More OO Fundamentals

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 4 — 09/01/2011

© Kenneth M. Anderson, 2011

Goals of the Lecture

- Continue a review of fundamental object-oriented concepts

Overview of OO Fundamentals

- Delegation
  - HAS-A
- More on Inheritance
  - IS-A
- More on Polymorphism
  - message passing
  - polymorphic arguments and return types
- Interfaces
  - Abstract Classes
- Object Identity

Delegation (I)

- When designing a class, there are four ways to handle an incoming message
  - Handle message by implementing code in a method
  - Let the class's superclass handle the request via inheritance
  - Pass the request to another object (delegation)
  - some combination of the previous three

© Kenneth M. Anderson, 2011

Delegation (II)

- Delegation 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 a private instance of the helper class and sends messages to it when appropriate
  - As such, delegation is often referred to as a "HAS-A" relationship
    - A Car object HAS-A Engine object

```java
import java.util.List;
import java.util.LinkedList;

public class GroceryList {

   private List<String> items;

   public GroceryList() {
      items = new LinkedList<String>();
   }

   public void addItem(String item) {
      items.add(item);
   }

   public void removeItem(String item) {
      items.remove(item);
   }

   public String toString() {
      String result = "Grocery List\n------------\n\n";
      int index = 1;
      for (String item: items) {
         result += String.format("%3d. %s", index++, item) + "\n";
      }
      return result;
   }

}

```

GroceryList delegates all of its work to Java's LinkedList class (which it accesses via the List interface).

© Kenneth M. Anderson, 2011

```java
1  public class Test {
2
3      public static void main(String[] args) {
4          GroceryList g = new GroceryList();
5          g.addItem("Granola");
6          g.addItem("Milk");
7          g.addItem("Eggs");
8          System.out.println("" + g);
9          g.removeItem("Milk");
10         System.out.println("" + g);
11     }
12
13 }
14
```

With the delegation, I get a nice abstraction in my client code. I can create grocery lists, add and remove items and get a printout of the current state of the list.

```java
import java.util.List;
import java.util.LinkedList;

public class TestWithout {

    public static void printList(List<String> items) {
       System.out.println("Grocery List");
       System.out.println("------------\n");
       int index = 1;
       for (String item : items) {
           System.out.println(String.format("%3d. %s", index++, item));
       }
       System.out.println();
    }

    public static void main(String[] args) {
        List<String> g = new LinkedList<String>();
        g.add("Granola");
        g.add("Milk");
        g.add("Eggs");
        printList(g);
        g.remove("Milk");
        printList(g);
    }

}
```

Without delegation, I get less abstraction. I'm using the List interface directly with its method names and I have to create a static method to handle the printing of the list rather than using toString().

© Kenneth M. Anderson, 2011

Delegation (III)

- Now, the two programs (with delegation and without delegation) produce exactly the same output
  - So, do we care which method we use?

Delegation (IV)

- Benefits of Delegation
  - Better abstraction
  - Less code in classes we write ourselves
  - We can change delegation relationships at runtime!
    - Unlike inheritance relationships; Imagine if we had created GroceryList as a subclass of LinkedList (*shudder*)
      - Why? Because GroceryList IS-NOT-A LinkedList

© Kenneth M. Anderson, 2011

Delegation (V)

- Changing delegation relationships at run-time
  - A class can use a set at run-time
    - `Set<String> items = new HashSet<String>();`
  - If the class suddenly needs to be sorted, it can do this
    - `items = new TreeSet<String>(items);`
  - We have changed the delegation to an entirely new object at run-time and now the items are sorted
    - In both cases, the type of items is Set<String> and we get the correct behavior via polymorphism

© Kenneth M. Anderson, 2011

Delegation (VI)

- Summary
  - Don't re-invent the wheel… delegate!
  - Delegation is dynamic (not static)
    - delegation relationships can change at run-time
  - Not tied to inheritance
    - indeed, considered much more flexible; In languages that support only single inheritance this is important!

