CISC 3115 TY3
C18a: Abstract Class and Method

Hui Chen

Department of Computer & Information Science

CUNY Brooklyn College

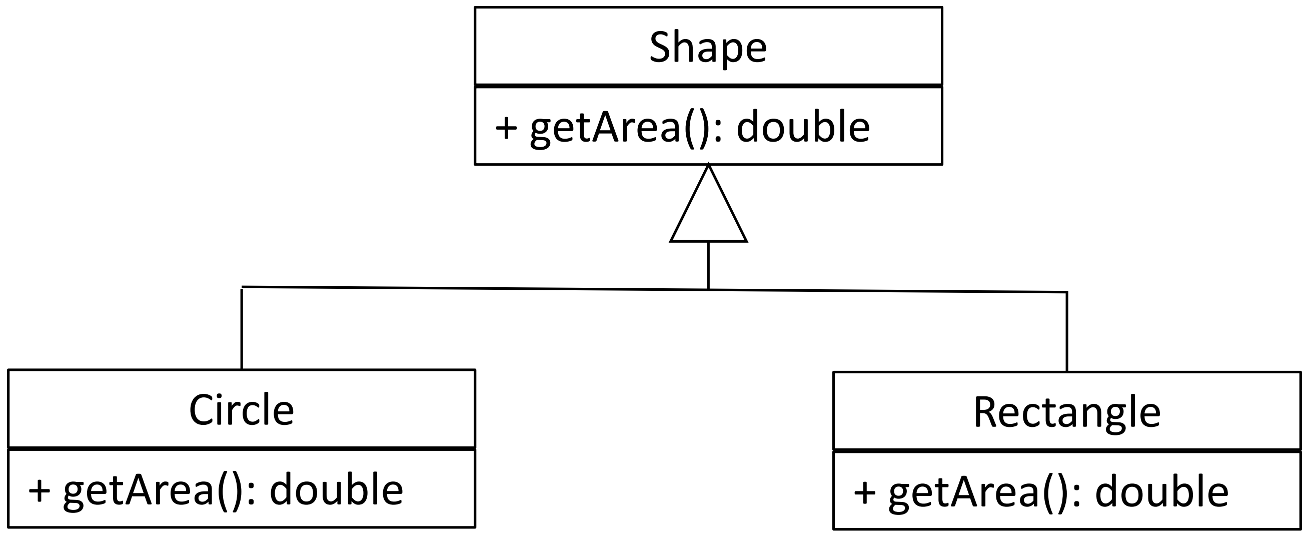
Outline

- Recap
  - Inheritance and polymorphism
- Abstract method and class
- Exercises
  - C18a-1
  - C18a-2
  - C18a-3

Recap: Inheritance and Polymorphism

- Problem: leveraging on polymorphism, write generic method to compute total areas of a list of geometric shapes.

The Shape Class Hierarchy



Recap: Inheritance and Polymorphism

- Problem: leveraging on polymorphism, write generic method to compute total areas of a list of geometric shapes.
- Solution

```
public double sumAreasOfShapes(ArrayList<Shape> shapeList) {
  double sum = 0.;
  for(Shape shape: shapeList) {
      sum += shape.getArea();
  }
  return sum;
}
```

The Dilemma

- Problem: leveraging on polymorphism, write generic method to compute total areas of a list of geometric shapes.
- Dilemma: but can we compute a shape's area without knowing the specification of the shape (e.g., type of shape, parameters of the shape)?

The Shape Class

- Do you like the getArea() method here?

```
public class Shape {          …
  public double getArea() {
     throw new UnsupportedOperationException("Cannot invoke the method");
  }
}
```

- Remarks
    - We know semantically that each shape has a behavior to compute its area
    - However, we don't know the algorithm without knowing the actual shape
    - The "dummy" method in the above is not only semantically undesired, but also can easily cause runtime errors.

Abstract Method

- An abstract method has no implementation

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

Abstract Method: No Implementation

- Let's declare an abstract method. How about these code snippets?

