

## 6 get to know the Java API

# Using the Java Library



**Java ships with hundreds of pre-built 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 pre-built 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.

we still have a **bug**

## 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 SimpleDotComGame
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 SimpleDotComGame
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?

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, he hasn't previously hit that cell. If he has, then we don't want to count it as a hit.

```
public String checkYourself(String stringGuess) {

    int guess = Integer.parseInt(stringGuess);

    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
```

Convert the String to an int.

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").

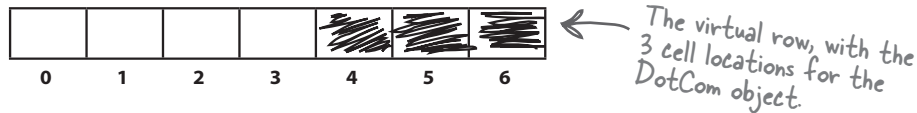
Return the result back to the calling method.

fixing the bug

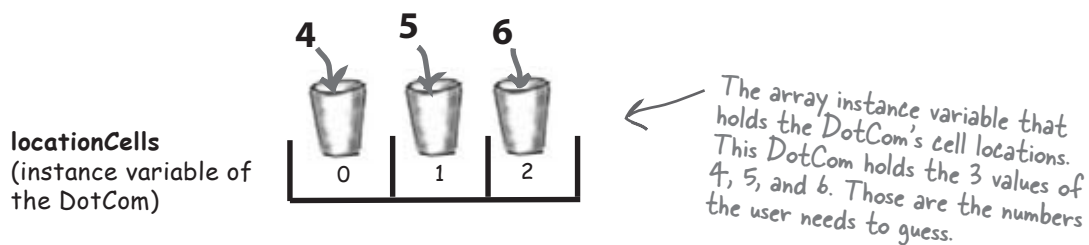
## How do we fix it?

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 7 cells, and a DotCom will occupy three consecutive cells somewhere in that row. This virtual row shows a DotCom placed at cell locations 4, 5 and 6.

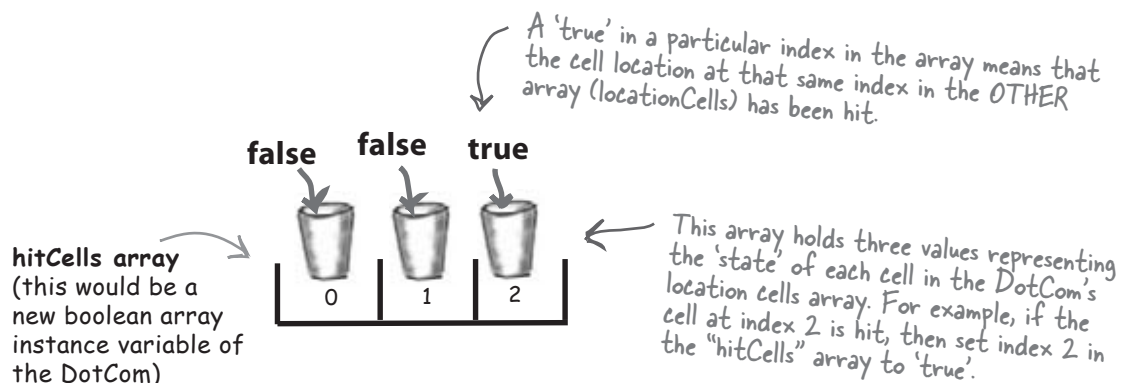


The DotCom has an instance variable—an int array—that holds that DotCom object's cell locations.



### 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.

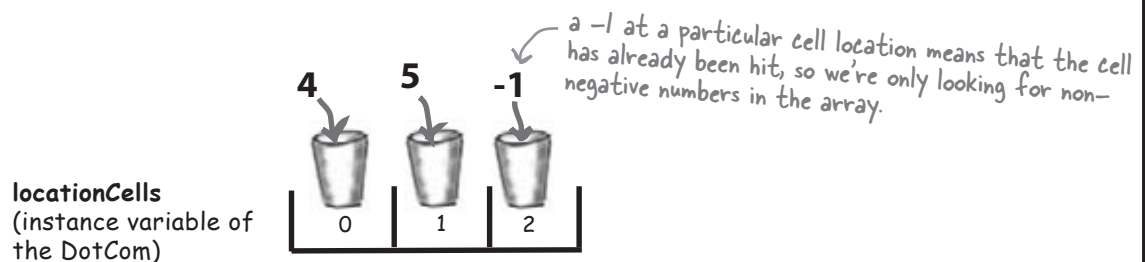


## Option one is too clunky

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...

## prep code

prep code

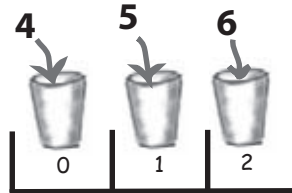
test code

real code

### ③ Option three

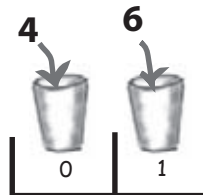
We delete each cell location as it gets hit, and then modify the array to be smaller. Except arrays can't change their size, so we have to make a **new** array and copy the remaining cells from the old array into the new smaller array.

**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 precode 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...



when arrays aren't enough

## 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 Standard Edition (which is what you have unless you're working on the Micro Edition for small devices and believe me, *you'd know*) ships with hundreds of pre-built classes. Just like our Ready-Bake code except that these built-in classes are already compiled.

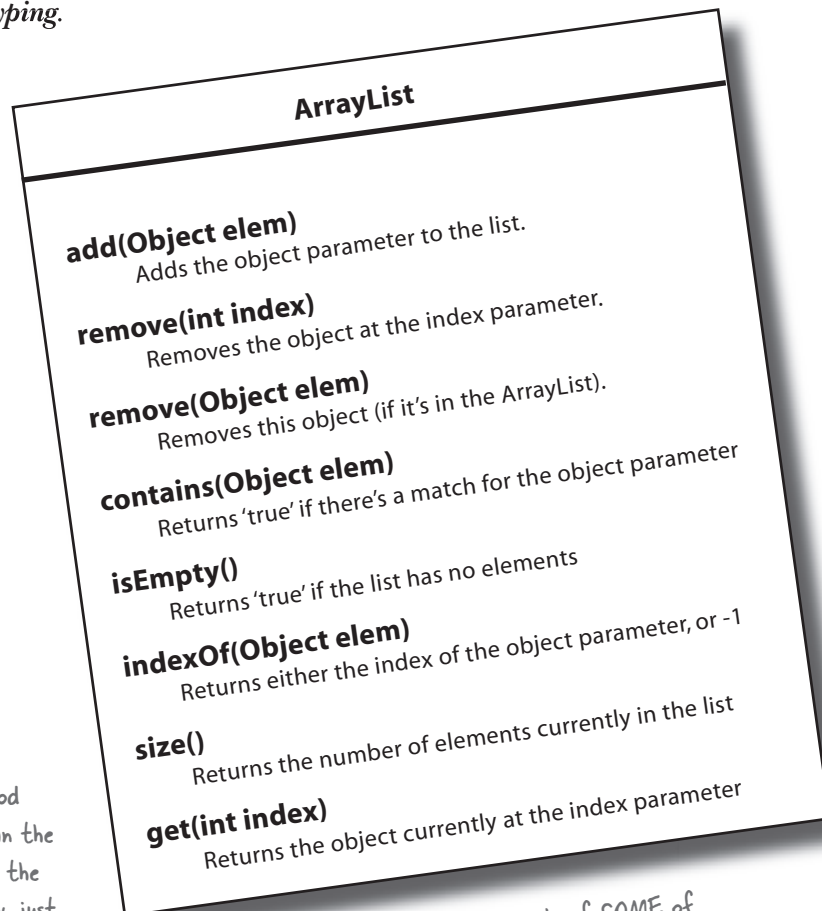
*That means no typing.*

Just use 'em.

