

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. Cats that are less than 2 years old.

```
public 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.

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.

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

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());
        }
    }
}
```

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`.

```
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("Purrrrrrrr");
    }

    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* `Geese`s.

```
/** 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.)

```
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) 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 Extra: 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);} */
```

super.super is invalid syntax. You cannot actually access the grandparent's x from this grand-child class in this case, since B's variable of the same name "hides" it. It'd be possible if B had a helper method that accessed its parent's (A's) x variable, which doesn't exist here.

```
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. If you're curious, you can read more about field hiding at this link.

```
} 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; no m3() in A.

        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
```