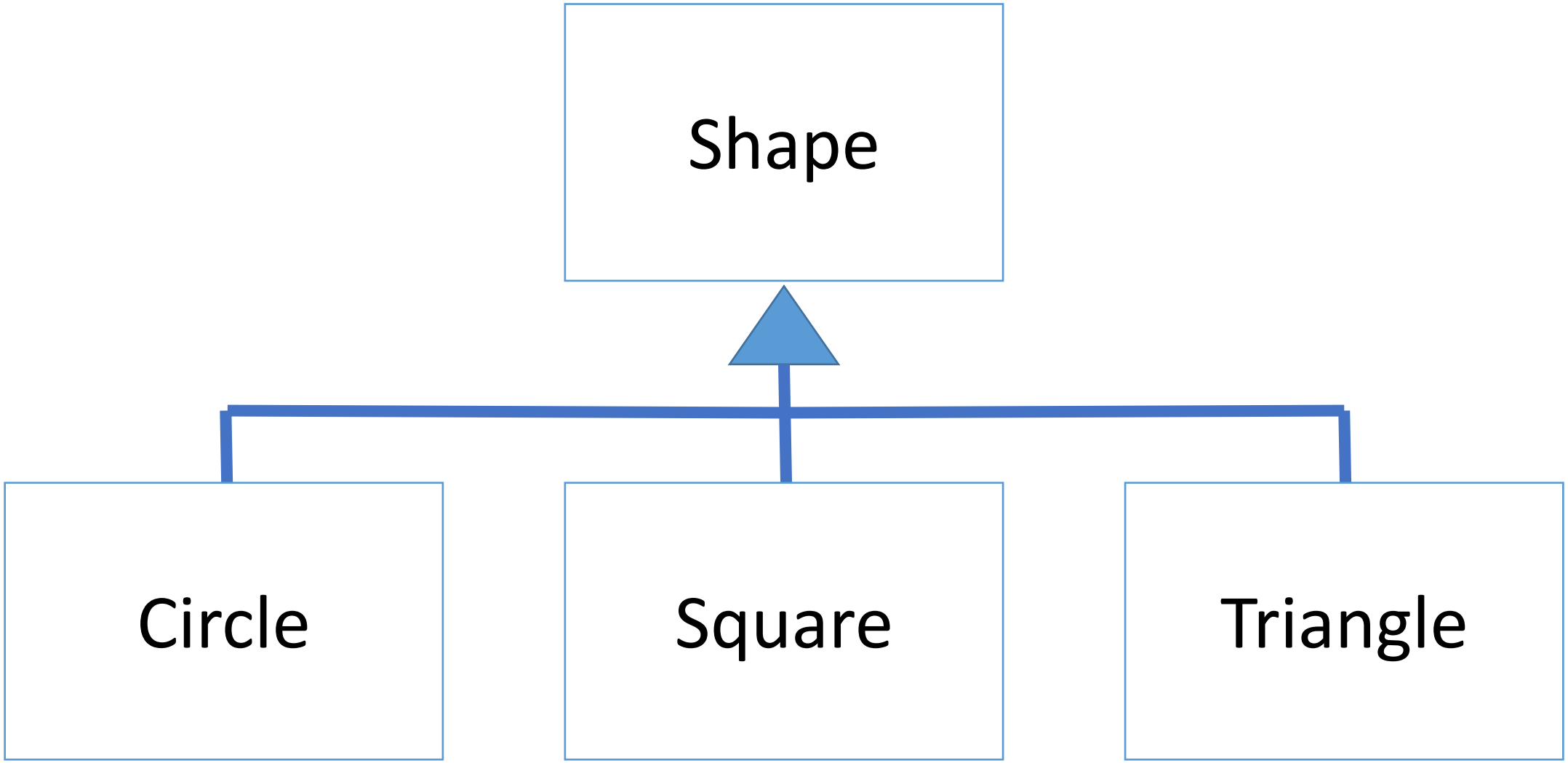
CSE 1325

Week of 10/17/2022

Instructor : Donna French

Inheritance

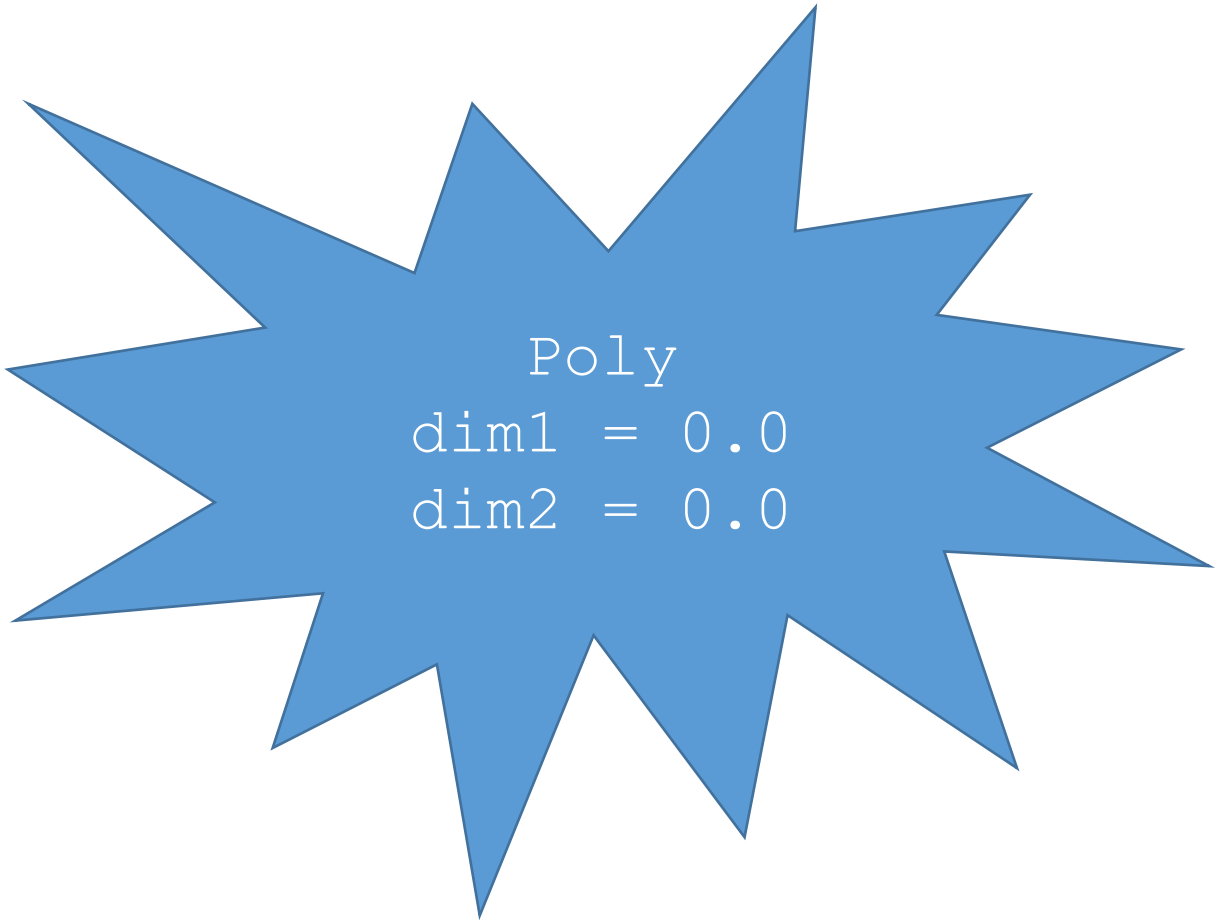


```
package shapedemo;

public class ShapeDemo
{
    public static void main(String[] args)
    {
        Shape A = new Shape("Poly");

        System.out.printf("\nMy name is %s\n", A.getName());
    }
}
```

```
package shapedemo;

public class Shape
{
    public String shapeName;
    public double dim1;
    public double dim2;

    public Shape(String name)
    {
        shapeName = name;
    }

    public String getName()
    {
        return shapeName;
    }
}
```

```
Shape A = new Shape("Poly");

System.out.printf("\nMy name is %s\n", A.getName());
```

```
My name is Poly
```



Now I want to create a class `Circle`.

The more abstract version of `Circle` is `Shape`.

`Shape` knows its name and how to get its name. `Shape` also knows dimensions.

We want `Circle` to know these same things, but we also want `Circle` to calculate its area.

When we create a `Circle` object, we want to construct it with its dimension/radius set already.

```
public class Circle
{
    private double radius = 0;
    private String name;

    public Circle(double radius,
                  String name)
    {
        this.radius = radius;
        this.name = name;
    }

    public double getArea()
    {
        return Math.PI *
               Math.pow(radius, 2);
    }
}
```

```
public class Circle
{
    private double radius = 0;
    private String name;

    public Circle(String name,
                  double radius)
    {
        this.radius = radius;
        this.name = name;
    }

    public double getArea()
    {
        return Math.PI *
               Math.pow(radius, 2);
    }
}
```

Creating the `Circle` class independently of class `Shape` does not take advantage of inheritance – we started over.

Not using inheritance now will not allow us to take advantage of polymorphism.

Adding "`extends Shape`" causes `Circle` to inherit from `Shape`

Pass `name` and `radius` to constructor

Pass `name` to the superclass's constructor

Why set `dim1` and `dim2` to `radius`?

```
public class Circle extends Shape
{
    public Circle(String name, double radius)
    {
        super(name);
        dim1 = dim2 = radius;
    }

    public double getArea()
    {
        return Math.PI * Math.pow(dim1, 2);
    }
}
```

```
package shapedemo;

public class ShapeDemo
{
    public static void main(String[] args)
    {
        Shape A = new Shape("Poly");
        Circle C = new Circle("Hoop", 5.0);

        System.out.printf("\nMy name is %s\n", A.getName());

        System.out.printf("\nMy name is %s and my area is %.2f\n",
                          C.getName(), C.getArea());
    }
}
```

Object `C` of class `Circle` is able to use the method `getName()` even though class `Circle` does not contain `getName()`

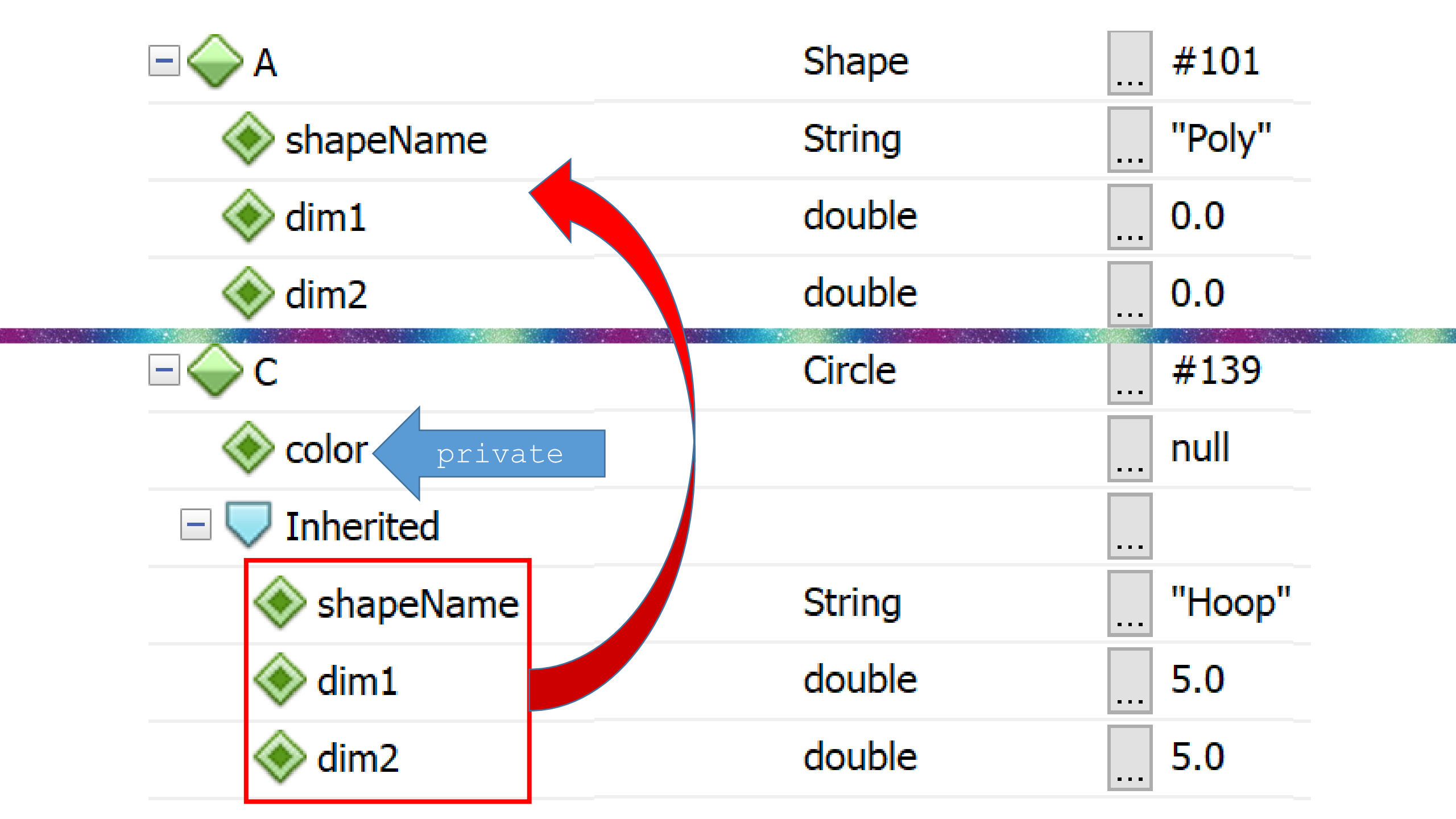
`Circle` inherited it from `Shape`

```
public class Circle extends Shape
{
    private String color;


    public void setColor(String color)
    {
        if (color == "Blue")
        {
            this.color = "Green";
        }
        else
        {
            this.color = color;
        }
    }
```

Let's add a private instance variable

Adding TO a subclass has NO effect on the superclass that the subclass inherited from.