Delegation (VII)

- Delegation, as a design pattern, is used throughout the iOS and Cocoa frameworks
  - Basic pattern involving two objects
    - Host and delegate; use delegate to customize host
    - Define an interface that a delegate will implement
      - some methods are required; the rest are optional
    - Host will invoke methods on delegate as needed to influence its behavior

iOS Delegation Example (I)

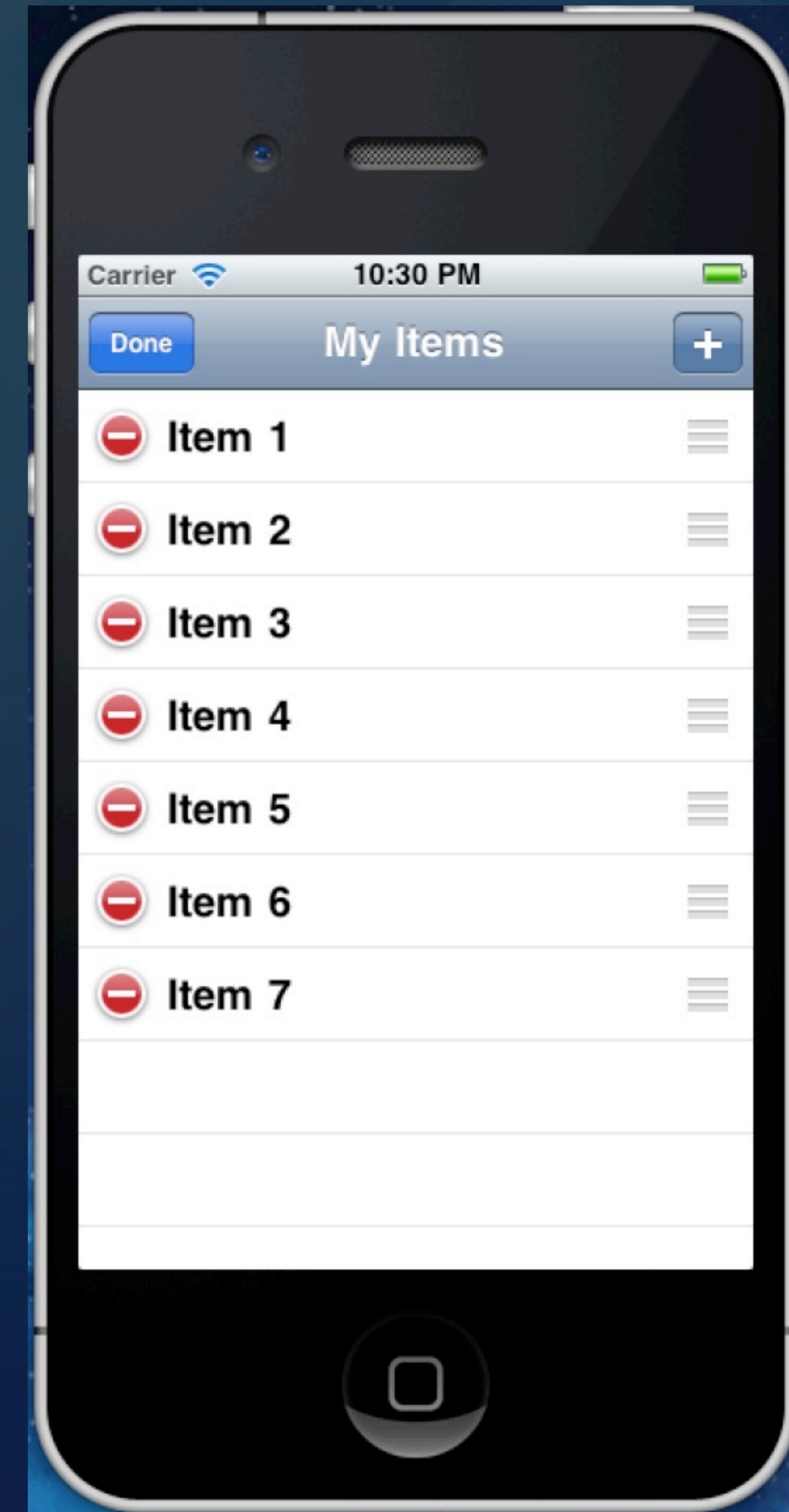
- UITableView displays a single-column table of rows
  - It requires two delegates
    - UITableViewDelegate
    - UITableViewDataSource
  - The first contains methods about how the table should look, how it should respond to selections, etc.
  - The second contains methods that populate the table and allow it to be edited

© Kenneth M. Anderson, 2011

iOS Delegation Example (II)

- iOS app with a UITableViewController
  - by default acts as both the
    - delegate and the
    - data source
  - Some cleverness
    - Move handles do not appear unless a delegate method is implemented

demo



© Kenneth M. Anderson, 2011

Inheritance (I)

- Inheritance is a mechanism for sharing (public/protected) features between classes
- Subclasses have an "IS-A" relationship with their superclass
    - 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 B is true

Inheritance (II)

