UML & OO Fundamentals

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 3 — 08/30/2011

Goals of the Lecture

- Review the material in Chapter 2 of the Textbook
  - Cover key parts of the UML notation
  - Demonstrate the ways in which I think it is useful
  - Give you a chance to apply the notation yourself to several examples
- **Warning: I repeat important information several times in this lecture; this is a hint to the future you when you are studying for the midterm.**

UML

- UML is short for **Unified Modeling Language**
  - The UML defines a standard set of **notations** for use in **modeling** object-oriented systems
- Throughout the semester we will encounter UML in the form of
  - class diagrams
  - sequence/collaboration diagrams
  - state diagrams
  - activity diagrams, use case diagrams, and more

(Very) Brief History of the UML

- In the 80s and early 90s, there were multiple OO A&D approaches (each with their own notation) available
- Three of the most popular approaches came from
  - James Rumbaugh: OMT (Object Modeling Technique)
  - Ivar Jacobson: Wrote "OO Software Engineering"
  - Grady Booch: Booch method of OO A&D
- In the mid-90's all three were hired by Rational and together developed the UML; "three amigos"

Big Picture View of OO Paradigm

- OO techniques view software systems as
  - **networks of communicating objects**
- Each object is an **instance** of a class
  - All objects of a class share similar **features**
    - **attributes**
    - **methods**
  - Classes can be **specialized** by **subclasses**
- Objects **communicate** by **sending messages**

Objects (I)

- Objects are **instances** of **classes**
  - They have state (**attributes**) and exhibit behavior (**methods**)
- 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**
  - Object A sends a message to Object B to request it perform a task
    - When done, B may pass a value back to A
    - Sometimes A == B
      - i.e., an object can send a message to itself

Objects (III)

- Sometimes messages can be **rerouted**
  - invoking a method defined in class A may in fact invoke an overridden version of that method in subclass B
  - a method of class B may in turn invoke messages on its superclass that are then handled by overridden methods from lower in the hierarchy
- The fact that messages (dynamic) can be rerouted distinguishes them from procedure calls (static) in non-OO languages

Objects (IV)

- In response to a message, an object may
  - update its internal state
  - return a value from its internal state
  - perform a calculation based on its state and return the calculated value
  - create a new object (or set of objects)
  - delegate part or all of the task to some other object

Objects (V)

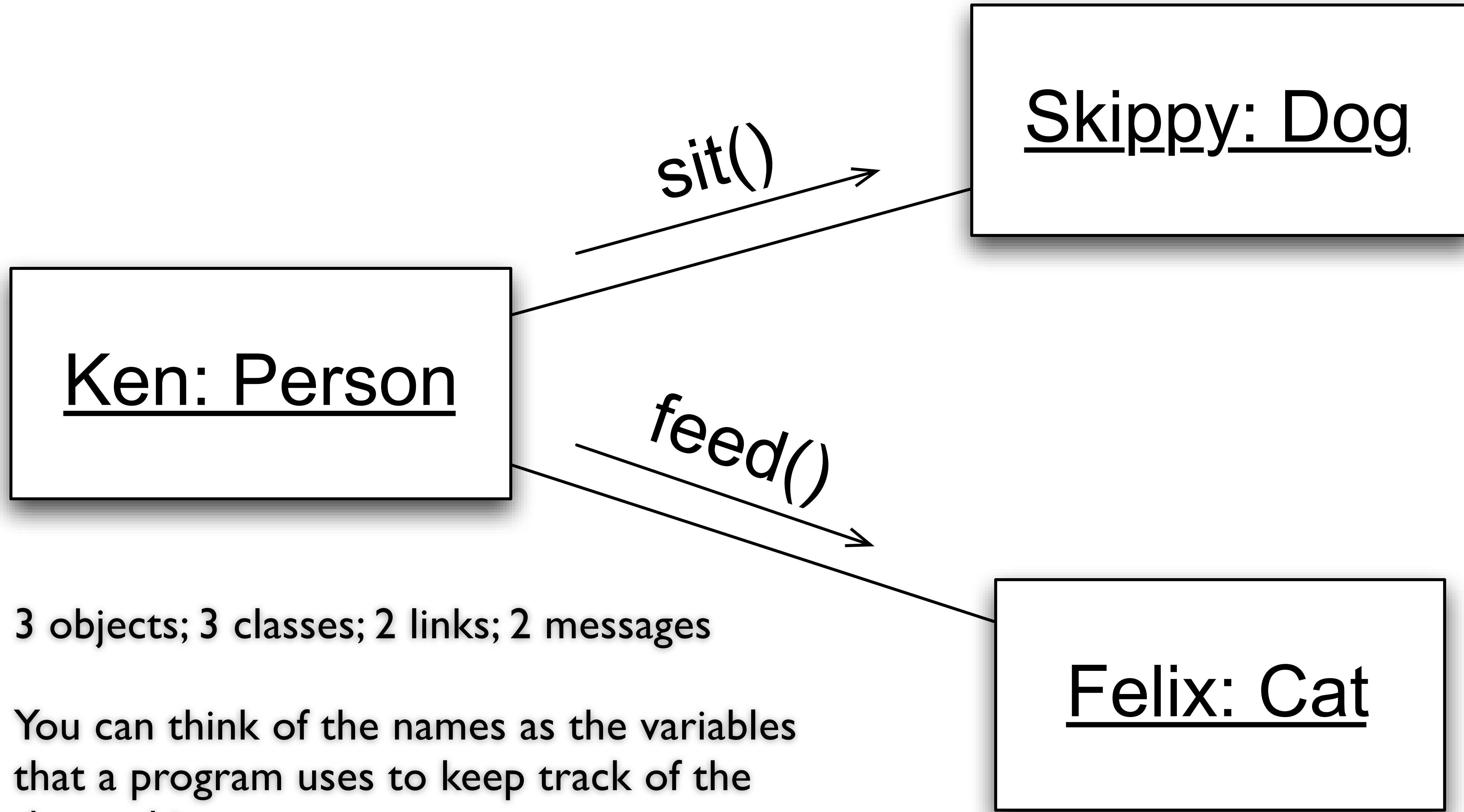
- As a result, objects can be viewed as **members of multiple object networks**
  - Object networks are also called **collaborations**
- Objects in an collaboration work together to perform a task for their host application

Objects (VI)

- UML notation
  - **Objects** are drawn as rectangles with their names and types (class names) underlined
    - <u>Ken : Person</u>
  - The name of an object is optional. The type is required
    - <u>: Person</u>
  - Note: The colon is not optional.

