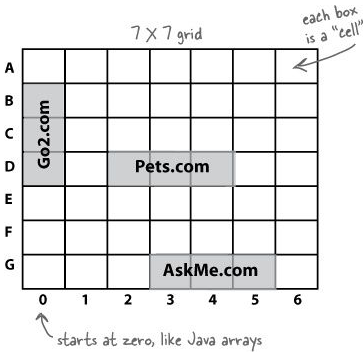
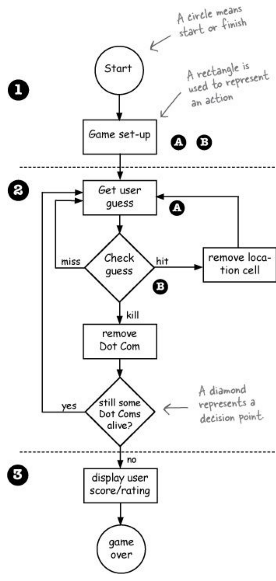
**B”H**

**Chapter 5**

* Over the next two chapters we’re going to build a Battleships game.
* You’re going to build the Sink a Dot Com game, with a 7 x 7 grid and three Dot Coms. Each Dot Com takes up three cells.

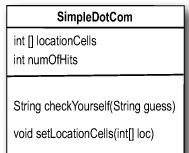
* FIRST thing to do, is at the app level (in contrast to the class level), to create a high-level design using a flow-chart. Every app should first have a flowchart before going down to the “class” level:



* Now we have an idea of the kinds of things the program needs to do. The next step is figuring out what kind of objects we’ll need to do the work. Remember; focus first on the things in the program rather than the procedures.
* In this chapter we’ll start with a stripped-down, simplified version, Simple Dot Com Game. Instead of a 2-D grid, we hide the Dot Com in just a single row. And instead of three Dot Coms, we use one.

**Developing a Class:**

1. Figure out what the class is supposed to do. List the instance variables and methods. I.e. make the UML diagrams.
2. Write the pseudo-code for the methods.
   * This helps you focus on the logic without stressing about syntax
3. Write test-code for the methods.
   * A class or methods that will test the real code and validate that it’s doing the right thing.
4. Implement the class in real-code.
   * The actual implementation of the class. This is where we write real Java code
5. Test the methods.
6. Debug and re-implement as needed.

Step 1 - UML: 

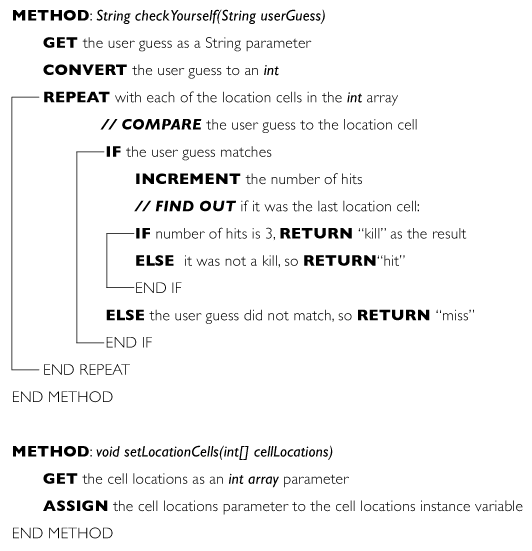
Step 2 – pseudo-code:

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

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

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

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



Step 3 – test-code: see SimpleDotComTester.java

* Why not wait until the real-code is written, and then whip out the test code?
  + The act of thinking through (and writing) the test code helps clarify your thoughts about what the method itself needs to do. As soon as your implementation code is done, you already have test code just waiting to validate it. Besides, you know if you don’t do it now, you’ll never do it.
  + Ideally, write a little test code, then write only the implementation code you need in order to pass that test. Then write a little more test code and write only the new implementation code needed to pass that new test. At each test iteration, you run all the previously-written tests, so that you always prove that your latest code additions don’t break previously-tested code.

