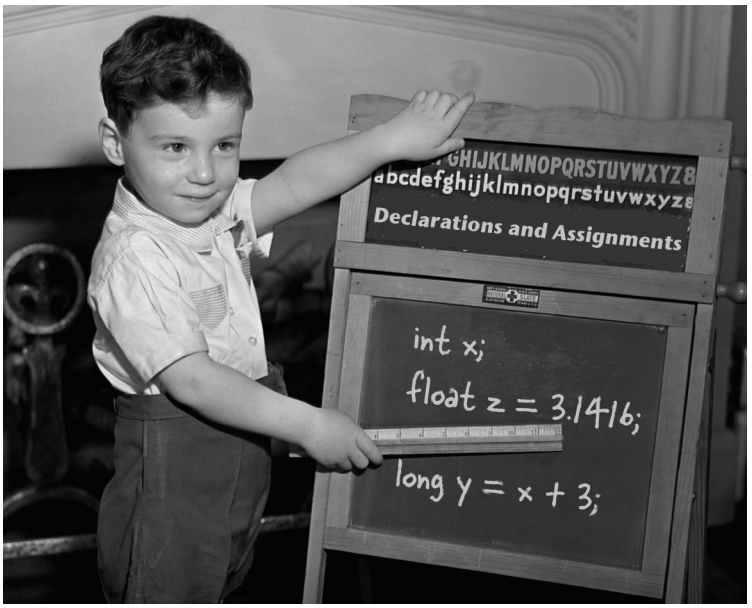
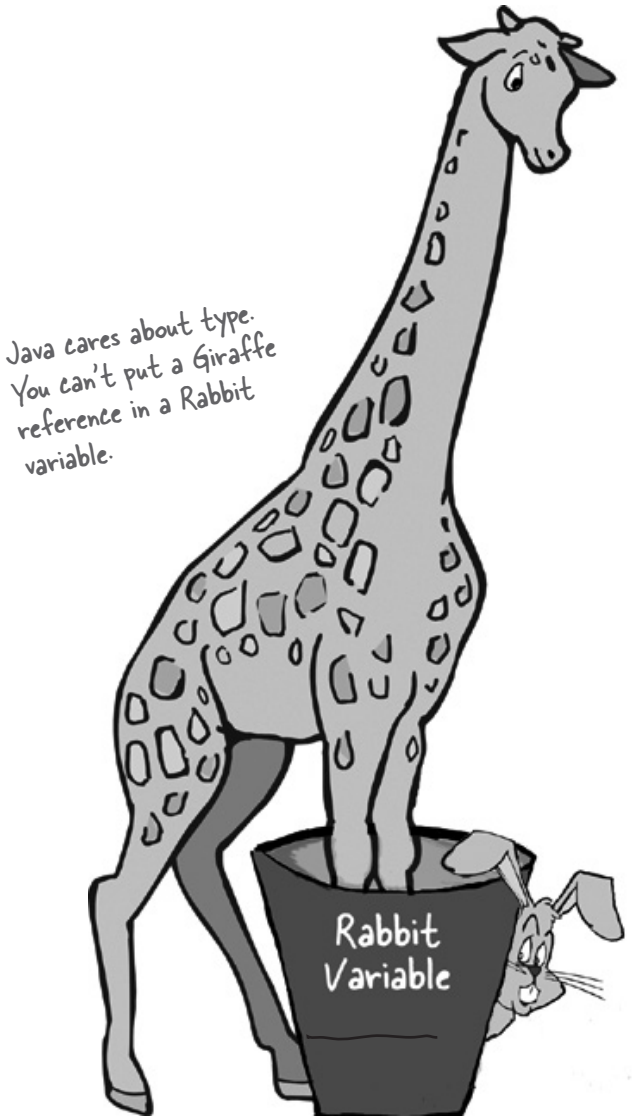
Know Your Variables



Variables can store two types of things: primitives and references.

So far you've used variables in two places—as object **state** (instance variables) and as **local** variables (variables declared within a *method*). Later, we'll use variables as **arguments** (values sent to a method by the calling code), and as **return types** (values sent back to the caller of the method). You've seen variables declared as simple **primitive** integer values (type `int`). You've seen variables declared as something more **complex** like a String or an array. But **there's gotta be more to life** than integers, Strings, and arrays. What if you have a PetOwner object with a Dog instance variable? Or a Car with an Engine? In this chapter we'll unwrap the mysteries of Java types (like the difference between primited and references) and look at what you can *declare* as a variable, what you can *put* in a variable, and what you can *do* with a variable. And we'll finally see what life is *truly* like on the garbage-collectible heap.



Declaring a variable

Java cares about type. It won't let you do something bizarre and dangerous like stuff a Giraffe reference into a Rabbit variable—what happens when someone tries to ask the so-called *Rabbit* to `hop()`? And it won't let you put a floating-point number into an integer variable, unless you *tell the compiler* that you know you might lose precision (like, everything after the decimal point).

The compiler can spot most problems:

```
Rabbit hopper = new Giraffe();
```

Don't expect that to compile. *Thankfully*.

For all this type-safety to work, you must declare the type of your variable. Is it an integer? a Dog? A single character? Variables come in two flavors: ***primitive*** and ***object reference***. Primitives hold fundamental values (think: simple bit patterns) including integers, booleans, and floating-point numbers. Object references hold, well, *references* to *objects* (gee, didn't *that* clear it up).

We'll look at primitives first and then move on to what an object reference really means. But regardless of the type, you must follow two declaration rules:

variables must have a type

Besides a type, a variable needs a name so that you can use that name in code.

variables must have a name

```
int count;
```

type → int

name → count

Note: When you see a statement like: "an object of **type** X," think of *type* and *class* as synonyms. (We'll refine that a little more in later chapters.)

"I'd like a double mocha, no, make it an int."

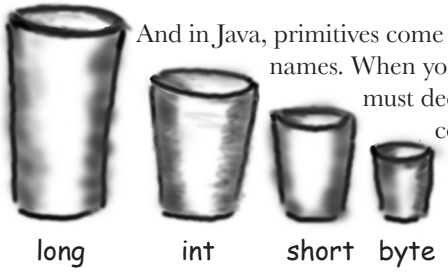
When you think of Java variables, think of cups. Coffee cups, tea cups, giant cups that hold lots and lots of your favorite drink, those big cups the popcorn comes in at the movies, cups with wonderful tactile handles, and cups with metallic trim that you learned can never, ever go in the microwave.

A variable is just a cup. A container. It *holds* something.

It has a size and a type. In this chapter, we're going to look first at the variables (cups) that hold **primitives**: then a little later we'll look at cups that hold *references to objects*. Stay with us here on the whole cup analogy—as simple as it is right now, it'll give us a common way to look at things when the discussion gets more complex. And that'll happen soon.

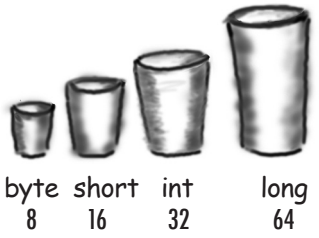
Primitives are like the cups they have at the coffee shop. If you've been to a Starbucks, you know what we're talking about here. They come in different sizes, and each has a name like "short," "tall," and, "I'd like a 'grande' mocha half-caff with extra whipped cream."

You might see the cups displayed on the counter so you can order appropriately:



And in Java, primitives come in different sizes, and those sizes have names. When you declare any variable in Java, you must declare it with a specific type. The four containers here are for the four integer primitives in Java.

Each cup holds a value, so for Java primitives, rather than saying, "I'd like a tall french roast," you say to the compiler, "I'd like an int variable with the number 90 please." Except for one tiny difference...in Java you also have to give your cup a *name*. So it's actually, "I'd like an int please, with the value of 2486, and name the variable ***height***." Each primitive variable has a fixed number of bits (cup size). The sizes for the six numeric primitives in Java are shown below:



Primitive Types

Type	Bit Depth	Value Range
boolean and char		
boolean	(JVM-specific)	***true*** or ***false***
char	16 bits	0 to 65535
numeric (all are signed)		
integer		
byte	8 bits	-128 to 127
short	16 bits	-32768 to 32767
int	32 bits	-2147483648 to 2147483647
long	64 bits	-huge to huge
floating point		
float	32 bits	varies
double	64 bits	varies

Primitive declarations with assignments:

```
int x;
x = 234;
byte b = 89;
boolean isFun = true;
double d = 3456.98;
char c = 'f';
int z = x;
boolean isPunkRock;
isPunkRock = false;
boolean powerOn;
powerOn = isFun;
long big = 3456789L;
float f = 32.5f;
```

Note the 'f' and 'L'. With some number types, you have to specifically tell the compiler what you mean, or it might get confused between similar-looking number types. You can use upper or lowercase.

You *really* don't want to spill that...

Be sure the value can fit into the variable.

You can't put a large value into a small cup.

Well, OK, you can, but you'll lose some. You'll get, as we say, *spillage*. The compiler tries to help prevent this if it can tell from your code that something's not going to fit in the container (variable/cup) you're using.

For example, you can't pour an int-full of stuff into a byte-sized container, as follows:

```
int x = 24;
byte b = x;
//won't work!!
```

Why doesn't this work, you ask? After all, the value of *x* is 24, and 24 is definitely small enough to fit into a byte. *You* know that, and *we* know that, but all the compiler cares about is that you're trying to put a big thing into a small thing, and there's the *possibility* of spilling. Don't expect the compiler to know what the value of *x* is, even if you happen to be able to see it literally in your code.

You can assign a value to a variable in one of several ways including:

- type a *literal* value after the equals sign (x=***12***, isGood = ***true***, etc.)
- assign the value of one variable to another (x = y)
- use an expression combining the two (x = y + ***43***)

In the examples below, the literal values are in bold italics:

`int size = 32;`	declare an int named *size*, assign it the value *32*
`char initial = 'j';`	declare a char named *initial*, assign it the value *'j'*
`double d = 456.709;`	declare a double named *d*, assign it the value *456.709*
`boolean isLearning;`	declare a boolean named *isCrazy* (no assignment)
`isLearning = true;`	assign the value *true* to the previously declared *isCrazy*
`int y = x + 456;`	declare an int named *y*, assign it the value that is the sum of whatever *x* is now plus *456*

Sharpen your pencil

The compiler won't let you put a value from a large cup into a small one. But what about the other way—pouring a small cup into a big one? ***No problem.***

Based on what you know about the size and type of the primitive variables, see if you can figure out which of these are legal and which aren't. We haven't covered all the rules yet, so on some of these you'll have to use your best judgment. ***Tip:*** The compiler always errs on the side of safety.

From the following list, ***Circle*** the statements that would be legal if these lines were in a single method:

```
1. int x = 34.5;
2. boolean boo = x;
3. int g = 17;
4. int y = g;
5. y = y + 10;
6. short s;
7. s = y;
8. byte b = 3;
9. byte v = b;
10. short n = 12;
11. v = n;
12. byte k = 128;
```

Answers on page 68.

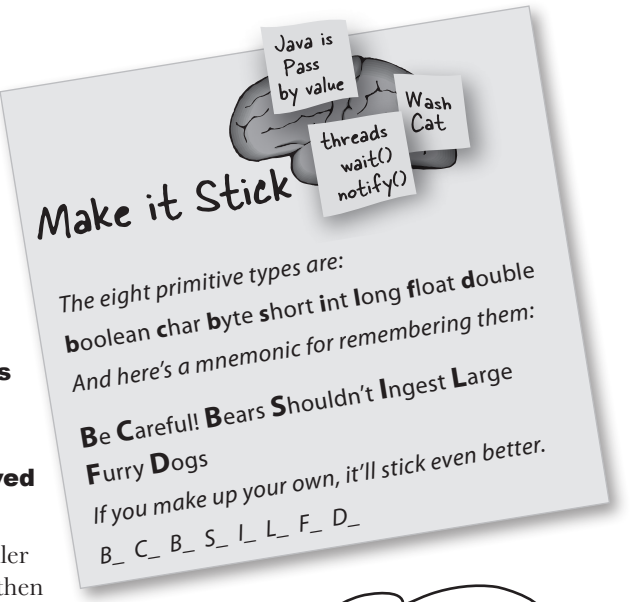
Back away from that keyword!

You know you need a name and a type for your variables.

You already know the primitive types.

But what can you use as names? The rules are simple. You can name a class, method, or variable according to the following rules (the real rules are slightly more flexible, but these will keep you safe):

- **It must start with a letter, underscore (_), or dollar sign ($). You can't start a name with a number.**
- **After the first character, you can use numbers as well. Just don't start it with a number.**
- **It can be anything you like, subject to those two rules, just so long as it isn't one of Java's reserved words.**



Reserved words are keywords (and other things) that the compiler recognizes. And if you really want to play confuse-a-compiler, then just *try* using a reserved word as a name.

You've already seen some reserved words:

```
public   static   void
```

don't use any of these for your own names.

And the primitive types are reserved as well:

```
boolean char byte short int long float double
```

But there are a lot more we haven't discussed yet. Even if you don't need to know what they mean, you still need to know you can't use 'em yourself. ***Do not****—under any circumstances—****try to memorize these now.*** To make room for these in your head, you'd probably have to lose something else. Like where your car is parked. Don't worry, by the end of the book you'll have most of them down cold.

This table reserved

_	catch	double	float	int	private	super	true
abstract	char	else	for	interface	protected	switch	try
assert	class	enum	goto	long	public	synchronized	void
boolean	const	extends	if	native	return	this	volatile
break	continue	false	implements	new	short	throw	while
byte	default	final	import	null	static	throws	
case	do	finally	instanceof	package	strictfp	transient	

Java's keywords, reserved words, and special identifiers. If you use these for names, the compiler will *probably* be very, *very* upset.

Controlling your Dog object

You know how to declare a primitive variable and assign it a value. But now what about non-primitive variables? In other words, *what about objects?*

- **There is actually no such thing as an object variable.**
- **There's only an object reference variable.**
- **An object reference variable holds bits that represent a way to *access* an object.**
- **It doesn't hold the object itself, but it holds something like a pointer. Or an address. Except, in Java we don't really know *what* is inside a reference variable. We *do* know that whatever it is, it represents one and only one object. And the JVM knows how to use the reference to get to the object.**

You can't stuff an object into a variable. We often think of it that way...we say things like, "I passed the String to the System.out.println() method." Or, "The method returns a Dog" or, "I put a new Foo object into the variable named myFoo."

But that's not what happens. There aren't giant expandable cups that can grow to the size of any object. Objects live in one place and one place only—the garbage-collectible heap! (You'll learn more about that later in this chapter.)

Although a primitive variable is full of bits representing the actual ***value*** of the variable, an object reference variable is full of bits representing ***a way to get to the object.***

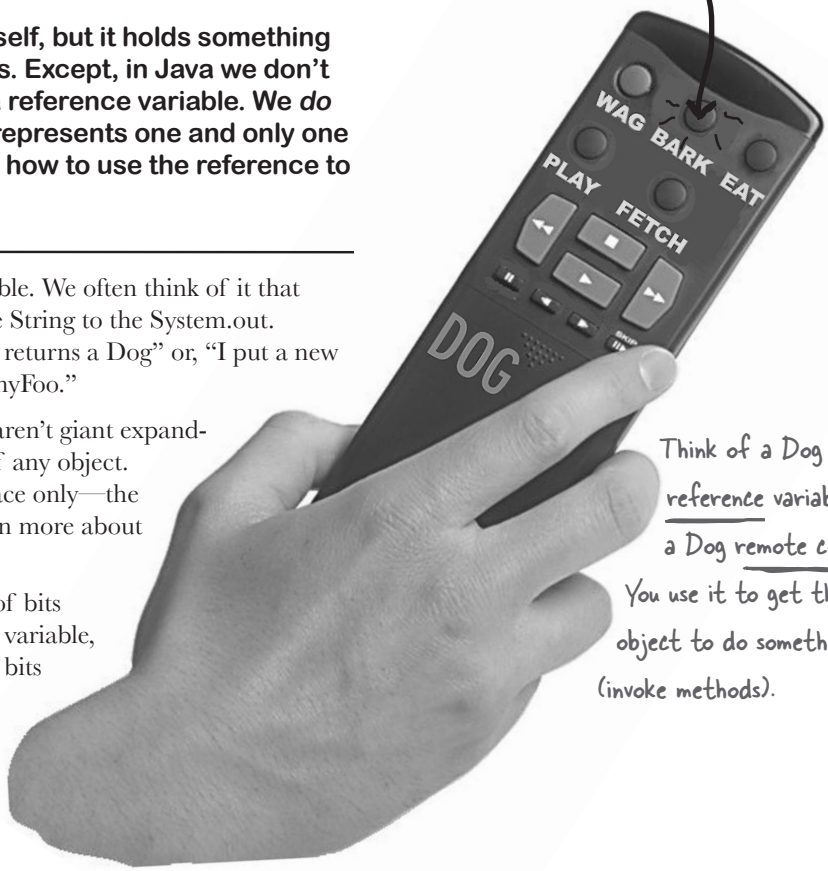
You use the dot operator (.) on a reference variable to say, "use the thing *before* the dot to get me the thing *after* the dot." For example:

```
myDog.bark();
```

means, "use the object referenced by the variable myDog to invoke the bark() method." When you use the dot operator on an object reference variable, think of it like pressing a button on the remote control for that object.

```
Dog d = new Dog();
d.bark();
```

think of this like this



Think of a Dog reference variable as a Dog remote control. You use it to get the object to do something (invoke methods).

An object reference is just another variable value

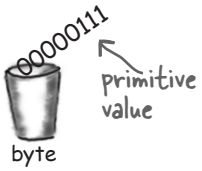
Something that goes in a cup.

Only this time, the value is a remote control.

Primitive Variable

```
byte x = 7;
```

The bits representing 7 go into the variable (00000111).

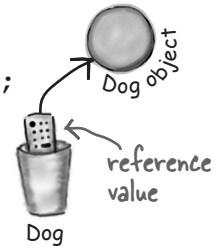


Reference Variable

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

The bits representing a way to get to the Dog object go into the variable.

The Dog object itself does not go into the variable!



With primitive variables, the value of the variable is...the *value* (5, -26.7, 'a').

With reference variables, the value of the variable is...*bits representing a way to get to a specific object.*

You don't know (or care) how any particular JVM implements object references. Sure, they might be a pointer to a pointer to...but even if you *know*, you still can't use the bits for anything other than accessing an object.

We don't care how many 1s and 0s there are in a reference variable. It's up to each JVM and the phase of the moon.

The 3 steps of object declaration, creation 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, and names that variable *myDog*. 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

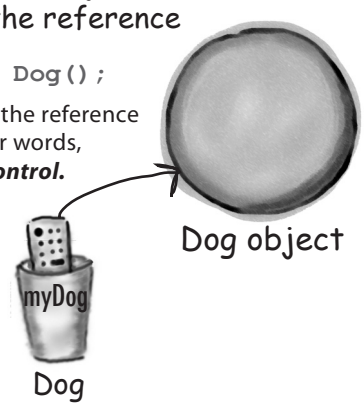
```
Dog myDog = new Dog();
```

Tells the JVM to allocate space for a new Dog object on the heap (we'll learn a lot more about that process, especially in Chapter 9, *Life and Death of an Object*).

3 Link the object and the reference

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

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



there are no Dumb Questions

Q: How big is a reference variable?

A: You don't know. Unless you're cozy with someone on the JVM's development team, you don't know how a reference is represented. There are pointers in there somewhere, but you can't access them. You won't need to. (OK, if you insist, you might as well just imagine it to be a 64-bit value.) But when you're talking about memory allocation issues, your Big Concern should be about how many *objects* (as opposed to object *references*) you're creating and how big *they* (the *objects*) really are.

Q: So, does that mean that all object references are the same size, regardless of the size of the actual objects to which they refer?

A: Yep. All references for a given JVM will be the same size regardless of the objects they reference, but each JVM might have a different way of representing references, so references on one JVM may be smaller or larger than references on another JVM.

Q: Can I do arithmetic on a reference variable, increment it, you know—C stuff?

A: Nope. Say it with me again, "Java is not C."

Java Exposed

This week's interview: Object Reference

HeadFirst: So, tell us, what's life like for an object reference?

Reference: Pretty simple, really. I'm a remote control, and I can be programmed to control different objects.

HeadFirst: Do you mean different objects even while you're running? Like, can you refer to a Dog and then five minutes later refer to a Car?

Reference: Of course not. Once I'm declared, that's it. If I'm a Dog remote control, then I'll never be able to point (oops—my bad, we're not supposed to say *point*), I mean, *refer* to anything but a Dog.

HeadFirst: Does that mean you can refer to only one Dog?

Reference: No. I can be referring to one Dog, and then five minutes later I can refer to some *other* Dog. As long as it's a Dog, I can be redirected (like reprogramming your remote to a different TV) to it. Unless...no never mind.

HeadFirst: No, tell me. What were you gonna say?

Reference: I don't think you want to get into this now, but I'll just give you the short version—if I'm marked as `final`, then once I am assigned a Dog, I can never be reprogrammed to anything else but *that* one and only Dog. In other words, no other object can be assigned to me.

HeadFirst: You're right, we don't want to talk about that now. OK, so unless you're `final`, then you can refer to one Dog and then refer to a different Dog later. Can you ever refer to *nothing at all*? Is it possible to not be programmed to anything?

Reference: Yes, but it disturbs me to talk about it.

HeadFirst: Why is that?

Reference: Because it means I'm `null`, and that's upsetting to me.

HeadFirst: You mean, because then you have no value?

Reference: Oh, `null` *is* a value. I'm still a remote control, but it's like you brought home a new universal remote control and you don't have a TV. I'm not programmed to control anything. They can press my buttons all day long, but nothing good happens. I just feel so...useless. A waste of bits. Granted, not that many bits, but still. And that's not the worst part. If I am the only reference to a particular object and then I'm set to `null` (deprogrammed), it means that now *nobody* can get to that object I had been referring to.

HeadFirst: And that's bad because...

Reference: You have to *ask*? Here I've developed a relationship with this object, an intimate connection, and then the tie is suddenly, cruelly, severed. And I will never see that object again, because now it's eligible for [producer, cue tragic music] *garbage collection*. Sniff. But do you think programmers ever consider *that*? Snif. Why, *why* can't I be a primitive? *I hate being a reference.* The responsibility, all the broken attachments...

