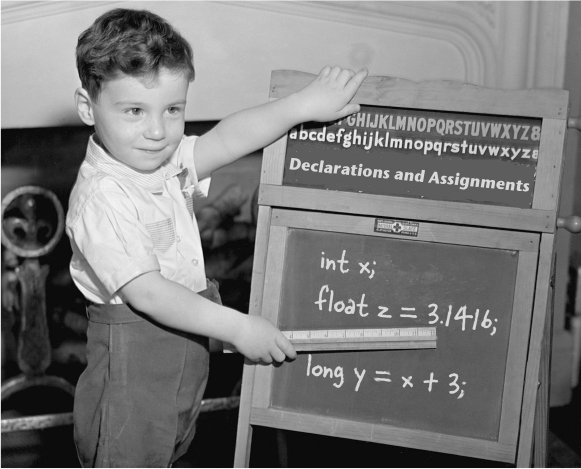
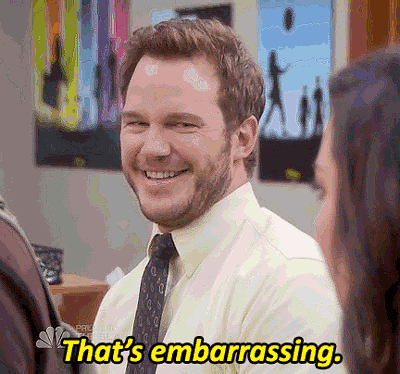
# Chapter 3. Primitives and References: Know Your Variables



****

**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 точность (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 лошадка, 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**



**Note**

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

**“I’d like a double mocha мокко, no, make it an int.”**

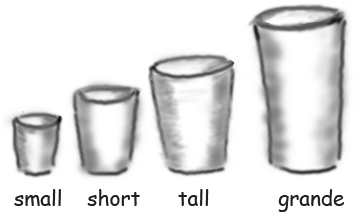
When you think of Java variables, think of cups. Coffee cups, tea cups, giant cups that hold lots and lots of your favorite drink, those big cups the popcorn comes in at the movies, cups with wonderful, tactile handles, and cups with metallic trim that you learned can never, ever go in the microwave.

**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”.

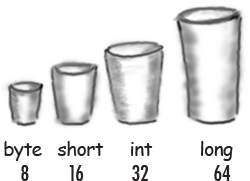
You might see the cups displayed on the counter, so you can order appropriately:



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 (in bits) 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)**

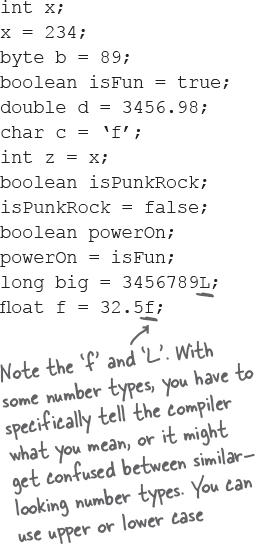
***integer***

|  |  |  |
| --- | --- | --- |
| 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:**



**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 разлив. 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**

image

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;**

**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.**

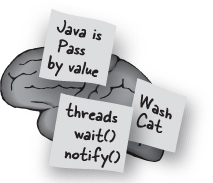
Reserved words are keywords (and other things) that the compiler recognizes. And if you really want to play confuse-a-compiler, then just *try* using a reserved word as a name.

You’ve already seen some reserved words:



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.

