Let's play

Here's what happens when we run it and enter the numbers 1,2,3,4,5,6. Lookin' good.

A complete game interaction
(your mileage may vary)

```
File Edit Window Help Smile
%java SimpleStartupGame
enter a number  1
miss
enter a number  2
miss
enter a number  3
miss
enter a number  4
hit
enter a number  5
hit
enter a number  6
kill
You took 6 guesses
```

What's this? A *bug*?

Gasp!

Here's what happens when we enter 1,1,1.

A different game interaction
(yikes)

```
File Edit Window Help Faint
%java SimpleStartupGame
enter a number  1
hit
enter a number  1
hit
enter a number  1
kill
You took 3 guesses
```

Sharpen your pencil

It's a cliff-hanger!

Will we ***find*** the bug?

Will we ***fix*** the bug?

Stay tuned for the next chapter, where we answer these questions and more...

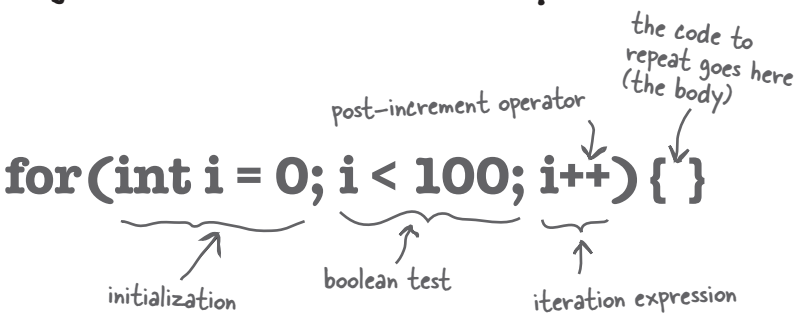
And in the meantime, see if you can come up with ideas for what went wrong and how to fix it.

→ Yours to solve.

More about for loops

We've covered all the game code for *this* chapter (but we'll pick it up again to finish the deluxe version of the game in the next chapter). We didn't want to interrupt your work with some of the details and background info, so we put it back here. We'll start with the details of for loops, and if you've seen this kind of syntax in another programming language, just skim these last few pages...

Regular (non-enhanced) for loops



What it means in plain English: "Repeat 100 times."

How the compiler sees it:

- create a variable *i* and set it to 0.
- repeat while *i* is less than 100.
- at the end of each loop iteration, add 1 to *i*.

Part One: *initialization*

Use this part to declare and initialize a variable to use within the loop body. You'll most often use this variable as a counter. You can actually initialize more than one variable here, but it's much more common to use a single variable.

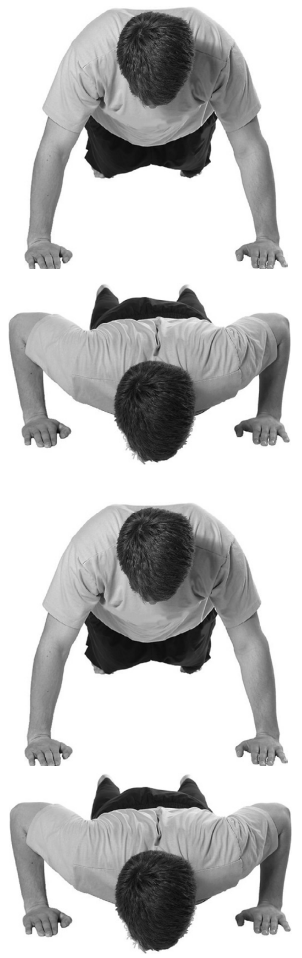
Part Two: *boolean test*

This is where the conditional test goes. Whatever's in there, it *must* resolve to a boolean value (you know, ***true*** or ***false***). You can have a test, like (x >= 4), or you can even invoke a method that returns a boolean.

Part Three: *iteration expression*

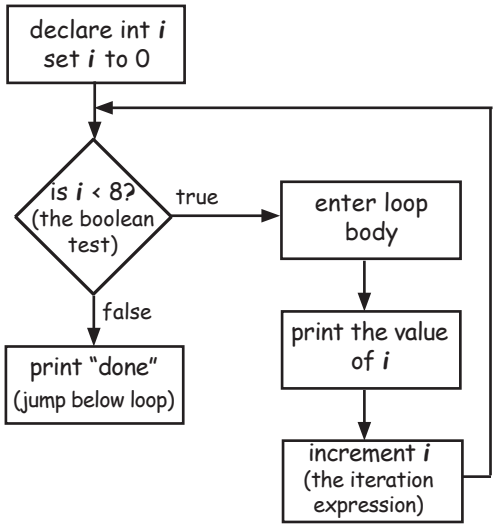
In this part, put one or more things you want to happen with each trip through the loop. Keep in mind that this stuff happens at the *end* of each loop.

repeat for 100 reps:



Trips through a loop

```
for (int i = 0; i < 8; i++) {
   System.out.println(i);
}
System.out.println("done");
```



output:

```
File Edit Window Help Repeat
%java Test
0
1
2
3
4
5
6
7
done
```

Difference between for and while

A *while* loop has only the boolean test; it doesn't have a built-in initialization or iteration expression. A *while* loop is good when you don't know how many times to loop and just want to keep going while some condition is true. But if you *know* how many times to loop (e.g., the length of an array, 7 times, etc.), a *for* loop is cleaner. Here's the loop above rewritten using *while:*

```
int i = 0;          <- we have to declare and initialize the counter
while (i < 8) {
   System.out.println(i);
   i++;             <- we have to increment the counter
}
System.out.println("done");
```

++ --

Pre and Post Increment/Decrement Operator

The shortcut for adding or subtracting 1 from a variable:

```
x++;
```

is the same as:

```
x = x + 1;
```

They both mean the same thing in this context:

"add 1 to the current value of x" or "***increment*** x by 1"

And:

```
x--;
```

is the same as:

```
x = x - 1;
```

Of course that's never the whole story. The placement of the operator (either before or after the variable) can affect the result. Putting the operator *before* the variable (for example, ++x), means, "*first*, increment x by 1, and *then* use this new value of x." This only matters when the ++x is part of some larger expression rather than just a single statement.

```
int x = 0;      int z = ++x;
```

produces: x is 1, z is 1

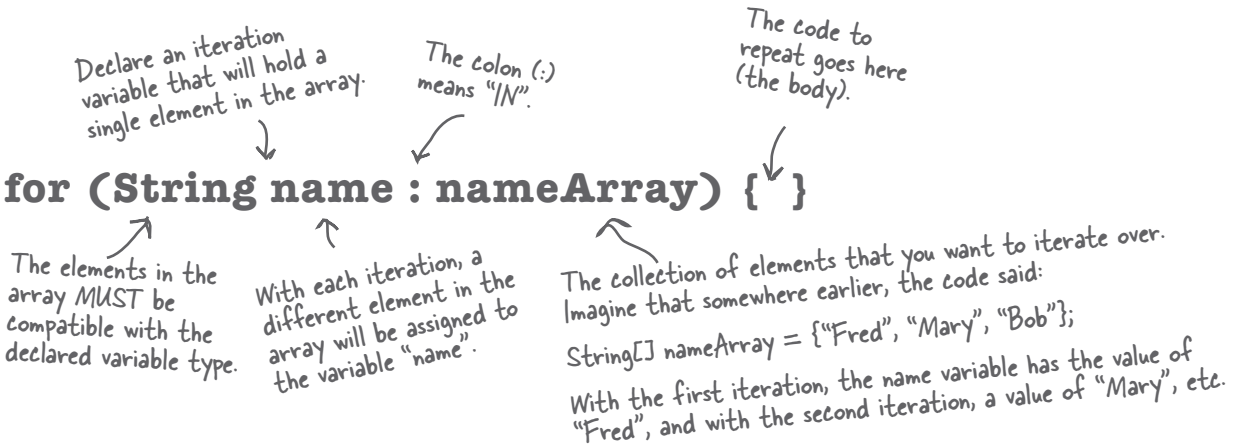
But putting the ++ *after* the x gives you a different result:

```
int x = 0;      int z = x++;
```

Once this code has run, x is 1, but ***z is 0***! z gets the value of x, and *then* x is incremented.

The enhanced for loop

The Java language added a second kind of *for* loop called the *enhanced for* back in Java 5. This makes it easier to iterate over all the elements in an array or other kinds of collections (you'll learn about *other* collections in the next chapter). That's really all that the enhanced for gives you—a simpler way to walk through all the elements in the collection. We'll see the enhanced for loop in the next chapter too, when we talk about collections that *aren't* arrays.



What it means in plain English: "For each element in nameArray, assign the element to the 'name' variable, and run the body of the loop."

How the compiler sees it:

- Create a String variable called *name* and set it to null.
- Assign the first value in *nameArray* to name.
- Run the body of the loop (the code block bounded by curly braces).
- Assign the next value in *nameArray* to name.
- Repeat while *there are still elements in the array.*

Note: depending on the programming language they've used in the past, some people refer to the enhanced for as the "for each" or the "for in" loop, because that's how it reads: "for EACH thing IN the collection..."

Part One: *iteration variable declaration*

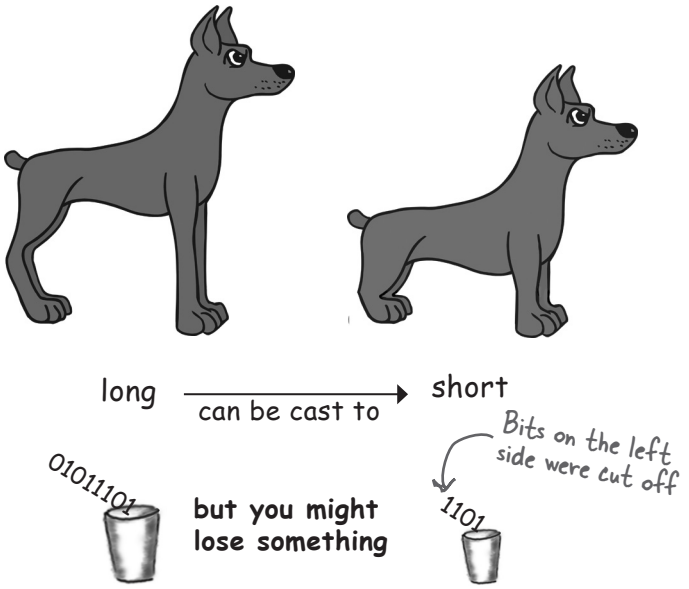
Use this part to declare and initialize a variable to use within the loop body. With each iteration of the loop, this variable will hold a different element from the collection. The type of this variable must be compatible with the elements in the array! For example, you can't declare an *int* iteration variable to use with a *String[]* array.

Part Two: *the actual collection*

This must be a reference to an array or other collection. Again, don't worry about the *other* non-array kinds of collections yet—you'll see them in the next chapter.

Casting primitives

Before we finish the chapter, we want to tie up a loose end. When we used Math.random(), we had to *cast* the result to an int. Casting one numeric type to another can change the value itself. It's important to understand the rules so you're not surprised by this.



In Chapter 3, *Know Your Variables*, we talked about the sizes of the various primitives and how you can't shove a big thing directly into a small thing:

```
long y = 42;
int x = y;       // won't compile
```

A long is bigger than an int, and the compiler can't be sure where that long has been. It might have been out partying with the other longs, and taking on really big values. To force the compiler to jam the value of a bigger primitive variable into a smaller one, you can use the cast operator. It looks like this:

```
long y = 42;     // so far so good
int x = (int) y; // x = 42 cool!
```

Putting in the cast tells the compiler to take the value of y, chop it down to int size, and set x equal to whatever is left. If the value of y was bigger than the maximum value of x, then what's left will be a weird (but calculable*) number:

```
long y = 40002;       // 40002 exceeds the 16-bit limit of a short
short x = (short) y;  // x now equals -25534!
```

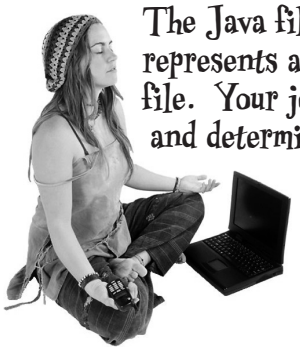
Still, the point is that the compiler lets you do it. And let's say you have a floating-point number and you just want to get at the whole number (int) part of it:

```
float f = 3.14f;
int x = (int) f;   //  x will equal 3
```

And don't even think about casting anything to a boolean or vice versa—just walk away.

**It involves sign bits, binary, "two's complement," and other geekery.*

BE the JVM



The Java file on this page represents a complete source file. Your job is to play JVM and determine what would be the output when the program runs.

```
class Output {
  public static void main(String[] args) {
    Output output = new Output();
    output.go();
  }

  void go() {
    int value = 7;
    for (int i = 1; i < 8; i++) {
      value++;
      if (i > 4) {
        System.out.print(++value + " ");
      }
      if (value > 14) {
        System.out.println(" i = " + i);
        break;
      }
    }
  }
}
```

File Edit Window Help OM

```
% java Output
12 14
```

-or-

File Edit Window Help Incense

```
% java Output
12 14 x = 6
```

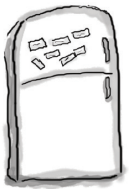
-or-

File Edit Window Help Believe

```
% java Output
13 15 x = 6
```

Answers on page 122.

Code Magnets



A working Java program is all scrambled up on the fridge. Can you reconstruct the code snippets to make a working Java program that produces the output listed below? Some of the curly braces fell on the floor and they were too small to pick up, so feel free to add as many of those as you need!

```
i++;
```

```
if (i == 1) {
```

```
System.out.println(i + " " + j);
```

```
class MultiFor {
```

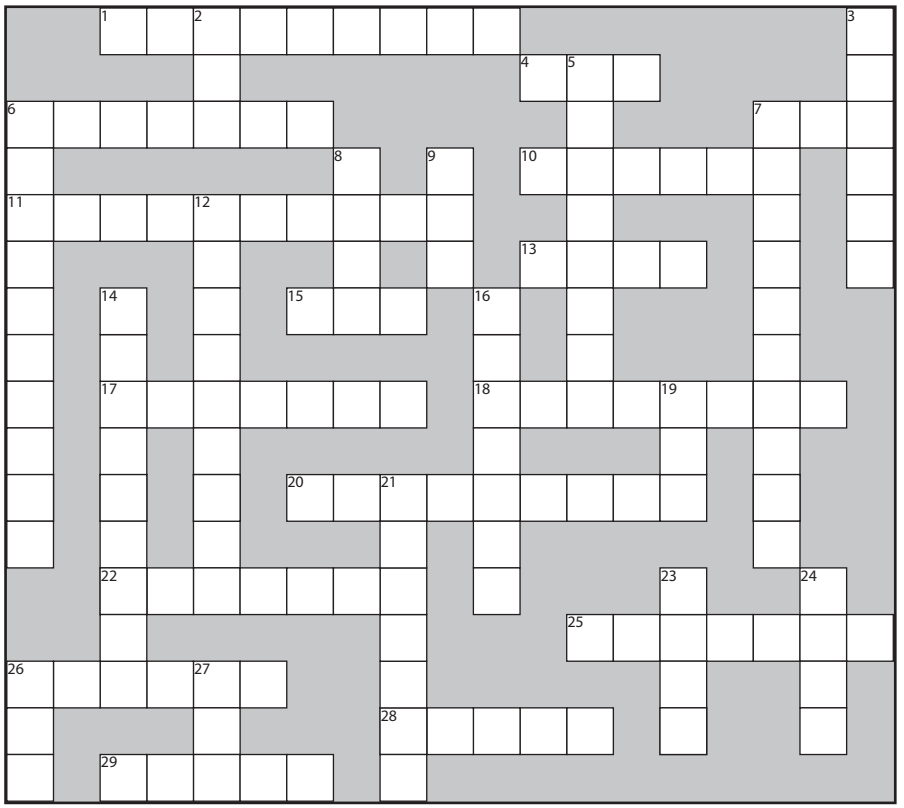
```
for(int  j = 4; j > 2; j--) {
```

```
for(int i = 0; i < 4; i++) {
```

```
public static void main(String[] args) {
```

```
File  Edit  Window  Help  Raid

% java MultiFor
0 4
0 3
1 4
1 3
3 4
3 3
```

Answers on page 122.



JavaCross

How does a crossword puzzle help you learn Java? Well, all of the words **are** Java related. In addition, the clues provide metaphors, puns, and the like. These mental twists and turns burn alternate routes to Java knowledge right into your brain!

Across

1. Fancy computer word for build
4. Multipart loop
6. Test first
7. 32 bits
10. Method's answer
11. Prep code-esque
13. Change
15. The big toolkit
17. An array unit
18. Instance or local
20. Automatic toolkit
22. Looks like a primitive, but..
25. Un-castable
26. Math method
28. Iterate over me
29. Leave early

Down

2. Increment type
3. Class's workhorse
5. Pre is a type of _____
6. For's iteration _____
7. Establish first value
8. While or For
9. Update an instance variable
12. Toward blastoff
14. A cycle
16. Talkative package
19. Method messenger (abbrev.)
21. As if
23. Add after
24. Pi house
26. Compile it and ____
27. ++ quantity

Answers on page 123.

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.

Answers on page 123.

```
public static void main(String[] args) {
  int x = 0;
  int y = 30;
  for (int outer = 0; outer < 3; outer++) {
    for (int inner = 4; inner > 1; inner--) {
      [                    ]   <- Candidate code goes here
      y = y - 2;
      if (x == 6) {
        break;
      }
      x = x + 3;
    }
    y = y - 2;
  }
  System.out.println(x + " " + y);
}
```

Candidates:

```
x = x + 3;
```

```
x = x + 6;
```

```
x = x + 2;
```

```
x++;
```

```
x--;
```

```
x = x + 0;
```

Possible output:

```
45 6
```

```
36 6
```

```
54 6
```

```
60 10
```

```
18 6
```

```
6 14
```

```
12 14
```

Match each candidate with one of the possible outputs

Exercise Solutions

Be the JVM (from page 118)

```
class Output {

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

  void go() {
    int value = 7;
    for (int i = 1; i < 8; i++) {
      value++;
      if (i > 4) {
        System.out.print(++value + " ");
      }
      if (value > 14) {
        System.out.println(" i = " + i);
        break;
      }
    }
  }
}
```

Did you remember to factor in the break statement? How did that affect the output?

```
File  Edit  Window  Help  MotorcycleMaintenance
% java Output
13 15 x = 6
```

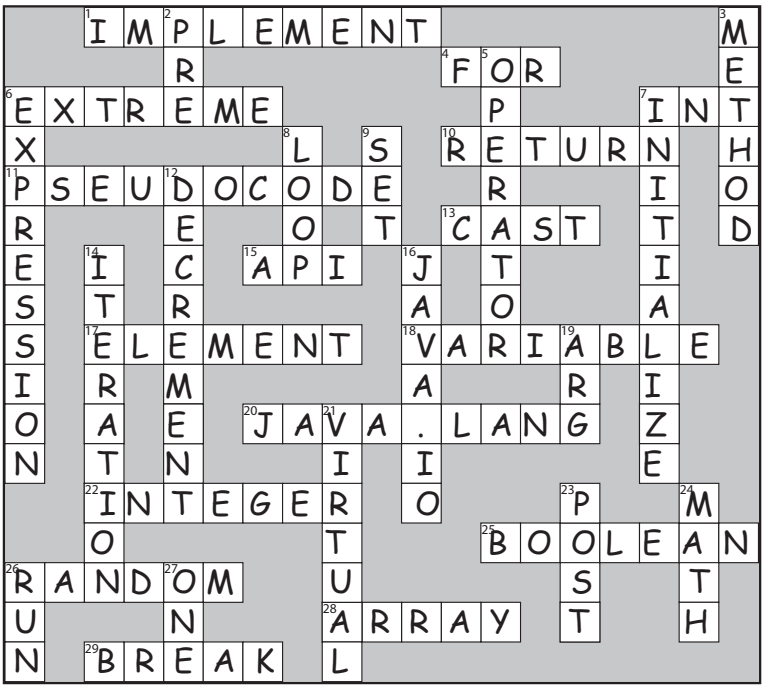
Code Magnets (from page 119)

```
class MultiFor {

  public static void main(String[] args) {
    for (int i = 0; i < 4; i++) {

      for (int j = 4; j > 2; j--) {
        System.out.println(i + " " + j);
      }

      if (i == 1) {
        i++;
      }
    }
  }
}
```

What would happen if this code block came before the 'j' for loop?

```
File  Edit  Window  Help  Monopole
% java MultiFor
0 4
0 3
1 4
1 3
3 4
3 3
```

Puzzle Solutions



JavaCross
(from page 120)

Mixed Messages
(from page 121)

Candidates:

```
x = x + 3;
x = x + 6;
x = x + 2;
x++;
x--;
x = x + 0;
```

Possible output:

```
45 6
36 6
54 6
60 10
18 6
6 14
12 14
```

Using the Java Library

Java ships with hundreds of prebuilt classes. You don't have to reinvent the wheel if you know how to find what you need in the Java library, known as the **Java API**. *You've got better things to do*. If you're going to write code, you might as well write *only* the parts that are truly custom for your application. You know those programmers who walk out the door each night at 5 PM? The ones who don't even show *up* until 10 AM? **They use the Java API.** And about eight pages from now, so will you. The core Java library is a giant pile of classes just waiting for you to use like building blocks, to assemble your own program out of largely prebuilt code. The Ready-Bake Java we use in this book is code you don't have to create from scratch, but you still have to type it. The Java API is full of code you don't even have to *type*. All you need to do is learn to use it.

In our last chapter, we left you with the cliff-hanger: a bug

How it's supposed to look

Here's what happens when we run it and enter the numbers 1,2,3,4,5,6. Lookin' good.

A complete game interaction
(your mileage may vary)

```
File  Edit  Window  Help  Smile
%java SimpleStartupGame
enter a number  1
miss
enter a number  2
miss
enter a number  3
miss
enter a number  4
hit
enter a number  5
hit
enter a number  6
kill
You took 6 guesses
```

How the bug looks

Here's what happens when we enter 2,2,2.

A different game interaction
(yikes)

```
File  Edit  Window  Help  Faint
%java SimpleStartupGame
enter a number  2
hit
enter a number  2
hit
enter a number  2
kill
You took 3 guesses
```

In the current version, once you get a hit, you can simply repeat that hit two more times for the kill!

So what happened?

```
public String checkYourself(int guess) {

    String result = "miss";

    for (int cell : locationCells) {

      if (guess == cell) {

        result = "hit";

        numOfHits++;

        break;

      } // end if

    } // end for

    if (numOfHits == locationCells.length) {

      result = "kill";

    } // end if

    System.out.println(result);

    return result;

} // end method
```

Make a variable to hold the result we'll return. Put "miss" in as the default (i.e., we assume a "miss").

Repeat with each thing in the array.

Compare the user guess to this element (cell), in the array.

we got a hit!

Get out of the loop; no need to test the other cells.

We're out of the loop, but let's see if we're now 'dead' (hit 3 times) and change the result String to "kill".

Display the result for the user ("miss" unless it was changed to "hit" or "kill").

Return the result back to the calling method.

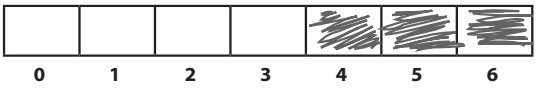
Here's where it goes wrong. We counted a hit every time the user guessed a cell location, *even if that location had already been hit!*

We need a way to know that when a user makes a hit, they haven't previously hit that cell. If they have, then we don't want to count it as a hit.

How do we fix it?

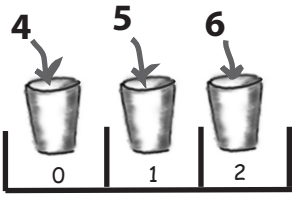
We need a way to know whether a cell has already been hit. Let's run through some possibilities, but first, we'll look at what we know so far...

We have a virtual row of seven cells, and a Startup will occupy three consecutive cells somewhere in that row. This virtual row shows a Startup placed at cell locations 4, 5, and 6.



The virtual row, with the 3 cell locations for the Startup object.

The Startup has an instance variable—an int array—that holds that Startup object's cell locations.



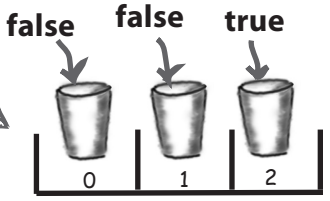
locationCells
(instance variable of the Startup)

The array instance variable that holds the Startup's cell locations. This Startup holds the three values of 4, 5, and 6. Those are the numbers the user needs to guess.

① Option one

We could make a second array, and each time the user makes a hit, we store that hit in the second array, and then check that array each time we get a hit, to see if that cell has been hit before.

A 'true' in a particular index in the array means that the cell location at that same index in the OTHER array (locationCells) has been hit.



hitCells array
(this would be a new boolean array instance variable of the Startup)

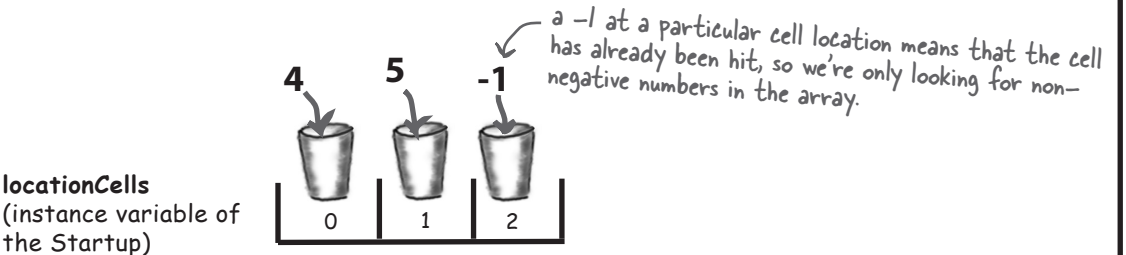
This array holds three values representing the 'state' of each cell in the Startup's location cells array. For example, if the cell at index 2 is hit, then set index 2 in the "hitCells" array to 'true'.

Option one is too clunky

Option one seems like more work than you'd expect. It means that each time the user makes a hit, you have to change the state of the *second* array (the hitCells array), oh—but first you have to CHECK the hitCells array to see if that cell has already been hit anyway. It would work, but there's got to be something better...

② Option two

We could just keep the one original array but change the value of any hit cells to -1. That way, we only have ONE array to check and manipulate.



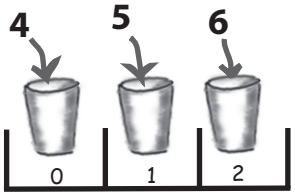
Option two is a little better, but still pretty clunky

Option two is a little less clunky than option one, but it's not very efficient. You'd still have to loop through all three slots (index positions) in the array, even if one or more are already invalid because they've been "hit" (and have a -1 value). There has to be something better...