Life on the garbage-collectible heap

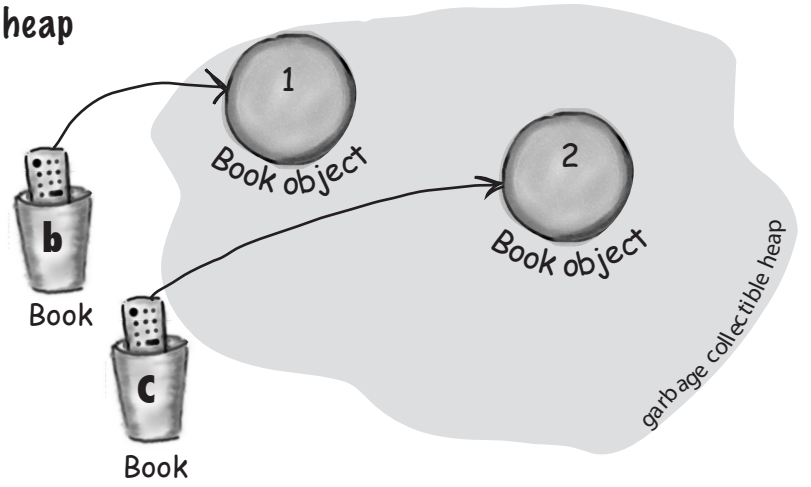
```
Book b = new Book();
Book c = new Book();
```

Declare two Book reference variables. Create two new Book objects. Assign the Book objects to the reference variables.

The two Book objects are now living on the heap.

References: 2

Objects: 2



```
Book d = c;
```

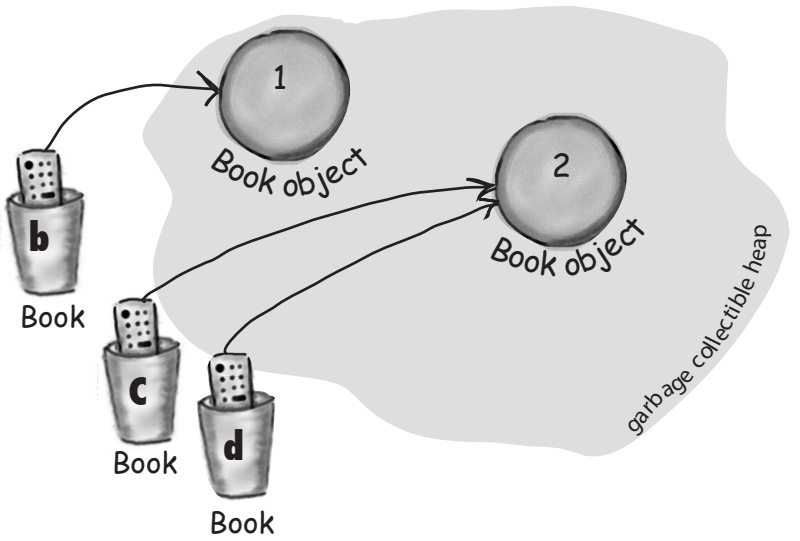
Declare a new Book reference variable. Rather than creating a new, third Book object, assign the value of variable ***c*** to variable ***d***. But what does this mean? It's like saying "Take the bits in ***c***, make a copy of them, and stick that copy into ***d***."

Both *c* and *d* refer to the same object.

The *c* and *d* variables hold two different copies of the same value. Two remotes programmed to one TV.

References: 3

Objects: 2



```
c = b;
```

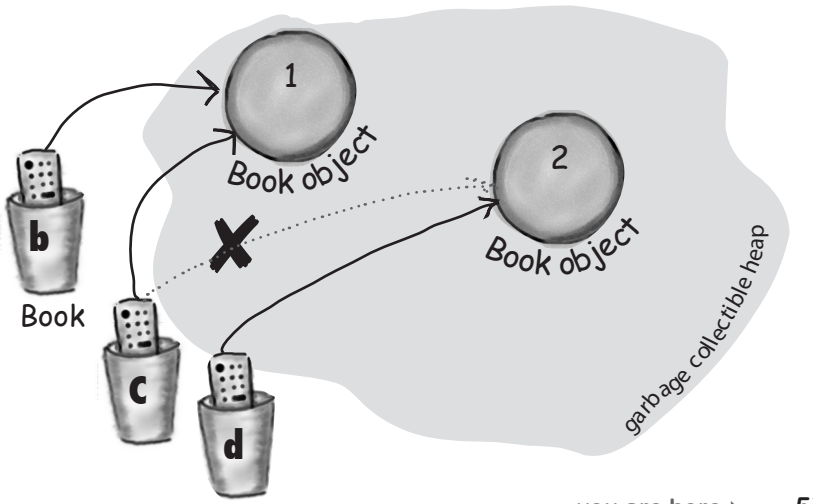
Assign the value of variable ***b*** to variable ***c***. By now you know what this means. The bits inside variable ***b*** are copied, and that new copy is stuffed into variable ***c***.

Both b and c refer to the same object.

The c variable no longer refers to its old Book object.

References: 3

Objects: 2



Life and death on the heap

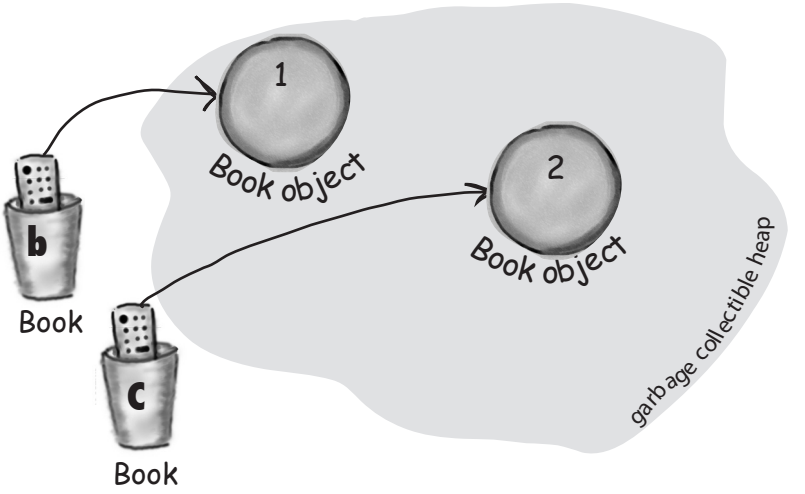
```
Book b = new Book();
Book c = new Book();
```

Declare two Book reference variables. Create two new Book objects. Assign the Book objects to the reference variables.

The two book objects are now living on the heap.

Active References: 2

Reachable Objects: 2



```
b = c;
```

Assign the value of variable ***c*** to variable ***b***. The bits inside variable ***c*** are copied, and that new copy is stuffed into variable ***b***. Both variables hold identical values.

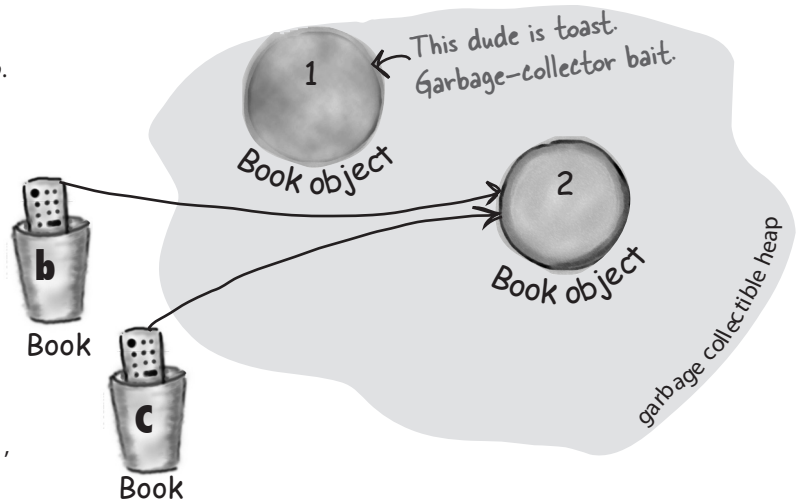
Both b and c refer to the same object. Object 1 is abandoned and eligible for Garbage Collection (GC).

Active References: 2

Reachable Objects: 1

Abandoned Objects: 1

The first object that ***b*** referenced, Object 1, has no more references. It's *unreachable*.



```
c = null;
```

Assign the value `null` to variable ***c***. This makes ***c*** a *null reference*, meaning it doesn't refer to anything. But it's still a reference variable, and another Book object can still be assigned to it.

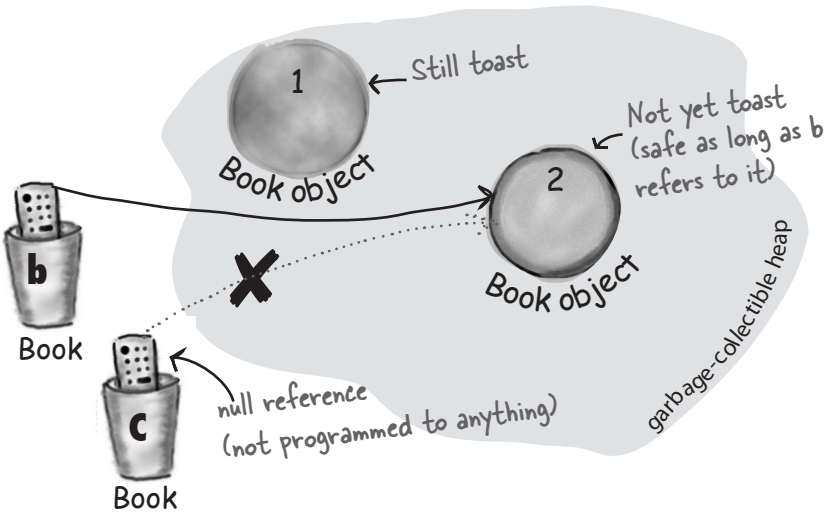
Object 2 still has an active reference (b), and as long as it does, the object is not eligible for GC.

Active References: 1

null References: 1

Reachable Objects: 1

Abandoned Objects: 1



An array is like a tray of cups

The Java standard library includes lots of sophisticated data structures including maps, trees, and sets (see Appendix B), but arrays are great when you just want a quick, ordered, efficient list of things. Arrays give you fast random access by letting you use an index position to get to any element in the array.

Every element in an array is just a variable. In other words, one of the eight primitive variable types (think: Large Furry Dog) or a reference variable. Anything you would put in a *variable* of that type can be assigned to an *array element* of that type. So in an array of type int (int[]), each element can hold an int. In a Dog array (Dog[]) each element can hold...a Dog? No, remember that a reference variable just holds a reference (a remote control), not the object itself. So in a Dog array, each element can hold a *remote control* to a Dog. Of course, we still have to make the Dog objects...and you'll see all that on the next page.

Be sure to notice one key thing in the picture—***the array is an object, even though it's an array of primitives***.

1 Declare an int array variable. An array variable is a remote control to an array object.

```
int[] nums;
```

2 Create a new int array with a length of 7, and assign it to the previously declared `int[]` variable `nums`

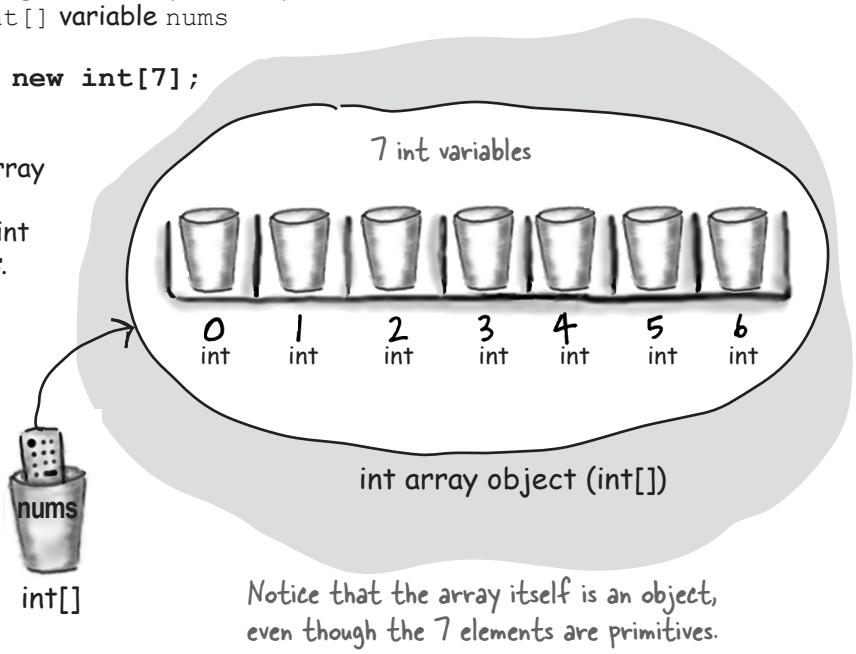
```
nums = new int[7];
```

3 Give each element in the array some int value. Remember, elements in an int *array* are just int *variables*.

```
nums[0] = 6;
nums[1] = 19;
nums[2] = 44;
nums[3] = 42;
nums[4] = 10;
nums[5] = 20;
nums[6] = 1;
```

7 int variables

Arrays are always objects, whether they're declared to hold primitives or object references.



Notice that the array itself is an object, even though the 7 elements are primitives.

Arrays are objects too

You can have an array object that's declared to *hold* primitive values. In other words, the array object can have *elements* that are primitives, but the array itself is *never* a primitive.
Regardless of what the array holds, the array itself is always an object!

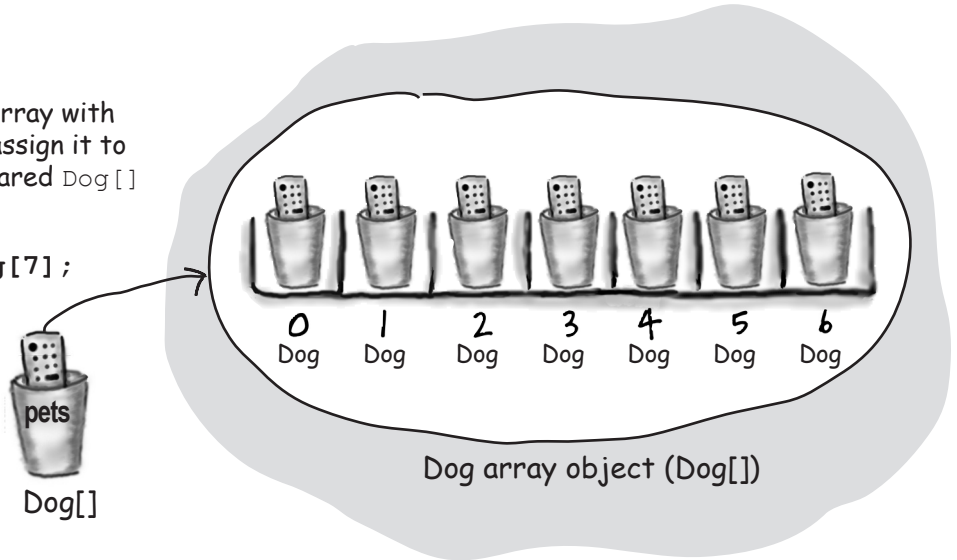
Make an array of Dogs

1. Declare a Dog array variable

```
Dog[] pets;
```

2. Create a new Dog array with a length of 7, and assign it to the previously declared `Dog[]` variable `pets`

```
pets = new Dog[7];
```

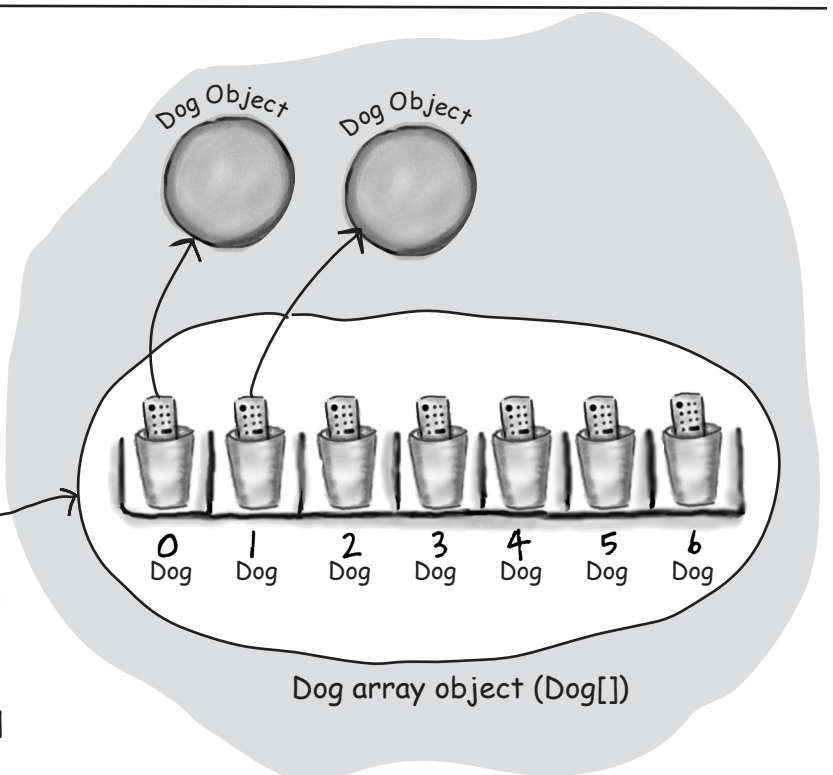


What's missing?

Dogs! We have an array of Dog *references*, but no actual Dog *objects*!

3. Create new Dog objects, and assign them to the array elements.
Remember, elements in a Dog *array* are just Dog reference *variables*. We still need Dogs!

```
pets[0] = new Dog();
pets[1] = new Dog();
```



→ Yours to solve.

Sharpen your pencil

What is the current value of pets[2]? _____

What code would make pets[3] refer to one of the two existing Dog objects?

_____

Dog
name
bark() eat() chaseCat()

Control your Dog

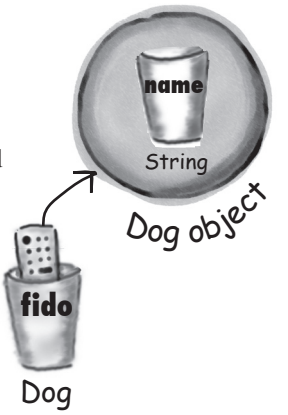
(with a reference variable)

```
Dog fido = new Dog();
fido.name = "Fido";
```

We created a Dog object and used the dot operator on the reference variable ***fido*** to access the name variable.*

We can use the ***fido*** reference to get the dog to bark() or eat() or chaseCat().

```
fido.bark();
fido.chaseCat();
```



What happens if the Dog is in a Dog array?

We know we can access the Dog's instance variables and methods using the dot operator, but *on what?*

When the Dog is in an array, we don't have an actual variable name (like ***fido***). Instead we use array notation and push the remote control button (dot operator) on an object at a particular index (position) in the array:

```
Dog[] myDogs = new Dog[3];
myDogs[0] = new Dog();
myDogs[0].name = "Fido";
myDogs[0].bark();
```

Java cares about type.

Once you've declared an array, you can't put anything in it except things that are of a compatible array type.

For example, you can't put a Cat into a Dog array (it would be pretty awful if someone thinks that only Dogs are in the array, so they ask each one to bark, and then to their horror discover there's a cat lurking.) And you can't stick a `double` into an `int` array (spillage, remember?). You can, however, put a `byte` into an `int` array, because a `byte` will always fit into an `int`-sized cup. This is known as an **implicit widening**. We'll get into the details later; for now just remember that the compiler won't let you put the wrong thing in an array, based on the array's declared type.

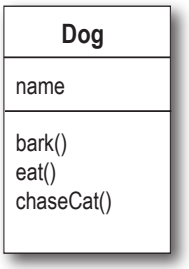
*Yes we know we're not demonstrating encapsulation here, but we're trying to keep it simple. For now. We'll do encapsulation in Chapter 4.

```
class Dog {
  String name;

  public static void main(String[] args) {
    // make a Dog object and access it
    Dog dog1 = new Dog();
    dog1.bark();
    dog1.name = "Bart";

    // now make a Dog array
    Dog[] myDogs = new Dog[3];
    // and put some dogs in it
    myDogs[0] = new Dog();
    myDogs[1] = new Dog();
    myDogs[2] = dog1;

    // now access the Dogs using the array
    // references
    myDogs[0].name = "Fred";
    myDogs[1].name = "Marge";

    // Hmmmm... what is myDogs[2] name?
    System.out.print("last dog's name is ");
    System.out.println(myDogs[2].name);

    // now loop through the array
    // and tell all dogs to bark
    int x = 0;
    while (x < myDogs.length) {
      myDogs[x].bark();
      x = x + 1;
    }
  }

  public void bark() {
    System.out.println(name + " says Ruff!");
  }

  public void eat() {
  }

  public void chaseCat() {
  }
}
```

Strings are a special type of object. You can create and assign them as if they were primitives (even though they're references).

Arrays have a variable 'length' that gives you the number of elements in the array.

A Dog example



Output

```
File  Edit  Window  Help  Howl
%java Dog
null says Ruff!
last dog's name is Bart
Fred says Ruff!
Marge says Ruff!
Bart says Ruff!
```

BULLET POINTS

- Variables come in two flavors: primitive and reference.
- Variables must always be declared with a name and a type.
- A primitive variable value is the bits representing the value (5, 'a', true, 3.1416, etc.).
- A reference variable value is the bits representing a way to get to an object on the heap.
- A reference variable is like a remote control. Using the dot operator (.) on a reference variable is like pressing a button on the remote control to access a method or instance variable.
- A reference variable has a value of `null` when it is not referencing any object.
- An array is always an object, even if the array is declared to hold primitives. There is no such thing as a primitive array, only an array that *holds* primitives.

BE the Compiler

Each of the Java files on this page represents a complete source file. Your job is to play compiler and determine whether each of these files will compile and run without exception. If they won't, how would you fix them?

A

```
class Books {
  String title;
  String author;
}

class BooksTestDrive {
  public static void main(String[] args) {
    Books[] myBooks = new Books[3];
    int x = 0;
    myBooks[0].title = "The Grapes of Java";
    myBooks[1].title = "The Java Gatsby";
    myBooks[2].title = "The Java Cookbook";
    myBooks[0].author = "bob";
    myBooks[1].author = "sue";
    myBooks[2].author = "ian";

    while (x < 3) {
      System.out.print(myBooks[x].title);
      System.out.print(" by ");
      System.out.println(myBooks[x].author);
      x = x + 1;
    }
  }
}
```

B

```
class Hobbits {
  String name;

  public static void main(String[] args) {
    Hobbits[] h = new Hobbits[3];
    int z = 0;

    while (z < 4) {
      z = z + 1;
      h[z] = new Hobbits();
      h[z].name = "bilbo";
      if (z == 1) {
        h[z].name = "frodo";
      }
      if (z == 2) {
        h[z].name = "sam";
      }
      System.out.print(h[z].name + " is a ");
      System.out.println("good Hobbit name");
    }
  }
}
```

Answers on page 68.

