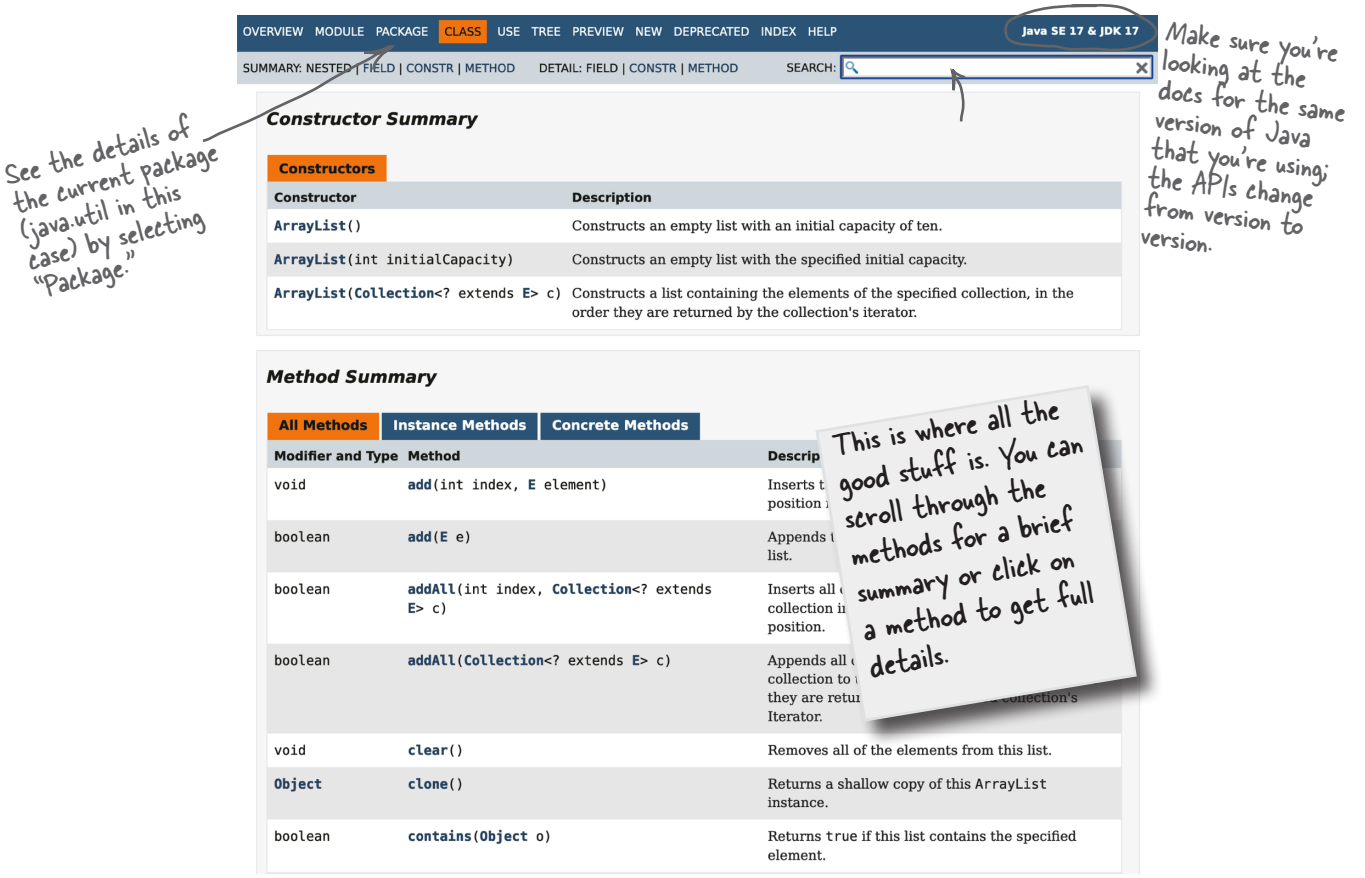
Using the class documentation

Whichever version of the Java docs you're using, they all have a similar layout for showing information about a specific class. This is where the juicy details are.

Let's say you were browsing through the reference book and found a class called ArrayList, in java.util. The book tells you a little about it, enough to know that this is indeed what you want to use, but you still need to know more about the methods. In the reference book, you'll find the method indexOf(). But if all you knew is that there is a method called indexOf() that takes an object and returns the index (an int) of that object, you still need to know one crucial thing: what happens if the object is not in the ArrayList? Looking at the method signature alone won't tell you how that works. But the API docs will (most of the time, anyway). The API docs tell you that the indexOf() method returns a -1 if the object parameter is not in the ArrayList. So now we know we can use it both as a way to check if an object is even in the ArrayList, and to get its index at the same time, if the object was there. But without the API docs, we might have thought that the indexOf() method would blow up if the object wasn't in the ArrayList.



OVERVIEW MODULE PACKAGE CLASS USE TREE PREVIEW NEW DEPRECATED INDEX HELP — Java SE 17 & JDK 17

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD SEARCH:

Constructor Summary

Constructors

Constructor	Description
`ArrayList()`	Constructs an empty list with an initial capacity of ten.
`ArrayList(int initialCapacity)`	Constructs an empty list with the specified initial capacity.
`ArrayList(Collection<? extends E> c)`	Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method	Descrip
void	`add(int index, E element)`	Inserts t position i
boolean	`add(E e)`	Appends t list.
boolean	`addAll(int index, Collection<? extends E> c)`	Inserts all collection i position.
boolean	`addAll(Collection<? extends E> c)`	Appends all collection to they are retu Iterator.
void	`clear()`	Removes all of the elements from this list.
`Object`	`clone()`	Returns a shallow copy of this ArrayList instance.
boolean	`contains(Object o)`	Returns true if this list contains the specified element.

In Chapters 11 and 12, you'll see how we use the API docs to figure out how to use the Java Libraries.

Code Magnets

Can you reconstruct the code snippets to make a working Java program that produces the output listed below? **NOTE:** To do this exercise, you need one NEW piece of info—if you look in the API for ArrayList, you'll find a *second* add method that takes two arguments:

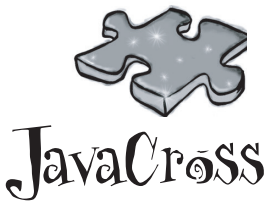
add(int index, Object o)

It lets you specify to the ArrayList *where* to put the object you're adding.

```
a.remove(2);
printList(a);
printList(a);
a.add(0, "zero");
a.add(1, "one");
public static void printList(ArrayList<String> list) {
if (a.contains("two")) {
    a.add("2.2");
}
a.add(2, "two");
  public static void main (String[] args) {
  System.out.print(element + "  ");
}
System.out.println();
if (a.contains("three")) {
    a.add("four");
}
public class ArrayListMagnet {
if (a.indexOf("four") != 4) {
    a.add(4, "4.2");
}
}
}
import java.util.ArrayList;
printList(a);
ArrayList<String> a = new ArrayList<String>();
for (String element : list) {
}
a.add(3, "three");
printList(a);
```

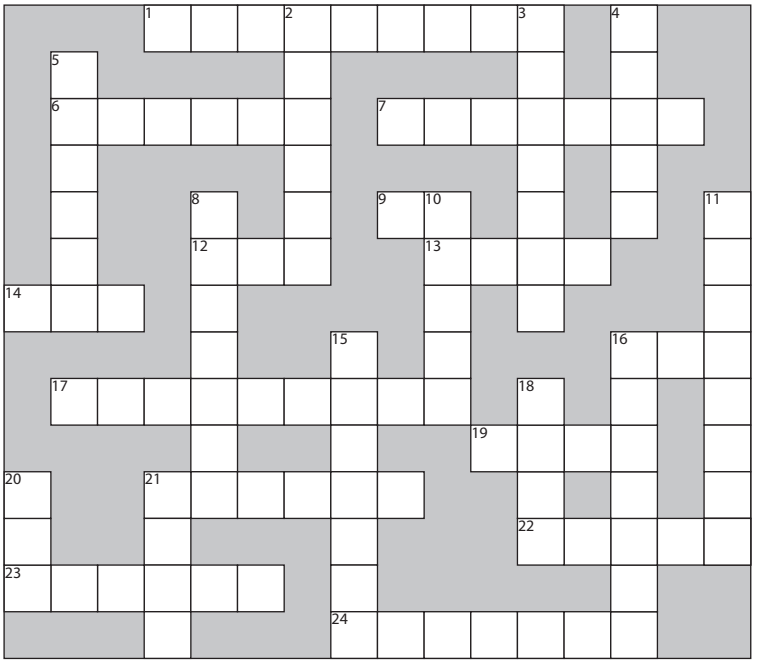
```
File  Edit  Window  Help  Dance
% java ArrayListMagnet
zero  one  two  three
zero  one  three  four
zero  one  three  four  4.2
zero  one  three  four  4.2
```

Answers on page 165.



How does this crossword puzzle help you learn Java? Well, all of the words **are** Java related (except one red herring).

Hint: When in doubt, remember ArrayList.



Across

1. I can't behave
6. Or, in the courtroom
7. Where it's at baby
9. A fork's origin
12. Grow an ArrayList
13. Wholly massive
14. Value copy
16. Not an object
17. An array on steroids
19. Extent
21. 19's counterpart
22. Spanish geek snacks (Note: This has nothing to do with Java.)
23. For lazy fingers
24. Where packages roam

Down

2. Where the Java action is
3. Addressable unit
4. 2nd smallest
5. Fractional default
8. Library's grandest
10. Must be low density
11. He's in there somewhere
15. As if
16. dearth method
18. What shopping and arrays have in common
20. Library acronym
21. What goes around

More Hints:

Across

1. 8 varieties
7. Think ArrayList
16. Common primitive
21. Array's extent
22. Not about Java—Spanish appetizers

Down

2. What's overridable?
3. Think ArrayList
4. & 10. Primitive
16. Think ArrayList
18. He's making a _____

Answers on page 166.

Exercise Solutions

```
File  Edit  Window  Help  Dance

% java ArrayListMagnet
zero  one  two  three
zero  one  three  four
zero  one  three  four  4.2
zero  one  three  four  4.2
```

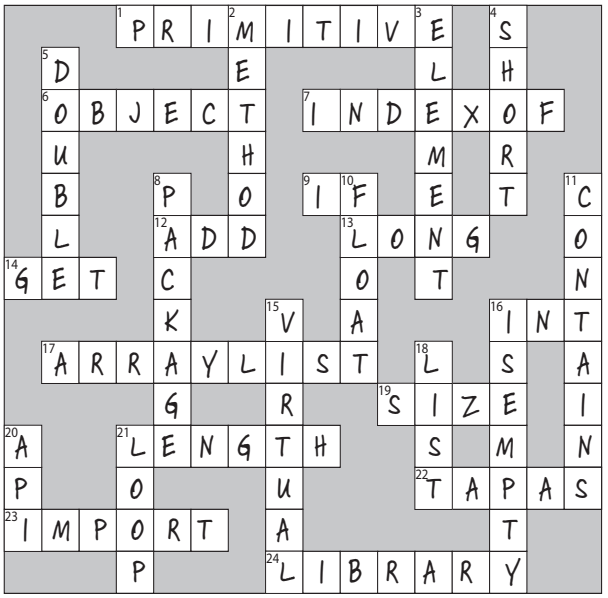
Code Magnets

(from page 163)

```
import java.util.ArrayList;

public class ArrayListMagnet {
  public static void main(String[] args) {
    ArrayList<String> a = new ArrayList<String>();
    a.add(0, "zero");
    a.add(1, "one");
    a.add(2, "two");
    a.add(3, "three");
    printList(a);

    if (a.contains("three")) {
      a.add("four");
    }
    a.remove(2);
    printList(a);

    if (a.indexOf("four") != 4) {
      a.add(4, "4.2");
    }
    printList(a);

    if (a.contains("two")) {
      a.add("2.2");
    }
    printList(a);
  }

  public static void printList(ArrayList<String> list) {
    for (String element : list) {
      System.out.print(element + "  ");
    }
    System.out.println();
  }
}
```

JavaCross

(from page 164)



Sharpen your pencil

Write your OWN set of clues! Look at each word, and try to write your own clues. Try making them easier, or harder, or more technical than the ones we have.

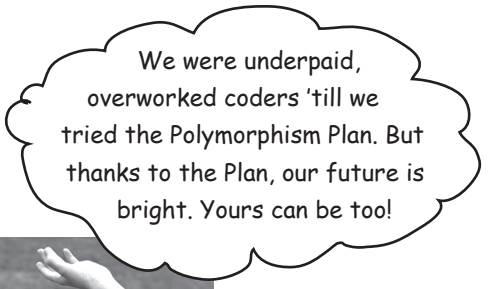
Across

1. _____
6. _____
7. _____
9. _____
12. _____
13. _____
14. _____
16. _____
17. _____
19. _____
21. _____
22. _____
23. _____
24. _____

Down

2. _____
3. _____
4. _____
5. _____
8. _____
10. _____
11. _____
15. _____
16. _____
18. _____
20. _____
21. _____

Better Living in Objectville



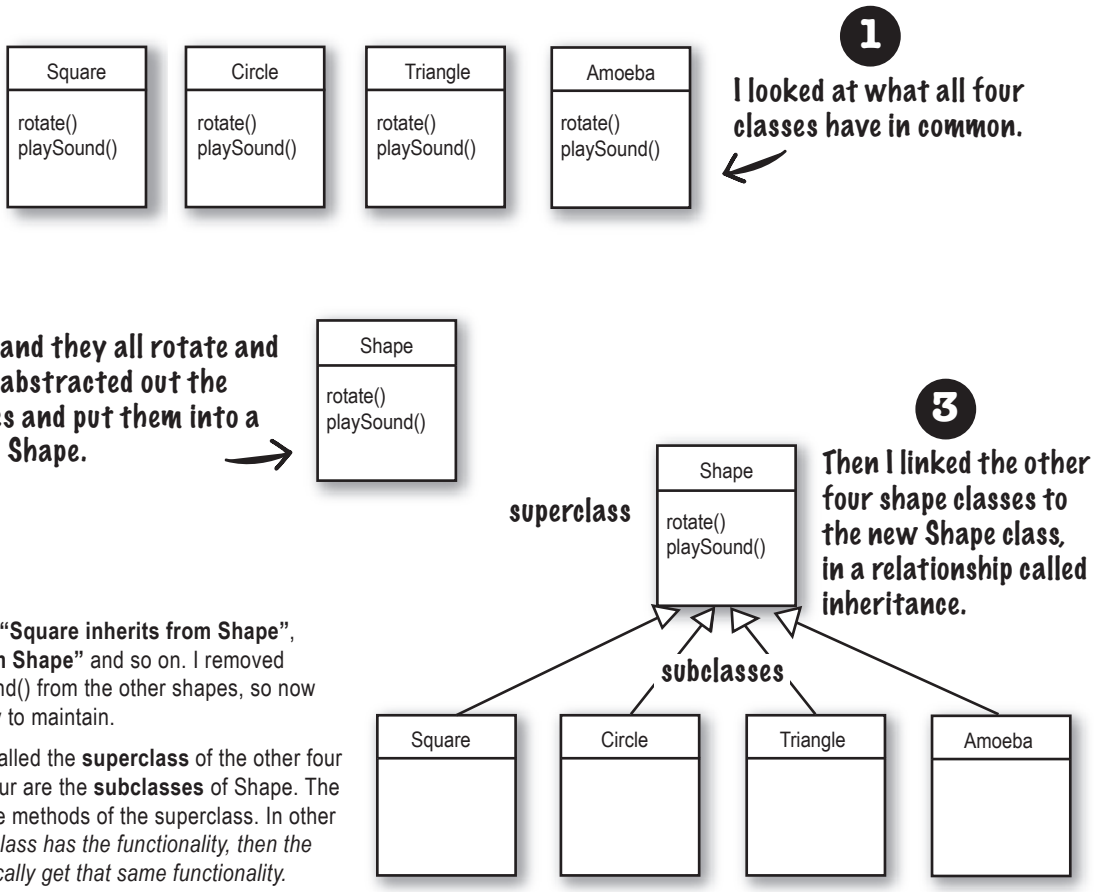
Plan your programs with the future in mind. If there were a way to write Java code such that you could take more vacations, how much would it be worth to you? What if you could write code that someone *else* could extend, **easily**? And if you could write code that was flexible, for those pesky last-minute spec changes, would that be something you'd be 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, when Laura (procedural programmer) and Brad (OO developer) were vying 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.

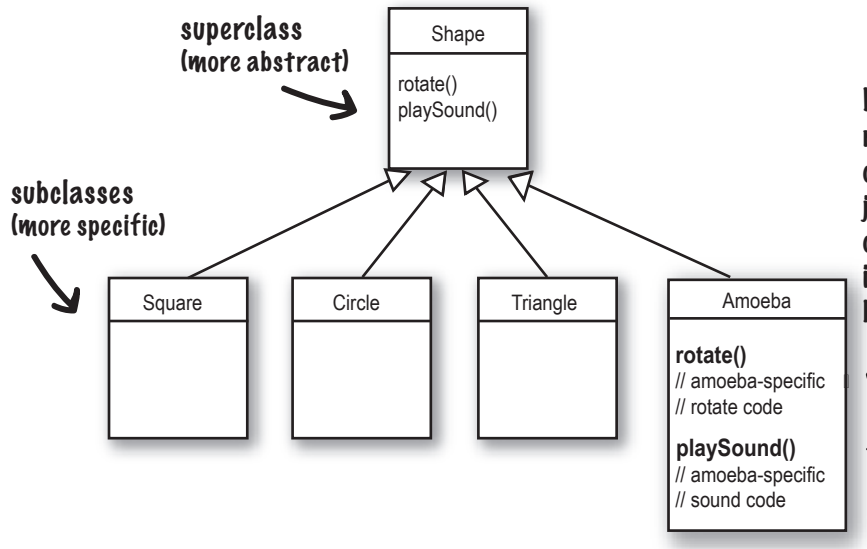
The Shape class is called the **superclass** of the other four classes. The other four are the **subclasses** of Shape. The subclasses inherit the methods of the superclass. In other words, *if the Shape class has the functionality, then the subclasses automatically get that same functionality.*

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?

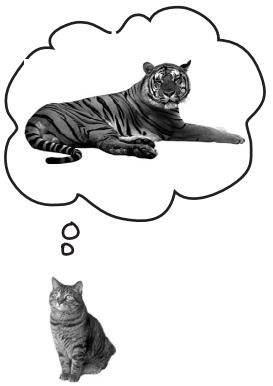
How can Amoeba do something different if it *inherits* its functionality from the Shape class?

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.



4

I made the Amoeba class override the rotate() and playSound() methods of the superclass Shape. Overriding just means that a subclass redefines one of its inherited methods when it needs to change or extend the behavior of that method.



BRAIN POWER

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?

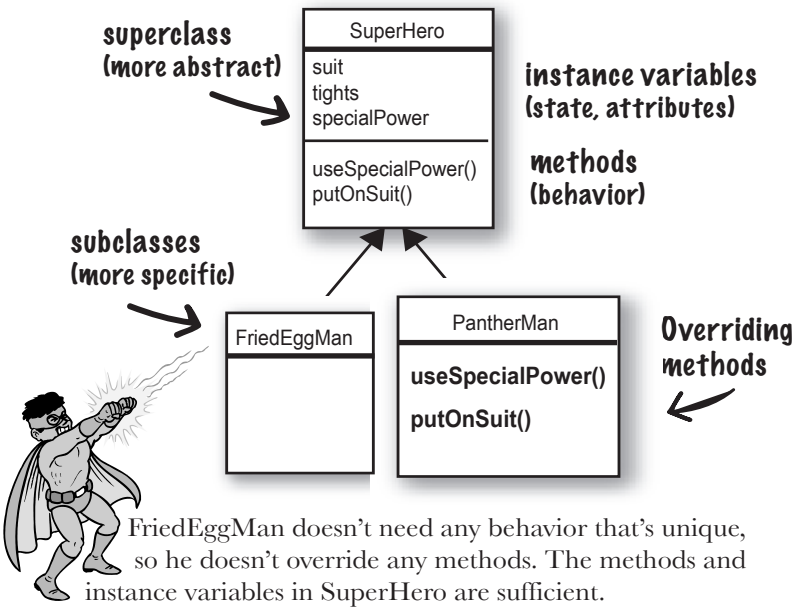
How would you design an inheritance structure? What methods would be overridden?

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`, `useSpecialPower()`, 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` to purple, while FriedEggMan sets his to white.

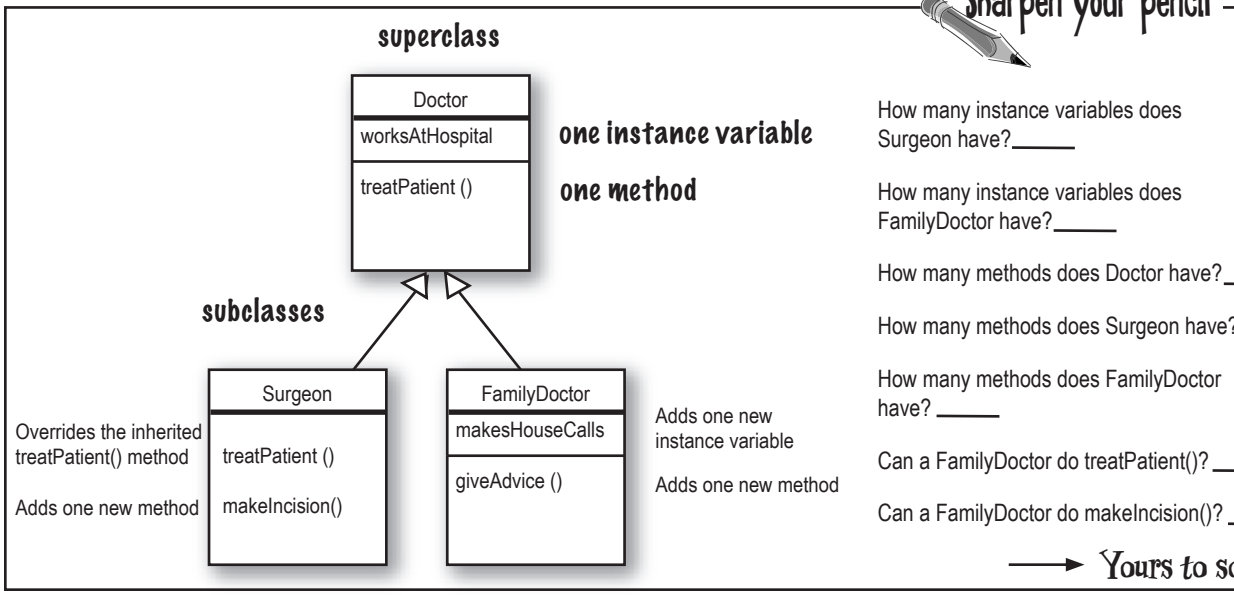
An inheritance example:

```
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



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()?_____

→ Yours to solve.

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.

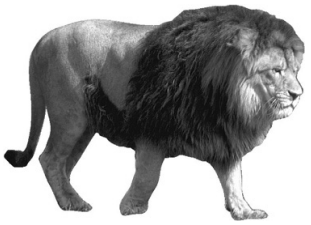
And we want other programmers to be able to add new kinds of animals to the program at any time.

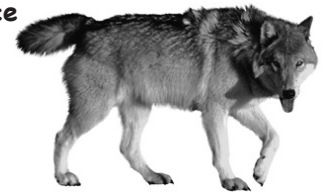
First we have to figure out the common, abstract characteristics that all animals have, and build those characteristics into a class that all animal classes can extend.

1 Look 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 (steps 4-5)





Using inheritance to avoid duplicating code in subclasses

We have five ***instance variables:***

picture – the filename 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 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.

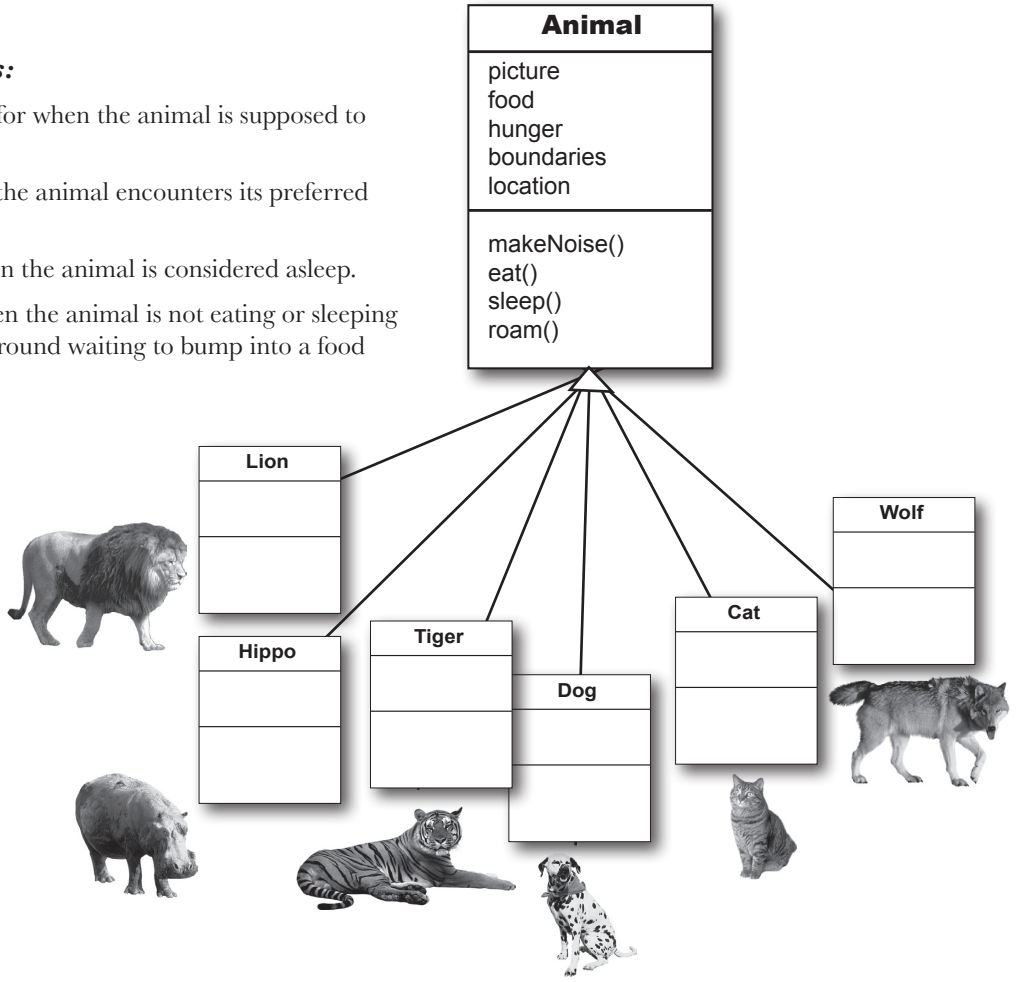
roam() – behavior for when the animal is not eating or sleeping (probably just wandering around waiting to bump into a food source or a boundary).

2

Design 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*?

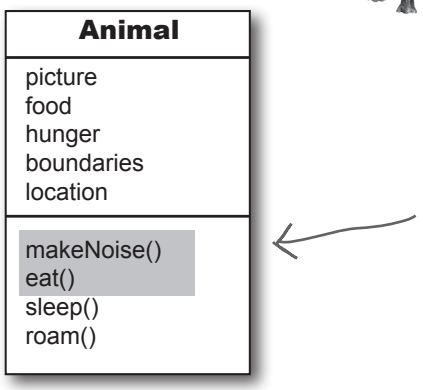
Which methods should we override?

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.

3 Decide 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.



We better override these two methods, eat() and makeNoise(), so that each animal type can define its own specific behavior for eating and making noise. For now, it looks like sleep() and roam() can stay generic.

Looking for more inheritance opportunities

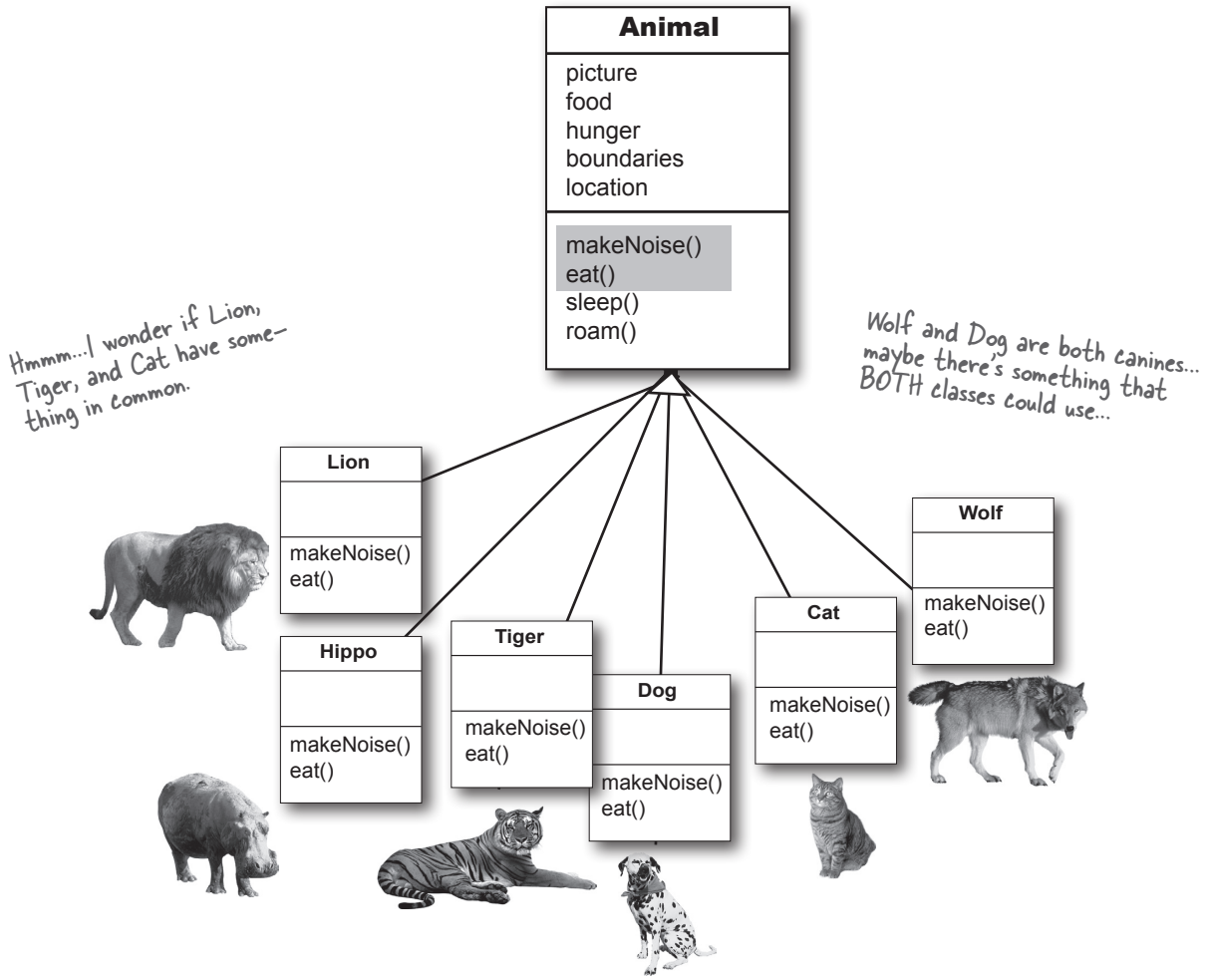
The class hierarchy is starting to shape up. We have each subclass override the *makeNoise()* and *eat()* methods so that there's no mistaking a Dog bark from a Cat meow (quite insulting to both parties). And a Hippo won't eat like a Lion.

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.

4

Look 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.

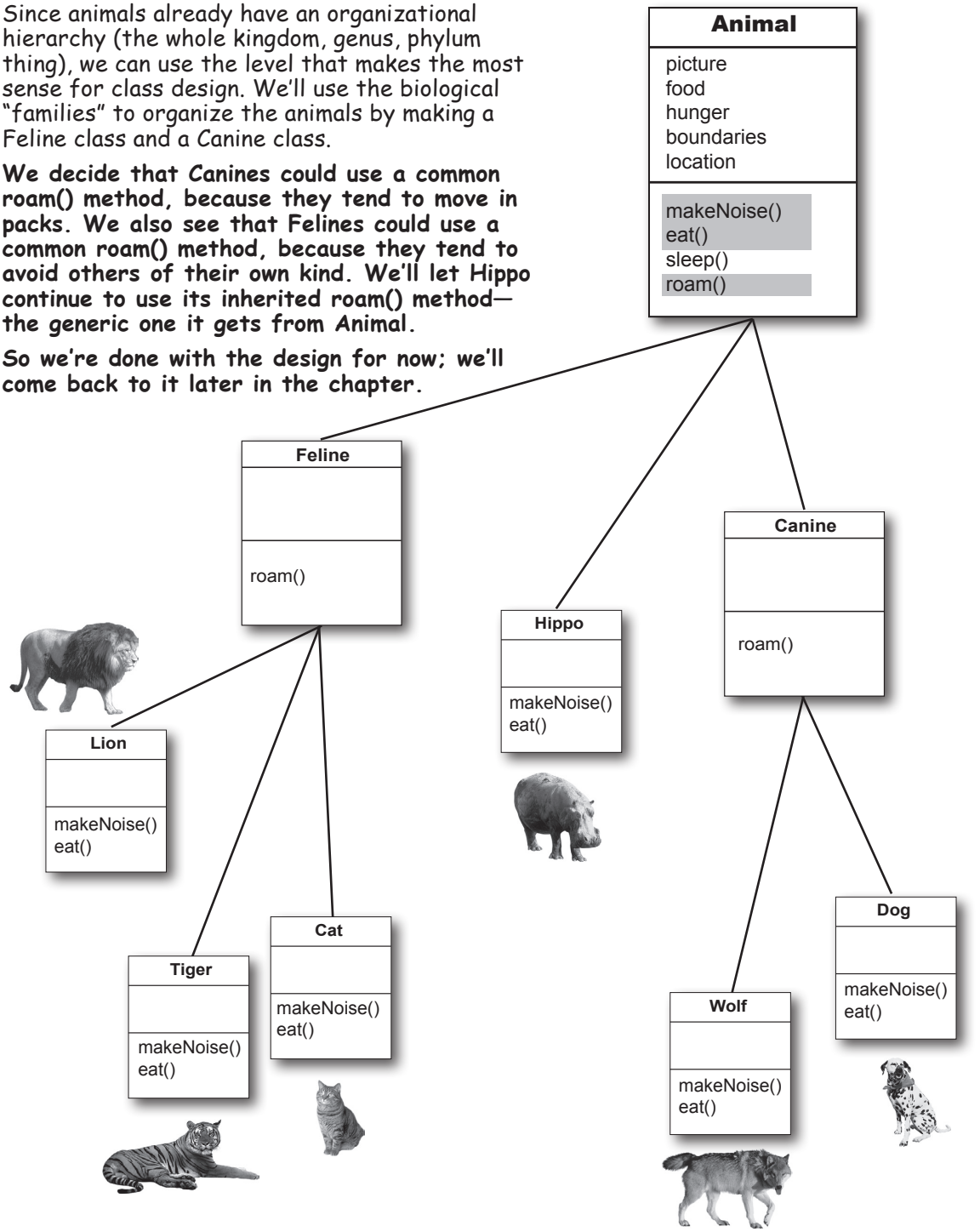


5 Finish the class hierarchy

Since animals already have an organizational hierarchy (the whole kingdom, genus, phylum 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 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?

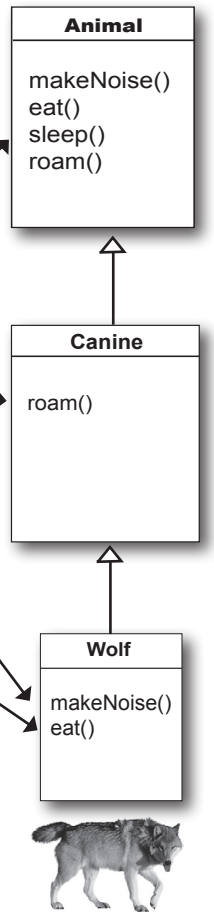
Make a new Wolf object
```
Wolf w = new Wolf();
```

Calls the version in Wolf
```
w.makeNoise();
```

Calls the version in Canine
```
w.roam();
```

Calls the version in Wolf
```
w.eat();
```

Calls the version in Animal
```
w.sleep();
```



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

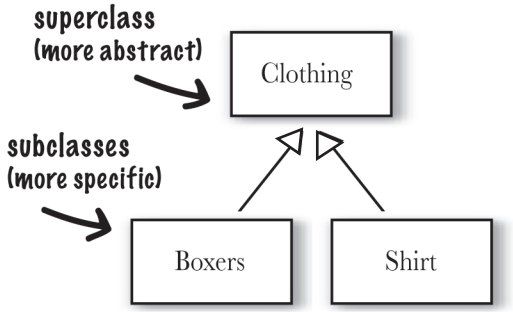
In other words, ***the lowest one wins!***

"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

Class	Superclasses	Subclasses
Clothing	---	Boxers, Shirt
Boxers	Clothing	
Shirt	Clothing	

Inheritance Table



Inheritance Class Diagram

Sharpen your pencil

Find the relationships that make sense. Fill in the last two columns

Class	Superclasses	Subclasses
Musician		
Rock Star		
Fan		
Bass Player		
Concert Pianist		

Hint: not everything can be connected to something else.

Hint: you're allowed to add to or change the classes listed.

Draw an inheritance diagram here.

→ Yours to solve.

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.

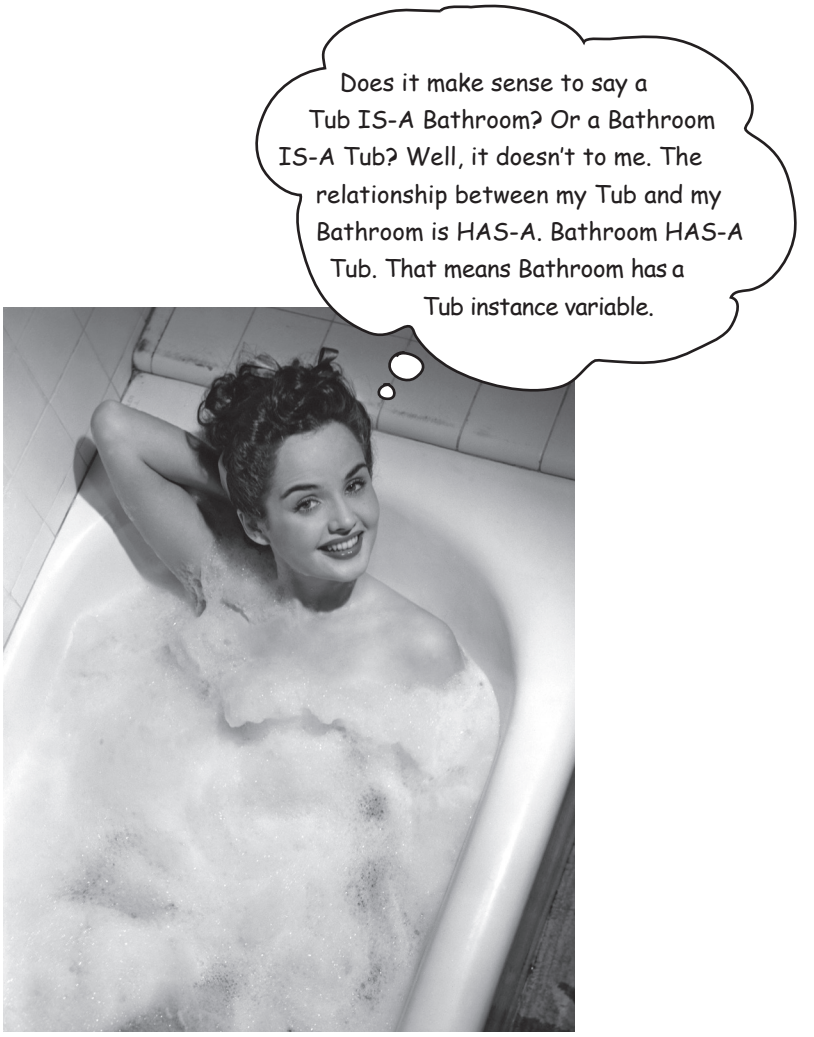
Surgeon IS-A Doctor, still good.

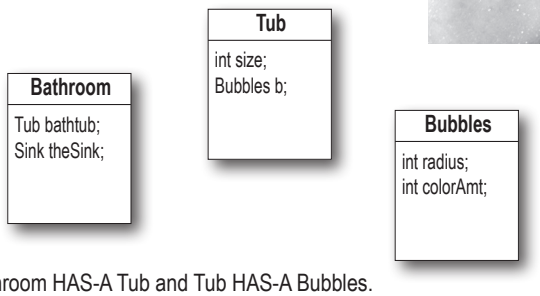
Tub extends Bathroom, sounds reasonable.
Until you apply the IS-A test.

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.

Canine extends Animal

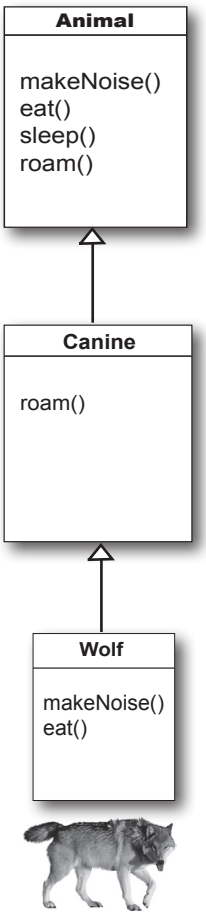
Wolf extends Canine

Wolf extends Animal

Canine IS-A Animal

Wolf IS-A Canine

Wolf IS-A Animal



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.

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 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.

- ☐ **Oven extends Kitchen**
- ☐ **Guitar extends Instrument**
- ☐ **Person extends Employee**
- ☐ **Ferrari extends Engine**
- ☐ **FriedEgg extends Food**
- ☐ **Beagle extends Pet**
- ☐ **Container extends Jar**
- ☐ **Metal extends Titanium**
- ☐ **GratefulDead extends Band**
- ☐ **Blonde extends Smart**
- ☐ **Beverage extends Martini**

→ Yours to solve.

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 *backward* 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."

