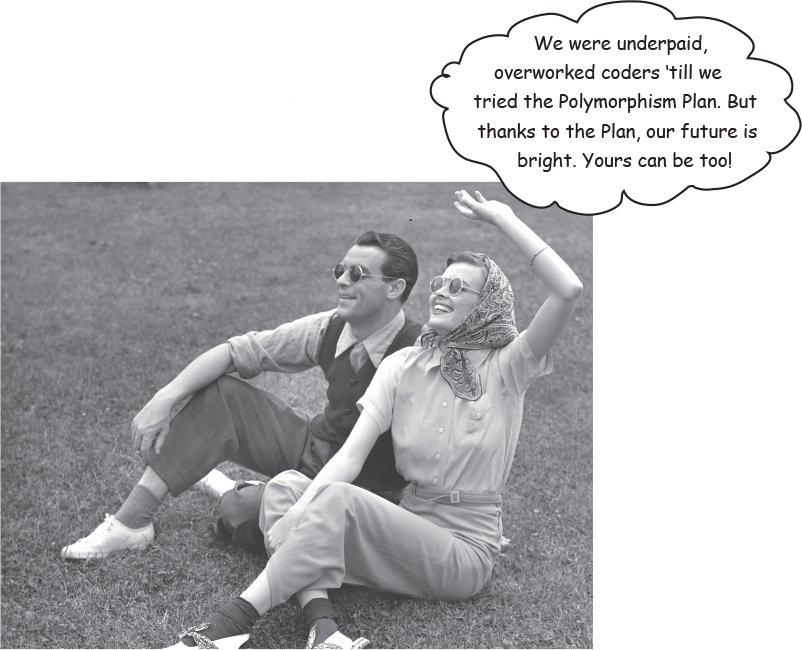
# Chapter 7. Inheritance and Polymorphism: Better Living in Objectville



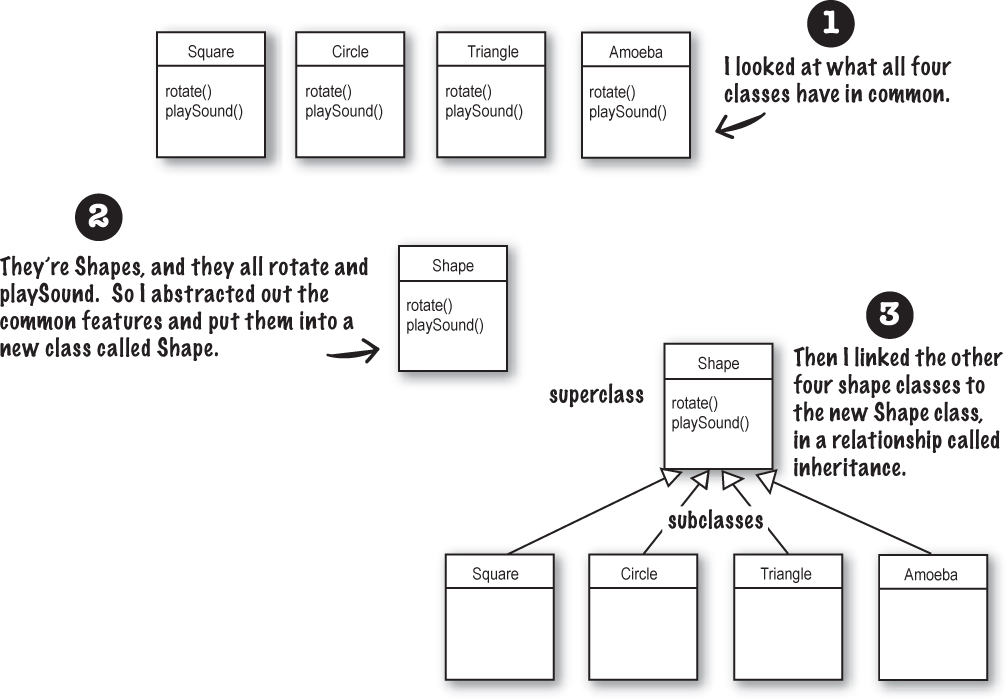
**Plan your programs with the future in mind.** If there were a way to write Java code such that you could take more vacations, how much would it be worth to you? What if you could write code that someone *else* could extend, **easily**? And if you could write code that was flexible, for those pesky надоедливый last-minute spec changes, would that be something you’re interested in? Then this is your lucky day. For just three easy payments of 60 minutes time, you can have all this. When you get on the Polymorphism Plan, you’ll learn the 5 steps to better class design, the 3 tricks to polymorphism, the 8 ways to make flexible code, and if you act now—a bonus lesson on the 4 tips for exploiting inheritance. Don’t delay, an offer this good will give you the design freedom and programming flexibility you deserve. It’s quick, it’s easy, and it’s available now. Start today, and we’ll throw in an extra level of abstraction!

**Chair Wars Revisited...**

***Remember way back in*** [***Chapter 2***](#_Chapter_2._Classes)***, when Laura (procedural programmer) and Brad (OO developer) were vying соперничал for the Aeron chair? Let’s look at a few pieces of that story to review the basics of inheritance.***

**LAURA:** You’ve got duplicated code! The rotate procedure is in all four Shape things. It’s a stupid design. You have to maintain four different rotate “methods”. How can that ever be good?

**BRAD:** Oh, I guess you didn’t see the final design. Let me show you how OO **inheritance** works, Laura.



You can read this as, **“Square inherits from Shape”**, **“Circle inherits from Shape”**, and so on. I removed rotate() and playSound() from the other shapes, so now there’s only one copy to maintain.

The Shape class is called the **superclass** of the other four classes. The other four are the **subclasses** of Shape. The subclasses inherit the methods of the superclass. In other words, *if the Shape class has the functionality, then the subclasses automatically get that same functionality.*

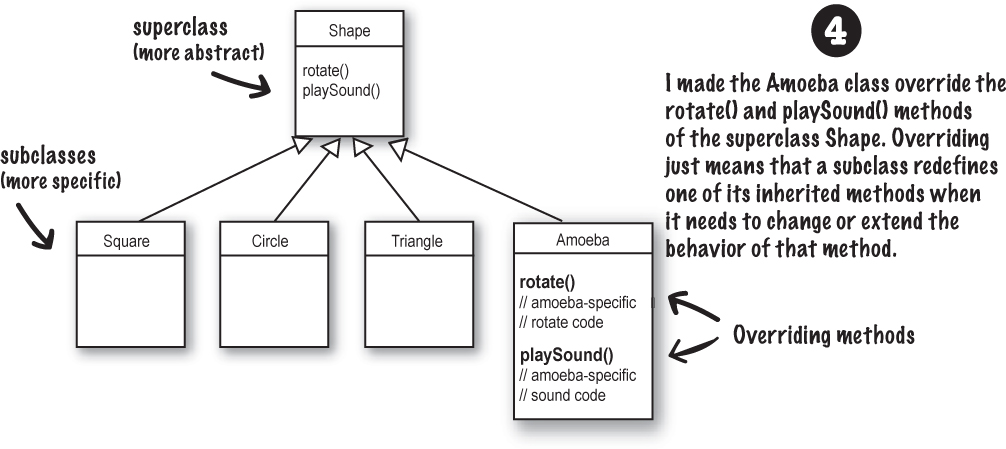
**What about the Amoeba rotate()?**



**LAURA:** Wasn’t that the whole problem here — that the amoeba shape had a completely different rotate and playSound procedure?

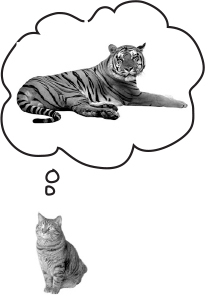
How can amoeba do something different if it *inherits* its functionality from the Shape class?

**BRAD:** That’s the last step. The Amoeba class ***overrides*** any methods of the Shape class that need specific amoeba behavior. Then at runtime, the JVM knows exactly which *rotate()* method to run when someone tells the Amoeba to rotate.



**Brain Power**

Images



How would you represent a house cat and a tiger, in an inheritance structure? Is a domestic cat a specialized version of a tiger? Which would be the subclass and which would be the superclass? Or are they both subclasses to some *other* class?

How would you design an inheritance structure? What methods would be overridden?

Think about it. *Before* you turn the page.

**Understanding Inheritance**

When you design with inheritance, you put common code in a class and then tell other more specific classes that the common (more abstract) class is their superclass. When one class inherits from another, **the subclass inherits from the superclass.**

In Java, we say that the **subclass** ***extends*** **the superclass**. An inheritance relationship means that the subclass inherits the **members** of the superclass. When we say “members of a class” we mean the instance variables and methods.

For example, if PantherMan is a subclass of SuperHero, the PantherMan class automatically inherits the instance variables and methods common to all superheroes including suit, tights, specialPower, useSpecialPowers(), and so on. But the PantherMan **subclass can add new methods and instance variables** of its own, and it **can override the methods it inherits from the superclass** SuperHero.



