COMP 110/L Lecture 21

Kyle Dewey

Outline

- `this`
- `instanceof`
- Casting
- `equals()`
- `protected`
- `interface`

this

`this`

Refers to whatever instance the given instance method is called on.

this

Refers to whatever instance the given instance method is called on.

```
public class Foo {
   public Foo returnMyself() {
      return this;
   }
}
```

Example:

`ThisExample.java`

Name Clashes

`this` can be used to refer to instance variables which have the same name as normal variables

Name Clashes

`this` can be used to refer to instance variables which have the same name as normal variables

```
public class NameClash {
   private int x;
   public NameClash(int x) {
      this.x = x;
   }
}
```

Example:

`NameClash.java`

instanceof

`instanceof`

Returns a boolean indicating if a given instance was made from or inherited from a given class

instanceof

Returns a boolean indicating if a given instance was made from or inherited from a given class

```
public class InstanceOf {
  public static void main(String[] a) {
    InstanceOf i = new InstanceOf();
    if (i instanceof InstanceOf &&
        i instanceof Object) {
      // code reaches this point
    }
  }
}
```

Example:

`InstanceOfExample.java`

Casting

Casting

Converts a value of one type into another.

Not always possible to perform.

Casting

Converts a value of one type into another.
Not always possible to perform.

```
int myInt0 = 16.0;
```

Casting

Converts a value of one type into another.
Not always possible to perform.

```
int myInt0 = 16.0;
```

Does not compile

Casting

Converts a value of one type into another.
Not always possible to perform.

```
int myInt0 = 16.0;

int myInt1 = (int)16.0;
```

Casting

Converts a value of one type into another.
Not always possible to perform.

```
int myInt0 = 16.0;

int myInt1 = (int)16.0;
```

`myInt1` **holds 16**

Casting

Converts a value of one type into another.
Not always possible to perform.

```
int myInt0 = 16.0;

int myInt1 = (int)16.0;

int myInt2 = (int)16.5;
```

Casting

Converts a value of one type into another.
Not always possible to perform.

```
int myInt0 = 16.0;

int myInt1 = (int)16.0;

int myInt2 = (int)16.5;
```

myInt2 **holds 16**

Casting

Converts a value of one type into another.

Not always possible to perform.

Casting

Converts a value of one type into another.
Not always possible to perform.

```
public class Foo { ... }
...
Foo f = new Foo();
```

I define Foo and later on I make an instance of it

Casting

Converts a value of one type into another.
Not always possible to perform.

```
public class Foo { ... }
...
Foo f = new Foo();
Object o = f;
```

I can assign f to an Object, since Foo is an instance of Object

Casting

Converts a value of one type into another.
Not always possible to perform.

```
public class Foo { ... }
...
Foo f = new Foo();
Object o = f;
Foo g = o;
```

But if I try to assign an object to a Foo...

Casting

Converts a value of one type into another.
Not always possible to perform.

```
public class Foo { ... }
...
Foo f = new Foo();
Object o = f;
Foo g = o;
```

Does not compile

...this fails to compile, because any arbitrary Object is not necessarily a Foo

Casting

Converts a value of one type into another.
Not always possible to perform.

```
public class Foo { ... }
...
Foo f = new Foo();
Object o = f;
Foo g = (Foo)o;
```

-If, however, we cast the object as a Foo...

Casting

Converts a value of one type into another.
Not always possible to perform.

```
public class Foo { ... }
...
Foo f = new Foo();
Object o = f;
Foo g = (Foo)o;
```

Compiles and runs ok

-...this will work, because we have performed the cast
-The cast effectively tells Java “I know what I’m doing, and this Object is a Foo”

Casting

Converts a value of one type into another.

Not always possible to perform.

