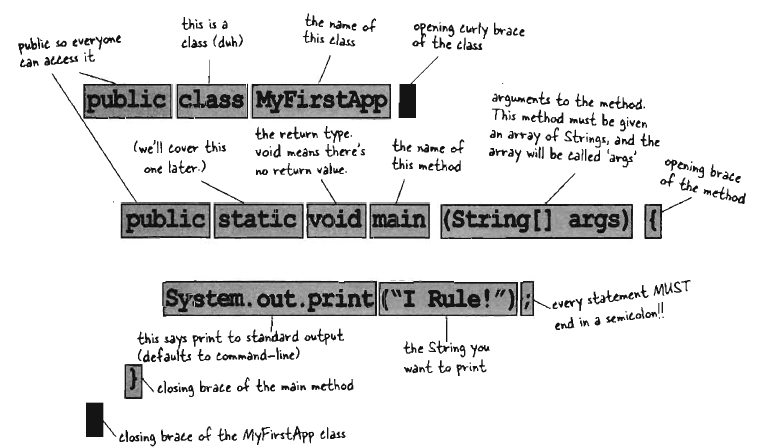
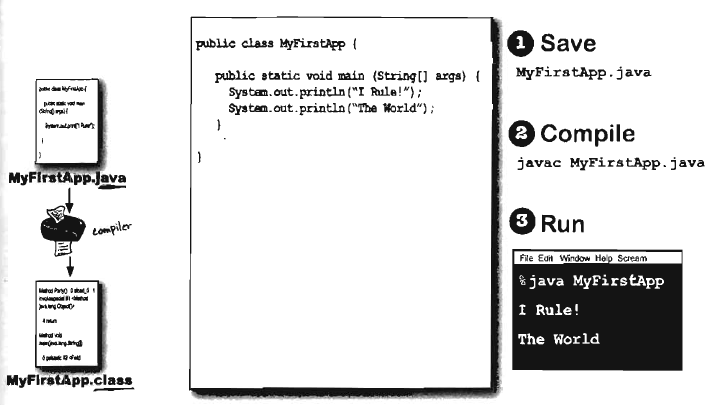
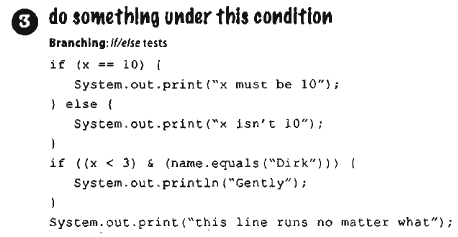
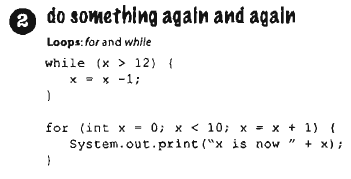
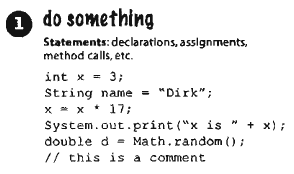
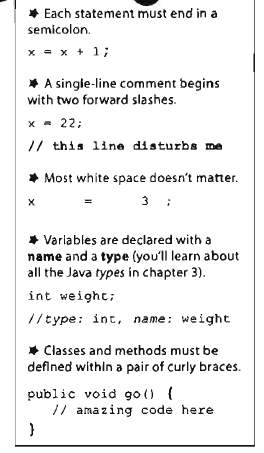
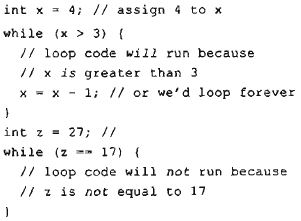
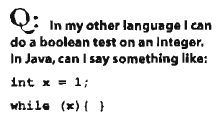
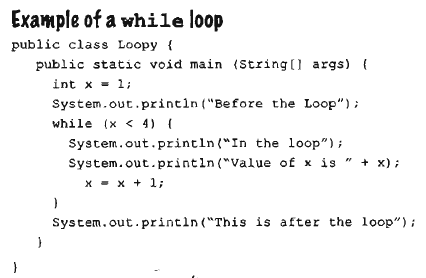
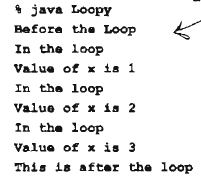
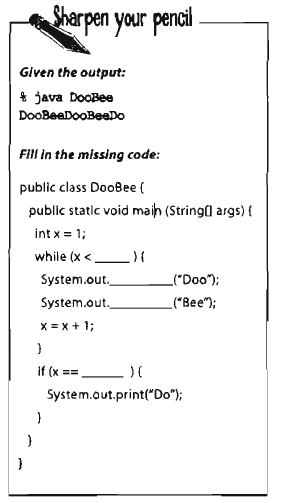
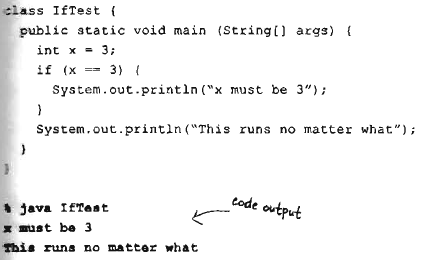
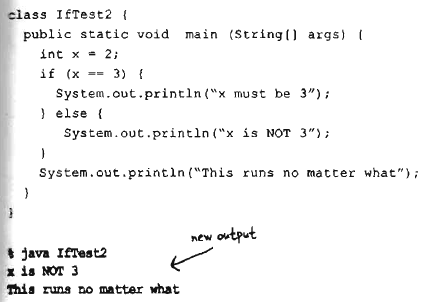
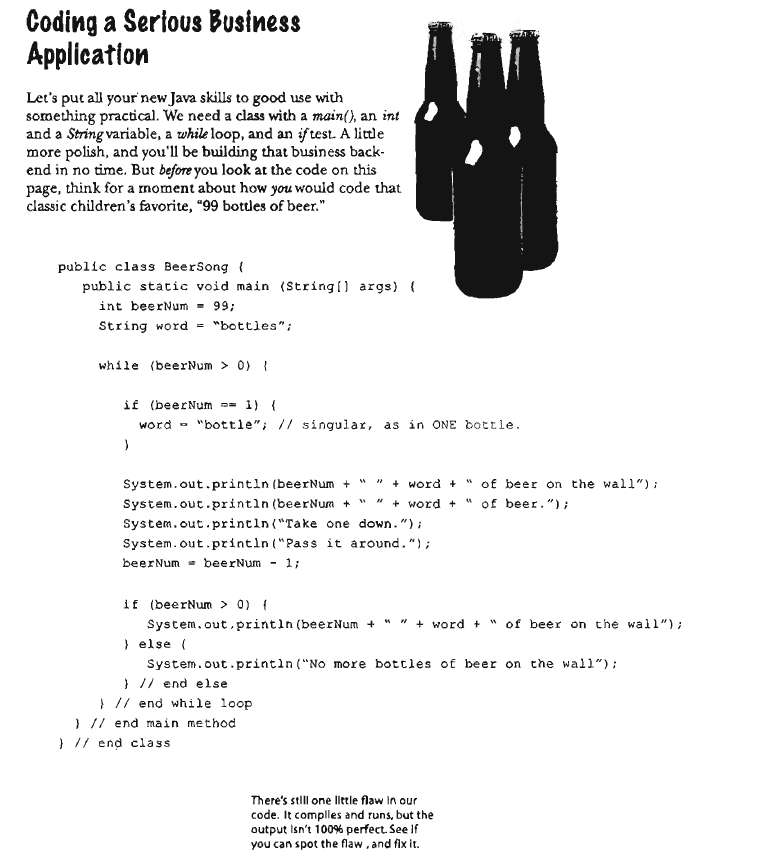
**The way Java works.**  
Source -> Compiler -> Output (Code) -> Virtual Machines (that runs the bytecode from the code)  
  
**Code Structure**Source code file (.java extension) holds one class definition, class is piece of the program.  
Class has one or more methods, holds instruction for how the class works.  
Method holds statement that tells how the method should be performed, set of statements.  
  
**Anatomy of a class**When JVM is running, it looks for the class you give at the command line, then it looks for method, then it runs everything between the bracket of the main method.  
  
Every Java application has to have at least one class and at least one method. (one main per application).  
  
  
**Writing a class with a main**Everything goes into a class, first type source code file with .java extension, then compile it into new class file (.class extension). Running program = Running class.  
  
Running program -> Load “Hello” Class, then execute its main { } method until all statement is ran.  
  
main { } method is where the program starts to run, no matter how many class exist, there has to be a main { } method.  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
**What can you say in the main method ?**

**Loops**Three standard looping construct: *while, do-while, for.*  
**While –** As long as condition is true, the statement inside the  
loop block is ran.  
  
Conditional test – expression that results in Boolean value.  
  
Example: “While IceCreamInTheTub is true, keep scooping”,  
It is Boolean test, either ice cream is in the tub or isn’t.  
  
**Simple Boolean tests**Perform simple Boolean test by checking the value of variable, using a comparison operator  
**< (less than), > (greater than) , == (equality)**.  
  
Single = (assignment operator)  
Double == (equals operator)

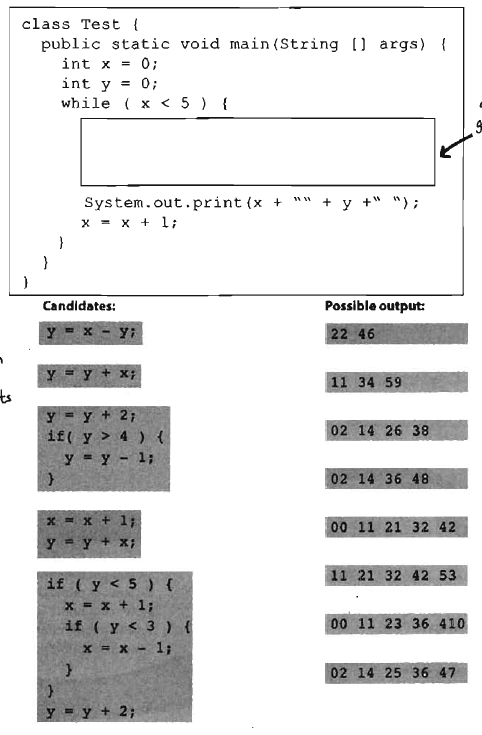
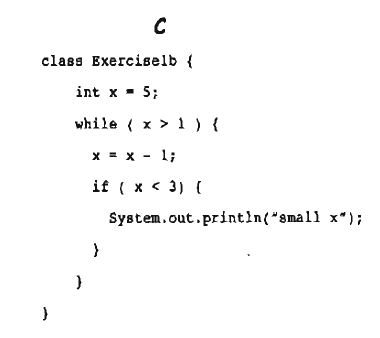
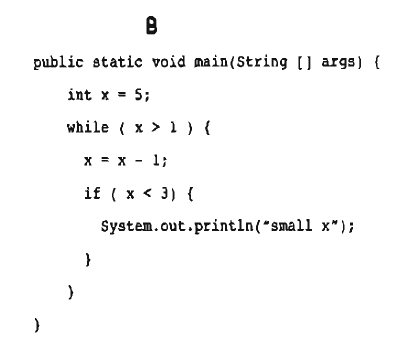
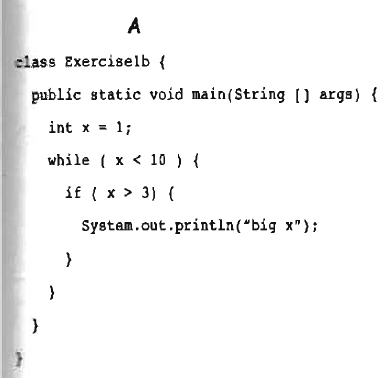
**Java** is object – oriented language, class is blue print for an object and nearly everything in java is an object.  
‘  
**You can have** only one main method that starts the program running even with multiple classes.