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!

```
int y = 0;
```

```
ref = index[y];
```

```
islands[0] = "Bermuda";
islands[1] = "Fiji";
islands[2] = "Azores";
islands[3] = "Cozumel";
```

```
int ref;
while (y < 4) {
```

```
System.out.println(islands[ref]);
```

```
index[0] = 1;
index[1] = 3;
index[2] = 0;
index[3] = 2;
```

```
String [] islands = new String[4];
```

```
System.out.print("island = ");
```

```
int [] index = new int[4];
```

```
y = y + 1;
```

```
File  Edit  Window  Help  Sunscreen
% java TestArrays
island = Fiji
island = Cozumel
island = Bermuda
island = Azores
```

```
class TestArrays {

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


Answers on page 68.

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 won't need to use all the snippets. Your ***goal*** is to make a class that will compile and run and produce the output listed.

Output

```
File  Edit  Window  Help  Bermuda
%java Triangle
triangle 0, area = 4.0
triangle 1, area = 10.0
triangle 2, area = 18.0
triangle 3, area = ____
y = _____________________
```

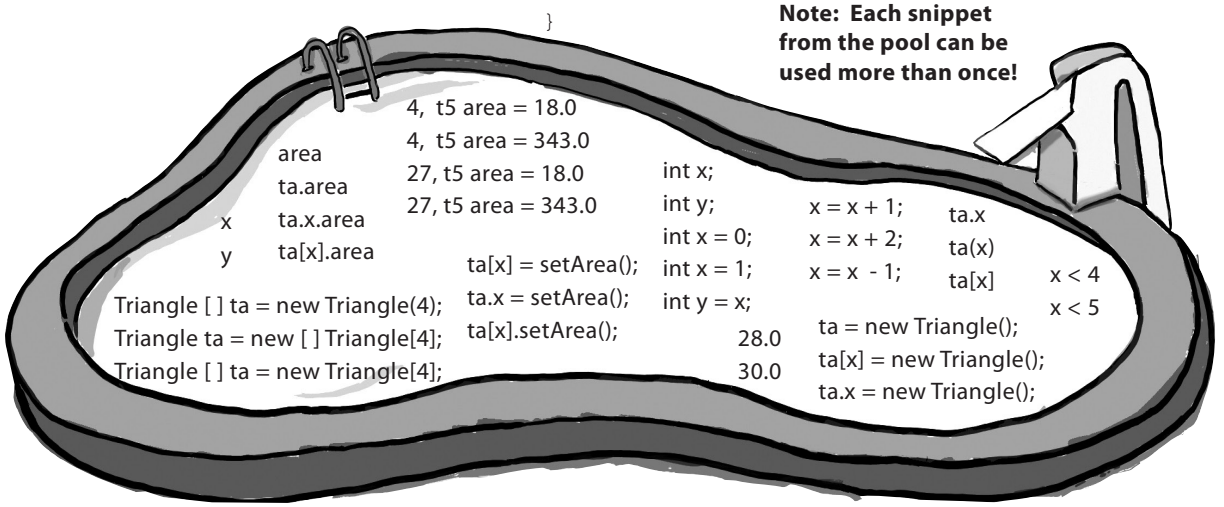
Bonus Question!

For extra bonus points, use snippets from the pool to fill in the missing output (above).

```
class Triangle {
  double area;
  int height;
  int length;

  public static void main(String[] args) {
    ____________
    ____________________________
    while ( ___________ ) {
      ____________________________
      _________.height = (x + 1) * 2;
      _________.length = x + 4;
      ____________________________
      System.out.print("triangle " + x + ", area");
      System.out.println(" = " + ________.area);
      ________________
    }
    _______________
    x = 27;
    Triangle t5 = ta[2];
    ta[2].area = 343;
    System.out.print("y = " + y);
    System.out.println(", t5 area = " + t5.area);
  }
  void setArea() {
    _____________ = (height * length) / 2;
  }
}
```

(Sometimes we don't use a separate test class, because we're trying to save space on the page.)

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



Answers on page 69.

A Heap o' Trouble

A short Java program is listed to the right. When "// do stuff" is reached, some objects and some reference variables will have been created. Your task is to determine which of the reference variables refer to which objects. Not all the reference variables will be used, and some objects might be referred to more than once. Draw lines connecting the reference variables with their matching objects.

Tip: Unless you're way smarter than we are, you probably need to draw diagrams like the ones on page 57–60 of this chapter. Use a pencil so you can draw and then erase reference links (the arrows going from a reference remote control to an object).

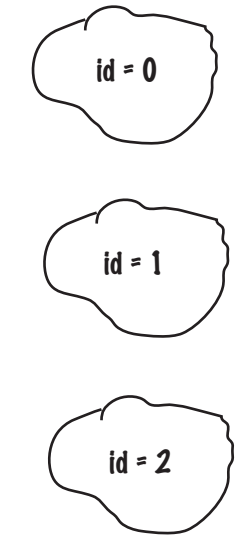
```
class HeapQuiz {
  int id = 0;

  public static void main(String[] args) {
    int x = 0;
    HeapQuiz[] hq = new HeapQuiz[5];
    while (x < 3) {
      hq[x] = new HeapQuiz();
      hq[x].id = x;
      x = x + 1;
    }
    hq[3] = hq[1];
    hq[4] = hq[1];
    hq[3] = null;
    hq[4] = hq[0];
    hq[0] = hq[3];
    hq[3] = hq[2];
    hq[2] = hq[0];
    // do stuff
  }
}
```

Match each reference variable with matching object(s).

You might not have to use every reference.

Reference Variables:

HeapQuiz Objects:



Answers on page 69.

The case of the pilfered references

It was a dark and stormy night. Tawny strolled into the programmers' bullpen like she owned the place. She knew that all the programmers would still be hard at work, and she wanted help. She needed a new method added to the pivotal class that was to be loaded into the client's new top-secret Java-enabled cell phone. Heap space in the cell phone's memory was tight, and everyone knew it. The normally raucous buzz in the bullpen fell to silence as Tawny eased her way to the white board. She sketched a quick overview of the new method's functionality and slowly scanned the room. "Well folks, it's crunch time," she purred. "Whoever creates the most memory efficient version of this method is coming with me to the client's launch party on Maui tomorrow...to help me install the new software."

Five-Minute Mystery

The next morning Tawny glided into the bullpen. "Ladies and Gentlemen," she smiled, "the plane leaves in a few hours, show me what you've got!" Bob went first; as he began to sketch his design on the white board, Tawny said, "Let's get to the point Bob, show me how you handled updating the list of contact objects." Bob quickly drew a code fragment on the board:

```
Contact [] contacts = new Contact[10];
while (x < 10 ) {   // make 10 contact objects
  contacts[x] = new Contact();
  x = x + 1;
}
// do complicated Contact list updating with contacts
```

"Tawny, I know we're tight on memory, but your spec said that we had to be able to access individual contact information for all ten allowable contacts; this was the best scheme I could cook up," said Bob. Kate was next, already imagining coconut cocktails at the party, "Bob," she said, "your solution's a bit kludgy, don't you think?" Kate smirked, "Take a look at this baby":

```
Contact contactRef;
while ( x < 10 ) {   // make 10 contact objects
  contactRef = new Contact();
  x = x + 1;
}
// do complicated Contact list updating with contactRef
```

"I saved a bunch of reference variables worth of memory, Bob-o-rino, so put away your sunscreen," mocked Kate. "Not so fast Kate!" said Tawny, "you've saved a little memory, but Bob's coming with me."

Why did Tawny choose Bob's method over Kate's, when Kate's used less memory?

Answers on page 69.

Exercise Solutions

Sharpen your pencil (from page 52)

1. int x = 34.5; ✗
2. boolean boo = x; ✗
3. int g = 17; ✓
4. int y = g; ✓
5. y = y + 10; ✓
6. short s; ✓
7. s = y; ✗
8. byte b = 3; ✓
9. byte v = b; ✓
10. short n = 12; ✓
11. v = n; ✗
12. byte k = 128; ✗

Code Magnets (from page 64)

```
class TestArrays {
  public static void main(String[] args) {
    int[] index = new int[4];
    index[0] = 1;
    index[1] = 3;
    index[2] = 0;
    index[3] = 2;
    String[] islands = new String[4];
    islands[0] = "Bermuda";
    islands[1] = "Fiji";
    islands[2] = "Azores";
    islands[3] = "Cozumel";
    int y = 0;
    int ref;
    while (y < 4) {
      ref = index[y];
      System.out.print("island = ");
      System.out.println(islands[ref]);
      y = y + 1;
    }
  }
}
```

```
File  Edit  Window  Help  Sunscreen
% java TestArrays
island = Fiji
island = Cozumel
island = Bermuda
island = Azores
```

BE the Compiler (from page 63)

```
class Books {
  String title;
  String author;
}

class BooksTestDrive {
  public static void main(String[] args) {
    Books[] myBooks = new Books[3];
    int x = 0;
    myBooks[0] = new Books();
    myBooks[1] = new Books();
    myBooks[2] = new Books();
    myBooks[0].title = "The Grapes of Java";
    myBooks[1].title = "The Java Gatsby";
    myBooks[2].title = "The Java Cookbook";
    myBooks[0].author = "bob";
    myBooks[1].author = "sue";
    myBooks[2].author = "ian";
    while (x < 3) {
      System.out.print(myBooks[x].title);
      System.out.print(" by ");
      System.out.println(myBooks[x].author);
      x = x + 1;
    }
  }
}
```

A

Remember: We have to actually make the Book objects!

```
class Hobbits {
  String name;

  public static void main(String[] args) {
    Hobbits[] h = new Hobbits[3];
    int z = -1;
    while (z < 2) {
      z = z + 1;
      h[z] = new Hobbits();
      h[z].name = "bilbo";
      if (z == 1) {
        h[z].name = "frodo";
      }
      if (z == 2) {
        h[z].name = "sam";
      }
      System.out.print(h[z].name + " is a ");
      System.out.println("good Hobbit name");
    }
  }
}
```

B

Remember: arrays start with element 0!

Puzzle Solutions

Pool Puzzle (from page 65)

```
class Triangle {
  double area;
  int height;
  int length;

  public static void main(String[] args) {
    int x = 0;
    Triangle[] ta = new Triangle[4];
    while (x < 4) {
      ta[x] = new Triangle();
      ta[x].height = (x + 1) * 2;
      ta[x].length = x + 4;
      ta[x].setArea();
      System.out.print("triangle " + x +
                       ", area");
      System.out.println(" = " + ta[x].area);
      x = x + 1;
    }
    int y = x;
    x = 27;
    Triangle t5 = ta[2];
    ta[2].area = 343;
    System.out.print("y = " + y);
    System.out.println(", t5 area = " +
                       t5.area);
  }

  void setArea() {
    area = (height * length) / 2;
  }
}
```

```
File  Edit  Window  Help  Bermuda
%java Triangle
triangle 0, area = 4.0
triangle 1, area = 10.0
triangle 2, area = 18.0
triangle 3, area = 28.0
y = 4, t5 area = 343.0
```

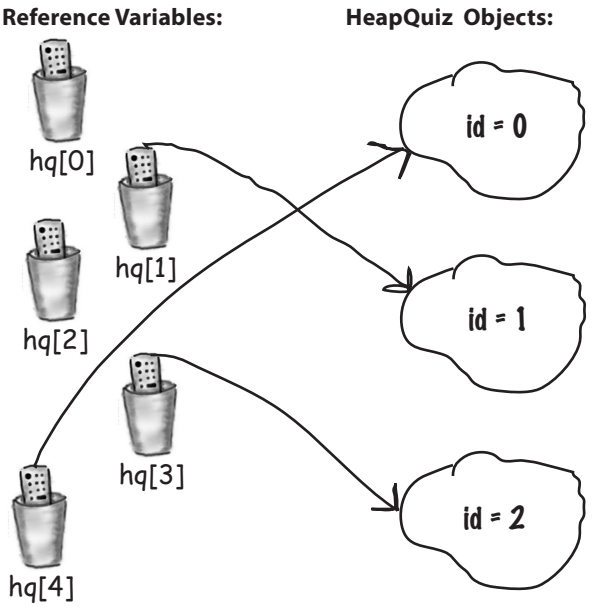
Five-Minute Mystery (from page 67)

The case of the pilfered references

Tawny could see that Kate's method had a serious flaw. It's true that she didn't use as many reference variables as Bob, but there was no way to access any but the last of the Contact objects that her method created. With each trip through the loop, she was assigning a new object to the one reference variable, so the previously referenced object was abandoned on the heap—*unreachable*. Without access to nine of the ten objects created, Kate's method was useless.

(The software was a huge success, and the client gave Tawny and Bob an extra week in Hawaii. We'd like to tell you that by finishing this book you too will get stuff like that.)

A Heap o' Trouble (from page 66)

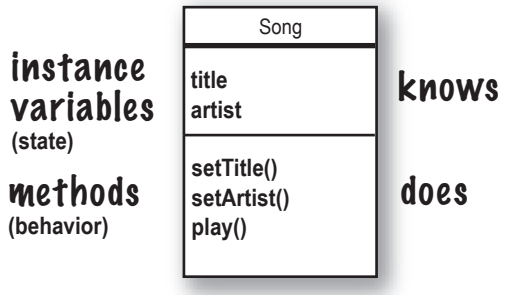


How Objects Behave

State affects behavior, behavior affects state. We know that objects have **state** and **behavior**, represented by **instance variables** and **methods**. But until now, we haven't looked at how state and behavior are *related*. We already know that each instance of a class (each object of a particular type) can have its own unique values for its instance variables. Dog A can have a *name* "Fido" and a *weight* of 70 pounds. Dog B is "Killer" and weighs 9 pounds. And if the Dog class has a method makeNoise(), well, don't you think a 70-pound dog barks a bit deeper than the little 9-pounder? (Assuming that annoying yippy sound can be considered a *bark*.) Fortunately, that's the whole point of an object—it has *behavior* that acts on its *state*. In other words, ***methods use instance variable values***. Like, "if dog is less than 14 pounds, make yippy sound, else..." or "increase weight by 5." ***Let's go change some state.***

Remember: a class describes what an object knows and what an object does

A class is the blueprint for an object. When you write a class, you're describing how the JVM should make an object of that type. You already know that every object of that type can have different *instance variable* values. But what about the methods?



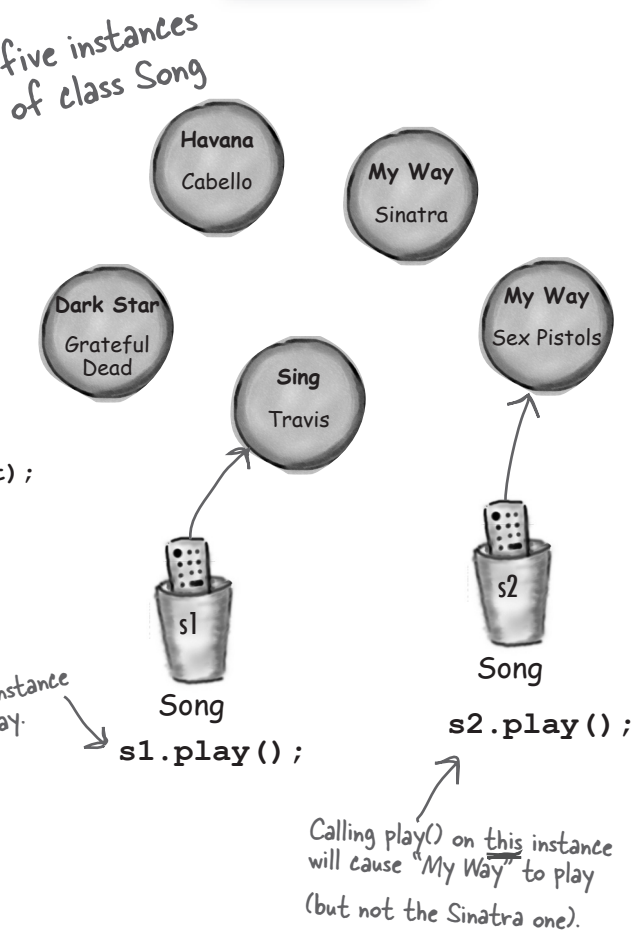
Can every object of that type have different method behavior?

Well...***sort of.****

Every instance of a particular class has the same methods, but the methods can *behave* differently based on the value of the instance variables.

The Song class has two instance variables, *title* and *artist*. When you call the play() method on an the instance, it will play the song represented by the value of the *title* and *artist* instance variables for that instance. So, if you call the play() method on one instance, you'll hear the song "Havana" by Cabello, while another instance plays "Sing" by Travis. The method code, however, is the same.

```
void play() {
    soundPlayer.playSound(title, artist);
}
```



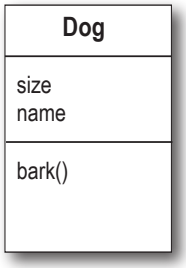

```
Song song1 = new Song();
song1.setArtist("Travis");
song1.setTitle("Sing");
Song song2 = new Song();
song2.setArtist("Sex Pistols");
song2.setTitle("My Way");
```

*Yes, another stunningly clear answer!

The size affects the bark

A small Dog's bark is different from a big Dog's bark.

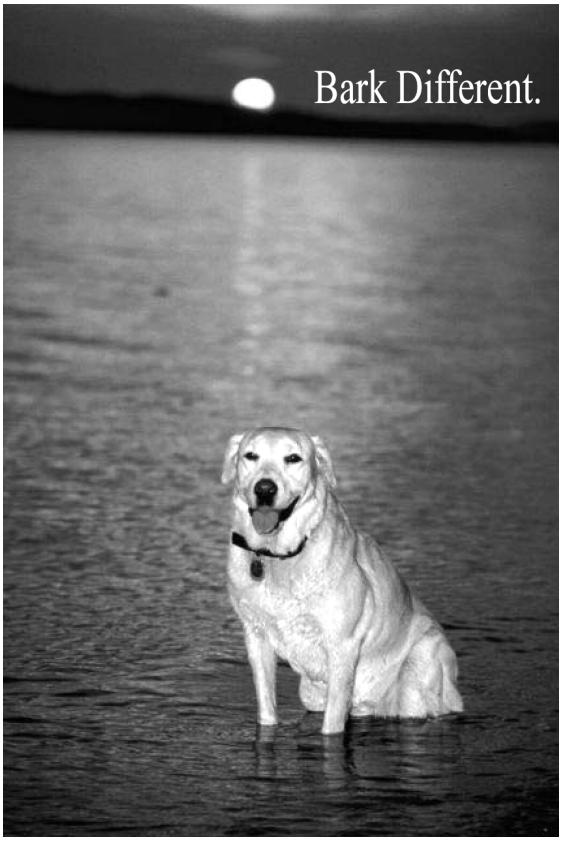
The Dog class has an instance variable *size* that the *bark()* method uses to decide what kind of bark sound to make.




```
class Dog {
  int size;
  String name;

  void bark() {
    if (size > 60) {
      System.out.println("Wooof! Wooof!");
    } else if (size > 14) {
      System.out.println("Ruff!  Ruff!");
    } else {
      System.out.println("Yip! Yip!");
    }
  }
}
```

```
class DogTestDrive {

  public static void main(String[] args) {
    Dog one = new Dog();
    one.size = 70;
    Dog two = new Dog();
    two.size = 8;
    Dog three = new Dog();
    three.size = 35;

    one.bark();
    two.bark();
    three.bark();
  }
}
```

```
File Edit Window Help Playdead
%java DogTestDrive
Wooof! Wooof!
Yip! Yip!
Ruff!  Ruff!
```



You can send things to a method

Just as you expect from any programming language, you can pass values into your methods. You might, for example, want to tell a Dog object how many times to bark by calling:

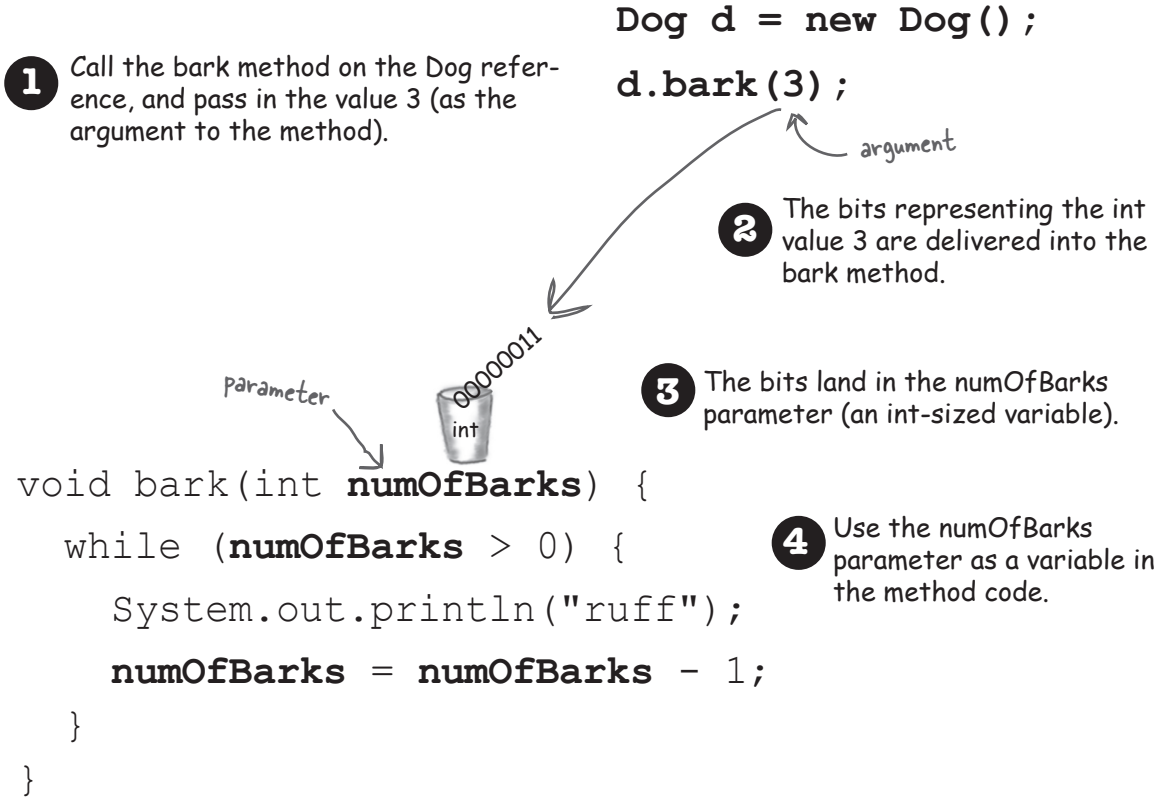
```
d.bark(3);
```

Depending on your programming background and personal preferences, *you* might use the term *arguments* or perhaps *parameters* for the values passed into a method. Although there *are* formal computer science distinctions that people who wear lab coats (and who will almost certainly not read this book) make, we have bigger fish to fry in this book. So *you* can call them whatever you like (arguments, donuts, hairballs, etc.) but we're doing it like this:

A caller passes arguments. A method takes parameters.

Arguments are the things you pass into the methods. An ***argument*** (a value like 2, Foo, or a reference to a Dog) lands face-down into a...wait for it...***parameter***. And a parameter is nothing more than a local variable. A variable with a type and a name that can be used inside the body of the method.

But here's the important part: **If a method takes a parameter, you *must* pass it something when you call it.** And that something must be a value of the appropriate type.



You can get things *back* from a method

Methods can also *return* values. Every method is declared with a return type, but until now we've made all of our methods with a **void** return type, which means they don't give anything back.

