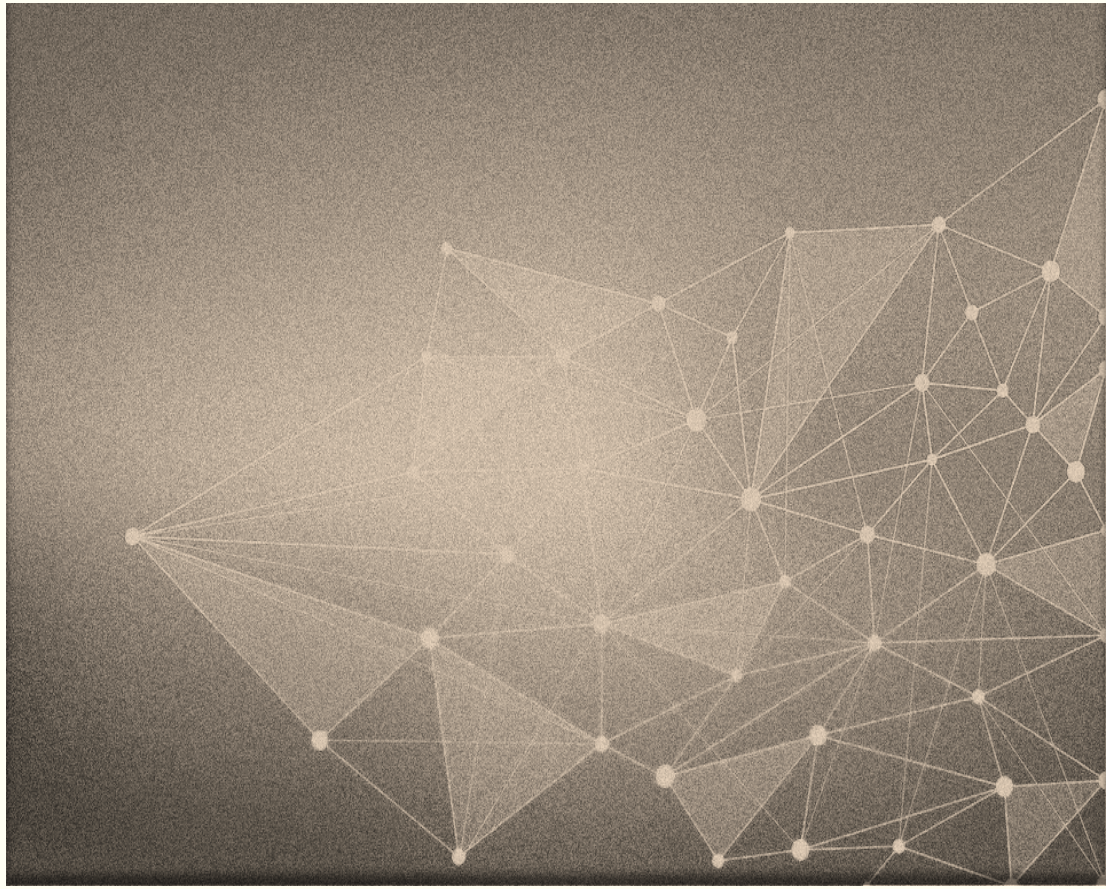
EECS2030:
ADVANCED OBJECT-ORIENTED PROGRAMMING

By: Dr. Marzieh Ahmadzadeh



Outline

- Last week:
  - Inheritance
  - Overridden methods
  - Object class
  - Composition vs Inheritance

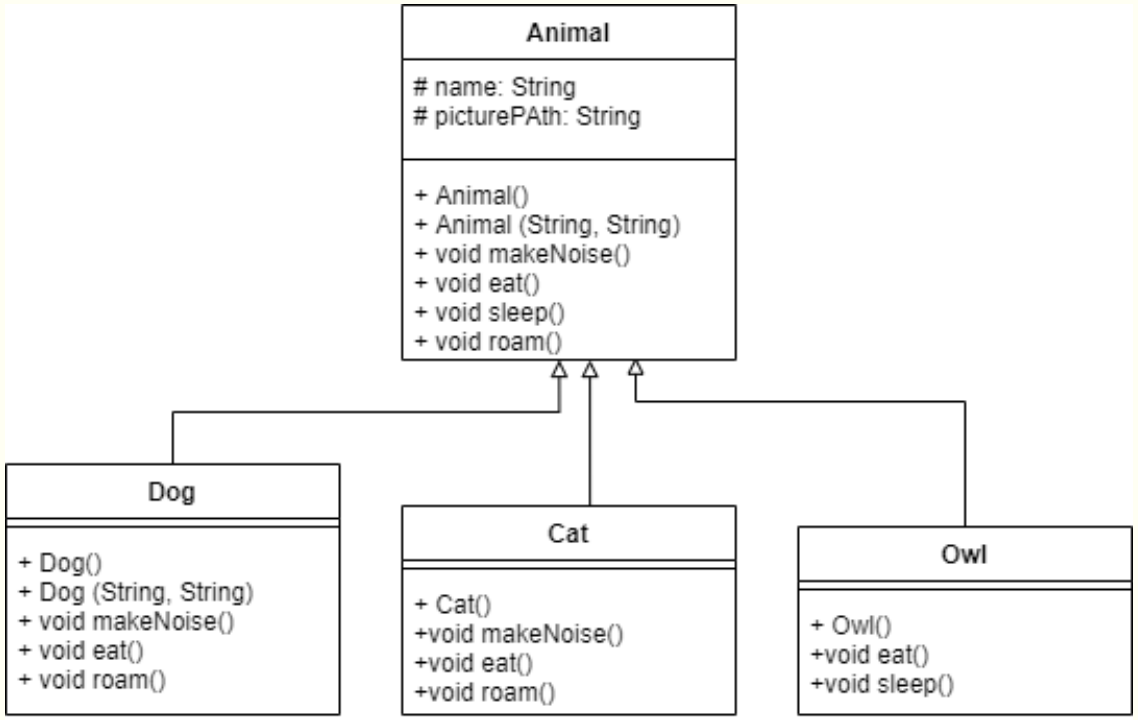
- This week
  - Polymorphism
    - Definition
    - How it works
  - Binding
    - Dynamic/ Late binding
    - Early binding
  - DBC wrt to Inheritance & Polymorphism
  - In-class activity