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

```
double getArea() ;
```

```
abstract double getArea() {
}
```

```
abstract double getArea() {}
```

```
double getArea() {}
```

Abstract Method: No Implementation

- Let's declare an abstract method. How about these code snippets?

```
abstract double getArea() ;
```
✓

```
double getArea() ;
```
✗

```
abstract double getArea() {
}
```
✗

```
abstract double getArea() {}
```
✗

```
double getArea() {}
```
✗

Abstract Class

- In Java, any class that has an abstract method must be declared “abstract”
- Example

```
abstract class Shape {
   public abstract double area() ;
}
```

- Abstract class: a class that is declared abstract
- Abstract classes cannot be instantiated, but they can be subclassed.

Class with Abstract Method

- Abstract method: a method that is declared without an implementation

```
abstract void makeNoise();
```

- A class that has an abstract method must be declared abstract
  - How about these two code snippets?

```
class Animal {
  abstract void makeNoise();
}
```

```
abstract class Animal {
  abstract void makeNoise();
}
```

Class with Abstract Method

- A class that has an abstract method must be declared abstract

```
class Animal {
   abstract void makeNoise();
}
```

✗

```
abstract class Animal {
   abstract void makeNoise();
}
```

✓

Abstract Class: Subclass & Instantiation

- Abstract classes ***cannot*** be instantiated, but they ***can be*** subclassed.

```
abstract class Shape {
 public abstract double area() ;
}
```

- How about these code snippets?

```
Shape s = new Shape();
```

```
class Circle extends Shape {…}
Shape s = new Circle();
```

Abstract Class: Subclass & Instantiation

- Abstract classes ***cannot*** be instantiated, but they ***can be*** subclassed.

```
abstract class Shape {
 public abstract double area() ;
}
```

- How about these code snippets?

```
Shape s = new Shape();
```

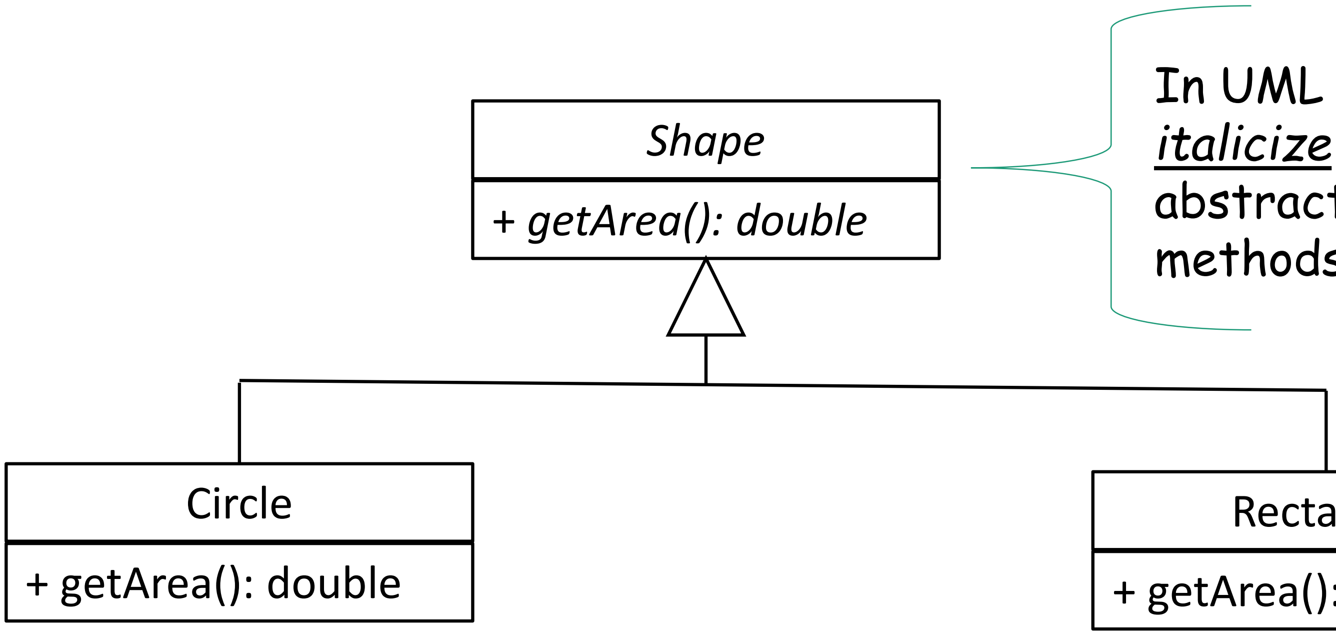
✗

```
class Circle extends Shape {…}

Shape s = new Circle();
```

