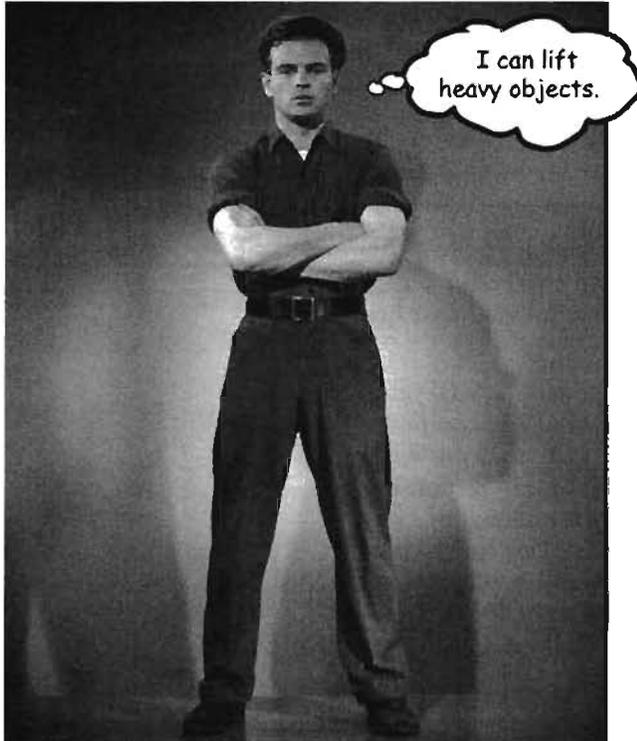


5 writing a program

Extra-Strength Methods



Let's put some muscle in our methods. We dabbled with variables, played with a few objects, and wrote a little code. But we were weak. We need more tools. Like **operators**. We need more operators so we can do something a little more interesting than, say, *bark*. And **loops**. We need loops, but what's with the wimpy *while* loops? We need **for** loops if we're really serious. Might be useful to **generate random numbers**. And **turn a String into an int**, yeah, that would be cool. Better learn that too. And why don't we learn it all by *building* something real, to see what it's like to write (and test) a program from scratch. **Maybe a game**, like Battleships. That's a heavy-lifting task, so it'll take two chapters to finish. We'll build a simple version in this chapter, and then build a more powerful deluxe version in chapter 6.

building a real game

Let's build a Battleship-style game: "Sink a Dot Com"

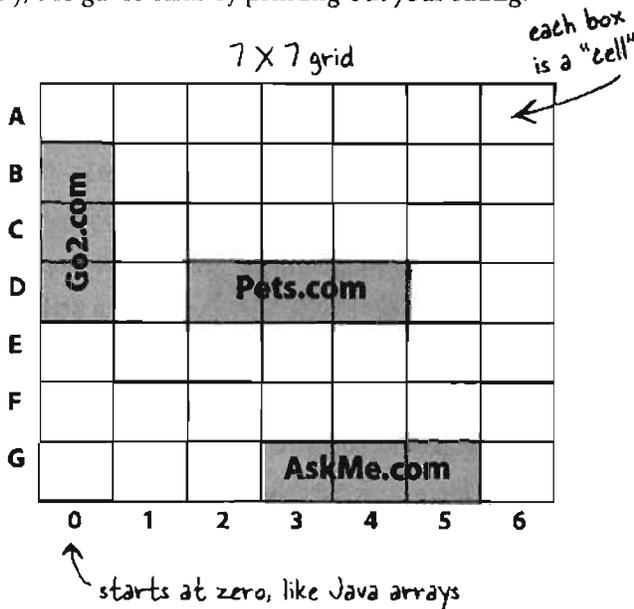
It's you against the computer, but unlike the real Battleship game, in this one you don't place any ships of your own. Instead, your job is to sink the computer's ships in the fewest number of guesses.

Oh, and we aren't sinking ships. We're killing Dot Coms. (Thus establishing business relevancy so you can expense the cost of this book).

Goal: Sink all of the computer's Dot Coms in the fewest number of guesses. You're given a rating or level, based on how well you perform.

Setup: When the game program is launched, the computer places three Dot Coms on a **virtual 7 x 7 grid**. When that's complete, the game asks for your first guess.

How you play: We haven't learned to build a GUI yet, so this version works at the command-line. The computer will prompt you to enter a guess (a cell), that you'll type at the command-line as "A3", "C5", etc.). In response to your guess, you'll see a result at the command-line, either "Hit", "Miss", or "You sunk Pets.com" (or whatever the lucky Dot Com of the day is). When you've sent all three Dot Coms to that big 404 in the sky, the game ends by printing out your rating.



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.

part of a game interaction

```
File Edit Window Help Shell
%java DotComBust
Enter a guess A3
miss
Enter a guess B2
miss
Enter a guess C4
miss
Enter a guess D2
hit
Enter a guess D3
hit
Enter a guess D4
Ouch! You sunk Pets.com : (
kill
Enter a guess B4
miss
Enter a guess G3
hit
Enter a guess G4
hit
Enter a guess G5
Ouch! You sunk AskMe.com : (
```

First, a high-level design

We know we'll need classes and methods, but what should they be? To answer that, we need more information about what the game should do.

First, we need to figure out the general flow of the game. Here's the basic idea:

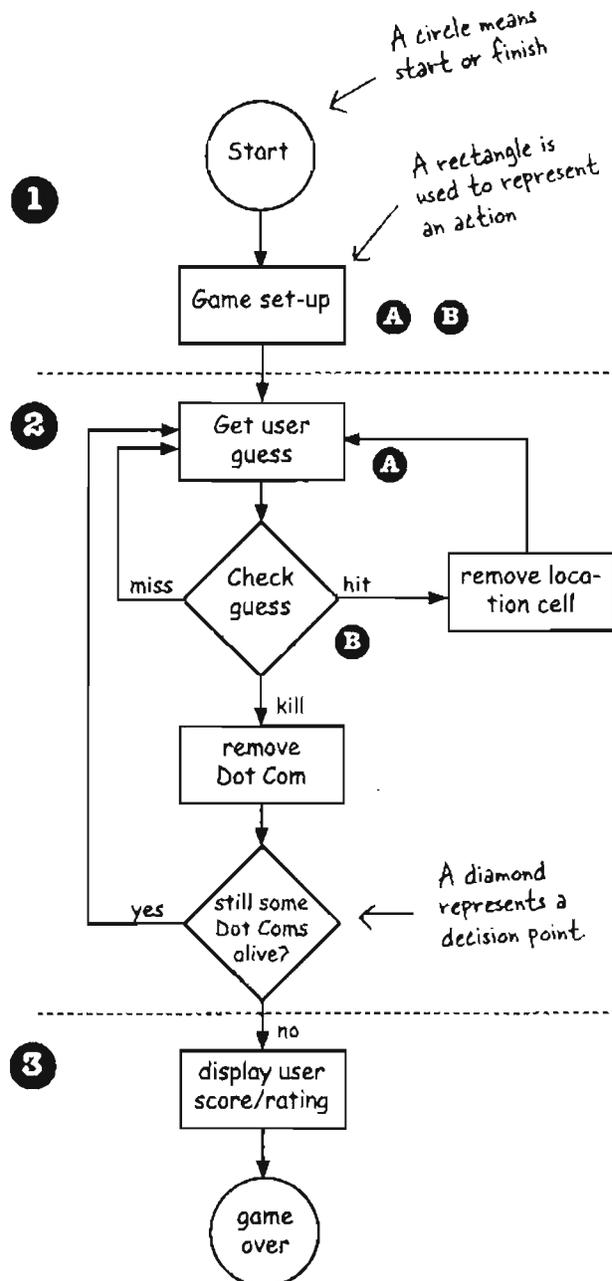
- 1** User starts the game
 - A** Game creates three Dot Coms
 - B** Game places the three Dot Coms onto a virtual grid
- 2** Game play begins

Repeat the following until there are no more Dot Coms:

 - A** Prompt user for a guess ("A2", "C0", etc.)
 - B** Check the user guess against all Dot Coms to look for a hit, miss, or kill. Take appropriate action: if a hit, delete cell (A2, D4, etc.). If a kill, delete Dot Com.
- 3** Game finishes

Give the user a rating based on the number of guesses.

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, think like Brad rather than Larry; focus first on the *things* in the program rather than the *procedures*.



Whoa. A real flow chart.

a simpler version of the game

The "Simple Dot Com Game" a gentler introduction

It looks like we're gonna need at least two classes, a Game class and a DotCom class. But before we build the full monty *Sink a Dot Com* game, we'll start with a stripped-down, simplified version, *Simple Dot Com Game*. We'll build the simple version in *this* chapter, followed by the deluxe version that we build in the *next* chapter.

Everything is simpler in this 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*.

The goal is the same, though, so the game still needs to make a DotCom instance, assign it a location somewhere in the row, get user input, and when all of the DotCom's cells have been hit, the game is over.

This simplified version of the game gives us a big head start on building the full game. If we can get this small one working, we can scale it up to the more complex one later.

In this simple version, the game class has no instance variables, and all the game code is in the `main()` method. In other words, when the program is launched and `main()` begins to run, it will make the one and only DotCom instance, pick a location for it (three consecutive cells on the single virtual seven-cell row), ask the user for a guess, check the guess, and repeat until all three cells have been hit.

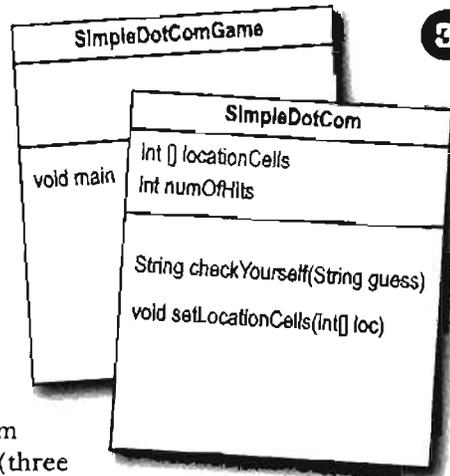
Keep in mind that the virtual row is... *virtual*. In other words, it doesn't exist anywhere in the program. As long as both the game and the user know that the DotCom is hidden in three consecutive cells out of a possible seven (starting at zero), the row itself doesn't have to be represented in code. You might be tempted to build an array of seven ints and then assign the DotCom to three of the seven elements in the array, but you don't need to. All we need is an array that holds just the three cells the DotCom occupies.

- 1 Game starts, and creates ONE DotCom and gives it a location on three cells in the single row of seven cells.

Instead of "A2", "C4", and so on, the locations are just integers (for example: 1,2,3 are the cell locations in this picture:



- 2 Game play begins. Prompt user for a guess, then check to see if it hit any of the DotCom's three cells. If a hit, increment the `numOfHits` variable.
- 3 Game finishes when all three cells have been hit (the `numOfHits` variable value is 3), and tells the user how many guesses it took to sink the DotCom.



A complete game interaction

```
File Edit Window Help Destroy
% java SimpleDotComGame
enter a number 2
hit
enter a number 3
hit
enter a number 4
miss
enter a number 1
kill
You took 4 guesses
```

Developing a Class

As a programmer, you probably have a methodology/process/approach to writing code. Well, so do we. Our sequence is designed to help you see (and learn) what we're thinking as we work through coding a class. It isn't necessarily the way we (or *you*) write code in the Real World. In the Real World, of course, you'll follow the approach your personal preferences, project, or employer dictate. We, however, can do pretty much whatever we want. And when we create a Java class as a "learning experience", we usually do it like this:

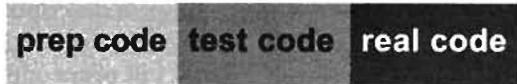
- Figure out what the class is supposed to *do*.
- List the **instance variables and methods**.
- Write **precode** for the methods. (You'll see this in just a moment.)
- Write **test code** for the methods.
- Implement** the class.
- Test** the methods.
- Debug** and **reimplement** as needed.
- Express gratitude that we don't have to test our so-called *learning experience* app on actual live users.



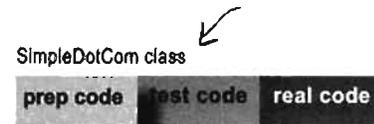
Flex those dendrites.

How would you decide which class or classes to build first, when you're writing a program? Assuming that all but the tiniest programs need more than one class (if you're following good OO principles and not having one class do many different jobs), where do you start?