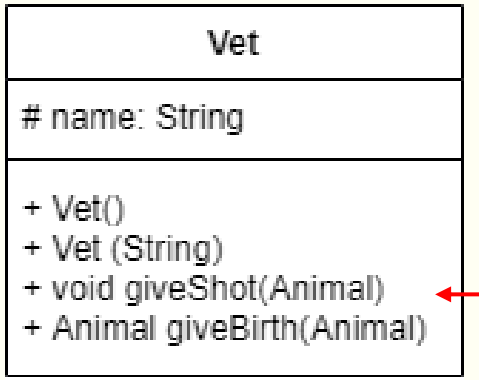
POLYMORPHISM

Another feature of object-oriented programming

Example:

- Subclasses in this hierarchy override `makenoise()` method.
- How should `giveShot()` be implemented so that each animal is able to make it's own noise?

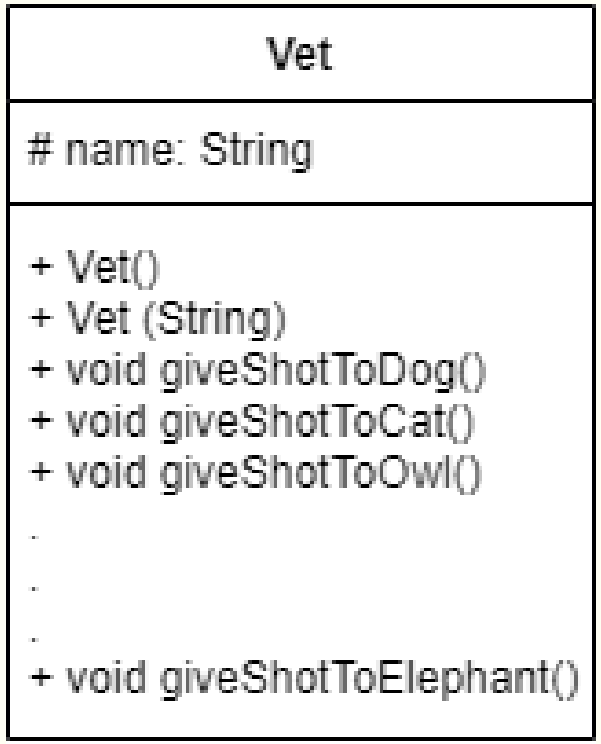




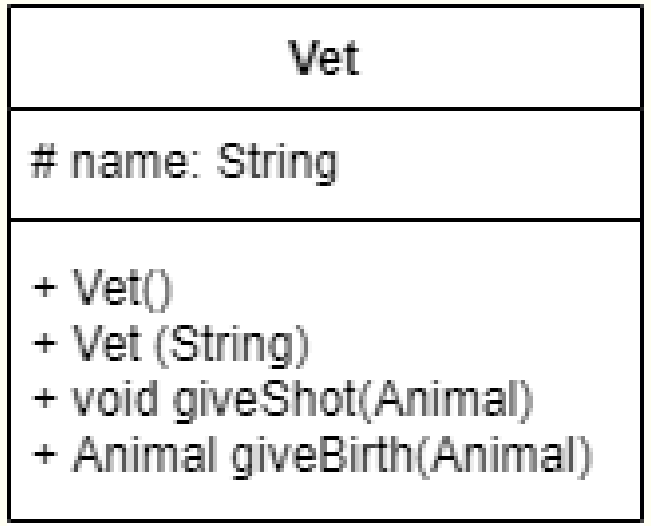

```
public void giveShot(?){
     // the vet gives a shot to an animal
    // the animal makes noise
}
```

Example

Without Polymorphism



With Polymorphism

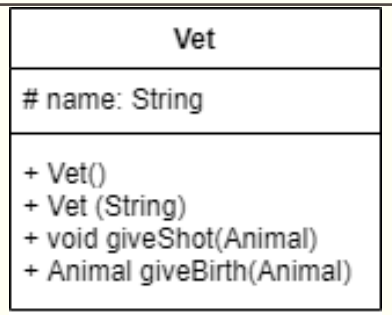


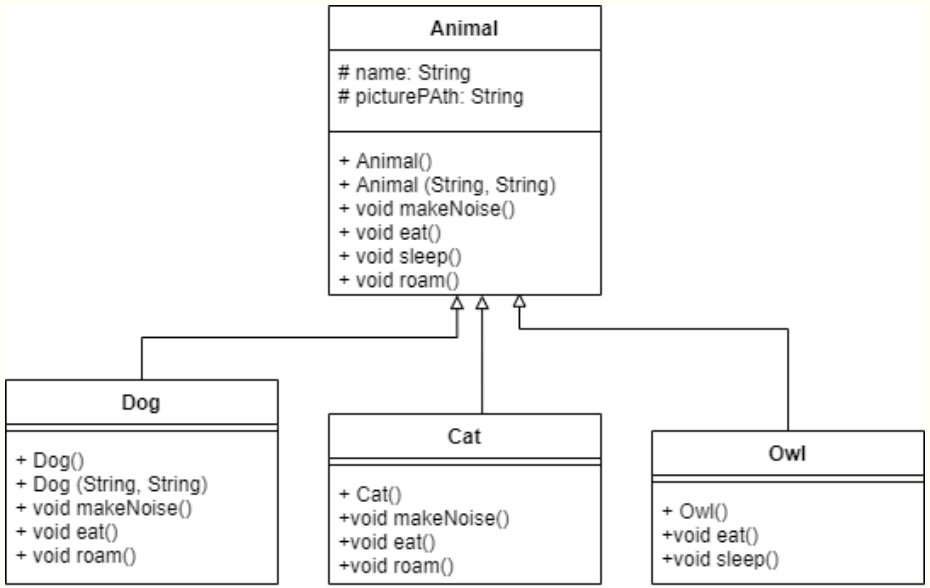

```
public void giveShot(Animal a){
     // the vet gives a shot to an animal
     a.makeNoise();
}
```

- In computer science `Polymorphism` refers to the substitution of an object of superclass by an object of its subclass.
  - The purpose of polymorphism is code-reuse.
- Polymorphism is only possible in the presence of inheritance.