**Make it Stick**



*The eight primitive types are:*

**b**oolean **c**har **b**yte **s**hort **i**nt **l**ong **f**loat **d**ouble

*And here’s a mnemonic for remembering them:*

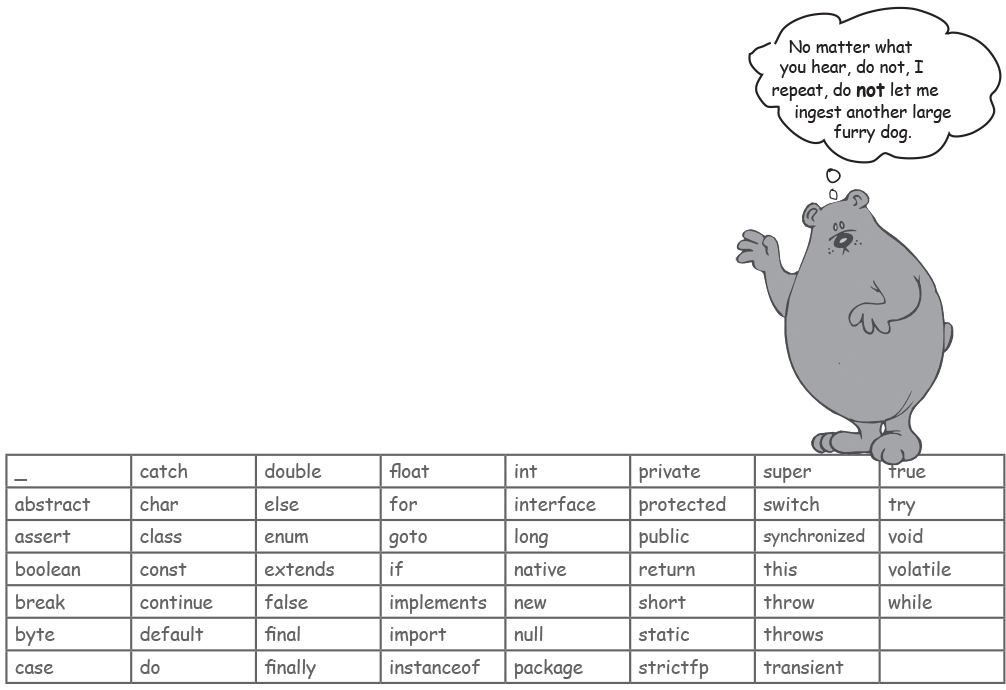
**B**e **C**areful! **B**ears **S**houldn’t **I**ngest **L**arge

**F**urry **D**ogs

*If you make up your own, it’ll stick even better.*

*B\_ C\_ B\_ S\_ I\_ L\_ F\_ D\_*

**This table reserved.**



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.”

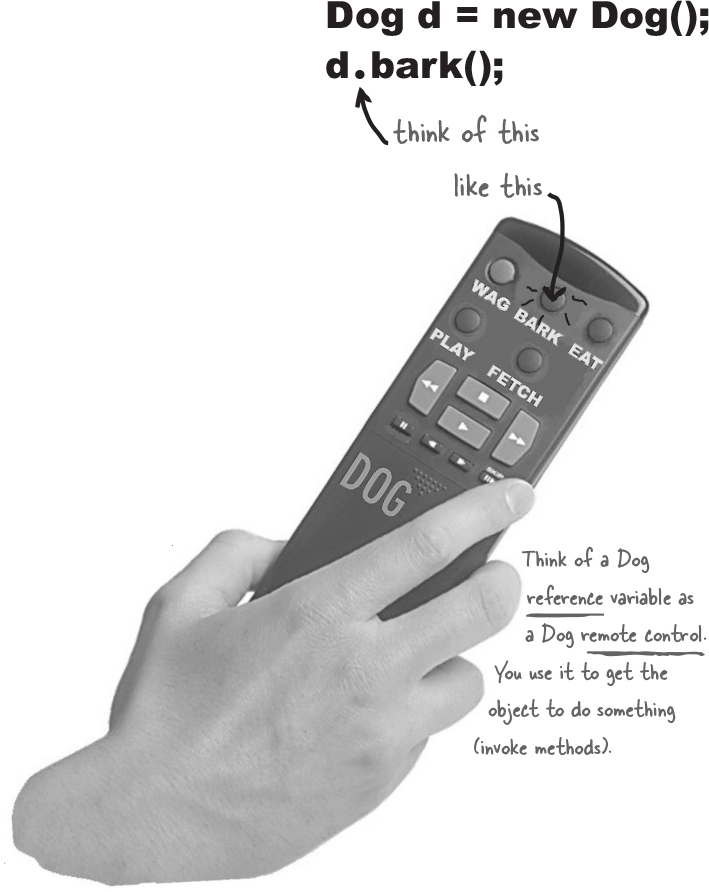
But that’s not what happens. There aren’t giant expandable cups гигантские расширяемые чашки that can grow to the size of any object. Objects live in one place and one place only—the garbage collectible heap! (You’ll learn more about that later in this chapter.)

