Object Fundamentals
Part Two

Kenneth M. Anderson
University of Colorado, Boulder
CSCI 4448/5448 — Lecture 3 — 09/01/2009

Lecture Goals

- Continue our tour of the basic concepts, terminology, and notations for object-oriented analysis, design, and programming
  - Some material for this lecture is drawn from **Head First Java** by Sierra & Bates, © O'Reilly, 2003

Overview

- Objects
- Classes
  - Relationships
    - Inheritance
    - Association
    - Aggregation/Composition
    - Qualification
- Interfaces
- Ken’s Corner: Multiple Inheritance

Objects (I)

- OO Techniques view software systems as being composed of objects
- Objects have
  - **state** (aka attributes)
  - **behavior** (aka methods or services)
- We would like objects to be
  - highly cohesive
    - have a single purpose; make use of all features
  - loosely coupled
    - be dependent on only a few other classes

Objects (II)

- Objects interact by **sending messages** to one another
  - Object A sends a message to Object B to request that it perform a task
    - When the task is complete, B may pass a value back to A
    - Note: sometimes A == B
      - that is, an object can send a message to itself
- Sometimes messages can be rerouted; invoking a method defined in class A may be rerouted to an overridden version of that method in subclass B
  - And, invoking a method on an object of class B may invoke an inherited version of that method defined by superclass A

Objects (III)

- In response to a message, an object may
  - update its internal state
  - retrieve a value from its internal state
  - create a new object (or set of objects)
  - **delegate** part or all of the task to some other object
- As a result, objects can be viewed as members of various **object networks**
  - Objects in an object network (aka **collaboration**) work together to perform a task for their host application

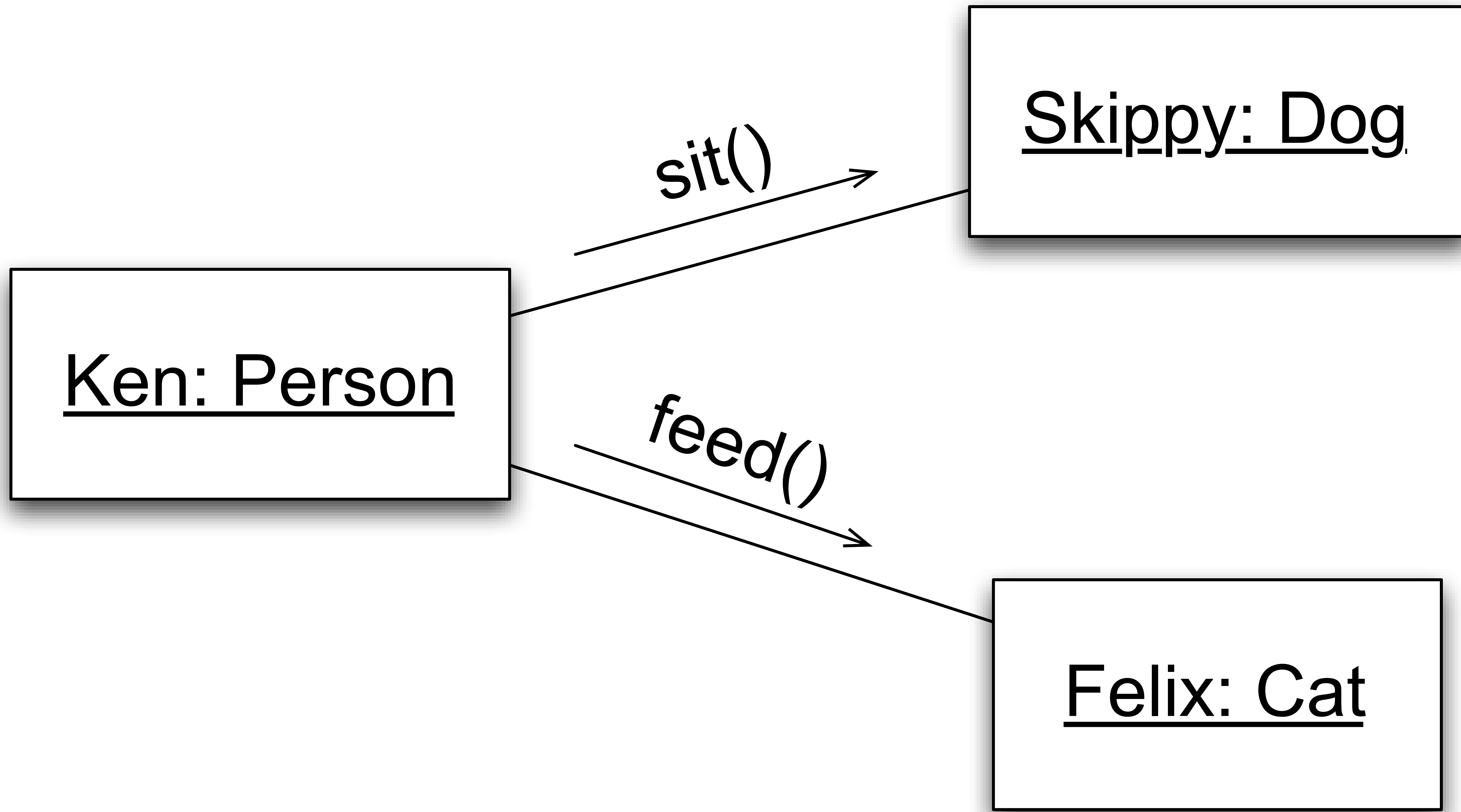
Objects (IV)

- UML notation
  - Objects are drawn as rectangles with their names and types underlined
    - <u>Ken : Person</u>
  - The name of an object is optional. The type, however, is required
    - <u>: Person</u>
  - Note: the colon is not optional. It’s another clue that you are talking about an object, not a class

Objects (V)

- Objects that know about each other have lines drawn between them
  - This connection has many names, the three most common are
    - object reference
    - reference
    - **link**
  - Messages are sent across links
    - Links are **instances** of **associations** (defined on slide 16)

Objects (Example)



Classes (I)

- A class is a blueprint for an object
  - The blueprint specifies the **attributes** (aka **instance variables**) and **methods** of the class
    - attributes are things an object of that class **knows**
    - methods are things an object of that class **does**
  - An object is **instantiated** (created) from the description provided by its class
    - Thus, objects are often called **instances**

Classes (II)

- An object of a class has its **own values** for the attributes of its class
  - For instance, two objects of the `Person` class can have different values for the `name` attribute
- In general, each object **shares the implementation** of a class's methods and thus **behave similarly**
  - When a class is defined, its developer provides an implementation for each of its methods
  - Thus, object `A` and `B` of type `Person` each share the same implementation of the `sleep()` method

Classes (III)

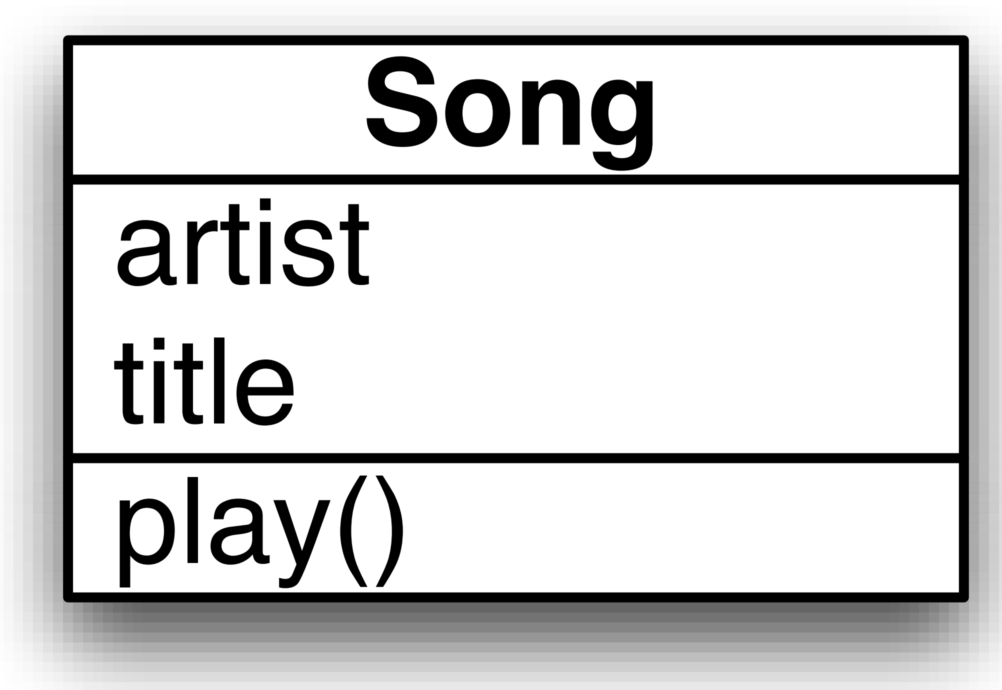
- Classes can define “class wide” (aka **static**) attributes and methods
  - A static attribute is shared among a class’s objects
    - That is, all objects of that class can read/write the static attribute
  - A static method does not have to be accessed via an object; you invoke static methods directly on a class
    - Static methods are often used to implement the notion of “library” in OO languages; it doesn’t make sense to have multiple instances of a Math class, each with their own sin() method
- We will see uses for static attributes and methods throughout the semester

