Object Oriented Programming Introduction

Steve Ryder
Session 1180
JSR Systems (JSR)
sryder@jsrsys.com

Note: zip files for all lab exercise materials can be found at

www.jsrsys.com

Objectives

- Compare/Contrast OO Programming to Procedural Programming
- Introduction to these Object Oriented concepts:
  - Classes (think Program Load Module++)
  - Objects (think one copy of a Program in memory)
  - Class Data (think Working Storage)
  - Methods (think Performed Procedures or functions())
  - Some COBOL comparisons (from 2002 OO-COBOL session)
- Understand the lifecycle of an object

COBOL vs. OO Comparisons

COBOL: **COBOL Concept Description**

Java: **Java/OO Similar Concept**

++: **What Java/OO adds to Concept**

COBOL: **Load Module/Program**

Java: **Class**

COBOL: **PERFORM**

Java: **method**

++: can pass parameters to method, more like FUNCTION
other programs/classes can call methods in different classes if declared public. public/private gives designer much control over what other classes can see inside a class.

COBOL vs. OO Comparisons (2)

COBOL: **Working Storage, statically linked sub-routine**

Java: **instance variables**

++: (see next)

COBOL: **Working Storge, dynamically loaded sub-routine**

Java: **Class variables**

++: Java can mix both Class variables (called static, just the reverse of our COBOL example, and instance variables (the default).

Shape Shifter Program

- Specifications
  - Shapes on a GUI
    - Square
    - Circle
    - Triangle
  - When user clicks on shape
    - Shape will rotate clockwise 360 degrees
    - An AIF sound file specific to that shape will play

Procedural Design

- Write Important procedures

```
rotate(shapenum)
{
  //make the shape rotate 360 degrees
}
playSound(shapenum)
{
  //use shapeNum to lookup which
  //AIF sound to play, and play it
}
```

Object Oriented Design

- Write a class for each of the shapes

Square	Circle	Triangle

rotate() {
//code to rotate square
}
 	
playSound(){
//code to play AIF
//for a square
}
	rotate() {
	//code to rotate circle
	}
	
	playSound(){
	//code to play AIF
	//for a circle
	}
		rotate() {
		//code to rotate
		// triangle
		}
		
		playSound(){
		//code to play AIF
		//for a triangle
		}

A Specification Change

- Add amoeba shape
- When user clicks on amoeba
  - Shape will rotate
  - An .hif sound file will play

Procedural Design

- Change previously-tested code
  - Rotate procedure will work as-is
  - PlaySound procedure must change

```
playSound(shapenum)
{
   //if the shape is not an amoeba,
        //use shapenum to look up the AIF
   //else
       //play amoeba .hif sound
}
```

Object Oriented Design

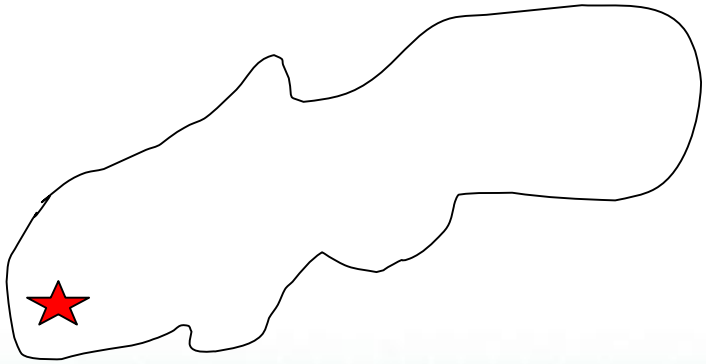
- Write one new class
- No need to touch previously-tested code

Amoeba

rotate()
{
//code to rotate
// amoeba
}
playSound()
{
//code to play .hif
//for a amoeba
}

User Testing – Another Change

- All of the shapes rotated around the center of the shape.
- The amoeba shape, however, should rotate around a point at one end. Like this:



Procedural Design

- Add rotation point arguments to the rotate procedure
- A lot of code was affected

```
rotate(shapenum, xPt, yPt)
{
    //if the shape is not an amoeba
      //calculate the center then rotate
    //else
      //us the xPt and yPt as the
      //rotation point then rotate

}
```

Object Oriented Design

- Change rotate only in the amoeba class

Amoeba

int xPoint;
int yPoint;
rotate()
{
  //code to rotate //amoeba using
  //x and y coordinates
}
playSound()
{
  //code to play .hif
  //for a amoeba
}

Object Oriented Design concepts

- Class (think Program Load Module++)
- Object (think one copy of a Program in memory)
- Method (think Performed Procedure or function)
- Class Data (think Working Storage)

Finding Classes

- Look for nouns in the specification
  - "Customers phone in and place an order for one or more items. The customer service representative creates a new order and adds the items to it. Next the shipping address and payment details are taken so that the order can be shipped and the customer's account charged."
  - Customer
  - Order
  - Item
  - Can you find others?

Objects

- What is the difference between a class and an object?
  - A class is not an object but…
  - It is used to construct them
- A class is a blueprint for an object
  - It explains *how* to make an object of that type
  - Each object made from that class can have its own instance variables

Objects

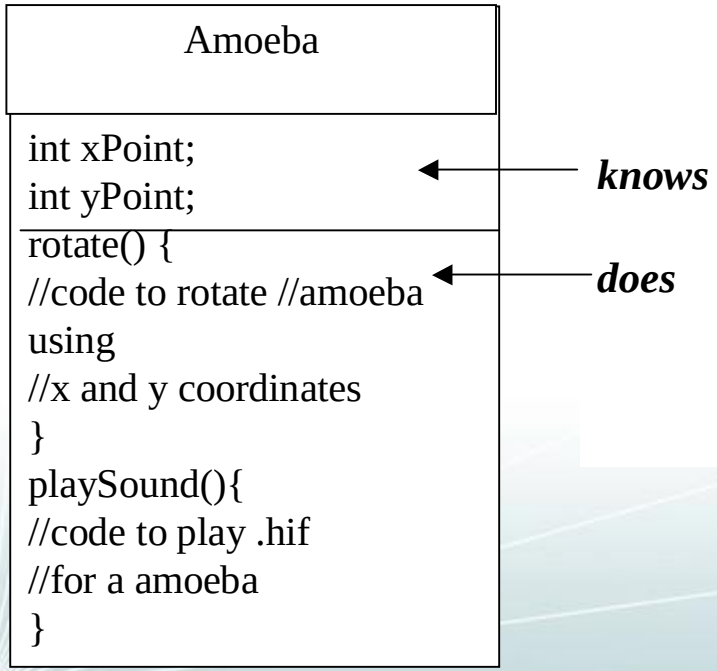
Think of an object like a pack of blank Rolodex™ cards.

- Each card has the same instance variables (blank fields)
- A completed card creates an object instance of a class
- The specific entries on each line represent the object's state (name, phone, address)

Class Data and Methods

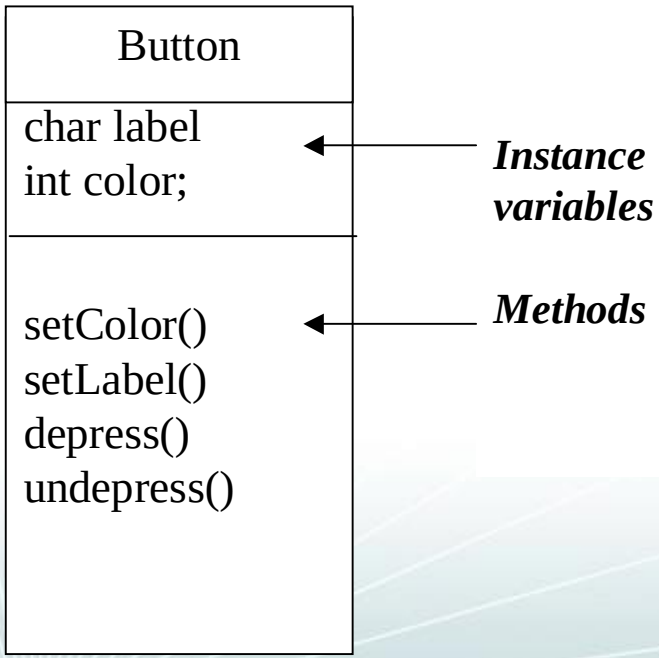
When you design a class, you think about the objects that will be created from that class. You think about:

- Things the object **knows**
- Things the object **does**



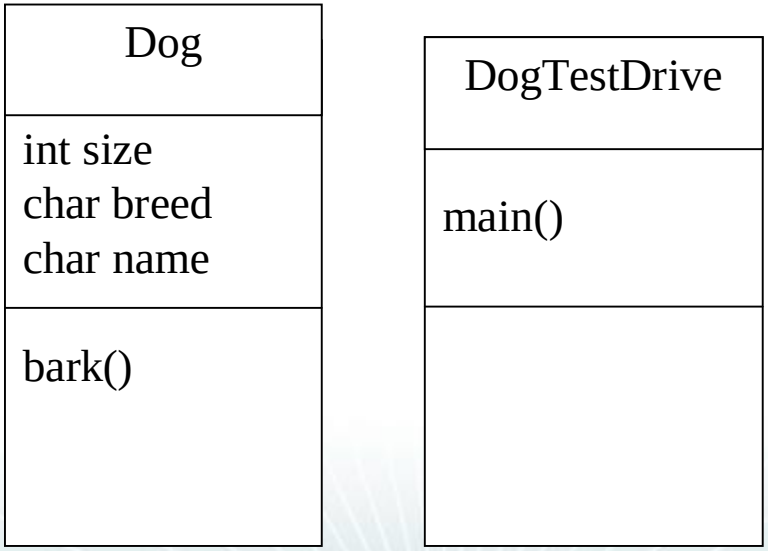
Class Data and Methods

- Things an object knows about itself are called
  - Instance variables
- Things an object can do are called
  - Methods



Your First Object

- What does it take to create and use an object?
  - You need two classes
    - One for the type of object you want to use
    - One to test your new class



Write the Dog class

```
class Dog
{
    int size;
    String breed;
    String name;

    void bark()
    {
        System.out.println("Ruff! Ruff!");
    }
}
```

Write the DogTestDrive class

```
class DogTestDrive
{
    public static void main ( String [] args)
    {
        Dog d = new Dog();
        d.size = 40;
        d.bark();
    }
} //  Need main(String[ ] args) to exec from command line
 //  could just add main to Dog class!
```

The Behavior of an Object

- Instance variables affect method behavior
  - Every instance of a particular class has the same methods
  - But, the methods can behave differently based on the value of the instance variables.

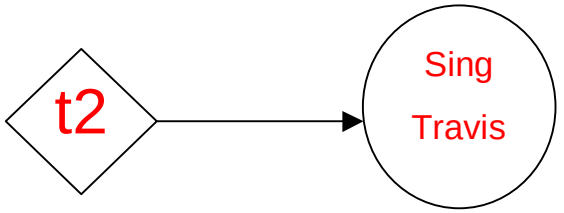
The Song class

- Two instance variables: title and artist.
- Methods to set the title and artist
- A method to play a song

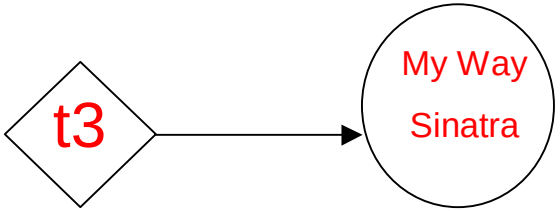
Song
String title=“ “; String artist
setTitle() setArtist() play()

