

4 methods use instance variables

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.**

objects have state and behavior

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*. The *play()* method plays a song, but the instance you call *play()* on will play the song represented by the value of the *title* instance variable for that instance. So, if you call the *play()* method on one instance you'll hear the song "Politik", while another instance plays "Darkstar". The method code, however, is the same.

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

```
Song t2 = new Song();  
t2.setArtist("Travis");  
t2.setTitle("Sing");  
Song s3 = new Song();  
s3.setArtist("Sex Pistols");  
s3.setTitle("My Way");
```

Calling *play()* on this instance will cause "Sing" to play.

t2.play();

Calling *play()* on this instance will cause "My Way" to play. (but not the Sinatra one)

s3.play();

instance
variables
(state)

methods
(behavior)

Song	
title	artist
setTitle()	setArtist()
play()	

knows

does

five instances
of class Song

Politik
Coldplay

My Way
Sinatra

Darkstar
Grateful
Dead

Sing
Travis

My Way
Sex Pistols



Song



Song

*Yes, another stunningly clear answer!

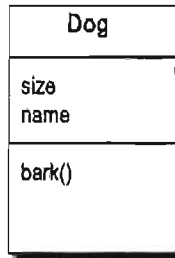
The size affects the bark

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("Woof! Woof!");
        } else if (size > 14) {
            System.out.println("Ruff! Ruff!");
        } else {
            System.out.println("Yip! Yip!");
        }
    }
}
```

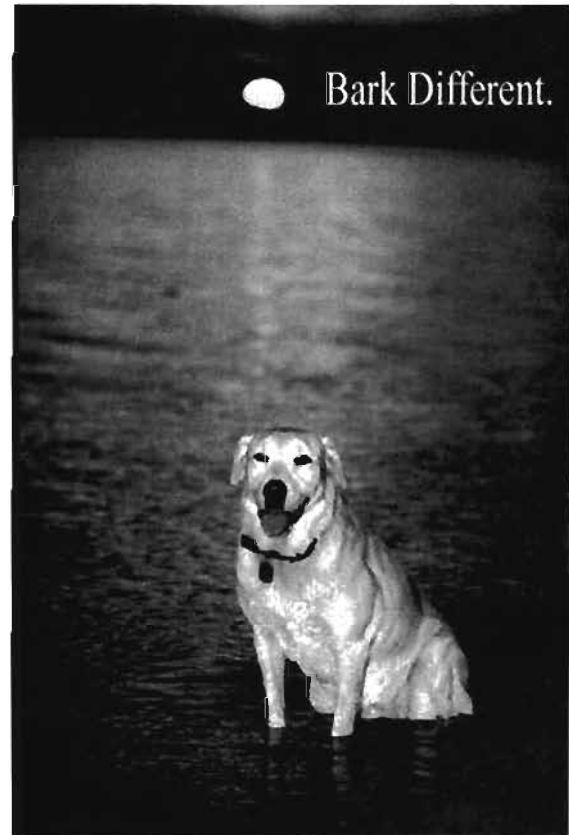


```
class DogTestDrive {

    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
Woof! Woof!
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:

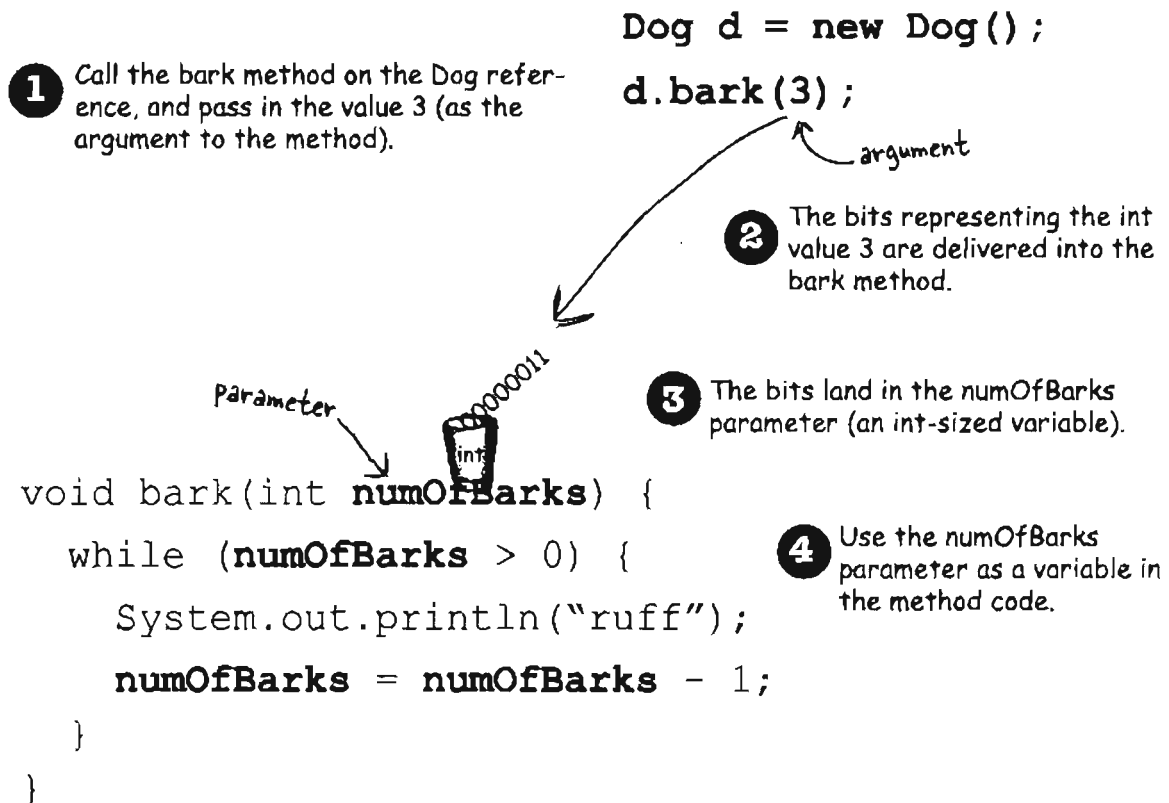
```
d.bark(3);
```

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 method uses parameters. A caller passes arguments.

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.** And that something must be a value of the appropriate type.



You can get things *back* from a method.

Methods can 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:

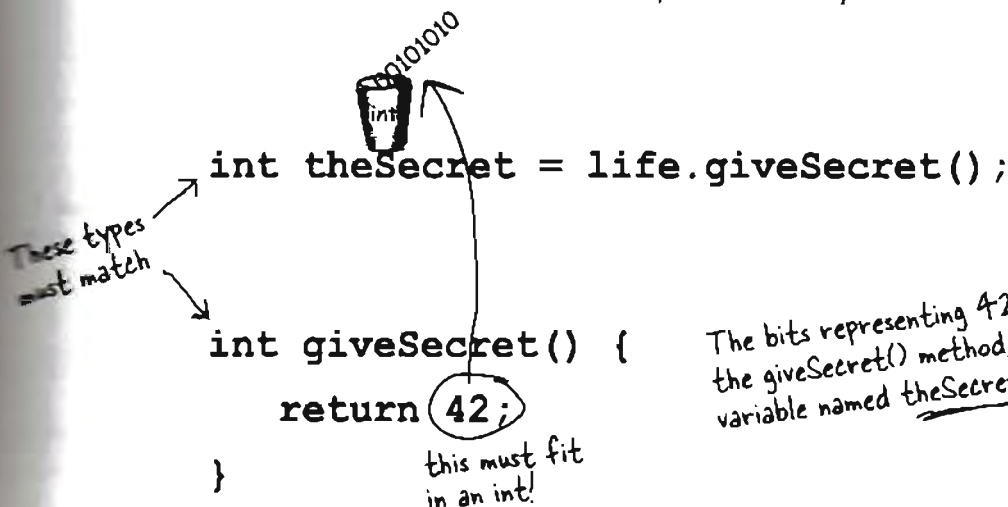
```
int giveSecret() {  
    return 42;  
}
```

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 chapter 7 and chapter 8.)