Altering `Circle` does not affect `Shape`.


A
Shape
#101
shapeName
String
"Poly"
dim1
double
0.0
dim2
double
0.0
C
Circle
#139
color
private
null
Inherited
shapeName
String
"Hoop"
dim1
double
5.0
dim2
double
5.0

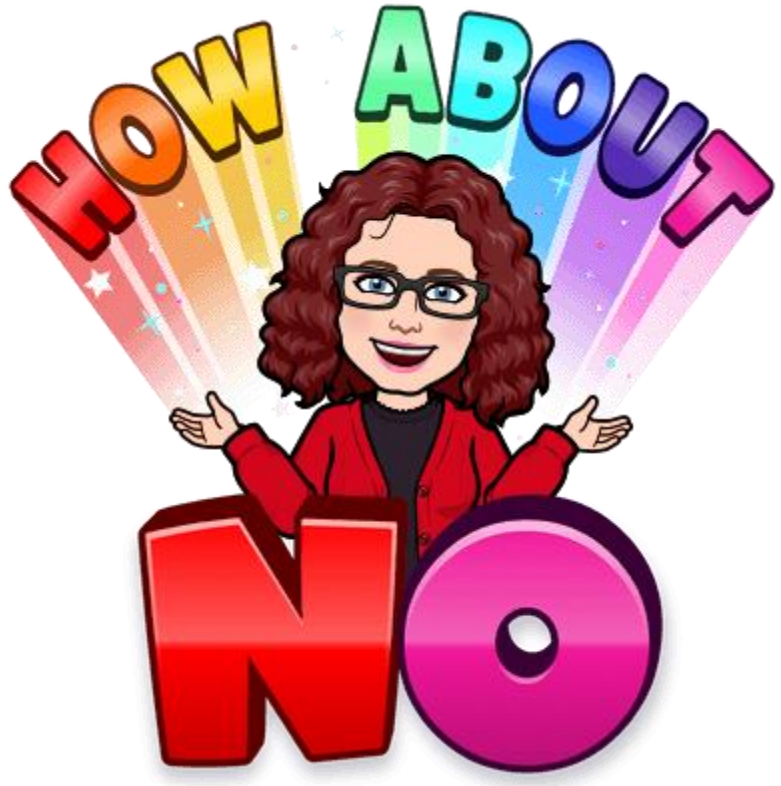
```
public static void main(String[] args)
{
    Shape A = new Shape("Poly");
    Circle C = new Circle("Hoop", 5.0);

    System.out.printf("\nMy name is %s\n", A.getName());

    System.out.printf("\nMy name is %s and my area is %.2f\n",
                      C.getName(), C.getArea());

    C.setColor("Blue");

    System.out.printf("%s's color is %s\n", C.getName(), C.getColor());
}
```

```
My name is Poly

My name is Hoop and my area is 78.54
Hoop's color is Green
```

```
public String getColor()
{
    return color;
}
```

If we make more shapes by inheriting from `Shape`, will we want those shapes to have a color?

Probably – why not?



Do we want to add

```
String color
```

```
setColor()
```

```
getColor()
```

to every shape we create?

```
package shapedemo;

public class Shape
{
    public String shapeName;
    public double dim1;
    public double dim2;
    private String color;

    public Shape(String name)
    {
        shapeName = name;
    }

    public String getName()
    {
        return shapeName;
    }
```

```
public void setColor(String color)
{
    if (color == "Blue")
    {
        this.color = "Green";
    }
    else
    {
        this.color = color;
    }
}

public String getColor()
{
    return color;
}
}
```

```
My name is Poly

My name is Hoop and my area is 78.54
Hoop's color is Green
```

Let's create another subclass using superclass `Shape`.

```
package shapedemo;

public class Rectangle extends Shape
{
    public Rectangle(String name, double height, double width)
    {
        super(name);
        dim1 = height;
        dim2 = width;
    }

    public double getArea()
    {
        return dim1 * dim2;
    }
}
```

```
public static void main(String[] args)
{
    Shape A = new Shape("Poly");
    Circle C = new Circle("Hoop", 5.0);
    Rectangle R = new Rectangle("NotQuiteSquare", 4.5, 3.2);

    System.out.printf("\nMy name is %s and my area is %.2f\n",
                      R.getName(), R.getArea());
}
```

```
My name is NotQuiteSquare and my area is 14.40
```

A	Shape	...	#103
shapeName	String	...	"Poly"
dim1	double	...	0.0
dim2	double	...	0.0
color		...	null
C	Circle	...	#104
Inherited		...	
shapeName	String	...	"Hoop"
dim1	double	...	5.0
dim2	double	...	5.0
color		...	null
R	Rectangle	...	#105
Inherited		...	
shapeName	String	...	"NotQuiteSquare"
dim1	double	...	4.5
dim2	double	...	3.2
color		...	null

Now, let's add a new shape – a square.

How is a square different from a rectangle?

A square is a special type of rectangle where all four sides have the same length.

So a square is a shape and a rectangle which means

- the area calculation for a square is the same as a rectangle.
- rectangle's area calculation requires two sides (l * w) so square could use the same calculation as long as length = width.

```
public class Square extends Rectangle
{
    public Square(String name, double size)
    {
        super(name, size, size);
    }
}
```

`Square` inherits from `Rectangle` (instead of `Shape`); therefore, only needs to call `Rectangle`'s constructor.

```
Square S = new Square("Quad", 3.4);

System.out.printf("\nMy name is %s and my area is %.2f\n",
                  S.getName(), S.getArea());
```

```
My name is Quad and my area is 11.56
```

Square inherited `getName()` and `getArea()` from `Rectangle` who inherited them from `Shape`.

Inheritance

With inheritance, the common instance variables and methods of all the classes in the hierarchy are declared in the superclass.

When changes are required for these common features, you need to make the changes only in the superclass.

Subclasses then inherit the changes.

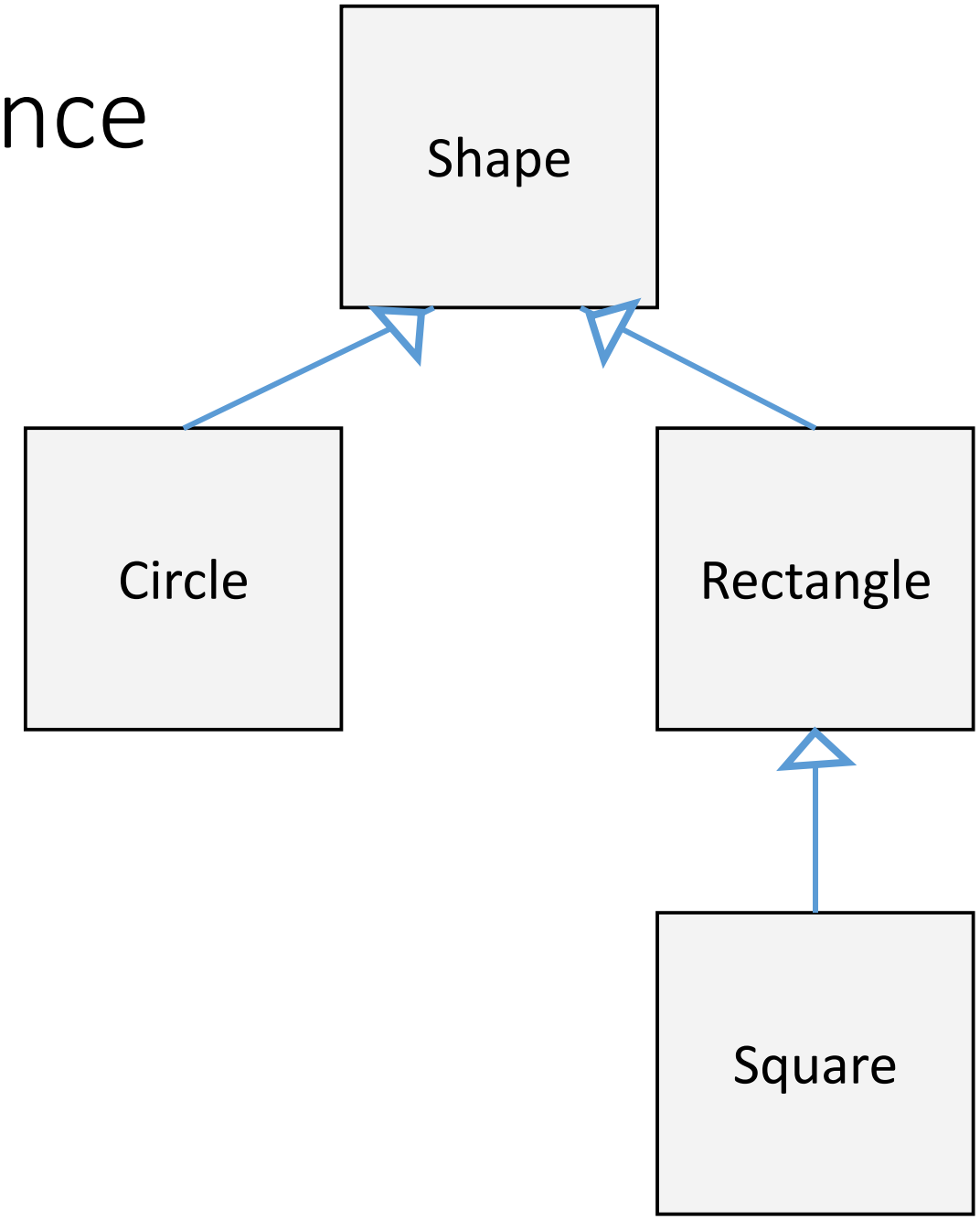
Without inheritance, changes would need to be made to all the source code files that contain a copy of the code in question.

Inheritance

When Java constructs subclass objects, it does so in phases.

First, the most-superclass (at the top of the inheritance tree) is constructed first.

Then each subclass is constructed in order, until the most-subclass (at the bottom of the inheritance tree) is constructed last.