One of a gazillion classes in the Java library.

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

(Note: the `add(Object elem)` method actually looks a little stranger than the one we've shown here... we'll get to the real one later in the book. For now, just think of it as an `add()` method that takes the object you want to add.)



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



## Some things you can do with ArrayList

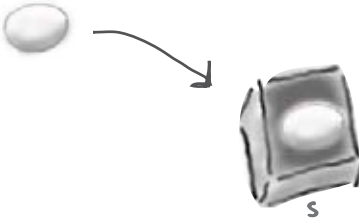
- ① Make one

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

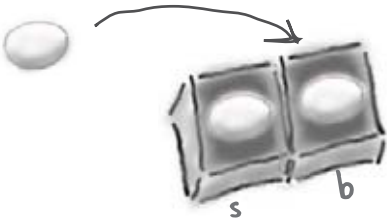
```
Egg s = new Egg();
```



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

```
myList.add(s);
```
- ③ Put another thing in it

```
Egg b = new Egg();
```



The ArrayList grows again to hold the second Egg object.

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

The ArrayList DOES contain the Egg object referenced by 's', so `contains()` returns true
- ⑥ Find out where something is (i.e. its index)


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

ArrayList is zero-based (means first index is 0) and since the object referenced by 'b' 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(s);
```

 Hey look — it shrank!

## when arrays aren't enough



### Sharpen your pencil

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

|  |  |
|--|--|
| <pre>ArrayList&lt;String&gt; myList = new<br/>ArrayList&lt;String&gt;();</pre> | <pre>String [] myList = new String[2];</pre> |
|  |  |
| <pre>String a = new String("whoohoo");<br/>myList.add(a);</pre>                | <pre>String a = new String("whoohoo");</pre> |
|  |  |
| <pre>String b = new String("Frog");<br/>myList.add(b);</pre>                   | <pre>String b = new String("Frog");</pre>    |
|  |  |
| <pre>int theSize = myList.size();</pre>  |  |
|  |  |
| <pre>Object o = myList.get(1);</pre>   |  |
|  |  |
| <pre>myList.remove(1);</pre>   |  |
|  |  |
| <pre>boolean isIn = myList.contains(b);</pre>                                  |  |

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? McJava's?)

**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 as of Java 5.0, 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.

## Comparing ArrayList to a regular array

| ArrayList  | regular array   |
|--|---|
| <pre>ArrayList&lt;String&gt; myList = new<br/>ArrayList&lt;String&gt;();</pre> | <pre>String [] myList = new String[2];</pre>  |
| <pre>String a = new String("whooohoo");<br/>myList.add(a);</pre>               | <pre>String a = new String("whooohoo");<br/>myList[0] = a;</pre>  |
| <pre>String b = new String("Frog");<br/>myList.add(b);</pre>                   | <pre>String b = new String("Frog");<br/>myList[1] = b;</pre>  |
| <pre>int theSize = myList.size();</pre>  | <pre>int theSize = myList.length;</pre>   |
| <pre>Object o = myList.get(1);</pre>   | <pre>String o = myList[1];</pre>  |
| <pre>myList.remove(1);</pre>   | <pre>myList[1] = null;</pre>  |
| <pre>boolean isIn = myList.contains(b);</pre>                                  | <pre>boolean isIn = false;<br/><br/>    for (String item : myList) {<br/><br/>        if (b.equals(item)) {<br/><br/>            isIn = true;<br/><br/>            break;<br/><br/>        }<br/><br/>    }</pre> |

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 a 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 in Java 5.0 are parameterized.

We just said that unlike arrays, ArrayLists have no special syntax. But they *do* use something special that was added to Java 5.0 Tiger—*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”.

Prior to Java 5.0, there was no way to declare the *type* of things that would go in the ArrayList, so to the compiler, all ArrayLists were simply heterogeneous collections of objects. But now, 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 the Collections chapter, 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.

## the buggy DotCom code

prep code

test code

real code

## Let's fix the DotCom code.

Remember, this is how the buggy version looks:

```
public class DotCom {
```

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

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

```
    public String checkYourself(String stringGuess) {  
        int guess = Integer.parseInt(stringGuess);  
        String result = "miss";
```

```
        for (int cell : locationCells) {
```

```
            if (guess == cell) {
```

```
                result = "hit";  
                numOfHits++;
```

```
                break;
```

```
            }
```

```
        } // out of the loop
```

```
        if (numOfHits == locationCells.length) {  
            result = "kill";
```

```
        }
```

```
        System.out.println(result);
```

```
        return result;
```

```
    } // close method
```

```
} // close class
```

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

← 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 DotCom class

```
import java.util.ArrayList;
```

```
public class DotCom {
```

```
    private ArrayList<String> locationCells;
```

```
    // private int numOfHits;
```

```
    // don't need that now
```

```
    public void setLocationCells(ArrayList<String> loc) {
```

```
        locationCells = loc;
```

```
    }
```

```
    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";
```

```
            } // close if
```

```
        } // close outer if
```

```
        return result;
```

```
    } // close method
```

```
} // close class
```

**Now with  
ArrayList  
power!**

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

Change the String array to an ArrayList that holds Strings.

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!

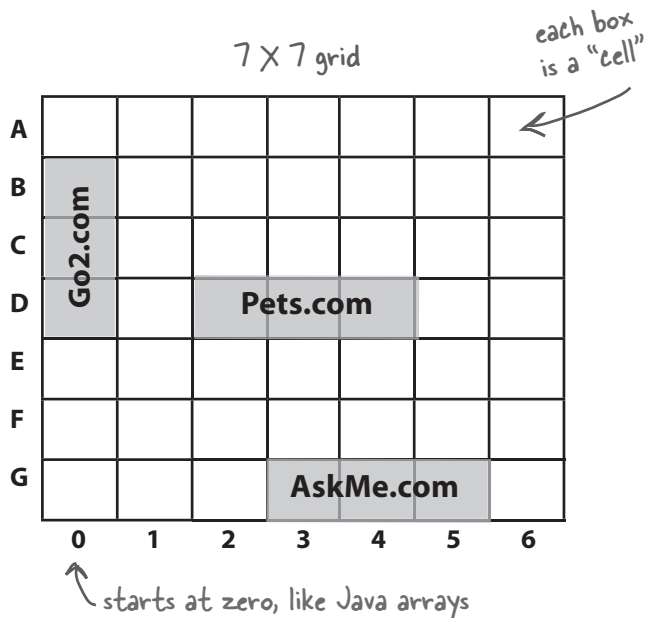
## Let's build the REAL game: "Sink a Dot Com"

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 DotCom, we'll use three.

**Goal:** Sink all of the computer's Dot Coms 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 Dot Coms, 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 Pets.com" (or whatever the lucky Dot Com of the day is). When you've sent all three Dot Coms to that big 404 in the sky, the game ends by printing out your rating.



**You're going to build the Sink a Dot Com game, with a 7 x 7 grid and three Dot Coms. Each Dot Com takes up three cells.**

### part of a game interaction