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



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




The bits representing 42 are returned from the `giveSecret()` method, and land in the variable named `theSecret`.

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);  
}
```

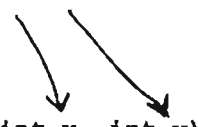


The diagram shows two curved arrows. One arrow starts at the number '12' in the call `t.takeTwo(12, 34);` and points to the parameter `int x` in the method signature `void takeTwo(int x, int y)`. The second arrow starts at the number '34' in the call and points to the parameter `int y` in the method signature.

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 diagram shows two curved arrows. One arrow starts at the variable `foo` in the call `t.takeTwo(foo, bar);` and points to the parameter `int x` in the method signature `void takeTwo(int x, int y)`. The second arrow starts at the variable `bar` in the call and points to the parameter `int y` in the method signature.

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;
```

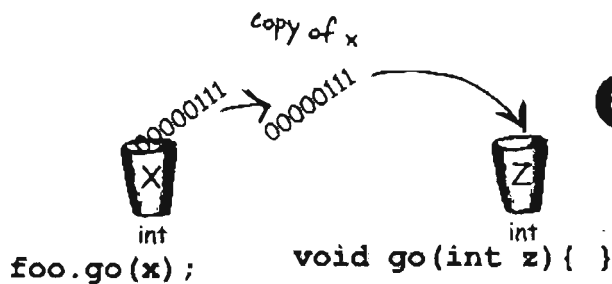


- 1 Declare an int variable and assign it the value '7'. The bit pattern for 7 goes into the variable named x.

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

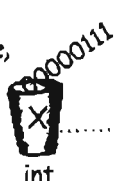


- 2 Declare a method with an int parameter named z.



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

x doesn't change, even if z does.



x and z aren't connected



```
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?

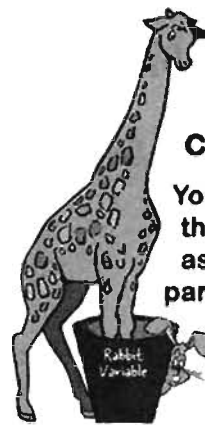
A: Sort of. A method can declare only one return value. BUT... if you want to return, say, three int values, then the declared return type can be an int *array*. Stuff those ints into the array, and pass it on back. It's a little more involved to return multiple values with different types; we'll be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

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: **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 the 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;
    }
}
```

ElectricGuitar	
brand	
numOfPickups	←
rockStarUsesIt	
getBrand()	
setBrand()	
getNumOfPickups()	
setNumOfPickups()	
getRockStarUsesIt()	
setRockStarUsesIt()	

Note: Using these naming conventions means you'll be following an important Java standard!



real developers encapsulate

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;
```

yikes! 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.



Jen says you're well-encapsulated...

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: Doesn't take a PhD here. Most instance variable values are coded with certain assumptions about the boundaries of the values. 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. Cell 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 do-able. 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("Woof! Woof!");
```

```
        } 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();
```

```
    }
```

```
}
```

GoodDog
size
getSize() setSize() bark()

Even though the methods don't really add new functionality, the cool 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:

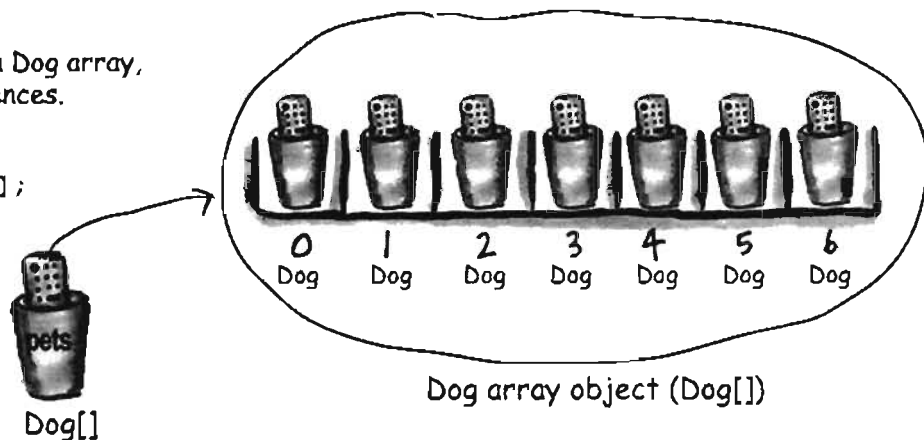
```
int x = 3 + one.getSize();
```

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 7 Dog references.

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

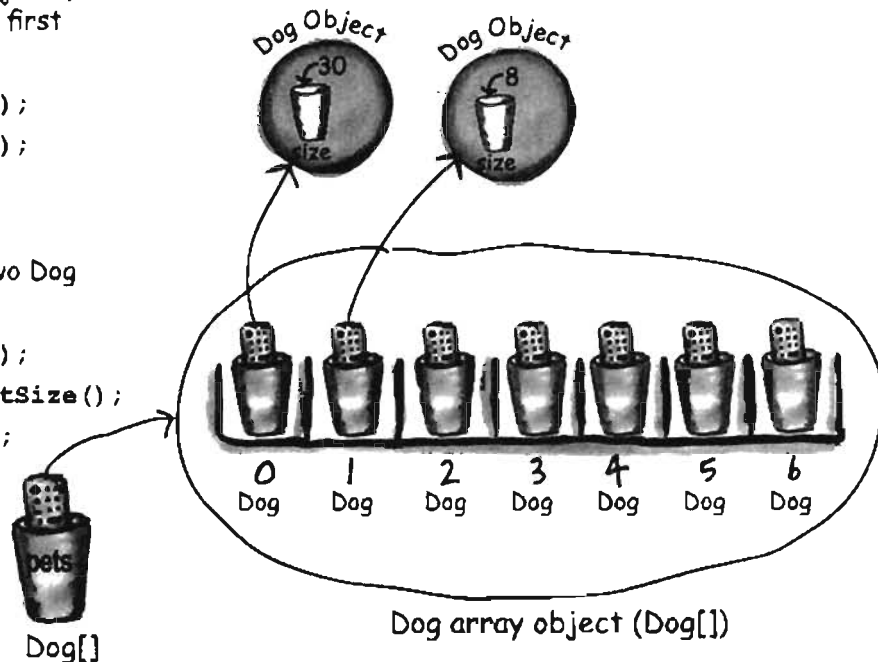


- 2 Create two new Dog objects, and assign them to the first two array elements.

```
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?

```
class PoorDog {
```

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

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

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

