Object Oriented Programming Part II of II

Steve Ryder
Session 8352
JSR Systems (JSR)
sryder@jsrsys.com

New Terms in this Section

- API
- Access Modifier
- Package
- Constructor

Polymorphism

- Three steps of object declaration and assignment
  1. Declare a reference variable

$$\overbrace{\texttt{Dog myDog}}^{1}\ \overset{3}{=}\ \overbrace{\texttt{new Dog();}}^{2}$$

  2. Create an object
  3. Link the object and the reference

Polymorphism

Power references

- The reference and the object can be different

```
Animal myDog = new Dog();
```

Polymorphic Arrays

- Power references allow you to make polymorphic arrays

```
Animal [ ] animals = new Animal[5];
animals[0] = new Dog();
animals[1] = new Cat();

for (int I = 0; I < animals.length; I ++) {
    animals[i].eat();
    animals[i].roam();
}
```

Arguments/Return Types

You can use polymorphic arguments and return types….

```
class Vet {
    public void giveShot(Animal a) {
        //do vet stuff
        a.makeNoise;
    }
}
```

Arguments/Return Types

You can use polymorphic arguments and return types….

```
class Vet {
    public void giveShot(Animal a) {
        //do vet stuff
        a.makeNoise;
    }
}
```

Any Animal subclass can be passed in as an argument to the Vet class.

Arguments/Return Types

You can use polymorphic arguments and return types….

```
class Vet {
    public void giveShot(Animal a) {
        //do vet stuff
        a.makeNoise;
    }
}
```

Invokes the method of the animal type (subclass) passed in as an argument

Review

- A reference variable can be thought of as a remote control that controls the behavior and state of an object.

```
Dog myDog = new Dog();
```

- You can set the reference variable to a more generic type than the object it controls.

```
Animal a = new Dog();
```

- You can define an argument and/or return type used by a method as a more generic type but pass/return the more specific type.

```
public void giveShot(Animal a)
```

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.

Do-overs a.k.a. 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 “bad” 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.

Serious Polymorphism

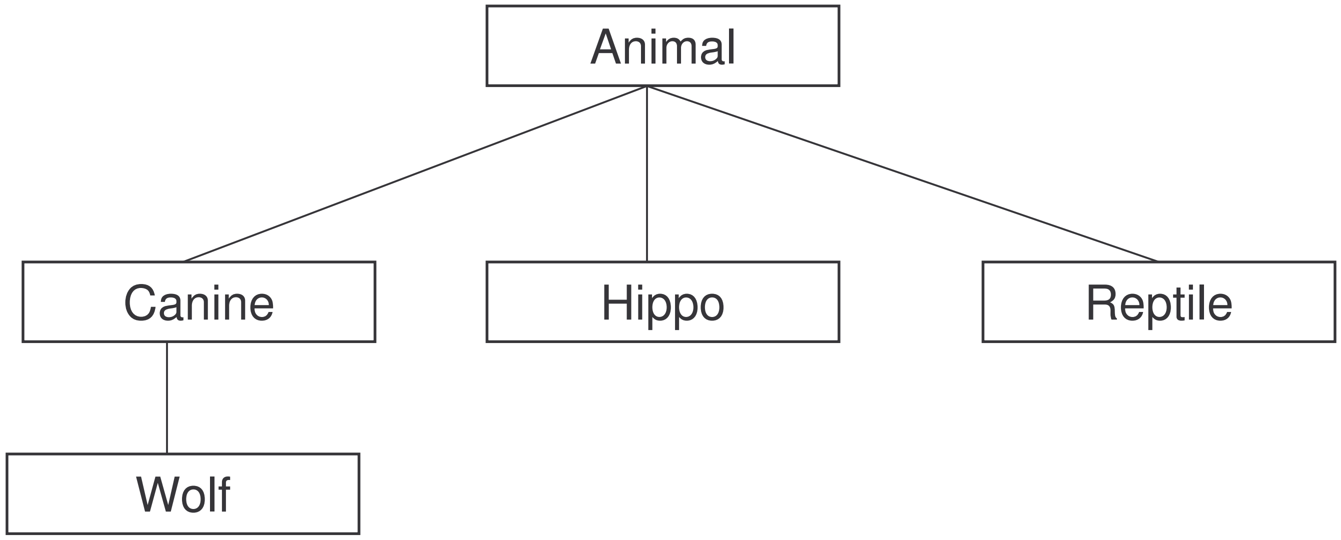
- Abstract classes
- Abstract methods
- Object class
- Arraylist
- Interfaces

