Classes, Interfaces, Inheritance

CSE 413, Autumn 2002

Programming Languages

http://www.cs.washington.edu/education/courses/413/02au/

Readings and References

- Reading in *Core Java Volume 1*

  Chapter 4, Objects and Classes, sections *Intro, Using Existing Classes, Building Your Own Classes, Method Parameters,* and *Object Construction*

  Chapter 5, Inheritance, sections *Extending Classes* and *Object : The cosmic superclass*

  Chapter 6, Interfaces and Inner Classes, section *Interfaces*

- Reading in Java tutorial
  - » Object Basics and Simple Data Objects
  - » Classes and Inheritance
  - » Interfaces and Packages
  - » http://java.sun.com/docs/books/tutorial/java/TOC.html#concepts

Recall: Objects and Classes

- A class is a definition of a *type of thing*
  - » The class definition is where we find a description of how things of this type behave.
- An object is a *particular thing*
  - » There can be many objects of a given class. An object is an *instance* of a class.
  - » All objects of a given class exhibit the same behavior.

Class Concepts

- Class definitions have two important components:
  - » state
  - » behavior or interface
- State is expressed using fields in the class definition
- Behavior is expressed using methods
- Together, fields and methods are called class members

Java Class Syntax

- Basic form:

```
[modifiers] class name { [body]  }
```

- Classes often written like:

```
class myClass {
  // public features
  // private features
}
```

- Be consistent, not religious about structure

Example : java.util.Random

```
package java.util;
public class Random implements java.io.Serializable {
   static final long serialVersionUID = 3905348978240129619L;
   private long seed;
   private final static long multiplier = 0x5DEECE66DL;
   private final static long addend = 0xBL;
   private final static long mask = (1L << 48) - 1;
   public Random() {...}
   public Random(long seed) {...}
   synchronized public void setSeed(long seed) {...}
   synchronized protected int next(int bits) {...}
   private static final int BITS_PER_BYTE = 8;
   private static final int BYTES_PER_INT = 4;
   public void nextBytes(byte[] bytes) {...}
   public int nextInt() {...}
   public int nextInt(int n) {...}
   public long nextLong() {...}
   public boolean nextBoolean() {...}
   public float nextFloat() {...}
   public double nextDouble() {...}
   private double nextNextGaussian;
   private boolean haveNextNextGaussian = false;
   synchronized public double nextGaussian() {...}
}
```

Instantiate - create an object

- Once we create a class definition using an editor and the compiler, we can *instantiate* it with the "**`new`**" operator
  - » to *instantiate* means to create objects based on the class definition
  - » `Oval moon = new Oval(100,100,20,20,Color.gray,true);`
- We can then manipulate these objects to do the work that needs to be done

Constructors

- A constructor is used to create a new object of a particular class
- Constructors are special methods that get called with the **`new`** operator

```
Dog rover = new Dog(10);
```

- The name of a constructor is the same as the name of the class
  - » in this case **`Dog(double rate)`** is a constructor for the class **`Dog`**
- You can think of the constructor as a method that initializes everything according to what the caller has specified, using whatever default values might be appropriate
- If you don't supply any constructor, the compiler inserts a simple constructor for you. This constructor takes no arguments, and simply calls the superclass constructor.

Multiple Constructors

- There are often several constructors for any one class
- They all have the same name (the name of the class)
- They must differ in their parameter lists
  - » the compiler can tell which constructor you mean by looking at the list of arguments you supply when you call the constructor

```
Rectangle deadTree;
Rectangle liveTrunk;

deadTree=new Rectangle(150,150,10,50);
liveTrunk=new
   Rectangle(200,210,10,50,Color.orange,true);
```

- There is no return value specified for any constructor, because a constructor always fills in the values in a new object

superclass constructor

- The first statement of a constructor is important. It always calls another constructor
  - » Most often, it calls the superclass' no-args constructor
  - » if you don't put in the call, the compiler will do it for you
- You can override this
  - » by calling a superclass constructor yourself using super(...)
  - » by calling another constructor of your class using this(...)

Methods

- A method is a block of statements that can be invoked to perform a particular action
  - » implementing and then calling methods are the way we specify what an object does
- The collection of all the methods defined for a class defines what objects of that class can do
  - » For example, if we define methods **`bark, sleep, eat,`** and **`getRate`** in the Dog class, then all Dog objects created from that class can do all those things.