**Bullet Points**Statement end in semicolon - **;**  
Code blocks are defined by pair of curly brackets – { }  
Declare an int variable with a name and a type: **int (type) x (name) ;**The assignment operator is one **=**the equals operator uses two **==**  
A while loop runs everything within its block as long as the conditional test is **true**If conditional test is false while loop code block won’t run, and execution will move down.  
Put a Boolean test inside parenthesis, **while (x == 4) { }**

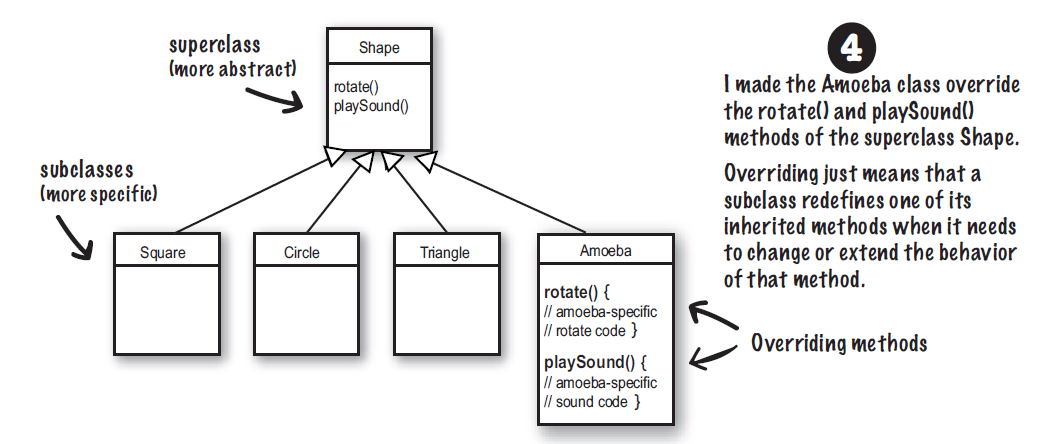
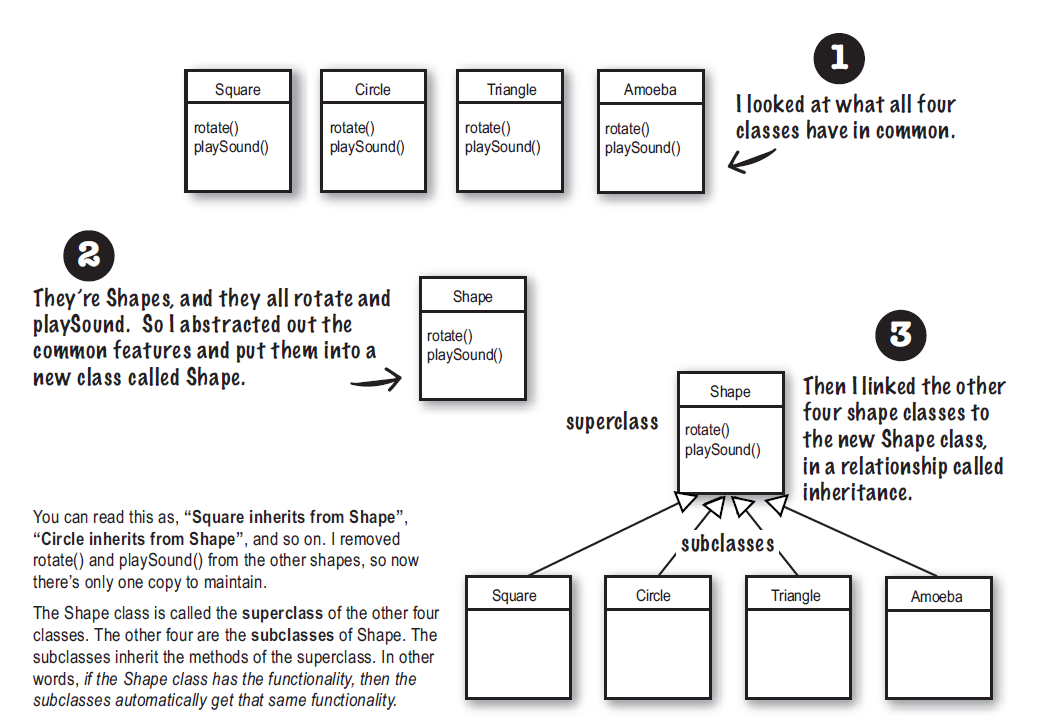
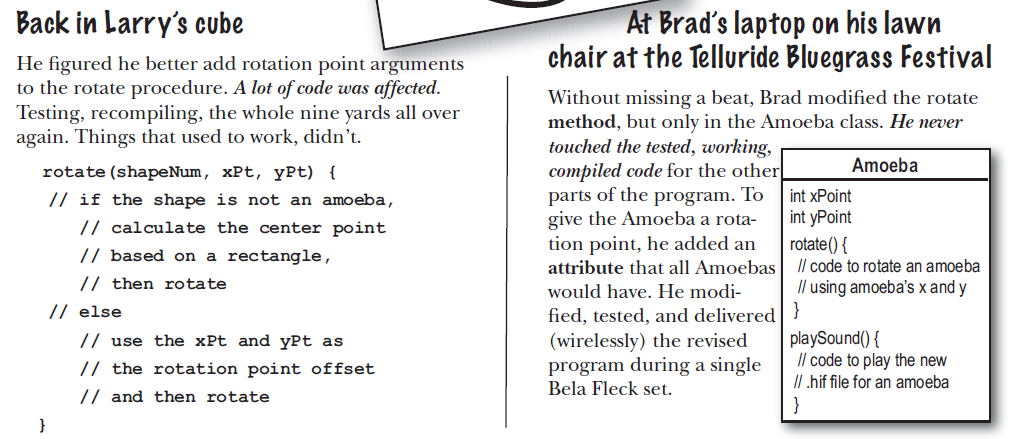
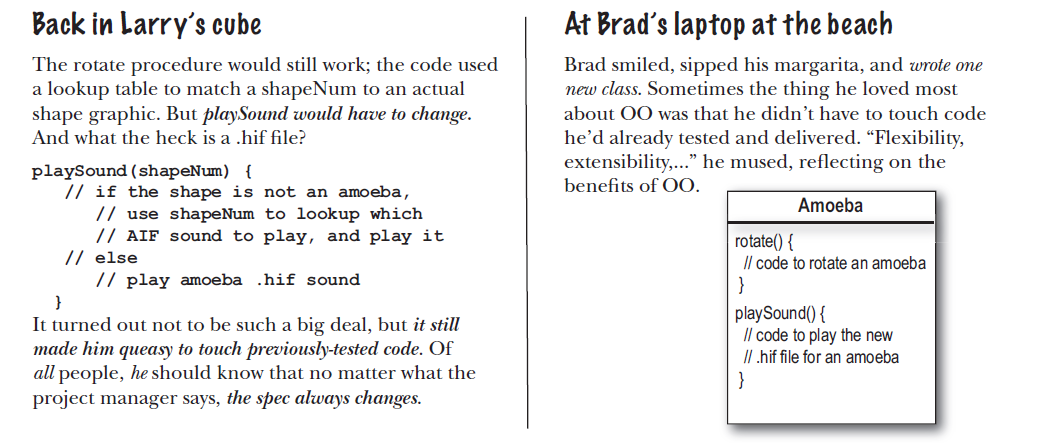
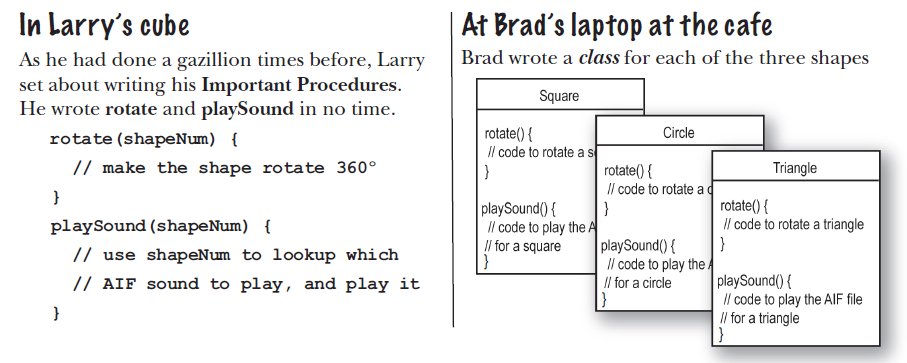
**Conditional Branching***if* test is same as the Boolean test in *while* **loop.**  
Code above executes if the condition is true.  
  
Adding *else* is also possible,  
  
 **System.out.printIn -> Prints a new line  
System.out.print -> Prints on a same line**

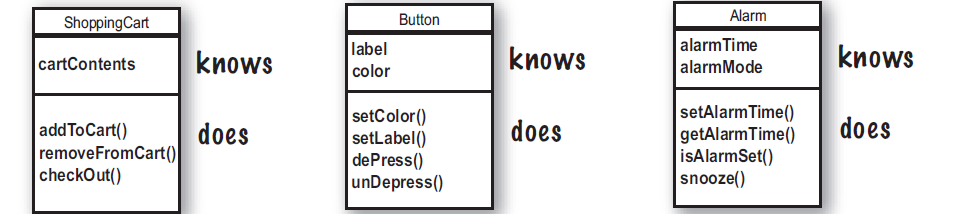
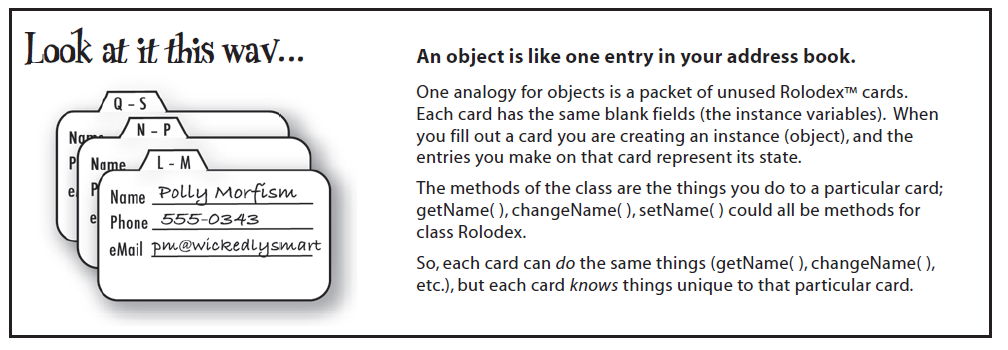
  
The problem is that when **beerNum** is **2**, it runs through the **while** loop, since it is not **2**, it passes through **if (beerNum == 1)**, so it prints the 4 line, **beerNum becomes 1**, but in the **next if (beerNum > 0) line,** the **beerNum is 1**, but the word string is still bottles, so it will print **“1 bottles of beer on the wall”** which is wrong, so the **if (beerNum ==1)** has to be after **beerNum = beerNum – 1;** to fix this issue and print **bottle** when **beerNum is 1.**

**Jini surrogate architecture –** You can still run a non-Java enabled device as if it were a java device by controlling it through some other interface that is running java.  
  
**Phrase – O – Matic**  
1. Creates three String arrays called wordListOne,Two,Three with lists of words.  
  
**Declaring and creating an array** String [ ] pets = { “Fido”, “Zeus”, “Bin” };  
  
2. We need the length of the array to pick out a word from the word list, to do this, we declare int for oneLength, twoLength, threeLength.  
  
**Getting the length of array** int x = pets.length;  
  
3. To randomly pick a word from the array, we need to pick a random number from the length of the array. **Random()** function can be used, the random method returns a random number between 0 and not-quite-1, so array length has to be multiplied with it. To force it into a whole number, we force it into integer.  
  
**Randomly generating an integer from the length of the array** int randx = (int) (Math.random() \* x)  
  
4. Build phrase, use “+” operator which concatenates string objects together, and to pull out the words from the array, you give the randomly generated array index number by **array[#]**.  
  