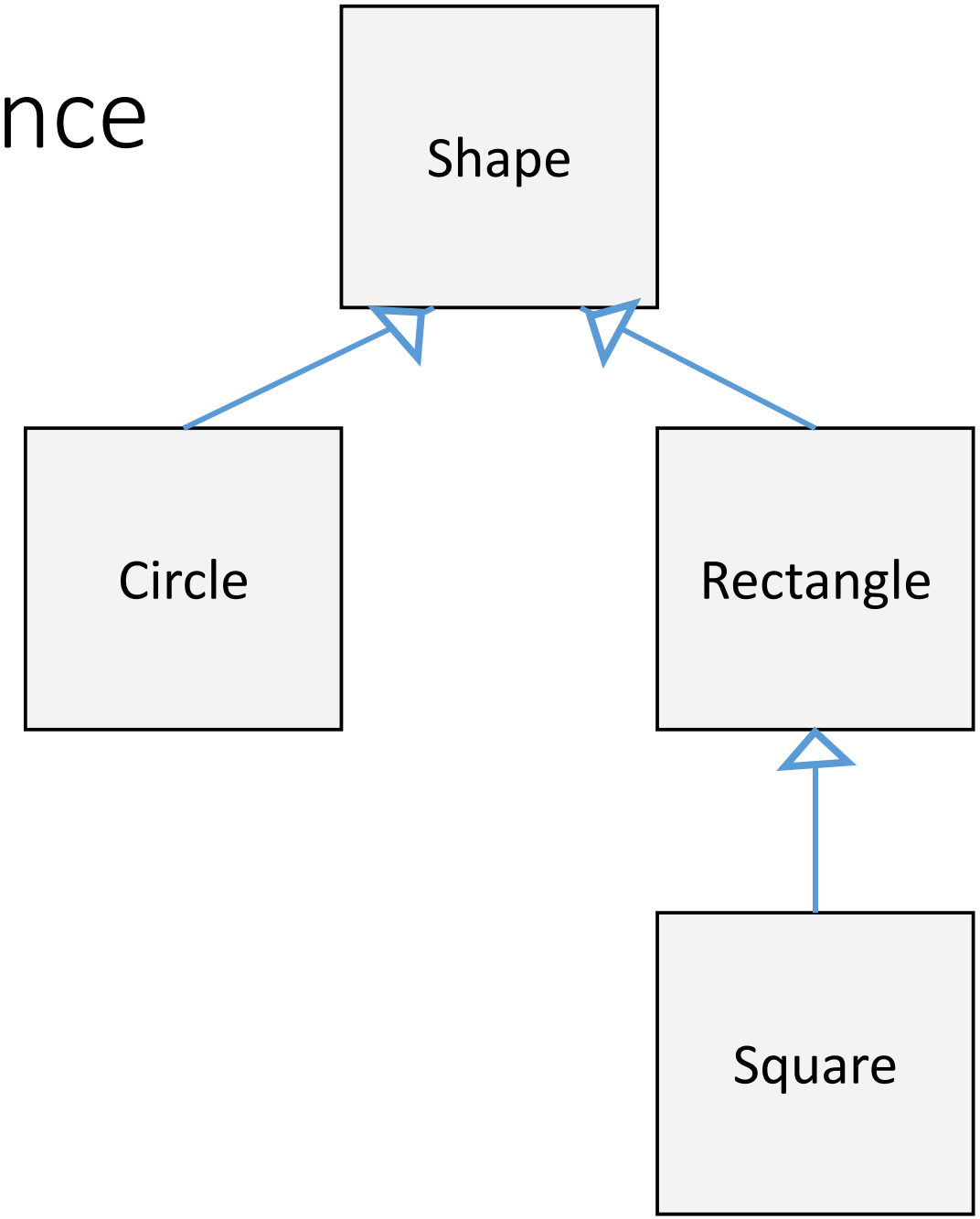


Inheritance

So when we construct a `Circle`, a `Shape` is constructed first and then the `Circle` is constructed.

When we construct a `Square`, a `Shape` is constructed and then a `Rectangle` and then a `Square`.

A subclass cannot exist until the superclass exists.



Inheritance

```
Shape A = new Shape("Poly");
Circle C = new Circle("Hoop", 5.0);
Rectangle R = new Rectangle("NotQuiteSquare", 4.5, 3.2);
Square S = new Square("Quad", 3.4);

System.out.printf("\nMy name is %s\n", A.getName());
```

```
Constructing Shape
Constructing Shape
Constructing Shape
Constructing Shape

My name is Poly
```

```
public Shape(String name)
{
    shapeName = name;
    System.out.println("Constructing Shape");
}
```

Inheritance

```
Shape A = new Shape("Poly");
Circle C = new Circle("Hoop", 5.0);
Rectangle R = new Rectangle("NotQuiteSquare", 4.5, 3.2);
Square S = new Square("Quad", 3.4);

System.out.printf("\nMy name is %s\n", A.getName());
```

```
Constructing Shape
Constructing Shape
Constructing Circle
Constructing Shape
Constructing Shape

My name is Poly
```

```
public Circle(String name, double radius)
{
    super(name);
    this.dim1 = this.dim2 = radius;
    System.out.println("Constructing Circle");
}
```

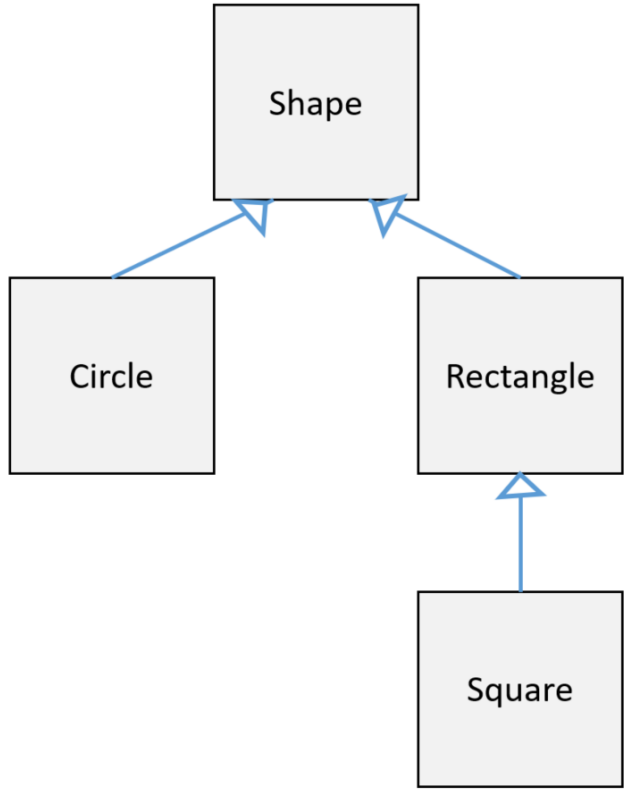
Inheritance

```
Shape A = new Shape("Poly");
Circle C = new Circle("Hoop", 5.0);
Rectangle R = new Rectangle("NotQuiteSquare", 4.5, 3.2);
Square S = new Square("Quad", 3.4);

System.out.printf("\nMy name is %s\n", A.getName());
```

```
Constructing Shape
Constructing Shape
Constructing Circle
Constructing Shape
Constructing Rectangle
Constructing Shape
Constructing Rectangle

My name is Poly
```

```
public Rectangle(String name, double height, double width)
{
    super(name);
    this.dim1 = height;
    this.dim2 = width;
    System.out.println("Constructing Rectangle");
}
```

```
public class Square extends Rectangle
{
    public Square(String name, double size)
    {
        super(name, size, size);
        System.out.println("Constructing Square");
    }
}
Shape A = new Shape("Poly");
Circle C = new Circle("Hoop", 5.0);
Rectangle R = new Rectangle("NotQuiteSquare", 4.5, 3.2);
Square S = new Square("Quad", 3.4);

System.out.printf("\nMy name is %s\n", A.getName());
```




```
Constructing Shape
Constructing Shape
Constructing Circle
Constructing Shape
Constructing Rectangle
Constructing Shape
Constructing Rectangle
Constructing Square

My name is Poly
```

Inheritance

The subclass often uses instance variables and methods from the superclass, but the superclass knows nothing about the subclass.

Instantiating the superclass first ensures those variables are already initialized by the time the subclass is created and ready to use them.

Remember that Java always constructs the “first” or “most super” class first. It then walks through the inheritance tree in order and constructs each successive subclass.

Inheritance

With non-subclasses, constructors only have to worry about their own instance variables.

For example, consider `Shape`. We can create a `Shape` object like this:

```
Shape A = new Shape("Poly");
```

Here's what actually happens when `Shape` is instantiated:

- Memory for `Shape` is set aside
- The appropriate `Shape` constructor is called
- The body of the constructor executes
- Control is returned to the caller

Inheritance

With subclasses, a few more things happen

For example, consider `Circle`. We can create a `Circle` object like this:

```
Circle C = new Circle("Hoop", 5.0);
```

Here's what actually happens when `Circle` is instantiated:

- Memory is set aside for both the `Shape` and `Circle` portions
- The appropriate `Circle` constructor is called who then calls the `Shape` constructor
- The `Shape` object is then constructed using the appropriate `Shape` constructor. If no constructor is specified, the default constructor will be used.
- The body of the constructor executes
- Control is returned to the caller which is the `Circle` constructor which then fires
- The body of the constructor executes
- Control is returned to the caller

Inheritance

The only real difference between constructing an object that inherits and an object that does not inherit is that before the subclass constructor can do anything substantial, the superclass constructor is called first.

The superclass constructor sets up the superclass portion of the object, control is returned to the subclass constructor, and the subclass constructor is allowed to finish up its job.

Constructors in Subclasses

- Instantiating a subclass object begins a *chain* of constructor calls in which the subclass constructor, before performing its own tasks, invokes its superclass's constructor

- If the superclass is derived from another class, the superclass constructor is required to invoke the constructor of the next class up in the hierarchy, and so on.

- The last constructor called in this chain is the constructor of the class at the base of the hierarchy, whose body actually finishes executing *first*.

- The most subclass constructor's body finishes executing *last*.

- Each superclass constructor initializes the superclass instance variables that the subclass object inherits.

Constructors in Subclasses

Superclass constructors are *not* inherited by subclasses.

Subclass constructors can call subclass versions.

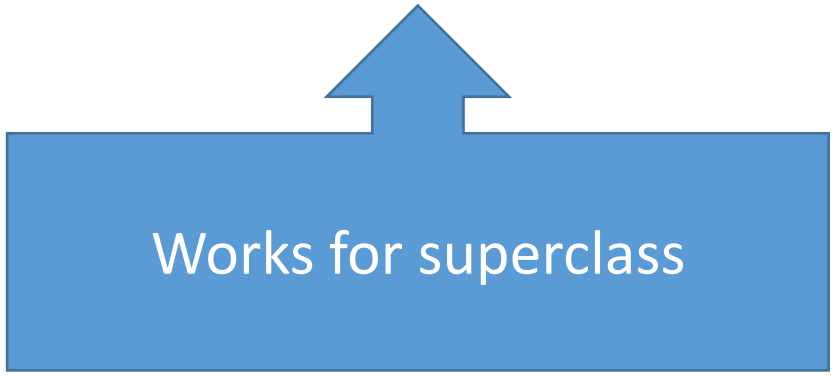
If the subclass does not explicitly define a constructor, the compiler still generates a default constructor in the subclass.

Inheritance

A class's **`public`** members are accessible wherever the program has a reference to an object of that class or one of its subclasses.

```
System.out.printf("\nMy name is %s\n", A.getName());
System.out.printf("\nMy name is %s\n", A.shapeName);
```

```
My name is Poly

My name is Poly
```

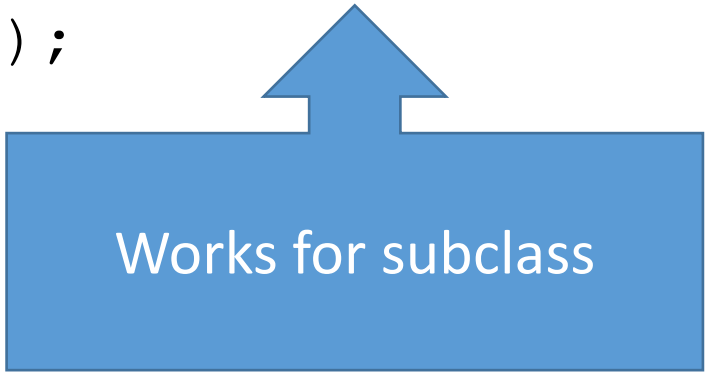


Inheritance

A class's **`public`** members are accessible wherever the program has a reference to an object of that class or one of its subclasses.

```
System.out.printf("\nMy name is %s and my area is %.2f\n",
                  C.getName(), C.getArea());
System.out.printf("\nMy name is %s and my area is %.2f\n",
                  C.shapeName, C.getArea());
```

```
My name is Hoop and my area is 78.54

My name is Hoop and my area is 78.54
```



Inheritance

A class's **`private`** members are accessible only within the class itself. Subclasses cannot directly access `private` members of their superclass.

