COMP 250
INTRODUCTION TO COMPUTER SCIENCE

Lecture 8 – OOD4 Polymorphism and Abstract Classes

Giulia Alberini, Fall 2018

McGill STEM Support Committee

Brunch

Friday, September 28th
9:45 to 11:30AM

Rutherford Boardroom - room 105

Come hear faculty speakers and talk about equity, diversity and student well-being in STEM while enjoying FREE food.

ALL students and staff welcome!

ANNOUNCEMENTS

- Assignment 1 is out!
  - Due on Monday October 8th at 23:59

FROM LAST CLASS

- **The `Object` class**
  - `hashCode()`
  - `toString()`
  - `equals()`
  - `clone()`
- **Type conversion: upcasting and downcasting**

TYPE CASTING – REFERENCE TYPES

- Casting allows us to use an object of one type in place of another type, if permitted.

- For example we can write

```
Animal myPet = new Dog();
```

This will not cause a compile-time error because there is an ***implicit upcasting*** since a `Dog` is for sure also an `Animal`.

TYPE CASTING – REFERENCE TYPES

On the other hand, consider the following

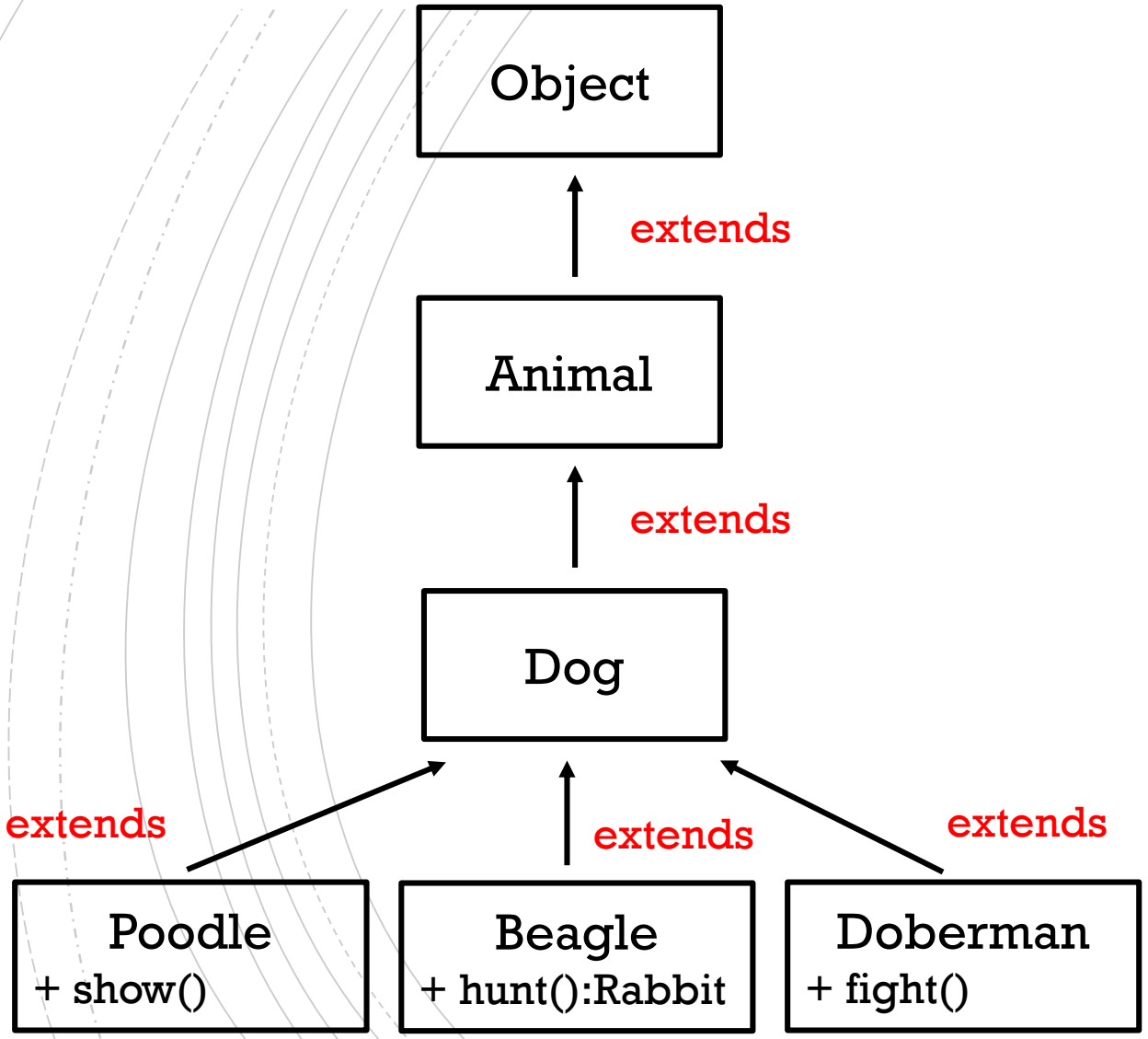
```
Animal myPet = new Dog();
Dog myDog = myPet;
```