✓

The Shape Class Hierarchy: Abstract Shape



In UML class diagram, *italicize* names of abstract classes and methods

Subclass an Abstract Class

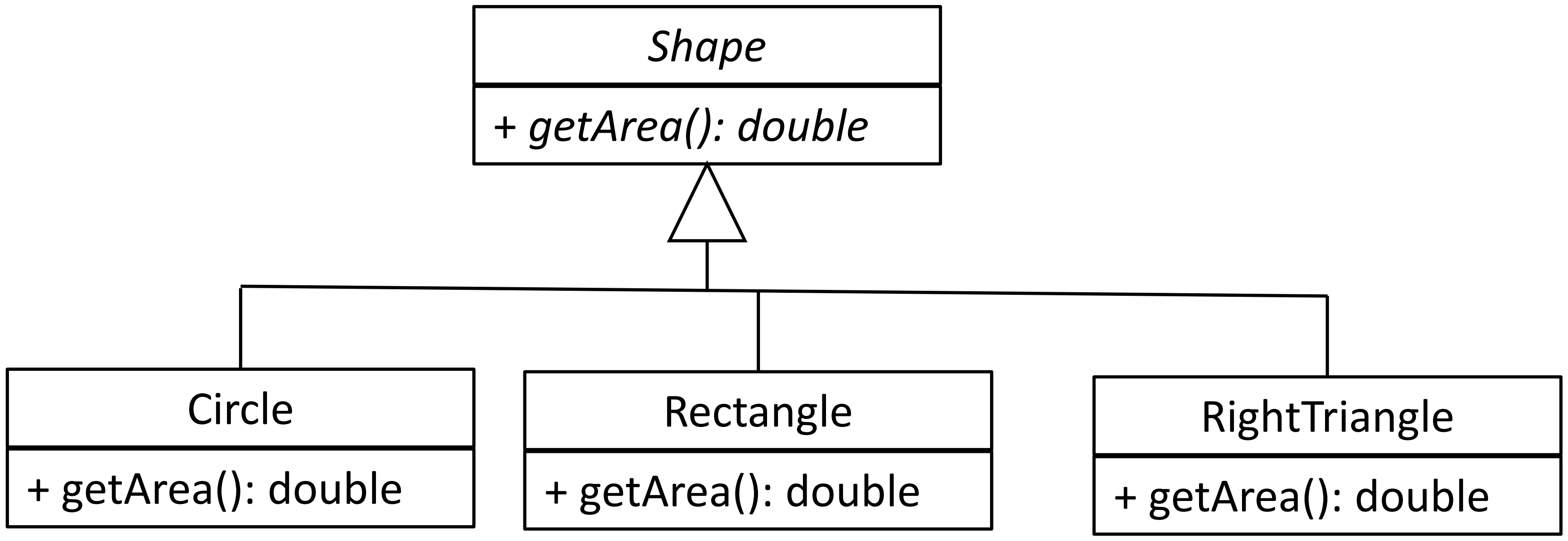
- Concrete subclass
  - A subclass may provide implementations for all of the abstract methods in its parent class.
- Abstract subclass
  - The subclass must also be declared abstract if it does not provide implementation of all of the abstract methods in its parent class.
  - The subclass may also declare abstract method itself

Concrete Subclass

- A subclass provides implementations of <u>all</u> of the abstract methods declared in its superclass.
- An abstract method thus can have many implementations.