③ Option three

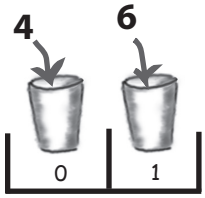
We delete each cell location as it gets hit and then modify the array to be smaller. Except arrays can't change their size, so we have to make a **new** array and copy the remaining cells from the old array into the new smaller array.

locationCells array BEFORE any cells have been hit



The array starts out with a size of 3, and we loop through all 3 cells (positions in the array) to look for a match between the user guess and the cell value (4,5, 6).

locationCells array AFTER cell '5', which was at index 1 in the array, has been hit



When cell '5' is hit, we make a new, smaller array with only the remaining cell locations, and assign it to the original locationCells reference.

Option three would be much better if the array could shrink so that we wouldn't have to make a new smaller array, copy the remaining values in, and reassign the reference.

The original prep code for part of the checkYourself() method:

REPEAT with each of the location cells in the *int* array
  // COMPARE the user guess to the location cell
  IF the user guess matches
    INCREMENT the number of hits
    // FIND OUT if it was the last location cell:
    IF number of hits is 3, **RETURN** "kill"
    ELSE it was not a kill, so **RETURN** "hit"
    END IF
  ELSE user guess did not match, so **RETURN** "miss"
  END IF
END REPEAT

Life would be good if only we could change it to:

REPEAT with each of the ***remaining*** location cells
  // COMPARE the user guess to the location cell
  IF the user guess matches
    REMOVE this cell from the array
    // FIND OUT if it was the last location cell:
    IF the array is now empty, **RETURN** "kill"
    ELSE it was not a kill, so **RETURN** "hit"
    END IF
  ELSE user guess did not match, so **RETURN** "miss"
  END IF
END REPEAT


If only I could find an array that could **shrink** when you **remove** something. And one that you didn't have to loop through to check each element, but instead you could just **ask it if it contains** what you're looking for. And it would let you **get** things out of it, without having to know exactly which slot the things are in. That would be dreamy. But I know it's just a fantasy...

Wake up and smell the library

As if by magic, there really *is* such a thing.

But it's not an *array*, it's an *ArrayList*.

A class in the core Java library (the API).

The Java Platform, Standard Edition (Java SE) ships with hundreds of pre-built classes. Just like our Ready-Bake Code. Except that these built-in classes are already compiled.

That means no typing.

Just use 'em.

ArrayList is one of a gazillion classes in the Java library.

You can use it in your code as if you wrote it yourself.

ArrayList

add(E e)
Appends the specified element to the end of this list.

remove(int index)
Removes the element at the specified position.

remove(Object o)
Removes the first occurrence of the specified element.

contains(Object o)
Returns true if this list contains the specified element.

isEmpty()
Returns "true" if the list contains no elements.

indexOf(Object o)
Returns either the first index of the element, or -1.

size()
Returns the number of elements in this list.

get(int index)
Returns the element at the specified position.

This is just a sample of SOME of the methods in ArrayList.

(Note: the add(E e) method looks a little strange...we'll get to this in Chapter 11. For now, just think of it as an add() method that takes the object you want to add.)

Some things you can do with ArrayList

1. Make one

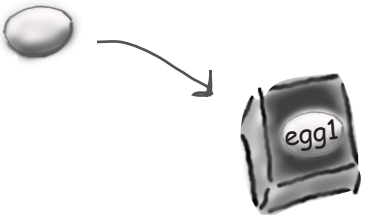
Don't worry about this new <Egg> angle-bracket syntax right now; it just means "make this a list of Egg objects."

```
ArrayList<Egg> myList = new ArrayList<Egg>();
```

A new ArrayList object is created on the heap. It's little because it's empty.

2. Put something in it

```
Egg egg1 = new Egg();

myList.add(egg1);
```



Now the ArrayList grows a "box" to hold the Egg object.

3. Put another thing in it

```
Egg egg2 = new Egg();

myList.add(egg2);
```

The ArrayList grows again to hold the second Egg object.

4. Find out how many things are in it

```
int theSize = myList.size();
```

The ArrayList is holding 2 objects so the size() method returns 2.

5. Find out if it contains something

```
boolean isIn = myList.contains(egg1);
```

The ArrayList DOES contain the Egg object referenced by 'egg1', so contains() returns **true**.

6. Find out where something is (i.e., its index)

```
int idx = myList.indexOf(egg2);
```

ArrayList is zero-based (means first index is 0) and since the object referenced by 'egg2' was the second thing in the list, indexOf() returns 1.

7. Find out if it's empty

```
boolean empty = myList.isEmpty();
```

it's definitely NOT empty, so isEmpty() returns **false**.

8. Remove something from it

```
myList.remove(egg1);
```

Hey look — it shrank!

Fill in the rest of the table below by looking at the ArrayList code on the left and putting in what you think the code might be if it were using a regular array instead. We don't expect you to get all of them exactly right, so just make your best guess.

ArrayList	Regular array
`ArrayList<String> myList = new ArrayList<String>();`	String [] myList = new String[2];
`String a = "whoohoo";`	String a = "whoohoo";
`myList.add(a);`	
`String b = "Frog";`	String b = "Frog";
`myList.add(b);`	
`int theSize = myList.size();`	
`String str = myList.get(1);`	
`myList.remove(1);`	
`boolean isIn = myList.contains(b);`	

Answers on page 136.

there are no Dumb Questions

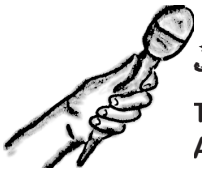
Q: So ArrayList is cool, but how would I know it exists?

A: The question is really, "How do I *know* what's in the API?" and that's the key to your success as a Java programmer. Not to mention your key to being as lazy as possible while still managing to build software. You might be amazed at how much time you can save when somebody else has already done most of the heavy lifting and all you have to do is step in and create the fun part.

But we digress...the short answer is that you spend some time learning what's in the core API. The long answer is at the end of this chapter, where you'll learn *how* to do that.

Q: But that's a pretty big issue. Not only do I need to know that the Java library comes with ArrayList, but more importantly I have to know that ArrayList is the thing that can do what I want! So how do I go from a need-to-do-something to a-way-to-do-it using the API?

A: Now you're really at the heart of it. By the time you've finished this book, you'll have a good grasp of the language, and the rest of your learning curve really is about knowing how to get from a problem to a solution, with you writing the least amount of code. If you can be patient for a few more pages, we start talking about it at the end of this chapter.



Java Exposed

This week's interview: ArrayList, on arrays

HeadFirst: So, ArrayLists are like arrays, right?

ArrayList: In their dreams! ***I*** am an *object*, thank you very much.

HeadFirst: If I'm not mistaken, arrays are objects too. They live on the heap right there with all the other objects.

ArrayList: Sure arrays go on the heap, ***duh***, but an array is still a wanna-be ArrayList. A poser. Objects have state *and* behavior, right? We're clear on that. But have you actually tried calling a method on an array?

HeadFirst: Now that you mention it, can't say I have. But what method would I call, anyway? I only care about calling methods on the stuff I put *in* the array, not the array itself. And I can use array syntax when I want to put things in and take things out of the array.

ArrayList: Is that so? You mean to tell me you actually *removed* something from an array? (Sheesh, where do they *train* you guys?)

HeadFirst: Of *course* I take something out of the array. I say Dog d = dogArray[1], and I get the Dog object at index 1 out of the array.

ArrayList: Alright, I'll try to speak slowly so you can follow along. You were *not,* I repeat *not,* removing that Dog from the array. All you did was make a copy of the *reference to the Dog* and assign it to another Dog variable.

HeadFirst: Oh, I see what you're saying. No, I didn't actually remove the Dog object from the array. It's still there. But I can just set its reference to null, I guess.

ArrayList: But I'm a first-class object, so I have methods, and I can actually, you know, *do* things like remove the Dog's reference from myself, not just set it to null. And I can change my size, *dynamically* (look it up). Just try to get an *array* to do that!

HeadFirst: Gee, hate to bring this up, but the rumor is that you're nothing more than a glorified but less-efficient array. That in fact you're just a wrapper for an array, adding extra methods for things like resizing that I would have had to write myself. And while we're at it, *you can't even hold primitives*! Isn't that a big limitation?

ArrayList: I can't *believe* you buy into that urban legend. No, I am *not* just a less-efficient array. I will admit that there are a few *extremely* rare situations where an array might be just a tad, I repeat, *tad* bit faster for certain things. But is it worth the *miniscule* performance gain to give up all this *power*? Still, look at all this *flexibility*. And as for the primitives, of *course* you can put a primitive in an ArrayList, as long as it's wrapped in a primitive wrapper class (you'll see a lot more on that in Chapter 10). And if you're using Java 5 or above, that wrapping (and unwrapping when you take the primitive out again) happens automatically. And alright, I'll *acknowledge* that yes, if you're using an ArrayList of *primitives*, it probably is faster with an array, because of all the wrapping and unwrapping, but still...who really uses primitives *these* days?

Oh, look at the time! *I'm late for Pilates.* We'll have to do this again sometime.

Solution

Sharpen your pencil (from page 134)

ArrayList	Regular array
`ArrayList<String> myList = new ArrayList<String>();`	`String [] myList = new String[2];`
`String a = "whoohoo";`	`String a = "whoohoo";`
`myList.add(a);`	`myList[0] = a;`
`String b = "Frog";`	`String b = "Frog";`
`myList.add(b);`	`myList[1] = b;`
`int theSize = myList.size();`	`int theSize = myList.length;`
`String str = myList.get(1);`	`String str = myList[1];`
`myList.remove(1);`	`myList[1] = null;`

`boolean isIn = myList.contains(b);`	`boolean isIn = false;`
	`for (String item : myList) {`
	`  if (b.equals(item)) {`
	`  isIn = true;`
	`  break;`
	`  }`
	`}`

Here's where it starts to look really different...

Notice how with ArrayList, you're working with an object of type ArrayList, so you're just invoking regular old methods on a regular old object, using the regular old dot operator.

With an *array*, you use *special array syntax* (like myList[0] = foo) that you won't use anywhere else except with arrays. Even though an array *is* an object, it lives in its own special world, and you can't invoke any methods on it, although you can access its one and only instance variable, *length*.

Comparing ArrayList to a regular array

1 A plain old array has to know its size at the time it's created.

But for ArrayList, you just make an object of type ArrayList. Every time. It never needs to know how big it should be, because it grows and shrinks as objects are added or removed.

```
new String[2]
```

Needs a size.

```
new ArrayList<String>()
```

No size required (although you can give it an initial size if you want to).

2 To put an object in a regular array, you must assign it to a specific location.

(An index from 0 to one less than the length of the array.)

```
myList[1] = b;
```

Needs an index.

If that index is outside the boundaries of the array (like the array was declared with a size of 2, and now you're trying to assign something to index 3), it blows up at runtime.

With ArrayList, you can specify an index using the *add(anInt, anObject)* method, or you can just keep saying *add(anObject)* and the ArrayList will keep growing to make room for the new thing.

```
myList.add(b);
```

No index.

3 Arrays use array syntax that's not used anywhere else in Java.

But ArrayLists are plain old Java objects, so they have no special syntax.

```
myList[1]
```

The array brackets [ ] are special syntax used only for arrays.

4 ArrayLists are parameterized.

We just said that unlike arrays, ArrayLists have no special syntax. But they *do* use something special—***parameterized types***.*

```
ArrayList<String>
```

The <String> in angle brackets is a "type parameter." ArrayList<String> means simply "a list of Strings," as opposed to ArrayList<Dog>, which means, "a list of Dogs."

Using the <TypeGoesHere> syntax, we can declare and create an ArrayList that knows (and restricts) the types of objects it can hold. We'll look at the details of parameterized types in ArrayLists in Chapter 11, *Data Structures*, so for now, don't think too much about the angle bracket <> syntax you see when we use ArrayLists. Just know that it's a way to force the compiler to allow only a specific type of object *(the type in angle brackets)* in the ArrayList.

**Parameterized types were added to Java in Java 5, which came out so long ago that you are almost definitely using a version that supports them!*

Let's fix the Startup code

Remember, this is how the buggy version looks:

We've renamed the class Startup now (instead of SimpleStartup), for the new advanced version, but this is the same code you saw in the last chapter.

```
class Startup {

  private int[] locationCells;
  private int numOfHits = 0;

  public void setLocationCells(int[] locs) {
    locationCells = locs;
  }

  public String checkYourself(int guess) {
    String result = "miss";
    for (int cell : locationCells) {
      if (guess == cell) {
        result = "hit";
        numOfHits++;

        break;
      }
    } // end for
    if (numOfHits == locationCells.length) {
      result = "kill";
    } // end if
    System.out.println(result);
    return result;
  } // end method
} // close class
```

Where it all went wrong. We counted each guess as a hit, without checking whether that cell had already been hit.

prep code | test code | real code

New and improved Startup class


```
import java.util.ArrayList;

public class Startup {

  private ArrayList<String> locationCells;
  // private int numOfHits;
  // don't need to track this now

  public void setLocationCells(ArrayList<String> locs) {
    locationCells = locs;
  }

  public String checkYourself(String userInput) {
    String result = "miss";
    int index = locationCells.indexOf(userInput);

    if (index >= 0) {

      locationCells.remove(index);

      if (locationCells.isEmpty()) {
        result = "kill";
      } else {
        result = "hit";
      } // end if
    } // end outer if
    return result;
  } // end method
} // close class
```

Ignore this line for now; we talk about it at the end of the chapter.

Change the int array to an ArrayList that holds Strings.

This is now a String – it needs to accept a value like "A3."

New and improved argument name.

Find out if the user guess is in the ArrayList, by asking for its index. If it's not in the list, then indexOf() returns a –1.

If index is greater than or equal to zero, the user guess is definitely in the list, so remove it.

If the list is empty, this was the killing blow!

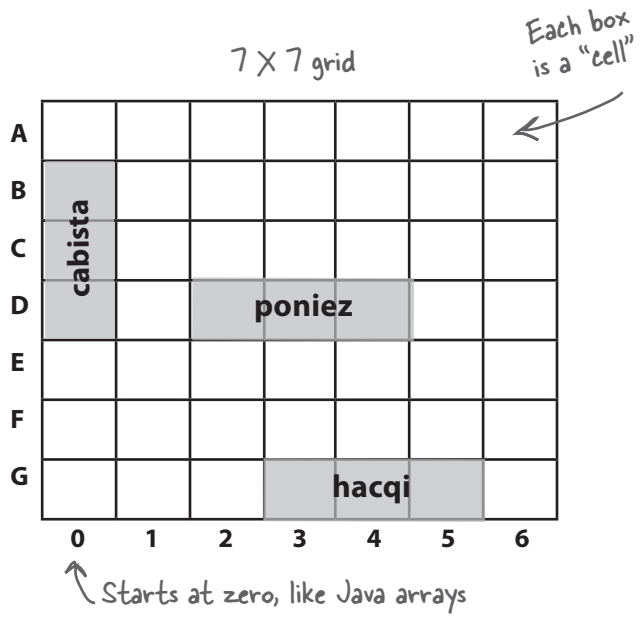
Let's build the REAL game: "Sink a Startup"

We've been working on the "simple" version, but now let's build the real one. Instead of a single row, we'll use a grid. And instead of one Startup, we'll use three.

Goal: Sink all of the computer's Startups in the fewest number of guesses. You're given a rating level based on how well you perform.

Setup: When the game program is launched, the computer places three Startups, randomly, on the **virtual 7 x 7 grid**. When that's complete, the game asks for your first guess.

How you play: We haven't learned to build a GUI yet, so this version works at the command line. The computer will prompt you to enter a guess (a cell), which you'll type at the command line (as "A3," "C5," etc.). In response to your guess, you'll see a result at the command-line, either "hit," "miss," or "You sunk poniez" (or whatever the lucky Startup of the day is). When you've sent all three Startups to that big 404 in the sky, the game ends by printing out your rating.



You're going to build the Sink a Startup game, with a 7 x 7 grid and three Startups. Each Startup takes up three cells.

part of a game interaction

```
File  Edit  Window  Help  Sell
%java StartupBust
Enter a guess  A3
miss
Enter a guess  B2
miss
Enter a guess  C4
miss
Enter a guess  D2
hit
Enter a guess  D3
hit
Enter a guess  D4
Ouch! You sunk poniez   : (
kill
Enter a guess  G3
hit
Enter a guess  G4
hit
Enter a guess  G5
Ouch! You sunk hacqi   : (
All Startups are dead! Your stock
is now worthless
Took you long enough. 62 guesses.
```

What needs to change?

We have three classes that need to change: the Startup class (which is now called Startup instead of SimpleStartup), the game class (StartupBust), and the game helper class (which we won't worry about now).

A Startup class

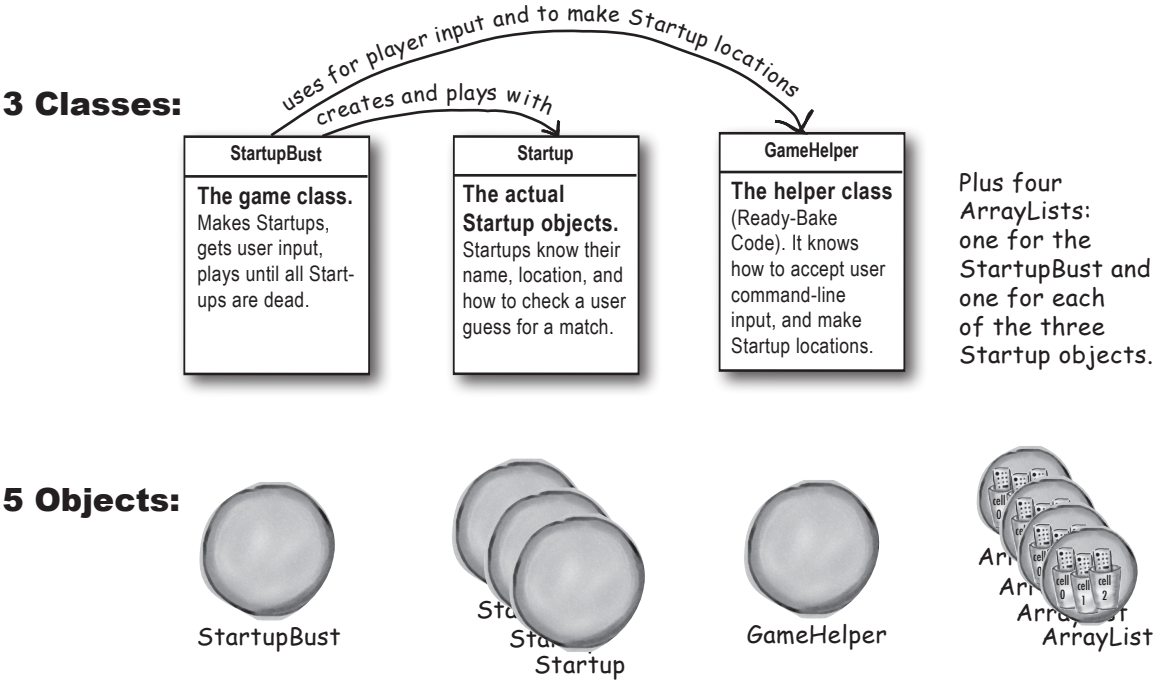
- **Add a *name* variable**
  to hold the name of the Startup ("poniez," "cabista," etc.) so each Startup can print its name when it's killed (see the output screen on the opposite page).

B StartupBust class (the game)

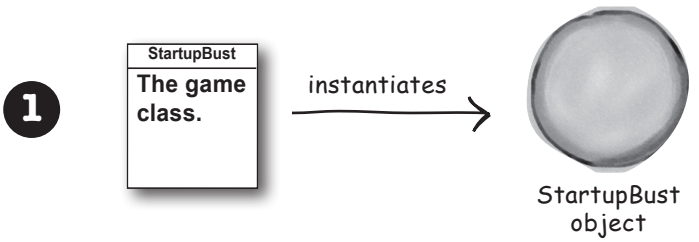
- **Create *three* Startups instead of one.**
- **Give each of the three Startups a *name*.**
  Call a setter method on each Startup instance so that the Startup can assign the name to its name instance variable.

StartupBust class continued...

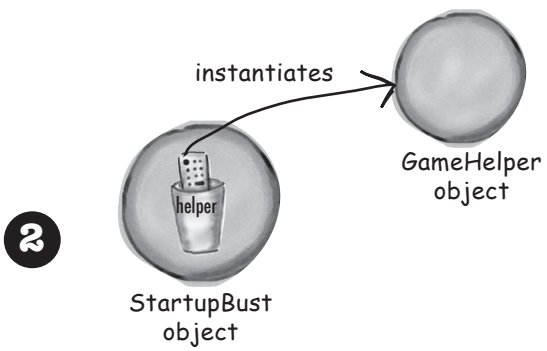
- **Put the Startups on a grid rather than just a single row, and do it for all three Startups.**
  This step is now way more complex than before, if we're going to place the Startups randomly. Since we're not here to mess with the math, we put the algorithm for giving the Startups a location into the GameHelper (Ready-Bake Code) class.
- **Check each user guess *with all three Startups*, instead of just one.**
- **Keep playing the game** (i.e., accepting user guesses and checking them with the remaining Startups) ***until there are no more live Startups.***
- **Get out of main.** We kept the simple one in main just to...keep it simple. But that's not what we want for the *real* game.



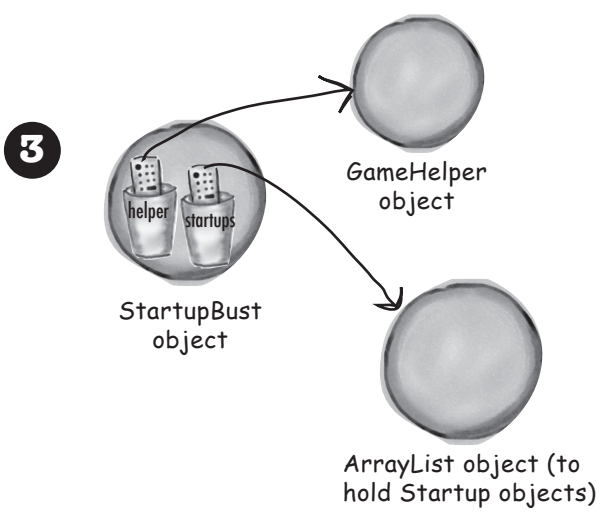
Who does what in the StartupBust game (and when)



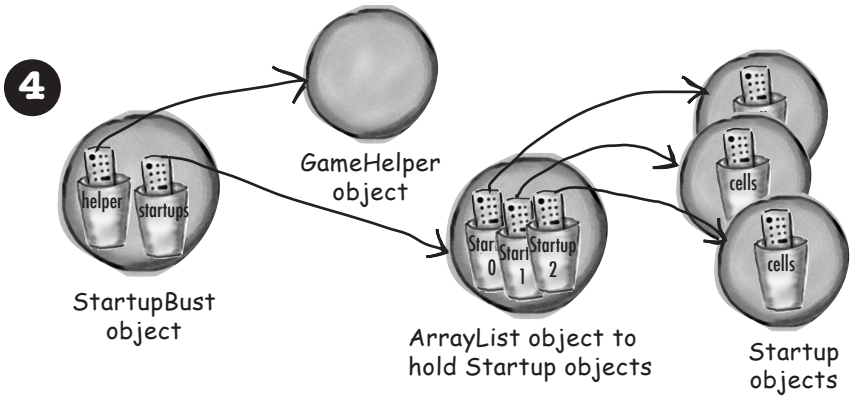
The main() method in the StartupBust class instantiates the StartupBust object that does all the game stuff.



The StartupBust (game) object instantiates an instance of GameHelper, the object that will help the game do its work.



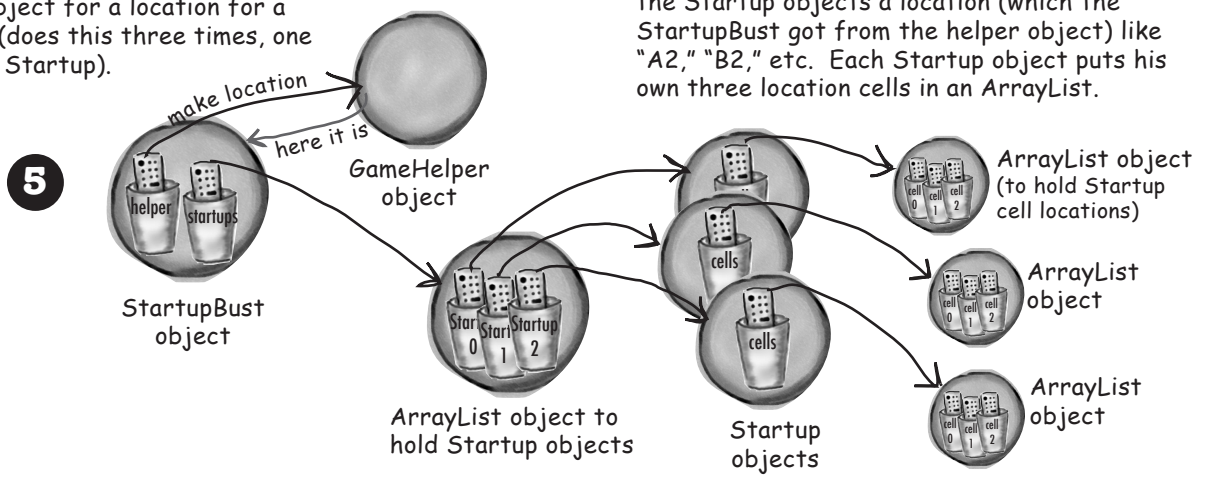
The StartupBust object instantiates an ArrayList that will hold the three Startup objects.



The StartupBust object creates three Startup objects (and puts them in the ArrayList).