The second line will cause a compile-time error. From the compiler point of view, `myPet` is of type `Animal` and an `Animal` might not be a `Dog`.
However, we can tell the compiler that `myPet` is of the correct type, by ***explicitly downcasting***:

```
Dog myDog = (Dog) myPet;
```

If `myPet` turns out to be of the wrong type we'll get a run-time error.

HIERARCHY FROM LAST CLASS



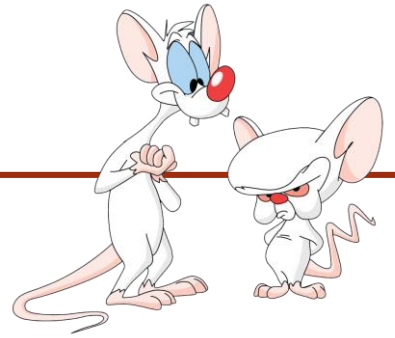
Upcasting

Happens automatically

Downcasting

The programmer has to manually do it.

WHAT ARE WE GOING TO DO TODAY?



- `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 `false`. If we apply the `instanceof` operator with any variable that has `null` value, it returns false.**

```
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 myMethod(Dog myDog) {
   if(myDog instanceof Beagle) {
      Beagle b = (Beagle) myDog; // downcasting
      b.hunt();
   }
}
```

instanceof AND equals()

