Lecture 2 and 3: Fundamental Object-Oriented Concepts

- Kenneth M. Anderson

- January 13, 2005
- January 18, 2005

Lecture Goals

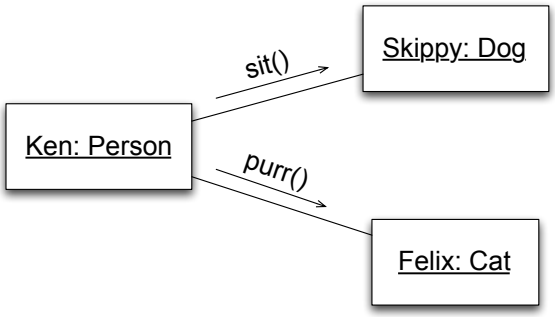
- Introduce the basic concepts of object-oriented analysis/design/programming techniques
  - A benefit of the object-oriented approach is that the same concepts appear in all three stages of software development
  - NOTE: some concepts in this lecture will be reintroduced in other lectures
    - That's okay because repetition is good!
- Cover a number of examples
- Demonstrate these concepts in code
- Credit where Credit is Due
  - Some examples have been taken from
    - Head First Java by Sierra & Bates, © O'Reilly, 2003

- Overview
  - Objects
    - Messages
  - Classes
    - Encapsulation
  - Composition
    - HAS-A
  - Inheritance
    - Abstraction
    - IS-A
  - Polymorphism
    - message passing
    - polymorphic arguments and return types

- Interfaces and Abstract Classes
- Object Identity
- Code Examples

- Objects
  - Object-oriented techniques view the world as consisting of objects
  - Objects have
    - **state** (aka attributes)
    - **behavior** (aka methods)
  - Objects interact by sending messages to one another
    - A message is a request by object A to have object B perform a particular task
      - When the task is complete, B may pass a value back to A
      - Note: sometimes B == A (i.e. an object can send a message to itself)
    - 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
      - Object networks will work together to perform a particular task for their host application
        - As a result, these networks are typically called ***Collaborations***
  - When drawn
    - objects appear as rectangles, with their names and types **underlined**
    - objects that know about each other have lines drawn between them
      - This connection is known as an ***object reference***, or just, ***reference***
      - Messages are sent across references

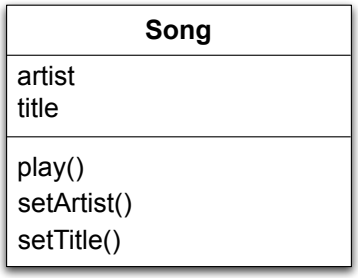


- Classes
  - A class is a blueprint for an object
    - The blueprint specifies the attributes (aka ***instance variables***) and methods of the class
      - attributes are things an object of that class **knows**
      - methods are things an object of that class **does**
    - An object is ***instantiated*** (created) from the description provided by its class
      - Thus objects are often called ***instances***
    - Each object 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
    - Each object shares the implementation of a class's methods
      - When a class is defined, a developer provides an implementation for each of its methods

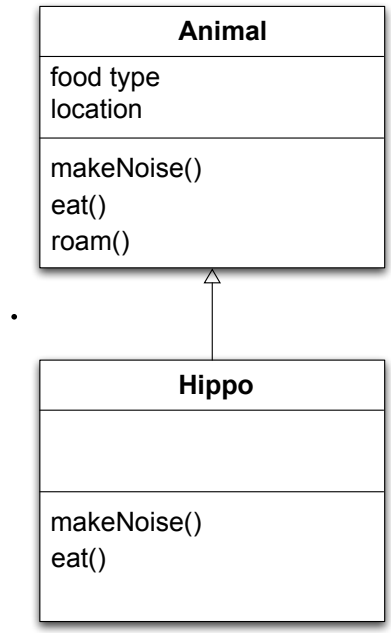
- Thus object A and B of type Person each share the same method implementation for the **`sleep()`** method; This approach ensures that objects of the same class behave similarly