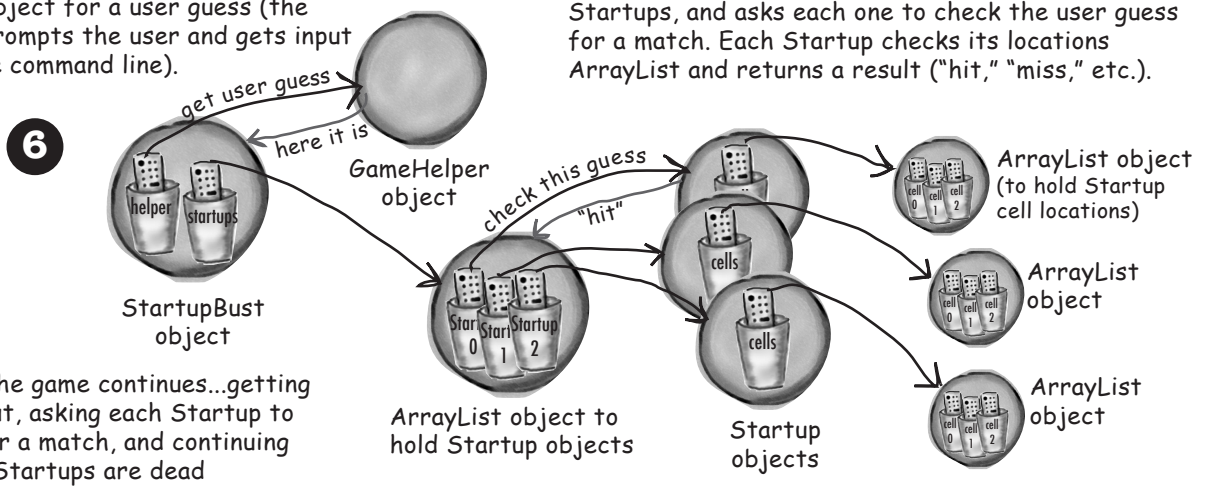
The StartupBust object asks the helper object for a location for a Startup (does this three times, one for each Startup).

The StartupBust object gives each of the Startup objects a location (which the StartupBust got from the helper object) like "A2," "B2," etc. Each Startup object puts his own three location cells in an ArrayList.

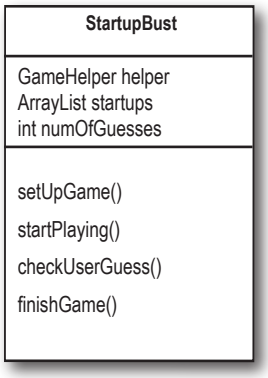


The StartupBust object asks the helper object for a user guess (the helper prompts the user and gets input from the command line).

The StartupBust object loops through the list of Startups, and asks each one to check the user guess for a match. Each Startup checks its locations ArrayList and returns a result ("hit," "miss," etc.).



And so the game continues...getting user input, asking each Startup to check for a match, and continuing until all Startups are dead



Prep code for the real StartupBust class

The StartupBust class has three main jobs: set up the game, play the game until the Startups are dead, and end the game. Although we could map those three jobs directly into three methods, we split the middle job (play the game) into *two* methods to keep the granularity smaller. Smaller methods (meaning smaller chunks of functionality) help us test, debug, and modify the code more easily.

Variable Declarations

DECLARE and instantiate the *GameHelper* instance variable, named ***helper.***

DECLARE and instantiate an *ArrayList* to hold the list of Startups (initially three) Call it ***startups***.

DECLARE an int variable to hold the number of user guesses (so that we can give the user a score at the end of the game). Name it ***numOfGuesses*** and set it to 0.

Method Declarations

DECLARE a ***setUpGame()*** method to create and initialize the Startup objects with names and locations. Display brief instructions to the user.

DECLARE a ***startPlaying()*** method that asks the player for guesses and calls the checkUserGuess() method until all the Startup objects are removed from play.

DECLARE a ***checkUserGuess()*** method that loops through all remaining Startup objects and calls each Startup object's checkYourself() method.

DECLARE a ***finishGame()*** method that prints a message about the user's performance, based on how many guesses it took to sink all of the Startup objects.

Method Implementations

METHOD: ***void setUpGame()***

// make three Startup objects and name them

CREATE three Startup objects.

SET a name for each Startup.

ADD the Startups to ***startups*** (the ArrayList).

REPEAT with each of the Startup objects in the ***startups*** List:

CALL the ***placeStartup()*** method on the helper object, to get a randomly-selected location for this Startup (three cells, vertically or horizontally aligned, on a 7 X 7 grid).

SET the location for each Startup based on the result of the ***placeStartup()*** call.

END REPEAT

END METHOD

prep code | test code | real code

Method implementations continued:

METHOD: ***void startPlaying()***

REPEAT while any Startups exist.

GET user input by calling the helper *getUserInput()* method.

EVALUATE the user's guess by *checkUserGuess()* method.

END REPEAT

END METHOD

METHOD: ***void checkUserGuess(String userGuess)***

// find out if there's a hit (and kill) on any Startup

INCREMENT the number of user guesses in the *numOfGuesses* variable.

SET the local *result* variable (a *String*) to "miss", assuming that the user's guess will be a miss.

REPEAT with each of the Startup objects in the *startups* List.

EVALUATE the user's guess by calling the Startup object's *checkYourself()* method.

SET the result variable to "hit" or "kill" if appropriate.

IF the result is "kill", **REMOVE** the Startup from the *startups* List.

END REPEAT

DISPLAY the *result* value to the user.

END METHOD

METHOD: ***void finishGame()***

DISPLAY a generic "game over" message, then:

IF number of user guesses is small,

DISPLAY a congratulations message.

ELSE

DISPLAY an insulting one.

END IF

END METHOD

→ Yours to solve.

Sharpen your pencil

How should we go from prep code to the final code? First we start with test code, and then test and build up our methods bit by bit. We won't keep showing you test code in this book, so now it's up to you to think about what you'd need to know to test these methods. And which method do you test and write first? See if you can work out some prep code for a set of tests. Prep code or even bullet points are good enough for this exercise, but if you want to try to write the *real* test code (in Java), knock yourself out.

the StartupBust code (the game)

prep code | test code | real code

```
import java.util.ArrayList;

public class StartupBust {
  private GameHelper helper = new GameHelper();
  private ArrayList<Startup> startups = new ArrayList<Startup>();
  private int numOfGuesses = 0;

  private void setUpGame() {
    // first make some Startups and give them locations
    Startup one = new Startup();
    one.setName("poniez");
    Startup two = new Startup();
    two.setName("hacqi");
    Startup three = new Startup();
    three.setName("cabista");
    startups.add(one);
    startups.add(two);
    startups.add(three);

    System.out.println("Your goal is to sink three Startups.");
    System.out.println("poniez, hacqi, cabista");
    System.out.println("Try to sink them all in the fewest number of guesses");

    for (Startup startup : startups) {
      ArrayList<String> newLocation = helper.placeStartup(3);
      startup.setLocationCells(newLocation);
    } // close for loop
  } // close setUpGame method

  private void startPlaying() {
    while (!startups.isEmpty()) {
      String userGuess = helper.getUserInput("Enter a guess");
      checkUserGuess(userGuess);
    } // close while
    finishGame();
  } // close startPlaying method
```

Sharpen your pencil

Annotate the code yourself!

Match the annotations at the bottom of each page with the numbers in the code. Write the number in the slot in front of the corresponding annotation.

You'll use each annotation just once, and you'll need all of the annotations.

Declare and initialize the variables we'll need

Get user input

Ask the helper for a Startup location

Repeat with each Startup in the list

Call the setter method on this Startup to give it the location you just got from the helper

Call our own checkUserGuess method

Print brief instructions for user

As long as the Startup list is NOT empty

Call our own finishGame method

Make three Startup objects, give 'em names, and stick 'em in the ArrayList

prep code | test code | real code

```
  private void checkUserGuess(String userGuess) {
    numOfGuesses++;                                          (11)
    String result = "miss";                                  (12)

    for (Startup startupToTest : startups) {                 (13)
      result = startupToTest.checkYourself(userGuess);       (14)

      if (result.equals("hit")) {
        break;                                               (15)
      }
      if (result.equals("kill")) {
        startups.remove(startupToTest);                      (16)
        break;
      }
    } // close for

    System.out.println(result);                              (17)
  } // close method


  private void finishGame() {
    System.out.println("All Startups are dead! Your stock is now worthless");
    if (numOfGuesses <= 18) {
      System.out.println("It only took you " + numOfGuesses + " guesses.");
      System.out.println("You got out before your options sank.");
    } else {
      System.out.println("Took you long enough. " + numOfGuesses + " guesses.");
      System.out.println("Fish are dancing with your options");
    }
  } // close method                                           (18)

  public static void main(String[] args) {
    StartupBust game = new StartupBust();                    (19)
    game.setUpGame();                                        (20)
    game.startPlaying();                                     (21)
  } // close method
}
```

Whatever you do, DON'T turn the page!

Not until you've finished this exercise.

Our version is on the next page.

Repeat with all Startups in the list

This one's dead, so take it out of the Startups list then get out of the loop

Print a message telling the user how they did in the game

Print the result for the user

Tell the game object to set up the game

Increment the number of guesses the user has made

Assume it's a 'miss,' unless told otherwise

Get out of the loop early, no point in testing the others

Tell the game object to start the main game play loop (keeps asking for user input and checking the guess)

Ask the Startup to check the user guess, looking for a hit (or kill)

Create the game object

the StartupBust code (the game)

prep code | test code | real code

```
import java.util.ArrayList;

public class StartupBust {

  private GameHelper helper = new GameHelper();
  private ArrayList<Startup> startups = new ArrayList<Startup>();
  private int numOfGuesses = 0;

  private void setUpGame() {
    // first make some Startups and give them locations
    Startup one = new Startup();
    one.setName("poniez");
    Startup two = new Startup();
    two.setName("hacqi");
    Startup three = new Startup();
    three.setName("cabista");
    startups.add(one);
    startups.add(two);
    startups.add(three);

    System.out.println("Your goal is to sink three Startups.");
    System.out.println("poniez, hacqi, cabista");
    System.out.println("Try to sink them all in the fewest number of guesses");

    for (Startup startup : startups) {
      ArrayList<String> newLocation = helper.placeStartup(3);
      startup.setLocationCells(newLocation);
    } // close for loop
  } // close setUpGame method

  private void startPlaying() {
    while (!startups.isEmpty()) {
      String userGuess = helper.getUserInput("Enter a guess");
      checkUserGuess(userGuess);
    } // close while
    finishGame();
  } // close startPlaying method
```

Declare and initialize the variables we'll need.

Make three Startup objects, give 'em names, and stick 'em in the ArrayList.

Print brief instructions for user.

Repeat with each Startup in the list.

Ask the helper for a Startup location (an ArrayList of Strings).

Call the setter method on this Startup to give it the location you just got from the helper.

As long as the Startup list is NOT empty (the ! means NOT, it's the same as (startups.isEmpty() == false).

Get user input.

Call our own checkUserGuess method.

Call our own finishGame method.

prep code | test code | real code

```
  private void checkUserGuess(String userGuess) {
    numOfGuesses++;
    String result = "miss";

    for (Startup startupToTest : startups) {
      result = startupToTest.checkYourself(userGuess);

      if (result.equals("hit")) {
        break;
      }
      if (result.equals("kill")) {
        startups.remove(startupToTest);
        break;
      }
    } // close for

    System.out.println(result);
  } // close method

  private void finishGame() {
    System.out.println("All Startups are dead! Your stock is now worthless");
    if (numOfGuesses <= 18) {
      System.out.println("It only took you " + numOfGuesses + " guesses.");
      System.out.println("You got out before your options sank.");
    } else {
      System.out.println("Took you long enough. " + numOfGuesses + " guesses.");
      System.out.println("Fish are dancing with your options");
    }
  } // close method

  public static void main(String[] args) {
    StartupBust game = new StartupBust();
    game.setUpGame();
    game.startPlaying();
  } // close method
}
```

Increment the number of guesses the user has made

Assume it's a 'miss', unless told otherwise

Repeat with all Startups in the list

Ask the Startup to check the user guess, looking for a hit (or kill)

Get out of the loop early, no point in testing the others

This one's dead, so take it out of the Startups list then get out of the loop

Print the result for the user

Print a message telling the user how they did in the game

Create the game object

Tell the game object to set up the game

Tell the game object to start the main game play loop (keeps asking for user input and checking the guess)

prep code | test code | real code

The final version of the Startup class

```
import java.util.ArrayList;

public class Startup {
  private ArrayList<String> locationCells;
  private String name;

  public void setLocationCells(ArrayList<String> loc) {
    locationCells = loc;
  }

  public void setName(String n) {
    name = n;
  }

  public String checkYourself(String userInput) {
    String result = "miss";
    int index = locationCells.indexOf(userInput);
    if (index >= 0) {
      locationCells.remove(index);

      if (locationCells.isEmpty()) {
        result = "kill";
        System.out.println("Ouch! You sunk " + name + "   : ( ");
      } else {
        result = "hit";
      } // end if
    } // end outer if
    return result;
  } // end method

} // close class
```

Startup's instance variables:
- an ArrayList of cell locations
- the Startup's name

A setter method that updates the Startup's location. (Random location provided by the GameHelper placeStartup() method.)

Your basic setter method

The ArrayList indexOf( ) method in action! If the user guess is one of the entries in the ArrayList, indexOf( ) will return its ArrayList location. If not, indexOf( ) will return –1.

Using ArrayList's remove() method to delete an entry.

Using the isEmpty() method to see if all of the locations have been guessed

Tell the user when a Startup has been sunk.

Return: 'miss' or 'hit' or 'kill'.

Super powerful Boolean expressions

So far, when we've used Boolean expressions for our loops or `if` tests, they've been pretty simple. We will be using more powerful boolean expressions in some of the Ready-Bake Code you're about to see, and even though we know you wouldn't peek, we thought this would be a good time to discuss how to energize your expressions.

"And" and "Or" Operators (&&, ||)

Let's say you're writing a chooseCamera( ) method, with lots of rules about which camera to select. Maybe you can choose cameras ranging from $50 to $1000, but in some cases you want to limit the price range more precisely. You want to say something like:

"If the price *range* is between $300 ***and*** $400, then choose X."

```
if (price >= 300 && price < 400) {
  camera = "X";
}
```

Let's say that of the ten camera brands available, you have some logic that applies to only a *few* of the list:

```
if (brand.equals("A") || brand.equals("B")) {
  // do stuff for only brand A or brand B
}
```

Boolean expressions can get really big and complicated:

```
if ((zoomType.equals("optical") &&
     (zoomDegree >= 3 && zoomDegree <= 8)) ||
    (zoomType.equals("digital") &&
     (zoomDegree >= 5 && zoomDegree <= 12))) {
  // do appropriate zoom stuff
}
```

If you want to get *really* technical, you might wonder about the *precedence* of these operators. Instead of becoming an expert in the arcane world of precedence, we recommend that you ***use parentheses*** to make your code clear.

Not equals (!= and !)

Let's say that you have a logic like "of the ten available camera models, a certain thing is *true for all but one*."

```
if (model != 2000) {
  //  do non-model 2000 stuff
}
```

or for comparing objects like strings...

```
if (!brand.equals("X")) {
  // do non-brand X stuff
}
```

Short-Circuit Operators (&& , ||)

The operators we've looked at so far, && and ||, are known as ***short-circuit*** operators. In the case of &&, the expression will be true only if *both* sides of the && are true. So if the JVM sees that the left side of a && expression is false, it stops right there! Doesn't even bother to look at the right side.

Similarly, with ||, the expression will be true if *either* side is true, so if the JVM sees that the left side is true, it declares the entire statement to be true and doesn't bother to check the right side.

Why is this great? Let's say that you have a reference variable and you're not sure whether it's been assigned to an object. If you try to call a method using this null reference variable (i.e., no object has been assigned), you'll get a NullPointerException. So, try this:

```
if (refVar != null &&
    refVar.isValidType()) {
  // do 'got a valid type' stuff
}
```

Non-Short-Circuit Operators (& , |)

When used in boolean expressions, the & and | operators act like their && and || counterparts, except that they force the JVM to *always* check *both* sides of the expression. Typically, & and | are used in another context, for manipulating bits.

Ready-Bake Code

This is the helper class for the game. Besides the user input method (that prompts the user and reads input from the command line), the helper's Big Service is to create the cell locations for the Startups. We tried to keep it fairly small so you wouldn't have to type so much. And remember, you won't be able to compile the StartupBust game class until you have *this* class.

```
import java.util.*;

public class GameHelper {
  private static final String ALPHABET = "abcdefg";
  private static final int GRID_LENGTH = 7;
  private static final int GRID_SIZE = 49;
  private static final int MAX_ATTEMPTS = 200;
  static final int HORIZONTAL_INCREMENT = 1;          // A better way to represent these two
  static final int VERTICAL_INCREMENT = GRID_LENGTH;  // things is an enum (see Appendix B)

  private final int[] grid = new int[GRID_SIZE];
  private final Random random = new Random();
  private int startupCount = 0;

  public String getUserInput(String prompt) {
    System.out.print(prompt + ": ");
    Scanner scanner = new Scanner(System.in);
    return scanner.nextLine().toLowerCase();
  } //end getUserInput

  public ArrayList<String> placeStartup(int startupSize) {
    // holds index to grid (0 - 48)
    int[] startupCoords = new int[startupSize];         // current candidate co-ordinates
    int attempts = 0;                                   // current attempts counter
    boolean success = false;                            // flag = found a good location?

    startupCount++;                                     // nth Startup to place
    int increment = getIncrement();                     // alternate vert & horiz alignment

    while (!success & attempts++ < MAX_ATTEMPTS) {      // main search loop
      int location = random.nextInt(GRID_SIZE);         // get random starting point

      for (int i = 0; i < startupCoords.length; i++) {  // create array of proposed coords
        startupCoords[i] = location;                    // put current location in array
        location += increment;                          // calculate the next location
      }
      // System.out.println("Trying: " + Arrays.toString(startupCoords));

      if (startupFits(startupCoords, increment)) {      // startup fits on the grid?
        success = coordsAvailable(startupCoords);       // ...and locations aren't taken?
      }                                                 // end loop
    }                                                   // end while
    savePositionToGrid(startupCoords);                  // coords passed checks, save
    ArrayList<String> alphaCells = convertCoordsToAlphaFormat(startupCoords);
    // System.out.println("Placed at: "+ alphaCells);
    return alphaCells;
  } //end placeStartup
```

Note: For extra credit, you might try "un-commenting" the System.out.println's, just to watch it work! These print statements will let you "cheat" by giving you the location of the Startups, but it will help you test it.

This is the statement that tells you exactly where the Startup is located.

Ready-Bake Code

GameHelper class code continued...

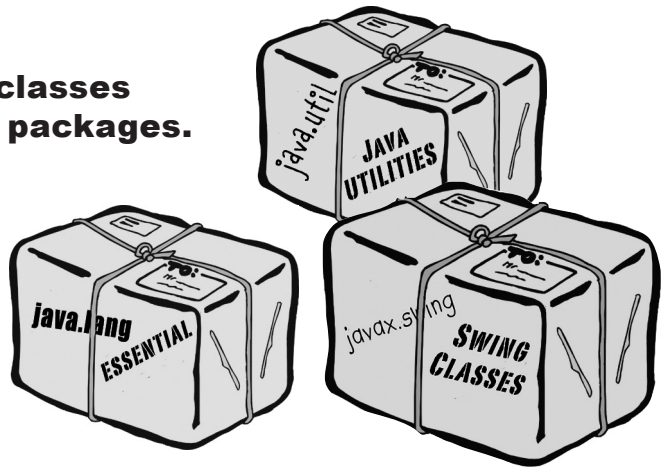
```
  private boolean startupFits(int[] startupCoords, int increment) {
    int finalLocation = startupCoords[startupCoords.length - 1];
    if (increment == HORIZONTAL_INCREMENT) {
      // check end is on same row as start
      return calcRowFromIndex(startupCoords[0]) == calcRowFromIndex(finalLocation);
    } else {
      return finalLocation < GRID_SIZE;                 // check end isn't off the bottom
    }
  } //end startupFits
  private boolean coordsAvailable(int[] startupCoords) {
    for (int coord : startupCoords) {                    // check all potential positions
      if (grid[coord] != 0) {                            // this position already taken
        // System.out.println("position: " + coord + " already taken.");
        return false;                                    // NO success
      }
    }
    return true;                                         // there were no clashes, yay!
  } //end coordsAvailable
  private void savePositionToGrid(int[] startupCoords) {
    for (int index : startupCoords) {
      grid[index] = 1;                                   // mark grid position as 'used'
    }
  } //end savePositionToGrid
  private ArrayList<String> convertCoordsToAlphaFormat(int[] startupCoords) {
    ArrayList<String> alphaCells = new ArrayList<String>();
    for (int index : startupCoords) {                    // for each grid coordinate
      String alphaCoords = getAlphaCoordsFromIndex(index); // turn it into an "a0" style
      alphaCells.add(alphaCoords);                       // add to a list
    }
    return alphaCells;                                   // return the "a0"-style coords
  } // end convertCoordsToAlphaFormat
  private String getAlphaCoordsFromIndex(int index) {
    int row = calcRowFromIndex(index);                   // get row value
    int column = index % GRID_LENGTH;                    // get numeric column value
    String letter = ALPHABET.substring(column, column + 1); // convert to letter
    return letter + row;
  } // end getAlphaCoordsFromIndex
  private int calcRowFromIndex(int index) {
    return index / GRID_LENGTH;
  } // end calcRowFromIndex
  private int getIncrement() {
    if (startupCount % 2 == 0) {                         // if EVEN Startup
      return HORIZONTAL_INCREMENT;                       // place horizontally
    } else {                                             // else ODD
      return VERTICAL_INCREMENT;                         // place vertically
    }
  } //end getIncrement
} //end class
```

This code, and a basic test, is available in the GitHub repo, https://oreil.ly/hfJava_3e_examples

Using the Library (the Java API)

You made it all the way through the StartupBust game, thanks to the help of ArrayList. And now, as promised, it's time to learn how to fool around in the Java library.

In the Java API, classes are grouped into packages.



To use a class in the API, you have to know which package the class is in.

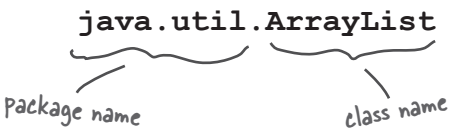
Every class in the Java library belongs to a package. The package has a name, like **javax.swing** (a package that holds some of the Swing GUI classes you'll learn about soon). ArrayList is in the package called **java.util**, which surprise surprise, holds a pile of *utility* classes. You'll learn a lot more about packages in Appendix B, including how to put your *own* classes into your *own* packages. For now, though, we're just looking to *use* some of the classes that come with Java.

Using a class from the API, in your own code, is simple. You just treat the class as though you wrote it yourself... as though you compiled it, and there it sits, waiting for you to use it. With one big difference: somewhere in your code you have to indicate the *full* name of the library class you want to use, and that means package name + class name.

Even if you didn't know it, ***you've already been using classes from a package.*** System (System.out.println), String, and Math (Math.random()) all belong to the **java.lang** package.

You have to know the full name* of the class you want to use in your code.

ArrayList is not the *full* name of ArrayList, just as Kathy isn't a full name (unless it's like Madonna or Cher, but we won't go there). The full name of ArrayList is actually:



You have to tell Java which ArrayList you want to use. You have two options:

A IMPORT

Put an import statement at the top of your source code file:

```
import java.util.ArrayList;
public class MyClass {... }
```

OR

B TYPE

Type the full name everywhere in your code. Each time you use it. *Everywhere* you use it.

When you declare and/or instantiate it:

```
java.util.ArrayList<Dog> list = new java.util.ArrayList<Dog>();
```

When you use it as an argument type:

```
public void go(java.util.ArrayList<Dog> list) { }
```

When you use it as a return type:

```
public java.util.ArrayList<Dog> foo() {...}
```

**Unless the class is in the java.lang package.*

there are no Dumb Questions

Q: Why does there have to be a full name? Is that the only purpose of a package?

A: Packages are important for three main reasons. First, they help the overall organization of a project or library. Rather than just having one horrendously large pile of classes, they're all grouped into packages for specific kinds of functionality (like GUI or data structures or database stuff, etc.).

Second, packages give you a name-scoping, to help prevent collisions if you and 12 other programmers in your company all decide to make a class with the same name. If you have a class named Set and someone else (including the Java API) has a class named Set, you need some way to tell the JVM *which* Set class you're trying to use.

Third, packages provide a level of security, because you can restrict the code you write so that only other classes in the same package can access it. The details are in Appendix B.

Q: OK, back to the name collision thing. How does a full name really help? What's to prevent two people from giving a class the same package name?

A: Java has a naming convention that usually prevents this from happening, as long as developers adhere to it.

BULLET POINTS

- **ArrayList** is a class in the Java API.
- To put something into an ArrayList, use **add()**.
- To remove something from an ArrayList use **remove()**.
- To find out where something is (and if it is) in an ArrayList, use i**ndexOf()**.
- To find out if an ArrayList is empty, use **isEmpty()**.
- To get the size (number of elements) in an ArrayList, use the **size()** ***method***.
- To get the **length** (number of elements) in a regular old array, remember, you use the **length** ***variable***.
- An ArrayList **resizes dynamically** to whatever size is needed. It grows when objects are added, and it **shrinks** when objects are removed.
- You declare the type of the array using a **type parameter**, which is a type name in angle brackets. Example: ArrayList<Button> means the ArrayList will be able to hold only objects of type Button (or subclasses of Button as you'll learn in the next couple of chapters).
- Although an ArrayList holds objects and not primitives, the compiler will automatically "wrap" (and "unwrap" when you take it out) a primitive into an Object and place that object in the ArrayList instead of the primitive. (More on this feature later in the book.)
- Classes are grouped into packages.
- A class has a full name, which is a combination of the package name and the class name. Class ArrayList is really java.util.ArrayList.
- To use a class in a package other than java.lang, you must tell Java the full name of the class.
- You can either use an import statement at the top of your source code, or you can type the full name every place you use the class in your code.

there are no Dumb Questions

Q: Does `import` make my class bigger? Does it actually compile the imported class or package into *my* code?

A: Perhaps you're a C programmer? An `import` is not the same as an `include`. So the answer is no and no. Repeat after me: "an `import` statement saves you from typing." That's really it. You don't have to worry about your code becoming bloated, or slower, from too many imports. An `import` is simply the way you give Java the *full name of a class*.

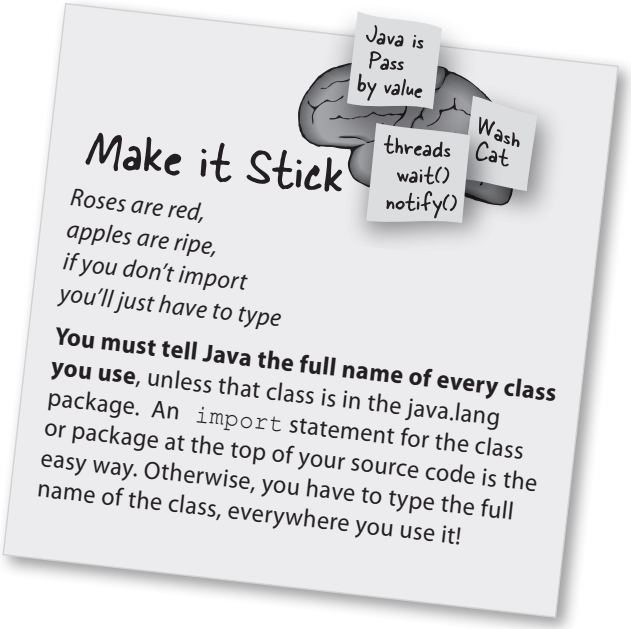
Q: OK, how come I never had to import the String class? Or System?