FriedEggMan doesn’t need any behavior that’s unique, so he doesn’t override any methods. The methods and instance variables in SuperHero are sufficient. PantherMan, though, has specific requirements for his suit and special powers, so useSpecialPower() and putOnSuit() are both overridden in the PantherMan class.

**Instance variables are not overridden** because they don’t need to be. They don’t define any special behavior, so a subclass can give an inherited instance variable any value it chooses. PantherMan can set his inherited tights трико to purple, while FriedEggMan sets his to white.

**An inheritance example:**



public class Doctor {

boolean worksAtHospital;

void treatPatient() {

// perform a checkup

}

}

public class FamilyDoctor extends Doctor {

boolean makesHouseCalls;

void giveAdvice() {

// give homespun advice

}

}

public class Surgeon extends Doctor {

void treatPatient() {

// perform surgery

}

void makeIncision() {

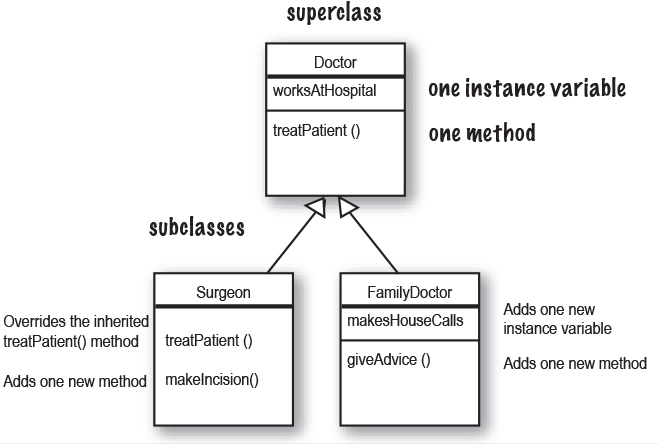
// make incision (yikes!)

}

}

**Sharpen your pencil**

image



How many instance variables does Surgeon have?\_\_\_\_\_\_\_\_\_

How many instance variables does FamilyDoctor have?\_\_\_\_\_\_\_\_\_

How many methods does Doctor have?\_\_\_\_\_\_\_\_\_

How many methods does Surgeon have?\_\_\_\_\_\_\_\_\_

How many methods does FamilyDoctor have?\_\_\_\_\_\_\_\_\_

Can a FamilyDoctor do treatPatient()?\_\_\_\_\_\_\_\_\_

Can a FamilyDoctor do makeIncision()?\_\_\_\_\_\_\_\_\_

**Let’s design the inheritance tree for an Animal simulation program**

Imagine you’re asked to design a simulation program that lets the user throw a bunch of different animals into an environment to see what happens. We don’t have to code the thing now, we’re mostly interested in the design.

We’ve been given a list of *some* of the animals that will be in the program, but not all. We know that each animal will be represented by an object, and that the objects will move around in the environment, doing whatever it is that each particular type is programmed to do.

***And we want other programmers to be able to add new kinds of animals to the program at any time.***

First we have to figure out the common, abstract characteristics that all animals have, and build those characteristics into a class that all animal classes can extend.

ImagesLook for objects that have common attributes and behaviors.

**What do these six types have in common? This helps you to abstract out behaviors. (step 2)**

**How are these types related? This helps you to define the inheritance tree relationships (step 4-5)**



**Using inheritance to avoid duplicating code in subclasses**

We have five ***instance variables: picture*** – the file name representing the JPEG of this animal

***food*** – the type of food this animal eats. Right now, there can be only two values: *meat* or *grass*.

***hunger*** – an int representing the hunger level of the animal. It changes depending on when (and how much) the animal eats.

***boundaries*** – values representing the height and width of the ‘space’ (for example, 640 x 480) that the animals will roam скитаться around in.

***location*** – the X and Y coordinates for where the animal is in the space.

We have four ***methods:***

***makeNoise*** () – behavior for when the animal is supposed to make noise.

***eat()*** – behavior for when the animal encounters its preferred food source, *meat* or *grass*.

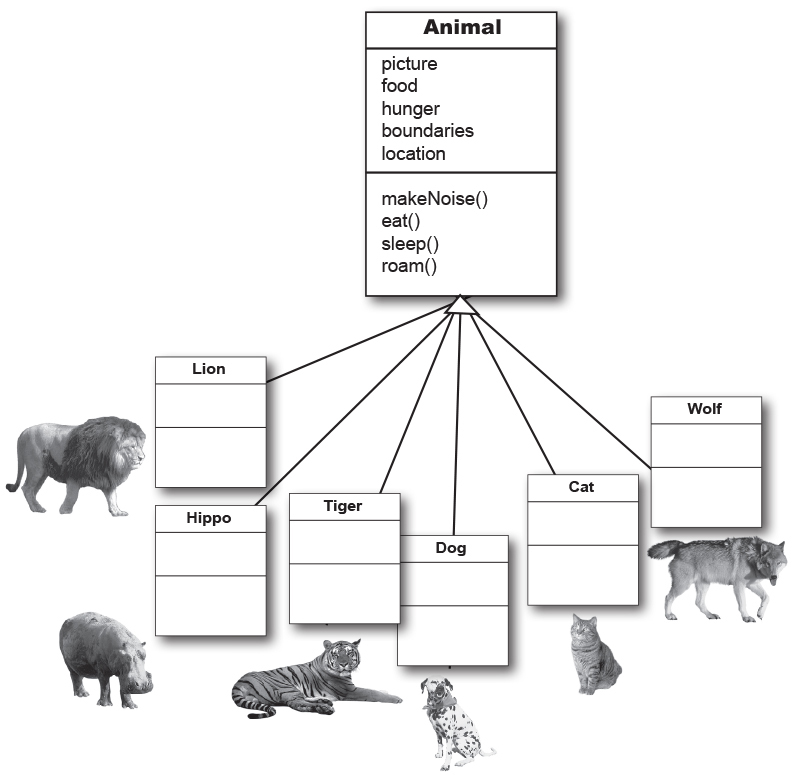
***sleep()*** – behavior for when the animal is considered asleep.

***roam()*** – behavior for when the animal is not eating or sleeping (probably just wandering around waiting to bump into a food source or a boundary).

ImagesDesign a class that represents the common state and behavior.

**These objects are all animals, so we’ll make a common superclass called Animal.**

**We’ll put in methods and instance variables that all animals might need.**



**Do all animals eat the same way?**

Assume that we all agree on one thing: the instance variables will work for *all* Animal types. A lion will have his own value for picture, food (we’re thinking *meat*), hunger, boundaries, and location. A hippo will have different *values* for his instance variables, but he’ll still have the same variables that the other Animal types have. Same with dog, tiger, and so on. But what about *behavior*?

ImagesDecide if a subclass needs behaviors (method implementations) that are specific to that particular subclass type.

**Looking at the Animal class, we decide that eat() and makeNoise() should be overridden by the individual subclasses.**

**Which methods should we override?**

Does a lion make the same **noise** as a dog? Does a cat **eat** like a hippo? Maybe in *your* version, but in ours, eating and making noise are Animal-type-specific. We can’t figure out how to code those methods in such a way that they’d work for any animal. OK, that’s not true. We could write the makeNoise() method, for example, so that all it does is play a sound file defined in an instance variable for that type, but that’s not very specialized. Some animals might make different noises for different situations (like one for eating, and another when bumping into an enemy, etc.)

So just as with the Amoeba overriding the Shape class rotate() method, to get more amoeba-specific (in other words, *unique*) behavior, we’ll have to do the same for our Animal subclasses.