Casting

Converts a value of one type into another.
Not always possible to perform.

```
public class Foo { ... }
public class Bar { ... }
...
```

-Let's define these two classes

Casting

Converts a value of one type into another.
Not always possible to perform.

```
public class Foo { ... }
public class Bar { ... }
...
Foo f = new Foo();
Bar b = new Bar();
```

...along with these two instances

Casting

Converts a value of one type into another.
Not always possible to perform.

```
public class Foo { ... }
public class Bar { ... }
...
Foo f = new Foo();
Bar b = new Bar();
f = b;
```

Casting

Converts a value of one type into another.
Not always possible to perform.

```
public class Foo { ... }
public class Bar { ... }
...
Foo f = new Foo();
Bar b = new Bar();
f = b;
```

Does not compile

-Doesn’t compile, because a Bar is not a Foo

Casting

Converts a value of one type into another.
Not always possible to perform.

```
public class Foo { ... }
public class Bar { ... }
...
Foo f = new Foo();
Bar b = new Bar();
f = (Foo)b;
```

-If we instead try to cast it...

Casting

Converts a value of one type into another.
Not always possible to perform.

```
public class Foo { ... }
public class Bar { ... }
...
Foo f = new Foo();
Bar b = new Bar();
f = (Foo)b;
```

Compiles, but doesn't run correctly
(gives a `ClassCastException`)

-If we instead try to cast it...

equals()

`equals()`

Used to determine if two arbitrary objects are equal.

Defined in `Object`.

`equals()`

Used to determine if two arbitrary objects are equal.

Defined in `Object`.

`"foo".equals("foo")`

Returns `true`

`equals()`

Used to determine if two arbitrary objects are equal.

Defined in `Object`.

`"foo".equals("foo")`

Returns `true`

`"foo".equals("bar")`

`equals()`

Used to determine if two arbitrary objects are equal.

Defined in `Object`.

`"foo".equals("foo")`

Returns `true`

`"foo".equals("bar")`

Returns `false`

`equals()` vs. ==

- With `equals()`, we test *object equality*, AKA *deep equality*
    - Look at the inside of the object
- With ==, we test *reference equality*, AKA *shallow equality*
    - Return `true` if two references refer to the exact same object

Example:
`StringEquals.java`

-This example shows off the difference between reference and object equality

Defining Your Own `equals()`

- Usual pattern: see if the given thing is an instance of my class
    - If `true`, cast it to the class, and do some deep comparisons
        - If `false`, return `false`
- Anything is possible

Example:

`CustomEquals.java`

protected

`protected`

Somewhere between `private` and `public`.

Like `private`, but subclasses can access it.

protected

Somewhere between `private` and `public`.

Like `private`, but subclasses can access it.

```
public class HasPrivate {
   private int x;
}
```

protected

Somewhere between `private` and `public`.

Like `private`, but subclasses can access it.

```
public class HasPrivate {
   private int x;
}
public class Sub extends HasPrivate {
   ...x...
}
```

protected

Somewhere between `private` **and** `public`**.**

Like `private`**, but subclasses can access it.**

```
public class HasPrivate {
   private int x;
}
public class Sub extends HasPrivate {
   ...x...
}
```

Not permitted - `x` **is** `private` **in** `HasPrivate`

protected

Somewhere between `private` and `public`.

Like `private`, but subclasses can access it.

```
public class HasPrivate {
   private int x;
}
public class Sub extends HasPrivate {
   ...x...
}
```

```
public class HasProt {
   protected int x;
}
public class Sub extends HasProt {
   ...x...
}
```

protected

Somewhere between `private` **and** `public`.