`Dog.java`

Method parameters

- Some methods know how to implement a little bit of behavior without needing any more information

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

  - » A Dog implemented this way will bark exactly the same way every time this method is called

- But many methods need to know something additional in order to actually perform their task

```
/**
 * Eat some goodies.  There is some weight gain after eating.
 * @param pounds the number of pounds of food provided.
 */
public void eat(int pounds) {
    double coverage = (double)pounds/(double)consumptionRate;
    …
```

  - » We use parameters (arguments) to provide this additional information

Specifying the required parameters

- The method header declares the type and name for each required parameter

- method **`eat`** has one parameter of type **`int`** named **`pounds`**

```
/**
 * Eat some goodies.  There is some weight gain after eating.
 * @param pounds the number of pounds of food provided.
 */
public void eat(int pounds) {
    double coverage = (double)pounds/(double)consumptionRate;
    …
```

  - » note that there is a javadoc comment describing the purpose of the parameter

Parameter declaration

- Declaring the parameter in the parameter list is just like declaring it in the body of the code
  - » The variable **`pounds`** has a type (**`int`**) and it can be used in expressions exactly the way any other variable in the method is used
- You can declare several parameters in the formal parameter list of a method
  - » but try to keep the number down
  - » if there are too many, the users of this method (you and other programmers) will have a hard time keeping straight just which parameter is which

Examples from class java.lang.String

- **`toLowerCase()`**

  Converts all of the characters in this String to lower case using the rules of the default locale

- **`startsWith(String prefix)`**

  Tests if this string starts with the specified prefix

- **`substring(int beginIndex, int endIndex)`**

  Returns a new string that is a substring of this string

- **`regionMatches(int toffset, String other, int ooffset, int len)`**

  Tests if two string regions are equal

Parameter variables in body of method

```
public String substring(int beginIndex, int endIndex) {
  if (beginIndex < 0) {
    throw new StringIndexOutOfBoundsException(beginIndex);
  }
  if (endIndex > count) {
        throw new StringIndexOutOfBoundsException(endIndex);
  }
  if (beginIndex > endIndex) {
        throw new StringIndexOutOfBoundsException(endIndex-beginIndex);
  }
  return ((beginIndex == 0) && (endIndex == count)) ? this :
        new String(offset+beginIndex, endIndex-beginIndex, value);
}
```

Supplying an actual value

- The actual values don't have to be literals like 2 or 14
- You can supply a variable name in the call, and the current value of the variable will be provided to the method

```
int currentFoodAmount = 4;
Dog jack = new Dog(2);
jack.eat(currentFoodAmount);
currentFoodAmount = 20;
jack.eat(currentFoodAmount);
```

- In this example, the method **`eat`** executes twice, once with **`pounds`** equal to 4, and then again with **`pounds`** equal to 20
- Notice that the method always associates the value with the name **`pounds`**, even though the caller might be using something else

Actual arguments can be expressions

- You can calculate the value to be passed right in the call to the method if that is appropriate
  - » Recall: **`substring(int beginIndex, int endIndex)`**

```
int beginIndex = 0;
String myName = "Doug Johnson";
String twoChar = myName.substring(beginIndex, beginIndex+2);
```

  - » **`twoChar`** is now a reference to a String containing "Do"
- If necessary and possible, the compiler will convert the value provided by the caller to the type of the value that was requested by the method in the formal parameter list

Returning a value to the caller

- A method can also return a value to its caller
- For example, there are often "accessor" methods that allow you to ask an object what some part of its current state is

```
public int getX()
public int getWidth()
```

- The word **`int`** in the above examples specifies the type of value that the method returns

```
/**
 * Get current X value.
 * @return the X coordinate
 */
public int getX() {
    return x;
}
```

Method Overloading

- Classes may declare multiple methods with the same name, provided the *signature* is different
- The signature of a method is:
  - » method name
  - » parameter list
  - » throws clause
- For example, **`System.out.println`** is overloaded for many types

```
println( char c);
println( double d);
println(String s);
```

Documentation for methods

- Short, useful description of the purpose of the method.
  - » javadoc takes the first sentence of this description and uses it in the summary part of the documentation page
  - » If there is important background information on how to use the method, it should follow the initial sentence.
- All parameters
  - » use an @param entry for each parameter
- The return value, if any
  - » use an @return tag if appropriate
- Error exceptions
  - » use a @throws tag if appropriate

Abstract the behavior of classes

- We sometimes want to use one or more methods that are available for various objects, even though they are not all of the same class
- Consider the Cat, Dog, and Sparrow
  - » They all have eat(), sleep(), getMealSize(), and a voice of some sort
- So we can promise that:
  - » We don't know exactly what kind of an animal it is, but we do know that it can eat, sleep, make a noise, and tell you its meal size