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

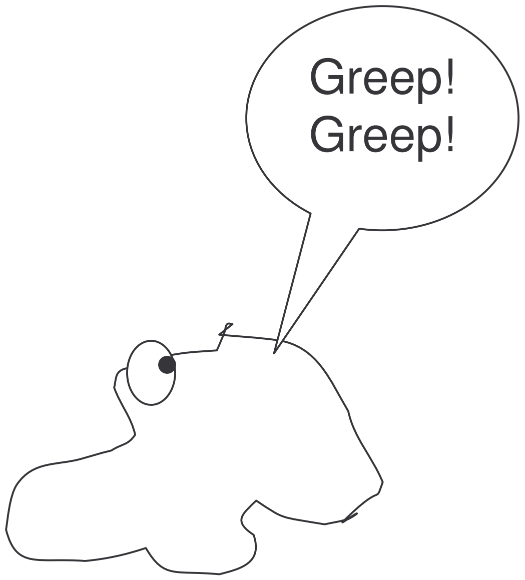
Given the class design on the previous slide, the following declarations are valid:

```
Animal aHippo = new Hippo();
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 or Hippo 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 methods from both Canine and Animal.

The Mother 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

Three methods available to every object.

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

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 “class Cat”

hashCode()

- Returns the hashcode (or unique id from memory) for the object.

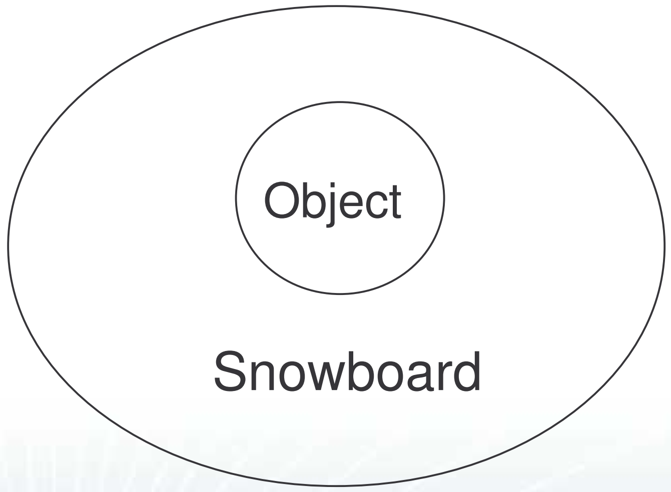
```
Cat c = new Cat();

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

Displays, for example: “8202111”

The Inner Object

- When you instantiate a new object, you get a single object in memory.
- The new object is wrapped around the Object class.

new Snowboard();

