

8 interfaces and abstract classes

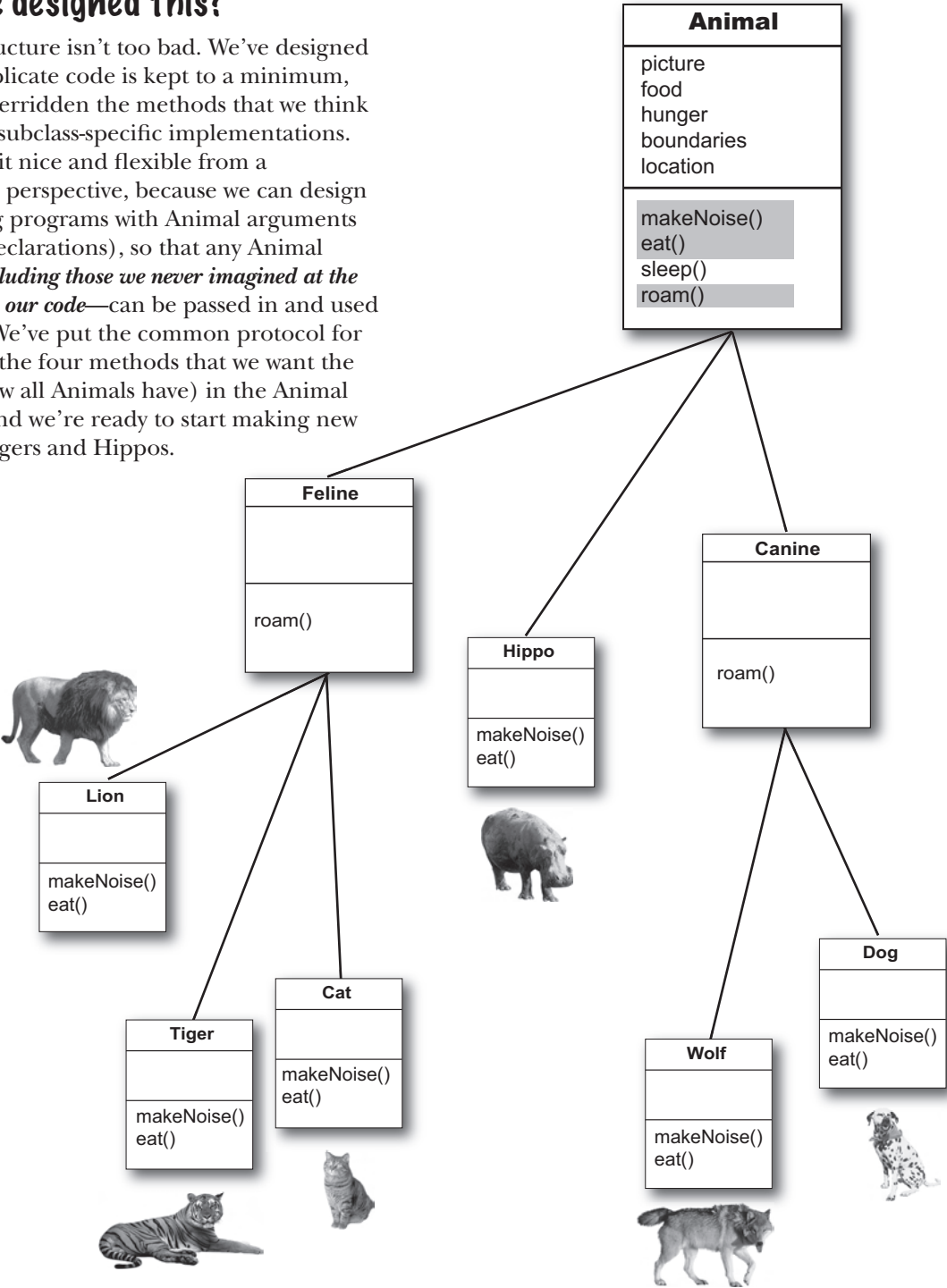
Serious Polymorphism



Inheritance is just the beginning. To exploit polymorphism, we need interfaces (and not the GUI kind). We need to go beyond simple inheritance to a level of flexibility and extensibility you can get only by designing and coding to interface specifications. Some of the coolest parts of Java wouldn't even be possible without interfaces, so even if you don't design with them yourself, you still have to use them. But you'll *want* to design with them. You'll *need* to design with them. ***You'll wonder how you ever lived without them.*** What's an interface? It's a 100% abstract class. What's an abstract class? It's a class that can't be instantiated. What's that good for? You'll see in just a few moments. But if you think about the end of the last chapter, and how we used polymorphic arguments so that a single `Vet` method could take `Animal` subclasses of all types, well, that was just scratching the surface. Interfaces are the ***poly*** in polymorphism. The ***ab*** in abstract. The ***caffeine*** in Java.

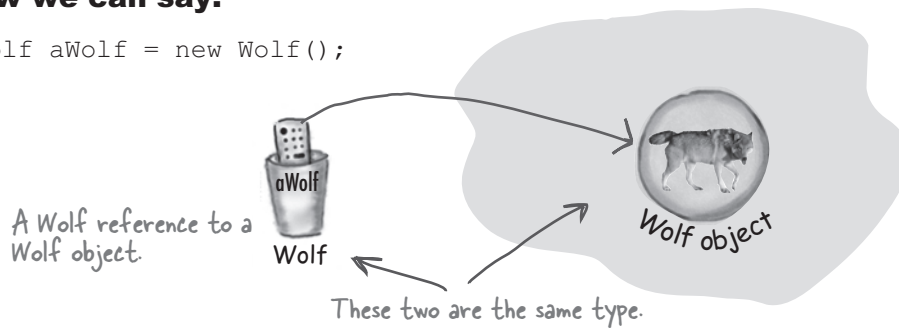
Did we forget about something when we designed this?

The class structure isn't too bad. We've designed it so that duplicate code is kept to a minimum, and we've overridden the methods that we think should have subclass-specific implementations. We've made it nice and flexible from a polymorphic perspective, because we can design Animal-using programs with Animal arguments (and array declarations), so that any Animal subtype—including those we never imagined at the time we wrote our code—can be passed in and used at runtime. We've put the common protocol for all Animals (the four methods that we want the world to know all Animals have) in the Animal superclass, and we're ready to start making new Lions and Tigers and Hippos.



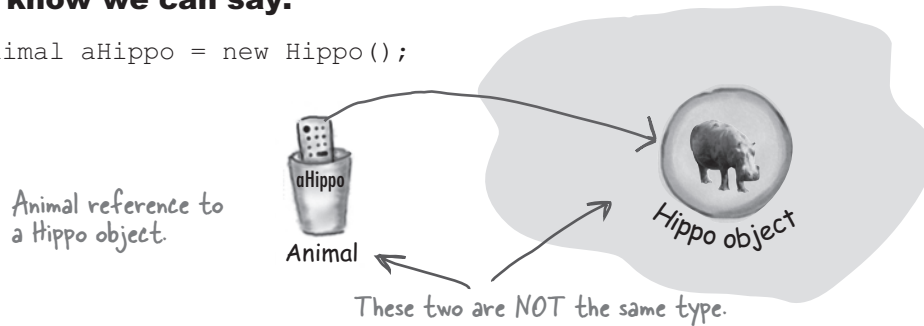
We know we can say:

```
Wolf aWolf = new Wolf();
```



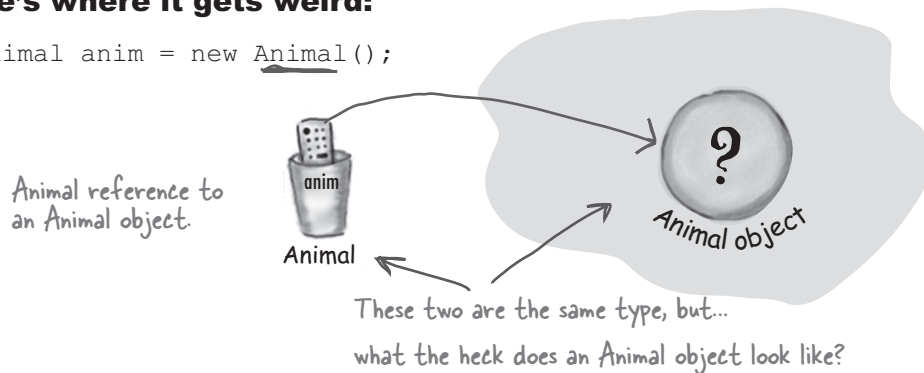
And we know we can say:

```
Animal aHippo = new Hippo();
```



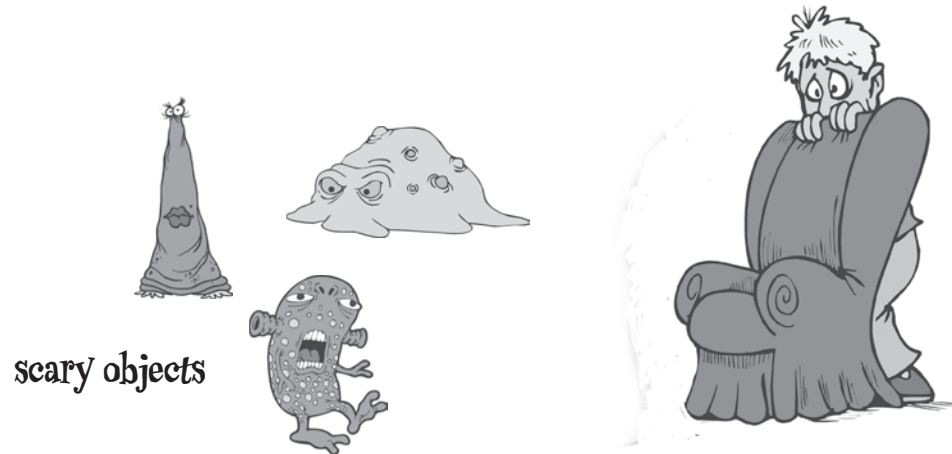
But here's where it gets weird:

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



when objects go bad

What does a new `Animal()` object look like?



What are the instance variable values?

Some classes just should not be instantiated!

It makes sense to create a `Wolf` object or a `Hippo` object or a `Tiger` object, but what exactly is an `Animal` object? What shape is it? What color, size, number of legs...

Trying to create an object of type `Animal` is like a **nightmare *Star Trek*TM transporter accident**. The one where somewhere in the beam-me-up process something bad happened to the buffer.

But how do we deal with this? We *need* an `Animal` class, for inheritance and polymorphism. But we want programmers to instantiate only the less abstract *subclasses* of class `Animal`, not `Animal` itself. We want `Tiger` objects and `Lion` objects, *not `Animal` objects*.

Fortunately, there's a simple way to prevent a class from ever being instantiated. In other words, to stop anyone from saying “**new**” on that type. By marking the class as **abstract**, the compiler will stop any code, anywhere, from ever creating an instance of that type.

You can still use that abstract type as a reference type. In fact, that's a big part of why you have that abstract class in the first place (to use it as a polymorphic argument or return type, or to make a polymorphic array).

When you're designing your class inheritance structure, you have to decide which classes are *abstract* and which are *concrete*. Concrete classes are those that are specific enough to be instantiated. A *concrete* class just means that it's OK to make objects of that type.

Making a class abstract is easy—put the keyword **abstract** before the class declaration:

```
abstract class Canine extends Animal {  
    public void roam() { }  
}
```

The compiler won't let you instantiate an abstract class

An abstract class means that nobody can ever make a new instance of that class. You can still use that abstract class as a declared reference type, for the purpose of polymorphism, but you don't have to worry about somebody making objects of that type. The compiler *guarantees* it.

```
abstract public class Canine extends Animal
{
    public void roam() { }
}
```

```
public class MakeCanine {
```

```
    public void go() {
```

```
        Canine c;
```

```
        c = new Dog();
```

```
        c = new Canine();
```

```
        c.roam();
```

```
    }
```

```
}
```

This is OK, because you can always assign a subclass object to a superclass reference, even if the superclass is abstract.

class Canine is marked abstract, so the compiler will NOT let you do this.

```
File Edit Window Help BeamMeUp
% javac MakeCanine.java
MakeCanine.java:5: Canine is abstract;
cannot be instantiated
    c = new Canine();
          ^
1 error
```

An **abstract class** has virtually* no use, no value, no purpose in life, unless it is **extended**.

