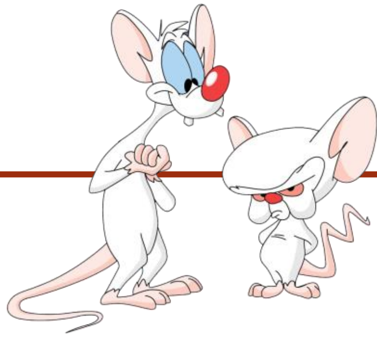
COMP 250

INTRODUC TER SCIENCE

Week 4-4 : OOD7

Giulia Alberini, Fall 2020

WHAT ARE WE GOING TO DO IN THIS VIDEO?



OOD7

- `instanceof`
- Intro to Polymorphism
- Abstract Classes and Methods

A LITTLE ABOUT `instanceof`

- The `instanceof` operator is used to test whether an object is an instance of the specified type.
- It returns either `true` or `f` `stanceof` operator with any variable that has `null` val

```
Dog myDog = new Dog();
Beagle snoopy = new Beagle();
Dog aDog = null;
System.out.println(myDog instanceof Dog); // true
System.out.println(snoopy instanceof Dog); // true
System.out.println(aDog instanceof Dog); // false
```

instanceof AND DOWNCASTING

- When can use `instanceof` to make sure that downcasting to a subclass will not cause a run time error.

```
public static void myMetho          og) {
    if(myDog instanceof Bea
        Beagle b = (Beagle) myDog; // downcasting
        b.hunt();
    }
}
```

instanceof AND equals()

- Note that in general we want to use `instanceof` as a last resort. We'll see why shortly.
- That said, we have to use `instanceof` when overriding `equals()`

```
public clas
    Person owner;
    :
    public boolean equals(Object obj) {
        if(obj instanceof Dog) {
            ...
        }
    }
}
```

WHERE WE LEFT OFF

class Dog

```
Person owner
public void bark() {
    print("woof!");
}
    :
```

↑ extends

class Beagle

```
void hunt ()
public void bark() {
    print("aowwwuuu");
}
    :
```

```
public class Test {
   public static void main(String[] args) {
      Dog snoopy = new Beagle();
      snoopy.bark();
   }
}
```

Is this allowed??

Yes, it's an example of upcasting!

If so, which `bark()` will execute???