```
public void roam() {
    super.roam();
     // my own roam stuff
}
```

This calls the inherited version of roam(), then comes back to do your own subclass-specific code

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? (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:

private **default** **protected** **public**

Access levels control *who sees what,* and are crucial to having well-designed, robust 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: get more details about default and protected in Appendix B.

When designing with inheritance, are you using or abusing?

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 *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. 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 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, return type, method name, etc.)

1 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) sees the change.

2 You define a common protocol for a group of classes.

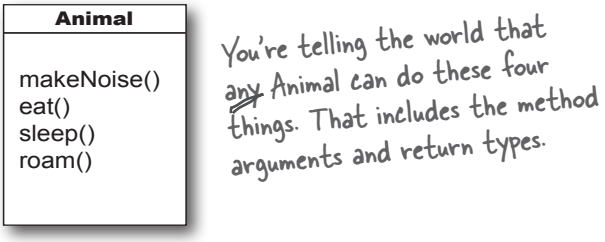
Inheritance lets you guarantee that all classes grouped under a certain supertype have all the methods that the supertype has*

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 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*. That 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 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 that definition a bit more.

And I care because...

You get to take advantage of polymorphism.

Which matters to me 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

```
1        3  2
Dog myDog = new Dog();
```

1 Declare 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.

2 Create an object

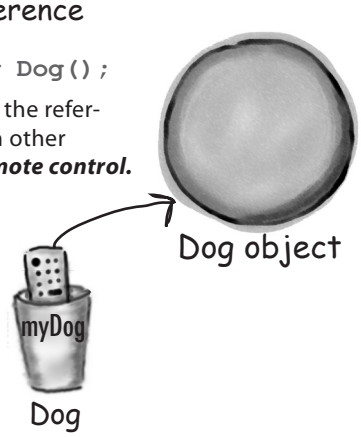
```
Dog myDog = new Dog();
```

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

3 Link the object and the reference

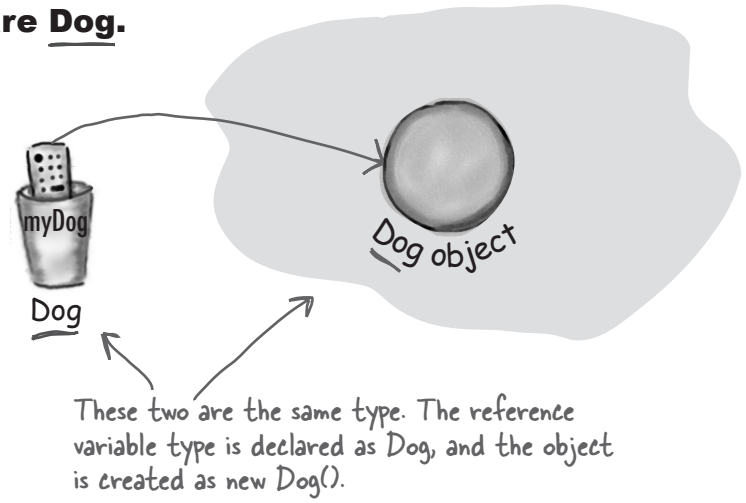
```
Dog myDog = new Dog();
```

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



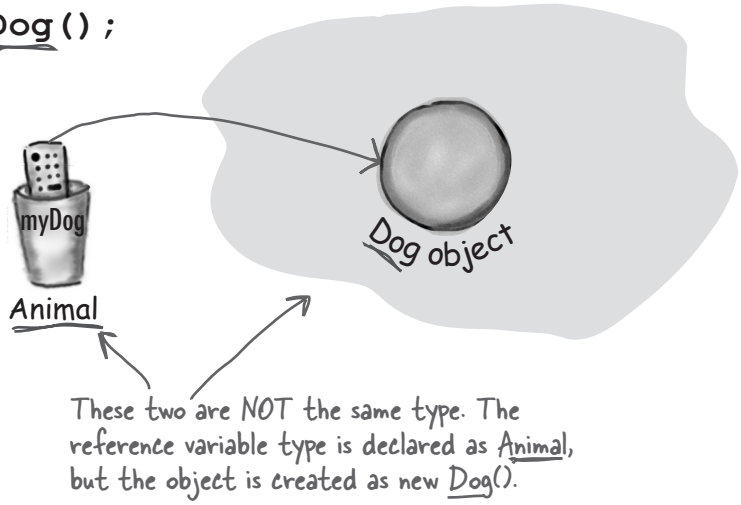
The important point is that the reference type AND the object type are the same.

In this example, both are Dog.



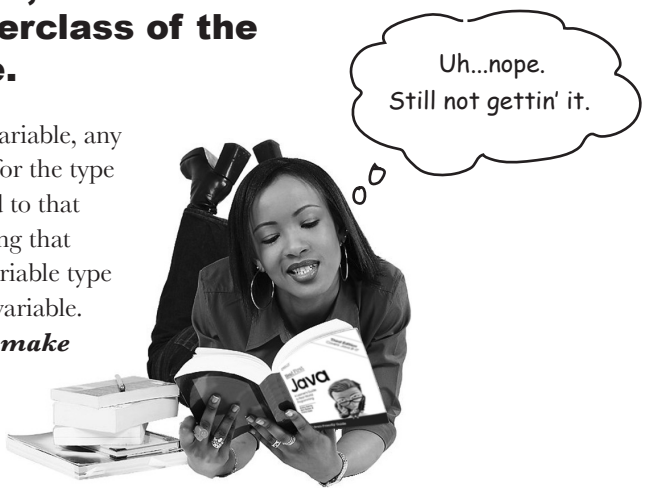
But with polymorphism, the reference type and the object type can be *different*.

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



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

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.

Declare an array of type Animal. In other words, an array that will hold objects of type Animal.

```
Animal[] animals = new Animal[5];
animals[0] = new Dog();
animals[1] = new Cat();
animals[2] = new Wolf();
animals[3] = new Hippo();
animals[4] = new Lion();

for (Animal animal : animals) {

  animal.eat();

  animal.roam();
}
```

But look what you get to do...you can put ANY subclass of Animal in the Animal array!

And here's the best polymorphic part (the raison d'être for the whole example): you get to loop through the array and call one of the Animal-class methods, and every object does the right thing!

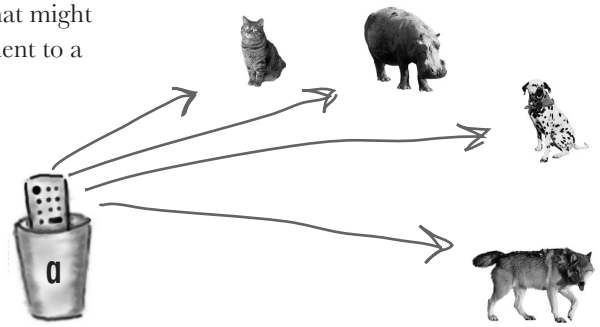
On the first pass through the loop, 'animal' is a Dog, so you get the Dog's eat() method. On the next pass, 'animal' is a Cat, so you get the Cat's eat() method.

Same with roam().

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...




```
class Vet {
  public void giveShot(Animal a) {
    // do horrible things to the Animal at
    // the other end of the 'a' parameter
    a.makeNoise();
  }
}
```

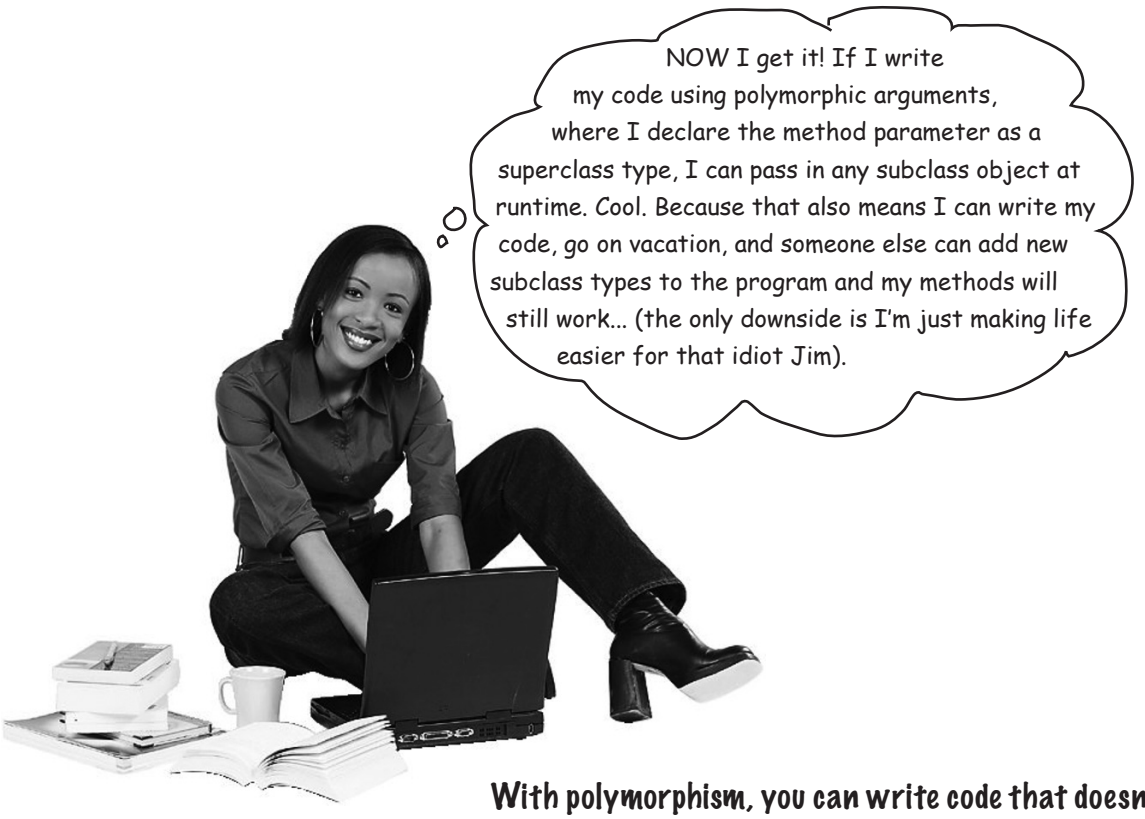
The 'a' parameter can take ANY Animal type as the argument. And when the Vet is done giving the shot, it tells the Animal to makeNoise(), and whatever Animal is really out there on the heap, that's whose makeNoise() method will run.

```
class PetOwner {
  public void start() {
    Vet vet = new Vet();
    Dog dog = new Dog();
    Hippo hippo = new Hippo();
    vet.giveShot(dog);
    vet.giveShot(hippo);
  }
}
```

The Vet's giveShot() method can take any Animal you give it. As long as the object you pass in as the argument is a subclass of Animal, it will work.

Dog's makeNoise() runs

Hippo's makeNoise() runs



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.

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, 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, which 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), 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 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.

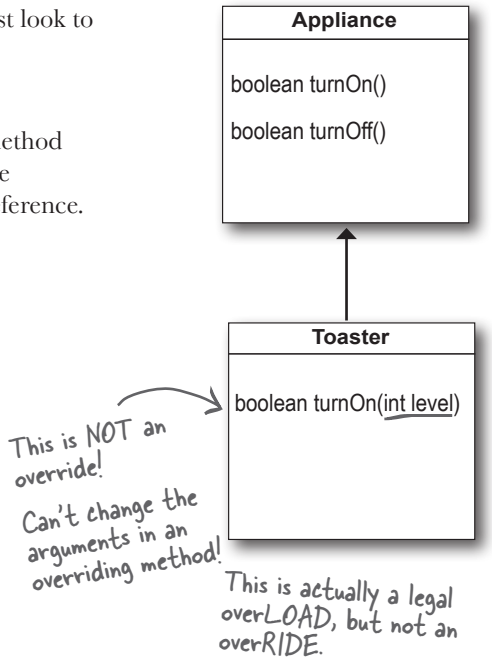
If polymorphism is going to work, the Toaster's version of the overridden method from Appliance has to work at runtime. Remember, the compiler looks at the reference type to decide whether you can call a particular method on that reference.

```
Appliance appliance = new Toaster();
```

Reference type — Object type

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!***



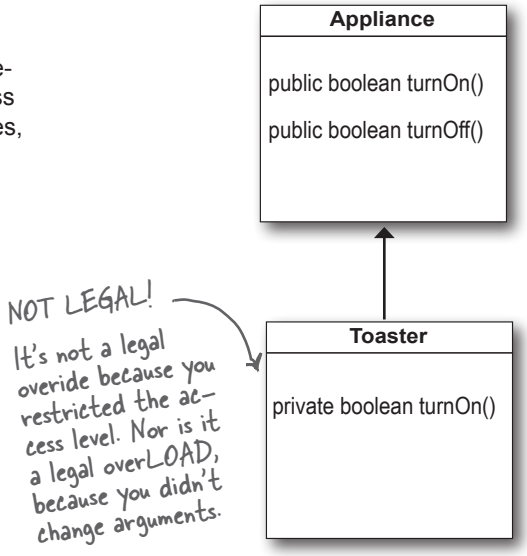
① Arguments must be the same, and return types must be compatible.

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.

② 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 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 Chapter 13, *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 Chapter 9, *Life and Death of an Object*.

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

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.

① 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.

② 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.

③ 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);
  }
}
```

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, ");
  }
}

class C extends B {
  void m3() {
    System.out.print("C's m3, " + (ivar + 6));
  }
}

public class Mixed2 {
  public static void main(String[] args) {
    A a = new A();
    B b = new B();
    C c = new C();
    A a2 = new C();

    // Candidate code goes here (three lines)

  }
}
```

Code candidates:

```
b.m1();
c.m2();
a.m3();
```

```
c.m1();
c.m2();
c.m3();
```

```
a.m1();
b.m2();
c.m3();
```

```
a2.m1();
a2.m2();
a2.m3();
```

Output:

A's m1, A's m2, C's m3, 6

B's m1, A's m2, A's m3,

A's m1, B's m2, A's m3,

B's m1, A's m2, C's m3, 13

B's m1, C's m2, A's m3,

B's m1, A's m2, C's m3, 6

A's m1, A's m2, C's m3, 13

Answers on page 197.

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.)

```
public class MonsterTestDrive {

  public static void main(String[] args) {
    Monster[] monsters = new Monster[3];
    monsters[0] = new Vampire();
    monsters[1] = new Dragon();
    monsters[2] = new Monster();
    for (int i = 0; i < monsters.length; i++) {
      monsters[i].frighten(i);
    }
  }
}

class Monster {
  A
}

class Vampire extends Monster {
  B
}

class Dragon extends Monster {
  boolean frighten(int degree) {
    System.out.println("breathe fire");
    return true;
  }
}
```

```
File  Edit  Window  Help  SaveYourself

% java MonsterTestDrive
a bite?
breathe fire
arrrgh
```

1

A
```
boolean frighten(int d) {
  System.out.println("arrrgh");
  return true;
}
```

B
```
boolean frighten(int x) {
  System.out.println("a bite?");
  return false;
}
```

2

A
```
boolean frighten(int x) {
  System.out.println("arrrgh");
  return true;
}
```

B
```
int frighten(int f) {
  System.out.println("a bite?");
  return 1;
}
```

3

A
```
boolean frighten(int x) {
  System.out.println("arrrgh");
  return false;
}
```

B
```
boolean scare(int x) {
  System.out.println("a bite?");
  return true;
}
```

4

A
```
boolean frighten(int z) {
  System.out.println("arrrgh");
  return true;
}
```

B
```
boolean frighten(byte b) {
  System.out.println("a bite?");
  return true;
}
```

Answers on page 197.

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");
   }
}
```

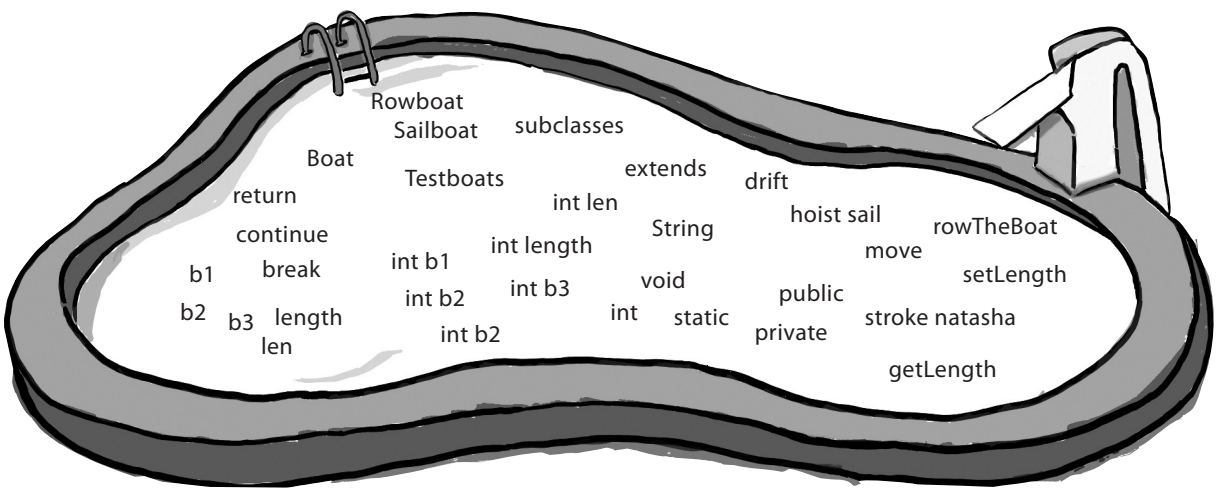
```
public class ________ {
   private int __________ ;
   _______ void _________ ( ______ ) {
      length = len;
   }
   public int getLength() {
      ________ _________ ;
   }
   public ___________ move() {
      System.out.print("___________");
   }
}
```

```
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
```



Answers on page 198.

Exercise Solutions

BE the Compiler (from page 195)

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 *override*.)

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.)

Code candidates:

Mixed Messages

(from page 194)

```
b.m1();
c.m2();
a.m3();
```

```
c.m1();
c.m2();
c.m3();
```

```
a.m1();
b.m2();
c.m3();
```

```
a2.m1();
a2.m2();
a2.m3();
```

Output:

```
A's m1, A's m2, C's m3, 6
B's m1, A's m2, A's m3,
A's m1, B's m2, A's m3,
B's m1, A's m2, C's m3, 13
B's m1, C's m2, A's m3,
B's m1, A's m2, C's m3, 6
A's m1, A's m2, C's m3, 13
```

Pool Puzzle

(from page 196)

```
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:

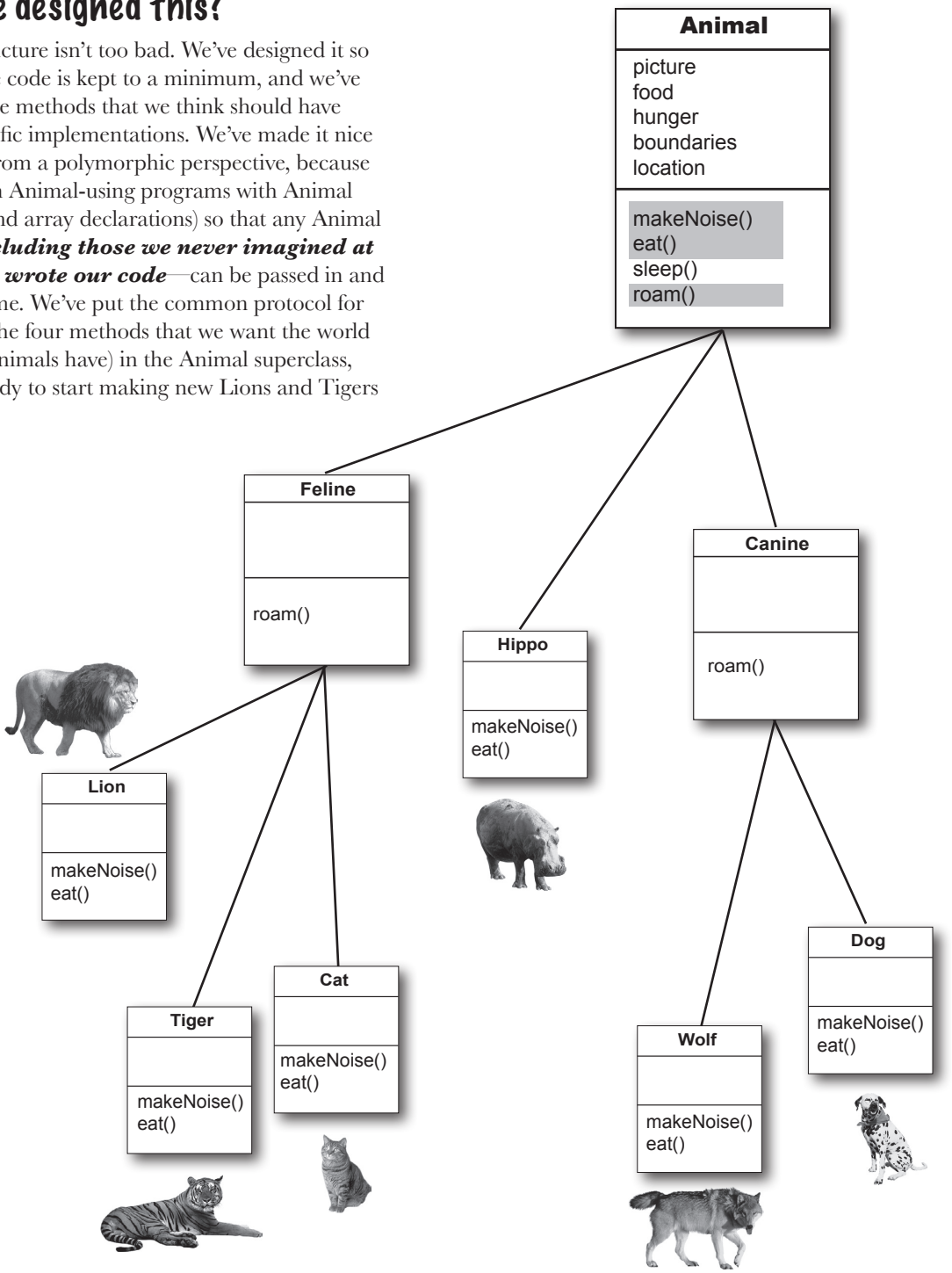
```
drift  drift  hoist sail
```

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 previous 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.

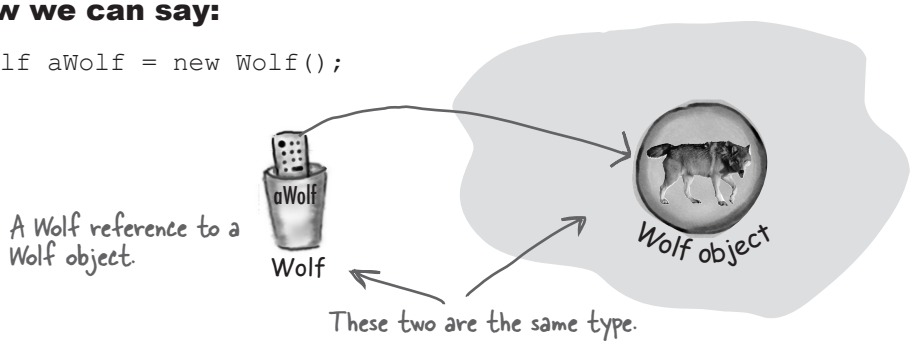
Did we forget about something when we designed this?

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.



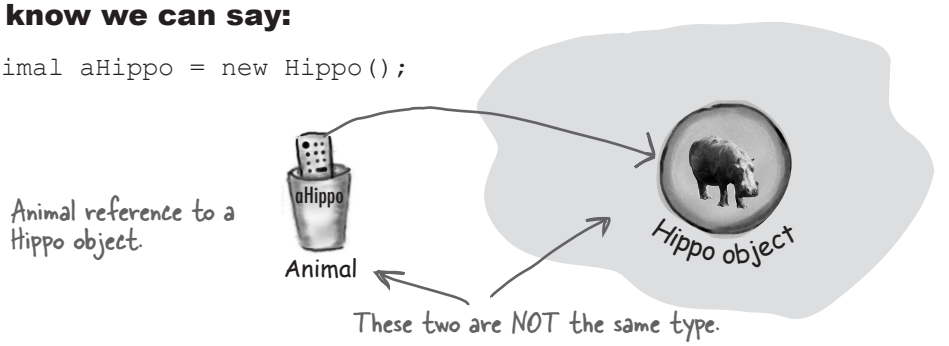
We know we can say:

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



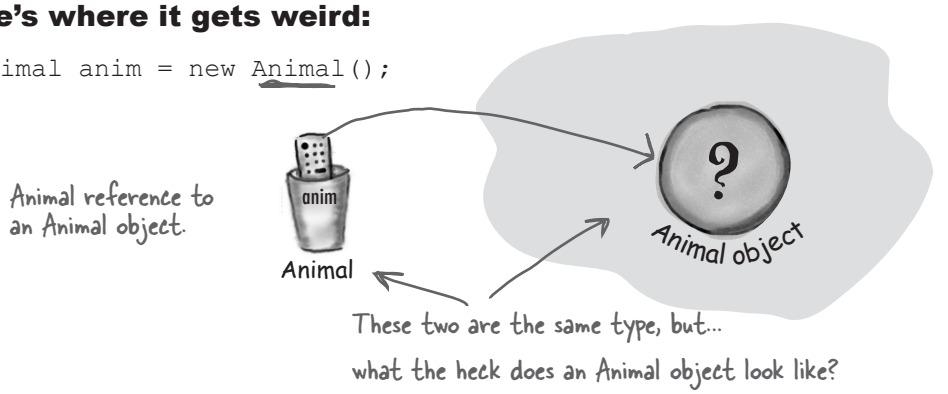
And we know we can say:

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

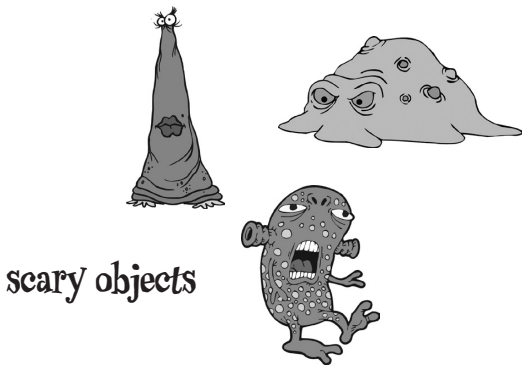


But here's where it gets weird:

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



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™ 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.***

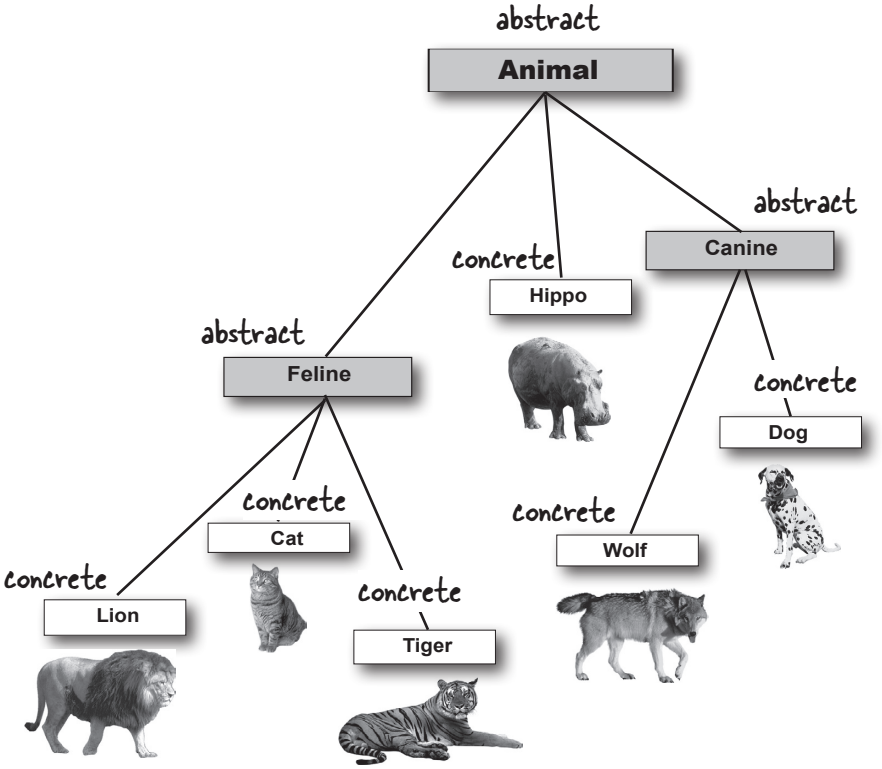
With an abstract class, it's the **instances of a *subclass*** of your abstract class that's doing the work at runtime

*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.



BRAIN POWER

Abstract or concrete?

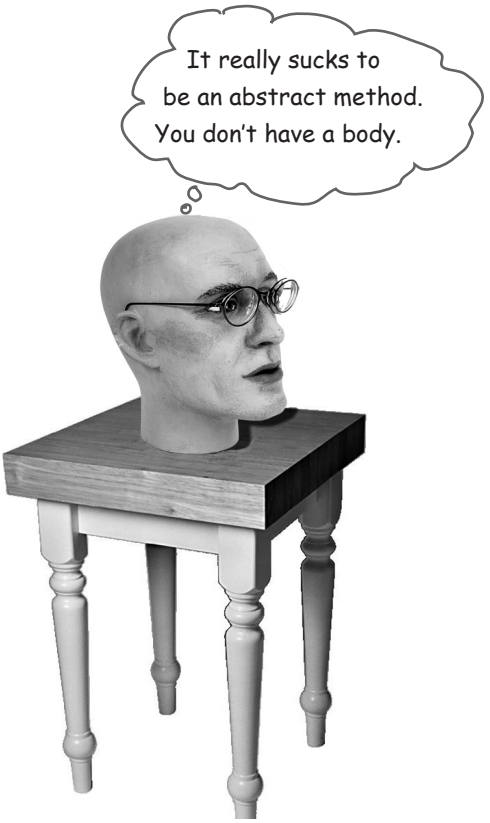
How do you know when a class should be abstract? ***Wine*** is probably abstract. But what about ***Red*** and ***White***? Again probably abstract (for some of us, anyway). But at what point in the hierarchy do things become concrete?

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 versus 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 and 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.)


```
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:

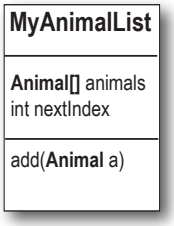
1. Make a separate class, MyCatList, to hold Cat objects. Pretty clunky.

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 a 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




```
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 dog = new Dog();
    Cat cat = new Cat();
    list.add(dog);
    list.add(cat);
  }
}
```

```
File Edit Window Help Harm
% java AnimalTestDrive
Animal added at 0
Animal added at 1
```

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...

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 library class*.

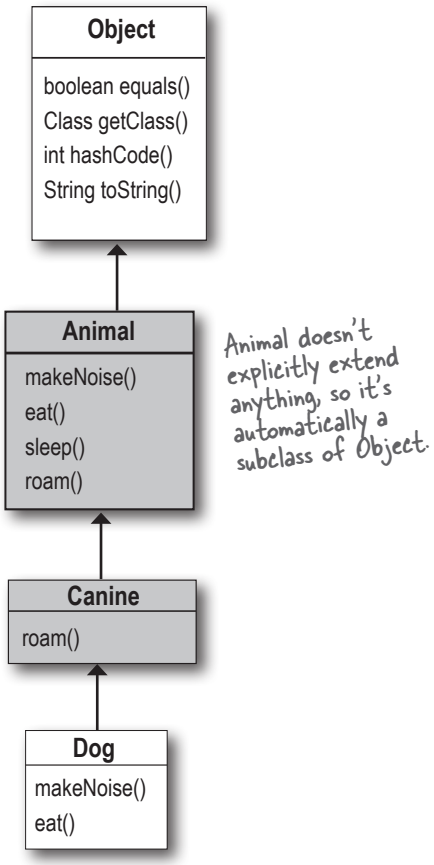
So you were making subclasses of class Object from the very beginning and you didn't even know it. ***Every class you write extends Object,*** without your ever having to say it. But you can think of it as though a class you write looks like this:

```
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.

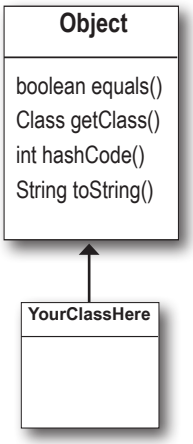
So, since Dog extends Canine, it doesn't *directly* extend Object (although it does extend it indirectly), and the same is true for Canine, but Animal *does* directly extend Object.



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 later). 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'.

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.

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).

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.

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: 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. 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?

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 as the last example.)

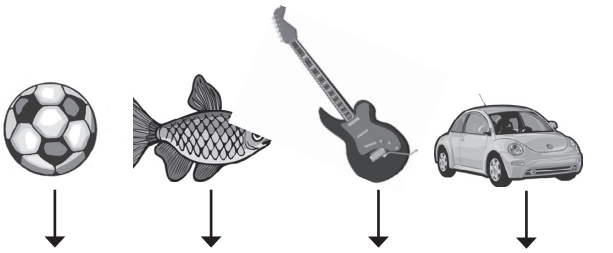
But what happens when you try to get the Dog object and assign it to a Dog reference?

```
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!!

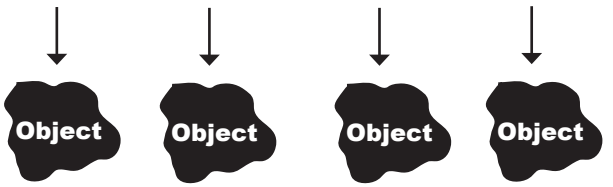
Everything comes out of an ArrayList<Object> as a reference of type Object, regardless of what the actual object is or what the reference type was when you added the object to the list.

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



ArrayList<Object>

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;
}
```

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.

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>.

```
File Edit Window Help Remember
DogPolyTest.java:10: incompatible types
found   : java.lang.Object
required: Dog
      Dog sameDog = getObject(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.)

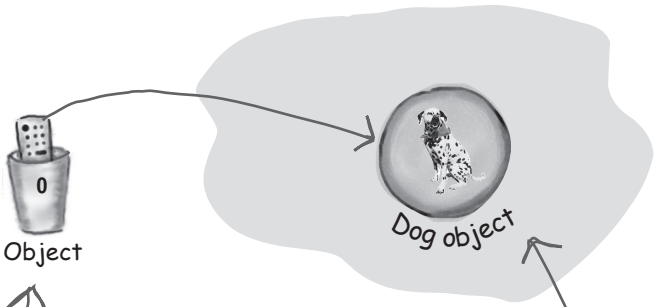
GOOD ☺

```
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:



When you get an object reference from an ArrayList<Object> (or any method that declares Object as the return type), it comes back as a polymorphic reference type of Object. So you have an Object reference to (in this case) a Dog instance.

```
Object o = al.get(index);
int i = o.hashCode();
o.bark();
```

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

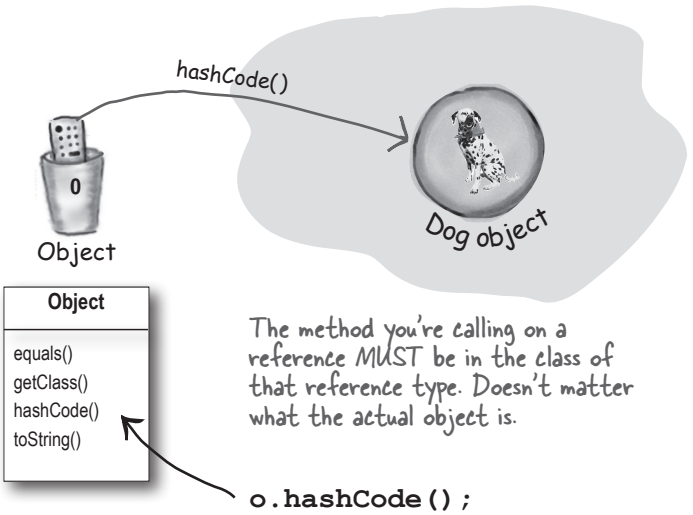
Won't compile!

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.

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.



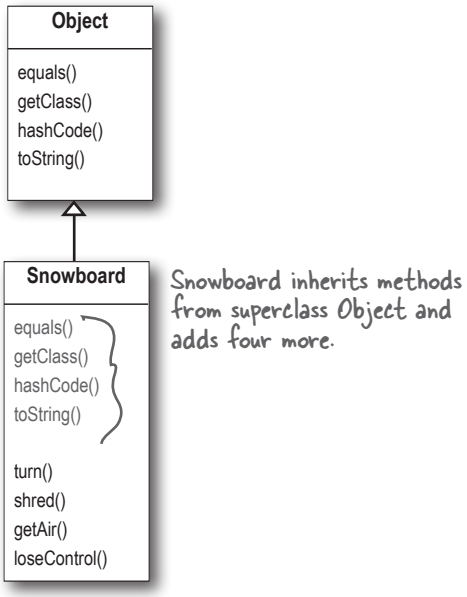
The method you're calling on a reference MUST be in the class of that reference type. Doesn't matter what the actual object is.

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

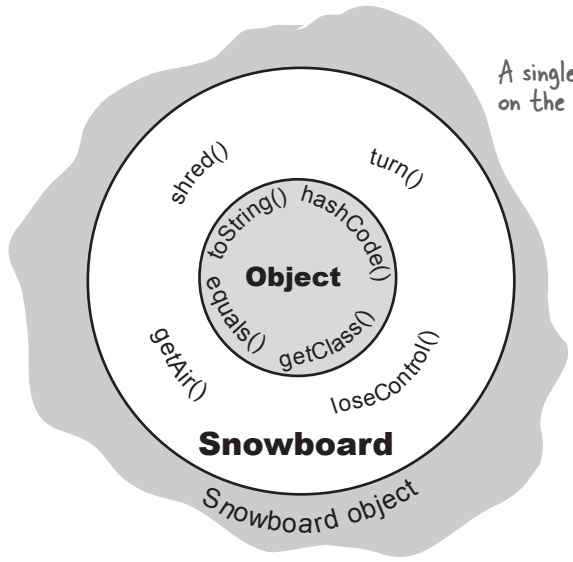
The "o" reference was declared as type Object, so you can call methods only if those methods are in class Object.

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.

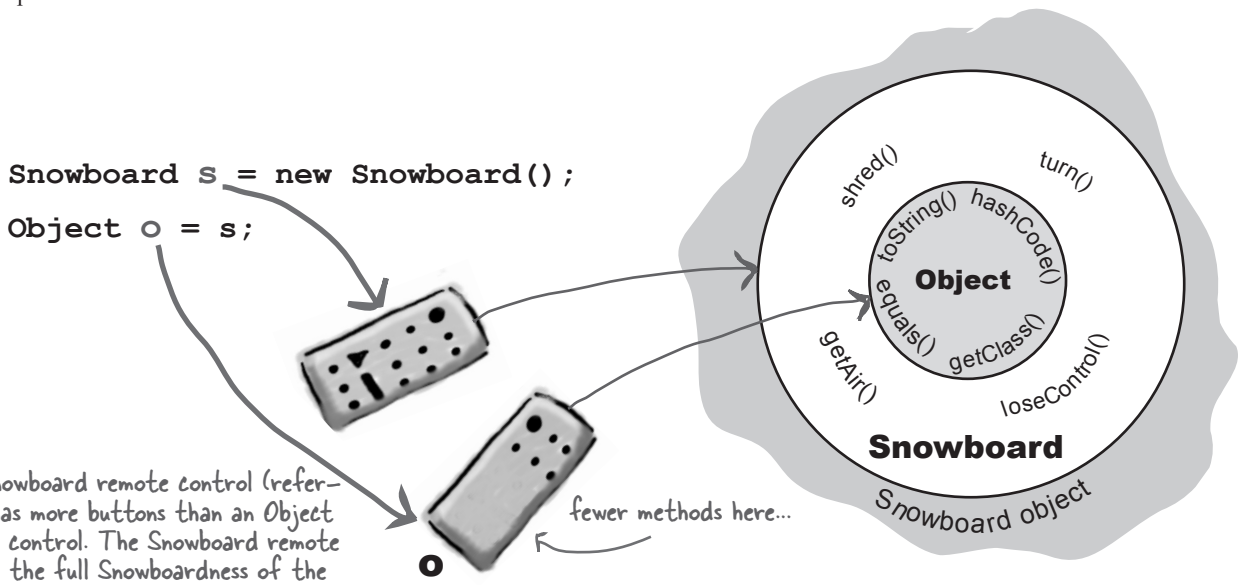
If a reference is like a remote control, the remote control takes on more and more buttons as you move down the inheritance tree. A remote control (reference) of type Object has only a few buttons—the buttons for the exposed methods of class Object. But a remote control of type Snowboard includes all 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.

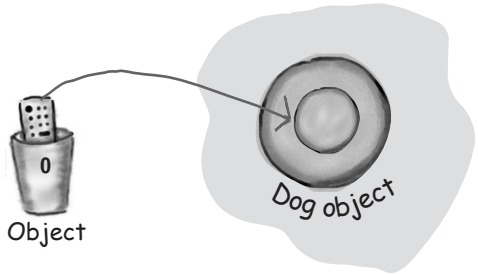
When you put an object in an ArrayList<Object>, you can treat it only as an Object, regardless of the type it was when you put it in.

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

That means you get an *Object* remote control.



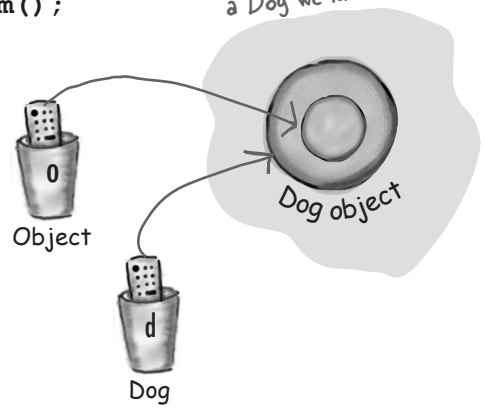
Casting an object reference back to its *real* type.



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;
d.roam();
```

← cast the Object back to a Dog we know is there.