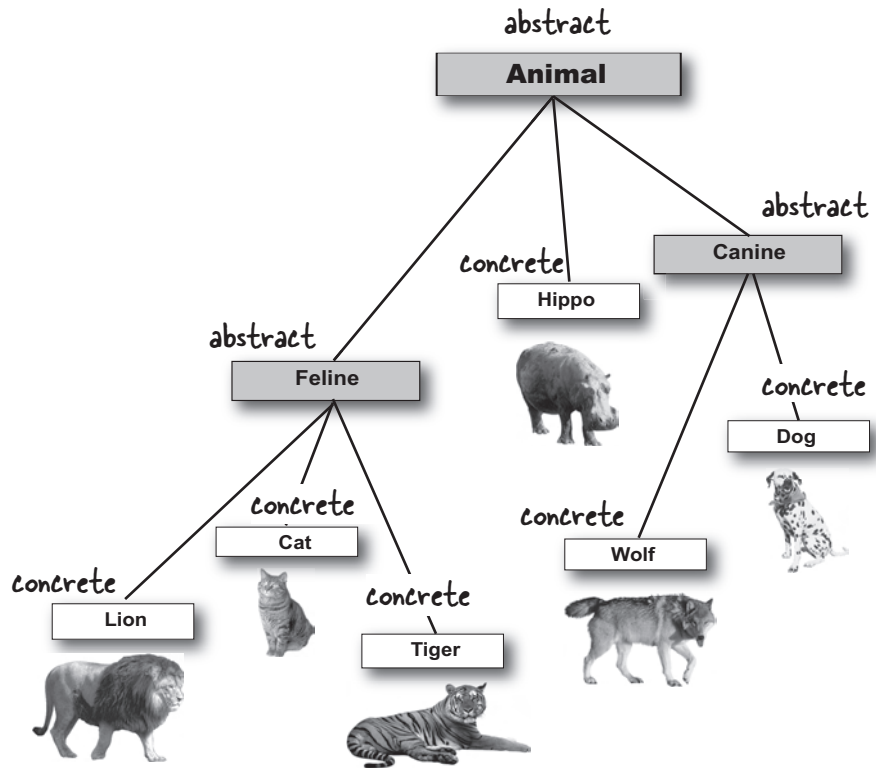
With an abstract class, the guys doing the work at runtime are **instances of a subclass** of your abstract class.

*There is an exception to this—an abstract class can have static members (see chapter 10).

Abstract vs. Concrete

A class that's not abstract is called a *concrete* class. In the Animal inheritance tree, if we make Animal, Canine, and Feline abstract, that leaves Hippo, Wolf, Dog, Tiger, Lion, and Cat as the concrete subclasses.

Flip through the Java API and you'll find a lot of abstract classes, especially in the GUI library. What does a GUI Component look like? The Component class is the superclass of GUI-related classes for things like buttons, text areas, scrollbars, dialog boxes, you name it. You don't make an instance of a generic *Component* and put it on the screen, you make a JButton. In other words, you instantiate only a *concrete subclass* of Component, but never Component itself.

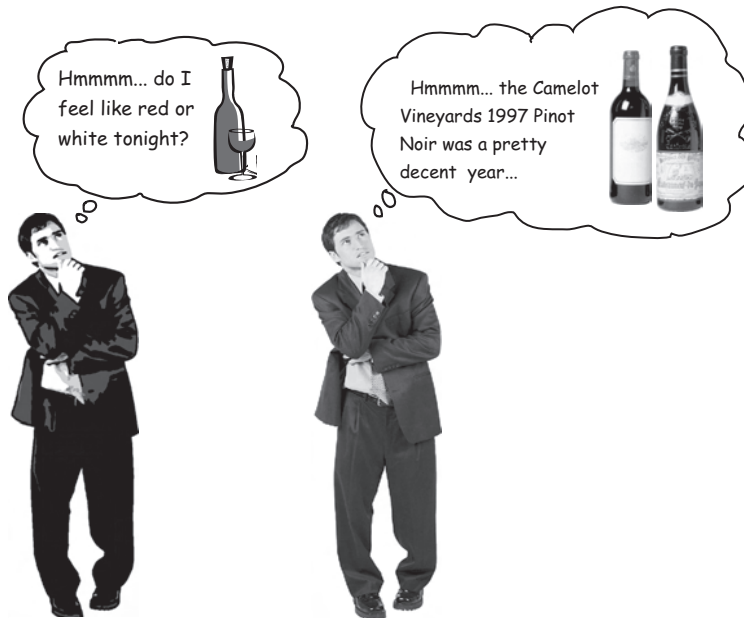


abstract or concrete?

How do you know when a class should be abstract? **Wine** is probably abstract. But what about **Red** and **White**? Again probably abstract (for some of us, anyway). But at what point in the hierarchy do things become concrete?

Do you make **PinotNoir** concrete, or is it abstract too? It looks like the Camelot Vineyards 1997 Pinot Noir is probably concrete no matter what. But how do you know for sure?

Look at the Animal inheritance tree above. Do the choices we've made for which classes are abstract and which are concrete seem appropriate? Would you change anything about the Animal inheritance tree (other than adding more Animals, of course)?



Abstract methods

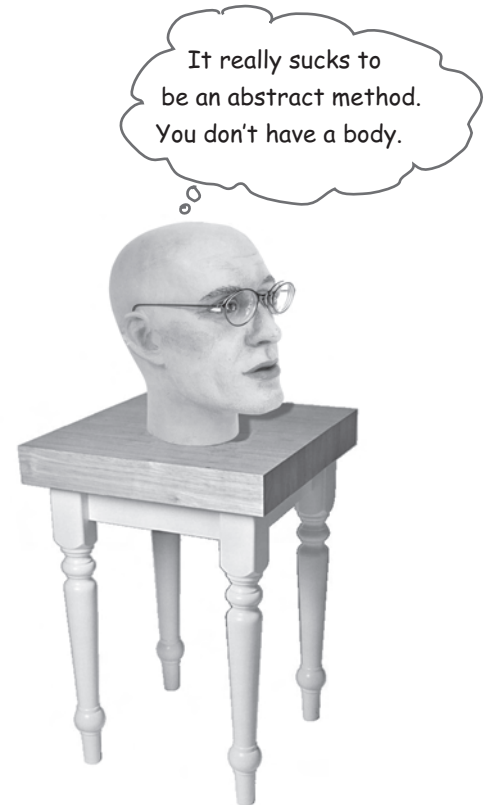
Besides classes, you can mark *methods* abstract, too. An abstract class means the class must be *extended*; an abstract method means the method must be *overridden*. You might decide that some (or all) behaviors in an abstract class don't make any sense unless they're implemented by a more specific subclass. In other words, you can't think of any generic method implementation that could possibly be useful for subclasses. What would a generic `eat()` method look like?

An abstract method has no body!

Because you've already decided there isn't any code that would make sense in the abstract method, you won't put in a method body. So no curly braces—just end the declaration with a semicolon.

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

No method body!
End it with a semicolon.



If you declare an abstract *method*, you *MUST* mark the *class* abstract as well. You can't have an abstract method in a non-abstract class.

If you put even a single abstract method in a class, you have to make the class abstract. But you *can* mix both abstract and non-abstract methods in the abstract class.

there are no Dumb Questions

Q: What is the *point* of an abstract method? I thought the whole point of an abstract class was to have common code that could be inherited by subclasses.

A: Inheritable method implementations (in other words, methods with actual *bodies*) are A Good Thing to put in a superclass. *When it makes sense*. And in an abstract class, it often *doesn't* make sense, because you can't come up with any generic code that subclasses would find useful. The point of an abstract method is that even though you haven't put in any actual method code, you've still defined part of the *protocol* for a group of subtypes (subclasses).

Q: Which is good because...

A: Polymorphism! Remember, what you want is the ability to use a superclass type (often abstract) as a method argument, return type, or array type. That way, you get to add new subtypes (like a new `Animal` subclass) to your program without having to rewrite (or add) new methods to deal with those new types. Imagine how you'd have to change the `Vet` class, if it didn't use `Animal` as its argument type for methods. You'd have to have a separate method for every single `Animal` subclass! One that takes a `Lion`, one that takes a `Wolf`, one that takes a... you get the idea. So with an abstract method, you're saying, "All subtypes of this type have THIS method." for the benefit of polymorphism.

You **MUST** implement all abstract methods



Implementing an abstract method is just like overriding a method.

Abstract methods don't have a body; they exist solely for polymorphism. That means the first concrete class in the inheritance tree must implement *all* abstract methods.

You can, however, pass the buck by being abstract yourself. If both `Animal` and `Canine` are abstract, for example, and both have abstract methods, class `Canine` does not have to implement the abstract methods from `Animal`. But as soon as we get to the first concrete subclass, like `Dog`, that subclass must implement *all* of the abstract methods from both `Animal` and `Canine`.

But remember that an abstract class can have both abstract and *non*-abstract methods, so `Canine`, for example, could implement an abstract method from `Animal`, so that `Dog` didn't have to. But if `Canine` says nothing about the abstract methods from `Animal`, `Dog` has to implement all of `Animal`'s abstract methods.

When we say "you must implement the abstract method", that means you *must provide a body*. That means you must create a non-abstract method in your class with the same method signature (name and arguments) and a return type that is compatible with the declared return type of the abstract method. What you put *in* that method is up to you. All Java cares about is that the method is *there*, in your concrete subclass.



Abstract vs. Concrete Classes

Let's put all this abstract rhetoric into some concrete use. In the middle column we've listed some classes. Your job is to imagine applications where the listed class might be concrete, and applications where the listed class might be abstract. We took a shot at the first few to get you going. For example, class Tree would be abstract in a tree nursery program, where differences between an Oak and an Aspen matter. But in a golf simulation program, Tree might be a concrete class (perhaps a subclass of Obstacle), because the program doesn't care about or distinguish between different types of trees. (There's no one right answer; it depends on your design.)

Concrete	Sample class	Abstract
golf course simulation	Tree	tree nursery application
_____	House	architect application
satellite photo application	Town	_____
_____	Football Player	coaching application
_____	Chair	_____
_____	Customer	_____
_____	Sales Order	_____
_____	Book	_____
_____	Store	_____
_____	Supplier	_____
_____	Golf Club	_____
_____	Carburetor	_____
_____	Oven	_____

Polymorphism in action

Let's say that we want to write our *own* kind of list class, one that will hold Dog objects, but pretend for a moment that we don't know about the ArrayList class. For the first pass, we'll give it just an *add()* method. We'll use a simple Dog array (Dog []) to keep the added Dog objects, and give it a length of 5. When we reach the limit of 5 Dog objects, you can still call the *add()* method but it won't do anything. If we're *not* at the limit, the *add()* method puts the Dog in the array at the next available index position, then increments that next available index (nextIndex).

Building our own Dog-specific list

(Perhaps the world's worst attempt at making our own ArrayList kind of class, from scratch.)

version
1

MyDogList
Dog[] dogs int nextIndex
add(Dog d)

```
public class MyDogList {
    private Dog [] dogs = new Dog[5];
    private int nextIndex = 0;

    public void add(Dog d) {
        if (nextIndex < dogs.length) {
            dogs[nextIndex] = d;
            System.out.println("Dog added at " + nextIndex);
            nextIndex++;
        }
    }
}
```

Use a plain old Dog array behind the scenes.

We'll increment this each time a new Dog is added.

If we're not already at the limit of the dogs array, add the Dog and print a message.

increment, to give us the next index to use

Uh-oh, now we need to keep Cats, too.

We have a few options here:

- 1) Make a separate class, MyCatList, to hold Cat objects. Pretty clunky.
- 2) Make a single class, DogAndCatList, that keeps two different arrays as instance variables and has two different add() methods: addCat(Cat c) and addDog(Dog d). Another clunky solution.
- 3) Make heterogeneous AnimalList class, that takes *any* kind of Animal subclass (since we know that if the spec changed to add Cats, sooner or later we'll have some *other* kind of animal added as well). We like this option best, so let's change our class to make it more generic, to take Animals instead of just Dogs. We've highlighted the key changes (the logic is the same, of course, but the type has changed from Dog to Animal everywhere in the code).

Building our own Animal-specific list

version
2

MyAnimalList
Animal[] animals int nextIndex
add(Animal a)

```
public class MyAnimalList {
    private Animal[] animals = new Animal[5];
    private int nextIndex = 0;

    public void add(Animal a) {
        if (nextIndex < animals.length) {
            animals[nextIndex] = a;
            System.out.println("Animal added at " + nextIndex);
            nextIndex++;
        }
    }
}
```

Don't panic. We're not making a new Animal object; we're making a new array object, of type Animal. (Remember, you cannot make a new instance of an abstract type, but you CAN make an array object declared to HOLD that type.)

```
public class AnimalTestDrive{
    public static void main (String[] args) {
        MyAnimalList list = new MyAnimalList();
        Dog a = new Dog();
        Cat c = new Cat();
        list.add(a);
        list.add(c);
    }
}
```

```
File Edit Window Help Harm

% java AnimalTestDrive
Animal added at 0
Animal added at 1
```

the ultimate superclass: **Object**

What about non-Animals? Why not make a class generic enough to take anything?