**Building a phrase** String phrase = “My dog’s name is “ + pets[randx];

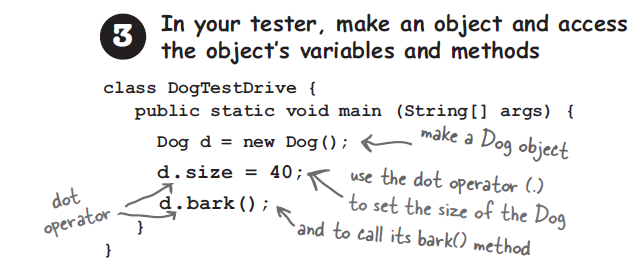
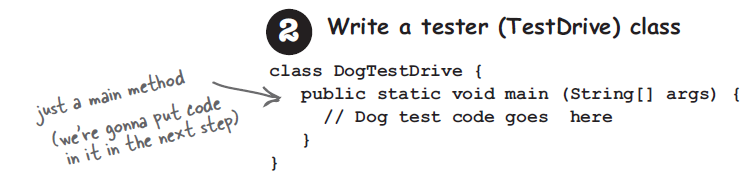
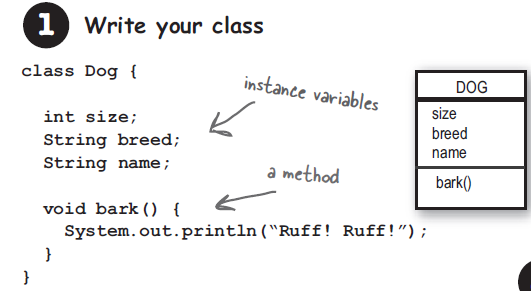
(PDF Page 42-43, difference between JVM & Complier)  
Code magnet – see CodeMagnet.java  
  
**A – Infinite while loop, X = X+1; is needed  
B – There is no class declaration  
C – There is no method declaration**

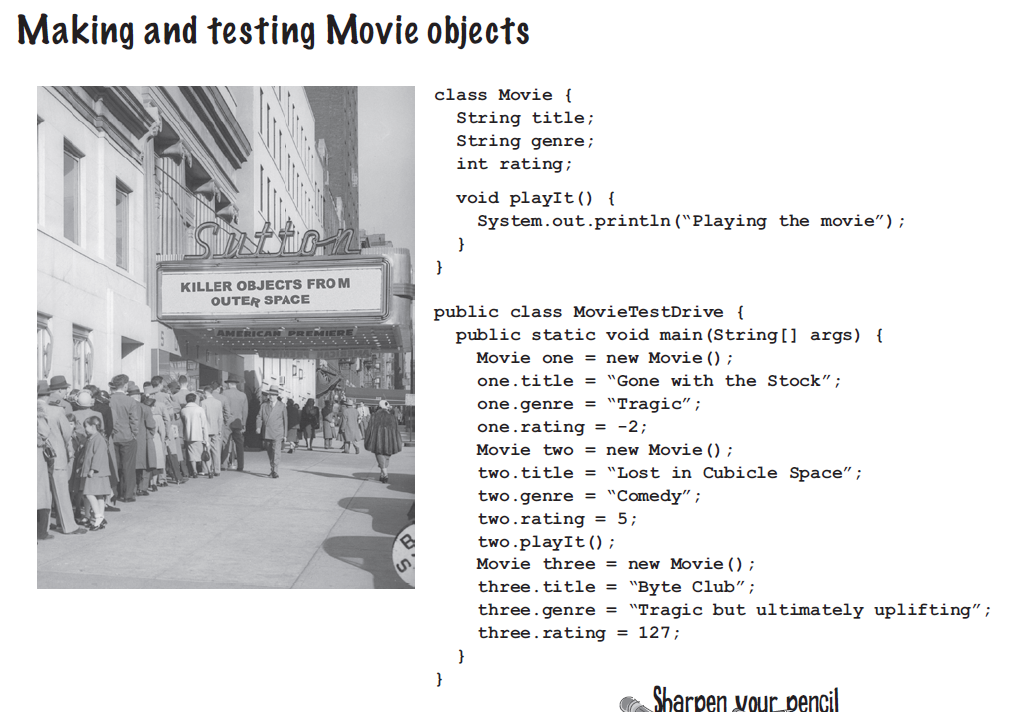
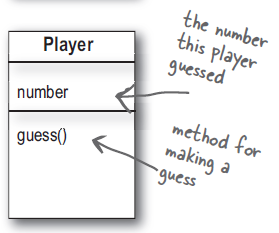
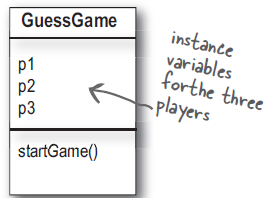
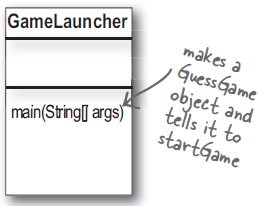
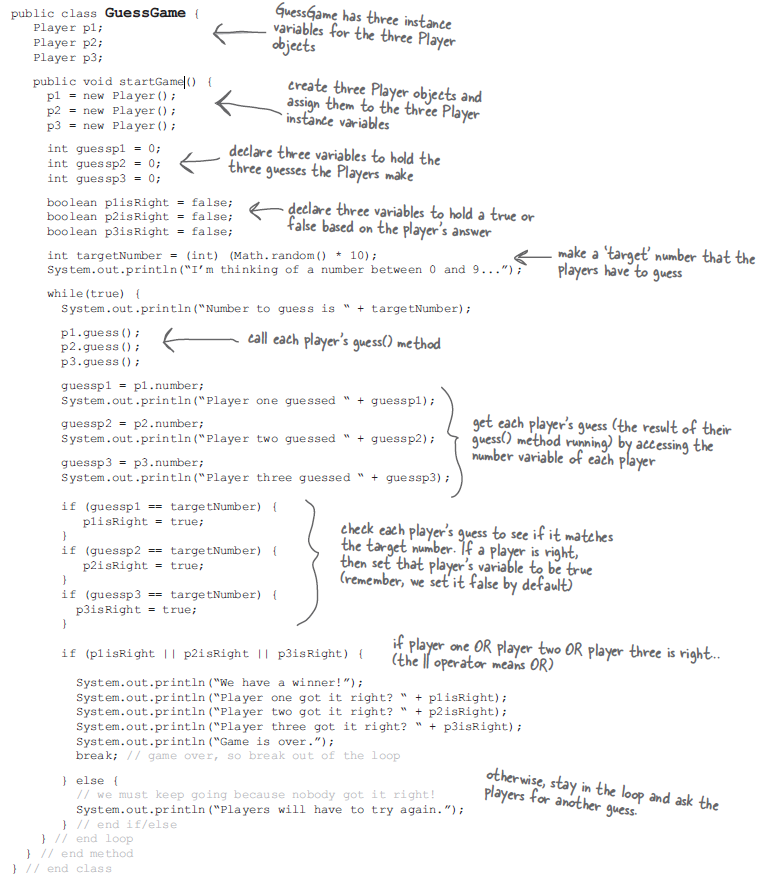
**Candidates -> Outputs  
A -> 00 11 21 32 42  
B -> 00 11 23 36 410  
C -> 02 14 25 36 47  
D -> 11 34 59  
E -> 02 14 36 48**

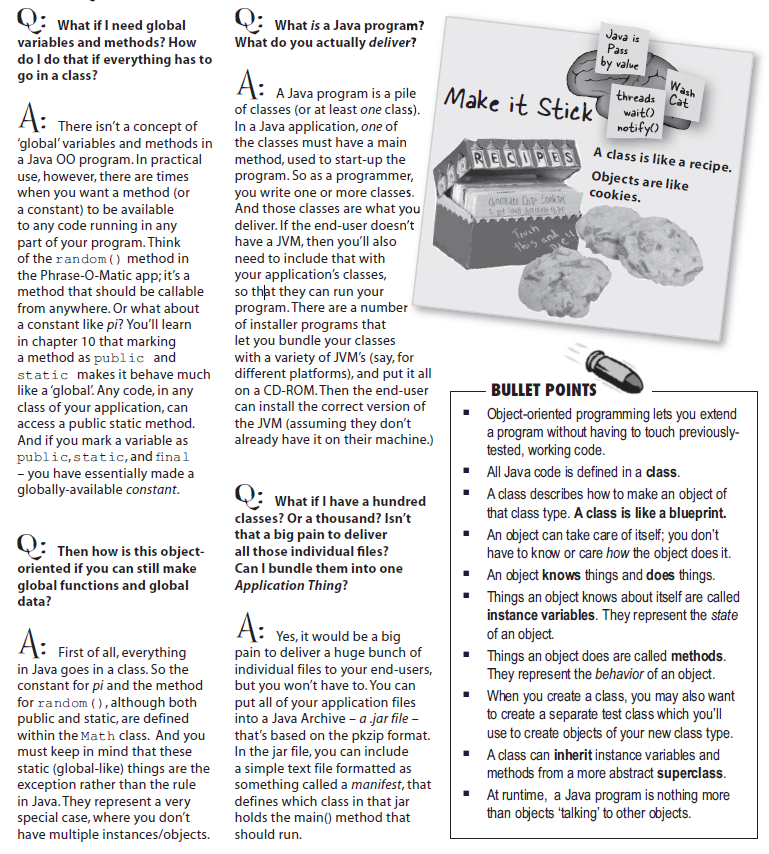
**Chapter 2 – Object Oriented  
Chair Wars**

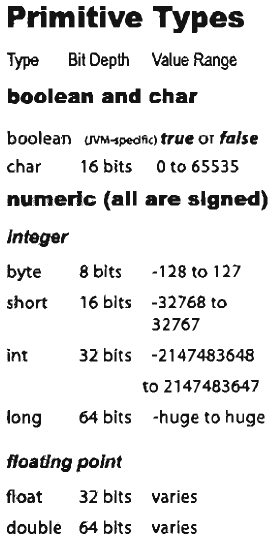
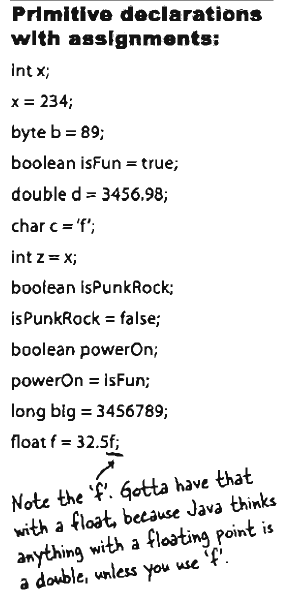
**When designing a class**Things an object knows about itself are called – Instance Variable  
Things an object can do are called – Methods  
  
  
**Difference between class and object**Class is blueprint for an object, it tells JVM how to make an object of that particular type. Each object made from that class has its own values for instance variables of that class.  
  
Button class -> Dozens of different buttons -> Each button have own color, size, shape, label, etc.  
  


**Making your first object**To create and use an object, you need two classes.  
One is for type of object that you want to use, one is a class to test your new class. “Tester” class.  
Tester class is where the main method goes in and the main method accesses the objects.