Polymorphism: how it works?

```java
public class Vet {
    protected String name;
    public Vet() {
        name = " ";
    }
    public Vet(String name) {
        this.name = name;
    }

    public void giveShot(Animal animal) {
        animal.makeNoise();
    }
```



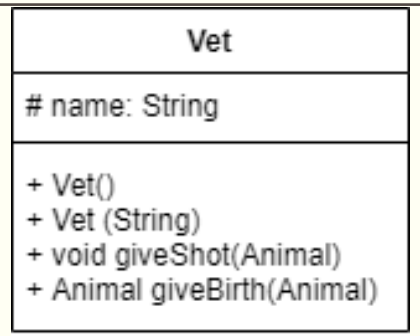


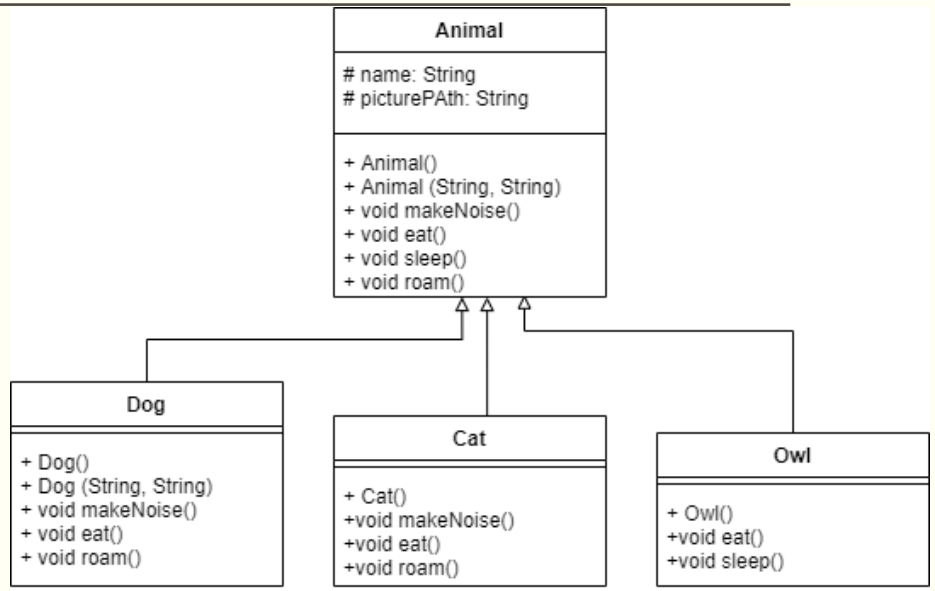
With polymorphism, whenever an animal object is expected, a dog, cat or an owl can be replaced.

```java
Animal anAnimal = new Animal();
Dog myDog = new Dog();
Cat herCat = new Cat();
Owl hisOwl = new Owl();

Vet john = new Vet("John");

john.giveShot(anAnimal);
john.giveShot(myDog);
john.giveShot(herCat);
john.giveShot(hisOwl);
```

Polymorphism: how it works?

```
public class Vet {
    protected String name;
    public Vet() {
        name = " ";
    }
    public Vet(String name) {
        this.name = name;
    }

    public void giveShot(Animal animal) {
        animal.makeNoise();
    }

    public Animal giveBirth(Animal animal) {
        animal.makeNoise();
        Animal baby;
        // not a good design
        if (animal instanceof Dog)
            baby = new Dog();
        else if (animal instanceof Cat)
            baby = new Cat();
        else baby = new Owl();
        return baby;
    }
}
```






```
Animal puppy1 = theVet.giveBirth(theDog);
Dog puppy2 = (Dog) theVet.giveBirth(theDog);
```

With polymorphism, whenever a superclass object is expected, a subclass can be replaced.

Definition (again)

- In computer science vocabulary:
  - A subclass object can substitute its superclass object, when the superclass is expected:
    - When an object is defined:
    - When an object is passed to a method
    - When an object is returned from a method

Because Dog IS-A(n) Animal

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

```
Vet theVet = new Vet("Jane");
theVet.giveShot(theDog);
```

```
Animal puppy = theVet.giveBirth(theDog);
```

Questions

- Knowing that the `animal` argument is a dog, can we write the following code?

```
Dog puppy = theVet.giveBirth(theDog);
```

```
    public Animal giveBirth(Animal animal) {
        animal.makeNoise();
        Animal baby = new Animal();
        return baby;
    }
}
```

- What if we know the animal that is returned from this method is of type Dog?
  - Cast it.

```
Animal puppy = theVet.giveBirth(theDog);
```

```
Dog puppy = (Dog) theVet.giveBirth(theDog);
```

- The followings is semantically wrong, but no compilation error is given. What do you think will happen when it is executed.

```
Cat puppy = (Cat) john.giveBirth(myDog);
```

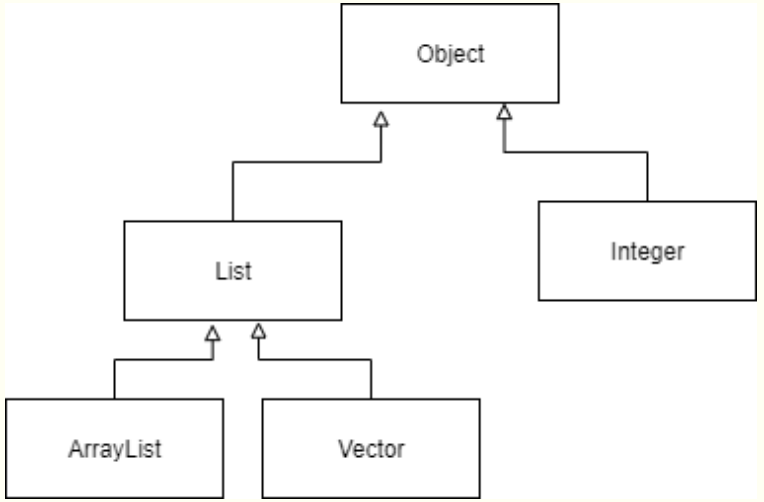
More Examples (1)