Classes by Analogy

- Address Book
    - Each card in an address book is an “instance” or “object” of the `AddressBookCard` class
        - Each card has the same blank fields (attributes)
        - You can do similar things to each card
            - each card has the same set of methods
    - The number of cards in the book is an example of a static attribute;
    - Sorting the cards alphabetically is an example of a static method

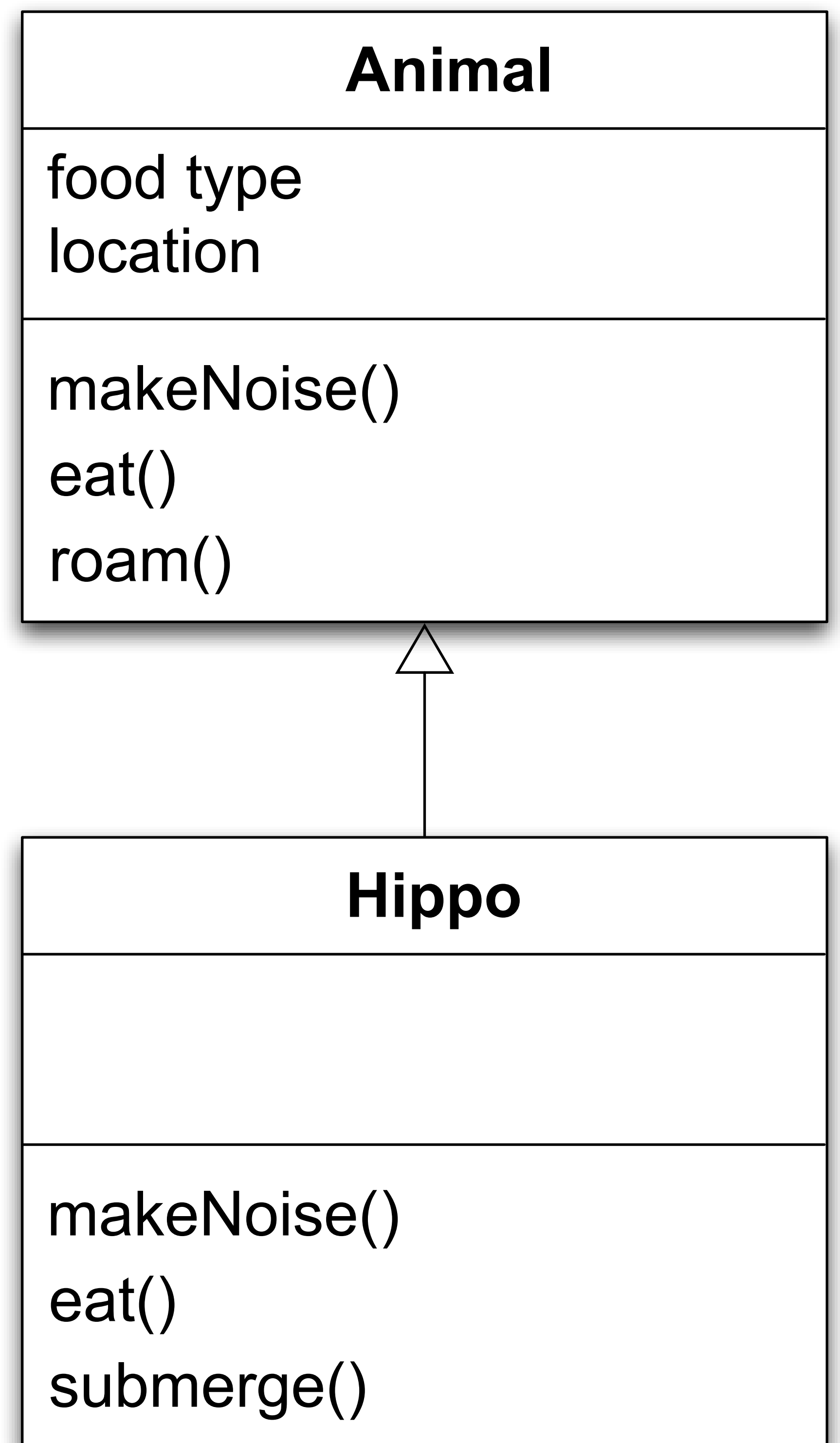
Classes (IV)

- UML Notation
  - Classes appear as rectangles with multiple parts
    - The first part contains its name (defines a type)
    - The second part contains the class's attributes
    - The third part contains the class's methods