```
void go() {
}
```

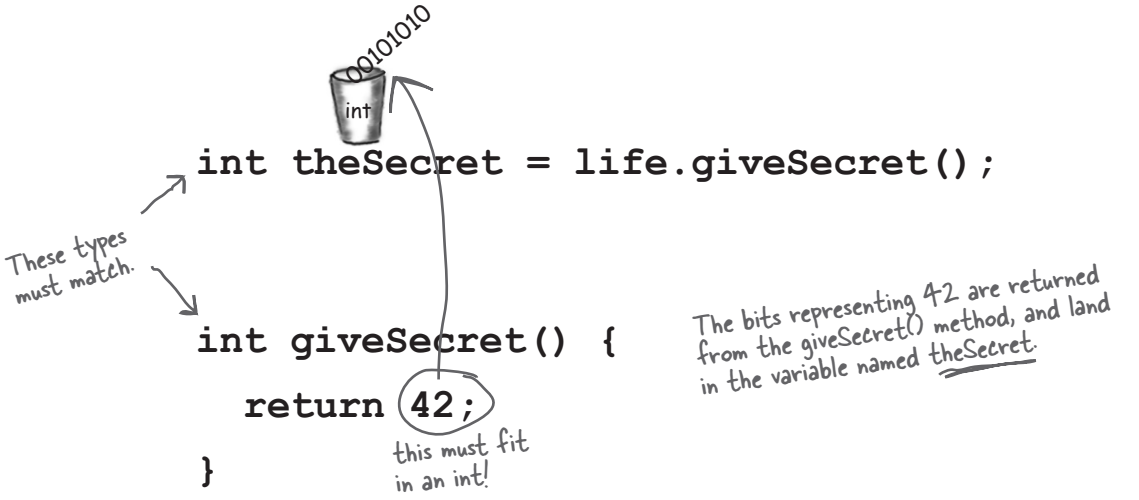
But we can declare a method to give a specific type of value back to the caller, such as:

```
int giveSecret() {
  return 42;
}
```

If you declare a method to return a value, you *must* return a value of the declared type! (Or a value that is *compatible* with the declared type. We'll get into that more when we talk about polymorphism in Chapters 7 and 8.)

Whatever you *say* you'll give back, you *better* give back!

The compiler won't let you return the wrong type of thing.

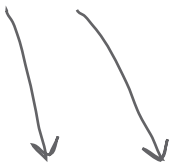


You can send more than one thing to a method

Methods can have multiple parameters. Separate them with commas when you declare them, and separate the arguments with commas when you pass them. Most importantly, if a method has parameters, you *must* pass arguments of the right type and order.

Calling a two-parameter method and sending it two arguments

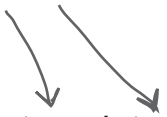
```
void go() {
  TestStuff t = new TestStuff();
  t.takeTwo(12, 34);
}

void takeTwo(int x, int y) {
  int z = x + y;
  System.out.println("Total is " + z);
}
```



The arguments you pass land in the same order you passed them. First argument lands in the first parameter, second argument in the second parameter, and so on.

You can pass variables into a method, as long as the variable type matches the parameter type

```
void go() {
   int foo = 7;
   int bar = 3;
   t.takeTwo(foo, bar);
}

void takeTwo(int x, int y) {
   int z = x + y;
   System.out.println("Total is " + z);
}
```



The values of foo and bar land in the x and y parameters. So now the bits in x are identical to the bits in foo (the bit pattern for the integer '7'), and the bits in y are identical to the bits in bar.

What's the value of z? It's the same result you'd get if you added foo + bar at the time you passed them into the takeTwo method.

Java is pass-by-value.

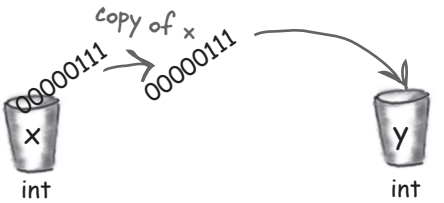
That means pass-by-copy.

```
int x = 7;
```

1 Declare an int variable and assign it the value '7'. The bit pattern for 7 goes into the variable named x.


```
void go(int z){ }
```

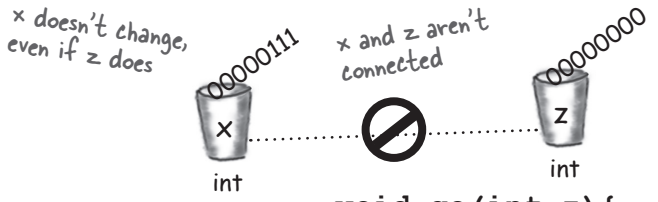
2 Declare a method with an int parameter named z.




```
foo.go(x);
```

```
void go(int z){ }
```

3 Call the go() method, passing the variable x as the argument. The bits in x are copied, and the copy lands in z.




```
void go(int z){
    z = 0;
}
```

4 Change the value of z inside the method. The value of x doesn't change! The argument passed to the z parameter was only a copy of x.

The method can't change the bits that were in the calling variable x.

there are no Dumb Questions

Q: What happens if the argument you want to pass is an object instead of a primitive?

A: You'll learn more about this in later chapters, but you already *know* the answer. Java passes *everything* by value. ***Everything***. But...*value* means *bits inside the variable*. And remember, you don't stuff objects into variables; the variable is a remote control—*a reference to an object*. So if you pass a reference to an object into a method, you're passing a *copy of the remote control*. Stay tuned, though, we'll have lots more to say about this.

Q: Can a method declare multiple return values? Or is there some way to return more than one value?

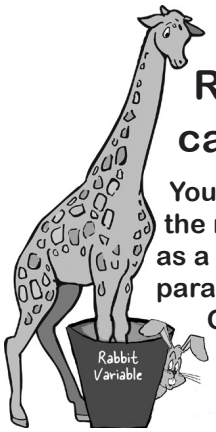
A: Sort of. A method can declare only one return value. BUT...if you want to return, say, three int values, then the declared return type can be an int *array*. Stuff those ints into the array, and pass it on back. It's a little more involved to return multiple values with different types; we'll be talking about that in a later chapter when we talk about ArrayList.

Q: Do I have to return the exact type I declared?

A: You can return anything that can be *implicitly* promoted to that type. So, you can pass a byte where an int is expected. The caller won't care, because the byte fits just fine into the int the caller will use for assigning the result. You must use an *explicit* cast when the declared type is *smaller* than what you're trying to return (we'll see these in Chapter 5).

Q: Do I have to do something with the return value of a method? Can I just ignore it?

A: Java doesn't require you to acknowledge a return value. You might want to call a method with a non-void return type, even though you don't care about the return value. In this case, you're calling the method for the work it does *inside* the method, rather than for what the method gives *returns*. In Java, you don't have to assign or use the return value.



Reminder: Java cares about type!

You can't return a Giraffe when the return type is declared as a Rabbit. Same thing with parameters. You can't pass a Giraffe into a method that takes a Rabbit.

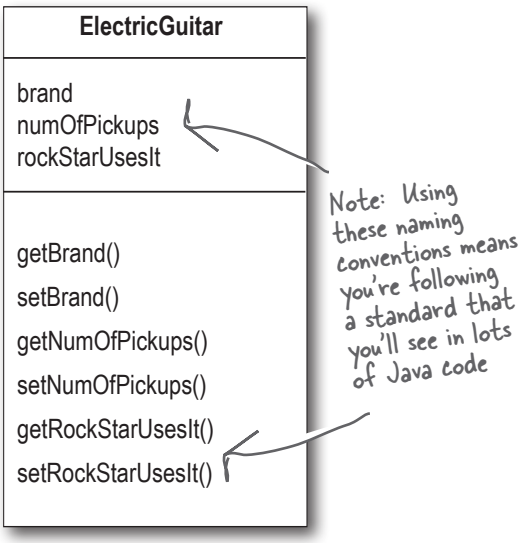
BULLET POINTS

- Classes define what an object knows and what an object does.
- Things an object knows are its **instance variables** (state).
- Things an object does are its **methods** (behavior).
- Methods can use instance variables so that objects of the same type can behave differently.
- A method can have parameters, which means you can pass one or more values in to the method.
- The number and type of values you pass in must match the order and type of the parameters declared by the method.
- Values passed in and out of methods can be implicitly promoted to a larger type or explicitly *cast* to a smaller type.
- The value you pass as an argument to a method can be a literal value (2, 'c', etc.) or a variable of the declared parameter type (for example, *x* where *x* is an int variable). (There are other things you can pass as arguments, but we're not there yet.)
- A method *must* declare a return type. A void return type means the method doesn't return anything.
- If a method declares a non-void return type, it *must* return a value compatible with the declared return type.

Cool things you can do with parameters and return types

Now that we've seen how parameters and return types work, it's time to put them to good use: let's create **Getters** and **Setters**. If you're into being all formal about it, you might prefer to call them *Accessors* and *Mutators*. But that's a waste of perfectly good syllables. Besides, Getters and Setters fits a common Java naming convention, so that's what we'll call them.

Getters and Setters let you, well, *get and set things*. Instance variable values, usually. A Getter's sole purpose in life is to send back, as a return value, the value of whatever it is that particular Getter is supposed to be Getting. And by now, it's probably no surprise that a Setter lives and breathes for the chance to take an argument value and use it to *set* the value of an instance variable.




```
class ElectricGuitar {
  String brand;
  int numOfPickups;
  boolean rockStarUsesIt;

  String getBrand() {
    return brand;
  }

  void setBrand(String aBrand) {
    brand = aBrand;
  }

  int getNumOfPickups() {
    return numOfPickups;
  }

  void setNumOfPickups(int num) {
    numOfPickups = num;
  }

  boolean getRockStarUsesIt() {
    return rockStarUsesIt;
  }

  void setRockStarUsesIt(boolean yesOrNo) {
    rockStarUsesIt = yesOrNo;
  }
}
```

Encapsulation

Do it or risk humiliation and ridicule.

Until this most important moment, we've been committing one of the worst OO faux pas (and we're not talking minor violation like showing up without the "B" in BYOB). No, we're talking Faux Pas with a capital "F." And "P."

Our shameful transgression?

Exposing our data!

Here we are, just humming along without a care in the world leaving our data out there for *anyone* to see and even touch.

You may have already experienced that vaguely unsettling feeling that comes with leaving your instance variables exposed.

Exposed means reachable with the dot operator, as in:

```
theCat.height = 27;
```

Think about this idea of using our remote control to make a direct change to the Cat object's size instance variable. In the hands of the wrong person, a reference variable (remote control) is quite a dangerous weapon. Because what's to prevent:


```
theCat.height = 0;
```

Oh my goodness! We can't let this happen!

This would be a Bad Thing. We need to build setter methods for all the instance variables, and find a way to force other code to call the setters rather than access the data directly.

By forcing everybody to call a setter method, we can protect the cat from unacceptable size changes.

```
public void setHeight(int ht) {
    if (ht > 9) {
        height = ht;
    }
}
```

We put in checks to guarantee a minimum cat height.

Hide the data

Yes, it *is* that simple to go from an implementation that's just begging for bad data to one that protects your data *and* protects your right to modify your implementation later.

OK, so how exactly do you *hide* the data? With the **`public`** and **`private`** access modifiers. You're familiar with **`public`**—we use it with every main method.

Here's an encapsulation *starter* rule of thumb (all standard disclaimers about rules of thumb are in effect): mark your instance variables ***private*** and provide ***public*** getters and setters for access control. When you have more design and coding savvy in Java, you will probably do things a little differently, but for now, this approach will keep you safe.

Mark instance variables `private`.

Mark getters and setters `public`.

"Sadly, Bill forgot to encapsulate his Cat class and ended up with a flat cat."

(overheard at the water cooler)

Java Exposed

This week's interview:
An Object gets candid about encapsulation.

HeadFirst: What's the big deal about encapsulation?

Object: OK, you know that dream where you're giving a talk to 500 people when you suddenly realize you're *naked*?

HeadFirst: Yeah, we've had that one. It's right up there with the one about the Pilates machine and...no, we won't go there. OK, so you feel naked. But other than being a little exposed, is there any danger?

Object: Is there any danger? Is there any *danger*? [starts laughing] Hey, did all you other instances hear that, *"Is there any danger?"* he asks? [falls on the floor laughing]

HeadFirst: What's funny about that? Seems like a reasonable question.

Object: OK, I'll explain it. It's [bursts out laughing again, uncontrollably]

HeadFirst: Can I get you anything? Water?

Object: Whew! Oh boy. No I'm fine, really. I'll be serious. Deep breath. OK, go on.

HeadFirst: So what does encapsulation protect you from?

Object: Encapsulation puts a force-field around my instance variables, so nobody can set them to, let's say, something *inappropriate*.

HeadFirst: Can you give me an example?

Object: Happy to. Most instance variable values are coded with certain assumptions about their boundaries. Like, think of all the things that would break if negative numbers were allowed. Number of bathrooms in an office. Velocity of an airplane. Birthdays. Barbell weight. Phone numbers. Microwave oven power.

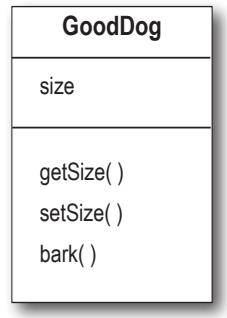
HeadFirst: I see what you mean. So how does encapsulation let you set boundaries?

Object: By forcing other code to go through setter methods. That way, the setter method can validate the parameter and decide if it's doable. Maybe the method will reject it and do nothing, or maybe it'll throw an Exception (like if it's a null Social Security number for a credit card application), or maybe the method will round the parameter sent in to the nearest acceptable value. The point is, you can do whatever you want in the setter method, whereas you can't do *anything* if your instance variables are public.

HeadFirst: But sometimes I see setter methods that simply set the value without checking anything. If you have an instance variable that doesn't have a boundary, doesn't that setter method create unnecessary overhead? A performance hit?

Object: The point to setters (and getters, too) is that ***you can change your mind later, without breaking anybody else's code!*** Imagine if half the people in your company used your class with public instance variables, and one day you suddenly realized, "Oops—there's something I didn't plan for with that value, I'm going to have to switch to a setter method." You break everyone's code. The cool thing about encapsulation is that *you get to change your mind.* And nobody gets hurt. The performance gain from using variables directly is so miniscule and would rarely—*if ever*—be worth it.

Encapsulating the GoodDog class



Make the instance variable private.

Make the getter and setter methods public.

```
class GoodDog {
  private int size;

  public int getSize() {
    return size;
  }

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

  void bark() {
    if (size > 60) {
      System.out.println("Wooof! Wooof!");
    } else if (size > 14) {
      System.out.println("Ruff!  Ruff!");
    } else {
      System.out.println("Yip! Yip!");
    }
  }
}

class GoodDogTestDrive {

  public static void main(String[] args) {
    GoodDog one = new GoodDog();
    one.setSize(70);
    GoodDog two = new GoodDog();
    two.setSize(8);
    System.out.println("Dog one: " + one.getSize());
    System.out.println("Dog two: " + two.getSize());
    one.bark();
    two.bark();
  }
}
```

Even though the methods don't really add new functionality, the nice thing is that you can change your mind later. You can come back and make a method safer, faster, better.

Any place where a particular value can be used, a *method call that returns that type can be used.*

instead of:

```
int x = 3 + 24;
```

you can say:

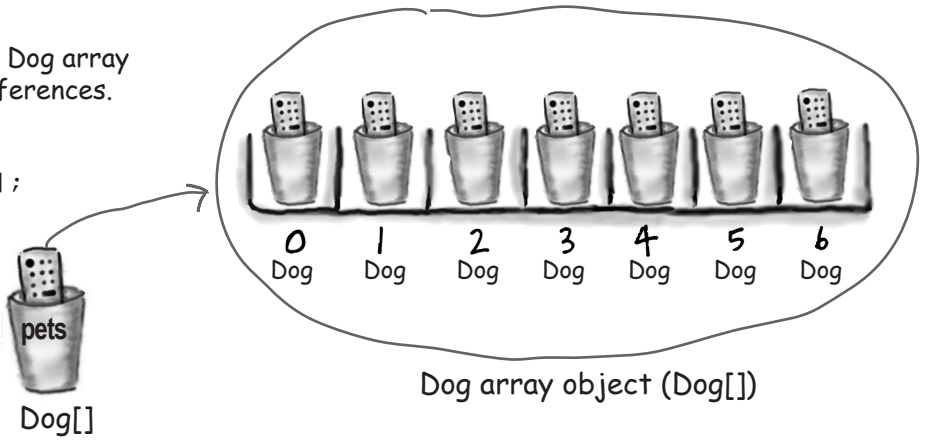
```
int x = 3 + one.getSize();
```

How do objects in an array behave?

Just like any other object. The only difference is how you *get* to them. In other words, how you get the remote control. Let's try calling methods on Dog objects in an array.

1 Declare and create a Dog array to hold seven Dog references.

```
Dog[] pets;
pets = new Dog[7];
```

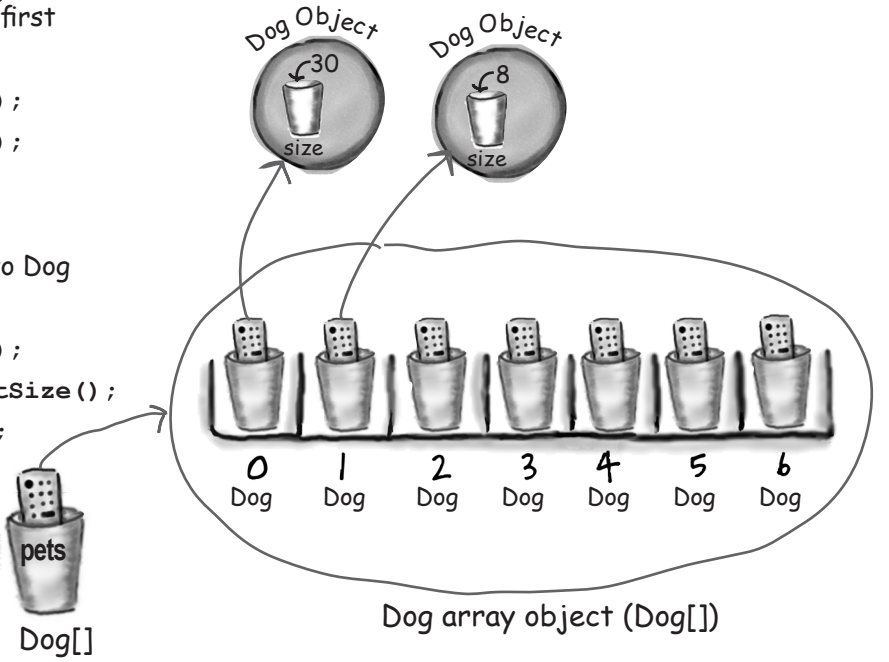


2 Create two new Dog objects, and assign them to the first two array elements.

```
pets[0] = new Dog();
pets[1] = new Dog();
```

3 Call methods on the two Dog objects.

```
pets[0].setSize(30);
int x = pets[0].getSize();
pets[1].setSize(8);
```



Declaring and initializing instance variables

You already know that a variable declaration needs at least a name and a type:

```
int size;
String name;
```

And you know that you can initialize (assign a value to) the variable at the same time:

```
int size = 420;
String name = "Donny";
```

But when you don't initialize an instance variable, what happens when you call a getter method? In other words, what is the *value* of an instance variable *before* you initialize it?

Instance variables always get a default value. If you don't explicitly assign a value to an instance variable or you don't call a setter method, the instance variable still has a value!

integers	0
floating points	0.0
booleans	false
references	null

```
class PoorDog {
  private int size;
  private String name;

  public int getSize() {
    return size;
  }

  public String getName() {
    return name;
  }
}
```

Declare two instance variables, but don't assign a value

What will these return??

```
public class PoorDogTestDrive {
  public static void main(String[] args) {
    PoorDog one = new PoorDog();
    System.out.println("Dog size is " + one.getSize());
    System.out.println("Dog name is " + one.getName());
  }
}
```

What do you think? Will this even compile?

```
File Edit Window Help CallVet
% java PoorDogTestDrive
Dog size is 0
Dog name is null
```

You don't have to initialize instance variables, because they always have a default value. Number primitives (including char) get 0, booleans get false, and object reference variables get null.

(Remember, null just means a remote control that isn't controlling / programmed to anything. A reference, but no actual object.)

The difference between instance and local variables

1 **Instance** variables are declared inside a class but not within a method.

```
class Horse {
  private double height = 15.2;
  private String breed;
  // more code...
}
```

2 **Local** variables are declared within a method.

```
class AddThing {
  int a;                    } INSTANCE variables
  int b = 12;

  public int add() {        <-- LOCAL variable
    int total = a + b;
    return total;
  }
}
```

3 **Local** variables MUST be initialized before use!

```
class Foo {
  public void go() {
    int x;
    int z = x + 3;
  }
}
```

Won't compile!! You can declare x without a value, but as soon as you try to USE it, the compiler freaks out.