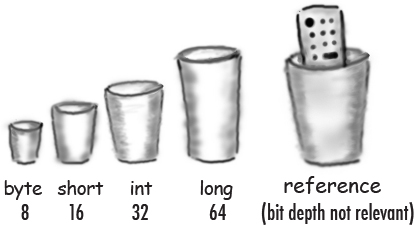
Although a primitive variable is full of bits representing the actual ***value*** of the variable, an object reference variable is full of bits representing ***a way to get to the object.***

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 Используйте объект, на который ссылается переменная myDog, для вызова метода Bark().” 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.





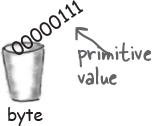
**An object reference is just another variable value.**

**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).

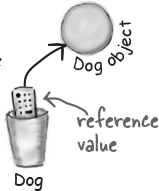


**Reference Variable**

**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!***



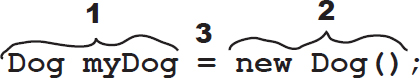
With primitive variables, the value of the variable is... the *value* (5, -26.7, ‘a’).

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 1’s and 0’s 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**



ImagesDeclare 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.

ImagesCreate an object

Dog myDog = new Dog();

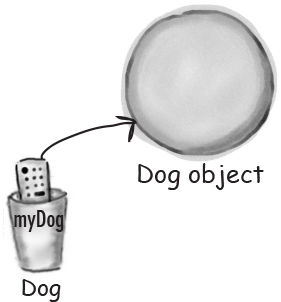
Tells the JVM to allocate space for a new Dog object on the heap (we’ll learn a lot more about that process, especially in [Chapter 9](#_Chapter_9._Constructors).)

ImagesLink 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 невзирая 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. C:\Users\Костя\Desktop\797-7972488_pikachu-pokemon-meme-wow-shook-shocked-wow-pikachu.png

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 Обнаженный**



**This week’s interview: Object Reference**

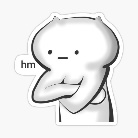
**HeadFirst:** So, tell us, what’s life like for an object reference?

**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 назначен 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 интимный connection, and then the tie is suddenly, cruelly, severed Разорвала. And I will never see that object again, because now it’s eligible подходящий 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();**

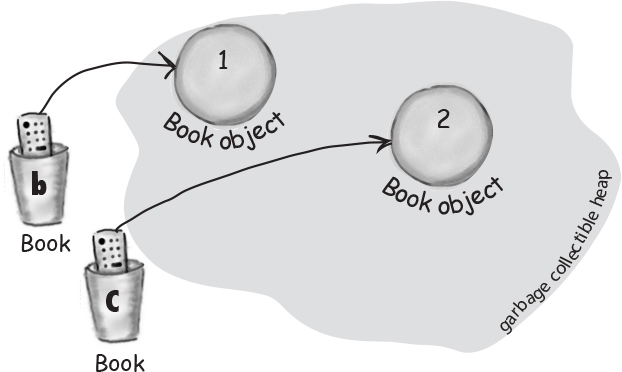
**Book c = new Book();**

Declare two Book reference variables. Create two different 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;**

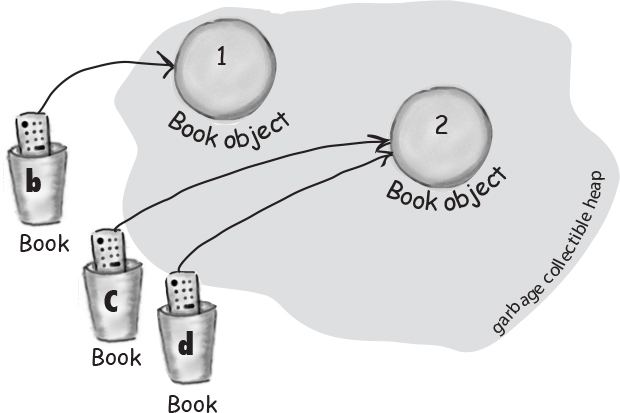
Declare a new Book reference variable. Rather than creating a new, third Book object, assign the value of variable ***c*** to variable ***d***. But what does this mean? It’s like saying, “Take the bits in ***c***, make a copy of them, and stick that copy into ***d***.”