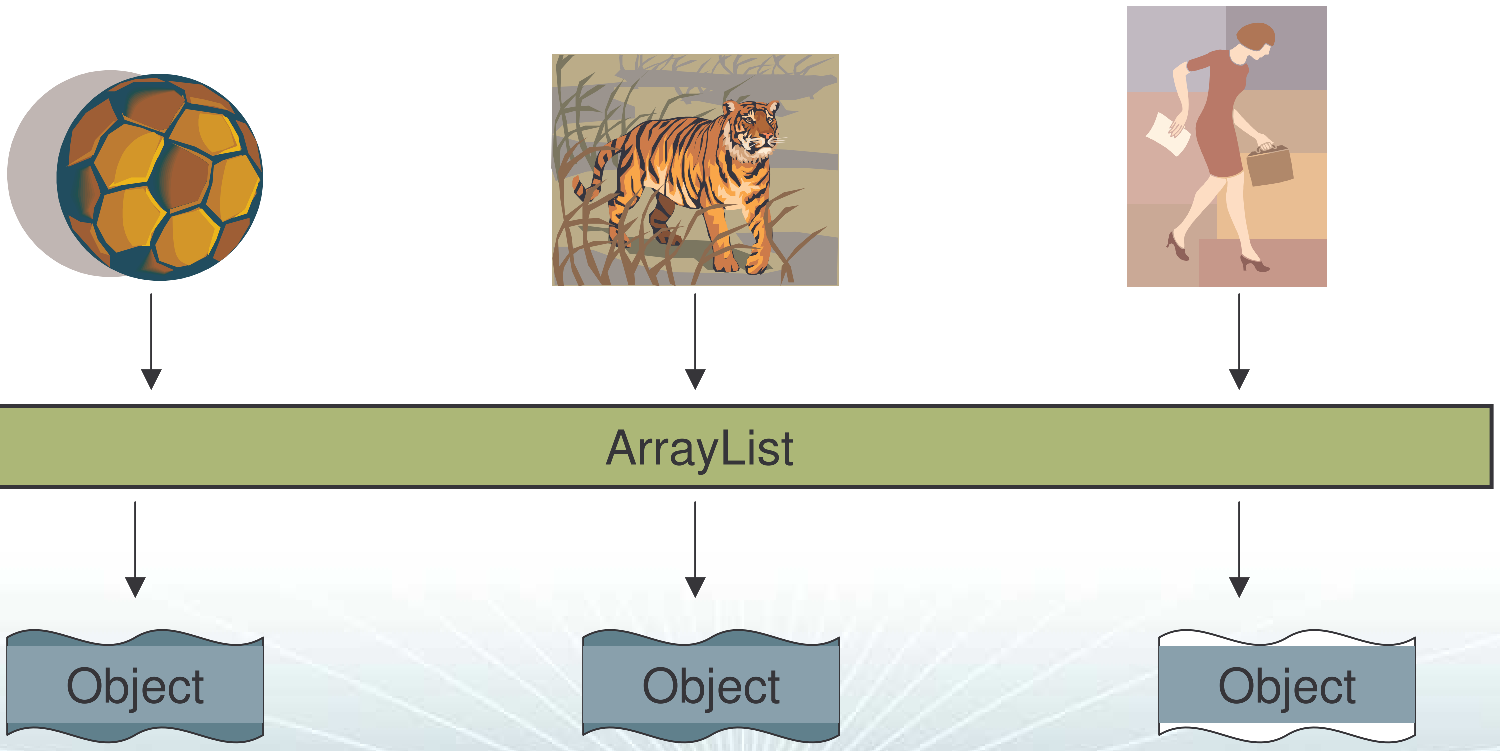


Review

- 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(), getClass(), hashCode() are a few examples.
- Each new object is considered a single object wrapped around an inner Object class.

Objects in an ArrayList

- Objects come out of an ArrayList acting like they are generic objects.



Objects in an ArrayList

```
1  ArrayList al = new ArrayList();
2  Tiger t = new Tiger();
3  al.add(t);
4  //Make Tiger in ArrayList growl
```

1) Instantiate ArrayList object
2) Instantiate Tiger object
3) Add Tiger object to ArrayList

Objects in an ArrayList

```
1 ArrayList al = new ArrayList();
2 Tiger t = new Tiger();
3 al.add(t);
4 //Make Tiger in ArrayList growl
```

Can you call the Tiger’s makeNoise method here?

No. Only the methods in the Object class are available at this point.

Casting an Object Reference Back to its Real Type

```
1 Object o = al.get(index);
2 Tiger t = (Tiger) o;
3 t.makeNoise();
```

1) Get the object from ArrayList
2) Generic object ‘o’ is casted to a Tiger object and assigned to the ‘t’ reference variable
3) The makeNoise() method of the Tiger class is called

Review

- Objects go into an ArrayList as the specified type but come out as generic objects.
- To access the methods of the specific type, you must cast the object to the specific object.