```
File Edit Window Help Yikes
% javac Foo.java
Foo.java:4: variable x might
not have been initialized
      int z = x + 3;
1 error       ^
```

Local variables do NOT get a default value! The compiler complains if you try to use a local variable before the variable is initialized.

there are no Dumb Questions

Q: **What about method parameters? How do the rules about local variables apply to them?**

A: Method parameters are virtually the same as local variables—they're declared *inside* the method (well, technically they're declared in the *argument list* of the method rather than within the *body* of the method, but they're still local variables as opposed to instance variables). But method parameters will never be uninitialized, so you'll never get a compiler error telling you that a parameter variable might not have been initialized.

Instead, the compiler will give you an error if you try to invoke a method without giving the arguments that the method needs. So parameters are *always* initialized, because the compiler guarantees that methods are always called with arguments that match the parameters. The arguments are assigned (automatically) to the parameters.

Comparing variables (primitives or references)

Sometimes you want to know if two *primitives* are the same; for example, you might want to check an int result with some expected integer value. That's easy enough: just use the == operator. Sometimes you want to know if two reference variables refer to a single object on the heap; for example, is this Dog object exactly the same Dog object I started with? Easy as well: just use the == operator. But sometimes you want to know if two *objects* are equal. And for that, you need the .equals() method.

Use == to compare two primitives or to see if two references refer to the *same* object.

Use the equals() method to see if two *different* objects are equal.

(E.g., two different String objects that both contain the characters "Fred")

The idea of equality for objects depends on the type of object. For example, if two different String objects have the same characters (say, "my name"), they are meaningfully equivalent, regardless of whether they are two distinct objects on the heap. But what about a Dog? Do you want to treat two Dogs as being equal if they happen to have the same size and weight? Probably not. So whether two different objects should be treated as equal depends on what makes sense for that particular object type. We'll explore the notion of object equality again in later chapters, but for now, we need to understand that the == operator is used *only* to compare the bits in two variables. *What* those bits represent doesn't matter. The bits are either the same, or they're not.

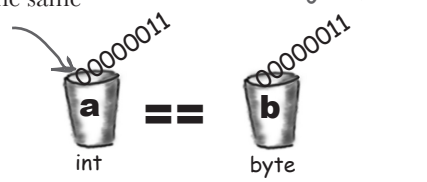
To compare two primitives, use the == operator

The == operator can be used to compare two variables of any kind, and it simply compares the bits.

if (a == b) {...} looks at the bits in a and b and returns true if the bit pattern is the same (although all the extra zeros on the left end don't matter).

```
int a = 3;
byte b = 3;
if (a == b) { // true }
```

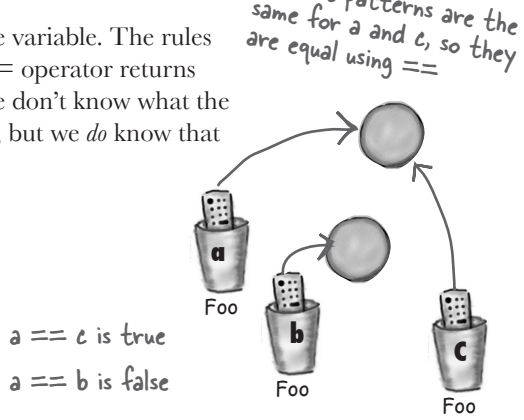
(There are more zeros on the left side of the int, but we don't care about that here)



To see if two references are the same (which means they refer to the same object on the heap) use the == operator

Remember, the == operator cares only about the pattern of bits in the variable. The rules are the same whether the variable is a reference or primitive. So the == operator returns true if two reference variables refer to the same object! In that case, we don't know what the bit pattern is (because it's dependent on the JVM and hidden from us), but we *do* know that whatever it looks like, *it will be the same for two references to a single object*.

```
Foo a = new Foo();
Foo b = new Foo();
Foo c = a;
if (a == b) { } // false
if (a == c) { } // true
if (b == c) { } // false
```



BULLET POINTS

- Encapsulation gives you control over who changes the data in your class and how.
- Make an instance variable *private* so it can't be changed by accessing the variable directly.
- Create a public mutator method, e.g., a setter, to control how other code interacts with your data. For example, you can add validation code inside a setter to make sure the value isn't changed to something invalid.
- *Instance* variables are assigned values by default, even if you don't set them explicitly.
- *Local* variables, e.g., variables inside methods, are not assigned a value by default. You always need to initialize them.
- Use == to check if two primitives are the same value.
- Use == to check if two references are the same, i.e., two object variables are actually the same object.
- Use .equals() to see if two objects are equivalent (but not necessarily the same object), e.g., to check if two String values contain the same characters.

Sharpen your pencil

What's legal?

Given the method below, which of the method calls listed on the right are legal?

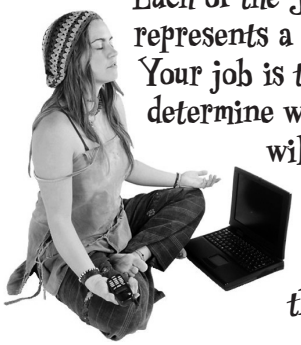
Put a checkmark next to the ones that are legal. (Some statements are there to assign values used in the method calls.)


```
int calcArea(int height, int width) {
   return height * width;
}
```

```
int a = calcArea(7, 12);
short c = 7;
calcArea(c, 15);

int d = calcArea(57);

calcArea(2, 3);

long t = 42;
int f = calcArea(t, 17);

int g = calcArea();

calcArea();

byte h = calcArea(4, 20);

int j = calcArea(2, 3, 5);
```

Answers on page 93.

BE the Compiler



Each of the Java files on this page represents a complete source file. Your job is to play compiler and determine whether each of these files will compile. If they won't compile, how would you fix them, and if they do compile, what would be their output?

A

```
class XCopy {

  public static void main(String[] args) {
    int orig = 42;
    XCopy x = new XCopy();
    int y = x.go(orig);
    System.out.println(orig + " " + y);
  }

  int go(int arg) {
    arg = arg * 2;
    return arg;
  }
}
```

B

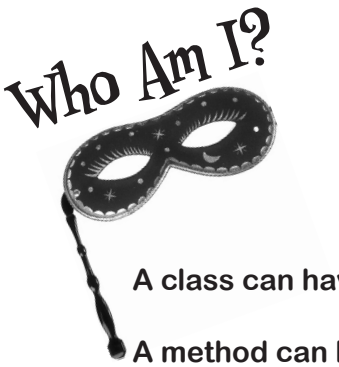
```
class Clock {
  String time;

  void setTime(String t) {
    time = t;
  }

  void getTime() {
    return time;
  }
}

class ClockTestDrive {
  public static void main(String[] args) {
    Clock c = new Clock();

    c.setTime("1245");
    String tod = c.getTime();
    System.out.println("time: "+tod);
  }
}
```

Answers on page 93.

A bunch of Java components, in full costume, are playing a party game, "Who am I?" They give you a clue, and you try to guess who they are, based on what they say. Assume they always tell the truth about themselves. If they happen to say something that could be true for more than one attendee, then write down all for whom that sentence applies. Fill in the blanks next to the sentence with the names of one or more attendees.

Tonight's attendees:

instance variable, argument, return, getter, setter, encapsulation, public, private, pass by value, method



A class can have any number of these. _____

A method can have only one of these. _____

This can be implicitly promoted. _____

I prefer my instance variables private. _____

It really means "make a copy." _____

Only setters should update these. _____

A method can have many of these. _____

I return something by definition. _____

I shouldn't be used with instance variables. _____

I can have many arguments. _____

By definition, I take one argument. _____

These help create encapsulation. _____

I always fly solo. _____

Answers on page 93.

A short Java program is listed to your right. Two blocks of the program are missing. Your challenge is to **match the candidate blocks of code** (below) **with the output** that you'd see if the blocks 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.

```
public class Mix4 {
  int counter = 0;

  public static void main(String[] args) {
    int count = 0;
    Mix4[] mixes = new Mix4[20];
    int i = 0;
    while ( [          ] ) {
      mixes[i] = new Mix4();
      mixes[i].counter = mixes[i].counter + 1;
      count = count + 1;
      count = count + mixes[i].maybeNew(i);
      i = i + 1;
    }
    System.out.println(count + " " +
                       mixes[1].counter);
  }

  public int maybeNew(int index) {
    if ( [          ] ) {
      Mix4 mix = new Mix4();
      mix.counter = mix.counter + 1;
      return 1;
    }
    return 0;
  }
}
```

Candidates:

```
i < 9

index < 5
```

```
i < 20

index < 5
```

```
i < 7

index < 7
```

```
i < 19

index < 1
```

Possible output:

```
14 7
```

```
9 5
```

```
19 1
```

```
14 1
```

```
25 1
```

```
7 7
```

```
20 1
```

```
20 5
```

Answers on page 94.

Pool Puzzle

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

→ Answers on page 94.

Output

```
File  Edit  Window  Help  BellyFlop
%java Puzzle4
result 543345
```

```
public class Puzzle4 {
  public static void main(String [] args) {

    _____________________________________
    int number = 1;
    int i = 0;
    while (i < 6) {

      _____________________________
      _____________________________
      number = number * 10;

      ___________________
    }

    int result = 0;
    i = 6;
    while (i > 0) {

      ___________________
      result = result + ____________________
    }
    System.out.println("result " + result);
  }
}

class ___________ {
  int intValue;
  _________  _______ doStuff(int __________) {
    if (intValue > 100) {
      return ______________________________
    } else {
      return ______________________________
    }
  }
}
```

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

doStuff(i);
values.doStuff(i);
values[i].doStuff(factor);
values[i].doStuff(i);
intValue = i;
values.intValue = i;
values[i].intValue = i;
values[i].intValue = number;
intValue
factor
public
private
intValue + factor;
intValue * (2 + factor);
intValue * (5 - factor);
intValue * factor;
Puzzle4
Value
Value( )
int
short
Puzzle4 [ ] values = new Puzzle4[6];
Value [ ] values = new Value[6];
Value [ ] values = new Puzzle4[6];
i = i + 1;
i = i - 1;
values [i] = new Value(i);
values [ ] = new Value( );
values [i] = new Value( );
values = new Value( );

Fast Times in Stim-City

When Buchanan roughly grabbed Jai's arm from behind, Jai froze. Jai knew that Buchanan was as stupid as he was ugly and he didn't want to spook the big guy. Buchanan ordered Jai into his boss's office, but Jai'd done nothing wrong (lately), so he figured a little chat with Buchanan's boss Leveler couldn't be too bad. He'd been moving lots of neural-stimmers in the west side lately, and he figured Leveler would be pleased. Black market stimmers weren't the best money pump around, but they were pretty harmless. Most of the stim-junkies he'd seen tapped out after a while and got back to life, maybe just a little less focused than before.

Five-Minute Mystery

Leveler's "office" was a skungy-looking skimmer, but once Buchanan shoved him in, Jai could see that it'd been modified to provide all the extra speed and armor that a local boss like Leveler could hope for. "Jai my boy," hissed Leveler, "pleasure to see you again." "Likewise I'm sure...," said Jai, sensing the malice behind Leveler's greeting, "We should be square Leveler, have I missed something?" "Ha! You're making it look pretty good, Jai. Your volume is up, but I've been experiencing, shall we say, a little 'breach' lately," said Leveler.

Jai winced involuntarily; he'd been a top drawer jack-hacker in his day. Anytime someone figured out how to break a street-jack's security, unwanted attention turned toward Jai. "No way it's me man," said Jai, "not worth the downside. I'm retired from hacking, I just move my stuff and mind my own business." "Yeah, yeah," laughed Leveler, "I'm sure you're clean on this one, but I'll be losing big margins until this new jack-hacker is shut out!" "Well, best of luck, Leveler. Maybe you could just drop me here and I'll go move a few more 'units' for you before I wrap up today," said Jai.

"I'm afraid it's not that easy, Jai. Buchanan here tells me that word is you're current on Java NE 37.3.2," insinuated Leveler. "Neural edition? Sure, I play around a bit, so what?" Jai responded, feeling a little queasy. "Neural edition's how I let the stim-junkies know where the next drop will be," explained Leveler. "Trouble is, some stim-junkie's stayed straight long enough to figure out how to hack into my Warehousing database." "I need a quick thinker like yourself, Jai, to take a look at my StimDrop Java NE class; methods, instance variables, the whole enchilada, and figure out how they're getting in. It should...," "HEY!" exclaimed Buchanan, "I don't want no scum hacker like Jai nosin' around my code!" "Easy big guy," Jai saw his chance, "I'm sure you did a top rate job with your access modi..." "Don't tell me, bit twiddler!" shouted Buchanan, "I left all of those junkie-level methods public so they could access the drop site data, but I marked all the critical WareHousing methods private. Nobody on the outside can access those methods, buddy, nobody!"

"I think I can spot your leak, Leveler. What say we drop Buchanan here off at the corner and take a cruise around the block?" suggested Jai. Buchanan clenched his fists and started toward Jai, but Leveler's stunner was already on Buchanan's neck, "Let it go, Buchanan," sneered Leveler, "Keep your hands where I can see them and step outside. I think Jai and I have some plans to make."

What did Jai suspect?

Will he get out of Leveler's skimmer with all his bones intact?

Answers on page 94.

Exercise Solutions

Sharpen your pencil (from page 87)

```
int a = calcArea(7, 12);
short c = 7;
calcArea(c, 15);

int d = calcArea(57);

calcArea(2, 3);

long t = 42;
int f = calcArea(t, 17);

int g = calcArea();

calcArea();

byte h = calcArea(4, 20);

int j = calcArea(2, 3, 5);
```

BE the Compiler (from page 88)

A

Class 'XCopy' compiles and runs as it stands! The output is: '42 84'. Remember, Java is pass by value, (which means pass by copy), and the variable 'orig' is not changed by the go( ) method.

B

```
class Clock {
  String time;

  void setTime(String t) {
    time = t;
  }

  String getTime() {
    return time;
  }
}

class ClockTestDrive {
  public static void main(String[] args) {
    Clock c = new Clock();
    c.setTime("1245");
    String tod = c.getTime();
    System.out.println("time: " + tod);
  }
}
```

Note: 'Getter' methods have a return type by definition.

Who Am I? (from page 89)

A class can have any number of these.	instance variables, getter, setter, method
A method can have only one of these.	return
This can be implicitly promoted.	return, argument
I prefer my instance variables private.	encapsulation
It really means "make a copy."	pass by value
Only setters should update these.	instance variables
A method can have many of these.	argument
I return something by definition.	getter
I shouldn't be used with instance variables	public
I can have many arguments.	method
By definition, I take one argument.	setter
These help create encapsulation.	getter, setter, public, private
I always fly solo.	return

Puzzle Solutions

Pool Puzzle (from page 91)

```
public class Puzzle4 {
  public static void main(String[] args) {
    Value[] values = new Value[6];
    int number = 1;
    int i = 0;
    while (i < 6) {
      values[i] = new Value();
      values[i].intValue = number;
      number = number * 10;
      i = i + 1;
    }

    int result = 0;
    i = 6;
    while (i > 0) {
      i = i - 1;
      result = result + values[i].doStuff(i);
    }
    System.out.println("result " + result);
  }
}

class Value {
  int intValue;

  public int doStuff(int factor) {
    if (intValue > 100) {
      return intValue * factor;
    } else {
      return intValue * (5 - factor);
    }
  }
}
```

Output

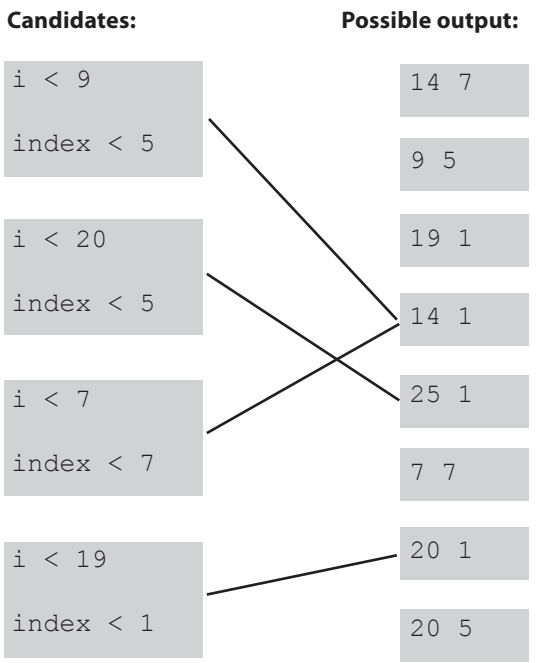
```
File  Edit  Window  Help  BellyFlop
%java Puzzle4
result 543345
```

Five-Minute Mystery (from page 92)

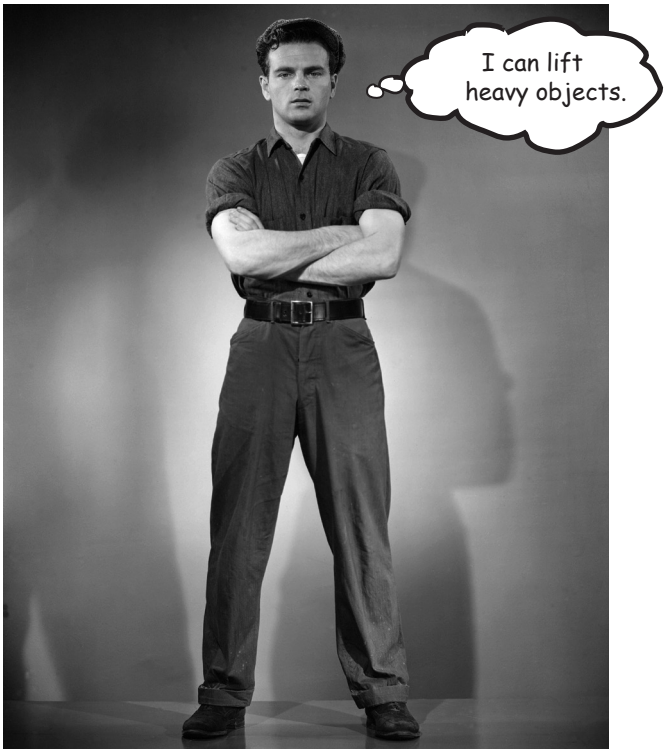
What did Jai suspect?

Jai knew that Buchanan wasn't the sharpest pencil in the box. When Jai heard Buchanan talk about his code, Buchanan never mentioned his instance variables. Jai suspected that while Buchanan did in fact handle his methods correctly, he failed to mark his instance variables `private`. That slip-up could have easily cost Leveler thousands.

Mixed Messages (from page 90)



Extra-Strength Methods



Let's put some muscle in our methods. We dabbled with variables, played with a few objects, and wrote a little code. But we were weak. We need more tools. Like **operators**. We need more operators so we can do something a little more interesting than, say, *bark*. And **loops**. We need loops, but what's with the wimpy *while* loops? We need ***for*** loops if we're really serious. Might be useful to **generate random numbers**. Better learn that too. And why don't we learn it all by *building* something real, to see what it's like to write (and test) a program from scratch. **Maybe a game**, like Battleships. That's a heavy-lifting task, so it'll take *two* chapters to finish. We'll build a simple version in this chapter and then build a more powerful deluxe version in Chapter 6, *Using the Java Library*.

Let's build a Battleship-style game: "Sink a Startup"

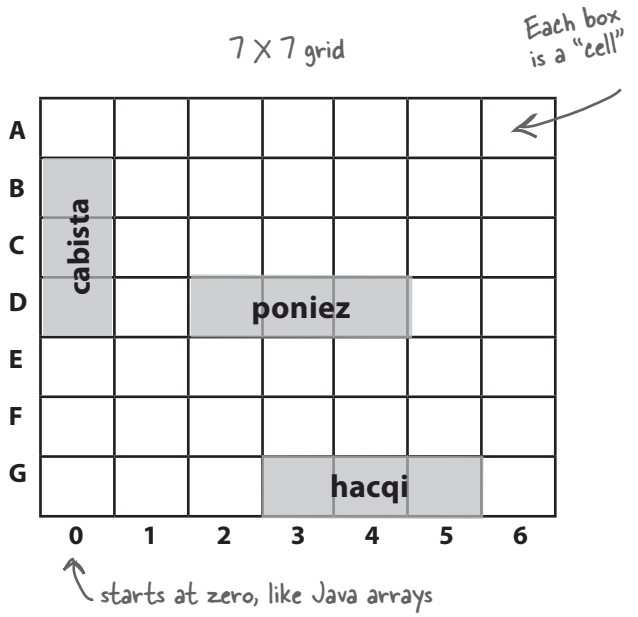
It's you against the computer, but unlike the real Battleship game, in this one you don't place any ships of your own. Instead, your job is to sink the computer's ships in the fewest number of guesses.

Oh, and we aren't sinking ships. We're killing ill-advised, Silicon Valley Startups (thus establishing business relevancy so you can expense the cost of this book).

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

Setup: When the game program is launched, the computer places three Startups on a **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) that 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.
```

First, a high-level design

We know we'll need classes and methods, but what should they be? To answer that, we need more information about what the game should do.

First, we need to figure out the general flow of the game. Here's the basic idea:

1 User starts the game.

- **A** Game creates three Startups
- **B** Game places the three Startups onto a virtual grid

2 Game play begins.

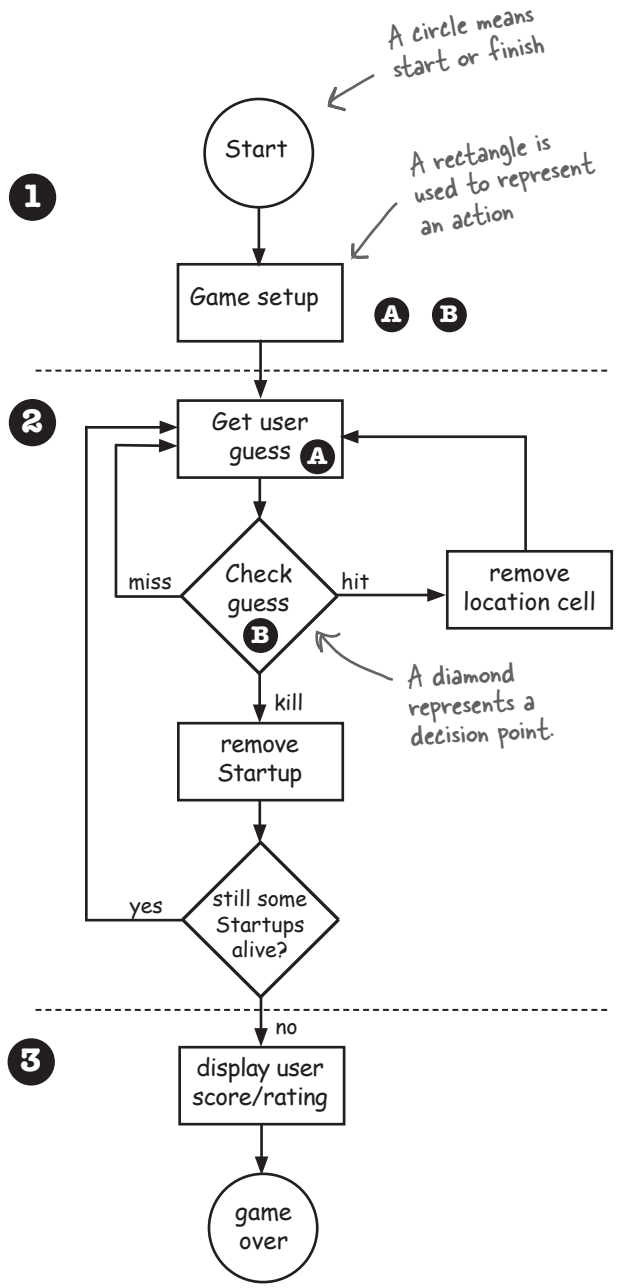
Repeat the following until there are no more Startups:

- **A** Prompt user for a guess ("A2," "C0," etc.)
- **B** Check the user guess against all Startups to look for a hit, miss, or kill. Take appropriate action: if a hit, delete cell (A2, D4, etc.). If a kill, delete Startup.