`color` is `private` in `Shape`

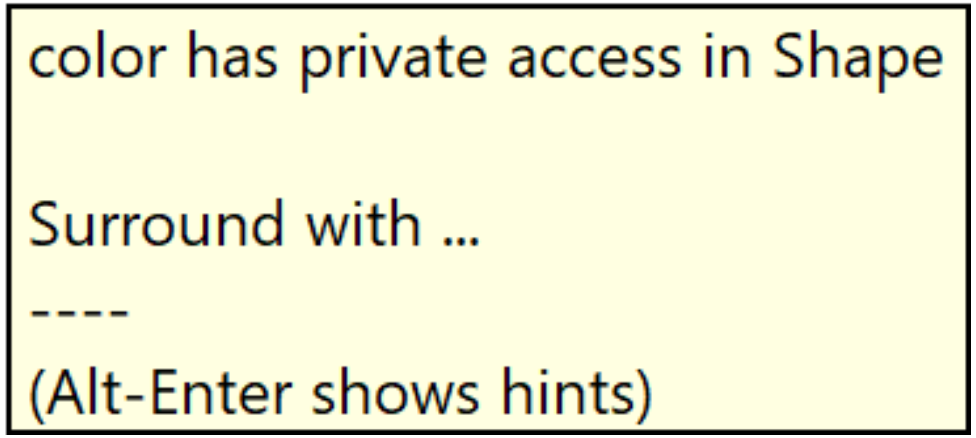
```
private String color;
```

We were able to set `color` for `Circle` by calling `setColor()`

```
C.setColor("Blue");
```

We cannot set `color` directly in `Circle`

```
C.color = "Blue";
```



Inheritance

`protected` access is an intermediate level of access between `public` and `private`.

A superclass's **`protected`** members can be accessed by

- members of that superclass
- by members of its subclasses
- by members of other classes in the same package

A class diagram shows **`protected`** access with a `#` (`+` for `public` and `–` for `private` and now `#` for `protected`)

Inheritance

Let's change `color` **from** `private` **access to** `protected` **in** `Shape`

```
public class Shape
{
    public String shapeName;
    public double dim1;
    public double dim2;
    private String color;    protected String color;
```

```
C.setColor("Blue");
System.out.printf("\nC's color is %s\n", C.getColor());

C.color = "Blue";
System.out.printf("\nC's color is %s\n", C.getColor());
```

```
C's color is Green

C's color is Blue
```

Because `color` is now `protected` instead of `private`, we can bypass any checks put into the class's setter for that instance variable.

```
public void setColor(String color)
{
    if (color == "Blue")
    {
        this.color = "Green";
    }
    else
    {
        this.color = color;
    }
}
```

`protected`

Inheriting `protected` instance variables enables direct access to the variables by subclasses.

In most cases, it's better to use `private` instance variables to encourage proper software engineering.

Code will be easier to maintain, modify and debug.

`protected`

Using `protected` instance variables creates several potential problems.

The subclass object can set an inherited variable's value directly without using a set method.

A subclass object can assign an invalid value to the variable

Subclass methods are more likely to be written so that they depend on the superclass's data implementation.

Subclasses should depend only on the superclass services and not on the superclass data implementation.

`protected`

With `protected` instance variables in the superclass, we may need to modify all the subclasses of the superclass if the superclass implementation changes.

Such a class is said to be fragile or brittle, because a small change in the superclass can “break” subclass implementation.

You should be able to change the superclass implementation while still providing the same services to the subclasses.

If the superclass services change, we must reimplement our subclasses.

A class’s `protected` members are visible to all classes in the same package as the class containing the protected members—this is not always desirable.

`protected`

Final thoughts on the usage of `protected` …

Avoid `protected` instance variables.

Instead, create non-`private` methods that access `private` instance variables.

This will help ensure that objects of the class maintain consistent states.

Inheritance

Instance variables with public access can be accessed by anybody.

Instance variables with private access can only be accessed by instance methods of the same class.

This means subclasses cannot access private instance variables of the superclass directly.

Inheritance

With `public` instance variables

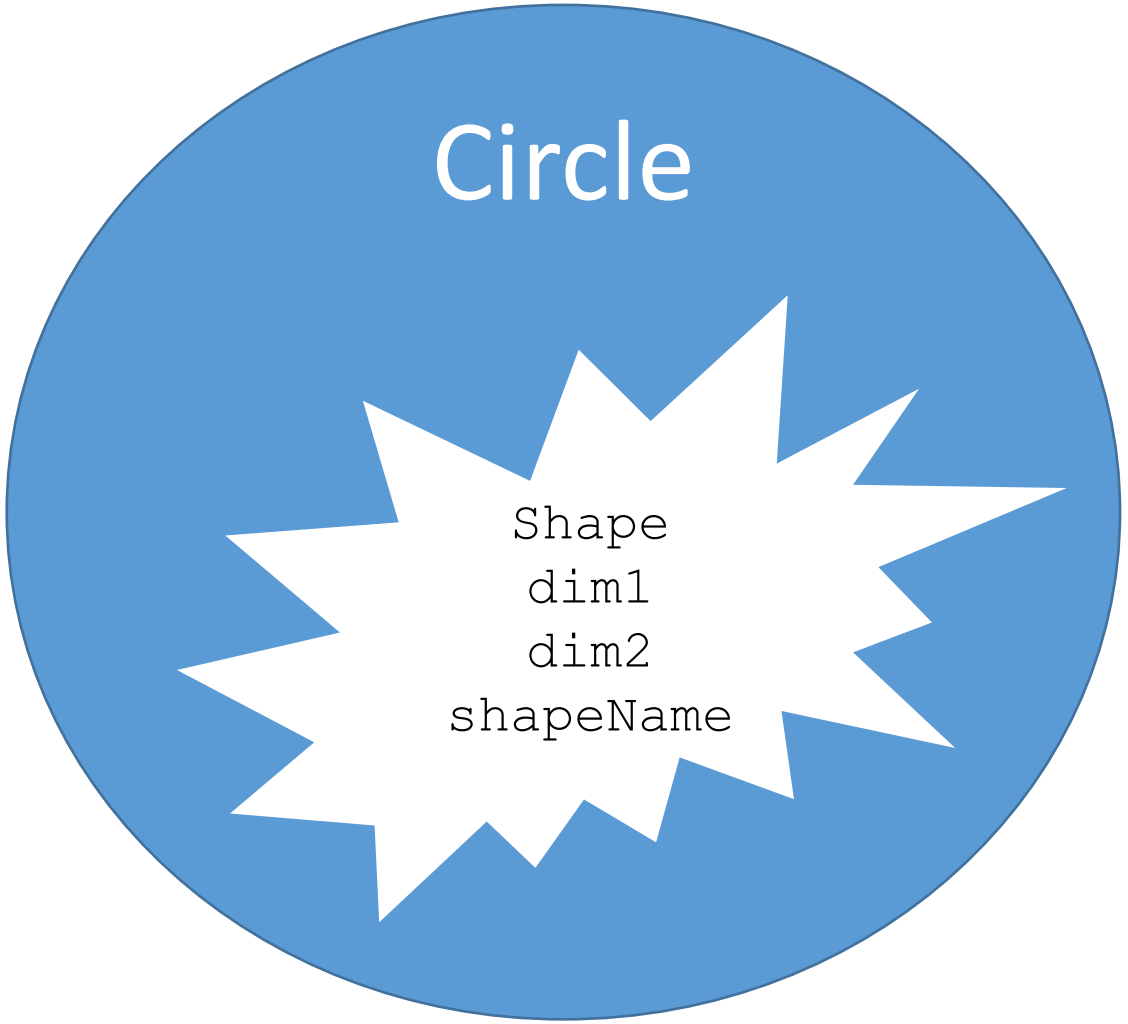
```
public String shapeName;
public double dim1;
public double dim2;
```

With `private` instance variables

```
private String shapeName;
private double dim1;
private double dim2;
```

Subclass `Circle` inherits `Shape`'s information but is not allowed to directly access the private instance variables inherited from `Shape`.

Subclasses will need to use access methods to access private instance variables of the superclass.

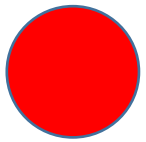


We would need to add getters and setters to `Shape` to provide access to private instance variables.

```
public void setDims(double Dim1, double Dim2)
{
    dim1 = Dim1;
    dim2 = Dim2;
}

public double getDim1()
{
    return dim1;
}

public double getDim2()
{
    return dim2;
}
```

We would then change `Circle` to use them.

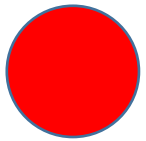
```
public class Circle extends Shape
{
    public Circle(String name, double radius)
    {
        super(name);
        dim1 = dim2 = radius;
    }

    public double getArea()
    {
        return Math.PI * Math.pow(dim1, 2);
    }
}
```



dim1 has private access in Shape

dim2 has private access in Shape

(Alt-Enter shows hints)



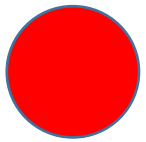
dim1 has private access in Shape

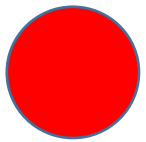
Flip operands of the binary operator

(Alt-Enter shows hints)

```
public class Circle extends Shape
{
    public Circle(String name, double radius)
    {
        super(name);
        setDims(radius, radius);
    }

    public double getArea()
    {
        return Math.PI * Math.pow(getDim1(), 2);
    }
}
```

```
public class Rectangle extends Shape
{
    public Rectangle(String name, double height, double width)
    {
        super(name);
        dim1 = height;
        dim2 = width;
    }

    public double getArea()
    {
        return dim1 * dim2;
    }
}
```





```
public class Rectangle extends Shape
{
    public Rectangle(String name, double height, double width)
    {
        super(name);
        setDims(height,width);
    }

    public double getArea()
    {
        return getDim1() * getDim2();
    }
}
```

```
My name is Poly

My name is Hoop and my area is 78.54
Hoop's color is Green

My name is NotQuiteSquare and my area is 14.40

My name is Quad and my area is 11.56
```

```
Shape A = new Shape("Poly");
Circle C = new Circle("Hoop", 5.0);
Rectangle R = new Rectangle("NotQuiteSquare", 4.5, 3.2);
Square S = new Square("Quad", 3.4);

System.out.printf("\nMy name is %s\n", A.getName());

System.out.printf("\nMy name is %s and my area is %.2f\n",
                  C.getName(), C.getArea());

System.out.printf("\nMy name is %s and my area is %.2f\n",
                  R.getName(), R.getArea());

System.out.printf("\nMy name is %s and my area is %.2f\n",
                  S.getName(), S.getArea());
```

`Circle`, `Rectangle` and `Square` all inherit `getName()`.

They each define their own version of `getArea()`.

Does it make sense to move `getArea()` to `Shape` and inherit it?

Moving `getArea()` to `Shape`

Yes

Every shape created from the `Shape` class will need an area.

No

Every shape created from the `Shape` class will calculate area differently so why bother putting it in `Shape`?

Decision

We are going to put `getArea()` into `Shape`, but for other reasons.

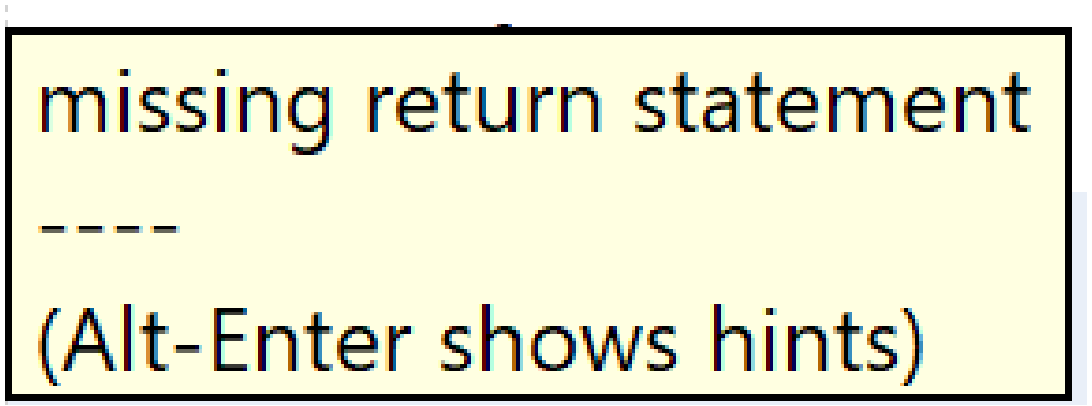
So what does `getArea()` in `Shape` look like??

Does a `Shape` have an area?

What calculation do we use?

