Inheritance

Discussion 04

Announcements

Week 4

- ❏ Project 1A is due Monday (9/14)
- ❏ Project 1B is due Friday (9/18)
- ❏ Don't be afraid to use slip days!
- ❏ Lab 4 has mandatory checkoff - but only go after you've turned in project 1A
- ❏ Do the weekly survey

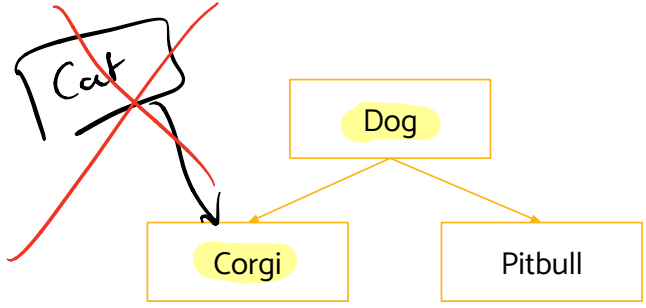
Content Review

Classes

Subclasses (or child classes) are classes that extend another class. This means that they have access to all of the functions and variables of their parent class in addition to any functions and variables defined in the child class.

Superclasses or parent classes are classes that are extended by another class.

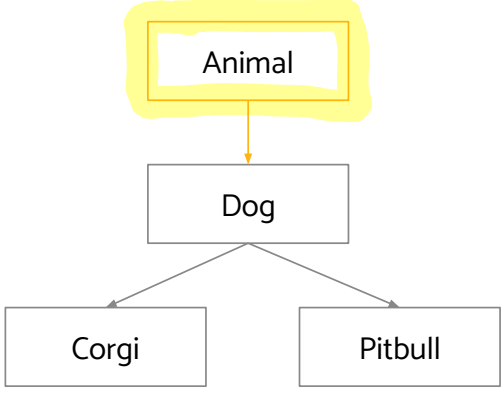
Classes can only extend one class but can be extended by many classes.



Abstract Classes

Abstract Classes are classes that cannot be directly referenced. Instead, they must be extended by a **concrete class**. They describe all of the functions that a class of their "type" must be able to do, with or without implementation.

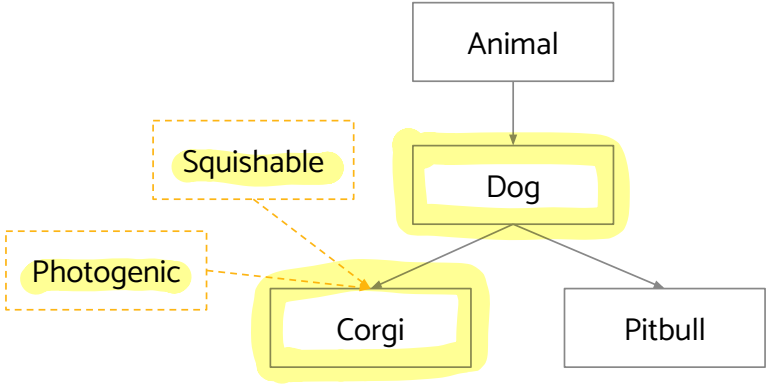
Ex. List



Interfaces

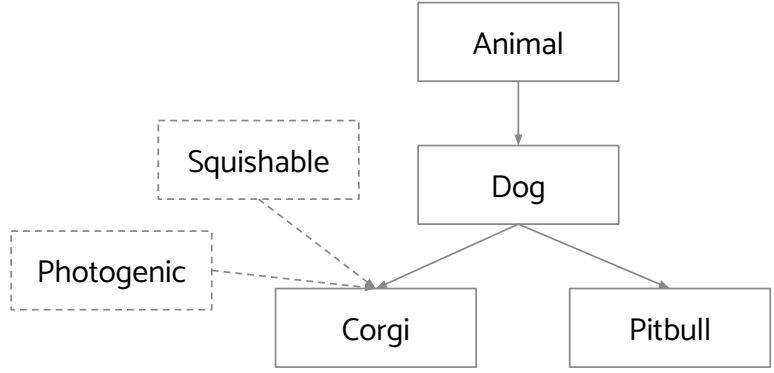
Interfaces are implemented by classes. They describe a narrow ability that can apply to many classes that may or may not be related to one another. They are like abstract classes in that they do not usually implement the methods they specify. One class can implement many interfaces.