You know where this is heading. We want to change the type of the array, along with the `add()` method argument, to something *above* `Animal`. Something even *more* generic, *more* abstract than `Animal`. But how can we do it? We don't *have* a superclass for `Animal`.

Then again, maybe we do...

Remember those methods of `ArrayList`? Look how the `remove`, `contains`, and `indexOf` method all use an object of type... ***Object!***

Every class in Java extends class **Object**.

Class `Object` is the mother of all classes; it's the superclass of *everything*.

Even if you take advantage of polymorphism, you still have to create a class with methods that take and return *your* polymorphic type. Without a common superclass for everything in Java, there'd be no way for the developers of Java to create classes with methods that could take *your* custom types... *types they never knew about when they wrote the `ArrayList` class.*

So you were making subclasses of class `Object` from the very beginning and you didn't even know it. ***Every class you write extends `Object`***, without your ever having to say it. But you can think of it as though a class you write looks like this:

```
public class Dog extends Object { }
```

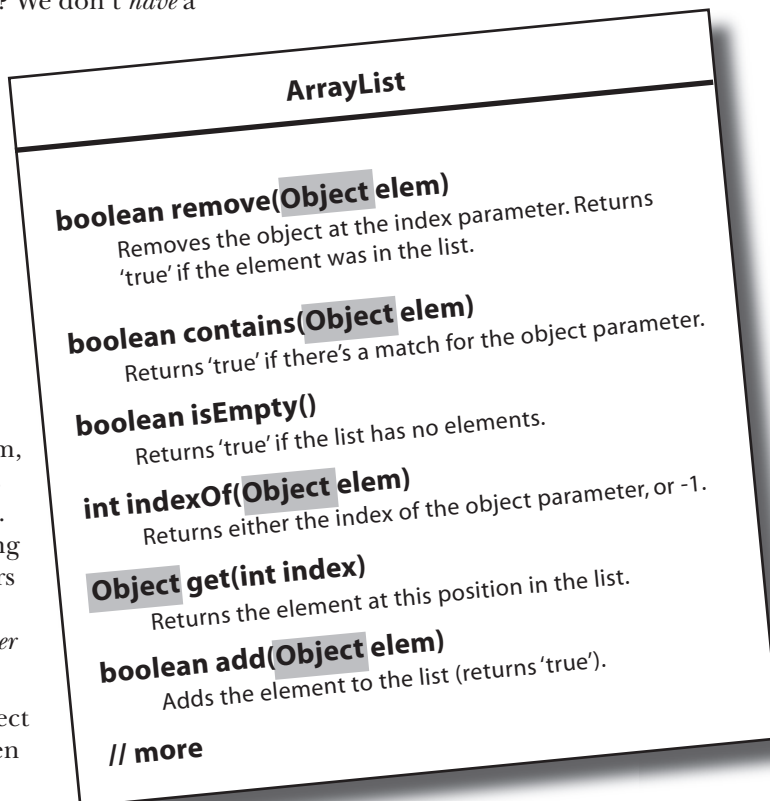
But wait a minute, `Dog` *already* extends something, `Canine`. That's OK. The compiler will make `Canine` extend `Object` instead. Except `Canine` extends `Animal`. No problem, then the compiler will just make `Animal` extend `Object`.

Any class that doesn't **explicitly** extend another class, **implicitly** extends **Object**.

So, since `Dog` extends `Canine`, it doesn't *directly* extend `Object` (although it does extend it indirectly), and the same is true for `Canine`, but `Animal` *does* directly extend `Object`.

version
3

(These are just a few of the methods in `ArrayList`...there are many more.)



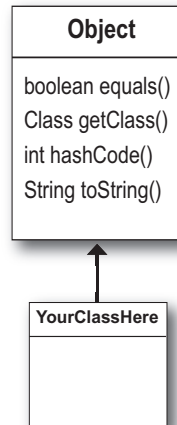
Many of the `ArrayList` methods use the ultimate polymorphic type, `Object`. Since every class in Java is a subclass of `Object`, these `ArrayList` methods can take anything!

(Note: as of Java 5.0, the `get()` and `add()` methods actually look a little different than the ones shown here, but for now this is the way to think about it. We'll get into the full story a little later.)

So what's in this ultra-super-megaclass Object?

If you were Java, what behavior would you want *every* object to have? Hmmmm... let's see... how about a method that lets you find out if one object is equal to another object? What about a method that can tell you the actual class type of that object? Maybe a method that gives you a hashCode for the object, so you can use the object in hashtables (we'll talk about Java's hashtables in chapter 17 and appendix B). Oh, here's a good one—a method that prints out a String message for that object.

And what do you know? As if by magic, class Object does indeed have methods for those four things. That's not all, though, but these are the ones we really care about.



Just *SOME* of the methods of class Object.

Every class you write inherits all the methods of class Object. The classes you've written inherited methods you didn't even know you had.

① equals(Object o)

```
Dog a = new Dog();
Cat c = new Cat();

if (a.equals(c)) {
    System.out.println("true");
} else {
    System.out.println("false");
}
```

```
File Edit Window Help Stop
% java TestObject
false
```

Tells you if two objects are considered 'equal' (we'll talk about what 'equal' really means in appendix B).

③ hashCode()

```
Cat c = new Cat();
System.out.println(c.hashCode());
```

```
File Edit Window Help Drop
% java TestObject
8202111
```

Prints out a hashCode for the object (for now, think of it as a unique ID).

② getClass()

```
Cat c = new Cat();
System.out.println(c.getClass());
```

```
File Edit Window Help Faint
% java TestObject
class Cat
```

Gives you back the class that object was instantiated from.

④ toString()

```
Cat c = new Cat();
System.out.println(c.toString());
```

```
File Edit Window Help LapseIntoComa
% java TestObject
Cat@7d277f
```

Prints out a String message with the name of the class and some other number we rarely care about.

Object and abstract classes

^{there are no} Dumb Questions

Q: Is class `Object` abstract?

A: No. Well, not in the formal Java sense anyway. `Object` is a non-abstract class because it's got method implementation code that all classes can inherit and use out-of-the-box, without having to override the methods.

Q: Then *can* you override the methods in `Object`?

A: Some of them. But some of them are marked `final`, which means you can't override them. You're encouraged (strongly) to override `hashCode()`, `equals()`, and `toString()` in your own classes, and you'll learn how to do that a little later in the book. But some of the methods, like `getClass()`, do things that must work in a specific, guaranteed way.

Q: If `ArrayList` methods are generic enough to use `Object`, then what does it mean to say `ArrayList<DotCom>`? I thought I was restricting the `ArrayList` to hold only `DotCom` objects?

A: You *were* restricting it. Prior to Java 5.0, `ArrayList`s couldn't be restricted. They were all essentially what you get in Java 5.0 today if you write `ArrayList<Object>`. In other words, ***an `ArrayList` restricted to anything that's an `Object`***, which means *any* object in Java, instantiated from *any* class type! We'll cover the details of this new `<type>` syntax later in the book.

Q: OK, back to class `Object` being non-abstract (so I guess that means it's concrete), **HOW** can you let somebody make an `Object` object? Isn't that just as weird as making an `Animal` object?

A: Good question! Why is it acceptable to make a new `Object` instance? Because sometimes you just want a generic object to use as, well, as an object. A *lightweight* object. By far, the most common use of an instance of type `Object` is for thread synchronization (which you'll learn about in chapter 15). For now, just stick that on the back burner and assume that you will rarely make objects of type `Object`, even though you *can*.

Q: So is it fair to say that the main purpose for type `Object` is so that you can use it for a polymorphic argument and return type? Like in `ArrayList`?

A: The `Object` class serves two main purposes: to act as a polymorphic type for methods that need to work on any class that you or anyone else makes, and to provide *real* method code that all objects in Java need at runtime (and putting them in class `Object` means all other classes inherit them). Some of the most important methods in `Object` are related to threads, and we'll see those later in the book.

Q: If it's so good to use polymorphic types, why don't you just make **ALL** your methods take and return type `Object`?

A: Ahhhh... think about what would happen. For one thing, you would defeat the whole point of 'type-safety', one of Java's greatest protection mechanisms for your code. With type-safety, Java guarantees that you won't ask the wrong object to do something you *meant* to ask of another object type. Like, ask a *Ferrari* (which you think is a *Toaster*) to *cook itself*. But the truth is, you *don't* have to worry about that fiery Ferrari scenario, even if you *do* use `Object` references for everything. Because when objects are referred to by an `Object` reference type, Java *thinks* it's referring to an instance of type `Object`. And that means the only methods you're allowed to call on that object are the ones declared in class `Object`! So if you were to say:

```
Object o = new Ferrari();  
o.goFast(); //Not legal!
```

You wouldn't even make it past the compiler.

Because Java is a strongly-typed language, the compiler checks to make sure that you're calling a method on an object that's actually capable of *responding*. In other words, you can call a method on an object reference *only* if the class of the reference type actually *has* the method. We'll cover this in much greater detail a little later, so don't worry if the picture isn't crystal clear.

Using polymorphic references of type Object has a price...

Before you run off and start using type `Object` for all your ultra-flexible argument and return types, you need to consider a little issue of using type `Object` as a reference. And keep in mind that we're not talking about making instances of type `Object`; we're talking about making instances of some other type, but using a reference of type `Object`.

When you put an object into an `ArrayList<Dog>`, it goes in as a `Dog`, and comes out as a `Dog`:

```
ArrayList<Dog> myDogArrayList = new ArrayList<Dog>();
Dog aDog = new Dog();
myDogArrayList.add(aDog);
Dog d = myDogArrayList.get(0);
```

← Make an `ArrayList` declared to hold `Dog` objects.
 ← Make a `Dog`.
 ← Add the `Dog` to the list.
 ← Assign the `Dog` from the list to a new `Dog` reference variable. (Think of it as though the `get()` method declares a `Dog` return type because you used `ArrayList<Dog>`.)

But what happens when you declare it as `ArrayList<Object>`? If you want to make an `ArrayList` that will literally take *any* kind of `Object`, you declare it like this:

```
ArrayList<Object> myDogArrayList = new ArrayList<Object>();
Dog aDog = new Dog();
myDogArrayList.add(aDog);
```

← Make an `ArrayList` declared to hold any type of `Object`.
 ← Make a `Dog`.
 ← Add the `Dog` to the list. (These two steps are the same.)

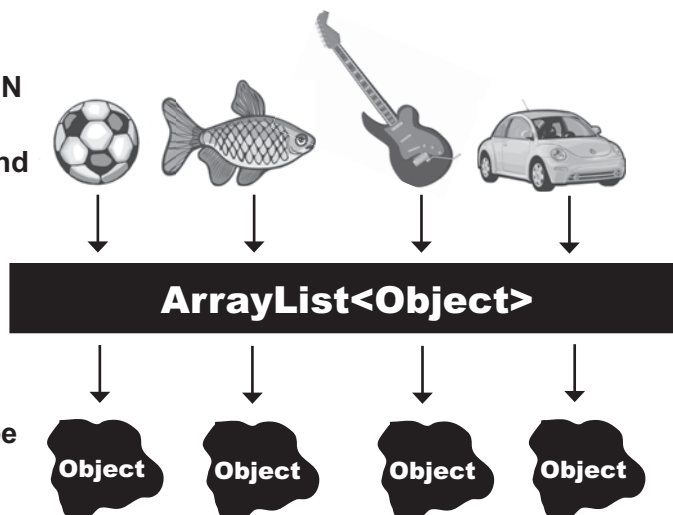
But what happens when you try to get the `Dog` object and assign it to a `Dog` reference?

```
Dog d = myDogArrayList.get(0);
```

NO!! Won't compile!! When you use `ArrayList<Object>`, the `get()` method returns type `Object`. The Compiler knows only that the object inherits from `Object` (somewhere in its inheritance tree) but it doesn't know it's a `Dog`!!

Everything comes out of an `ArrayList<Object>` as a reference of type ***Object***, regardless of what the actual object is, or what the reference type was when you added the object to the list.

The objects go IN as **SoccerBall, Fish, Guitar, and Car.**



But they come OUT as though they were of type **Object**.

Objects come out of an `ArrayList<Object>` acting like they're generic instances of class `Object`. The Compiler cannot assume the object that comes out is of any type other than `Object`.

When a Dog won't act like a Dog

The problem with having everything treated polymorphically as an Object is that the objects *appear* to lose (but not permanently) their true essence. *The Dog appears to lose its dogness.* Let's see what happens when we pass a Dog to a method that returns a reference to the same Dog object, but declares the return type as type Object rather than Dog.