- Good OO design strives to make sure that all IS-A relationships in a software system "make sense"
  - Consider Dog IS-A Canine vs. Dog IS-A Window
    - The latter might actually be tried by an inexperienced designer who wants to display each Dog object in its own separate window
      - This is known as **implementation inheritance**; it is considered poor design and something to be avoided

Inheritance (III)

- Inheritance enables significant code reuse since subclasses gain access to the code defined in their ancestors
- The next two slides show two ways of creating a set of classes modeling various types of Animals
  - The first uses no inheritance and likely contains a lot of duplicated code
  - The second uses inheritance and requires less code
    - even though it has more classes than the former

© Kenneth M. Anderson, 2011

Lion
makeNoise() roam() sleep()

Hippo
makeNoise() roam() sleep()

Dog
makeNoise() roam() sleep()

Cat
makeNoise() roam() sleep()

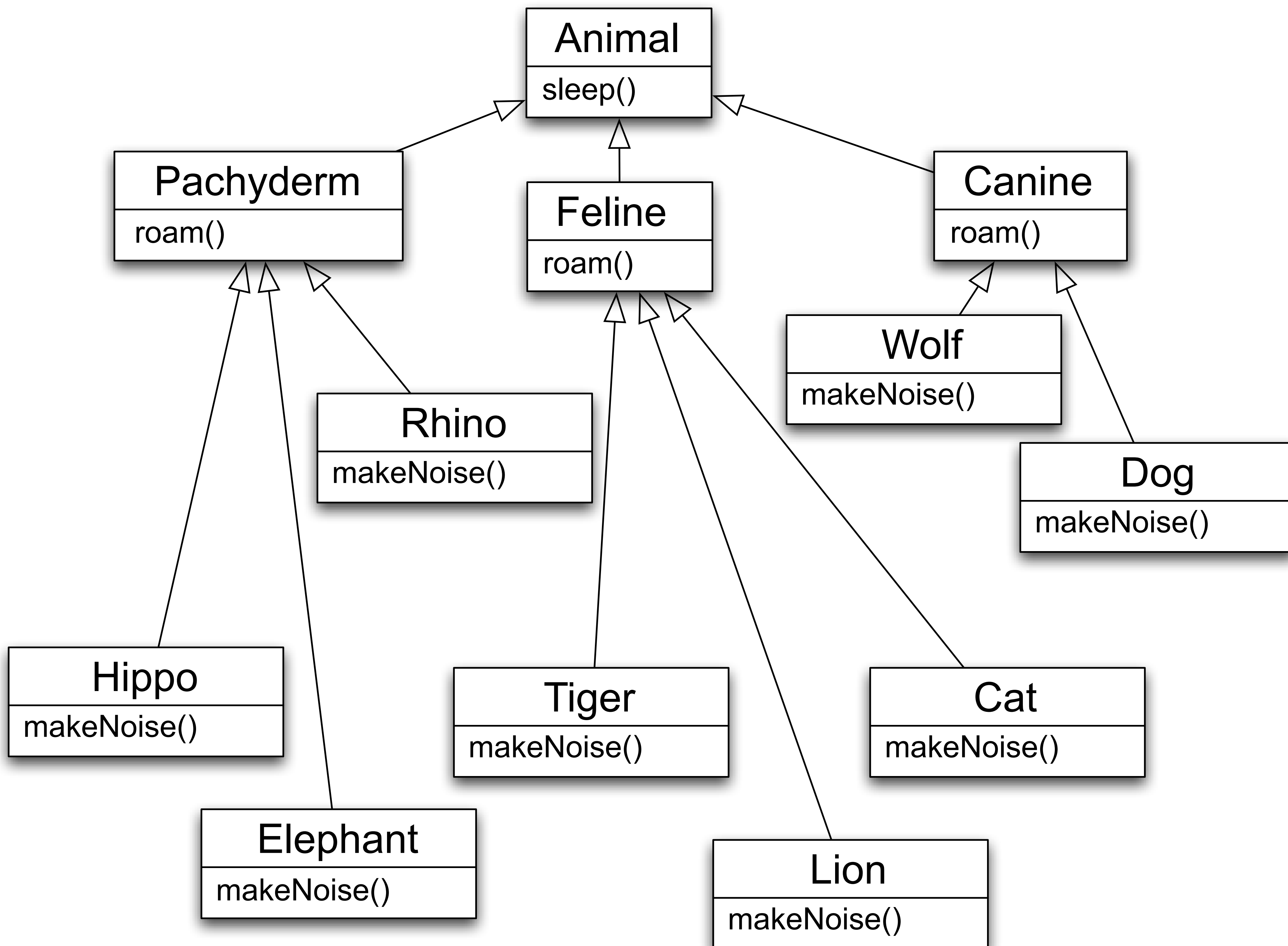
Elephant
makeNoise() roam() sleep()

