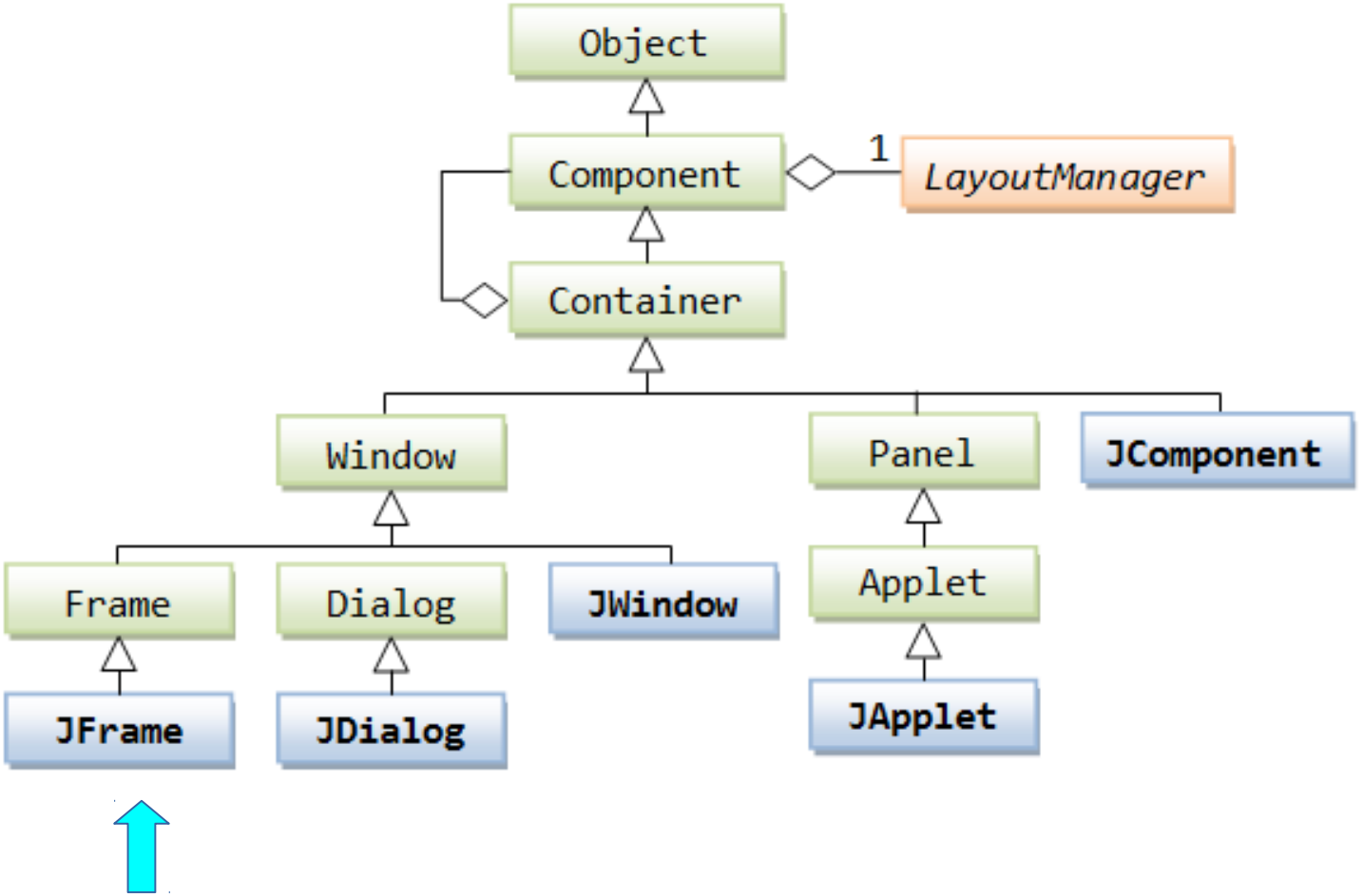
# A subclass can have subclasses




When we have more than one level of inheritance, we start using the term **class hierarchy**