The Shape Class Hierarchy: Concrete Subclasses

- *Abstract class Shape's getArea method* has many implementations in the abstract class's subclasses



Concrete Subclass

- Example:

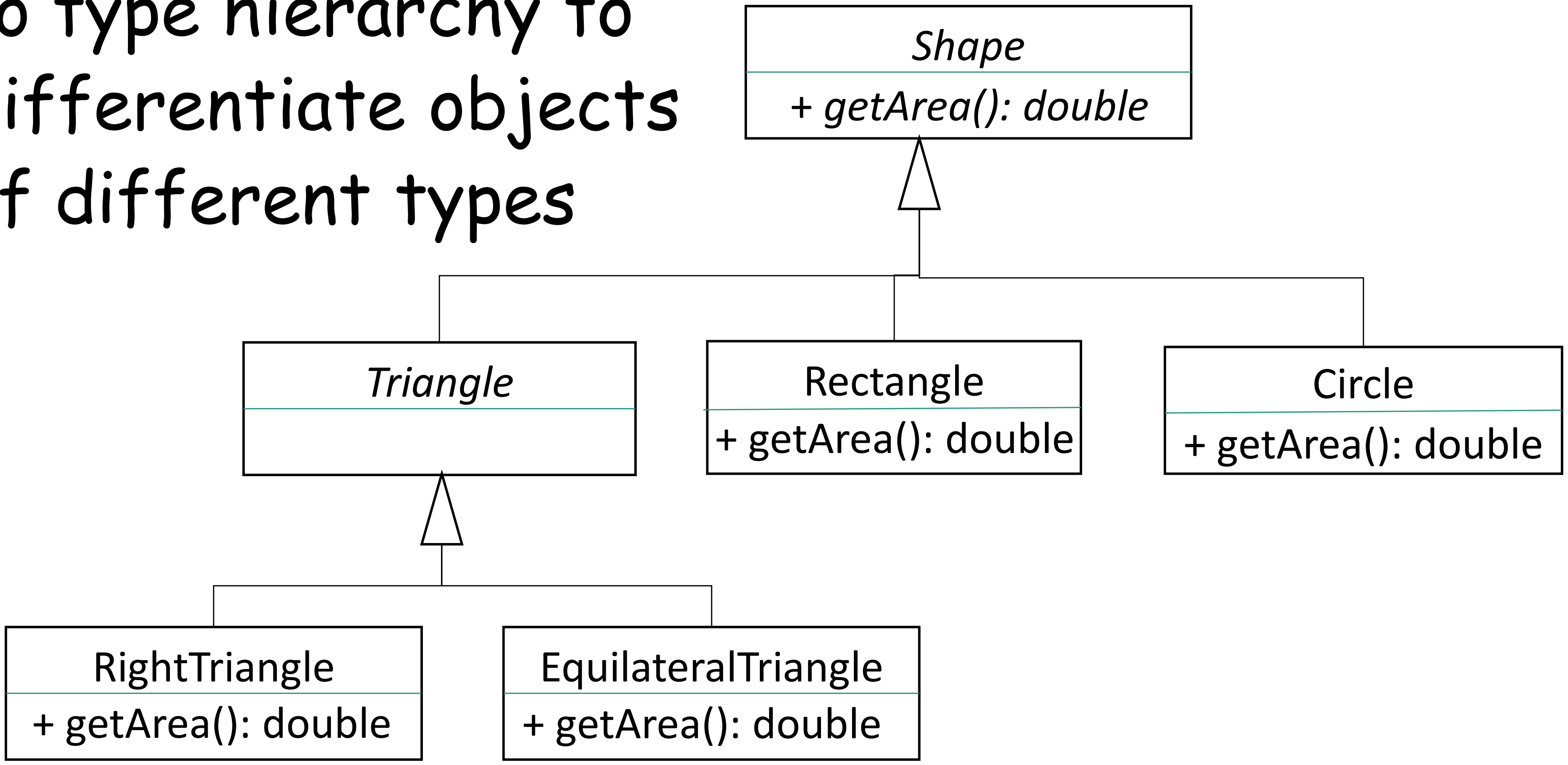
```
public class RightTriangle extends Shape {
  private double base;
  private double height;
  ……
  public double getArea() {
    return 0.5 * base * height;
  }
}
```

Abstract Subclass

- The subclass must also be declared abstract if it does not provide implementation of all of the abstract methods in its superclass.
- The subclass may also declare abstract method itself.