- Polymorphism in object definition:

```
List<Integer> vect = new Vector<Integer>();
Object array = new ArrayList<Integer>();
Object intValue = Integer.getInteger("10");
```

- Calling an overridden methods:

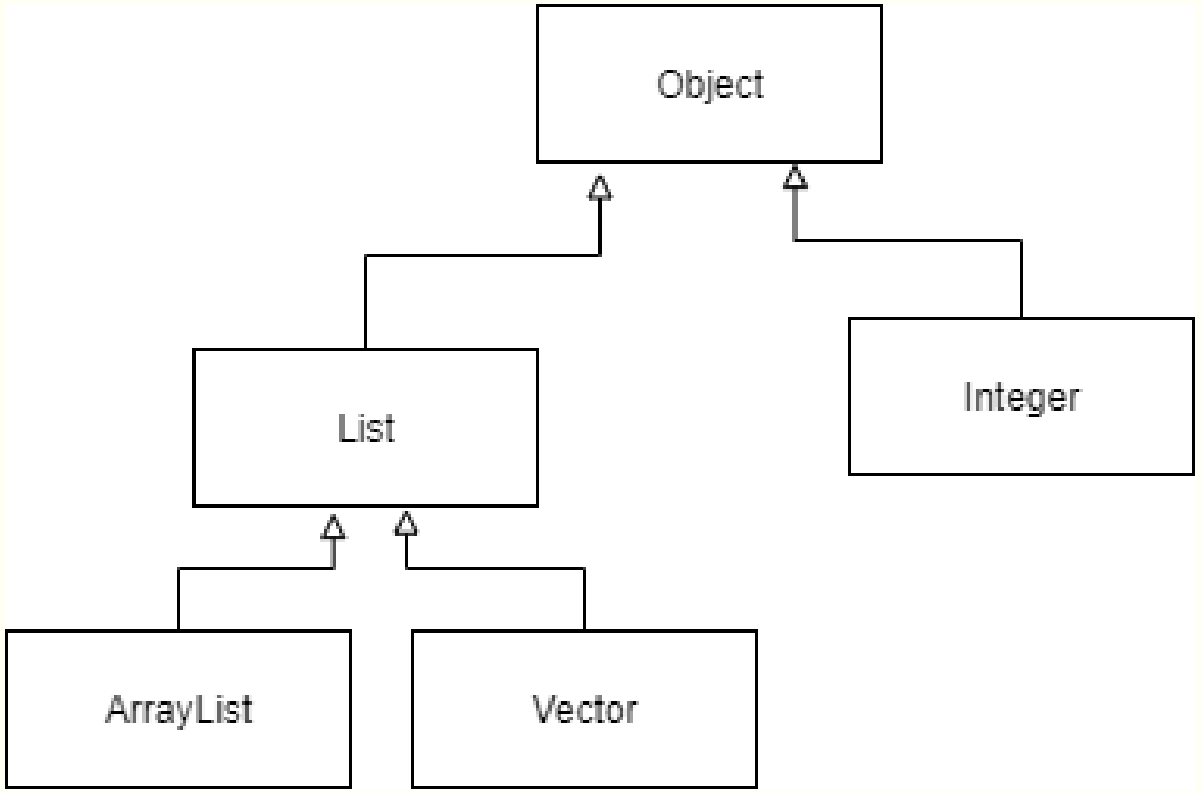
```
System.out.println(vect.toString());
System.out.println(array.toString());
System.out.println(intValue.toString());
```



Question

- Which statement is correct?

```
vect.add(0);
array.add(1);
```

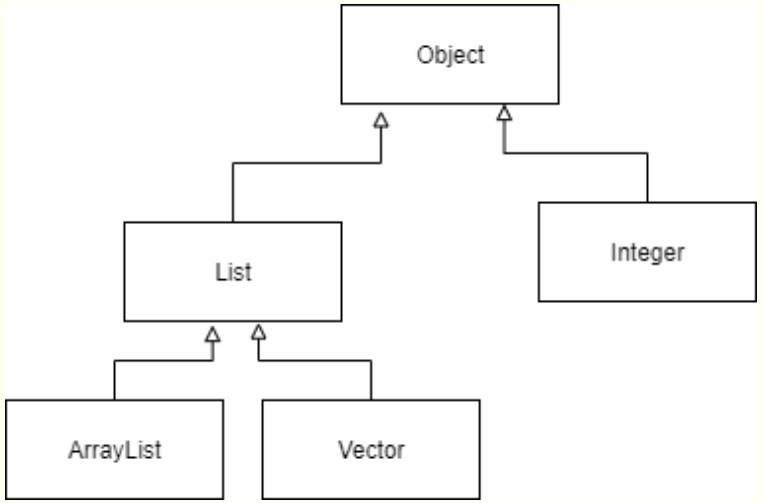



```
List<Integer> vect = new Vector<Integer>();
Object array = new ArrayList<Integer>();
```

More Examples (2)

- Polymorphism in passing a subtype to a method where the supertype is expected:

```
vect.add((Integer) intValue);
System.out.print(vect.contains(intValue));
```