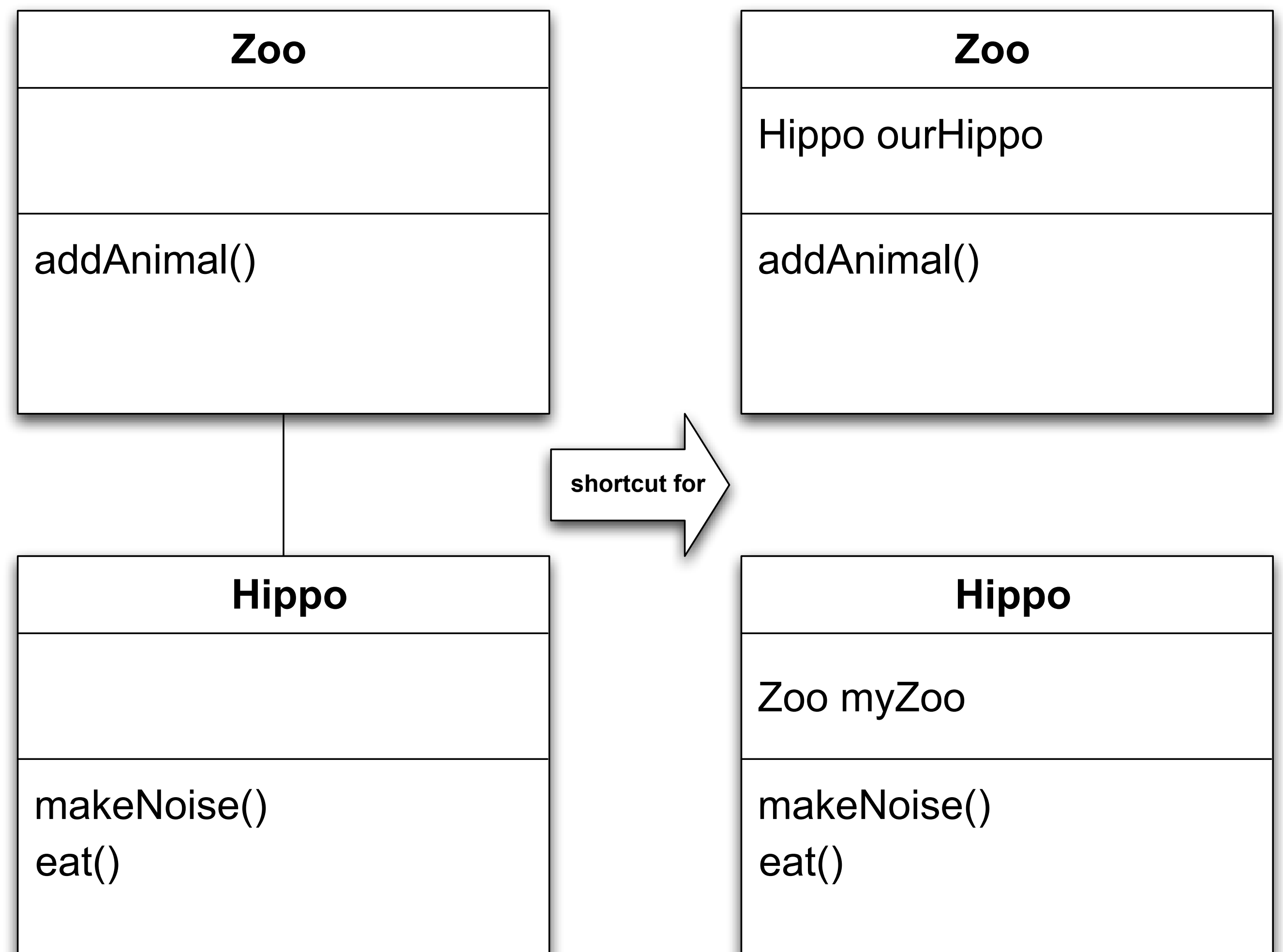
Relationships: Inheritance

- Classes can be related in various ways
    - One class can **extend** another (aka **inheritance**)
        - notation: an open triangle points to the superclass
    - As we learned last time, the **subclass** can add behaviors or **override** existing ones



Relationships: Association

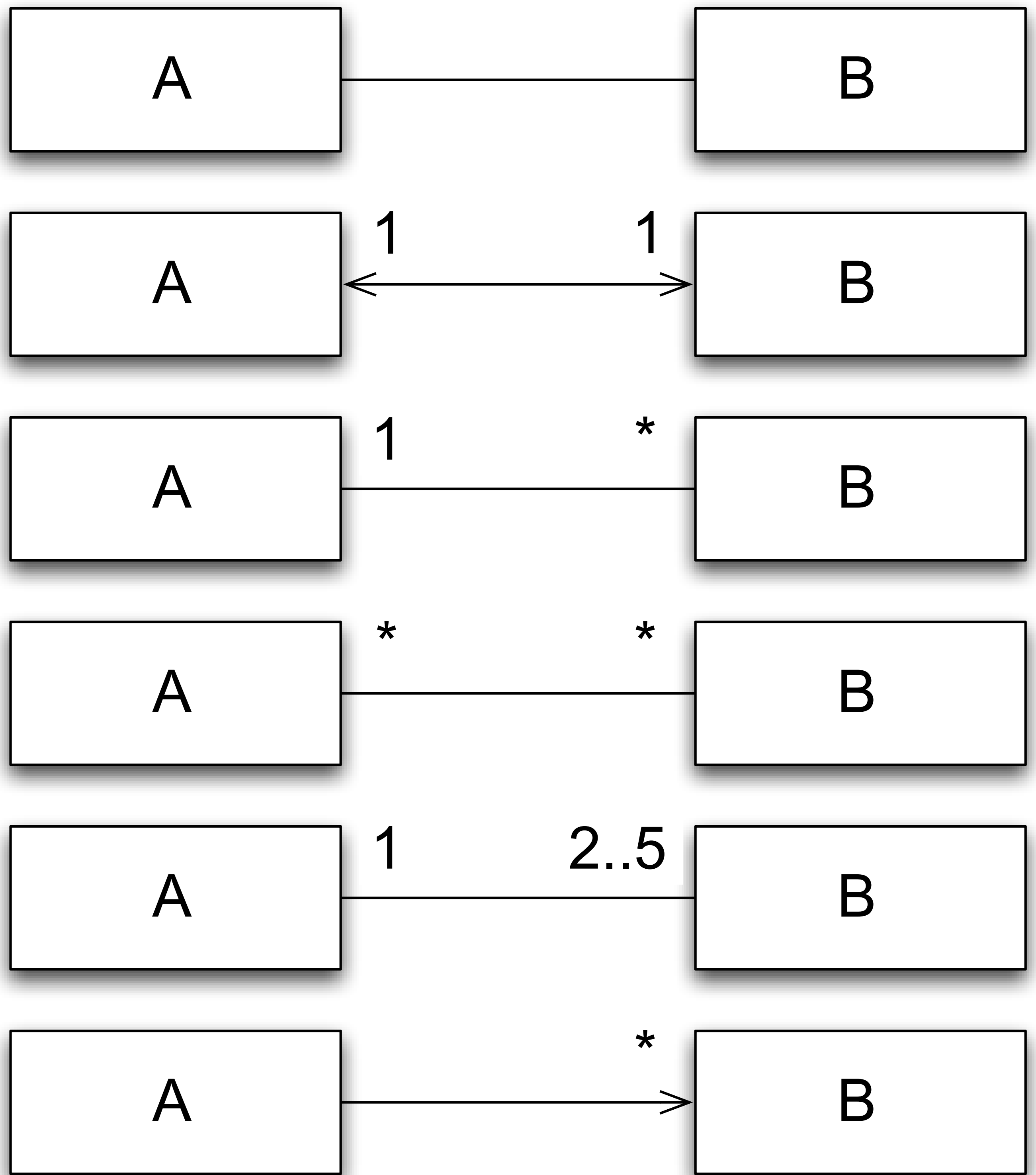
- One class can reference another (aka **association**)
    - notation: straight line
- This notation is a **graphical shorthand** that each class contains an attribute **whose type is the other class**



Multiplicity

- Associations can indicate the number of instances involved in the relationship
    - this is known as **multiplicity**
- An association with no markings is “one to one”
- An association can also indicate **directionality**

Multiplicity Examples



One B with each A; one A with each B

Same as above

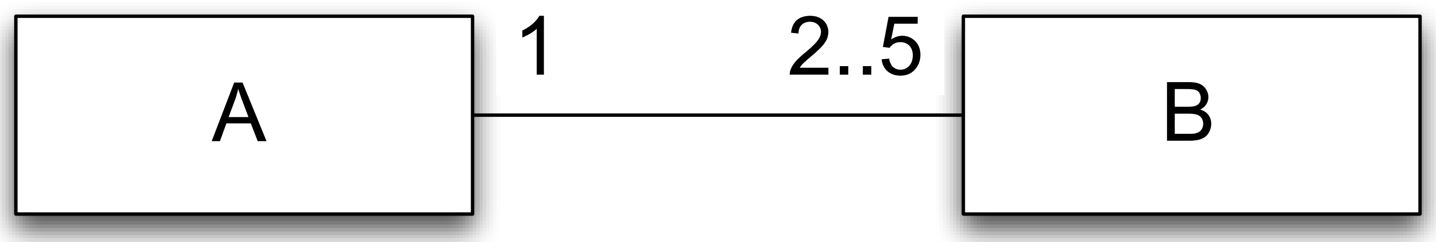
Zero or more Bs with each A; one A with each B

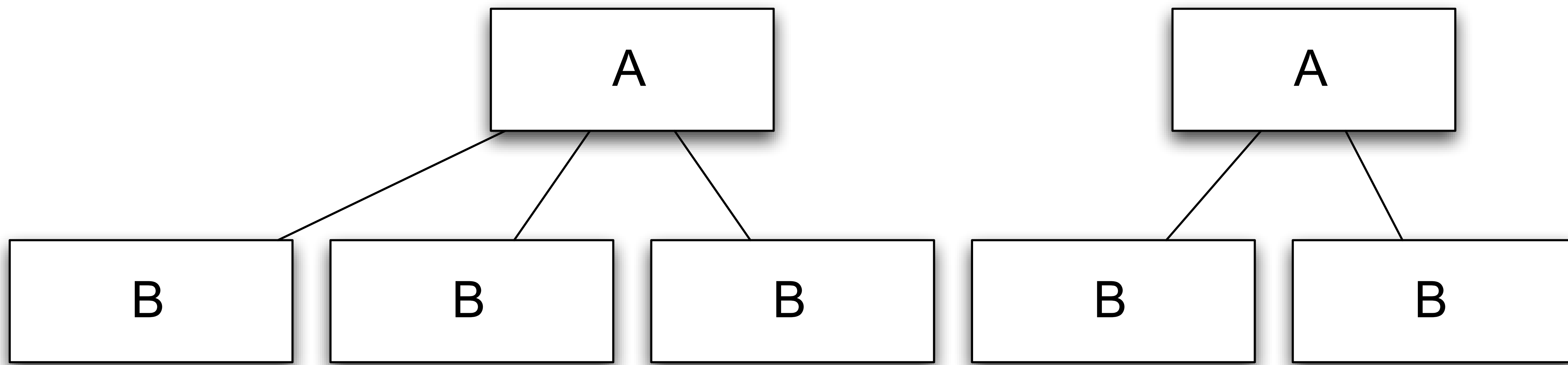
Zero or more Bs with each A; ditto As with each B