```
File Edit Window Help Sell
% java DotComBust
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 Pets.com : (
kill
Enter a guess  B4
miss
Enter a guess  G3
hit
Enter a guess  G4
hit
Enter a guess  G5
Ouch! You sunk AskMe.com : (
```



## What needs to change?

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

### A DotCom class

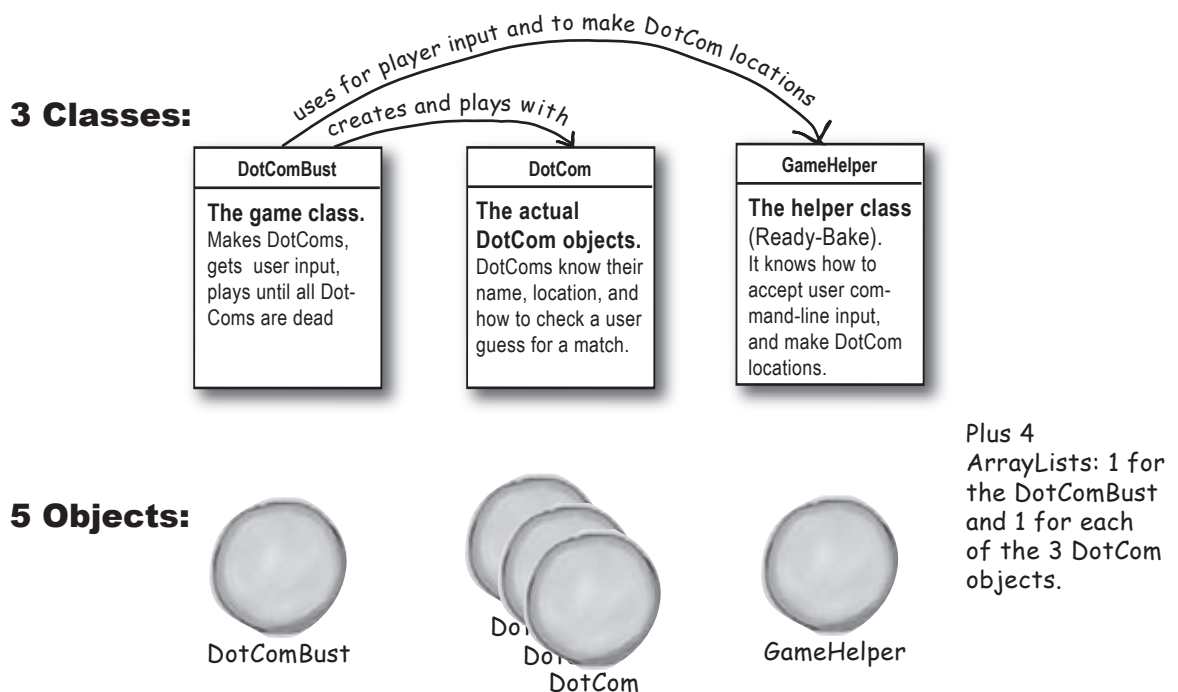
- ⊙ **Add a *name* variable**  
to hold the name of the DotCom ("Pets.com", "Go2.com", etc.) so each DotCom can print its name when it's killed (see the output screen on the opposite page).

### B DotComBust class (the game)

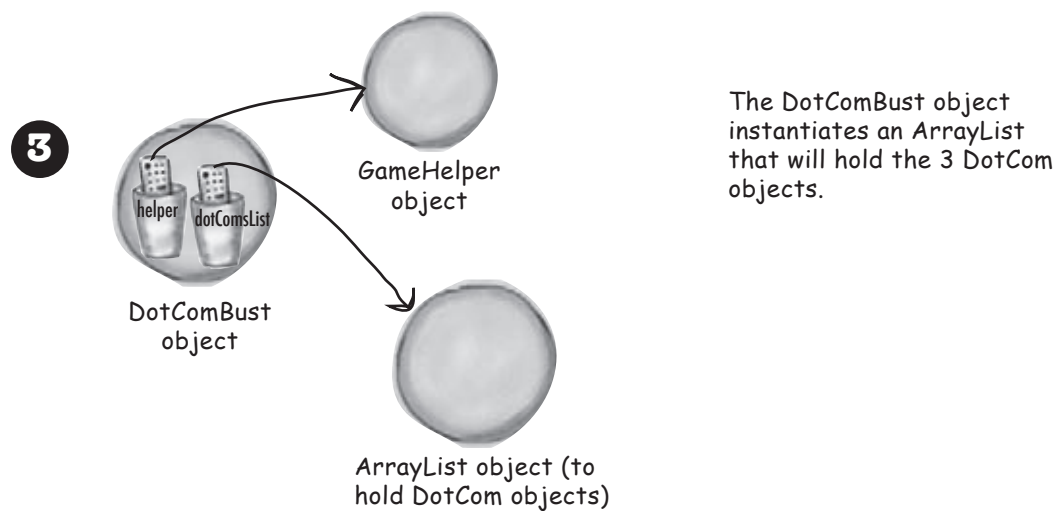
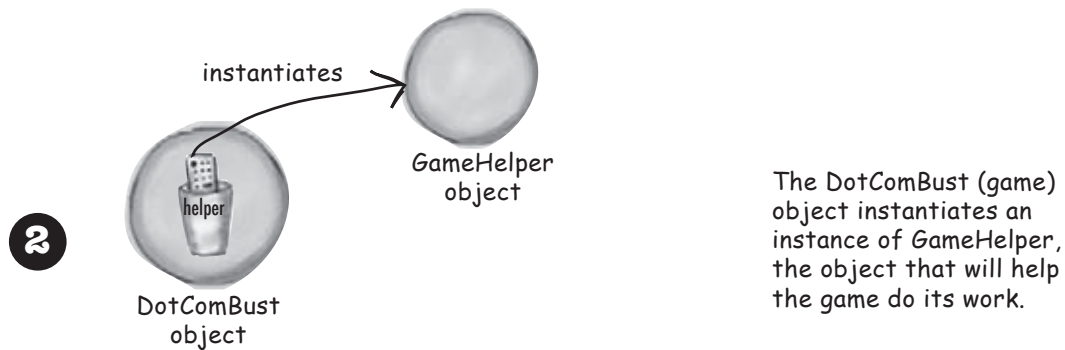
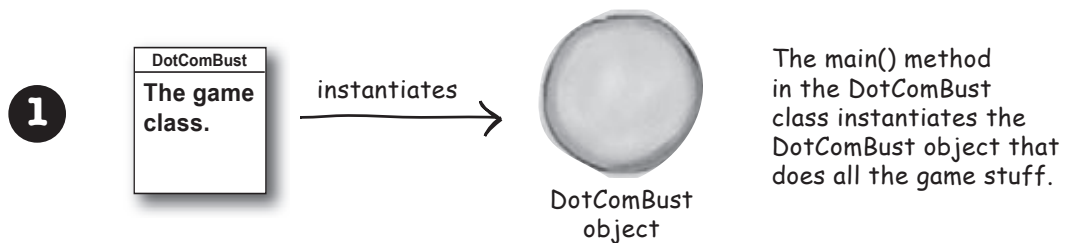
- ⊙ **Create *three* DotComs instead of one.**
- ⊙ **Give each of the three DotComs a *name*.**  
Call a setter method on each DotCom instance, so that the DotCom can assign the name to its name instance variable.

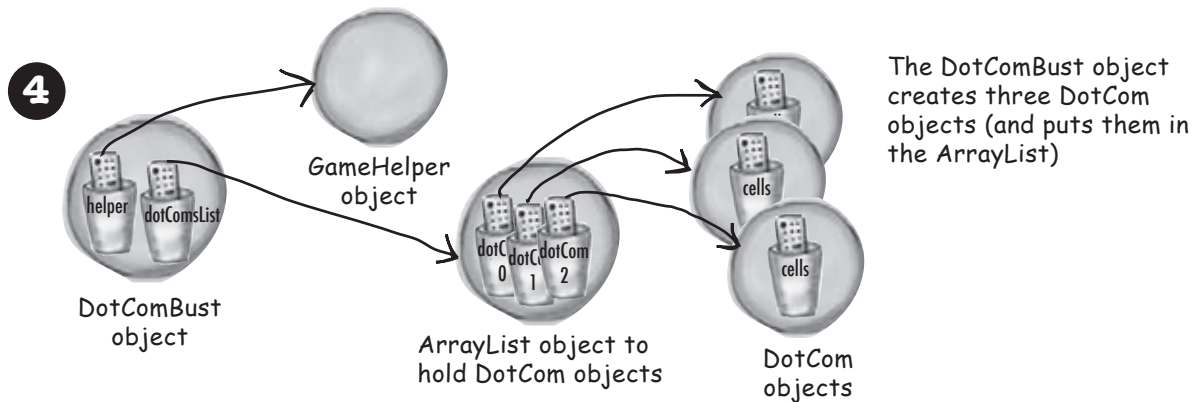
### DotComBust class continued...

- ⊙ **Put the DotComs on a grid rather than just a single row, and do it for all three DotComs.**  
This step is now way more complex than before, if we're going to place the DotComs randomly. Since we're not here to mess with the math, we put the algorithm for giving the DotComs a location into the GameHelper (Ready-bake) class.
- ⊙ **Check each user guess *with all three* DotComs, instead of just one.**
- ⊙ **Keep playing the game** (i.e accepting user guesses and checking them with the remaining DotComs) *until there are no more live DotComs*.
- ⊙ **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.

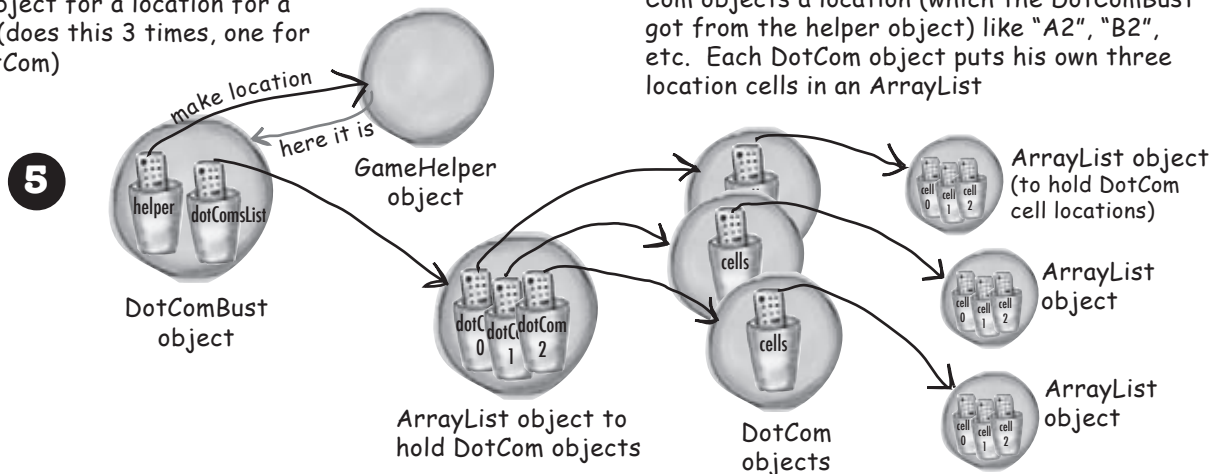


## Who does what in the DotComBust game (and when)

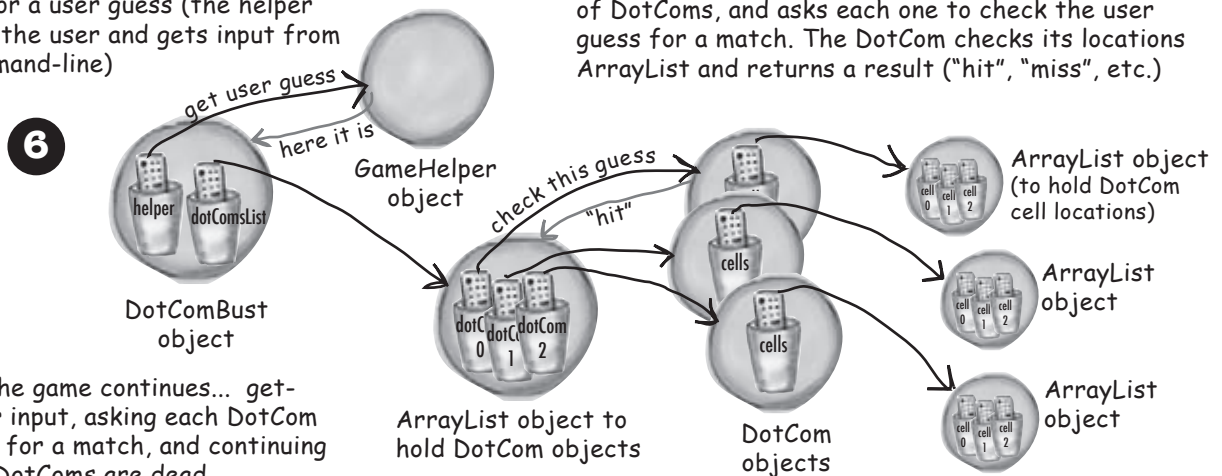




The DotComBust object asks the helper object for a location for a DotCom (does this 3 times, one for each DotCom)



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



the **DotComBust** class (the game)

prep code   test code   real code

| DotComBust  |
|---|