Ex. Comparable



Implementation

```
abstract class Animal {...}

interface Squishable {...}

interface Photogenic {...}

class Dog extends Animal {...}

class Pitbull extends Dog {...}

class Corgi extends Dog implements Squishable, Photogenic {...}
```



Fun with Methods

Method Overloading is done when there are multiple methods with the same name and return type, but different parameters.

```
public void barkAt(Dog d) { System.out.print("Woof, it's another dog!"); }
public void barkAt(Animal a) { System.out.print("Woof, what is this?"); }
```

Method Overriding is done when a subclass has a method with the exact same function signature as a method in its superclass.

- *In Dog class:*

```
public void speak() { System.out.print("Woof, I'm a dog!"); }
```

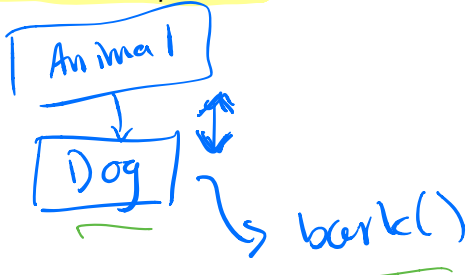
- *In Corgi Class:*

```
public void speak() { System.out.print("Woof, I'm a corgi!"); }
```

Casting

Casting allows our compiler to overlook cases where we are calling a method that belongs to a subclass on a variable that is statically typed to be the superclass.

```
Animal a = new Dog();
Dog d = a;
Dog d = (Dog) a;
```



✗ a.bark();

✓ ((Dog)a).bark();

Dynamic Method Selection

Your computer...

@ Compile Time:

1. Check for valid variable assignments
2. Check for valid method calls (only considering static type)

@ Run Time:

1. Check for overridden methods
2. Ensure casted objects can be assigned to their variables

Worksheet

CS 61B
Fall 2020

Inheritance

Discussion 4: September 14, 2020

1 JUnit Tests

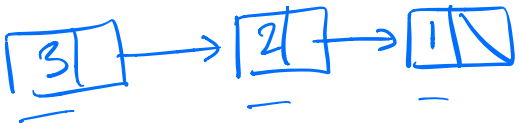
(a) What are the advantages and disadvantages of writing JUnit tests?

Pros: - easy to use
- compare results/ debug tool

Cons:
- hard to use for higher-levels

(b) Think about the lab you did last week where we did JUnit testing. Fill in the following tests so that they test the constructor and `dSquareList` functions of `IntList`.

```
1  public class IntListTest {
2
3      @Test
4      public void testList() {
5          IntList one = new IntList(1, null);
6          IntList twoOne = new IntList(2, one);
7          IntList threeTwoOne = new IntList(3, twoOne);
8
9          IntList x = IntList.list(_____3, 2, 1_____);
10         assertEquals(_____x_____, _____threeTwoOne_____);
11     }
12
13     @Test
14     public void testdSquareList() {
15         IntList L = IntList.list(1, 2, 3);
16         IntList.dSquareList(L);
17         assertEquals(_____IntList.list(1,4,9)_____, _____L_____);
18     }
19 }
```



Expected

actual

2 Creating Cats

Given the `Animal` class, fill in the definition of the `Cat` class so that when `greet()` is called, "Cat [name] says: Meow!" is printed (instead of "Animal [name] says: Huh?"). Cats less than the ages of 5 should say "MEOW!" instead of "Meow!". Don't forget to use `@Override` if you are writing a function with the same signature as a function in the superclass.

