More Inheritance, Abstract Classes, and Interfaces

Recall: Overriding of Methods

class Animal {
    protected String name = "";
    protected String noise = "";
    public setName (String myName) {
        name = myName;
    }
    public void identifySelf ( ) {
        System.out.println("My name is " + name);
    }
    public void perform ( ) {
    }
}

I Method perform( ) in Dog overrides perform( ) in Animal

class Dog extends Animal {
    public Dog ( ) {
        noise = "woof";
    }
    public void perform ( ) {
        identifySelf ( );
        System.out.println("I am a Dog");
        System.out.println(noise);
    }
}

Variable Shadowing

I What happens if Dog also has a field called name?

class Dog extends Animal {
    protected String name = "xxx";
    public Dog ( ) {
        noise = "woof";
    }
    public void perform ( ) {
        identifySelf ( );
        System.out.println("I am a Dog");
        System.out.println(noise);
    }
}

I The name field in Animal is hidden or shadowed
I Within Dog, super.name can be used to access the field in Animal
I When a method is invoked, the actual type of the reference is used (i.e., dynamic binding)
I When a variable is accessed, the declared type of the reference is used (i.e., static binding)
I Try to avoid shadowing

Constructors

I Goal: Modify Animal so that the animalType is specified when the Animal is created
I Need a new constructor in Animal
I Need a modified identifySelf( ) in Animal
I What changes are needed in Dog?

class Animal {
    protected String name = "";
    protected String noise = "";
    private String animalType = "";
    public Animal (String animalType) {
        this.animalType = animalType;
    }
    public setName (String myName) {
        name = myName;
    }
    public void identifySelf ( ) {
        System.out.println("My name is " + name);
        System.out.println("I am a " + animalType);
    }
    public void perform ( ) {
    }
}

Old Dog vs. New Dog

class Dog extends Animal {
    public Dog ( ) {
        noise = "woof";
    }
    public void perform ( ) {
        identifySelf ( );
        System.out.println("I am a Dog");
        System.out.println(noise);
    }
}

I The old Dog constructor starts by calling super( )
I This is now an error since there is no such constructor in Animal

The construction "super(xxx)" calls the constructor in Animal

Constructor Chaining

I Within the same class
I Use the construction this(xxx)
I Arguments are allowed
I Constructors can be overloaded
I Chaining to superclass
I Use the construction super(xxx)
I Arguments are allowed
I Uses constructor in superclass with matching signature
I Without an explicit occurrence of this( ) or super( ), an occurrence of super( ) (with no arguments) is assumed
I Implication: any use of this( ) or super( ) must occur in first statement of constructor
I Note: if no constructor is specified then a no-argument constructor is assumed
Use of `this` and `super` in Java

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>this(xxx)</code></td>
<td>Calls different constructor in current class (must be 1st statement)</td>
</tr>
<tr>
<td><code>this.xxx</code></td>
<td>Accesses a current-class variable</td>
</tr>
<tr>
<td><code>this.method(xxx)</code></td>
<td>Calls a current-class method</td>
</tr>
<tr>
<td><code>super(xxx)</code></td>
<td>Calls a superclass constructor (must be 1st statement)</td>
</tr>
<tr>
<td><code>super.xxx</code></td>
<td>Accesses a superclass variable</td>
</tr>
<tr>
<td><code>super.method(xxx)</code></td>
<td>Calls a superclass method</td>
</tr>
<tr>
<td><code>super.super.xxx</code></td>
<td>Invalid</td>
</tr>
</tbody>
</table>

Abstract Classes

- How do we keep users from defining generic Animals?
  - Make the class abstract