# Example Class Hierarchy

This is part of the class hierarchy for the Java GUI package called Swing




Our ***FigureViewer*** class extends ***JFrame***

# Packages

Java groups “related” classes into ***packages***

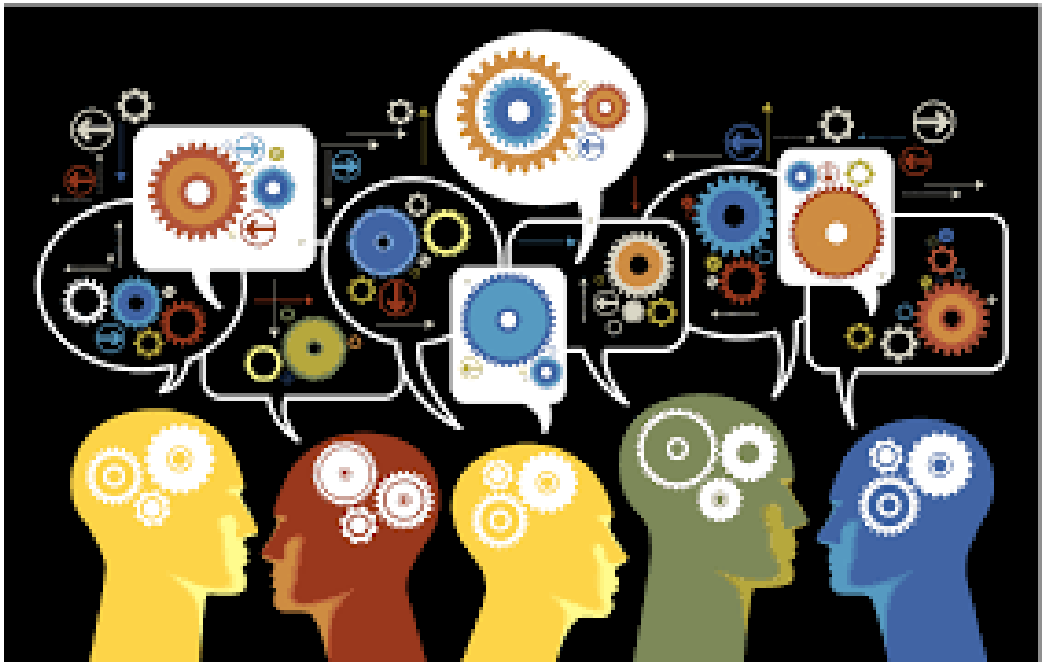
In the previous slide, the greenish classes belong to the package `java.awt`, while the blue classes are in the package `javax.swing`

If you want to use some class or package from the Java class libraries you must import it.

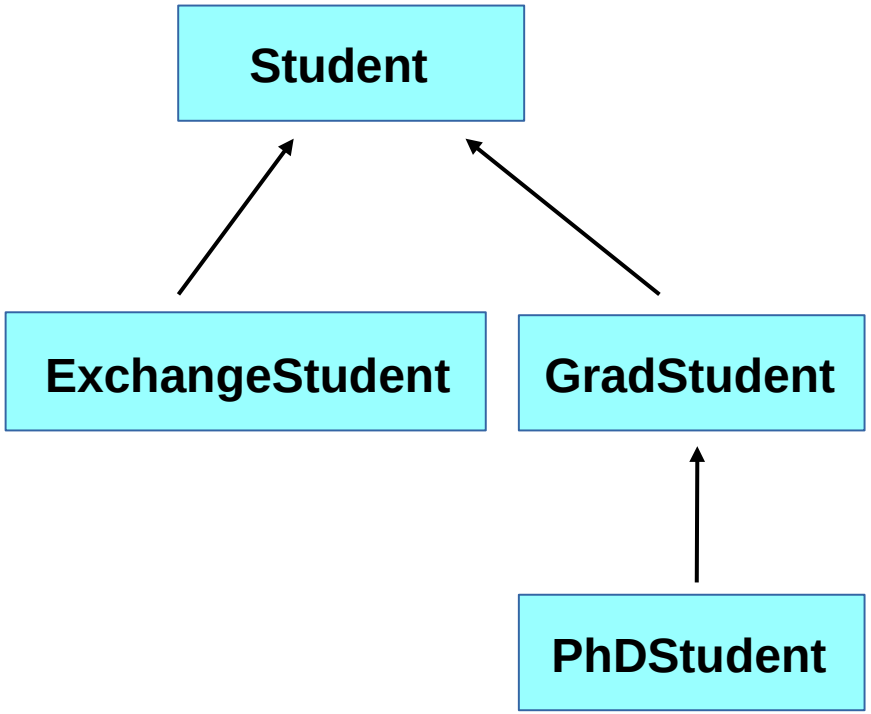
You can and should create your own packages to help manage the complexity of your code.