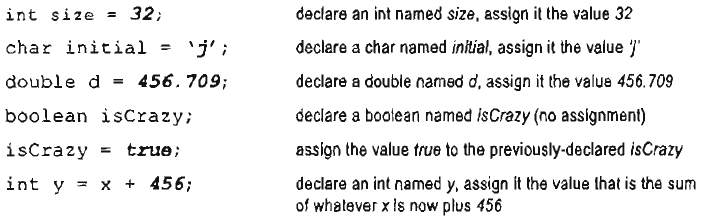


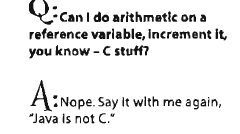
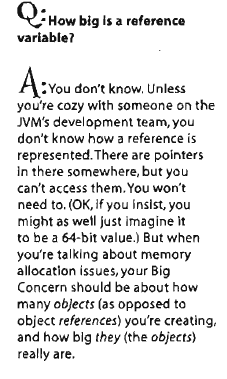
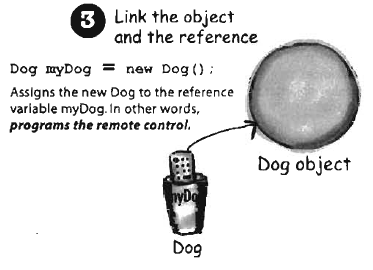
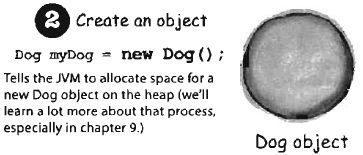
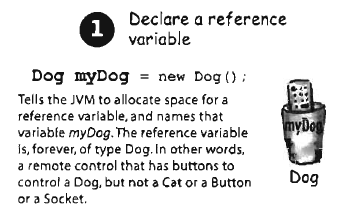
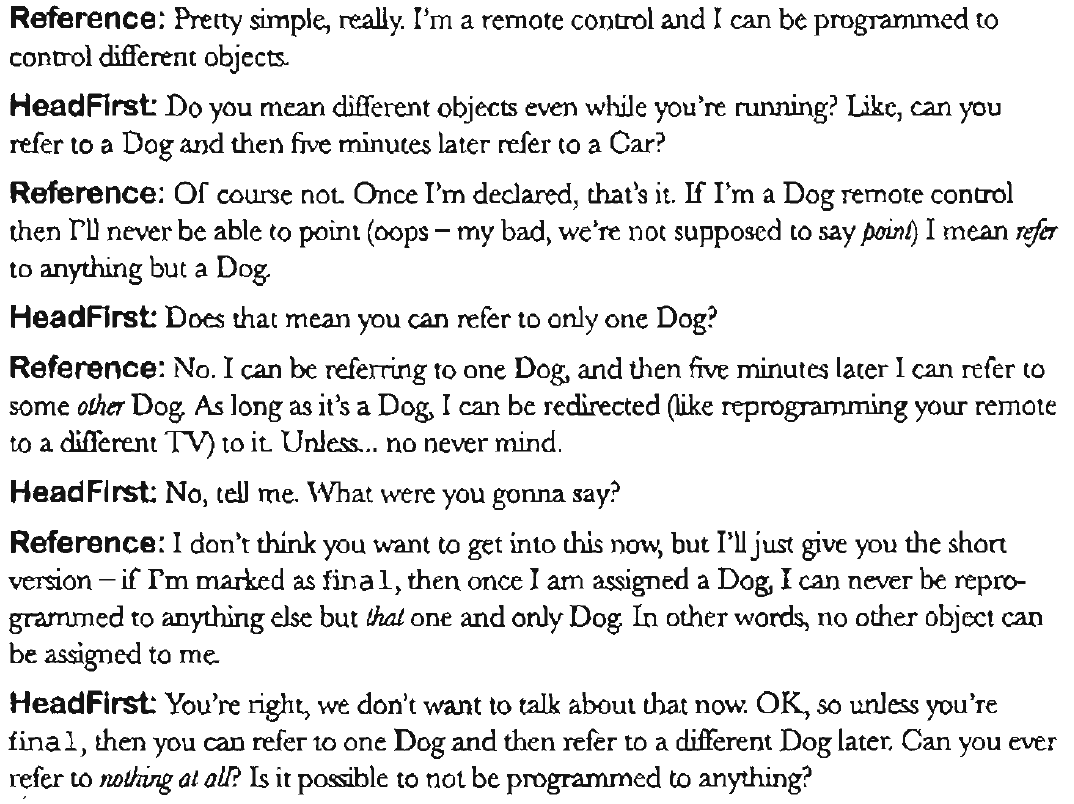
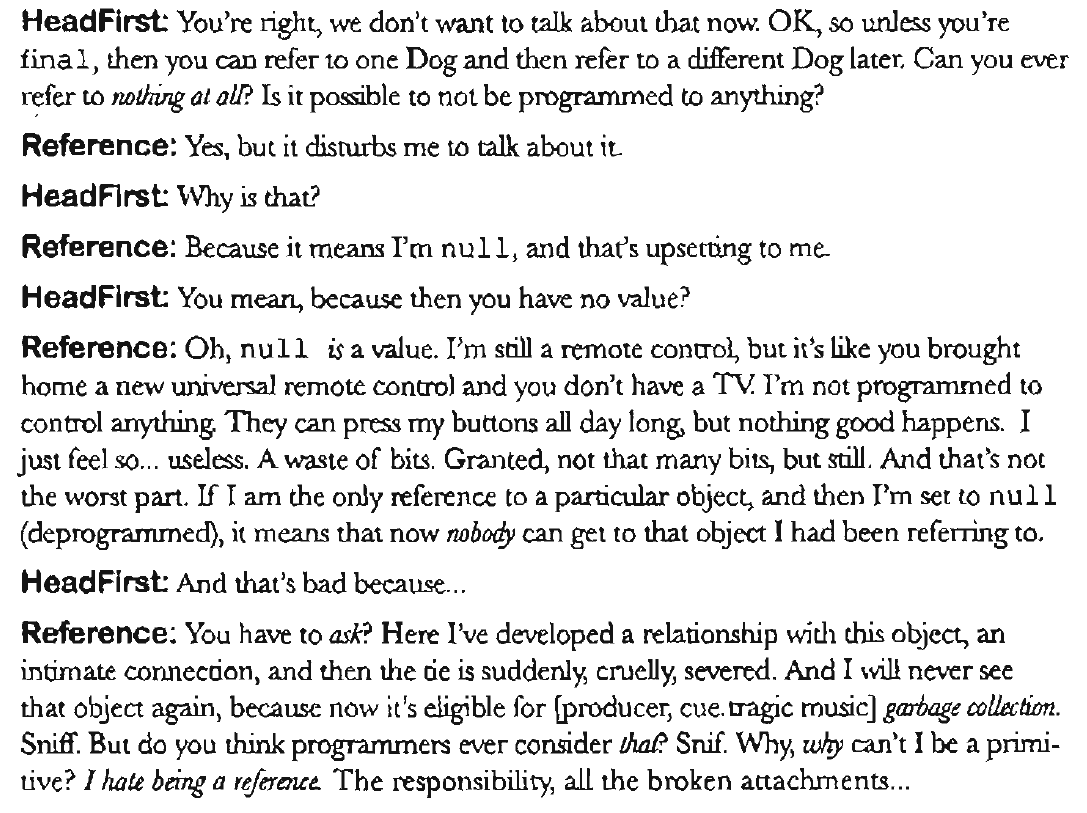
**The Dot Operator (.) –** gives access to an object’s state and behavior (instance variable and method)  
  
  
  
**Two uses of main:**Test your real class & Launch your Java application.  
A Java application is objects calling methods on one another  
  
**The Guessing Game (Sneak view of Java application)**“game” object and 3 “player” object. The game randomly generates a number between 0 to 9 and the three player objects try to guess it.  
  
The GameLauncher class starts application, it has the main () method.  
In the main () method, GuessGame object is created and its startGame () method is called.  
The GuessGame object’s startGame() is where the entire game is, creates 3 player and creates an target number, asks each player to guess, check results and prints out the result.  
  

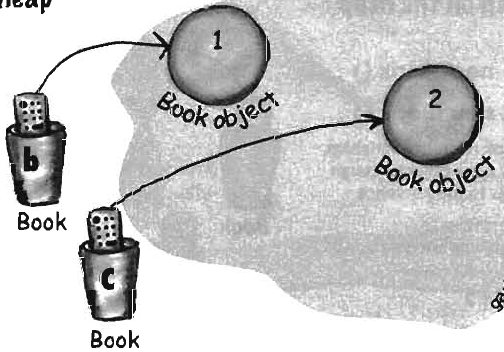



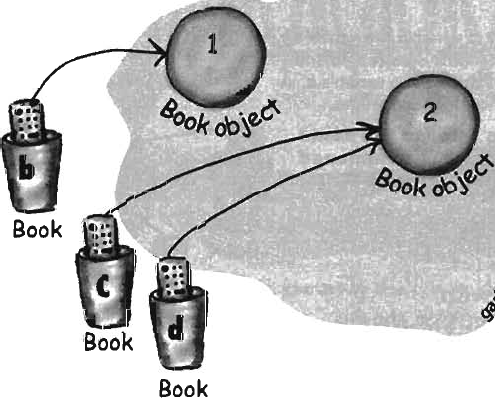
**Chapter 3 – Variables  
Declaring a variable**Java cares about type, ex) would not allow floating point number into an integer variable, unless it is acknowledged to the compiler that you might lose “precision”  
Declaration for the type of variable is must and variable comes in two.  
  
**Primitive and Object reference  
Primitive** – hold fundamental values (integer, Boolean, floating point numbers)  
**Object references –** hold references to objects.  
  
**Two Declaration Rules**Variable must have type and name. (int is type, count is name: **int count;**)  
Variable is a “container”, it holds something.  
  
**Primitives**When declaring variable in Java, you must declare it with a specific type.  
Each primitive variable has a fixed number of bits (“cup size”). The size for the six numeric primitives in Java are  
**byte – 8  
short – 16  
int – 32  
long – 64  
float – 32  
double – 64**  
  
  
  
  
  
  
  
  
  
  
  
Large value can go into a small “cup”, but there will be a spillage… complier warns if some value is not going to fit in the container (variable/cup) that is being used.  
  