**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;**

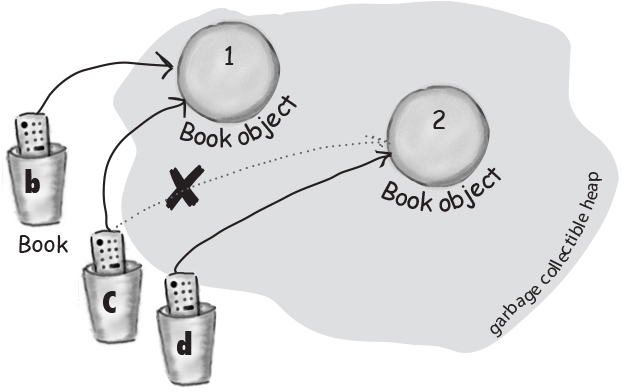
Assign the value of variable ***b*** to variable ***c***. By now you know what this means. The bits inside variable ***b*** are copied, and that new copy is stuffed into variable ***c***.

**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();**

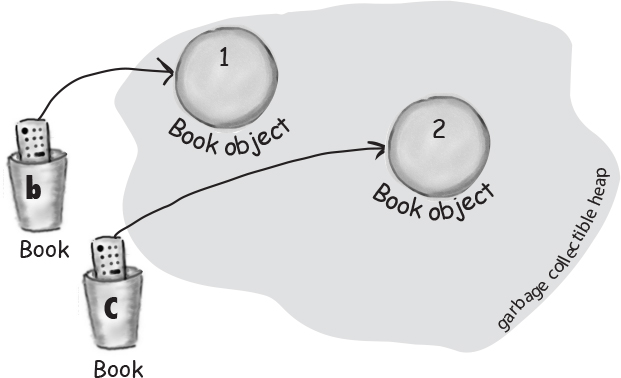
**Book c = new Book();**

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.

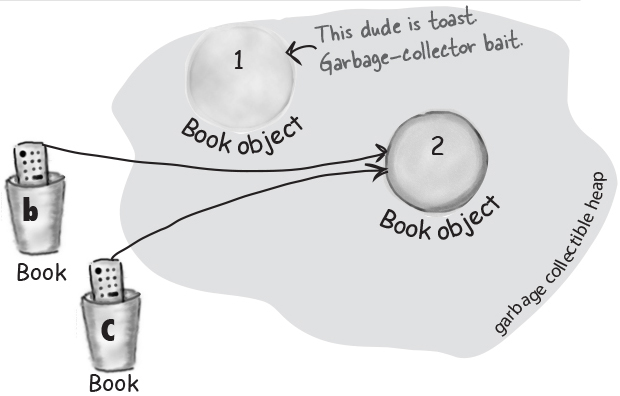
**Both b and c refer to the same object. Object 1 is abandoned and eligible подходящий for Garbage Collection (GC).**

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.

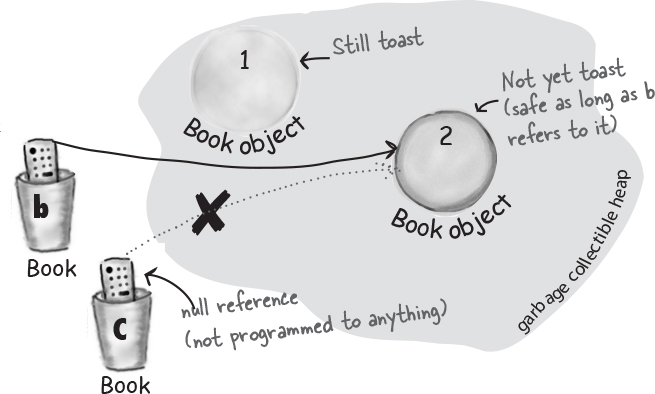
**Object 2 still has an active reference (b), and as long as it does, the object is not eligible for GC.**

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 изощрённый data structures including maps, trees, and sets (see [Appendix B](#_Appendix_B._The)), 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****.*