3 Game finishes.

Give the user a rating based on the number of guesses.

Now we have an idea of the kinds of things the program needs to do. The next step is figuring out what kind of **objects** we'll need to do the work. Remember, think like Brad rather than Laura (who we met in Chapter 2, *A Trip to Objectville*); focus first on the ***things*** in the program rather than the ***procedures***.



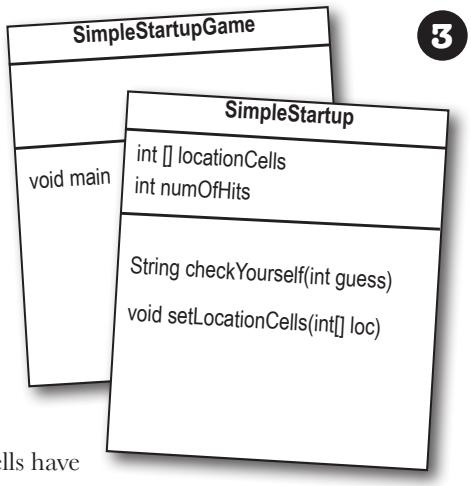
Whoa. A real flow chart.

The "Simple Startup Game"
a gentler introduction

It looks like we're gonna need at least two classes, a Game class and a Startup class. But before we build the full-monty ***Sink a Startup*** game, we'll start with a stripped-down, simplified version, ***Simple Startup Game.*** We'll build the simple version in *this* chapter, followed by the deluxe version that we build in the *next* chapter.

Everything is simpler in this game. Instead of a 2-D grid, we hide the Startup in just a single *row.* And instead of *three* Startups, we use *one*.

The goal is the same, though, so the game still needs to make a Startup instance, assign it a location somewhere in the row, get user input, and when all of the Startup's cells have been hit, the game is over. This simplified version of the game gives us a big head start on building the full game. If we can get this small one working, we can scale it up to the more complex one later.

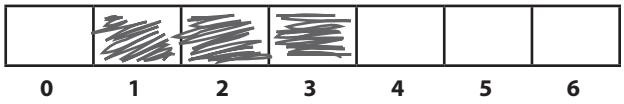


In this simple version, the game class has no instance variables, and all the game code is in the main() method. In other words, when the program is launched and main() begins to run, it will make the one and only Startup instance, pick a location for it (three consecutive cells on the single virtual seven-cell row), ask the user for a guess, check the guess, and repeat until all three cells have been hit.

Keep in mind that the virtual row is...*virtual*. In other words, it doesn't exist anywhere in the program. As long as both the game and the user know that the Startup is hidden in three consecutive cells out of a possible seven (starting at zero), the row itself doesn't have to be represented in code. You might be tempted to build an array of seven ints and then assign the Startup to three of the seven elements in the array, but you don't need to. All we need is an array that holds just the three cells the Startup occupies.

1. **Game starts** and creates ONE Startup and gives it a location on three cells in the single row of seven cells.

   Instead of "A2," "C4," and so on, the locations are just integers (for example: 1,2,3 are the cell locations in this picture):



2. **Game play begins**. Prompt user for a guess; then check to see if it hit any of the Startup's three cells. If a hit, increment the numOfHits variable.

3. **Game finishes** when all three cells have been hit (the numOfHits variable value is 3), and the user is told how many guesses it took to sink the Startup.

A complete game interaction

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

Developing a Class

As a programmer, you probably have a methodology/process/approach to writing code. Well, so do we. Our sequence is designed to help you see (and learn) what we're thinking as we work through coding a class. It isn't necessarily the way we (or *you)* write code in the Real World. In the Real World, of course, you'll follow the approach your personal preferences, project, or employer dictate. We, however, can do pretty much whatever we want. And when we create a Java class as a "learning experience," we usually do it like this:

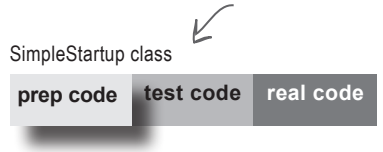
- ☐ Figure out what the class is supposed to *do*.
- ☐ List the **instance variables and methods**.
- ☐ Write **prep code** for the methods. (You'll see this in just a moment.)
- ☐ Write **test code** for the methods.
- ☐ **Implement** the class.
- ☐ **Test** the methods.
- ☐ **Debug** and **reimplement** as needed.
- ☐ Express gratitude that we don't have to test our so-called *learning experience* app on actual live users.

Flex those dendrites.

How would you decide which class or classes to build *first*, when you're writing a program? Assuming that all but the tiniest programs need more than one class (if you're following good OO principles and not having *one* class do many different jobs), where do you start?

The three things we'll write for each class:

This bar is displayed on the next set of pages to tell you which part you're working on. For example, if you see this picture at the top of a page, it means you're working on prep code for the SimpleStartup class.



prep code

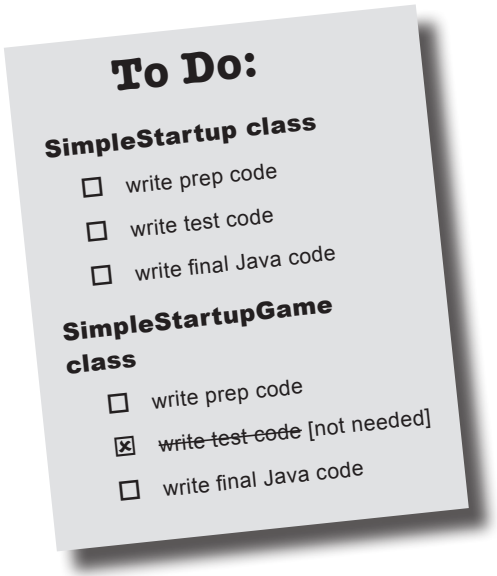
A form of pseudocode, to help you focus on the logic without stressing about syntax.

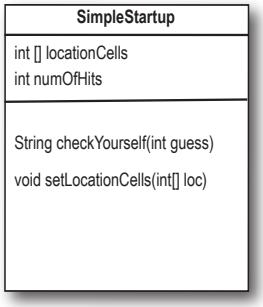
test code

A class or methods that will test the real code and validate that it's doing the right thing.

real code

The actual implementation of the class. This is where we write real Java code.





You'll get the idea of how prep code (our version of pseudocode) works as you read through this example. It's sort of halfway between real Java code and a plain English description of the class. Most prep code includes three parts: instance variable declarations, method declarations, method logic. The most important part of prep code is the method logic, because it defines *what* has to happen, which we later translate into *how* when we actually write the method code.

DECLARE an ***int array*** to hold the location cells. Call it ***locationCells***.

DECLARE an ***int*** to hold the number of hits. Call it ***numOfHits*** and **SET** it to 0.

DECLARE a ***checkYourself()*** method that takes a ***int*** for the user's guess (1, 3, etc.), checks it, and returns a result representing a "hit," "miss," or "kill."

DECLARE a ***setLocationCells()*** setter method that takes an ***int array*** (which has the three cell locations as ***ints*** (2, 3, 4, etc.)).

METHOD: *String checkYourself(int userGuess)*

```
    GET the user guess as an int parameter
    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" as the result
            ELSE  it was not a kill, so RETURN "hit"
            END IF
        ELSE the user guess did not match, so RETURN "miss"
        END IF
    END REPEAT
END METHOD
```

METHOD: *void setLocationCells(int[] cellLocations)*

```
    GET the cell locations as an int array parameter
    ASSIGN the cell locations parameter to the cell locations instance variable
END METHOD
```

Writing the method implementations

Let's write the real method code now and get this puppy working.

Before we start coding the methods, though, let's back up and write some code to *test* the methods. That's right, we're writing the test code *before* there's anything to test!

The concept of writing the test code first is one of the practices of Test-Driven Development (TDD), and it can make it easier (and faster) for you to write your code. We're not necessarily saying you should use TDD, but we do like the part about writing tests first. And TDD just *sounds* cool.

Test-Driven Development (TDD)

Back in 1999, Extreme Programming (XP) was a newcomer to the software development methodology world. One of the central ideas in XP was to write test code before writing the actual code. Since then, the idea of writing test code first has spun off of XP and become the core of a newer, more popular subset of XP called TDD. (Yes, yes, we know we've just grossly oversimplified this, please cut us a little slack here.)

TDD is a LARGE topic, and we're only going to scratch the surface in this book. But we hope that the way we're going about developing the "Sink a Startup" game gives you some sense of TDD.

Check out *Test Driven Development: By Example* by Kent Beck if you want to learn more about how TDD works.

Here is a partial list of key ideas in TDD:

- Write the test code first.
- Develop in iteration cycles.
- Keep it (the code) simple.
- Refactor (improve the code) whenever and wherever you notice the opportunity.
- Don't release anything until it passes all the tests.
- Don't put in anything that's not in the spec (no matter how tempted you are to put in functionality "for the future").
- No killer schedules; work regular hours.

Writing test code for the SimpleStartup class

We need to write test code that can make a SimpleStartup object and run its methods. For the SimpleStartup class, we really care about only the *checkYourself()* method, although we *will* have to implement the *setLocationCells()* method in order to get the *checkYourself()* method to run correctly.

Take a good look at the prep code below for the *checkYourself()* method (the *setLocationCells()* method is a no-brainer setter method, so we're not worried about it, but in a "real" application we might want a more robust "setter" method, which we *would* want to test).

Then ask yourself, "If the checkYourself() method were implemented, what test code could I write that would prove to me the method is working correctly?"

Based on this prep code:

```
METHOD String checkYourself(int userGuess)
  GET the user guess as an int parameter
  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" as the result
        ELSE  it was not a kill, so RETURN "Hit"
        END IF
    ELSE the user guess did not match, so RETURN "Miss"
    END IF
  END REPEAT
END METHOD
```

Here's what we should test:

1. Instantiate a SimpleStartup object.
2. Assign it a location (an array of 3 ints, like {2, 3, 4}).
3. Create an int to represent a user guess (2, 0, etc.).
4. Invoke the checkYourself() method passing it the fake user guess.
5. Print out the result to see if it's correct ("passed" or "failed").

prep code | test code | real code

there are no Dumb Questions

Q: Maybe I'm missing something here, but how exactly do you run a test on something that doesn't yet exist!?

A: You don't. We never said you start by *running* the test; you start by *writing* the test. At the time you write the test code, you won't have anything to run it against, so you probably won't be able to compile it until you write "stub" code that can compile, but that will always cause the test to fail (like, return null).

Q: Then I still don't see the point. Why not wait until the code is written, and then whip out the test code?

A: The act of thinking through (and writing) the test code helps clarify your thoughts about what the method itself needs to do.

As soon as your implementation code is done, you already have test code just waiting to validate it. Besides, you *know* if you don't do it now, you'll *never* do it. There's always something more interesting to do.

Ideally, write a little test code, then write *only* the implementation code you need in order to pass that test. Then write a little *more* test code and write *only* the new implementation code needed to pass *that* new test. At each test iteration, you run *all* the previously written tests to prove that your latest code additions don't break previously tested code.

Test code for the SimpleStartup class

```
public class SimpleStartupTestDrive {

  public static void main(String[] args) {

    SimpleStartup dot = new SimpleStartup();

    int[] locations = {2, 3, 4};

    dot.setLocationCells(locations);

    int userGuess = 2;

    String result = dot.checkYourself(userGuess);

    String testResult = "failed";

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

      testResult = "passed";

    }

    System.out.println(testResult);

  }

}
```

Instantiate a SimpleStartup object.

Make an int array for the location of the Startup (3 consecutive ints out of a possible 7).

Invoke the setter method on the Startup.

Make a fake user guess.

Invoke the checkYourself() method on the Startup object, and pass it the fake guess.

If the fake guess (2) gives back a "hit", it's working.

Print out the test result ("passed" or "failed").

Sharpen your pencil → Yours to solve.

In the next couple of pages we implement the SimpleStartup class, and then later we return to the test class. Looking at our test code above, what else should be added? What are we *not* testing in this code that we *should* be testing for? Write your ideas (or lines of code) below:

The checkYourself() method

There isn't a perfect mapping from prep code to Java code; you'll see a few adjustments. The prep code gave us a much better idea of *what* the code needs to do, and now we have to figure out the Java code that can do the *how*.

In the back of your mind, be thinking about parts of this code you might want (or need) to improve. The numbers (1) are for things (syntax and language features) you haven't seen yet. They're explained on the opposite page.

GET the user guess

REPEAT with each cell in the *int* array

IF the user guess matches

INCREMENT the number of hits

// FIND OUT if it was the last cell

IF number of hits is 3,

RETURN "kill" as the result

ELSE it was not a kill, so **RETURN** "hit"

ELSE

RETURN "miss"

```
public String checkYourself(int guess) {
  String result = "miss";
  (1)
  for (int cell : locationCells) {
    if (guess == cell) {
      result = "hit";
      (2)
      numOfHits++;
      (3)
      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 cell in the locationCells array (each cell location of the object)

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

prep code | test code | real code

Just the new stuff

The things we haven't seen before are on this page. Stop worrying! There are more details later in the chapter. This is just enough to get you going.

1 The **for** loop

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

Read this for loop declaration as "repeat for each element in the 'locationCells' array: take the next element in the array and assign it to the int variable 'cell'."

The colon (:) means "in", so the whole thing means "for each int value IN locationCells..."

Declare a variable that will hold one element from the array. Each time through the loop, this variable (in this case an int variable named "cell") will hold a different element from the array, until there are no more elements (or the code does a "break"... see #4 below).

The array to iterate over in the loop. Each time through the loop, the next element in the array will be assigned to the variable "cell". (More on this at the end of this chapter.)

2 The **post-increment operator**

```
numOfHits++
```

The ++ means add 1 to whatever's there (in other words, increment by 1).

numOfHits++ is the same (in this case) as saying numOfHits = numOfHits + 1, with less typing.

3 **break** statement

```
break;
```

Gets you out of a loop. Immediately. Right here. No iteration, no boolean test, just get out now!

there are no Dumb Questions

Q: In the beginning of the book, there was an example of a *for* loop that was really different from this one—are there two different styles of *for* loops?

A: Yes! From the first version of Java there has been a single kind of *for* loop (explained later in this chapter) that looks like this:

```
for (int i = 0; i < 10; i++)
{
    // do something 10 times
}
```

You can use this format for any kind of loop you need. But... since Java 5, you can also use the *enhanced* for loop (that's the official description) when your loop needs to iterate over the elements in an array (or *another* kind of collection, as you'll see in the *next* chapter). You can always use the plain old for loop to iterate over an array, but the *enhanced* for loop makes it easier.

Q: If you can add one to an int by using ++, can you also subtract one in some way?

A: Yep absolutely. Hopefully it's not too surprising to find out that the syntax is -- (two minuses), like this:

```
countdown = i--;
```

Final code for SimpleStartup and SimpleStartupTestDrive

```
public class SimpleStartupTestDrive {
  public static void main(String[] args) {
    SimpleStartup dot = new SimpleStartup();
    int[] locations = {2, 3, 4};
    dot.setLocationCells(locations);
    int userGuess = 2;
    String result = dot.checkYourself(userGuess);
    String testResult = "failed";
    if (result.equals("hit")) {
      testResult = "passed";
    }
    System.out.println(testResult);
  }
}
```

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

There's a little bug lurking here. It compiles and runs, but...don't worry about it for now, but we *will* have to face it a little later.

What should we see when we run this code?

The test code makes a SimpleStartup object and gives it a location at 2,3,4. Then it sends a fake user guess of "2" into the checkYouself() method. If the code is working correctly, we should see the result print out:

```
% java SimpleStartupTestDrive
hit
passed
```

prep code | test code | real code

Sharpen your pencil

We built the test class and the SimpleStartup class. But we still haven't made the actual *game*. Given the code on the opposite page and the spec for the actual game, write in your ideas for prep code for the game class. We've given you a few lines here and there to get you started. The actual game code is on the next page, so ***don't turn the page until you do this exercise!***

You should have somewhere between 12 and 18 lines (including the ones we wrote, but *not* including lines that have only a curly brace).

METHOD ***public static void main (String [] args)***

DECLARE an int variable to hold the number of user guesses, named *numOfGuesses*

COMPUTE a random number between 0 and 4 that will be the starting location cell position

WHILE the Startup is still alive:

GET user input from the command line

The SimpleStartupGame needs to do this:

1. Make the single SimpleStartup object.
2. Make a location for it (three consecutive cells on a single row of seven virtual cells).
3. Ask the user for a guess.
4. Check the guess.
5. Repeat until the Startup is sunk.
6. Tell the user how many guesses it took.

A complete game interaction

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

→ Yours to solve.

Prep code for the SimpleStartupGame class

Everything happens in main()

There are some things you'll have to take on faith. For example, we have one line of prep code that says "GET user input from command line." Let me tell you, that's a little more than we want to implement from scratch right now. But happily, we're using OO. And that means you get to ask some *other* class/object to do something for you, without worrying about ***how*** it does it. When you write prep code, you should assume that *somehow* you'll be able to do whatever you need to do, so you can put all your brainpower into working out the logic.

public static void main (String [] args)

DECLARE an int variable to hold the number of user guesses, named *numOfGuesses,* and set it to 0

MAKE a new SimpleStartup instance

COMPUTE a random number between 0 and 4 that will be the starting location cell position

MAKE an int array with 3 ints using the randomly generated number, that number incremented by 1, and that number incremented by 2 (example: 3,4,5)

INVOKE the *setLocationCells()* method on the SimpleStartup instance

DECLARE a boolean variable representing the state of the game, named *isAlive.* **SET** it to true

WHILE the Startup is still alive (isAlive == true):

GET user input from the command line

// CHECK the user guess

INVOKE the *checkYourself()* method on the SimpleStartup instance

INCREMENT *numOfGuesses* variable

// CHECK for Startup death

IF result is "kill"

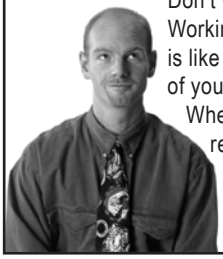
SET *isAlive* to false (which means we won't enter the loop again)

PRINT the number of user guesses

END IF

END WHILE

END METHOD



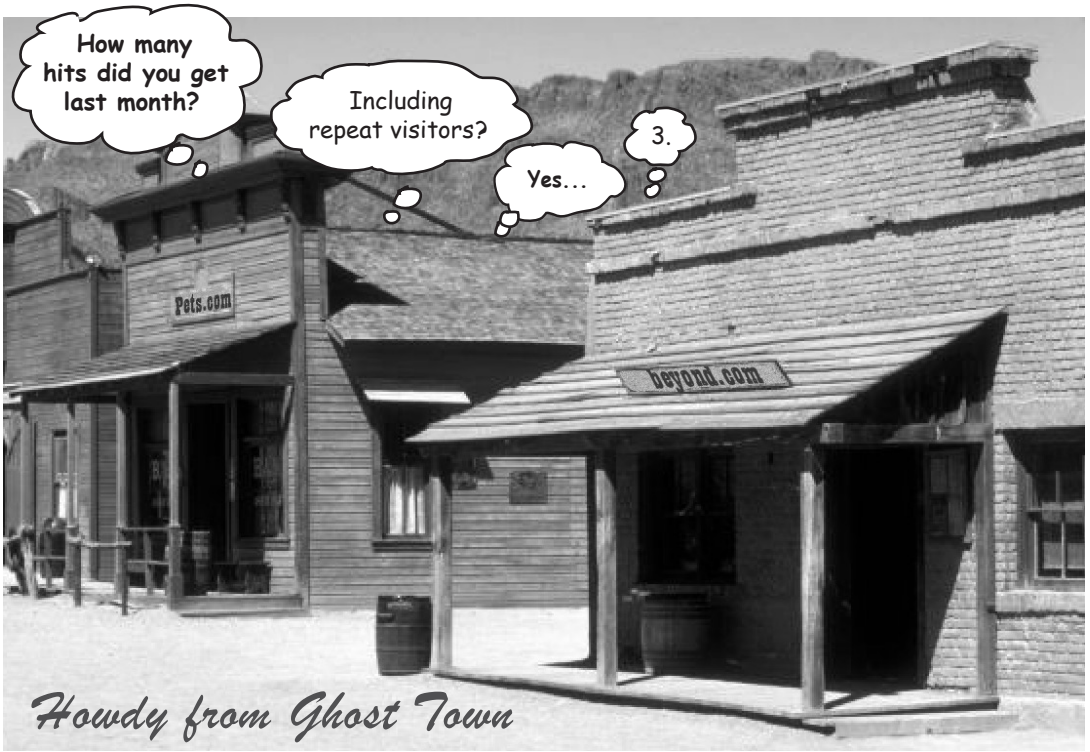
metacognitive tip

Don't work one part of the brain for too long a stretch at one time. Working just the left side of the brain for more than 30 minutes is like working just your left *arm* for 30 minutes. Give each side of your brain a break by switching sides at regular intervals.

When you shift to one side, the other side gets to rest and recover. Left-brain activities include things like step-by-step sequences, logical problem-solving, and analysis, while the right-brain kicks in for metaphors, creative problem-solving, pattern-matching, and visualizing.

BULLET POINTS

- Your Java program should start with a high-level design.
- Typically you'll write three things when you create a new class:
  - ***prep code***
  - ***test code***
  - ***real (Java) code***
- Prep code should describe *what* to do, not *how* to do it. Implementation comes later.
- Use the prep code to help design the test code.
- A class can have one superclass only.
- Write test code *before* you implement the methods.
- Choose *for* loops over *while* loops when you know how many times you want to repeat the loop code.
- The enhanced for loop is an easy way to loop over an array or collection.
- Use the *increment* operator to add 1 to a variable (x++;).
- Use the *decrement* operator to subtract 1 from a variable (x--;).
- Use *break* to leave a loop early (i.e., even if the boolean test condition is still true).



prep code | test code | real code

The game's main() method

Just as you did with the SimpleStartup class, be thinking about parts of this code you might want (or need) to improve. The numbered things (1) are for stuff we want to point out. They're explained on the opposite page. Oh, if you're wondering why we skipped the test code phase for this class, we don't need a test class for the game. It has only one method, so what would you do in your test code? Make a separate class that would call main() on this class? We didn't bother, we'll just run this to test it.

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

  int numOfGuesses = 0;

  GameHelper helper = new GameHelper();


  SimpleStartup theStartup = new SimpleStartup();

  int randomNum = (int) (Math.random() * 5);


  int[] locations = {randomNum, randomNum + 1, randomNum + 2};

  theStartup.setLocationCells(locations);

  boolean isAlive = true;


  while (isAlive) {

    int guess = helper.getUserInput("enter a number");

    String result = theStartup.checkYourself(guess);

    numOfGuesses++;

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

      isAlive = false;

      System.out.println("You took " + numOfGuesses + " guesses");

    } // close if

  } // close while
```

DECLARE a variable to hold user guess count, and set it to 0

Make a variable to track how many guesses the user makes.