Two to Five Bs with each A; one A with each B

Zero or more Bs with each A; B knows nothing about A

Multiplicity Example

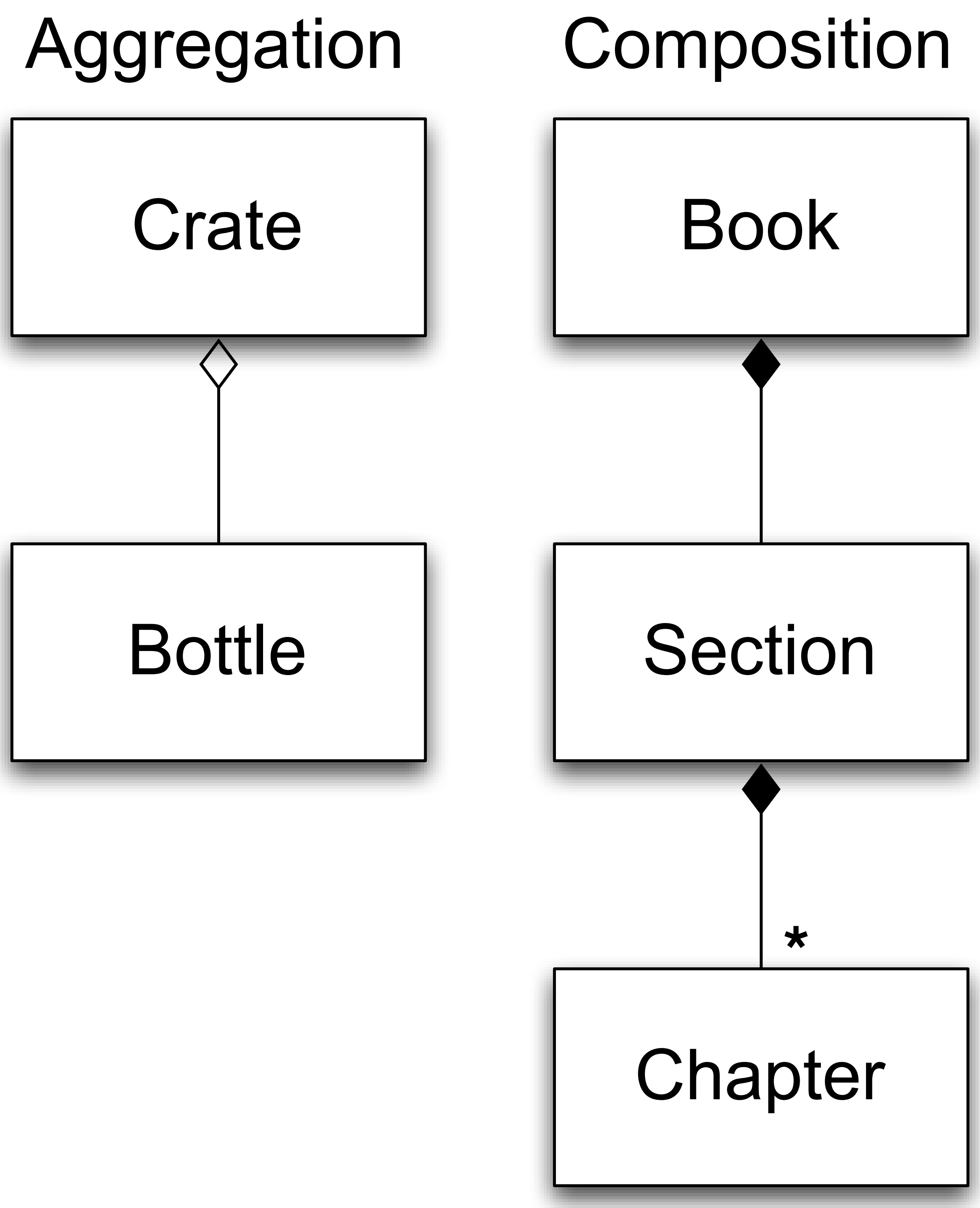




Relationships: whole-part

- Associations can also convey **semantic information** about themselves
  - In particular, **aggregations** indicate that one object contains a set of other objects
    - think of it as a **whole-part relationship** between
      - a class representing a **group** of components
      - a class representing the **components**
  - Notation: aggregation is indicated with a **white diamond** attached to the class playing the **container** role

Example: Aggregation



Composition will be defined on the next slide

Note: aggregation and composition relationships change the default multiplicity of associations;

instead of “one to one”, you should assume “one to many”

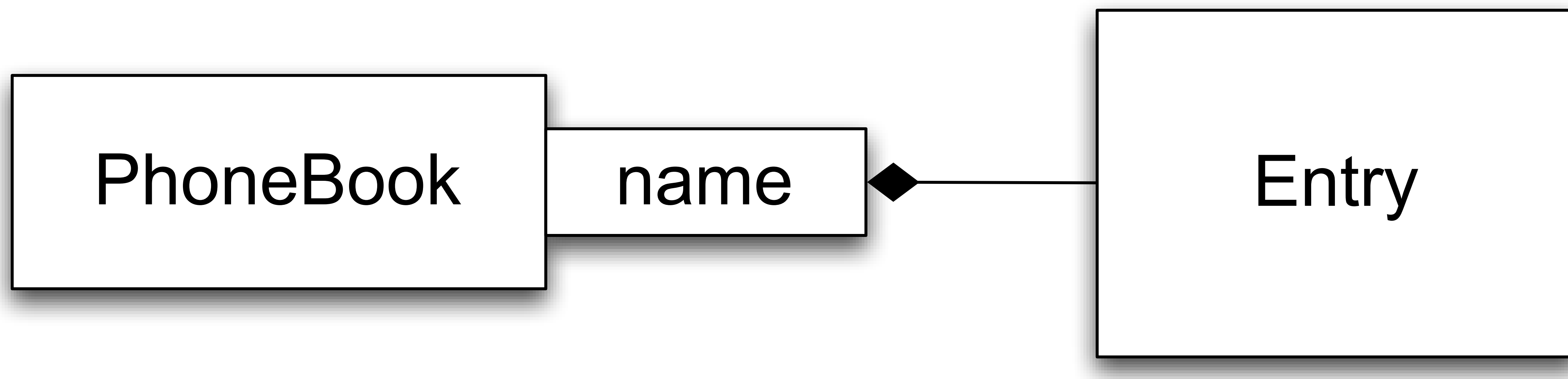
Semantics of Aggregation

- Aggregation relationships are **transitive**
  - if A contains B and B contains C, then A contains C
- Aggregation relationships are **asymmetric**
  - If A contains B, then B does not contain A
- A variant of aggregation is **composition** which adds the property of **existence dependency**
  - if A composes B, then if A is deleted, B is deleted
- Composition relationships are shown with a **black diamond** attached to the composing class

Relationships: Qualification

- An association can be **qualified** with information that indicates **how objects on the other end of the association are found**
  - This allows a designer to indicate that the association requires a query mechanism of some sort
  - e.g., an association between a phonebook and its entries might be qualified with a name, indicating that the name is required to locate a particular entry
  - Notation: a qualification is indicated with a rectangle attached to the end of an association indicating the attributes used in the query