The Song class

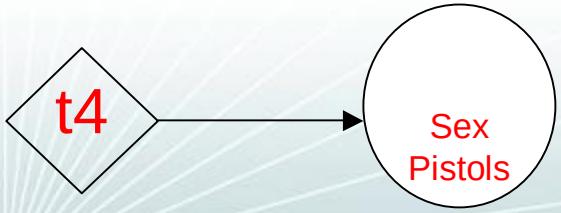
```
Song t2 = new Song();
t2.setArtist("Travis");
t2.setTitle("Sing");
```




```
Song t3 = new Song();
t3.setArtist("Sinatra");
t3.setTitle("My Way");
```




```
Song t4 = new Song();
t4.setArtist("Sex Pistols");
```



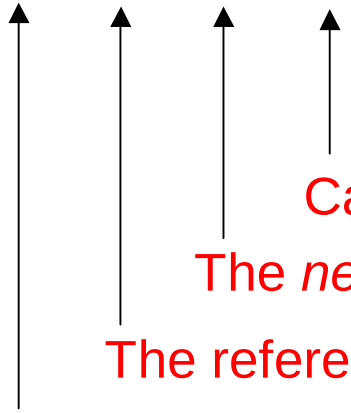
The Lifecycle of an Object

- Creating objects
- Using objects
- Cleaning up unused objects

Creating an object

- This statement initiates a reference to a new object and calls the constructor.

```
Dog d = new Dog();
```



Calls the constructor method of the Dog class

The *new* operator allocates memory for the object

The reference to the new object

Defines the type of reference

Constructors

- A special method defined in the class.
  - Initializes the state of an object
  - Makes sure the new object is ready for use
- Every class has a default constructor that takes no arguments
- You can also provide your own constructors
  - There can be many as long as each is differentiated by the number and type of arguments
  - Constructors with arguments are called with statements like this:
    - Dog d = new Dog(name, size);
    - Dog d = new Dog(breed, name, size);

Using an object

- The Dot Operator
  - The dot operator gives you access to an object's state and behavior
    - Make a new object

      Dog d = new Dog();
    - Call one of the object's methods

      d.bark();
    - Set one of the object's instance variables

      d.size = 40;

Revisiting the Objectives Part A

- Compare/Contrast OO Programming to Procedural Programming
  - Add/change features without touching tested code

Revisiting the Objectives Part A

- Introduction to these Object Oriented concepts:
  - Classes
    - Look for nouns in specification
    - The blueprint for an object
  - Objects
    - The realization of a class
  - Class Variables
    - Things an object knows
  - Methods
    - Things and object does

Objectives for Part B

- Inheritance (ex: Class Dog extends Class Animal)
  behaviors common to many animals would be coded in just Animal, classes such as Dog, Cat, or Tiger are said to inherit the behaviors of Animal (know as the super class).
- Polymorphism (ex: method overloading…)
- Abstract (classes with abstract methods can not be instantiated, only classes that extend them AND implement the abstract methods can be instantiated)
- Interfaces (the solution to multiple inheritance)

Inheritance

- COBOL: COPY or INCLUDE
  Java: Inheritance
  ++: Much more powerful
- Don't have to recompile to "Inherit". The "inheritance" happens at run time.