The three things we'll write for each class:



This bar is displayed on the next set of pages to tell you which part you're working on. For example, if you see this picture at the top of a page, it means you're working on precode for the SimpleDotCom class.



prep code

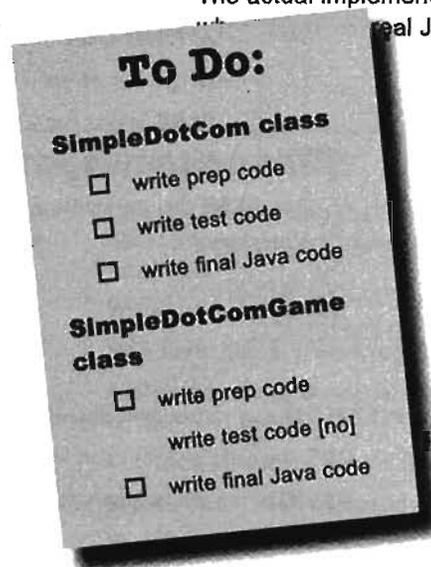
A form of pseudocode, to help you focus on the logic without stressing about syntax.

test code

A class or methods that will test the real code and validate that it's doing the right thing.

real code

The actual implementation of the class. This is the real Java code.



SimpleDotCom class

prep code test code real code

SimpleDotCom
int [] locationCells int numOfHits
String checkYourself(String guess) void setLocationCells(int[] loc)

You'll get the idea of how precode (our version of pseudocode) works as you read through this example. It's sort of half-way between real Java code and a plain English description of the class. Most precode includes three parts: instance variable declarations, method declarations, method logic. The most important part of precode is the method logic, because it defines *what* has to happen, which we later translate into *how*, when we actually write the method 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.).

METHOD: *String checkYourself(String userGuess)*

GET the user guess as a *String* parameter

CONVERT the user guess to an *int*

REPEAT with each of the location cells in the *int array*

 // **COMPARE** the user guess to the location cell

IF the user guess matches

INCREMENT the number of hits

 // **FIND OUT** if it was the last location cell:

IF number of hits is 3, **RETURN** "kill" as the result

ELSE it was not a kill, so **RETURN** "hit"

END IF

ELSE the user guess did not match, so **RETURN** "miss"

END IF

END REPEAT

END METHOD

METHOD: *void setLocationCells(int[] cellLocations)*

GET the cell locations as an *int array* parameter

ASSIGN the cell locations parameter to the cell locations instance variable

END METHOD

prep code test code **real code**

Writing the method implementations

let's write the real method code now, and get this puppy working.

Before we start coding the methods, though, let's back up and write some code to *test* the methods. That's right, we're writing the test code *before* there's anything to test!

The concept of writing the test code first is one of the practices of Extreme Programming (XP), and it can make it easier (and faster) for you to write your code. We're not necessarily saying you should use XP, but we do like the part about writing tests first. And XP just *sounds* cool.



Extreme Programming (XP)

Extreme Programming (XP) is a newcomer to the software development methodology world. Considered by many to be "the way programmers really want to work," XP emerged in the late 90's and has been adopted by companies ranging from the two-person garage shop to the Ford Motor Company. The thrust of XP is that the customer gets what he wants, when he wants it, even when the spec changes late in the game.

XP is based on a set of proven practices that are all designed to work together, although many folks do pick and choose, and adopt only a portion of XP's rules. 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.

SimpleDotCom class

prep code test code real code

Writing test code for the SimpleDotCom class

We need to write test code that can make a SimpleDotCom object and run its methods. For the SimpleDotCom class, we really care about only the *checkYourself()* method, although we *will* have to implement the *setLocationCells()* method in order to get the *checkYourself()* method to run correctly.

Take a good look at the precode below for the *checkYourself()* method (the *setLocationCells()* method is a no-brainer setter method, so we're not worried about it, but in a 'real' application we might want a more robust 'setter' method, which we *would* want to test).

Then ask yourself, "If the *checkYourself()* method were implemented, what test code could I write that would prove to me the method is working correctly?"

Based on this precode:

```
METHOD String checkYourself(String userGuess)
  GET the user guess as a String parameter
  CONVERT the user guess to an Int
  REPEAT with each of the location cells in the Int array
    // COMPARE the user guess to the location cell
    IF the user guess matches
      INCREMENT the number of hits
      // FIND OUT if it was the last location cell:
      IF number of hits is 3, RETURN "Kill" as the result
      ELSE it was not a kill, so RETURN "Hit"
    END IF
  ELSE the user guess did not match, so RETURN "Miss"
  END IF
END REPEAT
END METHOD
```

Here's what we should test:

1. Instantiate a SimpleDotCom object.
2. Assign it a location (an array of 3 ints, like {2,3,4}).
3. Create a String to represent a user guess ("2", "0", etc.).
4. Invoke the *checkYourself()* method passing it the fake user guess.
5. Print out the result to see if it's correct ("passed" or "failed").

prep code test code real code

There are no Dumb Questions

Q: Maybe I'm missing something here, but how exactly do you run a test on something that doesn't yet exist?

A: You don't. We never said you start by *running* the test; you start by *writing* the test. At the time you write the test code, you won't have anything to run it against, so you probably won't be able to compile it until you write 'stub' code that can compile, but that will always cause the test to fail (like, return null.)

Q: Then I still don't see the point. Why not wait until the code is written, and then whip out the test code?

A: 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. There's always something more interesting to do.

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.

Test code for the SimpleDotCom class

```
public class SimpleDotComTestDrive {
    public static void main (String[] args) {
        SimpleDotCom dot = new SimpleDotCom();
        int[] locations = {2,3,4};
        dot.setLocationCalls(locations);
        String userGuess = "2";
        String result = dot.checkYourself(userGuess);
        String testResult = "failed";
        if (result.equals("hit") ) {
            testResult = "passed";
        }
        System.out.println(testResult);
    }
}
```

instantiate a SimpleDotCom object

make an int array for the location of the dot com (3 consecutive ints out of a possible 7).

invoke the setter method on the dot com.

make a fake user guess

invoke the checkYourself() method on the dot com object, and pass it the fake guess.

if the fake guess (2) gives back a "hit", it's working

print out the test result ("passed or failed")

Sharpen your pencil

In the next couple of pages we implement the SimpleDotCom class, and then later we return to the test class. Looking at our test code above, what else should be added? What are we *not* testing in this code, that we *should* be testing for? Write your ideas (or lines of code) below:

SimpleDotCom class

prep code

real code

The checkYourself() method

There isn't a perfect mapping from precode to javacode; you'll see a few adjustments. The precode gave us a much better idea of *what* the code needs to do, and now we have to find the Java code that can do the *how*.