**int x = 24;**

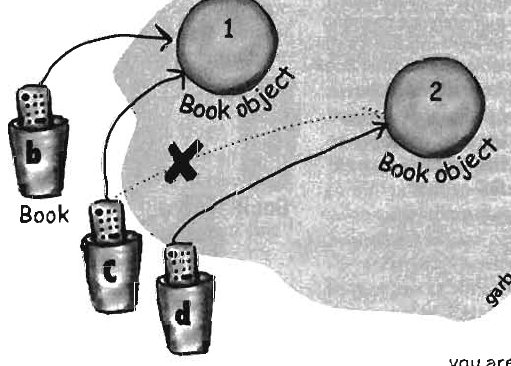
**Byte b = x;**

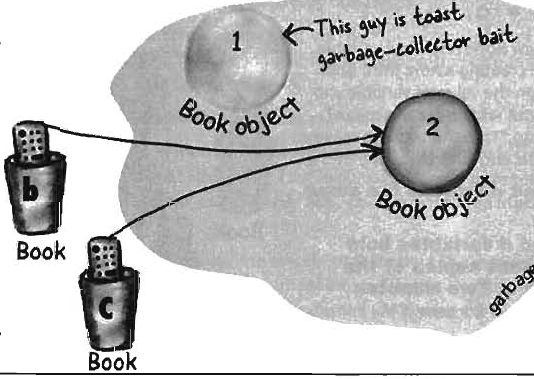
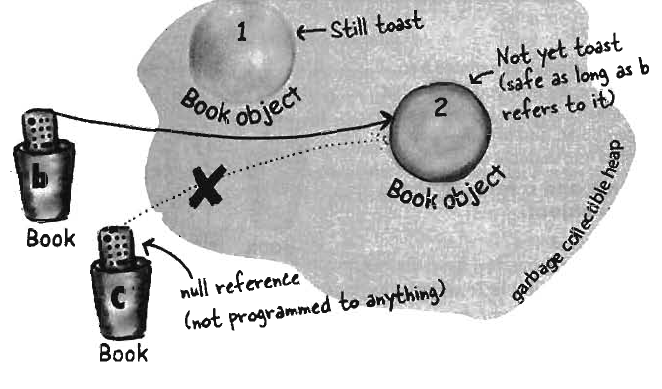
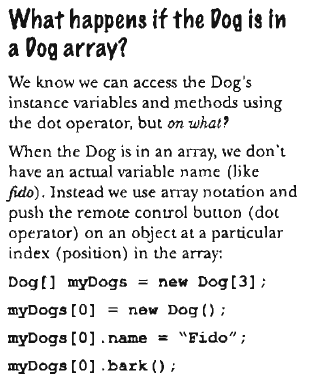
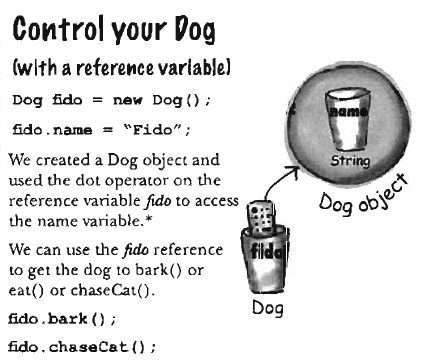
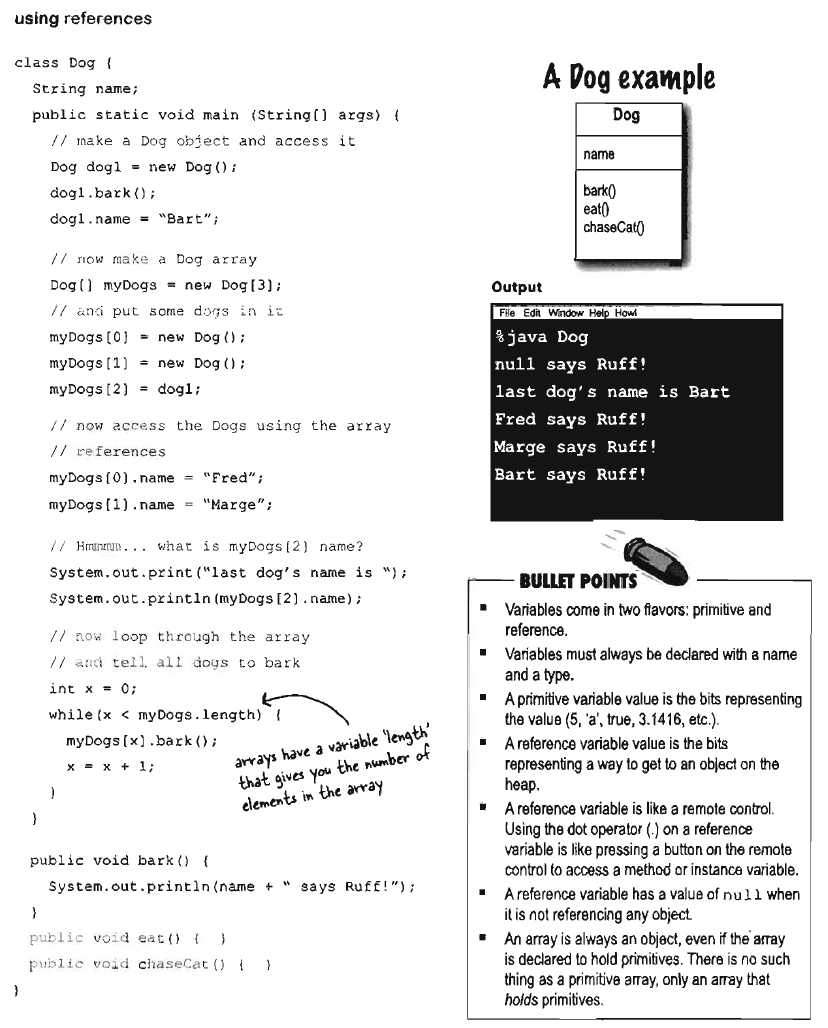
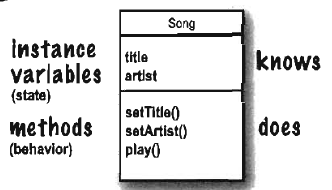
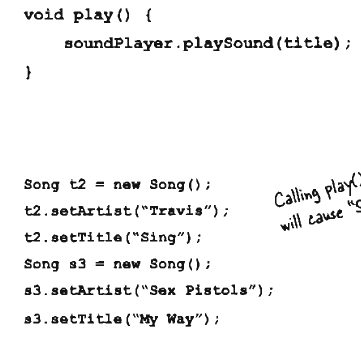
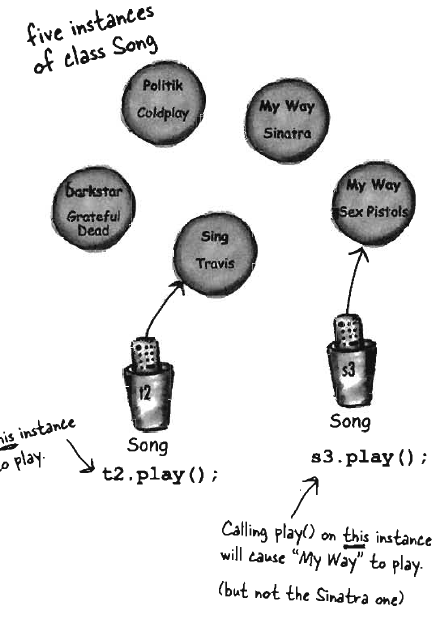
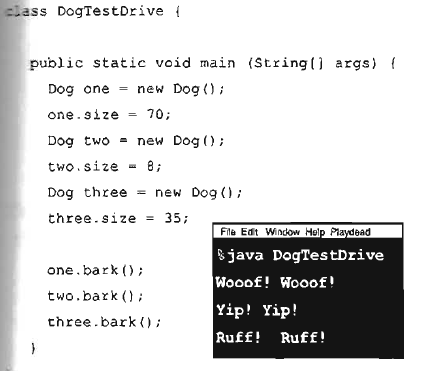
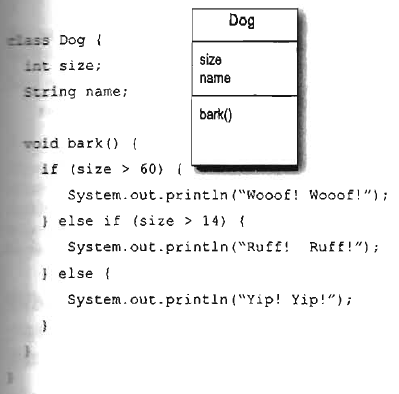
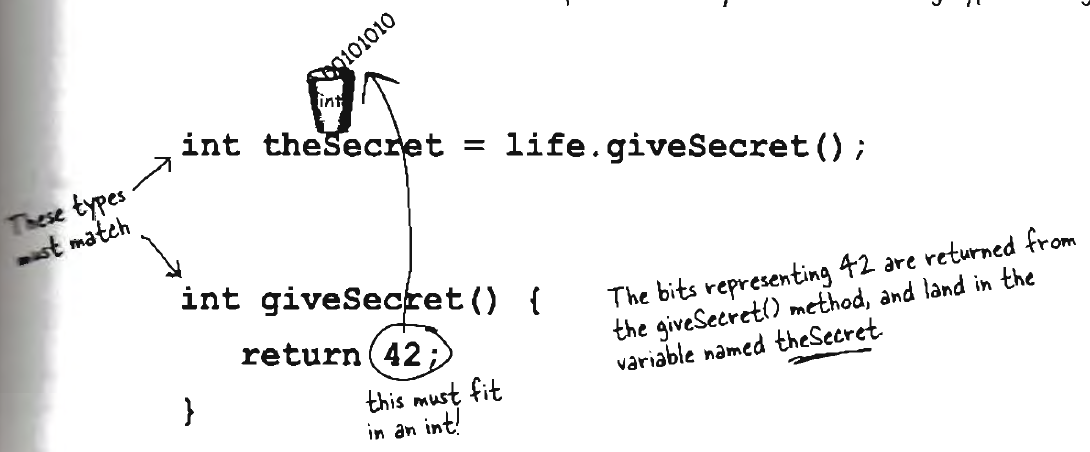
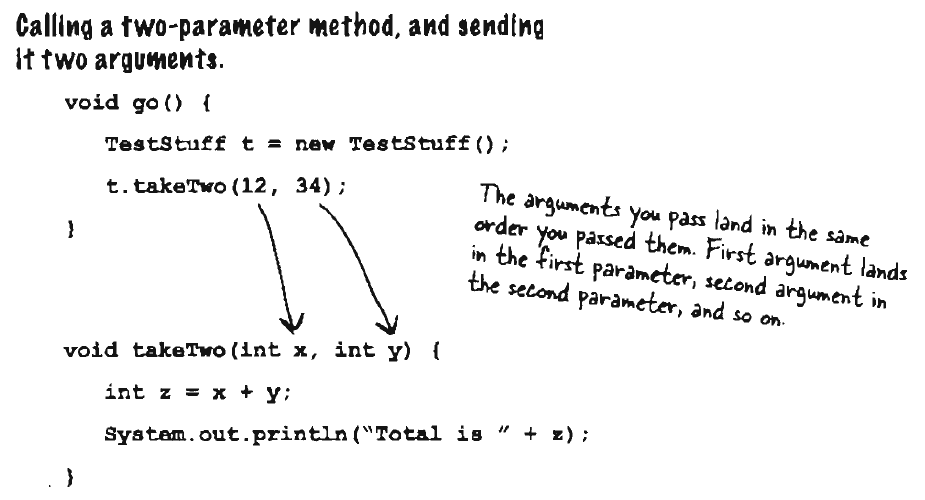
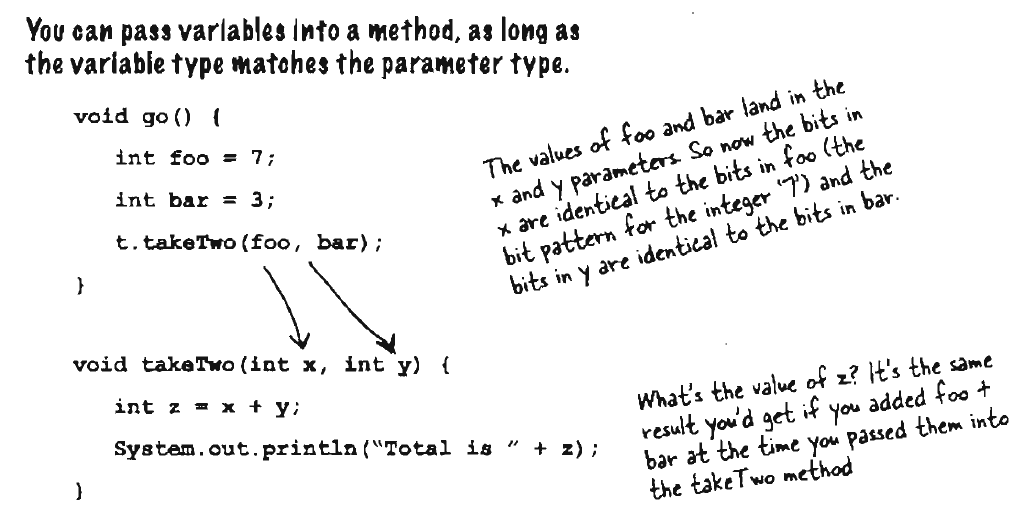
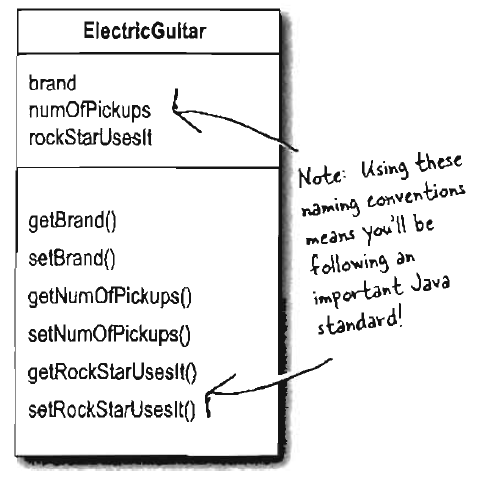
This would not work, 24 can go into byte, but since there is possibility of spilling when putting integer into byte, this would not work.  
  
The opposite, putting small type into larger type can work.  
  
  
**Assignment**You can type literal value after equals sign ( x = 12 )  
Assign the value of one variable to another ( x = y )  
Use an expression combining the two ( x = y + 43 )  
  
**Naming**Naming class, method, variable must start with a letter, underscore, or dollar sign.  
Numbers cannot start, but can be added.  
Java reserved words cannot be the names. (See PDF page 76)  
  
**Object Reference Variable (**Non-primitive variables)  
Object reference variable holds bits that represent a way to access an object.  
It doesn’t hold the object, but it holds a pointer… address. We don’t know what is inside a reference variable, we do know that whatever it is, it represents one object and the JVM knows how to use the reference to get to the object.  
  
Primitive variable is full of bits representing the actual value of the variable  
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 this before the dot to get me the thing after the dot.”  
  
**myDog.bark ();**use the object referenced by the variable myDog to invoke the bark () method.  
When dot operator is used, think of it like pressing a button on remote control for that object.  
  
**Object reference is another variable value**Primitive variable -> byte x = 7; the bits representing 7 goes into the variable.  
Reference variable -> Dog myDog = new Dog ();, bits representing ways to get the the Dog object.  
The dog object itself does not go inside the variable.