*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;
}
```

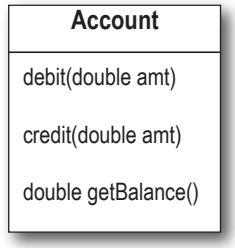
So now you've seen how much Java cares about the methods in the class of the reference variable.

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.



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 every time 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 trade-offs.

1 Option one

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

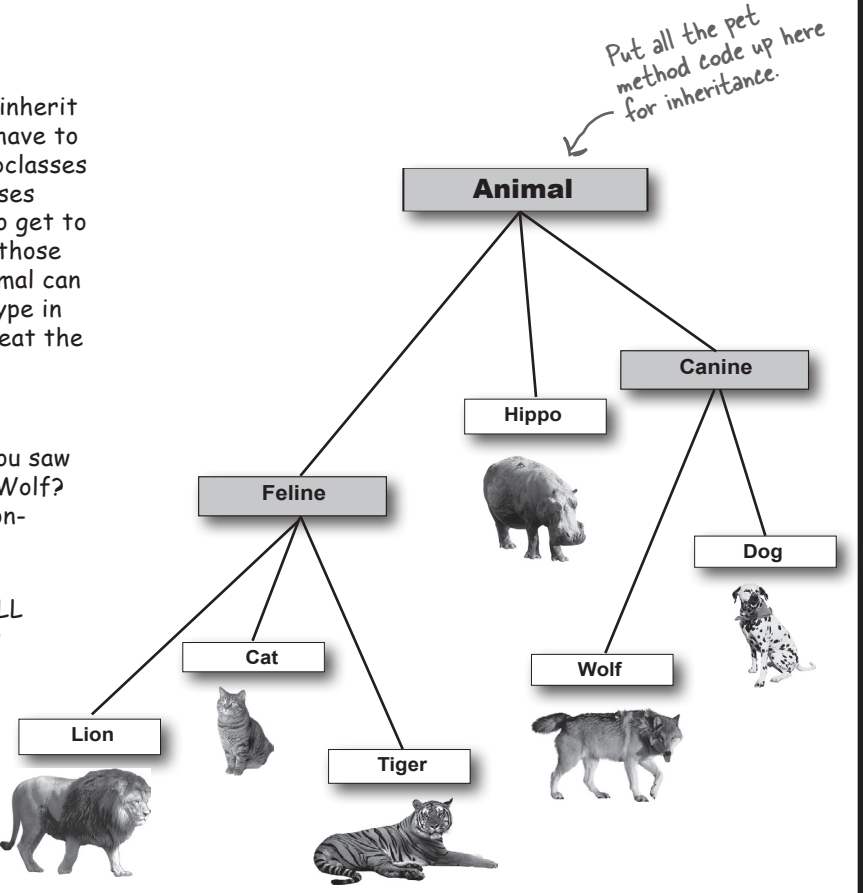
Pros:

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.

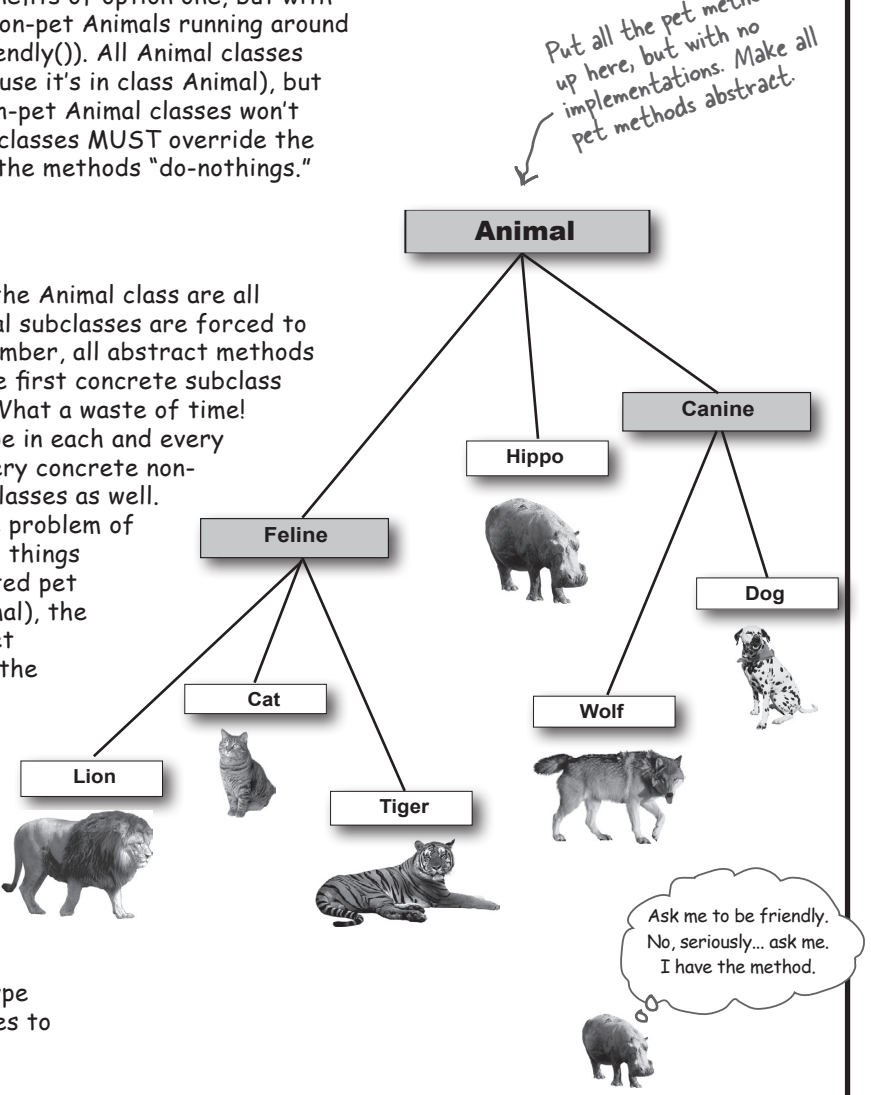
Pros:

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:

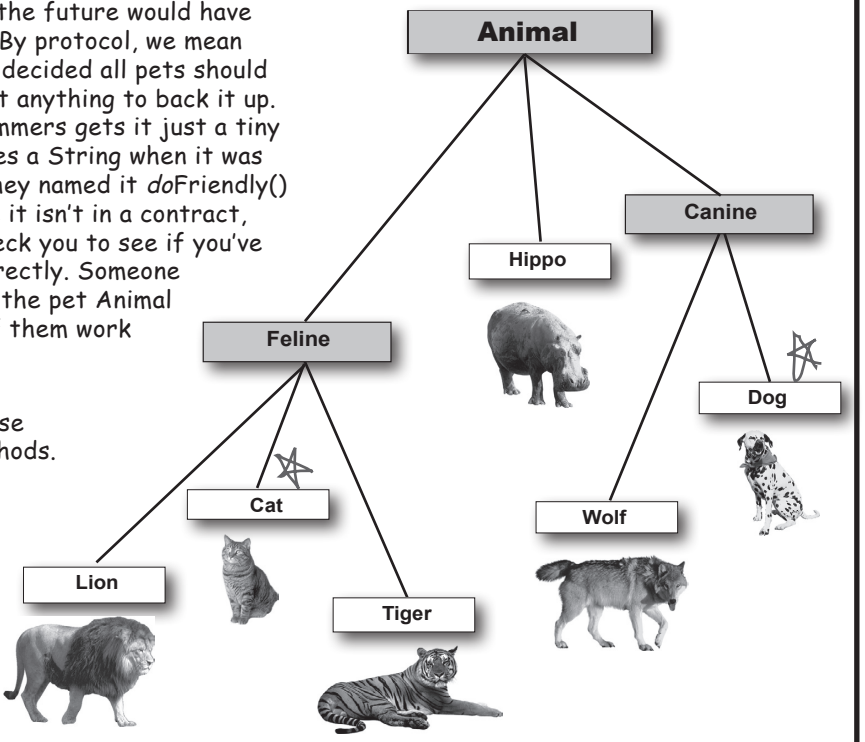
No more worries about Hippos greeting you at the door or licking your face. The methods are where they belong, and ONLY where they belong. Dogs can implement the methods and Cats can implement the methods, but nobody else has to know about them.

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 *do*Friendly() instead of *be*Friendly()? 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.

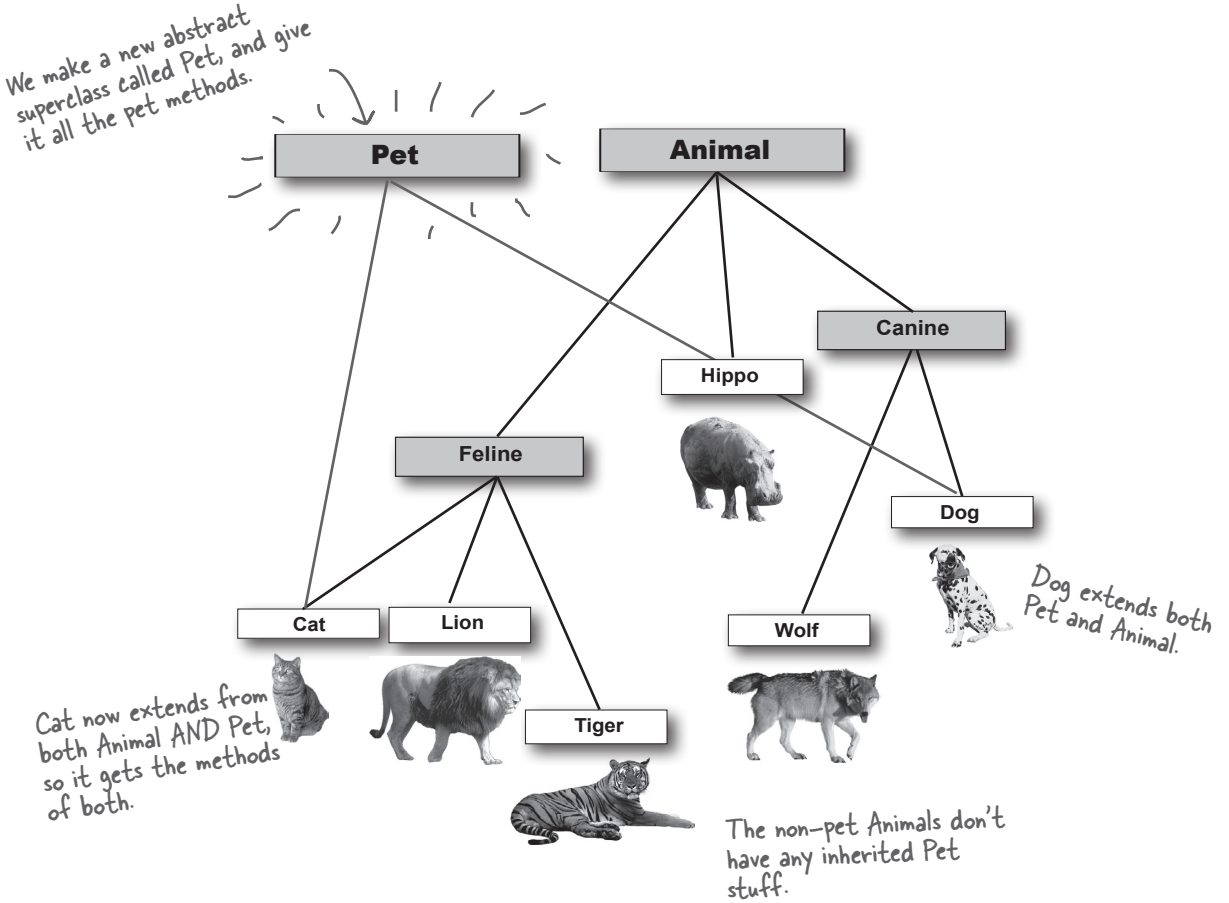
Put the pet methods ONLY in the Animal classes that can be pets, instead of in Animal.



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.



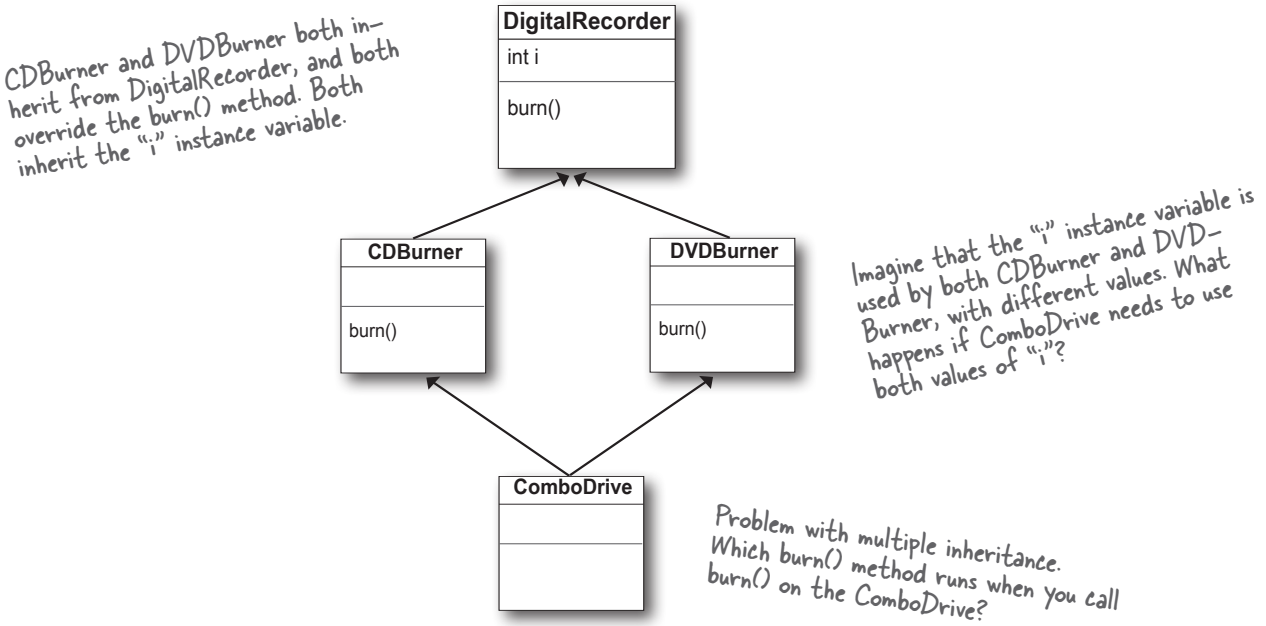
There's just one problem with the "two superclasses" approach...

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



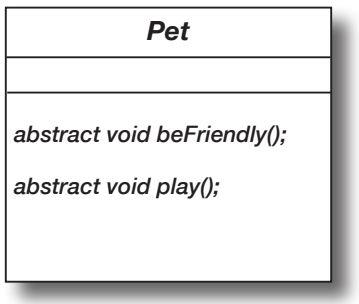
A language that allows the Deadly Diamond of Death can lead to some ugly complexities, because you have to have special rules to deal with the potential ambiguities. And extra rules means extra work for you both in *learning* those rules and watching out for those "special cases." Java is supposed to 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?*

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 {...}
```

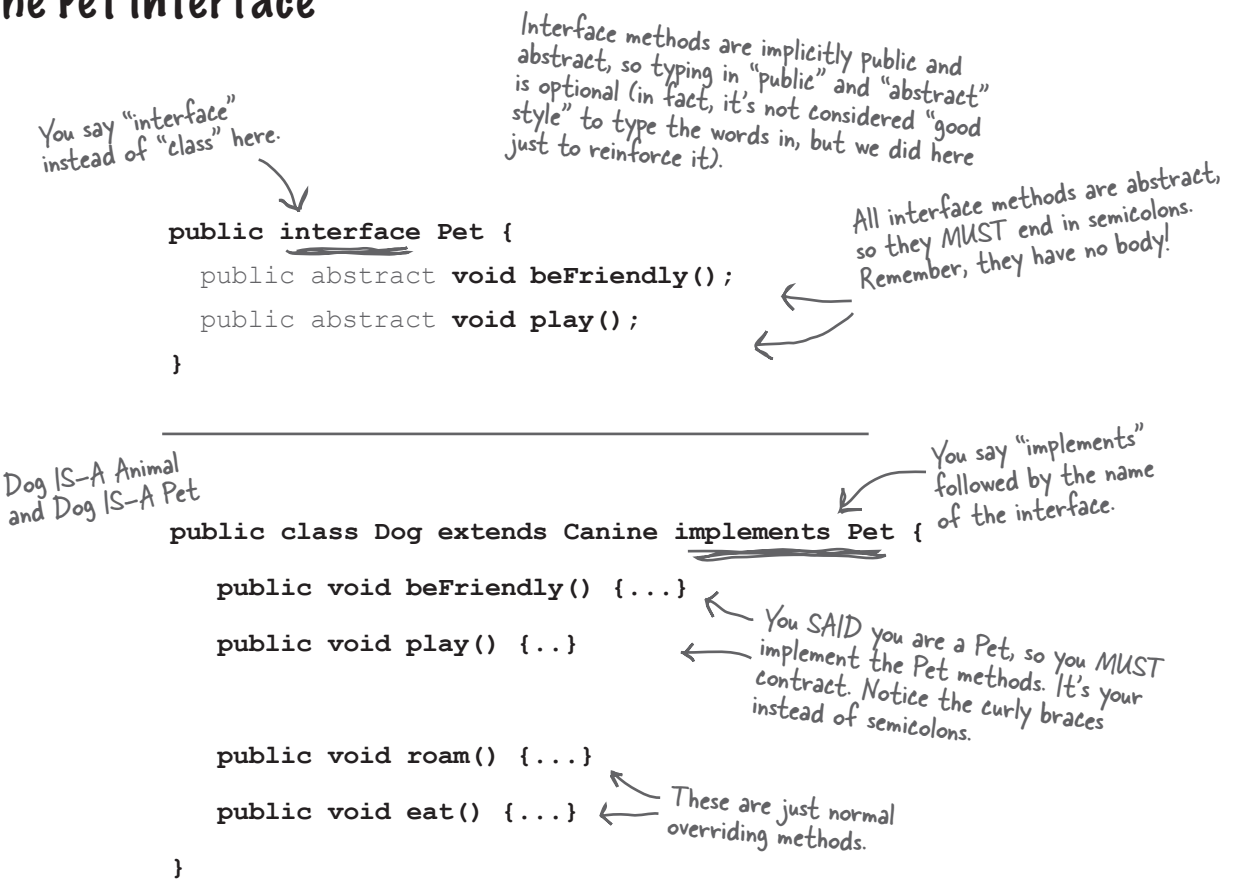
Use the keyword "interface" instead of "class."

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



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?

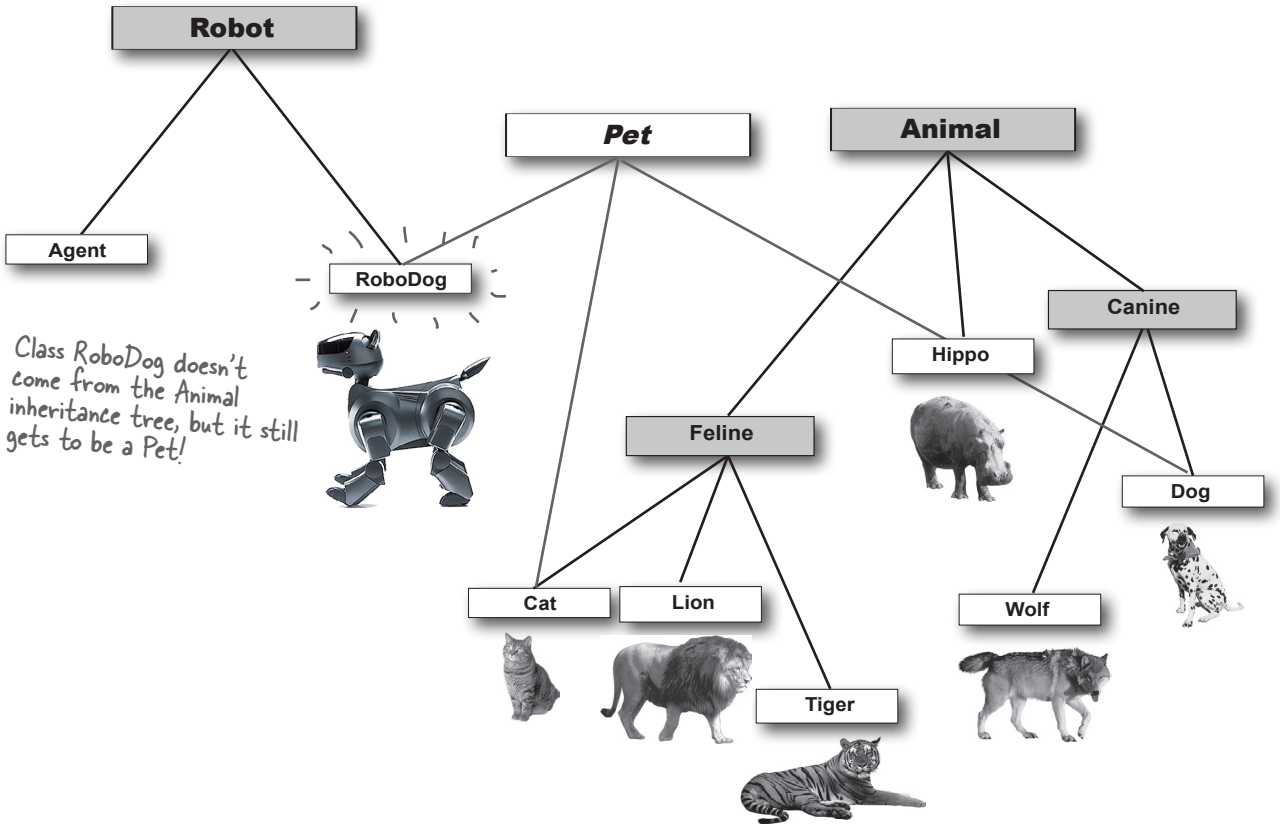
A: Well, actually...there are cases where interfaces can have implementation code (static and default methods, for example), but we're not going to go into them here.

The main purpose of interfaces is polymorphism, polymorphism, polymorphism. Interfaces are the ultimate in flexibility, because if you use interfaces instead of concrete classes (or even abstract classes) as arguments and return types, you can pass anything that implements that interface. And 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 write your code using interfaces, you don't even have to give anyone a superclass to extend. You can 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."

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



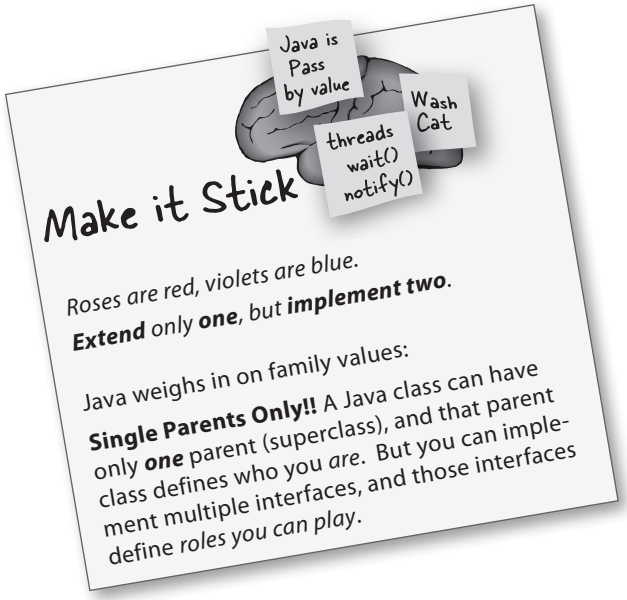
When you use a *class* as a polymorphic type (like an array of type Animal or a method that takes a Canine argument), the objects you can stick in that type must be from the same inheritance tree. But not just anywhere in the inheritance tree; the objects must be from a class that is a subclass of the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

Invoking the superclass version of a method

there are no Dumb Questions

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.

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

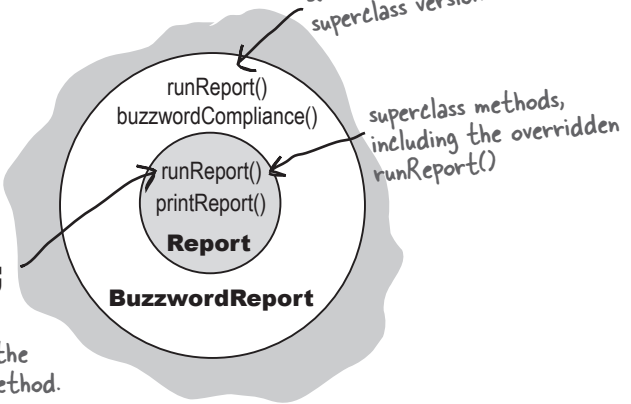
```
abstract class Report {
  void runReport() {
    // set up report
  }
  void printReport() {
    // generic printing
  }
}


class BuzzwordsReport 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.
- When a method is invoked, it will use the object type's implementation of that method.
- 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).
- 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, except default and static methods (which we'll see in Chapter 12).
- To invoke the superclass version of a method from a subclass that's overridden the method, use the **`super`** keyword. Example: `super.runReport();`

there are no Dumb Questions

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. 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 to 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 Chapter 11, *Data Structures*.

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 { }
```

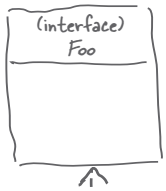
3.
```
public abstract class Muffie implements Whuffie { }
public class Fluffie extends Muffie { }
public interface Whuffie { }
```

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

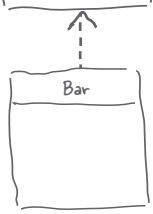
5.
```
public class Gamma extends Delta implements Epsilon { }
public interface Epsilon { }
public interface Beta { }
public class Alpha extends Gamma implements Beta { }
public class Delta { }
```

What's the Picture ?

1.



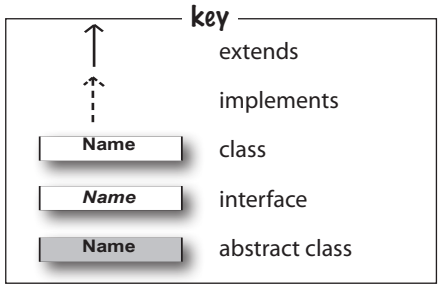
2.



3.

4.

5.

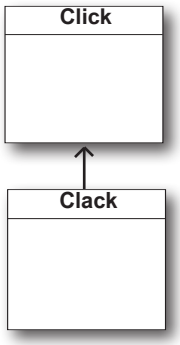


Answers on page 235.

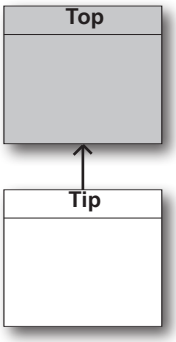
On the left you'll find sets of class diagrams. Your job is to turn these into valid Java declarations. We did number 1 for you (and it was a tough one).

Given:

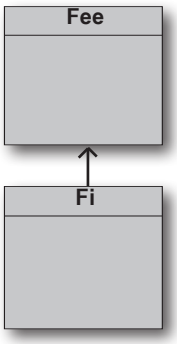
1.



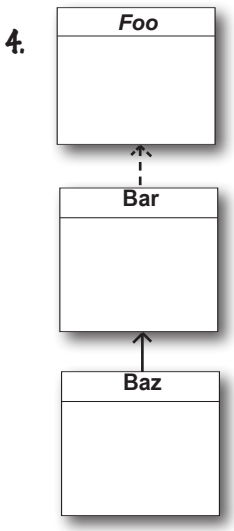
2.



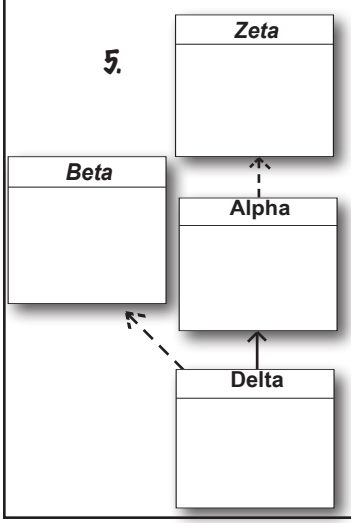
3.



4.



5.



What's the Declaration ?

1. public class Click { }

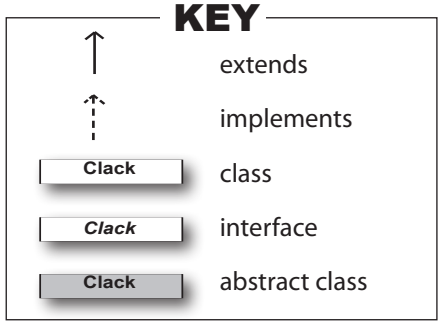
   public class Clack extends Click { }

2.

3.

4.

5.



Answers on page 235.

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.

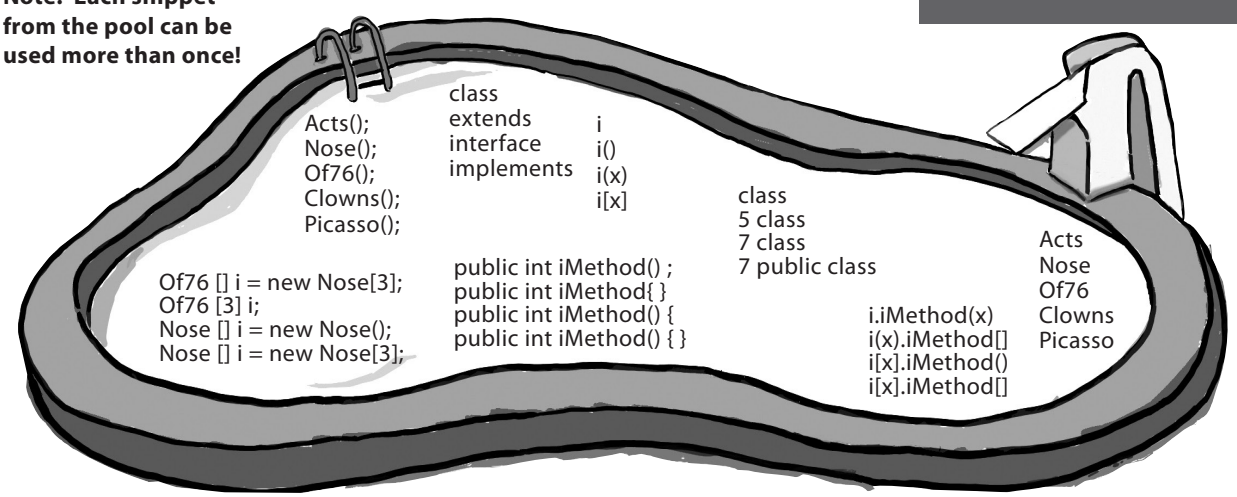
```
_____________ Nose {

  ____________________________

}

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

```
File  Edit  Window  Help  BeAfraid
%java _________________
5 class Acts
7 class Clowns
_________Of76
```

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

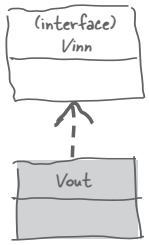


Answers on page 236.

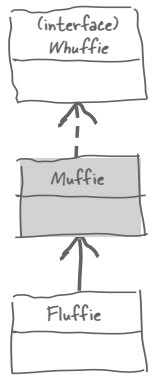
Exercise Solutions

What's the Picture? (from page 232)

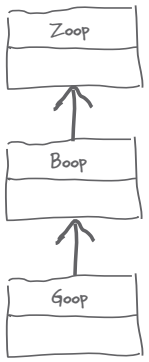
2.



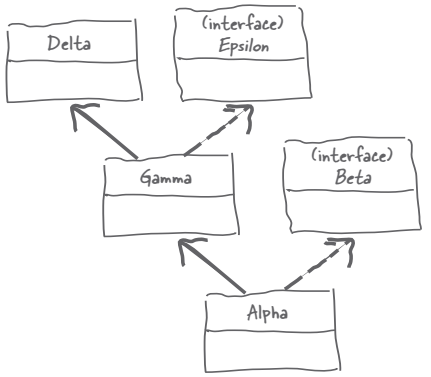
3.



4.



5.



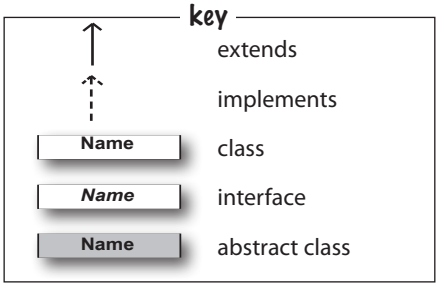
What's the Declaration? (from page 233)

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 { }
```



Pool Puzzle

(from page 234)

```
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
```

Life and Death of an Object

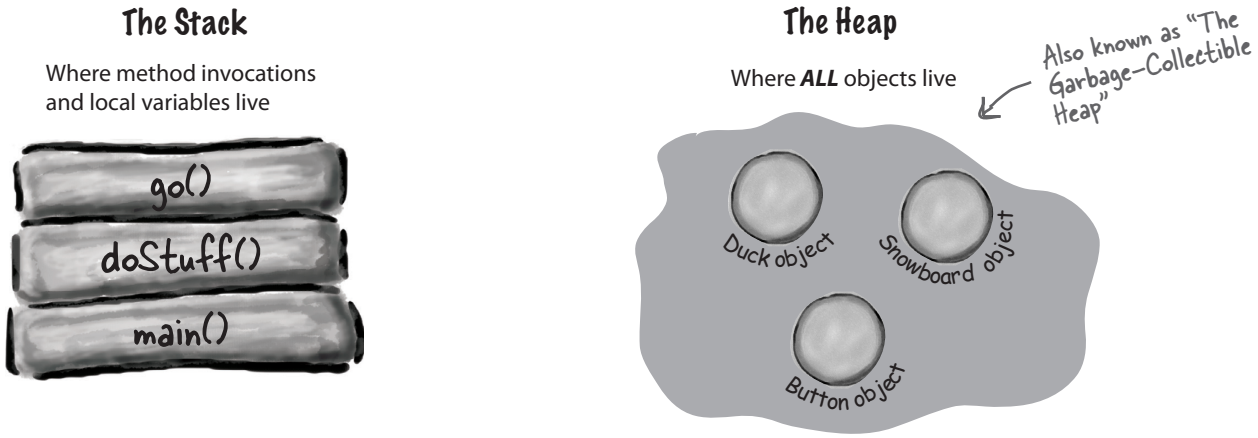
Objects are born and objects die. You're in charge of an object's lifecycle. You decide when and how to **construct** it. You decide when to **destroy** it. Except you don't actually *destroy* the object yourself, you simply *abandon* it. But once it's abandoned, the heartless **Garbage Collector (gc)** can vaporize it, reclaiming the memory that object was using. If you're gonna write Java, you're gonna create objects. Sooner or later, you're gonna have to let some of them go, or risk running out of RAM. In this chapter we look at how objects are created, where they live while they're alive, and how to keep or abandon them efficiently. That means we'll talk about the heap, the stack, scope, constructors, superclass constructors, null references, and more. Warning: this chapter contains material about object death that some may find disturbing. Best not to get too attached.

The Stack and the Heap: where things live

Before we can understand what really happens when you create an object, we have to step back a bit. We need to learn more about where everything lives (and for how long) in Java. That means we need to learn more about two areas of memory—the Stack and the Heap. When a JVM starts up, it gets a chunk of memory from the underlying OS and uses it to run your Java program. How *much* memory, and whether or not you can tweak it, is dependent on which version of the JVM (and on which platform) you're running. But usually you *won't* have any say in the matter. And with good programming, you probably won't care (more on that a little later).

In Java, we (programmers) care about the area of memory where objects live (the heap) and the one where method invocations and local variables live (the stack).

We know that all *objects* live on the garbage-collectible heap, but we haven't yet looked at where *variables* live. And where a variable lives depends on what *kind* of variable it is. And by "kind," we don't mean *type* (i.e., primitive or object reference). The two *kinds* of variables whose lives we care about now are *instance* variables and *local* variables. Local variables are also known as *stack* variables, which is a big clue for where they live.



Instance Variables

Instance variables are declared inside a *class* but *not* inside a method. They represent the "fields" that each individual object has (which can be filled with different values for each instance of the class). Instance variables live inside the object they belong to.

```
public class Duck {

    int size;

}
```

Every Duck has a "size" instance variable.

Local Variables

Local variables are declared inside a *method,* including method parameters. They're temporary and live only as long as the method is on the stack (in other words, as long as the method has not reached the closing curly brace).

```
public void foo(int x) {
  int i = x + 3;
  boolean b = true;
}
```

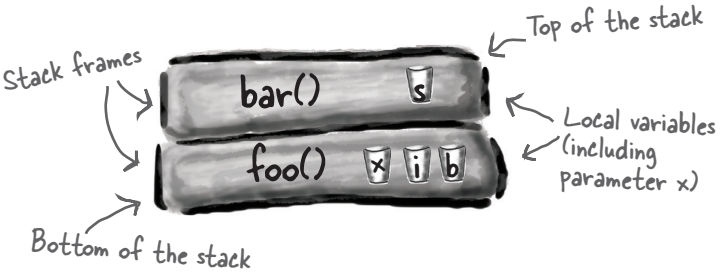
The parameter x and the variables i and b are all local variables.

Methods are stacked

When you call a method, the method lands on the top of a call stack. That new thing that's actually pushed onto the stack is the stack *frame*, and it holds the state of the method including which line of code is executing, and the values of all local variables.

The method at the *top* of the stack is always the currently running method for that stack (for now, assume there's only one stack, but in Chapter 14, *A Very Graphic Story*, we'll add more.) A method stays on the stack until the method hits its closing curly brace (which means the method's done). If method foo() calls method bar(), method bar() is stacked on top of method foo().

A call stack with two methods



The method on the top of the stack is always the currently executing method.

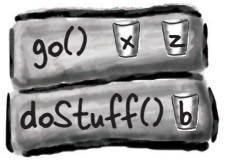
```
public void doStuff() {
  boolean b = true;
  go(4);
}

public void go(int x) {
  int z = x + 24;
  crazy();
  // imagine more code here
}

public void crazy() {
  char c = 'a';
}
```

A stack scenario

The code on the left is a snippet (we don't care what the rest of the class looks like) with three methods. The first method (doStuff()) calls the second method (go()), and the second method calls the third (crazy()). Each method declares one local variable within the body of the method (*b*, *z*, and *c*), and method go() also declares a parameter variable (which means go() has two local variables, *x* and *z*).

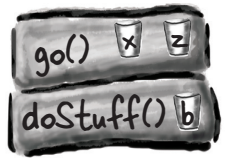
1. Code from another class calls **doStuff()**, and **doStuff()** goes into a stack frame at the top of the stack. The boolean variable named "**b**" goes on the **doStuff()** stack frame.

2. **doStuff()** calls **go()**, and **go()** is *pushed* on top of the stack. Variables "**x**" and "**z**" are in the **go()** stack frame.



3. **go()** calls **crazy()**, **crazy()** is now on the top of the stack, with variable "**c**" in the frame.

4. **crazy()** completes, and its stack frame is *popped* off the stack. Execution goes back to the **go()** method and picks up at the line following the call to **crazy()**.

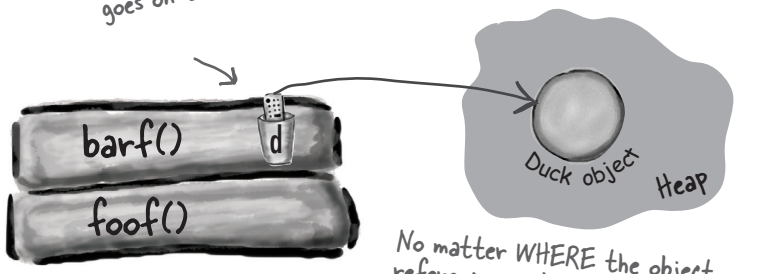


What about local variables that are objects?

Remember, a non-primitive variable holds a *reference* to an object, not the object itself. You already know where objects live—on the heap. It doesn't matter where they're declared or created. ***If the local variable is a reference to an object, only the variable (the reference/remote control) goes on the stack***.

```
public class StackRef {
   public void foof() {
      barf();
   }

   public void barf() {
      Duck d = new Duck();
   }
}
```

barf() declares and creates a new Duck reference variable "d" (since it's declared inside the method, it's a local variable and goes on the stack).



No matter WHERE the object reference variable is declared (inside a method vs. as an instance variable of a class), the object always, always, always goes on the heap.

there are no Dumb Questions

Q: One more time, WHY are we learning the whole stack/heap thing? How does this help me? Do I really need to learn about it?

A: Knowing the fundamentals of the Java Stack and Heap is crucial if you want to understand variable scope, object creation issues, memory management, threads, and exception handling. We cover threads and exception handling in later chapters. You do not need to know anything about *how* the Stack and Heap are implemented in any particular JVM and/or platform. Everything you need to know about the Stack and Heap is on this page and the previous one. If you nail these pages, all the other topics that depend on your knowing this stuff will go much, much, much easier. Once again, some day you will SO thank us for shoving Stacks and Heaps down your throat.

BULLET POINTS

- Java has two areas of memory we care about: the Stack and the Heap.
- Instance variables are variables declared inside a class but outside any method.
- Local variables are variables declared inside a method or method parameter.
- All local variables live on the stack, in the frame corresponding to the method where the variables are declared.
- Object reference variables work just like primitive variables—if the reference is declared as a local variable, it goes on the stack.
- All objects live in the heap, regardless of whether the reference is a local or instance variable.

If local variables live on the stack, where do instance variables live?

When you say new CellPhone(), Java has to make space on the Heap for that CellPhone. But how *much* space? Enough for the object, which means enough to house all of the object's instance variables. That's right, instance variables live on the Heap, *inside* the object they belong to.

Remember that the *values* of an object's instance variables live inside the object. If the instance variables are all primitives, Java makes space for the instance variables based on the primitive type. An int needs 32 bits, a long 64 bits, etc. Java doesn't care about the value inside primitive variables; the bit-size of an int variable is the same (32 bits) whether the value of the int is 32,000,000 or 32.

But what if the instance variables are *objects*? What if CellPhone HAS-A Antenna? In other words, CellPhone has a reference variable of type Antenna.

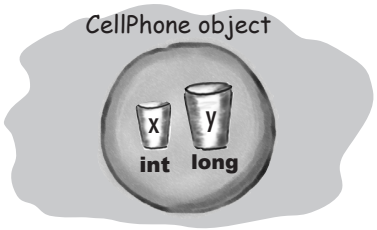
When the new object has instance variables that are object references rather than primitives, the real question is: does the object need space for all of the objects it holds references to? The answer is, *not exactly*. No matter what, Java has to make space for the instance variable *values*. But remember that a reference variable value is not the whole *object*, but merely a *remote control* to the object. So if CellPhone has an instance variable declared as the non-primitive type Antenna, Java makes space within the CellPhone object only for the Antenna's *remote control* (i.e., reference variable) but not the Antenna *object*.

Well, then, when does the Antenna *object* get space on the Heap? First we have to find out *when* the Antenna object itself is created. That depends on the instance variable declaration. If the instance variable is declared but no object is assigned to it, then only the space for the reference variable (the remote control) is created.

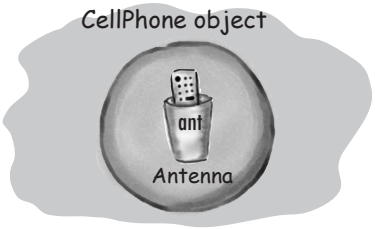
```
private Antenna ant;
```

No actual Antenna object is made on the heap unless or until the reference variable is assigned a new Antenna object.

```
private Antenna ant = new Antenna();
```

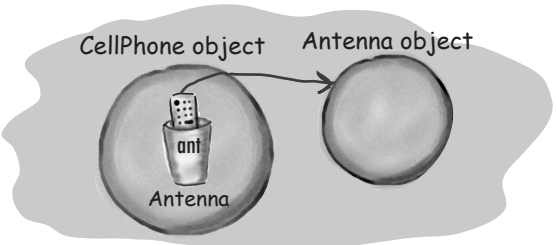


Object with two primitive instance variables. Space for the variables lives in the object.