Objects (VII)

- Objects that work together have lines drawn between them
  - This connection has many names
    - **object reference**
    - **reference**
    - **link**
  - Messages are sent across links
    - Links are instances of **associations** (see slide 30)



3 objects; 3 classes; 2 links; 2 messages

You can think of the names as the variables that a program uses to keep track of the three objects

Classes (I)

- A class is a **blueprint** for an object
  - The blueprint specifies a class's attributes and methods
    - 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)

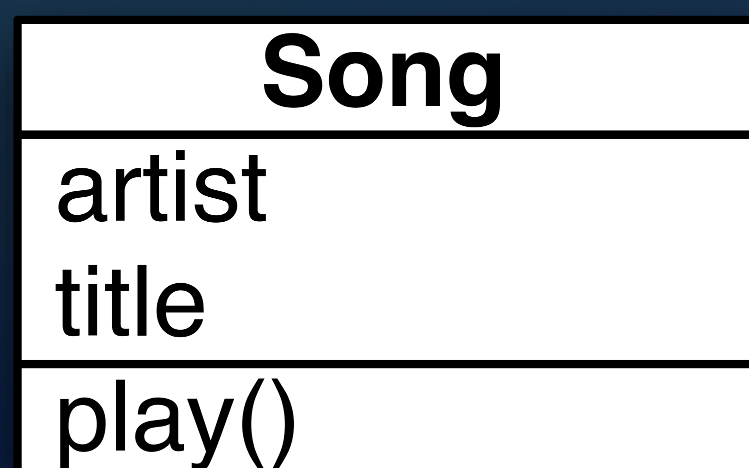
- An object of a class **has its own values** for the attributes of its class
  - For instance, two objects of the Person class can have different values for the name attribute
- Objects **share** the implementation of a class's **methods**
  - and thus behave similarly
    - i.e. Objects A and B of type Person each share the same implementation of the sleep() method

Classes (III)

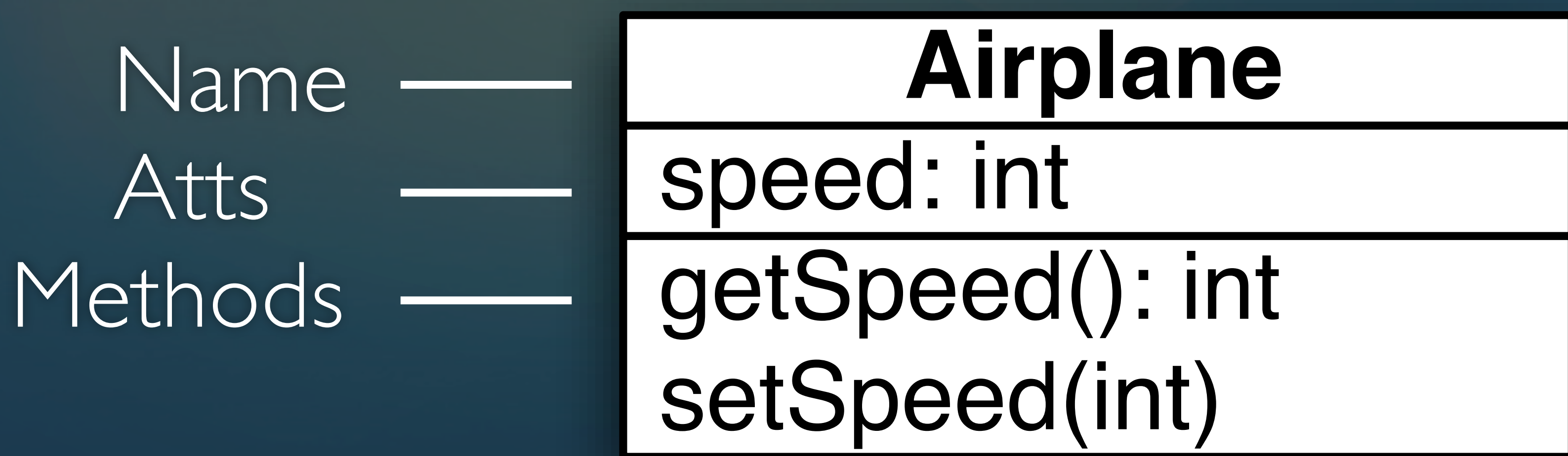
- Classes can define "class-based" (a.k.a. **static**) attributes and methods
  - A **static attribute** is shared among a class's objects
    - That is, all objects of that class can read/write the static attribute
  - A **static method** does not have to be accessed via an object; you invoke static methods directly on a class
    - In Lecture 2's Java code: String.format() was an example of a static method

Classes (IV)

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



Class Diagrams, 2nd Example



All parts are optional except the class name

A class is represented as a rectangle

This rectangle says that there is a class called Airplane that could potentially have many instances, each with its own speed variable and methods to access it

Translation to Code

- Class diagrams can be translated into code straightforwardly
  - Define the class with the specified name
  - Define specified attributes (**assume private access**)
  - Define specified method skeletons (**assume public**)
- May have to deal with unspecified information
  - Types are optional in class diagrams
  - Class diagrams typically do not specify constructors
    - just the class's **public interface**

Airplane in Java

Using Airplane

```
Airplane a = new Airplane(5);

a.setSpeed(10);

System.out.println(
    “” + a.getSpeed());
```

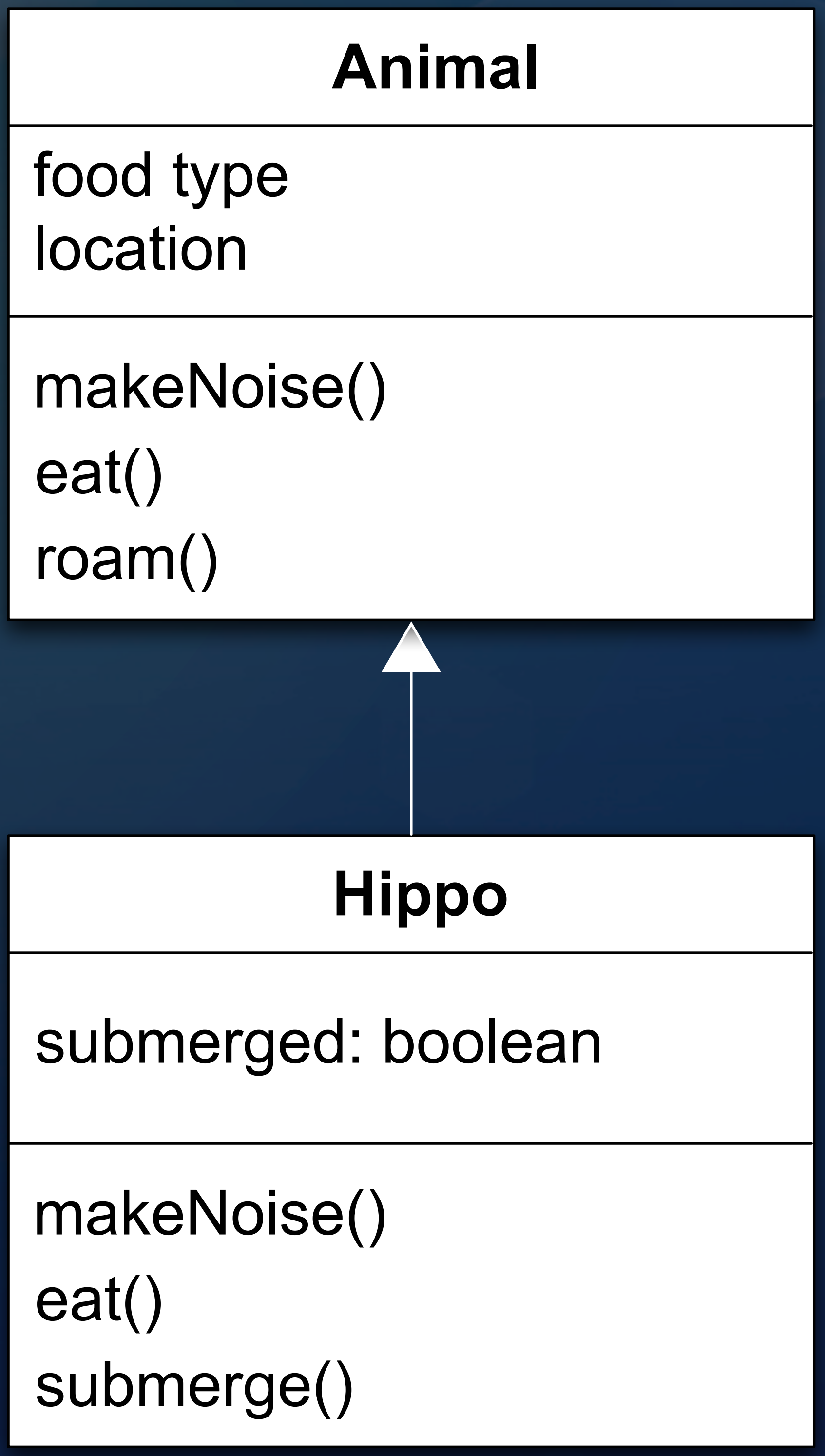
```java
public class Airplane {

    private int speed;

    public Airplane(int speed) {
        this.speed = speed;
    }

    public int getSpeed() {
        return speed;
    }

    public void setSpeed(int speed) {
        this.speed = speed;
    }

}
```