- Classes can define "class wide" (aka ***static***) attributes and methods
  - A static attribute is shared among all instances of a class (each object has the same value for the static attribute)
  - A static method does not have to be accessed via an object; you invoke static methods directly on a class
  - We will see uses for static attributes and methods throughout the semester
- 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);
    - when you fill out the fields for a particular card, you are setting its state
  - You can do similar things to each card, i.e., each card has the same set of methods

- Notations
  - classes appear as rectangles with potentially multiple compartments
    - The first compartment contains its name (this name defines a type)
    - The second compartment contains the class's attributes
    - The third compartment contains the class's methods

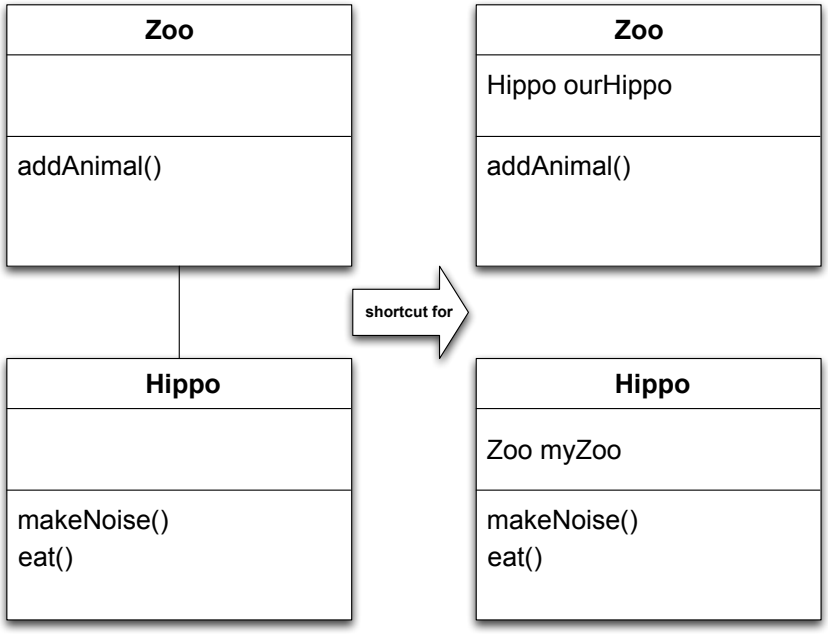


  - classes can be related to other classes in multiple ways
    - one class can extend another (aka ***inheritance***); indicated with an open triangle on the ***superclass***

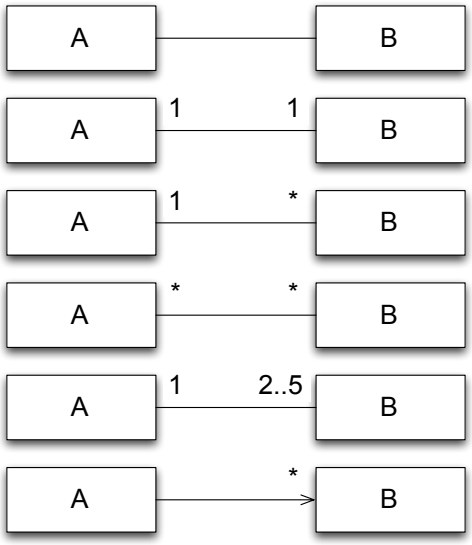


- one class can reference another (aka ***association***); indicated with a simple line

- Note: this notation is a graphical shorthand that one or both classes contain an attribute whose type is the other class

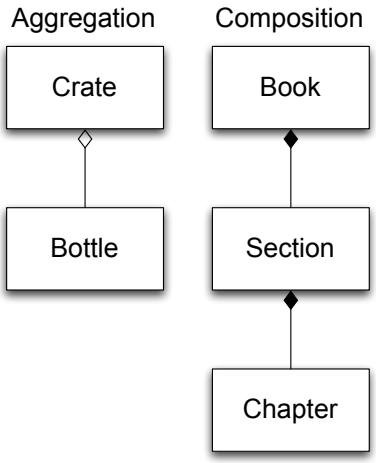


- associations can indicate the number of instances involved in the relationship; this is known as ***multiplicity;*** an association with no markings is assumed "one to one"; an association can also indicate directionality