| GameHelper helper<br>ArrayList dotComsList<br>int numOfGuesses    |
| setUpGame()<br>startPlaying()<br>checkUserGuess()<br>finishGame() |

## Variable Declarations

## Method Declarations

## Method Implementations

## Prep code for the real DotComBust class

The DotComBust class has three main jobs: set up the game, play the game until the DotComs 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.

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

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

**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.

---

**DECLARE** a *setUpGame()* method to create and initialize the DotCom 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 DotCom objects are removed from play.

**DECLARE** a *checkUserGuess()* method that loops through all remaining DotCom objects and calls each DotCom 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 DotCom objects.

---

**METHOD: void setUpGame()**

*// make three DotCom objects and name them*

**CREATE** three DotCom objects.

**SET** a name for each DotCom.

**ADD** the DotComs to the *dotComsList* ( the ArrayList).

**REPEAT** with each of the DotCom objects in the *dotComsList* array

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

**SET** the location for each DotCom based on the result of the *placeDotCom()* call.

END REPEAT

END METHOD

prep code

test code

real code

**Method implementations continued:****METHOD: void startPlaying()****REPEAT** while any DotComs 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 DotCom***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 DotObjects in the *dotComsList* array**EVALUATE** the user's guess by calling the DotCom object's *checkYourself()* method**SET** the result variable to "hit" or "kill" if appropriate**IF** the result is "kill", **REMOVE** the DotCom from the *dotComsList*

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

**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 DotComBust code (the game)

prep code   test code   real code