This is a special class we wrote that has the method for getting user input. For now, pretend it's part of Java.

MAKE a SimpleStartup object

Make the Startup object.

COMPUTE a random number between 0 and 4

(1)

Make a random number for the first cell, and use it to make the cell locations array.

MAKE an int array with the 3 cell locations, and

INVOKE setLocationCells on the Startup object

Give the Startup its locations (the array).

DECLARE a boolean isAlive

Make a boolean variable to track whether the game is still alive, to use in the while loop test.

WHILE the Startup is still alive

repeat while game is still alive.

GET user input

(2)

Get user guess.

// CHECK it

INVOKE checkYourself() on Startup

Ask the Startup to check the guess; save the returned result.

INCREMENT numOfGuesses

Increment guess count by one.

IF result is "kill"

Was it a "kill"? if so, set isAlive to false (so we won't re-enter the loop) and print user guess count.

SET isAlive to false

PRINT the number of user guesses

prep code | test code | real code

random() and getUserInput()

Two things that need a bit more explaining are on this page. This is just a quick look to keep you going; more details on the GameHelper class are at the end of this chapter.

1 Make a random number

This is a 'cast', and it forces the thing immediately after it to become the type of the cast (i.e., the type in the brackets). Math.random returns a double, so we have to cast it to an int (we want a nice whole number between 0 and 4). In this case, the cast chops off the fractional part of the double.

The Math.random method returns a number from zero to just less than one. So this formula (with the cast) returns a number from 0 to 4 (i.e., 0 – 4.999.., cast to an int).

```
int randomNum = (int) (Math.random() * 5)
```

We declare an int variable to hold the random number we get back.

A class that comes with Java.

A static method of the Math class.

Math.random() has been around forever, so you'll see code like this in the Real World. These days you can use java.util.Random's nextInt() method instead, which is more convenient (you don't have to cast the result to an int).

The Random class is in a different package. Since we haven't covered importing packages yet (it's in the next chapter), we've used Math.random() instead.

2 Getting user input using the GameHelper class

An instance we made earlier of a class that we built to help with the game. It's called GameHelper and you haven't seen it yet (you will).

This method takes a String argument that it uses to prompt the user at the command line. Whatever you pass in here gets displayed in the terminal just before the method starts looking for user input.

```
int guess = helper.getUserInput("enter a number");
```

We declare an int variable to hold the user input we get back (3, 5, etc.).

A method of the GameHelper class that asks the user for command-line input, reads it in after the user hits RETURN, and gives back the result as an int.

One last class: GameHelper

We made the *Startup* class.

We made the *game* class.

All that's left is the *helper* class—the one with the getUserInput() method. The code to get command-line input is more than we want to explain right now. It opens up topics best left for later. (Later, as in Chapter 16, *Saving Objects*.)

Whenever you see this logo, you're seeing code that you have to type as-is and take on faith. Trust it. You'll learn how that code works later.

Ready-Bake Code

Just copy* the code below and compile it into a class named GameHelper. Drop all three class files (SimpleStartup, SimpleStartupGame, GameHelper) into the same directory, and make it your working directory.


```
import java.util.Scanner;

public class GameHelper {
  public int getUserInput(String prompt) {
    System.out.print(prompt + ": ");
    Scanner scanner = new Scanner(System.in);
    return scanner.nextInt();
  }
}
```

*We know how much you enjoy typing, but for those rare moments when you'd rather do something else, we've made the Ready-Bake Code available on *https://oreil.ly/hfJava_3e_examples*.

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...

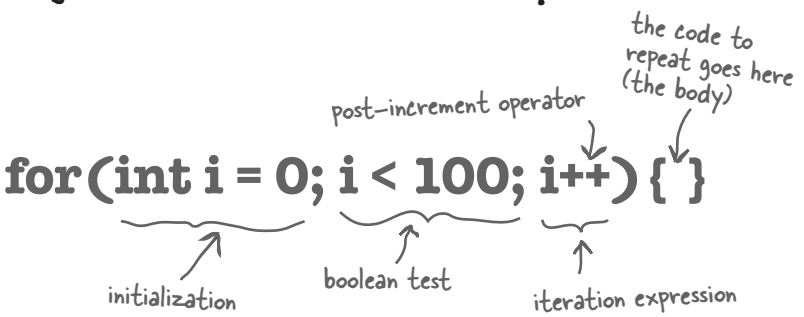
And in the meantime, see if you can come up with ideas for what went wrong and how to fix it.

→ 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.

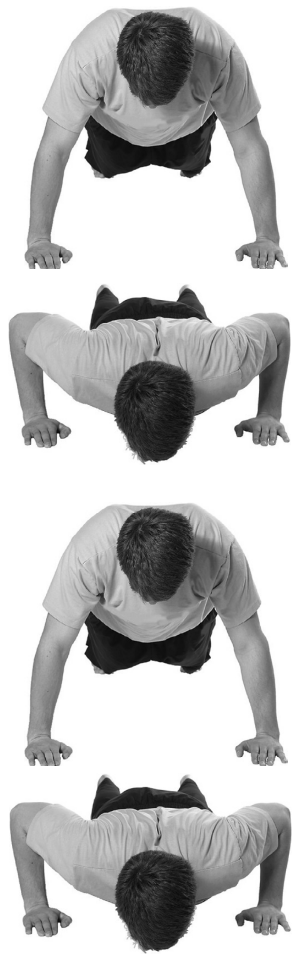
Part Two: *boolean test*

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*

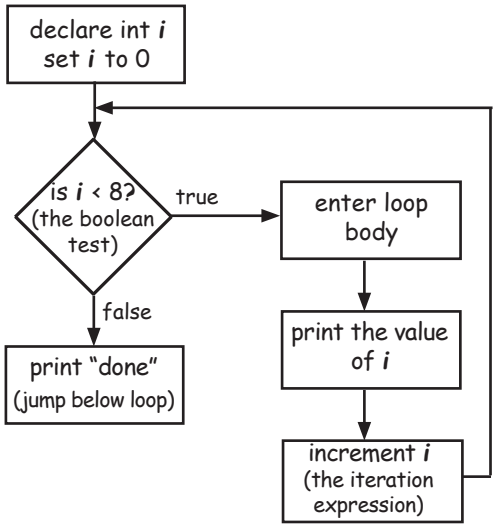
In this part, put one or more things you want to happen with each trip through the loop. Keep in mind that this stuff happens at the *end* of each loop.

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

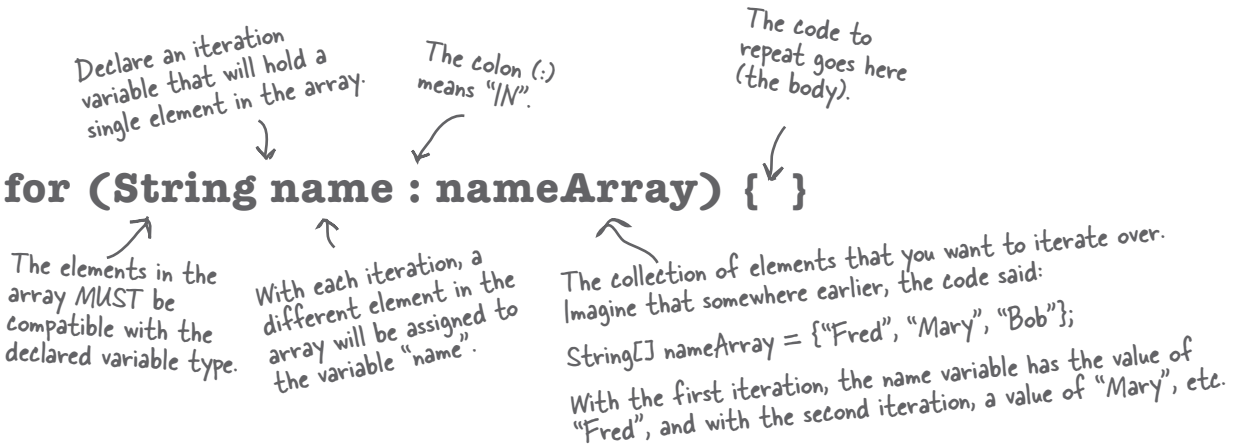
But putting the ++ *after* the x gives you a different result:

```
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*

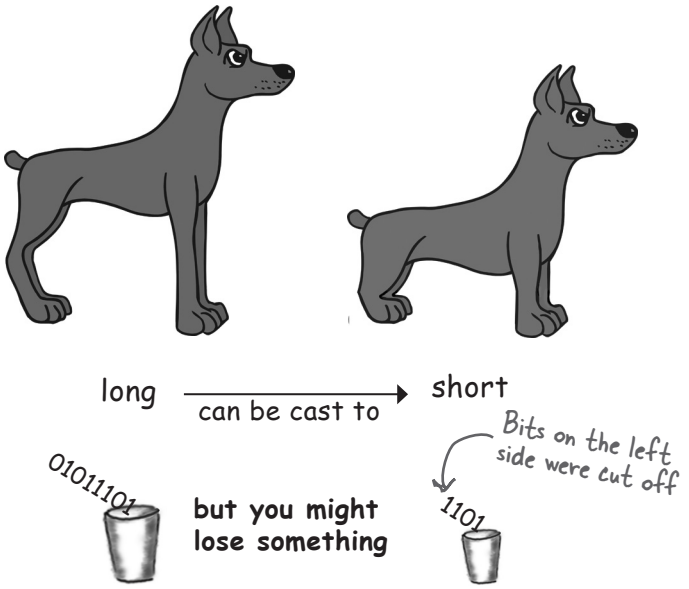
Use this part to declare and initialize a variable to use within the loop body. With each iteration of the loop, this variable will hold a different element from the collection. The type of this variable must be compatible with the elements in the array! For example, you can't declare an *int* iteration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

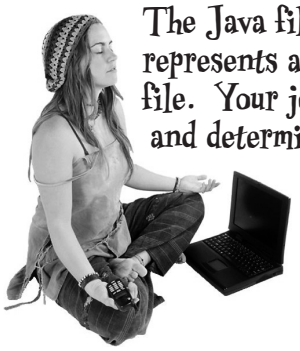
Still, the point is that the compiler lets you do it. And let's say you have a floating-point number and you just want to get at the whole number (int) part of it:

```
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
```

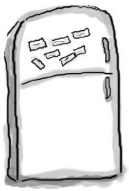
-or-

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 {
```

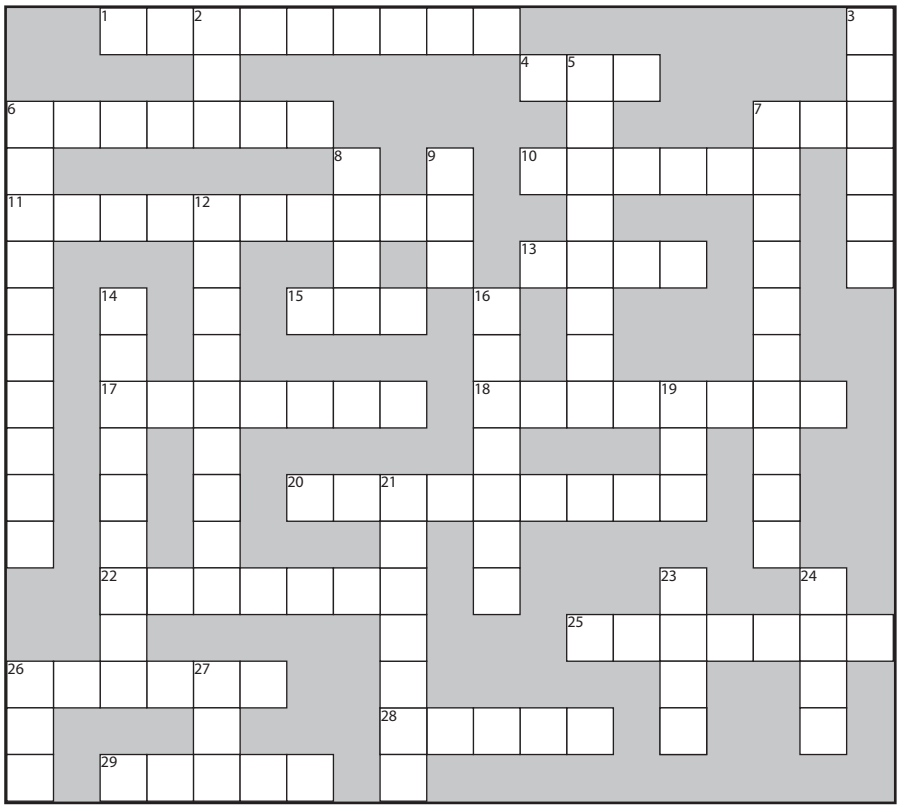
```
for(int  j = 4; j > 2; j--) {
```

```
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
```

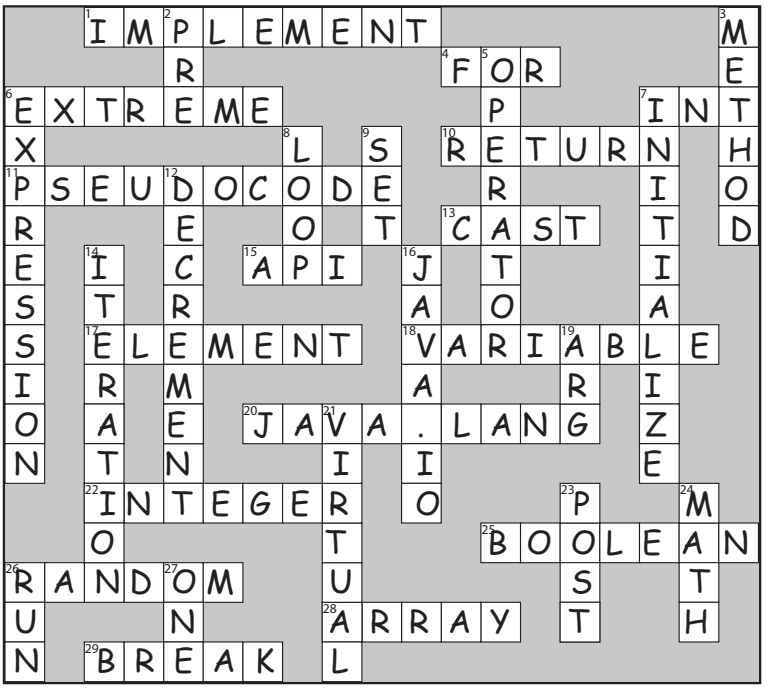
Code Magnets (from page 119)

```
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.

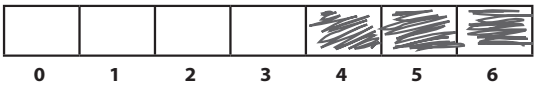
Here's where it goes wrong. We counted a hit every time the user guessed a cell location, *even if that location had already been hit!*

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?

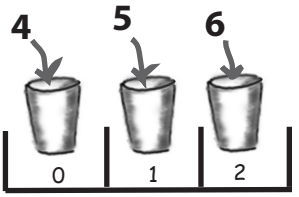
We need a way to know whether a cell has already been hit. Let's run through some possibilities, but first, we'll look at what we know so far...

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.



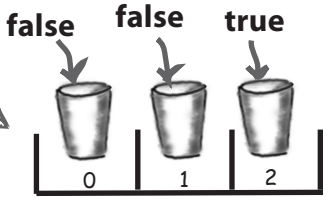
locationCells
(instance variable of the Startup)

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)

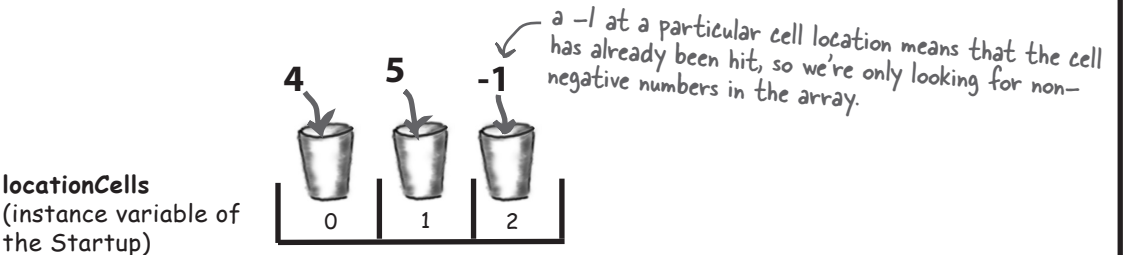
This array holds three values representing the 'state' of each cell in the Startup's location cells array. For example, if the cell at index 2 is hit, then set index 2 in the "hitCells" array to 'true'.

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.



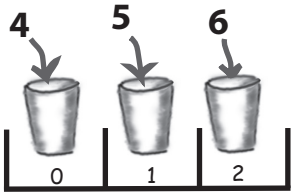
Option two is a little better, but still pretty clunky

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

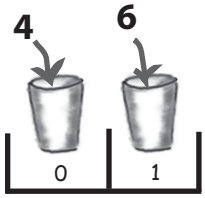
We delete each cell location as it gets hit and then modify the array to be smaller. Except arrays can't change their size, so we have to make a **new** array and copy the remaining cells from the old array into the new smaller array.

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

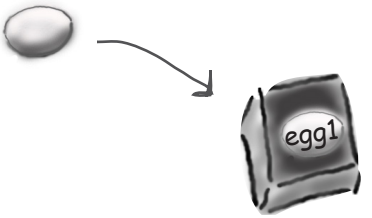
Don't worry about this new <Egg> angle-bracket syntax right now; it just means "make this a list of Egg objects."

```
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

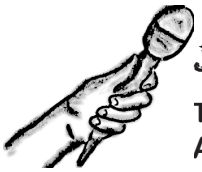
Q: So ArrayList is cool, but how would I know it exists?

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!

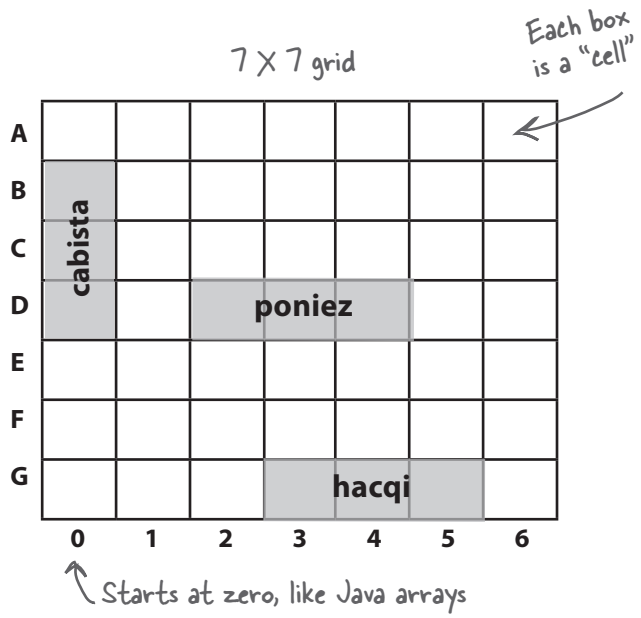
Let's build the REAL game: "Sink a Startup"

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

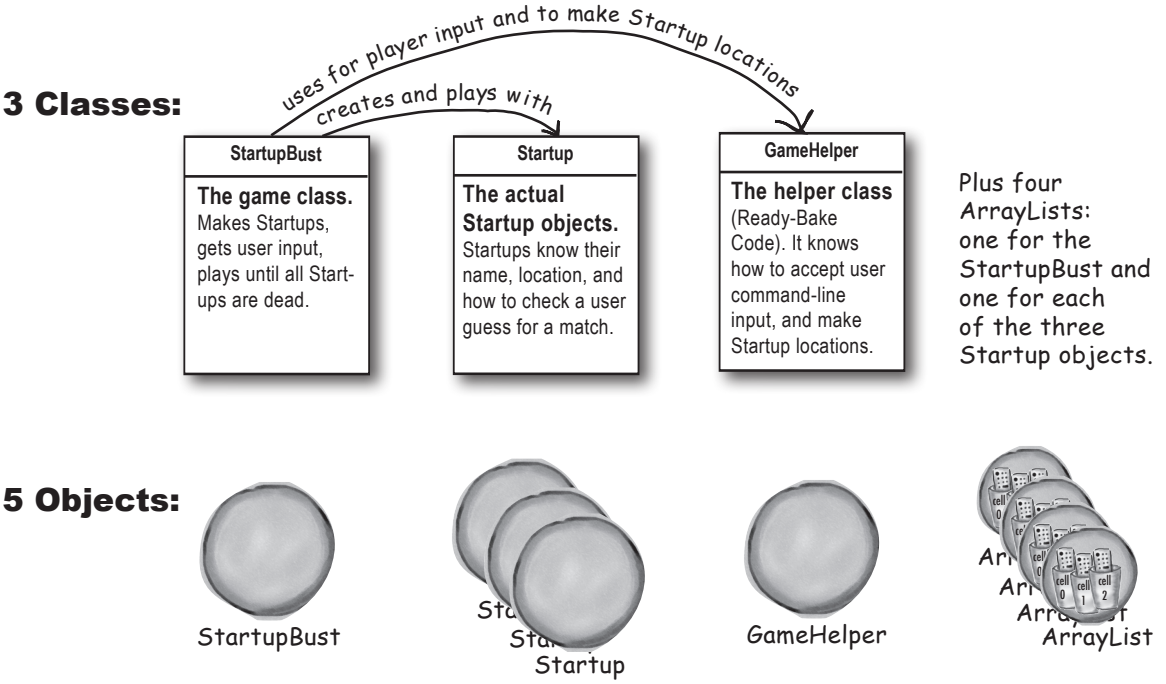
- **Add a *name* variable**
  to hold the name of the Startup ("poniez," "cabista," etc.) so each Startup can print its name when it's killed (see the output screen on the opposite page).

B StartupBust class (the game)

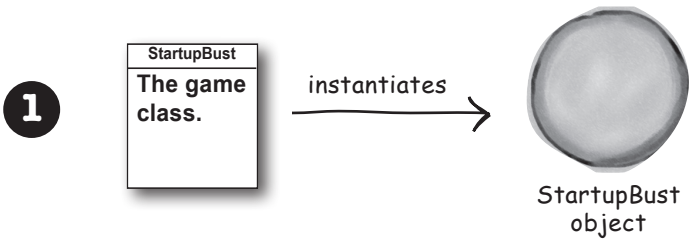
- **Create *three* Startups instead of one.**
- **Give each of the three Startups a *name*.**
  Call a setter method on each Startup instance so that the Startup can assign the name to its name instance variable.