A: Remember, you get the java.lang package sort of "pre-imported" for free. Because the classes in java.lang are so fundamental, you don't have to use the full name. There is only one java.lang.String class and one java.lang.System class, and Java darn well knows where to find them.

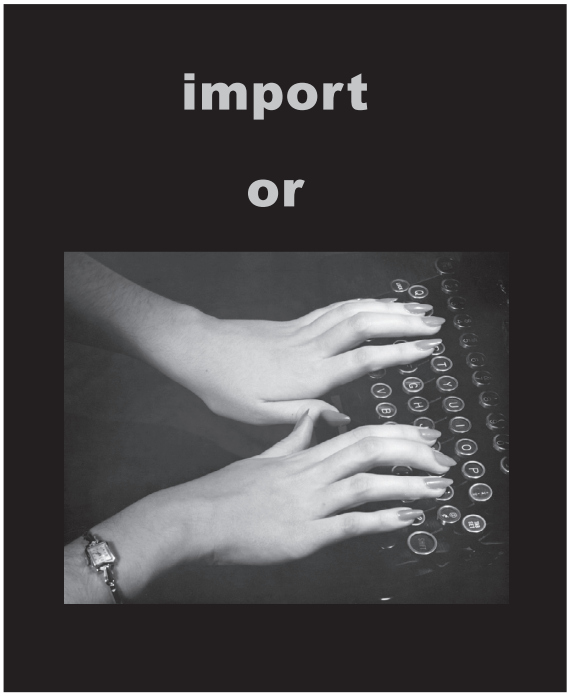
Q: Do I have to put my own classes into packages? How do I do that? *Can* I do that?

A: In the real world (which you should try to avoid), yes, you *will* want to put your classes into packages. We'll get into that in detail in Appendix B. For now, we won't put our code examples in a package.*

**But when you look at the code in the repo (https://oreil.ly/hfJava_3e_examples), you'll see we put the classes into packages.*



One more time, in the unlikely event that you don't already have this down:



*"Good to know there's an ArrayList in the java.util package. But by myself, how would **I** have figured that out?"*

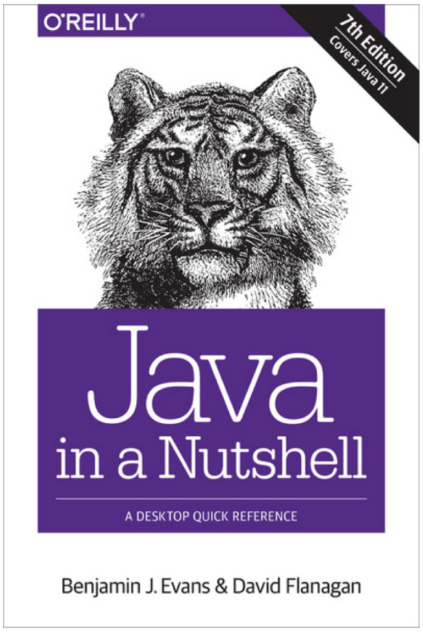
- Julia, 31, hand model

How to discover the API

Two things you want to know:

1. **What features are available in the library? (Which classes?)**
2. **How do you use these features? (Once you find a class, how do you know what it can do?)**

1 Browse a book



2 Use the HTML API docs

OVERVIEW MODULE PACKAGE CLASS USE TREE PREVIEW NEW DEPRECATED INDEX HELP

Java® Platform, Standard Edition & Java Development Kit Version 17 API Specification

This document is divided into two sections:

Java SE

The Java Platform, Standard Edition (Java SE) APIs define the core Java platform

JDK

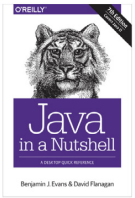
The Java Development Kit (JDK) APIs are specific to the JDK and will not necessar

All Modules	Java SE	JDK	Other Modules

Module	Description
java.base	Defines the foundational APIs of the Java SE Platform.
java.compiler	Defines the Language Model, Annotation Processing, a
java.datatransfer	Defines the API for transferring data between and with

https://docs.oracle.com/en/java/javase/17/docs/api/index.html

1 Browse a book



Flipping through a reference book is a good way to find out what's in the Java library. You can easily stumble on to a package or class that looks useful just by browsing pages.

O'REILLY

8. Working with Java Collections

12h 3m remaining

The List Interface

A `List` is an ordered collection of objects. Each element of a list has a position in the list, and the `List` interface defines methods to query or set the element at a particular position, or *index*. In this respect, a `List` is like an array whose size changes as needed to accommodate the number of elements it contains. Unlike sets, lists allow duplicate elements.

In addition to its index-based `get()` and `set()` methods, the `List` interface defines methods to add or remove an element at a particular index and also defines methods to return the index of the first or last occurrence of a particular value in the list. The `add()` and `remove()` methods inherited from `Collection` are defined to append to the list and to remove the first occurrence of the specified value from the list. The inherited `addAll()` appends all elements in the specified collection to the end of the list, and another version inserts the elements at a specified index. The `retainAll()` and `removeAll()` methods behave as they do for any `Collection`, retaining or removing multiple occurrences of the same value, if needed.

The `List` interface does not define methods that operate on a range of list indexes. Instead, it defines a single `subList()` method that returns a `List` object that represents just the specified range of the original list. The sublist is backed by the parent list, and any changes made to the sublist are immediately visible in the parent list. Examples of `subList()` and the other basic `List` manipulation methods are shown here:

```
// Create lists to work with
List<String> l = new ArrayList<String>(Arrays.asList(args));
List<String> words = Arrays.asList("hello", "world");
List<String> words2 = List.of("hello", "world");

// Querying and setting elements by index
```

2 Use the HTML API docs

Java comes with a fabulous set of online docs called, strangely, the Java API. You (or your IDE) can also download the docs to have on your hard drive just in case your internet connection fails at the Worst Possible Moment.

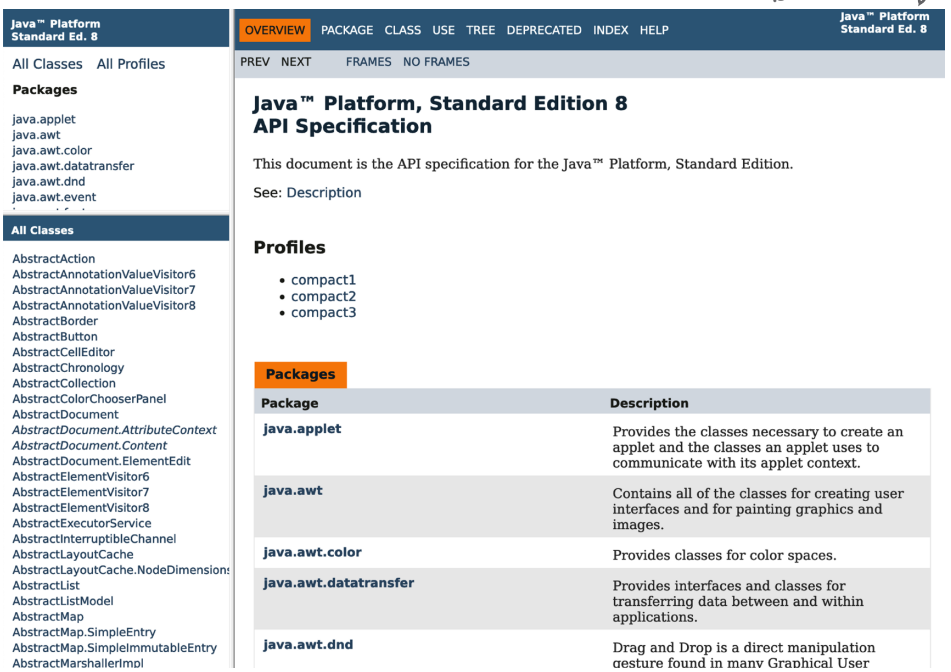
The API docs are the best reference for getting more details about what's in a package, and what the classes and interfaces in the package provide (e.g., in terms of methods and functionality).

The docs look different depending upon the version of Java you're using Make sure you're looking at the docs for your version of Java!

Java 8 and earlier

https://docs.oracle.com/javase/8/docs/api/index.html

Java version. This is Java 8 SE



Scroll through the packages and select one (click it) to restrict the list in the lower frame to only classes from that package.

Scroll through the classes and select one (click it) to choose the class that will fill the main browser frame.

You can navigate these docs:

- **Top down:** find a package you're interested in from the list in the top left and drill down.
- **Class-first:** find the class you want to know more about in the list in the bottom left, and click it.

The main panel will show you the details of whatever you're looking at. If you select a package, it will give summary information about that package and a list of the classes and interfaces.

If you select a class, it will show you a description of the class, and details of all the methods in the class, what they do, and how to use them.

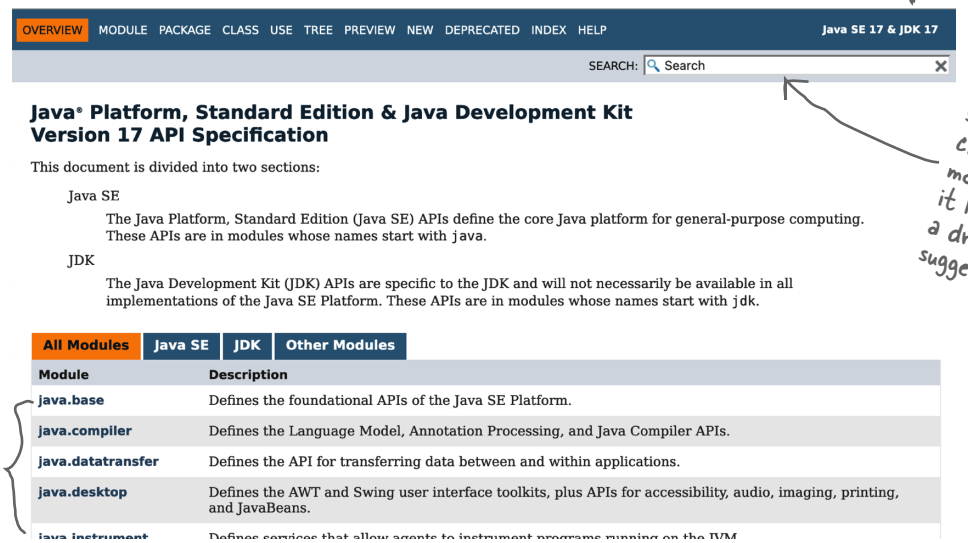
Java 9 and later

Java 9 introduced the Java Module System, which we're not going to cover in this book. What you do need to know to understand the docs is that the JDK is now split into *modules*. These modules group together related packages. This can make it easier to find the classes that interest you, because they're grouped by function. All of the classes we've covered in this book so far are in the **java.base** module; this contains core Java packages like java.lang and java.util.

Slightly different URL to the older docs

https://docs.oracle.com/en/java/javase/17/docs/api/index.html

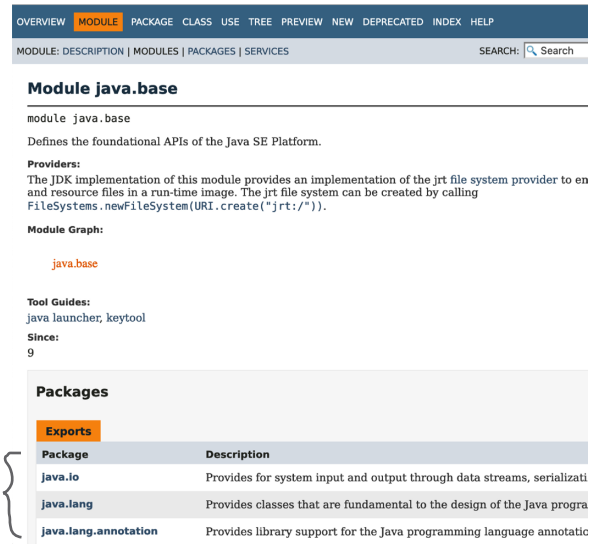
Java 17 is the current Long Term Support (LTS) version at the time of writing.



Search for a specific method/class/package/module by typing it here. You'll see a drop-down of suggestions.

The Java platform is now broken into a number of modules, which are listed on the home page of the docs.

We're mostly only interested in java.base. When we get to the Swing GUI, we'll care about java.desktop as well.



You can navigate these docs:

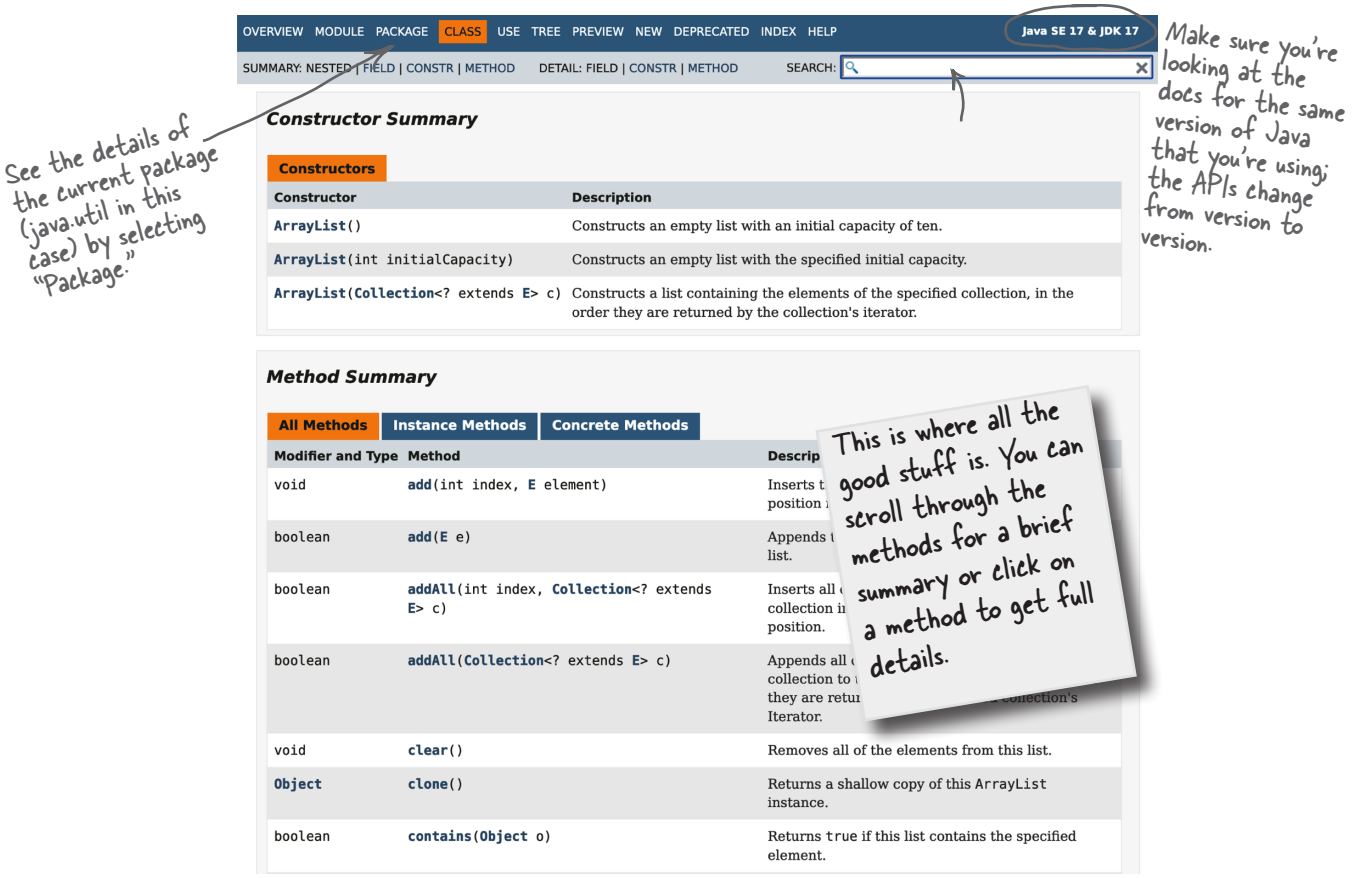
- **Top down:** find a module that looks like it covers the functionality you want, see its packages, and drill down from a package into its classes.
- **Search:** Use the search in the top right to go directly to the method, class, package, or module you want to read about.

When you've selected a module, you can see a list of all its packages and a description of what each package is for.

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:

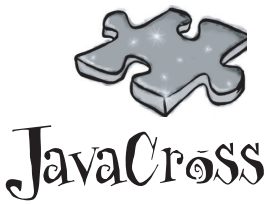
add(int index, Object o)

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);
```

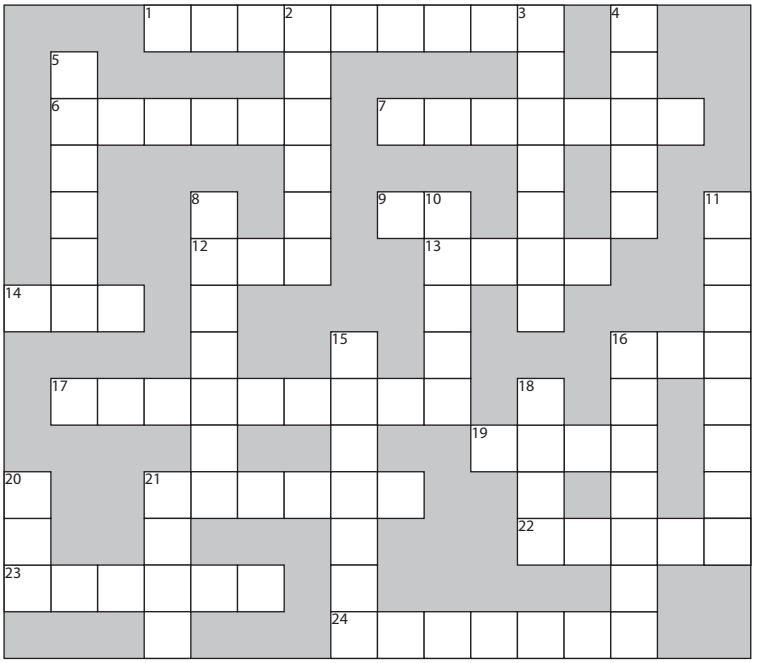
```
File  Edit  Window  Help  Dance
% java ArrayListMagnet
zero  one  two  three
zero  one  three  four
zero  one  three  four  4.2
zero  one  three  four  4.2
```

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
```

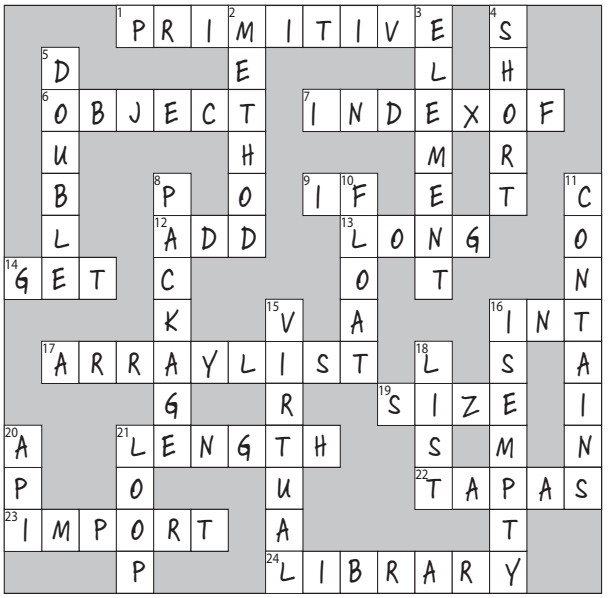
Code Magnets

(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.

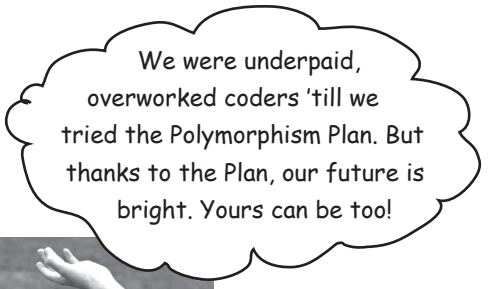
Across

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



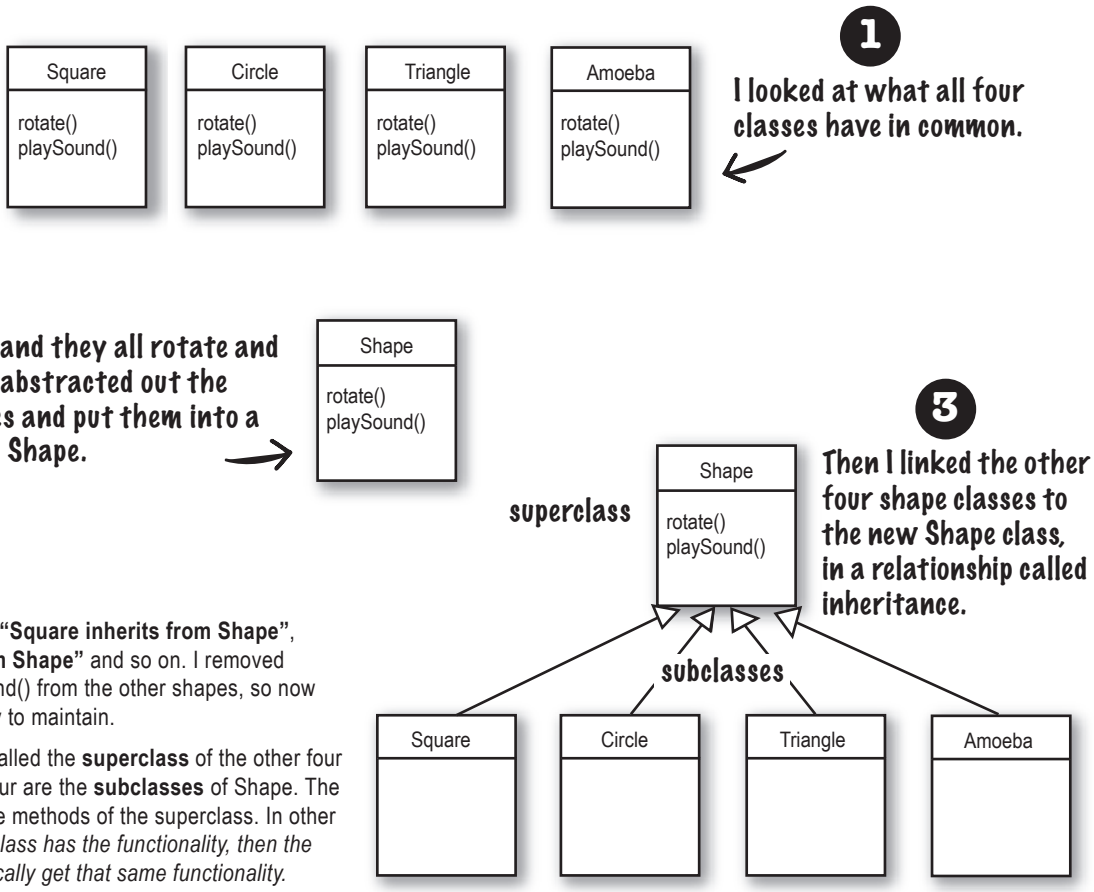
Plan your programs with the future in mind. If there were a way to write Java code such that you could take more vacations, how much would it be worth to you? What if you could write code that someone *else* could extend, **easily**? And if you could write code that was flexible, for those pesky last-minute 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.

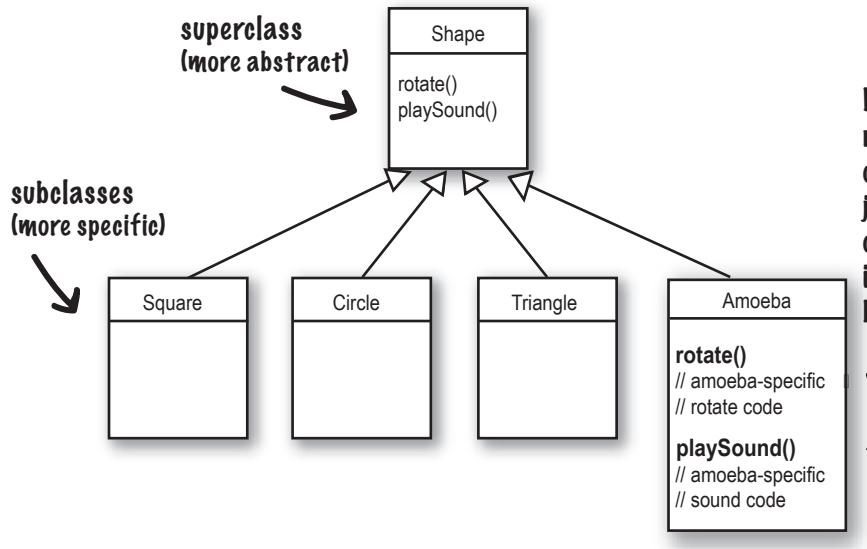
The Shape class is called the **superclass** of the other four classes. The other four are the **subclasses** of Shape. The subclasses inherit the methods of the superclass. In other words, *if the Shape class has the functionality, then the subclasses automatically get that same functionality.*

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?

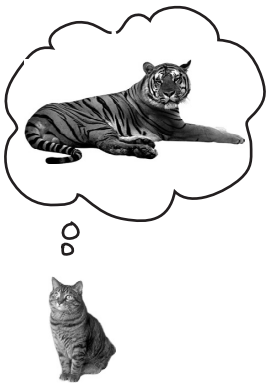
How can Amoeba do something different if it *inherits* its functionality from the Shape class?

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?