**Garbage-collectible heap  
  
Book b = new Book ( ) ;  
Book c = new Book ( ) ;**

(Two Book reference variable declared,  
created two new Book objects, Assigned  
Book objects to the reference variables)  
(2 Reference, 2 Objects)  
  
  
**Book d = c ;**

(Declare new book reference variable,  
assign value of c to variable d. Both  
c and d refer to the same object, the  
c and d variable holds two different copy  
of the same value, two remotes on one TV)  
(3 Reference, 2 Objects)  
  
**c = b ;**(Assigns value of variable b to variable c,  
bits inside variable b are copied, new copy is  
stuffed into variable c. Both b and c refer  
to the same object)

  
  
  
  
**Life and death on the heap  
  
b = c;**  
(Assigns 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 hold  
identical values, b and c refer to the same object  
Object 1 is abandoned and is unreachable.  
  
**c = null;**Assign value null to variable c, c does not refer  
to anything, but it is still a reference variable and another  
book object can be assigned to it, object 2 still has  
active reference, not eligible for GC.   
  
**Array**1. Declare an int array variable, **int [ ] nums ;**2. Create new int array with length of 7, assign it to previously declared int [ ] cariable nums  
**nums = new int [7] ;**3. Give each element in the array in int values, elements in int array are just int variables.  
**nums[0] = 6;, nums[1] = 19;, nums [2] = 44; …. Etc**Arrays are great when you want a quick, ordered list of things. Array gives fast random access by letting you use index position to get any element in an array.  
  
Every element in array is just a variable 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 array of type int, each element can hold an int. In Dog array, each element can hold a remote control to a dog, a reference. Array is an object.  
  
Arrays are always object, whether they are declared to hold primitives or object reference. But 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. Array itself is always an object.  
  
**Array of Dogs**1. Declare Dog array variable **Dog [ ] pets;**2. Create new Dog array with length 7, assign it to previously-declared Dog [] variable pets  
**pets = new Dog[7];**3. Create dog objects to assign them to array elements which are reference variables.  
**pets[0] = new Dog();, pets [1] = new Dog();, pets [2] = new Dog();**  
  
  
  
  
**Chapter 4 – How objects behave**A Class is blue print for an object, when writing a class, you are 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.  
  
Every instance of particular class has the same methods, but the methods can behave differently based on the values 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 presented by the value of the title instance variable for that instance. So, if you call play () method on one instance, you will hear the song “Politik” while another instance plays “Darkstar”, the method code is same.  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
**The size affects the bark (Value of I.V affect method behavior)**You can pass values into your methods, tell Dog object how many times to bark by calling **d.bark(3);**A method uses parameters, a caller passes arguments.  
Arguments are things you pass into the methods.  
**Argument (**Value like 2, “Foo”, or reference to a Dog) lands into a parameter.  
**Parameter** is local variable, variable with type and name that can be used inside the body of method.  
**If method takes a parameter, you must pass it something,** value of the appropriate type.  
  
  
**Methods can return values**.  
Every method is declared with a return type, but **void** return type does not give anything back.  
  
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 compatible).  
  
  
  
  
  
  
  
  
**Java is pass by value, this means pass by copy.  
  
int x = 7;** Declare int variable with value 7.  
**void go (int z) { }** Declare method with int parameter named z.  
**foo.go(x);** Call go() method, passing variable x as argument. 7 is coped into z.  
**void go (int z) { z = 0; }** Value of z is changed inside method, value of x is still 7, while z is 0.  
  
**Method can declare only one return value,** but if you need to return three int values, then the declared return type can be an int array.  
**Java doesn’t require you to acknowledge a return value**, you may want to call a method wit non-void return type, even though you don’t care about the return value. In this case, you are 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 values.  
  
**Bullet Points  
Class** defines what an object knows and what an object does.  
Things an object knows are **instance variables.**  
Things an object does are its **methods.**  
**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 into methods.  
**The number and type of values** you pass must match the order and the type of the parameter.  
**Value passed in and out of method** can be implicitly promoted to larger type or explicitly cast to smaller.  
**The value you pass as an argument** to method can be literal value or a variable of the declared parameter type.  
**A method** must declare a return type, a void return means the method doesn’t return anything.  
**If method** declares non-void return type, it must return a value compatible with the declared return type.  
  
  
  
**Parameters and return types**Getters and Setter lets you get and set, usually, instance variable values.  
Getter’s purpose is to send back as a return value, value whatever it is that particular getter is supposed to be getting.  
Setter takes an argument value and uses it to set the value of an instance variable.  
  
**class ElectricGuitar {  
 String brand;  
 int numOfPickups;  
 Boolean rockStarUsesIt;**

**String getBrand ( ) {  
 return brand;  
 }**

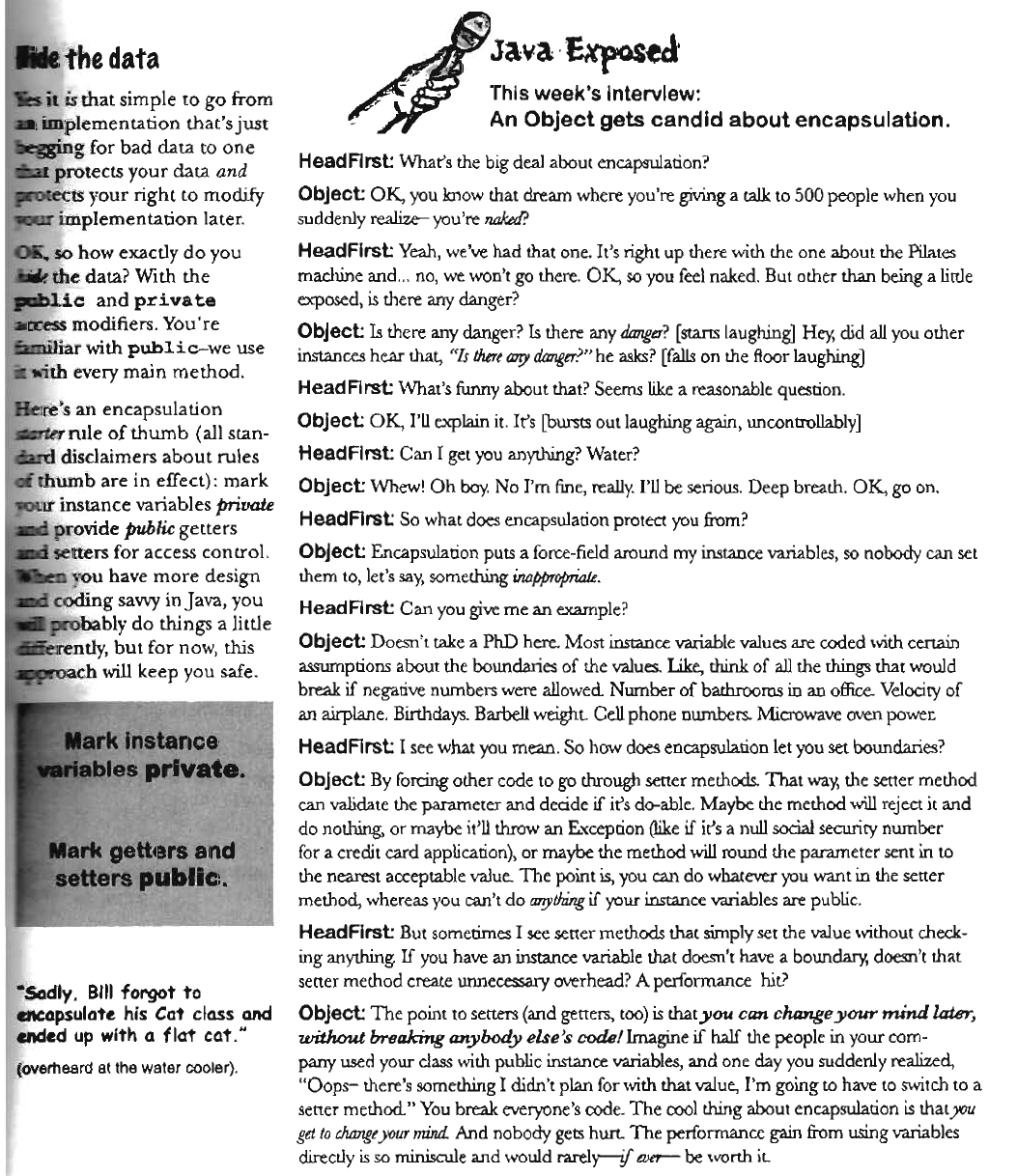
**Void setBrand (String aBrand) {  
 brand = aBrand;  
 }**

**Int getNumOfPickups ( ) {  
 return numOfPickups;  
 }**

**Void setNumOfPickups (int num) {  
 numOfPickups = num;  
 }**

**Boolean getRockStarUsesIt ( ) {  
 return rockStarUsesIt;  
 }**

**Void setRockStarUsesIt (Boolean yesOrNo) {  
 rockStarUsesIt = yesOrNo;  
 }**

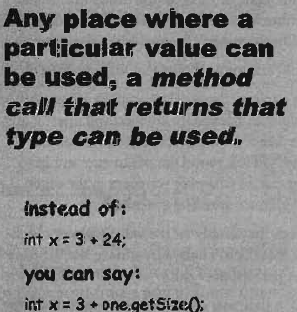
**}**  
**Encapsulation**Our data is exposed, exposed means reachable with the dot operator  
theCat.height = 27; Reference variable can be changed by anyone..  
This is bad, we need to build setter methods for all the instance variables and find a way to force other code to call the setters rather than access the data directly.  
  
**public void setHeight (int ht) {  
 if (ht > 9) {  
 height = ht;  
 }  
}**  
This puts in a check to guarantee a minimum height and by forcing everybody to call a setter method, it is protected.  
  
**Encapsulating GoodDog Class** class GoodDog {

private int size;

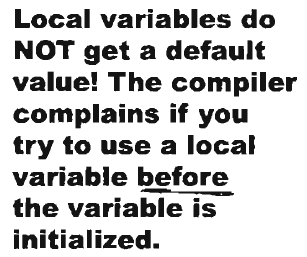
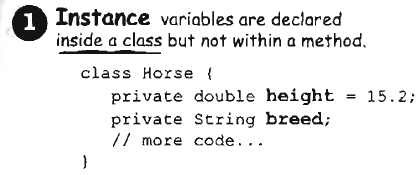
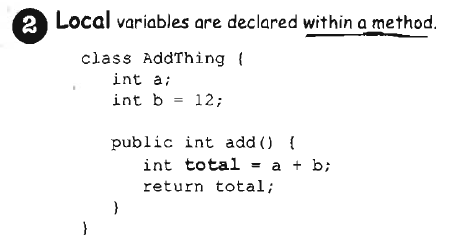
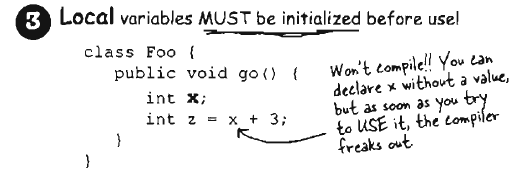
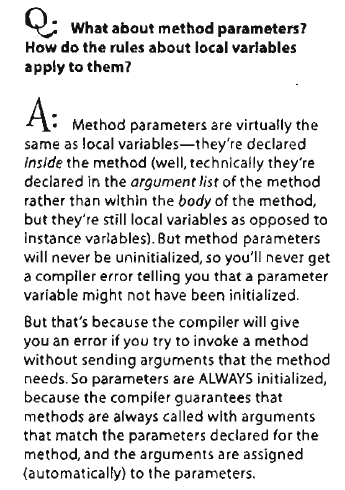
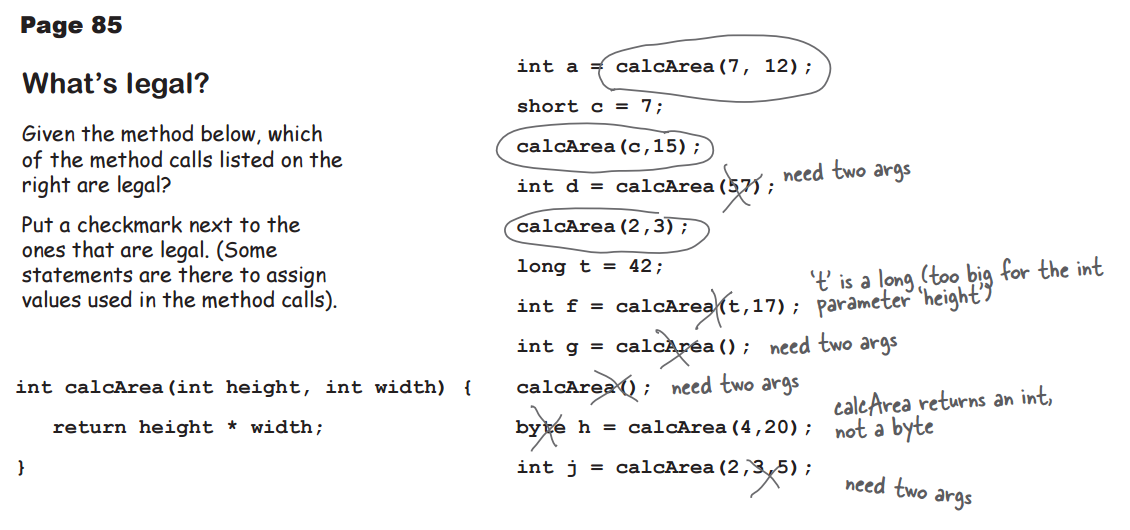
public int getSize ( ) {  
 return size;  
 }