Object with one non-primitive instance variable—a reference to an Antenna object, but no actual Antenna object. This is what you get if you declare the variable but don't initialize it with an actual Antenna object.

```
public class CellPhone {
  private Antenna ant;
}
```



Object with one non-primitive instance variable, and the Antenna variable is assigned a new Antenna object.

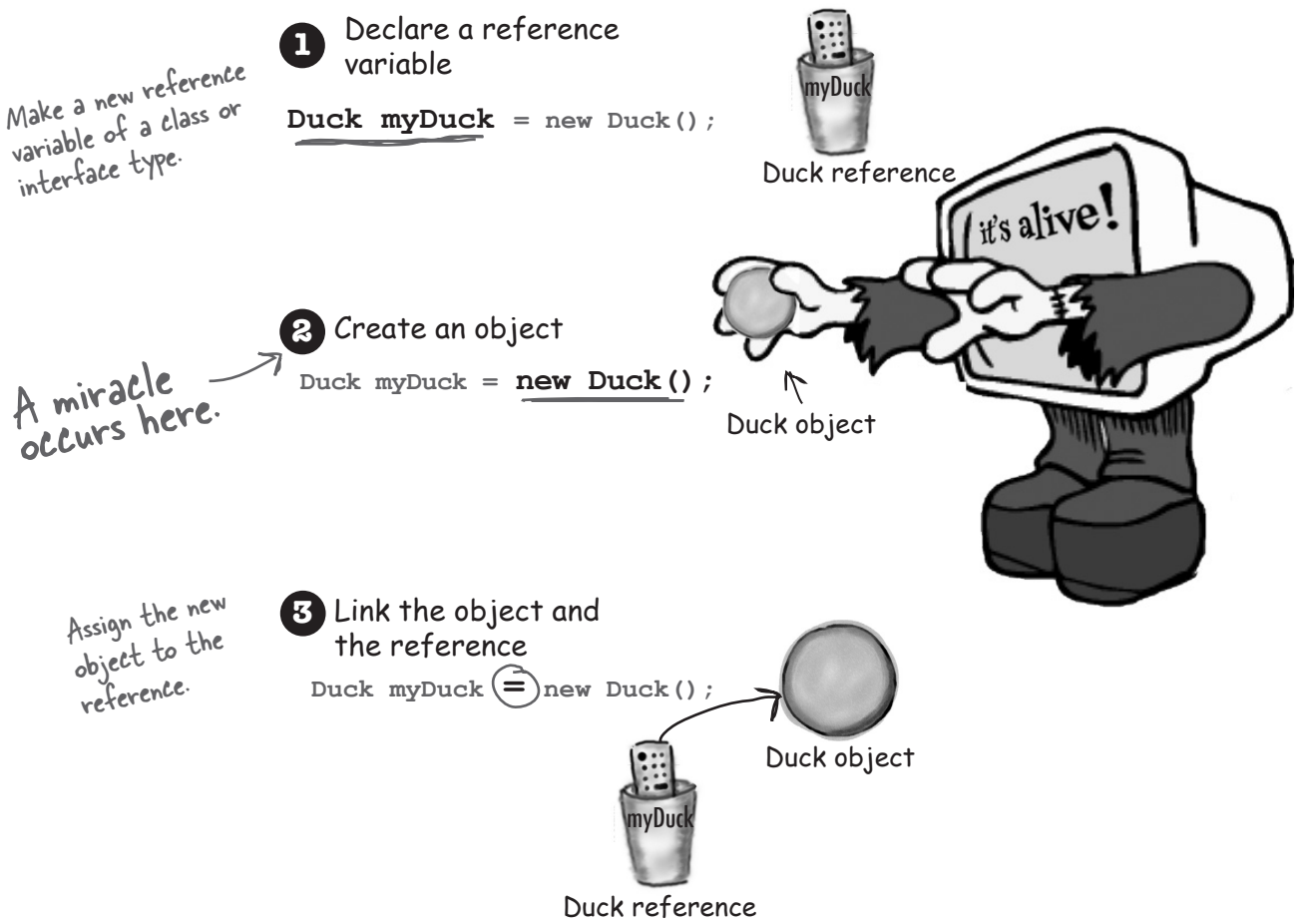
```
public class CellPhone {
  private Antenna ant = new Antenna();
}
```

The miracle of object creation

Now that you know where variables and objects live, we can dive into the mysterious world of object creation. Remember the three steps of object declaration and assignment: declare a reference variable, create an object, and assign the object to the reference.

But until now, step two—where a miracle occurs and the new object is "born"—has remained a Big Mystery. Prepare to learn the facts of object life. *Hope you're not squeamish.*

Let's review the 3 steps of object declaration, creation and assignment:



Are we calling a method named Duck()? Because it sure *looks* like it.

```
Duck myDuck = new Duck();
```

It looks like we're calling a method named Duck(), because of the parentheses.

No.

We're calling the Duck *constructor*.

A constructor *does* look and feel a lot like a method, but it's not a method. It's got the code that runs when you say **new**. In other words, *the code that runs when you instantiate an object.*

The only way to invoke a constructor is with the keyword **new** followed by the class name. The JVM finds that class and invokes the constructor in that class. (OK, technically this isn't the *only* way to invoke a constructor. But it's the only way to do it from *outside* a constructor. You *can* call a constructor from within another constructor, with restrictions, but we'll get into all that later in the chapter.)

A constructor has the code that runs when you instantiate an object. In other words, the code that runs when you say new on a class type.

Every class you write has a constructor, even if you don't write it yourself.

But where is the constructor?

If we didn't write it, who did?

You can write a constructor for your class (we're about to do that), but if you don't, ***the compiler writes one for you!***

Here's what the compiler's default constructor looks like:

```
public  Duck() {

}
```

Notice something missing? How is this different from a method?

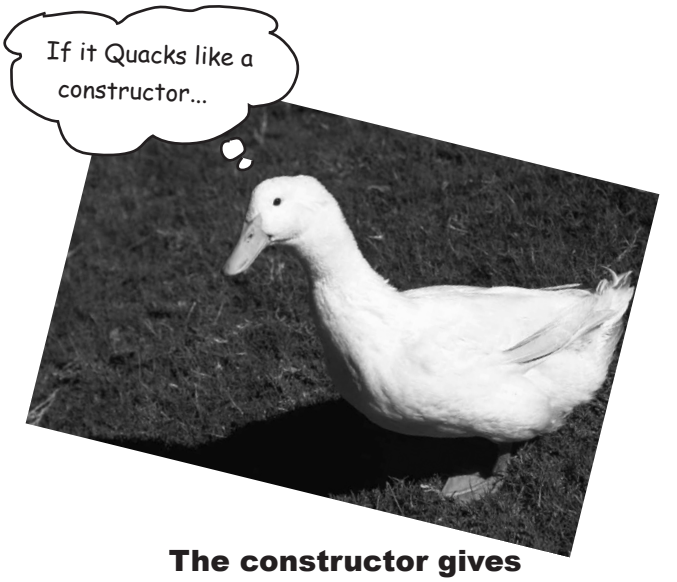
Where's the return type? If this were a method, you'd need a return type between "public" and "Duck()."

Its name is the same as the class name. That's mandatory.

```
public  Duck() {
    // constructor code goes here
}
```

Construct a Duck

The key feature of a constructor is that it runs *before* the object can be assigned to a reference. That means you get a chance to step in and do things to get the object ready for use. In other words, before anyone can use the remote control for an object, the object has a chance to help construct itself. In our Duck constructor, we're not doing anything useful, just demonstrating the sequence of events.



The constructor gives you a chance to step into the middle of `new`.

```
public class Duck {

   public Duck() {
      System.out.println("Quack");
   }
}
```

Constructor code.

```
public class UseADuck {

   public static void main (String[] args) {
      Duck d = new Duck();
   }
}
```

This calls the Duck constructor.

```
File Edit Window Help Quack
% java UseADuck
Quack
```

Sharpen your pencil

→ Yours to solve.

A constructor lets you jump into the middle of the object creation step—into the middle of **`new`**. Can you imagine conditions where that would be useful? Which of the actions on the right might be useful in a Car class constructor, if the Car is part of a Racing Game? Check off the ones that you came up with a scenario for.

- ■ Increment a counter to track how many objects of this class type have been made.
- ■ Assign runtime-specific state (data about what's happening NOW).
- ■ Assign values to the object's important instance variables.
- ■ Get and save a reference to the object that's *creating* the new object.
- ■ Add the object to an ArrayList.
- ■ Create HAS-A objects.
- ■ _____ (your idea here)

Initializing the state of a new Duck

Most people use constructors to initialize the state of an object. In other words, to make and assign values to the object's instance variables.

```
public Duck() {
    size = 34;
}
```

That's all well and good when the Duck class *developer* knows how big the Duck object should be. But what if we want the programmer who is *using* Duck to decide how big a particular Duck should be?

Imagine the Duck has a size instance variable, and you want the programmer using your Duck class to set the size of the new Duck. How could you do it?

Well, you could add a setSize() setter method to the class. But that leaves the Duck temporarily without a size* and forces the Duck user to write *two* statements—one to create the Duck, and one to call the setSize() method. The code below uses a setter method to set the initial size of the new Duck.

```
public class Duck {
  int size;                          ← Instance variable

  public Duck() {
    System.out.println("Quack");     ← Constructor
  }

  public void setSize(int newSize) {
    size = newSize;                  ← Setter method
  }
}
```

```
public class UseADuck {

  public static void main(String[] args) {
    Duck d = new Duck();

    d.setSize(42);   ← There's a bad thing here. The Duck is alive at this point in the code, but without a size!* And then you're relying on the Duck user to KNOW that Duck creation is a two-part process: one to call the constructor and one to call the setter.
  }
}
```

*Instance variables do have a default value. 0 or 0.0 for numeric primitives, false for booleans, and null for references.

there are no Dumb Questions

Q: Why do you need to write a constructor if the compiler writes one for you?

A: If you need code to help initialize your object and get it ready for use, you'll have to write your own constructor. You might, for example, be dependent on input from the user before you can finish making the object ready. There's another reason you might have to write a constructor, even if you don't need any constructor code yourself. It has to do with your superclass constructor, and we'll talk about that soon.

Q: How can you tell a constructor from a method? Can you also have a method that's the same name as the class?

A: Java lets you declare a method with the same name as your class. That doesn't make it a constructor, though. The thing that separates a method from a constructor is the return type. Methods *must* have a return type, but constructors *cannot* have a return type.

```
public Duck() { } Constructor

public void Duck() { } Method
          ↑ Return type
```

The compiler will allow these methods but **don't do this**. It's against normal naming conventions (methods start with a lowercase letter) but more importantly it's super confusing.

Q: Are constructors inherited? If you don't provide a constructor but your superclass does, do you get the superclass constructor instead of the default?

A: Nope. Constructors are not inherited. We'll look at that in just a few pages.

Using the constructor to initialize important Duck state*

If an object shouldn't be used until one or more parts of its state (instance variables) have been initialized, don't let anyone get hold of a Duck object until you're finished initializing! It's usually way too risky to let someone make—and get a reference to—a new Duck object that isn't quite ready for use until that someone turns around and calls the setSize() method. How will the Duck user even *know* that he's required to call the setter method after making the new Duck?

The best place to put initialization code is in the constructor. And all you need to do is make a constructor with arguments.


```
public class Duck {
  int size;

  public Duck(int duckSize) {
    System.out.println("Quack");

    size = duckSize;

    System.out.println("size is " + size);
  }
}
```

Add an int parameter to the Duck constructor.

Use the argument value to set the size instance variable. We could have called the setSize method instead.

```
public class UseADuck {

  public static void main (String[] args) {
    Duck d = new Duck(42);
  }
}
```

Pass a value to the constructor.

This time there's only one statement. We make the new Duck and set its size in one statement.

```
File  Edit  Window  Help  Honk
% java UseADuck
Quack
size is 42
```

*Not to imply that not all Duck state is not unimportant.

Make it easy to make a Duck

Be sure you have a no-arg constructor

What happens if the Duck constructor takes an argument? Think about it. On the previous page, there's only *one* Duck constructor—and it takes an int argument for the *size* of the Duck. That might not be a big problem, but it does make it harder for a programmer to create a new Duck object, especially if the programmer doesn't *know* what the size of a Duck should be. Wouldn't it be helpful to have a default size for a Duck so that if the user doesn't know an appropriate size, they can still make a Duck that works?

Imagine that you want Duck users to have TWO options for making a Duck—one where they supply the Duck size (as the constructor argument) and one where they don't specify a size and thus get your default Duck size.

You can't do this cleanly with just a single constructor. Remember, if a method (or constructor—same rules) has a parameter, you *must* pass an appropriate argument when you invoke that method or constructor. You can't just say, "If someone doesn't pass anything to the constructor, then use the default size" because they won't even be able to compile without sending an int argument to the constructor call. You *could* do something clunky like this:

```
public class Duck {
  int size;

  public Duck(int newSize) {
    if (newSize == 0) {
      size = 27;
    } else {
      size = newSize;
    }
  }
}
```

If the parameter value is zero, give the new Duck a default size; otherwise, use the parameter value for the size. NOT a very good solution.

But that means the programmer making a new Duck object has to *know* that passing a "0" is the protocol for getting the default Duck size. Pretty ugly. What if the other programmer doesn't know that? Or what if they really *do* want a zero-sized Duck? (Assuming a zero-sized Duck is allowed. If you don't want zero-sized Duck objects, put validation code in the constructor to prevent it.) The point is, it might not always be possible to distinguish between a genuine "I want zero for the size" constructor argument and a "I'm sending zero so you'll give me the default size, whatever that is" constructor argument.

You really want TWO ways to make a new Duck:

```
public class Duck2 {
  int size;

  public Duck2() {
    // supply default size
    size = 27;
  }

  public Duck2(int duckSize) {
    // use duckSize parameter
    size = duckSize;
  }
}
```

To make a Duck when you know the size:

```
Duck2 d = new Duck2(15);
```

To make a Duck when you do *not* know the size:

```
Duck2 d2 = new Duck2();
```

So this two-options-to-make-a-Duck idea needs *two* constructors. One that takes an int and one that doesn't. ***If you have more than one constructor in a class, it means you have overloaded constructors.***

Doesn't the compiler always make a no-arg constructor for you? No!

You might think that if you write *only* a constructor with arguments, the compiler will see that you don't have a no-arg constructor and stick one in for you. But that's not how it works. The compiler gets involved with constructor-making *only if you don't say anything at all about constructors*.

If you write a constructor that takes arguments and you *still* want a no-arg constructor, you'll have to build the no-arg constructor yourself!

As soon as *you* provide a constructor, ANY kind of constructor, the compiler backs off and says, "OK fair enough, looks like you're in charge of constructors now."

If you have more than one constructor in a class, the constructors MUST have different argument lists.

The argument list includes the order and types of the arguments. As long as they're different, you can have more than one constructor. You can do this with methods as well, but we'll get to that in another chapter.

Overloaded constructors means you have more than one constructor in your class.

To compile, each constructor must have a *different* argument list!

The class below is legal because all five constructors have different argument lists. If you had two constructors that took only an int, for example, the class wouldn't compile. What you name the parameter variable doesn't count. It's the variable *type* (int, Dog, etc.) and *order* that matters. You *can* have two constructors that have identical types, ***as long as the order is different***. A constructor that takes a String followed by an int is *not* the same as one that takes an int followed by a String.

Five different constructors means five different ways to make a new mushroom.

```
public class Mushroom {

  public Mushroom(int size) { }

  public Mushroom( ) { }

  public Mushroom(boolean isMagic) { }

  public Mushroom(boolean isMagic, int size) { }

  public Mushroom(int size, boolean isMagic) { }
}
```

When you know the size, but you don't know if it's magic

When you don't know anything

When you know if it's magic or not, but don't know the size

These two have the same args, but in a different order, so it's OK*

When you know whether or not it's magic, AND you know the size as well

*If the arguments were the same type, how would the compiler know they were two different things?

BULLET POINTS

- Instance variables live within the object they belong to, on the Heap.
- If the instance variable is a reference to an object, both the reference and the object it refers to are on the Heap.
- A constructor is the code that runs when you say **new** on a class type.
- A constructor must have the same name as the class, and must *not* have a return type.
- You can use a constructor to initialize the state (i.e., the instance variables) of the object being constructed.
- If you don't put a constructor in your class, the compiler will put in a default constructor.
- The default constructor is always a no-arg constructor.
- If you put a constructor—any constructor—in your class, the compiler will not build the default constructor.
- If you want a no-arg constructor and you've already put in a constructor with arguments, you'll have to build the no-arg constructor yourself.
- Always provide a no-arg constructor if you can, to make it easy for programmers to make a working object. Supply default values.
- Overloaded constructors means you have more than one constructor in your class.
- Overloaded constructors must have different argument lists.
- You cannot have two constructors with the same argument lists. An argument list includes the order and type of arguments.
- Instance variables are assigned a default value, even when you don't explicitly assign one. The default values are 0/0.0/false for primitives, and null for references.

Sharpen your pencil → Yours to solve.

Match the `new Duck()` call with the constructor that runs when that Duck is instantiated. We did the easy one to get you started.

```
public class TestDuck {

  public static void main(String[] args) {
    int weight = 8;
    float density = 2.3F;
    String name = "Donald";
    long[] feathers = {1, 2, 3, 4, 5, 6};
    boolean canFly = true;
    int airspeed = 22;

    Duck[] d = new Duck[7];

    d[0] = new Duck();

    d[1] = new Duck(density, weight);

    d[2] = new Duck(name, feathers);

    d[3] = new Duck(canFly);

    d[4] = new Duck(3.3F, airspeed);

    d[5] = new Duck(false);

    d[6] = new Duck(airspeed, density);
  }
}
```

```
class Duck {
  private int kilos = 6;
  private float floatability = 2.1F;
  private String name = "Generic";
  private long[] feathers = {1, 2, 3,
                             4, 5, 6, 7};
  private boolean canFly = true;
  private int maxSpeed = 25;

  public Duck() {
    System.out.println("type 1 duck");
  }

  public Duck(boolean fly) {
    canFly = fly;
    System.out.println("type 2 duck");
  }

  public Duck(String n, long[] f) {
    name = n;
    feathers = f;
    System.out.println("type 3 duck");
  }

  public Duck(int w, float f) {
    kilos = w;
    floatability = f;
    System.out.println("type 4 duck");
  }

  public Duck(float density, int max) {
    floatability = density;
    maxSpeed = max;
    System.out.println("type 5 duck");
  }
}
```

there are no Dumb Questions

Q: Earlier you said that it's good to have a no-argument constructor so that if people call the no-arg constructor, we can supply default values for the "missing" arguments. But aren't there times when it's impossible to come up with defaults? Are there times when you should not have a no-arg constructor in your class?

A: You're right. There are times when a no-arg constructor doesn't make sense. You'll see this in the Java API—some classes don't have a no-arg constructor. The Color class, for example, represents a...color. Color objects are used to, for example, set or change the color of a screen font or GUI button. When you make a Color instance, that instance is of a particular color (you know, Death-by-Chocolate Brown, Blue-Screen-of-Death Blue, Scandalous Red, etc.).

If you make a Color object, you must specify the color in some way.

```
Color c = new Color(3,45,200);
```

(We're using three ints for RGB values here. We'll get into using Color later, in Chapter 15, *Work on Your Swing*.) Otherwise, what would you get? The Java API programmers *could* have decided that if you call a no-arg Color constructor you'll get a lovely shade of mauve. But good taste prevailed. If you try to make a Color without supplying an argument:

```
Color c = new Color();
```

the compiler freaks out because it can't find a matching no-arg constructor in the Color class.

```
File Edit Window Help StopBeingStupid
cannot resolve symbol
:constructor Color()
location: class java.awt.
Color
Color c = new Color();
          ^
1 error
```

Nanoreview: four things to remember about constructors

1. A constructor is the code that runs when somebody says **new** on a class type:

```
Duck d = new Duck();
```

2. A constructor must have the same name as the class, and ***no*** return type:

```
public Duck(int size) { }
```

3. If you don't put a constructor in your class, the compiler puts in a default constructor. The default constructor is always a no-arg constructor.

```
public Duck() { }
```

4. You can have more than one constructor in your class, as long as the argument lists are different. Having more than one constructor in a class means you have overloaded constructors.

```
public Duck() { }

public Duck(int size) { }

public Duck(String name) { }

public Duck(String name, int size) { }
```

*Doing all the Brain Power exercises has been shown to produce a 42% increase in neuron size. And you know what they say, "Big neurons..."

What about superclasses?

When you make a Dog, should the Canine constructor run too?

If the superclass is abstract, should it even *have* a constructor?

We'll look at this on the next few pages, so stop now and think about the implications of constructors and superclasses.*

there are no Dumb Questions

Q: Do constructors have to be `public`?

A: No. Constructors can be **`public`**, **`protected`**, **`private`**, or *default* (which means no access modifier at all). We'll look more at *default* access in appendix B.

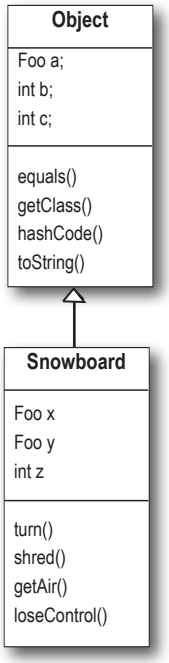
Q: How could a private constructor ever be useful? Nobody could ever call it, so nobody could ever make a new object!

A: Not exactly right. Marking something **`private`** doesn't mean *nobody* can access it; it just means that *nobody outside the class* can access it. Bet you're thinking Catch 22. Only code from the *same* class as the class-with-private-constructor can make a new object from that class, but without first making an object, how do you ever get to run code from that class in the first place? How do you ever get to anything in that class? *Patience grasshopper.* We'll get there in the next chapter.

Wait a minute...we never DID talk about superclasses and inheritance and how that all fits in with constructors

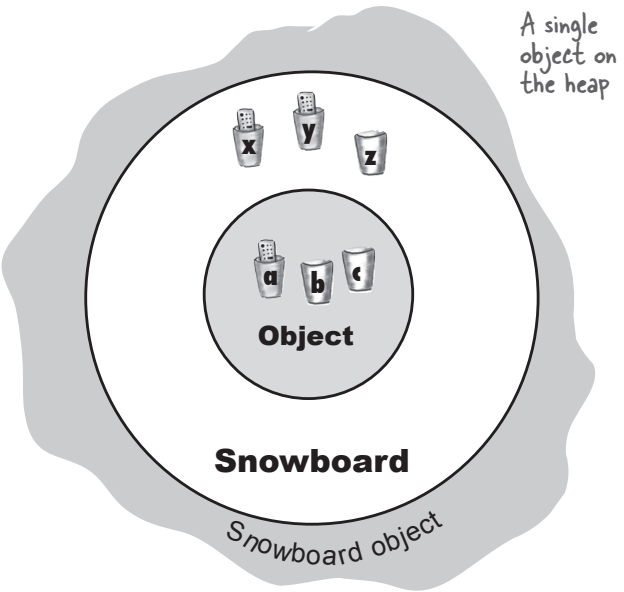
Here's where it gets fun. Remember in the previous chapter we looked at the Snowboard object wrapping around an inner core representing the Object portion of the Snowboard class? The Big Point there was that every object holds not just its *own* declared instance variables, but also *everything from its superclasses* (which, at a minimum, means class Object, since *every* class extends Object).

So when an object is created (because somebody said **new**; there is ***no other way*** to create an object other than someone, somewhere saying **new** on the class type), the object gets space for *all* the instance variables, from all the way up the inheritance tree. Think about it for a moment... a superclass might have setter methods encapsulating a private variable. But that variable has to live *somewhere*. When an object is created, it's almost as though *multiple* objects materialize—the object being new'd and one object per each superclass. Conceptually, though, it's much better to think of it like the picture below, where the object being created has *layers* of itself representing each superclass.



Object has instance variables encapsulated by access methods. Those instance variables are created when any subclass is instantiated. (These aren't the *REAL* Object variables, but we don't care what they are since they're encapsulated.)

Snowboard also has instance variables of its own, so to make a Snowboard object we need space for the instance variables of both classes.



There is only *ONE* object on the heap here. A Snowboard object. But it contains both the Snowboard parts of itself and the Object parts of itself. All instance variables from both classes have to be here.

The role of superclass constructors in an object's life

All the constructors in an object's inheritance tree must run when you make a new object.

Let that sink in.

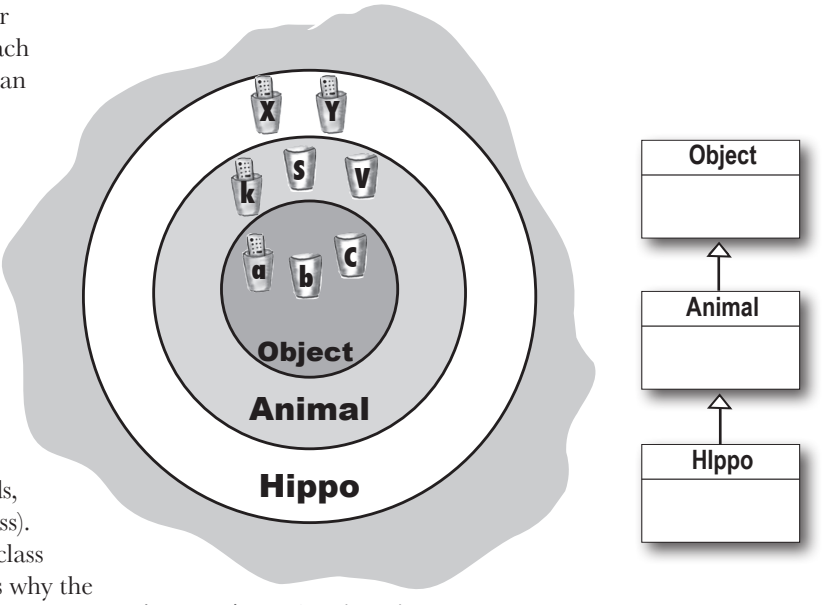
That means every superclass has a constructor (because every class has a constructor), and each constructor up the hierarchy runs at the time an object of a subclass is created.

Saying **new** is a Big Deal. It starts the whole constructor chain reaction. And yes, even abstract classes have constructors. Although you can never say new on an abstract class, an abstract class is still a superclass, so its constructor runs when someone makes an instance of a concrete subclass.

The superclass constructors run to build out the superclass parts of the object. Remember, a subclass might inherit methods that depend on superclass state (in other words, the value of instance variables in the superclass). For an object to be fully formed, all the superclass parts of itself must be fully formed, and that's why the superclass constructor *must* run. All instance variables from every class in the inheritance tree have to be declared and initialized. Even if Animal has instance variables that Hippo doesn't inherit (if the variables are private, for example), the Hippo still depends on the Animal methods that *use* those variables.

When a constructor runs, it immediately calls its superclass constructor, all the way up the chain until you get to the class Object constructor.

On the next few pages, you'll learn how superclass constructors are called, and how you can call them yourself. You'll also learn what to do if your superclass constructor has arguments!



A single Hippo object on the heap

A new Hippo object also IS-A Animal and IS-A Object. If you want to make a Hippo, you must also make the Animal and Object parts of the Hippo.

This all happens in a process called Constructor Chaining.

Making a Hippo means making the Animal and Object parts too...

```
public class Animal {
   public Animal() {
      System.out.println("Making an Animal");
   }
}
```

```
public class Hippo extends Animal {
  public Hippo() {
    System.out.println("Making a Hippo");
  }
}
```

```
public class TestHippo {
  public static void main(String[] args) {
    System.out.println("Starting...");
    Hippo h = new Hippo();
  }
}
```

Given the class hierarchy in the code above, we can step through the process of creating a new Hippo object.

Sharpen your pencil

What's the real output? Given the code on the left, what prints out when you run TestHippo? A or B?

(The answer is at the bottom of the page.)

A

```
File Edit Window Help Swear
% java TestHippo
Starting...
Making an Animal
Making a Hippo
```

B

```
File Edit Window Help Swear
% java TestHippo
Starting...
Making a Hippo
Making an Animal
```

1. Code from another class calls **new Hippo()**, and the **Hippo()** constructor goes into a stack frame at the top of the stack.

2. **Hippo()** invokes the superclass constructor, which pushes the **Animal()** constructor onto the top of the stack.

3. **Animal()** invokes the superclass constructor, which pushes the **Object()** constructor onto the top of the stack, since Object is the superclass of Animal.

4. **Object()** completes, and its stack frame is *popped* off the stack. Execution goes back to the **Animal()** constructor and picks up at the line following Animal's call to its superclass constructor.

The first one, A. The Hippo() constructor is invoked first, but it's the Animal constructor that finishes first.

How do you invoke a superclass constructor?

You might think that somewhere in, say, a Duck constructor, if Duck extends Animal you'd call Animal(). But that's not how it works:

```
public class Duck extends Animal {
  int size;

  public Duck(int newSize) {
    Animal();
    size = newSize;
  }
}
```

BAD! → `Animal();` ← NO! This is not legal!

The only way to call a superclass constructor is by calling ***super()***. That's right—***super()*** calls the ***superclass constructor.***

What are the odds?

```
public class Duck extends Animal {
  int size;

  public Duck(int newSize) {
    super();
    size = newSize;
  }
}
```

`super();` ← you just call super()

A call to super() in your constructor puts the superclass constructor on the top of the Stack. And what do you think that superclass constructor does? *Calls its superclass constructor.* And so it goes until the Object constructor is on the top of the Stack. Once Object() finishes, it's popped off the Stack, and the next thing down the Stack (the subclass constructor that called Object()) is now on top. *That* constructor finishes and so it goes until the original constructor is on the top of the Stack, where *it* can now finish.

And how is it that we've gotten away without calling super() before?

You probably figured that out.

Our good friend the compiler puts in a call to *super()* if you don't.

So the compiler gets involved in constructor-making in *two* ways:

1 If you *don't* provide a constructor

The compiler puts one in that looks like:

```
public ClassName() {
  super();
}
```

2 If you *do* provide a constructor but you do *not* put in the call to super()

The compiler will put a call to super() in each of your overloaded constructors.* The compiler-supplied call looks like:

```
super();
```

It always looks like that. The compiler-inserted call to super() is always a no-arg call. If the superclass has overloaded constructors, only the no-arg constructor is called.

*Unless the constructor calls another overloaded constructor (you'll see that in a few pages).

Can the child exist before the parents?

If you think of a superclass as the parent to the subclass child, you can figure out which has to exist first. ***The superclass parts of an object have to be fully formed (completely built) before the subclass parts can be constructed***. Remember, the subclass object might depend on things it inherits from the superclass, so it's important that those inherited things be finished. No way around it. The superclass constructor must finish before its subclass constructor.

Look at the Stack series on page 254 again, and you can see that while the Hippo constructor is the *first* to be invoked (it's the first thing on the Stack), it's the *last* one to complete! Each subclass constructor immediately invokes its own superclass constructor, until the Object constructor is on the top of the Stack. Then Object's constructor completes, and we bounce back down the Stack to Animal's constructor. Only after Animal's constructor completes do we finally come back down to finish the rest of the Hippo constructor. For that reason:

The call to super() must be the *first* statement in each constructor!*

Possible constructors for class Boop

☑
```
public Boop() {
    super();
}
```
These are OK because the programmer explicitly coded the call to super() as the first statement.

☑
```
public Boop(int i) {
    super();
    size = i;
}
```

☑
```
public Boop() {
}
```
These are OK because the compiler will put a call to super() in as the first statement.

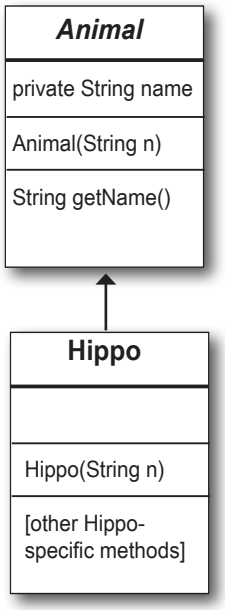
☑
```
public Boop(int i) {
    size = i;
}
```

⊘
```
public Boop(int i) {
    size = i;
    super();
}
```
BAD!! This won't compile! You can't explicitly put the call to super() below anything else.

*There's an exception to this rule; you'll learn it on page 258.

Superclass constructors with arguments

What if the superclass constructor has arguments? Can you pass something in to the super() call? Of course. If you couldn't, you'd never be able to extend a class that didn't have a no-arg constructor. Imagine this scenario: all animals have a name. There's a getName() method in class Animal that returns the value of the name instance variable. The instance variable is marked private, but the subclass (in this case, Hippo) inherits the getName() method. So here's the problem: Hippo has a getName() method (through inheritance) but does not have the name instance variable. Hippo has to depend on the Animal part of himself to keep the name instance variable, and return it when someone calls getName() on a Hippo object. But...how does the Animal part get the name? The only reference Hippo has to the Animal part of himself is through super(), so that's the place where Hippo sends the Hippo's name up to the Animal part of himself, so that the Animal part can store it in the private name instance variable.




```
public abstract class Animal {
  private String name;
  public String getName() {
    return name;
  }
  public Animal(String theName) {
    name = theName;
  }
}
```

All animals (including subclasses) have a name.

A getter method that Hippo inherits.

The constructor that takes the name and assigns it the name instance variable.

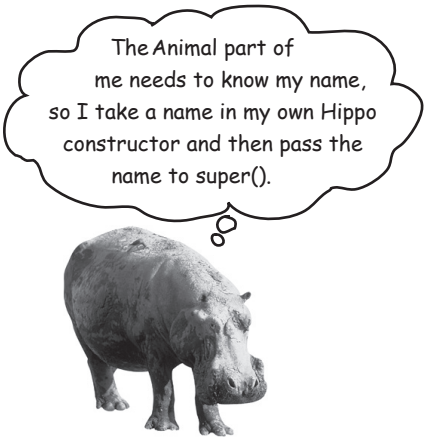
```
public class Hippo extends Animal {
  public Hippo(String name) {
    super(name);
  }
}
```

Hippo constructor takes a name.

It sends the name up the Stack to the Animal constructor.

```
public class MakeHippo {
  public static void main(String[] args) {
    Hippo h = new Hippo("Buffy");
    System.out.println(h.getName());
  }
}
```

Make a Hippo, passing the name "Buffy" to the Hippo constructor. Then call the Hippo's inherited getName().




```
File Edit Window Help Hide
%java MakeHippo
Buffy
```

Invoking one overloaded constructor from another

What if you have overloaded constructors that, with the exception of handling different argument types, all do the same thing? You know that you don't want *duplicate* code sitting in each of the constructors (pain to maintain, etc.), so you'd like to put the bulk of the constructor code (including the call to super()) in only *one* of the overloaded constructors. You want whichever constructor is first invoked to call The Real Constructor and let The Real Constructor finish the job of construction. It's simple: just say *this()*. Or *this(aString)*. Or *this(27, x)*. In other words, just imagine that the keyword *this* is a reference to **the current object.**

You can say *this()* only within a constructor, and it must be the first statement in the constructor!

But that's a problem, isn't it? Earlier we said that super() must be the first statement in the constructor. Well, that means you get a choice.

Use this() to call a constructor from another overloaded constructor in the same class.

The call to this() can be used only in a constructor, and must be the first statement in a constructor.

A constructor can have a call to super() OR this(), but never both!

Every constructor can have a call to super() or this(), but never both!

You'll need to choose which to call based on which values you have, which ones you need to set, and which constructors are provided in this class or the superclass.

```
import java.awt.Color;

class Mini extends Car {
  private Color color;

  public Mini() {
    this(Color.RED);
  }

  public Mini(Color c) {
    super("Mini");
    color = c;
    // more initialization
  }

  public Mini(int size) {
    this(Color.RED);
    super(size);
  }
}
```

The no-arg constructor supplies a default Color and calls the overloaded Real Constructor (the one that calls super()).

This is The Real Constructor that does The Real Work of initializing the object (including the call to super()).

Won't work!! Can't have super() and this() in the same constructor, because they each must be the first statement!

```
File Edit Window Help Drive
javac Mini.java

Mini.java:16: call to super must
be first statement in constructor

      super();
            ^
```

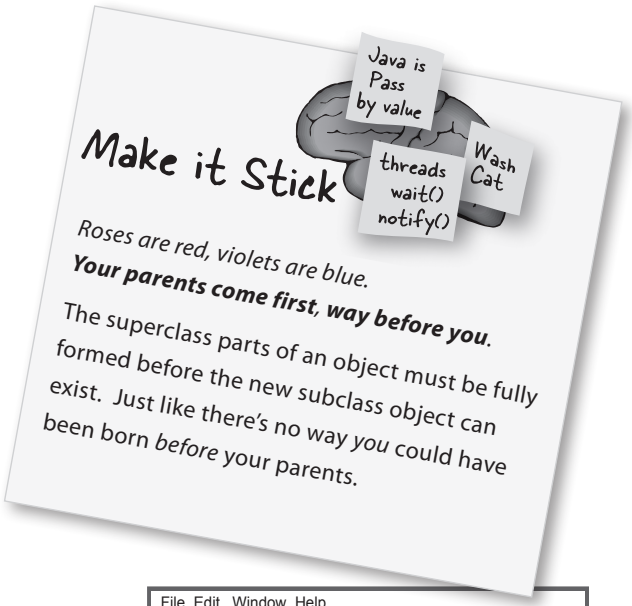
Sharpen your pencil

→ Yours to solve.

Some of the constructors in the SonOfBoo class will not compile. See if you can recognize which constructors are not legal. Match the compiler errors with the SonOfBoo constructors that caused them, by drawing a line from the compiler error to the "bad" constructor.

```
public class Boo {
  public Boo(int i) { }
  public Boo(String s) { }
  public Boo(String s, int i) { }
}
```

```
class SonOfBoo extends Boo {
  public SonOfBoo() {
    super("boo");
  }

  public SonOfBoo(int i) {
    super("Fred");
  }

  public SonOfBoo(String s) {
    super(42);
  }

  public SonOfBoo(int i, String s) {
  }

  public SonOfBoo(String a, String b, String c) {
    super(a, b);
  }

  public SonOfBoo(int i, int j) {
    super("man", j);
  }

  public SonOfBoo(int i, int x, int y) {
    super(i, "star");
  }
}
```




```
File Edit Window Help
%javac SonOfBoo.java
cannot resolve symbol
symbol : constructor Boo
(java.lang.String,java.
lang.String)
```

```
File Edit Window Help Yadayadayada
%javac SonOfBoo.java
cannot resolve symbol
symbol  : constructor Boo
(int,java.lang.String)
```

```
File Edit Window Help ImNotListening
%javac SonOfBoo.java
cannot resolve symbol
symbol:constructor Boo()
```

Now we know how an object is born, but how long does an object *live*?

An *object's* life depends entirely on the life of references referring to it. If the reference is considered "alive," the object is still alive on the Heap. If the reference dies (and we'll look at what that means in just a moment), the object will die.

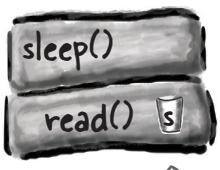
So if an object's life depends on the reference variable's life, how long does a *variable* live?

That depends on whether the variable is a *local* variable or an *instance* variable. The code below shows the life of a local variable. In the example, the variable is a primitive, but variable lifetime is the same whether it's a primitive or reference variable.

```
public class TestLifeOne {

  public void read() {
    int s = 42;
    sleep();
  }

  public void sleep() {
    s = 7;
  }
}
```

"s" is scoped to the read() method, so it can't be used anywhere else.

BAD!! Not legal to use "s" here!



sleep() can't see the "s" variable. Since it's not in sleep()'s own Stack frame, sleep() doesn't know anything about it.

The variable "s"" is alive, but in scope only within the read() method. When sleep() completes and read() is on top of the Stack and running again, read() can still see "s." When read() completes and is popped off the Stack, "s" is dead. Pushing up digital daisies.

1 A local variable lives only within the method that declared the variable.

```
public void read() {
  int s = 42;
  // 's' can be used only
  // within this method.
  // When this method ends,
  // 's' disappears completely.
}
```

Variable "**s**" can be used *only* within the *read()* method. In other words, ***the variable is in scope only within its own method***. No other code in the class (or any other class) can see "**s**."

2 An instance variable lives as long as the object does. If the object is still alive, so are its instance variables.

```
public class Life {
  int size;

  public void setSize(int s) {
    size = s;
    // 's' disappears at the
    // end of this method,
    // but 'size' can be used
    // anywhere in the class
  }
}
```

Variable '**s**' (this time a method parameter) is in scope only within the setSize() method. But instance variable size is scoped to the life of the *object* as opposed to the life of the *method*.

The difference between **life** and **scope** for local variables:

Life

A local variable is *alive* as long as its Stack frame is on the Stack. In other words, *until the method completes*.

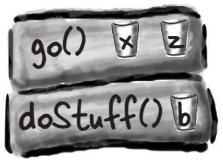
Scope

A local variable is in *scope* only within the method in which the variable was declared. When its own method calls another, the variable is alive, but not in scope until its method resumes. ***You can use a variable only when it is in scope***.

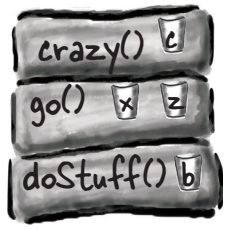
```
1 public void doStuff() {
    boolean b = true;
    go(4);
  }
2 public void go(int x) {
    int z = x + 24;
    crazy();
4   // imagine more code here
  }
  public void crazy() {
3   char c = 'a';
  }
```

Let's walk through what happens on the stack when something calls the doStuff() method.

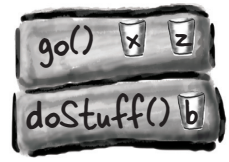
1 ***doStuff()*** **goes on the Stack. Variable "b" is alive and in scope.**



2 ***go()*** **plops on top of the Stack. "x" and "z" are alive and in scope, and "b" is alive but *not* in scope.**



3 ***crazy()*** **is pushed onto the Stack, with "c" now alive and in scope. The other three variables are alive but out of scope.**



4 ***crazy()*** **completes and is popped off the Stack, so 'c' is out of scope *and dead*. When go() resumes where it left off, "x" and "x" are both alive and back in scope. Variable "b" is still alive but out of scope (until go() completes).**

While a local variable is alive, its state persists. As long as method doStuff() is on the Stack, for example, the "b" variable keeps its value. But the "b" variable can be used only while doStuff()'s Stack frame is at the top. In other words, you can use a local variable *only* while that local variable's method is actually running (as opposed to waiting for higher Stack frames to complete).