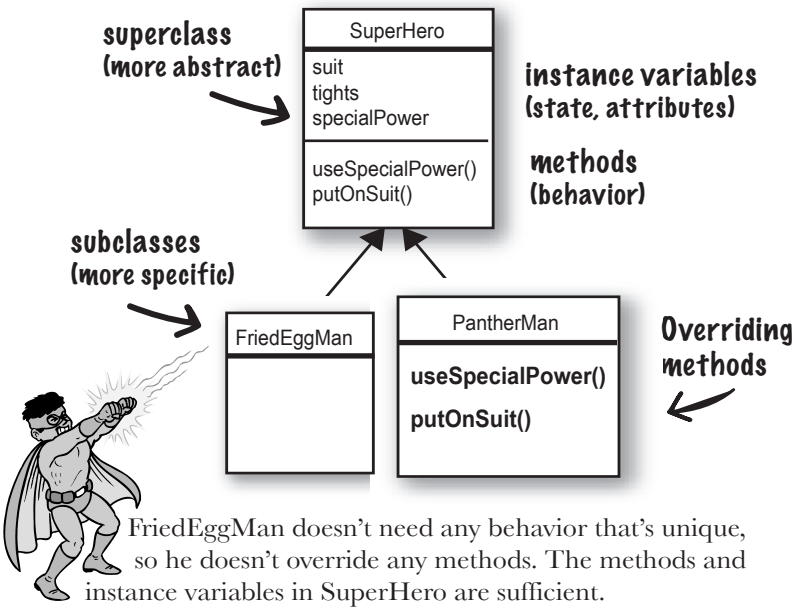
How would you design an inheritance structure? What methods would be overridden?

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.

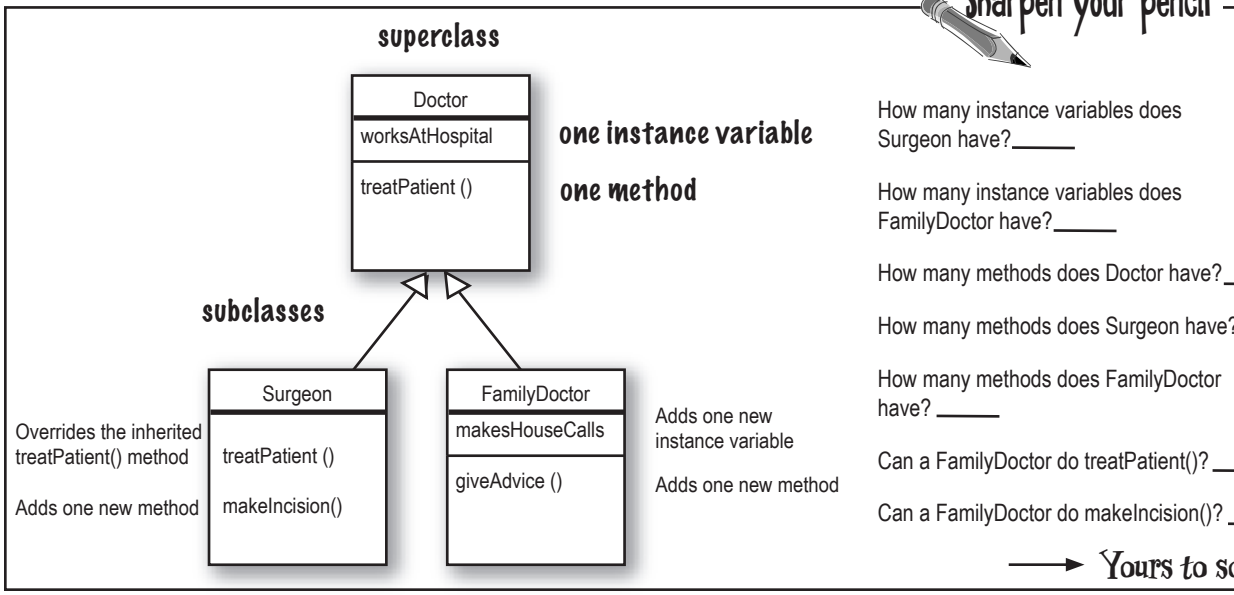
An inheritance example:

```
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.

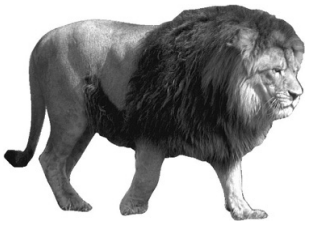
And we want other programmers to be able to add new kinds of animals to the program at any time.

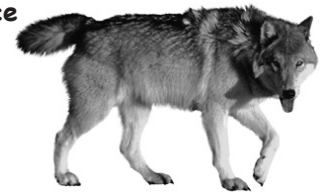
First we have to figure out the common, abstract characteristics that all animals have, and build those characteristics into a class that all animal classes can extend.

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.

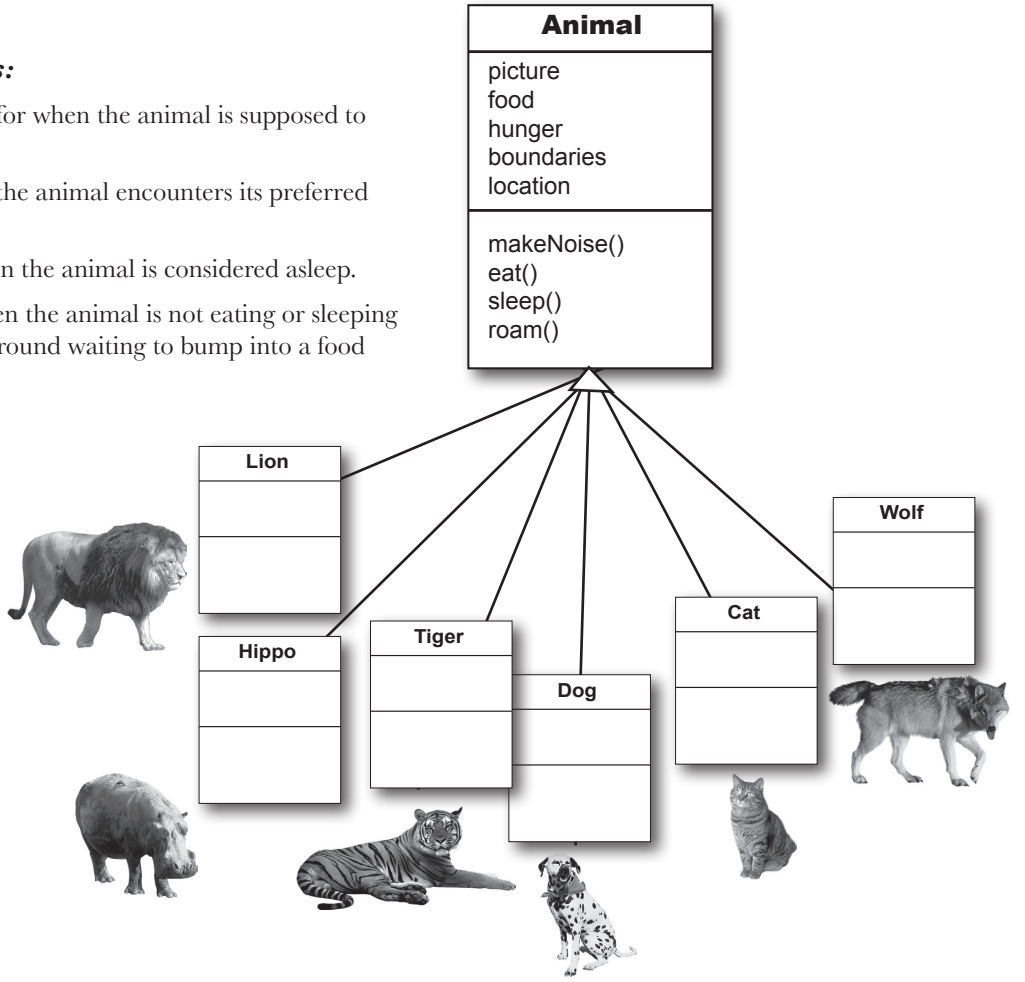
roam() – behavior for when the animal is not eating or sleeping (probably just wandering around waiting to bump into a food source or a boundary).

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*?

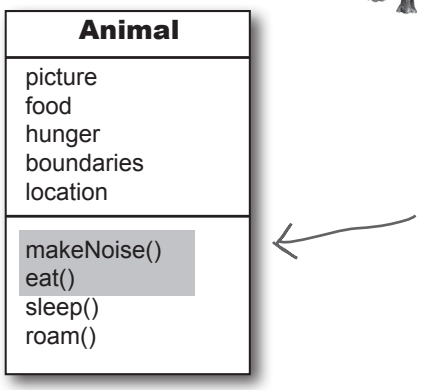
Which methods should we override?

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

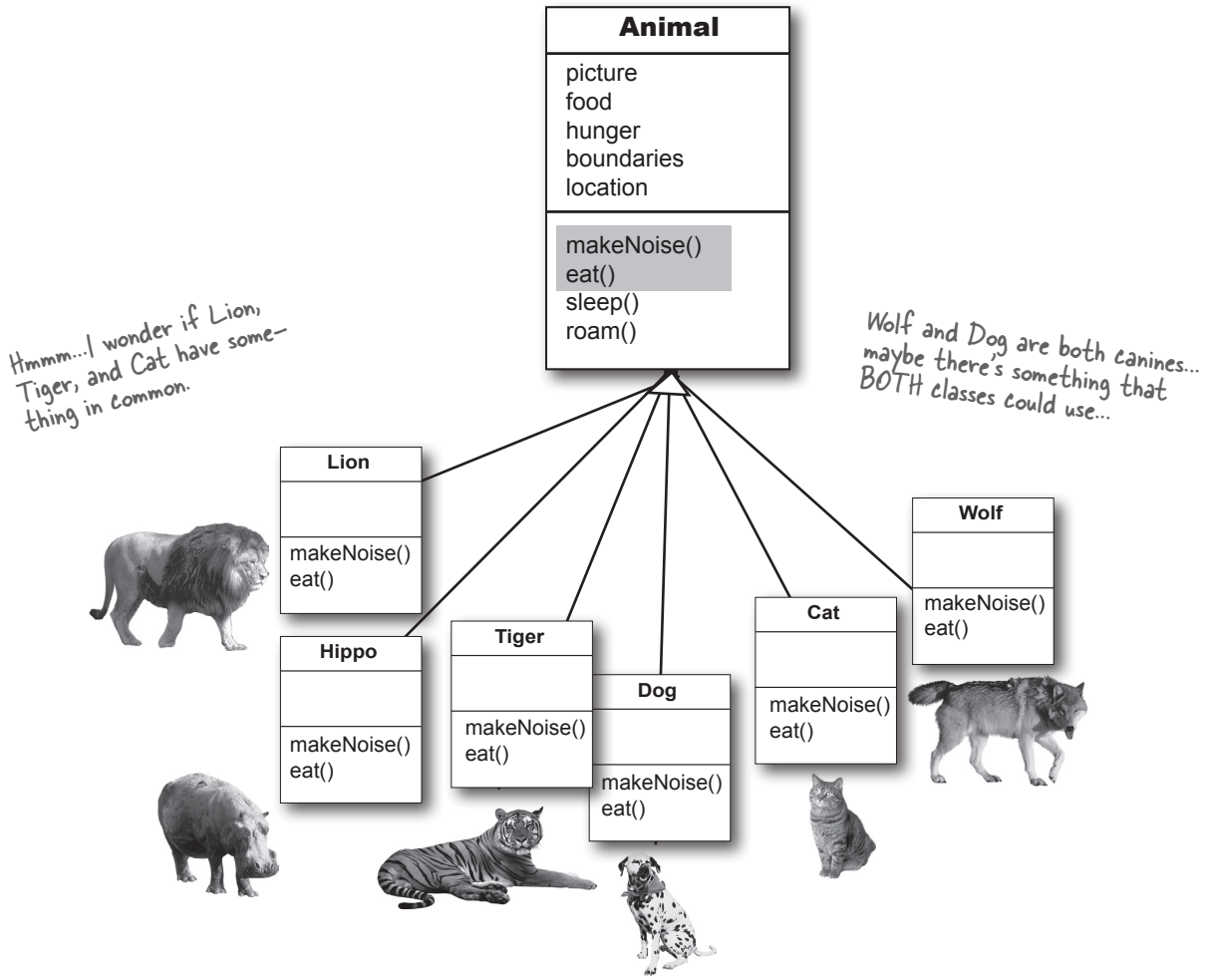
The class hierarchy is starting to shape up. We have each subclass override the *makeNoise()* and *eat()* methods so that there's no mistaking a Dog bark from a Cat meow (quite insulting to both parties). And a Hippo won't eat like a Lion.

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.

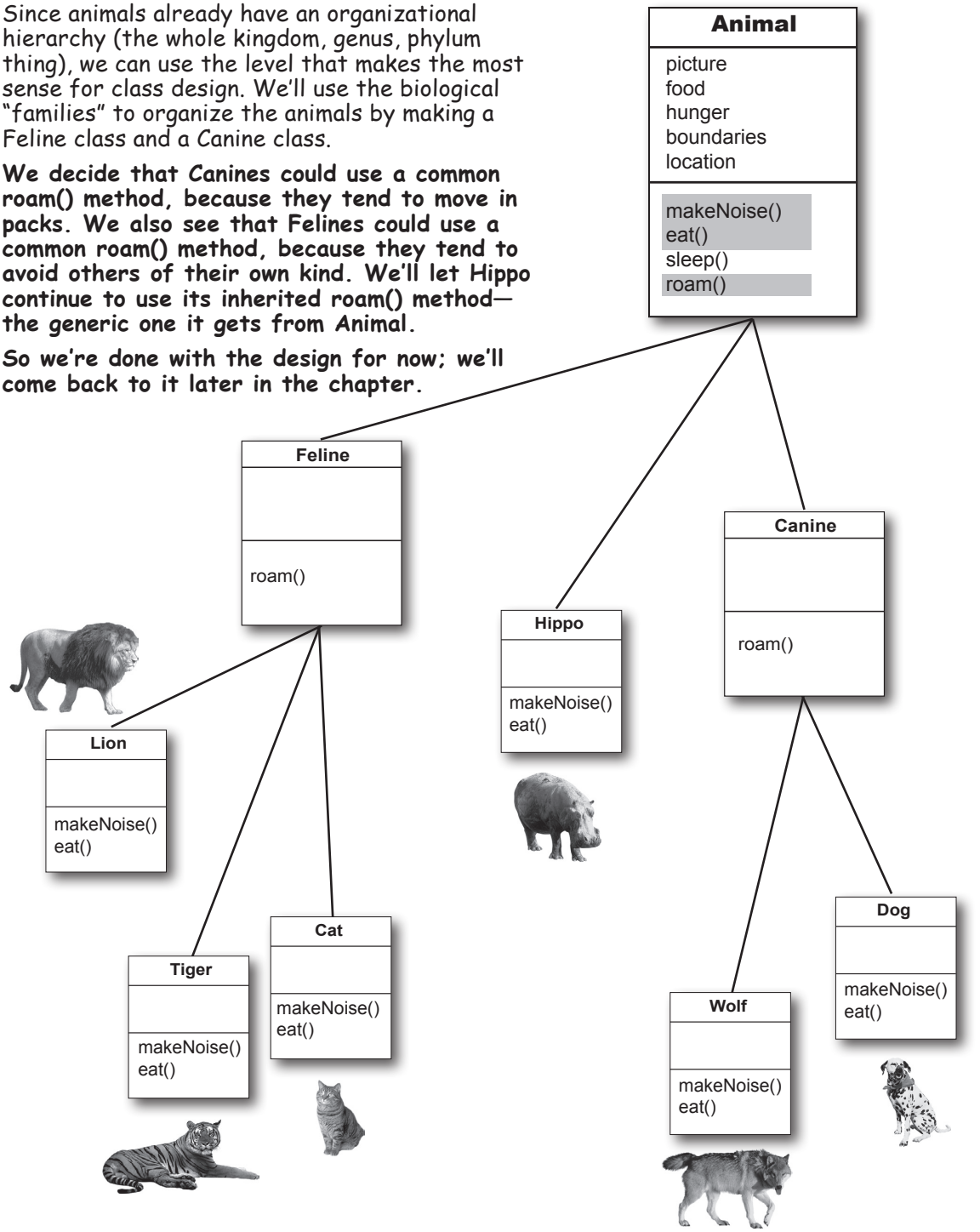


5 Finish the class hierarchy

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?

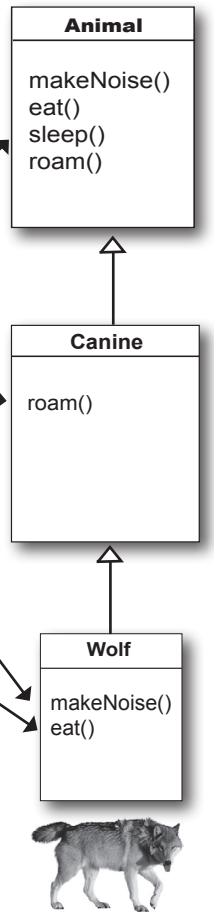
Make a new Wolf object
```
Wolf w = new Wolf();
```

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.

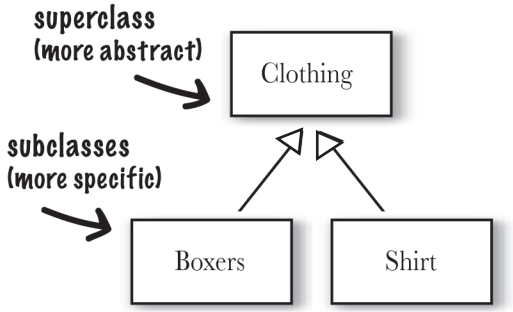
In other words, ***the lowest one wins!***

"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.

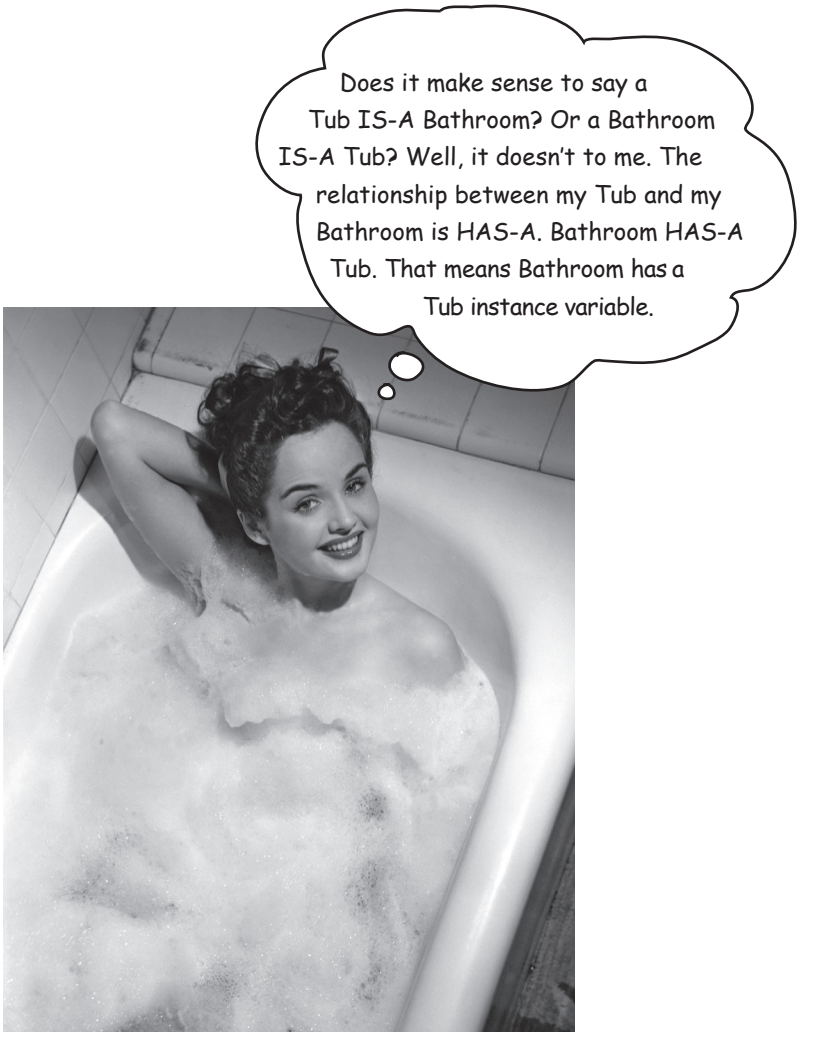
Surgeon IS-A Doctor, still good.

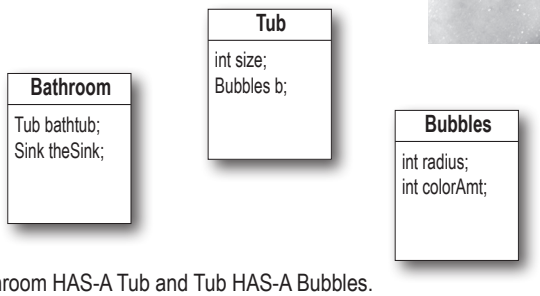
Tub extends Bathroom, sounds reasonable.
Until you apply the IS-A test.

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

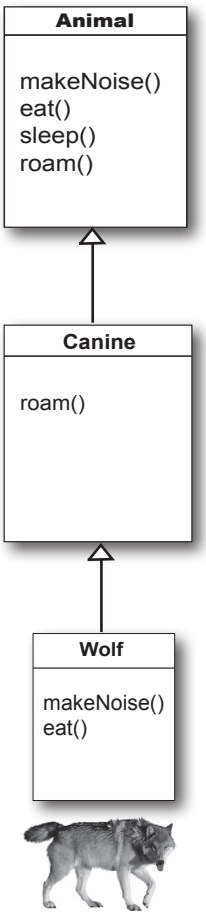
Wolf extends Canine

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.

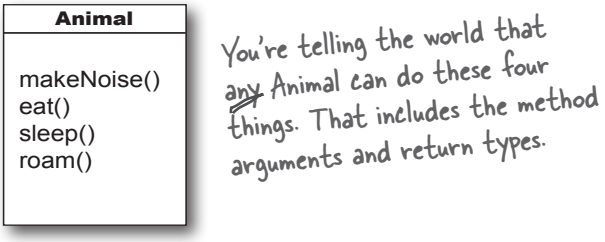
Inheritance lets you guarantee that all classes grouped under a certain supertype have all the methods that the supertype has*

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

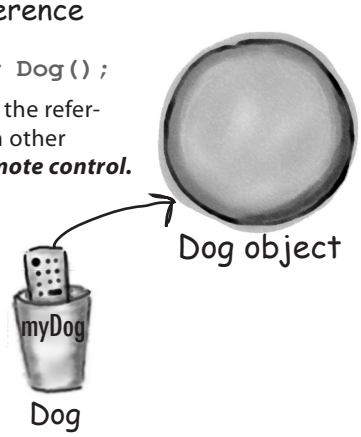
```
Dog myDog = new Dog();
```

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

3 Link the object and the reference

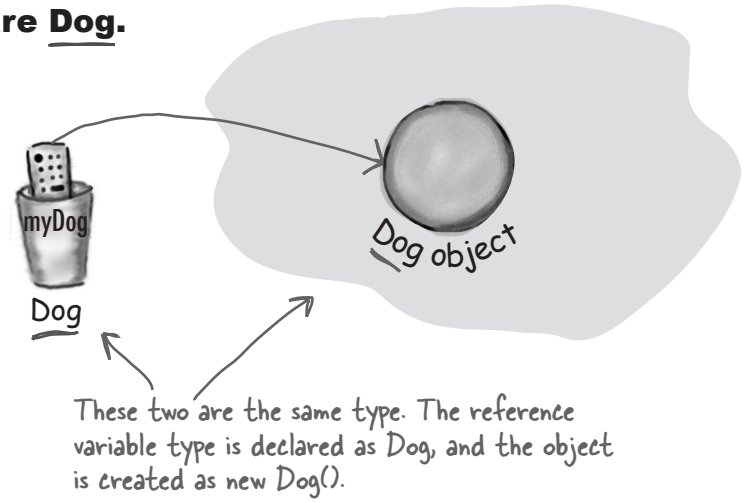
```
Dog myDog = new Dog();
```

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



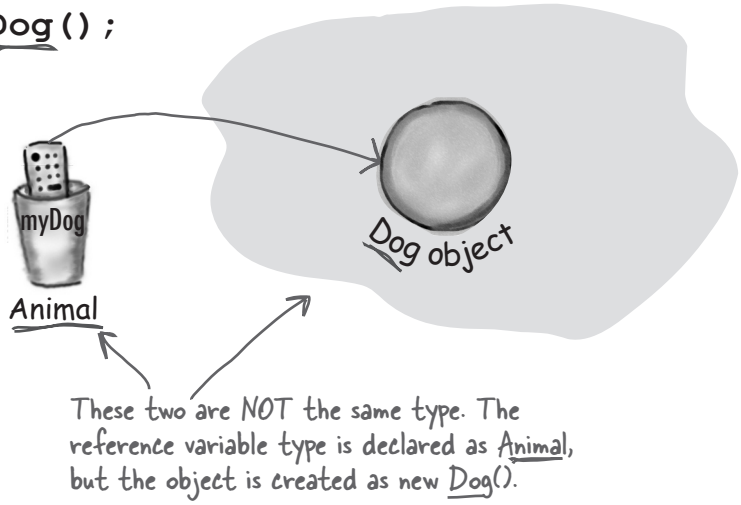
The important point is that the reference type AND the object type are the same.

In this example, both are Dog.



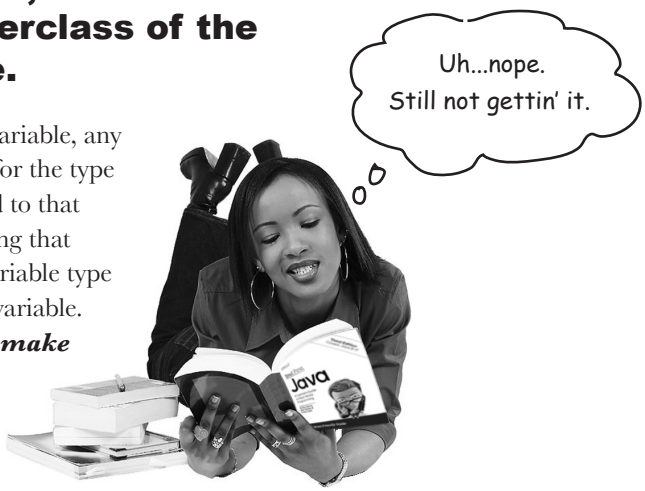
But with polymorphism, the reference type and the object type can be *different*.

```
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!

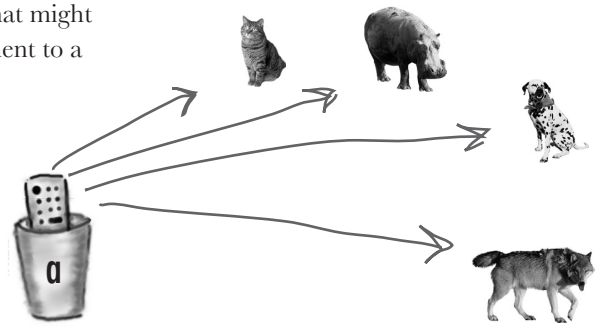
On the first pass through the loop, 'animal' is a Dog, so you get the Dog's eat() method. On the next pass, 'animal' is a Cat, so you get the Cat's eat() method.

