Interfaces

Object-Oriented Programming

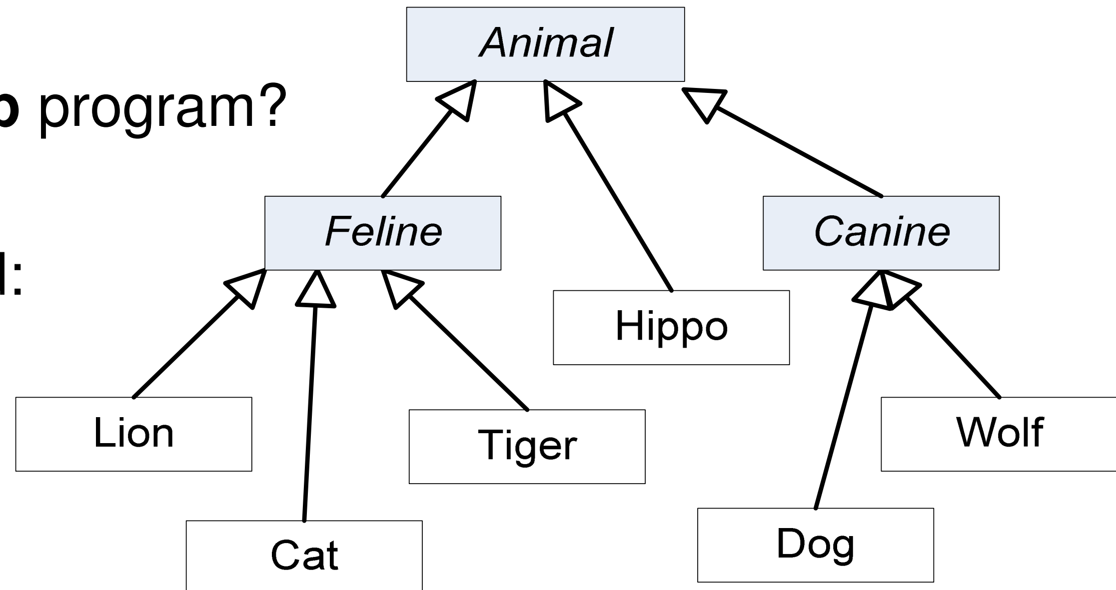
Outline

- Multi-inheritance
- The Diamond Problem
- Java interface
- Design pattern: Prototype

- Readings:
  - HFJ: Ch. 8.
  - GT: Ch. 8.

Our previous design

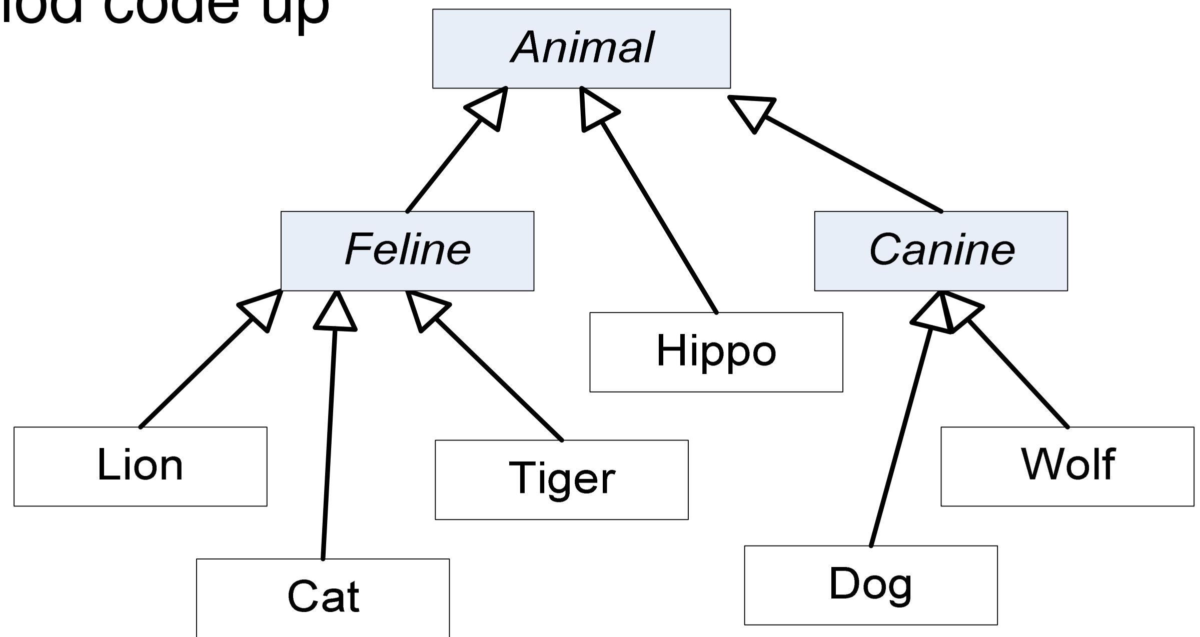
- designed for an animal simulation program
- reusable in educational software in zoology
- What about a **PetShop** program?
- Pet behaviors required:
  - beFriendly()
  - play()