```
import java.util.*;
public class DotComBust {

    ① {private GameHelper helper = new GameHelper();
    private ArrayList<DotCom> dotComsList = new ArrayList<DotCom>();
    private int numOfGuesses = 0;

    private void setUpGame() {
        // first make some dot coms and give them locations
        DotCom one = new DotCom();
        one.setName("Pets.com");
        DotCom two = new DotCom();
        two.setName("eToys.com");
        DotCom three = new DotCom();
        three.setName("Go2.com");
        dotComsList.add(one);
        dotComsList.add(two);
        dotComsList.add(three);

        System.out.println("Your goal is to sink three dot coms.");
        System.out.println("Pets.com, eToys.com, Go2.com");
        System.out.println("Try to sink them all in the fewest number of guesses");

        for (DotCom dotComToSet : dotComsList) {
            ArrayList<String> newLocation = helper.placeDotCom(3);
            dotComToSet.setLocationCells(newLocation);
        } // close for loop
    } // close setUpGame method

    private void startPlaying() {
        while(!dotComsList.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
- print brief instructions for user
- call our own finishGame method
- get user input
- call the setter method on this DotCom to give it the location you just got from the helper
- make three DotCom objects, give 'em names, and stick 'em in the ArrayList
- ask the helper for a DotCom location
- repeat with each DotCom in the list
- call our own checkUserGuess method
- as long as the DotCom list is NOT empty

prep code   test code   **real code**

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

    for (DotCom dotComToTest : dotComsList) { 13
        result = dotComToTest.checkYourself(userGuess); 14
        if (result.equals("hit")) {
            break; 15
        }
        if (result.equals("kill")) {
            dotComsList.remove(dotComToTest); 16
            break;
        }
    } // close for
    System.out.println(result); 17
} // close method

private void finishGame() {
    System.out.println("All Dot Coms 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) {
    DotComBust game = new DotComBust(); 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 DotComs in the list
  - this guy's dead, so take him out of the DotComs list then get out of the loop
  - increment the number of guesses the user has made
  - get out of the loop early, no point in testing the others
  - print a message telling the user how he did in the game
  - assume it's a 'miss', unless told otherwise
  - tell the game object to start the main game play loop (keeps asking for user input and checking the guess)
  - print the result for the user
  - ask the DotCom to check the user guess, looking for a hit (or kill)
  - tell the game object to set up the game
  - create the game object

## the DotComBust code (the game)

prep code   test code   real code

```
import java.util.*;
public class DotComBust {
```

Declare and initialize  
the variables we'll need.

```
    private GameHelper helper = new GameHelper();
    private ArrayList<DotCom> dotComsList = new ArrayList<DotCom>();
    private int numOfGuesses = 0;
```

Make an ArrayList of  
DotCom objects (in other  
words, a list that will hold  
ONLY DotCom objects,  
just as DotCom[] would  
mean an array of DotCom  
objects).

```
private void setUpGame() {
```

```
    // first make some dot coms and give them locations
```

```
    DotCom one = new DotCom();
    one.setName("Pets.com");
    DotCom two = new DotCom();
    two.setName("eToys.com");
    DotCom three = new DotCom();
    three.setName("Go2.com");
    dotComsList.add(one);
    dotComsList.add(two);
    dotComsList.add(three);
```

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

```
    System.out.println("Your goal is to sink three dot coms.");
```

```
    System.out.println("Pets.com, eToys.com, Go2.com");
```

```
    System.out.println("Try to sink them all in the fewest number of guesses");
```

Print brief  
instructions for user.

```
    for (DotCom dotComToSet : dotComsList) {
```

← Repeat with each DotCom in the list.

```
        ArrayList<String> newLocation = helper.placeDotCom(3);
```

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

```
        dotComToSet.setLocationCells(newLocation);
```

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

```
    } // close for loop
```

```
} // close setUpGame method
```

```
private void startPlaying() {
```

```
    while(!dotComsList.isEmpty()) {
```

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

```
        String userGuess = helper.getUserInput("Enter a guess");
```

← Get user input.

```
        checkUserGuess(userGuess);
```

← Call our own checkUserGuess method.

```
    } // close while
```

```
    finishGame();
```

← Call our own finishGame method.

```
} // close startPlaying method
```



prep code   test code   **real code**

```

private void checkUserGuess (String userGuess) {

    numOfGuesses++;           ← increment the number of guesses the user has made

    String result = "miss";   ← assume it's a 'miss', unless told otherwise

    for (DotCom dotComToTest : dotComsList) { ← repeat with all DotComs in the list

        result = dotComToTest.checkYourself(userGuess); ← ask the DotCom to check the user
                                                            guess, looking for a hit (or kill)

        if (result.equals("hit")) {

            break;           ← get out of the loop early, no point
                              in testing the others

        }

        if (result.equals("kill")) {

            dotComsList.remove(dotComToTest); ← this guy's dead, so take him out of the
            break;                               DotComs list then get out of the loop

        }

    } // close for

    System.out.println(result); ← print the result for the user
} // close method

private void finishGame () {
    System.out.println("All Dot Coms 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) {
    DotComBust game = new DotComBust(); ← create the game object
    game.setUpGame(); ← tell the game object to set up the game
    game.startPlaying(); ← tell the game object to start the main
                          game play loop (keeps asking for user
                          input and checking the guess)
} // close method
}

```

print a message telling the user how he did in the game

## the DotCom code

prep code

test code

real code

# The final version of the DotCom class

```
import java.util.*;
```

```
public class DotCom {
```

```
    private ArrayList<String> locationCells;
```

```
    private String name;
```

DotCom's instance variables:

- an ArrayList of cell locations
- the DotCom's name

```
    public void setLocationCells(ArrayList<String> loc) {
```

```
        locationCells = loc;
```

```
    }
```

A setter method that updates the DotCom's location. (Random location provided by the GametHelper placeDotCom() method.)

```
    public void setName(String n) {
```

```
        name = n;
```

```
}
```

Your basic setter method

```
    public String checkYourself(String userInput) {
```

```
        String result = "miss";
```

```
        int index = locationCells.indexOf(userInput);
```

```
        if (index >= 0) {
```

```
            locationCells.remove(index);
```

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

```
            if (locationCells.isEmpty()) {
```

```
                result = "kill";
```

```
                System.out.println("Ouch! You sunk " + name + " : ( ");
```

```
            } else {
```

```
                result = "hit";
```

```
            } // close if
```

```
        } // close if
```

```
        return result;
```

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

Tell the user when a DotCom has been sunk.