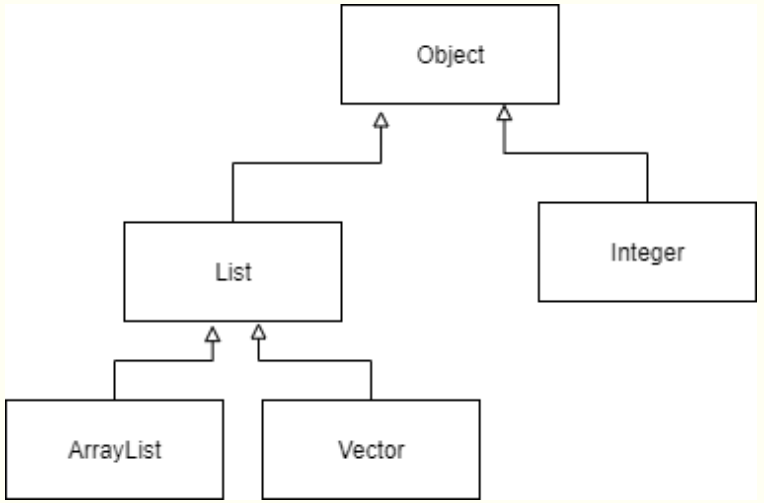

```
List<Integer> vect = new Vector<Integer>();
Object array = new ArrayList<Integer>();
Object intValue = Integer.getInteger("10");
```

boolean	add(E e)	Appends the specified element to the end of this Vector.
boolean	contains(Object o)	Returns true if this vector contains the specified element.

More Examples (3)

- Polymorphism in returning a subtype where the supertype is expected.

```
List<Integer> vect = new Vector<Integer>();
for (int i = 0; i < 10; i++)
    vect.add(i);

List<Integer> subList = vect.subList(0, 5);
for (int i = 0; i < subList.size(); i++)
    System.out.println(subList.get(i));
```



`List<E>`	`subList(int fromIndex, int toIndex)`	Returns a view of the portion of this List between fromIndex, inclusive, and toIndex, exclusive.

In-Class Activity

- Question 1

Polymorphism: Summary

- Another principle of object-oriented programming
- With polymorphism, a subtype can substitute its supertype.
- Polymorphism is only possible when an inheritance relationship exists.
- With polymorphism the `behaviour` of a method is changed depending on the object that it works on
  - See giveShot() example
  - This is possible because of **Late Binding** (aka **Dynamic Binding**) mechanism defined in Java.

BINDING

Binding

- Recall: With polymorphism the `behaviour` of a method is changed depending on the object that it works on
  - See giveShot() example
  - This is possible because of **Late Binding** (aka **Dynamic Binding**) mechanism defined in Java.

- Binding is the decision of selecting a `proper method definition` for execution, when the method is called.
  - This translates to potentially having more than one method definition that Java needs to choose from.

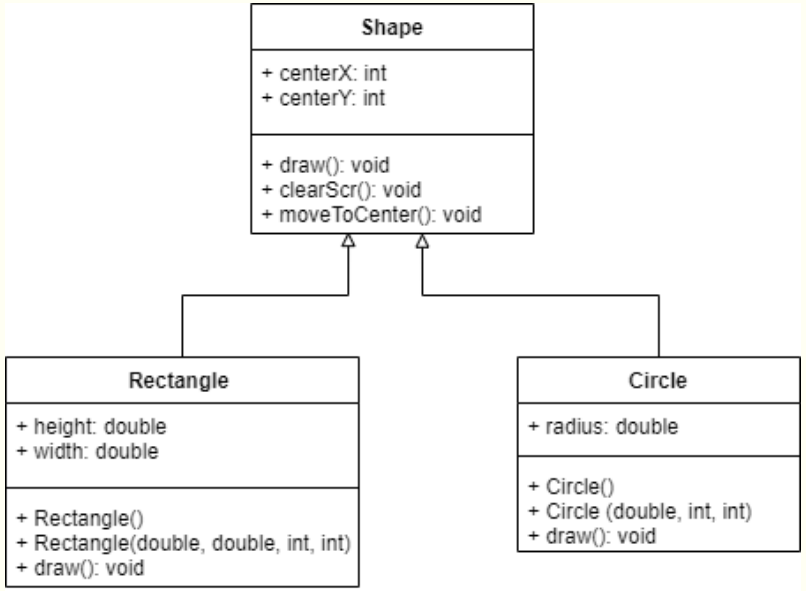
- Simpler definition: Which method definition in the hierarchy of inheritance is used for execution.
  - This depends on the `object (i.e., its type)` that requests the method.

- Depending on the time that this decision is made, two types of binding is defined.
  - Early Binding (aka static binding): association happens at compile time.
  - Late Binding (aka Dynamic Binding): association happens at run-time.

Binding: Example (1)

- Question: Which draw() method is called:

```
Shape firstShape = new Circle(4, 100, 100);
firstShape.draw();

Shape secondShape = new Rectangle(4, 2, 100, 100);
secondShape.draw();
```

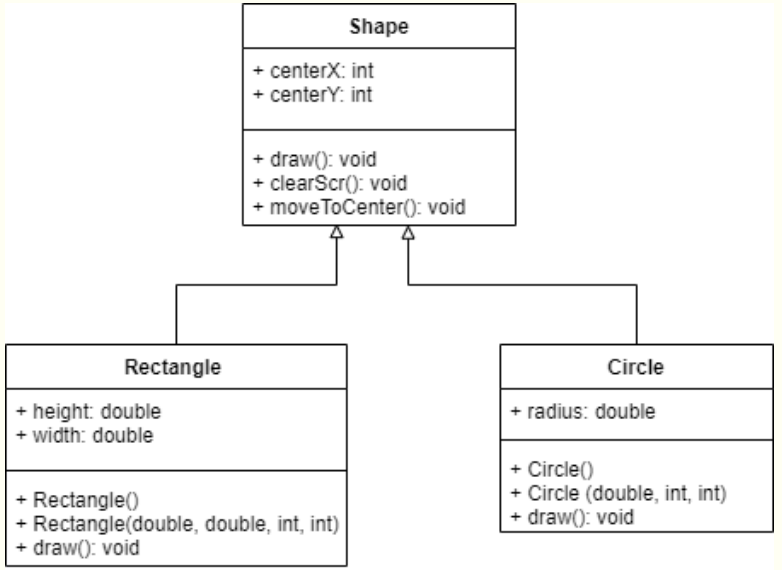


Binding: Example (2)