Relationships Between Classes

- Classes can be related in a variety of ways
  - Inheritance
  - Association
    - Multiplicity
  - Whole-Part (Aggregation and Composition)
  - Qualification
  - Interfaces

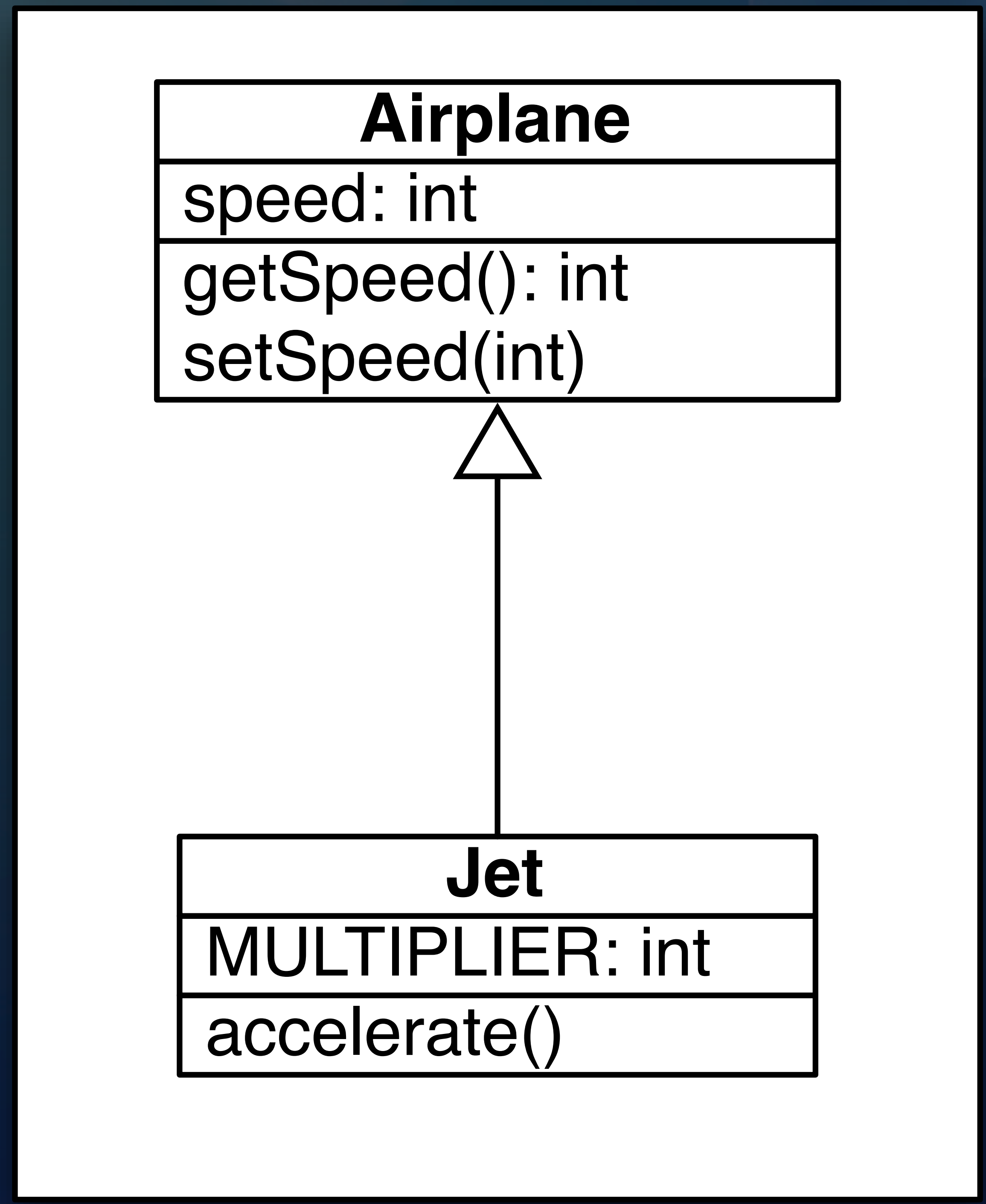
Relationships: **Inheritance**

- One class can **extend** another
- notation: a white triangle points to the superclass
  - the subclass can add attributes
    - Hippo adds **submerged** as new state
  - the subclass can add behaviors or **override** existing ones
    - Hippo is overriding makeNoise() and eat() and adding submerge()



Inheritance

- Inheritance lets you build classes based on other classes and **avoid duplicating code**
  - Here, Jet builds off the basics that Airplane provides



Inheriting From Airplane (in Java)

```java
public class Jet extends Airplane {

    private static final int MULTIPLIER = 2;

    public Jet(int id, int speed) {
        super(id, speed);
    }

    public void setSpeed(int speed) {
        super.setSpeed(speed * MULTIPLIER);
    }

    public void accelerate() {
        super.setSpeed(getSpeed() * 2);
    }

}
```

Note:

extends keyword indicates inheritance

super() and super keyword is used to refer to superclass

No need to define getSpeed() method; its inherited!

setSpeed() method overrides behavior of setSpeed() in Airplane

subclass can define new behaviors, such as accelerate()

Polymorphism: "Many Forms"

- **"Being able to refer to different derivations of a class in the same way, …"**
  - Implication: both of these are legal statements
    - Airplane plane = new Airplane();
    - Airplane plane = new Jet();
- **"…but getting the behavior appropriate to the derived class being referred to"**
  - when I invoke setSpeed() on the second plane variable above, I will get Jet's method, not Airplane's method

Encapsulation

- Encapsulation lets you
  - hide data and algorithms in one class from the rest of your application
  - limit the ability for other parts of your code to access that information
  - protect information in your objects from being used incorrectly

Encapsulation Example

- The "speed" instance variable is private in Airplane. That means that Jet doesn't have direct access to it.
    - Nor does any client of Airplane or Jet objects
- Imagine if we changed speed's visibility to public
- The encapsulation of Jet's setSpeed() method would be destroyed

```
Airplane

...
public void setSpeed(int speed) {
    this.speed = speed;
}
...

Jet

...
public void setSpeed(int speed) {
    super.setSpeed(speed * MULTIPLIER);
}
...