Wolf
makeNoise() roam() sleep()

Tiger
makeNoise() roam() sleep()

Rhino
makeNoise() roam() sleep()

Animal Classes with no inheritance



Animal Classes with inheritance

© Kenneth M. Anderson, 2011

Code Metrics

- Indeed, I coded these two examples and discovered
  - without inheritance: 9 files, 200 lines of code
  - with inheritance: 13 files, 167 lines of code
  - approximately a 15% savings, even for this simple example

© Kenneth M. Anderson, 2011

Inheritance (IV)

- An important aspect of inheritance is **substitutability**
  - Since a subclass can exhibit all of the behavior of its superclass, it can be used anywhere an instance of its superclass is used
    - The textbook describes this as polymorphism

Inheritance (VI)

- Furthermore, subclasses can add additional behaviors that make sense for it and override behaviors provided by the superclass, altering them to suit its needs
  - This is both powerful AND dangerous
    - Why? Stay tuned for the answer…

Polymorphism (I)

- OO programming languages support polymorphism ("many forms")
  - In practice, this allows code
    - to be written **with respect to the root of an inheritance hierarchy**
    - and **function correctly when applied to the root's subclasses**

Polymorphism (II)

- Message Passing vs. Method Invocation
  - With polymorphism, a message ostensibly sent to a superclass, may be handled by a subclass
    - as discussed in lecture 3

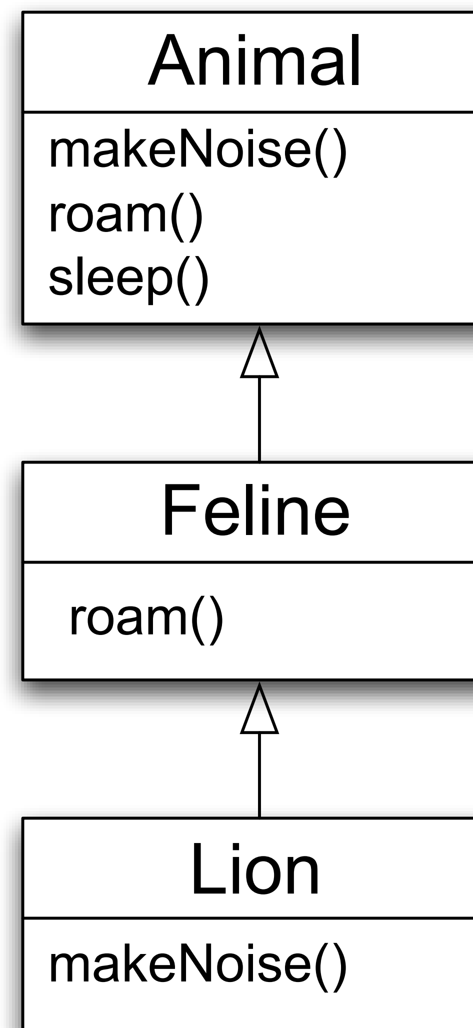
Polymorphism (III)

- Compare this
  - `Animal a = new Animal();`
  - `a.sleep(); // sleep() in Animal called`
- with this
  - `Animal a = new Lion();`
  - `a.sleep(); // sleep() in Lion called`

© Kenneth M. Anderson, 2011

Polymorphism Example

- Without polymorphism, the code on the right only calls methods in Animal
  - Think C++ non-virtual method invocations
- With polymorphism
  - a.roam() invokes Feline.roam()
  - a.makeNoise() invokes Lion.makeNoise()
- A message sent to Animal travels down the hierarchy looking for the "most specific" method body
  - In actuality, method lookup starts with Lion and goes up




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

Why is this important?