- **Where should we add those behaviors to?**

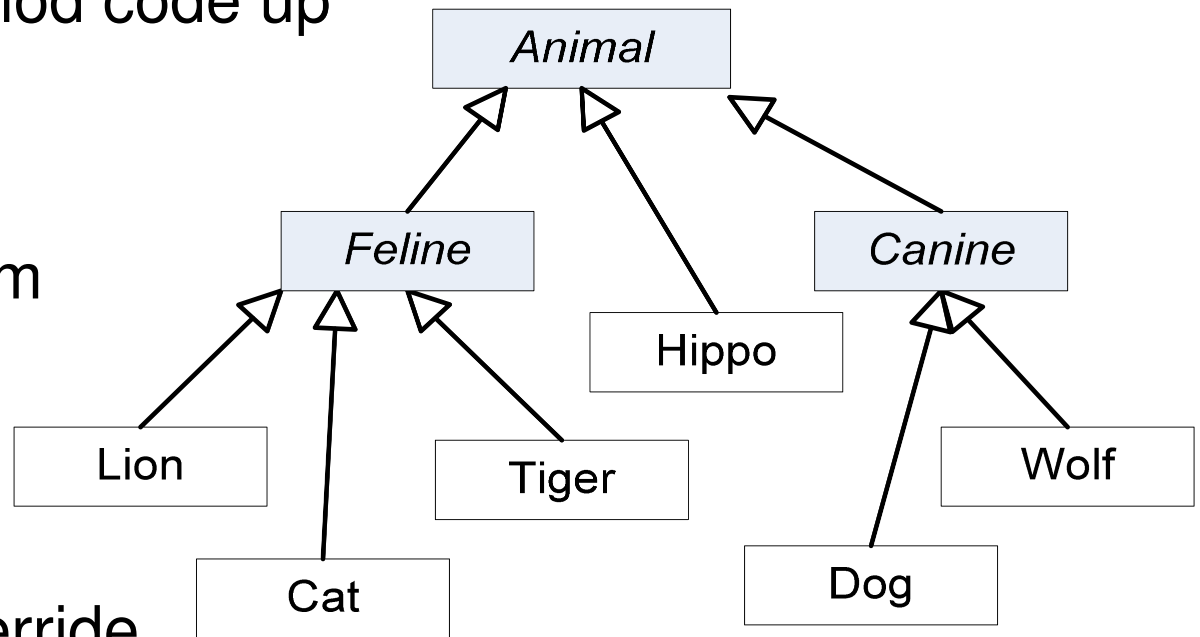
Option 1

- Put all the pet method code up
- Pros:
  - Pet polymorphism
  - Code inherited
- Cons:
  - Hippos as pets?
  - Lions and Wolves, too?
  - We still have to override pet methods in Cat and Dog