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

```
    } // close method
```

```
} // close class
```

## 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: GameHelper



### 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 DotComs. If we were you, we'd just back away slowly from this code, except to type it in and compile it. We tried to keep it fairly small to you wouldn't have to type so much, but that means it isn't the most readable code. And remember, you won't be able to compile the DotComBust game class until you have *this* class.

```
import java.io.*;
import java.util.*;
```

```
public class GameHelper {
```

```
    private static final String alphabet = "abcdefg";
    private int gridLength = 7;
    private int gridSize = 49;
    private int [] grid = new int[gridSize];
    private int comCount = 0;
```

```
    public String getUserInput(String prompt) {
        String inputLine = null;
        System.out.print(prompt + " ");
        try {
            BufferedReader is = new BufferedReader(
                new InputStreamReader(System.in));
            inputLine = is.readLine();
            if (inputLine.length() == 0 ) return null;
        } catch (IOException e) {
            System.out.println("IOException: " + e);
        }
        return inputLine.toLowerCase();
    }
```

```
    public ArrayList<String> placeDotCom(int comSize) {
        ArrayList<String> alphaCells = new ArrayList<String>();
        String [] alphacoords = new String [comSize]; // holds 'f6' type coords
        String temp = null; // temporary String for concat
        int [] coords = new int[comSize]; // current candidate coords
        int attempts = 0; // current attempts counter
        boolean success = false; // flag = found a good location ?
        int location = 0; // current starting location

        comCount++; // nth dot com to place
        int incr = 1; // set horizontal increment
        if ((comCount % 2) == 1) { // if odd dot com (place vertically)
            incr = gridLength; // set vertical increment
        }

        while ( !success & attempts++ < 200 ) { // main search loop (32)
            location = (int) (Math.random() * gridSize); // get random starting point
            //System.out.print(" try " + location);
            int x = 0; // nth position in dotcom to place
            success = true; // assume success
            while (success && x < comSize) { // look for adjacent unused spots
                if (grid[location] == 0) { // if not already used
```

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



## Ready-bake Code

### GameHelper class code continued...

```

        coords[x++] = location;                // save location
        location += incr;                      // try 'next' adjacent
        if (location >= gridSize){            // out of bounds - 'bottom'
            success = false;                   // failure
        }
        if (x>0 && (location % gridLength == 0)) { // out of bounds - right edge
            success = false;                   // failure
        }
    } else {                                  // found already used location
        // System.out.println(" used " + location);
        success = false;                       // failure
    }
}
}
// end while

int x = 0;                                    // turn location into alpha coords
int row = 0;
int column = 0;
// System.out.println("\n");
while (x < comSize) {
    grid[coords[x]] = 1;                      // mark master grid pts. as 'used'
    row = (int) (coords[x] / gridLength);      // get row value
    column = coords[x] % gridLength;           // get numeric column value
    temp = String.valueOf(alphabet.charAt(column)); // convert to alpha

    alphaCells.add(temp.concat(Integer.toString(row)));
    x++;
    // System.out.print(" coord "+x+" = " + alphaCells.get(x-1)); ← This is the statement that
}                                              tells you exactly where the
                                              DotCom is located.

// System.out.println("\n");

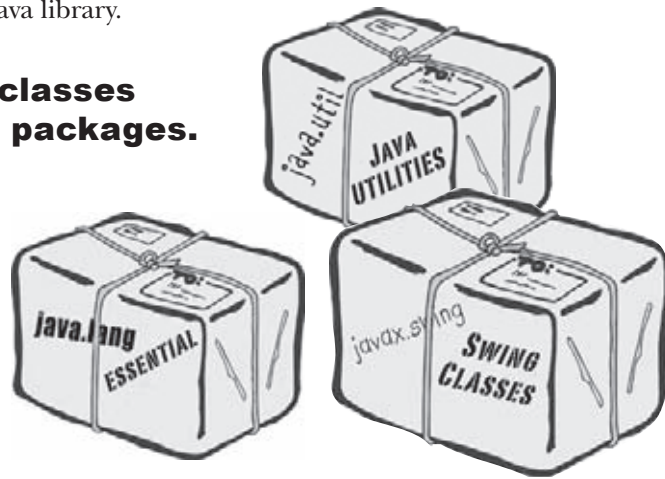
return alphaCells;
}
}

```

## Using the Library (the Java API)

You made it all the way through the DotComBust 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 chapter 16, 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:

`java.util.ArrayList`

└──┬──────────┬──────────┘  
package name      class name

## 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. *Anywhere* 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.

## <sup>there are no</sup> 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. You’ll learn all about that in chapter 16.

**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. We’ll get into that in more detail in chapter 16.



when arrays aren't enough

### Where'd that 'x' come from? (or, what does it mean when a package starts with javax?)

In the first and second versions of Java (1.02 and 1.1), all classes that shipped with Java (in other words, the standard library) were in packages that began with **java**. There was always **java.lang**, of course — the one you don't have to import. And there was **java.net**, **java.io**, **java.util** (although there was no such thing as **ArrayList** way back then), and a few others, including the **java.awt** package that held GUI-related classes.

Looming on the horizon, though, were other packages not included in the standard library. These classes were known as **extensions**, and came in two main flavors: *standard*, and *not standard*. Standard extensions were those that Sun considered official, as opposed to experimental, early access, or beta packages that might or might not ever see the light of day.

Standard extensions, by convention, all began with an 'x' appended to the regular **java** package starter. The mother of all standard extensions was the Swing library. It included several packages, all of which began with **javax.swing**.

But standard extensions can get promoted to first-class, ships-with-Java, standard-out-of-the-box library packages. And that's what happened to Swing, beginning with version 1.2 (which eventually became the first version dubbed 'Java 2').

"Cool," everyone thought (including us). "Now everyone who has Java will have the Swing classes, and we won't have to figure out how to get those classes installed with our end-users."

Trouble was lurking beneath the surface, however, because when packages get promoted, well of COURSE they have to start with **java**, not **javax**. Everyone KNOWS that packages in the standard library don't have that "x", and that only extensions have the "x". So, just (and we mean just) before version 1.2 went final, Sun changed the package names and deleted the "x" (among other changes). Books were printed and in stores featuring Swing code with the new names. Naming conventions were intact. All was right with the Java world.

Except the 20,000 or so screaming developers who realized that with that simple name change came disaster! All of their Swing-using code had to be changed! The horror! Think of all those import statements that started with **javax**...

And in the final hour, desperate, as their hopes grew thin, the developers convinced Sun to "screw the convention, save our code". The rest is history. So when you see a package in the library that begins with **javax**, you know it started life as an extension, and then got a promotion.



### 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 **indexOf()**.
- 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 use either 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?