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();
  }
}
```

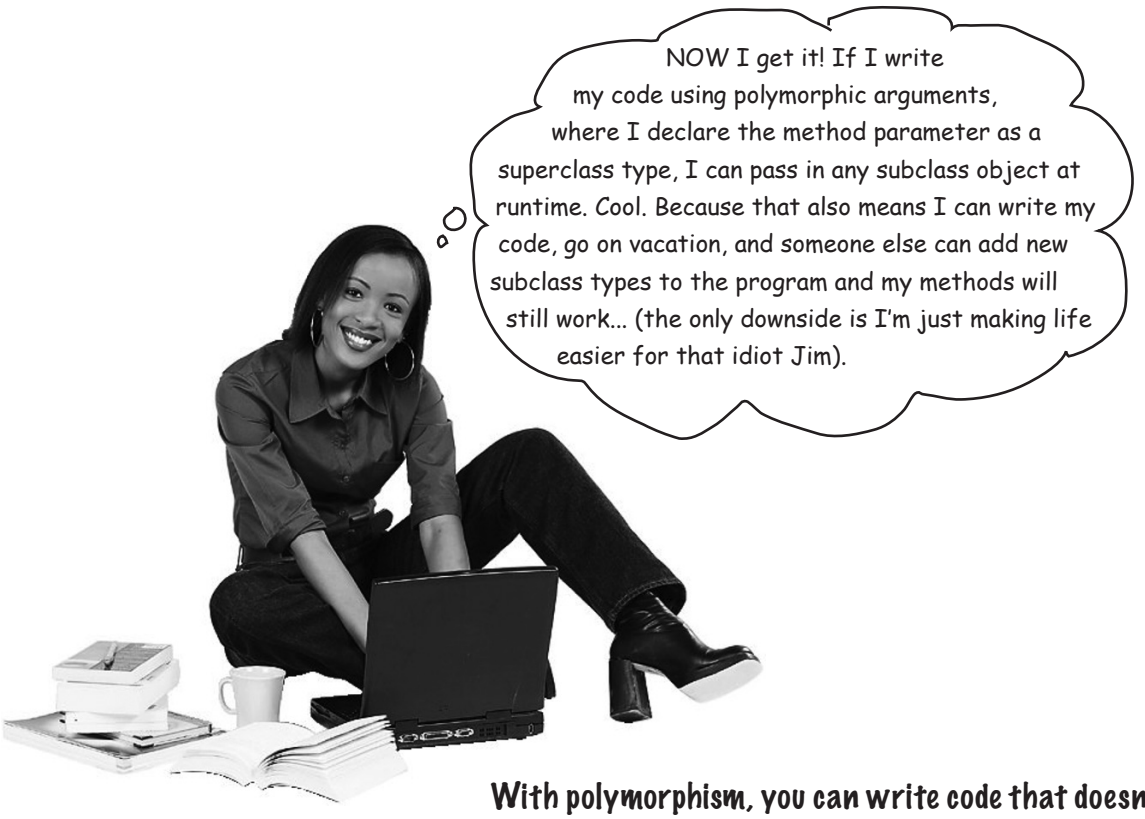
The 'a' parameter can take ANY Animal type as the argument. And when the Vet is done giving the shot, it tells the Animal to makeNoise(), and whatever Animal is really out there on the heap, that's whose makeNoise() method will run.

```
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.

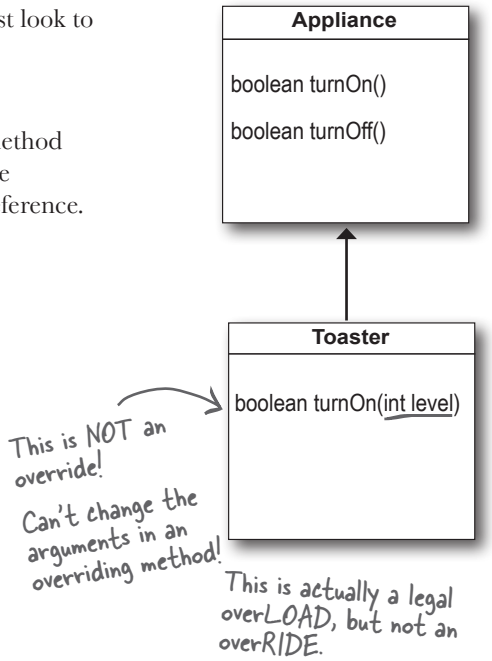
If polymorphism is going to work, the Toaster's version of the overridden method from Appliance has to work at runtime. Remember, the compiler looks at the reference type to decide whether you can call a particular method on that reference.

```
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!***



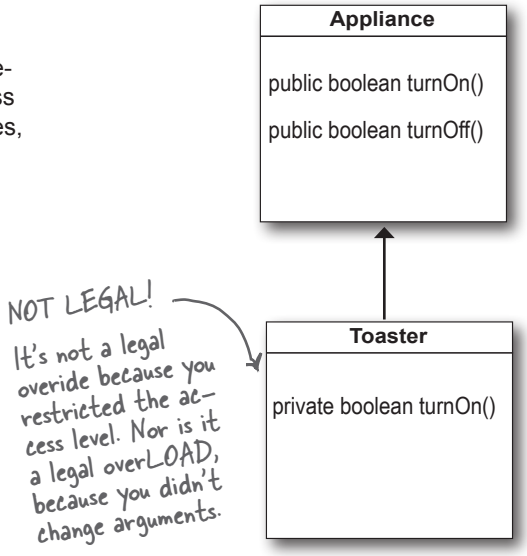
① Arguments must be the same, and return types must be compatible.

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");
   }
}
```

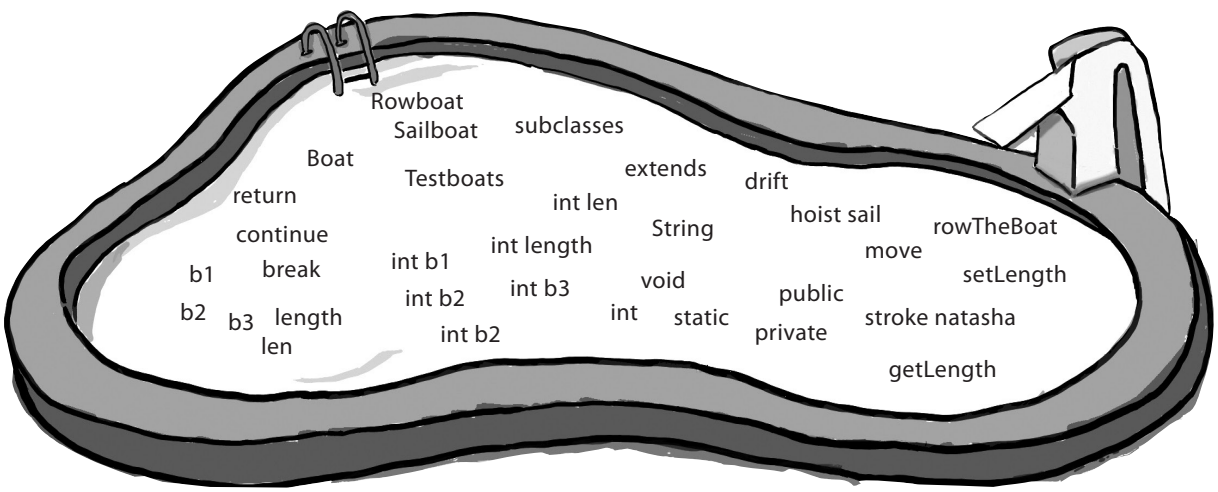
```
public class ________ {
   private int __________ ;
   _______ void _________ ( ______ ) {
      length = len;
   }
   public int getLength() {
      ________ _________ ;
   }
   public ___________ move() {
      System.out.print("___________");
   }
}
```

```
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:

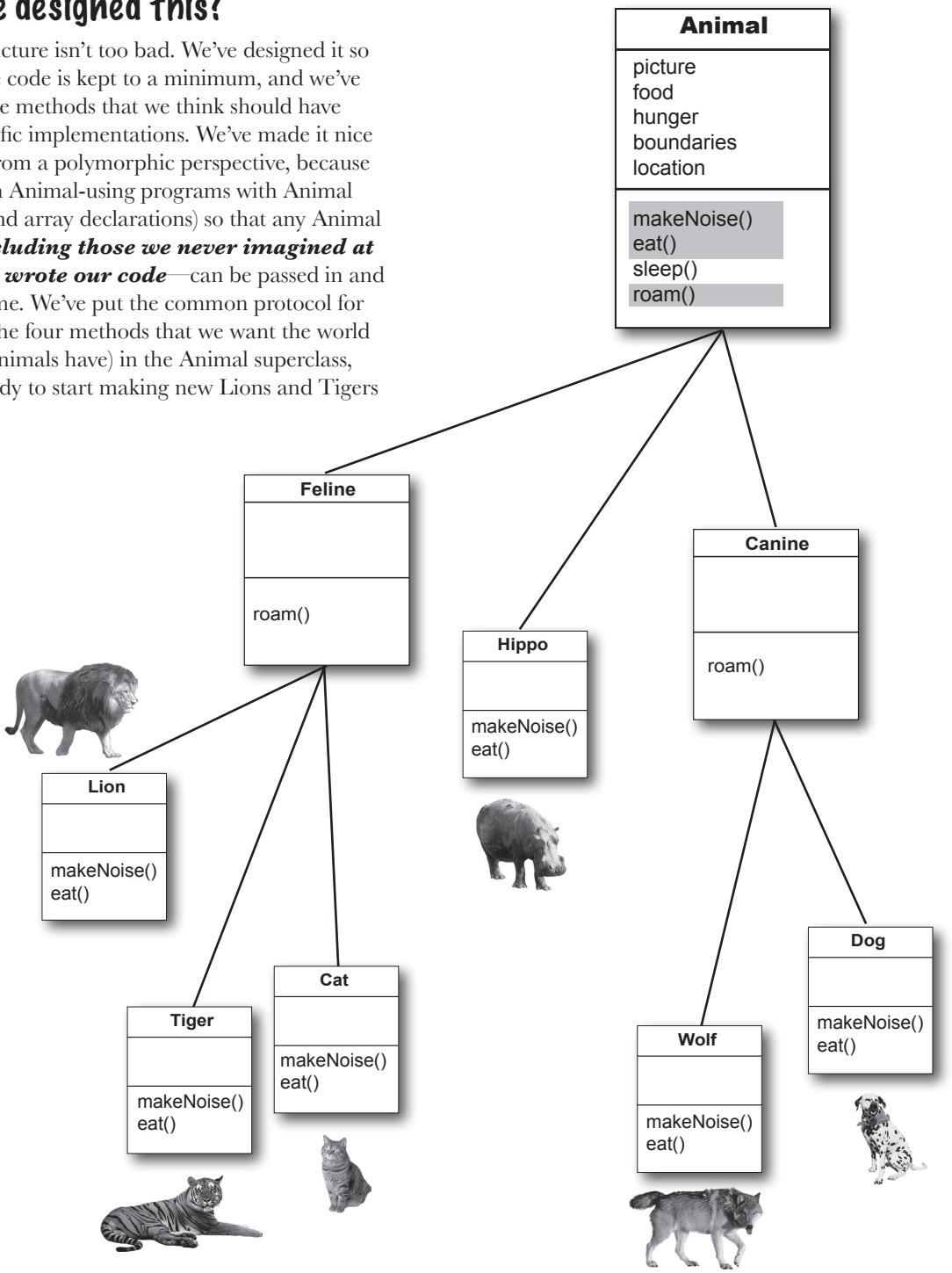
```
drift  drift  hoist sail
```

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.

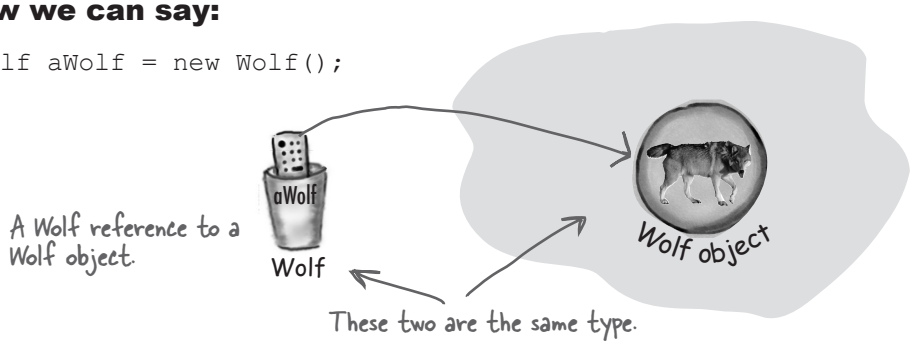
Did we forget about something when we designed this?

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.



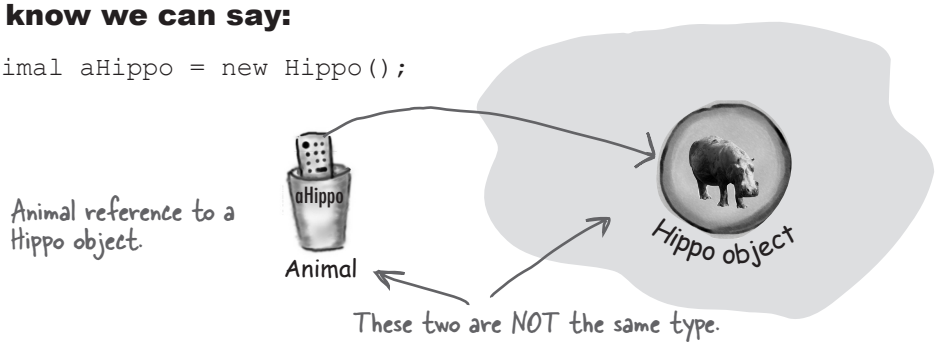
We know we can say:

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



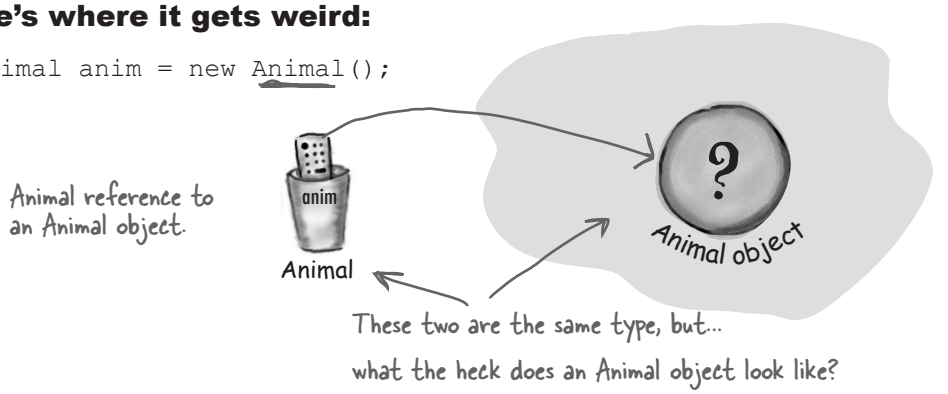
And we know we can say:

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

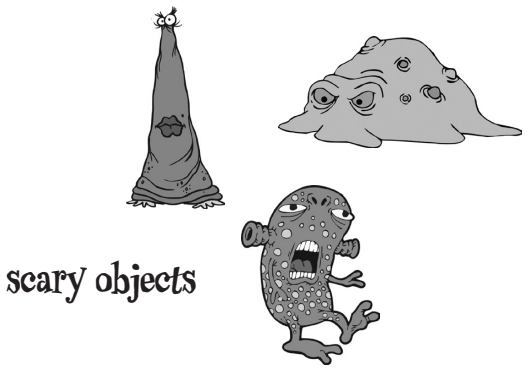


But here's where it gets weird:

```
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.***

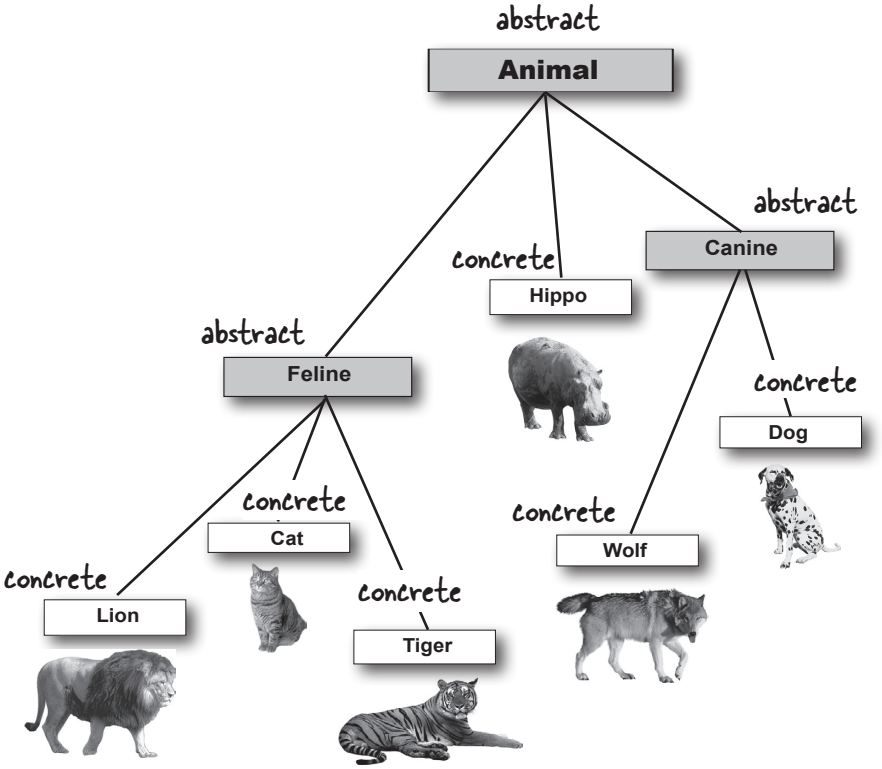
With an abstract class, it's the **instances of a *subclass*** of your abstract class that's doing the work at runtime

*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?

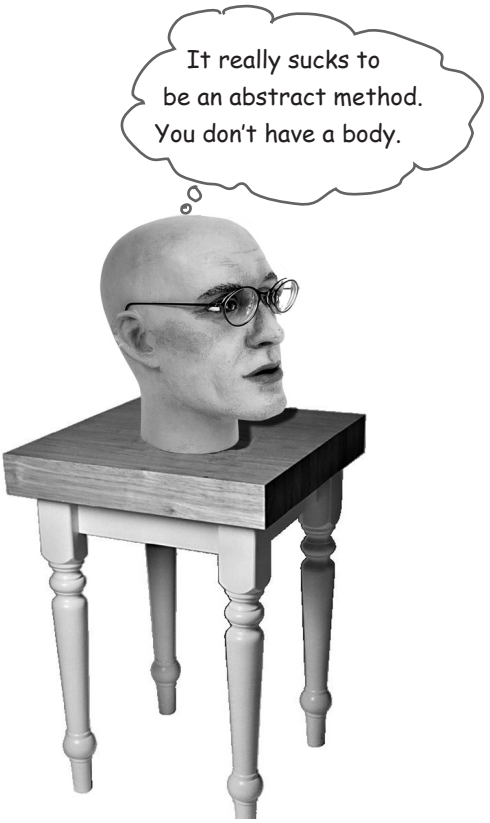
How do you know when a class should be abstract? ***Wine*** is probably abstract. But what about ***Red*** and ***White***? Again probably abstract (for some of us, anyway). But at what point in the hierarchy do things become concrete?

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:

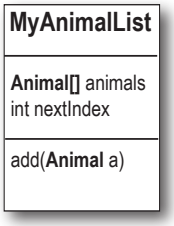
1. Make a separate class, MyCatList, to hold Cat objects. Pretty clunky.

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*.

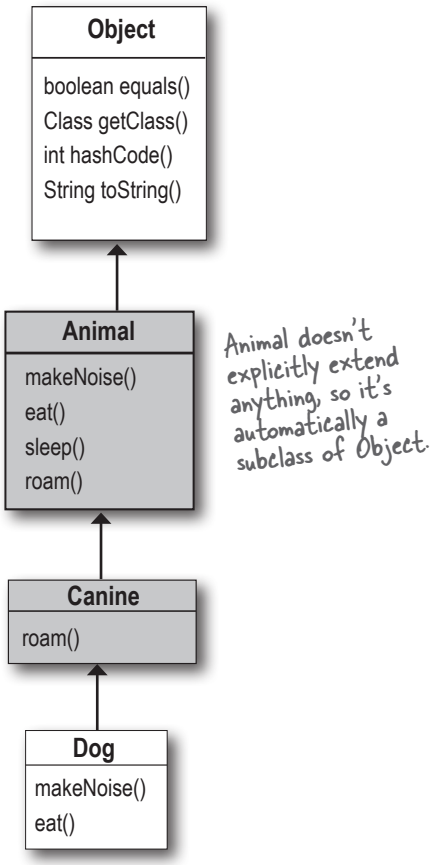
So you were making subclasses of class Object from the very beginning and you didn't even know it. ***Every class you write extends Object,*** without your ever having to say it. But you can think of it as though a class you write looks like this:

```
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.

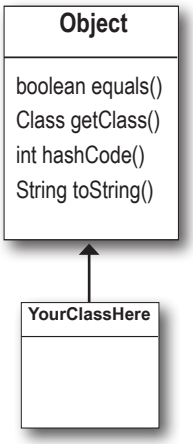
So, since Dog extends Canine, it doesn't *directly* extend Object (although it does extend it indirectly), and the same is true for Canine, but Animal *does* directly extend Object.



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.)

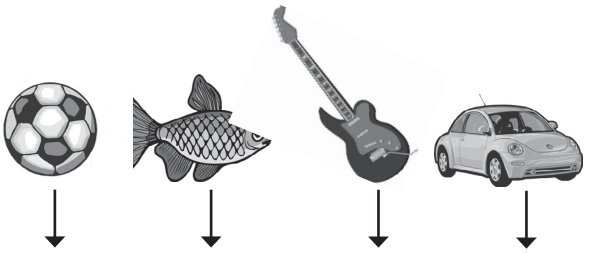
But what happens when you try to get the Dog object and assign it to a Dog reference?

```
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!!

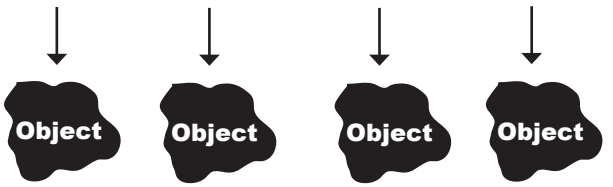
Everything comes out of an ArrayList<Object> as a reference of type Object, regardless of what the actual object is or what the reference type was when you added the object to the list.

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.)