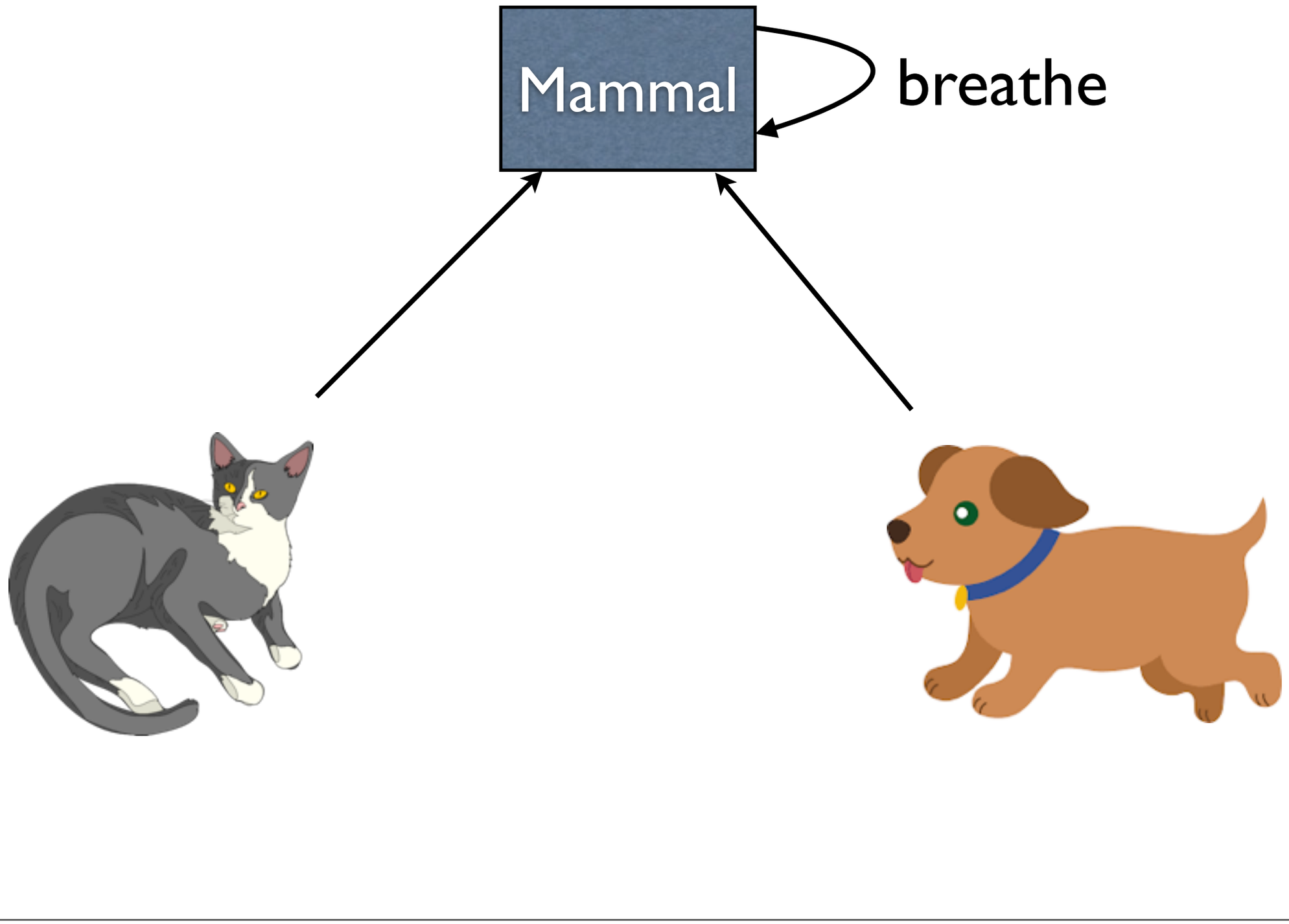
Like `private`**, but subclasses can access it.**

```
public class HasPrivate {
   private int x;
}
public class Sub extends HasPrivate {
   ...x...
}
```