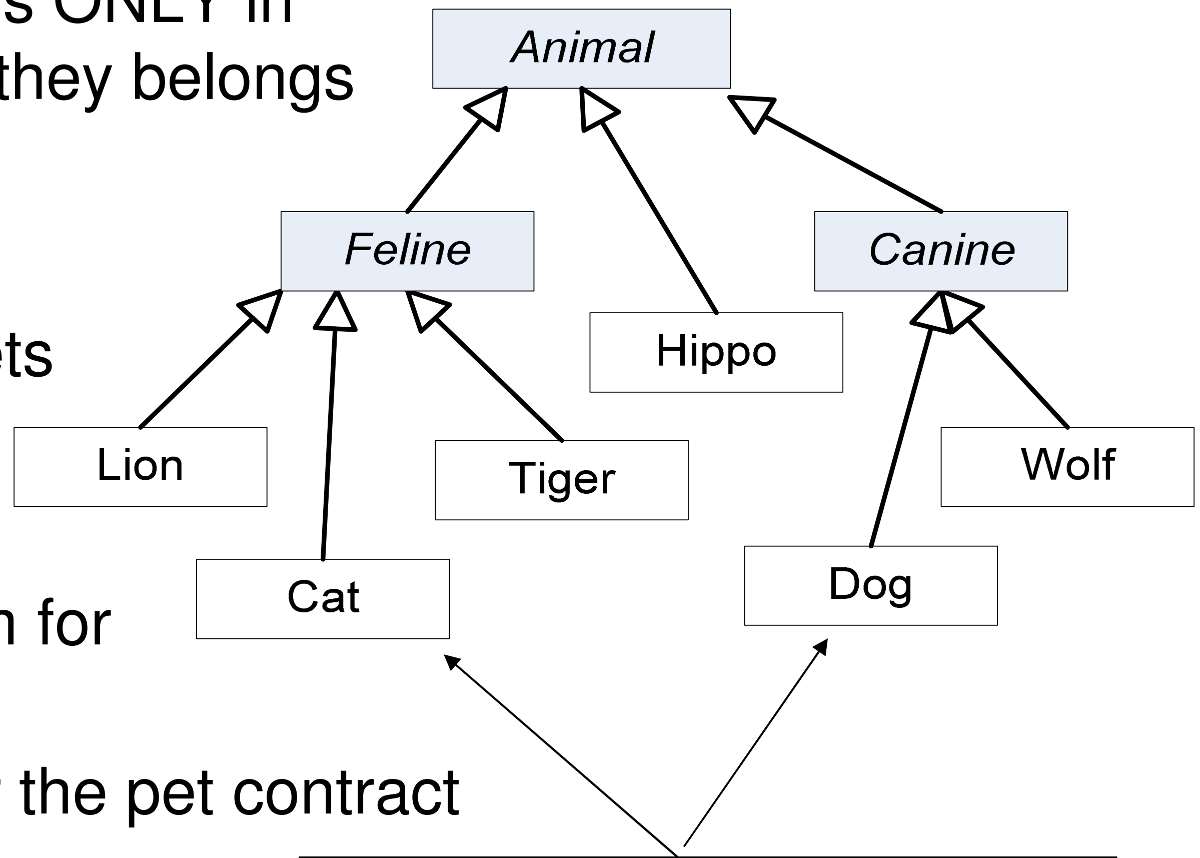
Option 2

- Put all the pet method code up
- Pros:
  - Pet polymorphism
- Cons:
  - ALL subclasses are forced to override
  - non-pet versions do nothing
- It's wrong to stuff in Animal things that not ALL Animal classes need



Option 3

- Put the pet methods ONLY in the classes where they belongs
- Pros:
  - No Hippos as pets
- Cons:
  - no polymorphism for pet methods
  - no guarantee for the pet contract