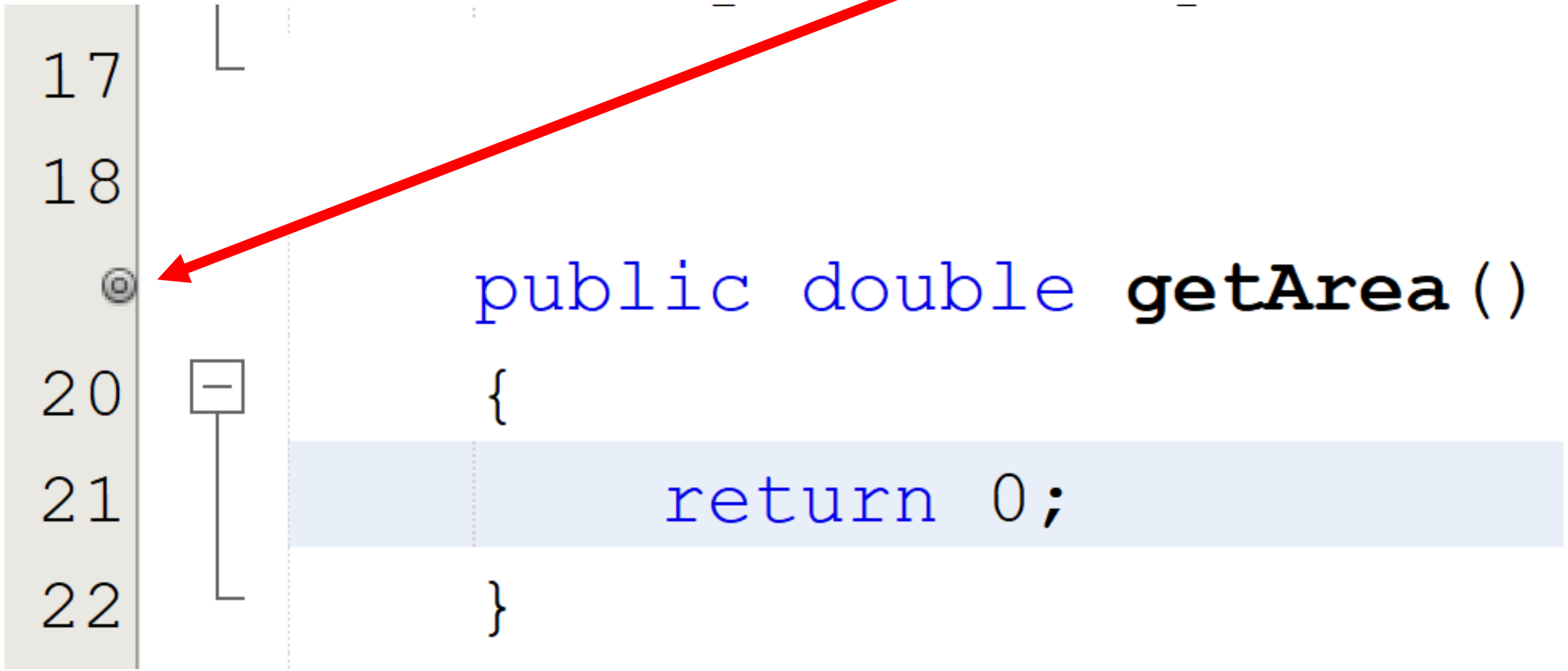
`getArea()` does not take any parameters and returns a `double` so let's start with that.

```
public double getArea()
{
    return 0;
}
```



So we have to return something…

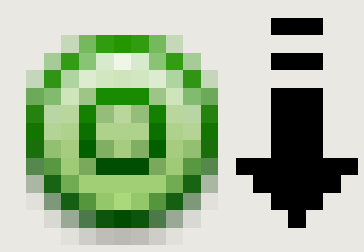
As soon as `getArea()` was added to `Shape`, a new symbol popped up in NetBeans



If we go back to `Circle.java`, another new symbol has popped up

```
13
14      public double getArea()
15      {
16          return Math.PI * Math.pow(getDim1(), 2);
17      }
18  }
```

Multiple annotations here [2] - click to cycle

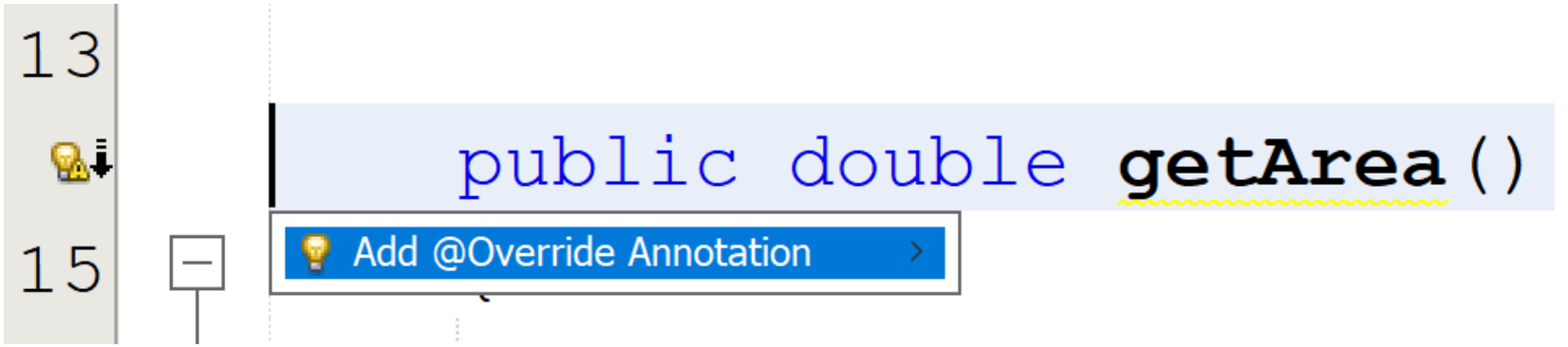


Overrides method from: shapedemo.Shape

(Ctrl+Shift+P goes to Ancestor Method)

Add @Override Annotation

(Alt-Enter shows hints)



If you click on the suggestion to "Add @Override Annotation", then you will get

```
14      @Override
        public double getArea()
16      {
17          return Math.PI * Math.pow(getDim1(), 2);
18      }
```

The `@Override` annotation is optional.

You should declare overridden methods with it to ensure that you defined the overridden method's signature correctly at compilation time.

Without the `@Override` annotation, signature mismatches may not be found until runtime.

`toString()` methods should actually be declared with the `@Override` annotation

However, you really won't see `toString()` with the `@Override` annotation since almost every instance of `toString()` is an override.

Remember – a non overridden version of `toString()` returns the object's reference.

Speaking of `toString()`

We have this code in our project

```
System.out.printf("\nMy name is %s and my area is %.2f\n",
                        C.getName(), C.getArea());
```

`shapeName` is a private instance variable of superclass `Shape`. We must use a getter method to retrieve the value of `shapeName`.

What if we created a `toString()` in `Circle` to print this statement?

We can create a `toString()` in `Circle` which would allow us to replace

```
System.out.printf("\nMy name is %s and my area is %.2f\n",
                  C.getName(), C.getArea());
```

with

```
System.out.println(C);
```


```
public String toString()
{
    return String.format("My name is %s and my area is %.2f\n",
                         getName(), getArea());
}
```

We still have to `getName()` to retrieve the private data.

Now, with the overload of `toString`() defined in `Circle`, we can print this message just by print the object.

```
System.out.printf("\nMy name is %s and my area is %.2f\n",
                                  C.getName(), C.getArea());
```

changes to

```
System.out.println(C);
```

and we get the same message

```
My name is Hoop and my area is 78.54
```

We would not need a getter if we accessed the private data from INSIDE the superclass.

```
public String toString()
{
    return String.format("My name is %s", shapeName);
}
```

Defined in `Shape`

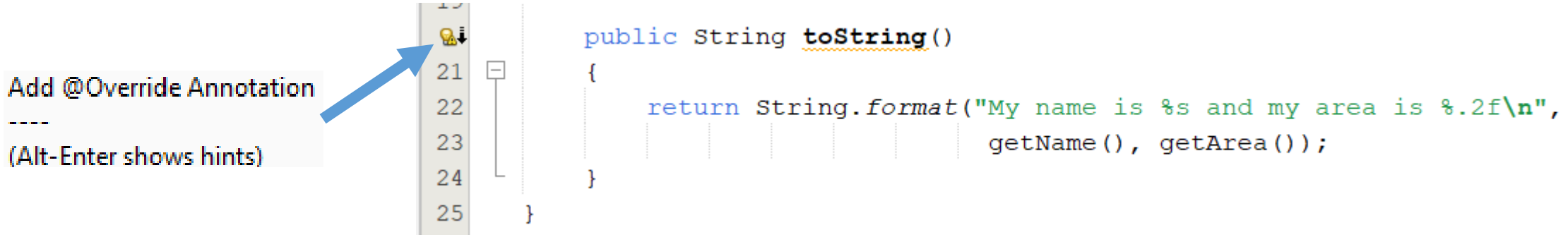
What about area?

Area is a calculation that is very specific to `Circle` so what happens if we call `getArea()` from inside `Shape`?

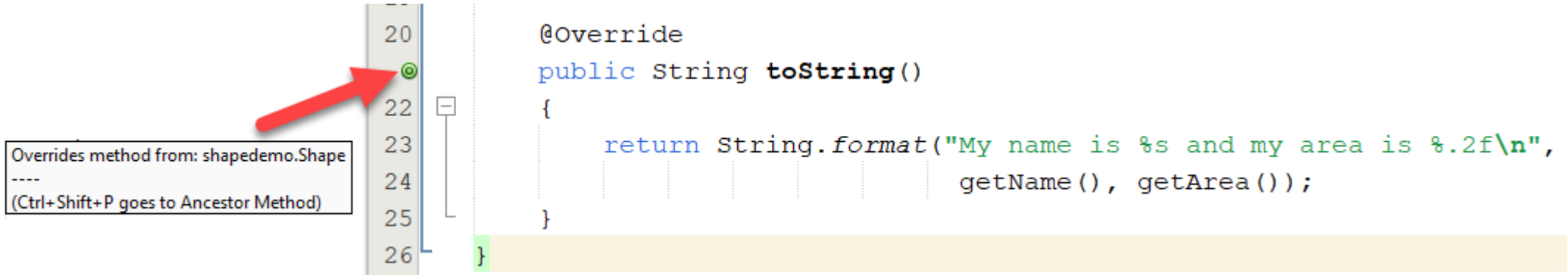
We could call `getArea()` since it is defined in `Shape`

```
public String toString()
{
    return String.format("My name is %s and my area is %.2f",
                         shapeName, getArea());
}
```

Defined in `Shape`



When we overloaded `toString()` inside `Shape`, the message to add `@Override` showed up in `Circle` because `Circle` already had a `toString()`.



So what happens when we override `toString()` in the superclass?

```
public String toString()
{
    return String.format("My name is %s and my area is %.2f",
                         shapeName, getArea());
}
```

and then run these statements?

```
System.out.printf("\nMy name is %s\n", A.getName());
System.out.println(A);

System.out.printf("\nMy name is %s and my area is %.2f\n", C.getName(), C.getArea());
System.out.println(C);

System.out.printf("\nMy name is %s and my area is %.2f\n", R.getName(), R.getArea());
System.out.println(R);

System.out.printf("\nMy name is %s and my area is %.2f\n", S.getName(), S.getArea());
System.out.println(S);
```

```
My name is Poly
My name is Poly and my area is 0.00

My name is Hoop and my area is 78.54
My name is Hoop and my area is 78.54

My name is NotQuiteSquare and my area is 14.40
My name is NotQuiteSquare and my area is 14.40

My name is Quad and my area is 11.56
My name is Quad and my area is 11.56
```

So how is the `toString()` in the superclass calling the subclass method to calculate the area of the right object?

When we moved `getArea()` to `Shape`, we kept the versions in the subclasses and marked them as overrides.

```
@Override
public double getArea()
{
    return Math.PI * Math.pow(getDim1(), 2);
}
```

```
@Override
public double getArea()
{
    return getDim1() * getDim2();
}
```

`Square` does not have its own area method – it uses the one it inherited from `Rectangle`.

When we called the superclass's `toString()`, we implicitly passed it the object.

You can think of the "`this`" reference being sent to the method.

When we invoked `getArea()`, the object used its version of `getArea()` instead of the superclass's version because the superclass's version has been overriden.

The `toString()` in `Shape` will use `Circle`'s version of `getArea()` instead of its own version of `getArea()` since a `Circle` object was passed implicitly to the `toString()` of the `Shape`.

Same thing for `Rectangle` and `Square`. The `toString()` in `Shape` will use the object's version of `getArea()` because of the override.

```
My name is Poly
My name is Poly and my area is 0.00

My name is Hoop and my area is 78.54
My name is Hoop and my area is 78.54

My name is NotQuiteSquare and my area is 14.40
My name is NotQuiteSquare and my area is 14.40

My name is Quad and my area is 11.56
My name is Quad and my area is 11.56
```

A `Shape` object printing an area of `0` does not really make sense.

When we call the `toString()` in `Shape` with a `Shape` object, the `getArea()` in `Shape` is used.

Maybe, it's not a great idea to put the printing of the name AND the area in the superclass.

