CS 61B Discussion 4: Inheritance Fall 2017

1 Creating Cats

Given the `Animal` class, fill in the definition of the `Cat` class so that it makes a "Meow!" noise when `greet()` is called. Assume this noise is all caps for kittens (less than 2 years old).

```
1  public class Animal {
2      protected String name, noise;
3      protected int age;
4      public Animal(String name, int age) {
5          this.name = name;
6          this.age = age;
7          this.noise = "Huh?";
8      }
9      public String makeNoise() {
10         if (age < 2) {
11             return noise.toUpperCase();
12         }
13         return noise;
14     }
15     public String greet() {
16         return name + ": " + makeNoise();
17     }
18 }

   class Cat extends Animal {
       public Cat(String name, int age) {
           super(name, age);
           this.noise = "Meow!";
       }
   }
```

Inheritance is powerful because it allows us to reuse code for related classes. With the `Cat` class here, we just have to re-write the constructor to get all the goodness of the `Animal` class.

A common question at this point may be, "Why is it necessary to call `super(name, age);` within the `Cat` constructor?" Great question! I'm glad you asked.

It turns out that a subclass' constructor by default always calls the superconstructor. If we didn't specify the call to the `Animal` superconstructor that takes in a `String` and a `int`, we'd get a compiler error. This is because the default superconstructor (`super();`) would have been called. Only problem is that the `Animal` class has no such zero-argument constructor!

By explicitly calling `super(name, age);` in the first line of the `Cat` constructor, we avoid calling the default superconstructor.

2 Impala-ments

a) We have two interfaces, `BigBaller` and `ShotCaller`. `LilTroy`, a concrete class, should implement `BigBaller` and `ShotCaller`. Fill out the blank lines below so that the code compiles correctly.

```
interface BigBaller {
    void ball();
}
interface ShotCaller {
    void callShots();
}
public class LilTroy implements BigBaller, ShotCaller {
    public void ball() {
        System.out.println("Wanna be a, baller");
    }
    public void callShots() {
        System.out.println("Shot caller");
    }
    public void rap() {
        System.out.println("Say: Twenty inch blades on the Impala");
    }
}
```

b) We have a `BallCourt` where ballers should be able to come and play. However, the below code demonstrates an example of bad program design. Right now, only `LilTroy` instances can ball.

```
public class BallCourt {
    public void play(LilTroy lilTroy) {
        lilTroy.ball();
    }
}
```

Fix the `play` method so that all the `BigBallers` can ball.

```
public class BallCourt {
    public void play(BigBaller baller) {
        baller.ball();
    }
}
```

c) We discover that `Rappers` have some common behaviors, leading to the following class.

```
class Rapper {
    public abstract String getLine();
    public final void rap() {
        System.out.println("Say: " + getLine());
    }
}
```

Will the above class compile? If not, why not? How can we fix it? This class will NOT compile. `Rapper` class has a method names `getLine`, which is declared `abstract`. It does not have any method implementation. Would it be possible to create an object from a class where a method

lacks the implementation? Definitely not! By adding the `abstract` keyword before the `class` keyword, the class will compile normally. The first line should look like `abstract class Rapper`.

d) Rewrite `LilTroy` so that `LilTroy` extends `Rapper` and displays exactly the same behavior as in part a) *without* overriding the `rap` method (in fact, you *cannot* override final methods).

```
class LilTroy extends Rapper implements BigBaller, ShotCaller {

    @Override
    public void ball() {
        System.out.println("Wanna be a, baller");
    }

    @Override
    public void callShots() {
        System.out.println("Shot caller");
    }

    @Override
    public String getLine() {
        return "Twenty inch blades on the Impala";
    }
}
```

Note that most of the `Rapper`'s implementation can be reused in all its subclasses, as long as they correctly implement `getLine`. `Rapper` captures a reusable and common behavior (`rap`), while delegating some parts of implementations to its subclasses.