In the back of your mind, be thinking about parts of this code you might want (or need) to improve. The numbers ① are for things (syntax and language features) you haven't seen yet. They're explained on the opposite page.

GET the user guess

CONVERT the user guess to an int

REPEAT with each cell in the int array

IF the user guess matches

INCREMENT the number of hits

// FIND OUT if it was the last cell

IF number of hits is 3,

RETURN "kill" as the result

ELSE it was not a kill, so

RETURN "hit"

ELSE

RETURN "miss"

```
public String checkYourself(String stringGuess) {
```

```
    int guess = Integer.parseInt(stringGuess);
```

```
    String result = "miss";
```

```
    for (int cell : locationCells) {
```

```
        if (guess == cell) {
```

```
            result = "hit";
```

```
            numOfHits++;
```

```
            break;
```

```
        } // end if
```

```
    } // end for
```

```
    if (numOfHits == locationCells.length) {
```

```
        result = "kill";
```

```
    } // end if
```

```
    System.out.println(result);
```

```
    return result;
```

```
    } // end method
```

convert the String to an int

make a variable to hold the result we'll return. put "miss" in as the default (i.e. we assume a "miss")

repeat with each cell in the locationCells array (each cell location of the object)

compare the user guess to this element (cell) in the array

we got a hit!

get out of the loop, no need to test the other cells

we're out of the loop, but let's see if we're now 'dead' (hit 3 times) and change the result String to "Kill"

display the result for the user ("Miss", unless it was changed to "Hit" or "Kill")

return the result back to the calling method

prep code test code real code

Just the new stuff

The things we haven't seen before are on this page. Stop worrying! The rest of the details are at the end of the chapter. This is just enough to let you keep going.

- ① Converting a String to an int

A class that ships with Java.

A method in the Integer class that knows how to "parse" a String into the int it represents.

Takes a String.

Integer.parseInt("3")

- ② The for loop

Read this for loop declaration as "repeat for each element in the 'locationCells' array: take the next element in the array and assign it to the int variable 'cell'."

The colon (:) means "in", so the whole thing means "for each int value IN locationCells..."

for (int cell : locationCells) { }

Declare a variable that will hold one element from the array. Each time through the loop, this variable (in this case an int variable named "cell"), will hold a different element from the array, until there are no more elements (or the code does a "break"... see #4 below).

The array to iterate over in the loop. Each time through the loop, the next element in the array will be assigned to the variable "cell". (More on this at the end of this chapter.)

- ③ The post-increment operator

numOfHits++

The ++ means add 1 to whatever's there (in other words, increment by 1).

numOfHits++ is the same (in this case) as saying numOfHits = numOfHits + 1, except slightly more efficient

- ④ break statement

break;

Gets you out of a loop. Immediately. Right here. No iteration, no boolean test, just get out now!

SimpleDotCom class

prep code test code real code

there are no Dumb Questions

Q: What happens in `Integer.parseInt()` if the thing you pass isn't a number? And does it recognize spelled-out numbers, like "three"?

A: `Integer.parseInt()` works only on Strings that represent the ASCII values for digits (0,1,2,3,4,5,6,7,8,9). If you try to parse something like "two" or "blorp", the code will blow up at runtime. (By *blow up*, we actually mean *throw an exception*, but we don't talk about exceptions until the Exceptions chapter. So for now, *blow up* is close enough.)

Q: In the beginning of the book, there was an example of a *for* loop that was really different from this one—are there two different styles of *for* loops?

A: Yes! 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 (that's the official description) 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.

Final code for SimpleDotCom and SimpleDotComTester

```
public class SimpleDotComTestDrive {  
  
    public static void main (String[] args) {  
        SimpleDotCom dot = new SimpleDotCom();  
        int[] locations = {2,3,4};  
        dot.setLocationCells(locations);  
        String userGuess = "2";  
        String result = dot.checkYourself(userGuess);  
    }  
}
```

```
public class SimpleDotCom {  
  
    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;  
            }  
        } // out of the loop  
  
        if (numOfHits ==  
            locationCells.length) {  
            result = "kill";  
        }  
        System.out.println(result);  
        return result;  
    } // close method  
} // close class
```

There's a little bug lurking here. It compiles and runs, but sometimes... don't worry about it for now, but we *will* have to face it a little later.

What should we see when we run this code?

The test code makes a `SimpleDotCom` object and gives it a location at 2,3,4. Then it sends a fake user guess of "2" into the `checkYourself()` method. If the code is working correctly, we should see the result print out:

```
java SimpleDotComTestDrive  
hit
```

prep code test code real code

Sharpen your pencil

We built the test class, and the SimpleDotCom class. But we still haven't made the actual *game*. Given the code on the opposite page, and the spec for the actual game, write in your ideas for precode for the game class. We've given you a few lines here and there to get you started. The actual game code is on the next page, so **don't turn the page until you do this exercise!**

You should have somewhere between 12 and 18 lines (including the ones we wrote, but *not* including lines that have only a curly brace).

METHOD `public static void main (String [] args)`

DECLARE an int variable to hold the number of user guesses, named `numOfGuesses`

COMPUTE a random number between 0 and 4 that will be the starting location cell position

WHILE the dot com is still alive :

GET user input from the command line

The SimpleDotComGame needs to do this:

1. Make the single SimpleDotCom Object.
2. Make a location for it (three consecutive cells on a single row of seven virtual cells).
3. Ask the user for a guess.
4. Check the guess.
5. Repeat until the dot com is dead .
6. Tell the user how many guesses it took.

A complete game interaction

```
File Edit Window Help Runaway
% java SimpleDotComGame
enter a number 2
hit
enter a number 3
hit
enter a number 4
miss
enter a number 1
kill
You took 4 guesses
```

Precode for the SimpleDotComGame class

Everything happens in main()

There are some things you'll have to take on faith. For example, we have one line of precode that says, "GET user input from command-line". Let me tell you, that's a little more than we want to implement from scratch right now. But happily, we're using OO. And that means you get to ask some *other* class/object to do something for you, without worrying about *how* it does it. When you write precode, you should assume that *somehow* you'll be able to do whatever you need to do, so you can put all your brainpower into working out the logic.

```
public static void main (String [] args)
```

DECLARE an int variable to hold the number of user guesses, named *numOfGuesses*, set it to 0.