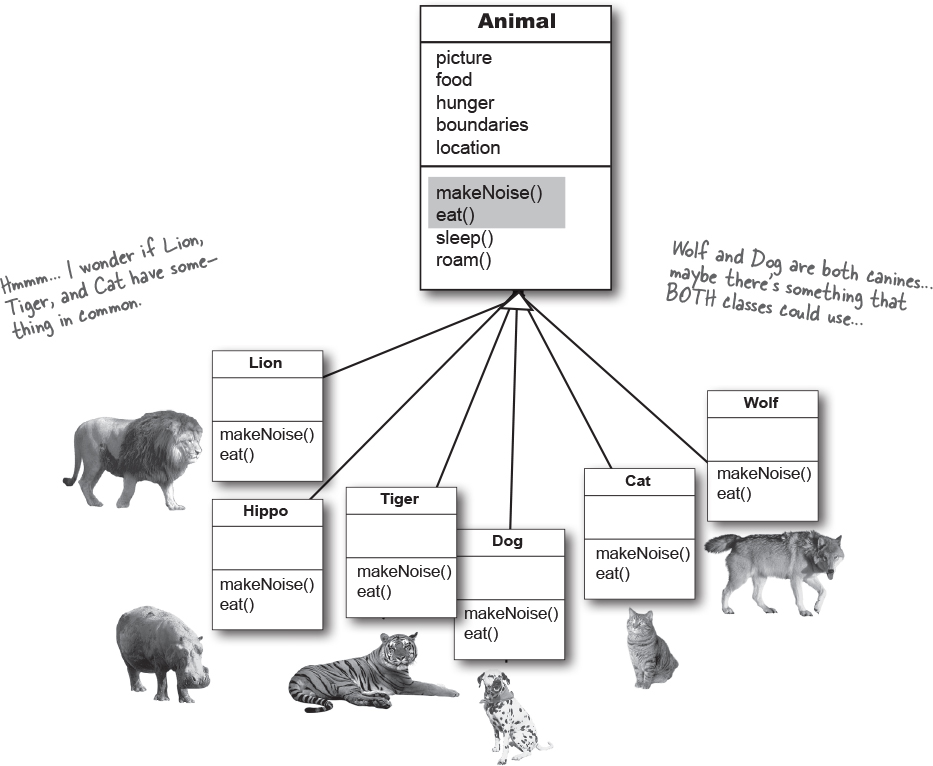
**Looking for more inheritance opportunities**

The class hierarchy is starting to shape up. We have each subclass override the *makeNoise()* and *eat()* methods, so that there’s no mistaking a Dog bark from a Cat meow (quite insulting to both parties). And a Hippo won’t eat like a Lion.

But perhaps there’s more we can do. We have to look at the subclasses of Animal, and see if two or more can be grouped together in some way, and given code that’s common to only *that* new group. Wolf and Dog have similarities. So do Lion, Tiger, and Cat.

ImagesLook for more opportunities to use abstraction, by finding two or more subclasses that might need common behavior.

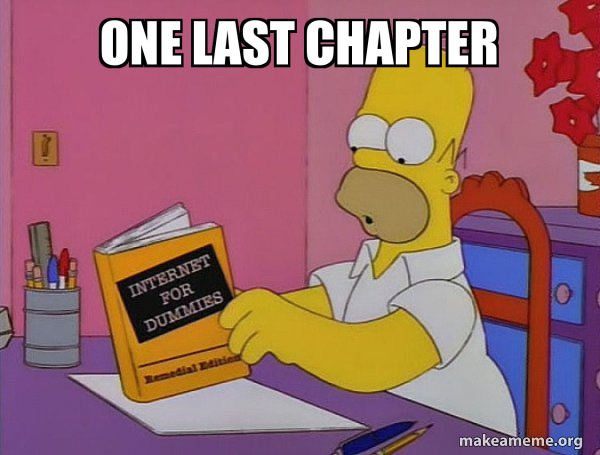
**We look at our classes and see that Wolf and Dog might have some behavior in common, and the same goes for Lion, Tiger, and Cat.**

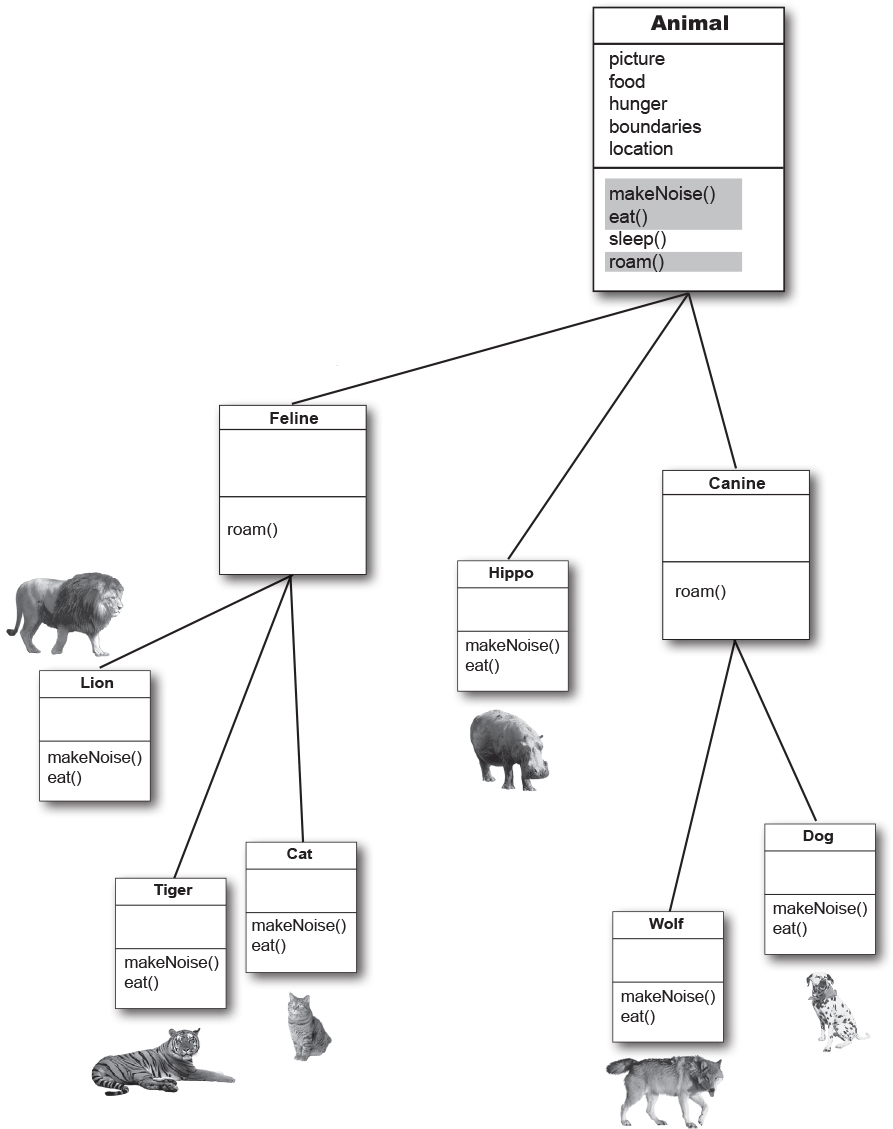


ImagesFinish the class hierarchy

Since animals already have an organizational hierarchy (the whole kingdom, genus род, phylum тип thing), we can use the level that makes the most sense for class design. We’ll use the biological “families” to organize the animals by making a Feline Кошачий class and a Canine class.

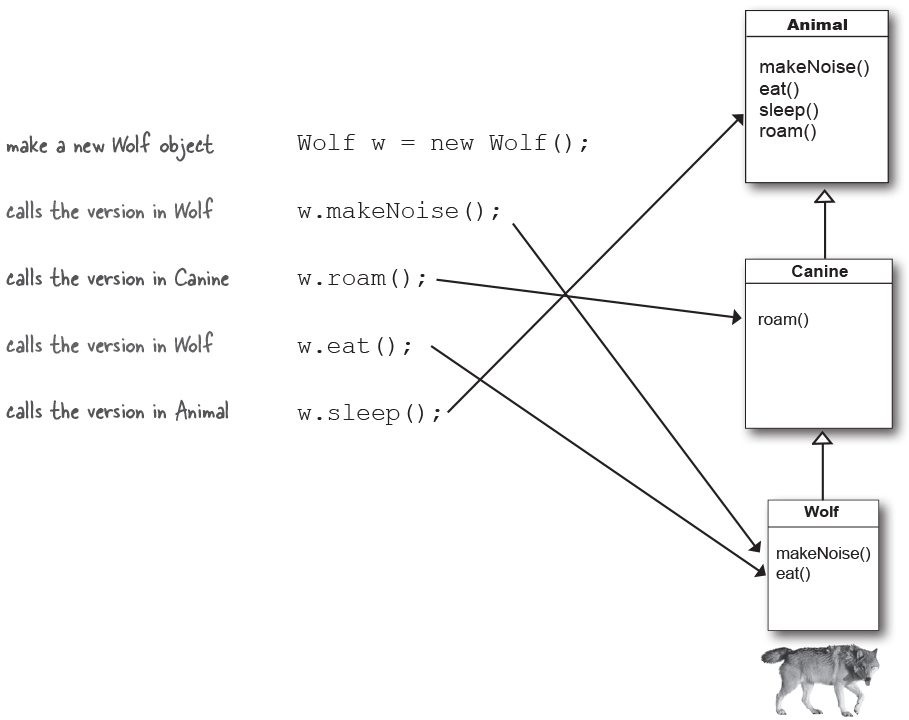
**We decide that Canines could use a common roam() method, because they tend to move in packs. We also see that Felines could use a common roam() method, because they tend to avoid others of their own kind. We’ll let Hippo continue to use its inherited roam() method— the generic one it gets from Animal.**

**So we’re done with the design for now; we’ll come back to it later in the chapter.**



**Which method is called?**

The Wolf class has four methods. One inherited from Animal, one inherited from Canine (which is actually an overridden version of a method in class Animal), and two overridden in the Wolf class. When you create a Wolf object and assign it to a variable, you can use the dot operator on that reference variable to invoke all four methods. But which *version* of those methods gets called?