BAD
☹️

```
public void go() {  
    Dog aDog = new Dog();  
    Dog sameDog = getObject(aDog);  
}
```

```
public Object getObject(Object o) {  
    return o;  
}
```

↖️ We're returning a reference to the same Dog, but as a return type of Object. This part is perfectly legal. Note: this is similar to how the get() method works when you have an ArrayList<Object> rather than an ArrayList<Dog>.

↖️ This line won't work! Even though the method returned a reference to the very same Dog the argument referred to, the return type Object means the compiler won't let you assign the returned reference to anything but Object.

```
File Edit Window Help Remember  
  
DogPolyTest.java:10: incompatible types  
found   : java.lang.Object  
required: Dog  
        Dog sameDog = takeObjects(aDog);  
1 error
```

↖️ The compiler doesn't know that the thing returned from the method is actually a Dog, so it won't let you assign it to a Dog reference. (You'll see why on the next page.)

GOOD
😊

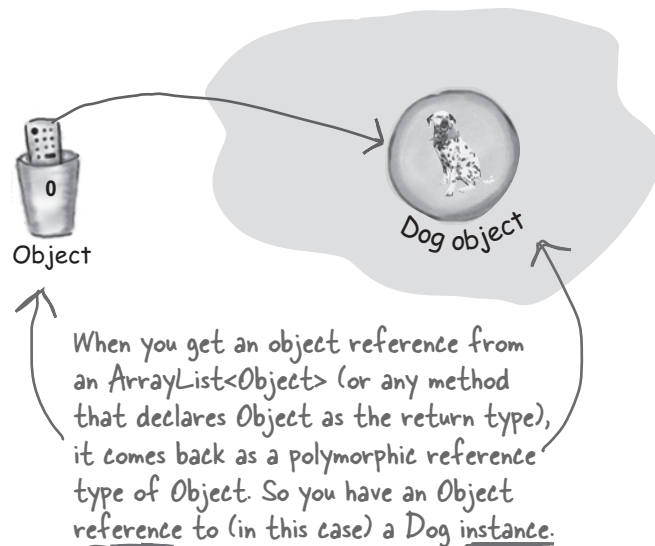
```
public void go() {  
    Dog aDog = new Dog();  
    Object sameDog = getObject(aDog);  
}
```

```
public Object getObject(Object o) {  
    return o;  
}
```

↖️ This works (although it may not be very useful, as you'll see in a moment) because you can assign **ANYTHING** to a reference of type Object, since every class passes the IS-A test for Object. Every object in Java is an instance of type Object, because every class in Java has Object at the top of its inheritance tree.

Objects don't bark.

So now we know that when an object is referenced by a variable declared as type `Object`, it can't be assigned to a variable declared with the actual object's type. And we know that this can happen when a return type or argument is declared as type `Object`, as would be the case, for example, when the object is put into an `ArrayList` of type `Object` using `ArrayList<Object>`. But what are the implications of this? Is it a problem to have to use an `Object` reference variable to refer to a `Dog` object? Let's try to call `Dog` methods on our `Dog-That-Compiler-Thinks-Is-An-Object`:



```
Object o = al.get(index);
```

```
int i = o.hashCode();
```

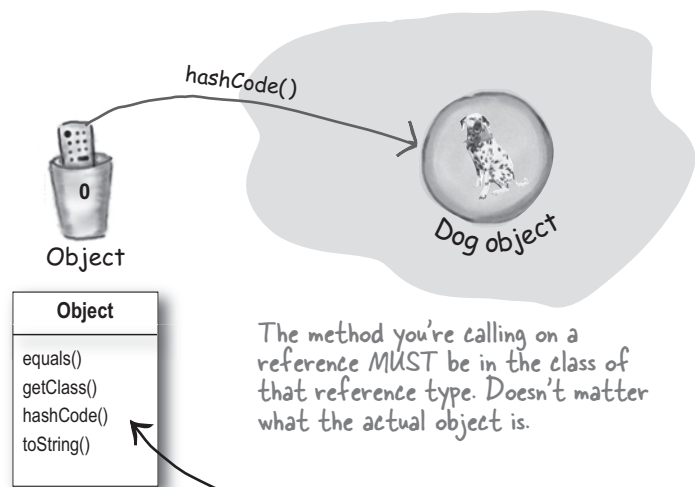
Won't compile! → `o.bark();` ←

Can't do this!! The `Object` class has no idea what it means to `bark()`. Even though YOU know it's really a `Dog` at that index, the compiler doesn't..

This is fine. Class `Object` has a `hashCode()` method, so you can call that method on ANY object in Java.

The compiler decides whether you can call a method based on the reference type, not the actual object type.

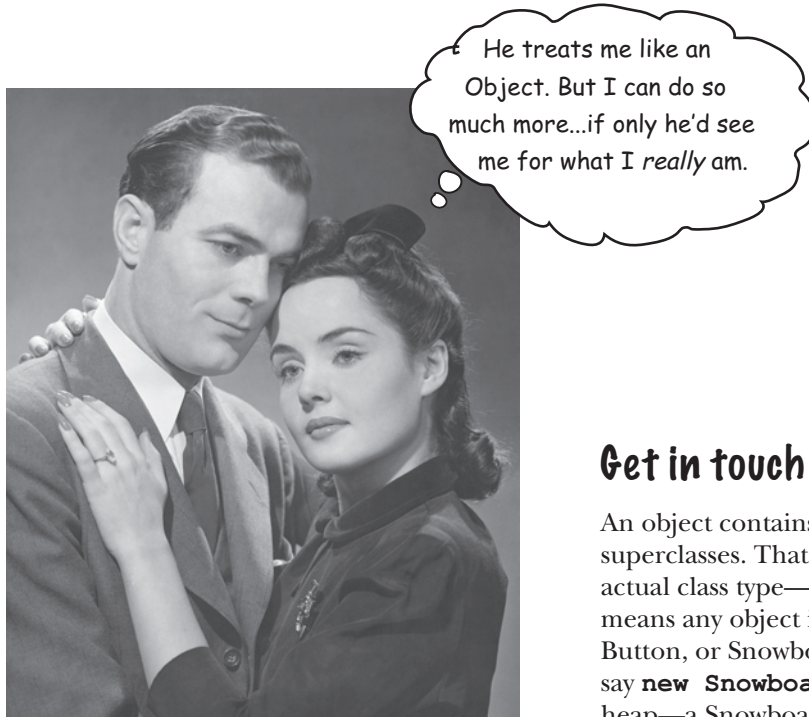
Even if you *know* the object is capable (“...but it really *is* a `Dog`, honest..”), the compiler sees it only as a generic `Object`. For all the compiler knows, you put a `Button` object out there. Or a `Microwave` object. Or some other thing that really doesn't know how to bark. The compiler checks the class of the *reference type*—not the *object type*—to see if you can call a method using that reference.



The method you're calling on a reference **MUST** be in the class of that reference type. Doesn't matter what the actual object is.

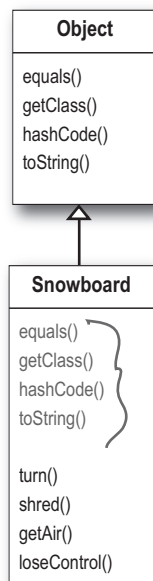
```
o.hashCode();
```

The “`o`” reference was declared as type `Object`, so you can call methods only if those methods are in class `Object`.

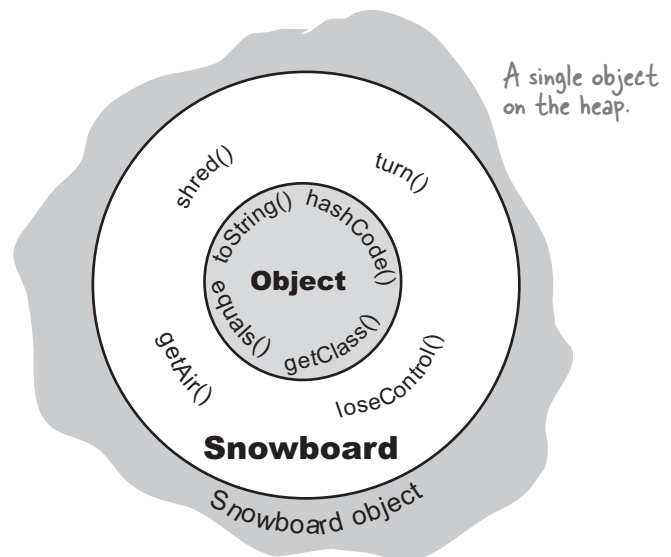


Get in touch with your inner Object.

An object contains *everything* it inherits from each of its superclasses. That means *every* object—regardless of its actual class type—is *also* an instance of class `Object`. That means any object in Java can be treated not just as a `Dog`, `Button`, or `Snowboard`, but also as an `Object`. When you say `new Snowboard()`, you get a single object on the heap—a `Snowboard` object—but that `Snowboard` wraps itself around an inner core representing the `Object` (capital “O”) portion of itself.



Snowboard inherits methods from superclass `Object`, and adds four more.



There is only *ONE* object on the heap here. A `Snowboard` object. But it contains both the `Snowboard` class parts of itself and the `Object` class parts of itself.

‘Polymorphism’ means ‘many forms’.

You can treat a Snowboard as a Snowboard or as an Object.

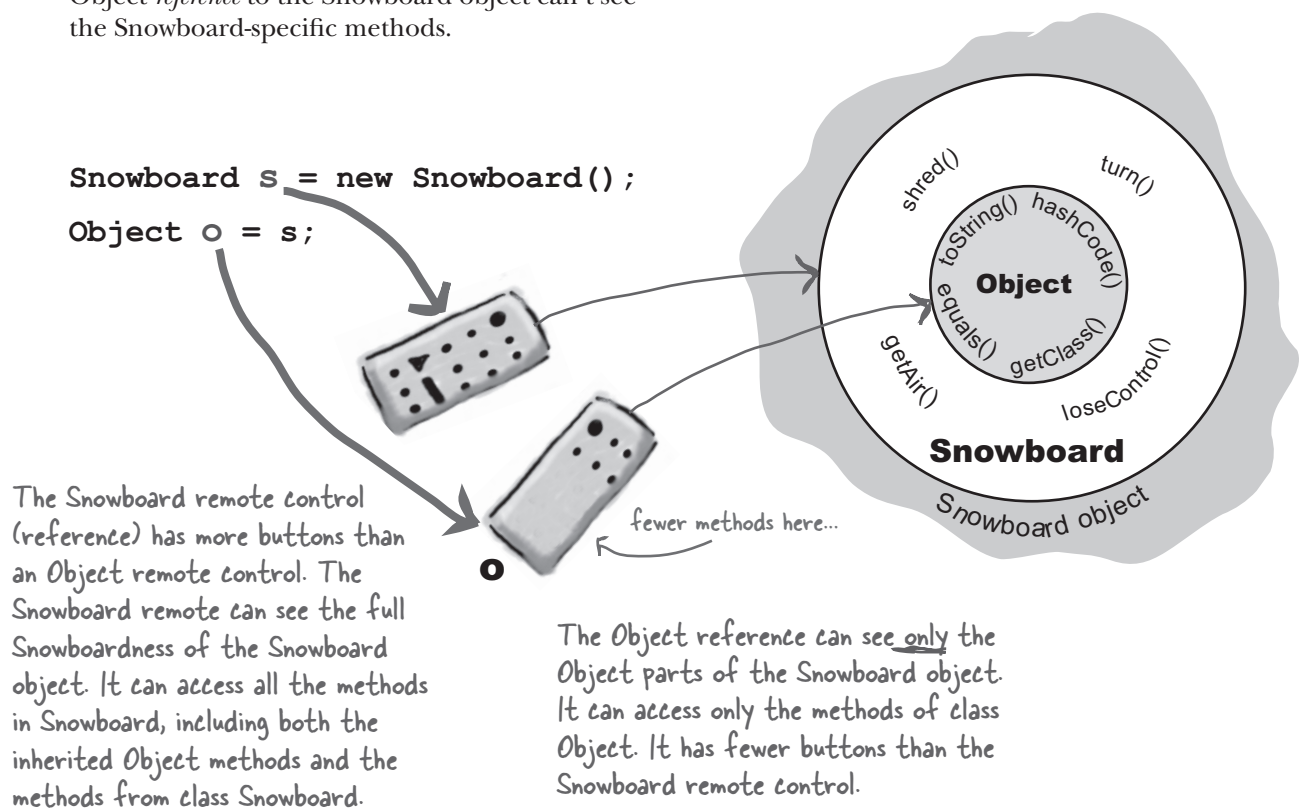
If a reference is like a remote control, the remote control takes on more and more buttons as you move down the inheritance tree. A remote control (reference) of type `Object` has only a few buttons—the buttons for the exposed methods of class `Object`. But a remote control of type `Snowboard` includes all the buttons from class `Object`, plus any new buttons (for new methods) of class `Snowboard`. The more specific the class, the more buttons it may have.