`PetList.java`

Interface

- Java has a nice mechanism for this
  - » an interface
- You can say that any class that claims to be an Animal will guarantee that it has methods for all the things that any Animal must do
- The definition of the interface shows exactly what the methods must look like
  - » the actual implementation is not in the interface

public interface Animal

```
/**
 * This interface specifies the behavior that a class
 * must implement in order to be considered a real Animal.
 *
 */
public interface Animal {
    /**
     * Provide this animal with a way to rest when weary.
     */
     public void sleep();
     /**
     * Eat some goodies.  There is some weight gain after eating.
     * @param pounds the number of pounds of food provided.
     */
     public void eat(double pounds);
     /**
      * get the meal size defined for this animal.
      * @return meal size in pounds
      */
     public double getMealSize();
     /**
     * Provide this animal with a voice.
     */
     public void noise();
}
```

using an interface in a class definition

- Each of the classes that wants to be considered an Animal must say so at the very beginning of the class definition

```
public class Dog implements Animal {...
public class Cat implements Animal {...
public class Sparrow implements Animal {...
```

- You are telling the compiler that this class guarantees that it will implement all the methods that are required in the interface

conform to expectations ...

- Each of the animal classes uses the same method name when they make their noise

```
public class Dog implements Animal {
...
  /**
   * Provide this animal with a voice.
   */
   public void noise() {
       System.out.println(name+" : Woof! Woof!");
   }


public class Cat implements Animal {
...
  /**
   * Provide this animal with a voice.
   */
   public void noise() {
       System.out.println(name+" : Meow! Meow!");
   }
```

using the Animal interface in PetList

- Now we know that all of the animals will satisfy the Animal interface, no matter what kind of object they are
- So PetList can guarantee to the compiler that the objects that it is dealing with are Animals, no matter what else they might be
  - » consequently, there is a known set of methods available for each of the objects, no matter what class was used to build it

Cast to Animal

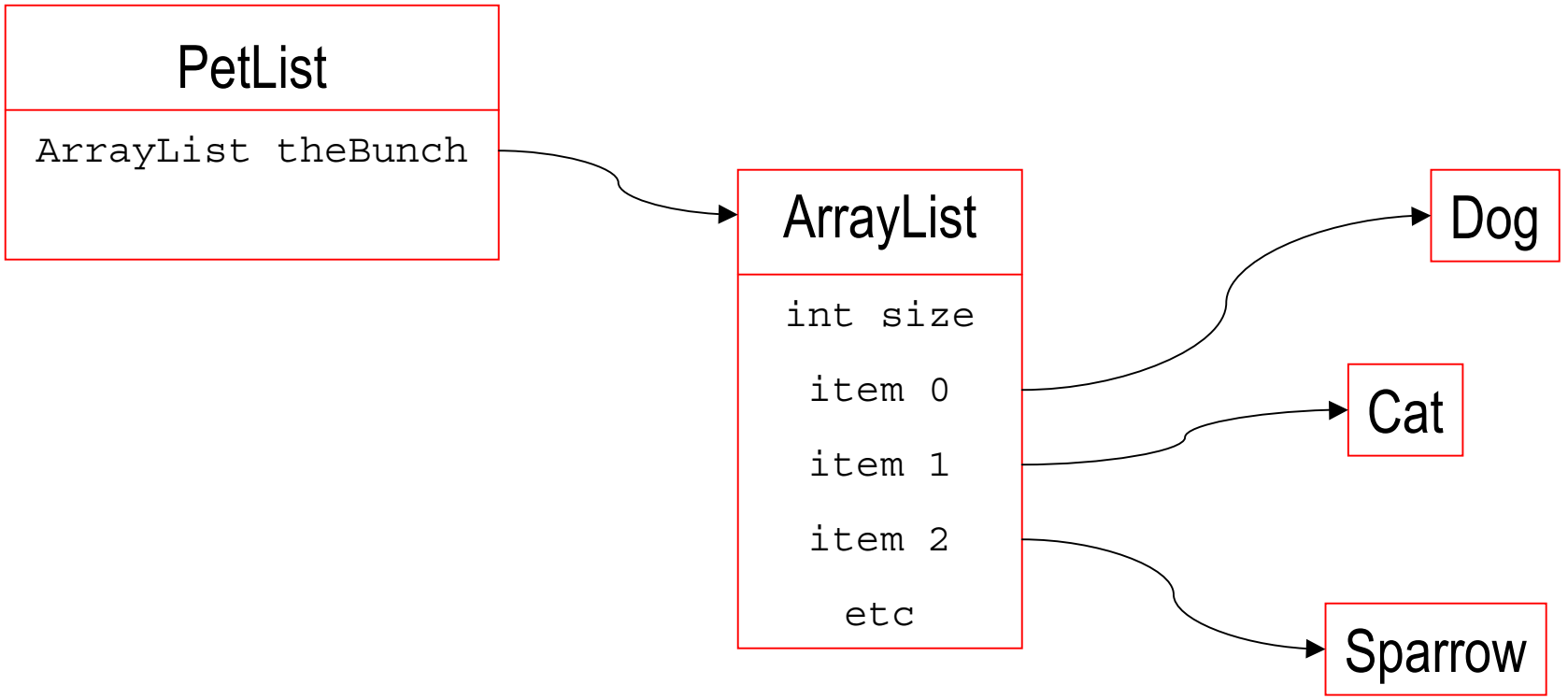
- Tell the compiler that the ArrayList contains objects that are Animals

```
public void dine() {
    for (int i=0; i<theBunch.size(); i++) {
        Animal pet = (Animal)theBunch.get(i);
        pet.eat(2*pet.getMealSize());
    }
}
```