The Circle class

```
class Circle {

  // Data encapsulated by the class
  private SimplePoint center;
  private int radius;

  // Methods that form external interface
  public double circumference() { ... }
  public double area() { ... }
  public SimplePoint getCenter() { return center; }
  public int getRadius() { return radius; }
}
```

The GraphicCircle class inherits..

```
class GraphicCircle extends Circle
{
  public void draw(Graphics g) { ... }
}
```

GraphicCircle extended/inherited…

- GraphicCircle is defined as an extension of the Circle class
- We call it a subclass
- GraphicCircle has all of the functionality of Circle, plus its own additional methods (and data)
- We say that GraphicCircle inherits the functionality of Circle
- We call the act of extending a class “inheritance”

Inheritance

- GraphicCircle is a Circle
- You can use it anywhere a Circle is required
  - public aMethod(Circle c)
- You can treat it just like a Circle when you use it
  - graphicCircle.getRadius()
- Use inheritance when you have an "is a" relationship

Other key relationships

- "is a" -> inheritance
- "has a" -> data member (e.g. Circle has a SimplePoint, its center) - containment
- "uses" class A uses class B if:
  - a method of A sends a message to an object of class B
  - a method of A creates, receives or returns objects of class B
  - try to minimize the number of classes that use each other

Limits to Subclassing

- How many levels deep can you go when designing subclasses?
  - Most Java API inheritance hierarchies are wide but not deep.
  - Most stay within one to two levels deep.
  - It's good practice, generally, to keep your hierarchy shallow but there is no hard limit that you are likely to encounter.

Method Overriding

If you are unable to change the code for a given class, yet you need to change how it works, you can extend a class and override the method with new, better code.

Do Not Extend…

There are three things that can prevent a class from being extended, or subclassed:

1. There is no *public* declaration.
2. The class has the a *final* access modifier.
3. The class has only *private* constructors.

Why use *final*?

- Make a class final only if you need the security of knowing that *all* methods will work as originally written.
- Make a method final if you want to protect only certain methods within a class.

Rules for Overriding

- When overriding a method from a superclass, you
  are, in effect, agreeing to a contract.
- Here are the rules for overriding a method:
  - Arguments and return types must be the same.
  - Access levels on the subclass must be equal to or more lenient than the superclass.

Overloading

- Overloading is having two methods with the same name but different arguments.
- Overloaded methods have great flexibility:
  1. Return types can be different as long as the arguments are different types.
  2. The return type can not be the only thing changed.
  3. You can vary the access levels in any manner.

Review

- Try to keep class hierarchies one to two levels deep.
- Method Overriding can be used as a do-over when you can not change existing code.
- You can not extend a class that has no public declaration, is declared final, or has private constructors.
- Use final to secure a class when you don't want any of the class to change.
- Use final to secure a method when you only want certain methods to remain unchanged.
- Overriding = agreeing to the superclass' contract. Arguments and return types must be the same. Access levels must be the same or less restrictive.
- Overloading = two or more methods with same name but different arguments (in type or number) and/or return types.

Abstract and Interfaces

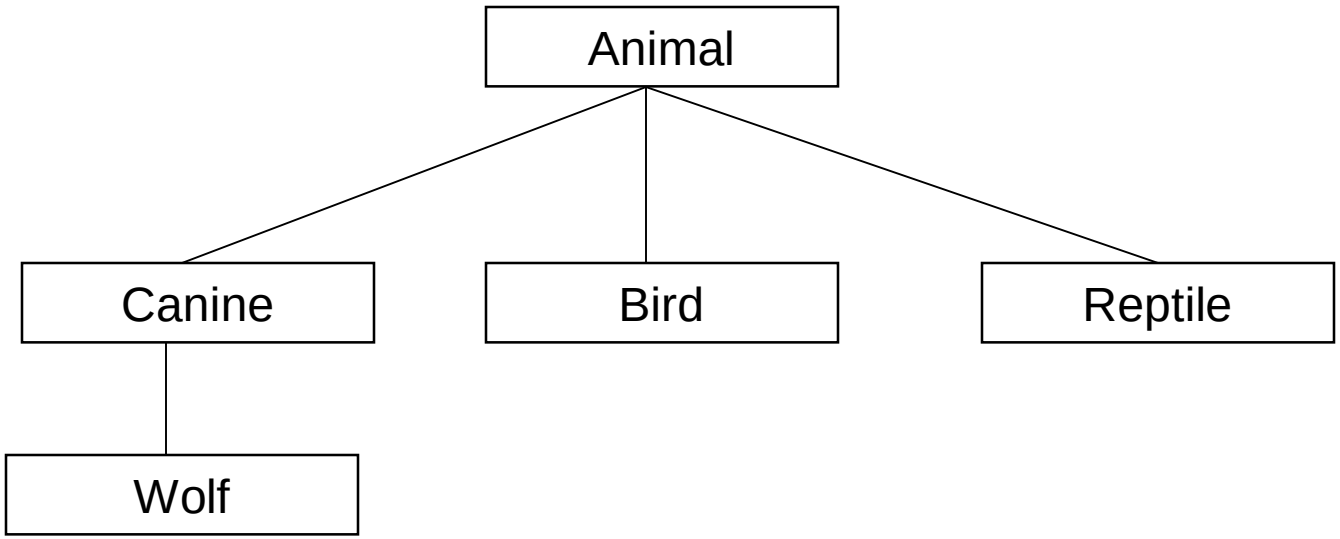
- Abstract classes
- Abstract methods
- Object class
- Interfaces