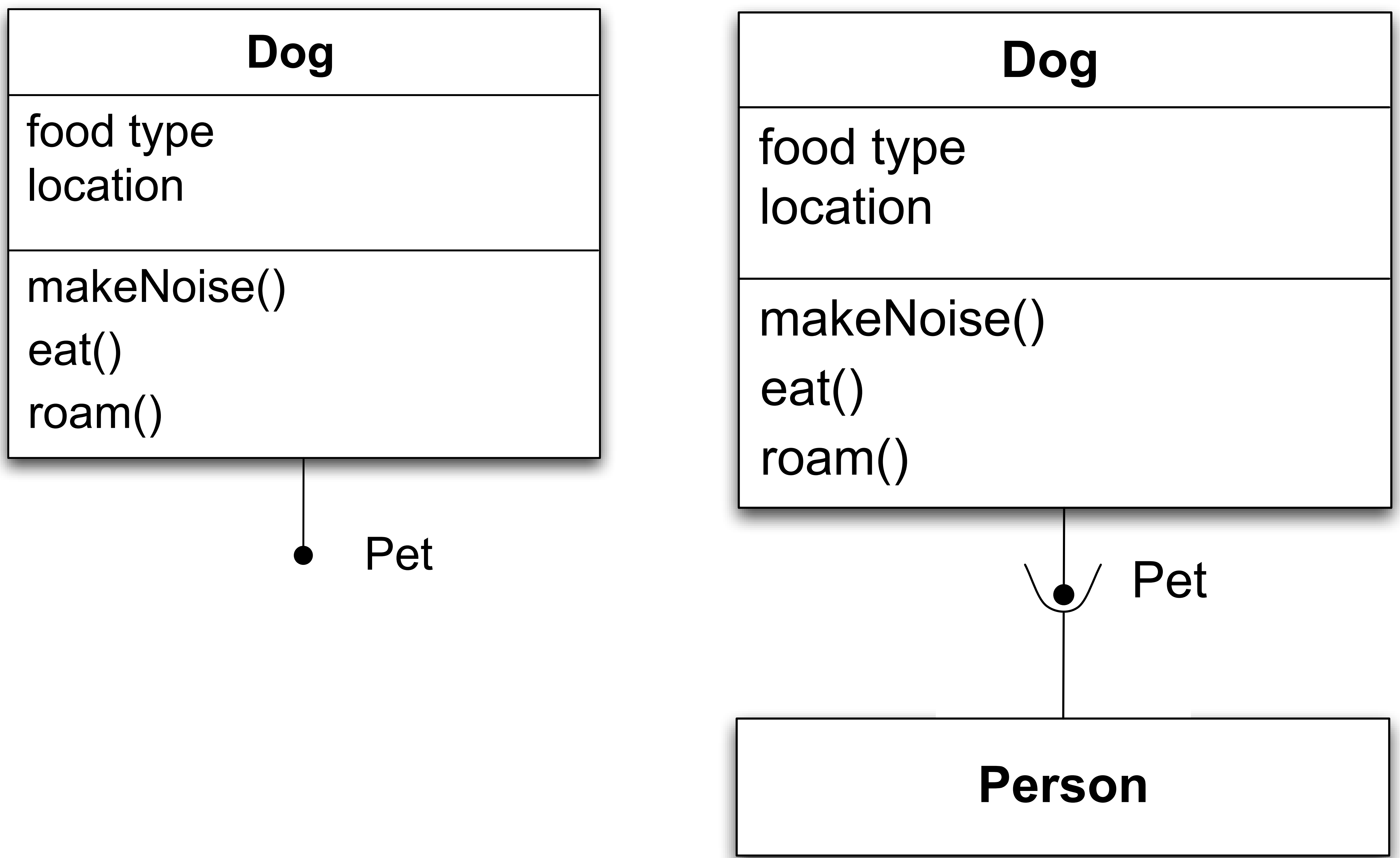
Qualification Example



Relationships: Interfaces

- A class can indicate that it implements an interface
  - An interface is a type of class definition in which only **method signatures** are defined
- A class **implementing** an interface provides method bodies for each defined method signature in that interface
  - This allows a class to play different roles, each role providing a different set of services
    - These roles are then independent of the class’s inheritance relationships
- Other classes can then access a class via its interface
  - This is indicated via a “ball and socket” notation

Example: Interfaces



Class Summary

- Classes are blue prints used to create objects
- Classes can participate in multiple relationship types
  - inheritance
  - association
    - associations have multiplicity
  - aggregation/composition
  - qualification
- Interfaces

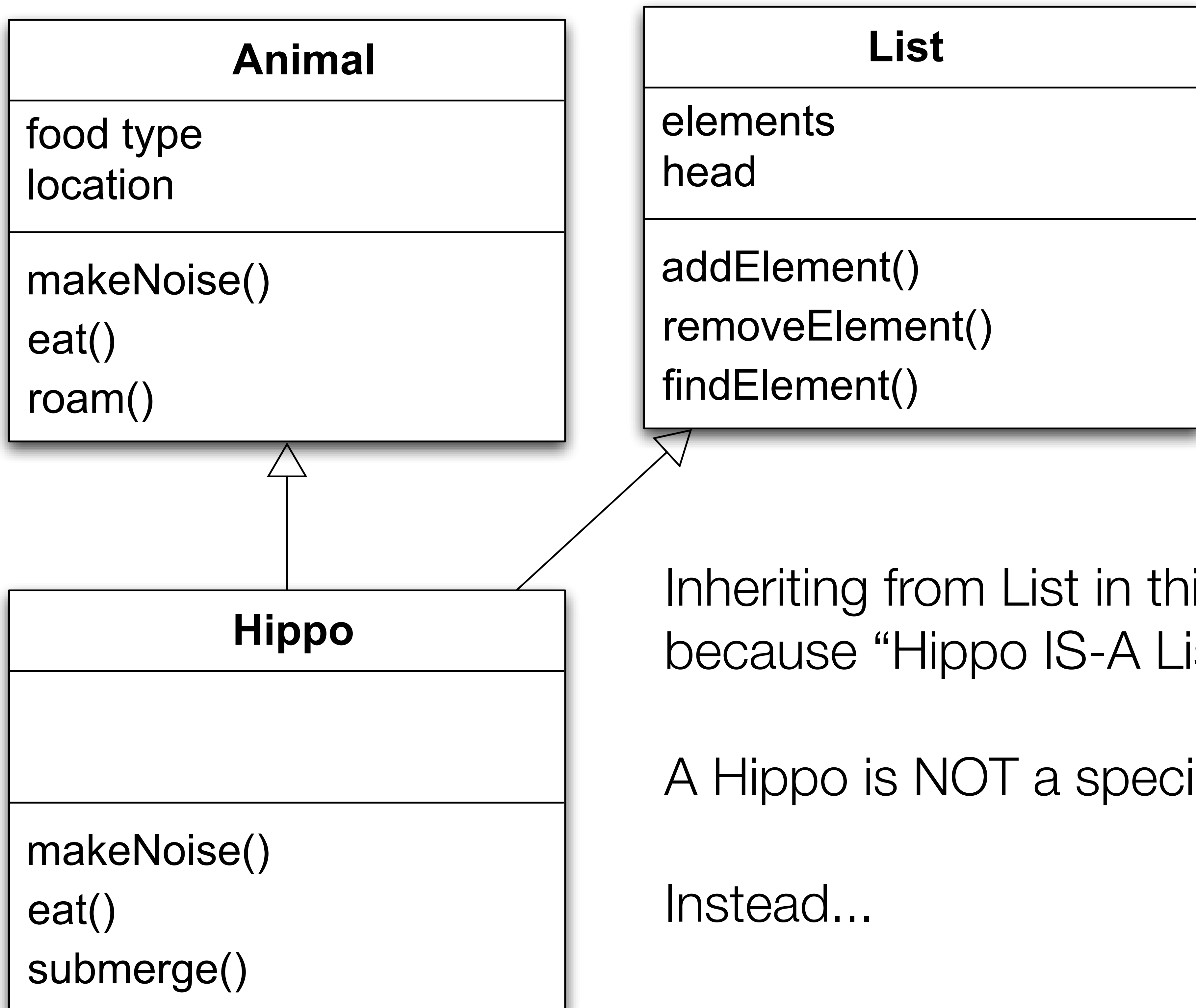
Ken’s Corner

- Multiple Inheritance
  - Some material for this section taken from
    - Object-Oriented Design Heuristics by Arthur J. Riel
      -
      - ISBN: 0-201-63385-X