Relationships between classes

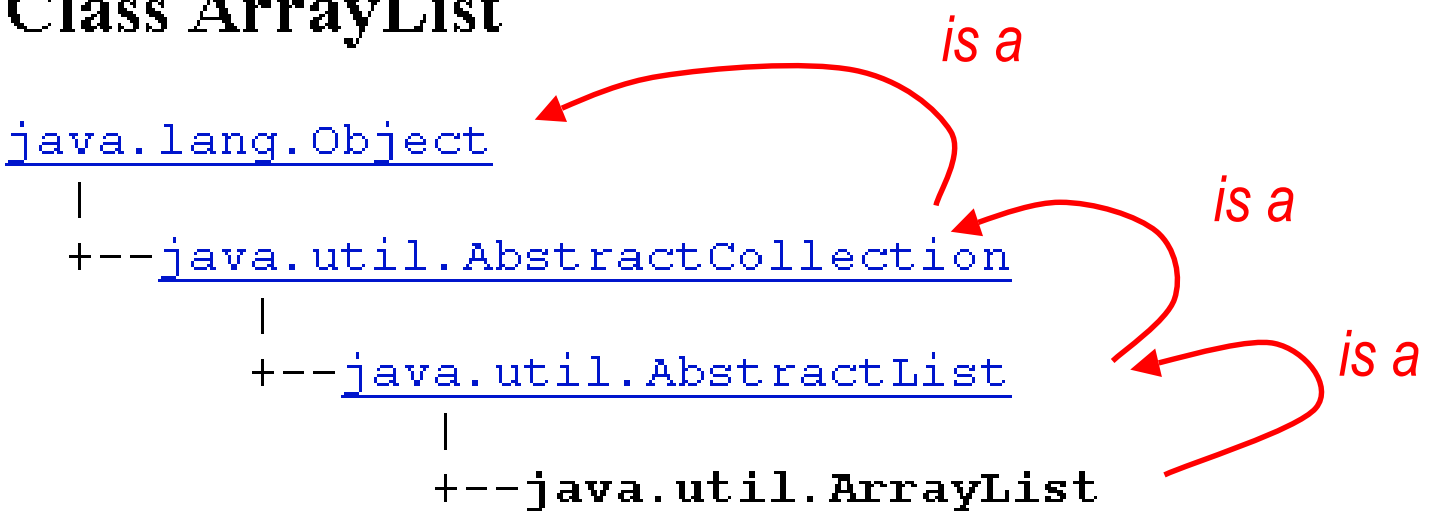
- Classes can be related via composition
  - » This is often referred to as the “has-a” relationship
  - » eg, a PetList *has a* list in an ArrayList of Animals
- Classes can also be related via inheritance
  - » This is often referred to as the “is-a” relationship
  - » eg, an ArrayList *is an* AbstractList

PetList *has a* list of Animals



java.util

Class ArrayList



All Implemented Interfaces:

Cloneable, Collection, List, Serializable

---

public class **ArrayList**
extends AbstractList
implements List, Cloneable, Serializable

Why use inheritance?