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

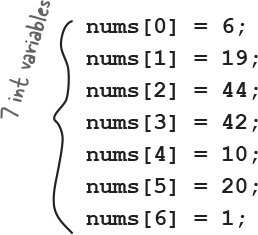
**int[] nums;**

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

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

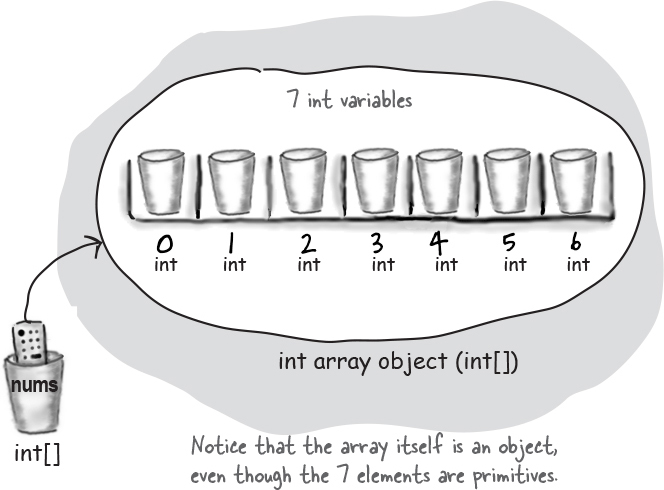
ImagesGive each element in the array some int value.

Remember, elements in an int array are just int variables.



**Note**

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



**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* which 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**

ImagesDeclare a Dog array variable

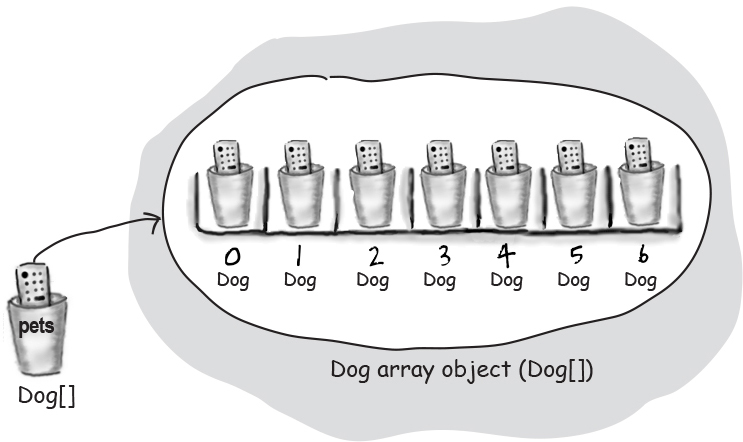
**Dog[] pets;**

ImagesCreate 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*!**

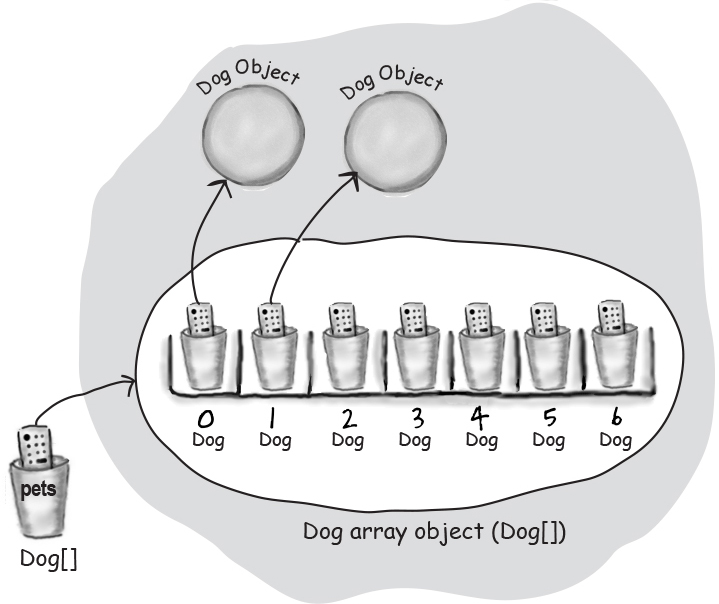


ImagesCreate new Dog objects, and assign them to the array elements.

Remember, elements in a Dog array are just Dog reference variables. We still need Dogs!