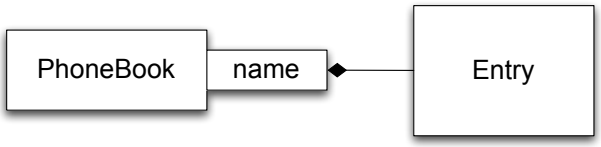
- associations can also convey semantic information about themselves; In particular, ***aggregations*** indicate that one object contains a set of other objects, think of it as a

whole-part relationship between a class representing an assembly of components and the classes representing the components

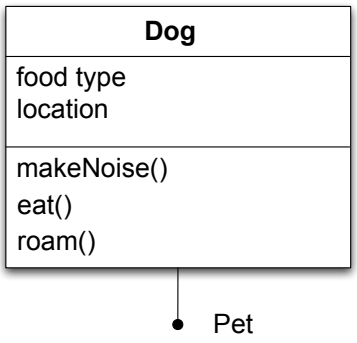


- aggregation relationships are ***transitive***; if A contains B and B contains C, then A contains C
- aggregation relationships are ***asymmetric***: if A contains B, then B cannot contain A
- An variant of aggregation is ***composition*** which adds the property of ***existence dependency***; if A composes B, then if A is deleted, B is deleted

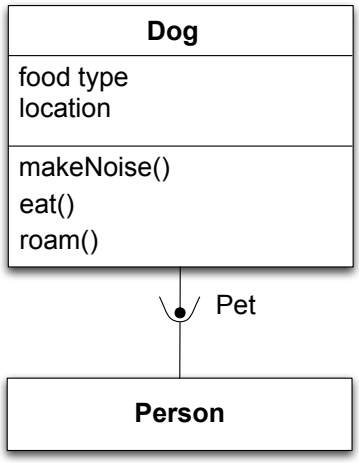
- Finally, associations can be ***qualified*** with information that indicates how objects on the other end of the association are found
  - This allows a designer to indicate that the association requires a query mechanism of some sort; For example, an association between an phonebook object and its entries might be qualified with a name attribute, indicating that a name is required to locate a particular entry



- a class can implement an interface; indicated with a "ball" sticking out of the class



- a class can communicate with another class through an interface; indicated via a "ball and socket" notation



Composition

- When designing a class, developers have three ways of dealing with requests made on its objects
  - One method is to simply deal with a request directly by implementing code in a method
  - A second method is to delegate the request to another object
    - This is called ***delegation*** or ***composition*** ("composing one object out of others")
  - A third method is to let a superclass handle the request
    - This is called ***inheritance*** which we will discuss next
- Composition is employed when some other class already exists to handle a request that might be made on the class being designed

- The "host" class simply creates an instance of the "helper" class and sends messages to it when appropriate
- These helper objects are typically stored internally to the host class (i.e. they are made **`private`** in the host class)
  - This means that other objects cannot access them directly, only indirectly via the host
- As such, composition is often referred to as a "HAS-A" relationship
  - For instance, a **`Bathroom`** HAS-A **`Bathtub`**

Advantages

- Composition is dynamic (not static); composition relationships can change at run-time
- Not tied to inheritance; in languages that support only single inheritance, this is important!