Abstract Classes

- Keep duplicate code to a minimum.
- Override generic methods.
- Flexible because of Animal subtypes that can be designed in the future and used in any method expecting an Animal object as an argument.
- Creates a common protocol for all animals that are related to the Animal superclass.

Abstract Classes

Sample Animal class hierarchy



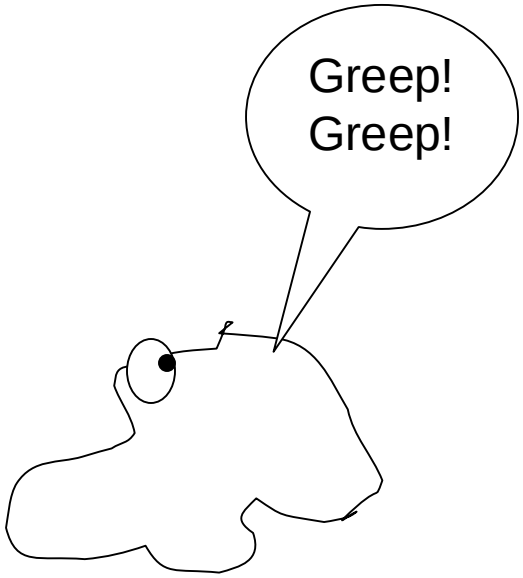
Abstract Classes

Given the class design on the previous slide, the following declarations are valid:

```
Animal aBird = new Bird();
Canine aWolf = new Wolf();
Wolf aWolf = new Wolf();
```

But what about this?

```
Animal anim = new Animal();
```



What would an Animal object look like?

Abstract Classes

- The Animal class is necessary for the inheritance and polymorphism we've been covering. However …
  - Programmers should only be able to instantiate the more concrete subclasses like Wolf because those have shapes, sizes, and behaviors that are well-defined.
- To stop a class from being instantiated, make the class abstract.

```
abstract class Animal

abstract class Canine extends Animal
```

Abstract Methods

- An abstract method must be overridden.
- An abstract method has no body.

```
public abstract void eat();
```

- If you declare a method as abstract, you must declare the class abstract as well.

Abstract Methods

- What can an abstract method be used for?
  - The point of an abstract method is that even without any actual code, you still have defined part of the protocol for a group of subclasses.

Abstract Methods

- What if there are two abstract classes in the hierarchy?
  - A subclass can ‘pass the buck.’
  - If Animal and Canine are both abstract, the first concrete class to extend Canine must implement all abstract methods from both Animal and Canine.