**pets[0] = new Dog();**

**pets[1] = new Dog();**



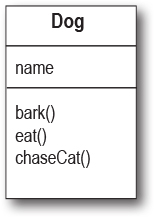
**Sharpen your pencil**

image

What is the current value of pets[2]? - null

What code would make pets[3] refer to one of the two existing Dog objects? pets[3] = pets[1] or pets[3]=pets[2]





**Control your Dog**

**(with a reference variable)**

**Dog fido = new Dog();**

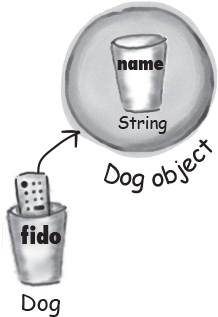
**fido.name = "Fido";**

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();**



**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 неявное расширение. 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.**

**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:

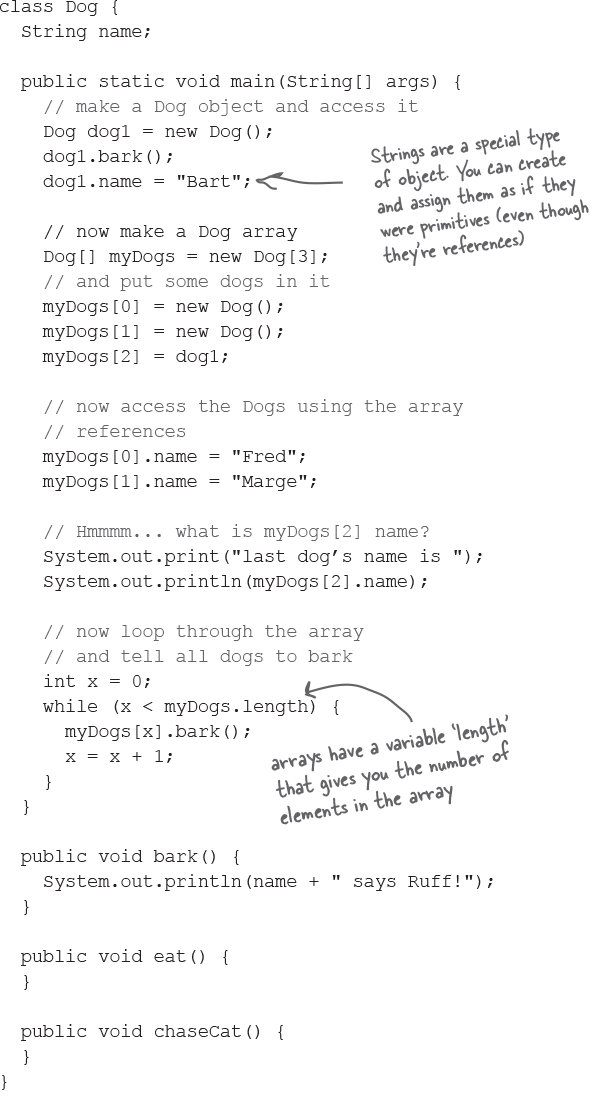
**Dog[] myDogs = new Dog[3];**

**myDogs[0] = new Dog();**

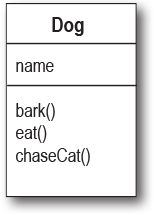
**myDogs[0].name = "Fido";**

**myDogs[0].bark();**

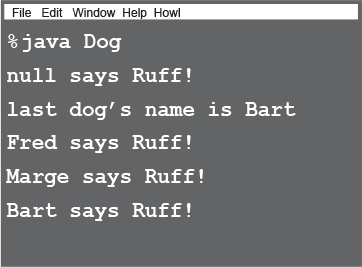
\*Yes we know we’re not demonstrating encapsulation here, but we’re trying to keep it simple. For now. We’ll do encapsulation in [Chapter 4](#_Chapter_4._Methods).



**A Dog example**



**Output**



**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 переменная экземпляра.

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.