StartupBust class continued...

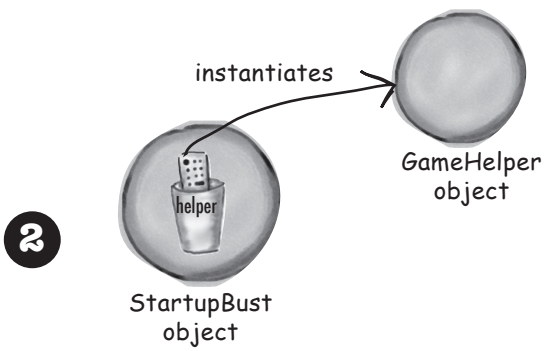
- **Put the Startups on a grid rather than just a single row, and do it for all three Startups.**
  This step is now way more complex than before, if we're going to place the Startups randomly. Since we're not here to mess with the math, we put the algorithm for giving the Startups a location into the 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.



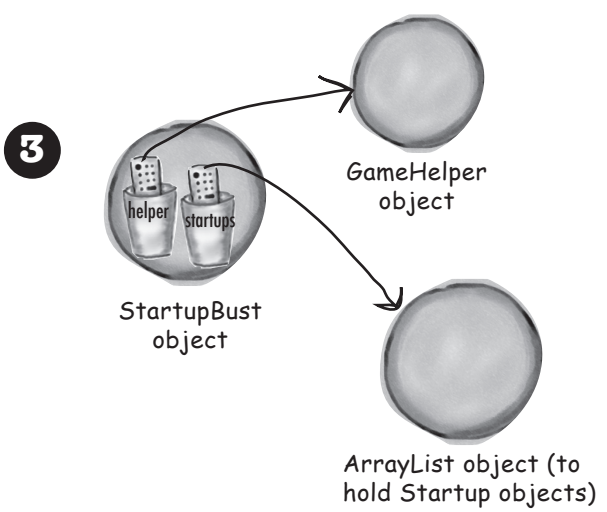
Who does what in the StartupBust game (and when)



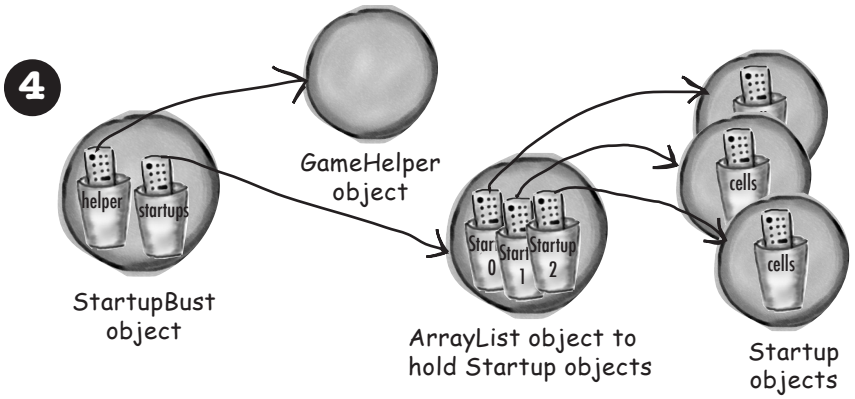
The main() method in the StartupBust class instantiates the StartupBust object that does all the game stuff.



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



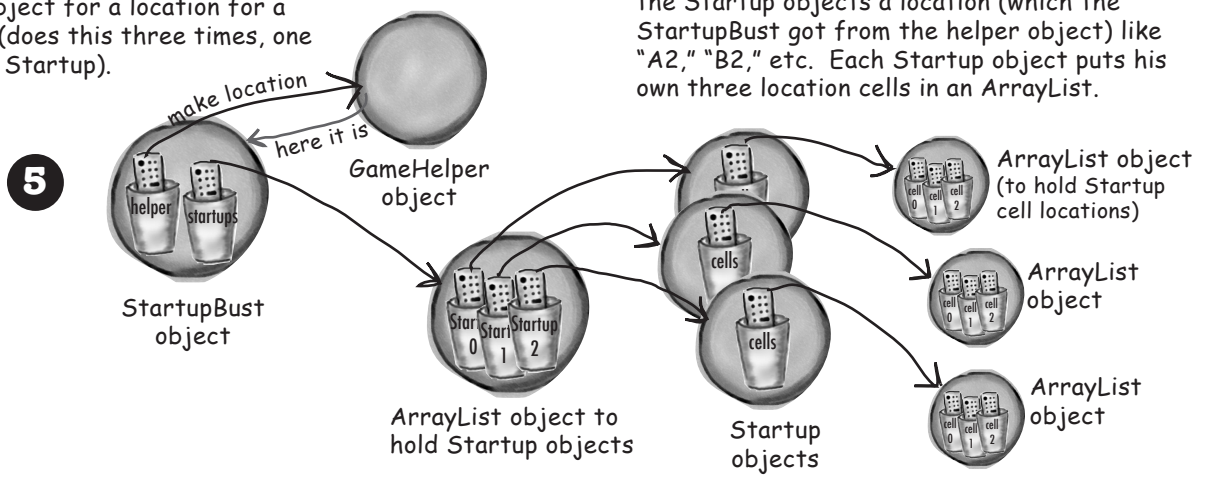
The StartupBust object instantiates an ArrayList that will hold the three Startup objects.



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

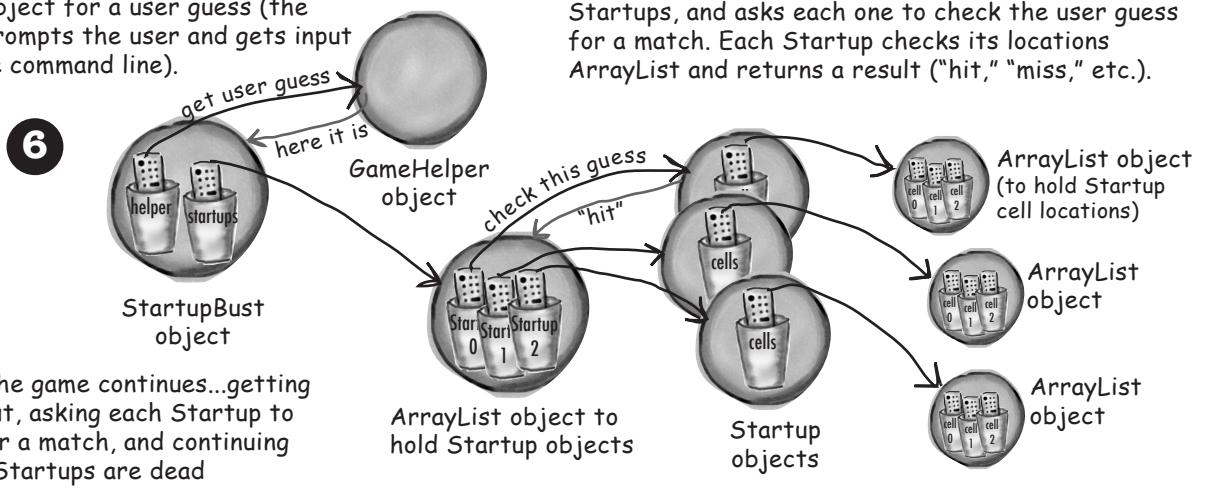
The StartupBust object asks the helper object for a location for a Startup (does this three times, one for each Startup).

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.

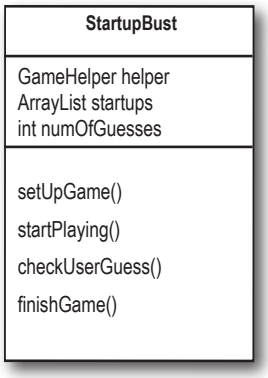


The StartupBust object asks the helper object for a user guess (the helper prompts the user and gets input from the command line).

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...

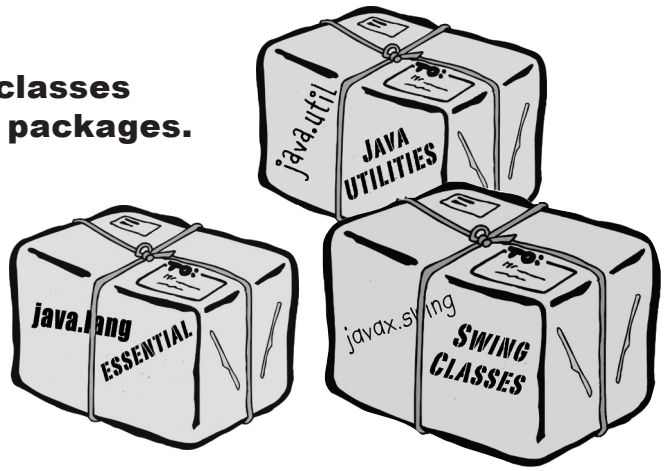
```
  private boolean startupFits(int[] startupCoords, int increment) {
    int finalLocation = startupCoords[startupCoords.length - 1];
    if (increment == HORIZONTAL_INCREMENT) {
      // check end is on same row as start
      return calcRowFromIndex(startupCoords[0]) == calcRowFromIndex(finalLocation);
    } 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.

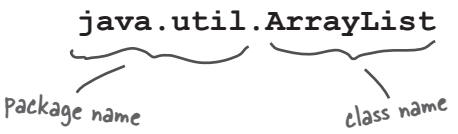
Every class in the Java library belongs to a package. The package has a name, like **javax.swing** (a package that holds some of the Swing GUI classes you'll learn about soon). ArrayList is in the package called **java.util**, which surprise surprise, holds a pile of *utility* classes. You'll learn a 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*.

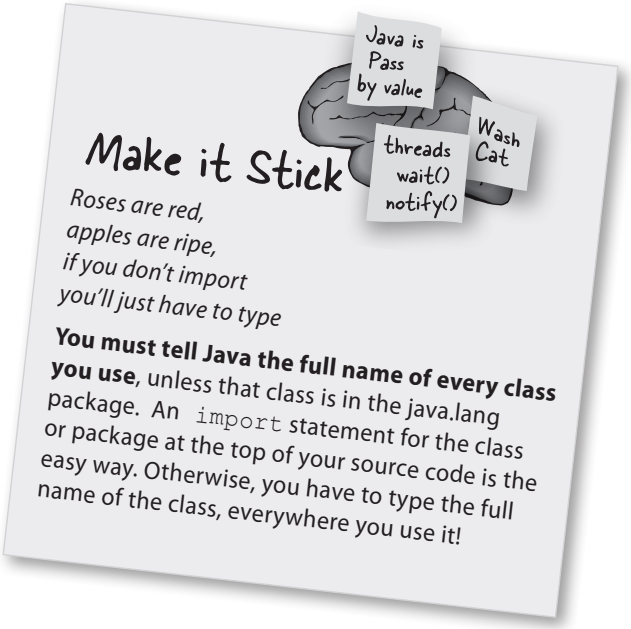
Q: OK, how come I never had to import the String class? Or System?

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.

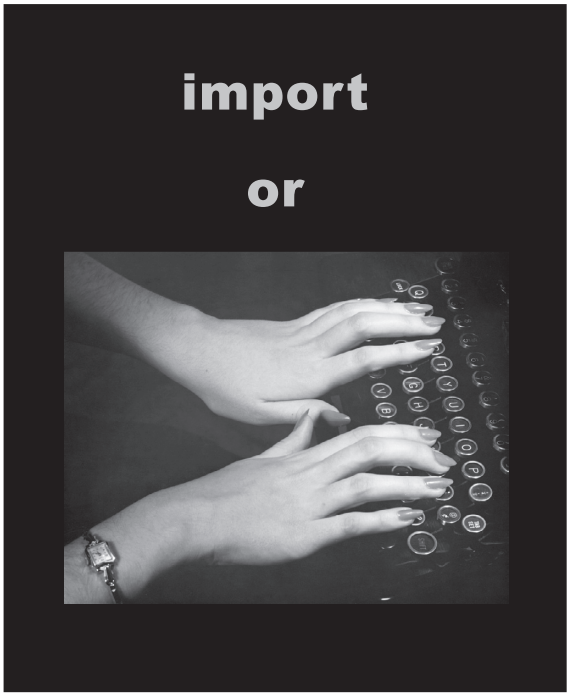
Q: Do I have to put my own classes into packages? How do I do that? *Can* I do that?

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?"*

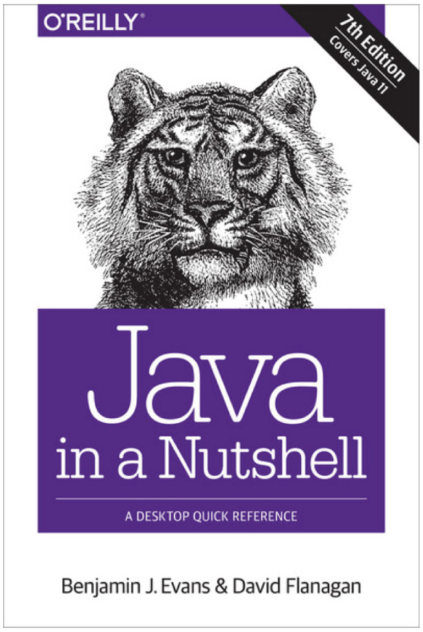
- Julia, 31, hand model

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

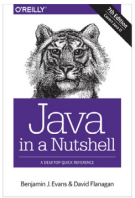
The Java Development Kit (JDK) APIs are specific to the JDK and will not necessar

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.

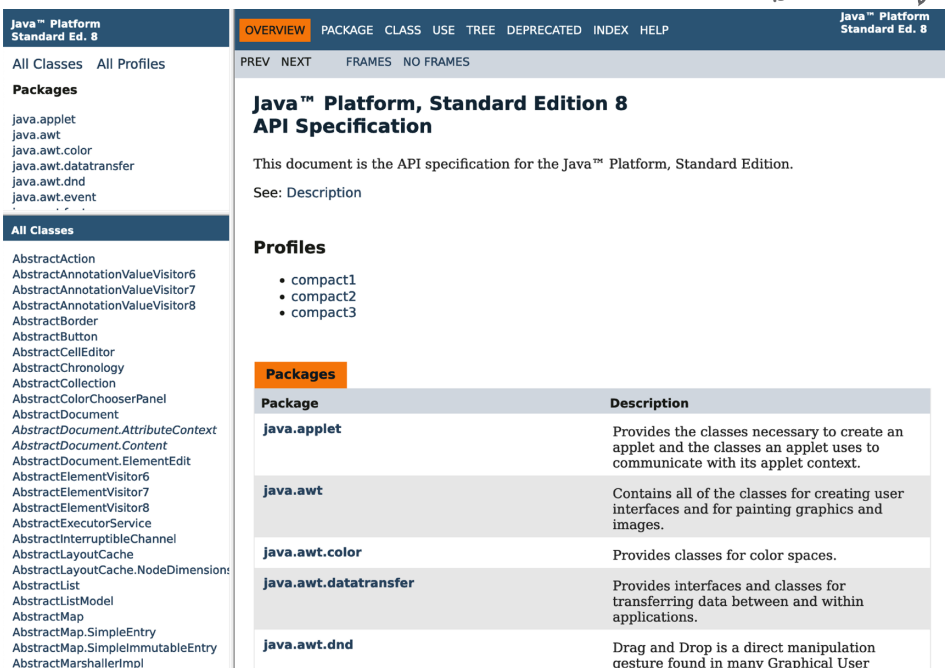
The API docs are the best reference for getting more details about what's in a package, and what the classes and interfaces in the package provide (e.g., in terms of methods and functionality).

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.

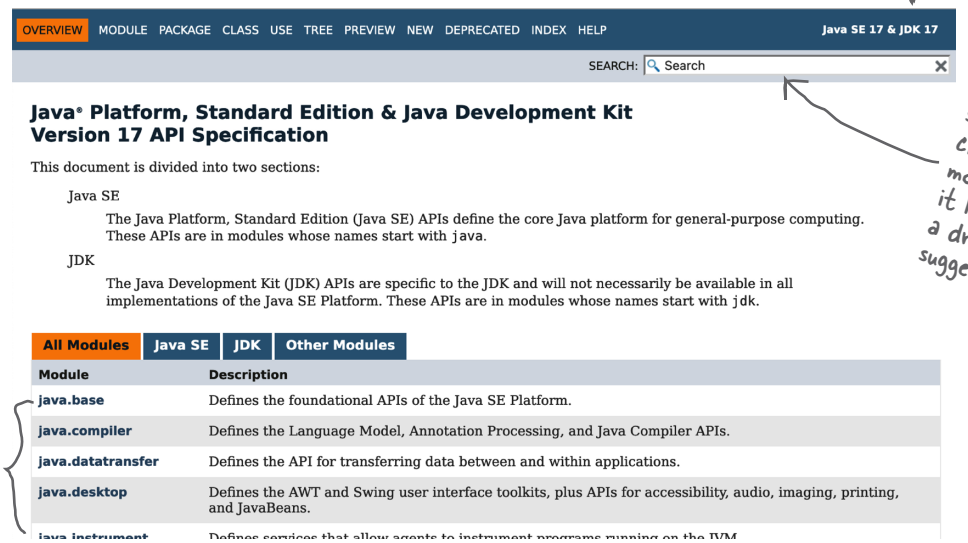
Java 9 and later

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

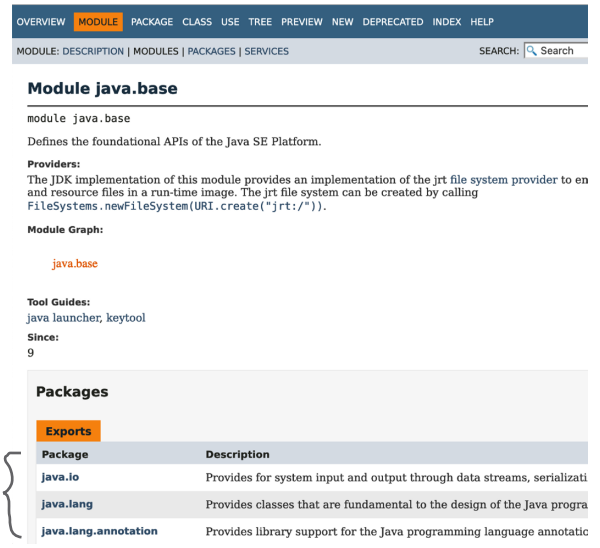
Java 17 is the current Long Term Support (LTS) version at the time of writing.



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:

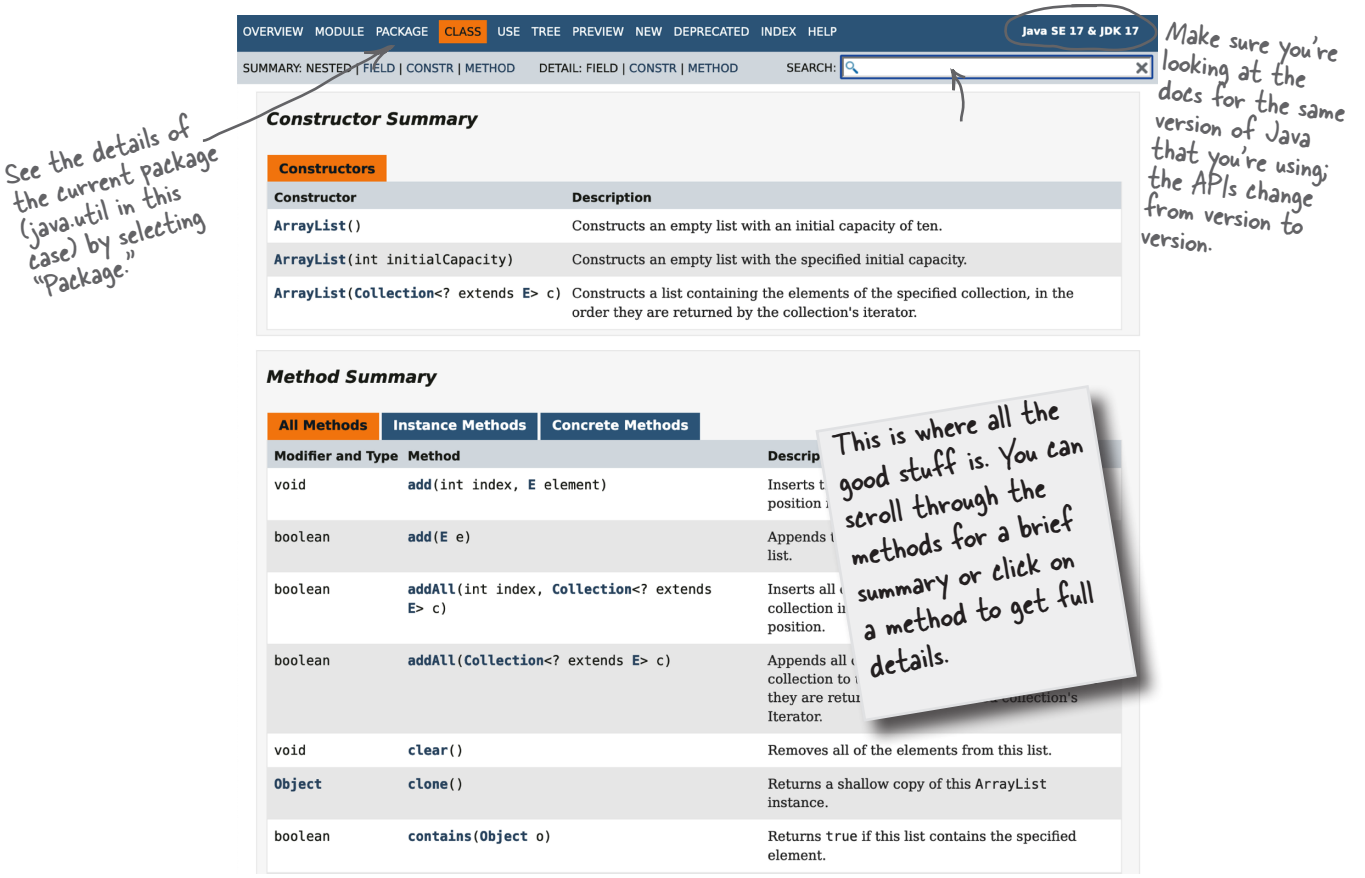
- **Top down:** find a module that looks like it covers the functionality you want, see its packages, and drill down from a package into its classes.
- **Search:** Use the search in the top right to go directly to the method, class, package, or module you want to read about.

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.