```

Reminder: Abstraction

- Abstraction is **distinct** from encapsulation
- It answers the questions
  - What features does a class provide to its users?
  - What services can it perform?
- Abstraction is the MOST IMPORTANT concern in A&D!
  - The choices you make in defining the abstractions of your system will live with you for a LONG time

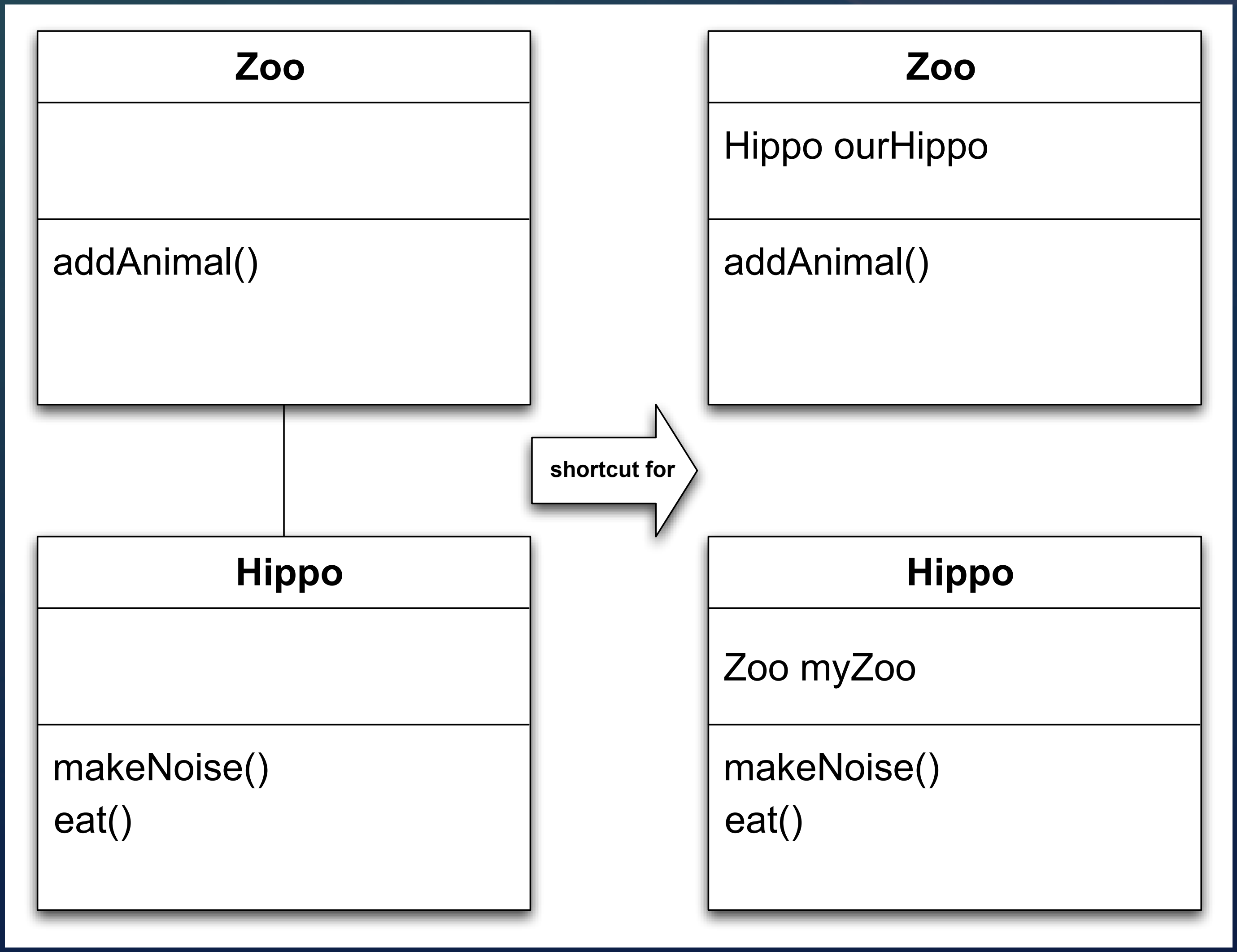
The Difference Illustrated

- The getSpeed() and setSpeed() methods represent Airplane's abstraction
    - Of all the possible things that we can model about airplanes, we choose just to model speed
- Making the speed attribute private is an example of encapsulation; if we choose to use a linked list to keep track of the history of the airplane's speed, we are free to do so

```java
public class Airplane {

    private int speed;

    public Airplane(int speed) {
        this.speed = speed;
    }

    public int getSpeed() {
        return speed;
    }

    public void setSpeed(int speed) {
        this.speed = speed;
    }

}
```