MAKE a new SimpleDotCom instance

COMPUTE a random number between 0 and 4 that will be the starting location cell position

MAKE an int array with 3 ints using the randomly-generated number, that number incremented by 1, and that number incremented by 2 (example: 3,4,5)

INVOKE the *setLocationCells()* method on the SimpleDotCom instance

DECLARE a boolean variable representing the state of the game, named *isAlive*. **SET** it to true

WHILE the dot com is still alive (*isAlive == true*) :

GET user input from the command line

// **CHECK** the user guess

INVOKE the *checkYourself()* method on the SimpleDotCom instance

INCREMENT *numOfGuesses* variable

// **CHECK** for dot com death

IF result is "kill"

SET *isAlive* to false (which means we won't enter the loop again)

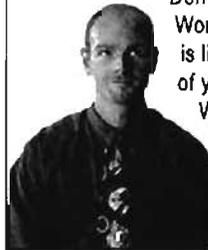
PRINT the number of user guesses

END IF

END WHILE

END METHOD

metacognitive tip



Don't work one part of the brain for too long a stretch at one time. Working just the left side of the brain for more than 30 minutes is like working just your left arm for 30 minutes. Give each side of your brain a break by switching sides at regular intervals.

When you shift to one side, the other side gets to rest and recover. Left-brain activities include things like step-by-step sequences, logical problem-solving, and analysis, while the right-brain kicks in for metaphors, creative problem-solving, pattern-matching, and visualizing.


BULLET POINTS

- Your Java program should start with a high-level design.
- Typically you'll write three things when you create a new class:
 - precode*
 - testcode*
 - real (Java) code*
- Precode should describe *what* to do, not *how* to do it. Implementation comes later.
- Use the precode to help design the test code.
- Write test code *before* you implement the methods.
- 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--`;)
 - Use `Integer.parseInt()` to get the int value of a String.
 - `Integer.parseInt()` works only if the String represents a digit ("0", "1", "2", etc.)
 - Use *break* to leave a loop early (i.e. even if the boolean test condition is still true).



Howdy from Ghost Town

SimpleDotComGame class

prep code test code real code

The game's main() method

Just as you did with the SimpleDotCom class, be thinking about parts of this code you might want (or need) to improve. The numbered things ● are for stuff we want to point out. They're explained on the opposite page. Oh, if you're wondering why we skipped the test code phase for this class, we don't need a test class for the game. It has only one method, so what would you do in your test code? Make a *separate* class that would call *main()* on *this* class? We didn't bother.

DECLARE a variable to hold user guess count, set it to 0

MAKE a SimpleDotCom object

COMPUTE a random number between 0 and 4

MAKE an int array with the 3 cell locations, and

INVOKE setLocationCells on the dot com object

DECLARE a boolean isAlive

WHILE the dot com is still alive

GET user input

// CHECK it

INVOKE checkYourself() on dot com

INCREMENT numOfGuesses

IF result is "kill"

SET gameAlive to false

PRINT the number of user guesses

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

```
    int numOfGuesses = 0;
```

```
    GameHelper helper = new GameHelper();
```

```
    SimpleDotCom theDotCom = new SimpleDotCom();
```

```
    int randomNum = (int) (Math.random() * 5);
```

```
    int[] locations = {randomNum, randomNum+1, randomNum+2};
```

```
    theDotCom.setLocationCells(locations);
```

```
    boolean isAlive = true;
```

```
    while(isAlive == true) {
```

```
        String guess = helper.getUserInput("enter a number");
```

```
        String result = theDotCom.checkYourself(guess);
```

```
        numOfGuesses++;
```

```
        if (result.equals("kill"))
```

```
            isAlive = false;
```

```
            System.out.println("You took " + numOfGuesses + " guesses");
```

```
        } // close if
```

```
    } // close while
```

```
} // close main
```

make a variable to track how many guesses the user makes

this is a special class we wrote that has the method for getting user input for now, pretend it's part of Java

make the dot com object

make a random number for the first cell, and use it to make the cell locations array

give the dot com its locations (the array)

make a boolean variable to track whether the game is still alive, to use in the while loop test repeat while game is still alive.

get user input String

ask the dot com to check the guess; save the returned result in a String

increment guess count

was it a "kill"? if so, set isAlive to false (so we won't re-enter the loop) and print user guess count

prep code test code **real code**

random() and getUserInput()

Two things that need a bit more explaining, are on this page. This is just a quick look to keep you going; more details on the GameHelper class are at the end of this chapter.

- 1 Make a random number

This is a 'cast', and it forces the thing immediately after it to become the type of the cast (i.e. the type in the parens). Math.random returns a double, so we have to cast it to be an int (we want a nice whole number between 0 and 4). In this case, the cast lops off the fractional part of the double.

The Math.random method returns a number from zero to just less than one. So this formula (with the cast), returns a number from 0 to 4. (i.e. 0 - 4.999..., cast to an int)

```
int randomNum = (int) (Math.random() * 5)
```

We declare an int variable to hold the random number we get back.

A class that comes with Java.

A method of the Math class

- 2 Getting user input using the GameHelper class

An instance we made earlier, of a class that we built to help with the game. It's called GameHelper and you haven't seen it yet (you will).

This method takes a String argument that it uses to prompt the user at the command-line. Whatever you pass in here gets displayed in the terminal just before the method starts looking for user input

```
String guess = helper.getUserInput("enter a number");
```

We declare a String variable to hold the user input String we get back ("3", "5", etc.).

A method of the GameHelper class that asks the user for command-line input, reads it in after the user hits RETURN, and gives back the result as a String.

GameHelper class (Ready-bake)

prep code test code real code

One last class: GameHelper

We made the *dot com* class.

We made the *game* class.

All that's left is the *helper* class—the one with the `getUserInput()` method. The code to get command-line input is more than we want to explain right now. It opens up way too many topics best left for later. (Later, as in chapter 14.)

Just copy* the code below and compile it into a class named `GameHelper`. Drop all three classes (`SimpleDotCom`, `SimpleDotComGame`, `GameHelper`) into the same directory, and make it your working directory.



Whenever you see the  logo, you're seeing code that you have to type as-is and take on faith. Trust it. You'll learn how that code works *later*.

I pre-cooked some code so you don't have to make it yourself.



Ready-bake
Code

```
import java.io.*;
public class GameHelper {
    public String getUserInput(String prompt) {
        String inputLine = null;
        System.out.print(prompt + " ");
        try {
            BufferedReader is = new BufferedReader(
                new InputStreamReader(System.in));
            inputLine = is.readLine();
            if (inputLine.length() == 0) return null;
        } catch (IOException e) {
            System.out.println("IOException: " + e);
        }
        return inputLine;
    }
}
```

*We know how much you enjoy typing, but for those rare moments when you'd rather do something else, we've made the Ready-bake Code available on wickedlysmart.com.