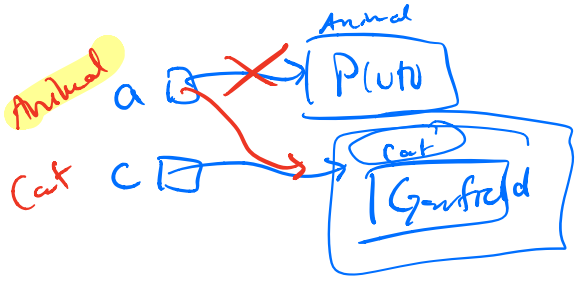
```
1  public class Animal {
2      protected String name, noise;
3      protected int age;
4
5      public Animal(String name, int age) {
6          this.name = name;
7          this.age = age;
8          this.noise = "Huh?";
9      }
10
11     public String makeNoise() {
12         if (age < 5) {
13             return noise.toUpperCase();
14         } else {
15             return noise;
16         }
17     }
18
19     public void greet() {
20         System.out.println("Animal " + name + " says: " + makeNoise());
21     }
22 }
```

```
public class Cat extends Animal {

    public Cat(String name, int age) {
        super(name, age);       // Call superclass' constructor.
        this.noise = "Meow!";   // Change the value of the field.
    }

    @Override
    public void greet() {
        System.out.println("Cat " + name + " says: " + makeNoise());
    }
}
```

3 Raining Cats and Dogs

(a) Assume that `Animal` and `Cat` are defined as above. What would Java print on each of the indicated lines?

```
1  public class Dog extends Animal {
2      public Dog(String name, int age) {
3          super(name, age);
4          noise = "Woof!";
5      }
6
7      @Override
8      public void greet() {
9          System.out.println("Dog " + name + " says: " + makeNoise());
10     }
11
12     public void playFetch() {
13         System.out.println("Fetch, " + name + "!");
14     }
15 }
16
17 public class TestAnimals {
18     public static void main(String[] args) {
19         Animal a = new Animal("Pluto", 10);
20         Cat c = new Cat("Garfield", 6);
21         Dog d = new Dog("Fido", 4);
22         a.greet();              // (A) Animal Pluto says: Huh?
23         c.greet();              // (B) Cat Garfield says: Meow!
24         d.greet();              // (C) Dog Fido says: WOOF!
25         a = c;
26         ((Cat) a).greet();      // (D) Cat Garfield says: Meow!
27         a.greet();              // (E) Cat Garfield says: Meow!
28     }
29 }
```



(b) Consider what would happen if we added the following to the bottom of `main` under line 27:

```
a = new Animal("Fluffy", 2);
c = a;
```

Would this code produce a compiler error? What if we set the second line to be `c = (Cat) a` instead?

(c) Consider what would happen if we instead added the following to the bottom of `main` under line 27:

```
a = new Dog("Spot", 10);
d = a;
```

Why would this code produce a compiler error? How could we fix this error?

4 An Exercise in Inheritance Misery *Extra*

Cross out any lines that cause compile-time errors or cascading errors (failures that occur because of an error that happened earlier in the program), and put an X through runtime errors (if any). Don't just limit your search to main, there could be errors in classes A,B,C. What does `D.main` output after removing these lines?

```
1  class A {
2      public int x = 5;
3      public void m1() {      System.out.println("Am1-> " + x);             }
4      public void m2() {      System.out.println("Am2-> " + this.x);        }
5      public void update() {  x = 99;                                        }
6  }
7  class B extends A {
8      public void m2() {      System.out.println("Bm2-> " + x);             }
9      public void m2(int y) { System.out.println("Bm2y-> " + y);            }
10     public void m3() {      System.out.println("Bm3-> " + "called");      }
11 }
12 class C extends B {
13     public int y = x + 1;
14     public void m2() {      System.out.println("Cm2-> " + super.x);       }
15     public void m4() {      System.out.println("Cm4-> " + super.super.x); }
16     public void m5() {      System.out.println("Cm5-> " + y);             }
17 }
18 class D {
19     public static void main (String[] args) {
20         B a0 = new A();
21         a0.m1();
22         a0.m2(16);
23         A b0 = new B();
24         System.out.println(b0.x);
25         b0.m1();
26         b0.m2();
27         b0.m2(61);
28         B b1 = new B();
29         b1.m2(61);
30         b1.m3();
31         A c0 = new C();
32         c0.m2();
33         C c1 = (A) new C();
34         A a1 = (A) c0;
35         C c2 = (C) a1;
36         c2.m3();
37         c2.m4();
38         c2.m5();
39         ((C) c0).m3();
40         (C) c0.m2();
```

```
41          b0.update();
42          b0.m1();
43      }
44  }
```