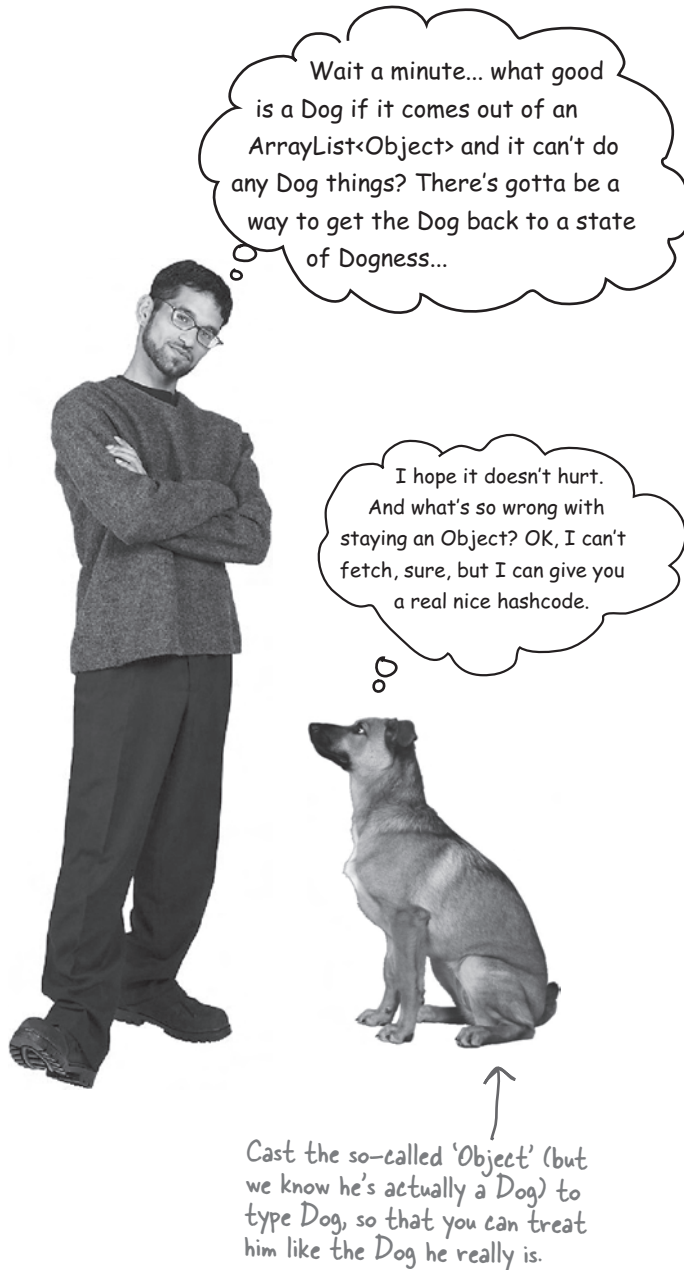
Of course that’s not always true; a subclass might not add any new methods, but simply override the methods of its superclass. The key point is that even if the *object* is of type `Snowboard`, an *Object* *reference* to the `Snowboard` object can’t see the `Snowboard`-specific methods.

When you put an object in an `ArrayList<Object>`, you can treat it only as an `Object`, regardless of the type it was when you put it in.

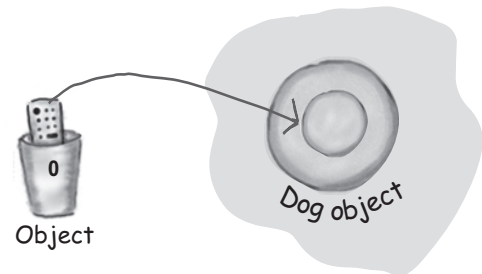
When you get a reference from an `ArrayList<Object>`, the reference is always of type `Object`.

That means you get an `Object` remote control.



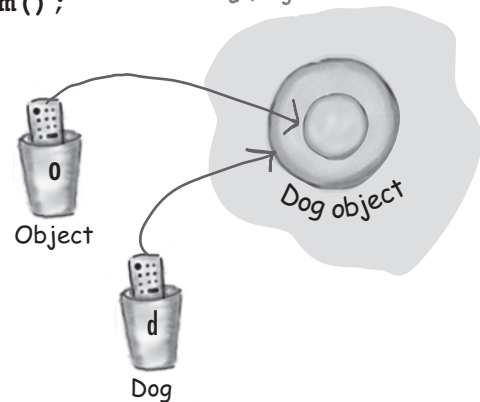


Casting an object reference back to its *real* type.



It's really still a *Dog object*, but if you want to call Dog-specific methods, you need a *reference* declared as type *Dog*. If you're *sure** the object is really a *Dog*, you can make a new *Dog* reference to it by copying the *Object* reference, and forcing that copy to go into a *Dog* reference variable, using a cast `(Dog)`. You can use the new *Dog* reference to call *Dog* methods.

```
Object o = al.get(index);
Dog d = (Dog) o; ← cast the Object back to
                  a Dog we know is there.
d.roam();
```



*If you're *not* sure it's a *Dog*, you can use the **instanceof** operator to check. Because if you're wrong when you do the cast, you'll get a `ClassCastException` at runtime and come to a grinding halt.

```
if (o instanceof Dog) {
    Dog d = (Dog) o;
}
```

So now you've seen how much Java cares about the methods in the class of the reference variable.

You can call a method on an object *only* if the class of the reference variable has that method.

Think of the public methods in your class as your contract, your promise to the outside world about the things you can do.



When you write a class, you almost always *expose* some of the methods to code outside the class. To *expose* a method means you make a method *accessible*, usually by marking it public.

Imagine this scenario: you're writing code for a small business accounting program. A custom application for "Simon's Surf Shop". The good re-user that you are, you found an Account class that appears to meet your needs perfectly, according to its documentation, anyway. Each account instance represents an individual customer's account with the store. So there you are minding your own business invoking the *credit()* and *debit()* methods on an account object when you realize you need to get a balance on an account. No problem—there's a *getBalance()* method that should do nicely.

Account
debit(double amt)
credit(double amt)
double getBalance()

Except... when you invoke the *getBalance()* method, the whole thing blows up at runtime. Forget the documentation, the class does not have that method. Yikes!

But that won't happen to you, because everytime you use the dot operator on a reference (*a.doStuff()*), the compiler looks at the *reference* type (the type 'a' was declared to be) and checks that class to guarantee the class has the method, and that the method does indeed take the argument you're passing and return the kind of value you're expecting to get back.

Just remember that the compiler checks the class of the *reference* variable, not the class of the actual *object* at the other end of the reference.

What if you need to change the contract?

OK, pretend you're a Dog. Your Dog class isn't the *only* contract that defines who you are. Remember, you inherit accessible (which usually means *public*) methods from all of your superclasses.

True, your Dog class defines a contract.

But not *all* of your contract.

Everything in class *Canine* is part of your contract.

Everything in class *Animal* is part of your contract.

Everything in class *Object* is part of your contract.

According to the IS-A test, you *are* each of those things—Canine, Animal, and Object.

But what if the person who designed your class had in mind the Animal simulation program, and now he wants to use you (class Dog) for a Science Fair Tutorial on Animal objects.

That's OK, you're probably reusable for that.

But what if later he wants to use you for a PetShop program? *You don't have any **Pet** behaviors.* A Pet needs methods like *beFriendly()* and *play()*.

OK, now pretend you're the Dog class programmer. No problem, right? Just add some more methods to the Dog class. You won't be breaking anyone else's code by *adding* methods, since you aren't touching the *existing* methods that someone else's code might be calling on Dog objects.

Can you see any drawbacks to that approach (adding Pet methods to the Dog class)?



Think about what **YOU** would do if **YOU** were the Dog class programmer and needed to modify the Dog so that it could do Pet things, too. We know that simply adding new Pet behaviors (methods) to the Dog class will work, and won't break anyone else's code.

But... this is a PetShop program. It has more than just Dogs! And what if someone wants to use your Dog class for a program that has *wild* Dogs? What do you think your options might be, and without worrying about how Java handles things, just try to imagine how you'd *like* to solve the problem of modifying some of your Animal classes to include Pet behaviors.

Stop right now and think about it, **before you look at the next page** where we begin to reveal *everything*.

(thus rendering the whole exercise completely useless, robbing you of your One Big Chance to burn some brain calories)

Let's explore some design options for reusing some of our existing classes in a PetShop program.

On the next few pages, we're going to walk through some possibilities. We're not yet worried about whether Java can actually *do* what we come up with. We'll cross that bridge once we have a good idea of some of the tradeoffs.

① Option one

We take the easy path, and put pet methods in class `Animal`.

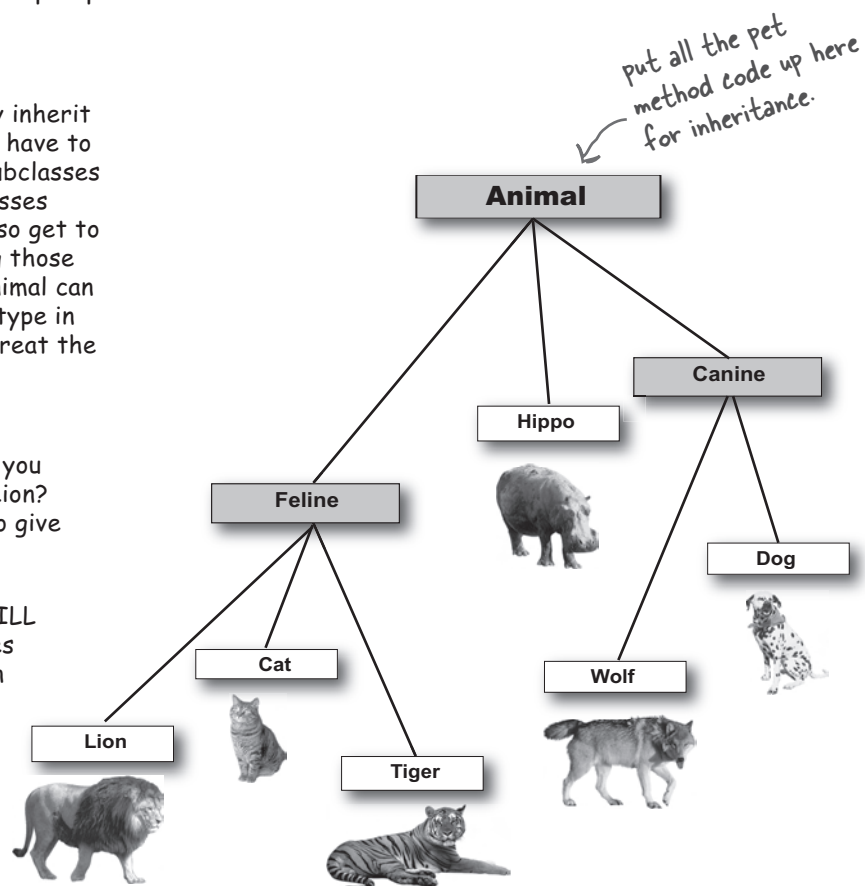
Pros:

All the `Animals` will instantly inherit the pet behaviors. We won't have to touch the existing `Animal` subclasses at all, and any `Animal` subclasses created in the future will also get to take advantage of inheriting those methods. That way, class `Animal` can be used as the polymorphic type in any program that wants to treat the `Animals` as pets.

Cons:

So... when was the last time you saw a Hippo at a pet shop? Lion? Wolf? Could be dangerous to give non-pets pet methods.

Also, we almost certainly **WILL** have to touch the pet classes like `Dog` and `Cat`, because (in our house, anyway) `Dogs` and `Cats` tend to implement pet behaviors **VERY** differently.



② Option two

We start with Option One, putting the pet methods in class `Animal`, but we make the methods abstract, forcing the `Animal` subclasses to override them.