# Core Skills in OOAD

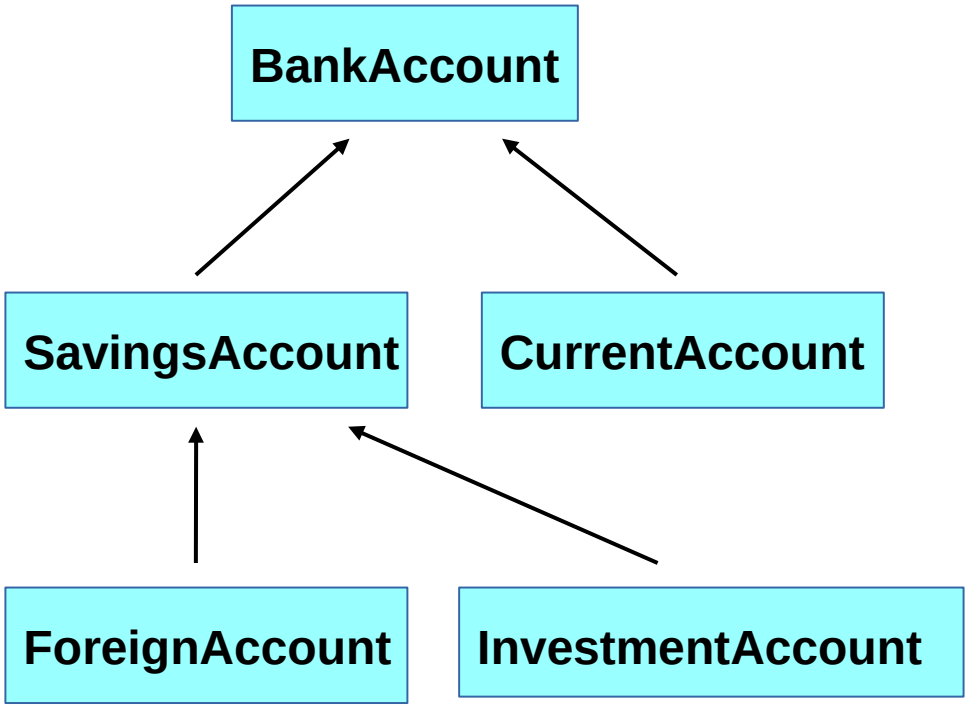
- Designing your own class hierarchies
- Understanding and extending class hierarchies provided by others



