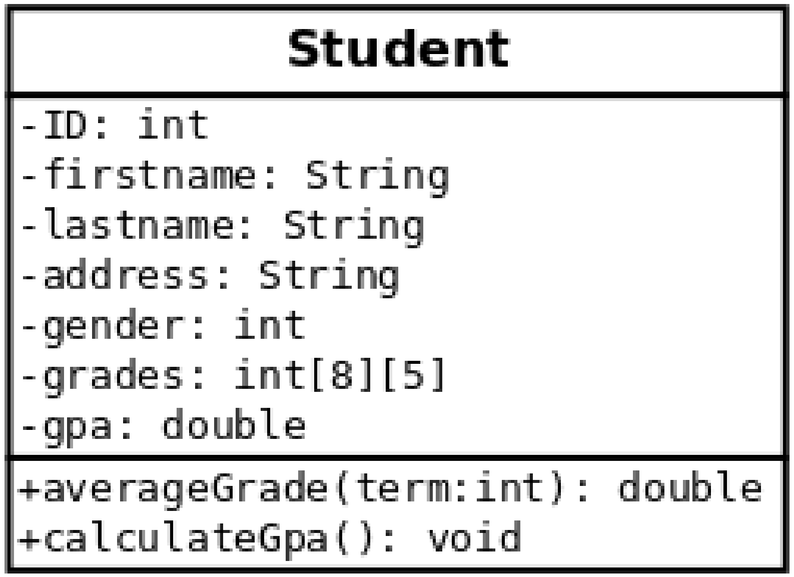
Object Oriented Design and Analysis CPE 372

Lecture 4

Overloading, Overriding and Polymorphism

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

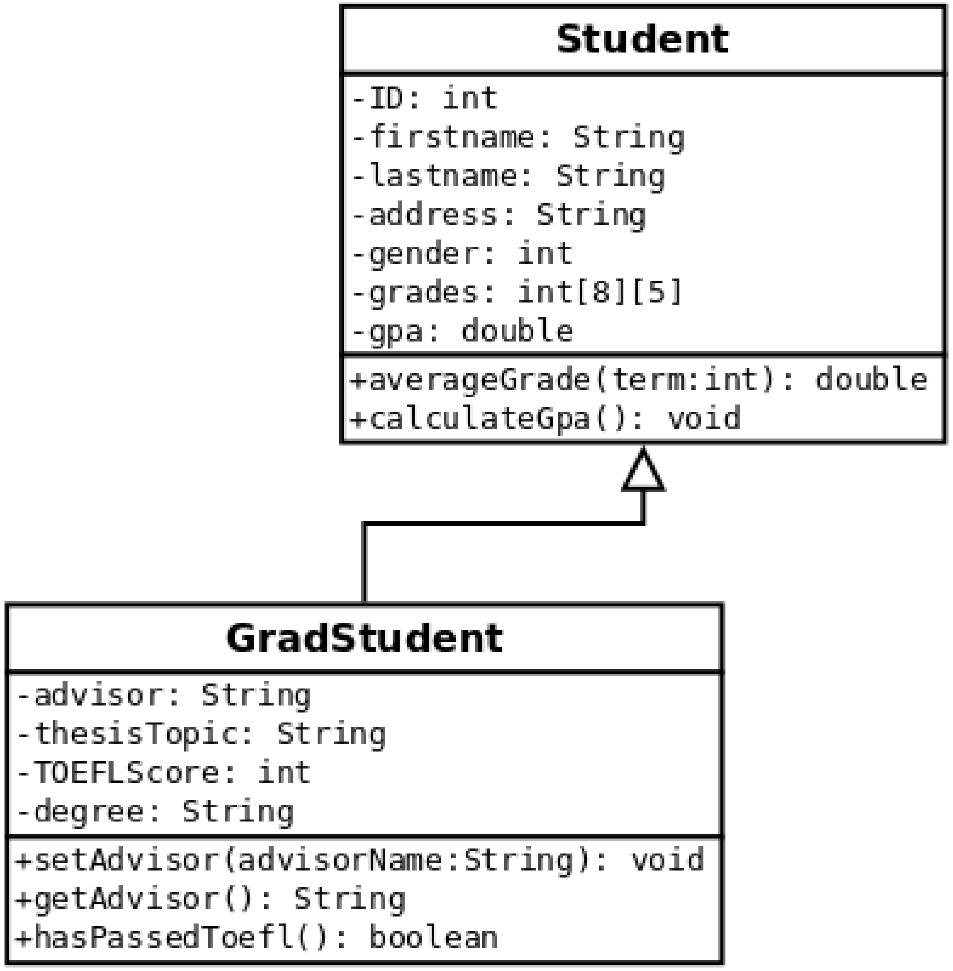
Inheritance and Class Hierarchies



Class ***Student*** represents a single university student at KMUTT.

Registration system will create one instance for each registered student.