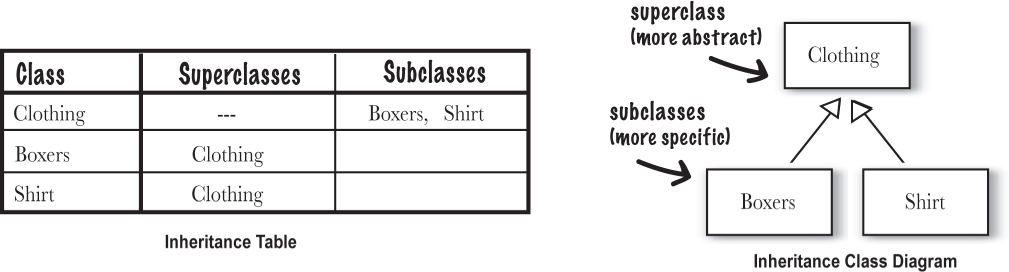


When you call a method on an object reference, you’re calling the most specific version of the method for that object type.

In other words, ***the lowest one wins!***

“Lowest” meaning lowest on the inheritance tree. Canine is lower than Animal, and Wolf is lower than Canine, so invoking a method on a reference to a Wolf object means the JVM starts looking first in the Wolf class. If the JVM doesn’t find a version of the method in the Wolf class, it starts walking back up the inheritance hierarchy until it finds a match.

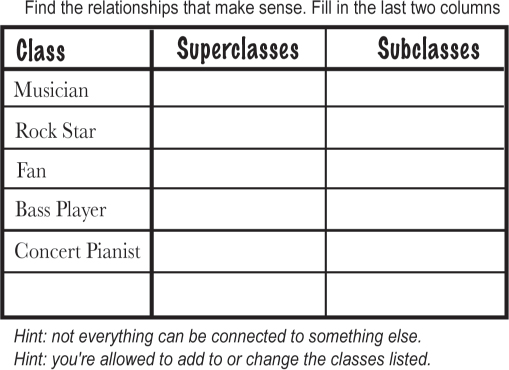
**Designing an Inheritance Tree**



**Sharpen your pencil**



Draw an inheritance diagram here.



**there are no Dumb Questions**

**Q: You said that the JVM starts walking up the inheritance tree, starting at the class type you invoked the method on (like the Wolf example on the previous page). But what happens if the JVM doesn’t ever find a match?**

**A:** Good question! But you don’t have to worry about that. The compiler guarantees that a particular method is callable for a specific reference type, but it doesn’t say (or care) from which *class* that method actually comes from at runtime. With the Wolf example, the compiler checks for a sleep() method, but doesn’t care that sleep() is actually defined in (and inherited from) class Animal. Remember that if a class *inherits* a method, it *has* the method.

*Where* the inherited method is defined (in other words, in which superclass it is defined) makes no difference to the compiler. But at runtime, **the JVM will always pick the right one**. And the right one means, ***the most specific version for that particular object.***

**Using IS-A and HAS-A**

Remember that when one class inherits from another, we say that the subclass *extends* the superclass. When you want to know if one thing should extend another, apply the IS-A test.

Triangle IS-A Shape, yeah, that works.

Cat IS-A Feline, that works too.

Surgeon IS-A Doctor, still good.

Tub Кадка extends Bathroom, sounds reasonable.

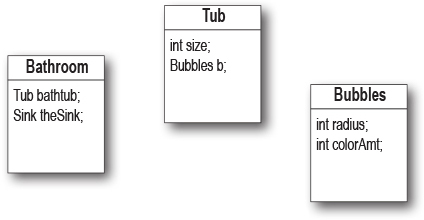
*Until you apply the IS-A test.*

To know if you’ve designed your types correctly, ask, “Does it make sense to say type X IS-A type Y?” If it doesn’t, you know there’s something wrong with the design, so if we apply the IS-A test, Tub IS-A Bathroom is definitely false.

What if we reverse it to Bathroom extends Tub? That still doesn’t work, Bathroom IS-A Tub doesn’t work.

Tub and Bathroom *are* related, but not through inheritance. Tub and Bathroom are joined by a HAS-A relationship. Does it make sense to say “Bathroom HAS-A Tub”? If yes, then it means that Bathroom has a Tub instance variable. In other words, Bathroom has a *reference* to a Tub, but Bathroom does not *extend* Tub and vice-versa.





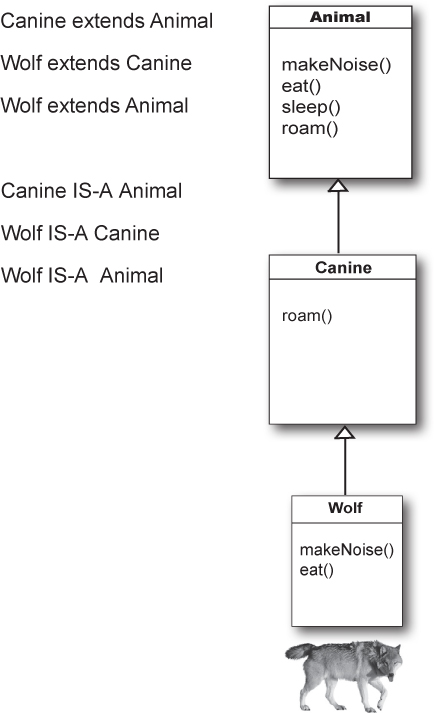
Bathroom HAS-A Tub and Tub HAS-A Bubbles. But nobody inherits from (extends) anybody else.

**But wait! There’s more!**

The IS-A test works *anywhere* in the inheritance tree. If your inheritance tree is well-designed, the IS-A test should make sense when you ask *any* subclass if it IS-A *any* of its supertypes.

**If class B extends class A, class B IS-A class A.**

**This is true anywhere in the inheritance tree. If class C extends class B, class C passes the IS-A test for both B and A.**



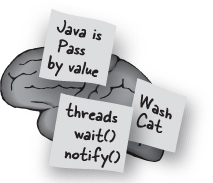
With an inheritance tree like the one shown here, you’re *always* allowed to say **“Wolf extends Animal”** or “**Wolf IS-A Animal”**. It makes no difference if Animal is the superclass of the superclass of Wolf. In fact, **as long as Animal is** ***somewhere*** **in the inheritance hierarchy above Wolf, Wolf IS-A Animal will always be true.**

The structure of the Animal inheritance tree says to the world:

“Wolf IS-A Canine, so Wolf can do anything a Canine can do. And Wolf IS-A Animal, so Wolf can do anything an Animal can do.”

It makes no difference if Wolf overrides some of the methods in Animal or Canine. As far as the world (of other code) is concerned, a Wolf can do those four methods. *How* he does them, or *in which class they’re overridden* makes no difference. A Wolf can makeNoise(), eat(), sleep(), and roam() because a Wolf extends from class Animal.

**Make it Stick**



*Roses are red, violets are blue.*

***Square*** *is-a* ***Shape****, the reverse isn’t true.*

*Roses are red, violets are dear.*

***Beer*** *is-a* ***Drink****, but not all* ***drinks*** *are* ***beer****.*

OK, your turn. Make one that shows the oneway-ness of the IS-A relationship. Remember, if X ***extends*** Y, X ***IS-A*** Y must make sense.

**How do you know if you’ve got your inheritance right?**

There’s obviously more to it than what we’ve covered so far, but we’ll look at a lot more OO issues in the next chapter (where we eventually refine очищать and improve on some of the design work we did in *this* chapter).

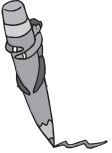
For now, though, a good guideline is to use the IS-A test. If “X IS-A Y” makes sense, both classes (X and Y) should probably live in the same inheritance hierarchy. Chances are, they have the same or overlapping behaviors.

**Keep in mind that the inheritance IS-A relationship works in only one direction!**

Triangle IS-A Shape makes sense, so you can have Triangle extend Shape.

But the reverse—Shape IS-A Triangle—does *not* make sense, so Shape should not extend Triangle. Remember that the IS-A relationship implies that if X IS-A Y, then X can do anything a Y can do (and possibly more).

**Sharpen your pencil**



Put a check next to the relationships that make sense.