**Exercise**



**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");

}

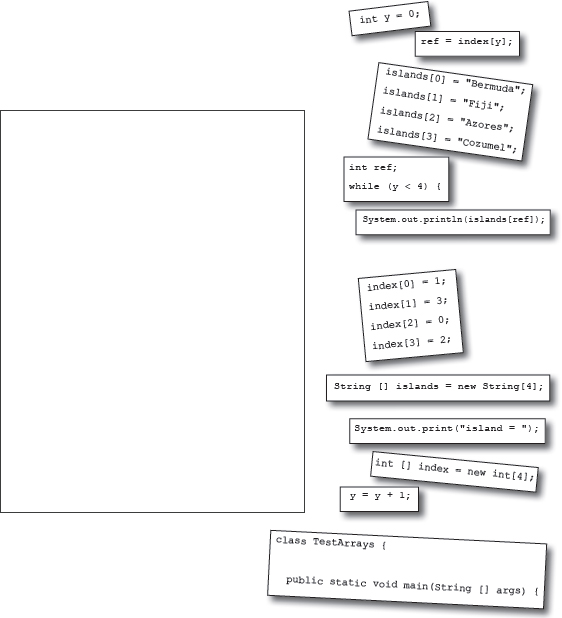
}

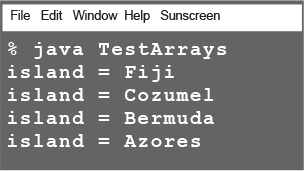
}

**Code Magnets**



A working Java program is all scrambled зашифрованный up on the fridge. Can you reconstruct the code snippets Фрагменты 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!





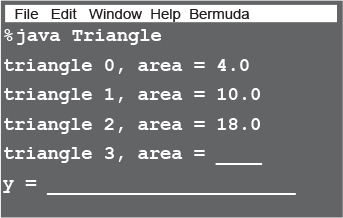
**Pool Puzzle**





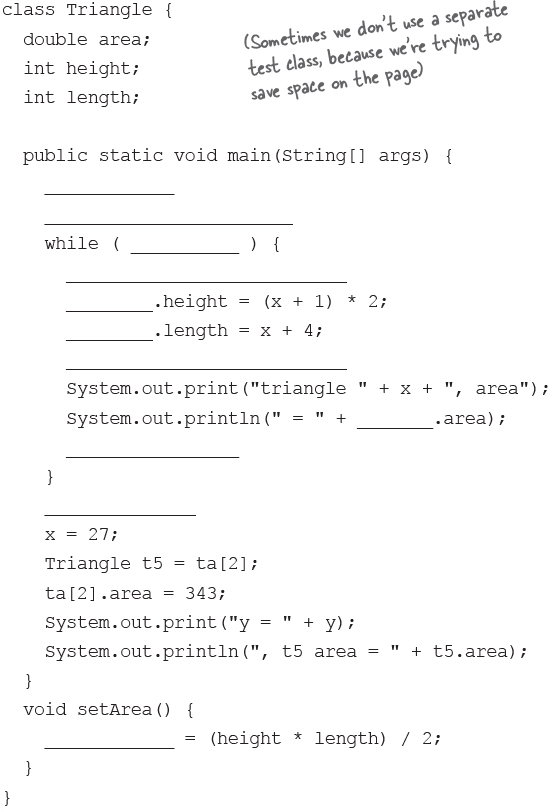
Your ***job*** is to take code snippets Фрагменты from the pool and place them into the blank lines in the code. You **may** use the same snippet more than once, and you won’t need to use all the snippets. Your ***goal*** is to make a class that will compile and run and produce the output listed.