- Polymorphism 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();
    }
}
```

© Kenneth M. Anderson, 2011

Importance (II)

- In the previous code
  - we don't care what type of animals are contained in the array
  - we just call sleep() and get the correct behavior for each type of animal

Importance (III)

- Indeed, if a new subclass of animal is created
  - the above code still functions correctly AND
  - it doesn't need to be recompiled
  - with dynamic class loading, if the above code was running in a server, you wouldn't even need to "stop the server"; you could simply load a new subclass and "keep on trucking" ☺
- It only cares about Animal, not its subclasses
  - as long as Animal doesn't change, the addition/removal of Animal subclasses has no impact

Importance (IV)

- We can view a class's public methods as establishing a contract that it and its subclasses promise to keep
  - if we code to the (root) contract, as we did in the previous example, we can create very robust and easy to maintain software systems
  - This perspective is known as design by contract

Importance (IV)

- Earlier, we referred to method overloading as "powerful AND dangerous"
  - The danger comes from the possibility that a subclass may change the behavior of a method such that it no longer follows the contract established by a superclass
    - such a change will break previously abstract and robust code

© Kenneth M. Anderson, 2011

Importance (V)

- Consider what would happen if an Animal subclass overrides the sleep() method to make its instances flee from a predator or eat a meal
  - Our goToSleep() method would no longer succeed in putting all of the Zoo's animals to sleep
- If we could not change the offending subclass, we would have to modify the goToSleep() method to contain special case code to handle it
  - this would break abstraction and seriously degrade the maintainability of that code

© Kenneth M. Anderson, 2011

Polymorphism (IV)

- Finally, polymorphism is supported in arguments to methods and method return types
  - In our goToSleep() method, we passed in a polymorphic argument, namely an array of Animals
    - The code doesn't care if the array contains Animal instances or any of its subclasses

Polymorphism (IV)

- 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
}
```

- When using the createRandomAnimal() method, we don't know ahead of time which instance of an Animal subclass will be returned
    - That's okay as long as we are happy to interact with it via the API provided by the Animal superclass

Abstract Classes (I)

- There are times when you want to make the "design by contract" principle explicit
  - **Abstract classes and Interfaces let you do this**
- An abstract class is simply one which cannot be directly instantiated
  - It is designed from the start to be subclassed
  - It does this by declaring a number of method signatures without providing method implementations for them
    - this sets a contract that each subclass must meet

© Kenneth M. Anderson, 2011

Abstract Classes (II)

- Abstract classes are useful since
  - they allow you to provide code for some methods (enabling code reuse)
  - while still defining an abstract interface that subclasses must implement

© Kenneth M. Anderson, 2011

Abstract Classes (III)

- Zoo example
  - `Animal a = new Lion(); // manipulate Lion via Animal interface`
  - `Animal a = new Animal(); // what Animal is this?`
- Animal, Feline, Pachyderm, and Canine are good candidates for being abstract classes

© Kenneth M. Anderson, 2011

Interfaces

- Interfaces go one step further and only allow the declaration of abstract methods
  - you cannot provide 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

© Kenneth M. Anderson, 2011

Interface Example (I)

- Consider modifying the Animal hierarchy to provide operations related to pets (e.g. play() or takeForWalk())
  - We have several options, all with pros and cons
    - add Pet-related methods to Animal
    - add abstract Pet methods to Animal
    - add Pet methods only in the classes they belong (no explicit contract)

© Kenneth M. Anderson, 2011

Interface Example (II)

- Options continued…
  - make a separate Pet superclass and have pets inherit from both Pet and Animal
  - make a Pet interface and have only pets implement it
    - This often makes the most sense although it hinders code reuse
    - Variation: create Pet interface, but then create Pet helper class that is then composed internally and Pet's delegate if they want the default behavior

Object Identity

- In OO programming languages, all objects have a unique id
  - This id might be its memory location or a unique integer assigned to it when it was created
- This id is used to enable a comparison of two variables to see if they point at the same object
  - See example next slide

```java
public class identity {

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

    public static void main(String[] args) {
        String ken = "Ken Anderson";
        String max = "Max Anderson";
        compare(ken, max);
        ken = max;
        compare(ken, max);
        max = new String("Max Anderson");
        compare(ken, max);
    }

}
```

Not Equal

Identical

Equal

© Kenneth M. Anderson, 2011

Identity in OO A&D (I)

- 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
    - Consider people in an elevator
      - Does the elevator care who pushes its buttons?

© Kenneth M. Anderson, 2011

Identity in OO A&D (II)

- Examples, continued
  - Consider a cargo tracking application
    - Does the system need to monitor every carrot that exists inside a bag? How about each bag of carrots in a crate?
  - Consider a flight between Denver and Chicago
    - What uniquely identifies that flight? The plane? The flight number? The cities? What?

Identity in OO A&D (III)

- When doing analysis, you will confront similar issues
  - you will be searching for uniquely identifiable objects that help you solve your problem

Coming Up Next

- Homework 2 assigned tomorrow, due next Friday
- Lecture 5: Example problem domain and initial OO solution (from book)
  - Read Chapters 3 and 4 of the Textbook
- Lecture 6: Introduction to Design Patterns
  - Read Chapter 5 of the Textbook

© Kenneth M. Anderson, 2011