What about reference variables?

The rules are the same for primitives and references. A reference variable can be used only when it's in scope, which means you can't use an object's remote control unless you've got a reference variable that's in scope. The *real* question is:

"How does *variable* life affect *object* life?"

An object is alive as long as there are live references to it. If a reference variable goes out of scope but is still alive, the object it *refers* to is still alive on the Heap. And then you have to ask..."What happens when the Stack frame holding the reference gets popped off the Stack at the end of the method?"

If that was the *only* live reference to the object, the object is now abandoned on the Heap. The reference variable disintegrated with the Stack frame, so the abandoned object is now, *officially*, toast. The trick is to know the point at which an object becomes ***eligible for garbage collection***.

Once an object is eligible for garbage collection (GC), you don't have to worry about reclaiming the memory that object was using. If your program gets low on memory, GC will destroy some or all of the eligible objects, to keep you from running out of RAM. You can still run out of memory, but *not* before all eligible objects have been hauled off to the dump. Your job is to make sure that you abandon objects (i.e., make them eligible for GC) when you're done with them, so that the garbage collector has something to reclaim. If you hang on to objects, GC can't help you, and you run the risk of your program dying a painful out-of-memory death.

An object's life has no value, no meaning, no point, unless somebody has a reference to it.

If you can't get to it, you can't ask it to do anything and it's just a big fat waste of bits.

But if an object is unreachable, the Garbage Collector will figure that out. Sooner or later, that object's goin' down.

An object becomes eligible for GC when its last live reference disappears.

Three ways to get rid of an object's reference:

1. The reference goes out of scope, permanently

```
void go() {
  Life z = new Life();
}
```

reference 'z' dies at end of method.

2. The reference is assigned another object

```
Life z = new Life();
z = new Life();
```

the first object is abandoned when z is 'reprogrammed' to a new object.

3. The reference is explicitly set to null

```
Life z = new Life();
z = null;
```

the first object is abandoned when z is 'deprogrammed.'

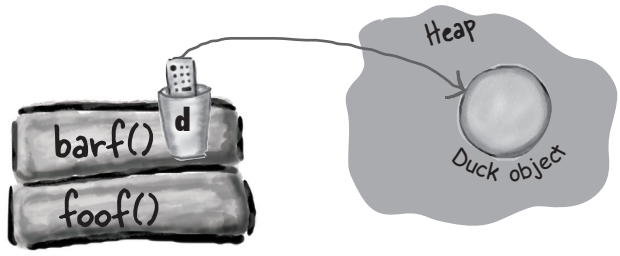
Object-killer #1

Reference goes out of scope, permanently.

```
public class StackRef  {
   public void foof() {
      barf();
   }

   public void barf() {
      Duck d = new Duck();
   }
}
```

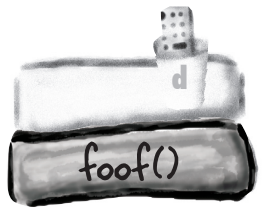
1. *foof()* is pushed onto the Stack; no variables are declared.

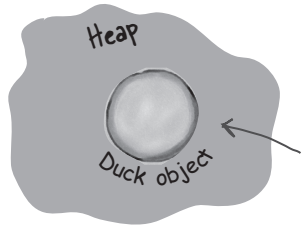
2. *barf()* is pushed onto the Stack, where it declares a reference variable, and creates a new object assigned to that reference. The object is created on the Heap, and the reference is alive and in scope.



The new Duck goes on the Heap, and as long as barf() is running, the 'd' reference is alive and in scope, so the Duck is considered alive.

3. *barf()* completes and pops off the Stack. Its frame disintegrates, so "d" is now dead and gone. Execution returns to *foof()*, but *foof()* can't use "d."



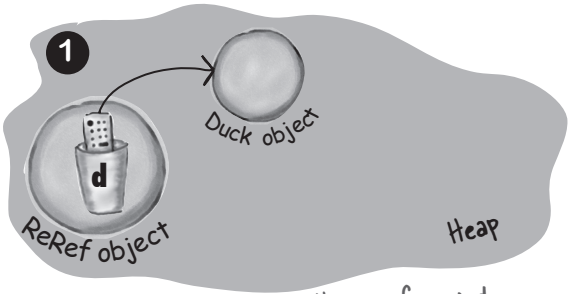


Uh–oh. The 'd' variable went away when the barf() Stack frame was blown off the stack, so the Duck is abandoned. Garbage–collector bait.

Object-killer #2

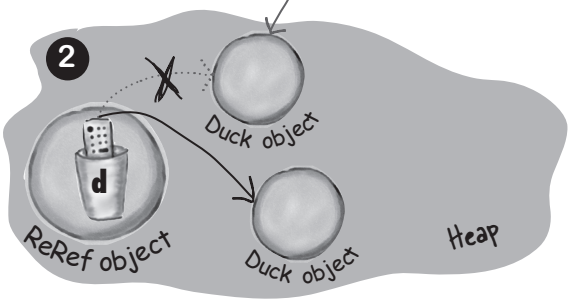
Assign the reference to another object

```
public class ReRef {
  Duck d = new Duck();

  public void go() {
    d = new Duck();
  }
}
```



The new Duck goes on the Heap, referenced by "d." Since "d" is an instance variable, the Duck will live as long as the ReRef object that instantiated it is alive. Unless...

When someone calls the go() method, this Duck is abandoned. Its only reference has been reprogrammed for a different Duck.



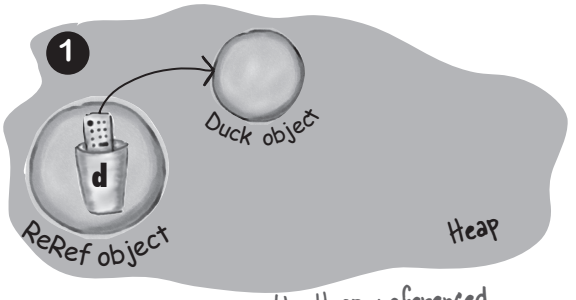
'd' is assigned a new Duck object, leaving the original (first) Duck object abandoned. That first Duck is now as good as dead.

Dude, all you had to do was reset the reference. Guess they didn't have memory management back then.

Object-killer #3

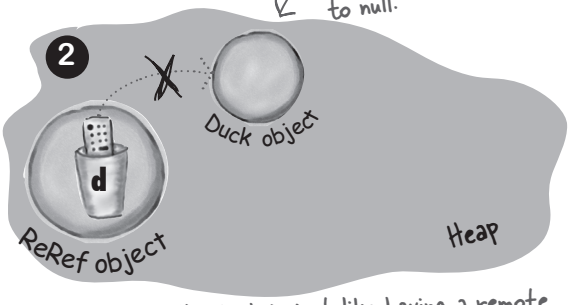
Explicitly set the reference to null

```
public class ReRef {
  Duck d = new Duck();

  public void go() {
    d = null;
  }
}
```



The new Duck goes on the Heap, referenced by "d." Since "d" is an instance variable, the Duck will live as long as the ReRef object that instantiated it is alive. Unless...

This Duck is abandoned. His only reference has been set to null.



'd' is set to null, which is just like having a remote control that isn't programmed to anything. You're not even allowed to use the dot operator on 'd' until it's reprogrammed (assigned an object).

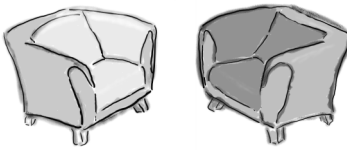
The meaning of `null`

When you set a reference to **`null`**, you're deprogramming the remote control. In other words, you've got a remote control, but no TV at the other end. A null reference has bits representing "null" (*we* don't know or care what those bits are, as long as the JVM knows).

If you have an unprogrammed remote control, in the real world, the buttons don't do anything when you press them. But in Java, you can't press the buttons (i.e., use the dot operator) on a null reference, because the JVM knows (this is a runtime issue, not a compiler error) that you're expecting a bark, but there's no Dog there to do it!

If you use the dot operator on a null reference, you'll get a NullPointerException at runtime. You'll learn all about exceptions in Chapter 13, *Risky Behavior*.

Fireside Chats



Tonight's Talk: **An instance variable and a local variable discuss life and death (with remarkable civility)**

Instance Variable

I'd like to go first, because I tend to be more important to a program than a local variable. I'm there to support an object, usually throughout the object's entire life. After all, what's an object without state? And what is state? Values kept in ***instance variables***.

Local Variable

I appreciate your point of view, and I certainly appreciate the value of object state and all, but I don't want folks to be misled. Local variables are *really* important. To use your phrase, "After all, what's an object without *behavior*?" And what is behavior? Algorithms in methods. And you can bet your bits there'll be some *local variables* in there to make those algorithms work.

No, don't get me wrong, I do understand your role in a method; it's just that your life is so short. So temporary. That's why they call you guys "temporary variables."

Within the local-variable community, the phrase "temporary variable" is considered derogatory. We prefer "local," "stack," "automatic," or "Scope-challenged."

My apologies. I understand completely.

Anyway, it's true that we don't have a long life, and it's not a particularly *good* life either. First, we're shoved into a Stack frame with all the other local variables. And then, if the method we're part of calls another method, another frame is pushed on top of us. And if *that* method calls *another* method...and so on. Sometimes we have to wait forever for all the other methods on top of the Stack to complete so that our method can run again.

I never really thought about it like that. What are you doing while the other methods are running and you're waiting for your frame to be the top of the Stack again?

Nothing. Nothing at all. It's like being in stasis—that thing they do to people in science-fiction movies when they have to travel long distances. Suspended animation, really. We just sit there on hold. As long as our frame is still there, we're safe and the value we hold is secure, but it's a mixed blessing when our frame gets to run again. On the one hand, we get to be active again. On the other

Instance Variable

We saw an educational video about it once. Looks like a pretty brutal ending. I mean, when that method hits its ending curly brace, the frame is literally *blown* off the Stack! Now *that's* gotta hurt.

I live on the Heap, with the objects. Well, not *with* the objects, actually *in* an object. The object whose state I store. I have to admit life can be pretty luxurious on the Heap. A lot of us feel guilty, especially around the holidays.

OK, hypothetically, yes, if I'm an instance variable of the Collar and the Collar gets GC'd, then the Collar's instance variables would indeed be tossed out like so many pizza boxes. But I was told that this almost never happens.

They let us *drink*?

Local Variable

hand, the clock starts ticking again on our short lives. The more time our method spends running, the closer we get to the end of the method. We *all* know what happens then.

Tell me about it. In computer science they use the term *popped* as in "the frame was popped off the Stack." That makes it sound fun, or maybe like an extreme sport. But, well, you saw the footage. So why don't we talk about you? I know what my little Stack frame looks like, but where do *you* live?

But you don't *always* live as long as the object who declared you, right? Say there's a Dog object with a Collar instance variable. Imagine *you're* an instance variable of the *Collar* object, maybe a reference to a Buckle or something, sitting there all happy inside the *Collar* object who's all happy inside the *Dog* object. But...what happens if the Dog wants a new Collar or *nulls* out its Collar instance variable? That makes the Collar object eligible for GC. So...if *you're* an instance variable inside the Collar and the whole *Collar* is abandoned, what happens to *you*?

And you believed it? That's what they say to keep us motivated and productive. But aren't you forgetting something else? What if you're an instance variable inside an object, and that object is referenced *only* by a *local* variable? If I'm the only reference to the object you're in, when I go, you're coming with me. Like it or not, our fates may be connected. So I say we forget about all this and go get a drink while we still can. Carpe RAM and all that.

BE the Garbage Collector

Which of the lines of code on the right, if added to the class on the left at point A, would cause exactly one additional object to be eligible for the Garbage Collector? (Assume that point A (//call more methods) will execute for a long time, giving the Garbage Collector time to do its stuff.)

Answers on page 272.

```
public class GC {
  public static GC doStuff() {
    GC newGC = new GC();
    doStuff2(newGC);
    return newGC;
  }

  public static void main(String[] args) {
    GC gc1;
    GC gc2 = new GC();
    GC gc3 = new GC();
    GC gc4 = gc3;
    gc1 = doStuff();

    A

    // call more methods
  }

  public static void doStuff2(GC copyGC) {
    GC localGC = copyGC;
  }
}
```

1 `copyGC = null;`

2 `gc2 = null;`

3 `newGC = gc3;`

4 `gc1 = null;`

5 `newGC = null;`

6 `gc4 = null;`

7 `gc3 = gc2;`

8 `gc1 = gc4;`

9 `gc3 = null;`

Popular Objects

In this code example, several new objects are created. Your challenge is to find the object that is "most popular," i.e., the one that has the most reference variables referring to it. Then list how many total references there are for that object, and what they are! We'll start by pointing out one of the new objects and its reference variable.

Good luck!

Answers on page 272.

```
class Bees {
  Honey[] beeHoney;
}

class Raccoon {
  Kit rk;
  Honey rh;
}

class Kit {
  Honey honey;
}

class Bear {
  Honey hunny;
}

public class Honey {
  public static void main(String[] args) {
    Honey honeyPot = new Honey();
    Honey[] ha = {honeyPot, honeyPot, honeyPot, honeyPot};
    Bees bees = new Bees();
    bees.beeHoney = ha;
    Bear[] bears = new Bear[5];
    for (int i = 0; i < 5; i++) {
      bears[i] = new Bear();
      bears[i].hunny = honeyPot;
    }
    Kit kit = new Kit();
    kit.honey = honeyPot;
    Raccoon raccoon = new Raccoon();
    raccoon.rh = honeyPot;
    raccoon.rk = kit;
    kit = null;
  } //  end of main
}
```

Here's a new Raccoon object!

Here's its reference variable 'raccoon.'

Five-Minute Mystery

"We've run the simulation four times, and the main module's temperature consistently drifts out of nominal toward cold," Sarah said, exasperated. "We installed the new temp-bots last week. The readings on the radiator bots, designed to cool the living quarters, seem to be within spec, so we've focused our analysis on the heat retention bots, the bots that help to warm the quarters." Tom sighed, at first it had seemed that nanotechnology was going to really put them ahead of schedule. Now, with only five weeks left until launch, some of the orbiter's key life support systems were still not passing the simulation gauntlet.

"What ratios are you simulating?" Tom asked.

"Well, if I see where you're going, we already thought of that," Sarah replied. "Mission control will not sign off on critical systems if we run them out of spec. We are required to run the v3 radiator bot's SimUnits in a 2:1 ratio with the v2 radiator's SimUnits," Sarah continued. "Overall, the ratio of retention bots to radiator bots is supposed to run 4:3."

"How's power consumption, Sarah?" Tom asked. Sarah paused, "Well, that's another thing, power consumption is running higher than anticipated. We've got a team tracking that down too, but because the nanos are wireless, it's been hard to isolate the power consumption of the radiators from the retention bots." "Overall power consumption ratios," Sarah continued, "are designed to run 3:2 with the radiators pulling more power from the wireless grid."

"OK, Sarah," Tom said. "Let's take a look at some of the simulation initiation code. We've got to find this problem, and find it quick!"

```
import java.util.ArrayList;

class V2Radiator {
  V2Radiator(ArrayList<SimUnit> list) {
    for (int x = 0; x < 5; x++) {
      list.add(new SimUnit("V2Radiator"));
    }
  }
}

class V3Radiator extends V2Radiator {
  V3Radiator(ArrayList<SimUnit> lglist) {
    super(lglist);
    for (int g = 0; g < 10; g++) {
      lglist.add(new SimUnit("V3Radiator"));
    }
  }
}

class RetentionBot {
  RetentionBot(ArrayList<SimUnit> rlist) {
    rlist.add(new SimUnit("Retention"));
  }
}
```

Five-Minute Mystery continued...

```
import java.util.ArrayList;

public class TestLifeSupportSim {
  public static void main(String[] args) {
    ArrayList<SimUnit> aList = new ArrayList<SimUnit>();
    V2Radiator v2 = new V2Radiator(aList);
    V3Radiator v3 = new V3Radiator(aList);
    for (int z = 0; z < 20; z++) {
      RetentionBot ret = new RetentionBot(aList);
    }
  }
}

class SimUnit {
  String botType;

  SimUnit(String type) {
    botType = type;
  }

  int powerUse() {
    if ("Retention".equals(botType)) {
      return 2;
    } else {
      return 4;
    }
  }
}
```

Tom gave the code a quick look, and a small smile crept across his lips. “I think I’ve found the problem, Sarah, and I bet I know by what percentage your power usage readings are off too!”

What did Tom suspect? How could he guess the power readings errors, and what few lines of code could you add to help debug this program?

Answers on page 273.

Exercise Solutions

Be the Garbage Collector

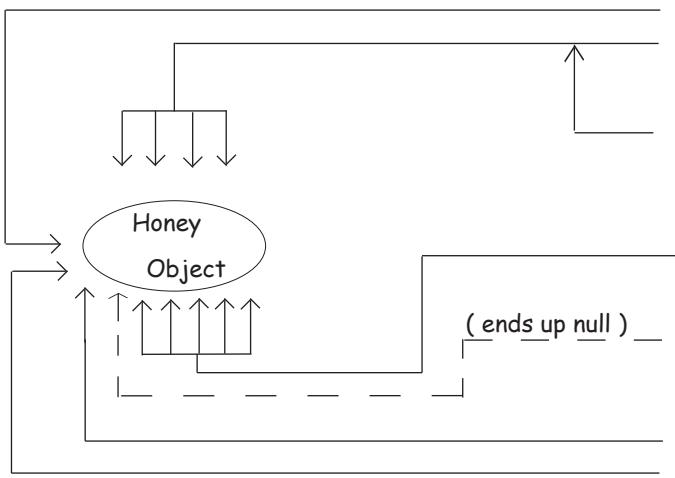
(from page 268)

1	`copyGC = null;`	No—this line attempts to access a variable that is out of scope.
2	`gc2 = null;`	OK—gc2 was the only reference variable referring to that object.
3	`newGC = gc3;`	No—another out of scope variable.
4	`gc1 = null;`	OK—gc1 had the only reference because newGC is out of scope.
5	`newGC = null;`	No—newGC is out of scope.
6	`gc4 = null;`	No—gc3 is still referring to that object.
7	`gc3 = gc2;`	No—gc4 is still referring to that object.
8	`gc1 = gc4;`	OK—Reassigning the only reference to that object.
9	`gc3 = null;`	No—gc4 is still referring to that object.

Popular Objects

(from page 269)

It probably wasn't too hard to figure out that the Honey object first referred to by the honeyPot variable is by far the most "popular" object in this class. But maybe it was a little trickier to see that all of the variables that point from the code to the Honey object refer to the ***same object***! There are a total of 12 active references to this object right before the main( ) method completes. The *kit.honeyPot* variable is valid for a while, but *kit* gets nulled at the end. Since *raccoon.rk* still refers to the Kit object, *raccoon.kit.honeyPot* (although never explicitly declared) refers to the object!



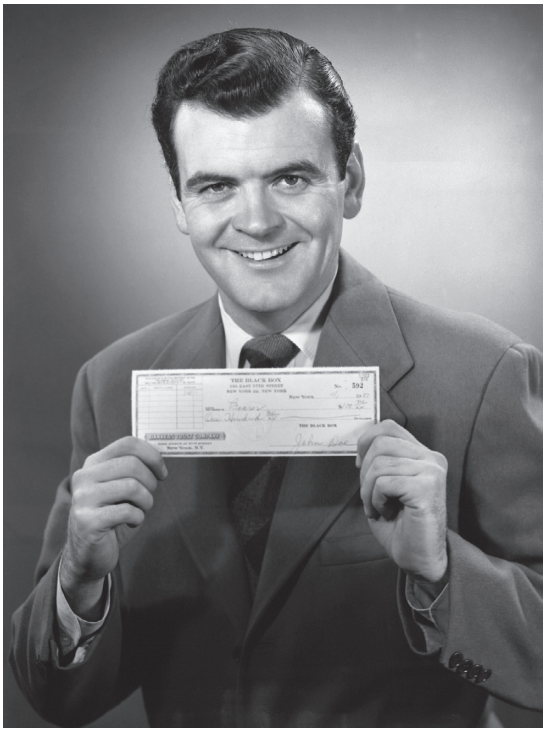

```
public class Honey {
  public static void main(String[] args) {
    Honey honeyPot = new Honey();
    Honey[] ha = {honeyPot, honeyPot,
                  honeyPot, honeyPot};
    Bees bees = new Bees();
    bees.beeHoney = ha;
    Bear[] bears = new Bear[5];
    for (int i = 0; i < 5; i++) {
      bears[i] = new Bear();
      bears[i].hunny = honeyPot;
    }
    Kit kit = new Kit();
    kit.honey = honeyPot;
    Raccoon raccoon = new Raccoon();

    raccoon.rh = honeyPot;
    raccoon.rk = kit;
    kit = null;
  } //  end of main
}
```

Five-Minute Mystery (from pages 270–271)

Tom noticed that the constructor for the V2Radiator class took an ArrayList. That meant that every time the *V3*Radiator constructor was called, it passed an ArrayList in its super() call to the *V2*Radiator constructor. That meant that an extra five V2Radiator SimUnits were created. If Tom was right, total power use would have been 120, not the 100 that Sarah's expected ratios predicted.

Since all the Bot classes create SimUnits, writing a constructor for the SimUnit class, which printed out a line every time a SimUnit was created, would have quickly highlighted the problem!

Numbers Matter



Do the Math. But there's more to working with numbers than just doing primitive arithmetic. You might want to get the absolute value of a number, or round a number, or find the larger of two numbers. You might want your numbers to print with exactly two decimal places, or you might want to put commas into your large numbers to make them easier to read. And what about parsing a String into a number? Or turning a number into a String? Someday you're gonna want to put a bunch of numbers into a collection like ArrayList that takes only objects. You're in luck. Java and the Java API are full of handy number-tweaking capabilities and methods, ready and easy to use. But most of them are **static**, so we'll start by learning what it means for a variable or method to be static, including constants in Java, also known as static *final* variables.

MATH methods: as close as you'll ever get to a *global* method

Except there's no global *anything* in Java. But think about this: what if you have a method whose behavior doesn't depend on an instance variable value. Take the round() method in the Math class, for example. It does the same thing every time—rounds a floating-point number (the argument to the method) to the nearest integer. Every time. If you had 10,000 instances of class Math, and ran the round(42.2) method, you'd get an integer value of 42. Every time. In other words, the method acts on the argument but is never affected by an instance variable state. The only value that changes the way the round() method runs is the argument passed to the method!

Doesn't it seem like a waste of perfectly good heap space to make an instance of class Math simply to run the round() method? And what about *other* Math methods like min(), which takes two numerical primitives and returns the smaller of the two? Or max(). Or abs(), which returns the absolute value of a number.

These methods never use instance variable values. In fact, the Math class doesn't *have* any instance variables. So there's nothing to be gained by making an instance of class Math. So guess what? You don't have to. As a matter of fact, you can't.

Methods in the Math class don't use any instance variable values. And because the methods are "static," you don't need to have an instance of Math. All you need is the Math class.

```
long x = Math.round(42.2);
int y = Math.min(56, 12);
int z = Math.abs(-343);
```

These methods never use instance variables, so their behavior doesn't need to know about a specific object.

If you try to make an instance of class Math:

```
Math mathObject = new Math();
```

You'll get this error:

```
File  Edit  Window  Help  IwasToldThereWouldBeNoMaths
%javac TestMath
TestMath.java:3: Math() has private
access in java.lang.Math
     Math mathObject = new Math();
                       ^
1 error
```

This error shows that the Math constructor is marked private! That means you can NEVER say 'new' on the Math class to make a new Math object.

The difference between regular (non-static) and static methods

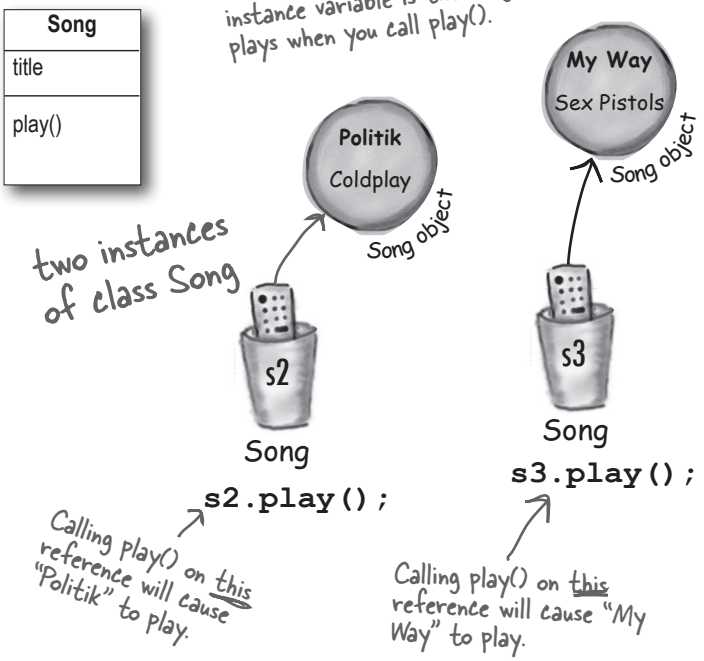
Java is object-oriented, but once in a while you have a special case, typically a utility method (like the Math methods), where there is no need to have an instance of the class. The keyword **`static`** lets a method run ***without any instance of the class***. A static method means "behavior not dependent on an instance variable, so no instance/object is required. Just the class."

regular (non-static) method

```
public class Song {
  String title;

  public Song(String t) {
    title = t;
  }

  public void play() {
    SoundPlayer player = new SoundPlayer();
    player.playSound(title);
  }
}
```

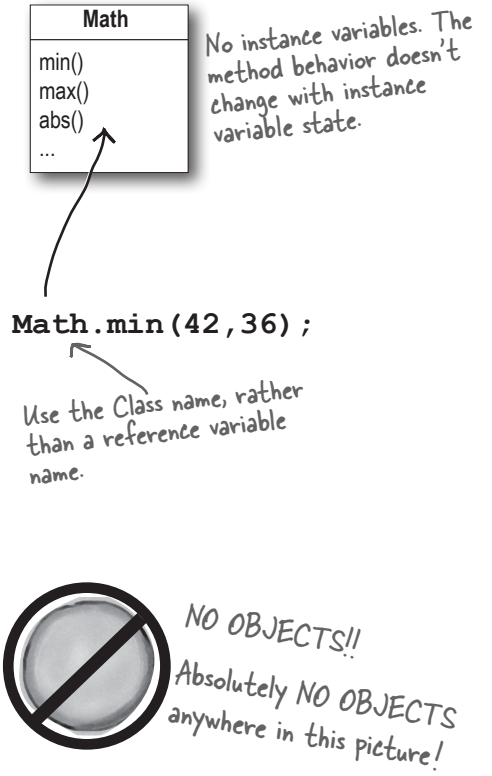
Instance variable value affects the behavior of the play() method.

The current value of the 'title' instance variable is the song that plays when you call play().

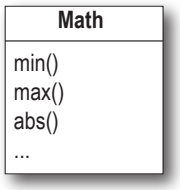


static method

```
public static int min(int a, int b) {
  //returns the smallest of a and b
}
```



Call a static method using a class name




```
Math.min(88,86);
```

Call a non-static method using a reference variable name


```
Song t2 = new Song();
t2.play();
```

What it means to have a class with static methods

Often (although not always), a class with static methods is not meant to be instantiated. In Chapter 8, *Serious Polymorphism*, we talked about abstract classes, and how marking a class with the **`abstract`** modifier makes it impossible for anyone to say "new" on that class type. In other words, ***it's impossible to instantiate an abstract class.***

But you can restrict other code from instantiating a *non*-abstract class by marking the constructor **`private`**. Remember, a *method* marked private means that only code from within the class can invoke the method. A *constructor* marked private means essentially the same thing—only code from within the class can invoke the constructor. Nobody can say "new" from *outside* the class. That's how it works with the Math class, for example. The constructor is private; you cannot make a new instance of Math. The compiler knows that your code doesn't have access to that private constructor.

This does *not* mean that a class with one or more static methods should never be instantiated. In fact, every class you put a main() method in is a class with a static method in it!

Typically, you make a main() method so that you can launch or test another class, nearly always by instantiating a class in main and then invoking a method on that new instance.

So you're free to combine static and non-static methods in a class, although even a single non-static method means there must be *some* way to make an instance of the class. The only ways to get a new object are through "new" or deserialization (or something called the Java Reflection API that we don't go into). No other way. But exactly *who* says new can be an interesting question, and one we'll look at a little later in this chapter.

Static methods can't use non-static (instance) variables!

Static methods run without knowing about any particular instance of the static method's class. And as you saw on the previous pages, there might not even *be* any instances of that class. Since a static method is called using the *class* (***Math***.random()) as opposed to an *instance reference* (***t2***.play()), a static method can't refer to any instance variables of the class. The static method doesn't know *which* instance's variable value to use.

If you try to use an instance variable from inside a static method, the compiler thinks, "I don't know *which* object's instance variable you're talking about!" If you have ten Duck objects on the heap, a static method doesn't know about *any* of them.

If you try to compile this code:

```
public class Duck {
  private int size;

  public static void main(String[] args) {
    System.out.println("Size of duck is " + size);
  }

  public void setSize(int s) {
    size = s;
  }

  public int getSize() {
    return size;
  }
}
```

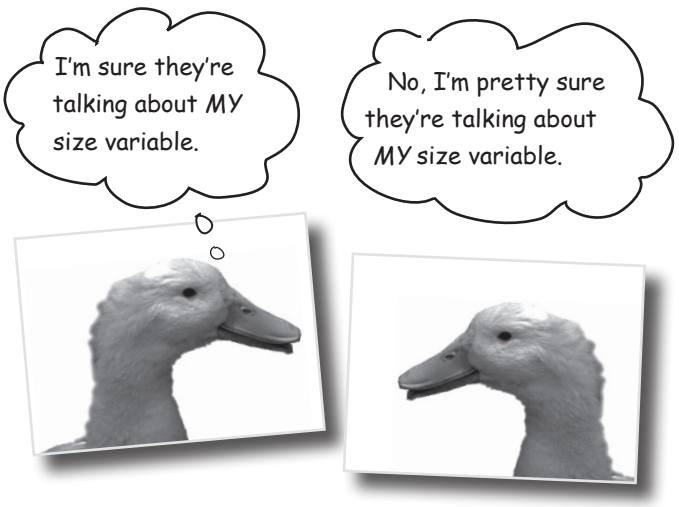
Which Duck? Whose size?

If there's a Duck on the heap somewhere, we don't know about it.

Static context. Everything else in the class is NOT static.

You'll get this error:

```
File Edit Window Help Quack
% javac Duck.java
Duck.java:6: non-static variable size cannot be referenced from a static context
      System.out.println("Size of duck is " + size);
                         ^
```



Static methods can't use non-static methods, either!

What do non-static methods do? ***They usually use instance variable state to affect the behavior of the method.*** A getName() method returns the value of the name variable. Whose name? The object used to invoke the getName() method.

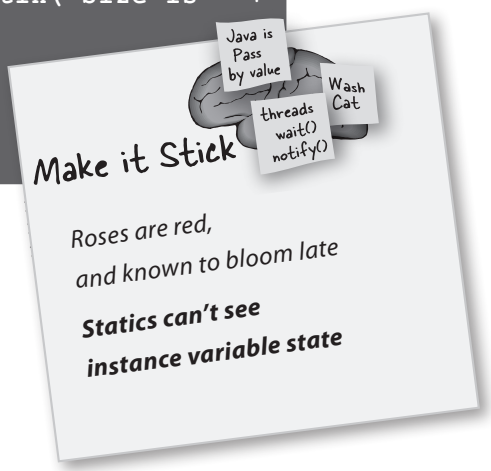
This won't compile:

Calling getSize() just postpones the inevitable—getSize() uses the size instance variable.

```
public class Duck {
  private int size;

  public static void main(String[] args) {
    System.out.println("Size is " + getSize());
  }

  public void setSize(int s) {
    size = s;
  }

  public int getSize() {
    return size;
  }
}
```

Back to the same problem... whose size?

```
File Edit Window Help Jack-in
% javac Duck.java
Duck.java:5: error: non-static
method getSize() cannot be refer-
enced from a static context
    System.out.println("Size is " +
getSize());
                       ^
1 error
```



there are no Dumb Questions

Q: What if you try to call a non-static method from a static method, but the non-static method doesn't use any instance variables. Will the compiler allow that?

A: No. The compiler knows that whether you do or do not use instance variables in a non-static method, you *can*. And think about the implications...if you were allowed to compile a scenario like that, then what happens if in the future you want to change the implementation of that non-static method so that one day it *does* use an instance variable? Or worse, what happens if a subclass *overrides* the method and uses an instance variable in the overriding version?

Q: I could swear I've seen code that calls a static method using a reference variable instead of the class name.

A: You *can* do that, but as your mother always told you, "Just because it's legal doesn't mean it's good." Although it *works* to call a static method using any instance of the class, it makes for misleading (less-readable) code. You *can* say,

```
Duck d = new Duck();
String[] s = {};
d.main(s);
```

This code is legal, but the compiler just resolves it back to the real class anyway ("OK, *d* is of type Duck, and main() is static, so I'll call the static main() in class Duck"). In other words, using *d* to invoke main() doesn't imply that main() will have any special knowledge of the object that *d* is referencing. It's just an alternate *way* to invoke a static method, but the method is still static!

Static variable:

value is the same for ALL instances of the class

Imagine you wanted to count how many Duck instances are being created while your program is running. How would you do it? Maybe an instance variable that you increment in the constructor?

```
class Duck {
  int duckCount = 0;
  public Duck() {
    duckCount++;
  }
}
```

this would always set duckCount to 1 each time a Duck was made

No, that wouldn't work because duckCount is an instance variable, and starts at 0 for each Duck. You could try calling a method in some other class, but that's kludgey. You need a class that's got only a single copy of the variable, and all instances share that one copy.

That's what a static variable gives you: a value shared by all instances of a class. In other words, one value per *class*, instead of one value per *instance*.

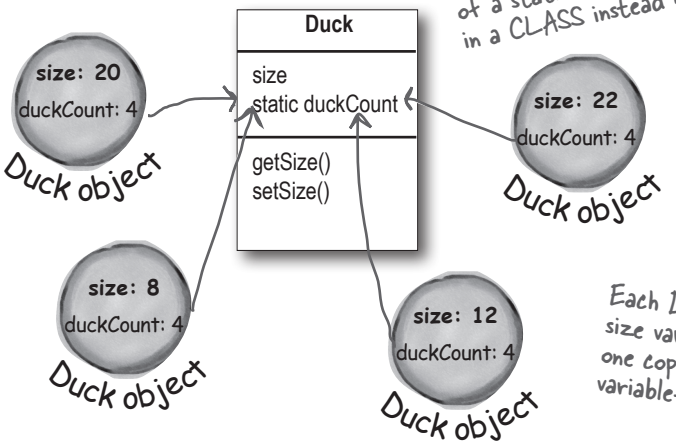
The static duckCount variable is initialized ONLY when the class is first loaded, NOT each time a new instance is made.

```
public class Duck {
  private int size;
  private static int duckCount = 0;

  public Duck() {
    duckCount++;
  }

  public void setSize(int s) {
    size = s;
  }

  public int getSize() {
    return size;
  }
}
```

Now it will keep incrementing each time the Duck constructor runs, because duckCount is static and won't be reset to 0.

A Duck object doesn't keep its own copy of duckCount.

Because duckCount is static, Duck objects all share a single copy of it. You can think of a static variable as a variable that lives in a CLASS instead of in an object.



Each Duck object has its own size variable, but there's only one copy of the duckCount variable—the one in the class.

Static variables are shared.

All instances of the same class share a single copy of the static variables.

instance variables: 1 per **instance**

static variables: 1 per **class**

Brain Barbell

Earlier in this chapter, we saw that a private constructor means that the class can't be instantiated from code running outside the class. In other words, only code from within the class can make a new instance of a class with a private constructor. (There's a kind of chicken-and-egg problem here.)

What if you want to write a class in such a way that only ONE instance of it can be created, and anyone who wants to use an instance of the class will always use that one, single instance?

Initializing a static variable

Static variables are initialized when a *class is loaded*. A class is loaded because the JVM decides it's time to load it. Typically, the JVM loads a class because somebody's trying to make a new instance of the class, for the first time, or use a static method or variable of the class. As a programmer, you also have the option of telling the JVM to load a class, but you're not likely to need to do that. In nearly all cases, you're better off letting the JVM decide when to *load* the class.

All static variables in a class are initialized *before* any object of that class can be created.

And there are two guarantees about static initialization:

- Static variables in a class are initialized before any *object* of that class can be created.
- Static variables in a class are initialized before any *static method* of the class runs.

```
class Player {
  static int playerCount = 0;
  private String name;
  public Player(String n) {
    name = n;
    playerCount++;
  }
}

public class PlayerTestDrive {
  public static void main(String[] args) {
    System.out.println(Player.playerCount);
    Player one = new Player("Tiger Woods");
    System.out.println(Player.playerCount);
  }
}
```

The playerCount is initialized when the class is loaded. We explicitly initialized it to 0, but we don't need to since 0 is the default value for ints. Static variables get default values just like instance variables.

Default values for declared but uninitialized static and instance variables are the same:

primitive integers (long, short, etc.): 0

primitive floating points (float, double): 0.0

boolean: false

object references: null

Access a static variable just like a static method—with the class name.

If you don't explicitly initialize a static variable (by assigning it a value at the time you declare it), it gets a default value, so int variables are initialized to zero, which means we didn't need to explicitly say playerCount = 0. Declaring, but not initializing, a static variable means the static variable will get the default value for that variable type, in exactly the same way that instance variables are given default values when declared.

```
File Edit Window Help What?
% java PlayerTestDrive
0
1
```

Before any instances are made

After an object is created

static final variables are constants

A variable marked **final** means that—once initialized—it can never change. In other words, the value of the static final variable will stay the same as long as the class is loaded. Look up Math.PI in the API, and you'll find:

```
public static final double PI = 3.141592653589793;
```

The variable is marked **public** so that any code can access it.

The variable is marked **static** so that you don't need an instance of class Math (which, remember, you're not allowed to create).

The variable is marked **final** because PI doesn't change (as far as Java is concerned).

There is no other way to designate a variable as a constant, but there is a naming convention that helps you to recognize one. ***Constant variable names are usually in all caps!***

A *static initializer* is a block of code that runs when a class is loaded, before any other code can use the class, so it's a great place to initialize a static final variable.

```
class ConstantInit1 {
    final static int X;
    static {
      X = 42;
    }
}
```

Initialize a *final* static variable:

1 At the time you declare it:

```
public class ConstantInit2 {
   public static final int X_VALUE = 25;
}
```

Notice the naming convention—static final variables are constants, so the name should be all uppercase, with an underscore separating the words.

OR

2 In a static initializer:

```
public class ConstantInit3 {
  public static final double VAL;

  static {
    VAL = Math.random();
  }
}
```

This code runs as soon as the class is loaded, before any static method is called and even before any static variable can be used.

If you don't give a value to a final variable in one of those two places:

```
public class ConstantInit3 {
  public static final double VAL;
}
```

no initialization!

The compiler will catch it:

```
File Edit Window Help Init?
% javac ConstantInit3.java
ConstantInit3.java:2: error: variable VAL not initialized in the default constructor
  public static final double VAL;
                             ^
1 error
```

final isn't just for static variables...

You can use the keyword **final** to modify non-static variables too, including instance variables, local variables, and even method parameters. In each case, it means the same thing: the value can't be changed. But you can also use final to stop someone from overriding a method or making a subclass.

non-static final variables

```
class Foof {
  final int size = 3;          ← now you can't change size
  final int whuffie;

  Foof() {
    whuffie = 42;              ← now you can't change whuffie
  }

  void doStuff(final int x) {
    // you can't change x
  }

  void doMore() {
    final int z = 7;
    // you can't change z
  }
}
```

final method

```
class Poof {
  final void calcWhuffie() {
    // important things
    // that must never be overridden
  }
}
```

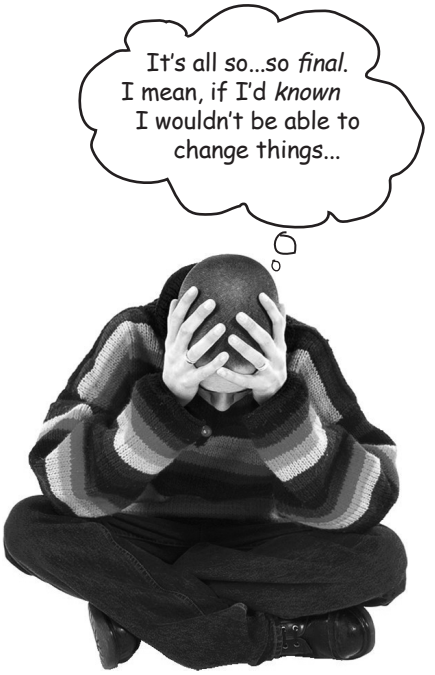
final class

```
final class MyMostPerfectClass {
  // cannot be extended
}
```

A final **variable** means you can't **change** its value.

A final **method** means you can't **override** the method.

A final **class** means you can't **extend** the class (i.e., you can't make a subclass).



there are no Dumb Questions

Q: A static method can't access a non-static variable. But can a non-static method access a static variable?

A: Of course. A non-static method in a class can always call a static method in the class or access a static variable of the class.

Q: Why would I want to make a class final? Doesn't that defeat the whole purpose of OO?

A: Yes and no. A typical reason for making a class final is for security. You can't, for example, make a subclass of the String class. Imagine the havoc if someone extended the String class and substituted their own String subclass objects, polymorphically, where String objects are expected. If you need to count on a particular implementation of the methods in a class, make the class final.

Q: Isn't it redundant to have to mark the methods final if the class is final?

A: If the class is final, you don't need to mark the methods final. Think about it—if a class is final, it can never be subclassed, so none of the methods can ever be overridden.

On the other hand, if you *do* want to allow others to extend your class and you want them to be able to override some, but not all, of the methods, then don't mark the class final, but go in and selectively mark specific methods as final. A final method means that a subclass can't override that particular method.