Abstract Subclass: Extending Type Hierarchy

- Motivation: add types to type hierarchy to differentiate objects of different types



Abstract Subclass: Extending Type Hierarchy

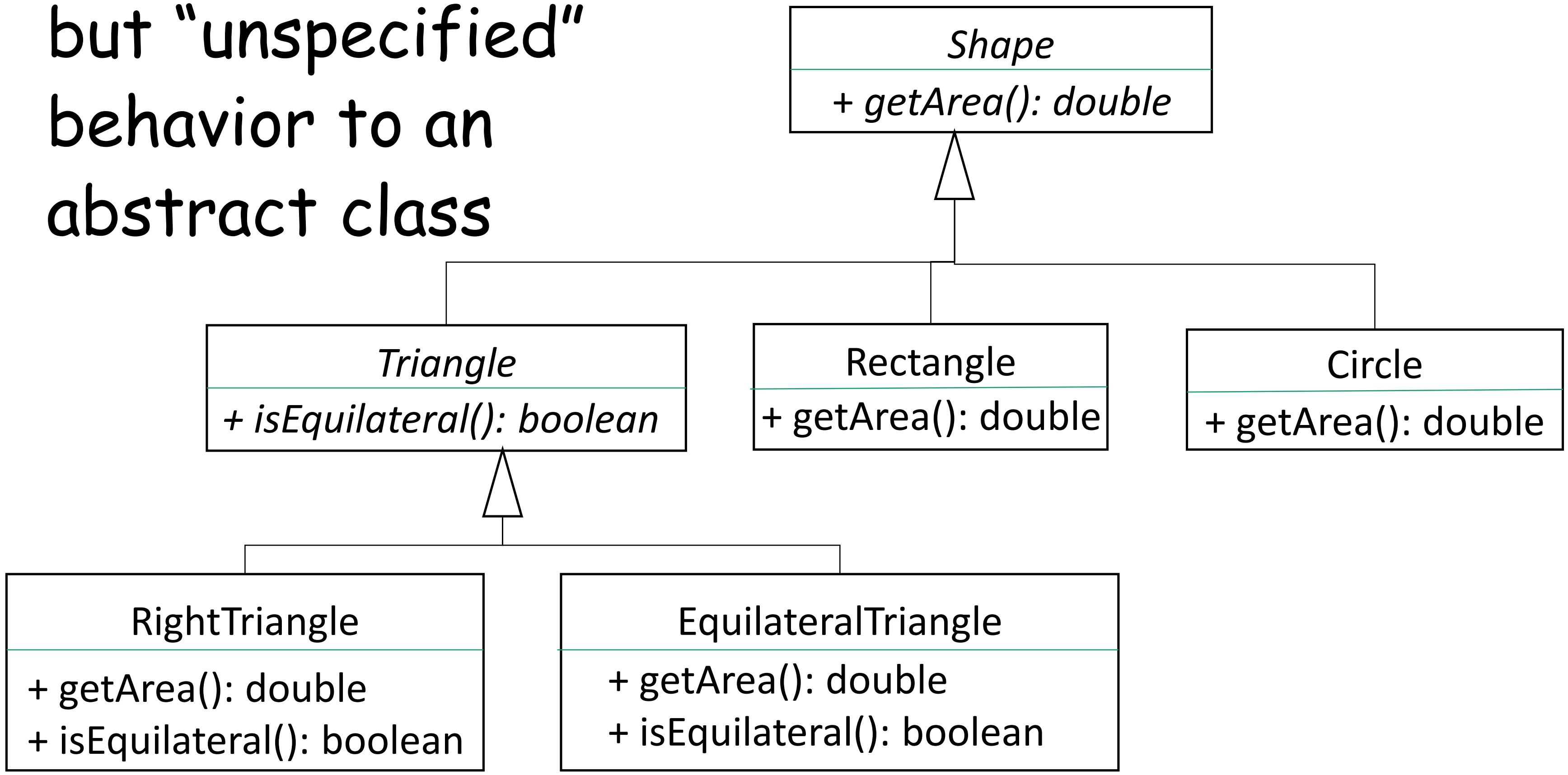
- Example:

```
public abstract class Triangle extends Shape {
        public Triangle(String name) {
                super(name);
        }
        public int getNumberOfSides() {
                return 3;
        }
}
```