Images**Oven extends Kitchen**

Images**Guitar extends Instrument**

Images**Person extends Employee**

Images**Ferrari extends Engine**

Images**FriedEgg extends Food**

Images**Beagle Гончая extends Pet**

Images**Container extends Jar**

Images**Metal extends Titanium**

Images**GratefulDead extends Band**

Images**Blonde extends Smart**

Images**Beverage extends Martini**

*Hint: apply the IS-A test*

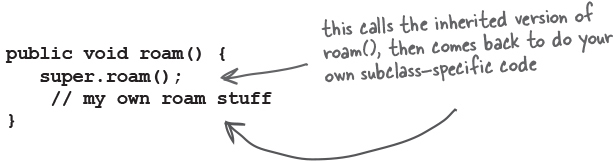
**there are no Dumb Questions**

**Q: So we see how a subclass gets to inherit a superclass method, but what if the superclass wants to use the subclass version of the method?**

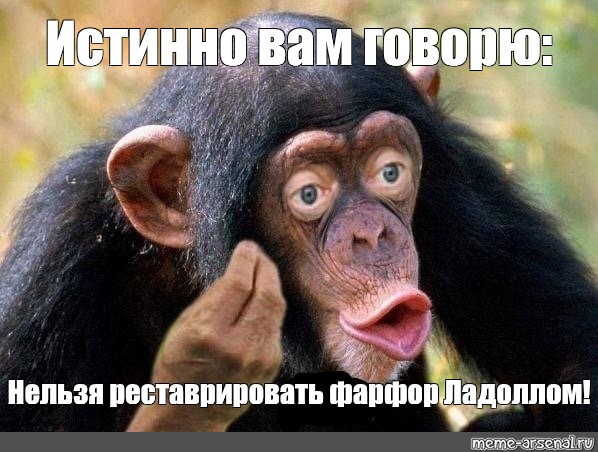
**A**: A superclass won’t necessarily *know* about any of its subclasses. You might write a class and much later someone else comes along and extends it. But even if the superclass creator does know about (and wants to use) a subclass version of a method, there’s no sort of *reverse* or *backwards* inheritance. Think about it, children inherit from parents, not the other way around.

**Q: In a subclass, what if I want to use BOTH the superclass version and my overriding subclass version of a method? In other words, I don’t want to completely** ***replace*** **the superclass version, I just want to add more stuff to it.**

**A**: You can do this! And it’s an important design feature. Think of the word “extends” as meaning, “I want to *extend* the functionality of the superclass”.



You can design your superclass methods in such a way that they contain method implementations that will work for any subclass, even though the subclasses may still need to ‘append’ more code. In your subclass overriding method, you can call the superclass version using the keyword **super**. It’s like saying, “first go run the superclass version, then come back and finish with my own code...”

**Who gets the Porsche, who gets the porcelain фарфор? (how to know what a subclass can inherit from its superclass)**



A subclass inherits members of the superclass. Members include instance variables and methods, although later in this book we’ll look at other inherited members. A superclass can choose whether or not it wants a subclass to inherit a particular member by the level of access the particular member is given.

There are four access levels that we’ll cover in this book. Moving from most restrictive to least, the four access levels are:



**Access levels control** ***who sees what,*** and are crucial to having well-designed, robust крепкий Java code. For now we’ll focus just on public and private. The rules are simple for those two:

***public members******are******inherited******private members are******not******inherited***

When a subclass inherits a member, it is ***as if the subclass defined the member itself***. In the Shape example, Square inherited the rotate() and playSound() methods and to the outside world (other code) the Square class simply *has* a rotate() and playSound() method.

The members of a class include the variables and methods defined in the class plus anything inherited from a superclass.

**Note**

Note: get more details about default and protected in appendix B.

**When designing with inheritance, are you using or abusing Злоупотребление?**

Although some of the reasons behind these rules won’t be revealed until later in this book, for now, simply *knowing* a few rules will help you build a better inheritance design.

**DO** use inheritance when one class is a more specific type of a superclass. Example: Willow Ива *is a* more specific type of Tree, so Willow *extends* Tree makes sense.

**DO** consider inheritance when you have behavior (implemented code) that should be shared among multiple classes of the same general type. Example: Square, Circle, and Triangle all need to rotate and play sound, so putting that functionality in a superclass Shape might make sense, and makes for easier maintenance and extensibility растяжимость. Be aware, however, that while inheritance is one of the key features of object-oriented programming, it’s not necessarily the best way to achieve behavior reuse. It’ll get you started, and often it’s the right design choice, but design patterns will help you see other more subtle тонкий and flexible options. If you don’t know about design patterns, a good follow-on to this book would be *Head First Design Patterns.*

**DO NOT** use inheritance just so that you can reuse code from another class, if the relationship between the superclass and subclass violate either of the above two rules. For example, imagine you wrote special printing code in the Animal class and now you need printing code in the Potato class. You might think about making Potato extend Animal so that Potato inherits the printing code. That makes no sense! A Potato is *not* an Animal! (So the printing code should be in a Printer class, that all printable objects can take advantage of via a HAS-A relationship.)

**DO NOT** use inheritance if the subclass and superclass do not pass the IS-A test. Always ask yourself if the subclass IS-A more specific type of the superclass. Example: Tea IS-A Beverage makes sense. Beverage IS-A Tea does not.

### Bullet Points

A subclass *extends* a superclass.

A subclass inherits all *public* instance variables and methods of the superclass, but does not inherit the *private* instance variables and methods of the superclass.

Inherited methods *can* be overridden; instance variables *cannot* be overridden (although they can be *redefined* in the subclass, but that’s not the same thing, and there’s almost never a need to do it.)

Use the IS-A test to verify that your inheritance hierarchy is valid. If X *extends* Y, then X *IS-A* Y must make sense.

The IS-A relationship works in only one direction. A Hippo is an Animal, but not all Animals are Hippos.

When a method is overridden in a subclass, and that method is invoked on an instance of the subclass, the overridden version of the method is called. (*The lowest one wins*.)

If class B extends A, and C extends B, class B IS-A class A, and class C IS-A class B, and class C also IS-A class A.

**So what does all this inheritance really buy you?**

You get a lot of OO mileage by designing with inheritance. You can get rid of duplicate code by abstracting out the behavior common to a group of classes, and sticking that code in a superclass. That way, when you need to modify it, you have only one place to update, and *the change is magically reflected in all the classes that inherit that behavior.*

Well, there’s no magic involved, but it *is* pretty simple: make the change and compile the class again. That’s it. **You don’t have to touch the subclasses!**

**Just deliver the newly-changed superclass, and all classes that extend it will automatically use the new version.**

A Java program is nothing but a pile of classes, so the subclasses don’t have to be recompiled in order to use the new version of the superclass. As long as the superclass doesn’t *break* anything for the subclass, everything’s fine. (We’ll discuss what the word ‘break’ means in this context, later in the book. For now, think of it as modifying something in the superclass that the subclass is depending on, like a particular method’s arguments or return type, or method name, etc.)

Images**You avoid duplicate code.**

Put common code in one place, and let the subclasses inherit that code from a superclass. When you want to change that behavior, you have to modify it in only one place, and everybody else (i.e. all the subclasses) see the change.

Images**You define a common protocol for a group of classes.**



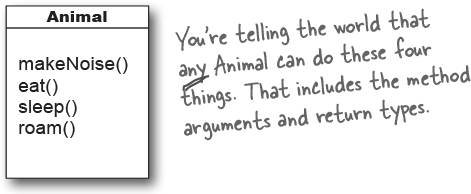
**Inheritance lets you guarantee that all classes grouped under a certain supertype have all the methods that the supertype has.\***

**In other words, you define a common protocol for a set of classes related through inheritance.**

When you define methods in a superclass, that can be inherited by subclasses, you’re announcing a kind of protocol to other code that says, “All my subtypes (i.e. subclasses) can do these things, with these methods that look like this...”

In other words, you establish устанавливать a *contract*.

Class Animal establishes a common protocol for all Animal subtypes:



And remember, when we say *any Animal*, we mean Animal *and any class that extends from Animal*. Which again means, *any class that has Animal somewhere above it in the inheritance hierarchy*.

But we’re not even at the really cool part yet, because we saved the best—*polymorphism*—for last.

When you define a supertype for a group of classes, *any subclass of that supertype can be substituted замещенный where the supertype is expected.*

Say, what?

Don’t worry, we’re nowhere near done explaining it. Two pages from now, you’ll be an expert.

\*When we say “all the methods” we mean “all the *inheritable* methods”, which for now actually means, “all the *public* methods”, although later we’ll refine уточним that definition a bit more.

**And I care because...**

Because you get to take advantage of polymorphism.

**Which matters to me because...**

Because you get to refer to a subclass object using a reference declared as the supertype.

**And that means to me...**

You get to write really flexible code. Code that’s cleaner (more efficient, simpler). Code that’s not just easier to *develop*, but also much, much easier to *extend*, in ways you never imagined at the time you originally wrote your code.

That means you can take that tropical vacation while your co-workers update the program, and your co-workers might not even need your source code.

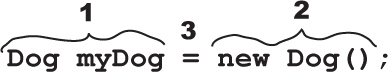
You’ll see how it works on the next page.

We don’t know about you, but personally, we find the whole tropical vacation thing particularly motivating.