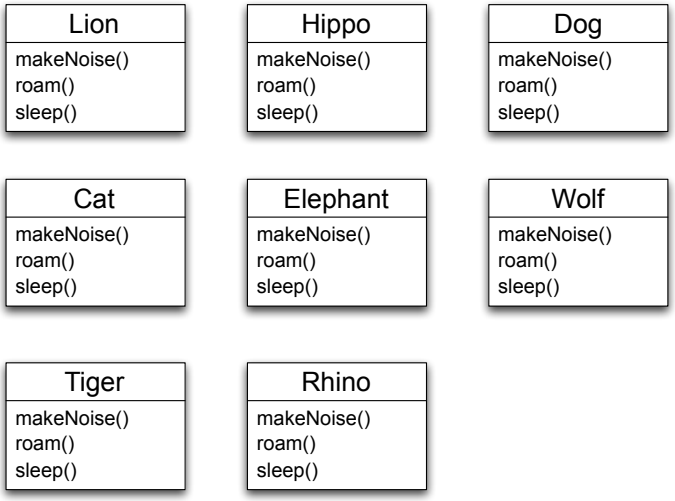
Inheritance

- Inheritance is a mechanism that allows a designer to state that one class is an . of several other classes
  - The former is called a ***superclass***
  - The latter are called ***subclasses***
- Single inheritance systems only allow a class to have one parent, or superclass
  - Multiple inheritance systems allow a class to have multiple parents
    - Multiple inheritance is tricky to use and presents several challenges to designers and programmers, as we will see later in the semester

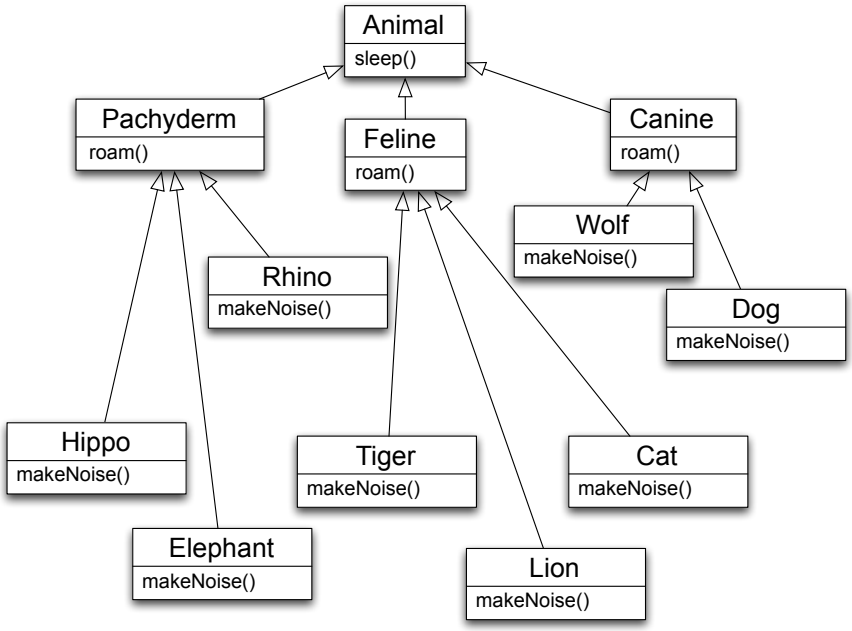
- Subclasses are also called subtypes because they are "more specific" versions of their superclasses; they restrict the "legal" values for the type
  - For instance, Real Numbers → Integers → Positive Integers
  - Or, Component → Container → Control → Button → Checkbox
- Subclasses have an "IS-A" relationship with their superclass; an IS-A relationship is one directional
  - A **`Hippo`** IS-A **`Animal`** makes sense while the reverse does not
  - IS-A relationships are transitive
    - If D is a subclass of C and C is a subclass of B, then D IS-A C and D IS-A B are both true
- Why is it called inheritance?

- The reason is that subclasses inherit attributes and methods from their superclasses
  - In particular, all **`public`** attributes and methods; **`private`** attributes and methods are not inherited
    - This enables significant code reuse since shared code can be located in root classes and shared by all subclasses

- Inheritance Example, Part 1



- Inheritance Example, Part 2



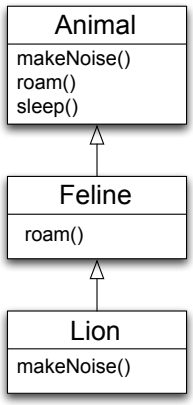
- Thus, a subclass can exhibit all of the behaviors of its superclasses