# More Example Hierarchies




*means “extends”*



# Extend? Or just add members?

## Extend

***Student***: name, id, year, grades, courses, etc.

***GradStudent***: advisor, publications

***PhDStudent***: thesisTitle, committee

## Add members

***Student:*** name, id, year, grades, courses

***degree*** (BS,BE,MS,ME,PhD)

advisor *(can be blank)*

publications *(can be blank)*

thesisTitle *(can be blank)*

committee *(can be blank)*

**Not an easy choice! Depends on:**

**1) degree of difference in class instance behavior;**

**2) future plans for extension.**

**Sometimes you will end up changing your mind!**

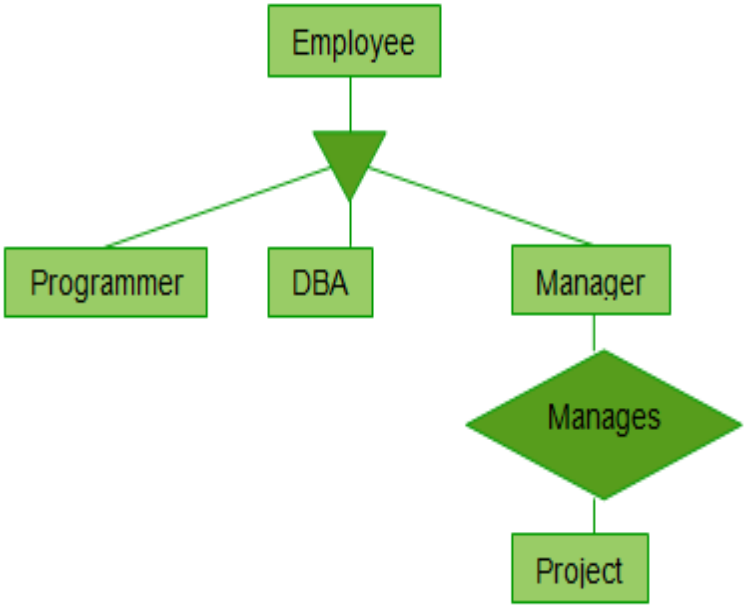
# Specialization versus Generalization

All these examples involve ***specialization***

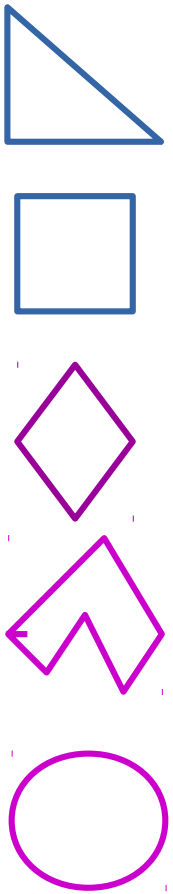
We started with a more general class (Square, Student, BankAccount), then created subclasses to represent special cases with additional attributes or behavior

Class hierarchies can also develop through ***generalization***.

Generalization involves looking for common features within a set of specific classes, and creating a superclass to capture these common features.



# Generalization of Exercise 2



***Triangle***: defined by 3 points; draw, calculate perimeter, calculate area

***Square***: defined by 1 point plus side length; draw, move, calculate perimeter, calculate area

***Diamond***: defined by 1 point plus major, minor axis; draw, move, calculate perimeter, calculate area

***Polygon***: defined by N points; draw, move, calculate perimeter, calculate area

***Circle:*** defined by center plus radius; draw, move, calculate perimeter, calculate radius

**These classes have a lot in common. We can reduce duplicate or similar code and improve modularity by creating a generalized superclass, *Shape*.**

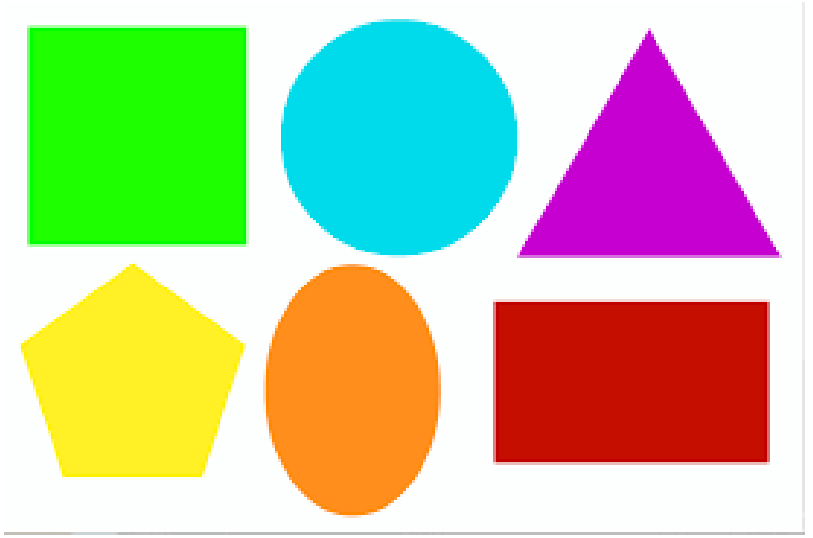
# ***Shape*** Superclass?

## Member data

- Anchor point X,Y  (e.g. upper left X,Y for Square)
- Arrays (or array lists) of X and Y coordinates for vertices
- Vertex count
- Color
- Figure number
- counter, allFigures list (static)

## Methods

- draw()
- move()
- drawAll()
- calcPerimeter(), calcArea()



# Try It!

```
public class Shape
{
    /** Anchor point X,Y */
    protected Point anchor;   /* determines the "position" of a shape */
    /* Point is a class in package java.awt that has a public x and y member */

    /** list of points */
    protected ArrayList<Point> vertices = new ArrayList<Point>();

    /** how many points? */
    protected int pointCount;

    /** color */
    protected Color color;

    /** so we can count and label figures */
    protected static int counter = 0;

    /** collection of all squares */
    protected static ArrayList<Shape> allFigures = new ArrayList<Shape>();

    /** used to cycle through display colors */
    protected static Color colors[] = {Color.RED, Color.GREEN, Color.BLUE,
                   Color.MAGENTA, Color.ORANGE};
    ...
}
```

# So far, so good...but...

**What about the common methods?**

*Square*, *Triangle*, *Diamond* etc. use (somewhat) different algorithms for drawing, moving, calculating perimeter and area

How can we create a superclass that has all the methods that apply to every shape, when the details are different for each specific shape?