**To see how polymorphism works, we have to step back and look at the way we normally declare a reference and create an object...**

**The 3 steps of object declaration and assignment**



ImagesDeclare a reference variable

**Dog myDog = new Dog();**

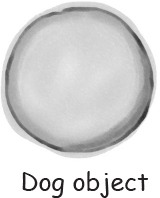
Tells the JVM to allocate space for a reference variable. The reference variable is, forever, of type Dog. In other words, a remote control that has buttons to control a Dog, but not a Cat or a Button or a Socket Розетка.



ImagesCreate an object

**Dog myDog = new Dog();**

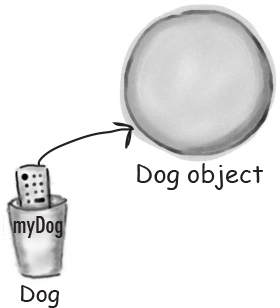
Tells the JVM to allocate space for a new Dog object on the garbage collectible heap.



ImagesLink the object and the reference

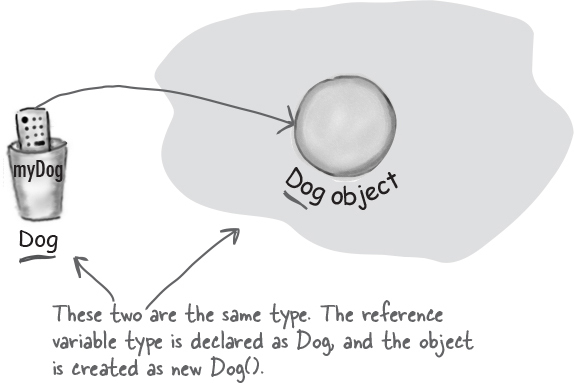
**Dog myDog = new Dog();**

Assigns the new Dog to the reference variable myDog. In other words, ***program the remote control.***



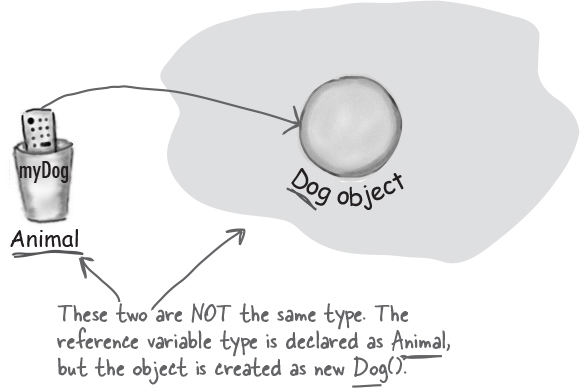
**The important point is that the reference type AND the object type are the same.**

**In this example, both are Dog.**

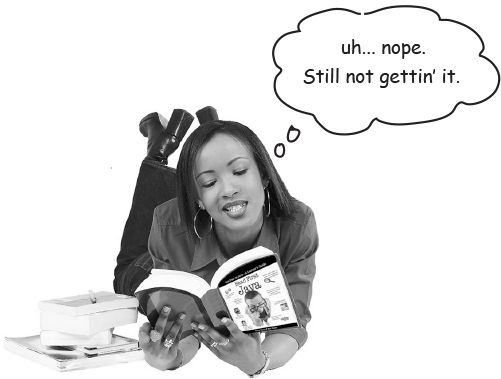


**But with polymorphism, the reference type and the object type can be different.**

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

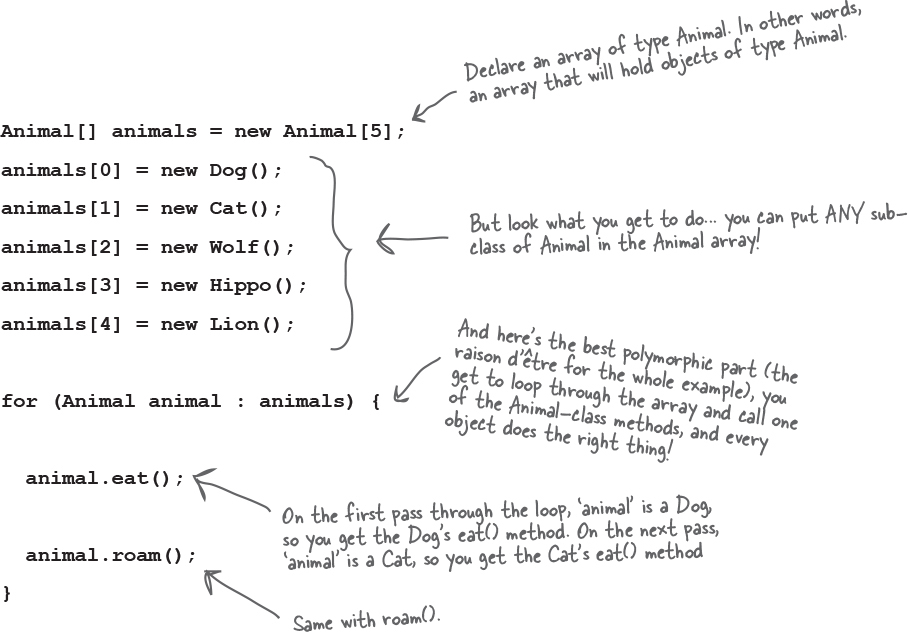


**With polymorphism, the reference type can be a superclass of the actual object type. uh... nope.**



When you declare a reference variable, any object that passes the IS-A test for the type of the reference can be assigned to that variable. In other words, anything that *extends* the declared reference variable type can be *assigned* to the reference variable. ***This lets you do things like make polymorphic arrays.***

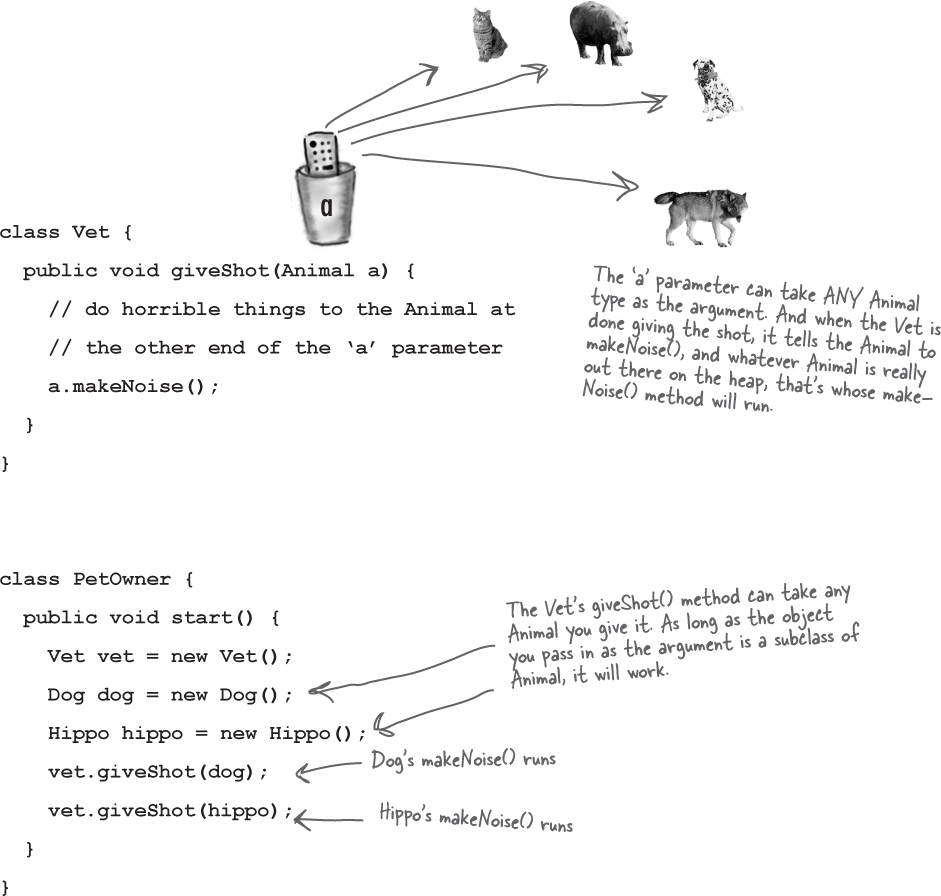
**OK, OK maybe an example will help.**

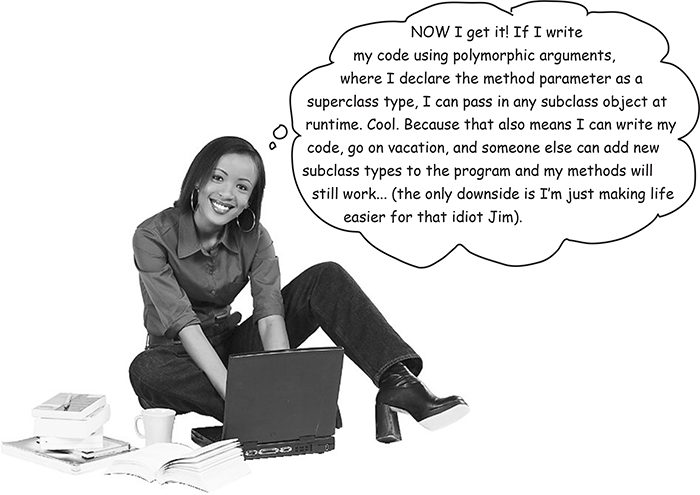


**But wait! There’s more!**

**You can have polymorphic arguments and return types.**

If you can declare a reference variable of a supertype, say, Animal, and assign a subclass object to it, say, Dog, think of how that might work when the reference is an argument to a method...