- As such, it can be used anywhere an instance of its superclass can (more on this later)
- Furthermore, a subclass can extend its superclass, providing additional behaviors that make sense for it
- In addition, a subclass can ***override*** the behaviors provided by the superclass, altering them to suit its needs
  - This is both powerful and dangerous, as we will discuss later in the semester

Polymorphism

- Object-Oriented programming languages support ***polymorphism***, which means "many forms"
  - In practice, this allows code to be written with respect to the root of an inheritance hierarchy and function correctly if applied to an instance of one of its subclasses
- To begin, consider why invoking a method on an object is known as "message passing" rather than say "method invocation"
  - The reason is that you may think you are sending a message to the method body of a superclass object and the run-time engine of the programming language dynamically "re-routes" the message to the method body of one of that object's subclasses

- Consider the following example




```
Animal a = new Lion()
a.makeNoise();
a.roam();
a.sleep();
```

  - We have created a **`Lion`** object but we are "looking at it" with a **`Animal`** variable; Without polymorphism, the code above would invoke the method bodies defined in the **`Animal`** class
  - But with polymorphism, "a.roam()" invokes the method body contained in the **`Feline`** class and "a.makeNoise()" invokes the method body contained in the **`Lion`** class

- Why is this important?
  - It allows us to write very abstract code that is robust with respect to the creation of new subclasses. For instance:

```
public void goToSleep(Animal[] zoo) {
    for (int i = 0; i < zoo.length; i++) {
        zoo[i].sleep();
    }
}
```

    - In the above code, it doesn't matter what type of animals are contained in the array; we simply iterate through the array and ask each animal to go to sleep. If a new subclass is created that overrides the **`sleep()`** method, we don't care, the above code will correctly invoke that overridden method; indeed the above code wouldn't even need to be re-compiled!

- Polymorphism is also supported when used as arguments to methods or as method return types
  - In the above **`goToSleep()`** method, we passed in a polymorphic argument, namely an array of **`Animal`**s; the

code does not care if the array contains **`Animal`** instances or any of its subclasses

- In addition, we can create methods that return polymorphic return values. For example:

```
public Animal createRandomAnimal() {
    // code that randomly creates and
    // returns one of Animal's subclasses
}
```

  - In the above code, we don't know ahead of time which instance of an **`Animal`** subclass will be returned by the **`createRandomAnimal()`** method; that's okay as long as we are happy to interact with it via the interface provided by the **`Animal`** class

- Indeed, we can view inheritance as establishing a contract by which a root class and any of its subclasses can be used; if we "code to the

contract" we can create very robust and easy to maintain software systems

- This notion of a class's interface specifying a contract is often referred to as "***design by contract***"
- In fact, this is why overriding methods in subclasses can be considered dangerous
  - if a subclass overrides a superclass method and then alters the behavior of that method such that it violates the intentions of the superclass, we have violated the contract specified by the superclass and this can indeed harm previously abstract and robust code that had been coded to that contract
    - Consider what would happen if an **`Animal`** subclass overrides the **`sleep()`** method to make its instances eat instead; our **`goToSleep()`** method above would fail in its goal of putting all of the Zoo's animals to sleep.

Interfaces and Abstract Classes

- There are times when you want to make the "design by contract" principle explicit rather than implicit
  - ***Abstract classes*** and ***Interfaces*** allow you to do this
    - An abstract class is simply one which cannot be instantiated
      - It is designed from the start to be sub-classed
      - It does this by declaring a number of methods but not providing method implementations for them; this sets a contract, since a subclass is required to provide implementations for each abstract method
      - Abstract classes are useful because they allow you to provide code for some of the methods (enabling code reuse) while still defining an abstract interface that subclasses must implement
      - Consider our Zoo example; while it makes sense to write code like this
      - Animal a = new Lion(); -- manipulate a **`Lion`** object via its **`Animal`** superclass
      - it makes less sense to write code like this:
      - Animal a = new Animal(); -- what animal is being created/manipulated?