Step 4 – real-code: see SimpleDotCom.java

* Some highlights from SimpleDotCom.java:



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

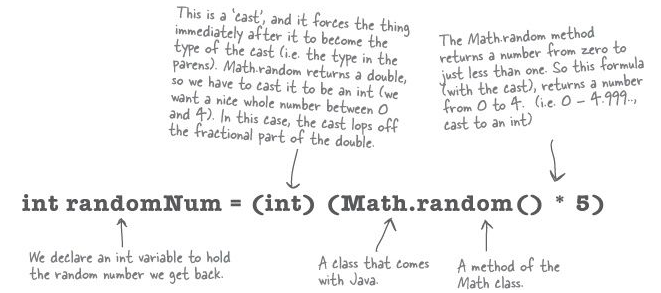
**for (int i = 0; i < 10; i + +) {**

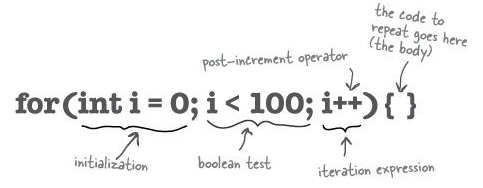
**// do something 10 times**

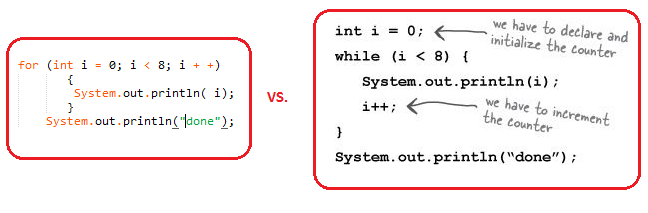
**}**

You can use this format for any kind of loop you need. But beginning with Java 5.0 (Tiger), you can also use the enhanced for loop (used in the above SimpleDotCom class) when your loop needs to iterate over the elements in an array (or another kind of collection, as you’ll see in the next chapter). You can always use the plain old for loop to iterate over an array, but the enhanced for loop makes it easier.

* Some people refer to the enhanced for as the “for each” or the “for in” loop, because that’s how it reads: “for EACH thing IN the collection...”
* Choose **for** loops over **while** loops when you know how many times you want to repeat the loop code.
* Use the pre/ post increment operator to add 1 to a variable **(x++;)**
* Use the pre/ post decrement to subtract 1 from a variable **(x--;)**
* A note on the **main** method in the **Game** class:



* Regular (non-enhanced) for loops: 
* A **while** loop has only the boolean test; it doesn’t have a built-in initialization or iteration expression. A **while** loop is good when you don’t know how many times to loop and just want to keep going while some condition is true.
* But if you know how many times to loop (e.g. the length of an array, 7 times, etc.), a **for** loop is cleaner. Here’s the loop above rewritten using while:



Casting primitives:

* To force the compiler to jam the value of a bigger primitive variable into a smaller one, you can use the cast operator. It looks like this:

**long y = 42; // so far so good**

**int x = (int) y; // x = 42 cool!**

* If the value of y was bigger than the maximum value of x, then what’s left will be a weird number. Still, the point is that the compiler lets you do it:

**long y = 40002; // 40002 exceeds the 16-bit limit of a short**

**short x = (short) y; // x now equals -25534!**

**EXTREME PROGRAMMING (XP):**

These practices include things like:

* Make small, but frequent, releases.
* Develop in iteration cycles.
* Don’t put in anything that’s not in the spec (no matter how tempted you are to put in functionality “for the future”).
* Write the test code first.
* No killer schedules; work regular hours.
* Refactor (improve the code) whenever and wherever you notice the opportunity.
* Don’t release anything until it passes all the tests.
* Set realistic schedules, based around small releases.
* Keep it simple.
* Program in pairs, and move people around so that everybody knows pretty much everything about the code.