**With polymorphism, you can write code that doesn’t have to change when you introduce new subclass types into the program.**

Remember that Vet class? If you write that Vet class using arguments declared as type *Animal*, your code can handle any Animal *subclass*. That means if others want to take advantage of your Vet class, all they have to do is make sure *their* new Animal types extend class Animal. The Vet methods will still work, even though the Vet class was written without any knowledge of the new Animal subtypes the Vet will be working on.

**Brain Power**

Images

Why is polymorphism guaranteed to work this way? Why is it always safe to assume that any *subclass* type will have the methods you think you’re calling on the *superclass* type (the superclass reference type you’re using the dot operator on)?

**there are no Dumb Questions**

**Q: Are there any practical limits on the levels of subclassing? How deep can you go?**

**A:** If you look in the Java API, you’ll see that most inheritance hierarchies are wide but not deep. Most are no more than one or two levels deep, although there are exceptions (especially in the GUI classes). You’ll come to realize that it usually makes more sense to keep your inheritance trees shallow мелкий, but there isn’t a hard limit (well, not one that you’d ever run into).

**Q: Hey, I just thought of something... if you don’t have access to the source code for a class, but you want to change the way a method of that class works, could you use subclassing to do that? To extend the “bad” class and override the method with your own better code?**

**A:** Yep. That’s one cool feature of OO, and sometimes it saves you from having to rewrite the class from scratch, or track down the programmer who hid the source code.

**Q: Can you extend** ***any*** **class? Or is it like class members where if the class is private you can’t inherit it...**

**A:** There’s no such thing as a private class, except in a very special case called an *inner* class, that we haven’t looked at yet. But there *are* three things that can prevent a class from being subclassed.

The first is access control. Even though a class *can’t* be marked private, a class *can* be non-public (what you get if you don’t declare the class as public). A non-public class can be subclassed only by classes in the same package as the class. Classes in a different package won’t be able to subclass (or even *use*, for that matter) the non-public class.

The second thing that stops a class from being subclassed is the keyword modifier final. A final class means that it’s the end of the inheritance line. Nobody, ever, can extend a final class.

The third issue is that if a class has only private constructors (we’ll look at constructors in [Chapter 9](#_Chapter_9._Constructors)), it can’t be subclassed.

**Q: Why would you ever want to make a final class? What advantage would there be in preventing a class from being subclassed?**

**A:** Typically, you won’t make your classes final. But if you need security — the security of knowing that the methods will always work the way that you wrote them (because they can’t be overridden), a final class will give you that. A lot of classes in the Java API are final for that reason. The String class, for example, is final because, well, imagine the havoc опустошение if somebody came along and changed the way Strings behave!

**Q: Can you make a** ***method*** **final, without making the whole** ***class*** **final?**

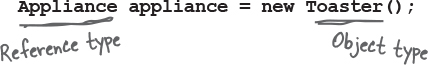
**A:** If you want to protect a specific method from being @Overridden, mark the *method* with the final modifier. Mark the whole *class* as final if you want to guarantee that *none* of the methods in that class will ever be @Overridden.

**Keeping the contract: rules for @Overriding**

When you override a method from a superclass, you’re agreeing to fulfill the contract. The contract that says, for example, “I take no arguments and I return a boolean.” In other words, the arguments and return types of your overriding method must look to the outside world *exactly* like the overridden method in the superclass.

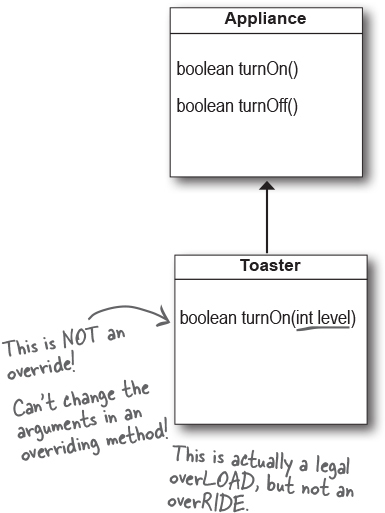
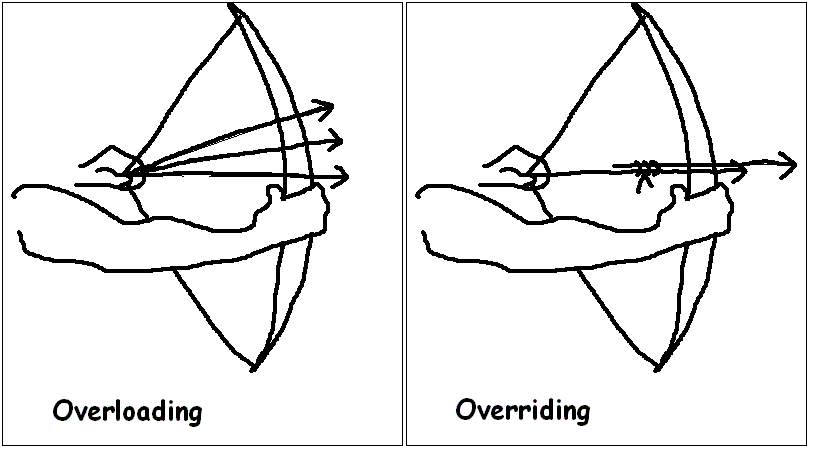
**The methods** ***are*** **the contract.**

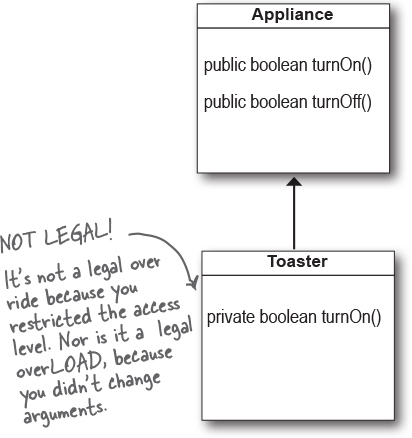
If polymorphism is going to work, the Toaster’s version of the overridden method from Appliance has to work at runtime. Remember, the compiler looks at the reference type to decide whether you can call a particular method on that reference.



With an Appliance reference to a Toaster, the compiler cares only if class *Appliance* has the method you’re invoking on an Appliance reference. But at runtime, the JVM does not look at the **reference** type (*Appliance*) but at the actual *Toaster* **object** on the heap.

So if the compiler has already approved the method call, the only way it can work is if the overriding method has the same arguments and return types. Otherwise, someone with an Appliance reference will call turnOn() as a no-arg method, even though there’s a version in Toaster that takes an int. Which one is called at runtime? The one in Appliance. In other words, ***the turnOn(int level) method in Toaster is not an override!***





Images**Arguments must be the same, and return types must be compatible.**

The contract of superclass defines how other code can use a method. Whatever the superclass takes as an argument, the subclass overriding the method must use that same argument. And whatever the superclass declares as a return type, the overriding method must declare either the same type, or a subclass type. Remember, a subclass object is guaranteed to be able to do anything its superclass declares, so it’s safe to return a subclass where the superclass is expected.

Images**The method can’t be less accessible.**

That means the access level must be the same, or friendlier. You can’t, for example, override a public method and make it private. What a shock that would be to the code invoking what it thinks (at compile time) is a public method, if suddenly at runtime the JVM slammed Захлопнул the door shut because the overriding version called at runtime is private!

So far we’ve learned about two access levels: private and public. The other two are in appendix B. There’s also another rule about overriding related to exception handling, but we’ll wait until the chapter on exceptions (Risky Behavior) to cover that.

**Overloading a method**

Method overloading is nothing more than having two methods with the same name but different argument lists. Period. There’s no polymorphism involved with overloaded methods!

Overloading lets you make multiple versions of a method, with different argument lists, for convenience to the callers. For example, if you have a method that takes only an int, the calling code has to convert, say, a double into an int before calling your method. But if you overloaded the method with another version that takes a double, then you’ve made things easier for the caller. You’ll see more of this when we look into constructors in the object lifecycle chapter.

Since an overloading method isn’t trying to fulfill the polymorphism contract defined by its superclass, overloaded methods have much more flexibility.

**Note**

**An overloaded method is just a different method that happens to have the same method name. It has nothing to do with inheritance and polymorphism. An overloaded method is NOT the same as an overridden method.**

Images**The return types can be different.**

You’re free to change the return types in overloaded methods, as long as the argument lists are different.