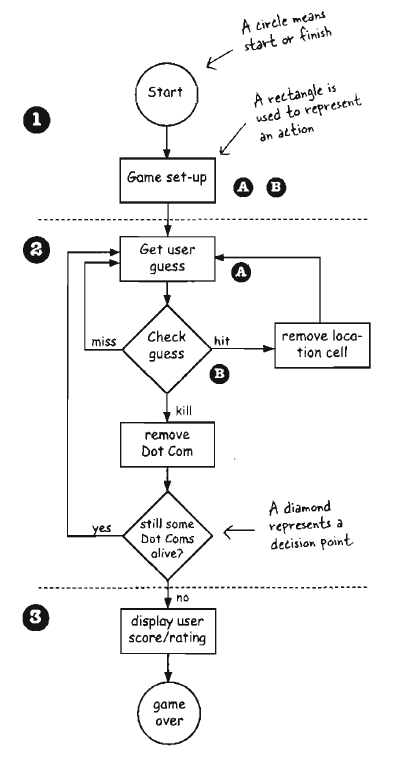
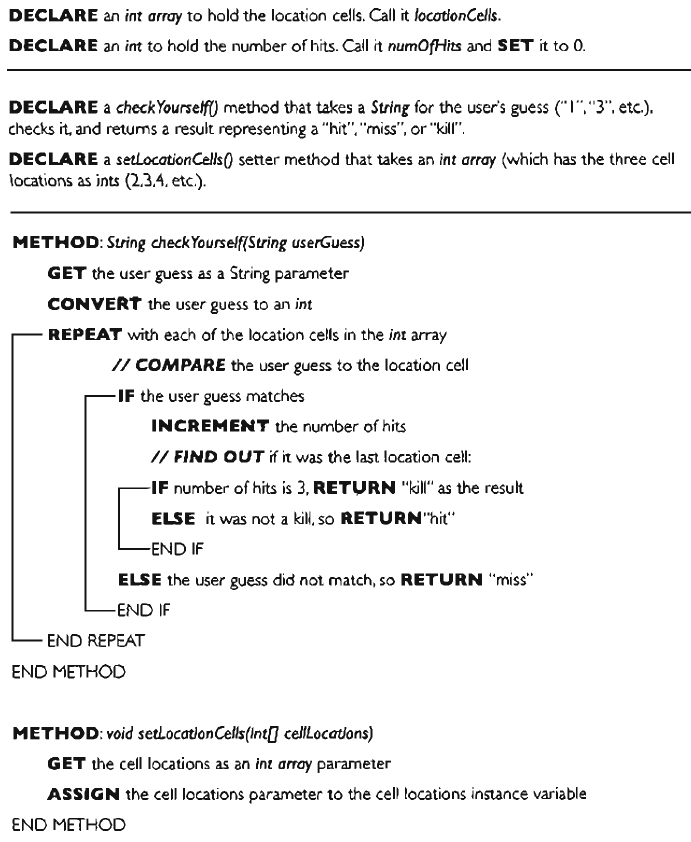
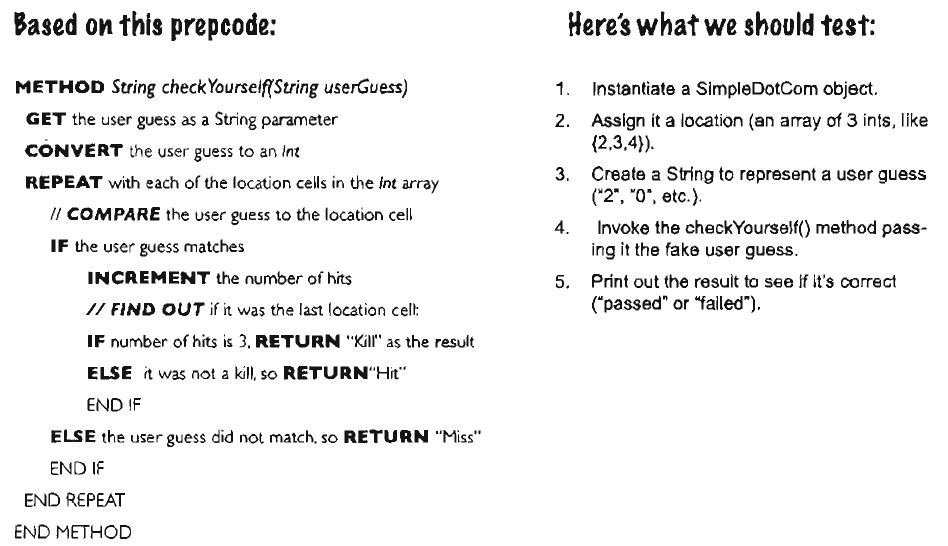
public void setSize(int s) {  
 size = s;  
 }

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!”);  
 }  
 }  
 }  
  
**The instance variable is private, getter and setter methods are public.**class GoodDogTestDrive {

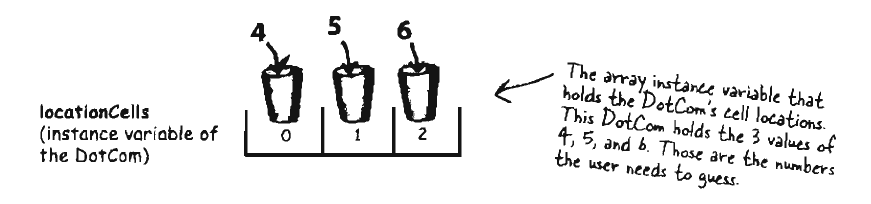
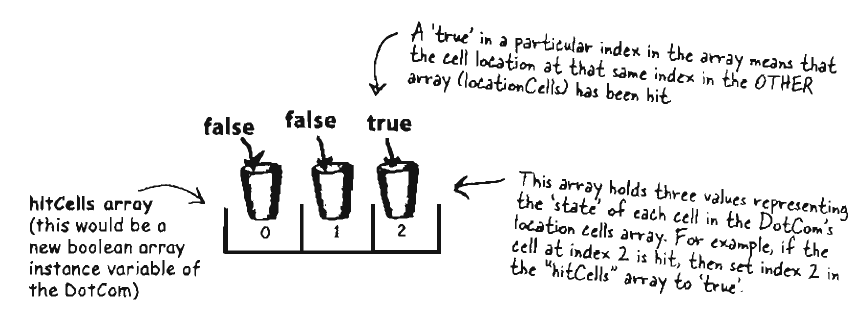
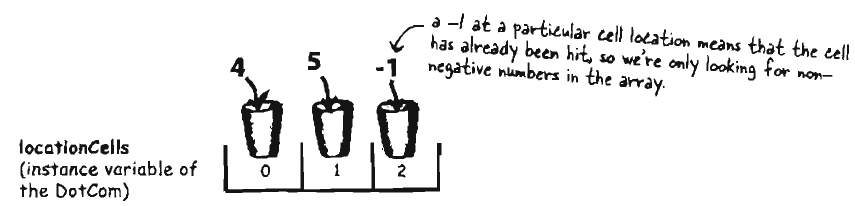
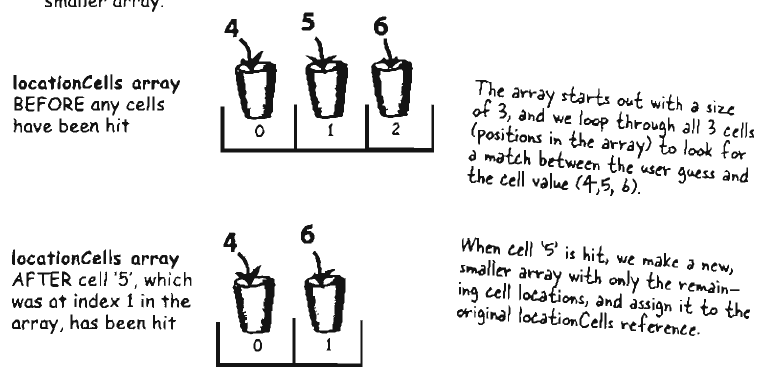
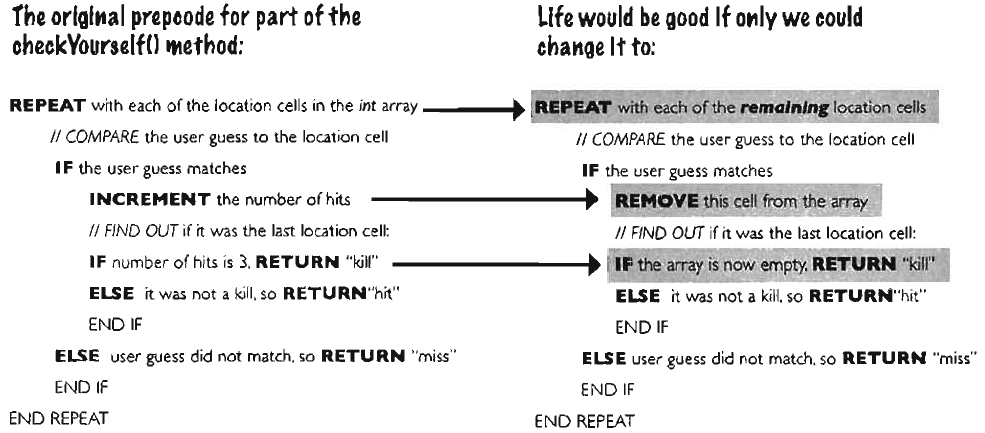
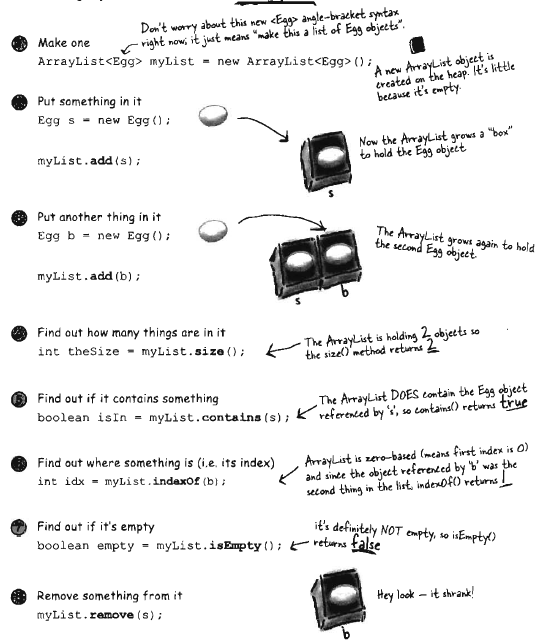
 public static void main (String[] args){  
 GoodDog one = new GoodDog ( );  
 one.setSize(70);  
 GoodDog two = new GoodDog ( );  
 two.setSize(8);  
 System.out.println(“Dog one: “ + one.getSize());  
 System.out.println(‘Dog two: “ + two.getSize());  
 one.bark();  
 two.bark();  
 }  
}

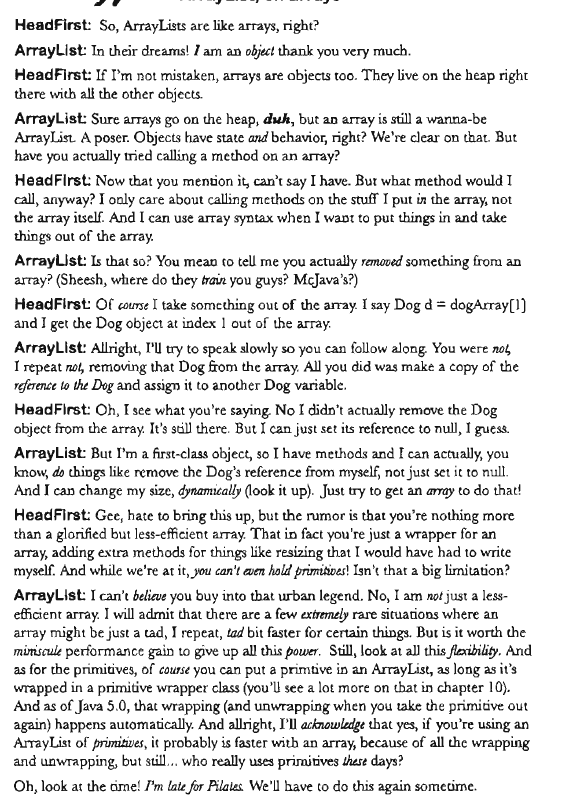
**Output**Dog one: 70  
Dog two: 8  
Woof Woof  
Yip Yip **Remember, there has to be 2 classes, so 2 .java files.  
  