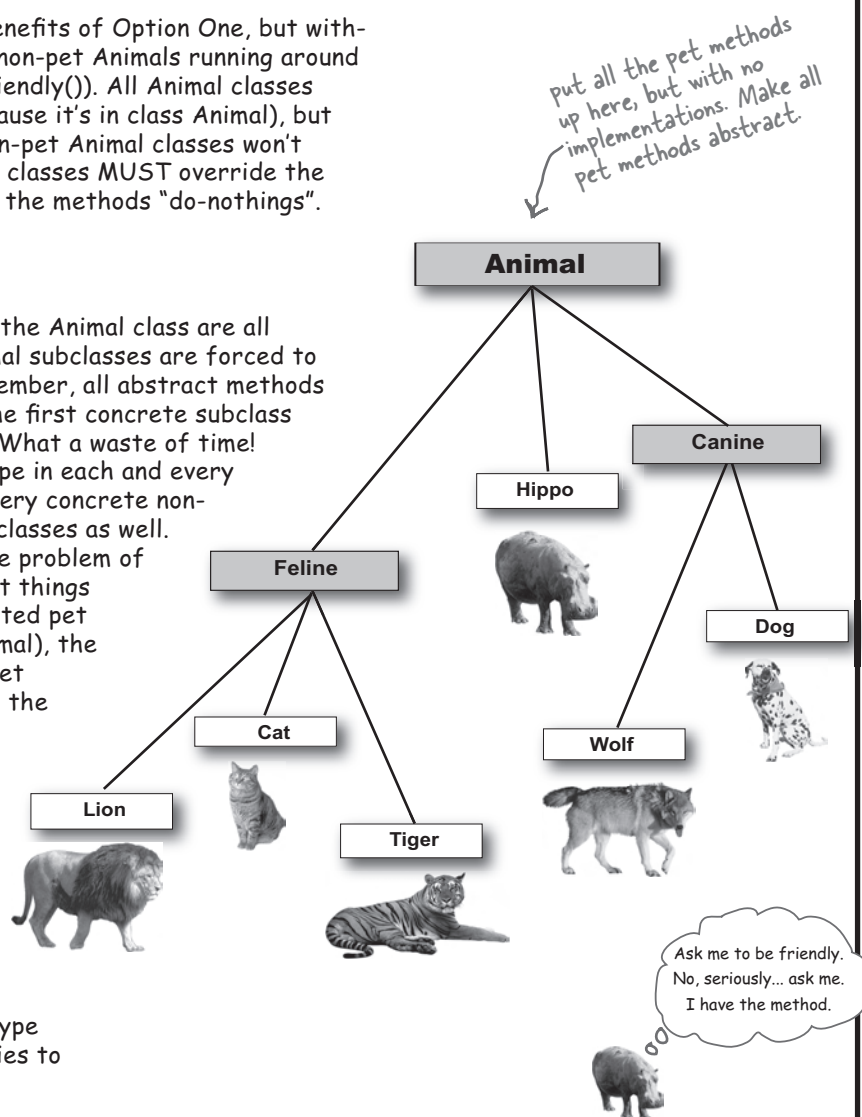
Pros:

That would give us all the benefits of Option One, but without the drawback of having non-pet `Animals` running around with pet methods (like `beFriendly()`). All `Animal` classes would have the method (because it's in class `Animal`), but because it's abstract the non-pet `Animal` classes won't inherit any functionality. All classes **MUST** override the methods, but they can make the methods "do-nothings".

Cons:

Because the pet methods in the `Animal` class are all abstract, the concrete `Animal` subclasses are forced to implement all of them. (Remember, all abstract methods **MUST** be implemented by the first concrete subclass down the inheritance tree.) What a waste of time! You have to sit there and type in each and every pet method into each and every concrete non-pet class, and all future subclasses as well. And while this does solve the problem of non-pets actually **DOING** pet things (as they would if they inherited pet functionality from class `Animal`), the contract is bad. Every *non-pet* class would be announcing to the world that it, too, has those pet methods, even though the methods wouldn't actually **DO** anything when called.

This approach doesn't look good at all. It just seems wrong to stuff everything into class `Animal` that more than one `Animal` type might need, **UNLESS** it applies to **ALL** `Animal` subclasses.



③ Option three

Put the pet methods **ONLY** in the classes where they belong.

Pros:

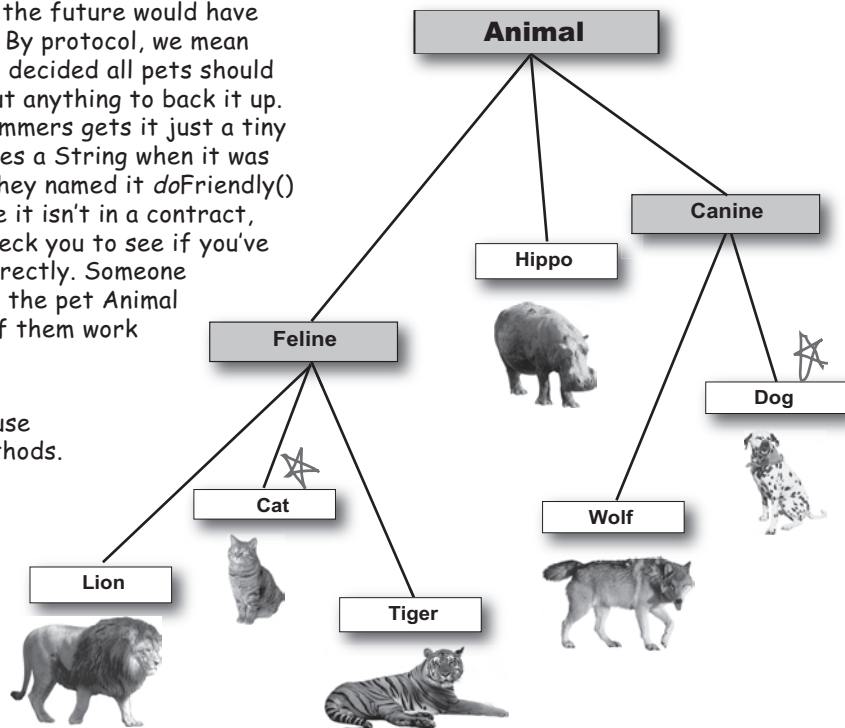
No more worries about Hippos greeting you at the door or licking your face. The methods are where they belong, and **ONLY** where they belong. Dogs can implement the methods and Cats can implement the methods, but nobody else has to know about them.

Cons:

Two Big Problems with this approach. First off, you'd have to agree to a protocol, and all programmers of pet Animal classes now and in the future would have to **KNOW** about the protocol. By protocol, we mean the exact methods that we've decided all pets should have. The pet contract without anything to back it up. But what if one of the programmers gets it just a tiny bit wrong? Like, a method takes a String when it was supposed to take an int? Or they named it `doFriendly()` instead of `beFriendly()`? Since it isn't in a contract, the compiler has no way to check you to see if you've implemented the methods correctly. Someone could easily come along to use the pet Animal classes and find that not all of them work quite right.

And second, you don't get to use polymorphism for the pet methods. Every class that needs to use pet behaviors would have to know about each and every class! In other words, you can't use Animal as the polymorphic type now, because the compiler won't let you call a Pet method on an Animal reference (even if it's really a Dog object) because class Animal doesn't have the method.

★ Put the pet methods **ONLY** in the Animal classes that can be pets, instead of in Animal.

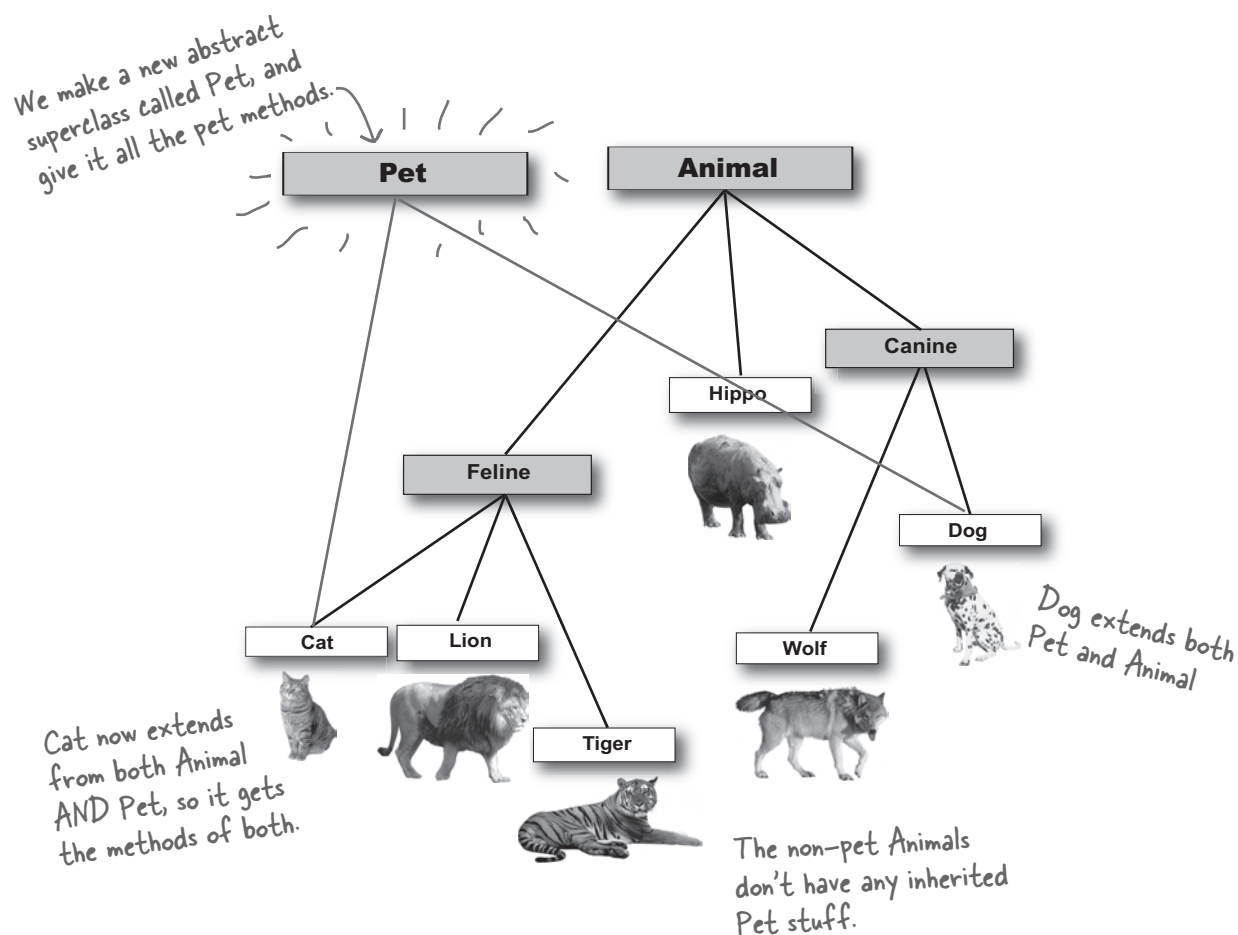


multiple inheritance?

So what we REALLY need is:

- ☀ A way to have pet behavior in **just** the pet classes
- ☀ A way to guarantee that all pet classes have all of the same methods defined (same name, same arguments, same return types, no missing methods, etc.), without having to cross your fingers and hope all the programmers get it right.
- ☀ A way to take advantage of polymorphism so that all pets can have their pet methods called, without having to use arguments, return types, and arrays for each and every pet class.

It looks like we need TWO superclasses at the top



There's just one problem with the "two superclasses" approach...

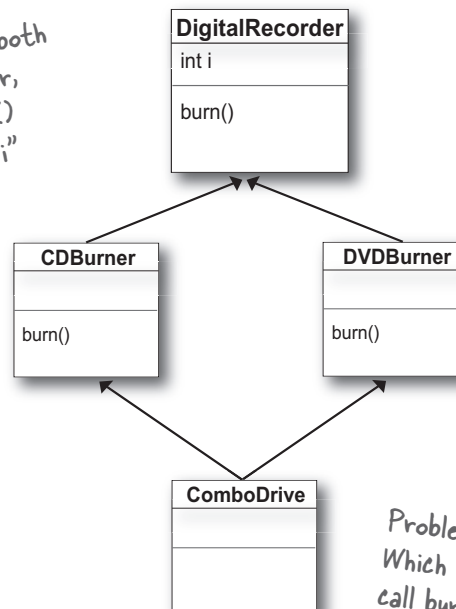
It's called "multiple inheritance" and it can be a Really Bad Thing.

That is, if it were possible to do in Java.

But it isn't, because multiple inheritance has a problem known as The Deadly Diamond of Death.

Deadly Diamond of Death

CDBurner and DVDBurner both inherit from DigitalRecorder, and both override the burn() method. Both inherit the "i" instance variable.



Imagine that the "i" instance variable is used by both CDBurner and DVDBurner, with different values. What happens if ComboDrive needs to use both values of "i"?

Problem with multiple inheritance. Which burn() method runs when you call burn() on the ComboDrive?

A language that allows the Deadly Diamond of Death can lead to some ugly complexities, because you have to have special rules to deal with the potential ambiguities. And extra rules means extra work for you both in *learning* those rules and watching out for those "special cases". Java is supposed to be *simple*, with consistent rules that don't blow up under some scenarios. So Java (unlike C++) protects you from having to think about the Deadly Diamond of Death. But that brings us back to the original problem! *How do we handle the Animal/Pet thing?*

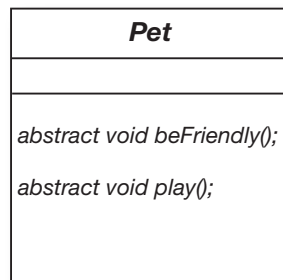
interfaces

Interface to the rescue!