Create Subclasses to Specialize

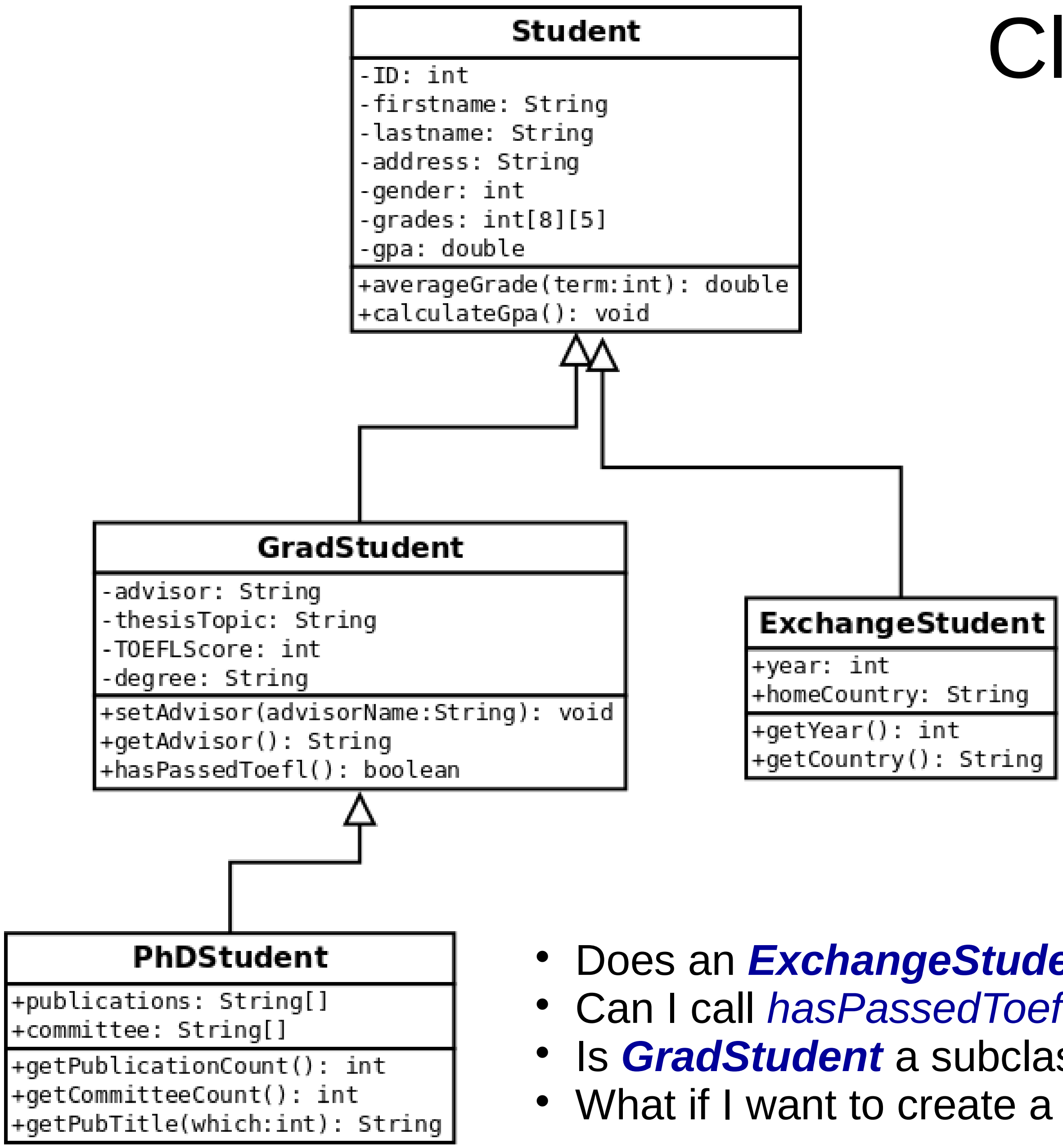


Class ***GradStuden***t has some additional members and methods.

However, it also inherits the members and methods of its parent class.

So we can call *calculateGpa()* on a ***GradStudent*** instance.

Class Hierarchy

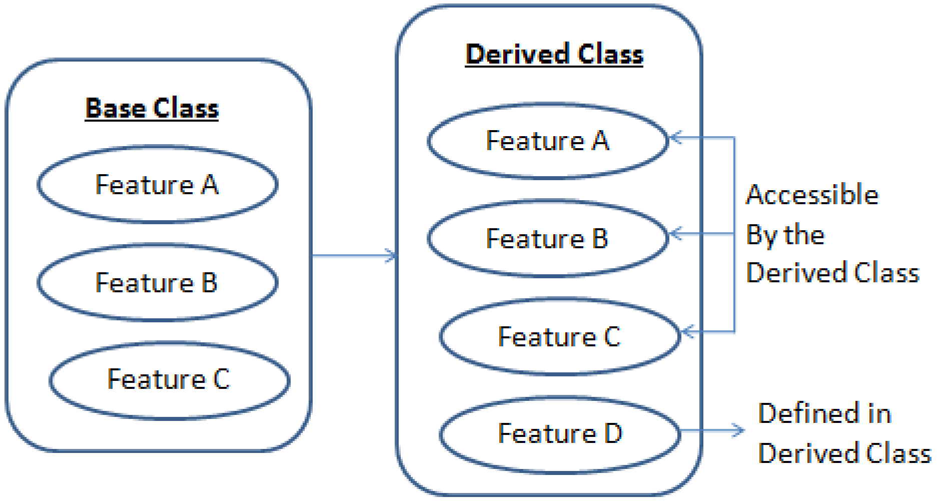


- Does an ***ExchangeStudent*** have an advisor?
- Can I call *hasPassedToefl()* method on a ***PhDStudent***?
- Is ***GradStudent*** a subclass of ***PhDStudent***?
- What if I want to create a class for high school students?