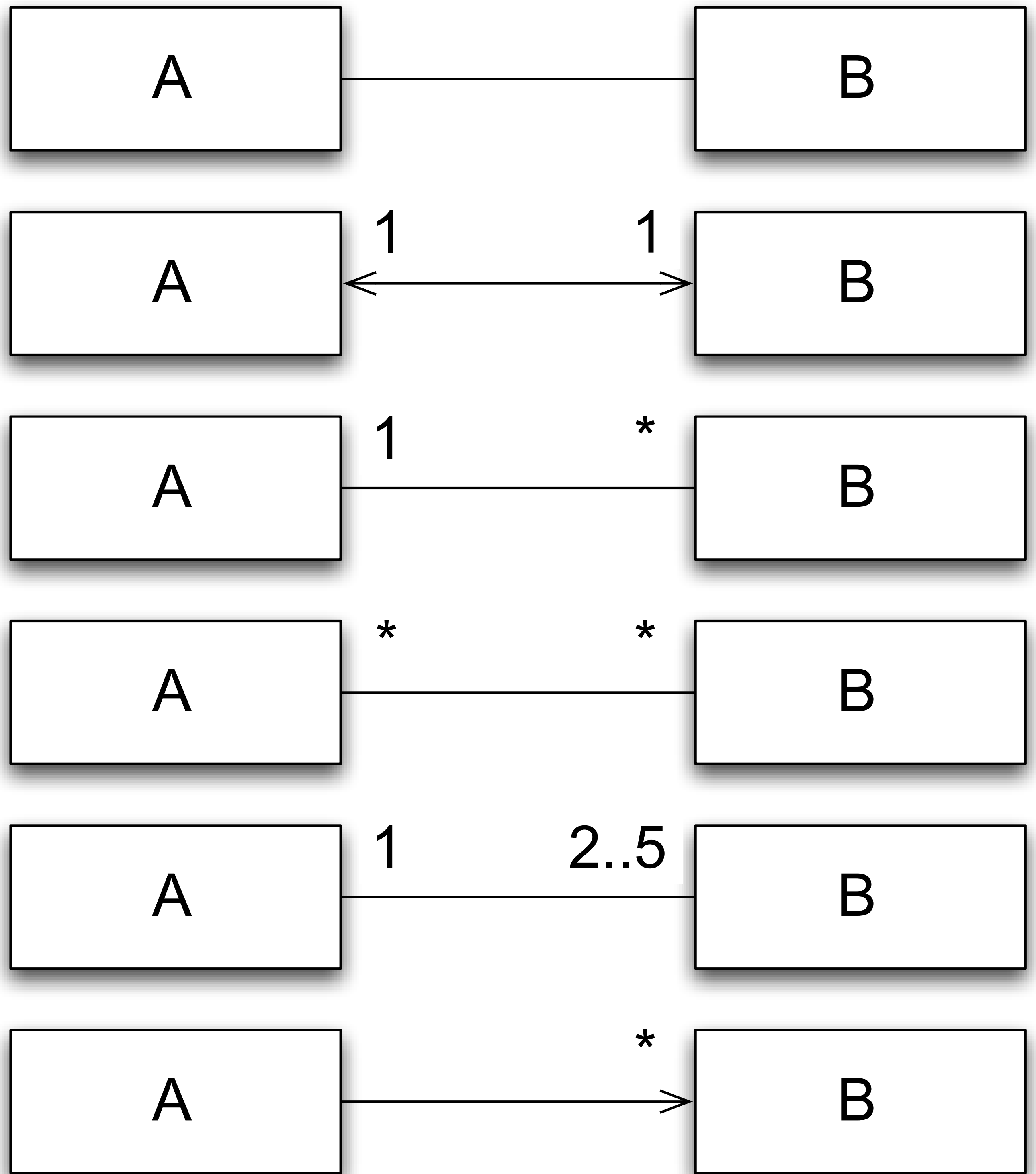
Relationships: **Association**

- One class can reference another (a.k.a. association)
    - notation: straight line
- This notation is a graphical shorthand that each class contains an attribute whose type is the other class
    - This is just one way to implement this; there are MANY others



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
  - if so, it indicates that the "knowledge" of the relationship is not bidirectional
- Examples on next slide



One B with each A; one A with each B

Same as above

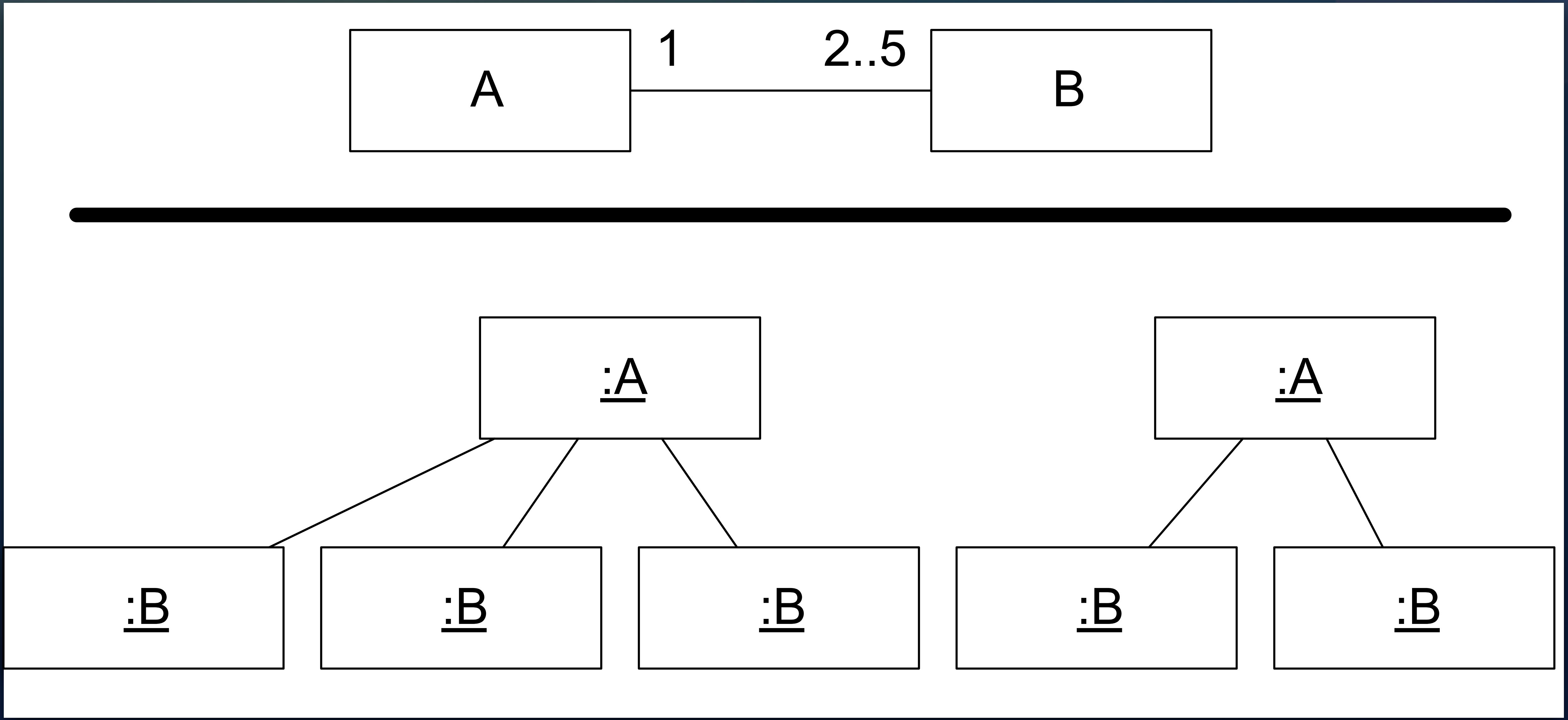
Zero or more Bs with each A; one A with each B