Let's play

Here's what happens when we run it and enter the numbers 1,2,3,4,5,6. Lookin' good.

A complete game interaction (your mileage may vary)

```
File Edit Window Help Smile
%java SimpleDotComGame
enter a number 1
miss
enter a number 2
miss
enter a number 3
miss
enter a number 4
hit
enter a number 5
hit
enter a number 6
kill
You took 6 guesses
```

What's this? A bug?

Gaspl

Here's what happens when we enter 1,1,1.

A different game interaction (yikes)

```
File Edit Window Help Faint
%java SimpleDotComGame
enter a number 1
hit
enter a number 1
hit
enter a number 1
kill
You took 3 guesses
```

Sharpen your pencil



It's a cliff-hanger!
Will we *find* the bug?
Will we *fix* the bug?

Stay tuned for the next chapter, where we answer these questions and more...

And in the meantime, see if you can come up with ideas for what went wrong and how to fix it.

for loops

More about for loops

We've covered all the game code for *this* chapter (but we'll pick it up again to finish the deluxe version of the game in the next chapter). We didn't want to interrupt your work with some of the details and background info, so we put it back here. We'll start with the details of for loops, and if you're a C++ programmer, you can just skim these last few pages...

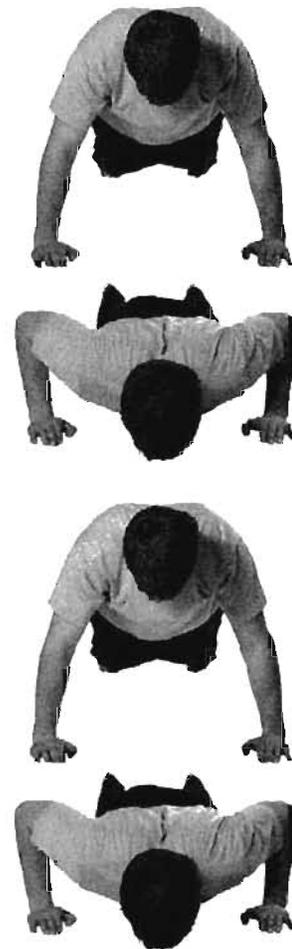
Regular (non-enhanced) for loops

```
for (int i = 0; i < 100; i++) { }
```

Handwritten annotations:

- Initialization: `int i = 0;`
- Boolean test: `i < 100;`
- Iteration expression: `i++`
- Post-increment operator: `++`
- The code to repeat goes here (the body): `{ }`

repeat for 100 reps:



What it means in plain English: "Repeat 100 times."

How the compiler sees it:

- ✦ create a variable *i* and set it to 0.
- ✦ repeat while *i* is less than 100.
- ✦ at the end of each loop iteration, add 1 to *i*

Part One: *initialization*

Use this part to declare and initialize a variable to use within the loop body. You'll most often use this variable as a counter. You can actually initialize more than one variable here, but we'll get to that later in the book.

Part Two: *boolean test*

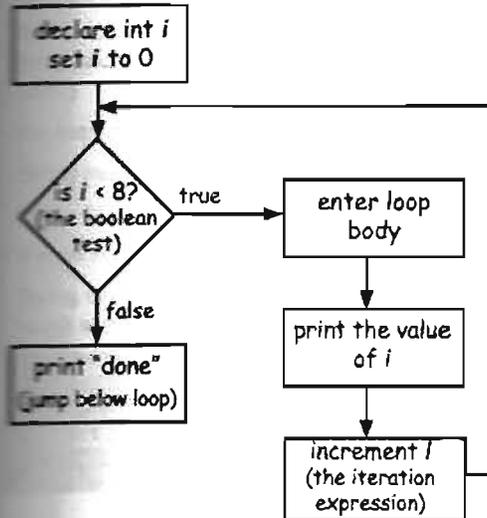
This is where the conditional test goes. Whatever's in there, it *must* resolve to a boolean value (you know, *true* or *false*). You can have a test, like `(x >= 4)`, or you can even invoke a method that returns a boolean.

Part Three: *iteration expression*

In this part, put one or more things you want to happen with each trip through the loop. Keep in mind that this stuff happens at the *end* of each loop.

Trips through a loop

```
for (int i = 0; i < 8; i++) {
    System.out.println(i);
}
System.out.println("done");
```



Difference between for and while

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:

```
int i = 0;
while (i < 8) {
    System.out.println(i);
    i++;
}
System.out.println("done");
```

we have to declare and initialize the counter

we have to increment the counter

output:

```
File Edit Window Help Repeat
%java Test
0
1
2
3
4
5
6
7
done
```

++ --

Pre and Post Increment/Decrement Operator

The shortcut for adding or subtracting 1 from a variable.

x++;

Is the same as:

x = x + 1;

They both mean the same thing in this context:

"add 1 to the current value of x" or "increment x by 1"

And:

x--;

Is the same as:

x = x - 1;

Of course that's never the whole story. The placement of the operator (either before or after the variable) can affect the result. Putting the operator *before* the variable (for example, ++x), means, "first, increment x by 1, and then use this new value of x." This only matters when the ++x is part of some larger expression rather than just in a single statement.

```
int x = 0;    int z = ++x;
```

produces: x is 1, z is 1

But putting the ++ *after* the x give you a different result:

```
int x = 0;    int z = x++;
```

produces: x is 1, but z is 0! z gets the value of x and then x is incremented.

enhanced for

The enhanced for loop

Beginning with Java 5.0 (Tiger), the Java language has a second kind of *for* loop called the *enhanced for*, that makes it easier to iterate over all the elements in an array or other kinds of collections (you'll learn about *other* collections in the next chapter). That's really all that the enhanced for gives you—a simpler way to walk through all the elements in the collection, but since it's the most common use of a *for* loop, it was worth adding it to the language. We'll revisit the *enhanced for loop* in the next chapter, when we talk about collections that *aren't* arrays.

Declare an iteration variable that will hold a single element in the array.

The colon (:) means "IN".

The code to repeat goes here (the body).

```
for (String name : nameArray) { }
```

The elements in the array **MUST** be compatible with the declared variable type.

With each iteration, a different element in the array will be assigned to the variable "name".