Pet Shop Program

- 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

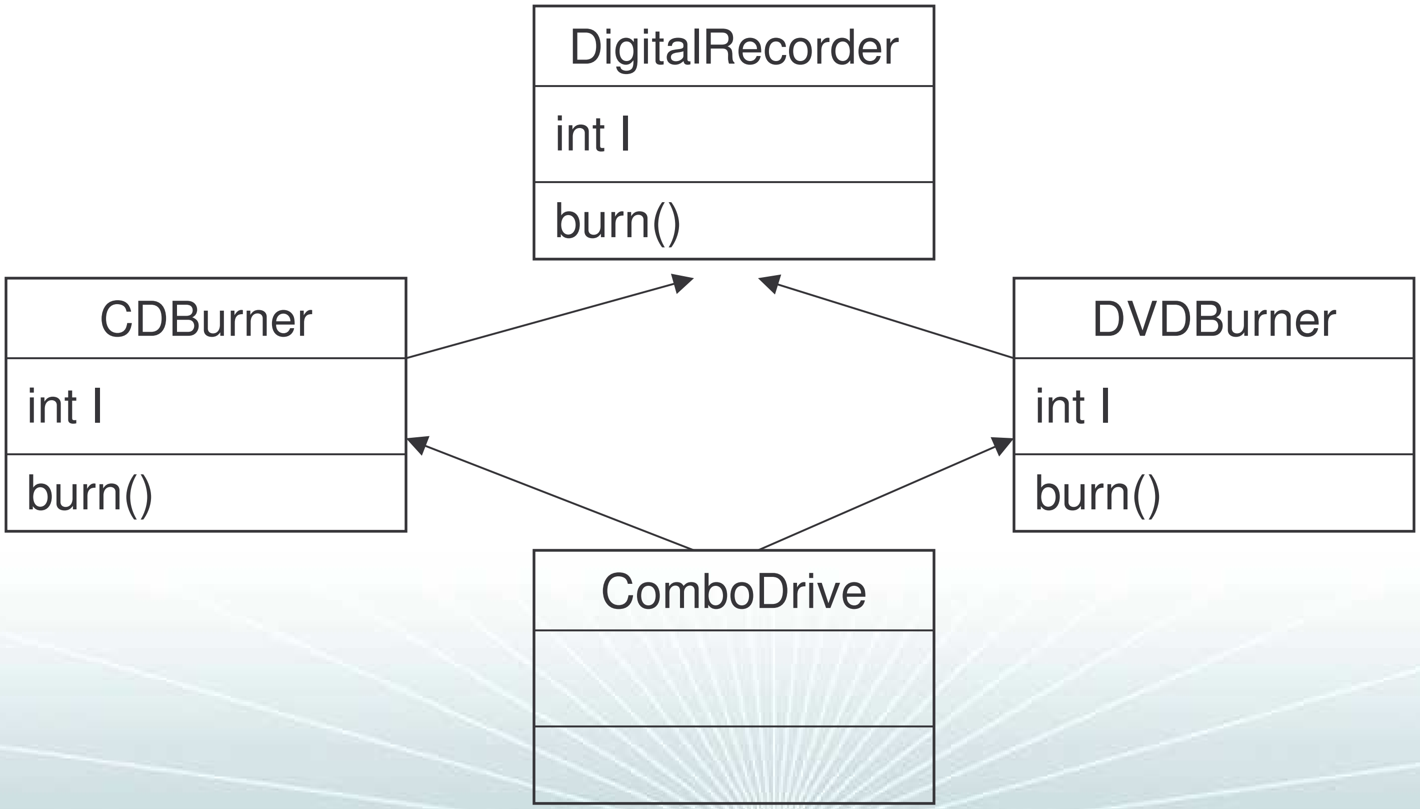
- Put the pet methods only in the classes where they belong.
- Pros
  - The pet methods are only where they belong.
- Cons
  - There is no way for other programmers to know what the protocol for establishing or using pet behaviors and no way for the compiler to make sure pet-like methods are implemented 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.

Deadly Diamond of Death

- Multiple Inheritance is not allowed in Java because of the possible creation of the Deadly Diamond of Death.



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 Deadly Diamond of Death.
- An interface has all abstract methods.
- A class can inherit multiple interfaces.