More Terminology

The parent class is sometimes called the **base class**

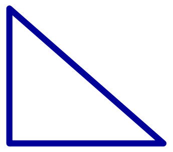
The child class is called the **derived class**



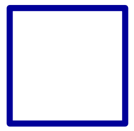
General OO terminology – not Java-specific

Exercise 3 Solution

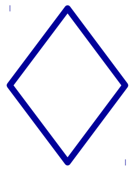
AbstractShape class has three subclasses



Triangle: defined by 3 points

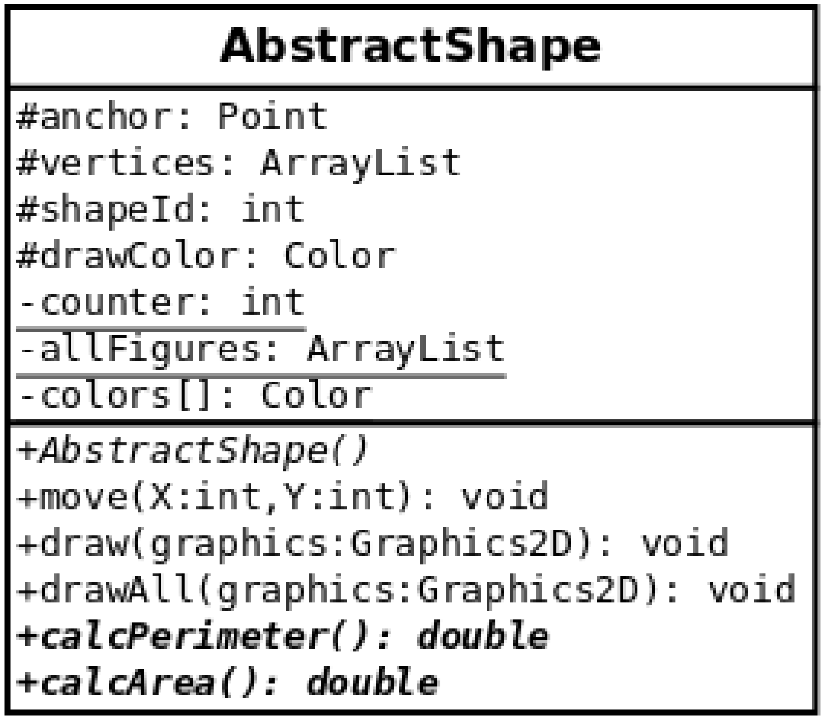


Square: defined by 1 point plus side length



Diamond: defined by 1 point plus vertical and horizontal axes

AbstractShape Class



```
AbstractShape
-------------------------------------------
#anchor: Point
#vertices: ArrayList
#shapeId: int
#drawColor: Color
-counter: int
-allFigures: ArrayList
-colors[]: Color
-------------------------------------------
+AbstractShape()
+move(X:int,Y:int): void
+draw(graphics:Graphics2D): void
+drawAll(graphics:Graphics2D): void
+calcPerimeter(): double
+calcArea(): double
```

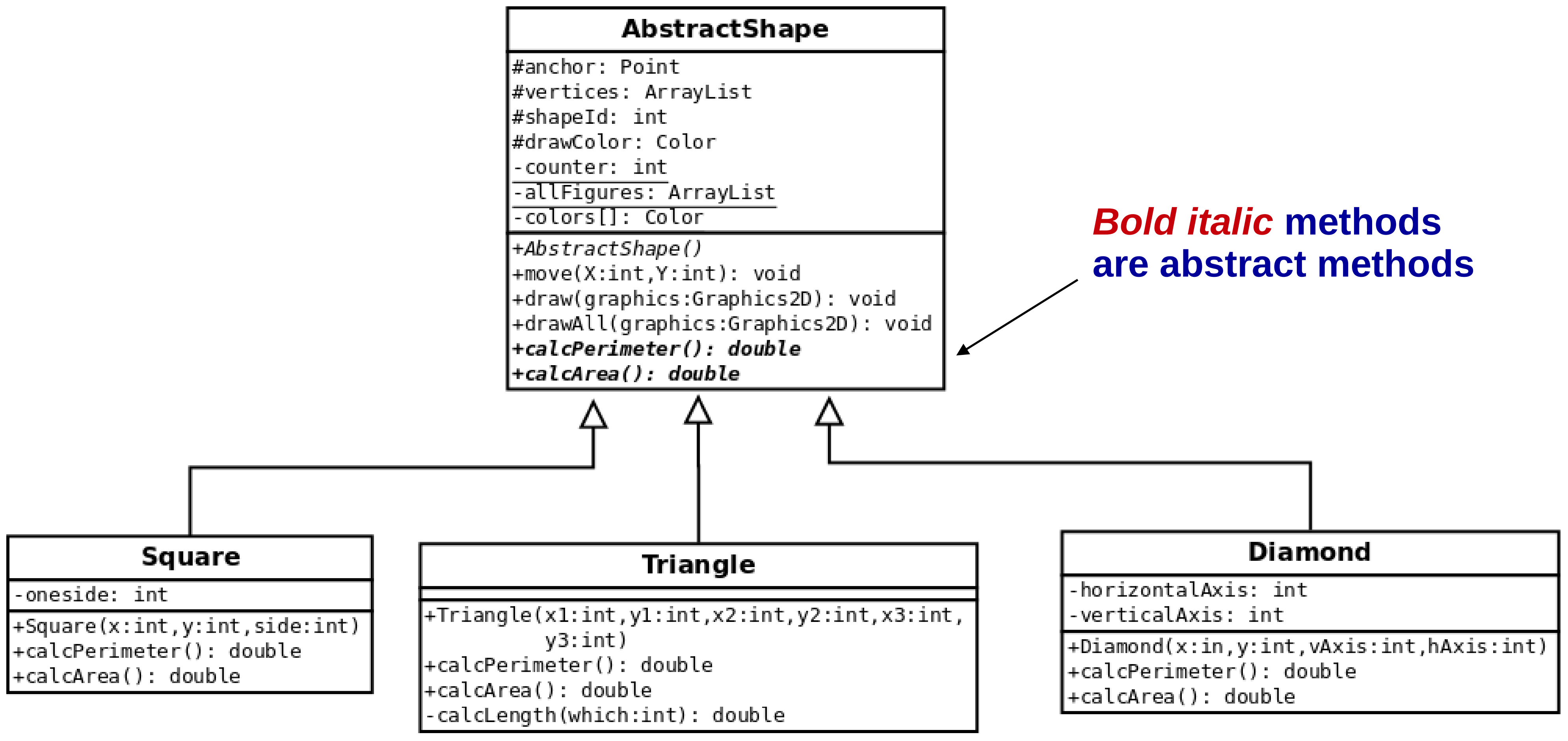
Almost all data items are stored in the superclass