Java gives you a solution. An *interface*. Not a *GUI* interface, not the generic use of the *word* interface as in, “That’s the public interface for the Button class API,” but the Java *keyword* **interface**.

A Java interface solves your multiple inheritance problem by giving you much of the polymorphic *benefits* of multiple inheritance without the pain and suffering from the Deadly Diamond of Death (DDD).

The way in which interfaces side-step the DDD is surprisingly simple: **make all the methods abstract!** That way, the subclass **must** implement the methods (remember, abstract methods *must* be implemented by the first concrete subclass), so at runtime the JVM isn’t confused about *which* of the two inherited versions it’s supposed to call.



A Java interface is like a 100% pure abstract class.

All methods in an interface are abstract, so any class that IS-A Pet **MUST** implement (i.e. override) the methods of Pet.

To DEFINE an interface:

```
public interface Pet {...}
```

↖ Use the keyword “interface” instead of “class”

To IMPLEMENT an interface:

```
public class Dog extends Canine implements Pet {...}
```

↖ Use the keyword “implements” followed by the interface name. Note that when you implement an interface you still get to extend a class

Making and Implementing the Pet interface

You say 'interface' instead of 'class' here

```
public interface Pet {
    public abstract void beFriendly();
    public abstract void play();
}
```

interface methods are implicitly public and abstract, so typing in 'public' and 'abstract' is optional (in fact, it's not considered 'good style' to type the words in, but we did here just to reinforce it, and because we've never been slaves to fashion...)

All interface methods are abstract, so they **MUST** end in semicolons. Remember, they have no body!

Dog IS-A Animal
and Dog IS-A Pet

```
public class Dog extends Canine implements Pet {
    public void beFriendly() {...}
    public void play() {...}

    public void roam() {...}
    public void eat() {...}
}
```

You say 'implements' followed by the name of the interface.

You SAID you are a Pet, so you **MUST** implement the Pet methods. It's your contract. Notice the curly braces instead of semicolons.

These are just normal overriding methods.

^{there are no} Dumb Questions

Q: Wait a minute, interfaces don't really give you multiple inheritance, because you can't put any implementation code in them. If all the methods are abstract, what does an interface really buy you?

A: Polymorphism, polymorphism, polymorphism. Interfaces are the ultimate in flexibility, because if you use interfaces instead of concrete subclasses (or even abstract superclass types) as arguments and return

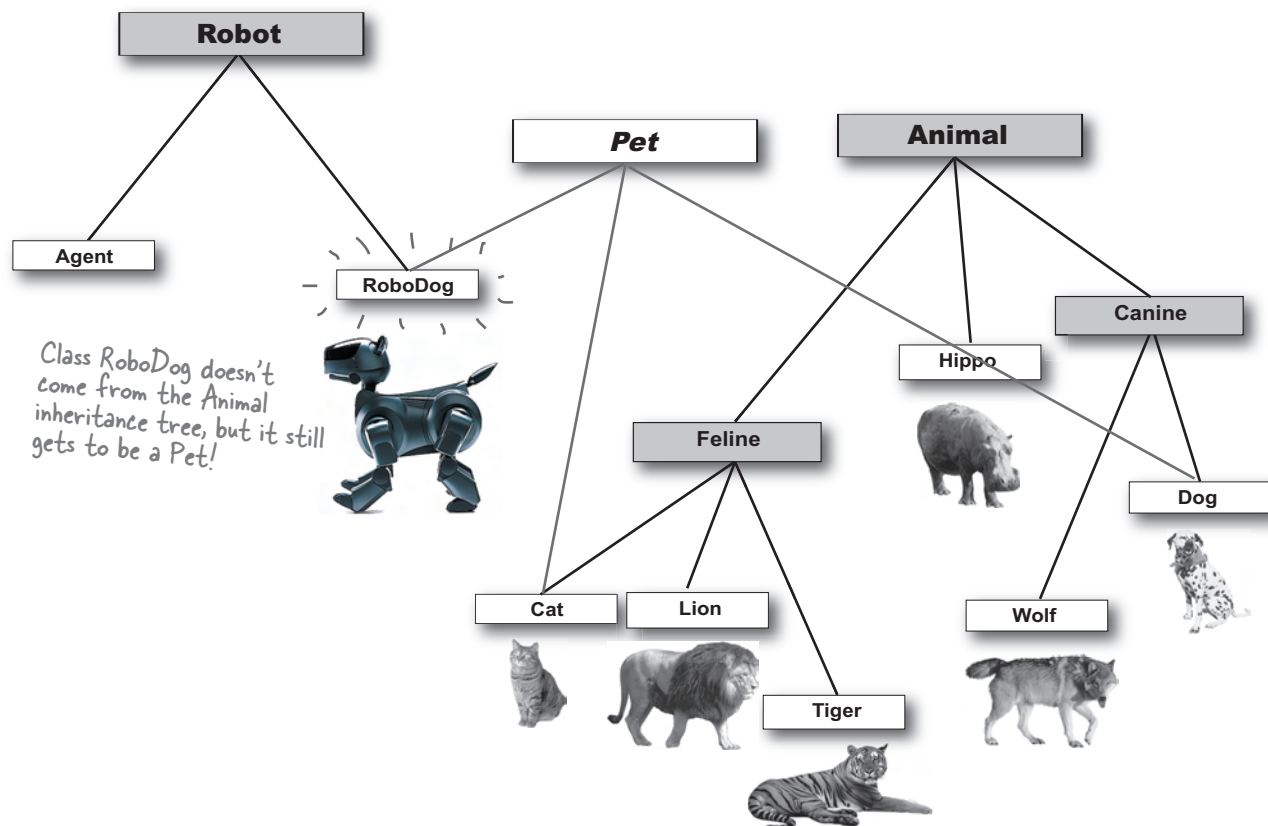
types, you can pass anything that implements that interface. And think about it—with an interface, a class doesn't have to come from just one inheritance tree. A class can extend one class, and implement an interface. But another class might implement the same interface, yet come from a completely different inheritance tree! So you get to treat an object by the role it plays, rather than by the class type from which it was instantiated.

In fact, if you wrote your code to use interfaces, you wouldn't even have to give anyone a superclass that they had

to extend. You could just give them the interface and say, "Here, I don't care what kind of class inheritance structure you come from, just implement this interface and you'll be good to go."

The fact that you can't put in implementation code turns out not to be a problem for most good designs, because most interface methods wouldn't make sense if implemented in a generic way. In other words, most interface methods would need to be overridden even if the methods weren't *forced* to be abstract.

Classes from *different* inheritance trees can implement the *same* interface.



When you use a *class* as a polymorphic type (like an array of type `Animal` or a method that takes a `Canine` argument), the objects you can stick in that type must be from the same inheritance tree. But not just anywhere in the inheritance tree; the objects must be from a class that is a subclass of the polymorphic type. An argument of type `Canine` can accept a `Wolf` and a `Dog`, but not a `Cat` or a `Hippo`.

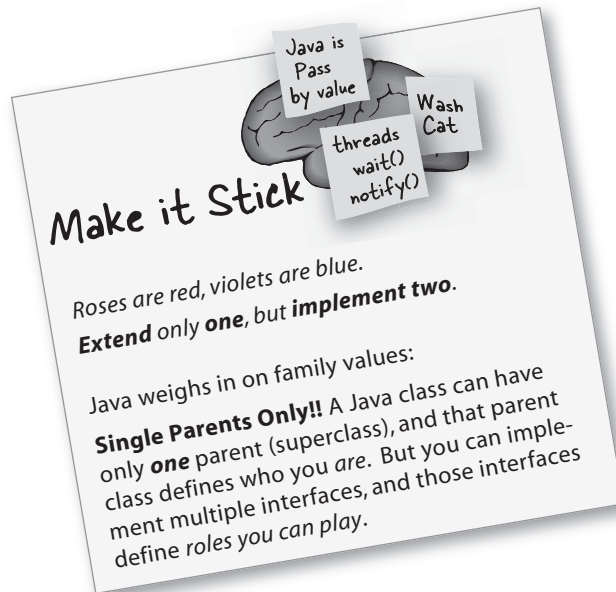
But when you use an *interface* as a polymorphic type (like an array of `Pets`), the objects can be from *anywhere* in the inheritance tree. The only requirement is that the objects are from a class that *implements* the interface. Allowing classes in different inheritance trees to implement a common interface is crucial in the Java API. Do you want an object to be able to save its state to a file? Implement the `Serializable` interface. Do you need objects to run

their methods in a separate thread of execution? Implement `Runnable`. You get the idea. You'll learn more about `Serializable` and `Runnable` in later chapters, but for now, remember that classes from *any* place in the inheritance tree might need to implement those interfaces. Nearly *any* class might want to be saveable or runnable.

Better still, a class can implement *multiple* interfaces!

A `Dog` object IS-A `Canine`, and IS-A `Animal`, and IS-A `Object`, all through inheritance. But a `Dog` IS-A `Pet` through interface implementation, and the `Dog` might implement other interfaces as well. You could say:

```
public class Dog extends Animal implements
    Pet, Saveable, Paintable { ... }
```



How do you know whether to make a class, a subclass, an *abstract* class, or an interface?

- Make a class that doesn't extend anything (other than `Object`) when your new class doesn't pass the IS-A test for any other type.
- Make a subclass (in other words, *extend* a class) only when you need to make a **more specific** version of a class and need to override or add new behaviors.
- Use an abstract class when you want to define a **template** for a group of subclasses, and you have at least *some* implementation code that all subclasses could use. Make the class abstract when you want to guarantee that nobody can make objects of that type.
- Use an interface when you want to define a **role** that other classes can play, regardless of where those classes are in the inheritance tree.

using `super`

Invoking the superclass version of a method

Q: What if you make a concrete subclass and you need to override a method, but you want the behavior in the superclass version of the method? In other words, what if you don't need to *replace* the method with an override, but you just want to *add* to it with some additional specific code.

A: Ahhh... think about the meaning of the word 'extends'. One area of good OO design looks at how to design concrete code that's *meant* to be overridden. In other words, you write method code in, say, an abstract class, that does work that's generic enough to support typical concrete implementations. But, the concrete code isn't enough to handle *all* of the subclass-specific work. So the subclass overrides the method and *extends* it by adding the rest of the code. The keyword `super` lets you invoke a superclass version of an overridden method, from within the subclass.

```
abstract class Report {  
    void runReport() {  
        // set-up report  
    }  
    void printReport() {  
        // generic printing  
    }  
}
```

superclass version of the method does important stuff that subclasses could use

```
class BuzzwordReport extends Report {  
  
    void runReport() {  
        super.runReport();  
        buzzwordCompliance();  
        printReport();  
    }  
    void buzzwordCompliance() {...}  
}
```

call superclass version, then come back and do some subclass-specific stuff

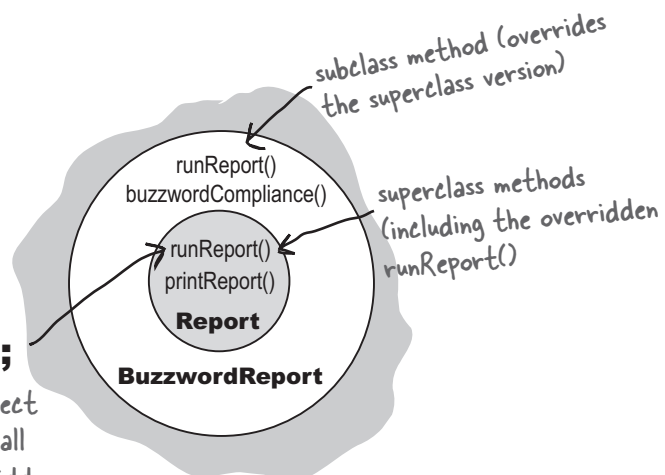
If method code inside a BuzzwordReport subclass says:

```
super.runReport();
```

the `runReport()` method inside the superclass `Report` will run

super.runReport();

A reference to the subclass object (`BuzzwordReport`) will always call the subclass version of an overridden method. That's polymorphism. But the subclass code can call `super.runReport()` to invoke the superclass version.



The `super` keyword is really a reference to the superclass portion of an object. When subclass code uses `super`, as in `super.runReport()`, the superclass version of the method will run.



