OBJECT-ORIENTED SYSTEMS DESIGN

[Exercise]: Polymorphism and Abstract Classes

2022/05/04

Department of Computer Science,
Hanyang University

Lecturer: Taeuk Kim
TA: Taejun Yoon, Seong Hoon Lim, Young Hyun Yoo

Today's Plan

1. **Chapter Review: 20 min.**

2. **Practice: 30 min.**

Introduction to Polymorphism

- **There are three main programming mechanisms that constitute object-oriented programming (OOP).**
  - Encapsulation
  - Inheritance
  - **Polymorphism**

- **Polymorphism is the ability to associate many meanings to one method name.**
  - polymorphism overrides the method definition of the derived class and allows these changes to be applied to software written for the base class.
  - It does this through a special mechanism known as *late binding* or *dynamic binding.*

Details on Polymorphism

- **Q: Is overloading an example of polymorphism?**
  - **A: Maybe not.** In our book, polymorphism is considered **only for** cases where functions whose names are identical but differently defined (i.e. overidden functions).
  - For overloading, we **change the heading of each function** (such as the number and types of parameters). Therefore, **in a narrow perspective**, it is hard to say functions made with overloading are exactly the same.
  - However, some people say overloading is also a case of polymorphism **in a broad perspective,** which might be quite controversial.
  - As a conclusion, we do not consider overloading as polymorphism.

- **Q: Is overriding an example of polymorphism?**
  - **A: Yes. Overriding** (cause)**- polymorphism** (result) **- late binding** (solution) are closely related.
  - When we **override** (re-define) a function, there comes a possibility of **polymorphism**.
  - **Late binding** is the way of eliminating ambiguousness in Java when it comes to selecting a specific implementation of a function among various candidates.

Binding

```java
class SuperClass{
    public static void test(){
        System.out.println("Super Class");
    }
    public void test2(){
        System.out.println("SuperClass");
    }
}

class SubClass extends SuperClass{
    @Override
    public static void test(){
        System.out.println("Sub Class");
    }
    @Override
    public void test2() {
        System.out.println("Sub Class");
    }
}

public class BindingDemo {
    public static void main(String[] args) {
        SuperClass superClass = new SuperClass();
        SubClass subClass = new SubClass();

        superClass.test();
        subClass.test();

        superClass.test2();
        subClass.test2();
    }
}
```

- The process of associating a method definition with a method invocation is called binding.
- If the method definition is associated with its invocation when the method is invoked (at run time), that is called late binding or dynamic binding.
  - When the method definition is associated with a call at compile time, it is called an early binding or a static binding.
    Java uses static binding for **`private`**, **`final`**, and **`static`** methods.
    Therefore, a static method cannot be overridden by a derived class.

```
java: static methods cannot be annotated with @Override
```

Late Binding Example

```
class A{
    public String x(){
        return "A.x";
    }
}

class B extends A{
    public String y() { return "y"; }

    @Override
    public String x() {
        return "B.x";
    }
}

public class polymorphism_demo {
    public static void main(String[] args) {
        A obj = new B();
        System.out.println(obj.x());
        //System.out.println(obj.y()); Error!!
    }
}
```