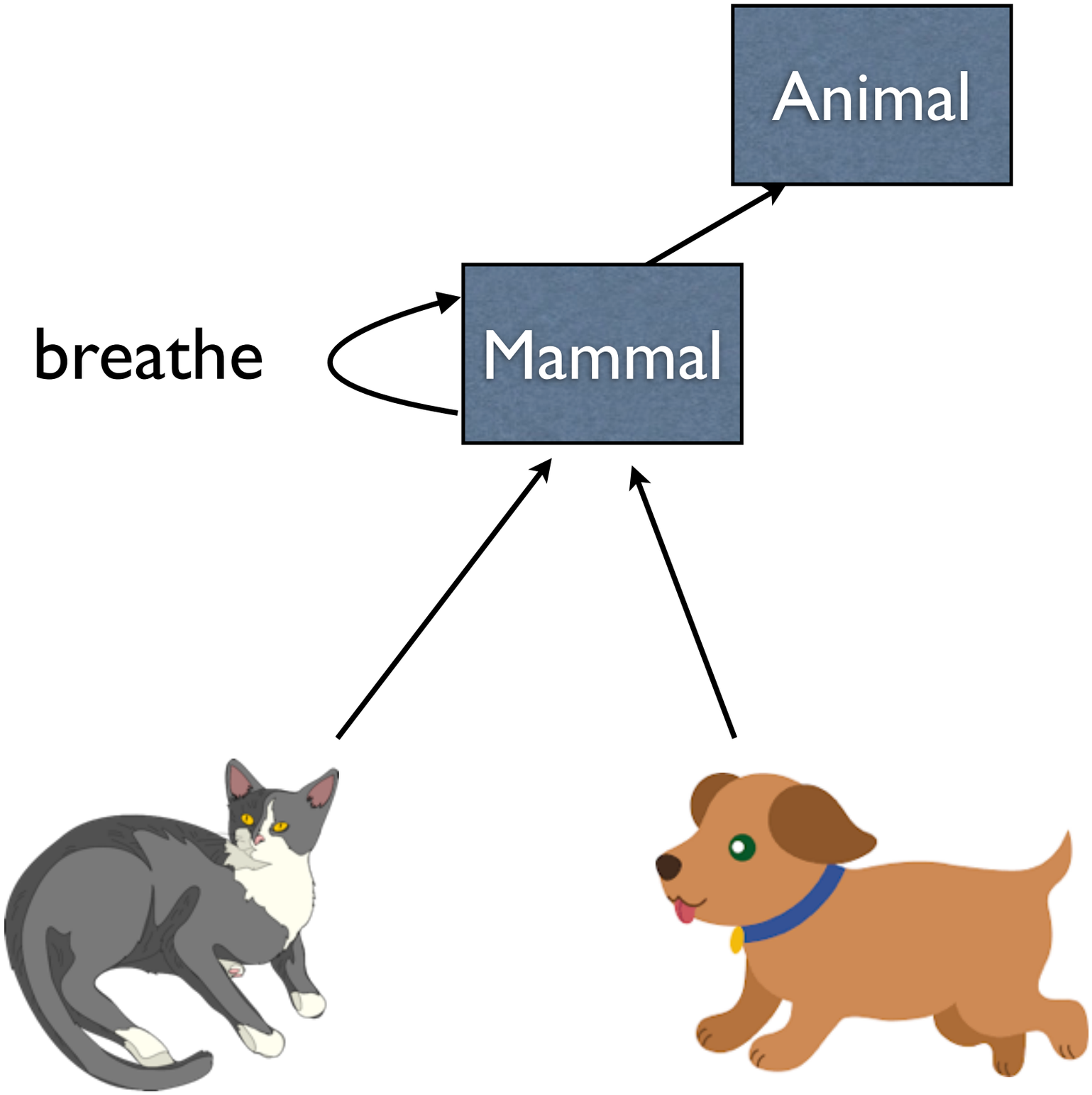
```
public class HasProt {
   protected int x;
}
public class Sub extends HasProt {
   ...x...
}
```