So, we want the benefit of using the superclass's `toString()` to print the name since the superclass has direct access, but we don't want the superclass's `toString()` printing the area since that causes a `Shape` object to print an area of `0`.

```
public String toString()
{
    return String.format("My name is %s", shapeName);
}
```

We still want to be able to just print the object though and get the name and area.

```
System.out.println(A);
System.out.println(C);
System.out.println(R);
System.out.println(S);
```

without having to call `getName()` and/or `getArea()`

```
System.out.printf("\nMy name is %s\n", A.getName());
System.out.printf("\nMy name is %s and my area is %.2f\n", C.getName(), C.getArea());
System.out.printf("\nMy name is %s and my area is %.2f\n", R.getName(), R.getArea());
System.out.printf("\nMy name is %s and my area is %.2f\n", S.getName(), S.getArea());
```

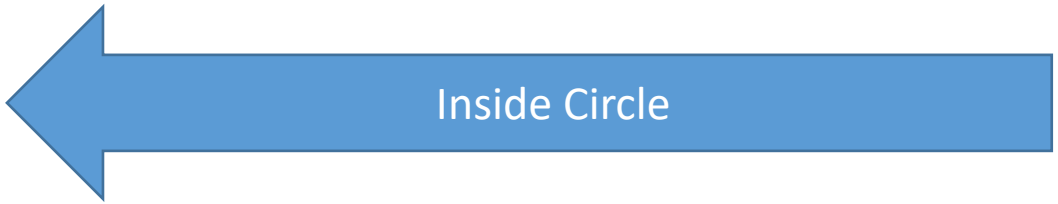
If we override `toString()` in the subclasses….

```
public String toString()
{
    return String.format("My name is %s and my area is %.2f\n", getName(), getArea());
}
```

Then we still have to use the getters

How do we use the `toString()` in `Shape` AND the `toString()` in `Circle`?

```
public String toString()
{
    return String.format("%s and my area is %.2f\n", super.toString(), getArea());
}
```



```
public String toString()
{
    return String.format("%s and my area is %.2f\n", super.toString(), getArea());
}
```

Placing the keyword **super** and a dot (.) separator before the superclass method name invokes the superclass version of the overridden method.

```
System.out.println(C);
System.out.printf("\nMy name is %s and my area is %.2f\n",
                  C.getName(), C.getArea());
```

```
My name is Hoop and my area is 78.54
```

So what did we get from moving `getArea()` to `Shape` other than the work of adding the `@Override` annotation to our classes?

Let's add another class `Triangle`

```
package shapedemo;

public class Triangle extends Shape
{
    Triangle(String name, double base, double height)
    {
        super(name);
        setDims(base, height);
    }

    public String toString()
    {
        return String.format("%s and my area is %.2f\n",
                             super.toString(), getArea());
    }
```

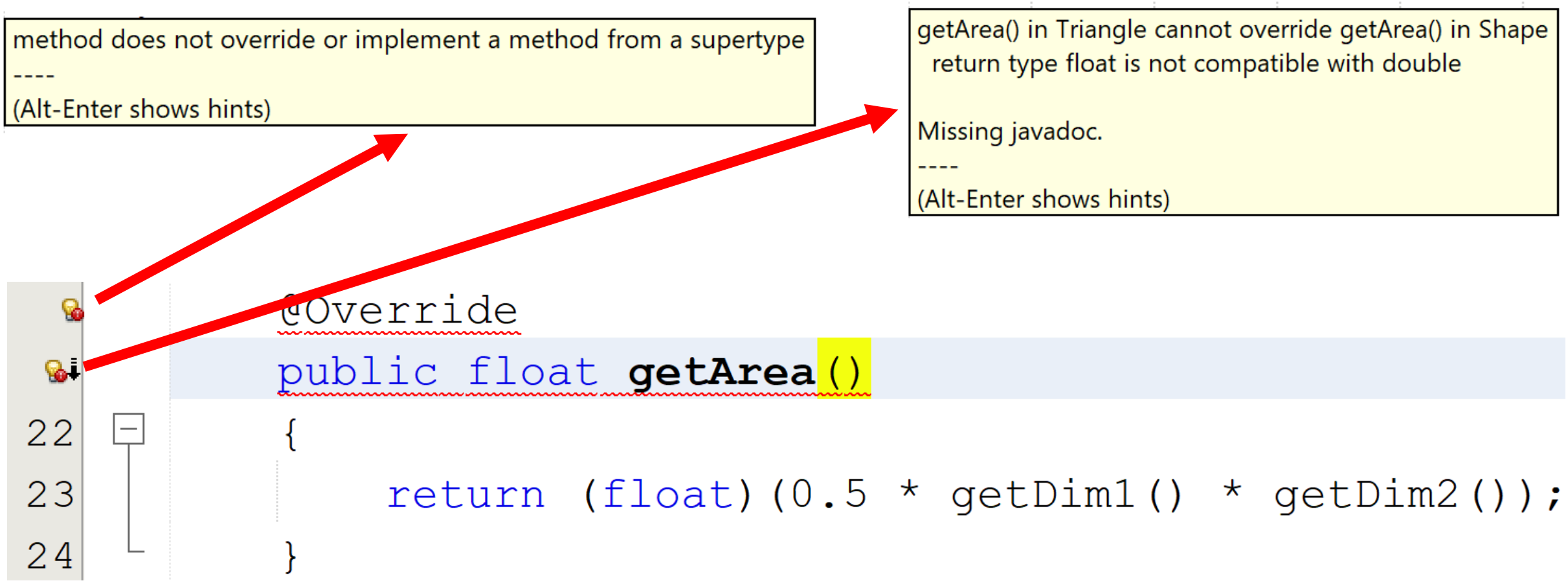
For whatever reason, it is decided that the `getArea()` method for `Triangle` should return a `float` instead of a `double`.

so instead of

```
@Override
public double getArea()
{
    return 0.5 * getDim1() * getDim2();
}
```

we want

```
@Override
public float getArea()
{
    return (float)(0.5 * getDim1() * getDim2());
}
```



Because `getArea()` was overridden, we are not allowed to change the return value.

Moving `getArea()` to `Shape` allows us to enforce the interface of the method.

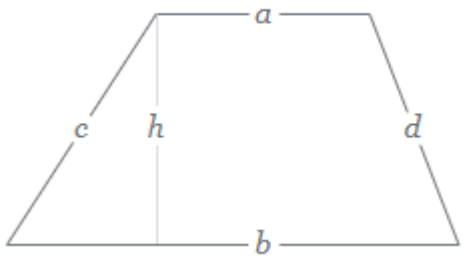
This will be a crucial feature in Polymorphism.

In Class Exercise

Add a new class to find the area of a trapezoid

Trapezoid

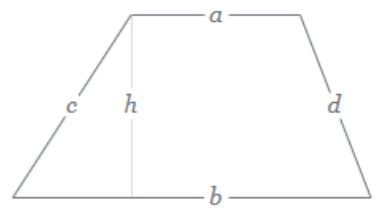
$$A = \frac{a+b}{2}h$$



```
package shapedemo;

public class Trapezoid extends Shape
{
    public Trapezoid(String name, double base1, double base2, double height)
    {
        super(name);
        setDims(base1, base2, height);
    }

    @Override
    public double getArea()
    {
        return (((getDim1() + getDim2())/2) * getDim3());
    }

    @Override
    public String toString()
    {
        return String.format("%s and my area is %.2f\n",
                             super.toString(), getArea());
    }
}
```

Trapezoid

$$A = \frac{a+b}{2}h$$



```
public void setDims(double Dim1, double Dim2)
{
    dim1 = Dim1;
    dim2 = Dim2;
}
public void setDims(double Dim1, double Dim2, double Dim3)
{
    dim1 = Dim1;
    dim2 = Dim2;
    dim3 = Dim3;
}
public double getDim1()
{
    return dim1;
}
public double getDim2()
{
    return dim2;
}
public double getDim3()
{
    return dim3;
}
```

```
Trapezoid Z = new Trapezoid("Trap", 2.3, 3.4, 4.5);


System.out.printf("\nMy name is %s and my area is %.2f\n",
                                  Z.getName(), Z.getArea());
System.out.println(Z);
```

```
My name is Trap and my area is 12.83
My name is Trap and my area is 12.83
```

Coding Assignment 3

`buyACoke` is not listed as having a return value in the class diagram.

The assignment states that it should return an enumerated value.

That enumeration is part of the implementation; therefore, cannot be listed as part of the class diagram.

`buyACoke` should have a return value of the enumeration

PIE

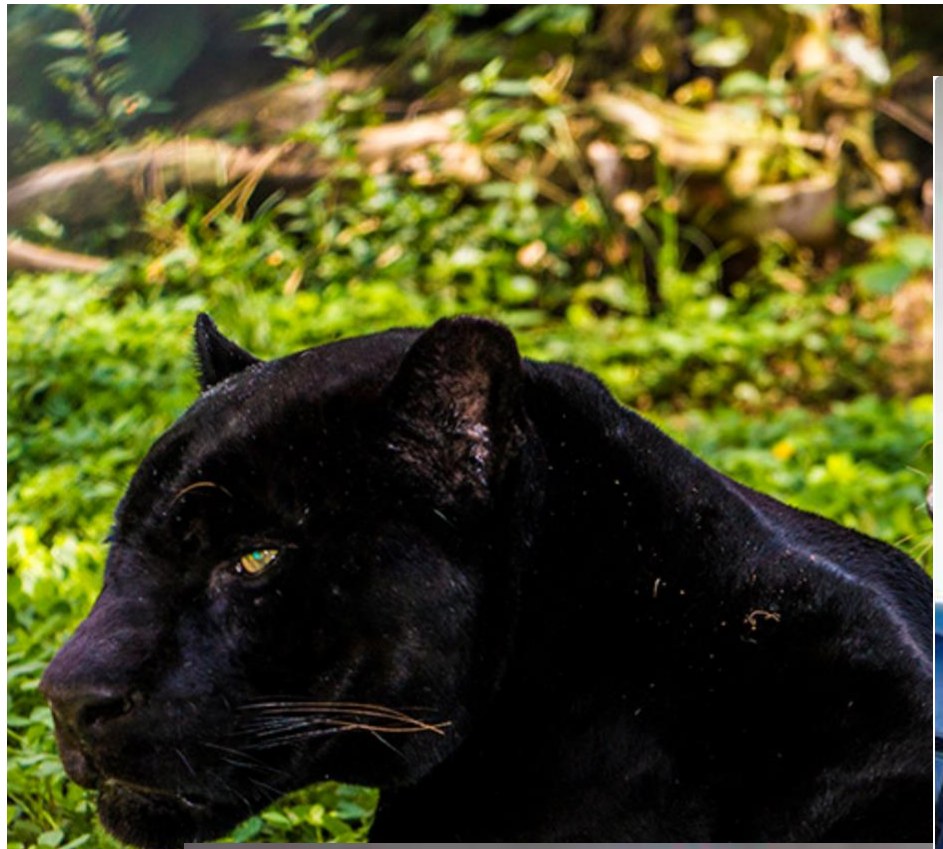
Polymorphism
Inheritance
Encapsulation

Polymorphism

Polymorphism occurs in biology.

Polymorphism in biology and zoology is the occurrence of two or more clearly different morphs or forms, also referred to as alternative phenotypes, in the population of a species.

To be classified as such, morphs must occupy the same habitat at the same time and belong to a panmictic population



Polymorphism

The word **polymorphism** means having many forms.

Typically, **polymorphism** occurs when there is a hierarchy of classes and they are related by inheritance.

Java polymorphism means that a call to an instance method will cause a different method to be executed depending on the type of object that invokes the method.

Polymorphism

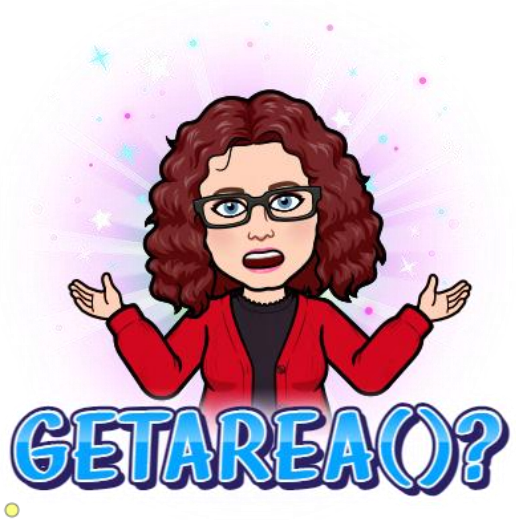
Polymorphism enables us to

program in general

rather than

program in the specific

Polymorphism allow us to rely on an object to know how to "do the right thing" when a given method is called even when that method is implemented differently in each subclass.