# Abstract Classes

Java solves this problem using the concept of an ***abstract class***.

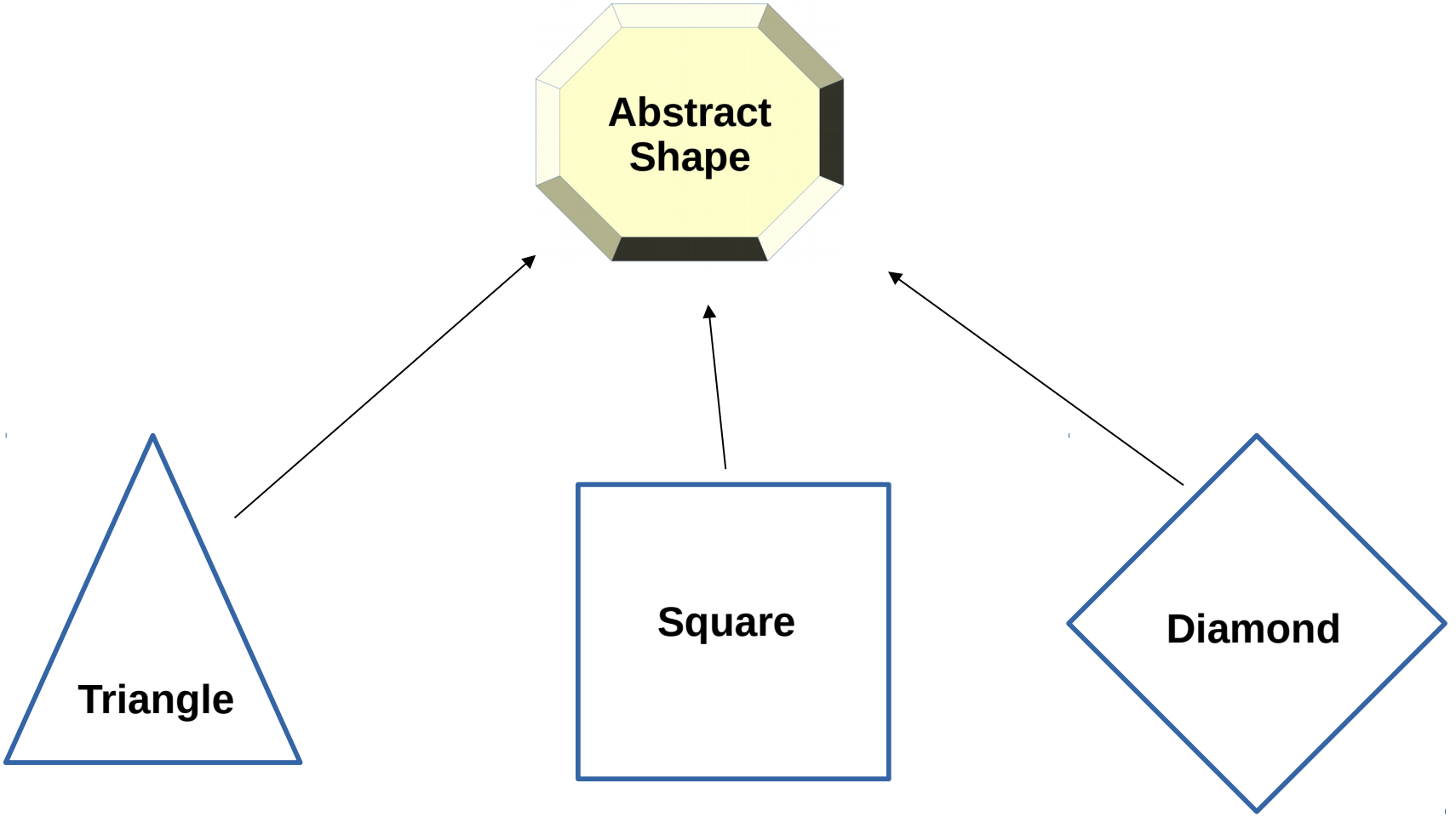
Created to be a superclass for a set of concrete classes.

Incomplete – specifies the existence of some methods (***abstract methods***) but has no code for these methods

Concrete classes that extend the abstract class must provide their own code for each abstract method.

*It is not possible to create an instance of an abstract class.*

# Class Hierarchy



# In Exercise 3,

# you will build this class hierarchy!