**Output**



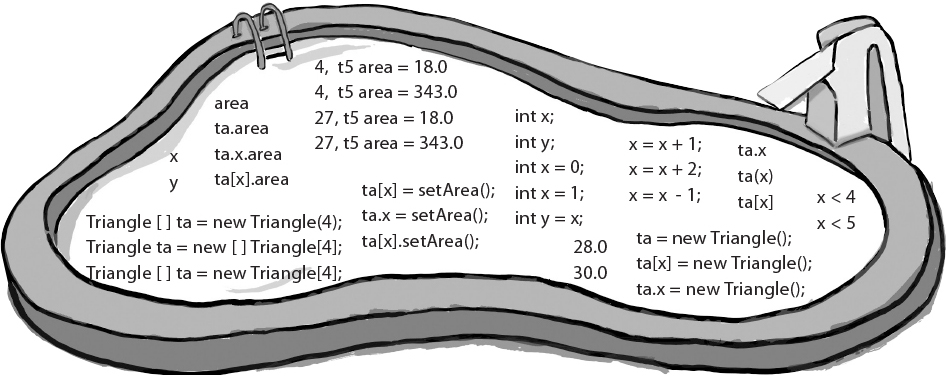
**Bonus Question!**

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



**Note**

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



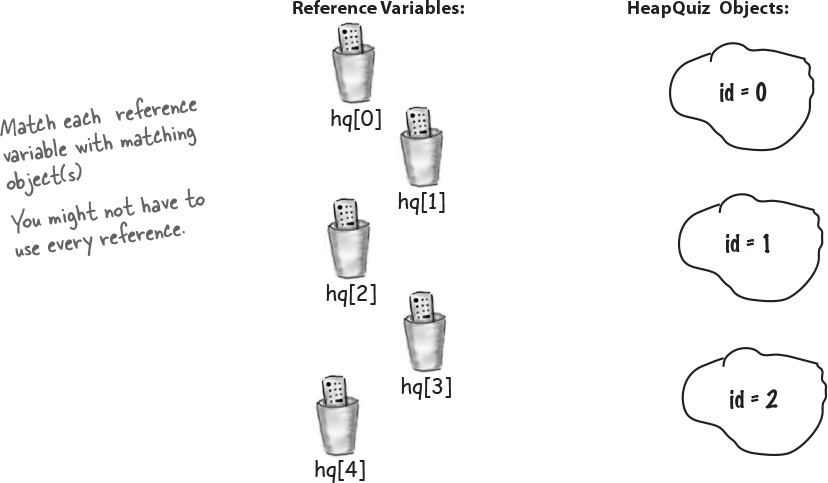
3x

**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 [“Life on the garbage-collectible heap”](file:///C:\Users\Костя\AppData\Local\Temp\bxvibni1.hgu\OEBPS\ch03.xhtml#life_on_the_garbage-collectible_heap)–“Make an array of Dogs” 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

}

}

**The case of the pilfered вороватый references**

It was a dark and stormy night. Tawny Рыжевато-коричневый strolled Гуляли into the programmers’ bullpen кутузка 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 осевой 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 хриплый гул in the bullpen кутузка fell to silence as Tawny eased Облегчить 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 Горячая пора”, she purred мурлыкает. “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 Скользили into the bullpen кутузка. “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 неуклюжий don’t you think?” Kate smirked ухмыльнулся, “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 Издевались 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?*** ******

### Exercise Solutions



**Sharpen your pencil:**



**Code Magnets:**

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 = ");

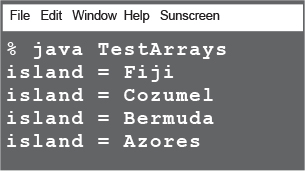
System.out.println(islands[ref]);

y = y + 1;

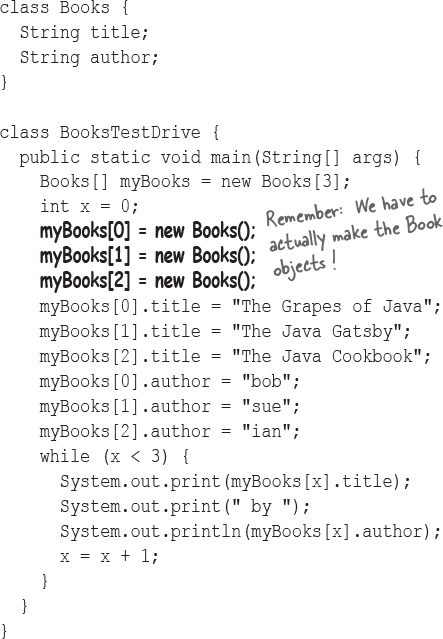
}

}

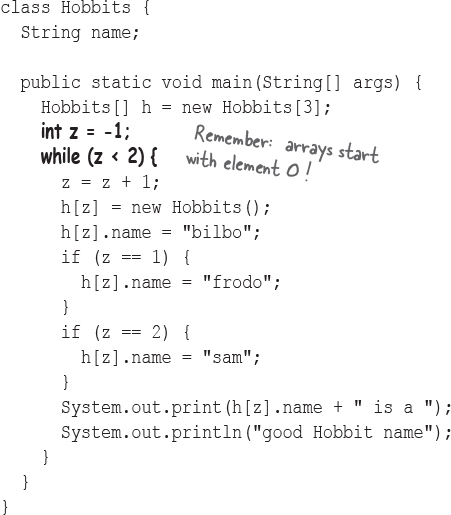
}



**A**



**B**



**Puzzle Solutions**



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);

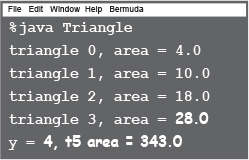
}

void setArea() {

**area** = (height \* length) / 2;

}

}



**The case of the pilfered вороватый references**

Tawny could see that Kate’s method had a serious flaw изъян. 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.)