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

```
public class Dog {
    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!

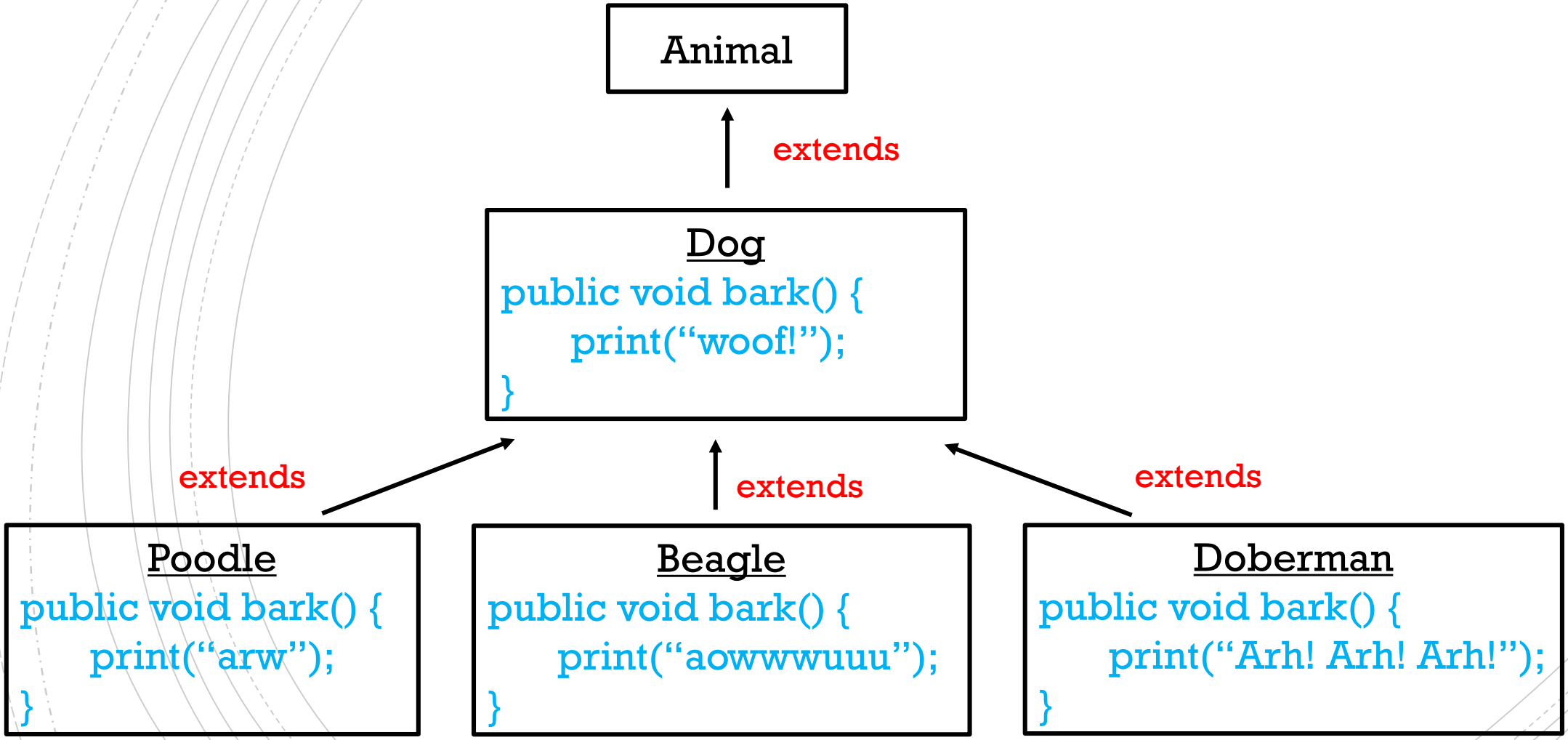
If so, which `bark()` will execute???

POLYMORPHISM

POLYMORPHISM

- Each object can have different "forms".
- One important aspect of polymorphism in Java: "the Java virtual machine (JVM) calls the appropriate method for the object that is referred to in each variable. It does not call the method that is defined by the variable's type".
- More general discussion about polymorphism in higher level courses. (e.g. COMP 302)

RECALL HIERARCHY FROM OUR EXAMPLES



EXAMPLE

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

OUTPUT