P M

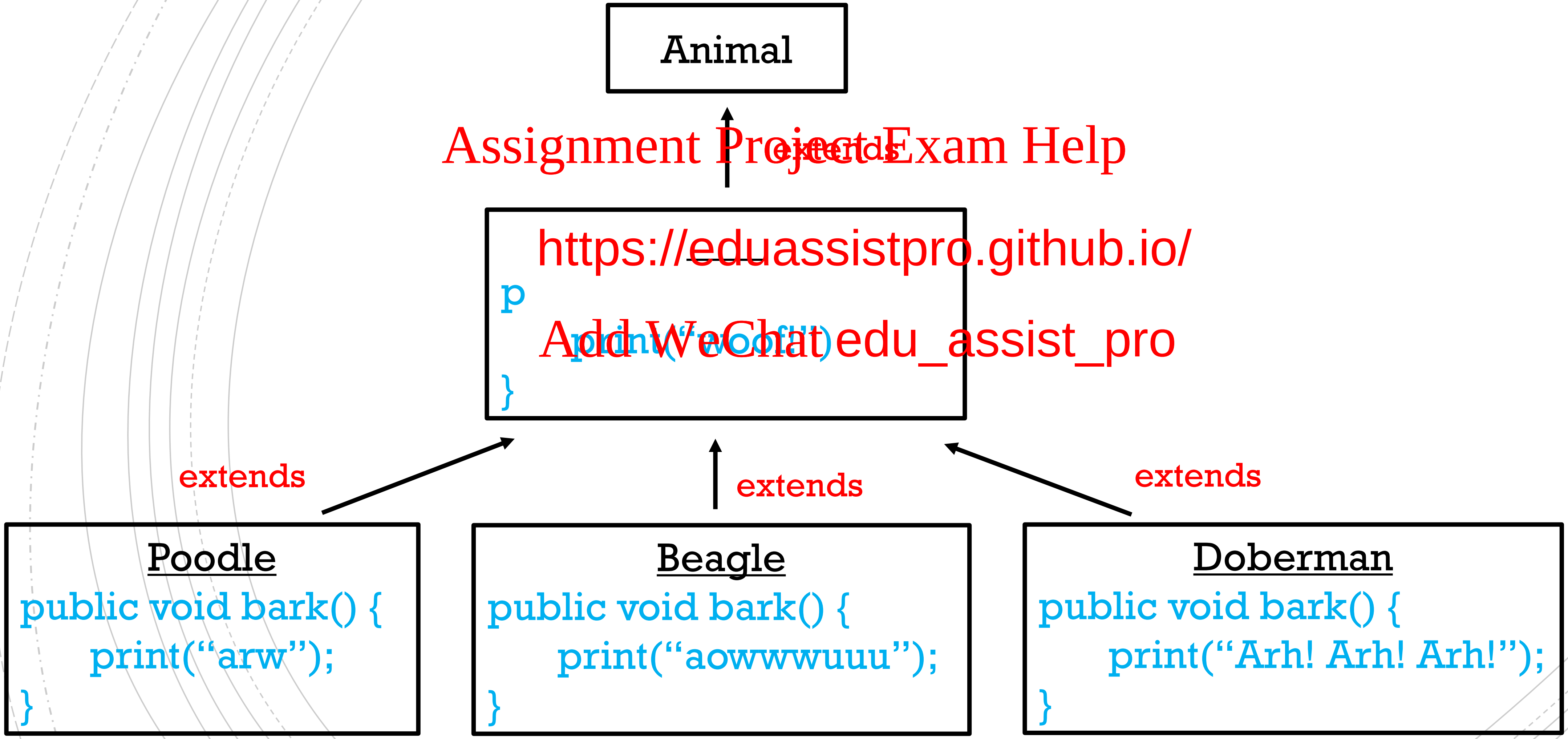
POLYMORPHISM

- Each object can have different "forms".

- One important aspect of "the Java virtual machine (JVM) calls the appropriate method for that is referred to in each variable. It does not call the method th d by the variable's type".

- More general discussion about polymorphism in higher level courses. (e.g. COMP 302)

RECALL HIERARCHY FROM OUR EXAMPLES



```
p
    print("woof!")
}
```

Poodle
```
public void bark() {
    print("arw");
}
```

Beagle
```
public void bark() {
    print("aowwwuuu");
}
```

Doberman
```
public void bark() {
    print("Arh! Arh! Arh!");
}
```

EXAMPLE

```
Dog myDog = new Dog();
myDog.bark();
Dog snoopy = new Beagl
snoopy.bark();
```

The compiler sees `bark()` in the `Dog` class (`myDog` is of type `Dog`) at compile time, the JVM invokes `bark()` from the `Dog` class at run g points to an object of type `Dog`).

At compile time, the compiler uses `bark()` in the `Dog` class to validate the statement. At run time, however, the JVM invokes `bark()` from the `Beagle` class. (`snoopy` is actually referring to a `Beagle` object)

OUTPUT

```
woof!
aowwwuuu
```

THE "OO WAY"

- Favor polymorphism and dynamic binding to downcasting and `instanceOf`.

- From *Effective C++*, by Scott Mey

"Anytime you find yourself writing code of the form 'if the object is of type T1, then do something, but if it's of type T2, then do something else', slap yourself".

TRY IT!

- Let’s go back to our Shape classes. On the first video I suggested you to [illegible] Info() in the Shape class and override [illegible] gle. Let’s then create an array of Shapes and see how [illegible] loit polymorphism when displaying the info of the elements in the array.

LET'S LOOK AT AN EXAMPLE

- Suppose we created the following two classes to work with Circles and Triangles.

<u>Circle</u>

- color: String
- radius: double
+ getColor(): String
+ setColor(c:String)
+ getRadius():double
+ getArea(): double

- color: String
- base: double
- height: double
+ getColor(): String
+ setColor(c:String)
+getArea(): double

LET'S LOOK AT AN EXAMPLE

- Suppose we created the following two classes to work with Circles and Triangles.

<u>Circle</u>

- color: String
- radius: double
+ getColor(): String
+ setColor(c:String)
+ getRadius():double
+ getArea(): double

[illegible]

- color: String
- base: double
- height: double
+ getColor(): String
+ setColor(c:String)
+getArea(): double

Observations:

- The two classes are closely related. They are both used to represent geometrical shapes.

LET'S LOOK AT AN EXAMPLE

- Suppose we created the following two classes to work with Circles and Triangles.