Polymorphism

Subclasses `Fish`, `Frog` and `Bird` are created from superclass `Animal`.

`Animal` contains a method called `move`. `Animal` maintains an animal's current location as x-y coordinates.

Each subclass implements its own version of `move()`.

The instantiations of `Fish`, `Frog` and `Bird` are kept in an `ArrayList` of type `Animal`.

```
public static void main(String[] args)
{
    ArrayList <Animal> Zoo = new ArrayList<>();
    Scanner in = new Scanner(System.in);
    String AnimalFile;

    System.out.print("Enter the filename of Fish ");
    AnimalFile = in.nextLine();
    ReadFile(AnimalFile, "Fish", Zoo);

    System.out.print("Enter the filename of Frog ");
    AnimalFile = in.nextLine();
    ReadFile(AnimalFile, "Frog", Zoo);

    System.out.print("Enter the filename of Bird ");
    AnimalFile = in.nextLine();
    ReadFile(AnimalFile, "Bird", Zoo);

    System.out.println(Zoo);
}
```

Frog.txt

glass,true,tree,poison dart,Kermit,South American,wood

Bird.txt

owl,hummingbird,penguin,heron,ostrich,crane,stork,Big

Fish.txt

goldfish,guppy,carp,trout,Nemo,oscar,catfish,dogfish,lionfish

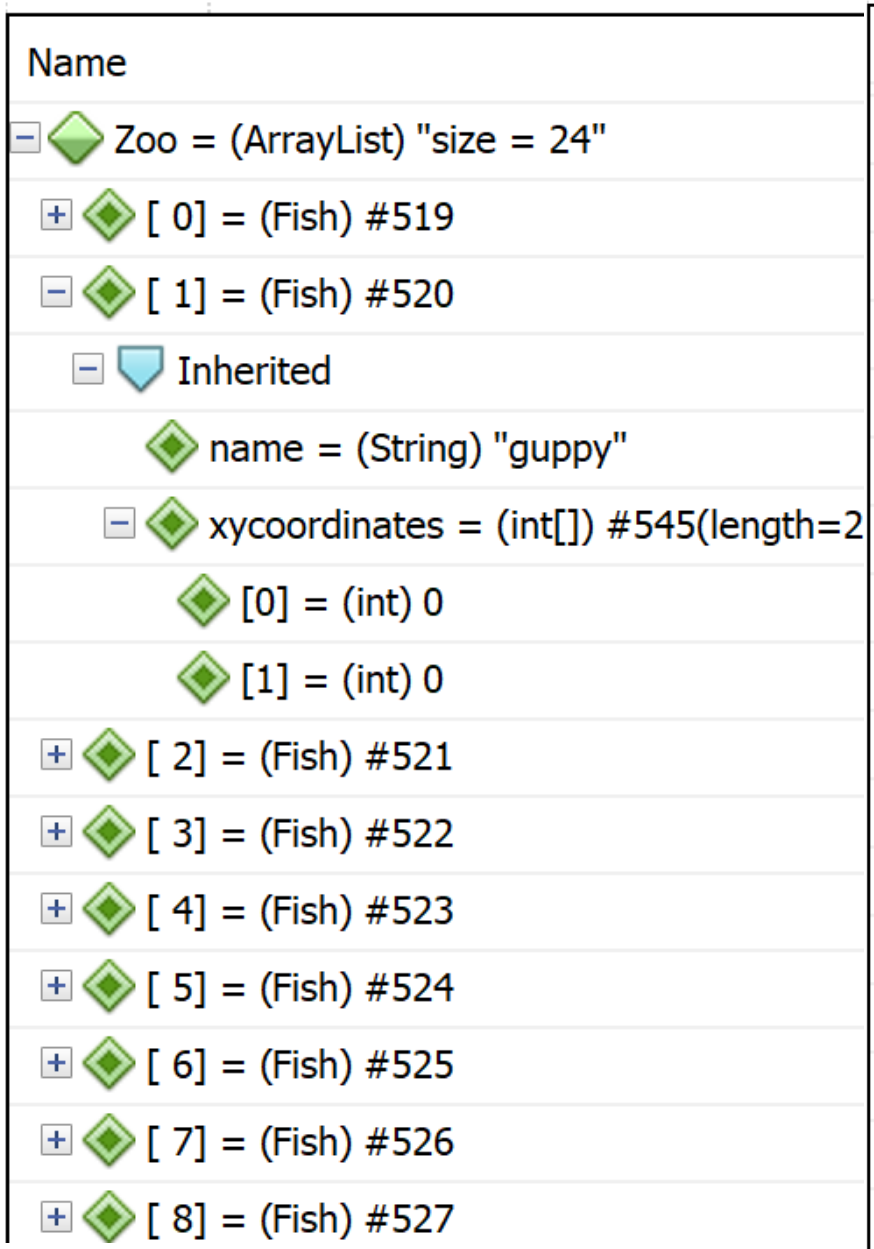
```
public static void ReadFile(String filename, String AnimalType, ArrayList <Animal> Zoo)
{
    File FH  = new File(filename);
    Scanner FileReader = null;

    try
    {
        FileReader = new Scanner(FH);
    }
    catch (Exception e)
    {
        System.out.printf("%s file name does not exist...exiting\n", filename);
        System.exit(0);
    }
```

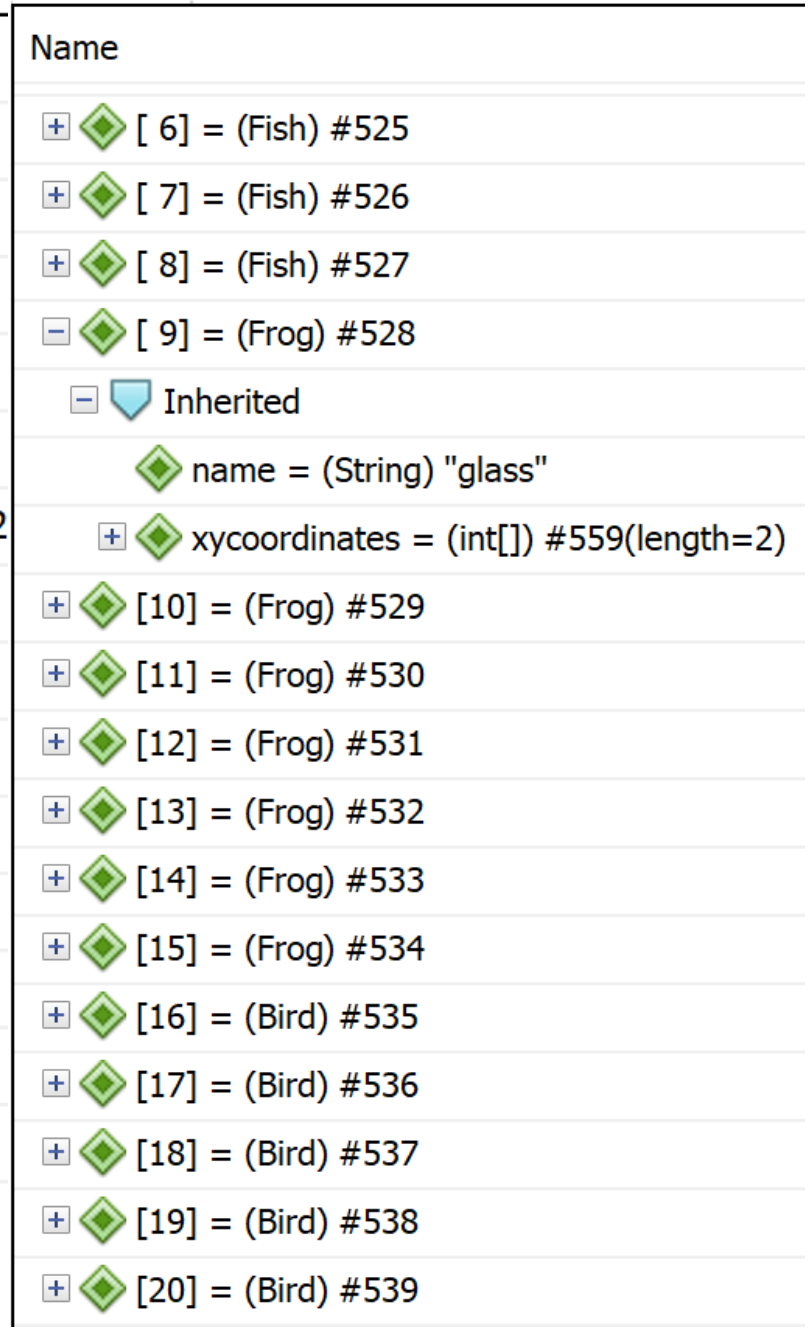
```
    String FileLine[] = FileReader.nextLine().split(",");

    switch(AnimalType)
    {
        case "Fish" :
            for(String it : FileLine)
                Zoo.add(new Fish(it));
            break;
        case "Frog" :
            for(String it : FileLine)
                Zoo.add(new Frog(it));
            break;
        case "Bird" :
            for(String it : FileLine)
                Zoo.add(new Bird(it));
            break;

    }

    FileReader.close();
}
```

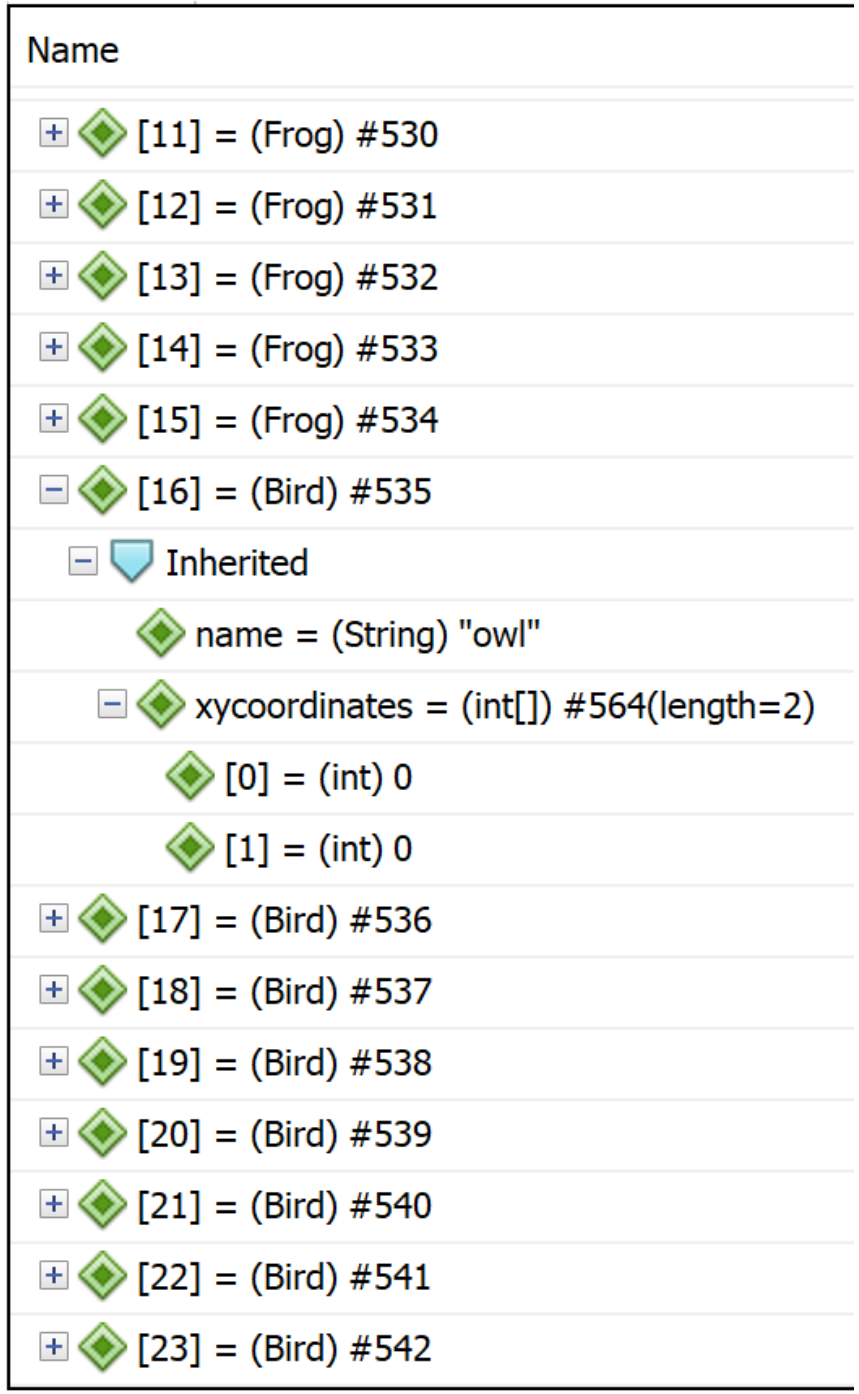
```
ReadFile(AnimalFile, "Fish", Zoo);
ReadFile(AnimalFile, "Frog", Zoo);
ReadFile(AnimalFile, "Bird", Zoo);
```

```
System.out.println(Zoo);
```

```
[animaldemo.Fish@30dae81, animaldemo.Fish@1b2c6ec2, animaldemo.Fish@4edde6e5,
animaldemo.Fish@70177ecd, animaldemo.Fish@1e80bfe8, animaldemo.Fish@66a29884,
animaldemo.Fish@4769b07b, animaldemo.Fish@cc34f4d, animaldemo.Fish@17a7cec2,
animaldemo.Frog@65b3120a, animaldemo.Frog@6f539caf, animaldemo.Frog@79fc0f2f,
animaldemo.Frog@50040f0c, animaldemo.Frog@2dda6444, animaldemo.Frog@5e9f23b4,
animaldemo.Frog@4783da3f, animaldemo.Bird@378fd1ac, animaldemo.Bird@49097b5d,
animaldemo.Bird@6e2c634b, animaldemo.Bird@37a71e93, animaldemo.Bird@7e6cbb7a,
animaldemo.Bird@7c3df479, animaldemo.Bird@7106e68e, animaldemo.Bird@7eda2dbb]
```