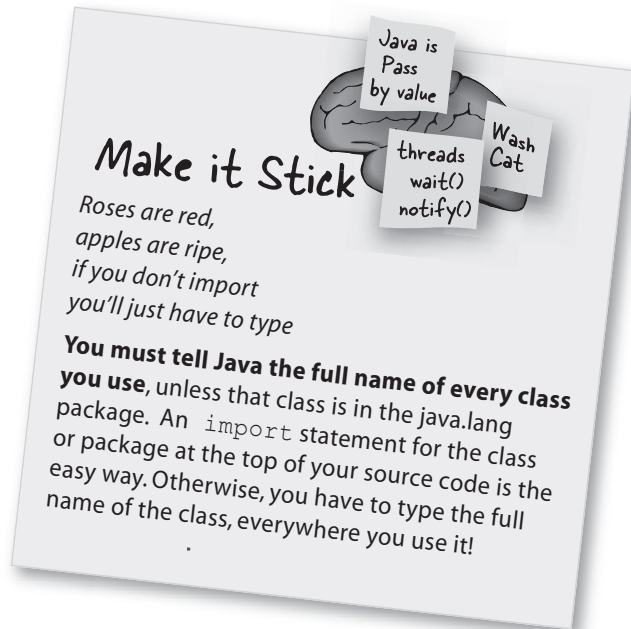
**A:** Perhaps you're a C programmer? An `import` is not the same as an `include`. So the answer is no and no. Repeat after me: "an `import` statement saves you from typing." That's really it. You don't have to worry about your code becoming bloated, or slower, from too many imports. An `import` is simply 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 chapter 16. For now, we won't put our code examples in a package.



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



getting to know the API

*“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

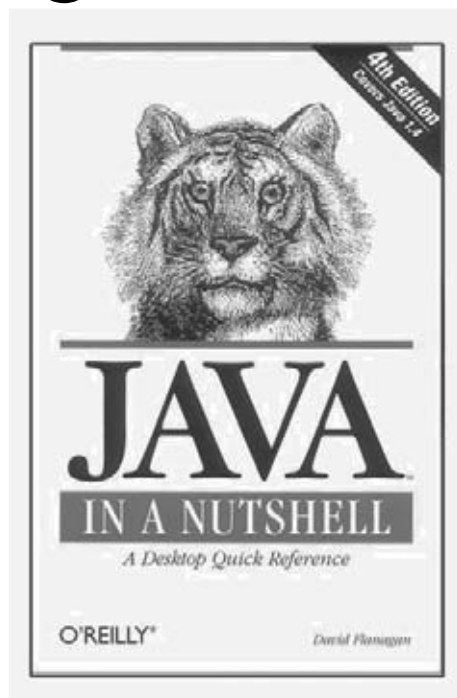
## How to play with the API

Two things you want to know:

- 1 What classes are in the library?
- 2 Once you find a class, how do you know what it can do?



### 1 Browse a Book



### 2 Use the HTML API docs



# 1 Browse a Book



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

class name

package name

class description

methods (and other things we'll talk about later)

*java.util.Currency*

*Returned By:* `java.text.DecimalFormat.getCurrency()`, `java.text.DecimalFormatSymbols.getCurrency()`, `java.text.NumberFormat.getCurrency()`, `Currency.getInstance()`

---

**Date** Java 1.0

`java.util` *cloneable serializable comparable*

This class represents dates and times and lets you work with them in a system-independent way. You can create a `Date` by specifying the number of milliseconds from the epoch (midnight GMT, January 1st, 1970) or the year, month, date, and, optionally, the hour, minute, and second. Years are specified as the number of years since 1900. If you call the `Date` constructor with no arguments, the `Date` is initialized to the current time and date. The instance methods of the class allow you to get and set the various date and time fields, to compare dates and times, and to convert dates to and from string representations. As of Java 1.1, many of the date methods have been deprecated in favor of the methods of the `Calendar` class.

Object — Date

Cloneable — Comparable — Serializable

```

public class Date implements Cloneable, Comparable, Serializable {
    // Public Constructors
    public Date();
    public Date(long date);
    # public Date(String s);
    # public Date(int year, int month, int date);
    # public Date(int year, int month, int date, int hrs, int min);
    # public Date(int year, int month, int date, int hrs, int min, int sec);
    // Property Accessor Methods (by property name)
    public long getTime();
    public void setTime(long time);
    // Public Instance Methods
    public boolean after(java.util.Date when);
    public boolean before(java.util.Date when);
    1.2 public int compareTo(java.util.Date anotherDate);
    // Methods implementing Comparable
    1.2 public int compareTo(Object o);
    // Public Methods Overriding Object
    1.2 public Object clone();
    public boolean equals(Object obj);
    public int hashCode();
    public String toString();
    // Deprecated Public Methods
    # public int getDate();
    # public int getDay();
    # public int getHours();
    # public int getMinutes();
    # public int getMonth();
    # public int getSeconds();
    # public int getTimezoneOffset();
    # public int getYear();
    # public static long parse(String s);
    # public void setDate(int date);
    # public void setHours(int hours);
    # public void setMinutes(int minutes);
    # public void setMonth(int month);
  
```

## 2

### Use the HTML API docs

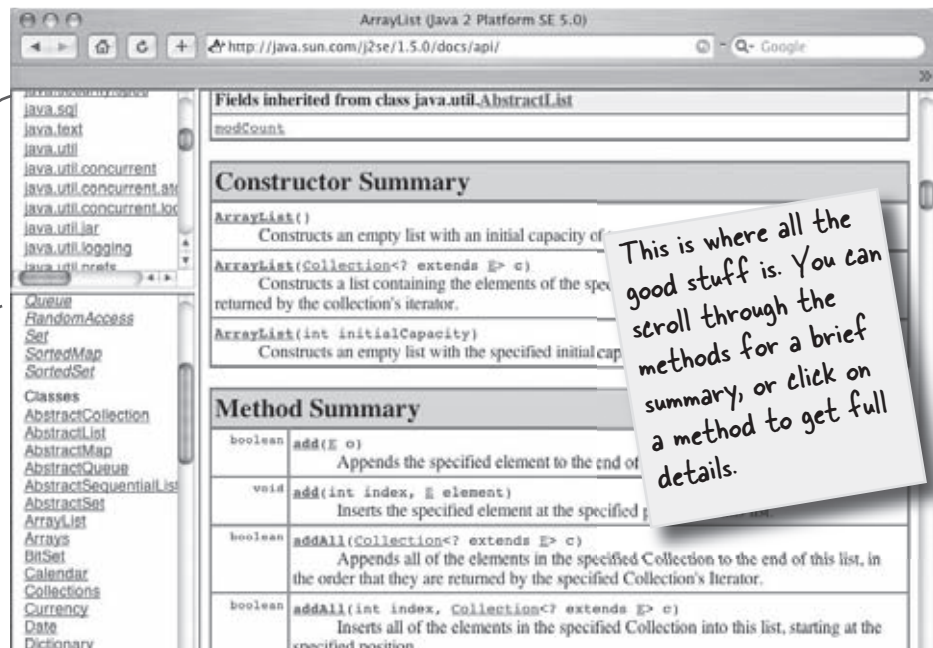
Java comes with a fabulous set of online docs called, strangely, the Java API. They're part of a larger set called the Java 5 Standard Edition Documentation (which, depending on what day of the week you look, Sun may be referring to as "Java 2 Standard Edition 5.0"), and you have to download the docs separately; they don't come shrink-wrapped with the Java 5 download. If you have a high-speed internet connection, or tons of patience, you can also browse them at [java.sun.com](http://java.sun.com). Trust us, you probably want these on your hard drive.

The API docs are the best reference for getting more details about a class and its methods. Let's say you were browsing through the reference book and found a class called `Calendar`, 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.

The reference book, for example, tells you what the methods take, as arguments, and what they return. Look at `ArrayList`, for example. In the reference book, you'll find the method `indexOf()`, that we used in the `DotCom` class. 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`. That's how we knew we could 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`.