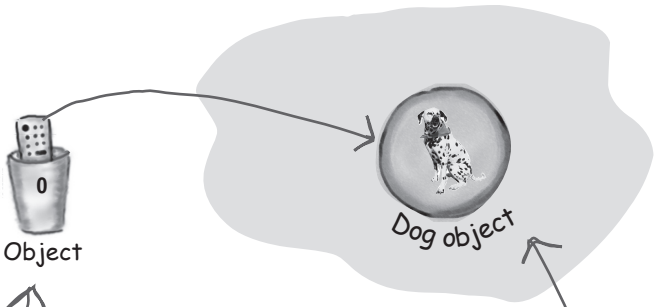
GOOD ☺

```
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.

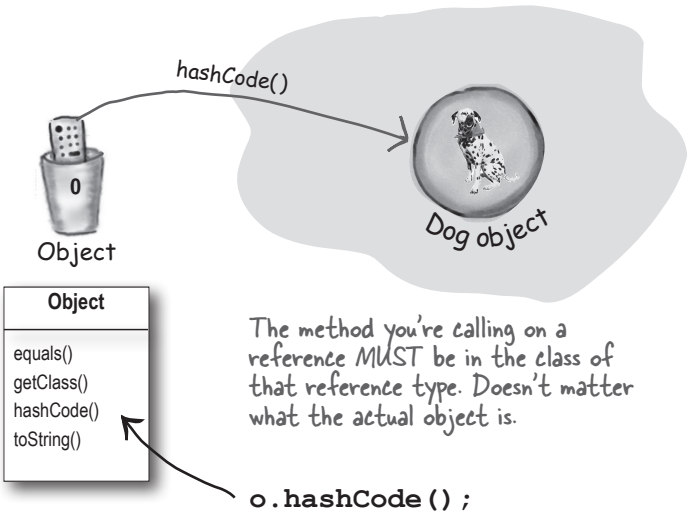
Won't compile!

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.



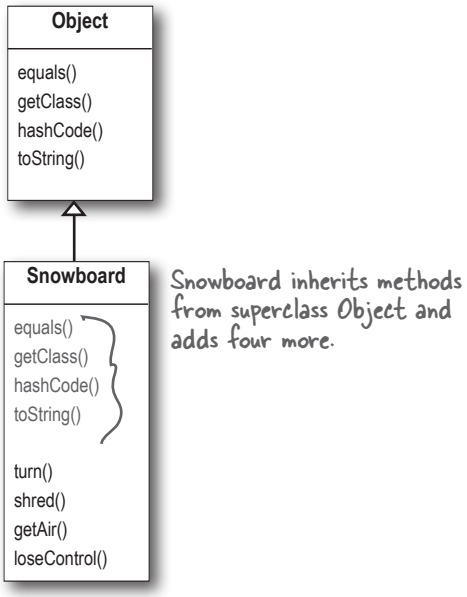
The method you're calling on a reference MUST be in the class of that reference type. Doesn't matter what the actual object is.

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

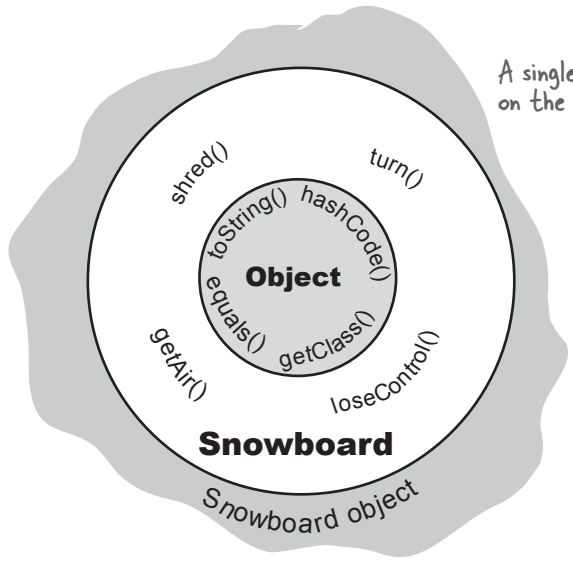
The "o" reference was declared as type Object, so you can call methods only if those methods are in class Object.

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.

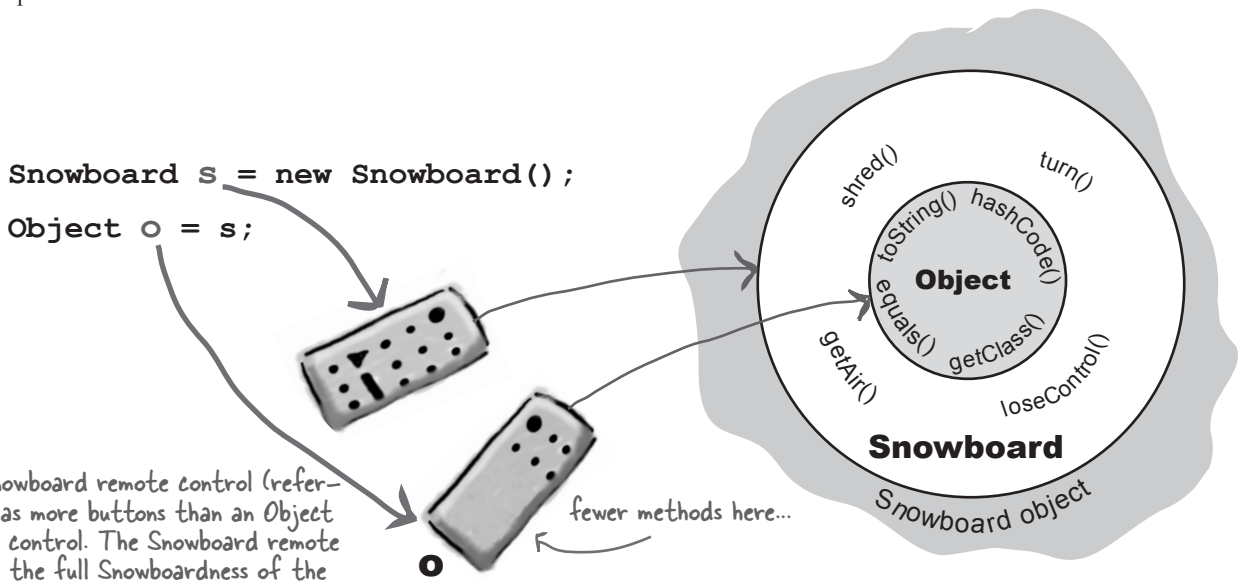
If a reference is like a remote control, the remote control takes on more and more buttons as you move down the inheritance tree. A remote control (reference) of type Object has only a few buttons—the buttons for the exposed methods of class Object. But a remote control of type Snowboard includes all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

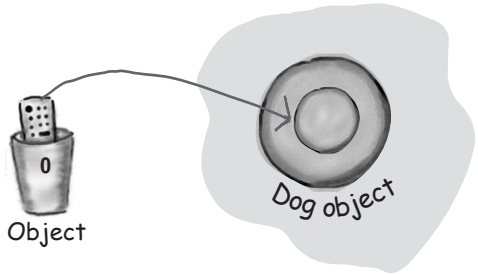
When you put an object in an ArrayList<Object>, you can treat it only as an Object, regardless of the type it was when you put it in.

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

That means you get an *Object* remote control.



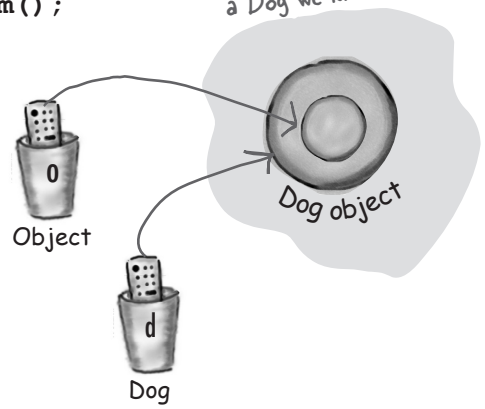
Casting an object reference back to its *real* type.



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;
}
```

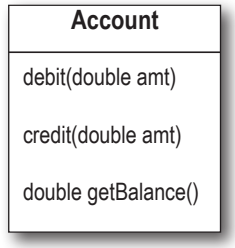
So now you've seen how much Java cares about the methods in the class of the reference variable.

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.

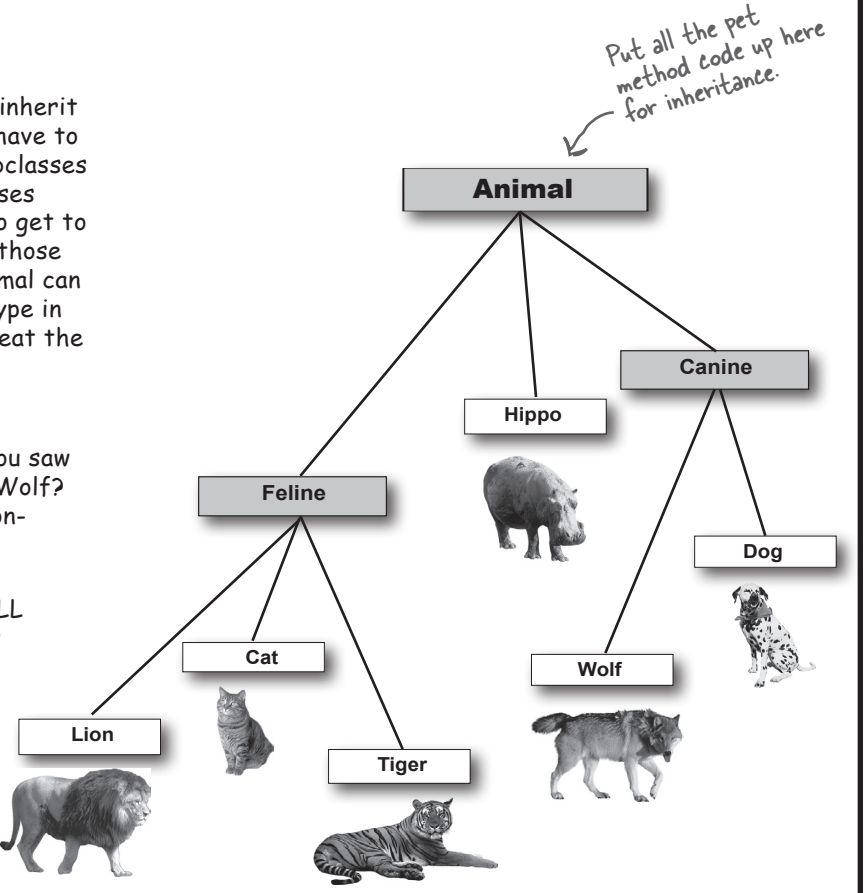
Pros:

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.

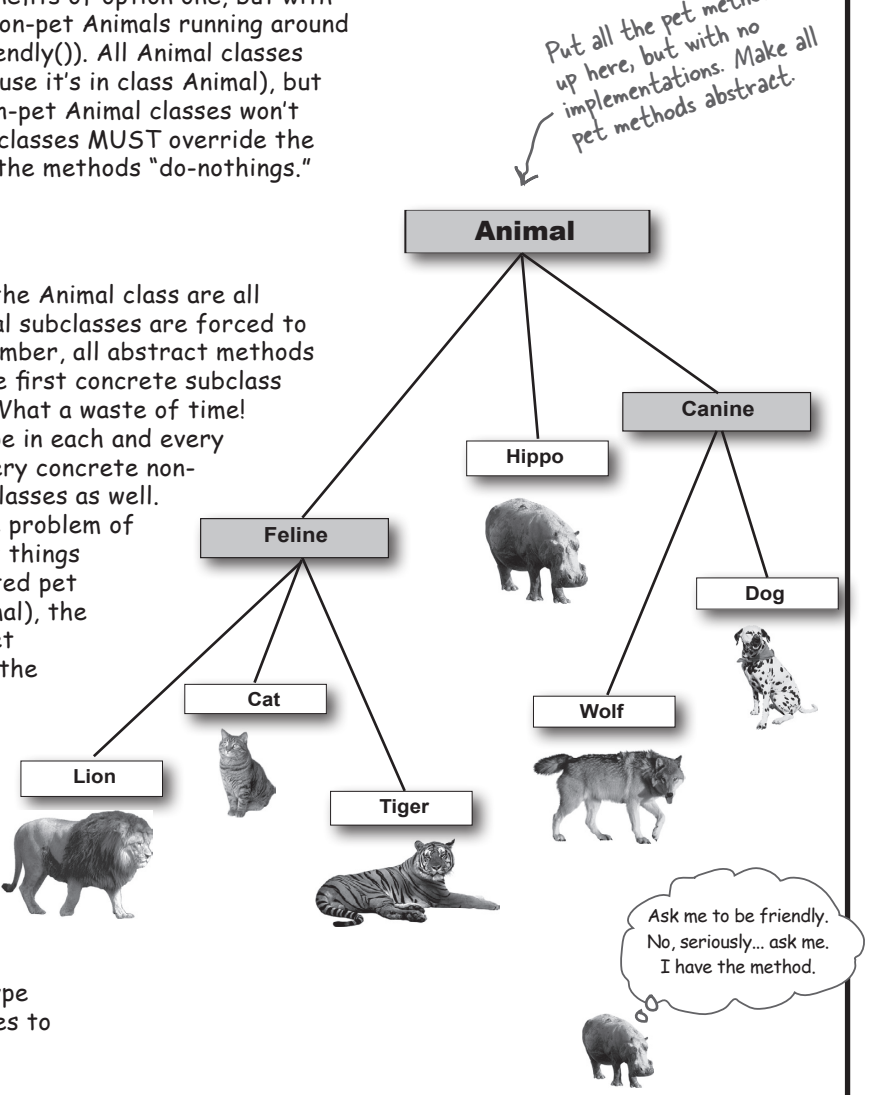
Pros:

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:

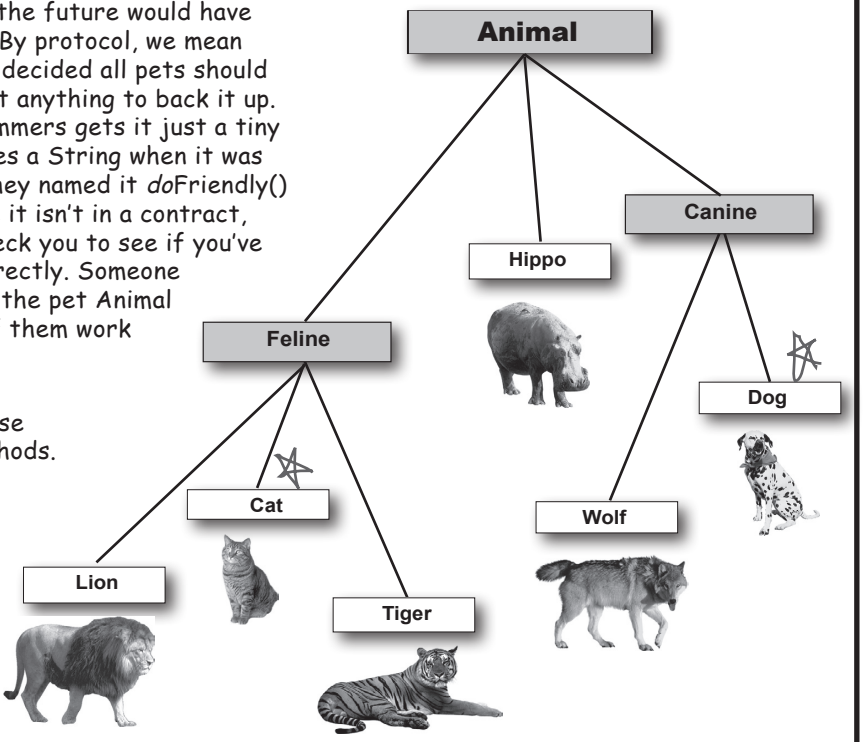
No more worries about Hippos greeting you at the door or licking your face. The methods are where they belong, and ONLY where they belong. Dogs can implement the methods and Cats can implement the methods, but nobody else has to know about them.

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.

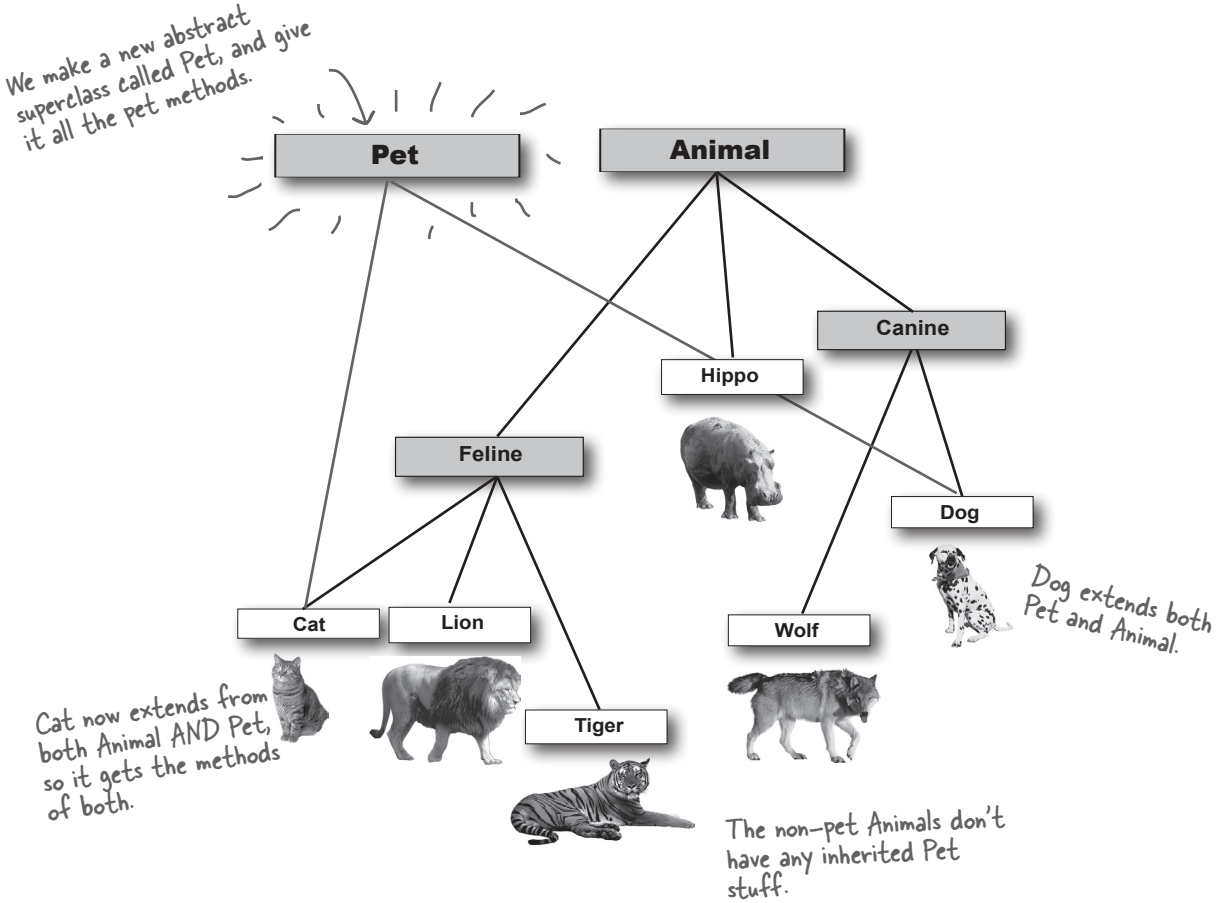
Put the pet methods ONLY in the Animal classes that can be pets, instead of in Animal.



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.



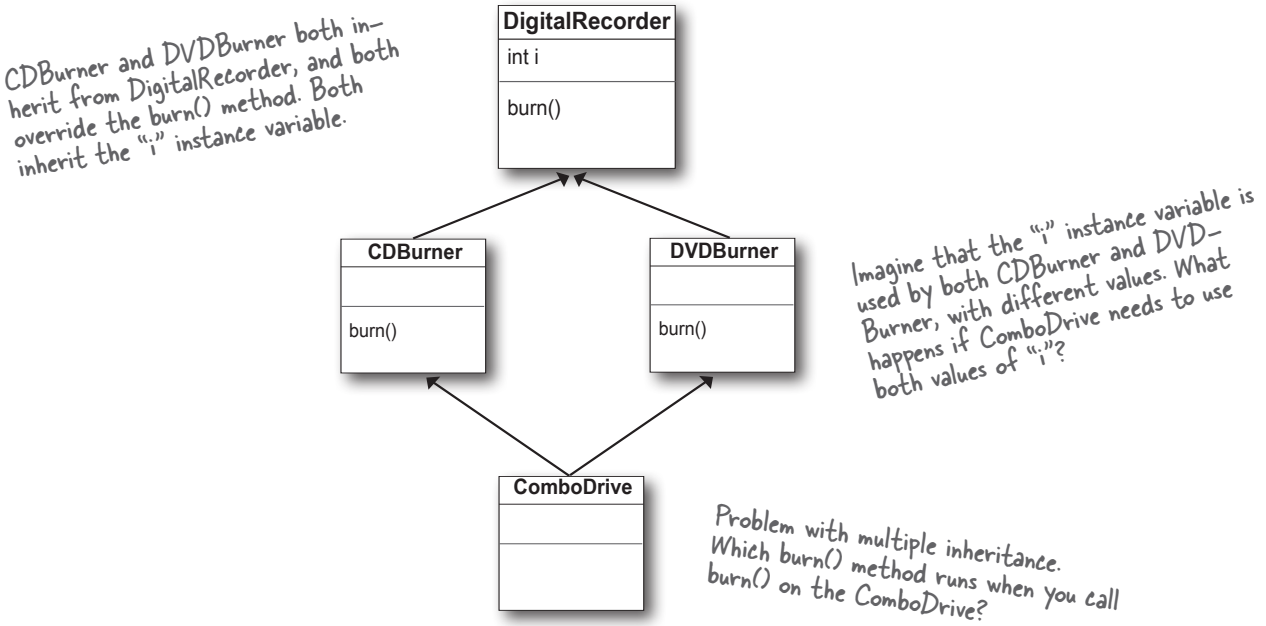
There's just one problem with the "two superclasses" approach...

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



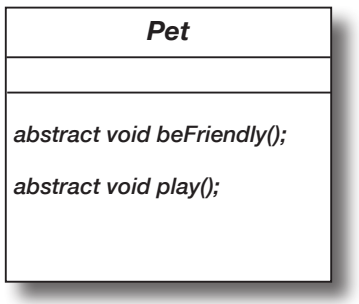
A language that allows the Deadly Diamond of Death can lead to some ugly complexities, because you have to have special rules to deal with the potential ambiguities. And extra rules means extra work for you both in *learning* those rules and watching out for those "special cases." Java is supposed to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

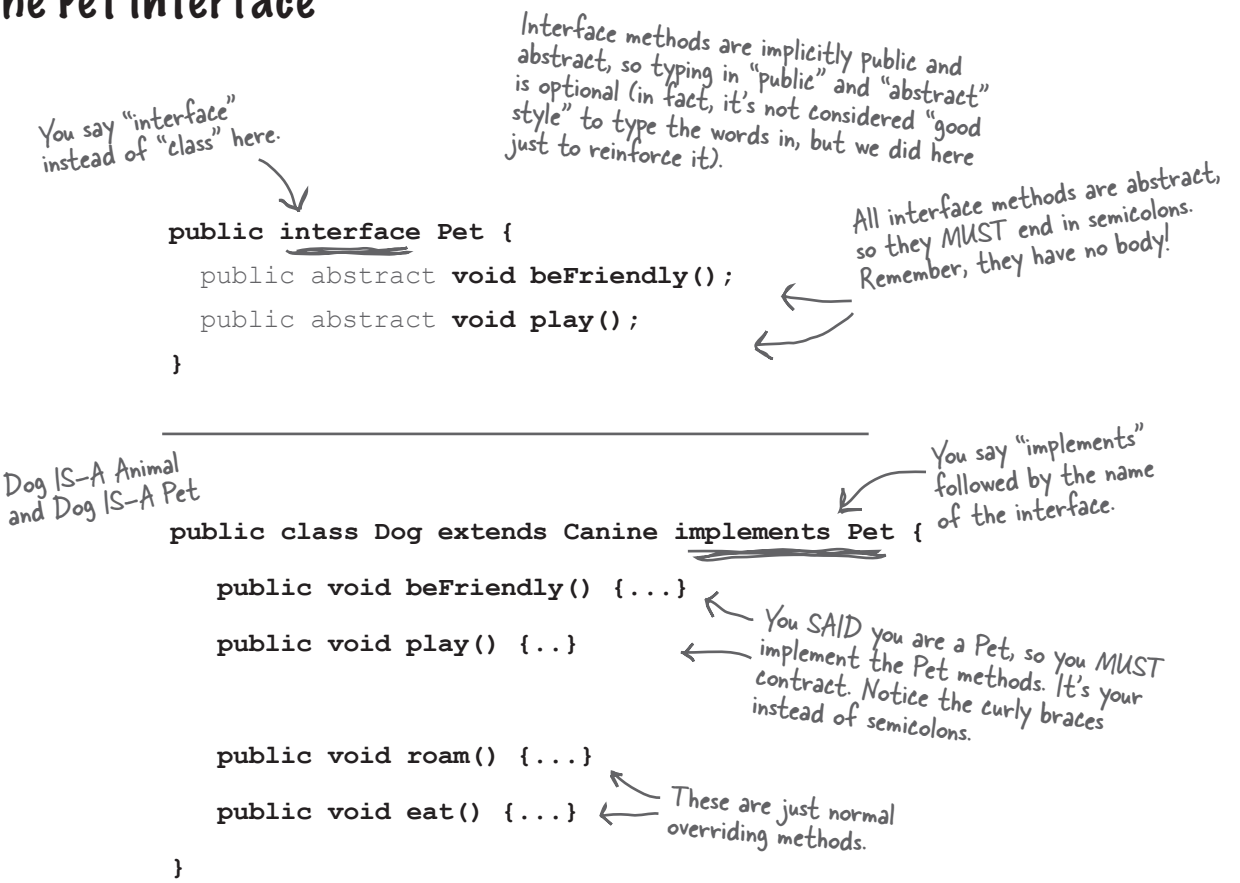
Use the keyword "interface" instead of "class."

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?

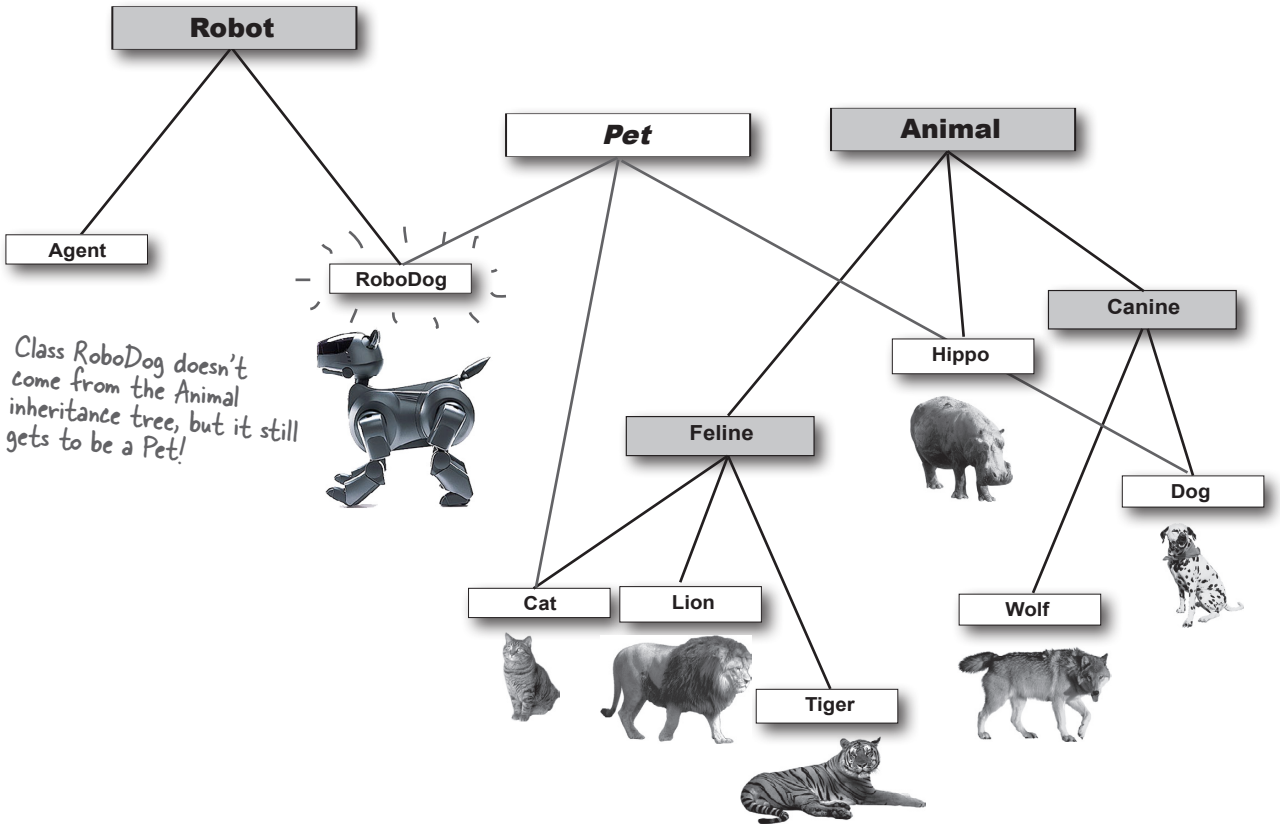
A: Well, actually...there are cases where interfaces can have implementation code (static and default methods, for example), but we're not going to go into them here.

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.