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

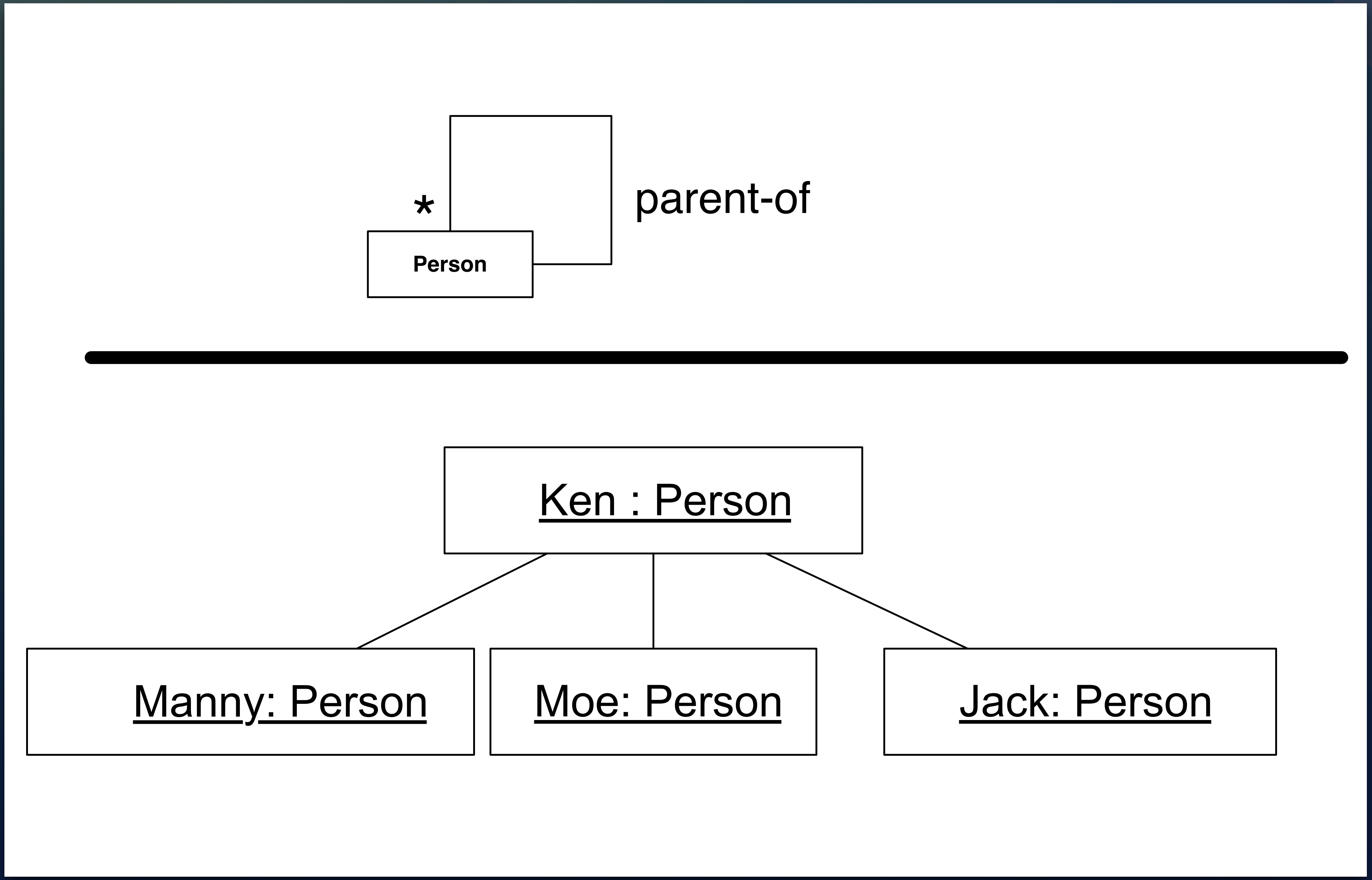
Two to Five Bs with each A; one A with each B

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

Multiplicity Example



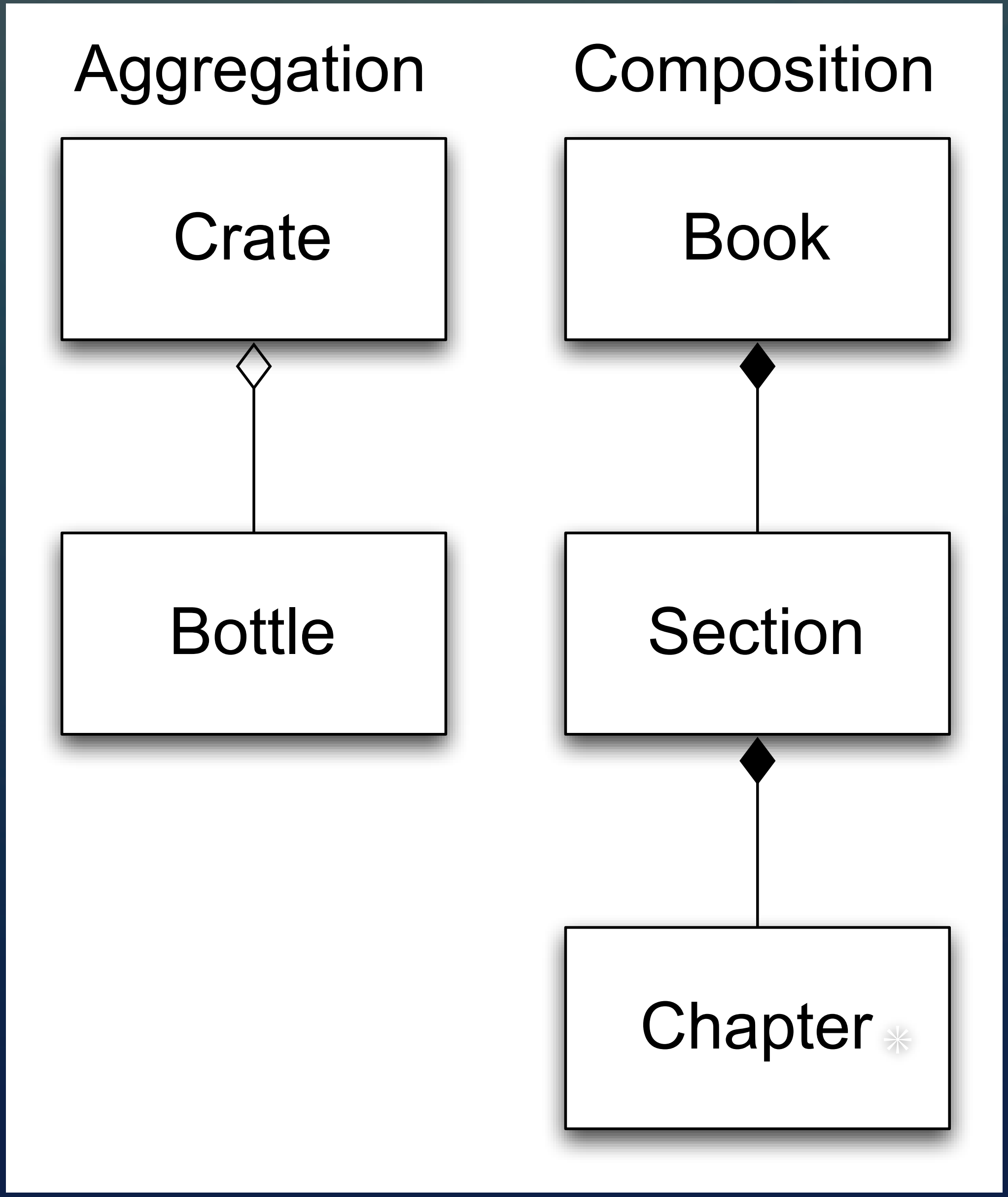
Self Association



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: multiplicity annotations for aggregation/composition is tricky

Some authors assume "one to many" when the diamond is present; others assume "one to one" and then add multiplicity indicators to the other end