The collection of elements that you want to iterate over. Imagine that somewhere earlier, the code said:
`String[] nameArray = {"Fred", "Mary", "Bob"};`
With the first iteration, the name variable has the value of "Fred", and with the second iteration, a value of "Mary", etc.

What it means in plain English: "For each element in nameArray, assign the element to the 'name' variable, and run the body of the loop."

How the compiler sees it:

- ✦ Create a String variable called *name* and set it to null.
- ✦ Assign the first value in *nameArray* to *name*.
- ✦ Run the body of the loop (the code block bounded by curly braces).
- ✦ Assign the next value in *nameArray* to *name*.
- ✦ Repeat while *there are still elements in the array*.

Note: depending on the programming language they've used in the past, 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..."

Part One: iteration variable declaration

Use this part to declare and initialize a variable to use within the loop body. With each iteration of the loop, this variable will hold a different element from the collection. The type of this variable must be compatible with the elements in the array! For example, you can't declare an *int* iteration variable to use with a *String[]* array.

Part Two: the actual collection

This must be a reference to an array or other collection. Again, don't worry about the *other* non-array kinds of collections yet—you'll see them in the next chapter.

Converting a String to an int

```
int guess = Integer.parseInt(stringGuess);
```

The user types his guess at the command-line, when the game prompts him. That guess comes in as a *String* ("2"; "0"; etc.), and the game passes that *String* into the `checkYourself()` method.

But the cell locations are simply *ints* in an array, and you can't compare an *int* to a *String*.

For example, *this won't work*:

```
String num = "2";
```

```
int x = 2;
```

```
if (x == num) // horrible explosion!
```

Trying to compile that makes the compiler laugh and mock you:

```
operator == cannot be applied to
int, java.lang.String
```

```
if (x == num) {
```

```
^
```

So to get around the whole apples and oranges thing, we have to make the *String* "2" into the *int* 2. Built into the Java class library is a class called *Integer* (that's right, an *Integer class*, not the *int primitive*), and one of its jobs is to take *Strings* that represent numbers and convert them into *actual numbers*.

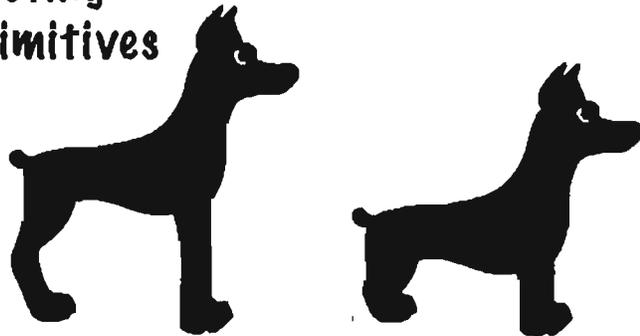
a class that ships
with Java

takes a *String*

Integer.parseInt("3")

a method in the *Integer*
class that knows how to
"parse" a *String* into the
int it represents.

Casting primitives



long $\xrightarrow{\text{can be cast to}}$ short



but you might
lose something



In chapter 3 we talked about the sizes of the various primitives, and how you can't shove a big thing directly into a small thing:

```
long y = 42;
```

```
int x = y; // won't compile
```

A *long* is bigger than an *int* and the compiler can't be sure where that *long* has been. It might have been out drinking with the other *longs*, and taking on really big values. 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!
```

Putting in the *cast* tells the compiler to take the value of *y*, chop it down to *int* size, and set *x* equal to whatever is left. If the value of *y* was bigger than the maximum value of *x*, then what's left will be a weird (but calculable*) number:

```
long y = 40002;
```

```
// 40002 exceeds the 16-bit limit of a short
```

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

Still, the point is that the compiler lets you do it. And let's say you have a floating point number, and you just want to get at the whole number (*int*) part of it:

```
float f = 3.14f;
```

```
int x = (int) f; // x will equal 3
```

And don't even *think* about casting anything to a *boolean* or vice versa—just walk away.

*It involves sign bits, binary, 'two's complement' and other geekery, all of which are discussed at the beginning of appendix B.

exercise: Be the JVM



BE the JVM



The Java file on this page represents a complete source file. Your job is to play JVM and determine what would be the output when the program runs?

```
class Output {  
  
    public static void main(String [] args) {  
        Output o = new Output();  
        o.go();  
    }  
  
    void go() {  
        int y = 7;  
        for(int x = 1; x < 8; x++) {  
            y++;  
            if (x > 4) {  
                System.out.print(++y + " ");  
            }  
            if (y > 14) {  
                System.out.println(" x = " + x);  
                break;  
            }  
        }  
    }  
}
```

```
File Edit Window Help OM  
% java Output  
12 14
```

-or-

```
File Edit Window Help Incons  
% java Output  
12 14 x = 6
```

-or-

```
File Edit Window Help Believe  
% java Output  
13 15 x = 6
```



Code Magnets



A working Java program is all scrambled up on the fridge. Can you reconstruct the code snippets to make a working Java program that produces the output listed below? Some of the curly braces fell on the floor and they were too small to pick up, so feel free to add as many of those as you need!