- An abstract class is “incomplete” and thus cannot be instantiated
  - A method can also be abstract (e.g., `perform()`)  
  - A class that inherits (without overriding) or contains an abstract method is also abstract

```
abstract class Animal {
  protected String name = "";
  protected String noise = "";
  private String animalType = "";
  public Animal (String animalType) {
    this.animalType = animalType;
  }
  public setName (String myName) {
    name = myName;
  }
  public void identifySelf  {
    System.out.println("My name is " + name);
    System.out.println("I am a " + animalType);
  }
  abstract public void perform ();
}
```

**final** Methods, Classes, and Variables

- What if we don’t want any subclasses of Animal to mess with `identifySelf()`?
  - Make `identifySelf()` a final method
  - A final method cannot be overridden
  - A final class cannot be extended (e.g., `Integer`, `String` in Java)
  - A final variable cannot be changed (i.e., it’s constant)

```
final public void identifySelf  {
  System.out.println("My name is " + name);
  System.out.println("I am a " + animalType);
}
```

Multiple Inheritance

- Multiple inheritance allows the creation of classes that inherit from more than one superclass

```
public interface Comparable {
  public int compareTo (object o);
}
```

- Not allowed in Java
- But other object-oriented languages allow it (e.g., C++, Lisp(CLOS))
- Java allows only single inheritance (or linear inheritance)
  - Simpler to implement
  - More efficient
  - Less confusing

```
final public void identifySelf  {
  System.out.println("My name is " + name);
  System.out.println("I am a " + animalType);
}
```

Interfaces

- In Java, an interface is a special kind of “class” that has only abstract methods (and constants)
  - The method signatures are known, but no implementations are given
- In Java, a class extends a superclass, but it implements an interface

**Example:**
```
java.lang.Comparable
```
```
public interface Comparable {
  public int compareTo (object o);
}
```
```
A class that implements Comparable must provide a method compareTo (with matching signature)
```

A Kind of Multiple Inheritance

- A class can extend just one superclass
  - Multiple inheritance can cause conflicts
  - Example: Which of 2 inherited methods to use when both have identical signatures?
- But it can implement multiple interfaces
  - Multiple interfaces don’t conflict because there are no implementations
Interfaces Define New Types

- An interface cannot be instantiated (e.g., Comparable c = new Comparable(); is illegal)
- But you can declare a variable using the interface type (e.g., Comparable c = new String("hello"); is legal because the class String implements Comparable)

```java
interface Pet {
    void perform( );
}
class Dog extends Animal implements Pet {
    public Dog ( ) {
        super("Dog");
        noise = "woof";
    }
    public void perform ( ) {
        System.out.print(noise);
    }
    Dog d = new Dog ( );
    System.out.println(d instanceof Pet); // true
    Pet p = d; // OK
    p.perform ( ); // OK
    p.identifySelf ( ); // Compile-time error
}
```

More on Interfaces

- Interface methods
  - Interface methods are implicitly public and abstract
  - No static methods are allowed in interfaces
- Interface constants
  - Interface constants are public, static, and final
  - Can inherit multiple versions of constants
  - Compiler detects this
  - When this occurs, fully qualified names are required

Why Interfaces and Abstract Classes?

Why have both?
- Because an abstract class can include method implementations
- We used this in Animal
  - identifySelf
  - constructor for Animal
- Useful in Shape class in text

```java
abstract class Animal {
    protected String name = "";
    protected String noise = "";
    private String animalType = "";
    public Animal (String animalType) {this.animalType = animalType;}
    public setName (String myName) {
        name = myName;
    }
    abstract public void perform ( );
    public void identifySelf( ) {
        System.out.println("My name is "+ name);
        System.out.println("I am a "+ animalType);
    }
}
```

Aggregation

- Two major mechanisms for code reuse in Object Oriented Programming
  - Inheritance
  - Aggregation
- Aggregation is based on the “has-a” relationship
  - a Car has an Engine
  - an Order has a Customer
  - a Customer has a CreditRecord
- The idea of aggregation is to use objects as parts of other objects
- Example: The programmer who writes the Order class does not need to know implementation details about the Customer class even though the Order class uses a Customer field