- Remark:
    - An abstract class must be declared abstract.
    - An abstract class can have concrete methods.

Abstract Subclass: Add New Abstract Behavior

- Motivation: add new, but “unspecified” behavior to an abstract class



Abstract Subclass: Add New Behavior

- Example:

```
public abstract class Triangle extends Shape {
        public Triangle(String name) {
                super(name);
        }
        public int getNumberOfSides() {
                return 3;
        }
        public abstract boolean isEquilateral();
}
```

Questions?

- Abstract class
- Abstract method
- Extending abstract class
    - Concrete subclass
    - Abstract subclass
- Example programs

Exercise C18a-1

- In this exercise, you are to compare the Shape class hierarchies with and without making Shape class abstract.
  - Create directory C18a-1 in your weekly practice repo, and copy the two directories "ConcreteShape" and "AbstractShape" to your C18a-1 directory
  - Add a RightTriangle class with two data fields, base and height to the programs in both of the "ConcreteShape" and "AbstratShape" directories, the class is a subclass of the Shape class. (As seen in UML class diagram in this Slide)
  - Revise the 4 client classes (e.g., ShapeClient, ShapeClientError, etc) to use your RightTriangle class.
  - Compile and run the programs, and observe compilation error and runtime error if any.
  - Write a comment in the client programs to explain what you observe.
  - Submit your work using git