Circle

- color: String
- radius: double
+ getColor(): String
+ setColor(c:String)
+ getRadius():double
+ getArea(): double

- color: String
- base: double
- height: double
+ getColor(): String
+ setColor(c:String)
+getArea(): double

Observations:

- There's code that is repeated: the two classes share fields and methods that are implemented in the same way.

LET'S LOOK AT AN EXAMPLE

- Suppose we created the following two classes to work with Circles and Triangles.

```
Circle
- color: String
- radius: double
+ getColor(): String
+ setColor(c:String)
+ getRadius():double
+ getArea(): double
```

```
- color: String
- base: double
- height: double
+ getColor(): String
+ setColor(c:String)
+getArea(): double
```

Observations:

- There's a method that serves the same purpose in both classes, but it's implemented differently depending on the class.

LET'S LOOK AT AN EXAMPLE

- Suppose we created the following two classes to work with Circles and Triangles.

Circle

- color: String

- radius: double

+ getColor(): String

+ setColor(c:String)

+ getRadius():double

+ getArea(): double

- color: String

- base: double

- height: double

+ getColor(): String

+ setColor(c:String)

+getArea(): double

Observations:

- There are fields and methods that are specific to each class.

LET'S LOOK AT AN EXAMPLE

- Suppose we created the following two classes to work with Circles and Triangles.

<u>Circle</u>

- color: String
- radius: double
+ getColor(): String
+ setColor(c:String)
+ getRadius():double
+ getArea(): double

- color: String
- base: double
- height: double
+ getColor(): String
+ setColor(c:String)
+getArea(): double

Observations:

- It is the perfect situation to create an abstract superclass!

abstract METHODS

- If you want a class to contain a particular method, but you would like the implementation of this method to be specified by the subclasses, then you can declare the method to be `abstract`.

- An `abstract` method is a method that is ithout implementation:

```
public abstract double getArea();
```

The method has no body! Instead of the curly braces, we use the semicolon at the end of the header.

`abstract` METHODS

Declaring a method as abstract has 2 consequences:

- The class containing `abstract`.

- Every subclass of the current class MUST either override the abstract method or declare it itself as `abstract`.

abstract CLASSES

- An abstract class must be declared using the `abstract` keyword.
- It can have `abstract` and non-`abstract` methods.

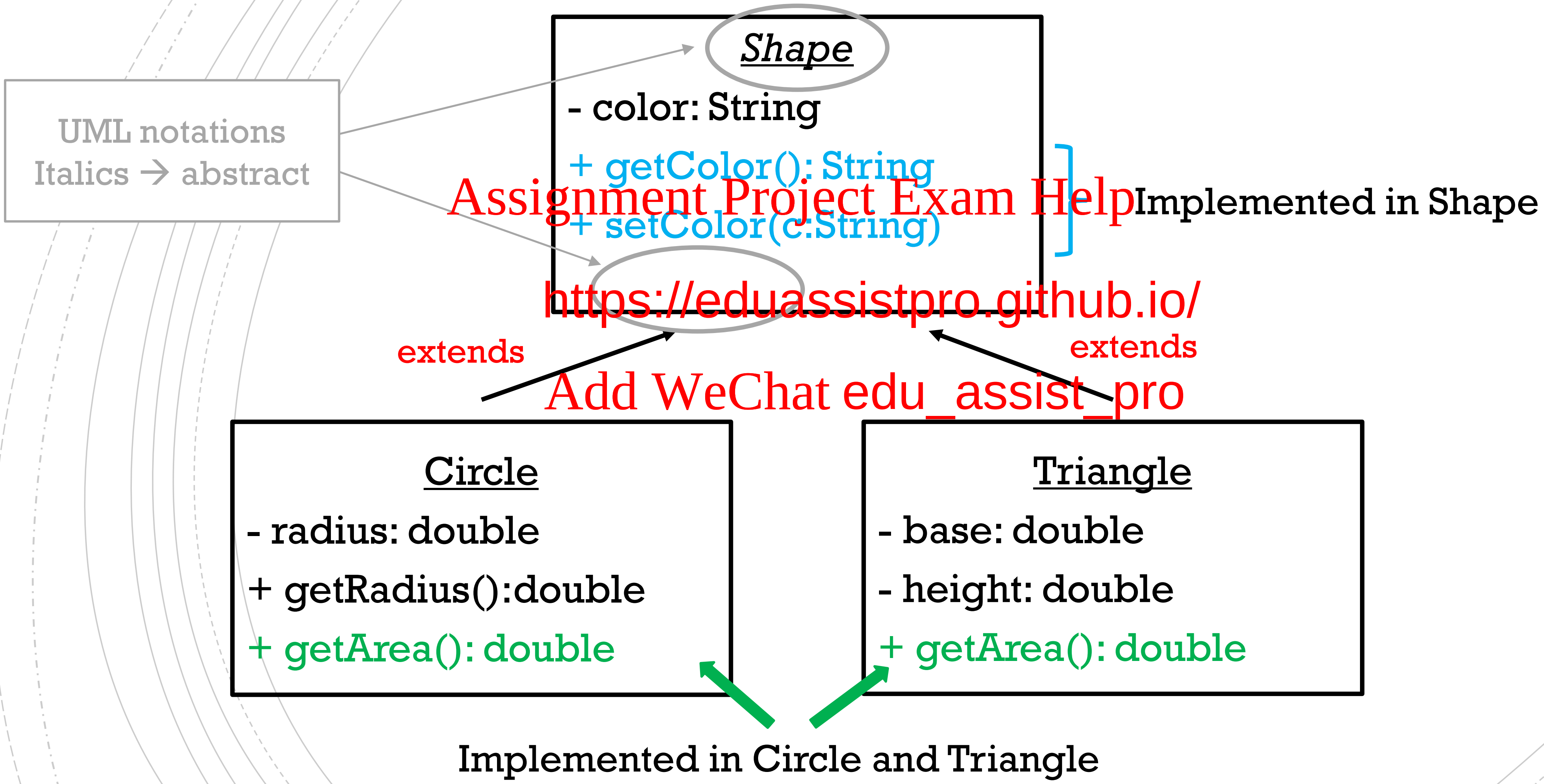
- It cannot be instant
- It can have constructors and static methods.
- It can have `final` methods which will force the subclass not to change the body of the method