```
woof!
aowwwuuu
```

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 time (`myDog` 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)

THE "OO WAY"

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

- From *Effective C++*, by Scott Meyers:

  "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".

HOW DOES (RUN TIME) POLYMORPHISM WORK?

To answer this question, we first need to understand how classes are represented in a running program. To do so, we have to talk about the `Class` class. If time permits, we will do so in a couple of weeks when we come back to OOD (Oct. 15th)

TRY IT!

- Let's go back to our `Shape` classes. Last time we wrote a method `displayInfo()` in the `Shape` class and we overrode it in `Circle` and `Triangle`. Let's now create an array of `Shape`s and see how we can exploit polymorphism when displaying the info of the elements in the array.

abstract

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

Triangle
- 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.

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

Triangle
- 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

Triangle
- 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

Triangle
- 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

Triangle
- 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.

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

Triangle
- 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 declared without 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 it must be also declared `abstract`.
- Every subclass of the current class MUST either override the abstract method or declare it itself as `abstract`.

`abstract` CLASSES

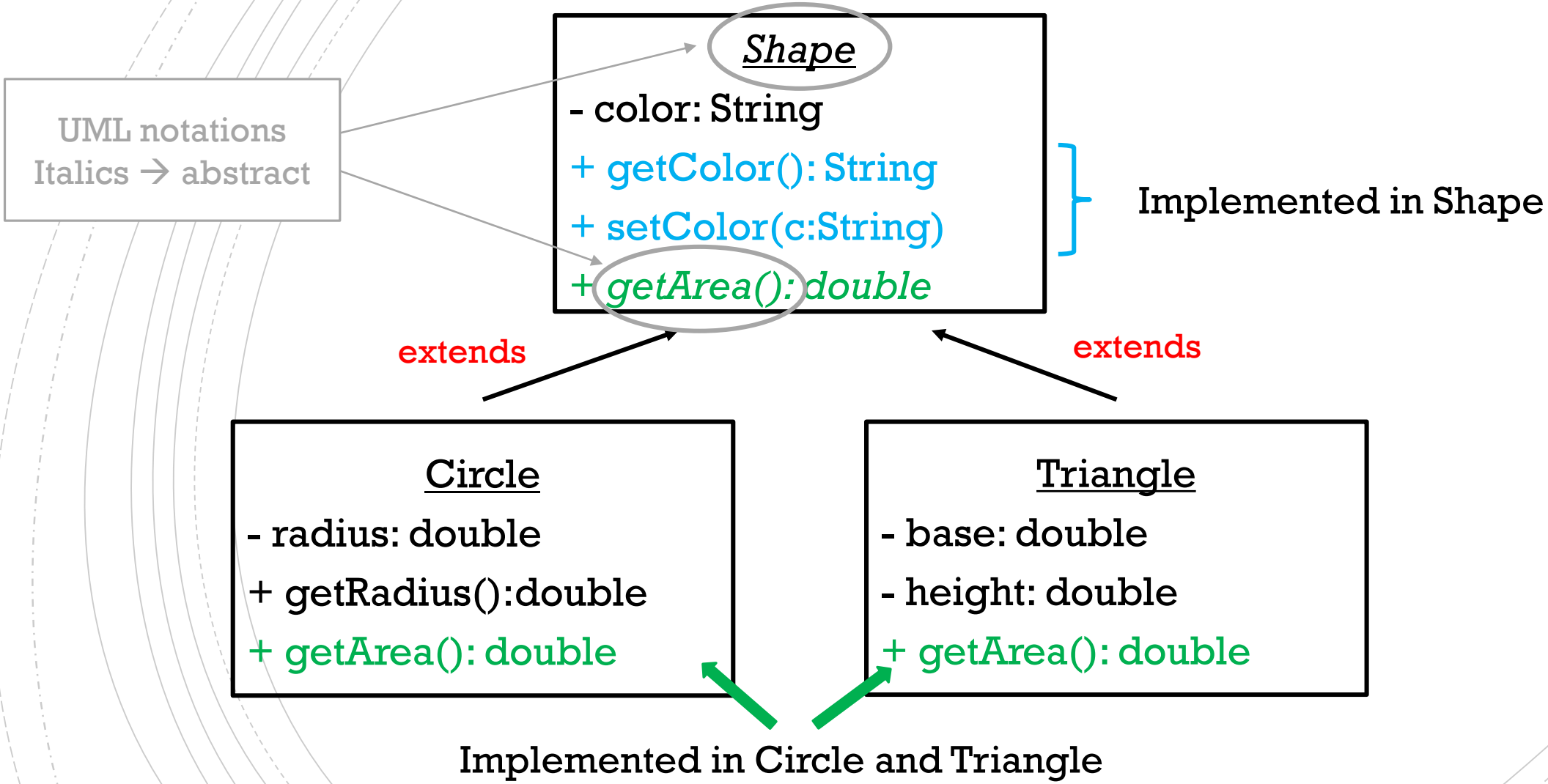
- An abstract class must be declared with an `abstract` keyword.
- It can have `abstract` and non-`abstract` methods.
- It cannot be instantiated.
- It can have constructors and static methods.
- It can have `final` methods which will force the subclass not to change the body of the method

`abstract` CLASSES – OBSERVATIONS

- We can have abstract classes with no abstract methods. This allow us to create classes that cannot be 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 the three classes.
- *IF TIME PERMITS*: In the Shape class add 2 static methods
  - getShallowCopy()
  - getDeepCopy()

  Both methods take an array of Shape as input and return a copy of it (one shallow, the other deep).
- Play around with the classes!

- From Wednesday Mike will talk to you about ArrayList and LinkedList.
- I'll be back briefly on Friday Oct. 5th with slow sorting algorithms.
- And then I'll see you again on Oct. 15th with even more about Java and OOD! ☺