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 {
}
```
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.

Why is it necessary to call `super(name, age);` within the `Cat` constructor? It turns out that a subclass's constructor by default always calls its parent class's constructor (aka a super constructor). If we didn't specify the call to the `Animal` super constructor that takes in a `String` and a `int`, we'd get a compiler error. This is because the default super constructor (`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 super constructor.

Similarly, not providing any explicit constructor at all in the `Cat` implementation would also result in code that does not compile. This is because when there are no constructors available in a class, Java automatically inserts a no-argument constructor for you. In that no-argument constructor, Java will then attempt to call the default super constructor, which again, does not exist.

Also note that declaring a `noise` field at the top of the `Cat` class would not be correct. Since in Java, fields are bound at compile time, when the parent class's `makeNoise()` function calls upon `noise`, we will receive "Huh?". Because of this confusing subtlety of Java, which is called field hiding, it is generally a bad idea to have an instance variable in both a superclass and a subclass with the same name.
(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.

```java
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.

```java
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());
        }
    }
}
```
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
    public void count(int x) {
        for(int i = 0; i < x; i++) {
            System.out.println(makeNoise());
        }
    }
}

To override an abstract method, the method signature’s access modifiers must match exactly. Since shout is declared to be public abstract in Animal, our Cat class must declare it to be public to ensure that access modifiers match. The default access modifier for abstract classes is the same as the default access modifier for regular Java classes. Since count has the default access modifier in the Animal abstract class, count has the default access modifier when we override it in the Cat class.
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.

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

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

public class CatBus extends Cat, implements Vehicle, Honker {
    public void revEngine() {
        System.out.println("Purrrrrrr");
    }

    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 Gooses.

```java
/** Allows CatBus to honk at 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) _____________________________
    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) _____________________________
}
```
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 - i.e. in this example, not all Animals are Cats, so an attempt at a dangerous initialization like this would be caught as an error. Note that doing the opposite, as in the next line, is fine, since all Cats are Animals.

    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, as in the first error case.

    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 tells the compiler to treat the following variable as a specified static type, and its effects only last for the line on which it was used. 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. More formally, when casting to a specific type, the new type must be in the same inheritance hierarchy as the existing type (in this case, void (i.e. null) isn’t in the same inheritance family as Dog, since it can never be a Dog). 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. Since imposter’s static type is Animal, so it’s actually possible that its dynamic type is Dog, so the casting will compile (unlike in the previous case). However, at runtime, we see a ClassCastException because imposter’s dynamic type (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 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();
Solution:

```java
A aa = new A(); //None
B bb = new B(); //None
A ab = new B(); //None
C ca = new A(); //CE
C cb = new B(); //CE

aa.f(ab); //1
ab.f(aa); //2
bb.f(ab); //2
ab.f(bb); //3
bb.f(bb); //3
ab.h(); //A.h
bb.h(); //B.h
((A) bb).h(); //A.h
```

Explanation:

Lines 1-3 initialize instances using the default Java constructor, which produces no output. Lines 4 and 5 violate the rule that variables' static types must be a parent or the same class as their dynamic type.

**Line 7:** `aa` has the same static and dynamic type `A`, and `A.f(A other)` is allowed to take in a type `A` argument. We output the result of `A.f(A other)`, which prints `A.x=1`.

**Line 8:** `ab` has static type `A` and dynamic type `B`. During compile time, we bind `ab.f(aa)` to `A.f(A other)` (we only use static types during compile time, and `aa` is of static type `A`). During runtime, we override `A.f(A other)` with `B.f(A other)` due to `ab`'s dynamic type. During override, `B.f(A other)` prints `B.x`, which is 2.

**Line 9:** `bb` has static type and dynamic type `B`. During compile time, we bind `bb.f(ab)` to `B.f(A other)`. During runtime, no override occurs (arguments' dynamic types are not considered during DMS), and we execute `B.f(A other)`, which prints `B.x=2`.

**Line 10:** `ab` has static type `A` and dynamic type `B`. During compile time, we bind `ab.f(bb)` to `A.f(B other)`. During runtime, `ab`'s type becomes `B`, but no override occurs (remember that in order to override, the dynamic class must have a method signature that exactly matches `void f(B other)`). So because no override occurs, we print `3`.

**Line 11:** `bb` has static type `B` so the compiler first looks for a method in class `B` with header `f(B other)`. It doesn’t find a method with that header, so it then looks at the superclass `A` of `B` and finds the method with header `f(B other)`. At runtime, the dynamic type of `bb` is the same as the static type, so nothing is overridden and the method `A.f(B other)`. This method prints the value of `x + 2`, which evaluates to `3`.

**Line 12:** `ab` has static type `A` so the static method `A.h()` is called and prints `A.h`. Static methods are determined at compile time, so it is not overridden. In general, static methods bound at compile time cannot be overridden by DMS, even if a more specific method exists.
**Line 13:** bb has static type B so the static method B.h() is called and prints B.h.

**Line 14:** bb is cast to static type A so the static method A.h() is called and prints A.h. Again, static methods are determined at compile time, so it is not overridden.
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}`.

```java
public static int[] flatten(int[][] x) {
    int totalLength = 0;

    for (____________________________________) {
        ________________________________________________
    }

    int[] a = new int[totalLength];
    int aIndex = 0;
    for (____________________________________) {
        ________________________________________________
    }

    return a;
}
```
Solution:

```java
public static int[] flatten(int[][] x) {
    int totalLength = 0;
    for (int[] arr: x) {
        totalLength += arr.length;
    }
    int[] a = new int[totalLength];
    int aIndex = 0;
    for (int[] arr: x) {
        for (int value: arr) {
            a[aIndex] = value;
            aIndex++;
        }
    }
    return a;
}
```

Alternate Solutions:

```java
public static int[] flatten(int[][] x) {
    int totalLength = 0;
    for (int[] arr: x) {
        totalLength += arr.length;
    }
    int[] a = new int[totalLength];
    int aIndex = 0;
    for (int[] arr: x) {
        System.arraycopy(arr, 0, a, aIndex, arr.length);
        aIndex += arr.length;
    }
    return a;
}
```

```java
public static int[] flatten(int[][] x) {
    int totalLength = 0;
    for (int i = 0; i < x.length; i++) {
        totalLength += x[i].length;
    }
    int[] a = new int[totalLength];
    int aIndex = 0;
    for (int i = 0; i < x.length; i++) {
        for (int j = 0; j < x[i].length; j++) {
            a[aIndex] = x[i][j];
            aIndex++;
        }
    }
    return a;
}
```
Here is a video walkthrough of the solutions for this problem.

**Explanation:** All these solutions do essentially the same thing. In Java, an array’s length must be known before we can instantiate it–as such, we have to loop over all inner arrays to get the `totalLength` of our flattened array. Then, we iterate over the elements of `x`, filling `a` as we go. `aIndex` keeps track of where we are in the `a` array.