CS 61B Spring 2022

Inheritance

Discussion 5: February 14, 2022

1 Classy Cats

Look at the `Animal` class defined below.

```
1  public class Animal {
2      protected String name, noise;
3      protected int age;
4
5      public Animal(String name, int age) {
6          this.name = name;
7          this.age = age;
8          this.noise = "Huh?";
9      }
10
11     public String makeNoise() {
12         if (age < 2) {
13             return noise.toUpperCase();
14         }
15         return noise;
16     }
17
18     public String greet() {
19         return name + ": " + makeNoise();
20     }
21 }
```

(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. `Cat`s that are less than 2 years old.

```
public class Cat extends Animal {









}
```

(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
    ____________ ____________() {
        System.out.println(noise.toUpperCase());
    }

    @Override
    ____________ ____________(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. `CatBus`es are `Cat`s 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 `CatBus`es can rev their engines and honk at other `CatBus`es.

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

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

public class CatBus extends _______, implements _____________, _____________ {

    @Override
    _____________ _____________ revEngine() {
        System.out.println("Purrrrrrr");
    }

    @Override
    _____________ _____________ 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 `CatBus`es can `honk` at `CatBus`es *and* `Goose`es.

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

public void conversation(_____________ 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.)

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

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

    Animal a = new Cat("Olivia Benson", 3);  (B) _____________________________
    a = new Dog("Fido", 7);                  (C) _____________________________
    System.out.println(a.greet());           (D) _____________________________
    a.playFetch();                           (E) _____________________________

    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 Back to ABC's!

Suppose we have the `A`, `B`, and `C` classes defined below.

```
class A {
    int x = 1;
    void f(A other) { System.out.println(x); }
    void f(B other) { System.out.println(x + 2); }
    static void h() { System.out.println("A.h"); }
}

class B extends A {
    int x = 2;
    void f(A other) { System.out.println(x); }
    static void h() { System.out.println("B.h"); }
}
```

For each line below, write what, if anything, is printed after its execution. Write CE if there is a compiler error and RE if there is a runtime error. If a line errors, continue executing the rest of the lines.

```
A aa = new A();
B bb = new B();
A ab = new B();
C ca = new A();
C cb = new B();

aa.f(ab);
ab.f(aa);
bb.f(ab);
ab.f(bb);
bb.f(bb);
ab.h();
bb.h();
((A) bb).h();
```

5 Flatten

Write a method `flatten` that takes in a 2-D array x and returns a 1-D array that contains all of the arrays in x concatenated together.

For example, `flatten({{1, 2, 3}, {}, {7, 8}})` should return `{1, 2, 3, 7, 8}`.

```
1  public static int[] flatten(int[][] x) {
2      int totalLength = 0;
3
4      for (_____________________________________) {
5
6          ___________________________________________
7      }
8
9      int[] a = new int[totalLength];
10     int aIndex = 0;
11     for (_____________________________________) {
12
13         ___________________________________________
14
15         ___________________________________________
16
17         ___________________________________________
18
19         ___________________________________________
20     }
21
22     return a;
23 }
```