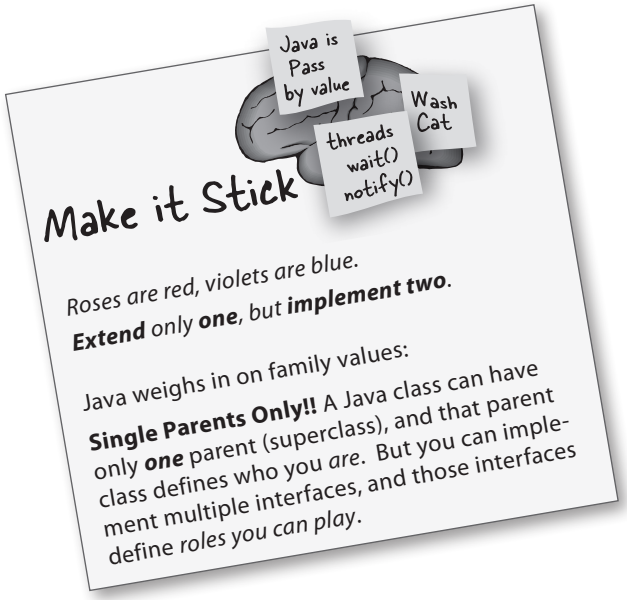
When you use a *class* as a polymorphic type (like an array of type Animal or a method that takes a Canine argument), the objects you can stick in that type must be from the same inheritance tree. But not just anywhere in the inheritance tree; the objects must be from a class that is a subclass of the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

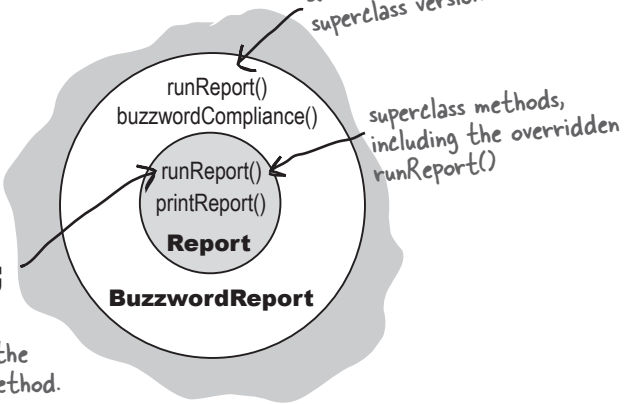
```
abstract class Report {
  void runReport() {
    // set up report
  }
  void printReport() {
    // generic printing
  }
}


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 { }
```

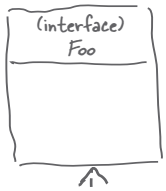
3.
```
public abstract class Muffie implements Whuffie { }
public class Fluffie extends Muffie { }
public interface Whuffie { }
```

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

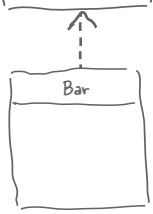
5.
```
public class Gamma extends Delta implements Epsilon { }
public interface Epsilon { }
public interface Beta { }
public class Alpha extends Gamma implements Beta { }
public class Delta { }
```

What's the Picture ?

1.



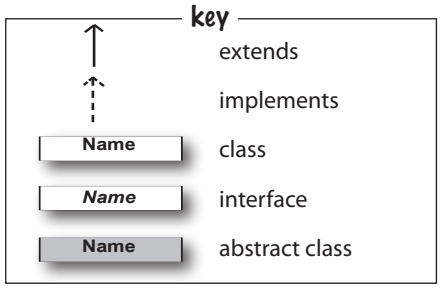
2.



3.

4.

5.

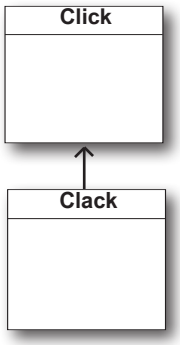


Answers on page 235.

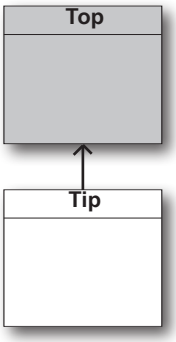
On the left you'll find sets of class diagrams. Your job is to turn these into valid Java declarations. We did number 1 for you (and it was a tough one).

Given:

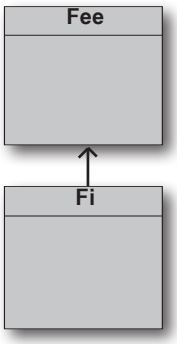
1.



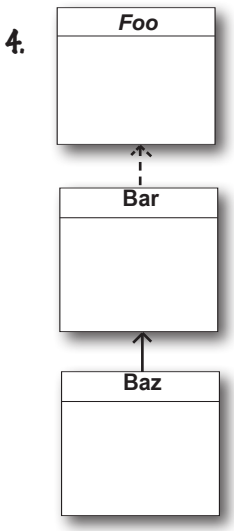
2.



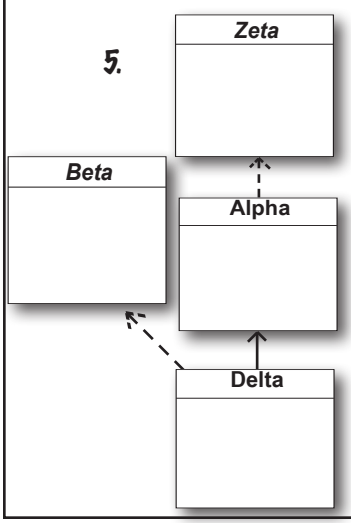
3.



4.



5.



What's the Declaration ?

1. public class Click { }

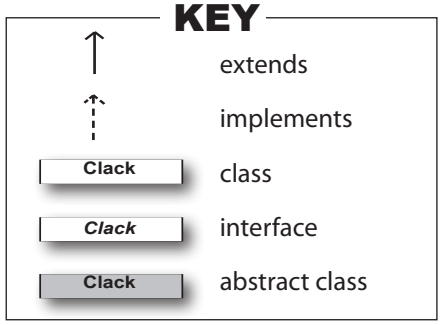
   public class Clack extends Click { }

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.

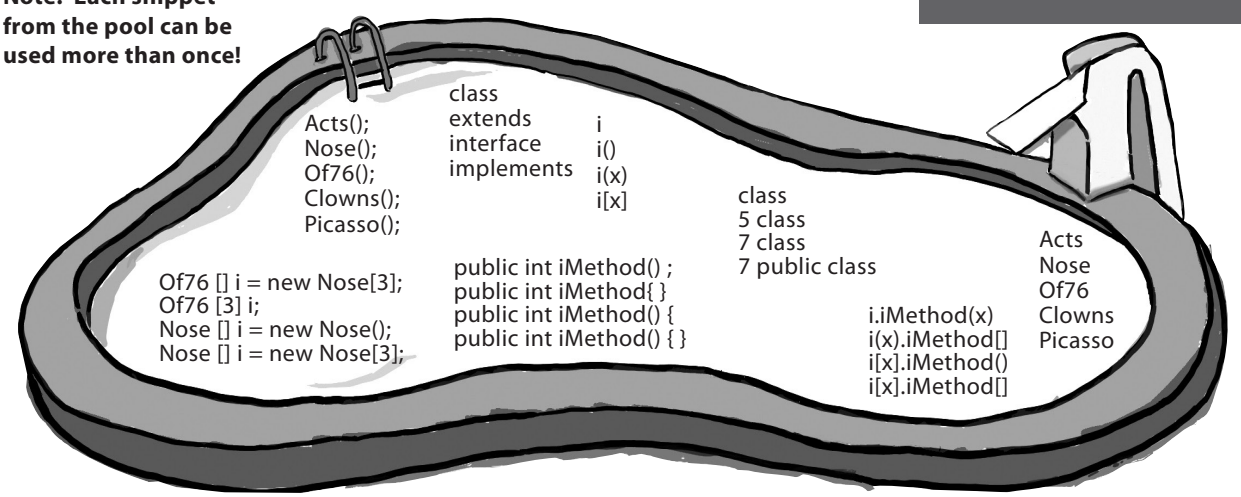
```
_____________ Nose {

  ____________________________

}

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!

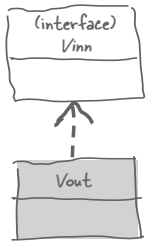


Answers on page 236.

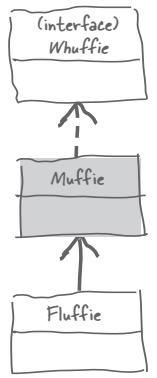
Exercise Solutions

What's the Picture? (from page 232)

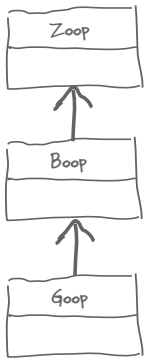
2.



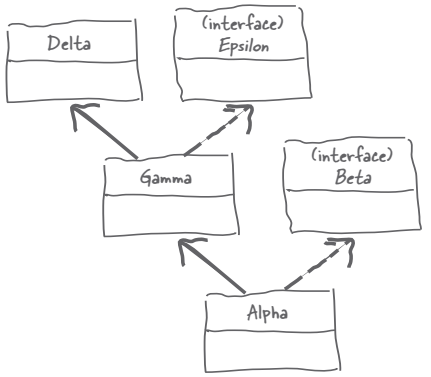
3.



4.



5.



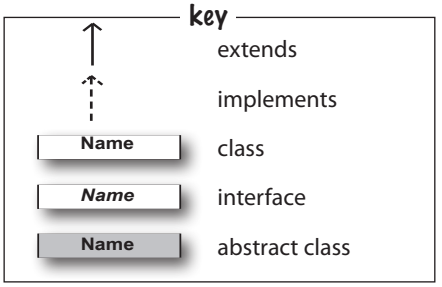
What's the Declaration? (from page 233)

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

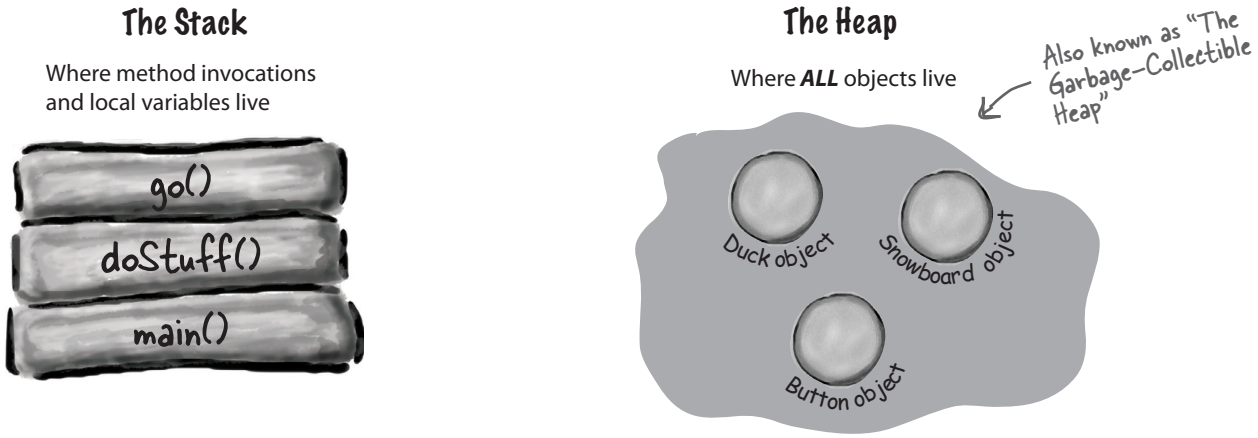
Objects are born and objects die. You're in charge of an object's lifecycle. You decide when and how to **construct** it. You decide when to **destroy** it. Except you don't actually *destroy* the object yourself, you simply *abandon* it. But once it's abandoned, the heartless **Garbage Collector (gc)** 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;
}
```

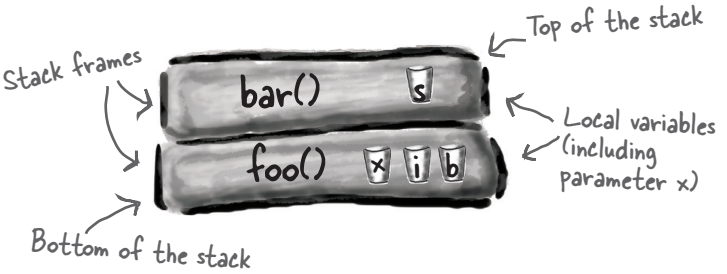
The parameter x and the variables i and b are all local variables.

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.

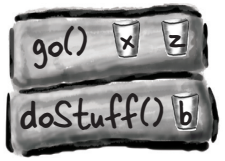
```
public void doStuff() {
  boolean b = true;
  go(4);
}

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*).

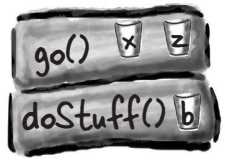
1. Code from another class calls **doStuff()**, and **doStuff()** goes into a stack frame at the top of the stack. The boolean variable named "**b**" goes on the **doStuff()** stack frame.

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()**.

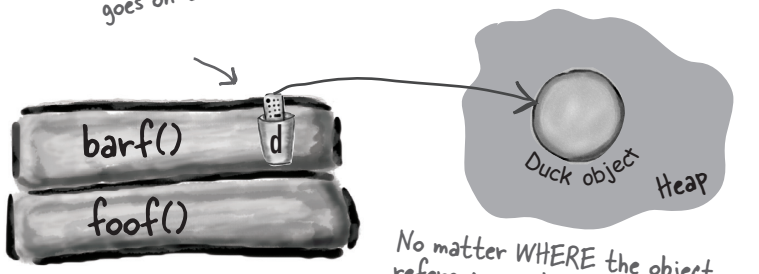


What about local variables that are objects?

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.

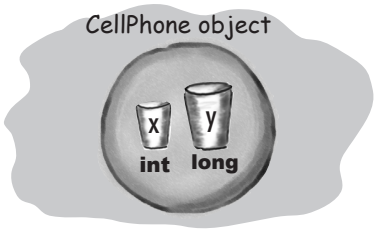
When the new object has instance variables that are object references rather than primitives, the real question is: does the object need space for all of the objects it holds references to? The answer is, *not exactly*. No matter what, Java has to make space for the instance variable *values*. But remember 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.

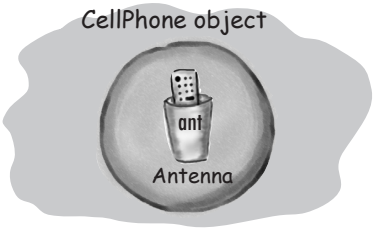
```
private Antenna ant;
```

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();
```

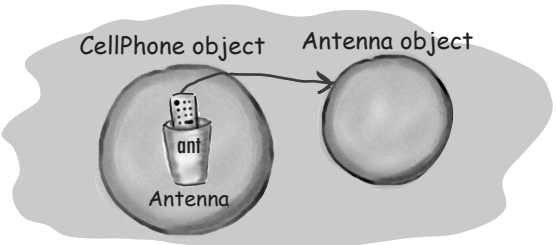


Object with two primitive instance variables. Space for the variables lives in the object.



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.

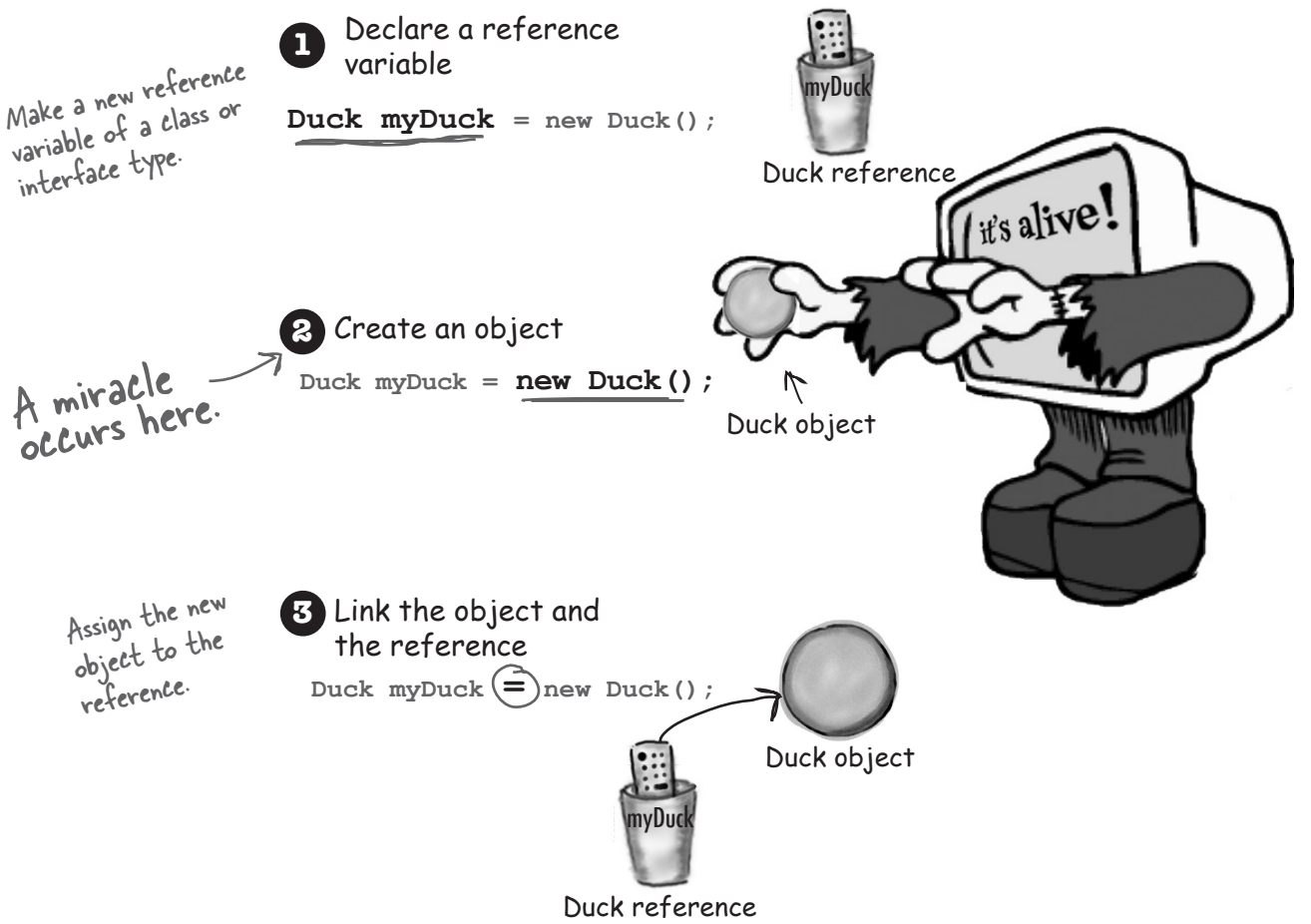
```
public class CellPhone {
  private Antenna ant = new Antenna();
}
```

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?

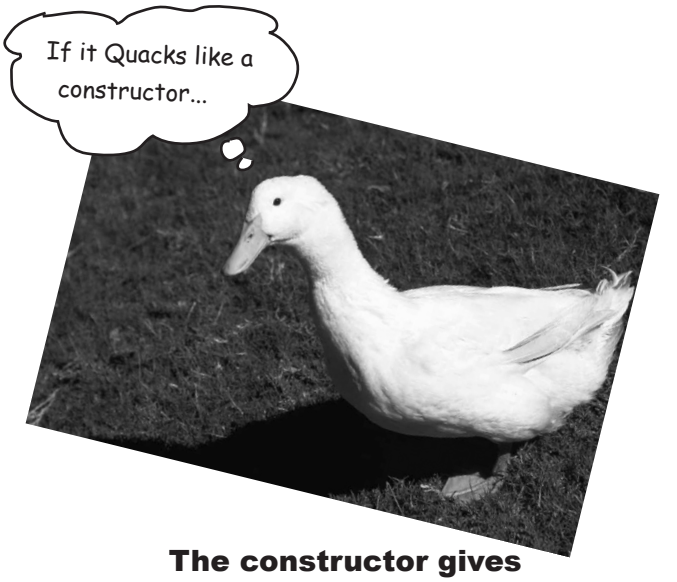
Where's the return type? If this were a method, you'd need a return type between "public" and "Duck()."

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.

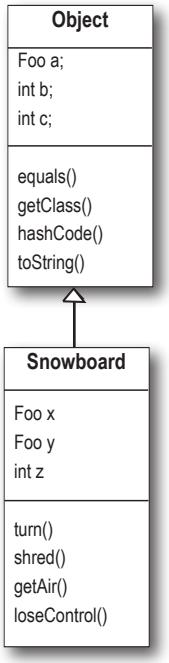
Q: How could a private constructor ever be useful? Nobody could ever call it, so nobody could ever make a new object!

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

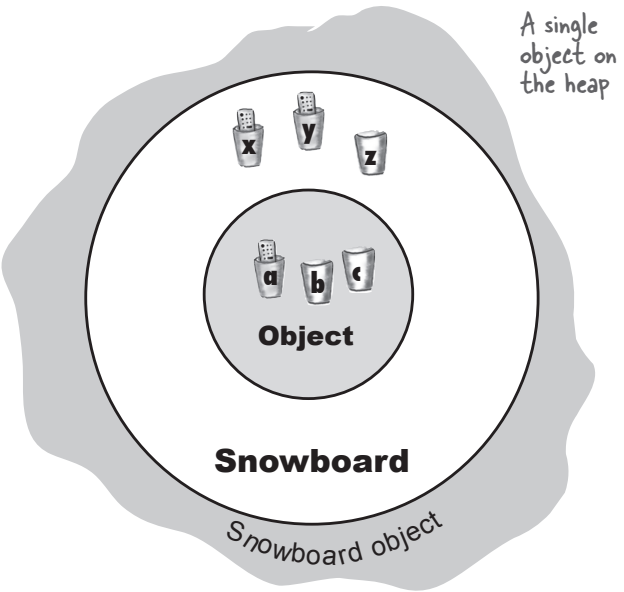
Here's where it gets fun. Remember in the previous chapter we looked at the Snowboard object wrapping around an inner core representing the Object portion of the Snowboard class? The Big Point there was that every object holds not just its *own* declared instance variables, but also *everything from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

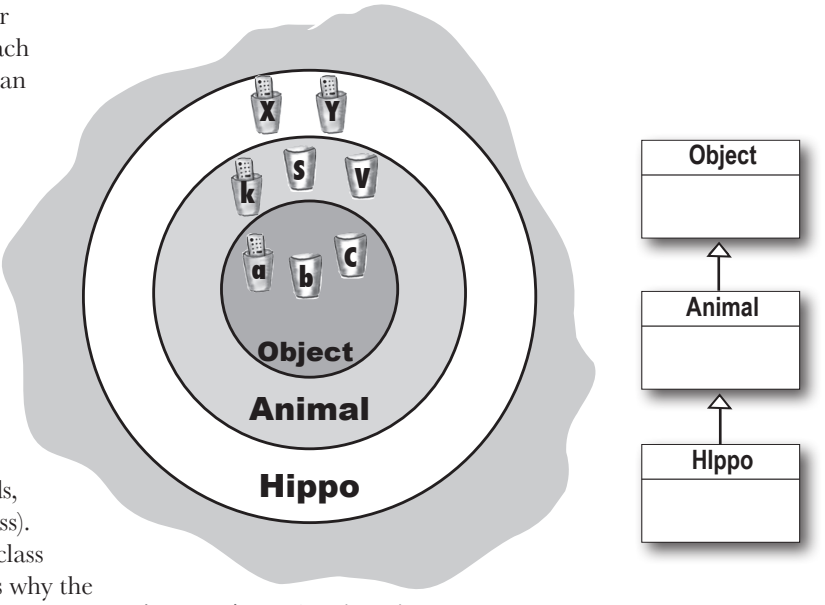
That means every superclass has a constructor (because every class has a constructor), and each constructor up the hierarchy runs at the time an object of a subclass is created.

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.

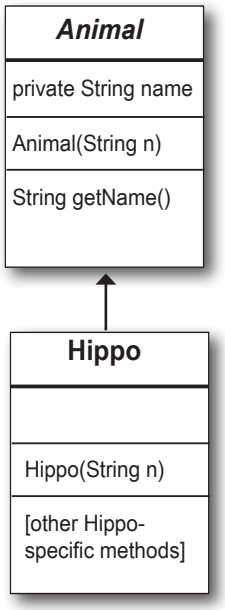
☑
```
public Boop(int i) {
    size = i;
}
```

⊘
```
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.

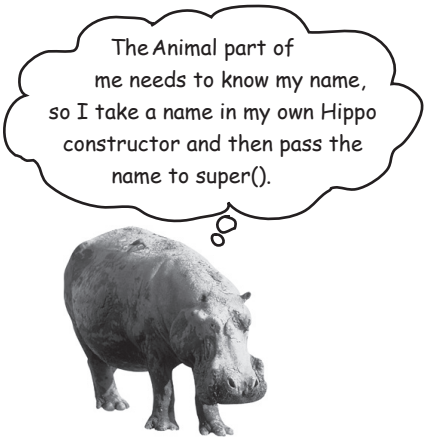
```
public class Hippo extends Animal {
  public Hippo(String name) {
    super(name);
  }
}
```

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();
            ^
```

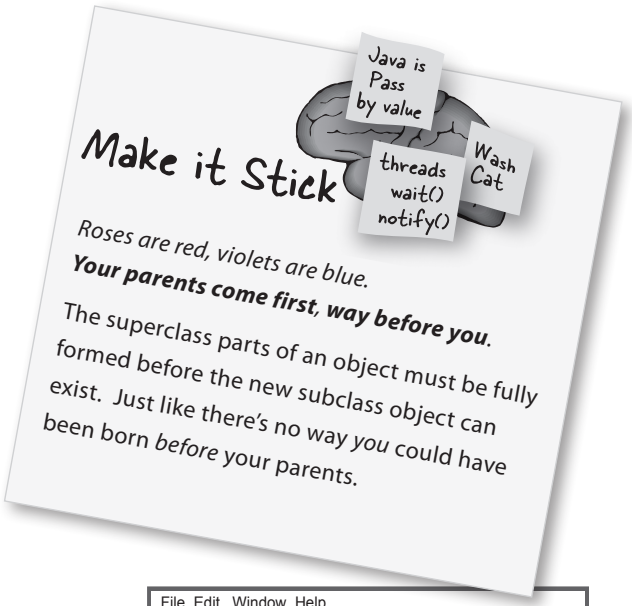
Sharpen your pencil

→ 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.

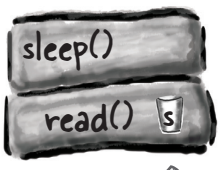
So if an object's life depends on the reference variable's life, how long does a *variable* live?

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*.

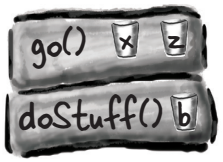
Scope

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***.

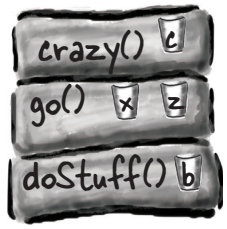
```
1 public void doStuff() {
    boolean b = true;
    go(4);
  }
2 public void go(int x) {
    int z = x + 24;
    crazy();
4   // imagine more code here
  }
  public void crazy() {
3   char c = 'a';
  }
```

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

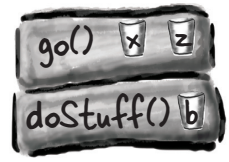
1 ***doStuff()*** **goes on the Stack. Variable "b" is alive and in scope.**



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.'