counter, *allFigures*, *colors* are static members

I was able to implement *move(),* *draw()* and *drawAll()* as concrete methods in the superclass, rather than as abstract methods

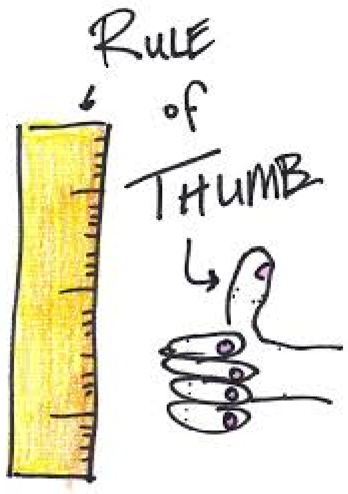
Perimeter and area methods must be abstract because the formula depends on the type of shape

Exercise 3 Class Hierarchy



Bold italic **methods are abstract methods**

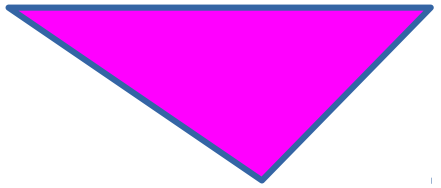
OO Design Rule of Thumb: 3



Maximize the amount of data and behavior that you implement in the superclass

Let’s add new behavior

What if we want to sometimes not only draw the outline of the shape but also fill it with some color?



We could add a new method

drawFilled(Graphic2D graphics, Color fillColor)

That method could call *draw()*, then add extra code for doing the fill process

However, a more “object-oriented” way to do this is to have a second version of *draw()* that has an extra argument.

To Use This Method...

```
Triangle myshape = new Triangle(12,10,32,11,20,22);
myshape.draw(graphics, Color.magenta);
```

Where would we define this new version of *draw()*?

Overloading

This is an example of ***overloading*** a method.

“Overloading” means creating multiple versions of a method that have the *same function name*, but *different arguments* and/or a *different return value*.

(We can also say the different implementations have ***different signatures***.)

Another Example

Original constructor for ***Triangle:***

```
public Triangle(int x1, int y1, int x2, int y2, int x3, int y3);
```

Or you might implement it like this:

```
public Triangle(Point p1, Point p2, Point p3);
```

It’s possible to do both!

You could also have a default constructor:

```
public Triangle()
{
    Triangle(100,0,100,100,150,200);
}
```

What are the advantages of overloading?

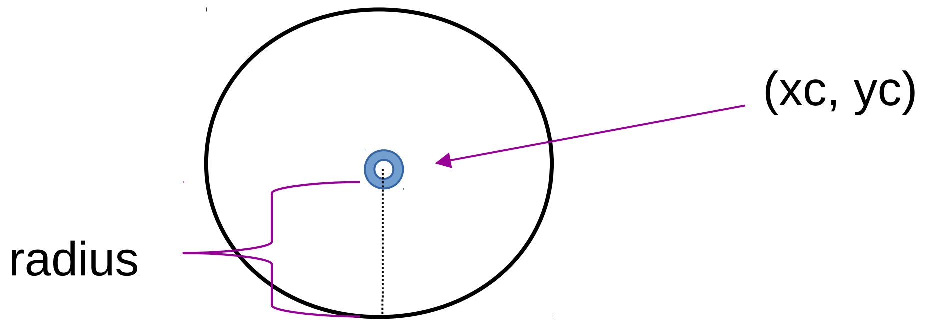
Focus attention on ***behavior*** rather than implementation

Provide greater convenience for different calling code

Only create overloaded methods when all variants do basically the same thing!

Adding a New Subclass

What if we want to create and draw circles?