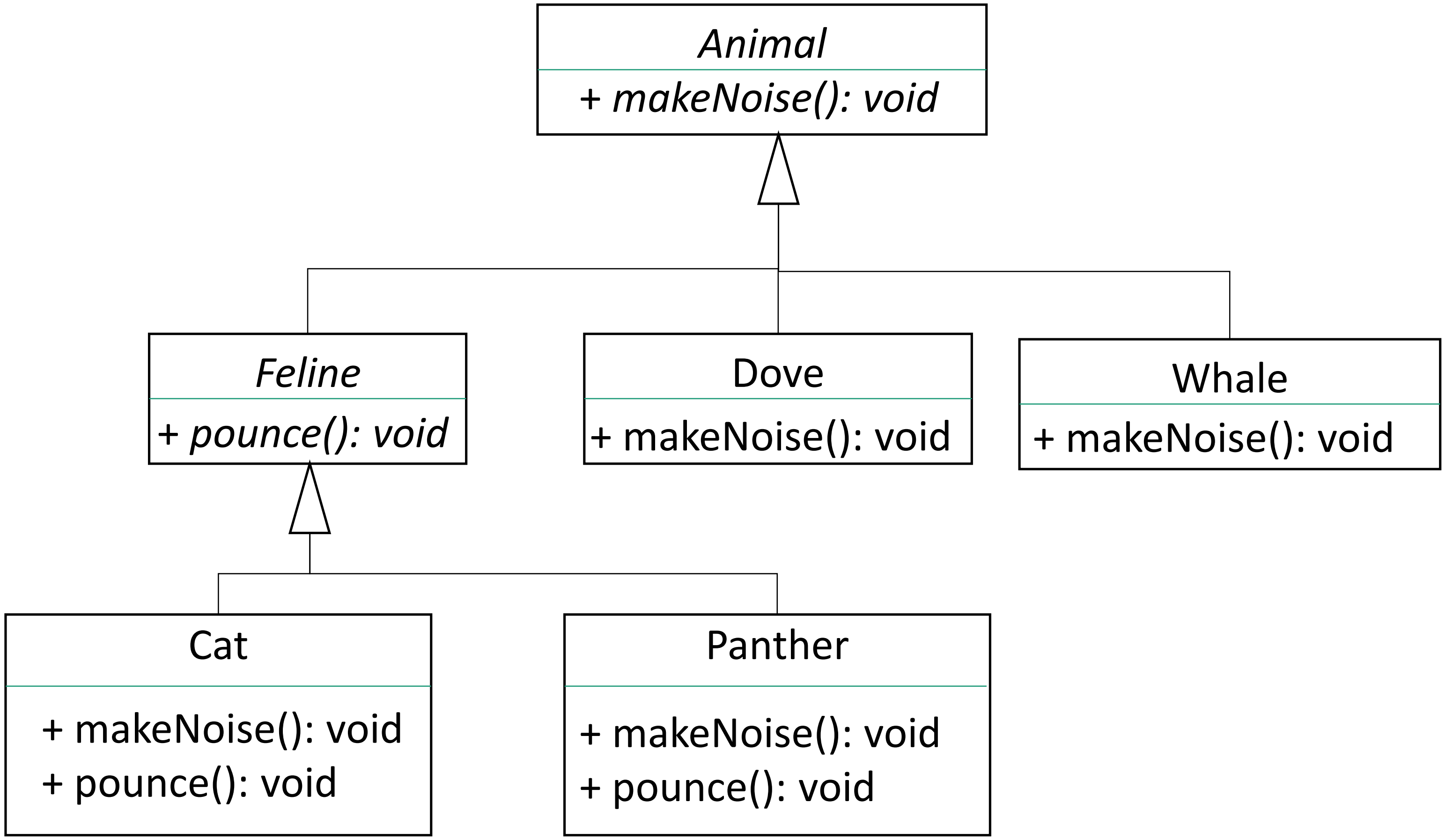
Exercise C18a-2

- In this exercise, you are to realize the UML class diagram in the slide.
  - Create directory C18a-2 in your weekly practice repo, and copy the programs in the "AbstractShape" directory in your answers to C18a-1 to the directory
  - Add an abstract class Triangle extending the Shape class as shown in the UML class diagram (in the slide).
  - Conforming to the UML class diagram, revise the RightTriangle class so that it extends the Triangle class
  - Conforming to the UML class diagram, create the EquilateralTriangle class
  - Revise the ShapeClient class to use all the concrete classes in the Shape class hierarchy, and also add statements to demonstrate the use of the isEquilateral method.
  - Use git to make a submission

Exercise C18a-3

- In this exercise, you are to realize the UML class diagram in next slide.
  - Create directory C18a-3 in your weekly practice repo, create all the classes from scratch in the UML class diagram shown in the next slide.
    - Note that Animal and Feline are abstract classes
    - Note that makeNoise and pounce are abstract methods. To implement the methods in concrete subclasses, simply print out an appropriate message, such as, "Cat purrs", "Panther roars", "Dove coos", "Whale clicks", "Cat pounces", "Panther pounces", etc.
  - Write a client class, called the AnimalClient class. In the client, write three methods
    - Two generic methods, one of which takes an ArrayList of Animals and invokes each Animal's makeNoise method, and the other of which takes an ArrayList of Felines and invokes each Feline's pounce method
    - A main method that demonstrates the use of the two generic methods.
  - Use git to make a submission

C18a-3: The Animal Class Hierarchy


Animal
+ *makeNoise(): void*
Feline
+ *pounce(): void*
Dove
+ makeNoise(): void
Whale
+ makeNoise(): void
Cat
+ makeNoise(): void
+ pounce(): void
Panther
+ makeNoise(): void
+ pounce(): void