BULLET POINTS

- A ***static method*** should be called using the class name rather than an object reference variable: `Math.random()` versus `myFoo.go()`
- A static method can be invoked without any instances of the method's class on the heap.
- A static method is good for a utility method that does not (and will never) depend on a particular instance variable value.
- A static method is not associated with a particular instance—only the class—so it cannot access any instance variable values of its class. It wouldn't know *which* instance's values to use.
- A static method cannot access a non-static method, since non-static methods are usually associated with instance variable state.
- If you have a class with only static methods and you do not want the class to be instantiated, you can mark the constructor private.
- A ***static variable*** is a variable shared by all members of a given class. There is only one copy of a static variable in a class, rather than one copy *per each object* for instance variables.
- A static method can access a static variable.
- To make a constant in Java, mark a variable as both static and final.
- A final static variable must be assigned a value either at the time it is declared or in a static initializer:

```
static {
    DOG_CODE = 420;
}
```

- The naming convention for constants (final static variables) is to make the name all uppercase and use underscores (_) to separate the words.
- A final variable value cannot be changed once it has been assigned.
- Assigning a value to a final *instance* variable must be either at the time it is declared or in the constructor.
- A final method cannot be overridden.
- A final class cannot be extended (subclassed).

Sharpen your pencil

What's Legal?

Given everything you've just learned about static and final, which of these would compile?

1
```
public class Foo {
  static int x;

  public void go() {
    System.out.println(x);
  }
}
```

2
```
public class Foo2 {
  int x;

  public static void go() {
    System.out.println(x);
  }
}
```

3
```
public class Foo3 {
  final int x;

  public void go() {
    System.out.println(x);
  }
}
```

4
```
public class Foo4 {
  static final int x = 12;

  public void go() {
    System.out.println(x);
  }
}
```

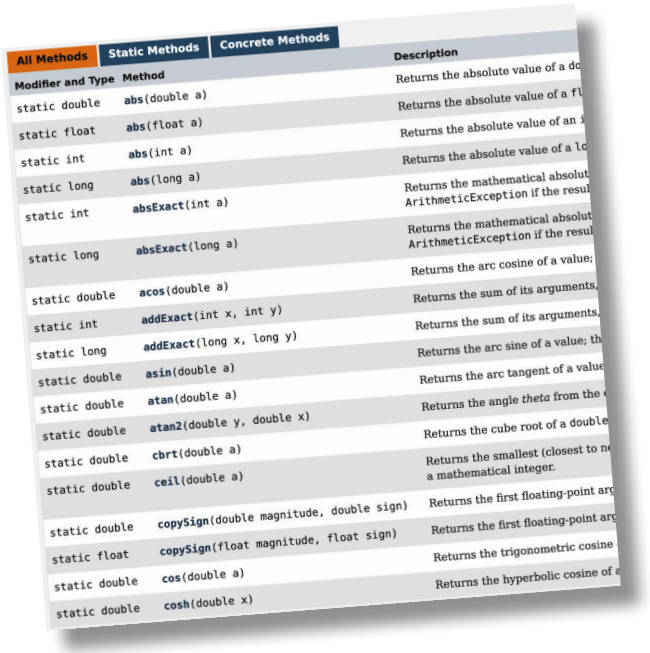
5
```
public class Foo5 {
  static final int x = 12;

  public void go(final int x) {
    System.out.println(x);
  }
}
```

6
```
public class Foo6 {
  int x = 12;

  public static void go(final int x) {
    System.out.println(x);
  }
}
```

Answers on page 308.

Math methods

Now that we know how static methods work, let's look at some static methods in class Math. This isn't all of them, just the highlights. Check your API for the rest including cos(), sin(), tan(), ceil(), floor(), and asin().



All Methods | Static Methods | Concrete Methods

Modifier and Type	Method	Description
static double	abs(double a)	Returns the absolute value of a do
static float	abs(float a)	Returns the absolute value of a fl
static int	abs(int a)	Returns the absolute value of an i
static long	abs(long a)	Returns the absolute value of a lo
static int	absExact(int a)	Returns the mathematical absolut ArithmeticException if the resul
static long	absExact(long a)	Returns the mathematical absolut ArithmeticException if the resul
static double	acos(double a)	Returns the arc cosine of a value;
static int	addExact(int x, int y)	Returns the sum of its arguments,
static long	addExact(long x, long y)	Returns the sum of its arguments,
static double	asin(double a)	Returns the arc sine of a value; th
static double	atan(double a)	Returns the arc tangent of a value
static double	atan2(double y, double x)	Returns the angle *theta* from the c
static double	cbrt(double a)	Returns the cube root of a double
static double	ceil(double a)	Returns the smallest (closest to ne a mathematical integer.
static double	copySign(double magnitude, double sign)	Returns the first floating-point arg
static float	copySign(float magnitude, float sign)	Returns the first floating-point arg
static double	cos(double a)	Returns the trigonometric cosine
static double	cosh(double x)	Returns the hyperbolic cosine of a

Math.abs()

Returns a double that is the absolute value of the argument. The method is overloaded, so if you pass it an int, it returns an int. Pass it a double, it returns a double.

```
int x = Math.abs(-240);        // returns 240
double d = Math.abs(240.45);   // returns 240.45
```

Math.random()

Returns a double between (and including) 0.0 through (but not including) 1.0.

```
double r1 = Math.random();
int r2 = (int) (Math.random() * 5);
```

We've been using this method so far, but there's also java.util.Random, which is a bit nicer to use.

Math.round()

Returns an int or a long (depending on whether the argument is a float or a double) rounded to the nearest integer value.

Remember, floating-point literals are assumed to be doubles unless you add the 'f.'

```
int x = Math.round(-24.8f);  // returns -25
int y = Math.round(24.45f);  // returns 24

long z = Math.round(24.45);  // returns 24L
```

This is a double.

Math.min()

Returns a value that is the minimum of the two arguments. The method is overloaded to take ints, longs, floats, or doubles.

```
int x = Math.min(24,240);                // returns 24
double y = Math.min(90876.5, 90876.49); // returns 90876.49
```

Math.max()

Returns a value that is the maximum of the two arguments. The method is overloaded to take ints, longs, floats, or doubles.

```
int x = Math.max(24,240);                // returns 240
double y = Math.max(90876.5, 90876.49); // returns 90876.5
```

Math.sqrt()

Returns the positive square root of the argument. The method takes a double, but of course you can pass in anything that fits in a double.

```
double x = Math.sqrt(9);    //return 3
double y = Math.sqrt(42.0); // returns 6.48074069840786
```

Wrapping a primitive

Sometimes you want to treat a primitive like an object. For example, collections like ArrayList only work with Objects:

```
ArrayList<???> list;
```

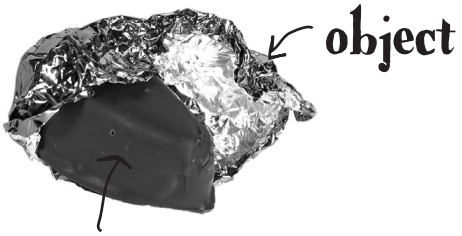
Can we create an ArrayList for ints?

There's a wrapper class for every primitive type, and since the wrapper classes are in the java.lang package, you don't need to import them. You can recognize wrapper classes because each one is named after the primitive type it wraps, but with the first letter capitalized to follow the class naming convention.

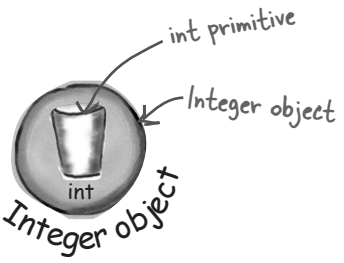
Oh yeah, for reasons absolutely nobody on the planet is certain of, the API designers decided not to map the names *exactly* from primitive type to class type. You'll see what we mean:

Boolean
Character
Byte
Short
Integer
Long
Float
Double

Watch out! The names aren't mapped exactly to the primitive types. The class names are fully spelled out.



When you need to treat a primitive like an object, wrap it.



Give the primitive to the wrapper constructor. That's it.

wrapping a value

```
int i = 288;
Integer iWrap = new Integer(i);
```

All the wrappers work like this. Boolean has a booleanValue(), Character has a charValue(), etc.

*un*wrapping a value

```
int unWrapped = iWrap.intValue();
```

Note: the picture at the top is a chocolate in a foil wrapper. Get it? Wrapper? Some people think it looks like a baked potato, but that works too.

This is stupid. You mean I can't just make an ArrayList of ints??? I have to wrap every single frickin' one in a new Integer object and then unwrap it when I try to access that value in the ArrayList? That's a waste of time and an error waiting to happen...

Java will Autobox primitives for you

In The Olden Days (pre–Java 5), we did have to do all this ourselves, manually wrapping and unwrapping primitives. Fortunately, now it's all done for us *automatically*.

Let's see what happens when we want to make an ArrayList to hold ints.

An ArrayList of primitive ints

```
public void autoboxing() {
  int x = 32;
  ArrayList<Integer> list = new ArrayList<Integer>();
  list.add(x);

  int num = list.get(0);
}
```

Make an ArrayList of type Integer.

Just add it!

Although there is NOT a method in ArrayList for add(int), the compiler does all the wrapping (boxing) for you. In other words, there really IS an Integer object stored in the ArrayList, but you get to "pretend" that the ArrayList takes ints. (You can add both ints and Integers to an ArrayList<Integer>.)

And the compiler automatically unwraps (unboxes) the Integer object so you can assign the int value directly to a primitive without having to call the intValue() method on the Integer object.

there are no Dumb Questions

Q: **Why not declare an ArrayList<int> if you want to hold ints?**

A: Because...*you can't.* At least, not in the versions of Java this book covers (the language is constantly evolving and things may change!). Remember, the rule for generic types is that you can specify only class or interface types, *not primitives*. So ArrayList<int> will not compile. But as you can see from the code above, it doesn't really matter, since the compiler lets you put ints into the ArrayList<Integer>. In fact, there's really no way to *prevent* you from putting primitives into an ArrayList where the type of the list is the type of that primitive's wrapper, since autoboxing will happen automatically. So, you can put boolean primitives in an ArrayList<Boolean> and chars into an ArrayList<Character>.

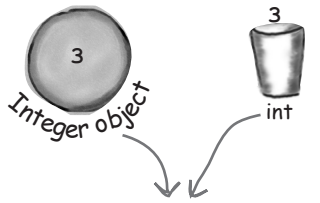
Autoboxing works almost everywhere

Autoboxing lets you do more than just the obvious wrapping and unwrapping to use primitives in a collection...it also lets you use either a primitive or its wrapper type virtually anywhere one or the other is expected. Think about that!

Fun with autoboxing

Method arguments

If a method takes a wrapper type, you can pass a reference to a wrapper or a primitive of the matching type. And of course the reverse is true—if a method takes a primitive, you can pass in either a compatible primitive or a reference to a wrapper of that primitive type.

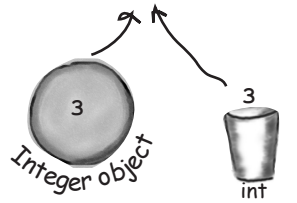



```
void takeNumber(Integer i) { }
```

Return values

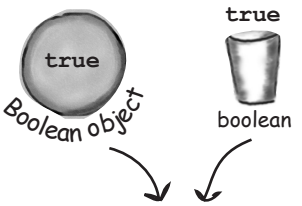
If a method declares a primitive return type, you can return either a compatible primitive or a reference to the wrapper of that primitive type. And if a method declares a wrapper return type, you can return either a reference to the wrapper type or a primitive of the matching type.

```
int giveNumber() {
  return x;
}
```



Boolean expressions

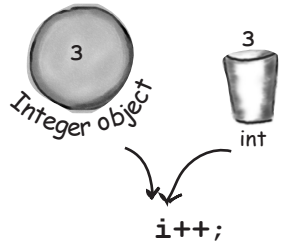
Any place a boolean value is expected, you can use either an expression that evaluates to a boolean (4 > 2), a primitive boolean, or a reference to a Boolean wrapper.




```
if (bool) {
   System.out.println("true");
}
```

Operations on numbers

This is probably the strangest one—yes, you can use a wrapper type as an operand in operations where the primitive type is expected. That means you can apply, say, the increment operator against a reference to an Integer object!



But don't worry—this is just a compiler trick. The language wasn't modified to make the operators work on objects; the compiler simply converts the object to its primitive type before the operation. It sure looks weird, though.

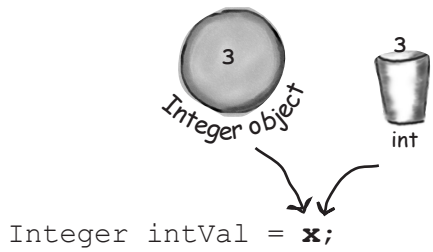
```
Integer i = new Integer(42);
i++;
```

And that means you can also do things like:

```
Integer j = new Integer(5);
Integer k = j + 3;
```

Assignments

You can assign either a wrapper or primitive to a variable declared as a matching wrapper or primitive. For example, a primitive int variable can be assigned to an Integer reference variable, and vice versa—a reference to an Integer object can be assigned to a variable declared as an int primitive.



Sharpen your pencil

Will this code compile? Will it run? If it runs, what will it do?

Take your time and think about this one; it brings up an implication of autoboxing that we didn't talk about.

You'll have to go to your compiler to find the answers. (Yes, we're forcing you to experiment, for your own good, of course.)

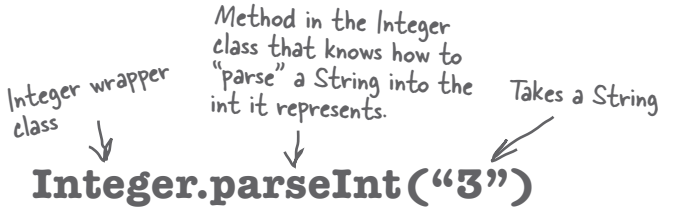
```
public class TestBox {
  private Integer i;
  private int j;

  public static void main(String[] args) {
    TestBox t = new TestBox();
    t.go();
  }

  public void go() {
    j = i;
    System.out.println(j);
    System.out.println(i);
  }
}
```

→ Yours to solve.

But wait! There's more! Wrappers have static utility methods too!

Besides acting like a normal class, the wrappers have a bunch of really useful static methods.

For example, the *parse* methods take a String and give you back a primitive value.



Converting a String to a primitive value is easy:

No problem to parse "2" into 2.

```
String s = "2";
int x = Integer.parseInt(s);
double d = Double.parseDouble("420.24");

boolean b = Boolean.parseBoolean("True");
```

The parseBoolean() method ignores the cases of the characters in the String argument.

But if you try to do this:

```
String t = "two";
int y = Integer.parseInt(t);
```

Uh-oh. This compiles just fine, but at runtime it blows up. Anything that can't be parsed as a number will cause a NumberFormatException.

You'll get a runtime exception:

```
File Edit Window Help Clue
% java Wrappers
Exception in thread "main" java.lang.NumberFormatException:
For input string: "two"
   at java.base/java.lang.NumberFormatException.forInputStri
ng(NumberFormatException.java:65)
   at java.base/java.lang.Integer.parseInt(Integer.java:652)
   at java.base/java.lang.Integer.parseInt(Integer.java:770)
   at Snippets.badParse(Snippets.java:48)
   at Snippets.main(Snippets.java:54)
```

Every method or constructor that parses a String can throw a NumberFormatException. It's a runtime exception, so you don't have to handle or declare it. But you might want to.

(We'll talk about exceptions in Chapter 13, *Risky Behavior*.)

And now in reverse...turning a primitive number into a String

You may want to turn a number into a String, for example when you want to show this number to a user or put it into a message. There are several ways to turn a number into a String. The easiest is to simply concatenate the number to an existing String.

```
double d = 42.5;
String doubleString = "" + d;
```

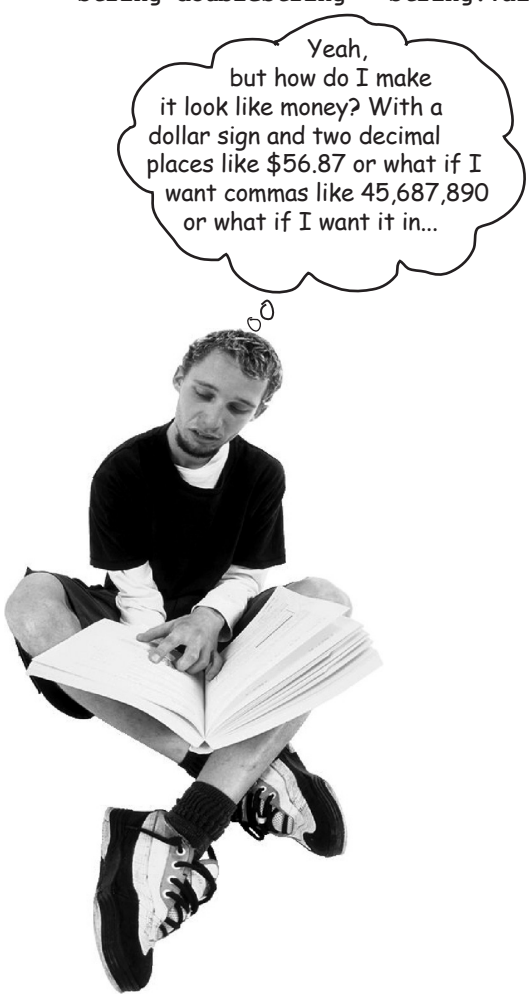
Remember the '+' operator is overloaded in Java (the only overloaded operator) as a String concatenator. Anything added to a String becomes Stringified.

```
double d = 42.5;
String doubleString = Double.toString(d);
```

Another way to do it using a static method in class Double.

```
double d = 42.5;
String doubleString = String.valueOf(d);
```

There's also an overloaded a static method "valueOf" on String that will get the String value of pretty much anything.



Number formatting

In Java, formatting numbers and dates doesn't have to be coupled with I/O. Think about it. One of the most typical ways to display numbers to a user is through a GUI. You put Strings into a scrolling text area, or maybe a table. If formatting was built only into print statements, you'd never be able to format a number into a nice String to display in a GUI.

The Java API provides powerful and flexible formatting using the Formatter class in java.util. But often you don't need to create and call methods on the Formatter class yourself, because the Java API has convenience methods in some of the I/O classes (including printf()) and the String class. So it can be a simple matter of calling a static String.format() method and passing it the thing you want formatted along with formatting instructions.

Of course, you do have to know how to supply the formatting instructions, and that takes a little effort unless you're familiar with the ***printf()*** function in C/C++. Fortunately, even if you *don't* know printf(), you can simply follow recipes for the most basic things (that we're showing in this chapter). But you *will* want to learn how to format if you want to mix and match to get *anything* you want.

We'll start here with a basic example and then look at how it works. (Note: we'll revisit formatting again in Chapter 16, *Saving Objects*.)

Making big numbers more readable with underscores, a quick detour

Before we get into formatting numbers, let's take a small, useful detour. Sometimes you'll want to declare variables with large initial values. Let's look at three declarations that assign the same large value, a billion, to long primitives:

```
long hardToRead = 1000000000;
long easierToRead = 1_000_000_000;
long legalButSilly = 10_0000_0000;
```

When you're assigning large values, properly located underscores will make your life easier!

Formatting a number to use commas

```
public class TestFormats {
  public static void main(String[] args) {
    long myBillion = 1_000_000_000;
    String s = String.format("%,d", myBillion);
    System.out.println(s);
  }
}
```

The number to format (we want it to have commas).

The formatting instructions for how to format the second argument (which in this case is an int value). Remember, there are only two arguments to this method here—the first comma is *INSIDE* the String literal, so it isn't separating arguments to the format method.

```
1,000,000,000
```

Now we get commas inserted into the number.

Formatting deconstructed...

At the most basic level, formatting consists of two main parts (there is more, but we'll start with this to keep it cleaner):

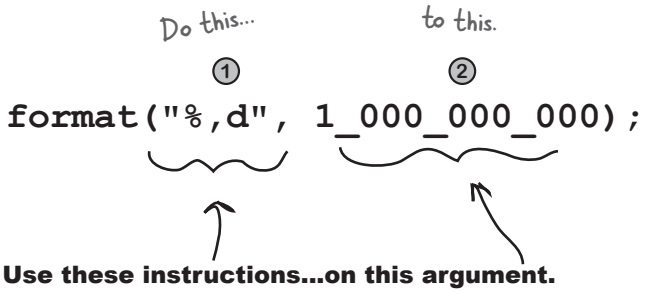
1. **Formatting instructions**

   You use special format specifiers that describe how the argument should be formatted.

2. **The argument to be formatted.**

   Although there can be more than one argument, we'll start with just one. The argument type can't be just *anything*...it has to be something that can be formatted using the format specifiers in the formatting instructions. For example, if your formatting instructions specify a *floating-point number*, you can't pass in a Dog or even a String that looks like a floating-point number.

Note: if you already know printf() from c/C++, you can probably just skim the next few pages. Otherwise, read carefully!



What do these instructions actually say?

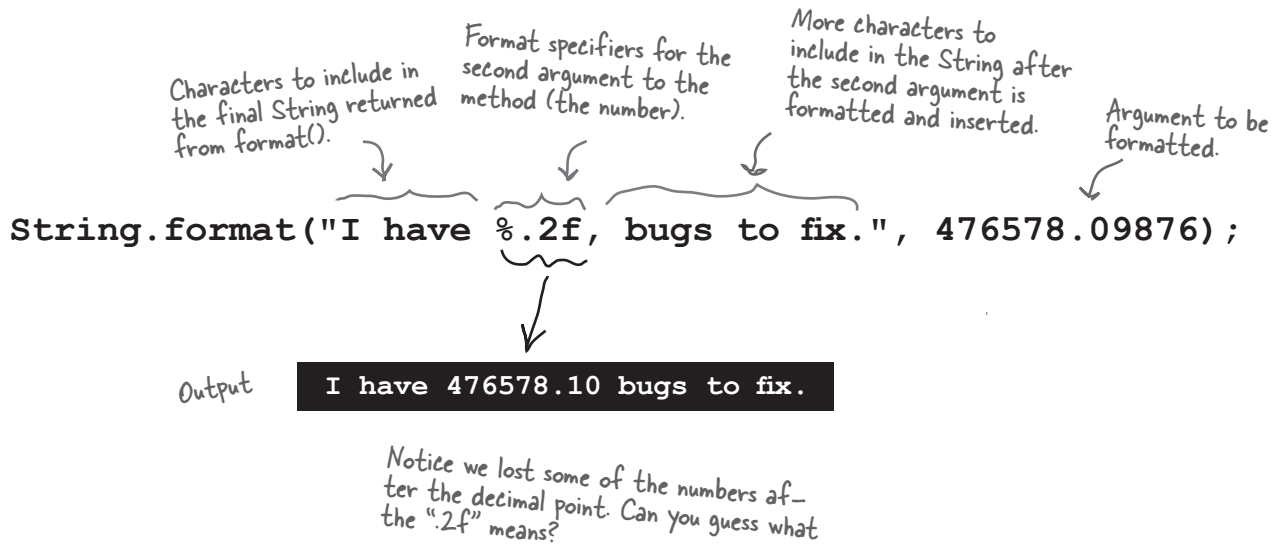
"Take the second argument to this method, and format it as a **d**ecimal integer and insert **commas**."

How do they say that?

On the next page we'll look in more detail at what the syntax "%,d" actually means, but for starters, any time you see the percent sign (%) in a format String (which is always the first argument to a format() method), think of it as representing a variable, and the variable is the other argument to the method. The rest of the characters after the percent sign describe the formatting instructions for the argument.

The percent (%) says, "insert argument here"
(and format it using these instructions)

The first argument to a format() method is called the format String, and it can actually include characters that you just want printed as-is, without extra formatting. When you see the % sign, though, think of the percent sign as a variable that represents the other argument to the method.



The "%" sign tells the formatter to insert the other method argument (the second argument to format(), the number) here, AND format it using the ".2f" characters after the percent sign. Then the rest of the format String, "bugs to fix," is added to the final output.

Adding a comma

```
String.format("I have %,.2f bugs to fix.", 476578.09876);
```

```
I have 476,578.10 bugs to fix.
```

By changing the format instructions from "%.2f" to %,.2f", we got a comma in the formatted number.

The format String uses its own little language syntax

You obviously can't put just *anything* after the "%" sign. The syntax for what goes after the percent sign follows very specific rules, and describes how to format the argument that gets inserted at that point in the result (formatted) String.

You've already seen some examples:

%,d means "insert commas and format the number as a decimal integer."

and

%.2f means "format the number as a floating point with a precision of two decimal places."

and

%,.2f means "insert commas and format the number as a floating point with a precision of two decimal places."

Really the question is: "How do I know what to put after the percent sign to get it to do what I want?" And that includes knowing the symbols (like "d" for decimal and "f" for floating point) as well as the order in which the instructions must be placed following the percent sign. For example, if you put the comma after the "d" like "%d," instead of "%,d" it won't work!

Or will it? What do you think this will do:

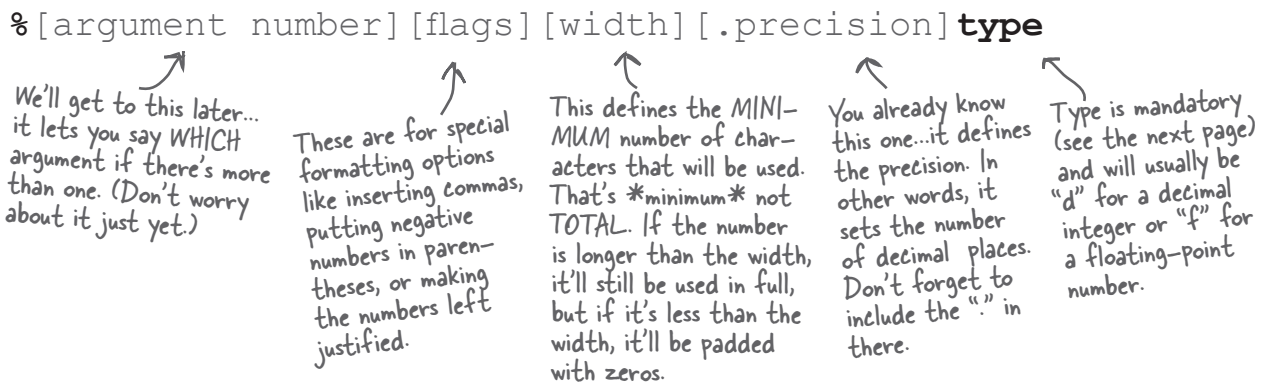
```
String.format("I have %.2f, bugs to fix.", 476578.09876);
```

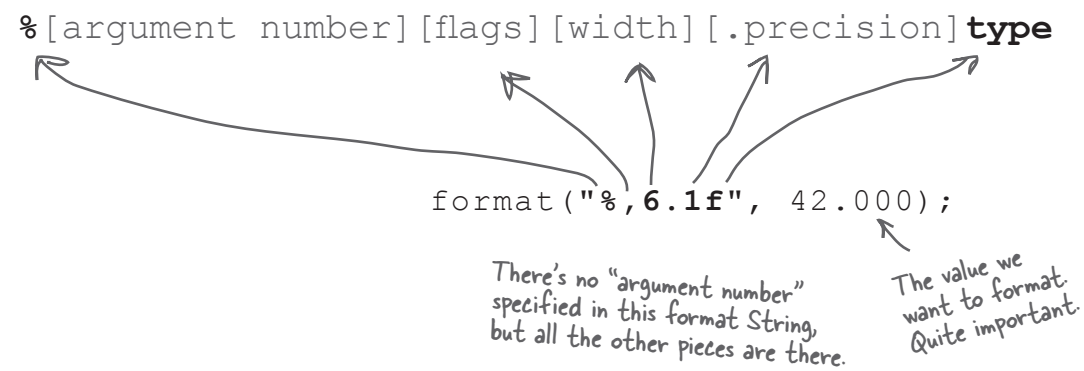
(We'll answer that on the next page.)

The format specifier

Everything after the percent sign up to and including the type indicator (like "d" or "f") is part of the formatting instructions. After the type indicator, the formatter assumes the next set of characters is meant to be part of the output String, until or unless it hits another percent (%) sign. Hmmmm...is that even possible? Can you have more than one formatted argument variable? Put that thought on hold for right now; we'll come back to it in a few minutes. For now, let's look at the syntax for the format specifiers—the things that go after the percent (%) sign and describe how the argument should be formatted.

A format specifier can have up to five different parts (not including the "%"). Everything in brackets [ ] below is optional, so only the percent (%) and the type are required. But the order is also mandatory, so any parts you DO use must go in this order.





The only required specifier is for TYPE

Although type is the only required specifier, remember that if you *do* put in anything else, type must always come last! There are more than a dozen different type modifiers (not including dates and times; they have their own set), but most of the time you'll probably use %d (decimal) or %f (floating point). And typically you'll combine %f with a precision indicator to set the number of decimal places you want in your output.

The TYPE is mandatory, everything else is optional.

%d ***decimal***

```
format("%d", 42);
```

A 42.25 would not work! It would be the same as trying to directly assign a double to an int variable.

```
42
```

The argument must be compatible with an int, so that means only byte, short, int, and char (or their wrapper types).

%f ***floating point***

```
format("%.3f", 42.000000)
```

Here we combined the "f" with a precision indicator ".3" so we ended up with three zeros.

```
42.000
```

The argument must be of a floating-point type, so that means only a float or double (primitive or wrapper) as well as something called BigDecimal (which we don't look at in this book).

%x ***hexadecimal***

```
format("%x", 42)
```

```
2a
```

The argument must be a byte, short, int, long (including both primitive and wrapper types), and BigInteger.

%c ***character***

```
format("%c", 42)
```

The number 42 represents the char "*".

```
*
```

The argument must be a byte, short, char, or int (including both primitive and wrapper types).

You must include a type in your format instructions, and if you specify things besides type, the type must always come last. Most of the time, you'll probably format numbers using either "d" for decimal or "f" for floating point.

What happens if I have more than one argument?

Imagine you want a String that looks like this:

"The rank is ***20,456,654*** out of ***100,567,890.24***."

But the numbers are coming from variables. What do you do? You simply add *two* arguments after the format String (first argument), so that means your call to format() will have three arguments instead of two. And inside that first argument (the format String), you'll have two different format specifiers (two things that start with "%"). The first format specifier will insert the second argument to the method, and the second format specifier will insert the third argument to the method. In other words, the variable insertions in the format String use the order in which the other arguments are passed into the format() method.

```
int one = 20456654;
double two = 100567890.248907;
String s = String.format("The rank is %,d out of %,.2f", one, two);
```

```
The rank is 20,456,654 out of 100,567,890.25
```

We added commas to both variables and restricted the floating-point number (the second variable) to two decimal places.

When you have more than one argument, they're inserted using the order in which you pass them to the format() method.

As you'll see when we get to date formatting, you might actually want to apply different formatting specifiers to the same argument. That's probably hard to imagine until you see how *date* formatting (as opposed to the *number* formatting we've been doing) works. Just know that in a minute, you'll see how to be more specific about which format specifiers are applied to which arguments.

there are no Dumb Questions

Q: Um, there's something REALLY strange going on here. Just how many arguments *can* I pass? I mean, how many overloaded format() methods are IN the String class? So, what happens if I want to pass, say, ten different arguments to be formatted for a single output String?

A: Good catch. Yes, there *is* something strange going on, and no there are *not* a bunch of overloaded format() methods to take a different number of possible arguments. In order to support this formatting (printf-like) API in Java, the language needed another feature—*variable argument lists* (called *varargs* for short). We'll talk about varargs more in Appendix B.

Just one more thing...static imports

Static imports are a real mixed blessing. Some people love this idea, some people hate it. Static imports exist to make your code a little shorter. If you hate to type or hate long lines of code, you might just like this feature. The downside to static imports is that—if you're not careful—using them can make your code a lot harder to read.

The basic idea is that whenever you're using a static class, a static variable, or an enum (more on those later), you can import them and save yourself some typing.

Use carefully:

Static imports can make your code confusing to read.

Always re-read your code after using a static import and think: "Will I understand this in six months time?"

Without static imports:

```
class NoStatic {

  public static void main(String[] args) {

    System.out.println("sqrt " + Math.sqrt(2.0));

    System.out.println("tan " + Math.tan(60));

  }

}
```

Same code, with static imports:

The syntax to use when declaring static imports.

```
import static java.lang.Math.*;

import static java.lang.System.out;

class WithStatic {

  public static void main(String[] args) {

    out.println("sqrt " + sqrt(2.0));

    out.println("tan " + tan(60));

  }

}
```

Static imports in action.

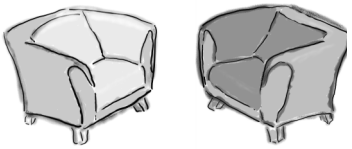
This might be a BAD place to use a static import. Removing the "System" class makes it unclear what this is and where it came from. Also it may lead to naming conflicts; you can't create any other variables called "out" now.

You might want to use static imports for these methods. It makes the code shorter, and you don't need the "Math." prefix to understand what these operations are.

Caveats & Gotchas

- Using a static import removes the information about which class the static came from. We'd advise using static imports only when the static method or variable still means something when it's not prefixed with the class name.
- A big issue with static imports is that it's easy to create naming conflicts. For example, if you have two different classes with an "add()" method, how will you and the compiler know which one to use? So it's best not to use a static import if it's possible to create a conflict.
- Notice that you can use wildcards (.*), in your static import declaration.

Fireside Chats



Tonight's Talk: **An instance variable takes cheap shots at a static variable**

Instance Variable

I don't even know why we're doing this. Everyone knows static variables are just used for constants. And how many of those are there? I think the whole API must have, what, four? And it's not like anybody ever uses them.

Static Variable

You really should check your facts. When was the last time you looked at the API? It's frickin' loaded with statics! It even has entire classes dedicated to holding constant values. There's a class called SwingConstants, for example, that's just full of them.

Full of it. Yeah, you can say that again. OK, so there are a few in the Swing library, but everybody knows Swing is just a special case.

It might be a special case, but it's a really important one! And what about the Color class? What a pain if you had to remember the RGB values to make the standard colors! But the color class already has constants defined for blue, purple, white, red, etc. Very handy.

Ok, but besides a few GUI things, give me an example of just one static variable that anyone would actually use. In the real world.

How's System.out for starters? The out in System.out is a static variable of the System class. You personally don't make a new instance of the System; you just ask the System class for its out variable.

Well, that's another special case. And nobody uses that except for debugging anyway.

Oh, like debugging isn't important?

And here's something that probably never crossed your narrow mind—let's face it, static variables are more efficient. One per class instead of one per instance. The memory savings might be huge!

Instance Variable

Um, aren't you forgetting something?

Static variables are about as un-OO as it gets!! Gee, why not just go take a giant backward step and do some procedural programming while we're at it.

You're like a global variable, and any programmer worth their sticker-covered laptop knows that's usually a Bad Thing.

Yeah, you live in a class, but they don't call it *Class*-Oriented programming. That's just stupid. You're a relic. Something to help the old-timers make the leap to Java.

Well, OK, every once in a while sure, it makes sense to use a static, but let me tell you, abuse of static variables (and methods) is the mark of an immature OO programmer. A designer should be thinking about *object* state, not *class* state.

Static methods are the worst things of all, because it usually means the programmer is thinking procedurally instead of about objects doing things based on their unique object state.

Riiiiiight. Whatever you need to tell yourself.

Static Variable

What?

What do you mean *un*-OO?

I am NOT a global variable. There's no such thing. I live in a class! That's pretty OO you know, a CLASS. I'm not just sitting out there in space somewhere; I'm a natural part of the state of an object; the only difference is that I'm shared by all instances of a class. Very efficient.

Alright just stop right there. THAT is definitely not true. Some static variables are absolutely crucial to a system. And even the ones that aren't crucial sure are handy.

Why do you say that? And what's wrong with static methods?

Sure, I know that objects should be the focus of an OO design, but just because there are some clueless programmers out there...don't throw the baby out with the bytecode. There's a time and place for statics, and when you need one, nothing else beats it.

BE the Compiler

The Java file on this page represents a complete program. Your job is to play compiler and determine whether this file will compile. If it won't compile, how would you fix it? When it runs, what would be its output?

```
class StaticSuper {
  static {
    System.out.println("super static block");
  }

  StaticSuper () {
    System.out.println("super constructor");
  }
}

public class StaticTests extends StaticSuper {
  static int rand;

  static {
    rand = (int) (Math.random() * 6);
    System.out.println("static block " + rand);
  }

  StaticTests() {
    System.out.println("constructor");
  }

  public static void main(String[] args) {
    System.out.println("in main");
    StaticTests st = new StaticTests();
  }
}
```

Which of these is the output?

Possible Output

```
File  Edit  Window  Help  Cling
%java StaticTests
static block 4
in main
super static block
super constructor
constructor
```

Possible Output

```
File  Edit  Window  Help  Electricity
%java StaticTests
super static block
static block 3
in main
super constructor
constructor
```

→ Answers on page 308.

This chapter explored the wonderful, static world of Java. Your job is to decide whether each of the following statements is true or false.

True or False

1. To use the Math class, the first step is to make an instance of it.

2. You can mark a constructor with the **`static`** keyword.

3. Static methods don't have access to instance variable state of the "this" object.

4. It is good practice to call a static method using a reference variable.

5. Static variables could be used to count the instances of a class.

6. Constructors are called before static variables are initialized.

7. MAX_SIZE would be a good name for a static final variable.

8. A static initializer block runs before a class's constructor runs.

9. If a class is marked final, all of its methods must be marked final.

10. A final method can be overridden only if its class is extended.

11. There is no wrapper class for boolean primitives.

12. A wrapper is used when you want to treat a primitive like an object.

13. The parseXxx methods always return a String.

14. Formatting classes (which are decoupled from I/O) are in the java.format package.

Answers on page 308.

Exercise Solutions

Sharpen your pencil (from page 287)

1, 4, 5, and 6 are legal.

2 doesn't compile because the static method references a non-static instance variable.

3 doesn't compile because the instance variable is final but hasn't been initialized.

BE the Compiler (from page 306)

```
StaticSuper () {
  System.out.println(
    "super constructor");
}
```

StaticSuper is a constructor and must have ( ) in its signature. Notice that as the output below demonstrates, the static blocks for both classes run before either of the constructors run.

Note that this will be a randomly generated number from 0 to 5 inclusive.

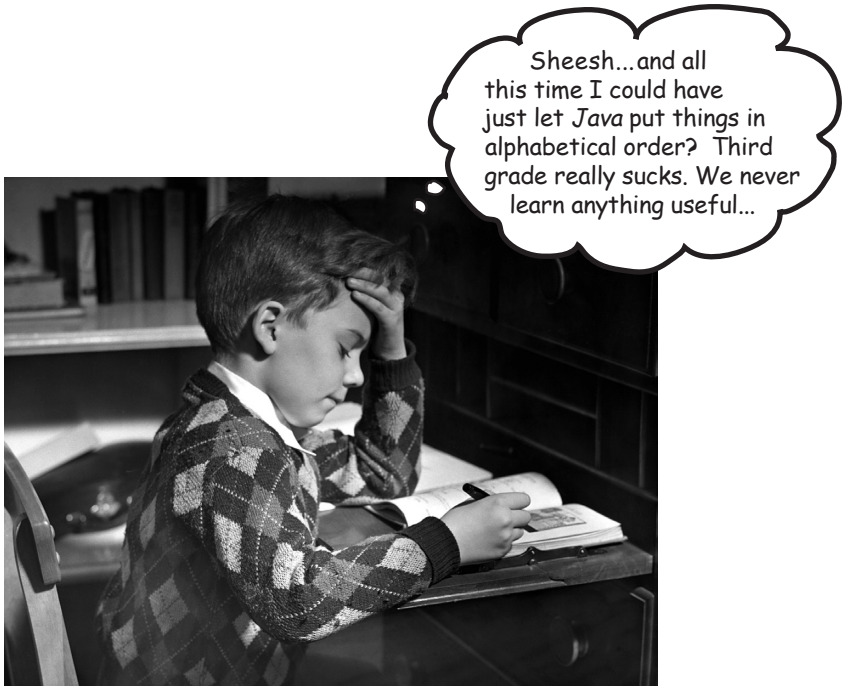
Output

```
File  Edit  Window  Help  Cling
%java StaticTests
super static block
static block 3
in main
super constructor
constructor
```

True or False (from page 307)

1. To use the Math class, the first step is to make an instance of it. **False**
2. You can mark a constructor with the keyword "static." **False**
3. Static methods don't have access to an object's instance variables. **True**
4. It is good practice to call a static method using a reference variable. **False**
5. Static variables could be used to count the instances of a class. **True**
6. Constructors are called before static variables are initialized. **False**
7. MAX_SIZE would be a good name for a static final variable. **True**
8. A static initializer block runs before a class's constructor runs. **True**
9. If a class is marked final, all of its methods must be marked final. **False**
10. A final method can be overridden only if its class is extended. **False**
11. There is no wrapper class for boolean primitives. **False**
12. A wrapper is used when you want to treat a primitive like an object. **True**
13. The parseXxx methods always return a String. **False**
14. Formatting classes (which are decoupled from I/O) are in the java.format package. **False**

Data Structures



Sorting is a snap in Java. You have all the tools for collecting and manipulating your data without having to write your own sort algorithms (unless you're reading this right now sitting in your Computer Science 101 class, in which case, trust us—you are SO going to be writing sort code while the rest of us just call a method in the Java API). In this chapter, you're going to get a peek at when Java can save you some typing and figure out the types that you need.

The Java Collections Framework has a data structure that should work for virtually anything you'll ever need to do. Want to keep a list that you can easily keep adding to? Want to find something by name? Want to create a list that automatically takes out all the duplicates? Sort your co-workers by the number of times they've tried to speak with their mic muted on a video call? Sort your pets by number of tricks learned? It's all here...

Tracking song popularity on your jukebox

Congratulations on your new job—managing the automated jukebox system at Lou's Diner. There's no Java inside the jukebox itself, but each time someone plays a song, the song data is appended to a simple text file.

Your job is to manage the data to track song popularity, generate reports, and manipulate the playlists. You're not writing the entire app—some of the other software developers are involved as well, but you're responsible for managing and sorting the data inside the Java app. And since Lou has a thing against databases, this is strictly an in-memory data collection. Another programmer will be writing the code to read the song data from a file and put the songs into a List. (In a few chapters you'll learn how to read data from files, and write data to files.) All you're going to get is a List with the song data the jukebox keeps adding to.

Let's not wait for that other programmer to give us the actual file of songs; let's create a small test program to provide us with some sample data we can work with. We've agreed with the other programmer that she'll ultimately provide a Songs class with a getSongs method we'll use to get the data. Armed with that information, we can write a small class to temporarily "stand in" for the actual code. Code that stands in for other code is often called "**mock**" code.

You'll often want to write some temporary code that stands in for the real code that will come later. This is called "mocking."

We'll use this "mock" class to test our code.

We'll make this method static, because this class doesn't have any instance fields and doesn't need any.

```
class MockSongs {

  public static List<String> getSongStrings() {

    List<String> songs = new ArrayList<>();
    songs.add("somersault");
    songs.add("cassidy");
    songs.add("$10");
    songs.add("havana");
    songs.add("Cassidy");
    songs.add("50 Ways");
    return songs;
  }
}
```

This will be our list of six song titles to work with.