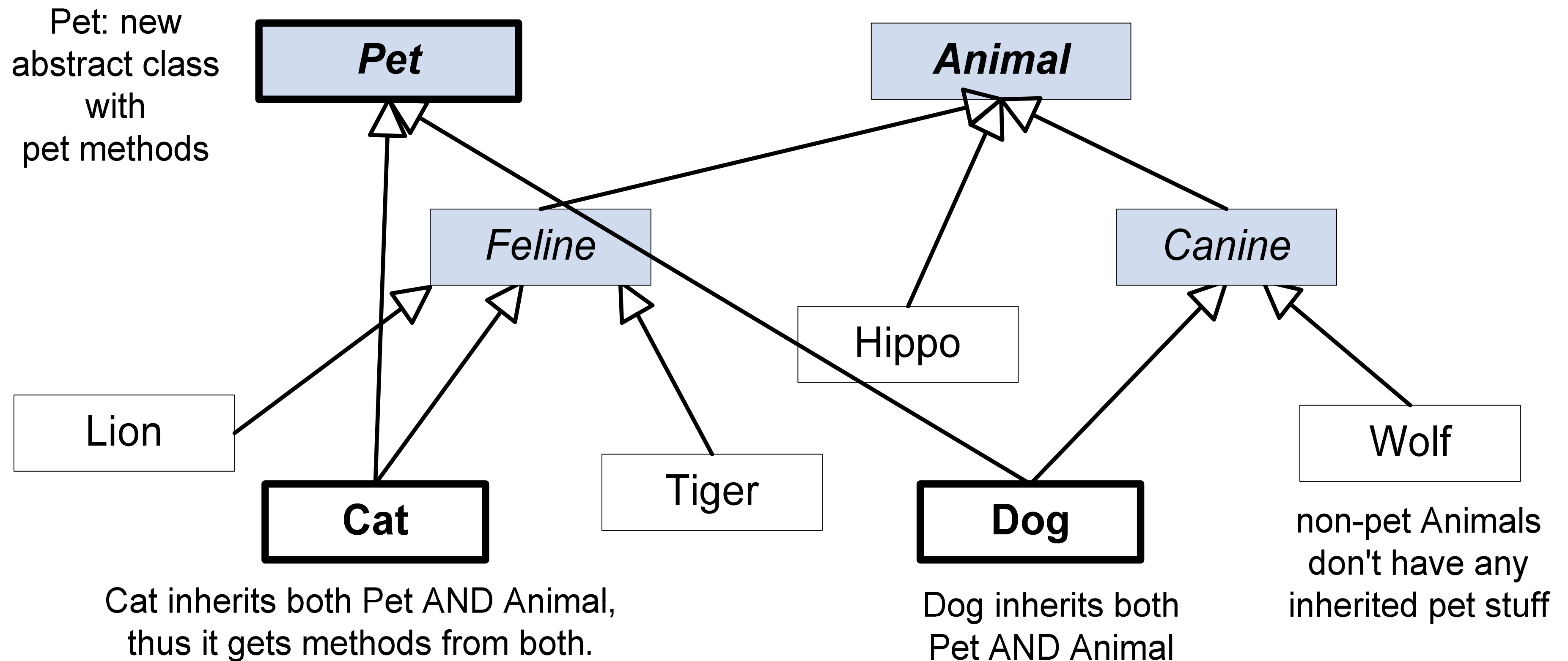


Put the pet methods ONLY in the classes that can be pets

What we really need

- A way to have pet behavior in **just** the pet classes
- A way to guarantee that all pet classes have all of the same methods defined
  - same name, same parameters, same return types, no missing methods, etc.
- A way to take advantage of polymorphism for pets
  - methods that works on all types of pets,
  - arrays contains all types of pets.
  - …

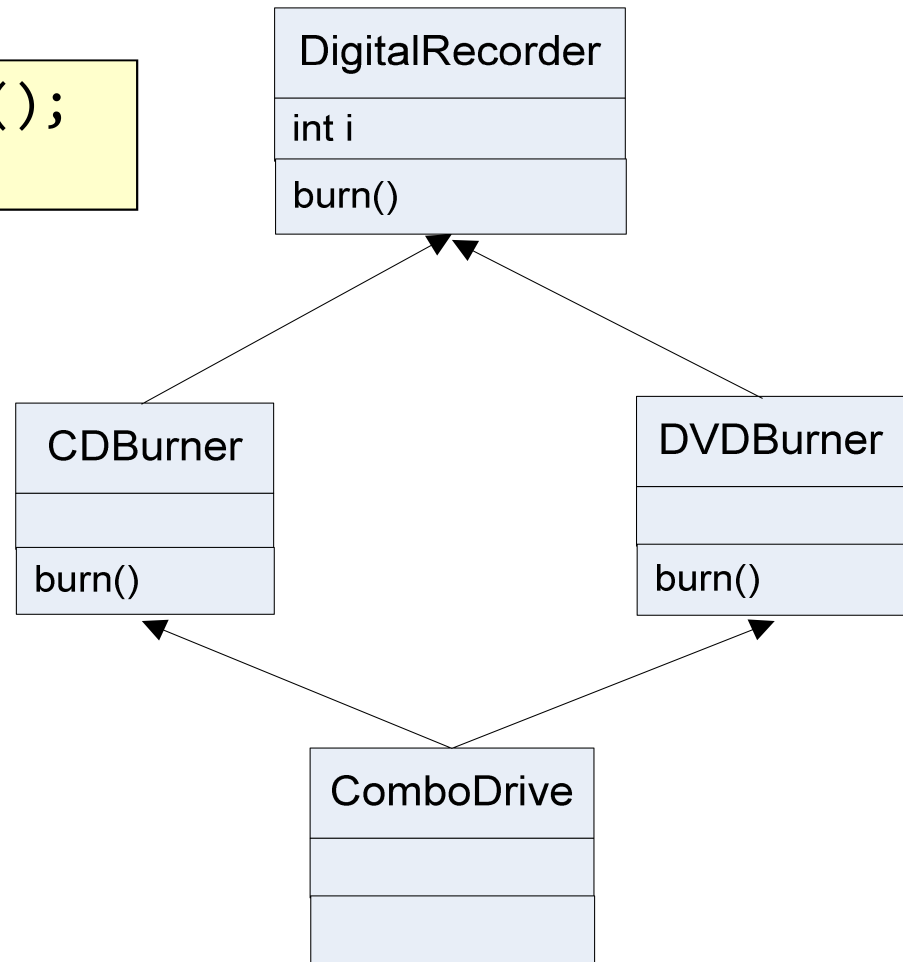
Multi-inheritance!



