Discussion 5: Inheritance

Hi everyone! My name is Adel and I will be your CS 61B TA this semester 🙂

Discussion: Wed 2-3pm 3111 Etcheverry Hall
Lab: Fri 1-3pm 275 Soda Hall
OH: Tues 3-4pm in 109 Morgan Hall

email: asetoodehnia@berkeley.edu
website: asetoodehnia.github.io (or find it through the CS 61B staff webpage)
(I post my discussion materials here if you are interested in referring to them after section)

Announcements!

1. Midterm 1 is **Thurs, Feb. 27**
2. HW3 Due **Fri, Feb. 21**
3. HKN/CSM review sessions this weekend!
4. Extra Topical Section **Fri, Feb. 21**

public class A implements –, –, –

Interfaces and Abstract Classes

- Interfaces are similar to classes, except rather than describe an object they describe a capability of an object (and you cannot create an instance of an interface)
  - Basically just a blueprint for other classes you want to create in the future
  - All methods are abstract and not concrete, must be overwritten by the class that implements the interface
  - Classes can implement many interfaces
- Abstract classes are basically the same but with a few key differences
  - Methods can either be abstract or concrete
  - The subclass that extends this can then override these concrete methods if it chooses to, and must implement the abstract methods
  - Classes can extend only one abstract class

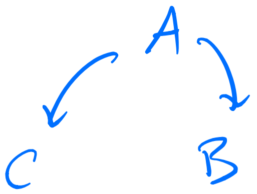
Access Modifiers

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected:** The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

NOTE: To override an abstract method, the method signature's access modifiers must match exactly!!!

Static and Dynamic Type

- **Static Types** are the types of containers
  - This is what is checked by the compiler, and any inconsistencies in the static types can lead to compile time errors
- **Dynamic Types** are the types of a value
  - This is determined at runtime, and any inconsistencies here can lead to runtime errors



```
B b = new A();
A a = new C();
```

CS 61B Discussion 5: Inheritance Spring 2020

1 Classy Cats

Look at the `Animal` class defined below.

```java
public class Animal {
    protected String name, noise;
    protected int age;

    public Animal(String name, int age) {
        this.name = name;
        this.age = age;
        this.noise = "Huh?";
    }

    public String makeNoise() {
        if (age < 2) {
            return noise.toUpperCase();
        }
        return noise;
    }

    public String greet() {
        return name + ": " + makeNoise();
    }
}
```

(a) 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, i.e. `Cats` that are less than 2 years old.

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

        }
}
```

(b) "Animal" is an extremely broad classification, so it doesn't really make sense to have it be a class. Look at the new definition of the `Animal` class below.

```
public abstract class Animal {
    protected String name;
    protected String noise = "Huh?";
    protected int age;

    public String makeNoise() {
        if (age < 2) {
            return noise.toUpperCase();
        }
        return noise;
    }

    public String greet() {
        return name + ": " + makeNoise();
    }

    public abstract void shout();
    abstract void count(int x);
}
```

Fill out the `Cat` class again below to allow it to be compatible with `Animal` (which is now an abstract class) and its two new methods.

```
public class Cat extends Animal {
    public Cat() {
        this.name = "Kitty";
        this.age = 1;
        this.noise = "Meow!";
    }

    public Cat(String name, int age) {
        this();
        this.name = name;
        this.age = age;
    }

    @Override
    __public__ ___void___ shout() {
        System.out.println(noise.toUpperCase());
    }

    @Override
    __________ ___void___ count(int x) {
        for (int i = 0; i < x; i++) {
            System.out.println(makeNoise());
        }
    }
}
```

2 The Interfacing CatBus

After discovering that we can implement the `Cat` class with minimal effort, Professor Hilfinger decided that he wants to create a `CatBus` class. `CatBuses` are `Cats` that act like vehicles and have the ability to honk (safety is important!).

a) Given the `Vehicle` and `Honker` interfaces, fill out the `CatBus` class so that `CatBuses` can rev their engines and honk at other `CatBuses`.

```
interface Vehicle {
    /** Gotta go fast! */
    public void revEngine();
}
                                                default void honk();

interface Honker {
    /** HONQUE! */
    void honk();
}

public class CatBus extends Cat, implements Vehicle, Honker {

    @Override
→   public void revEngine() {
        System.out.println("Purrrrrrr");
    }

    @Override
    public void honk() {
        System.out.println("CatBus says HONK");
    }

    /** Allows CatBus to honk at other CatBuses. */
    public void conversation(CatBus target, int duration) {
        for (int i = 0; i < duration; i++) {
            honk();
            target.honk();
        }
    }
}
```

b) After a few hours of research, Professor Hilfinger discovered that animals of type `Goose` are also avid `Honkers`! Modify the `conversation` method so that `CatBuses` can `honk` at `CatBuses` *and* `Goosees`.

```
/** Allows CatBus to honk ANY target that can honk back. */

public void conversation(Honker target, int duration) {
    for (int i = 0; i < duration; i++) {
        honk();
        target.honk();
    }
}
```

3 Raining Cats & Dogs

In addition to Animal and Cat from Problem 1a, we now have the Dog class! (Assume that the Cat and Dog classes are both in the same file as the Animal class.)

```java
class Dog extends Animal {
    public Dog(String name, int age) {
        super(name, age);
        noise = "Woof!";
    }
    public void playFetch() {
        System.out.println("Fetch, " + name + "!");
    }
}
```

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. It may be useful to refer to the Animal class back on the first page.

```java
public static void main(String[] args) {
    Cat nyan = new Animal("Nyan Cat", 5);     (A) compile error

    Animal a = new Cat("Olivia Benson", 3);   (B) ✓
    a = new Dog("Fido", 7);                   (C) ✓
    System.out.println(a.greet());            (D) Fido: woof!
    a.playFetch();                            (E) compile error

    Dog d1 = a;                               (F) ____________
    Dog d2 = (Dog) a;                         (G) ____________
    d2.playFetch();                           (H) ____________
    (Dog) a.playFetch();                      (I) ____________

    Animal imposter = new Cat("Pedro", 12);   (J) ____________
    Dog fakeDog = (Dog) imposter;             (K) ____________

    Cat failImposter = new Cat("Jimmy", 21); (L) ____________
    Dog failDog = (Dog) failImposter;         (M) ____________
}
```

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?

Moral of the story: Fields become hidden when you redefine them in the subclass. If possible, you should avoid doing so or else your code may become confusing.

```
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);}
    public void m4() {System.out.println("Cm4-> " + y);}
    public void m5() {System.out.println("Cm5-> " + super.y);}
}
class D {
    public static void main (String[] args) {
        A b0 = new B();
        System.out.println(b0.x);    (A) _____________________
        b0.m1();                     (B) _____________________
        b0.m2();                     (C) _____________________
        b0.m3();                     (D) _____________________

        B b1 = new B();
        b1.m3();                     (E) _____________________
        b1.m4();                     (F) _____________________

        A c0 = new C();
        c0.m1();                     (G) _____________________

        A a1 = (A) c0;
        C c2 = (C) a1;
        c2.m4();                     (H) _____________________
        ((C) c0).m3();               (I) _____________________

        b0.update();
        b0.m1();                     (J) _____________________
    }
}
```