Because ArrayList IS-A List, we can create an ArrayList, store it in a List, and return List from the method.

In the real world you'll often see Java code that returns the interface type (List) and hides the implementation type (ArrayList).

Your first job, sort the songs in alphabetical order

We'll start by creating code that reads in data from the mock Songs class and prints out what it got. Since an ArrayList's elements are placed in the order in which they were added, it's no surprise that the song titles are not yet alphabetized.

```
import java.util.*;

public class Jukebox1 {
  public static void main(String[] args) {
    new Jukebox1().go();
  }

  public void go() {
    List<String> songList = MockSongs.getSongStrings();
    System.out.println(songList);
  }
}

// Below is the "mock" code. A stand in for the actual
// I/O code that the other programmer will provide later

class MockSongs {
  public static List<String> getSongStrings() {
    List<String> songs = new ArrayList<>();
    songs.add("somersault");
    songs.add("cassidy");
    songs.add("$10");
    songs.add("havana");
    songs.add("Cassidy");
    songs.add("50 Ways");
    return songs;
  }
}
```

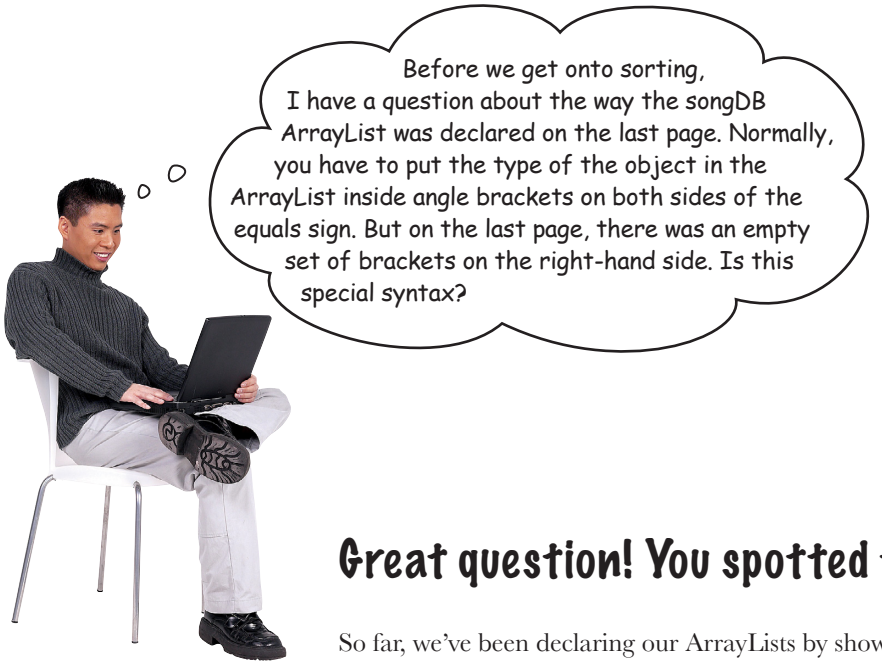
We'll store the song titles in a List of Strings.

Then print the contents of the songList.

Nothing special here...just some sample data we can use to work on our sorting code.

```
File Edit Window Help Dance
%java Jukebox1
[somersault, cassidy, $10, havana,
Cassidy, 50 Ways]
```

This is definitely NOT alphabetical!



Great question! You spotted the diamond operator

So far, we've been declaring our ArrayLists by showing the element type twice:

```
ArrayList<String> songs = new ArrayList<String>();
```

Most of the time, we don't need to say the same thing twice. The compiler can tell from what you wrote on the left-hand side what you probably want on the right-hand side. It uses *type inference* to infer (work out) the type you need.

```
ArrayList<String> songs = new ArrayList<>();
```

No type needed

This syntax is called the diamond operator (because, well, it's diamond-shaped!) and was introduced in Java 7, so it's been around a while and is probably available in your version of Java.

Over time, Java has evolved to remove unnecessary code duplication from its syntax. If the compiler can figure out a type, you don't always need to write it out in full.

there are no Dumb Questions

Q: Should I be using the diamond operator all the time? Are there any downsides?

A: The diamond operator is "syntactic sugar," which means it's there to make our lives easier as we write (and read) code, but it doesn't make any difference to the underlying byte code. So if you're worried about whether using the diamond means something different to using the specific type, don't worry! It's basically the same thing.

However, sometimes you might choose to write out the full type. The main reason you might want to is to help people reading your code. For example, if the variable is declared a long way from where it's initialized, you might want to use the type when you initialize it so you can see clearly what objects go into it.

```
ArrayList<String> songs;
// lots of code between these lines...
songs = new ArrayList<String>();
```

Q: Are there any other places the compiler can work out the types for me?

A: Yes! For example, the **var** keyword ("Local Variable Type Inference"), which we'll talk about in Appendix B. And another important example, lambda expressions, which we will see later in this chapter.

Q: I saw you were creating an ArrayList but assigning it to a List reference, and that you created an ArrayList but returned a List from the method. Why not just use ArrayList everywhere?

A: One of the advantages of polymorphism is that code doesn't need to know the specific implementation type of an object to work well with it. List is a well-known, well-understood interface (which we'll see more of in this chapter). Code that is working with an ArrayList doesn't usually need to know it's an ArrayList. It could be a LinkedList. Or a specialized type of List. Code that's working with the List only needs to know it can call List methods on it (like add(), size() etc). It's usually safer to pass the interface type (i.e., List) around instead of the implementation. That way, other code can't go rooting around inside your object in a way that was never intended.

It also means that should you ever want to change from an ArrayList to a LinkedList, or a CopyOnWriteArrayList (see Chapter 18, *Dealing with Concurrency Issues*) at a later date, you can without having to change all the places the List is used.

Exploring the java.util API, List and Collections

We know that with an ArrayList, or any List, the elements are kept in the order in which they were added. So we're going to need to sort the elements in the song list. In this chapter, we'll be looking at some of the most important and commonly used collection classes in the java.util package, but for now, let's limit ourselves to two classes: java.util.List and java.util.Collections.

We've been using ArrayList for a while now. Because ArrayList IS-A List and because many of the methods we're familiar with on ArrayList come from List, we can comfortably transfer most of what we know about working with ArrayLists to List.

The Collections class is known as a "utility" class. It's a class that has a lot of handy methods for working with the various collection types.

Excerpts from the API

java.util.List
sort(**Comparator**): Sorts this list according to the order induced by the specified **Comparator**.

java.util.Collections
sort(List): Sorts the specified list into ascending order, according to the **natural ordering** of its elements.

sort(List, **Comparator**): Sorts the specified list according to the order defined by the **Comparator**.

In the "Real-World"™ there are lots of ways to sort

We don't always want our lists sorted alphabetically. We might want to sort clothes by size, or movies by how many five-star reviews they get. Java lets you sort the good old-fashioned way, alphabetically, and it also lets you create your own custom sorting approaches. Those references you see above to "Comparator" have to do with custom sorting, which we'll get to later this chapter. So for now, let's stick with "natural ordering" (alphabetical).

Since we know we have a List, it looks like we've found the perfect method, **Collections.sort()**.

"Natural Ordering," what Java means by alphabetical

Lou wants you to sort songs "alphabetically," but what exactly does that mean? The A–Z part is obvious, but how about lowercase versus uppercase letters? How about numbers and special characters? Well, this is another can-of-worms topic, but Java uses Unicode, and for many of us in "the West" that means that numbers sort before uppercase letters, uppercase letters sort before lowercase letters, and some special characters sort before numbers and some sort after numbers. That's pretty clear, right? Ha! Well, the upshot is that, by default, sorting in Java happens in what's called "natural order," which is more or less alphabetical. Let's take a look at what happens when we sort our list of songs:

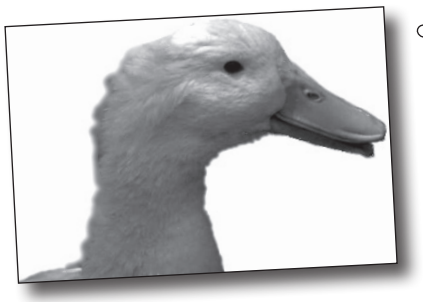
```
  public void go() {
    List<String> songList = MockSongs.getSongStrings();
    System.out.println(songList);
    Collections.sort(songList);
    System.out.println(songList);
  }
}
```

Sort our song titles using "natural ordering."

```
File Edit Window Help Dance
%java Jukebox1
[somersault, cassidy, $10, havana, Cassidy, 50 Ways]
[$10, 50 Ways, Cassidy, cassidy, havana, somersault]
```

Our songs unsorted, in the order they were added.

Our songs sorted. Notice how the special characters, numbers, and uppercase letters got sorted.



Just FYI, we ducks are very particular about how we get sorted.

But now you need Song objects, not just simple Strings

Now your boss Lou wants actual Song class instances in the list, not just Strings, so that each Song can have more data. The new jukebox device outputs more information, so the actual song file will have *three* pieces of information for each song.

The Song class is really simple, with only one interesting feature—the overridden toString() method. Remember, the toString() method is defined in class Object, so every class in Java inherits the method. And since the toString() method is called on an object when it's printed (System.out.println(anObject)), you should override it to print something more readable than the default unique identifier code. When you print a list, the toString() method will be called on each object.

```
class SongV2 {
  private String title;
  private String artist;
  private int bpm;

  SongV2(String title, String artist, int bpm) {
    this.title = title;
    this.artist = artist;
    this.bpm = bpm;
  }
  public String getTitle() {
    return title;
  }
  public String getArtist() {
    return artist;
  }
  public int getBpm() {
    return bpm;
  }
  public String toString() {
    return title;
  }
}
```

Three instance variables for the three song attributes in the file.

The variables are all set in the constructor whenever a new Song is created.

The getter methods for the three attributes.

We override toString(), because when you do a System.out.println(aSongObject), we want to see the title. When you do a System.out.println(aListOfSongs), it calls the toString() method of EACH element in the list.

```
class MockSongs {
  public static List<String> getSongStrings() { ... }

  public static List<SongV2> getSongsV2() {
    List<SongV2> songs = new ArrayList<>();
    songs.add(new SongV2("somersault", "zero 7", 147));
    songs.add(new SongV2("cassidy", "grateful dead", 158));
    songs.add(new SongV2("$10", "hitchhiker", 140));

    songs.add(new SongV2("havana", "cabello", 105));
    songs.add(new SongV2("Cassidy", "grateful dead", 158));
    songs.add(new SongV2("50 ways", "simon", 102));
    return songs;
  }
}
```

We made a new method in the MockSongs class to mock the new song data.

Changing the Jukebox code to use Songs instead of Strings

Your code changes only a little. The big change is that the List will be of type <SongV2> instead of <String>.

```
import java.util.*;

public class Jukebox2 {
  public static void main(String[] args) {
    new Jukebox2().go();
  }

  public void go() {

    List<SongV2> songList = MockSongs.getSongsV2();
    System.out.println(songList);

    Collections.sort(songList);
    System.out.println(songList);
  }
}
```

Change to a List of SongV2 objects instead of Strings.

Call the mock class to load song data into our List of songs.

And once again, call the sort method to sort the songs.

It won't compile!

He's right to be curious, something's wrong...the Collections class clearly shows there's a sort() method that takes a List. It *should* work.

But it doesn't!

The compiler says it can't find a sort method that takes a List<SongV2>, so maybe it doesn't like a List of Song objects? It didn't mind a List<String>, so what's the important difference between Song and String? What's the difference that's making the compiler fail?

File Edit Window Help Bummer

```
%javac Jukebox2.java
Jukebox2.java:13: error: no suitable method found for
sort(List<SongV2>)
    Collections.sort(songList);
               ^
...
1 error
```

And of course you probably already asked yourself, "What would it be sorting *on*?" How would the sort method even *know* what made one Song greater or less than another Song? Obviously if you want the song's *title* to be the value that determines how the songs are sorted, you'll need some way to tell the sort method that it needs to use the title and not, say, the beats per minute.

We'll get into all that a few pages from now, but first, let's find out why the compiler won't even let us pass a SongV2 List to the sort() method.

The sort() method declaration

```
static <T extends Comparable<? super T>> void sort(List<T> list)
```

Sorts the specified list into ascending order, according to the natural ordering of its elements.

From the API docs (looking up the java.util.Collections class and scrolling to the sort() method), it looks like the sort() method is declared...*strangely*. Or at least different from anything we've seen so far.

That's because the sort() method (along with other things in the whole collection framework in Java) makes heavy use of *generics*. Any time you see something with angle brackets in Java source code or documentation, it means generics—a feature added in Java 5. So it looks like we'll have to learn how to interpret the documentation before we can figure out why we were able to sort String objects in a List, but not a List of Song objects.

Generics means more type-safety

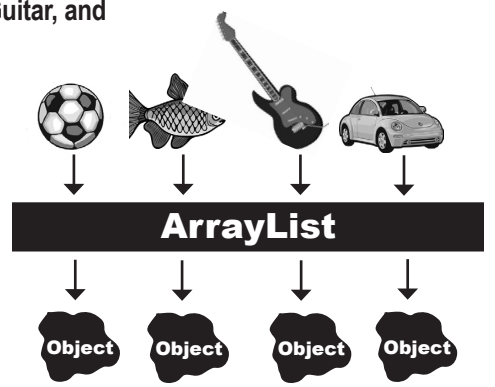
Although generics can be used in other ways, you'll often use generics to write type-safe collections. In other words, code that makes the compiler stop you from putting a Dog into a list of Ducks.

Without generics the compiler could not care less what you put into a collection, because all collection implementations hold type Object. You could put *anything* in any ArrayList without generics; it's like the ArrayList is declared as ArrayList<Object>.

Before generics, there was no way to declare the type of an ArrayList, so its add() method took type Object.

WITHOUT generics

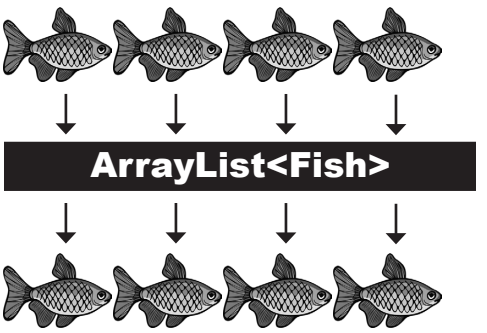
Objects go IN as a reference to SoccerBall, Fish, Guitar, and Car objects



And come OUT as a reference of type Object

WITH generics

Objects go IN as a reference to only Fish objects



And come out as a reference of type Fish

Without generics, the compiler would happily let you put a Pumpkin into an ArrayList that was supposed to hold only Cat objects.

With generics, you can create type-safe collections where more problems are caught at compile-time instead of runtime.

Now with generics, you can put only Fish objects in the ArrayList<Fish>, so the objects come out as Fish references. You don't have to worry about someone sticking a Volkswagen in there, or that what you get out won't really be castable to a Fish reference.

Learning generics

Of the dozens of things you could learn about generics, there are really only three that matter to most programmers:

1 **Creating instances of generic *classes* (like ArrayList)**

When you make an ArrayList, you have to tell it the type of objects you'll allow in the list, just as you do with plain old arrays.

```
new ArrayList<Song>()
```

2 **Declaring and assigning *variables* of generic types**

How does polymorphism really work with generic types? If you have an ArrayList<Animal> reference variable, can you assign an ArrayList<Dog> to it? What about a List<Animal> reference? Can you assign an ArrayList<Animal> to it? You'll see...

```
List<Song> songList =
      new ArrayList<Song>()
```

3 **Declaring (and invoking) *methods* that take generic types**

If you have a method that has as a parameter, say, an ArrayList of Animal objects, what does that really mean? Can you also pass it an ArrayList of Dog objects? We'll look at some subtle and tricky polymorphism issues that are very different from the way you write methods that take plain old arrays.

(This is actually the same point as #2, but that shows you how important we think it is.)

```
void foo(List<Song> list)

x.foo(songList)
```

there are no Dumb Questions

Q: **But don't I also need to learn how to create my OWN generic classes? What if I want to make a class type that lets people instantiating the class decide the type of things that class will use?**

A: You probably won't do much of that. Think about it—the API designers made an entire library of collections classes covering most of the data structures you'd need, and the things that really need to be generic are collection classes or classes and methods that work on collections. There are some other cases too (like Optional, which we'll see in the next chapter). Generally, generic classes are classes that hold or operate on objects of some other type they don't know about.

Yes, it is possible that you might want to *create* generic classes, but that's pretty advanced, so we won't cover it here. (But you can figure it out from the things we *do* cover, anyway.)

Using generic CLASSES

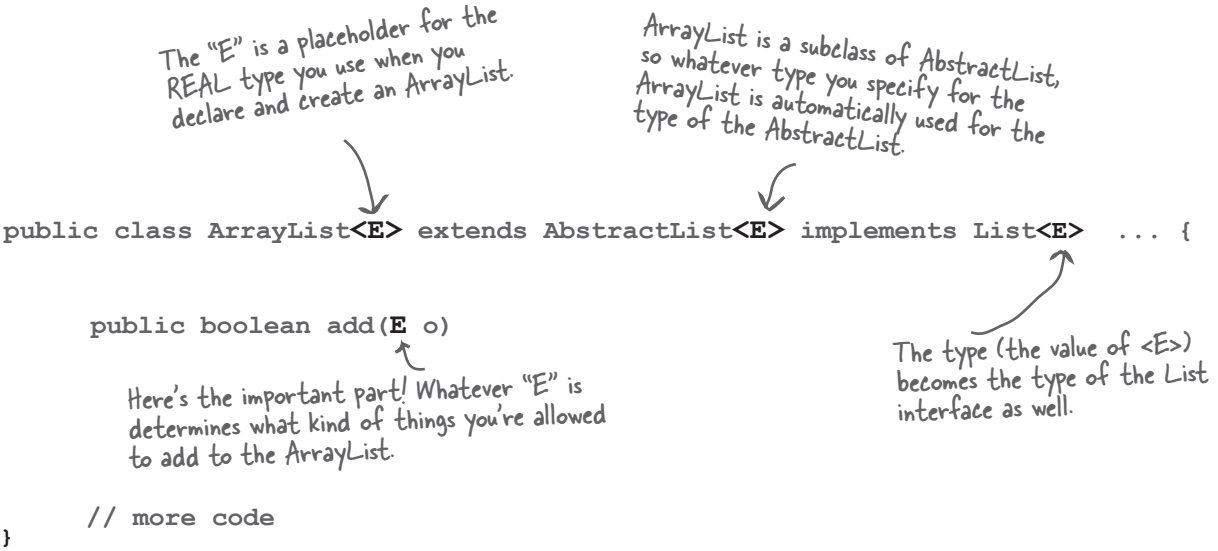
Since ArrayList is one of our most-used generic classes, we'll start by looking at its documentation. The two key areas to look at in a generic class are:

1. The *class* declaration

2. The *method* declarations that let you add elements

Understanding ArrayList documentation

(Or, what's the true meaning of "E"?)



The "E" represents the type used to create an instance of ArrayList. When you see an "E" in the ArrayList documentation, you can do a mental find/replace to exchange it for whatever <type> you use to instantiate ArrayList.

So, new ArrayList<Song> means that "E" becomes "Song" in any method or variable declaration that uses "E."

Think of "E" as a stand-in for "the type of element you want this collection to hold and return." (E is for Element.)

Using type parameters with ArrayList

THIS code:

```
List<String> thisList = new ArrayList<>
```

Means ArrayList:

```
public class ArrayList<E> extends AbstractList<E> ... {

      public boolean add(E o)
      // more code
}
```

Is treated by the compiler as:

```
public class ArrayList<String> extends AbstractList<String>... {

      public boolean add(String o)
      // more code
}
```

In other words, the "E" is replaced by the *real* type (also called the *type parameter*) that you use when you create the ArrayList. And that's why the add() method for ArrayList won't let you add anything except objects of a reference type that's compatible with the type of "E." So if you make an ArrayList**<String>**, the add() method suddenly becomes **add(String o)**. If you make the ArrayList of type **Dog**, suddenly the add() method becomes **add(Dog o)**.

there are no Dumb Questions

Q: **Is "E" the only thing you can put there? Because the docs for sort used "T"....**

A: You can use anything that's a legal Java identifier. That means anything that you could use for a method or variable name will work as a type parameter. But you'll often see single letter used. Another convention is to use "T" ("Type") unless you're specifically writing a collection class, where you'd use "E" to represent the "type of the Element the collection will hold." Sometimes you'll see "R" for "Return type."

Using generic METHODS

A generic *class* means that the *class declaration* includes a type parameter. A generic *method* means that the *method declaration* uses a type parameter in its signature.

You can use type parameters in a method in several different ways:

1 Using a type parameter defined in the class declaration

```
public class ArrayList<E> extends AbstractList<E> ... {
  public boolean add(E o)
```

You can use the "E" here ONLY because it's already been defined as part of the class.

When you declare a type parameter for the class, you can simply use that type any place that you'd use a *real* class or interface type. The type declared in the method argument is essentially replaced with the type you use when you instantiate the class.

2 Using a type parameter that was NOT defined in the class declaration

```
public <T extends Animal> void takeThing(ArrayList<T> list)
```

If the class itself doesn't use a type parameter, you can still specify one for a method, by declaring it in a really unusual (but available) space—*before the return type*. This method says that T can be "any type of Animal."

Here we can use <T> because we declared "T" at the start of the method declaration.

Here's where it gets weird...

This:

```
public <T extends Animal> void takeThing(ArrayList<T> list)
```

Is NOT the same as this:

```
public void takeThing(ArrayList<Animal> list)
```

Both are legal, but they're *different*!

The first one, where **<T extends Animal>** is part of the method declaration, means that any ArrayList declared of a type that is Animal, or one of Animal's subtypes (like Dog or Cat), is legal. So you could invoke the top method using an ArrayList<Dog>, ArrayList<Cat>, or ArrayList<Animal>.

But...the one on the bottom, where the method argument is (ArrayList<Animal> list) means that *only* an ArrayList<Animal> is legal. In other words, while the first version takes an ArrayList of any type that is a type of Animal (Animal, Dog, Cat, etc.), the second version takes *only* an ArrayList of type Animal. Not ArrayList<Dog> or ArrayList<Cat>, but only ArrayList<Animal>.

And yes, it does appear to violate the point of polymorphism, but it will become clear when we revisit this in detail at the end of the chapter. For now, remember that we're only looking at this because we're still trying to figure out how to sort() that SongList, and that led us into looking at the API for the sort() method, which had this strange generic type declaration.

For now, all you need to know is that the syntax of the top version is legal and that it means you can pass in a ArrayList object instantiated as Animal or any Animal subtype.

And now back to our sort() method...

Remember where we were...

```
File Edit Window Help Bummer
%javac Jukebox2.java
Jukebox2.java:13: error: no suitable method found
for sort(List<SongV2>)
    Collections.sort(songList);
               ^
...
1 error
```

```
import java.util.*;

public class Jukebox2 {
  public static void main(String[] args) {
    new Jukebox2().go();
  }

  public void go() {

    List<SongV2> songList = MockSongs.getSongsV2();
    System.out.println(songList);

    Collections.sort(songList);
    System.out.println(songList);
  }
}
```

This is where it breaks! It worked fine when passed in a List<String>, but as soon as we tried to sort a List<SongV2>, it failed.

Revisiting the sort() method

So here we are, trying to read the sort() method docs to find out why it was OK to sort a list of Strings, but not a list of Song objects. And it looks like the answer is...

```
static <T extends Comparable<? super T>> void sort(List<T> list)
```

Sorts the specified list into ascending order, according to the natural ordering of its elements.

The sort() method can take only lists of Comparable objects.

Song is NOT a subtype of Comparable, so you cannot sort() the list of Songs.

At least not yet...

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

This says "Whatever 'T' is must be of type Comparable."

(Ignore this part for now. But if you can't, it just means that the type parameter for Comparable must be of type T or one of T's supertypes.)

You can pass in only a List (or subtype of list, like ArrayList) that uses a parameterized type that "extends Comparable."

Um...I just checked the docs for String, and String doesn't EXTEND Comparable—it IMPLEMENTS it. **Comparable is an interface**. So it's nonsense to say <T ***extends*** Comparable>.

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {
```

Great point, and one that deserves a full answer! Turn the page...

In generics, "extends" means "extends or implements"

The Java engineers had to give you a way to put a constraint on a parameterized type so that you can restrict it to, say, only subclasses of Animal. But you also need to constrain a type to allow only classes that implement a particular interface. So here's a situation where we need one kind of syntax to work for both situations—inheritance and implementation. In other words, that works for both *extends* and *implements*.

And the winning word was...*extends*. But it really means "IS-A" and works regardless of whether the type on the right is an interface or a class.

In generics, the keyword "extends" really means "IS-A" and works for BOTH classes and interfaces.

Comparable is an interface, so this REALLY reads, "T must be a type that implements the Comparable interface."

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

It doesn't matter whether the thing on the right is a class or interface...you still say "extends."

there are no Dumb Questions

Q: **Why didn't they just make a new keyword, "is"?**

A: Adding a new keyword to the language is a REALLY big deal because it risks breaking Java code you wrote in an earlier version. Think about it—you might be using a variable "is" (which we do use in this book to represent input streams). And since you're not allowed to use keywords as identifiers in your code, that means any earlier code that used the keyword *before* it was a reserved word, would break. So whenever there's a chance for the language engineers to reuse an existing keyword, as they did here with "extends," they'll usually choose that. But sometimes they don't have a choice.

In recent years, new "keyword-like" terms have been added to the language, without actually making it a keyword that would trample all over your earlier code. For example, the identifier **`var`**, which we'll talk about in Appendix B, is a *reserved type name*, **not** a keyword. This means any existing code that used var (for example, as a variable name) will not break if it's compiled with a version of Java that supports **`var`**.

Finally we know what's wrong...
The Song class needs to implement Comparable

We can pass the ArrayList<Song> to the sort() method only if the Song class implements Comparable, since that's the way the sort() method was declared. A quick check of the API docs shows the Comparable interface is really simple, with only one method to implement:

java.lang.Comparable

```
public interface Comparable<T> {
   int compareTo(T o);
}
```

And the method documentation for compareTo() says:

Returns:
a negative integer if this object is less than the specified object;
a zero if they're equal;
a positive integer if this object is greater than the specified object.

It looks like the compareTo() method will be called on one Song object, passing that Song a reference to a different Song. The Song running the compareTo() method has to figure out if the Song it was passed should be sorted higher, lower, or the same in the list.

Your big job now is to decide what makes one song greater than another, and then implement the compareTo() method to reflect that. A negative number (any negative number) means the Song you were passed is greater than the Song running the method. Returning a positive number says that the Song running the method is greater than the Song passed to the compareTo() method. Returning zero means the Songs are equal (at least for the purpose of sorting...it doesn't necessarily mean they're the same object). You might, for example, have two Songs by different artists with the same title.

(That brings up a whole different can of worms we'll look at later...)

The big question is: what makes *one* song less than, equal to, or greater than *another* song?

You can't implement the Comparable interface until you make that decision.

Sharpen your pencil

Write in your idea and pseudocode (or better, REAL code) for implementing the compareTo() method in a way that will sort() the Song objects by title.

Hint: if you're on the right track, it should take less than three lines of code!

→ Yours to solve.

The new, improved, comparable Song class

We decided we want to sort by title, so we implement the compareTo() method to compare the title of the Song passed to the method against the title of the song on which the compareTo() method was invoked. In other words, the song running the method has to decide how its title compares to the title of the method parameter.

Hmmm...we know that the String class must know about alphabetical order, because the sort() method worked on a list of Strings. We know String has a compareTo() method, so why not just call it? That way, we can simply let one title String compare itself to another, and we don't have to write the comparing/alphabetizing algorithm!

Usually these match...we're specifying the type that the implementing class can be compared against.

This means that SongV3 objects can be compared to other SongV3 objects, for the purpose of sorting.

```
class SongV3 implements Comparable<SongV3> {
  private String title;
  private String artist;
  private int bpm;

  public int compareTo(SongV3 s) {
    return title.compareTo(s.getTitle());
  }

  SongV3(String title, String artist, int bpm) {
    this.title = title;
    this.artist = artist;
    this.bpm = bpm;
  }

  public String getTitle() {
    return title;
  }

  public String getArtist() {
    return artist;
  }

  public int getBpm() {
    return bpm;
  }

  public String toString() {
    return title;
  }
}
```

The sort() method sends a Song to compareTo() to see how that Song compares to the Song on which the method was invoked.

Simple! We just pass the work on to the title String objects, since we know Strings have a compareTo() method.

This time it worked. It prints the list and then calls sort, which puts the Songs in alphabetical order by title.

```
File Edit Window Help Ambient
%java Jukebox3
[somersault, cassidy, $10, havana, Cassidy, 50 ways]
[$10, 50 ways, Cassidy, cassidy, havana, somersault]
```

We can sort the list, but...

There's a new problem—Lou wants two different views of the song list, one by song title and one by artist!

But when you make a collection element comparable (by having it implement Comparable), you get only one chance to implement the compareTo() method. So what can you do?

The horrible way would be to use a flag variable in the Song class and then do an *if* test in compareTo() and give a different result depending on whether the flag is set to use title or artist for the comparison.

But that's an awful and brittle solution, and there's something much better. Something built into the API for just this purpose—when you want to sort the same thing in more than one way.

Look at API documentation again. There's a second sort() method on Collections—and it takes a Comparator. There's also a sort method on List that takes a Comparator.

Excerpts from the API

`java.util.Collections`
sort(List): Sorts the specified list into ascending order, according to the **natural ordering** of its elements.

sort(List, **Comparator**): Sorts the specified list according to the order induced by the specified **Comparator**.

`java.util.List`
sort(**Comparator**): Sorts this list according to the order induced by the specified **Comparator**.

The sort() method on Collections is overloaded to take something called a Comparator.

There's also a sort() on List, which takes a Comparator.

Note to self: figure out how to get/make a Comparator that can compare and order the songs by artist instead of title.

Using a custom Comparator

A **Comparable** element in a list can compare *itself* to another of its own type in only one way, using its compareTo() method. But a **Comparator** is external to the element type you're comparing—it's a separate class. So you can make as many of these as you like! Want to compare songs by artist? Make an ArtistComparator. Sort by beats per minute? Make a BpmComparator.

Then all you need to do is call a sort() method that takes a Comparator (Collections.sort or List.sort), which will use this Comparator to put things in order.

The sort() method that takes a Comparator will use the Comparator instead of the element's own compareTo() method when it puts the elements in order. In other words, if your sort() method gets a Comparator, it won't even *call* the compareTo() method of the elements in the list. The sort() method will instead invoke the **compare()** method on the Comparator.

To summarize, the rules are:

- Invoking the **Collections.sort(List list)** method means the list element's compareTo() method determines the order. The elements in the list MUST implement the **Comparable** interface.
- Invoking **List.sort(Comparator c)** or **Collections.sort(List list, Comparator c)** means the **Comparator**'s compare() method will be used. That means the elements in the list do NOT need to implement the Comparable interface, but if they do, the list element's compareTo() method will NOT be called.

java.util.Comparator

```
public interface Comparator<T> {
   int compare(T o1, T o2);
}
```

If you pass a Comparator to the sort() method, the sort order is determined by the Comparator.

If you don't pass a Comparator and the element is Comparable, the sort order is determined by the element's compareTo() method.

there are no Dumb Questions

Q: Why are there two sort methods that take a comparator on two different classes? Which sort method should I use?

A: Both methods that take a comparator, Collections.sort(List, Comparator) and List.sort(Comparator), do the same thing; you can use either and expect exactly the same results.

List.sort was introduced in Java 8, so older code *must* use Collections.sort(List, Comparator).

We use List.sort because it's a bit shorter, and generally you already have the list you want to sort, so it makes sense to call the sort method on that list.

Updating the Jukebox to use a Comparator

We're going to update the Jukebox code in three ways:

1. Create a separate class that implements Comparator (and thus the ***compare()*** method that does the work previously done by ***compareTo()***).

2. Make an instance of the Comparator class.

3. Call the List.sort() method, giving it the instance of the Comparator class.

```
import java.util.*;

public class Jukebox4 {
  public static void main(String[] args) {
    new Jukebox4().go();
  }

  public void go() {
    List<SongV3> songList = MockSongs.getSongsV3();
    System.out.println(songList);

    Collections.sort(songList);
    System.out.println(songList);

    ArtistCompare artistCompare = new ArtistCompare();
    songList.sort(artistCompare);
    System.out.println(songList);
  }
}

class ArtistCompare implements Comparator<SongV3> {
  public int compare(SongV3 one, SongV3 two) {
    return one.getArtist().compareTo(two.getArtist());
  }
}
```

Make an instance of the Comparator class.

Invoke sort() on our list, passing it a reference to the new custom Comparator object.

This is a String (the artist)

We're letting the String variables (for artist) do the actual comparison, since Strings already know how to alphabetize themselves.

```
File Edit Window Help Ambient
%java Jukebox4
[somersault, cassidy, $10, havana, Cassidy, 50 ways]
[$10, 50 ways, Cassidy, cassidy, havana, somersault]
[havana, Cassidy, cassidy, $10, 50 ways, somersault]
```

Unsorted songList

Sorted by title (using the Song's compareTo method)

Sorted by artist name (using ArtistComparator)

Fill-in-the-blanks

For each of the questions below, fill in the blank with one of the words from the "possible answers" list, to correctly answer the question.

Possible Answers:

Comparator,

Comparable,

compareTo( ),

compare( ),

yes,

no

Given the following compilable statement:

```
Collections.sort(myArrayList);
```

1. What must the class of the objects stored in `myArrayList` implement? _____

2. What method must the class of the objects stored in `myArrayList` implement? _____

3. Can the class of the objects stored in `myArrayList` implement both `Comparator` AND `Comparable`? _____

Given the following compilable statements (they both do the same thing):

```
Collections.sort(myArrayList, myCompare);
myArrayList.sort(myCompare);
```

4. Can the class of the objects stored in `myArrayList` implement `Comparable`? _____

5. Can the class of the objects stored in `myArrayList` implement `Comparator`? _____

6. Must the class of the objects stored in `myArrayList` implement `Comparable`? _____

7. Must the class of the objects stored in `myArrayList` implement `Comparator`? _____

8. What must the class of the `myCompare` object implement? _____

9. What method must the class of the `myCompare` object implement? _____

Answers on page 364.

But wait! We're sorting in two different ways!

Now we're able to sort the song list two ways:

1. Using Collections.sort(songList), because Song implements **Comparable**

2. Using songLists.sort(artistCompare) because the ArtistCompare class implements **Comparator**

While our new code allows us to sort songs by title and by artist, it is reminiscent of Frankenstein's monster, cobbled together bit by bit.

```
public void go() {
 List<SongV3> songList = MockSongs.getSongsV3();
 System.out.println(songList);

 Collections.sort(songList);
 System.out.println(songList);

 ArtistCompare artistCompare = new ArtistCompare();
 songList.sort(artistCompare);
 System.out.println(songList);
}
```

This uses Comparable to sort

This uses a custom Comparator to sort

A better approach would be to handle all of the sorting definitions in classes that implement Comparator.

there are no Dumb Questions

Q: So does this mean that if you have a class that doesn't implement Comparable, and you don't have the source code, you could still put the things in order by creating a Comparator?

A: That's right. The other option (if it's possible) would be to subclass the element and make the subclass implement Comparable.

Q: But why doesn't *every* class implement Comparable?

A: Do you really believe that *everything* can be ordered? If you have element types that just don't lend themselves to any kind of natural ordering, then you'd be misleading other programmers if you implement Comparable. And there's no problem if you *don't* implement Comparable, since a programmer can compare anything in any way that they choose using their own custom Comparator.

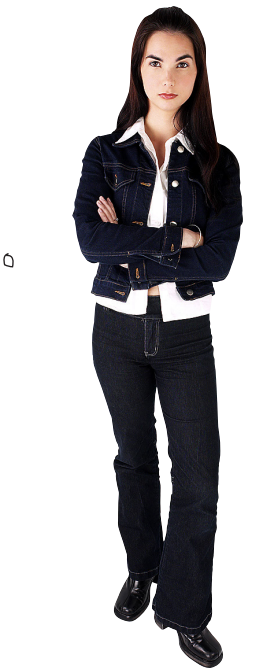
Sorting using only Comparators

Having Song implement Comparable and creating a custom Comparator for sorting by Artist absolutely works, but it's confusing to rely on two different mechanisms for our sort. It's much clearer if our code uses the same technique to sort, regardless of how Lou wants his songs sorted. The code below has been updated to use Comparators for sorting by both Title and Artist; the new code is in bold.

```
public class Jukebox5 {
  public static void main(String[] args) {
    new Jukebox5().go();
  }

  public void go() {
    List<SongV3> songList = MockSongs.getSongsV3();
    System.out.println(songList);

    TitleCompare titleCompare = new TitleCompare();
    songList.sort(titleCompare);
    System.out.println(songList);

    ArtistCompare artistCompare = new ArtistCompare();
    songList.sort(artistCompare);
    System.out.println(songList);
  }
}
class TitleCompare implements Comparator<SongV3> {
  public int compare(SongV3 one, SongV3 two) {
    return one.getTitle().compareTo(two.getTitle());
  }
}
class ArtistCompare implements Comparator<SongV3> {
  public int compare(SongV3 one, SongV3 two) {
    return one.getArtist().compareTo(two.getArtist());
  }
}

// more specialized Comparator classes could go here,
// e.g. BpmCompare
```

Make an instance of the Comparator class and use the sort() method on List.

This is the new class that implements Comparator.



That's an awful lot of code for sorting our songs in just two different orders. Isn't there a better way?

Just the code that matters

The Jukebox class does have a lot of code that's needed for sorting. Let's zoom in on one of the Comparator classes we wrote for Lou. The first thing to notice is that all we really want to sort our collection is the one line of code in the middle of the class. The rest of the code is just the long-winded syntax that's necessary to let the compiler know what type of class this is and which method it implements.

Code Up Close

```
class TitleCompare implements Comparator<Song> {
   public int compare(Song one, Song two) {
     return one.getTitle().compareTo(two.getTitle());
   }
}
```

The one line of code that's doing all the work

There's more than one way to declare small pieces of functionality like this. One approach is inner classes, which we'll look at in a later chapter. You can even declare the inner class right where you use it (instead of at the end of your class file); this is sometimes called an "argument-defined anonymous inner class." Sounds fun already:

```
songList.sort(new Comparator<SongV3>() {
  public int compare(SongV3 one, SongV3 two) {
    return one.getTitle().compareTo(two.getTitle());
  }
});
```

While this lets us declare the sorting logic in exactly the location we need it (where we call the sort method, instead of in a separate class), there's still a lot of code there for saying "sort by title please").

Relax

We're not going to learn how to write "argument-defined anonymous inner classes"!

We just wanted you to see this example, in case you stumble across it in Real Code.

What do we REALLY need in order to sort?

```
public class Jukebox5 {
  public void go() {
1   List<SongV3> songList = MockSongs.getSongsV3();
    ...
    TitleCompare titleCompare = new TitleCompare();
2   songList.sort(titleCompare);
    ...
  }
}
class TitleCompare implements Comparator<SongV3> {
  public int compare(SongV3 one, SongV3 two) {
    return one.getTitle().compareTo(two.getTitle());
  }
}
```

The compiler knows the List contains SongV3 objects.

The compiler understands that sort() expects a Comparator for SongV3 objects.

Let's take a look at the API documentation for the sort method on List:

`default void sort(Comparator<? super E> c)`	Sorts this list according to the order induced by the specified `Comparator`.

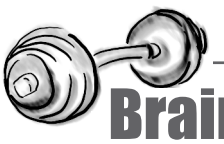
If we were to explain out loud the following line of code:

```
songList.sort(titleCompare);
```

We could say:

"Call the sort method on the list of songs 1 and pass it a reference to a Comparator object, which is designed specifically to sort Song objects 2."

If we're honest, we could say all that without even looking at the TitleCompare class. We can work it all out just by looking at the documentation for sort and the type of the List that we're sorting!



Brain Barbell

Do you think the compiler cares about the name "TitleCompare"? If the class was called "FooBar" instead, would the code still work?

Wouldn't it be wonderful if the code that described HOW you want to sort your Songs wasn't so far away from the sort method? And wouldn't it be great if you didn't have to write a bunch of code the compiler could probably figure out on its own...

Enter lambdas! Leveraging what the compiler can infer

We could write a whole bunch of code to say how to sort a list (like we have been doing)...

```
...
songList.sort(titleCompare);
...
```

This is just a reference to an object that implements Comparator. Its name doesn't matter to the compiler.

The compiler cares not a whit what you call the class.

Doh! The compiler can infer this from the sort() docs.

```
class TitleCompare implements Comparator<Song> {
    public int compare(Song one, Song two) {
        return one.getTitle().compareTo(two.getTitle());
    }
}
...
```

Yup, the compiler knows what this method should look like.

The compiler can figure out that the two objects have to be Song objects since songList is a List of Song objects.

This is ALL THE COMPILER NEEDS. Just tell it HOW to do the sort!

Or, we could use a lambda...

```
songList.sort((one, two) -> one.getTitle().compareTo(two.getTitle()));
```

These are Song one and Song two, the parameters to the compare method.

Brain Barbell

What do you think would happen if Comparator needed you to implement more than one method? How much could the compiler fill in for you?

Where did all that code go?

To answer this question, let's take a look at the API documentation for the Comparator interface.

Method Summary

Modifier and Type	Method	Description
`int`	`compare(T o1, T o2)`	Compares its two arguments for order.
`boolean`	`equals(Object obj)`	Indicates whether some other object is "equal to" this comparator.

Remember from Chapter 8 that every class and interface inherits methods from class Object, and that equals() is implemented in class Object.

Because equals has been implemented by Object, if we create a custom comparator, we know we only *need* to implement the compare method.

We also know exactly the shape of that method—it has to return an int, and it takes two arguments of type T (remember generics?). Our lambda expression implements the compare() method, without having to declare the class or the method, only the details of what goes into the body of the compare() method.

Some interfaces have only ONE method to implement

With interfaces like Comparator, we only have to implement a ***single abstract method***, SAM for short. These interfaces are so important that they have several special names:

SAM Interfaces a.k.a. Functional Interfaces

If an interface has only one method that needs to be implemented, that interface can be implemented as a ***lambda expression***. You don't need to create a whole class to implement the interface; the compiler knows what the class and method would look like. What the compiler *doesn't* know is the logic that goes *inside* that method.

We'll look at lambda expressions and functional interfaces in much more detail in the next chapter. For now, back to Lou's diner.

Updating the Jukebox code with lambdas

```
import java.util.*;

public class Jukebox6 {
  public static void main(String[] args) {
    new Jukebox6().go();
  }

  public void go() {
    List<SongV3> songList = MockSongs.getSongsV3();
    System.out.println(songList);

    songList.sort((one, two) -> one.getTitle().compareTo(two.getTitle()));
    System.out.println(songList);

    songList.sort((one, two) -> one.getArtist().compareTo(two.getArtist()));
    System.out.println(songList);
  }
}
```

Here's our lambda expression in action—no need to create a custom Comparator class; just put the sorting logic right inside sort method call.

You can tell what the list will be sorted by, just by looking at the field used in the lambda.

The output is exactly the same as when we used Comparator classes, but our code was much shorter.