      - Thus the **`Animal`** class, along with **`Feline`**, **`Pachyderm`**, and **`Canine`** classes, are good candidates for being abstract classes
    - Interfaces go one step further and only allow the declaration of abstract methods; you can not provide any method implementations for any of the methods declared by an interface
      - Interfaces are useful when you want to define a role in your software system that could be played by any number of classes
      - As we will see, interfaces allow you to address many of the needs that multiple inheritance was designed for
  - To demonstrate the utility of interfaces, consider wanting to modify the **`Animal`** class hierarchy to provide operations related to pets (e.g. play())
    - We have several options, all with their pros and cons
      - add pet methods and code to **`Animal`**
      - add pet methods to **`Animal`** but make them abstract
      - add pet methods only in the classes where they belong (no explicit contract)
      - make a separate **`Pet`** superclass and have pets inherit from both **`Pet`** and **`Animal`**
      - make a **`Pet`** ***interface*** and have only pets implement that interface

- Object Identity
  - All objects have identity
    - A property that allows us to distinguish one object from another
    - Considering two cups from the same set of china, we might say that they are equal but not identical
      - They are equal because they have the same set of attributes (size, shape, color, ...) and they have the same values for each of their attributes
      - But they are two distinct cups and we can select among them; no two objects can have the same identity (otherwise they would be the same object)
  - In object-oriented programming languages, all objects (i.e. instances) have a unique identifier
    - This identifier may be, for instance, the object's location in memory; or a unique integer that was assigned to it when it was created

- This identifier is used to enable a comparison of two variables to see if they point at the same object, e.g.,

```
public void compare(String a, String b) {
    if (a == b) {
        System.out.println("identical");
    } else if (a.equals(b)) {
        System.out.println("equal");
    } else {
        System.out.println("not equal");
    }
}

String ken = "Ken Anderson";
String max = "Max Anderson";
compare(ken, max); -- not equal
ken = max;
compare(ken, max); -- identical
max = new String("Max Anderson");
compare(ken, max); -- equal
```

- However, identity is also important in analysis and design
  - We do not want to create a class for objects that do not have unique identity in our problem domain
    - If there is a concept in our problem domain and we can not distinguish between separate instances of that concept, then we do not need a class for that concept
      - Consider people in an elevator; does the elevator care who pushes its buttons?
      - Consider a cargo tracking application; does the system need to monitor every carrot that exists inside a bag? how about each bag of carrots inside a crate?
      - Consider a flight between Denver and Chicago; what uniquely identifies that flight? The flight number? The plane? The cities? What?
      - Consider a telephone "chat line"; what constitutes a call?
    - When performing analysis, you will confront issues like these; you will be searching for uniquely identifiable objects that help you solve your problem

- Class Activity
  - Lets practice using OO concepts to model a few example scenarios
    - Scenario 1
      - A structural computing system is made up of elements. There are two types of elements, atoms and collections. Atoms are used to store application-specific objects supplied by clients; Collections are used to group other elements. All elements have a unique id and a set of attribute value pairs. The name of an attribute is a string but its value can be any number of different types all of which share a common interface. Elements are stored by a repository, which manages their persistence and which also can be used to search for specific elements via their attributes.
    - Scenario 2
      - The InfiniTe information integration environment provides a homogenous repository for performing requirements traceability tasks; InfiniTe contains two types of agents for manipulating the repository, translators and integrators. Translators are used to import information into documents; Translators can also be used to export information from documents out of the repository. Integrators are used to search for relationships between documents; The information space is partitioned into a number of contexts; All contexts are a member of another context except for the "global" context which serves as the root. Documents can be assigned to any number of contexts and can be stored in a number of

different formats. Each format consists of a number of format objects that represent the content of that document in that format. An anchor is used to indicate an item of interest within a document; Anchors are context-specific, allowing different items of a document to be highlighted in different contexts; A relationship is a set of anchors and can thus be used to link within a document, across documents, and across contexts.

- Code Examples
  - Basic class definition and object creation in Java, Python, and Objective-C
  - Inheritance hierarchies and code reuse
  - Polymorphism and examples of its benefits
  - Use of Interfaces
  - Object identity