```
x++;
```

```
if (x == 1) {
```

```
System.out.println(x + " " + y);
```

```
class MultiFor {
```

```
for(int y = 4; y > 2; y--) {
```

```
for(int x = 0; x < 4; x++) {
```

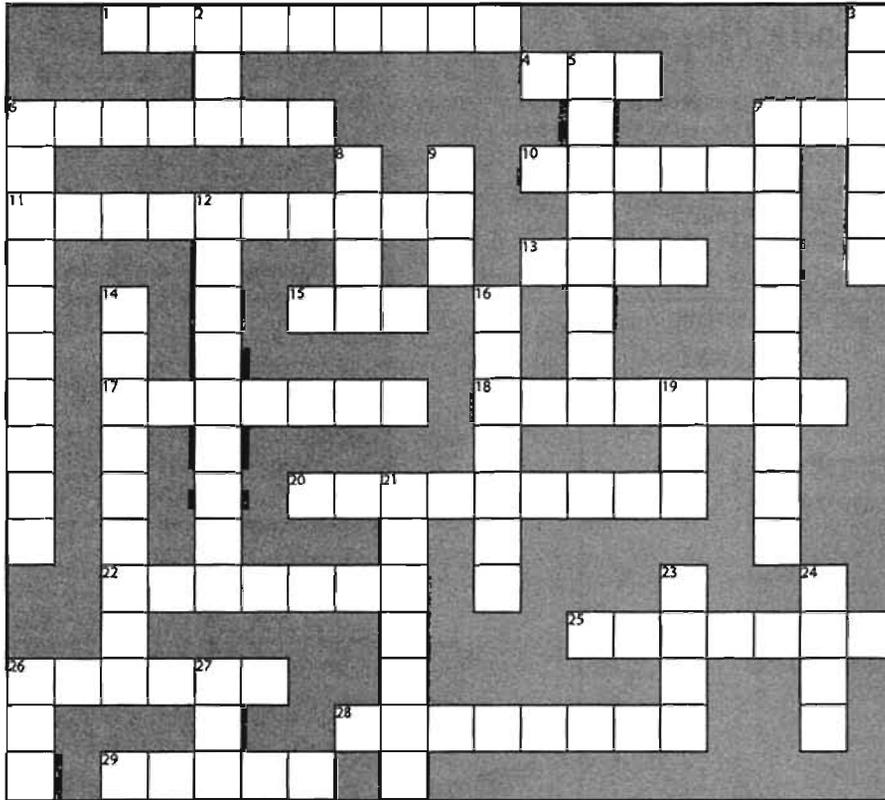
```
public static void main(String [] args) {
```

```

File Edit Window Help Rsd
java MultiFor
0 4
1 3
2 4
3 3
4 4
5 3

```

puzzle: JavaCross



JavaCross

How does a crossword puzzle help you learn Java? Well, all of the words **are** Java related. In addition, the clues provide metaphors, puns, and the like. These mental twists and turns burn alternate routes to Java knowledge, right into your brain!

Across

- 1. Fancy computer word for build
- 4. Multi-part loop
- 6. Test first
- 7. 32 bits
- 10. Method's answer
- 11. Precode-esque
- 13. Change
- 15. The big toolkit
- 17. An array unit
- 18. Instance or local
- 20. Automatic toolkit
- 22. Looks like a primitive, but_
- 25. Un-castable
- 26. Math method
- 28. Converter method
- 29. Leave early

Down

- 2. Increment type
- 3. Class's workhorse
- 5. Pre is a type of ____
- 6. For's iteration ____
- 7. Establish first value
- 8. While or For
- 9. Update an instance variable
- 12. Towards blastoff
- 14. A cycle
- 16. Talkative package
- 19. Method messenger (abbrev.)
- 21. As if
- 23. Add after
- 24. Pi house
- 26. Compile it and ____
- 27. ++ quantity



A short Java program is listed below. One block of the program is missing. Your challenge is to **match the candidate block of code** (on the left), **with the output** that you'd see if the block 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.

```
class MixFor5 {
    public static void main(String [] args) {
        int x = 0;
        int y = 30;
        for (int outer = 0; outer < 3; outer++) {
            for (int inner = 4; inner > 1; inner--) {
                
                y = y - 2;
                if (x == 6) {
                    break;
                }
                x = x + 3;
            }
            y = y - 2;
        }
        System.out.println(x + " " + y);
    }
}
```

← candidate code goes here

Candidates:

`x = x + 3;`

`x = x + 6;`

`x = x + 2;`

`x++;`

`x--;`

`x = x + 0;`

Possible output:

45 6

36 6

54 6

60 10

18 6

6 14

12 14

match each candidate with one of the possible outputs



Exercise Solutions

Be the JVM:

```
class Output {  
  
    public static void main(String [] args) {  
        Output o = new Output();  
        o.go();  
    }  
    void go() {  
        int y = 7;  
        for(int x = 1; x < 8; x++) {  
            y++;  
            if (x > 4) {  
                System.out.print(++y + " ");  
            }  
            if (y > 14) {  
                System.out.println(" x = " + x);  
                break;  
            }  
        }  
    }  
}
```

Did you remember to factor in the break statement? How did that affect the output?

```
File Edit Window Help MotorcycleMaintenance  
% java Output  
13 15 x = 6
```

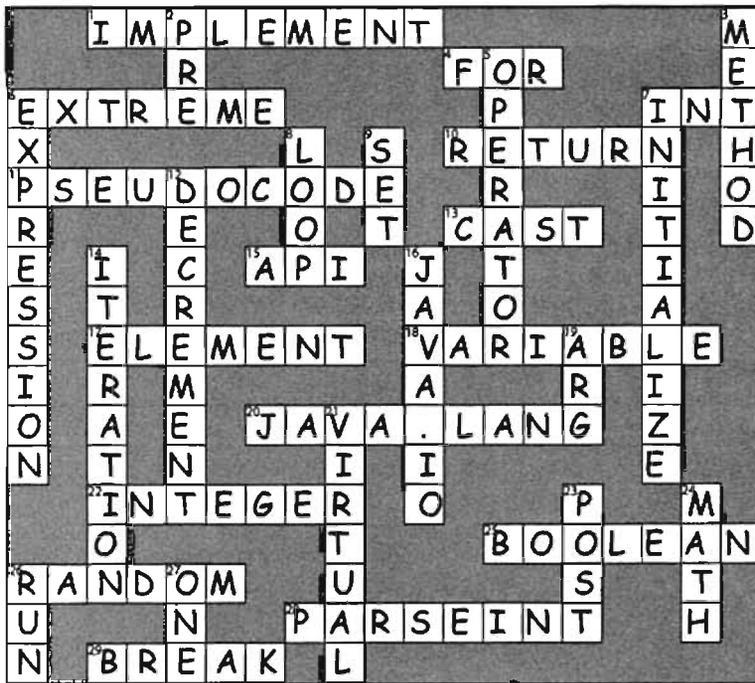
Code Magnets:

```
class MultiFor {  
  
    public static void main(String [] args) {  
  
        for(int x = 0; x < 4; x++) {  
  
            for(int y = 4; y > 2; y--) {  
                System.out.println(x + " " + y);  
            }  
  
            if (x == 1) {  
                x++;  
            }  
        }  
    }  
}
```

What would happen if this code block came before the 'y' for loop?

```
File Edit Window Help Monopole  
% java MultiFor  
0 4  
0 3  
1 4  
1 3  
3 4  
3 3
```

 Puzzle Solutions



Candidates:

- `x = x + 3;`
- `x = x + 6;`
- `x = x + 2;`
- `x++;`
- `x--;`
- `x = x + 0;`

Possible output:

- 45 6
- 36 6
- 54 6
- 60 10
- 18 6
- 6 14
- 12 14