```
File Edit Window Help Ambient
%java Jukebox6
[somersault, cassidy, $10, havana, Cassidy, 50 ways]
[$10, 50 ways, Cassidy, cassidy, havana, somersault]
[havana, Cassidy, cassidy, $10, 50 ways, somersault]
```

How could you sort the songs differently?

Write lambda expressions to sort the songs in these ways (the answers are at the end of the chapter):

- Sort by BPM
- Sort by title in descending order

Answers on page 366.

Reverse Engineer

Assume this code exists in a single file. Your job is to fill in the blanks so the program will create the output shown.

```
import ____________________;

public class SortMountains {
  public static void main(String [] args) {
    new SortMountains().go();
  }

  public void go() {
    List_________________ mountains = new ArrayList<>();
    mountains.add(new Mountain("Longs", 14255));
    mountains.add(new Mountain("Elbert", 14433));
    mountains.add(new Mountain("Maroon", 14156));
    mountains.add(new Mountain("Castle", 14265));
    System.out.println("as entered:\n" + mountains);

    mountains._______(____->_____________);
    System.out.println("by name:\n" + mountains);

    _________._______(____->_____________);
    System.out.println("by height:\n" + mountains);
  }
}

class Mountain {
  ____________________;
  ____________________;

  _____________________ {
    ____________________;
    ____________________;
  }
  ________________________ {
    ____________________________________;
  }
}
```

Output:

```
File  Edit  Window  Help  ThisOne'sForBob
%java SortMountains
as entered:
[Longs 14255, Elbert 14433, Maroon 14156, Castle 14265]
by name:
[Castle 14265, Elbert 14433, Longs 14255, Maroon 14156]
by height:
[Elbert 14433, Castle 14265, Longs 14255, Maroon 14156]
```

Answers on page 365.

Uh-oh. The sorting all works, but now we have duplicates...

The sorting works great; now we know how to sort on both *title* and *artist*. But there's a new problem we didn't notice with a test sample of the jukebox songs—***the sorted list contains duplicates.***

Unlike the mock code, Lou's real jukebox application appears to just keep writing to the file regardless of whether the same song has already been played (and thus written) to the text file. The *SongListMore.txt* jukebox text file is an example. It's a complete record of every song that was played, and might contain the same song multiple times.

```
File Edit Window Help TooManyNotes
%java Jukebox7

[somersault: zero 7, cassidy: grateful dead, $10: hitchhiker,
havana: cabello, $10: hitchhiker, cassidy: grateful dead, 50
ways: simon]

[$10: hitchhiker, $10: hitchhiker, 50 ways: simon, cassidy:
grateful dead, cassidy: grateful dead, havana: cabello,
somersault: zero 7]

[havana: cabello, cassidy: grateful dead, cassidy: grateful
dead, $10: hitchhiker, $10: hitchhiker, 50 ways: simon,
somersault: zero 7]
```

Before sorting

After sorting by song title

After sorting by artist name

Note that we changed the Song's toString to output the title and the artist.

SongListMore.txt

This is what the actual song data file looks like.

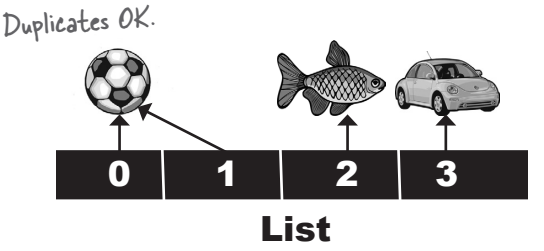
```
somersault, zero 7, 147
cassidy, grateful dead, 158
$10, hitchhiker, 140
havana, cabello, 105
$10, hitchhiker, 140
cassidy, grateful dead, 158
50 ways, simon, 102
```

The SongListMore text file now has duplicates in it, because the jukebox machine is writing every song played, in order.

To get the output above, we wrote a MockMoreSongs class with a getSongs() method that returned a List that has all the same entries as this text file.

We need a Set instead of a List

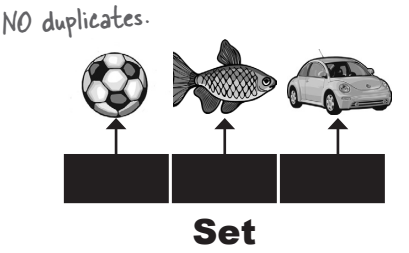
From the Collection API, we find three main interfaces, **List**, **Set**, and **Map**. ArrayList is a **List**, but it looks like ***Set*** is exactly what we need.



- **LIST - when *sequence* matters**

  Collections that know about ***index position***.

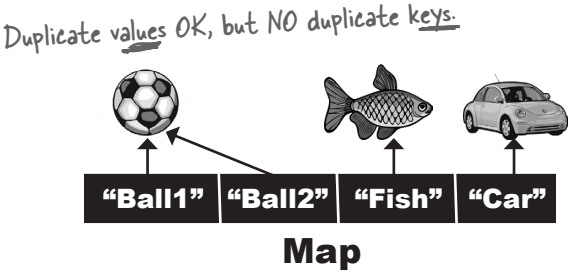
  Lists know where something is in the list. You can have more than one element referencing the same object.



- **SET - when *uniqueness* matters**

  Collections that ***do not allow duplicates***.

  Sets know whether something is already in the collection. You can never have more than one element referencing the same object (or more than one element referencing two objects that are considered equal—we'll look at what object equality means in a moment).



- **MAP - when *finding something by key* matters**

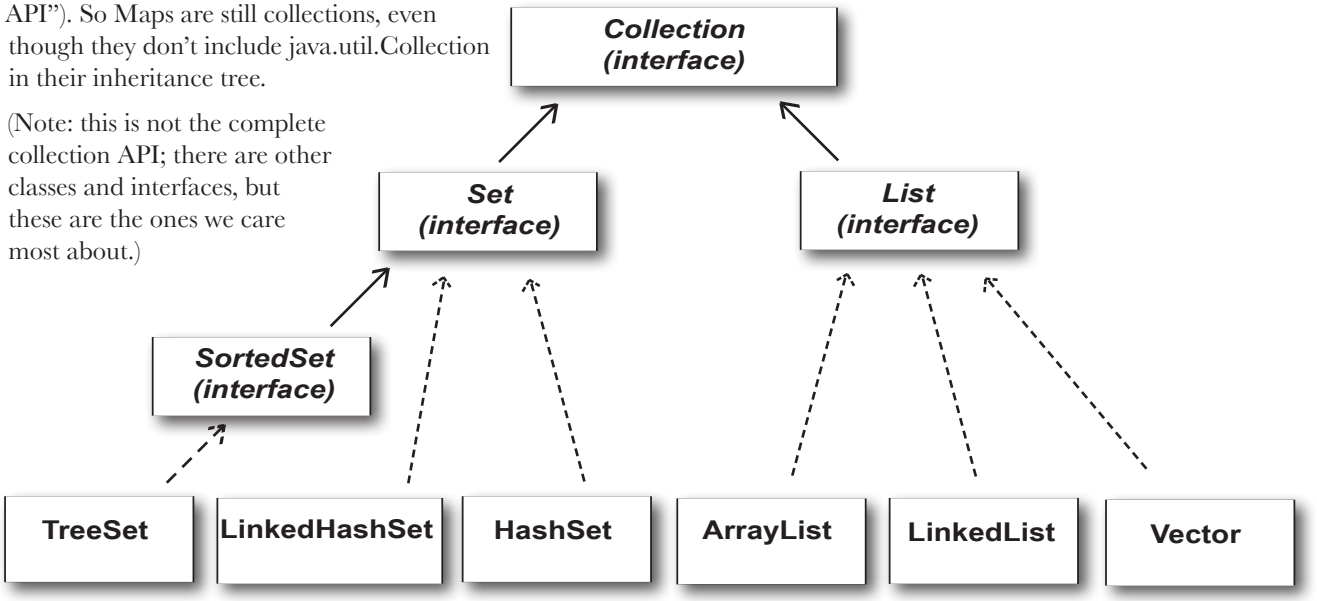
  Collections that use ***key-value pairs***.

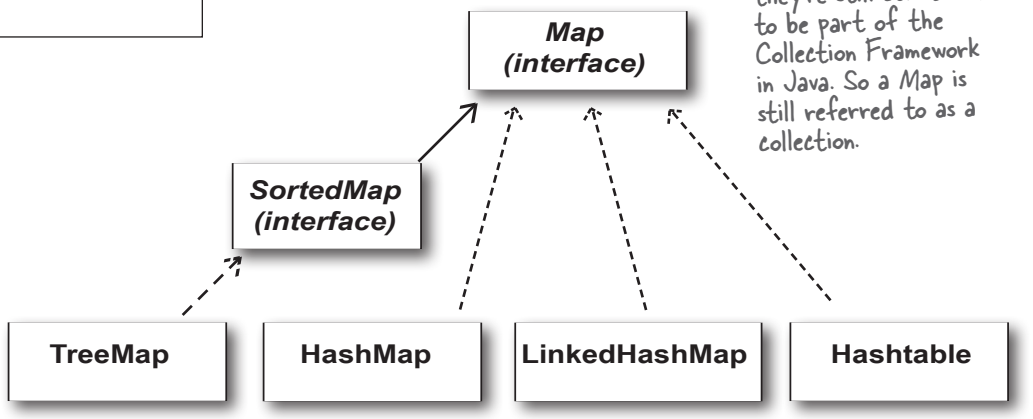
  Maps know the value associated with a given key. You can have two keys that reference the same value, but you cannot have duplicate keys. A key can be any object.

The Collection API (part of it)

Notice that the Map interface doesn't actually extend the Collection interface, but Map is still considered part of the "Collection Framework" (also known as the "Collection API"). So Maps are still collections, even though they don't include java.util.Collection in their inheritance tree.

(Note: this is not the complete collection API; there are other classes and interfaces, but these are the ones we care most about.)





Maps don't extend from java.util.Collection, but they're still considered to be part of the Collection Framework in Java. So a Map is still referred to as a collection.

Using a HashSet instead of ArrayList

We updated the Jukebox code to put the songs in a HashSet to try to eliminate our duplicate songs. (Note: we left out some of the Jukebox code, but you can copy it from earlier versions.)

```
import java.util.*;

public class Jukebox8 {
  public static void main(String[] args) {
    new Jukebox8().go();
  }

  public void go() {
    List<SongV3> songList = MockMoreSongs.getSongsV3();
    System.out.println(songList);

    songList.sort((one, two) -> one.getTitle().compareTo(two.getTitle()));
    System.out.println(songList);

    Set<SongV3> songSet = new HashSet<>(songList);
    System.out.println(songSet);
  }
}
```

We created a MockMoreSongs class to return a List of SongV3 objects that contain the same values as SongListMore.txt.

HashSet has a constructor that takes a Collection, and it will create a set with all the items from that collection.

We want the Set to hold SongV3 objects. HashSet IS-A Set, so we can store the HashSet in this Set variable.

```
File Edit Window Help GetBetterMusic
%java Jukebox8

[somersault, cassidy, $10, havana, $10, cassidy, 50 ways]

[$10, $10, 50 ways, cassidy, cassidy, havana, somersault]

[$10, 50 ways, havana, cassidy, $10, cassidy, somersault]
```

Before sorting the List.

After sorting the List (by title).

After putting it into a HashSet and printing the HashSet (we didn't call sort() again).

The Set didn't help!!

We still have all the duplicates!

(And it lost its sort order when we put the list into a HashSet, but we'll worry about that one later...)

What makes two objects equal?

To figure out why using a Set didn't remove the duplicates, we have to ask—what makes two Song references duplicates? They must be considered ***equal***. Is it simply two references to the very same object, or is it two separate objects that both have the same *title*?

This brings up a key issue: *reference* equality vs. *object* equality.

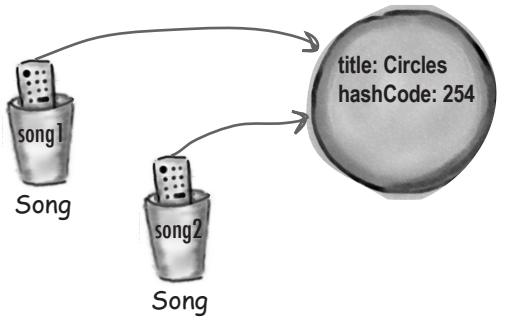
If two objects *foo* and *bar* are equal, *foo.equals(bar)* and *bar.equals(foo)* must be *true*, and both *foo* and *bar* must return the same value from *hashCode()*. For a Set to treat two objects as duplicates, you must override the hashCode() and equals() methods inherited from class Object so that you can make two different objects be viewed as equal.

▶ Reference equality

Two references, one object on the heap.

Two references that refer to the same object on the heap are equal. Period. If you call the **hashCode()** method on both references, you'll get the same result. If you don't override the hashCode() method, the default behavior (remember, you inherited this from class Object) is that each object will get a unique number (most versions of Java assign a hashcode based on the object's memory address on the heap, so no two objects will have the same hashcode).

If you want to know if two *references* are really referring to the same object, use the == operator, which (remember) compares the bits in the variables. If both references point to the same object, the bits will be identical.



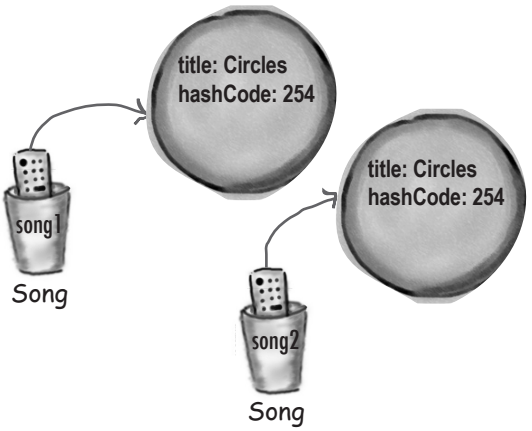

```
if (song1 == song2) {
   // both references are referring
   // to the same object on the heap
}
```

▶ Object equality

Two references, two objects on the heap, but the objects are considered *meaningfully equivalent.*

If you want to treat two different Song objects as equal (for example if you decided that two Songs are the same if they have matching *title* variables), you must override *both* the **hashCode()** and **equals()** methods inherited from class Object.

As we said above, if you *don't* override hashCode(), the default behavior (from Object) is to give each object a unique hashcode value. So you must override hashCode() to be sure that two equivalent objects return the same hashcode. But you must also override equals() so that if you call it on *either* object, passing in the other object, always returns ***true***.




```
if (song1.equals(song2) && song1.hashCode() == song2.hashCode()) {
   // both references are referring to either a
   // a single object, or to two objects that are equal
}
```

How a HashSet checks for duplicates: hashCode() and equals()

When you put an object into a HashSet, it calls the object's hashCode method to determine where to put the object in the Set. But it also compares the object's hash code to the hash code of all the other objects in the HashSet, and if there's no matching hash code, the HashSet assumes that this new object is *not* a duplicate.

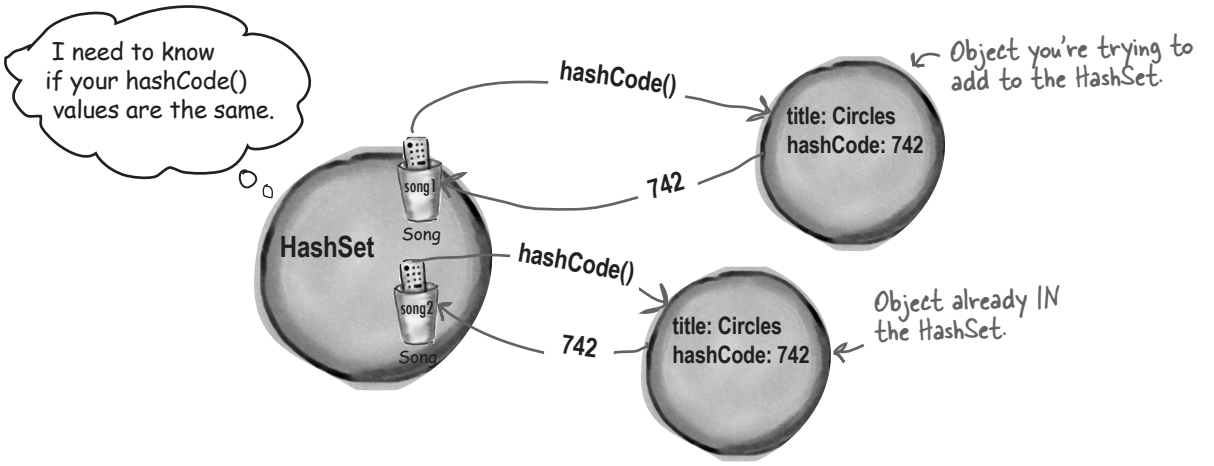
In other words, if the hash codes are different, the HashSet assumes there's no way the objects can be equal!

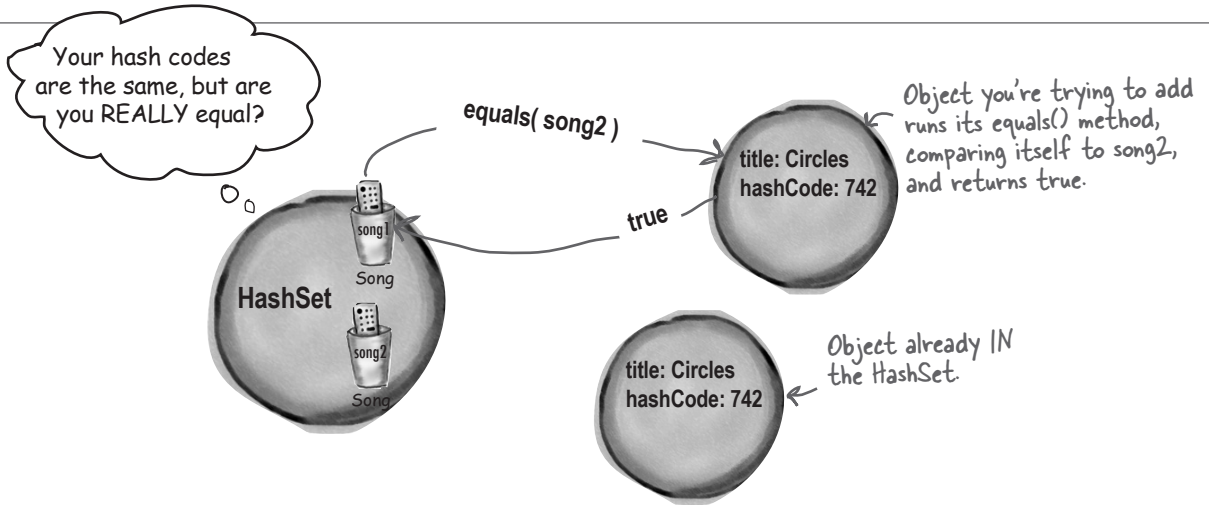
So you must override hashCode() to make sure the objects have the same value.

But two objects with the same hash code might *not* be equal (more on this on the next page), so if the HashSet finds a matching hash code for two objects—one you're inserting and one already in the set—the HashSet will then call one of the object's equals() methods to see if these hash code–matched objects really *are* equal.

And if they're equal, the HashSet knows that the object you're attempting to add is a duplicate of something in the Set, so the add doesn't happen.

You don't get an exception, but the HashSet's add() method returns a boolean to tell you (if you care) whether the new object was added. So if the add() method returns *false*, you know the new object was a duplicate of something already in the set.





The Song class with overridden hashCode() and equals()

```
class SongV4 implements Comparable<SongV4> {
  private String title;
  private String artist;
  private int bpm;

  public boolean equals(Object aSong) {
    SongV4 other = (SongV4) aSong;
    return title.equals(other.getTitle());
  }

  public int hashCode() {
    return title.hashCode();
  }

  public int compareTo(SongV4 s) {
    return title.compareTo(s.getTitle());
  }

  SongV4(String title, String artist, int bpm) {
    this.title = title;
    this.artist = artist;
    this.bpm = bpm;
  }

  public String getTitle() {
    return title;
  }

  public String getArtist() {
    return artist;
  }

  public int getBpm() {
    return bpm;
  }

  public String toString() {
    return title;
  }
}
```

The HashSet (or anyone else calling this method) sends it another Song.

The GREAT news is that title is a String, and Strings have an overridden equals() method. So all we have to do is ask one title if it's equal to the other song's title.

Same deal here...the String class has an overridden hashCode() method, so you can just return the result of calling hashCode() on the title. Notice how hashCode() and equals() are using the SAME instance variable (title).

Now it works! No duplicates when we print out the HashSet. But we didn't call sort() again, and when we put the ArrayList into the HashSet, the HashSet didn't preserve the sort order.

File Edit Window Help HashingItOut

```
%java Jukebox9

[somersault, cassidy, $10, havana, $10,
cassidy, 50 ways]

[$10, $10, 50 ways, cassidy, cassidy, havana,
somersault]

[havana, $10, 50 ways, cassidy, somersault]
```

Java Object Law for hashCode() and equals()

The API docs for class Object state the rules you MUST follow:

- **If two objects are equal, they MUST have matching hash codes.**
- **If two objects are equal, calling equals() on either object MUST return true. In other words, if (a.equals(b)) then (b.equals(a)).**
- **If two objects have the same hash code value, they are NOT required to be equal. But if they're equal, they MUST have the same hash code value.**
- **So, if you override equals(), you MUST override hashCode().**
- **The default behavior of hashCode() is to generate a unique integer for each object on the heap. So if you don't override hashCode() in a class, no two objects of that type can EVER be considered equal.**
- **The default behavior of equals() is to do an == comparison. In other words, to test whether the two references refer to a single object on the heap. So if you don't override equals() in a class, no two objects can EVER be considered equal since references to two different objects will always contain a different bit pattern.**

a.equals(b)* must also mean that *a.hashCode() == b.hashCode()

But *a.hashCode() == b.hashCode()* does NOT have to mean *a.equals(b)*

there are no Dumb Questions

Q: How come hash codes can be the same even if objects aren't equal?

A: HashSets use hash codes to store the elements in a way that makes it much faster to access. If you try to find an object in an ArrayList by giving the ArrayList a copy of the object (as opposed to an index value), the ArrayList has to start searching from the beginning, looking at each element in the list to see if it matches. But a HashSet can find an object much more quickly, because it uses the hash code as a kind of label on the "bucket" where it stored the element. So if you say, "I want you to find an object in the set that's exactly like this one...," the HashSet gets the hash code value from the copy of the Song you give it (say, 742), and then the HashSet says, "Oh, I know exactly where the object with hash code #742 is stored...," and it goes right to the #742 bucket.

This isn't the whole story you get in a computer science class, but it's enough for you to use HashSets effectively. In reality, developing a good hashing algorithm is the subject of many a PhD thesis, and more than we want to cover in this book.

The point is that hash codes can be the same without necessarily guaranteeing that the objects are equal, because the "hashing algorithm" used in the hashCode() method might happen to return the same value for multiple objects. And yes, that means that multiple objects would all land in the same hash code bucket in the HashSet, but that's not the end of the world. The HashSet might be a little less efficient, because if the HashSet finds more than one object in the same hash code bucket, it has to use the equals() on all those objects to see if there's a perfect match.

If we want the set to stay sorted, we've got TreeSet

TreeSet is similar to HashSet in that it prevents duplicates. But it also *keeps* the list sorted. It works just like the sort() method in that if you make a TreeSet without giving it a Comparator, the TreeSet uses each object's compareTo() method for the sort. But you have the option of passing a Comparator to the TreeSet constructor, to have the TreeSet use that instead.

The downside to TreeSet is that if you don't *need* sorting, you're still paying for it with a small performance hit. But you'll probably find that the hit is almost impossible to notice for most apps.

```
public class Jukebox10 {
  public static void main(String[] args) {
    new Jukebox10().go();
  }

  public void go() {
    List<SongV4> songList = MockMoreSongs.getSongsV4();
    System.out.println(songList);

    songList.sort((one, two) -> one.getTitle().compareTo(two.getTitle()));
    System.out.println(songList);

    Set<SongV4> songSet = new TreeSet<>(songList);
    System.out.println(songSet);
  }
}
```

Create a TreeSet instead of HashSet. The TreeSet will use SongV4's compareTo() method to sort the items in songList.

If we want the TreeSet to sort on something different (i.e., to NOT use SongV4's compareTo() method), we need to pass in a Comparator (or a lambda) to the TreeSet constructor. Then we'd use songSet.addAll() to add the songList values into the TreeSet.

```
Set<SongV4> songSet = new TreeSet<>((o1, o2) -> o1.getBpm() - o2.getBpm());
songSet.addAll(songList);
```

Yep, another lambda for sorting. This one sorts by BPM. Remember, this lambda implements Comparator.

What you MUST know about TreeSet...

TreeSet looks easy, but make sure you really understand what you need to do to use it. We thought it was so important that we made it an exercise so you'd *have* to think about it. Do NOT turn the page until you've done this. *We mean it.*

Answers on page 366.

Look at this code. Read it carefully, then answer the questions below. (Note: there are no syntax errors in this code.)

```
import java.util.*;

public class TestTree {
  public static void main(String[] args) {
    new TestTree().go();
  }

  public void go() {
    Book b1 = new Book("How Cats Work");
    Book b2 = new Book("Remix your Body");
    Book b3 = new Book("Finding Emo");

    Set<Book> tree = new TreeSet<>();
    tree.add(b1);
    tree.add(b2);
    tree.add(b3);
    System.out.println(tree);
  }
}

class Book {
  private String title;
  public Book(String t) {
    title = t;
  }
}
```

1. What is the result when you compile this code?

2. If it compiles, what is the result when you run the TestTree class?

3. If there is a problem (either compile-time or runtime) with this code, how would you fix it?

TreeSet elements MUST be comparable

TreeSet can't read the programmer's mind to figure out how the objects should be sorted. You have to tell the TreeSet *how*.

To use a TreeSet, one of these things must be true:

- **The elements in the list must be of a type that implements *Comparable***

The Book class on the previous page didn't implement Comparable, so it wouldn't work at runtime. Think about it, the poor TreeSet's sole purpose in life is to keep your elements sorted, and once again—it had no idea how to sort Book objects! It doesn't fail at compile-time, because the TreeSet add() method doesn't take a Comparable type. The TreeSet add() method takes whatever type you used when you created the TreeSet. In other words, if you say new TreeSet<Book>(), the add() method is essentially add(Book). And there's no requirement that the Book class implement Comparable! But it fails at runtime when you add the second element to the set. That's the first time the set tries to call one of the object's compareTo() methods and...can't.

```
class Book implements Comparable<Book> {
  private String title;
  public Book(String t) {
    title = t;
  }

  public int compareTo(Book other) {
    return title.compareTo(other.title);
  }
}
```

OR

- **You use the TreeSet's overloaded constructor that takes a *Comparator***

TreeSet works a lot like the sort() method—you have a choice of using the element's compareTo() method, assuming the element type implemented the Comparable interface, OR you can use a custom Comparator that knows how to sort the elements in the set. To use a custom Comparator, you call the TreeSet constructor that takes a Comparator.

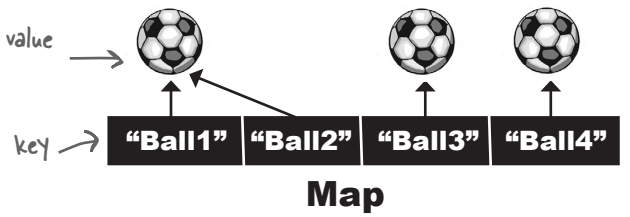
```
class BookCompare implements Comparator<Book> {
  public int compare(Book one, Book two) {
    return one.title.compareTo(two.title);
  }
}
public class TestTreeComparator {
  public void go() {
    Book b1 = new Book("How Cats Work");
    Book b2 = new Book("Remix your Body");
    Book b3 = new Book("Finding Emo");
    BookCompare bookCompare = new BookCompare();
    Set<Book> tree = new TreeSet<>(bookCompare);
    tree.add(b1);
    tree.add(b2);
    tree.add(b3);
    System.out.println(tree);
  }
}
```

You could use a lambda instead of declaring a new Comparator class.

We've seen Lists and Sets, now we'll use a Map

Lists and Sets are great, but sometimes a Map is the best collection (not Collection with a capital "C"—remember that Maps are part of Java collections but they don't implement the Collection interface).

Imagine you want a collection that acts like a property list, where you give it a name and it gives you back the value associated with that name. Keys can be any Java object (or, through autoboxing, a primitive), but you'll often see String keys (i.e., property names) or Integer keys (representing unique IDs, for example).



Each element in a Map is actually TWO objects—a *key* and a *value*. You can have duplicate *values*, but NOT duplicate *keys*.

Map example

```
public class TestMap {
  public static void main(String[] args) {
    Map<String, Integer> scores = new HashMap<>();

    scores.put("Kathy", 42);
    scores.put("Bert", 343);
    scores.put("Skyler", 420);

    System.out.println(scores);
    System.out.println(scores.get("Bert"));
  }
}
```

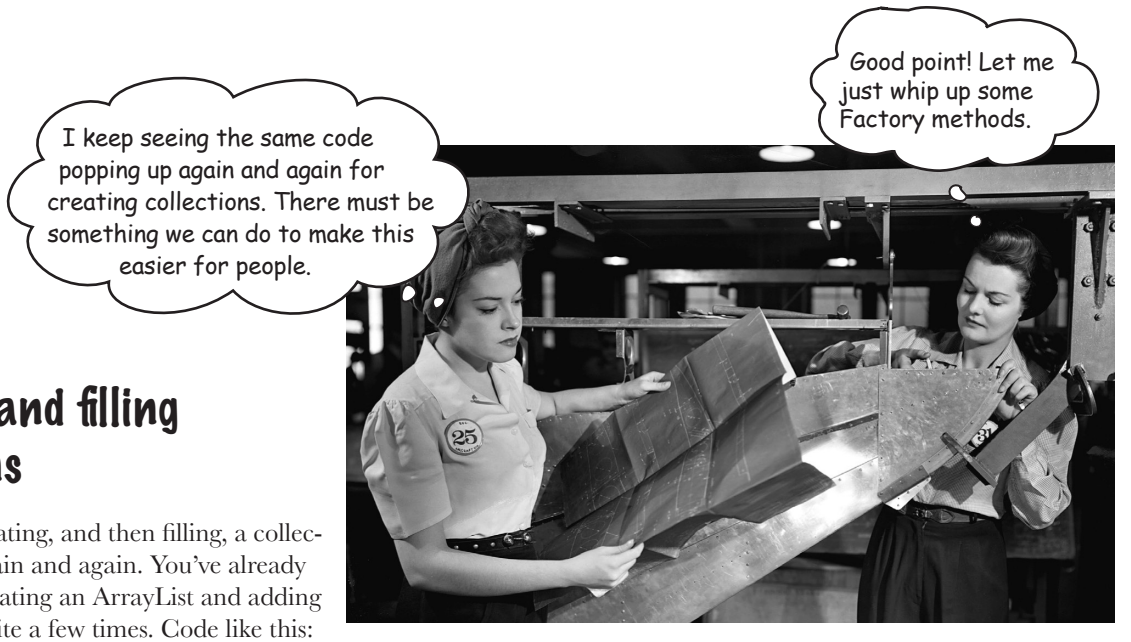
HashMap needs TWO type parameters—one for the key and one for the value.

Use put() instead of add(), and now of course it takes two arguments (key, value).

The get() method takes a key and returns the value (in this case, an Integer).

```
File Edit Window Help WhereAmI
%java TestMap

{Skyler=420, Bert=343, Kathy=42}
343
```

When you print a Map, it gives you the key=value pairs, in braces { } instead of the brackets [ ] you see when you print lists and sets.



Creating and filling collections

The code for creating, and then filling, a collection crops up again and again. You've already seen code for creating an ArrayList and adding elements to it quite a few times. Code like this:

```
List<String> songs = new ArrayList<>();
songs.add("somersault");
songs.add("cassidy");
songs.add("$10");
```

Whether you're creating a List, a Set, or a Map, it looks pretty similar. What's more, these types of collections are often ones where we know what the data is right at the start, and then we don't intend to change it at all during the lifetime of the collection. If we wanted to really make sure that no-one changed the collection after we'd created it, we'd have to add an extra step:

```
List<String> songs = new ArrayList<>();
songs.add("somersault");
songs.add("cassidy");
songs.add("$10");
return Collections.unmodifiableList(songs);
```

Return an "unmodifiable" version of the list we just created so we know no one else can change it.

We'll see in Chapters 12 and 18 why we might want to create data structures that can't be changed.

That's a lot of code! And it's a lot of code for something common that we probably want to do a lot.

Fortunately for us, Java now has "Convenience Factory Methods of Collections" (they were added in Java 9). We can use these methods to create common data structures and fill them with data, with just one method call.

Convenience Factory Methods for Collections

Convenience Factory Methods for Collections allow you to easily create a List, Set, or Map that's been prefilled with known data. There are a couple of things to understand about using them:

1. **The resulting collections cannot be changed**. You can't add to them or alter the values; in fact, you can't even do the sorting that we've seen in this chapter.

2. **The resulting collections are not the standard Collections we've seen**. These are not ArrayList, HashSet, HashMap, etc. You can rely on them to behave according to their interface: a List will always preserve the order in which the elements were placed; a Set will never have duplicates. But you can't rely on them being a specific implementation of List, Set, or Map.

Convenience Factory Methods are just that—a convenience that will work for most of the cases where you want to create a collection prefilled with data. And for those cases where these factory methods don't suit you, you can still use the Collections constructors and add() or put() methods instead.

▶ Creating a List: `List.of()`

To create the list of Strings from the last page, we don't need five lines of code; we just need one:

```
List<String> strings = List.of("somersault", "cassidy", "$10");
```

If you want to add Song objects instead of simple Strings, it's still short and descriptive:

```
List<SongV4> songs = List.of(new SongV4("somersault", "zero 7", 147),
                             new SongV4("cassidy", "grateful dead", 158),
                             new SongV4("$10", "hitchhiker", 140));
```

▶ Creating a Set: `Set.of()`

Creating a Set uses very similar syntax:

```
Set<Book> books = Set.of(new Book("How Cats Work"),
                         new Book("Remix your Body"),
                         new Book("Finding Emo"));
```

▶ Creating a Map: `Map.of()`, `Map.ofEntries()`

Maps are different, because they take two objects for each "entry"—a key and a value. If you want to put less than 10 entries into your Map, you can use Map.of, passing in key, value, key, value, etc.:

```
Map<String, Integer> scores = Map.of("Kathy", 42,
                                     "Bert", 343,
                                     "Skyler", 420);
```

If you have more than 10 entries, or if you want to be clearer about how your keys are paired up to their values, you can use Map.ofEntries instead:

```
Map<String, String> stores = Map.ofEntries(Map.entry("Riley", "Supersports"),
                                           Map.entry("Brooklyn", "Camera World"),
                                           Map.entry("Jay", "Homecase"));
```

To make the line shorter, you can use a *static import* on Map.entry (we talked about static imports in Chapter 10).

Finally, back to generics

Remember earlier in the chapter we talked about how methods that take arguments with generic types can be...*weird*. And we mean weird in the polymorphic sense. If things start to feel strange here, just keep going—it takes a few pages to really tell the whole story. The examples are going to use a class hierarchy of Animals.

```
abstract class Animal {
  void eat() {
    System.out.println("animal eating");
  }
}
class Dog extends Animal {
  void bark() { }
}
class Cat extends Animal {
  void meow() { }
}
```

The simplified Animal class hierarchy

Using polymorphic arguments and generics

Generics can be a little...counterintuitive when it comes to using polymorphism with a generic type (the class inside the angle brackets). Let's create a method that takes a List<Animal> and use this to experiment.

Passing in List<Animal>

```
public class TestGenerics1 {
  public static void main(String[] args) {

    List<Animal> animals = List.of(new Dog(), new Cat(), new Dog());
    takeAnimals(animals);
  }

  public static void takeAnimals(List<Animal> animals) {
    for (Animal a : animals) {
      a.eat();
    }
  }
}
```

Using the List.of factory method we just looked at

Pass a List<Animal> into our testAnimals method

Method that has a generic class (List) as a parameter

Remember, we can call ONLY the methods declared in type Animal, since the animals parameter is of type List<Animal>.

Compiles and runs just fine

```
File Edit Window Help CatFoodIsBetter
%java TestGenerics1

animal eating
animal eating
animal eating
```

But will it work with List<Dog>?

A List<Animal> argument can be passed to a method with a List<Animal> parameter. So the big question is, will the List<Animal> parameter accept a List<Dog>? Isn't that what polymorphism is for?

Passing in List<Dog>

```
public void go() {
  List<Animal> animals = List.of(new Dog(), new Cat(), new Dog());
  takeAnimals(animals);
```

We know this line worked fine.

```
  List<Dog> dogs = List.of(new Dog(), new Dog());
  takeAnimals(dogs);
}
```

Make a Dog List and put a couple dogs in.

Will this work now that we changed from an array to a List?

```
public void takeAnimals(List<Animal> animals) {
  for (Animal a : animals) {
    a.eat();
  }
}
```

When we compile it:

File Edit Window Help CatsAreSmarter

```
%javac TestGenerics2.java

TestGenerics2.java:20: error: incompatible types:
List<Dog> cannot be converted to List<Animal>
    takeAnimals(dogs);
                ^
1 error
```

It looked so right, but went so wrong...

And I'm supposed to be OK with this? That totally screws my animal simulation where the veterinary program takes a list of any type of animal so that a dog kennel can send a list of dogs, and a cat kennel can send a list of cats...now you're saying I can't do that?

What could happen if it *were* allowed...?

Imagine the compiler let you get away with that. It let you pass a List<Dog> to a method declared as:

```
public void takeAnimals(List<Animal> animals) {
  for (Animal a : animals) {
    a.eat();
  }
}
```

There's nothing in that method that *looks* harmful, right? After all, the whole point of polymorphism is that anything an Animal can do (in this case, the eat() method), a Dog can do as well. So what's the problem with having the method call eat() on each of the Dog references?

There's nothing wrong with *that* code. But imagine *this* code instead:

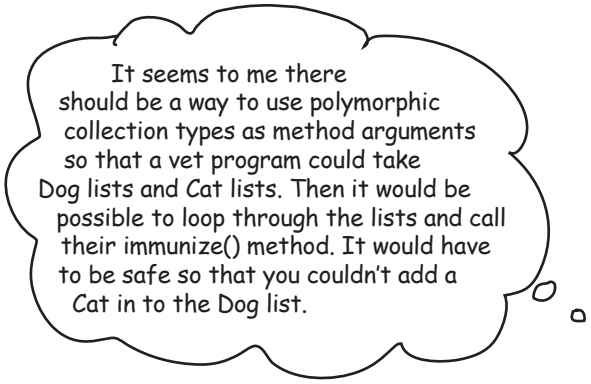
```
public void takeAnimals(List<Animal> animals) {
   animals.add(new Cat());
}
```

Yikes!! We just stuck a Cat in what might be a Dogs-only List.

So that's the problem. There's certainly nothing wrong with adding a Cat to a List<Animal>, and that's the whole point of having a List of a supertype like Animal—so that you can put all types of animals in a single Animal List.

But if you passed a Dog List—one meant to hold ONLY Dogs—to this method that takes an Animal List, then suddenly you'd end up with a Cat in the Dog list. The compiler knows that if it lets you pass a Dog List into the method like that, someone could, at runtime, add a Cat to your Dog list. So instead, the compiler just won't let you take the risk.

If you declare a method to take List<Animal>, it can take ONLY a List<Animal>, not List<Dog> or List<Cat>.



We can do this with wildcards

It looks unusual, but there *is* a way to create a method argument that can accept a List of any Animal subtype. The simplest way is to use a **wildcard**.

```
public void takeAnimals(List<? extends Animal> animals) {
  for (Animal a : animals) {
    a.eat();
  }
}
```

Remember, the keyword "extends" here means either extends OR implements.

So now you're wondering, "What's the *difference*? Don't you have the same problem as before?"

And you'd be right for wondering. The answer is NO. When you use the wildcard <?> in your declaration, the compiler won't let you do anything that adds to the list!

When you use a wildcard in your method argument, the compiler will STOP you from doing anything that could hurt the list referenced by the method parameter.

You can still call methods on the elements in the list, but you cannot add elements to the list.

In other words, you can do things *with* the list elements, but you can't put *new* things in the list.

there are no Dumb Questions

Q: **Back when we first saw generic methods, there was a similar-looking method that declared the generic type in front of the method name. Does that do the same thing as this takeAnimals method?**

A: Well spotted! Back at the start of the chapter, there was a method like this:

```
<T extends Animal> void takeThing(List<T> list)
```

We actually could use this syntax to achieve a similar thing, but it works in a slightly different way. Yes, you can pass List<Animal> and List<Dog> into the method, but you get the added benefit of being able to use the generic type, T, elsewhere too.

Using the method's generic type parameter

What can we do if we define our method like this instead?

```
public <T extends Animal> void takeAnimals(List<T> list) { }
```

Well, not much as the method stands right now, we don't need to use "T" for anything. But if we made a change to our method to return a List, for example of all the animals we had successfully vaccinated, we can declare that the List that's returned has the same generic type as the List that's passed in:

```
public <T extends Animal> List<T> takeAnimals(List<T> list) { }
```

When you call the method, you know you're going to get the same type back as you put in.

```
List<Dog> dogs = List.of(new Dog(), new Dog());
List<Dog> vaccinatedDogs = takeAnimals(dogs);

List<Animal> animals = List.of(new Dog(), new Cat());
List<Animal> vaccinatedAnimals = takeAnimals(animals);
```

The List we get back from the takeAnimals method is always the same type as the list we pass in.

If the method used the wildcard for both method parameter and return type, there's nothing to guarantee they're the same type. In fact, anything calling the method has almost no idea what's going to be in the collection, other than "some sort of animal."

```
public void go() {
  List<Dog> dogs = List.of(new Dog(), new Dog());
  List<? extends Animal> vaccinatedSomethings = takeAnimals(dogs);
}

public List<? extends Animal> takeAnimals(List<? extends Animal> animals) { }
```

Using the wildcard ("? extends") is fine when you don't care much about the generic type, you just want to allow all subtypes of some type.

Using a type parameter ("T") is more helpful when you want to do more with the type itself, for example in the method's return.