Multiple Inheritance

- Riel does not advocate the use of multiple inheritance (its too easy to misuse it). As such, his first heuristic is
  - *If you have an example of multiple inheritance in your design, assume you have made a mistake and prove otherwise!*
- Most common mistake
  - Using multiple inheritance in place of containment
    - That is, you need the services of a List to complete a task
      - Rather than creating an instance of a List internally, you instead use multiple inheritance to inherit from your semantic superclass as well as from List to gain direct access to List's methods
        - You can then invoke List's methods directly and complete the task

Graphically

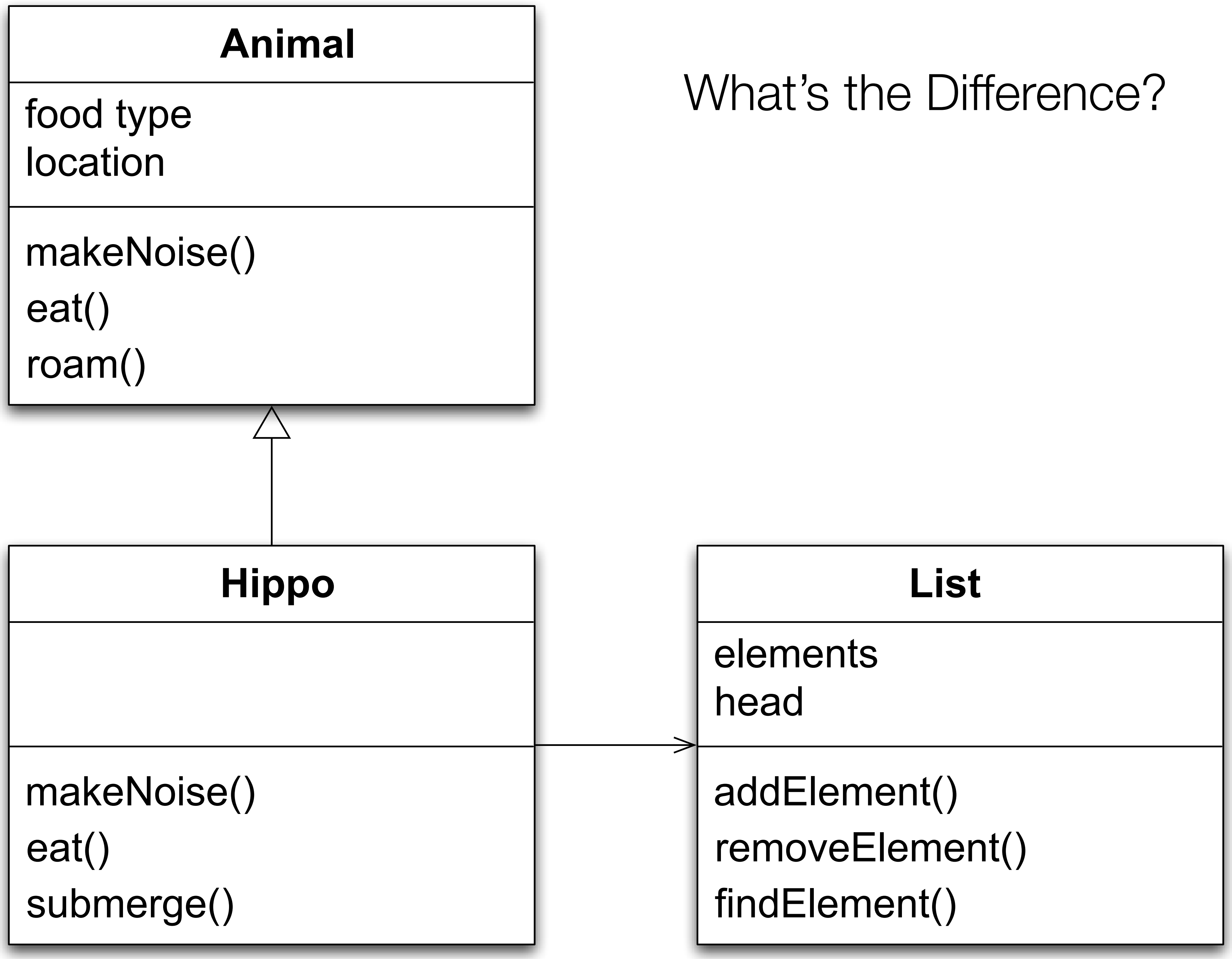


Inheriting from List in this way is bad, because “Hippo IS-A List” is FALSE

A Hippo is NOT a special type of List

Instead...

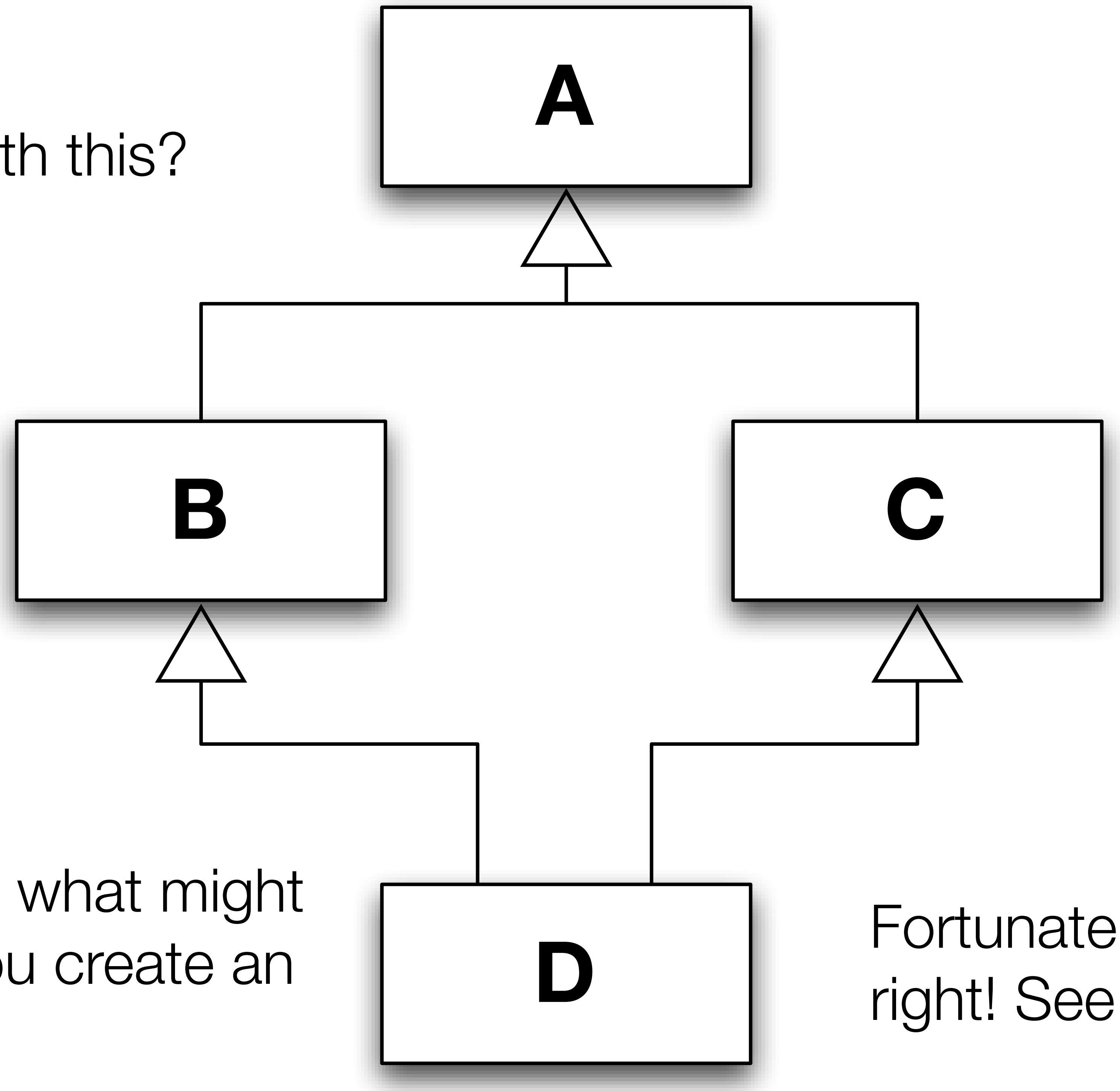
Do This



What’s the Difference?