```
}
```

```
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());
    }
```

```
}
```

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

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

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

What will these return??

What do you think? Will this even compile?

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;
    int b = 12;

    public int add() {
        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.

But that's because the compiler will give you an error if you try to invoke a method without sending 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 declared for the method, and the arguments are assigned (automatically) to the parameters.

Comparing variables (primitives or references)

Sometimes you want to know if two *primitives* are the same. 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. 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. The idea of equality for objects depends on the type of object. For example, if two different `String` objects have the same characters (say, "expeditious"), 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 (and appendix B), 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.

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.

(Such as two different `String` objects that both represent the characters in "Fred")

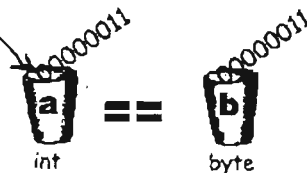
To compare two primitives, use the `==` operator

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 it doesn't care about the size of the variable, so all the extra zeroes on the left end don't matter).

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

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

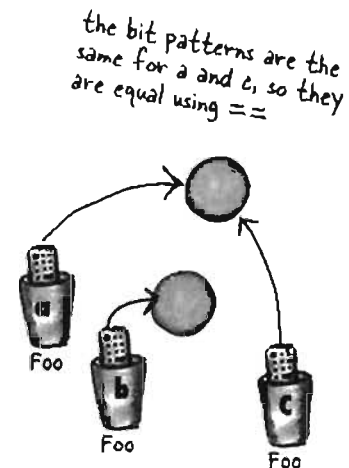


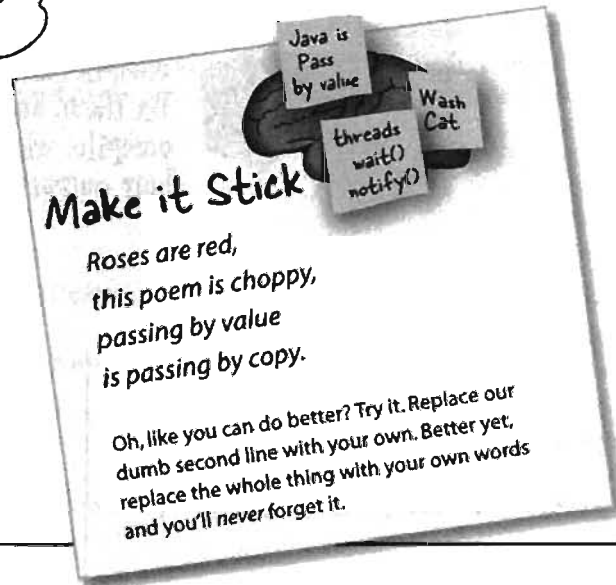
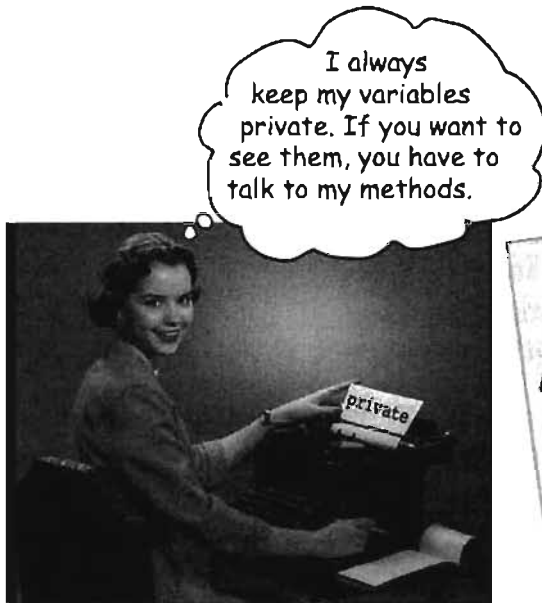
To see if two references are the same (which means they refer to the same object on the heap) use the `==` operator

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 }
```

`a == c` is true
`a == b` is false





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);
```

exercise: Be the Compiler



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);  
    }  
}
```



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 guy, 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.

puzzle: Mixed Messages



Mixed Messages

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.

Candidates:

`x < 9`

`index < 5`

`x < 20`

`index < 5`

`x < 7`

`index < 7`

`x < 19`

`index < 1`

Possible output:

14 7

9 5

19 1

14 1

25 1

7 7

20 1

20 5

```
public class Mix4 {
    int counter = 0;
    public static void main(String [] args) {
        int count = 0;
        Mix4 [] m4a = new Mix4[20];
        int x = 0;
        while (  ) {
            m4a[x] = new Mix4();
            m4a[x].counter = m4a[x].counter + 1;
            count = count + 1;
            count = count + m4a[x].maybeNew(x);
            x = x + 1;
        }
        System.out.println(count + " "
                            + m4a[1].counter);
    }

    public int maybeNew(int index) {
        if (  ) {
            Mix4 m4 = new Mix4();
            m4.counter = m4.counter + 1;
            return 1;
        }
        return 0;
    }
}
```



