Object Fundamentals
Part Three

Kenneth M. Anderson
University of Colorado, Boulder
CSCI 4448/6448 — Lecture 4 — 09/04/2008
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

• Delegation
  • HAS-A

• Inheritance
  • IS-A

• Polymorphism
  • message passing
  • polymorphic arguments and return types

• Interfaces
  • Abstract Classes

• Object Identity
• Code Examples
Delegation (I)

- When designing a class, there are three ways to handle an incoming message
  - Handle message by implementing code in a method
  - Let the class’s superclass handle the request via inheritance
  - Delegate the request to another object (delegation)
    - Note: goes hand in hand with composition (not to be confused with aggregation/composition which is a design concept)
      - You compose one object out of others
      - The host object delegates requests to its internal objects
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
Delegation (III)

• Advantages
  • 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!
Inheritance (I)

- Inheritance is a mechanism for sharing (public/protected) features between classes.

- A class defines a type.
  
  - A superclass is a more generic instance of that type.
  
  - A subclass is a more specific instance of that type.
    
    - A subtype typically restricts the legal values of its superclass
      
      - Real Numbers → Integers → Positive Integers
      
      - Component → Container → Control → Button → Checkbox
Inheritance (II)

• 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
• 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 would likely require less code,
    - even though it has more classes than the former.
# Animals (No Inheritance)

<table>
<thead>
<tr>
<th>Animal</th>
<th>makeNoise()</th>
<th>roam()</th>
<th>sleep()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lion</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hippo</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dog</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cat</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Elephant</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wolf</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tiger</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rhino</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Animals (With Inheritance)

- Animal
  - sleep()
  - Feline
    - roam()
  - Pachyderm
    - roam()
  - Canine
    - roam()
  - Rhino
    - makeNoise()
  - Wolf
    - makeNoise()
  - Dog
    - makeNoise()
  - Hippo
    - makeNoise()
  - Tiger
    - makeNoise()
  - Cat
    - makeNoise()
  - Elephant
    - makeNoise()
  - Lion
    - makeNoise()
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
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**
• 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
  - Compare this
    
    ```java
    Animal a = new Animal()
    a.sleep() // sleep() in Animal called
    ```
  - with this
    
    ```java
    Animal a = new Lion()
    a.sleep() // sleep() in Lion called
    ```
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

  makeNoise()
  roam()
  sleep()

Feline

  roam()

Lion

  makeNoise()

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

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

- 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
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

  • Why?
Polymorphism (III)

• 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

```java
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 API provided by the Animal superclass
Abstract Classes/Interfaces

• 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
Abstract Classes, Continued

• 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

• Zoo example
  • Animal a = new Lion(); // manipulate Lion via Animal
  • Animal a = new Animal(); // what Animal is this?

• Animal, Feline, Pachyderm, and Canine are good candidates for being abstract classes
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
Interface Example

• 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)
    • 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
Object Identity

- In OO programming languages, all objects have a unique identifier
  - This identifier might be its memory location or a unique integer 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
  - See example next slide
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); ← Not Equal
        ken = max;
        compare(ken, max); ← Identical
        max = new String("Max Anderson");
        compare(ken, max); ← Equal
    }
}
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?
    - 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 (II)

• When doing analysis, you will confront similar issues

  • you will be searching for uniquely identifiable objects that help you solve your problem
Ken’s Corner (I)

- Big names in OO circles (this list is dreadfully incomplete)
    - One of the “fathers” of OO programming
    - Co-inventor of UML; long time advocate of OO A&D
    - Co-inventor of UML; wrote OO software engineering
    - Co-inventor of UML; developed Object Modeling Technique (OMT)
  - **Martin Fowler** [http://www.martinfowler.com/](http://www.martinfowler.com/)
    - Prolific author on OO topics such as refactoring, patterns, UML, etc.
Ken’s Corner (II)

- Big names in OO circles (this list is still dreadfully incomplete)
    - Inventor of Extreme Programming; popularized test-driven design/JUnit
    - Inventor of the wiki; long time advocate for design patterns
    - Wrote *Design Patterns* with “The Gang of Four (GoF)” which also includes Richard Helm, Ralph Johnson, John Vlissides
  - Many, many more… for instance, the designers of OO languages: Alan Kay (Smalltalk), Bjarne Stroustrup (C++), Guido van Rossum (Python), Yukihiro Matsumoto “Matz” (Ruby), Anders Hejlsberg (C#), Brad Cox (Objective C), Brendan Eich (Javascript), Bertrand Meyer (Eiffel); See [http://en.wikipedia.org/wiki/Object-oriented_programming_language](http://en.wikipedia.org/wiki/Object-oriented_programming_language) for more!
Coming Up Next

• Lecture 5: Great Software
  • Read Chapter 1 of the OO A&D book
• Lecture 6: Give Them What They Want
  • Read Chapter 2 of the OO A&D book