BULLET POINTS

- ▶ When you don't want a class to be instantiated (in other words, you don't want anyone to make a new object of that class type) mark the class with the **abstract** keyword.
- ▶ An abstract class can have both abstract and non-abstract methods.
- ▶ If a class has even *one* abstract method, the class must be marked abstract.
- ▶ An abstract method has no body, and the declaration ends with a semicolon (no curly braces).
- ▶ All abstract methods must be implemented in the first concrete subclass in the inheritance tree.
- ▶ Every class in Java is either a direct or indirect subclass of class **Object** (java.lang.Object).
- ▶ Methods can be declared with Object arguments and/or return types.
- ▶ You can call methods on an object *only* if the methods are in the class (or interface) used as the *reference* variable type, regardless of the actual *object* type. So, a reference variable of type Object can be used only to call methods defined in class Object, regardless of the type of the object to which the reference refers.
- ▶ A reference variable of type Object can't be assigned to any other reference type without a *cast*. A cast can be used to assign a reference variable of one type to a reference variable of a subtype, but at runtime the cast will fail if the object on the heap is NOT of a type compatible with the cast.
Example: `Dog d = (Dog) x.getObject(aDog) ;`
- ▶ All objects come out of an ArrayList<Object> as type Object (meaning, they can be referenced only by an Object reference variable, unless you use a *cast*).
- ▶ Multiple inheritance is not allowed in Java, because of the problems associated with the "Deadly Diamond of Death". That means you can extend only one class (i.e. you can have only one immediate superclass).
- ▶ An interface is like a 100% pure abstract class. It defines *only* abstract methods.
- ▶ Create an interface using the **interface** keyword instead of the word **class**.
- ▶ Implement an interface using the keyword **implements**
Example: `Dog implements Pet`
- ▶ Your class can implement multiple interfaces.
- ▶ A class that implements an interface *must* implement all the methods of the interface, since **all interface methods are implicitly public and abstract**.
- ▶ To invoke the superclass version of a method from a subclass that's overridden the method, use the **super** keyword. Example: `super.runReport() ;`

interfaces and polymorphism

Q: There's still something strange here... you never explained how it is that **ArrayList<Dog>** gives back **Dog** references that don't need to be cast, yet the **ArrayList** class uses **Object** in its methods, not **Dog** (or **DotCom** or anything else). What's the special trick going on when you say **ArrayList<Dog>**?

A: You're right for calling it a special trick. In fact it is a special trick that **ArrayList<Dog>** gives back **Dogs** without you having to do any cast, since it looks like **ArrayList** methods don't know anything about **Dogs**, or any type besides **Object**.

The short answer is that *the compiler puts in the cast for you!* When you say **ArrayList<Dog>**, there is no special class that has methods to take and return **Dog** objects, but instead the **<Dog>** is a signal to the compiler that you want the compiler to let you put **ONLY Dog** objects in and to stop you if you try to add any other type to the list. And since the compiler stops you from adding anything but **Dogs** to the **ArrayList**, the compiler also knows that it's safe to cast anything that comes out of that **ArrayList** do a **Dog** reference. In other words, using **ArrayList<Dog>** saves you from having to cast the **Dog** you get back. But it's much more important than that... because remember, a cast can fail at runtime, and wouldn't you rather have your errors happen at compile time rather than, say, when your customer is using it for something critical?

But there's a lot more to this story, and we'll get into all the details in the **Collections** chapter.

exercise: What's the Picture?



Here's your chance to demonstrate your artistic abilities. On the left you'll find sets of class and interface declarations. Your job is to draw the associated class diagrams on the right. We did the first one for you. Use a dashed line for "implements" and a solid line for "extends".

Given:

1) `public interface Foo { }`
`public class Bar implements Foo { }`

2) `public interface Vinn { }`
`public abstract class Vout implements Vinn { }`

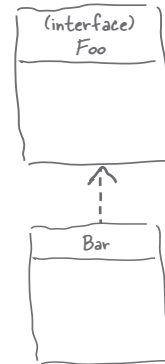
3) `public abstract class Muffie implements Whuffie { }`
`public class Fluffie extends Muffie { }`
`public interface Whuffie { }`

4) `public class Zoop { }`
`public class Boop extends Zoop { }`
`public class Goop extends Boop { }`

5) `public class Gamma extends Delta implements Epsilon { }`
`public interface Epsilon { }`
`public interface Beta { }`
`public class Alpha extends Gamma implements Beta { }`
`public class Delta { }`

What's the Picture?

1)



3)

2)

4)

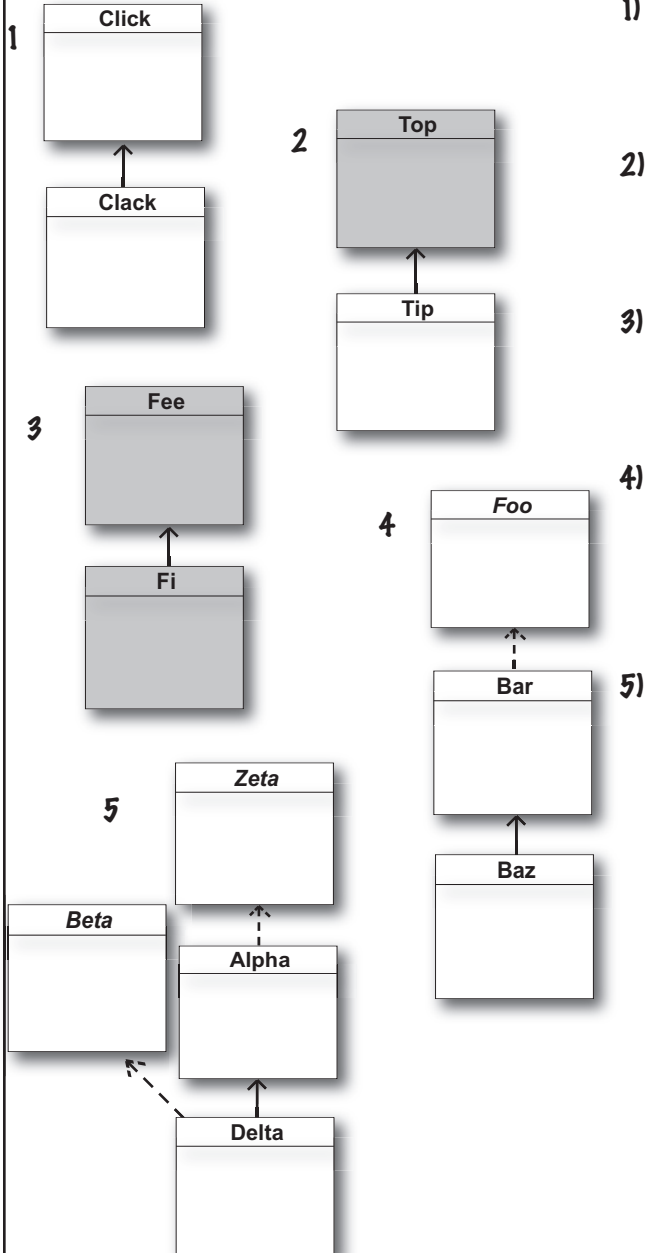
5)



On the left you'll find sets of class diagrams. Your job is to turn these into valid Java declarations. We did number 1 for you (and it was a tough one).

What's the Declaration?

Given:



1) `public class Click { }`
`public class Clack extends Click { }`

2)

3)

4)

5)

KEY



extends



implements

Clack

class

Clack

interface

Clack

abstract class

puzzle: Pool Puzzle



Pool
Puzzle

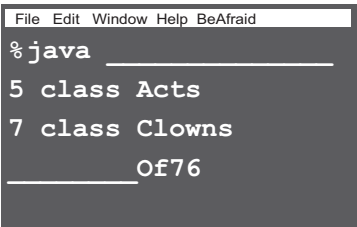


Your **job** is to take code snippets from the pool and place them into the blank lines in the code and output. You **may** use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to make a set of classes that will compile and run and produce the output listed.

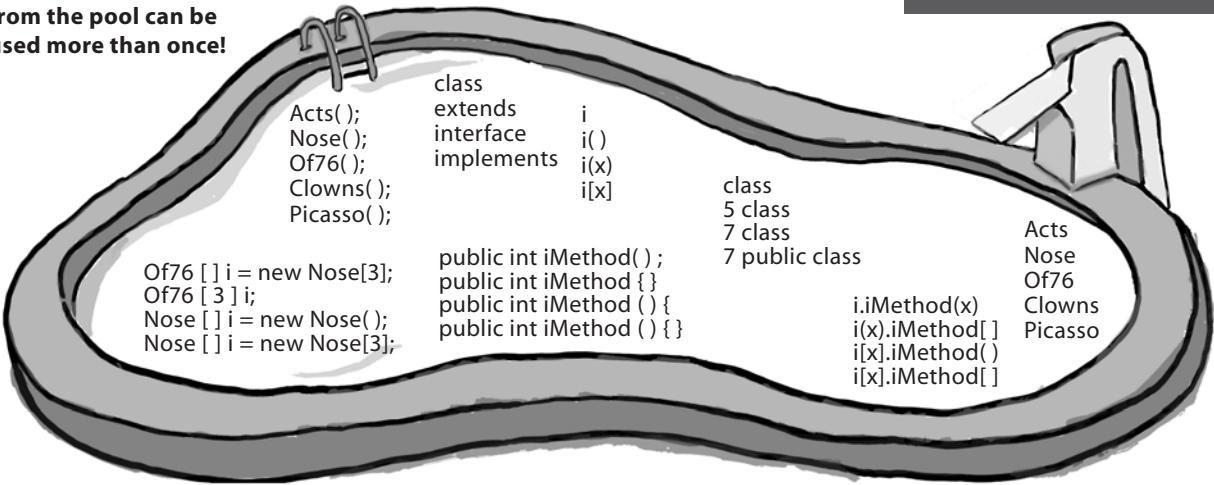
```
_____ Nose {  
    _____  
}  
  
abstract class Picasso implements _____{  
    _____  
    return 7;  
}  
  
class _____ { }  
  
class _____ {  
    _____  
    return 5;  
}  
}
```

```
public _____ extends Clowns {  
  
    public static void main(String [] args) {  
        _____  
        i[0] = new _____  
        i[1] = new _____  
        i[2] = new _____  
        for(int x = 0; x < 3; x++) {  
            System.out.println(_____  
                + " " + _____getClass( ) );  
        }  
    }  
}
```

Output



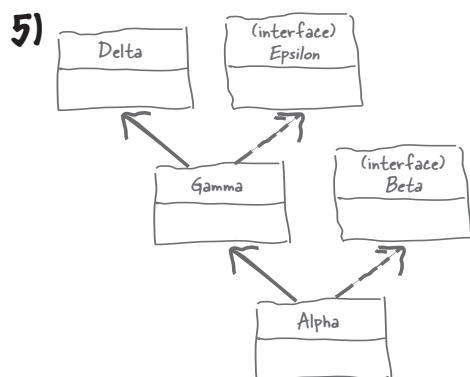
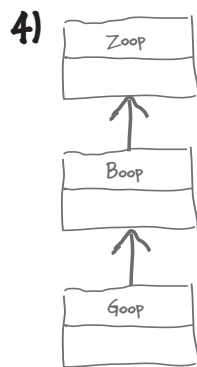
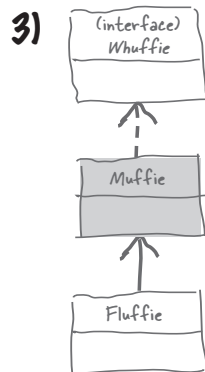
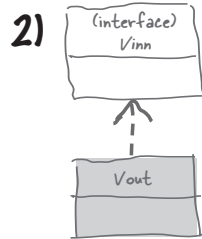
Note: Each snippet from the pool can be used more than once!





Exercise Solutions

What's the Picture ?



What's the Declaration ?

2) `public abstract class Top { }`
`public class Tip extends Top { }`

3) `public abstract class Fee { }`
`public abstract class Fi extends Fee { }`

4) `public interface Foo { }`
`public class Bar implements Foo { }`
`public class Baz extends Bar { }`

5) `public interface Zeta { }`
`public class Alpha implements Zeta { }`
`public interface Beta { }`
`public class Delta extends Alpha implements Beta { }`

puzzle solution



```
interface Nose {
    public int iMethod( );
}
abstract class Picasso implements Nose {
    public int iMethod( ){
        return 7;
    }
}
class Clowns extends Picasso { }

class Acts extends Picasso {
    public int iMethod( ){
        return 5;
    }
}
```

```
public class Of76 extends Clowns {
    public static void main(String [] args) {
        Nose [ ] i = new Nose [3];
        i[0] = new Acts( );
        i[1] = new Clowns( );
        i[2] = new Of76( );
        for(int x = 0; x < 3; x++) {
            System.out.println( i [x]. iMethod( )
                                + " " + i [x].getClass( ) );
        }
    }
}
```

Output

```
File Edit Window Help KillTheMime
%java Of76
5 class Acts
7 class Clowns
7 class Of76
```