- Question: Which draw() method is called:

```
Shape firstShape = new Circle(4, 100, 100);

Shape secondShape = new Rectangle(4, 2, 100, 100);
```

```
firstShape.moveToCenter();
secondShape.moveToCenter();
```



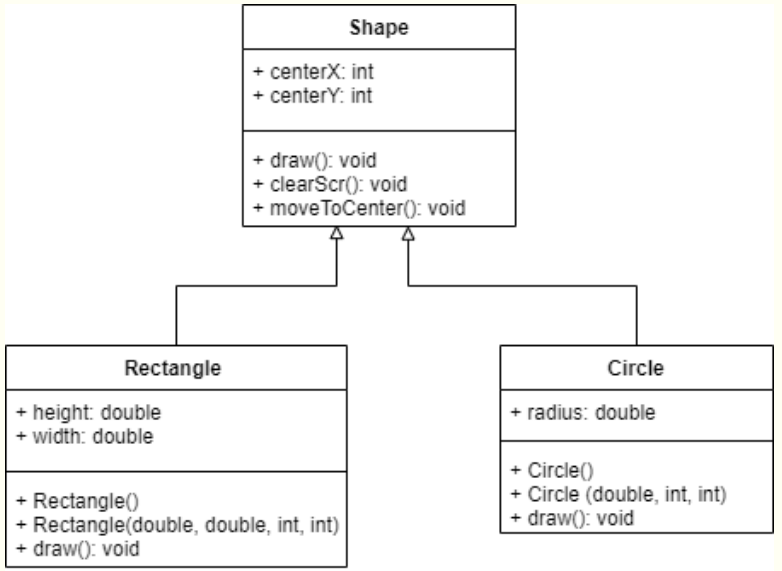

```
public void moveToCenter() {
    clearScr();
    this.draw();
}
```

```
public void clearScr() {
    System.out.println("Clear screen is in process....");
    centerX = 0;
    centerY = 0;

}
```

Dynamic Binding in this example:

- At Compilation: The following code is correct as there is a draw method in class Shape.

```
public void moveToCenter() {
    clearScr();
    this.draw();
}
```



- At Runtime: it is decided, which draw() method should be executed for the following codes.
  - i.e. late binding: the binding is postponed to run-time

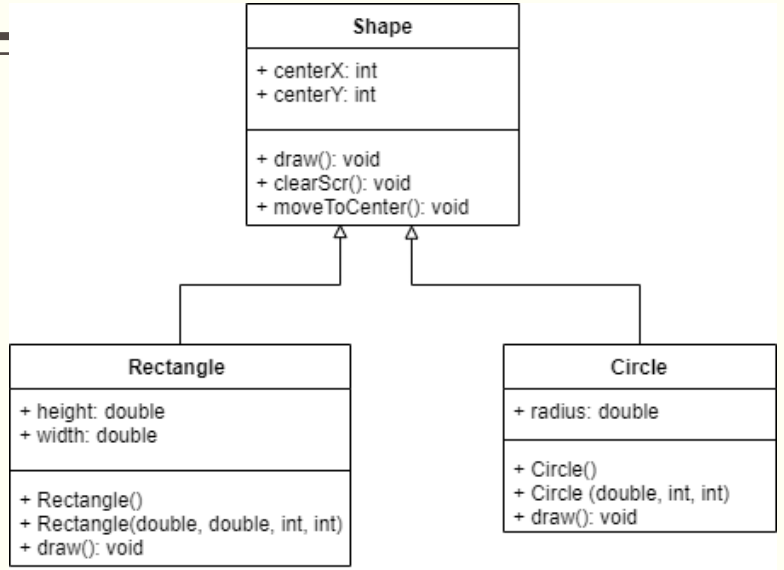
```
firstShape.moveToCenter();
secondShape.moveToCenter();
```

Now the name POLYMORPHISM (Multiple Forms) should make sense to you.

Dynamic Binding: Back to the early examples:

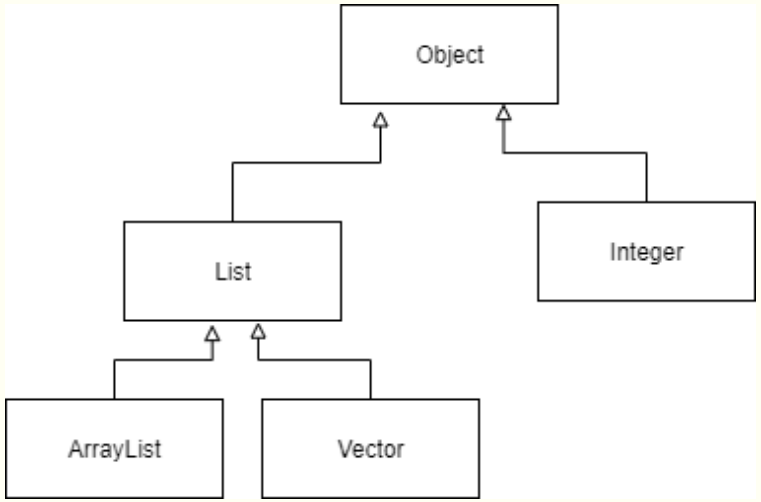
- At Compilation: The following code is correct:

```
Shape firstShape = new Circle(4, 100, 100);
Shape secondShape = new Rectangle(4, 2, 100, 100);

firstShape.moveToCenter();
secondShape.moveToCenter();
```



- At Compilation: The following code is NOT correct:

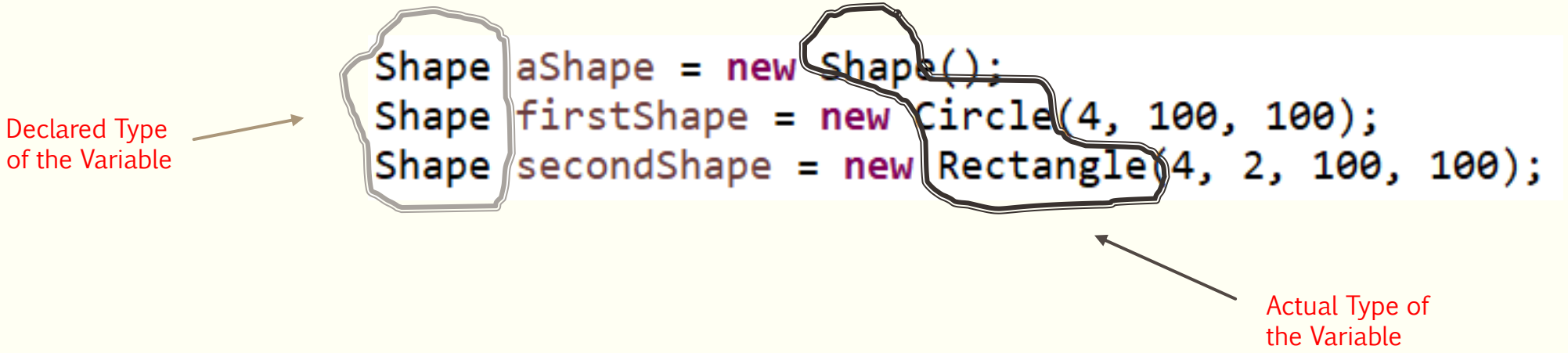
```
Object array = new ArrayList<Integer>();
 array.add(1);
```



- At Compilation: The following code is correct:

```
List<Integer> vect = new Vector<Integer>();
vect.add(0);
```

Dynamic Dispatching (or why dynamic bindings works the way it works)

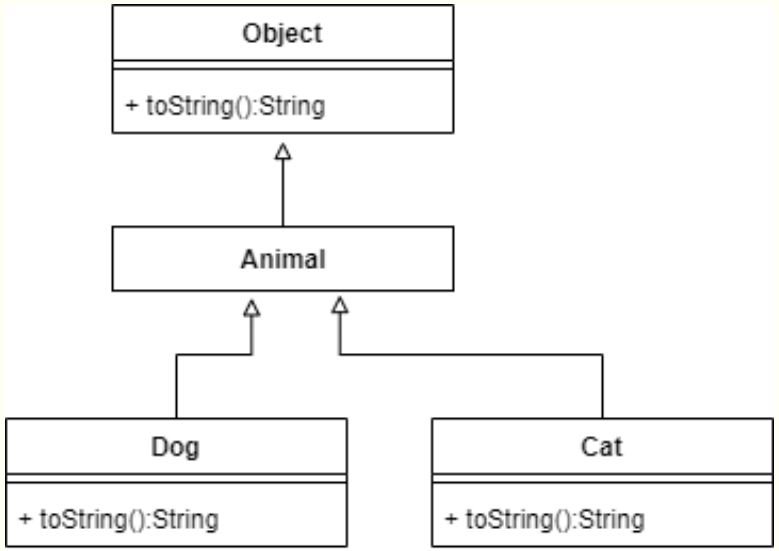


- The `declared type` of an object reference determines where the method is invoked.
  - Starting point
  - Checked at compile time
- The `actual type` of an object reference determines which polymorphic method is run.
  - Dispatching point
  - Done at run time
- Dynamic Dispatching: is the mechanism by which dynamic binding is explained/performed.

Dynamic Dispatching: toString()

- Question1: Where does toString() is invoked for the following code?
- Question2: To which class the execution of toString is dispatched to?

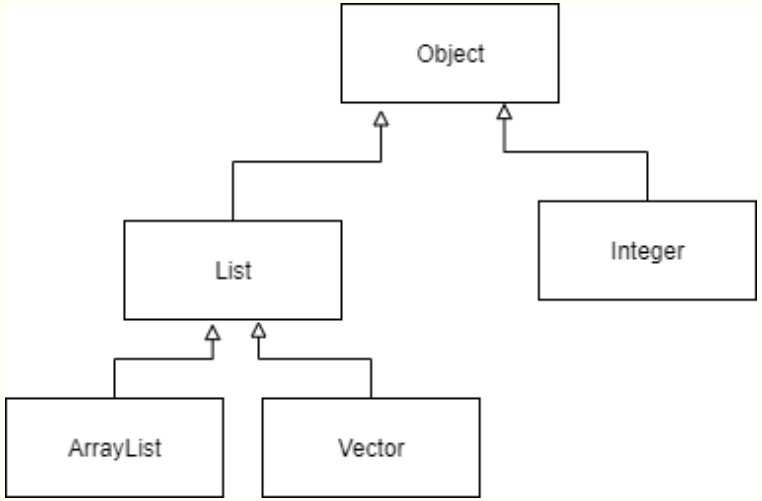
```
Object obj1 = new Dog();
System.out.println(obj1.toString());
Animal obj2 = new Dog();
System.out.println(obj2.toString());
Dog obj3 = new Dog();
System.out.println(obj3.toString());
Object obj4 = new Cat();
System.out.println(obj4.toString());
Animal obj5 = new Cat();
System.out.println(obj5.toString());
Cat obj6 = new Cat();
System.out.println(obj6.toString());
```