3 Raining Cats & Dogs

We now have the `Dog` class! (Assume that the `Cat` and `Dog` classes are both in the same file as the `Animal` class.)

```
1 class Dog extends Animal {
2     public Dog(String name, int age) {
3         super(name, age);
4         noise = "Woof!";
5     }
6     public void playFetch() {
7         System.out.println("Fetch, " + name + "!");
8     }
9 }
```

Consider the following `main` function in the `Animal` class. Decide whether each line causes a compile time error, a runtime error, or no error. If a line works correctly, draw a box-and-pointer diagram and/or note what the line prints.

```
public static void main(String[] args) {

    Cat nyan = new Animal("Nyan Cat", 5);     (A) compile time error
```

The static type of `nyan` must be the same class or a superclass of the dynamic type. It doesn't make sense for the dynamic type to be the superclass of the static type.

```
Animal a = new Cat("Olivia Benson", 3);   (B) no error
a = new Dog("Fido", 7);                   (C) no error
System.out.println(a.greet());            (D) "Fido: Woof!"
a.playFetch();                            (E) compile time error
```

The compiler attempts to find the method `playFetch` in the `Animal` class (`a`'s static type). Because it does not find it there, there is an error because the compiler does not check the `Dog` class (dynamic type) at compile time.

```
Dog d1 = a;                               (F) compile time error
```

The compiler views the type of variable `a` to be `Animal` because that is its static type. It doesn't make sense to assign an `Animal` to a `Dog` variable.

```
Dog d2 = (Dog) a;                         (G) no error
```

The `(Dog) a` part is a cast. Casting tells the compiler to treat `a` as if it were a `Dog`. Casting changes the compiler's perception of a variable's dynamic type for the one line of the cast. After that line, `a`'s static type goes back to being `Animal`.

```
d2.playFetch();                           (H) "Fetch, Fido!"
(Dog) a.playFetch();                      (I) compile time error
```

Parentheses are important when casting. Here, the cast happens after `a.playFetch()` is evaluated. The return type of `playFetch()` is `void`, and it makes no sense to cast something `void` to a `Dog`. This is simply invalid. Something that would work is: `((Dog) a).playFetch();`

```
Animal imposter = new Cat("Pedro", 12);   (J) no error
Dog fakeDog = (Dog) imposter;             (K) runtime error
```

The compiler sees that we'd like to treat `imposter` like a `Dog`. `imposter`'s static type is `Animal`, so it's possible that its dynamic type is actually `Dog`. However, at runtime, when the cast actually happens, we see a `ClassCastException` because the dynamic type of `imposter` (`Cat`) is not compatible with `Dog`.

```
Cat failImposter = new Cat("Jimmy", 21); (L) no error
Dog failDog = (Dog) failImposter;         (M) compile time error
```

The compiler sees that we'd like to treat `failImposter` like a `Dog`. However, unlike the example above, `failImposter`'s static type is `Cat`, so it's impossible that its dynamic type is actually `Dog`. Thus, the compiler states that these are inconvertible (incompatible) types.

```
}
```

4 Bonus: An Exercise in Inheritance Misery

Cross out any lines that cause compile or runtime errors. What does the `main` program output after removing those lines?

```
class A {
    int x = 5;
    public void m1() {System.out.println("Am1-> " + x);}
    public void m2() {System.out.println("Am2-> " + this.x);}
    public void update() {x = 99;}
}
class B extends A {
    int x = 10;
    public void m2() {System.out.println("Bm2-> " + x);}
    public void m3() {System.out.println("Bm3-> " + super.x);}
    public void m4() {System.out.print("Bm4-> "); super.m2();}
}
class C extends B {
    int y = x + 1;
    public void m2() {System.out.println("Cm2-> " + super.x);}
    /* public void m3() {System.out.println("Cm3-> " + super.super.x);} */
```

`super.super` is invalid syntax.

```
    public void m4() {System.out.println("Cm4-> " + y);}
    /* public void m5() {System.out.println("Cm5-> " + super.y);} */
```

`C`'s superclass `B`, and `B`'s superclass `A` both don't have the variable `y`.

```
}
class D {
    public static void main (String[] args) {
        A b0 = new B();
        System.out.println(b0.x);    (A) 5
        b0.m1();                     (B) Am1->5
        b0.m2();                     (C) Bm2->10
        /* b0.m3(); */               (D) compile time error because A does
                                         not have method m3

        B b1 = new B();
        b1.m3();                     (E) Bm3->5
        b1.m4();                     (F) Bm4->Am2->5

        A c0 = new C();
        c0.m1();                     (G) Am1->5

        A a1 = (A) c0;
        C c2 = (C) a1;
        c2.m4();                     (H) Cm4->11
        ((C) c0).m3();               (I) Bm3->5

        b0.update();
        b0.m1();                     (J) Am1->99
    }
}
```