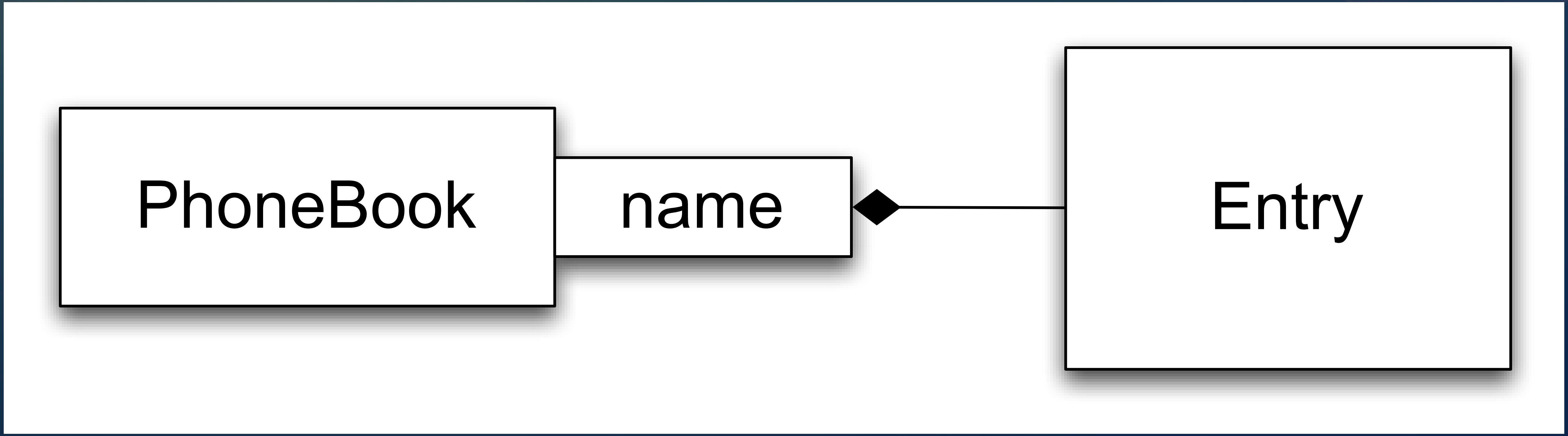
Semantics of Aggregation

- 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
  - Notation: a qualification is indicated with a rectangle attached to the end of an association indicating the attributes used in the query

Qualification Example

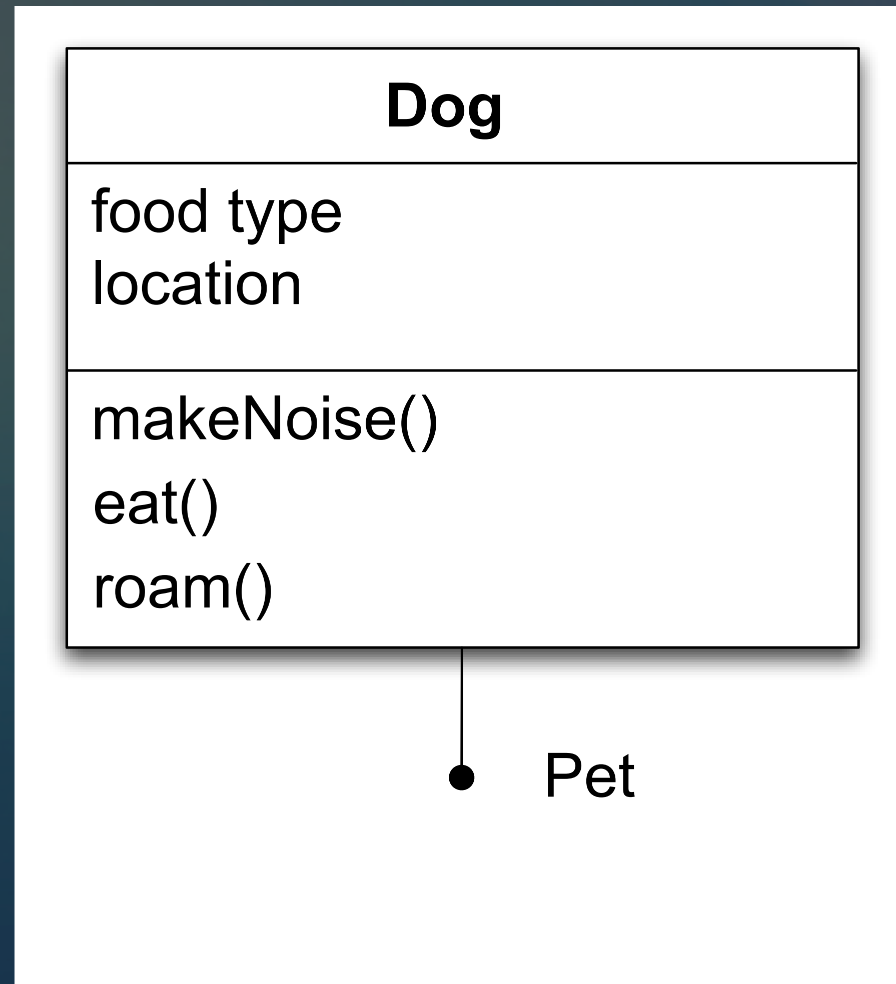


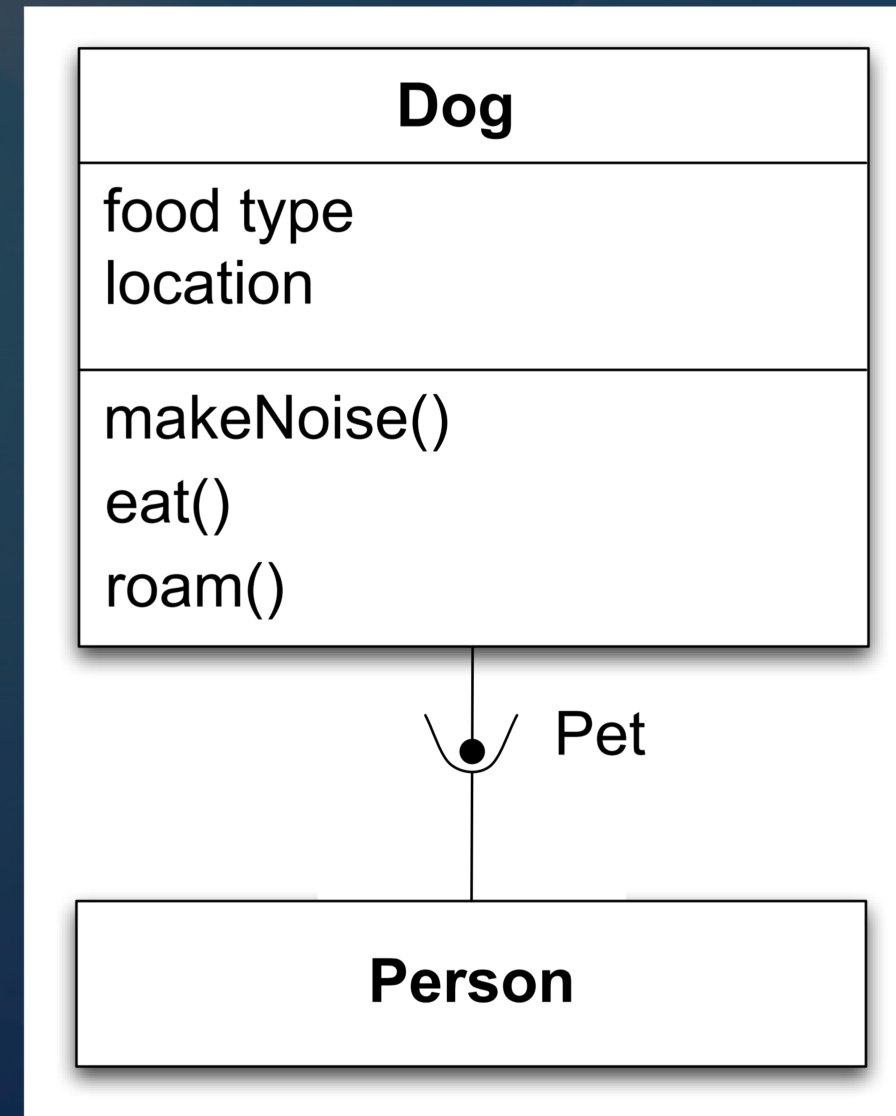
Qualification is not used very often; the same information can be conveyed via a note or a use case that accompanies the class diagram