- Code simplification
  - » Avoid doing the same operation in two places
  - » Avoid storing “matching state” in two places
- Code simplification
  - » We can deal with objects based on their common behavior, and don’t need to have special cases for each subtype
- Code simplification
  - » Lots of elegant code has already been written - use it, don’t try to rewrite everything from scratch

Reduce the need for duplicated code

- In our collection of pets:
  - » Dog has getMealSize() and eat(double w) methods
  - » Cat has getMealSize() and eat(double w) methods
  - » and they were implemented exactly the same way
- We can define a class named BasicAnimal that implements these methods once, and then the subclasses can extend it and add their own implementations of other methods if they like

BasicAnimal class

Class Tree Deprecated Index

PREV CLASS NEXT CLASS
SUMMARY: INNER | FIELD | CONSTR | METHOD

Class BasicAnimal

```
java.lang.Object
   |
   +--BasicAnimal
```

All Implemented Interfaces:
Animal

Direct Known Subclasses:
Cat, Dog, Sparrow

public abstract class **BasicAnimal**
extends java.lang.Object
implements Animal

Constructor Summary

	BasicAnimal(java.lang.String theName, double serving, double weight) Create a new BasicAnimal, using supplied parameter values.

Method Summary

void	**eat**(double pounds) Eat some goodies.
double	**getMealSize**() get the meal size defined for this animal.
void	**sleep**() Provide this animal with a way to rest when weary.
java.lang.String	**toString**() print information about this animal.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Methods inherited from interface Animal

noise

Syntax of inheritance

- Specify inheritance relationship using **`extends`**

```
public class Dog extends BasicAnimal { …
```

```
public abstract class BasicAnimal implements Animal {
        …
        public double getMealSize() {
                return mealSize;
        }
}
```

- Dog can use existing BasicAnimal methods if desired

Dog as a subclass of BasicAnimal

Package **Class** Tree De

PREV CLASS NEXT CLASS
SUMMARY: NESTED | FIELD | CONSTR

Class Dog

```
java.lang.Object
   |
   +--BasicAnimal
         |
         +--Dog
```

All Implemented Interfaces:
Animal

public class **Dog**
extends BasicAnimal

Constructor Summary

Dog(java.lang.String theName)
Create a new Dog with default characteristics.

Dog(java.lang.String theName, double serving, double weight)
Create a new Dog, using supplied parameter values.

Method Summary

static void	**main**(java.lang.String[] args) Run this animal through a typical day.
void	**noise**() Provide this animal with an appropriate voice.

Methods inherited from class BasicAnimal

eat, getMealSize, sleep, toString

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Using the superclass constructor

- Constructor of the superclass is called to do much (or all) of the initialization for the subclass

```
public class Dog extends BasicAnimal {
    public Dog(String theName) {
        super(theName,0.5,20);
    }
    public Dog(String theName,double serving,double weight) {
        super(theName,serving,weight);
    }
```

```
public class BasicAnimal implements Animal {
    public BasicAnimal(String theName,double serving,double weight) {
        name = theName;
        mealSize = serving;
        currentWeight = weight;
        System.out.println("Created "+name);
    }
```

this() and super() as constructors

- You can use an alias to call another constructor
  - » `super(...)` to call a superclass constructor
  - » `this(…)` to call another constructor from same class
- The call to the other constructor must be the first line of the constructor
  - » If neither this() nor super() is the first line in a constructor, a call to super() is inserted automatically by the compiler. This call takes no arguments. If the superclass has no constructor that takes no arguments, the class will not compile.

Overriding methods

- Overriding methods is how a subclass refines or extends the behavior of a superclass method
- Manager and Executive classes extend Employee
- How do we specify different behavior for Managers and Executives?
  - » Employee:

    ```
    double pay() {return hours*rate + overtime*(rate+5.00);}
    ```

  - » Manager:

    ```
    double pay() {return hours*rate;}
    ```

  - » Executive:

    ```
    double pay() {return salary + bonus;}
    ```

Overriding methods

```
public class Employee {
   // other stuff
   public float pay() {
       return hours*rate + overtime*(rate+5.00);
   }
}
public class Manager extends Employee {
   // other stuff
   public float pay() {
       return hours*rate;
   }
}
```

instanceof

- Used to test an object for class membership

```
if (bunch.get(i) instanceof Dog) {…}
```

- One way to ensure that a cast will succeed
- Tests for a relationship anywhere along the hierarchy
  - » Also tests whether a class implements an interface
- What class must <classname> represent for the following expression to be true always?

```
if (v instanceof <classname>) { … }
```

`instanceof` example with interface

```
ArrayList onStage = theStage.getActors();
for (int i=0; i<onStage.size(); i++) {
    if (onStage.get(i) instanceof ClickableActor) {
        ClickableActor clickee = (ClickableActor)onStage.get(i);
        if (clickee.intersects(cursor)) {
            clickee.doClickAction(theStage);
            if (clickee == runButton) {
                if (runButton.isEnabled()) {
                    theStage.animate();
                } else {
                    theStage.quitAnimation();
                }
            }
        }
    }
}
```