1  
Scroll through the packages and select one (click it) to restrict the list in the lower frame to only classes from that package.

2  
Scroll through the classes and select one (click it) to choose the class that will fill the main browser frame.





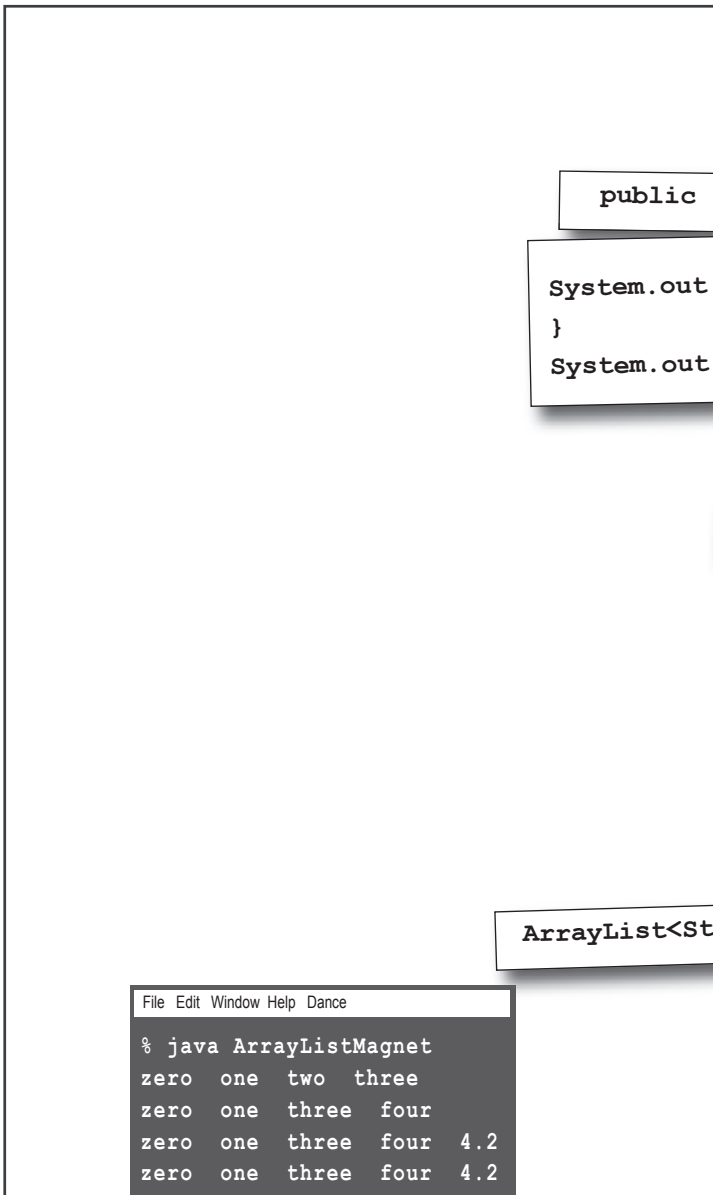
## Exercise

## 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.



```
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
```

```
a.remove(2);
```

```
printAL(a);
```

```
printAL(a);
```

```
a.add(0, "zero");
a.add(1, "one");
```

```
public static void printAL(ArrayList<String> al) {
```

```
    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.*;
```

```
printAL(a);
```

```
ArrayList<String> a = new ArrayList<String>();
```

```
for (String element : al) {
```

```
    a.add(3, "three");
    printAL(a);
```

puzzle: crossword



# JavaCross 7.0

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

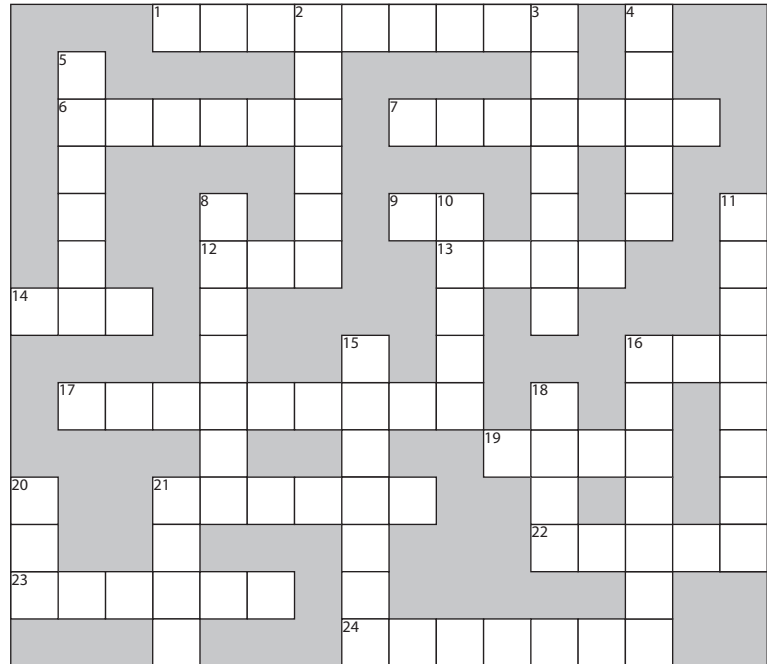
**Hint:** When in doubt, remember ArrayList.

## 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

## More Hints:

- Down
2. What's overridable?
  3. Think ArrayList
  4. & 10. Primitive
  16. Think ArrayList
  21. Array's extent
  22. Not about Java - Spanish appetizers
- Across
1. 8 varieties
  7. Think ArrayList
  16. Common primitive
  21. Array's extent
  22. Not about Java - Spanish appetizers



## 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



## 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
```

```
import java.util.*;

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");
        printAL(a);

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

        a.remove(2);
        printAL(a);

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

        printAL(a);

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

        printAL(a);
    }

    public static void printAL(ArrayList<String> al) {

        for (String element : al) {

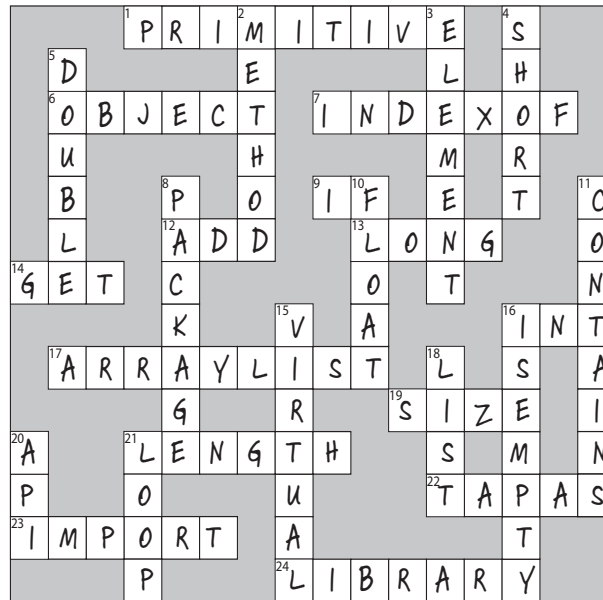
            System.out.print(element + " ");
        }

        System.out.println(" ");
    }
}
```

## puzzle answers



# JavaCross answers



## 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. \_\_\_\_\_