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**, with each role providing a different set of services
    - These roles are then independent of the class's inheritance relationships

Example





- Other classes can then access a class via its interface
  - This is indicated via a "ball and socket" notation

Class Summary

- Classes are blue prints used to create objects
- Classes can participate in multiple types of relationships
  - inheritance, association (with multiplicity), aggregation/composition, qualification, interfaces

Your Turn

- Draw the following UML diagrams
  - A can have zero or more B's; each B can have 3-4 C's
  - A inherits from B; B implements an interface called C; D accesses B via C's interface
  - B's are accessed from A via an id
  - A composes zero or more B's; C aggregates zero or more A's

Questions

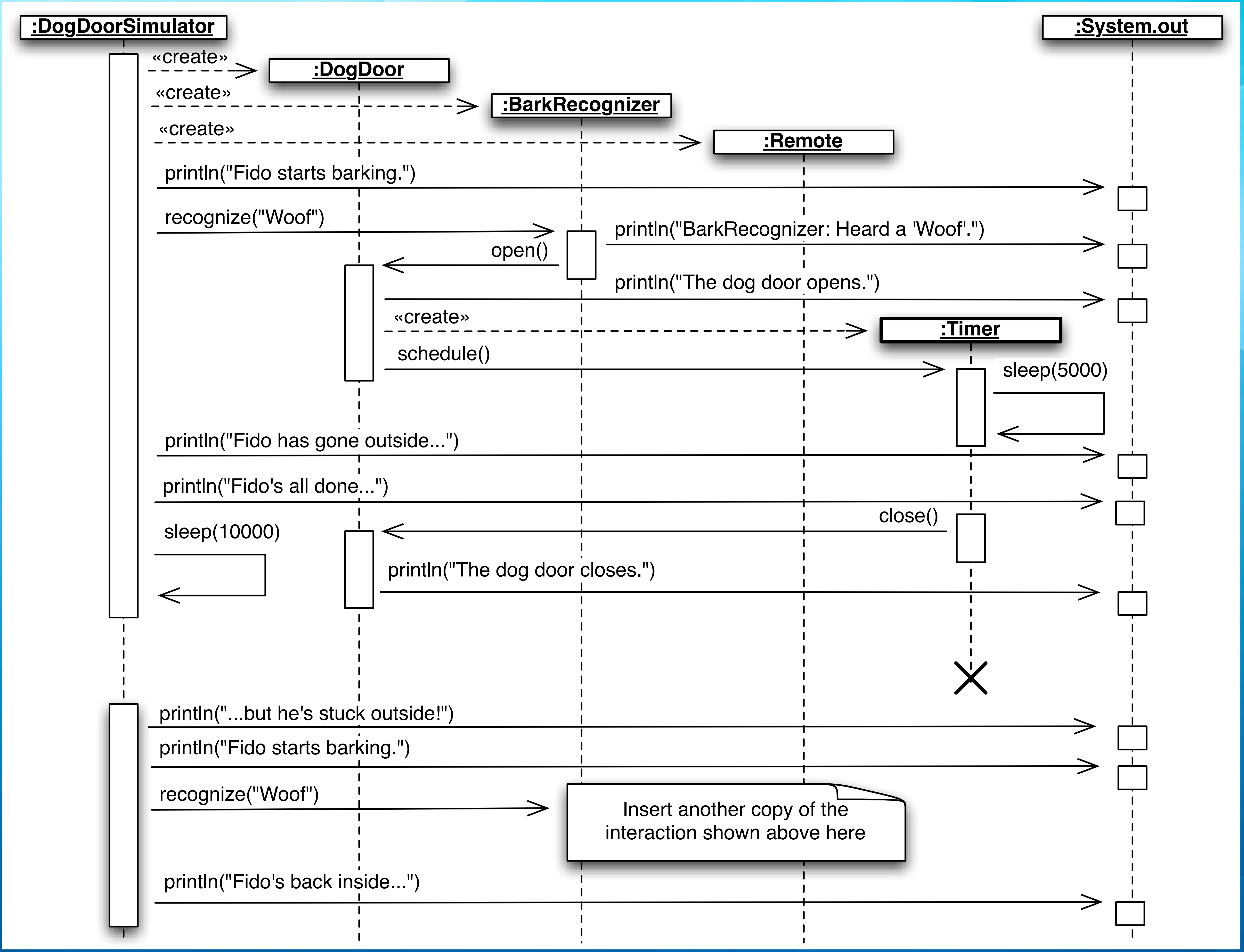
- Given
  - *A inherits from B; B implements an interface called C; D accesses B via C's interface*
- Can D access an instance of A via C's interface?
- How would you implement the following?
  - A can contain zero or more B's
  - B's are accessed from A via an id
  - A composes zero or more B's; C aggregates zero or more A's

Sequence Diagrams (I)

- Objects are shown across the top of the diagram
  - Objects at the top of the diagram existed when the scenario begins
    - All other objects are created during the execution of the scenario
- Each object has a vertical dashed line known as its lifeline
  - When an object is active, the lifeline has a rectangle placed above its lifeline
  - If an object dies during the scenario, its lifeline terminates with an "X"

Sequence Diagrams (II)

- Messages between objects are shown with lines pointing at the object receiving the message
  - The line is labeled with the method being called and (optionally) its parameters
- All UML diagrams can be annotated with "notes"
- Sequence diagrams can be useful, but they are also labor intensive (!)


:DogDoorSimulator
:System.out
«create»
:DogDoor
«create»
:BarkRecognizer
«create»
:Remote
println("Fido starts barking.")
recognize("Woof")
println("BarkRecognizer: Heard a 'Woof'.")
open()
println("The dog door opens.")
«create»
:Timer
schedule()
sleep(5000)
println("Fido has gone outside...")
println("Fido's all done...")
close()
sleep(10000)
println("The dog door closes.")
println("...but he's stuck outside!")
println("Fido starts barking.")
recognize("Woof")
Insert another copy of the
interaction shown above here
println("Fido's back inside...")

Coming Up Next

- Lecture 4: **More OO Fundamentals**
- Homework 1 due Thursday night by 11:59 PM
- Lecture 5: Example problem domain and traditional OO solution
  - Read Chapters 3 and 4 of the Textbook
- Homework 2 assigned on Thursday