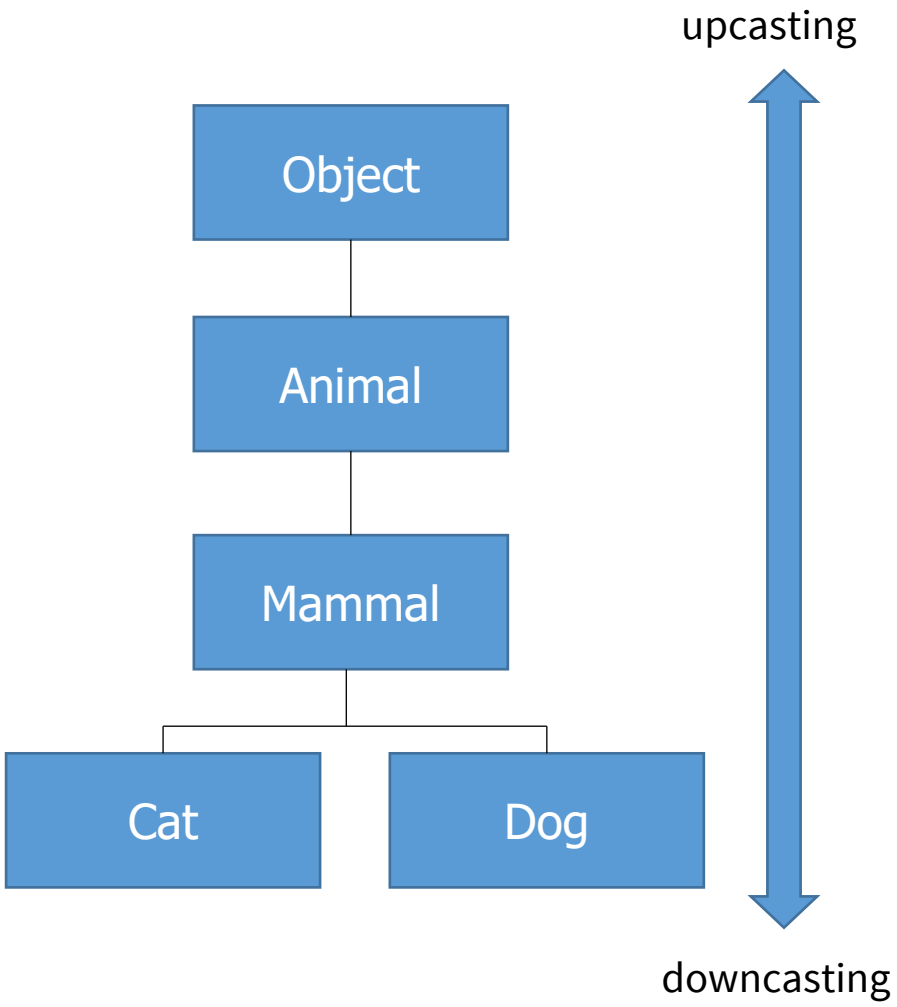
What does `obj.x()` return ?

```
B.x
```

`obj.x()` returns **B.x** not **A.x** !

Upcasting and Downcasting

- **Upcasting**
  - When an object of a derived class type is assigned to a variable of its base (or ancestor) class type.
- **Downcasting**
  - When type casting is performed from a base class to its derived class.



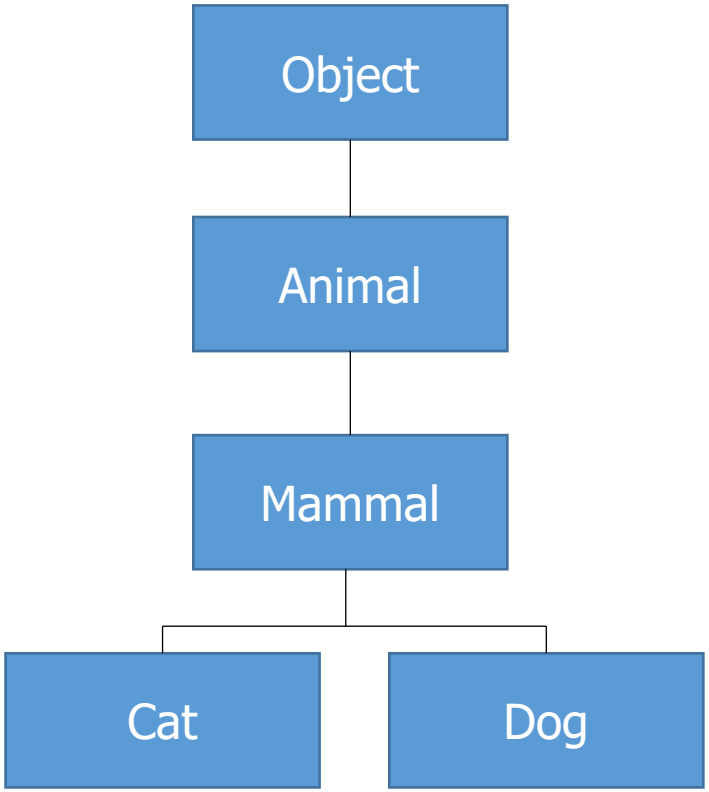
Upcasting and Downcasting

```
public static void main(String[] args) {
    Animal a1 = new Dog();
    //upcasting

    Dog d1 = new Animal();
    //Compile error

    Dog d2 = (Dog)new Animal();
    //Downcasting
    //throw ClassCastException

    Animal a2 = new Dog();  //upcasting
    Dog d3 = (Dog) a2;      //Downcasting
}
```



- **Downcasting** only makes sense when the object to be cast is an instance of the target class.

Abstract Method

- **An abstract method is like a placeholder for a method that will be fully defined in a descendent class.**

An abstract method must be public or protected.