Another Problem

What's wrong with this?



Hint: think about what might happen when you create an instance of D

Fortunately: Python gets it right! See example code.

Multiple Inheritance

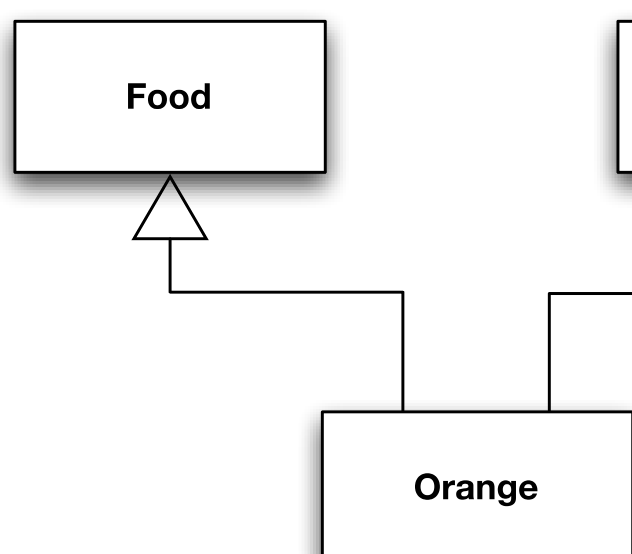
- A Second Heuristic
  - *Whenever there is inheritance in an OO design, ask two questions:*

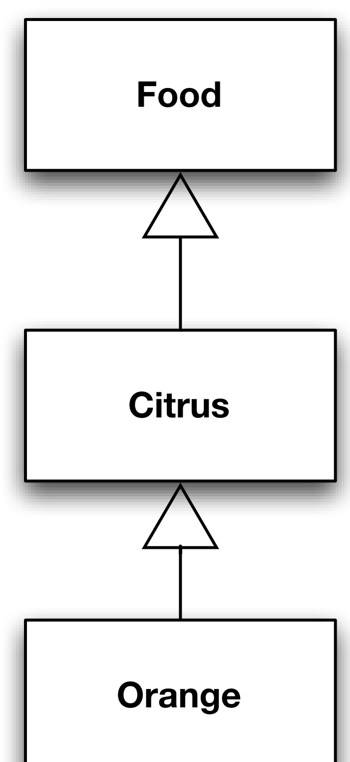
    1) Am I a special type of the thing from which I'm inheriting?

    2) Is the thing from which I'm inheriting part of me?
- A yes to 1) and no to 2) implies the need for inheritance
- A no to 1) and a yes to 2) implies the need for composition
  - Recall Hippo/List example
- Example
  - Is an airplane a special type of fuselage? No
  - Is a fuselage part of an airplane? Yes

Multiple Inheritance

- A third heuristic
  - *Whenever you have found a multiple inheritance relationship in an object-oriented design, be sure that no base class is actually a derived class of another base class*
- Otherwise you have what Riel calls **accidental multiple inheritance**
  - Consider the classes “Citrus”, “Food”, and “Orange”; you can have Orange multiply inherit from both Citrus and Food…but Citrus IS-A Food, and so the proper hierarchy can be achieved with single inheritance

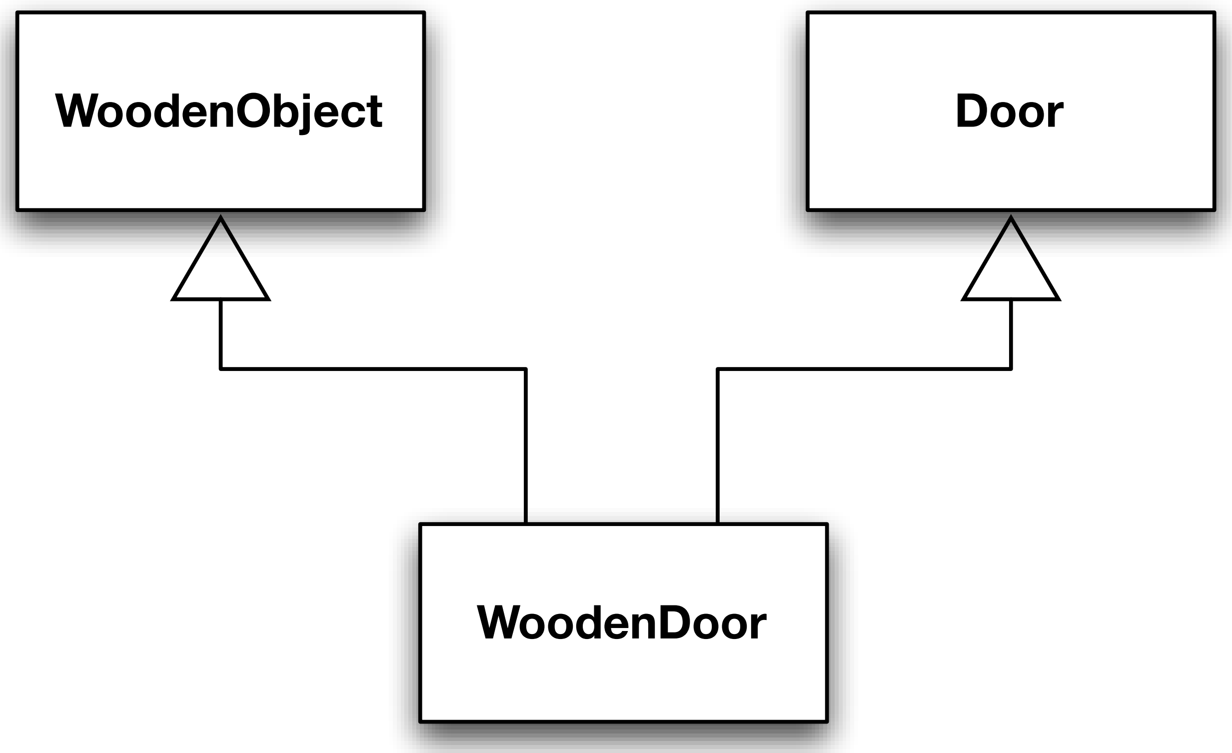




Multiple Inheritance

- So, is there a valid use of multiple inheritance?
    - Yes, sub-typing for combination
        - It is used to define a new class that is a special type of two other classes where those two base classes are from different domains
    - In such cases, the derived class can then legally combine data and behavior from the two different base classes in a way that **makes semantic sense**