Dynamic Dispatching: going back to the previous example

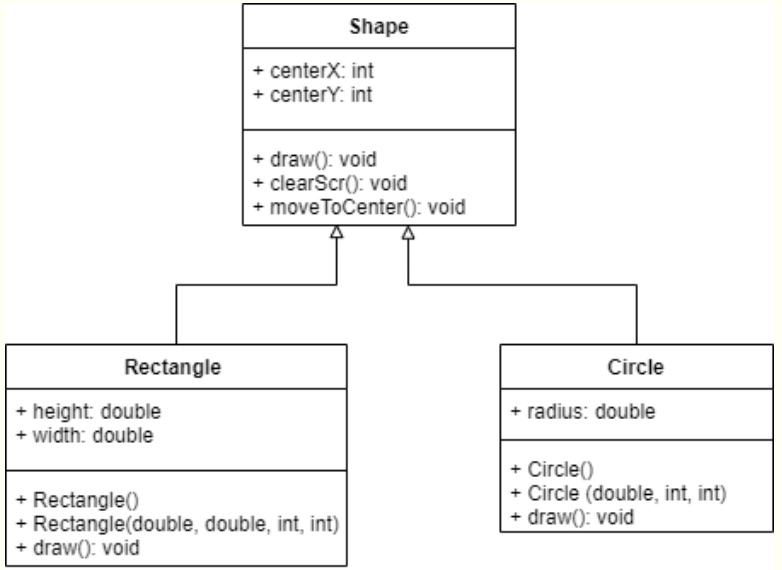
- The following code generates a compiler error because:
  - The actual type determines which polymorphic method should be dispatched. (i.e. at run time – no problem)
  - However, the declared type determines, where the method is invoked. (i.e. compile time - error)
    - Object class does not contain add() method.
    - Compiler error is issued.

```
Object array = new ArrayList<Integer>();
array.add(1);
```



Early Binding

- At compile time the binding between a method call and method definition is determined.
- If a method is `final or private`, it cannot be overridden by the derived classes.
  - Example: if `clearScr()` in shape had been final ➔ the association between the object and this method would have been at compile time.




```
Shape aShape = new Shape();
Shape firstShape = new Circle(4, 100, 100);
Shape secondShape = new Rectangle(4, 2, 100, 100);
```

```
aShape.clearScr();
firstShape.clearScr();
secondShape.clearScr();
```

In-class Activity

- Question 2

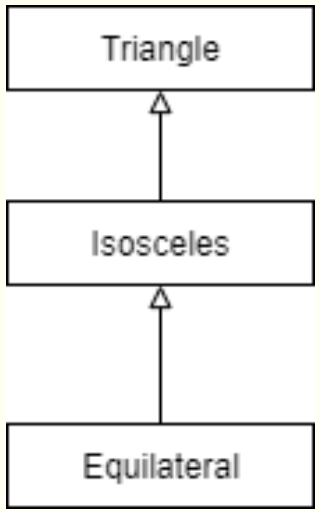
DBC & INHERITANCE & POLYMORPHISM

DBC: Recall

- A precondition for a method is a logical expression (aka assertion) which must be true just before the method is called
- A postcondition for a method is a logical expression which must be true just after the method has been completed.
- An invariant is an assertion that must be true just before the method is called and after the method has been completed.

DBC w.r.t. to Inheritance & Polymorphism

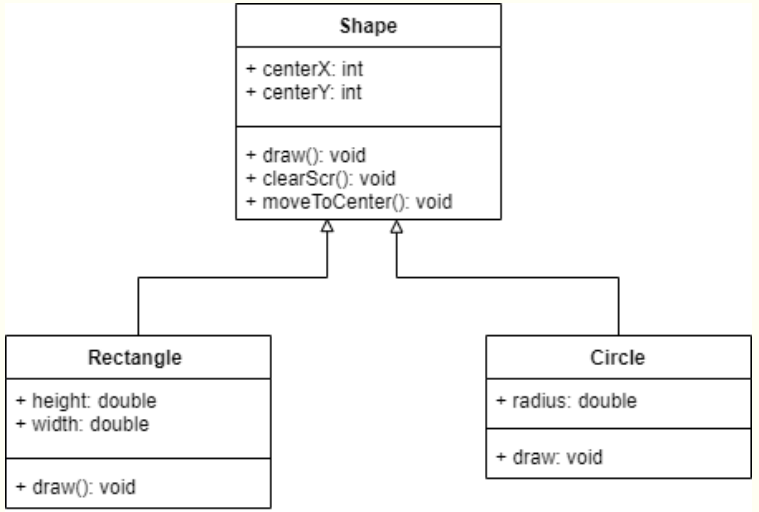
- Recall :subclass is-a superclass (inheritance) means subclass is-substitutable for a superclass (Polymorphism).
  - So whatever the superclass can do, the subclass can do too (even more)
- DBC rules for inheritance:
  - The class invariant must not be weaker than the invariant in the superclass
  - Think about the subclass as a more specific type of the superclass.
  - Invariant for Triangle:
    - Sum of angles = 180 degrees
    - Three sides
  - Invariant for Isosceles:
    - Sum of angles = 180 degrees
    - Three sides
    - Two sides have equal length
  - Invariant for Equilateral:
    - Sum of angles = 180 degrees
    - Three sides
    - Three sides have equal length



DBC w.r.t. to Inheritance & Polymorphism

- Recall : subclass is-a superclass (inheritance) means subclass is-substitutable for a superclass (Polymorphism).
  - So whatever the superclass can do, the subclass can do too (even more)
- DBC rules for inheritance:
  - The precondition for a subclass must not be stronger than the precondition in its superclass
    - Otherwise it contradicts with “whatever the superclass can do, the subclass can do too (even more)”

```
Shape firstShape = new Circle(4, 100, 100);
firstShape.draw();

Shape secondShape = new Rectangle(4, 2, 100, 100);
secondShape.draw();
```



DBC w.r.t. to Inheritance & Polymorphism

- Recall : subclass is-a superclass (inheritance) means subclass is-substitutable for a superclass (Polymorphism).
  - So whatever the superclass can do, the subclass can do too (even more)
- DBC rules for inheritance:
  - The postcondition for a subclass must not be weaker than the postcondition in the superclass ·
    - whatever the superclass can do, the subclass can do too (even more). So a subclass must not provide less than what the superclass is promised to provide .

Expectations & Reading

- Expectations
  - You should have a good understanding of what polymorphism is and how it is used.
  - You should be able to design a program that manipulates polymorphism
  - You should have a good understanding of what binding is.
  - You should be able to explain the difference between two types of binding.
  - You should be able to explain which method is called due to binding property.
  - You should be able to identify a correct DBC for a subclass, if the DBC of the superclass is given.
- Reading:
  - None required.
- One-Minute Paper