Abstract method has no body and ends with a semicolon.

```
public abstract double getPay();
public abstract void doIt(int count);
```

Add `abstract` to declare an abstract method.

Abstract Class

```
abstract class Car{
    public abstract void start();
    public abstract void stop();

    public void engine (boolean run){
        if(run == true){
            System.out.println("Start");
        }else{
            System.out.println("Stop");
        }
    }
}
class Bus extends Car{
    @Override
    public void start() {
        engine( run: true);
    }
    @Override
    public void stop() {
        engine( run: false);
    }
}

public class abstract_demo {
    public static void main(String[] args) {
        Bus bus = new Bus();
        bus.start();
        bus.stop();
    }
}
```

An abstract class is a class that contains one or more abstract methods and therefore cannot be instantiated.

Abstract classes can have concrete methods.

A concrete class is a class that contains no abstract methods and therefore can be instantiated.

If **`Bus`** extends **`Car`**, **`Bus`** should override **`Car`**'s abstract methods **`start`** and **`stop`** (Otherwise, **`Bus`** would remain as an abstract class).

Practice

Exercise/WeekN/Taxi.java,
GeneralTaxi.java
DeluxeTaxi.java
Main.java

Taxi

- **Define a class `Taxi` .**
  - Define **`Taxi`** as an abstract class.
  - Taxi has three **instance variables**.
    - **`int carNum:`** ranges from 1000 to 9999.
    - **`double distance:`** total driving distance, initialized to 0 by a constructer.
    - **`double income:`** total income , initialized to 0 by a constructer.
  - A **Constructer** **`public Taxi(int carNum) .`**
  - Define the **`toString()`** method that returns the information of **`Taxi`**.
  - Define an abstract method **`getPaid(double distance) .`**
  - Define **`doDrive(double dis)`** that calls the **`getPaid`** method inside the method to update the total income.
  - Define **`earnMore(Taxi t)`** for comparing the total incomes of two **`Taxi`** objects.

GeneralTaxi

- **Define `GeneralTaxi` derived from `Taxi`.**
  - An instance variable `double farePerKilometer`.
  - An instance variable `double baseDistance` (initialized as 3).
  - An instance variable `double baseFee` (initialized as 3).
  - A Constructor
    - Parameters: as car number and a rate per kilometer.
    - Requirement: the value of `farePerKilometer` should be larger than (`baseFee / baseDistance`).
  - Override the `toString()` method.
  - Override the `getPaid(double dis)` method to calculate a driving fee.
    - The base fee is charged until the driving distance reaches the base distance.
    - Additional charges are requested for the extra distance (the distance – the base distance). In this case, `farePerKilometer` is utilized to compute the extra fee.

DeluxeTaxi

- **Define `DeluxeTaxi` derived from `Taxi`.**
  - An instance variable **`double farePerKilometer.`**
  - An instance variable **`double baseDistance`** (iniitialized as 3).
  - An instance variable **`double baseFee`** (initialized as 5).
  - **`DeluxeTaxi`** has a constructer.
    - Parameters: as car number and a rate per kilometer.
    - Requirement: the value of **`farePerKilometer`** should be larger than (**`baseFee / baseDistance`**).
  - Override a **`toString()`** method.
  - Override a **`getPaid(double dis)`** method to calculate a driving fee.
    - The rule is same as that of **`GeneralTaxi.`**

Main

- **`main` method**
  - Create a **`GeneralTaxi t1`** and a **`DeluxeTaxi t2`** that are upcasted as **`Taxi`**.
  - Print the status of taxi before and after driving using the **`toString()`** method.
  - Print out which taxi will earn more money using the **`earnMore()`** method.

```
public static void main(String[] args) {
    Taxi t1 = new GeneralTaxi( car_num: 1234, farePerKilometer: 2.1);
    Taxi t2 = new DeluxeTaxi( car_num: 2345,  farePerKilometer: 6.1);

    System.out.println(t1.toString());
    System.out.println(t2.toString());

    t1.doDrive( dis: 5.2);
    t1.doDrive( dis: 2.4);

    t2.doDrive( dis: 5);

    System.out.println(t1.toString());
    System.out.println(t2.toString());
    if (t1.earnMore(t2)){
        System.out.println("t1 earn more than t2");
    }else {
        System.out.println("t2 earn more than t1");
    }
}
```

Time for Practice

Get it started, and ask TAs if you are in a trouble.