But… the Deadly Diamond Problem.

```
ComboDrive c = new ComboDrive();
c.burn();
```

- Which burn() gets to run?
  - CDBurner.burn() ?
  - DVDBurner.burn() ?



Java interfaces

- Java does not support multiple inheritance
  - The Deadly Diamond Problem
- Java interfaces
  - A special type of class which
    - Defines a set of method prototypes
    - Does not provide the implementation for the prototypes
    - Can also define final constants

```
public interface Pet {
  public abstract void beFriendly();
  public abstract void play();
}
```

Java interfaces - Example

- To **define** an interface:

```
public interface Pet {
  public abstract void beFriendly();
  public abstract void play();
}
```

keyword **interface** instead of **class**

the methods are ALL abstract

- To **implement** an interface:

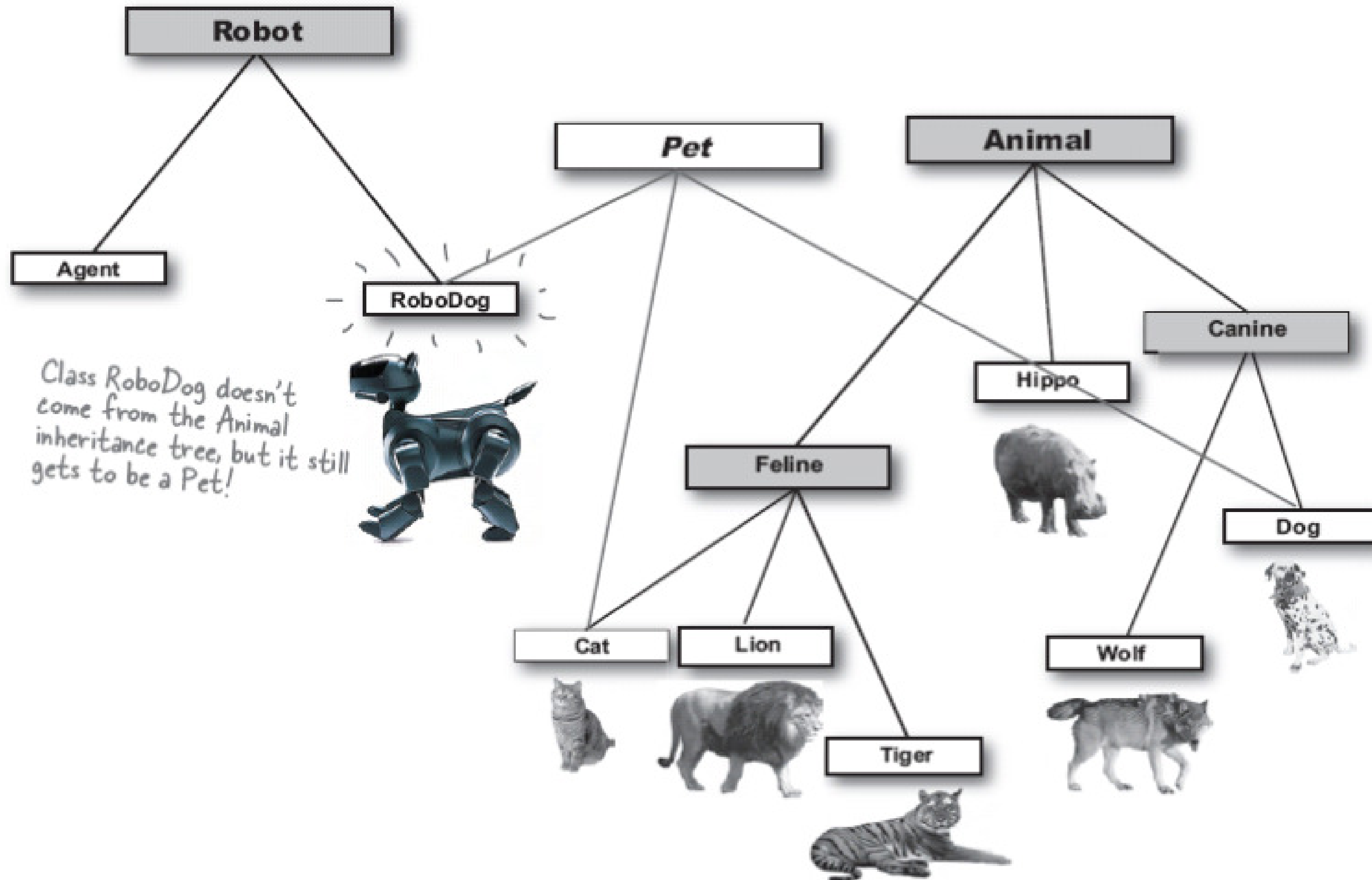
```
public class Dog extends Canine implements Pet {
  public void beFriendly() {...}
  public void play() {...}

  public void roam() {...}
  public void eat() {...}
}
```