Review

- Abstract classes and methods are useful for keeping duplicate code to a minimum while maintaining a protocol for a group of classes.
- An abstract class can not be instantiated. This forces the programmer to instantiate only the more specific (or concrete) subclasses.
- Abstract methods define the behaviors that all subclasses must have. Each subclass has its own unique way to implement the behaviors.
- The first concrete class in the hierarchy (Wolf from Canine and Animal) must implement all abstract methods from both Canine and Animal.

The Parent of all Classes

class Object

- Every class in Java extends the Object class
- Any class that does not explicitly extend another class implicitly extends Object.

The Dot Operator

- The Dot operator (.) gives you access to an object's state and behavior

```
//Make a new Object
Dog d = new Dog();
//Call the Dog's bark method
d.bark();
//Set the size of the Dog
d.size = 40;
```

Object Class Methods

Two methods available to every object

1. equals(Object o)
2. getClass()

equals()

- Tests if one object is equal to another object

```
Object object1 = new Object;
Object object2 = new Object;
if object1.equals(object2)
{
        System.out.println(“True”);
}
else
{
        System.out.println(“False”);
}
```

getClass()

- Returns the class from which a particular object was instantiated

```
Cat c = new Cat();

System.out.println(c.getClass());
```

Displays “Cat”

Review

- All objects that do not explicitly extend another class implicitly extend the Object class
- There are a number of useful methods in the Object class that can be used with any object -- equals() and getClass() are a few examples

Pet Shop Program: Intro to Interfaces

- What if the Dog class that was written for any type of dog was needed as a pet in another program?
- The Dog class would need new pet-oriented methods such as play(), sit(), rollover(), etc..
- Let's review three design options to make this happen…

Pet Shop – Design Option 1

- Put pet methods in Animal class
- Pros
  - All Animals instantly inherit pet behaviors
  - We won't have to touch existing Animal subclasses
  - Any Animal subclass created in the future will get the pet methods
  - Any program wanting to treat animals as pets can use the Animal class as a polymorphic argument or return type
- Cons
  - ALL animals inherit pet behaviors even lions, tigers, and bears – oh, my!
  - There are sure to be changes required to the subclasses like Dog and Cat because they would implement pet behaviors very differently

Pet Shop – Design Option 2

- Put pet methods in the Animal class but make the methods abstract, forcing the subclasses to override them
- Pros
  - All the benefits of option1 are realized plus there would be no unwanted animals with pet attributes
  - The abstract methods that must be overridden can be empty
- Cons
  - Every subclass of Animal would have to have pet methods even if they aren't needed
  - The existence of Pet methods in the subclasses would be misleading as pet behaviors would be expected from those methods

Pet Shop – Design Option 3

- Put the pet methods only in the classes where they belong
- Pros
  - The pet methods are only where they belong.
- Cons
  - There is no way for other programmers to know what the protocol for establishing or using pet behaviors and no way for the compiler to make sure pet-like methods are implemented correctly
  - The Animal class could not be used as the polymorphic type because the compiler will not let you call a pet method on an Animal reference

Pet Shop – Best Design

- Create two superclasses: Animal and Pet
- Give the Pet class all of the Pet methods
- Have subclasses that should use Pet methods extend both the Animal and Pet classes

Interfaces

- Java provides a tool called an interface because you can not extend two classes
- An interface is a class with the keyword interface as part of the class declaration
- In an interface, all methods are abstract
- All subclasses (of the interface) must implement the interface's methods

Interfaces

- To define an interface

  public interface Pet { … }

  To implement an interface

  public class Dog extends Animal implements Pet { … }

Interfaces

- Interfaces are extremely flexible because…
  - You can use interfaces instead of concrete subclasses as arguments and return types
  - The classes that implement an interface can come from any inheritance tree. This allows you to treat an object by the role it plays and not the class type used to instantiate it
  - A class can implement multiple interfaces

Review

- You can not extend two classes in Java
- An interface allows multiple inheritance without the complications of Multiple Inheritance
- An interface has all abstract methods
- A class can inherit multiple interfaces