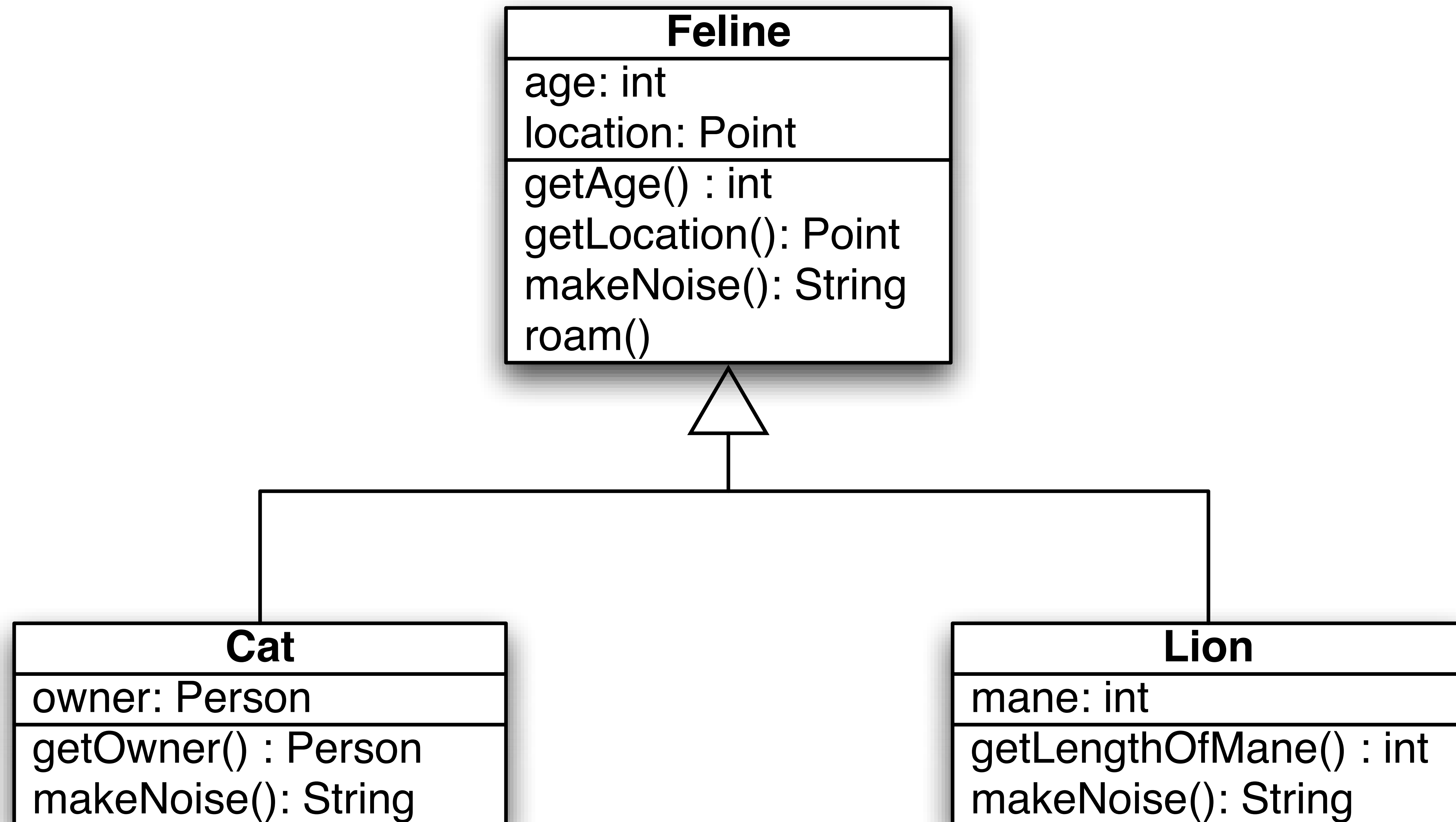
Multiple Inheritance Example



Is a wooden door a special type of door? **Yes**
Is a door part of a wooden door? **No**
Is a wooden door a special type of wooden object? **Yes**
Is a wooden object part of a door? **No**
Is a wooden object a special type of door? **No**
Is a door a special type of wooden object? **No**
All Heuristics Pass!

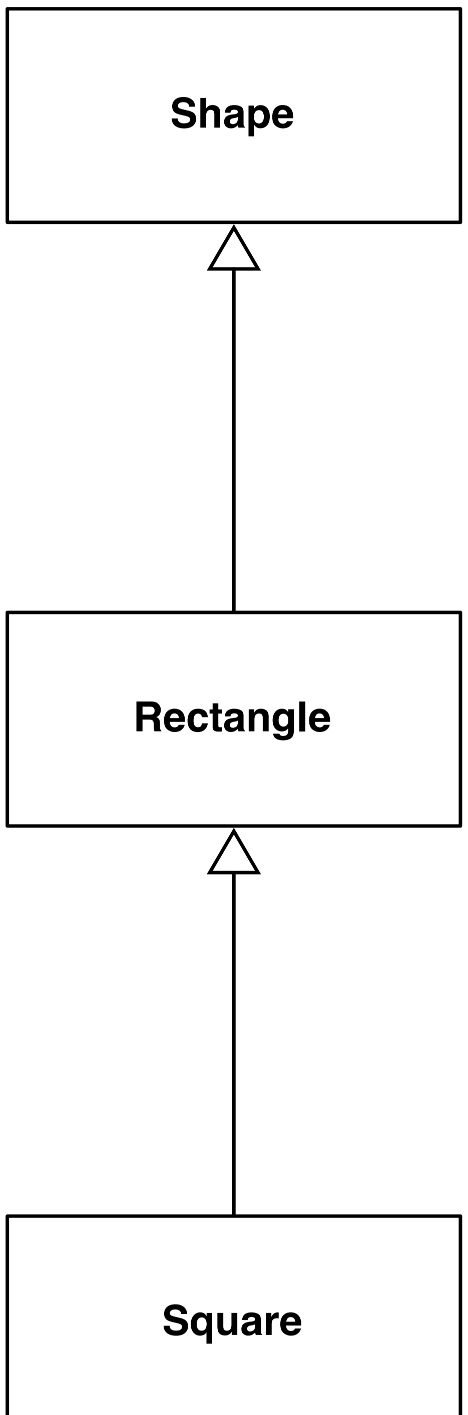
Homework 1: On Its Way

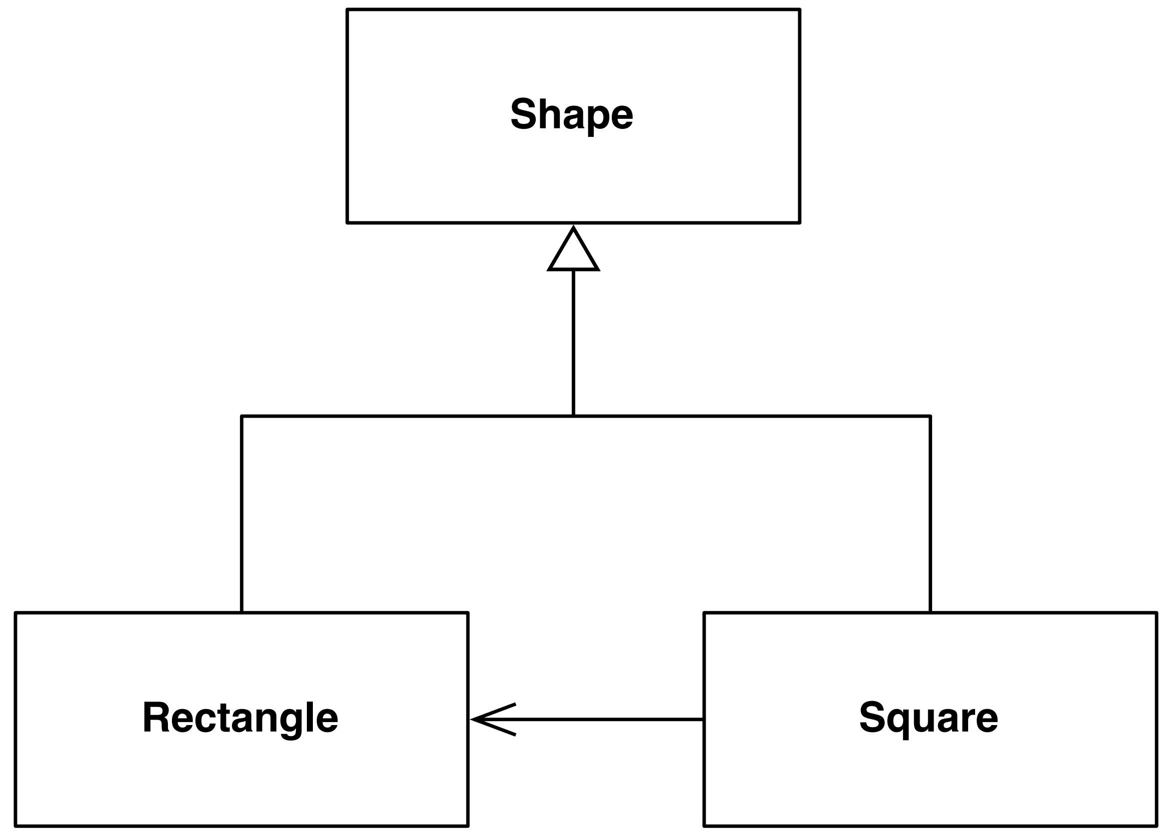
- Will involve questions concerning this diagram



Homework 1: On Its Way

- And these two diagrams...





Coming Up Next

- Lecture 4: Object Fundamentals, Part 3
- Lecture 5: Great Software
    - Read Chapter 1 of the OO A&D book