Images**You can’t change ONLY the return type.**

If only the return type is different, it’s not a valid over*load*—the compiler will assume you’re trying to over*ride* the method. And even *that* won’t be legal unless the return type is a subtype of the return type declared in the superclass. To overload a method, you MUST change the argument list, although you *can* change the return type to anything.

Images**You can vary the access levels in any direction.**

You’re free to overload a method with a method that’s more restrictive. It doesn’t matter, since the new method isn’t obligated to fulfill the contract of the overloaded method.

**Legal examples of method overloading:**

public class Overloads {

String uniqueID;

public **int** addNums**(int a, int b)** {

return a + b;

}

public **double** addNums**(double a, double b)** {

return a + b;

}

public void setUniqueID**(String theID)** {

// lots of validation code, and then:

uniqueID = theID;

}

public void setUniqueID**(int ssNumber)** {

String numString = "" + ssNumber;

setUniqueID(numString);

}

}

**Exercise**



**Mixed Messages**

image

A short Java program is listed below. One block of the program is missing! Your challenge is to match the candidate block of code (on the left), with the output that you’d see if the block were inserted. Not all the lines of output will be used, and some of the lines of output might be used more than once. Draw lines connecting the candidate blocks of code with their matching command-line output.

**the program:**

**class A {**

**int ivar = 7;**

**void m1() {**

**System.out.print("A's m1, ");**

**}**

**void m2() {**

**System.out.print("A's m2, ");**

**}**

**void m3() {**

**System.out.print("A's m3, ");**

**}**

**}**

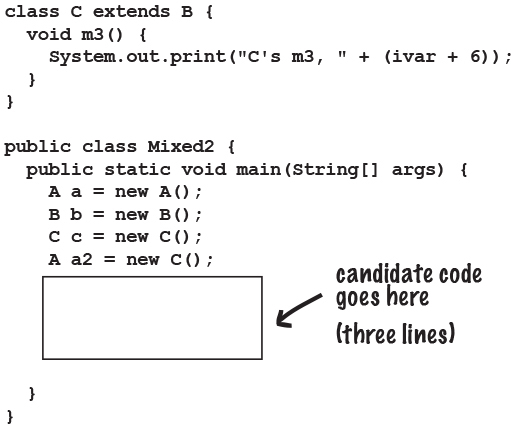
**class B extends A {**

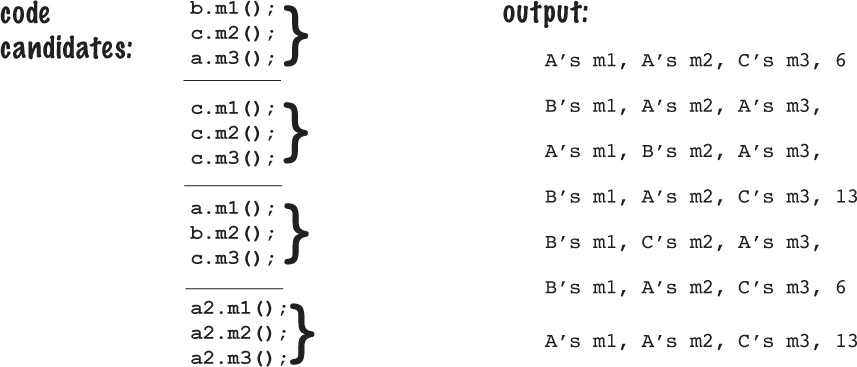
**void m1() {**

**System.out.print("B's m1, ");**

**}**

**}**





**BE the Compiler**



**Which of the A-B pairs of methods listed on the right, if inserted into the classes on the left, would compile and produce the output shown? (The A method inserted into class Monster, the B method inserted into class Vampire.)**



**Pool Puzzle**





Your ***job*** is to take code snippets from the pool and place them into the blank lines in the code. You may use the same snippet more than once, and you might not need to use all the snippets. Your ***goal*** is to make a set of classes that will compile and run together as a program. Don’t be fooled – this one’s harder than it looks.

public class Rowboat \_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_ {

public \_\_\_\_\_\_\_\_\_\_\_ rowTheBoat() {

System.out.print("stroke natasha");

}

}

public class \_\_\_\_\_\_\_\_ {

private int \_\_\_\_\_\_\_\_\_\_ ;

\_\_\_\_\_\_\_ void \_\_\_\_\_\_\_\_\_ ( \_\_\_\_\_\_ ) {

length = len;

}

public int getLength() {

\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_ ;

}

public \_\_\_\_\_\_\_\_\_\_\_ move() {

System.out.print("\_\_\_\_\_\_\_\_\_\_\_");

}

}

public class TestBoats {

\_\_\_\_\_\_ \_\_\_\_\_\_ \_\_\_\_\_\_\_ main(String[] args){

\_\_\_\_\_\_\_\_\_ b1 = new Boat();

Sailboat b2 = new \_\_\_\_\_\_\_\_\_\_();

Rowboat \_\_\_\_\_\_\_\_ = new Rowboat();

b2.setLength(32);

b1.\_\_\_\_\_\_\_\_\_\_();

b3.\_\_\_\_\_\_\_\_\_\_();

\_\_\_\_\_\_\_.move();

}

}

public class \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_ Boat {

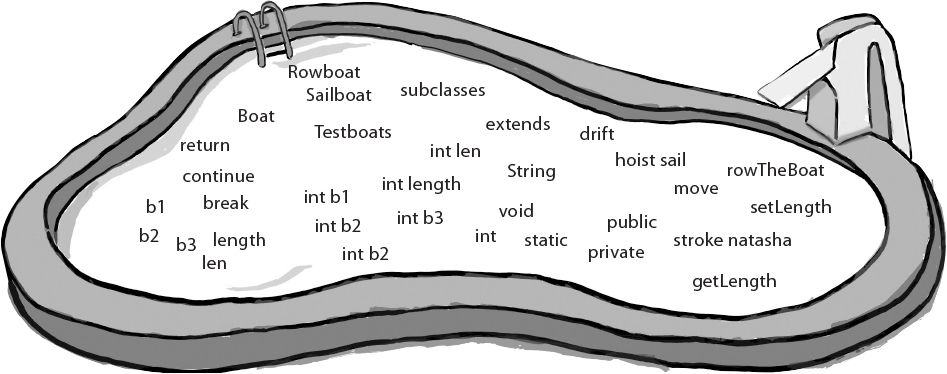
public \_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_() {

System.out.print("\_\_\_\_\_\_\_\_\_\_\_");

}

}

**OUTPUT:** drift drift hoist sail



### Exercise Solutions



**BE the Compiler**



Set 1 **will** work.

Set 2 **will not** compile because of Vampire’s return type (int).

The Vampire’s frighten() method (B) is not a legal override OR overload of Monster’s frighten() method. Changing ONLY the return type is not enough to make a valid overload, and since an int is not compatible with a boolean, the method is not a valid override. (Remember, if you change ONLY the return type, it must be to a return type that is compatible with the superclass version’s return type, and then it’s an over*ride*.

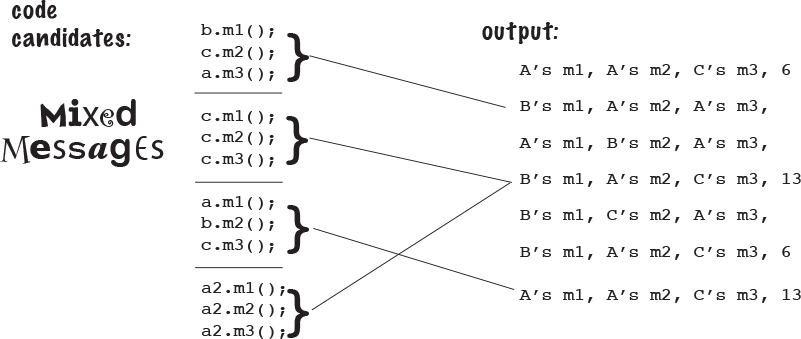
Sets 3 and 4 **will** compile, but produce:

**arrrgh**

**breathe fire**

**arrrgh**

Remember, class Vampire did not override class Monster’s frighten() method. (The frighten() method in Vampire’s set 4 takes a byte, not an int.)





public class Rowboat extends Boat {

public void rowTheBoat() {

System.out.print("stroke natasha");

}

}

public class Boat {

private int length ;

public void setLength ( int len ) {

length = len;

}

public int getLength() {

return length ;

}

public void move() {

System.out.print("drift ");

}

}

public class TestBoats {

public static void main(String[] args){

Boat b1 = new Boat();

Sailboat b2 = new Sailboat();

Rowboat b3 = new Rowboat();

b2.setLength(32);

b1.move();

b3.move();

b2.move();

}

}

public class Sailboat extends Boat {

public void move() {

System.out.print("hoist sail ");

}

}

**OUTPUT:** drift drift hoist sail