Name
Zoo = (ArrayList) "size = 24"
[0] = (Fish) #519
[1] = (Fish) #520
Inherited
name = (String) "guppy"
xycoordinates = (int[]) #545(length=2
[0] = (int) 0
[1] = (int) 0
[2] = (Fish) #521
[3] = (Fish) #522
[4] = (Fish) #523
[5] = (Fish) #524
[6] = (Fish) #525
[7] = (Fish) #526
[8] = (Fish) #527


Name
[6] = (Fish) #525
[7] = (Fish) #526
[8] = (Fish) #527
[9] = (Frog) #528
Inherited
name = (String) "glass"
xycoordinates = (int[]) #559(length=2)
[10] = (Frog) #529
[11] = (Frog) #530
[12] = (Frog) #531
[13] = (Frog) #532
[14] = (Frog) #533
[15] = (Frog) #534
[16] = (Bird) #535
[17] = (Bird) #536
[18] = (Bird) #537
[19] = (Bird) #538
[20] = (Bird) #539


Name
[11] = (Frog) #530
[12] = (Frog) #531
[13] = (Frog) #532
[14] = (Frog) #533
[15] = (Frog) #534
[16] = (Bird) #535
Inherited
name = (String) "owl"
xycoordinates = (int[]) #564(length=2)
[0] = (int) 0
[1] = (int) 0
[17] = (Bird) #536
[18] = (Bird) #537
[19] = (Bird) #538
[20] = (Bird) #539
[21] = (Bird) #540
[22] = (Bird) #541
[23] = (Bird) #542

```
package animaldemo;

public class Animal
{
    private String name;
    private int xycoordinates[] = new int [2];

    Animal(String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return name;
    }
}
```

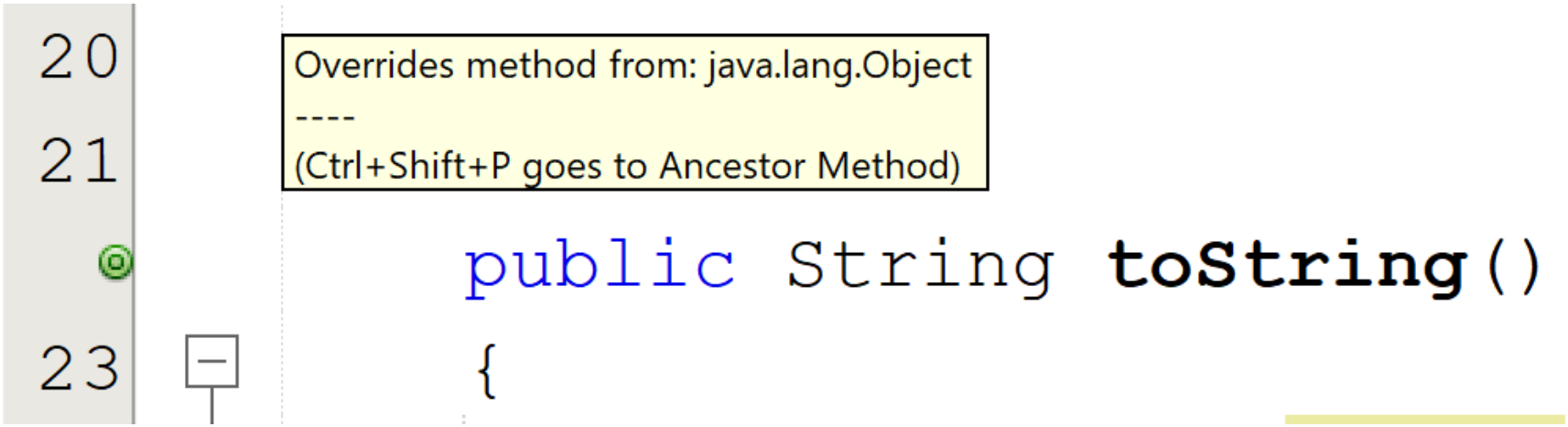
By adding the override of `toString()` to the superclass, all of the subclasses can use it.

```
@Override
public String toString()
{
    return String.format("%s", name);
}
```

`toString()` is in `Animal`; therefore, we have direct access to `name`

```
Enter the filename of Fish Fish.txt
Enter the filename of Frog Frog.txt
Enter the filename of Bird Bird.txt
[goldfish, guppy, carp, trout, Nemo, oscar, catfish, dogfish,
lionfish, glass, true, tree, poison dart, Kermit, South
American, wood, owl, hummingbird, penguin, heron, ostrich,
crane, stork, Big]
```

```
@Override
public String toString()
{
    return String.format("\n%s is at %d-%d",
                         name, xycoordinates[0], xycoordinates[1]);
}
```




```
public String toString()
{
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

```
[
goldfish is at 0-0,
guppy is at 0-0,
carp is at 0-0,
trout is at 0-0,
Nemo is at 0-0,
oscar is at 0-0,
catfish is at 0-0,
dogfish is at 0-0,
lionfish is at 0-0,
glass is at 0-0,
true is at 0-0,
tree is at 0-0,
poison dart is at 0-0,
Kermit is at 0-0,
South American is at 0-0,
wood is at 0-0,
owl is at 0-0,
hummingbird is at 0-0,
penguin is at 0-0,
heron is at 0-0,
ostrich is at 0-0,
crane is at 0-0,
stork is at 0-0,
Big is at 0-0]
```

```
[
goldfish is at 1-1,
guppy is at 0-0,
carp is at 1-1,
trout is at 2-1,
Nemo is at 1-2,
oscar is at 2-2,
catfish is at 0-2,
dogfish is at 0-0,
lionfish is at 1-1,
glass is at 0-3,
true is at 0-2,
tree is at 2-3,
poison dart is at 4-2,
Kermit is at 0-3,
South American is at 3-3,
wood is at 2-2,
owl is at 1-7,
hummingbird is at 4-3,
penguin is at 7-6,
heron is at 6-2,
ostrich is at 2-0,
crane is at 0-1,
stork is at 6-3,
Big is at 8-9]
```

```
public String toString()
{
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

```
[animaldemo.Fish@30dae81, animaldemo.Fish@1b2c6ec2, animaldemo.Fish@4edde6e5,
animaldemo.Fish@70177ecd, animaldemo.Fish@1e80bfe8, animaldemo.Fish@66a29884,
animaldemo.Fish@4769b07b, animaldemo.Fish@cc34f4d, animaldemo.Fish@17a7cec2,
animaldemo.Frog@65b3120a, animaldemo.Frog@6f539caf, animaldemo.Frog@79fc0f2f,
animaldemo.Frog@50040f0c, animaldemo.Frog@2dda6444, animaldemo.Frog@5e9f23b4,
animaldemo.Frog@4783da3f, animaldemo.Bird@378fd1ac, animaldemo.Bird@49097b5d,
animaldemo.Bird@6e2c634b, animaldemo.Bird@37a71e93, animaldemo.Bird@7e6cbb7a,
animaldemo.Bird@7c3df479, animaldemo.Bird@7106e68e, animaldemo.Bird@7eda2dbb]
[animaldemo.Fish@30dae81, animaldemo.Fish@1b2c6ec2, animaldemo.Fish@4edde6e5,
animaldemo.Fish@70177ecd, animaldemo.Fish@1e80bfe8, animaldemo.Fish@66a29884,
animaldemo.Fish@4769b07b, animaldemo.Fish@cc34f4d, animaldemo.Fish@17a7cec2,
animaldemo.Frog@65b3120a, animaldemo.Frog@6f539caf, animaldemo.Frog@79fc0f2f,
animaldemo.Frog@50040f0c, animaldemo.Frog@2dda6444, animaldemo.Frog@5e9f23b4,
animaldemo.Frog@4783da3f, animaldemo.Bird@378fd1ac, animaldemo.Bird@49097b5d,
animaldemo.Bird@6e2c634b, animaldemo.Bird@37a71e93, animaldemo.Bird@7e6cbb7a,
animaldemo.Bird@7c3df479, animaldemo.Bird@7106e68e, animaldemo.Bird@7eda2dbb]
```

```
@Override          `
public String toString()
{
    return String.format("\n%s %s is at %d-%d",
                         getClass().getSimpleName(),
                         name, xycoordinates[0],
                         xycoordinates[1]);
}
```

Our `Animal` class is overriding `Object`'s `toString`() method.

When we call `toString()` for a given object, the object is implicitly passed to the method.

`toString()` retrieves the `name` and `xycoordinates` for the implicitly object and also calls `getClass().getSimpleName()` for the implicit object.

```
[
Fish goldfish is at 0-0,
Fish guppy is at 0-0,
Fish carp is at 0-0,
Fish trout is at 0-0,
Fish Nemo is at 0-0,
Fish oscar is at 0-0,
Fish catfish is at 0-0,
Fish dogfish is at 0-0,
Fish lionfish is at 0-0,
Frog glass is at 0-0,
Frog true is at 0-0,
Frog tree is at 0-0,
Frog poison dart is at 0-0,
Frog Kermit is at 0-0,
Frog South American is at 0-0,
Frog wood is at 0-0,
Bird owl is at 0-0,
Bird hummingbird is at 0-0,
Bird penguin is at 0-0,
Bird heron is at 0-0,
Bird ostrich is at 0-0,
Bird crane is at 0-0,
Bird stork is at 0-0,
Bird Big is at 0-0]
```

To simulate the animals' movements, the program sends each object the same message once per second

- a Fish might swim three feet
- a Frog might jump five feet
- a Bird might fly ten feet.

We'll create a method in `Animal` called `move()`

```
public void move()
{

}
```

Each instance of `Animal` (each subclass) moves differently

- a Fish might swim three feet
- a Frog might jump five feet
- a Bird might fly ten feet.

We add `move()` to `Animal` and then override it in each subclass.

Why??

To guarantee that all objects instantiated from `Animal` use the same interface for `move()`. If we did not, we could not do this…

```
for (Animal it : Zoo)
{
    it.move();
}
```

```
for (Animal it : Zoo)
{
    it.move();
}
```

The same message sent to a variety of objects has "many forms" of results

Polymorphism

```
@Override
public void move()
{
    // Fish swim 3 feet every second
    Random rn = new Random();
    xycoordinates[0] += rn.nextInt(4);
    xycoordinates[1] += rn.nextInt(4);
}
```

This will create a random number between 0 and 3

Since a fish can swim 3 feet (or less) per second, we get a random number between 0 and 3.

We add that random value to the existing coordinates to indicate movement.

```
@Override
public void move()
{
    // Birds fly 10 feet every second
    Random rn = new Random();
    xycoordinates[0] += rn.nextInt(11);
    xycoordinates[1] += rn.nextInt(11);
}

@Override
public void move()
{
    // Frogs jump 5 feet every second
    Random rn = new Random();
    xycoordinates[0] += rn.nextInt(6);
    xycoordinates[1] += rn.nextInt(6);
}
```

Action Items

Mon, Oct 24

Due 11:59pm	Crash Course : Quiz 8
Due 11:59pm	Coding Assignment 3
Due 11:59pm	Homework 5