keyword **implements**

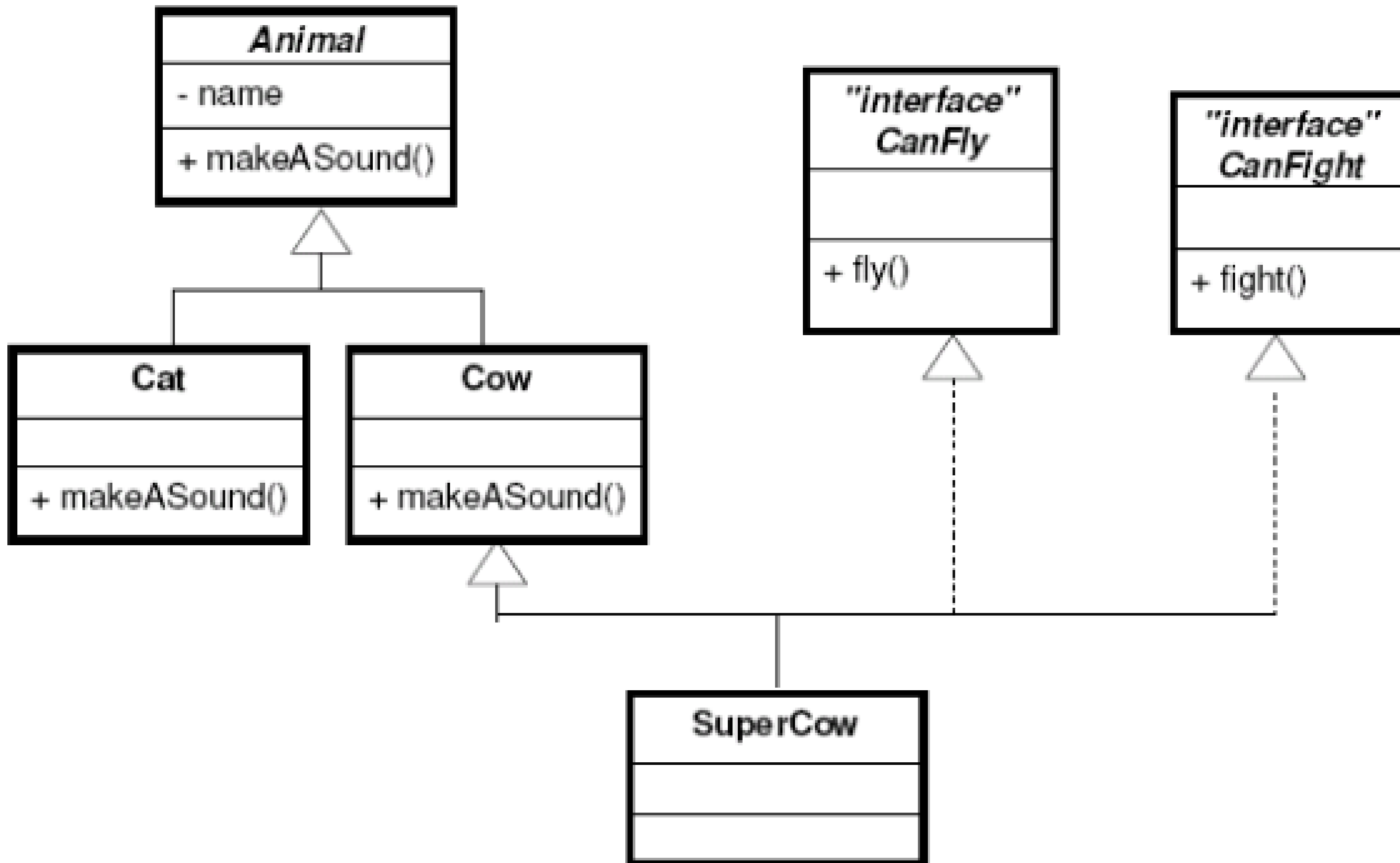
implements ALL Pet methods

normal overriding methods

Classes from different inheritance tree can implement the same interface

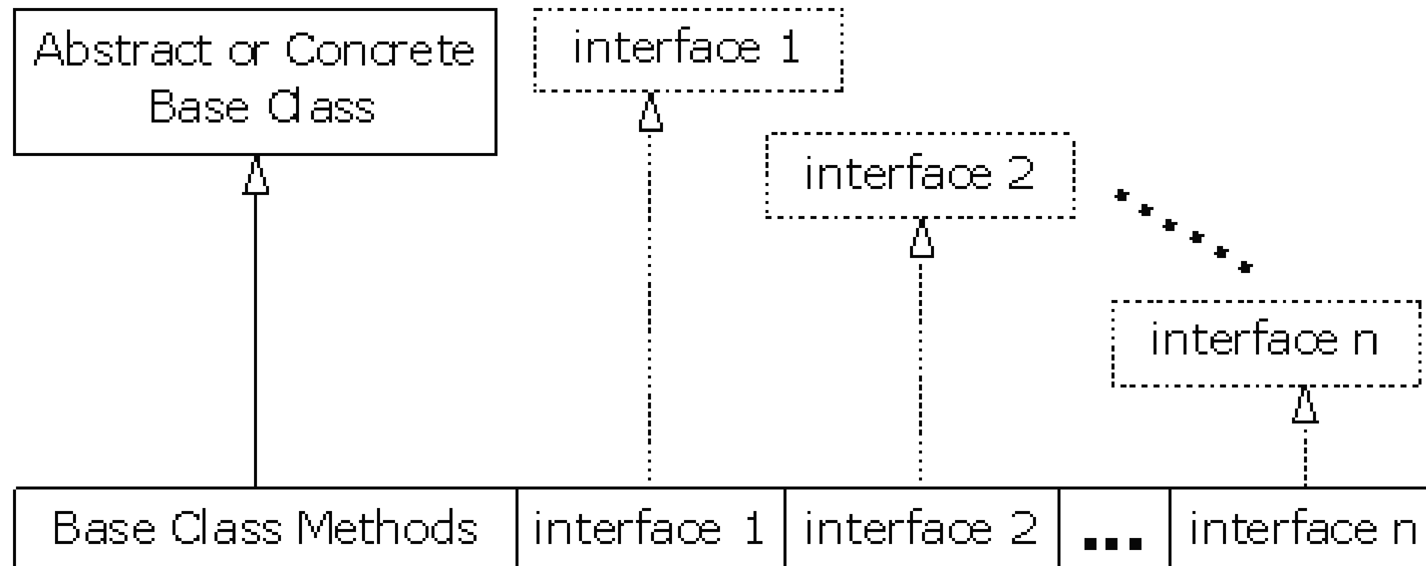


A class can implement multiple interfaces



Extends and implements

- A class
  - Can “extend” only one class, i.e. ONE superclass
  - Can “implement” MULTIPLE interfaces



Java interfaces

- Lightweight
  - Allow multiple classes to respond to a common set of messages but without the implementation complexity.
- Similar to Subclassing but…
  - Good news
    - Class has only one superclass
    - Can implement multiple interfaces
  - Bad news:
    - Interface only gives the method prototype and not the implementation

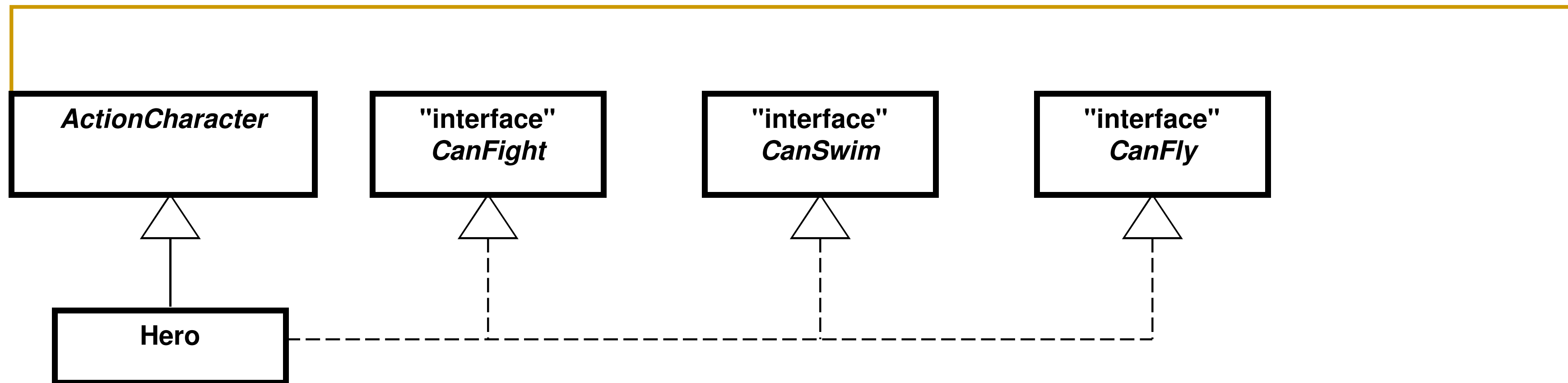
```
interface Action {
   void moveTo(int x, int y);
   void erase();
   void draw();
}
```

```
class Circle1 implements Action {
   int x, y, r;
   Circle1(int _x, int _y, int _r) { ... }

   public void erase() {...}
   public void draw() {...}
   public void moveTo(int x1, int y1) {...}
}
```

```
class ImageBuffer {
...
}

class Animation extends ImageBuffer implements Action {
...
   public void erase() {...}
   public void draw() {...}
   public void moveTo() {...}
}
```

```java
interface CanFight {
    void fight();
}
interface CanSwim {
    void swim();
}
interface CanFly {
    void fly();
}
class ActionCharacter {
    public void fight() {...}
}
```

```java
class SuperHero extends ActionCharacter
implements CanFight, CanSwim, CanFly {
    public void swim() {...}
    public void fly() {...}
}
```




```
public class Adventure {
    public static void t(CanFight x) { x.fight(); }
    public static void u(CanSwim x) { x.swim(); }
    public static void v(CanFly x) { x.fly(); }
    public static void w(ActionCharacter x) { x.fight(); }
    public static void main(String[] args) {
        SuperHero h = new SuperHero();
        t(h); // Treat it as a CanFight
        u(h); // Treat it as a CanSwim
        v(h); // Treat it as a CanFly
        w(h); // Treat it as an ActionCharacter
    }
}
```