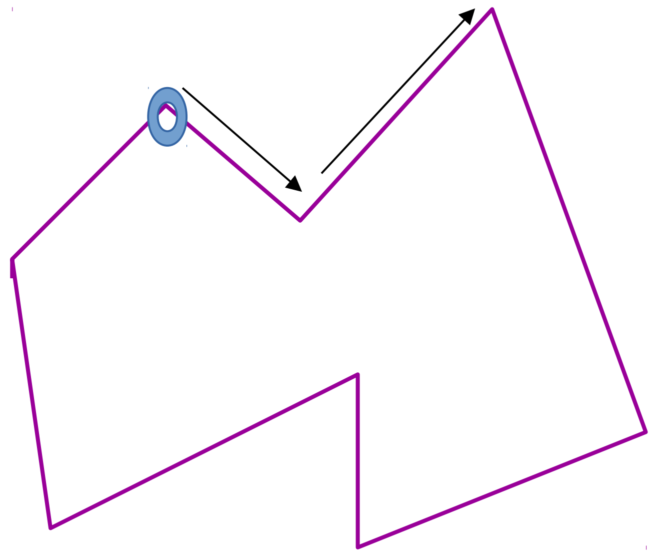


Defined by center point coordinates plus radius.

Let’s add the constructor, perimeter calculator and area calculator and see if it works.

What went wrong???

Draw Method in AbstractShape



Starts at the anchor point (point 0)

Iterates through all the points in the *vertices* ArrayList

Draws lines from current point to the next point

Uses modulus so last line will close the shape

But Circle does not store any points!

The Solution

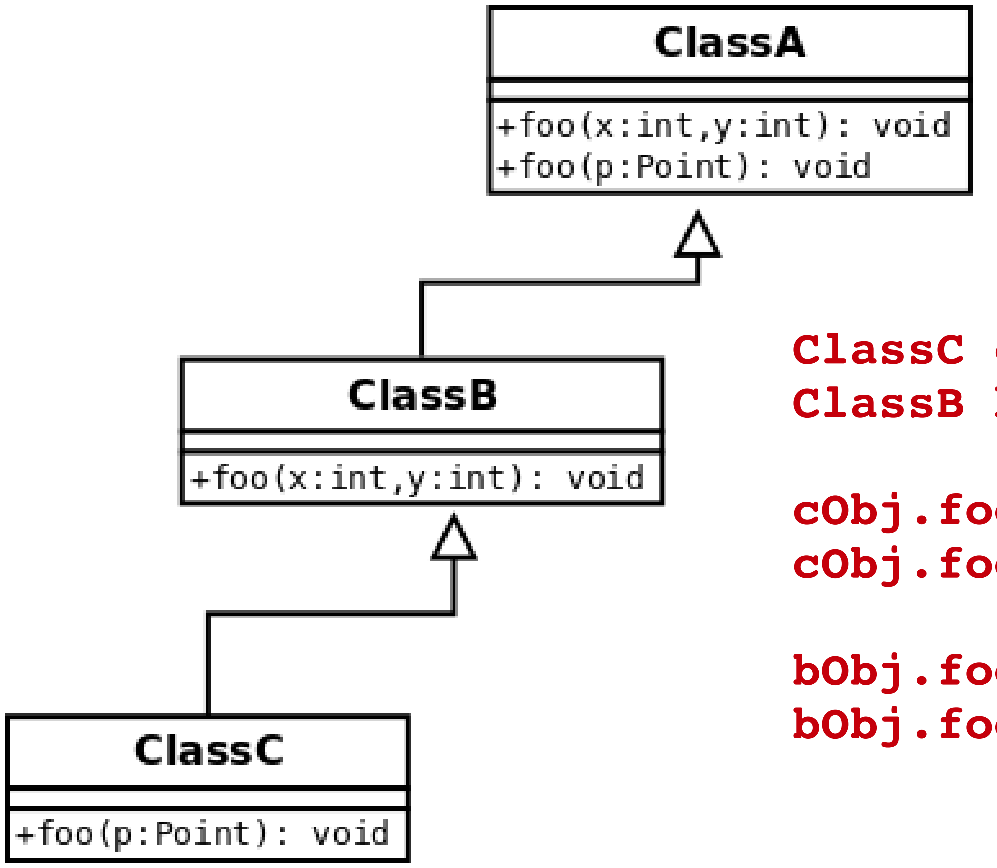
Implement specialized version of *draw()* in the *Circle* class

```
public void draw(Graphics2D graphics)
 {
     graphics.setPaint(drawColor);
     /* drawOval takes center plus width and height */
     graphics.drawOval(anchor.x,anchor.y,2*radius,2*radius);
     /* label it near the anchor point */
     int labelx = anchor.x + 5;
     int labely = anchor.y - 5;
     graphics.drawString(new String(" " + shapeId),labelx,labely);
 }
```

Creating a method in a subclass that has the same name and signature as one in the superclass is called ***overriding*** the method.

Java will call the right method!

The program will start at the level of the subclass and search ***upwards*** in the class hierarchy until it finds a matching method (matching name and signature)