`abstract` CLASSES – OBSERVATIONS

- We can have abstract classes with no abstract methods. This allow us to create instantiated, but can only be inherited

- We cannot instantiate an abstract class, but we can define constructors. These constructors are called when an instance of a subclass is created.

BACK TO OUR EXAMPLE

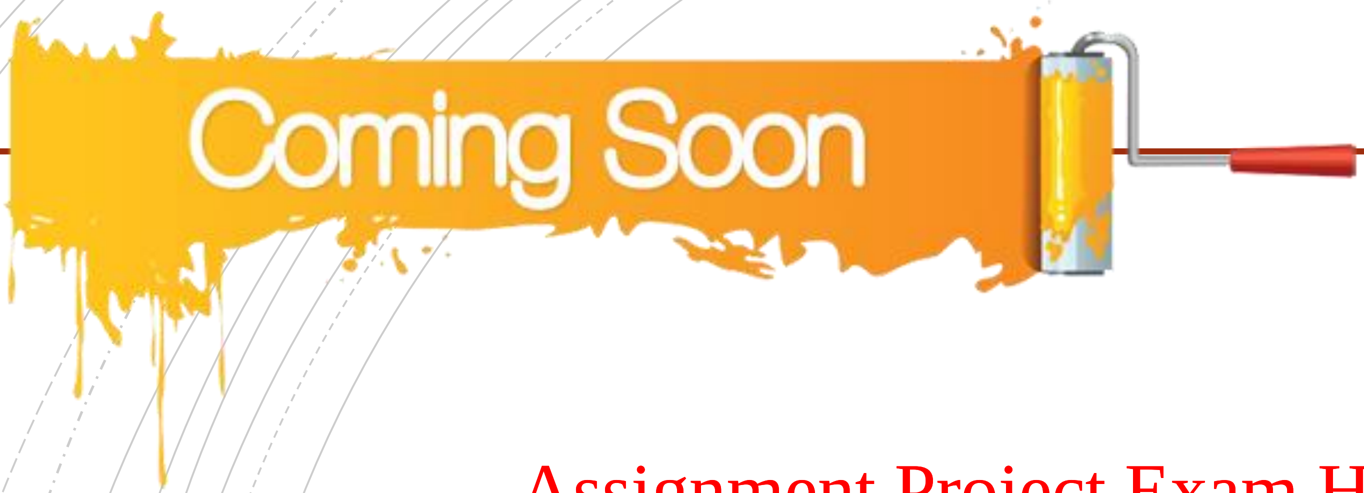


TRY IT!

- Go back to the Shape class and modified it by adding an abstract getArea() method.

- Add constructors to t

- Play around with the classes!



In the next

- Linear d s and linked lists!