OK: `Sub` **is a subclass of** `HasProt`

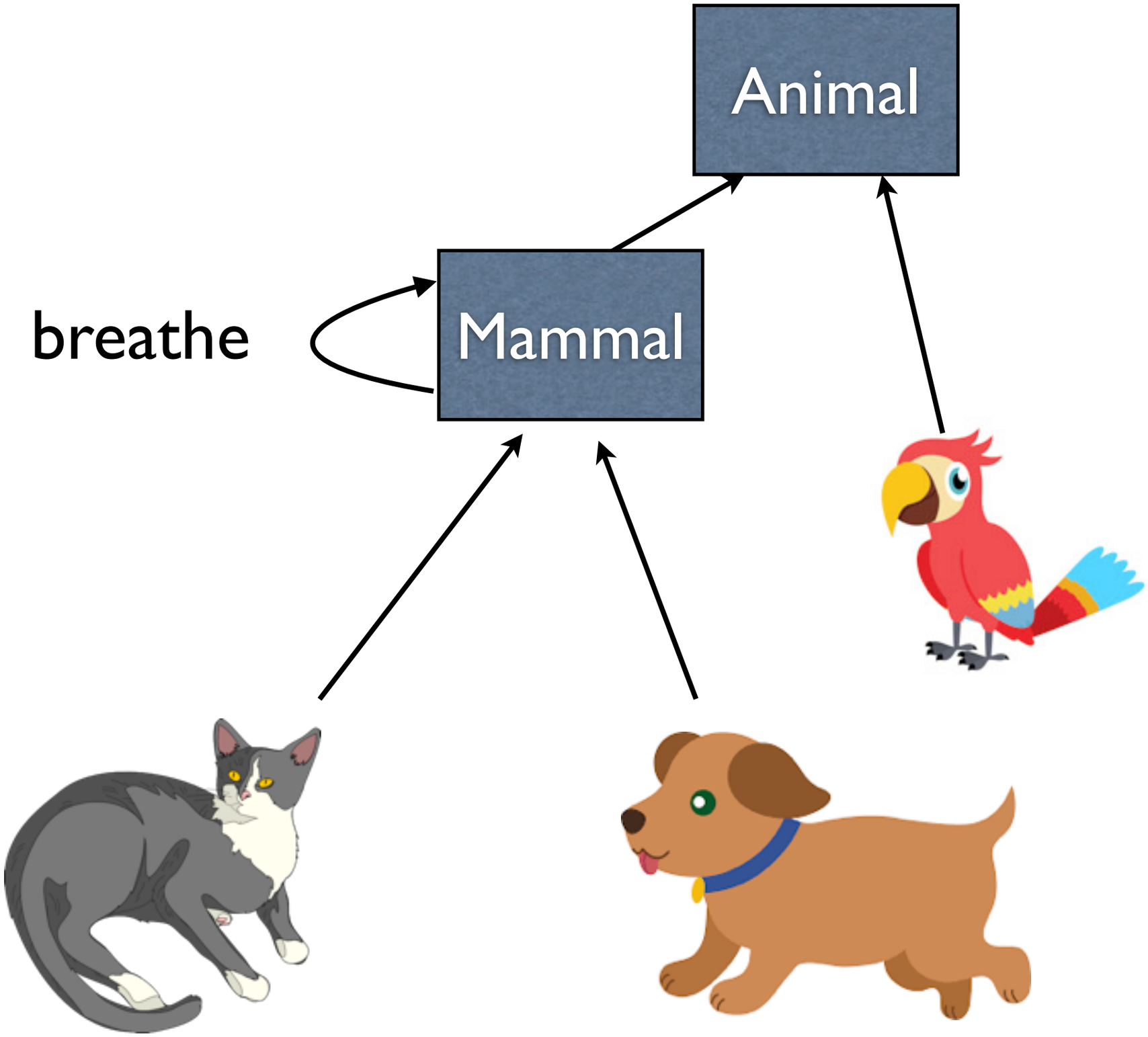
interface



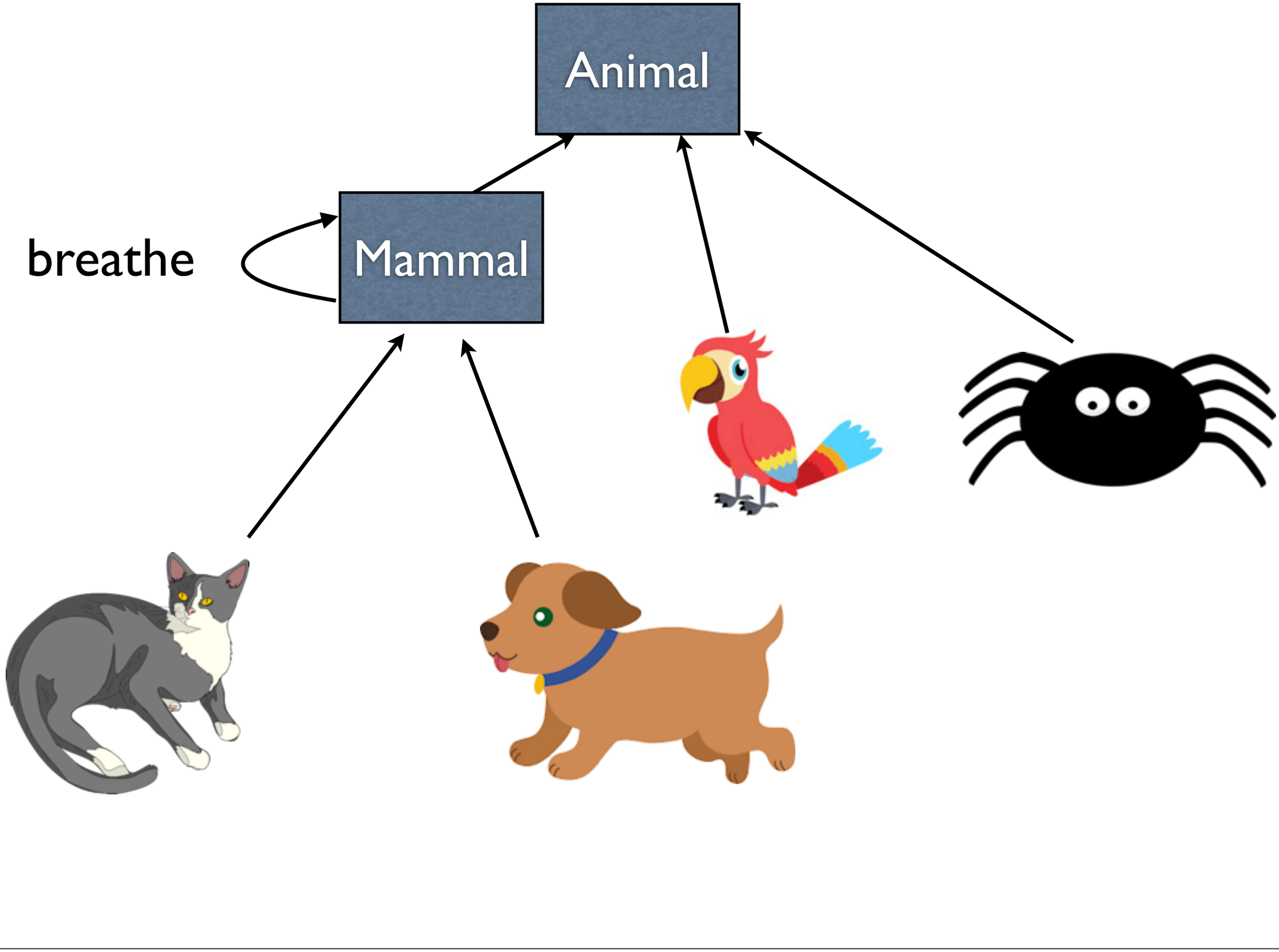
-We had this setup before



-Restructure: add Animal, birds, and spiders



-Restructure: add Animal, birds, and spiders



-Restructure: add Animal, birds, and spiders
-Now we have a problem: birds breathe too, but they aren't mammals
-The property of breathing is not unique to mammals
-Potential solution: restructure things to add BreathingAnimals in between Animal and Mammal; have Birds inherit from BreathingAnimals but not Mammal.
-That solution is specific to this case; doesn't work in general

interface

- Like an abstract class with the following restrictions:
  - Cannot have constructors
  - Cannot have instance variables
- However, we can inherit from them anywhere, and we can inherit from multiple interfaces

Using `interfaces`

```
public interface CanBreathe {
   public void breathe();
}
```

-We can define an interface like so...

Using `interfaces`

```
public interface CanBreathe {
  public void breathe();
}
```

```
public class Foo extends Bar
implements CanBreathe {
  public void breathe() { ... }
}
```

-And implement it like this
-We can extend a class along with implementing an interface

Using `interfaces`

```
public interface CanBreathe {
  public void breathe();
}
```

```
public class Foo extends Bar
  implements CanBreathe {
    public void breathe() { ... }
  }
```

```
public class Multi extends Alpha
implements Beta, Gamma, Delta { ... }
```

-Can inherit from multiple interfaces, separated with commas

Example

- `Animal.java`
- `CanBreathe.java`
- `Mammal.java`
- `Dog.java`
- `Cat.java`
- `CanFly.java`
- `Parrot.java`
- `Bat.java`
- `Spider.java`
- `AnimalMain.java`