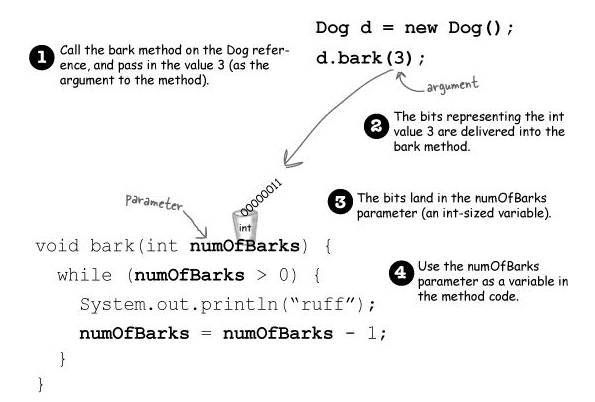
**B”H**

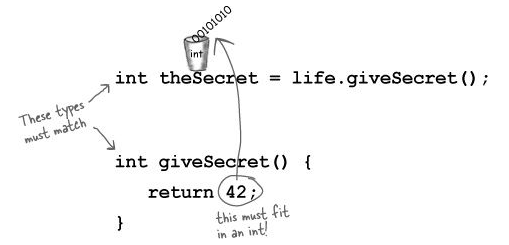
**Chapter 4**

**NOTE: CORRECTION: a “.java” file can contain multiple classes however when compiled it results in one “.class” file per class**

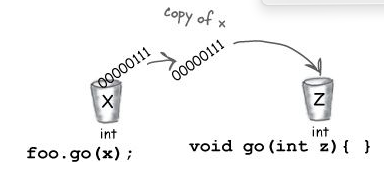
* State affects behavior, behavior affects state.
* Every instance of a particular class has the same methods, but the methods can behave differently based on the value of the instance variables.
* You can pass values into your methods:
  + A method uses parameters. A caller passes arguments.
  + In other words: an argument (a value like 2, “Foo”, or a reference to a Dog) lands face-down into a parameter.
  + A parameter is nothing more than a local variable.
* If a method takes a parameter, you must pass it something.



* Methods can return values. Every method is declared with a return type (a void return type means they don’t give anything back).
* 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.)



* Methods can have multiple parameters.
* You can pass variables into a method, as long as the variable type matches the parameter type
* Java is pass-by-value. That means pass-by-copy



* 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.
* 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.
* 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.. 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.
* Java doesn’t require you to acknowledge a return value. You don’t have to assign or use the return value.
* Any place where a particular value can be used, a method call that returns that type can be used.

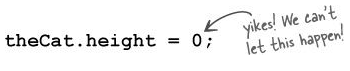
Instead of: **int x = 3 + 24;**

You can say: **int x = 3 + one.getSize();**

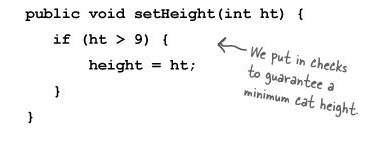
**Encapsulation:**

Encapsulation puts a force-field around the instance variables, so nobody can set them to something inappropriate.

* Until this most important moment, we’ve been committing one of the worst OO faux pas: Exposing our data!
* Exposed means reachable with the dot operator, as in: **theCat.height = 27;**
* Because what’s to prevent:

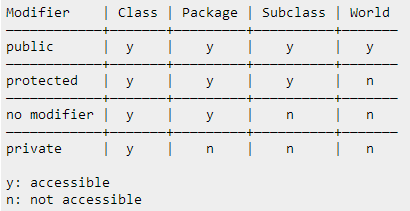


* By forcing everybody to call a setter method, we can protect the cat from unacceptable size changes.

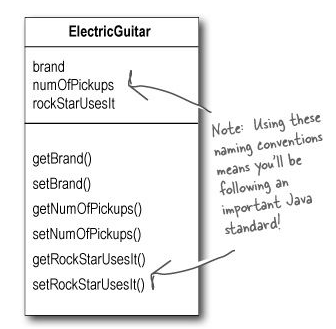


.

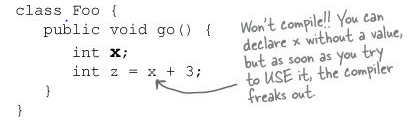
* How exactly do you hide the data? With the **public** and **private** access modifiers



* Getters and Setters (fancy names are Accessors and Mutators):



* General rule of thumb: mark all instance variables private; and make getters and setters and mark them public.
* Even if at the moment you don’t have any special code to put in the setter or getter, still add them, because:
  + 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.
* 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
* The difference between instance and local variables:
  + Instance variables are declared inside a class but not within a method.
  + Local variables are declared within a method.
  + Local variables do NOT get a default value! The compiler complains if you try to use a local variable before the variable is initialized.



* Method parameters are also local variables — but method parameters will never be uninitialized, because the compiler guarantees that methods are always called with arguments that match the parameters.
* Use **==** to compare two primitives, or to see if two references refer to the same object.
  + It simply compares the bits:

**int a = 3;**

**byte b = 3;**

**if (a == b) { // true }**

**Foo a = new Foo();**

**Foo b = new Foo();**

**Foo c = a;**

**if (a == b) { // false }**

**if (a == c) { // true }**

**if (b == c) { // false }**

* 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”)
  + Note we’ll explore the notion of object equality again in later chapters (and Appendix B ),