  
  
Arrays**Declare a Dog array that has the length of 7.  
**Dog [ ] pets;  
pets = new Dog [7] ;**Create two new Dog objects  
**pets[0] = new Dog () ;  
pets[1] = new Dog () ;**Call method  
**pets[0].setSize(30);  
int x = pets[0].getSize();**  (So x will be 30)  
**pets[1].setSize(8);  
  
  
  
Declaring and initializing instance variables**A variable declaration needs name and type.  
If you don’t initialize an instance variable, the value is defaulted, meaning that  
**0 for int, 0.0 float, false Boolean, null references**  
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());  
 }

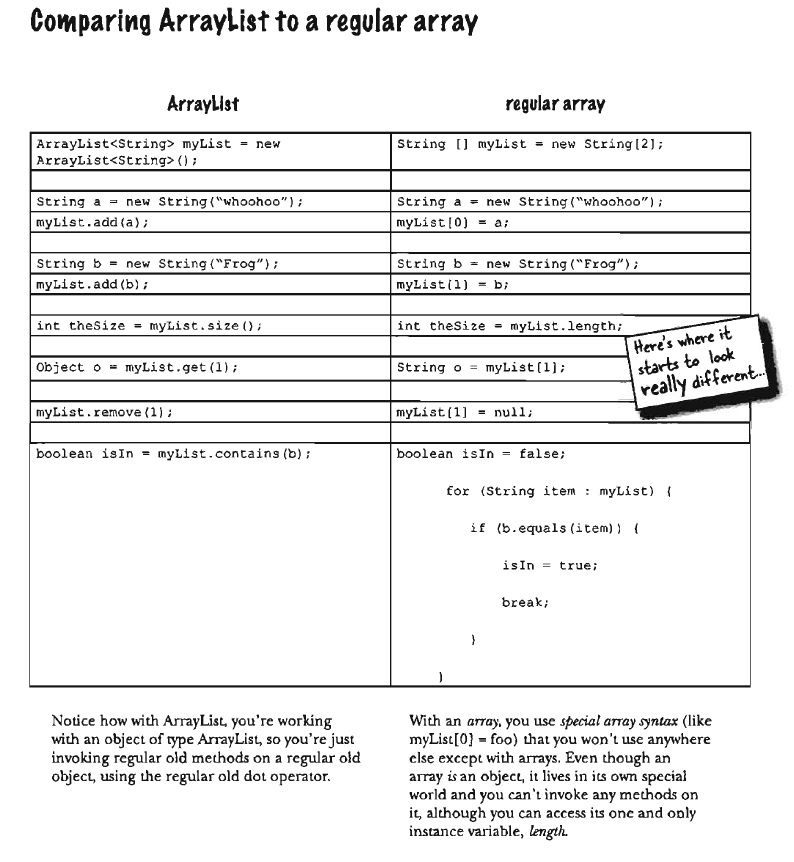
}  
  
**Output**Dog size is 0  
Dog name is null  
  
  
  
  
**Difference between instance and local variables.**      
**To compare primitives or references, use == operator.**  
int a = 3;  
byte b = 3;  
if (a == b) { // True }  
  
**Comparing references**Foo a = new Foo();  
Foo b = new Foo();  
Foo c = a;  
if (a == b) { // False}  
if (a == c) { // true}  
if (b == c) { // false}  
  
**It looks at the big pattern and returns true or false, so int == byte work for that example.**

**Chapter 5 – Extra Strength Methods  
‘**Building a battleship-style game, “Sink a Dot Com” – Virtual 7 x 7 grid.  
  
**Flow of the game**1. User starts the game (Game creates three Dot Coms and places three Dot Coms onto a virtual grid)  
2. Game play begins (Repeat Prommpt user for a guess and check user guess against all Dot Coms to look for a hit, miss, or kill. If hit, delete cell. If kill, delete Dot Com)  
3. Game finishes (Give user a rating based on the number of guesses)  
  
**Simplified Dot Com Game**Single row, one Dot Coms, but the goal is the  
same. Assign it a location somewhere in the  
row, get user input, when all has been hit, the  
game is over.  
  
In this simple version, the game class has no  
instance variable and all the game code is in   
the main () method. When the program is  
launched and main () begins to run, it will  
make the one and only DotCom instance,  
pick a location for it, ask user for guess,  
check the guess, and repeat until all  
three cells have been hit.  
  
Virtual row is virtual, as long as the game  
and the user know that the DotCom is   
hidden in three consecutive cells out of   
seven, the row itself doesn’t have to  
be represented in the code.  
  
We don’t need 7 array with 3 of them for   
DotCom, we just need array that holds   
the 3 DotCom cells.  
  
  
  
  
  
  
  
  
  
  
  
  
**Developing a class**Figure out what class is supposed to do  
List the instance variables and methods  
Write prepcode for the methods  
Write test code for the methodds  
Implement the test  
Test the methods  
Debug & reimplement   
  
**Prep code –** Form of pseudocode, focus on logic without stress on syntax  
**Test code –** Class or methods that will test the real code and validate that its doing the right thing  
**Real code –** Actual implementation of the class.  
‘  
**Prep code of Simple DotCom Class**  
**Extreme Programming (XP)**Based on design, small but frequent releases. Develop in iteration cycles. Don’t pout anything that’s not in the spec. Write the test code first, work regular hours.  
Refactor whenever and whatever you notice the opportunity, don’t release anything until it passes all the tests. Set realistic schedules, based on small releases, simple, program in pairs and move people around so that everybody knows pretty much everything about the code.  
  
**Writing prep code for the Simple DotCom class**

**The test code** is written, not ran. It will not compile, but the reason for writing the test code is to clarify your thoughts about what the method itself needs to do.  
  
As soon as implementation code is done, you already have the test code to validate.  
  
Write a little test code, write only the implementation code you need to pass the test. Each iteration, you run all the previously written tests, so that you always prove that your latest code additions don’t break previosuly – tested code.

Rest of notes were printed directly from the book  
  
  
  
**Chapter 6 – Java Library**The problem with simpledotcom from chapter 5 is that we count a hit every time the user guesses a cel location, regardless of whether that location has already been hit or not. This makes the “bug”.  
  
The DotCom has an instance variable (int array) that holds that DotCom object’s cell locations.  
  
  
**Option 1**Make second boolean array called **hitCells array**, each time a user makes a hit, we store that hit in the second array and then check that array each time we get a hit to see if that cell has been hit before.  
  
  
  
  
  
**Option 2**Keep original array, but change the value of any hit cells to -1, This does not create any additional array.  
  
  
**Option 3**Delete each cell location as it gets hit and modify the array to be smaller, but since arrays cannot change length, we have to make new array and copy the remaining cells from the old array into the new smaller array.  
  
  
  
**Option 1 –** Each time user makes hit, you have to change state of boolean array but first check the boolean array to see if that cell has already been hit…  
  
**Option 2 –** Less clunky, you still have to loop through all three slots in array even if one or more are already invalid because they have been hit.  
  
**Option 3 –** No need to make new array, copy remaning value and reassign reference. Best method.  
  
  
One of the compiled class in Java library (API) is **ArrayList  
  
Arraylist**add(Object elem) – Adds object parameter to the list  
remove(int index) – Removes the object at the index parameter  
remove(Object elem) – Removes this object (if it’s in the ArrayList)  
contains(Object elem) – Returns true if there’s a match for the object parameter  
IsEmpty() – Returns true if the list has no elements  
indexOf(Object elem) – Returns either the index of the object parameter or -1  
size() – Returns the number of elements currently in the list  
get(int Index) – Returns object currently at the index parameter.  


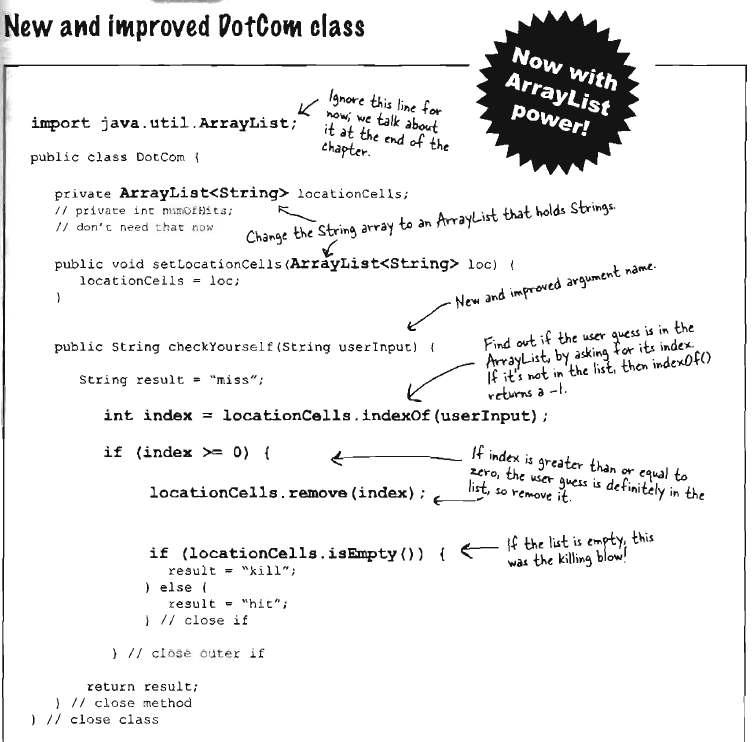


  
**Array** needs size when being declared, size is not required for ArrayList  
**Array** uses special syntax, Array List doesn’t have any special syntax  
**To put object** in regular Array, you have to assign it to a specific location, ArrayList, you can specify index using the add method or you can just keep saying add and the ArrayList will keep growing to make room for the new thing.  
**ArrayLists** are parametrized, angle brackets < > are type parameter, so <String> is “A list of String”  
as opposed to ArrayList<Dog> “a list of Dogs”.  
  
**The existing DotCom Code (Same as chapter 5)**public class DotCom {

int [ ] locationCells;  
 int numOfHits = 0;

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

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

for (int cell : locationCells) {  
 **if (guess == cell) {  
 result = “hit”;  
 numOfHits++;  
 break;  
 }**  
 }  
   
 if (numOfHits == locationCells.length) {  
 result = “kill”;  
 }  
 System.out.println(result);  
 return result;  
 }  
 }  
  
The Bolded part is the problem, we count each guess as hit without checking whether that cell had already been hit or not.  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
**Building the REAL game**Goal – Sink Dot Coms in fewest number of guesses, rating given at the end.  
Setup – Virtual 7 x 7 grid  
How to play – 7 x 7 grid, prompts you to enter a guess, see result hit, miss, or you sunk –NAME-. When you sunk all three Dot Coms, the game ends by printing out your rating.