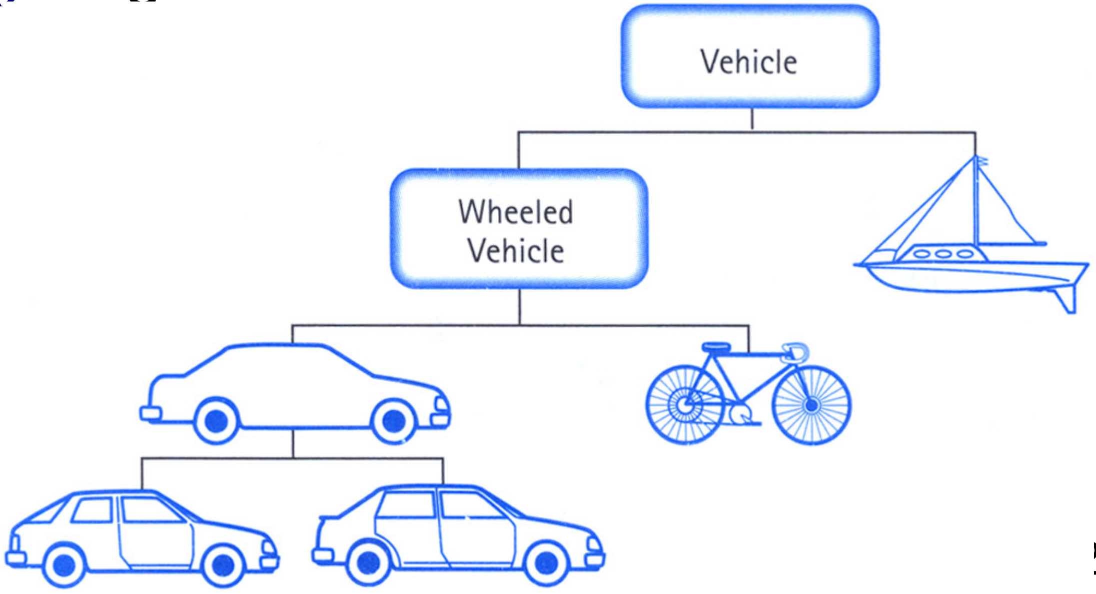
```
ClassC cObj= new ClassC();
ClassB bObj= new ClassB();

cObj.foo(200,200);
cObj.foo(new Point(230,80));

bObj.foo(200,200);
bObj.foo(new Point(230,80));
```

Overriding allows you to escape from strict inheritance

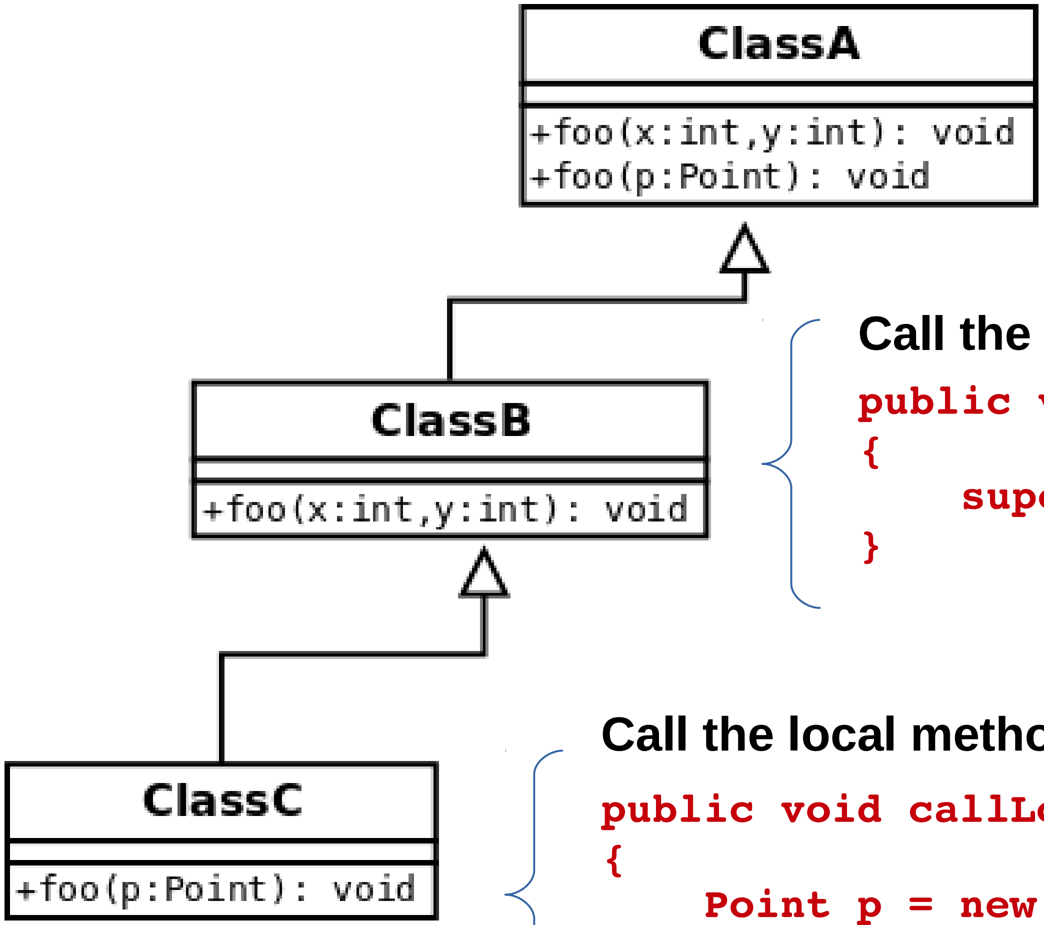
- Get the benefits of subclassing
- Provides customization if necessary
- Overriding can either *replace* or *expand* behavior

Circle *draw()* replaces AbstractShape.draw()

3DSquare *draw()* might extend it



If necessary, can invoke a specific version



Call the superclass method:

```
public void callParentFoo()
{
    super.foo(12,23);
}
```

Call the local method:

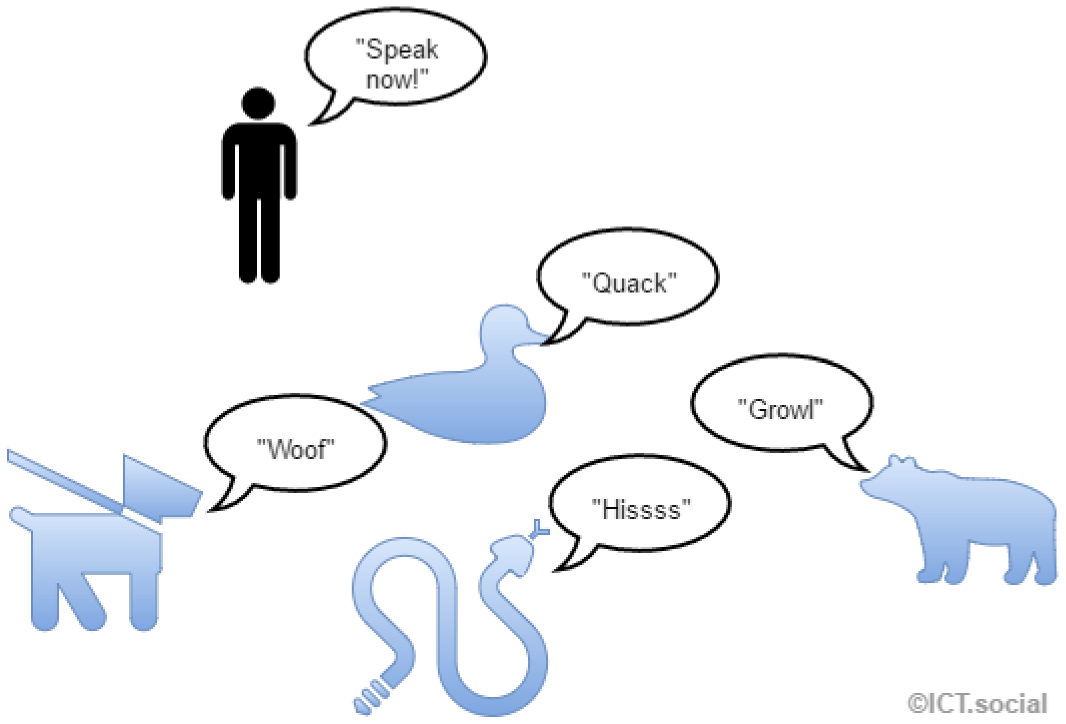
```
public void callLocalFoo()
{
    Point p = new Point(12,23);
    this.foo(p);
}
```

Polymorphism

Consider the method *drawAll()* in ***AbstractShape***

```
/**
 * Static method to draw all the shapes
 * that have been created so far.
 * @param  graphics   Graphics context for drawing.
 */
 public static void drawAll(Graphics2D graphics)
 {
    for (int i=0; i < allFigures.size(); i++)
    {
        AbstractShape shape = allFigures.get(i);
        shape.draw(graphics);
    }
 }
```

If we have instances of ***Circle*** in our list, we will be calling different methods depending on which concrete subclass of ***AbstractShape*** is stored in the *shape* variable. *drawAll()* doesn't know and doesn't care.



“Polymorphism” comes from Greek meaning “many forms”.

The most common use of polymorphism in OOD occurs when a parent class reference is used to refer to a child class object and to call child class methods that override the parent method.

Another Example

In this example, every call to *calcArea()* invokes a different child method.

```
/**
 * Static method to print the area of all the shape
 * that have been created so far.
 * @param  graphics   Graphics context for drawing.
 */
 public static void showAreas()
 {
    for (int i=0; i < allFigures.size(); i++)
    {
        AbstractShape shape = allFigures.get(i);
        double area = shape.calcArea();
        System.out.println(“The area of shape “ + i +
                “ is “ + area);
    }
 }
```

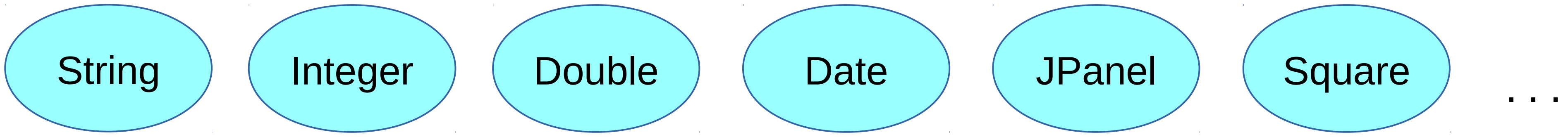
Another Example: PolyDemo

Every class in Java is a subclass of ***Object***

Object provides a standard method *toString()*

Some classes override that method

Contents of *objectlist*

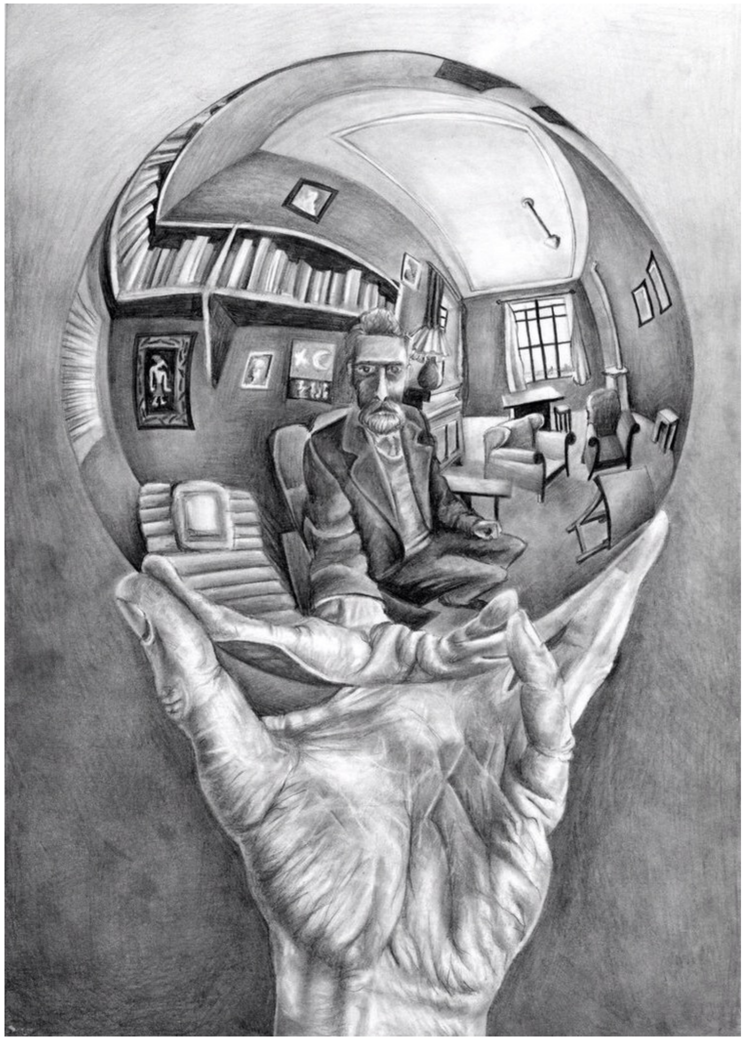


What if we want to know the actual class?

Java has the ability to "*introspect*" - to allow us to examine classes and methods while the program is running

The *getClass()* method returns the runtime class of an object (as an Object!)

We can use *toString()* to print the name of the class



Summary of Terms

Overloading

To create multiple methods in one class which have the *same method name* but *different signatures* (arguments)

Summary of Terms (2)

Overriding

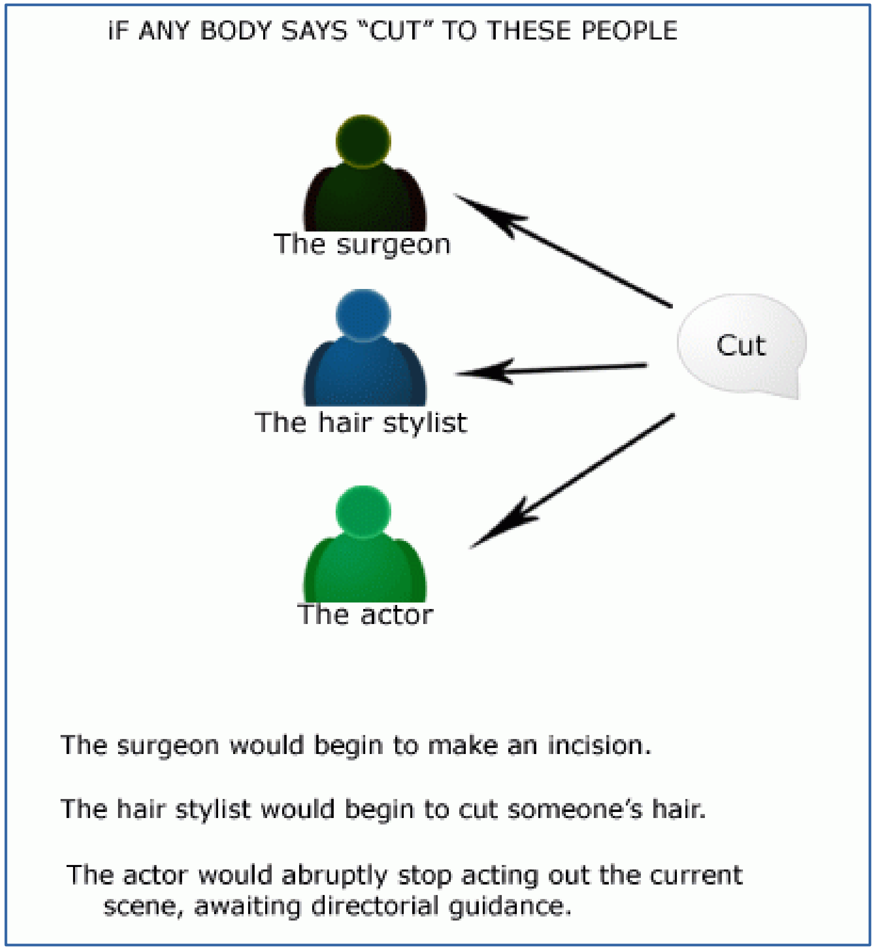
To create a *method in a subclass* that has the *same name and signature as a method in its superclass*, but a different implementation which is specific to that subclass

Summary of Terms (3)

Polymorphism

A situation in which multiple subclasses of a superclass have *methods with the same names and signatures,* but *different implementations*.

When the polymorphic method is called on an object identified as the superclass, *the correct subclass method is automatically invoked*.



The surgeon would begin to make an incision.

The hair stylist would begin to cut someone's hair.

The actor would abruptly stop acting out the current scene, awaiting directorial guidance.