Extending an interface with inheritance

```
interface Monster {
   void menace();
}
interface Lethal {
   void kill();
}
interface Vampire extends Monster, Lethal {
   void drinkBlood();
}
```

```
class VeryBadVampire implements Vampire {
   public void menace() {...}
   public void kill() {...}
   public void drinkBlood() {...}
}
```

Conflict (1)

```
interface I1 { void f(); }
interface I2 { int f(int i); }
interface I3 { int f(); }
class C {
    public int f() { return 1; }
}
```

```
class C2 implements I1, I2 {
    public void f() {}
    public int f(int i) { return 1; } // overloaded
}
```

```
class C3 extends C implements I2 {
    public int f(int i) { return 1; } // overloaded
}
```

Conflict (2)

```
interface I1 { void f(); }
interface I2 { int f(int i); }
interface I3 { int f(); }
class C {
    public int f() { return 1; }
}
```

```
class C4 extends C implements I3 {
    // Identical, no problem:
    public int f() { return 2; }
}
```

```
class C5 extends C implements I1 {...} //error
```

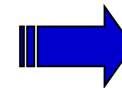
```
interface I4 extends I1, I3 {...} //error
```

Cloning objects

Copy constructors

```
class Animal {
   String name;
   public Animal( String name_) { name = name_; }
   public Animal(Animal b) { name = b.name; }
   public void sayHi() { System.out.println( "Uh oh!"); }
}
class Cat extends Animal {
   public Cat(String name_) { super(name_); }
   public Cat(Cat d) { super(d);     }
   public void sayHi() { System.out.println( "Meow..."); }
}
```

```
Cat tom = new Cat("Tom");
Cat c = new Cat(tom); c.sayHi();
Animal a = new Animal(tom); a.sayHi();
```

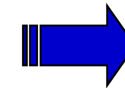


Meow…

Uh oh!

Cloning objects

```
Cat tom = new Cat("Tom");
Cat c = new Cat(tom); c.sayHi();
Animal a = new Animal(tom); a.sayHi();
```



Meow…

Uh oh!

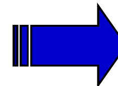
Actually, the Animal constructor is called. Not Cat constructor.

- How to clone objects without knowing their actual type?
  - copy constructor? Nope!
  - copy method?
    - write a `clone()` method

Method clone()

```
class Animal {
   String name;
   public Animal( String name_) { name = name_; }
   public Animal(Animal b) { name = b.name; }
   public Animal clone() { return new Animal(this); }
   public void sayHi() { System.out.println( "Uh oh!"); }
}
class Cat extends Animal {
   public Cat(String name_) { super(name_); }
   public Cat(Cat d) { super(d);     }
   public Cat clone() { return new Cat(this); }
   public void sayHi() { System.out.println( "Meow..."); }
}
```

```
Cat tom = new Cat("Tom");
Cat c = tom.clone(); c.sayHi();
Animal a = tom;
Animal b = a.clone(); b.sayHi();
```



Meow…

Meow…

Now we have polymorphism

Design pattern: Prototype