Pool Puzzle

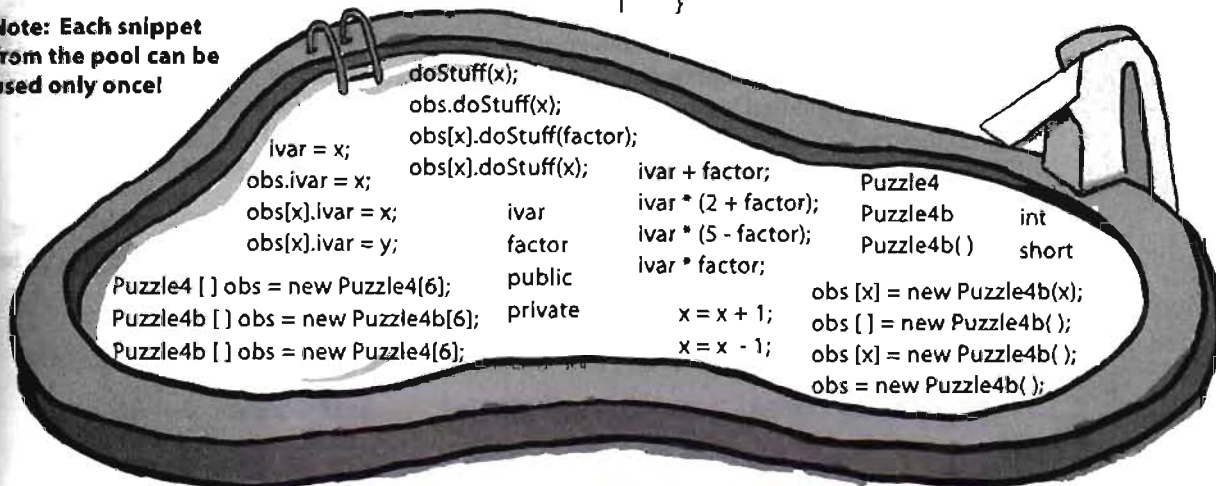


Your **job** is to take code snippets from the pool and place them into the blank lines in the code. You may **not** use the same snippet more than once, and you won't need to use all the snippets. Your **goal** is to make a class that will compile and run and produce the output listed.

Output

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

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



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

        _____

        int y = 1;
        int x = 0;
        int result = 0;
        while (x < 6) {

            _____

            y = y * 10;

            _____

        }
        x = 6;
        while (x > 0) {

            _____

            result = result + _____

        }
        System.out.println("result " + result);
    }

    class _____ {
        int ivar;

        _____ doStuff(int _____) {
            if (ivar > 100) {
                return _____
            } else {
                return _____
            }
        }
    }
}
```



Fast Times in Stim-City

When Buchanan jammed his twitch-gun into Jai's side, 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 J37NE", 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 J37NE 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 reached for his twitch-gun but Leveler's stunner was already on Buchanan's neck, "Let it go Buchanan", sneered Leveler, "Drop the twitcher 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?

Exercise Solutions

```

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.

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

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

puzzle answers

Puzzle Solutions

```
public class Puzzle4 {
    public static void main(String [] args) {
        Puzzle4b [] obs = new Puzzle4b[6];
        int y = 1;
        int x = 0;
        int result = 0;
        while (x < 6) {
            obs[x] = new Puzzle4b();
            obs[x].ivar = y;
            y = y * 10;
            x = x + 1;
        }
        x = 5;
        while (x > 0) {
            x = x - 1;
            result = result + obs[x].doStuff(x);
        }
        System.out.println("result " + result);
    }
}

class Puzzle4b {
    int ivar;
    public int doStuff(int factor) {
        if (ivar > 100) {
            return ivar * factor;
        } else {
            return ivar * (5 - factor);
        }
    }
}
```

Output

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

Answer to the 5-minute mystery...

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.

Candidates:

Possible output:

