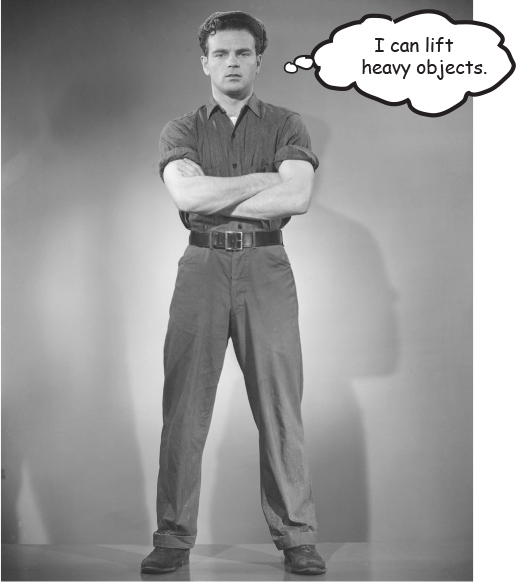
# Chapter 5. Writing a Program: Extra-Strength Methods



**Let’s put some muscle in our** **methods.** We dabbled балуется 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 слабак *while* loops? We need ***for*** loops if we’re really serious. Might be useful to **generate random numbers**. 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](#_Chapter_6._Get).

**Let’s build a Battleship-style game: “Sink a Startup”**

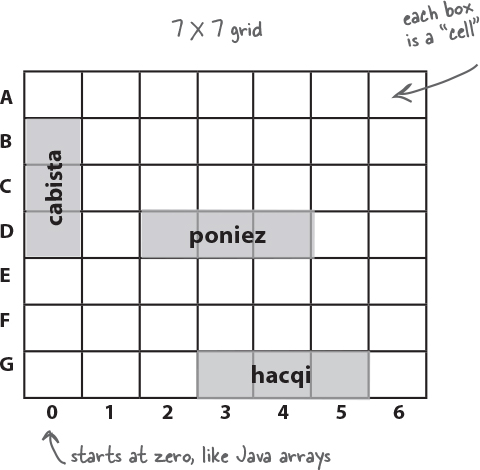
It’s you against the computer, but unlike the real Battleship game, in this one you don’t place any ships of your own. Instead, your job is to sink the computer’s ships in the fewest number of guesses.

Oh, and we aren’t sinking ships. We’re killing ill-advised, Silicon Valley Startups. (Thus establishing business relevancy so you can expense the cost of this book).

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

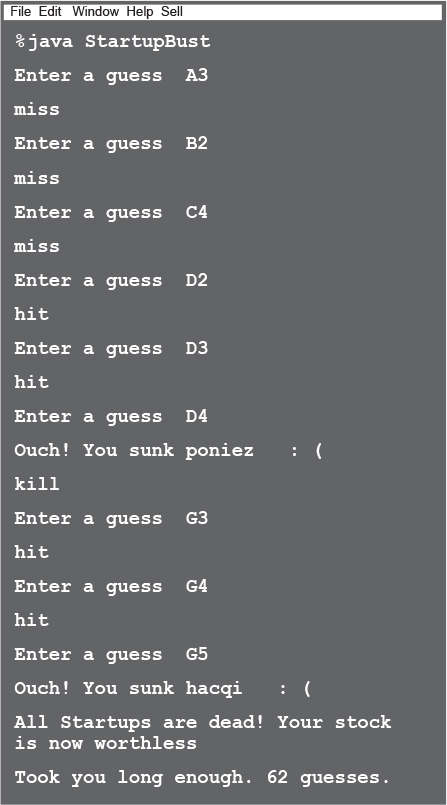
**Setup**: When the game program is launched, the computer places three Startups on a **virtual 7 x 7 grid** сетка. When that’s complete, the game asks for your first guess.

**How you play:** We haven’t learned to build a GUI yet, so this version works at the command-line. The computer will prompt подсказывать 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 poniez” (or whatever the lucky Startup of the day is). When you’ve sent all three Startups to that big 404 in the sky, the game ends by printing out your rating.



**You’re going to build the Sink a Startup game, with a 7 x 7 grid сетка and three Startups. Each Startup takes up three cells.**

**part of a game interaction**



**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:

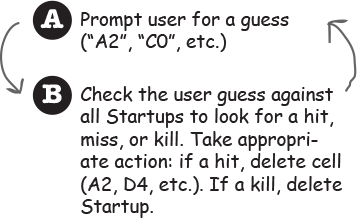
ImagesUser starts the game

ImagesGame creates three Startups

ImagesGame places the three Startups onto a virtual grid

ImagesGame play begins

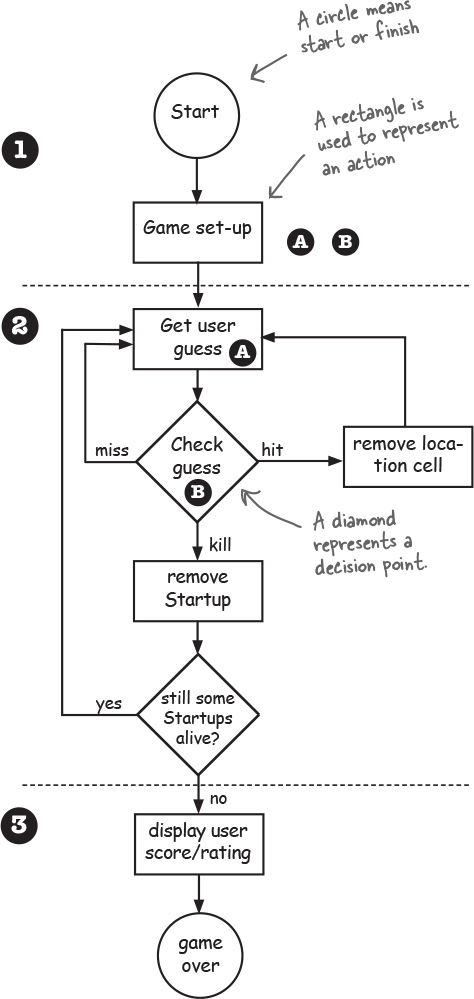
Repeat the following until there are no more Startups:



ImagesGame 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 Laura (who we met in [Chapter 2](#_Chapter_2._Classes)); focus first on the ***things*** in the program rather than the ***procedures***.



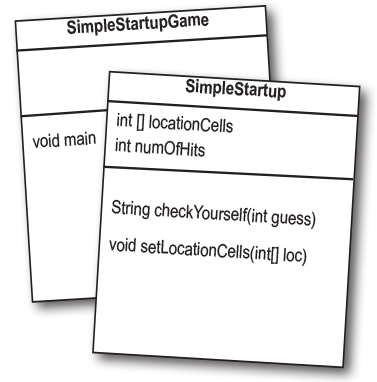
Whoa. A real flow chart блок-схема.

**The “Simple Startup Game”a gentler introduction**

It looks like we’re gonna need at least two classes, a Game class and a Startup class. But before we build the full monty ***Sink a Startup*** game, we’ll start with a stripped-down, simplified version, ***Simple Startup 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 сетка, we hide the Startup in just a single *row.* And instead of *three* Startups, we use *one*.

The goal is the same, though, so the game still needs to make a Startup instance, assign it a location somewhere in the row, get user input, and when all of the Startup’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 Startup instance, pick a location for it (three consecutive последовательный 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 Startup is hidden in three consecutive последовательный 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 Startup 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 Startup occupies Занимает.

Images**Game starts**, and creates ONE Startup and gives it a location on three cells in the single row of seven cells.

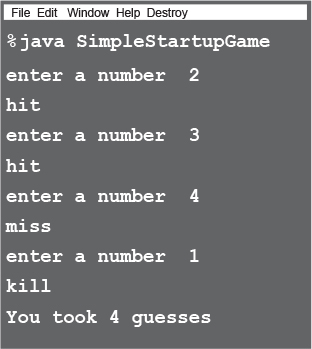
Instead of “A2”, “C4”, and so on, the locations are just integers (for example: 1,2,3 are the cell locations in this picture):



Images**Game play begins**. Prompt user Пригласить пользователя for a guess, then check to see if it hit any of the Startup’s three cells. If a hit, increment the numOfHits variable.

Images**Game finishes** when all three cells have been hit (the numOfHits variable value is 3), and the user is told how many guesses it took to sink the Startup.

**A complete game interaction**



**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:

ImagesFigure out what the class is supposed to *do*.

ImagesList the **instance variables and methods**.

ImagesWrite **prep code** for the methods. (You’ll see this in just a moment.)

ImagesWrite **test code** for the methods.

Images**Implement** the class.

Images**Test** the methods.

Images**Debug** and **reimplement** as needed.

ImagesExpress gratitude благодарность that we don’t have to test our so-called *learning experience* app on actual live users.

**Brain Power**

Images

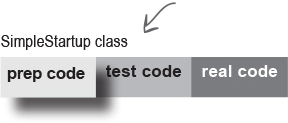
**Flex those dendrites.**

**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 prep code for the SimpleStartup 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 where we write real Java code.

**To Do:**

**SimpleStartup class**

Imageswrite prep code

Imageswrite test code

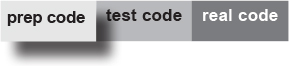
Imageswrite final Java code

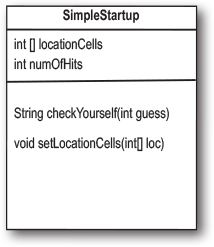
**SimpleStartupGame class**

Imageswrite prep code

Imageswrite test code [not needed]

Imageswrite final Java code





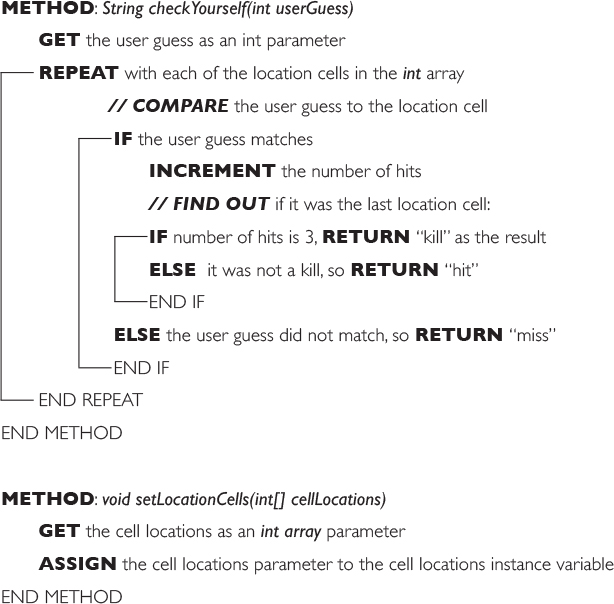
You’ll get the idea of how prep code (our version of pseudocode) works as you read through this example. It’s sort of half-way between real Java code and a plain English description of the class. Most prep code includes three parts: instance variable declarations, method declarations, method logic. The most important part of prep code 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 *int* 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.).





**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 Test-Driven Development (TDD), and it can make it easier (and faster) for you to write your code. We’re not necessarily saying you should use TDD but we do like the part about writing tests first. And TDD just *sounds* cool.



**Test-Driven Development (TDD)**

Back in 1999, Extreme Programming (XP) was a newcomer новичок to the software development methodology world. One of the central ideas in XP was to write test code before writing the actual code. Since then, the idea of writing test code first has spun off отпочковывается of XP and become the core of a newer, more popular subset of XP called TDD. (Yes, yes, we know we’ve just grossly oversimplified this, please cut us a little slack слабины here.)

TDD is a LARGE topic, and we’re only going to scratch the surface in this book. But we hope that the way we’re going about developing the “Sink a Startup” game gives you some sense of TDD.

Check out Test Driven Development: By Example by Kent Beck if you want to learn more about how TDD works.

Here is a partial list of key ideas in TDD:

- Write the test code *first*.

- Develop in iteration cycles.

- Keep it (the code) simple.

- Refactor (improve the code) whenever and wherever you notice the opportunity.

- Don’t release anything until it passes all the tests.

- Don’t put in anything that’s not in the spec (no matter how tempted you are to put in functionality “for the future”).

- No killer schedules; work regular hours.



**Writing test code for the SimpleStartup class**

We need to write test code that can make a SimpleStartup object and run its methods. For the SimpleStartup 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 prep code below for the *checkYourself()* method (the *setLocationCells()* method is a no-brainer Проще пареной репы setter method, so we’re not worried about it, but in a ‘real’ application we might want a more robust крепкий ‘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 prep code:**

**METHOD** *String checkYourself(int userGuess)*

**GET** the user guess as an int parameter

**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:**

Instantiate a SimpleStartup object.

Assign it a location (an array of 3 ints, like {2, 3, 4}).

Create an int to represent a user guess (2, 0, etc.).

Invoke Взывать the checkYourself() method passing it the fake user guess.

Print out the result to see if it’s correct (“passed” or “failed”).

image

**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’ заглушка' code that can compile, but that will always cause the test to fail (like, return null.)

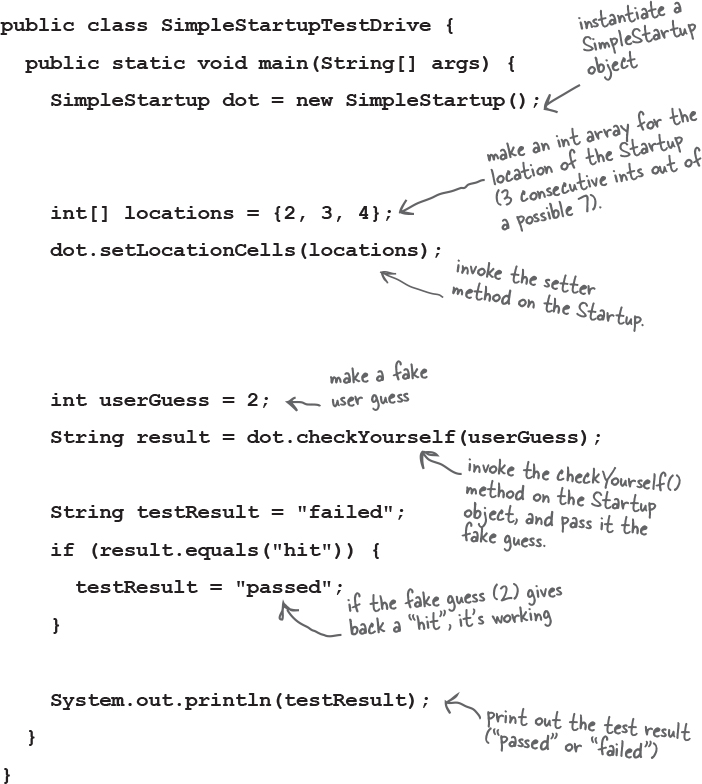
Q: **Then I still don’t see the point. Why not wait until the code is written, and then whip out вытащить the test code?**

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, to prove that your latest code additions don’t break previously-tested code.

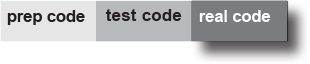
**Test code for the SimpleStartup class**



**Sharpen your pencil**

image

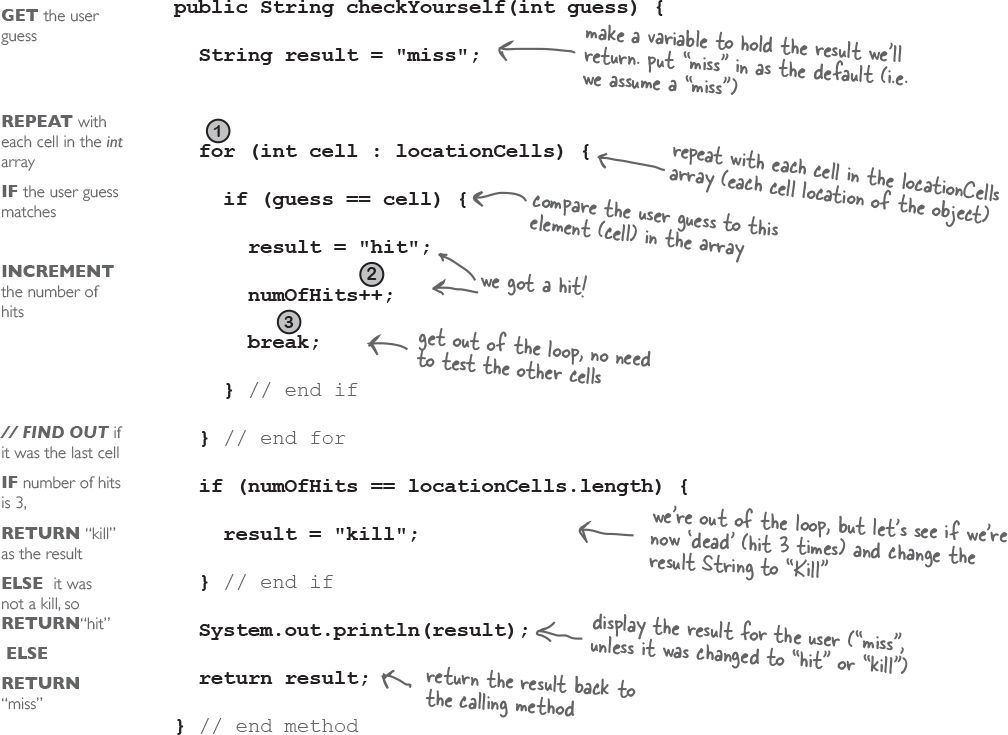
In the next couple of pages we implement the SimpleStartup class, and then later we return to the test class. Looking at our test code above, what else should be added? What are we *not* testing in this code, that we *should* be testing for? Write your ideas (or lines of code) below:



**The checkYourself() method**

There isn’t a perfect mapping from prep code to Java code; you’ll see a few adjustments. The prep code gave us a much better idea of *what* the code needs to do, and now we have to figure out 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 Imagesare for things (syntax and language features) you haven’t seen yet. They’re explained on the opposite page.





**Just the new stuff**

The things we haven’t seen before are on this page. Stop worrying! There are more details later in the chapter. This is just enough to get you going.



image

**there are no Dumb Questions**

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... since Java 5, you can also use the *enhanced* улучшенный 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 простой старый for loop to iterate over an array, but the *enhanced* for loop makes it easier.

Q: **If you can add one to an int by using ++, can you also subtract вычитать one in some way?**

A: Yep absolutely. Hopefully it’s not too surprising to find out that the syntax is “--” (two minuses), like this:

countdown = i--;

**Final code for SimpleStartup and SimpleStartupTestDrive**

public class **SimpleStartupTestDrive** {

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

SimpleStartup dot = new SimpleStartup();

int[] locations = {2, 3, 4};

dot.setLocationCells(locations);

int userGuess = 2;

String result = dot.checkYourself(userGuess);

String testResult = "failed";

if (result.equals("hit")) {

testResult = "passed";

}

System.out.println(testResult);

}

}

class **SimpleStartup** {

private int[] locationCells;

private int numOfHits = 0;

public void **setLocationCells**(int[] locs) {

locationCells = locs;

}

public String **checkYourself**(int guess) {

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

} // close class

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

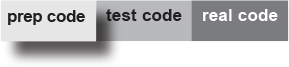
**What should we see when we run this code?**

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

% java SimpleStartupTestDrive

hit

passed



**Sharpen your pencil**

image

We built the test class, and the SimpleStartup 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 prep code 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 Startup is still alive :

**GET** user input from the command line

**The SimpleStartupGame needs to do this:**

Make the single SimpleStartup Object.

Make a location for it (three consecutive последовательный cells on a single row of seven virtual cells).

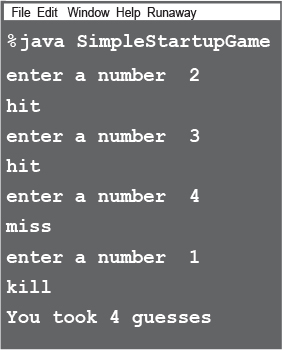
Ask the user for a guess.

Check the guess.

Repeat until the Startup is sunk .

Tell the user how many guesses it took.

**A complete game interaction**



image

**Prep code for the SimpleStartupGame class**

**Everything happens in main()**

There are some things you’ll have to take on faith. For example, we have one line of prep code 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 prep code, 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 SimpleStartup 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 SimpleStartup instance

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

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

**GET** user input from the command line

***// CHECK*** the user guess

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

**INCREMENT** *numOfGuesses* variable

***// CHECK*** for Startup 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:

***prep code***

***test code***

***real (Java) code***

Prep code should describe *what* to do, not *how* to do it. Implementation comes later.

Use the prep code to help design the test code.

A class can have one superclass only.

Write test code *before* you implement the methods.

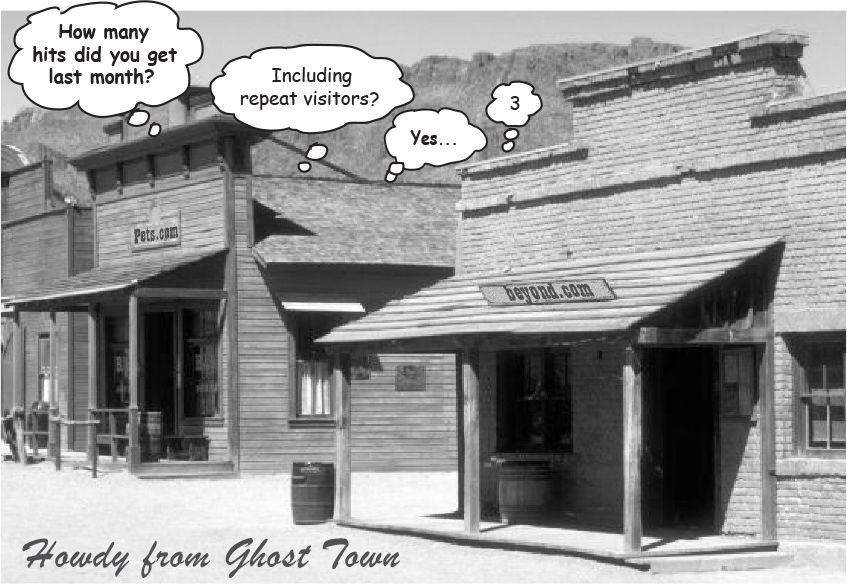
Choose *for* loops over *while* loops when you know how many times you want to repeat the loop code.

The enhance улучшенный for loop is an easy way to loop over an array or collection.

Use the *increment* operator to add 1 to a variable (x++;)

Use the *decrement* operator to subtract 1 from a variable (x--;)

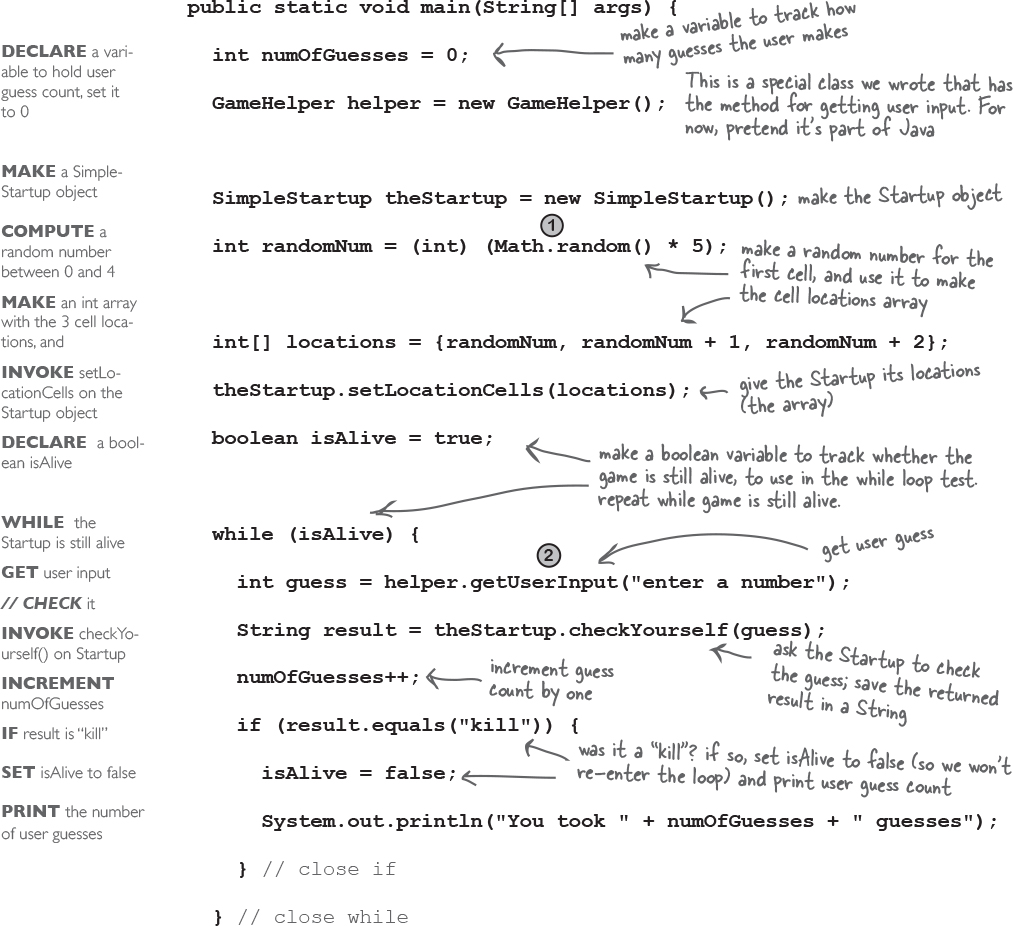
Use *break* to leave a loop early (i.e. even if the boolean test condition is still true).



image

**The game’s main() method**

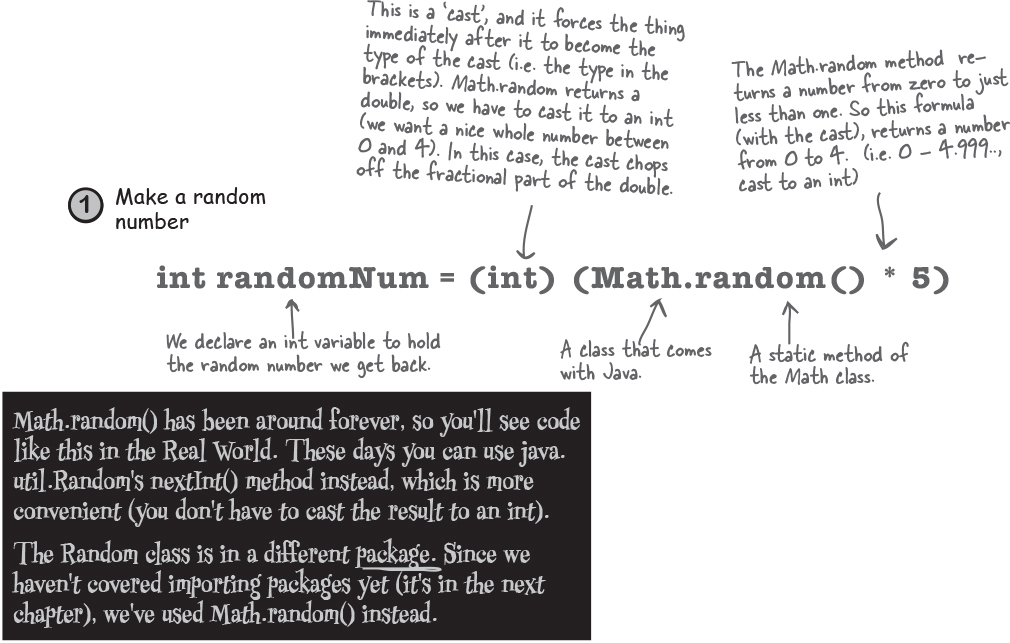
Just as you did with the SimpleStartup class, be thinking about parts of this code you might want (or need) to improve. The numbered things Imagesare for stuff we want to point out. They’re explained on the opposite page. Oh, if you’re wondering why we skipped the test code phase for this class, we don’t 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, we’ll just run this to test it.

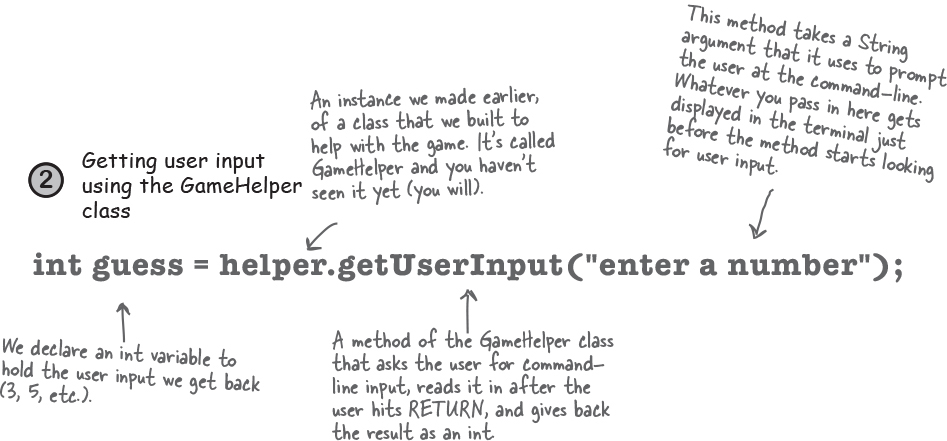


image

**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.





image

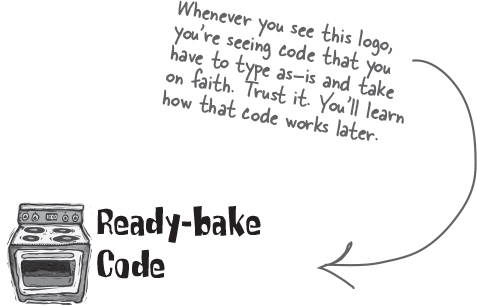
**One last class: GameHelper**

**We made the** ***Startup*** **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 topics best left for later. (Later, as in [Chapter 16](#_Chapter_16._Serialization).)

Just copy\* the code below and compile it into a class named GameHelper. Drop all three class files (SimpleStartup, SimpleStartupGame, GameHelper) into the same directory, and make it your working directory.





**import java.util.Scanner;**

**public class GameHelper {**

**public int getUserInput(String prompt) {**

**System.out.print(prompt + ": ");**

**Scanner scanner = new Scanner(System.in);**

**return scanner.nextInt();**

**}**

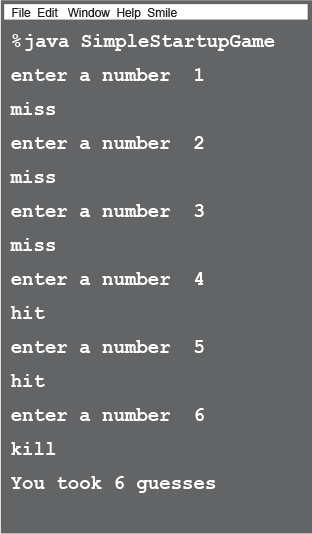
**}**

\*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 [*https://oreil.ly/hfJava\_3e\_examples*](https://oreil.ly/hfJava_3e_examples).

**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)

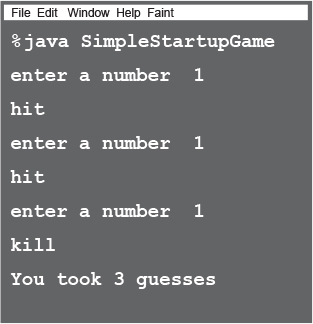


**What’s this? A bug ?**

**Gasp!**

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

**A different game interaction** (yikes)



**Sharpen your pencil**

image



**It’s a cliff-hanger!**

Will we ***find*** the bug?

Will we ***fix*** the bug?

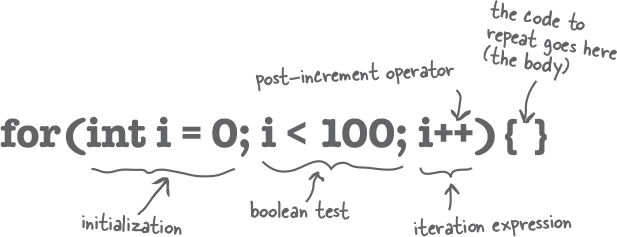
Stay tuned for the next chapter, where we answer these questions and more...

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

**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’ve seen this kind of syntax in another programming language, just skim снимать these last few pages...

**Regular (non-enhanced) for loops**



**What it means in plain English:** “Repeat 100 times.”

**How the compiler sees it:**

Imagescreate a variable *i* and set it to 0.

Imagesrepeat while *i* is less than 100.

Imagesat 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 it’s much more common to use a single variable.

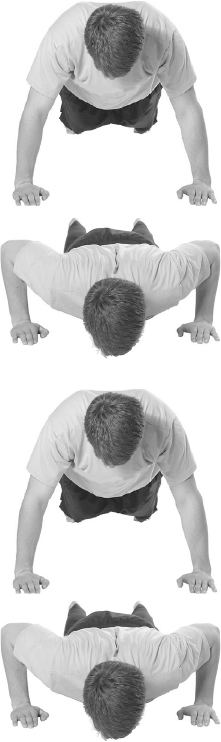
**Part Two:** ***boolean test***

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.

**repeat for 100 reps:**



**Trips through a loop**

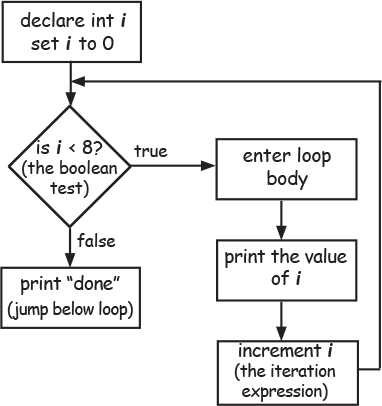
Images

**for (int i = 0; i < 8; i++) {**

**System.out.println(i);**

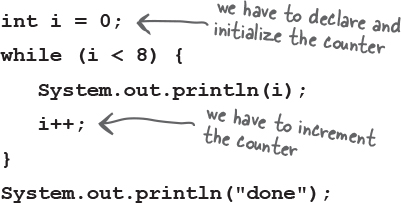
**}**

**System.out.println("done");**

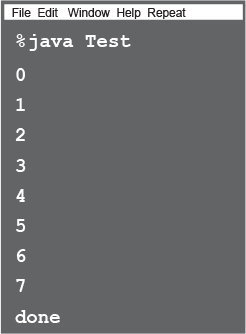


**Difference between for and while**

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:*



**output:**



++ --

**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 a single statement.

**int x = 0;    int z = ++x;**

produces: x is 1, z is 1

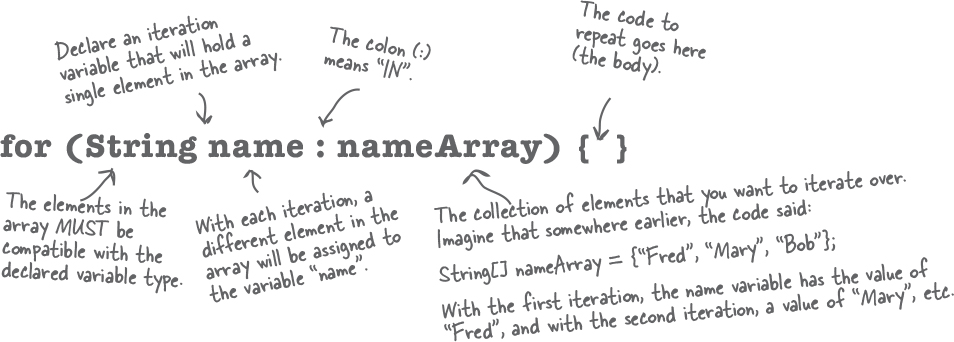
But putting the ++ *after* the x gives you a different result:

**int x = 0;    int z = x++;**

Once this code has run, x is 1, but ***z is 0***! z gets the value of x and *then* x is incremented.

**The enhanced for loop**

The Java language added a second kind of *for* loop called the *enhanced for* back in Java 5. This 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. We’ll see the *enhanced for loop* in the next chapter too, when we talk about collections that *aren’t* arrays.



**What it means in plain простом 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:**

ImagesCreate a String variable called *name* and set it to null.

ImagesAssign the first value in *nameArray* to name.

ImagesRun the body of the loop (the code block bounded by curly braces).

ImagesAssign the next value in *nameArray* to name.

ImagesRepeat while *there are still elements in the array.*

**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***

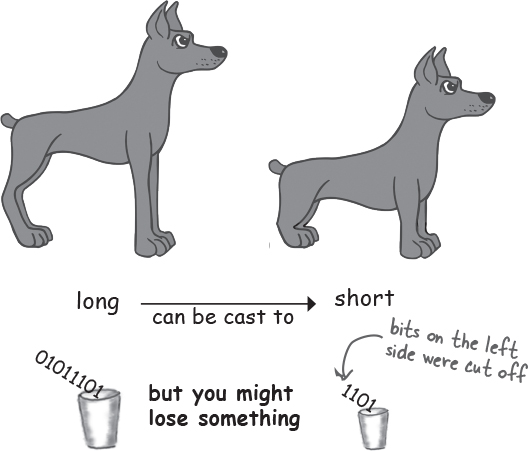
This must be a reference to an array or other collection. Again, don’t worry about the *other* non-array kinds of collections yet—you’ll see them in the next chapter.

**Note**

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...”

**Casting primitives**

Before we finish the chapter, we want to tie up a loose end. When we used Math.random() we had to *cast* the result to an int. Casting one numeric type to another can change the value itself. It’s important to understand the rules so you’re not surprised by this.

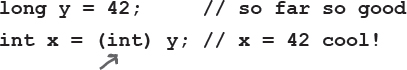


In [Chapter 3](#_Chapter_3._Primitives) we talked about the sizes of the various primitives, and how you can’t shove толкать a big thing directly into a small thing:

**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 partying гулянка with the other longs, and taking on really big values. To force the compiler to jam заглушить the value of a bigger primitive variable into a smaller one, you can use the cast operator. It looks like this:



Putting in the cast tells the compiler to take the value of y, chop рубить 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.*

**Exercise**



**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 output = new Output();

output.go();

}

void go() {

int value = 7;

for (int i = 1; i < 8; i++) {

value++;

if (i > 4) {

System.out.print(++value + " ");

}

if (value > 14) {

System.out.println(" i = " + i);

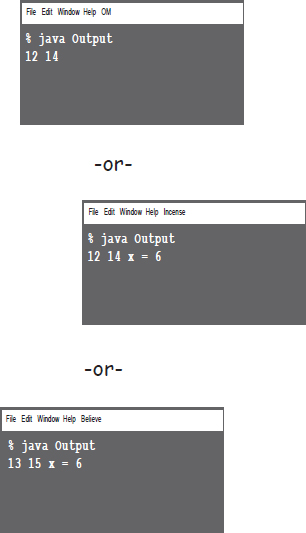
break;

}

}

}

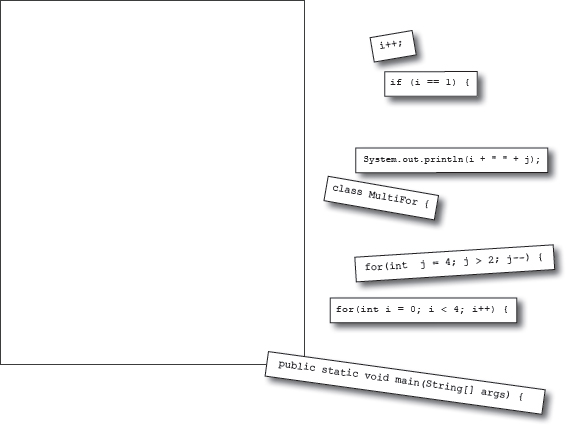
}

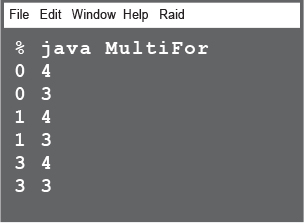


**Code Magnets**



A working Java program is all scrambled зашифрованный 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!

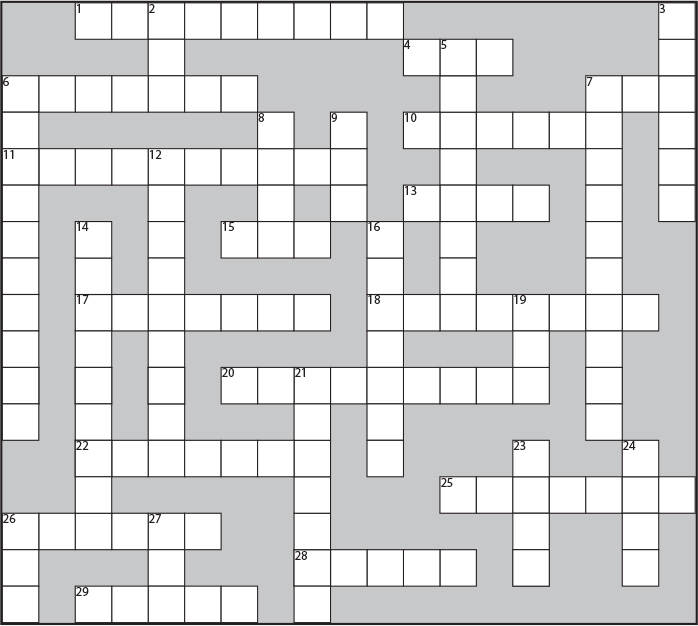




**JavaCross**



How does a crossword puzzle help you learn Java? Well, all of the words **are** Java related. In addition, the clues provide metaphors, puns Каламбуры, and the like. These mental twists Повороты 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. Prep code-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. Iterate over me

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

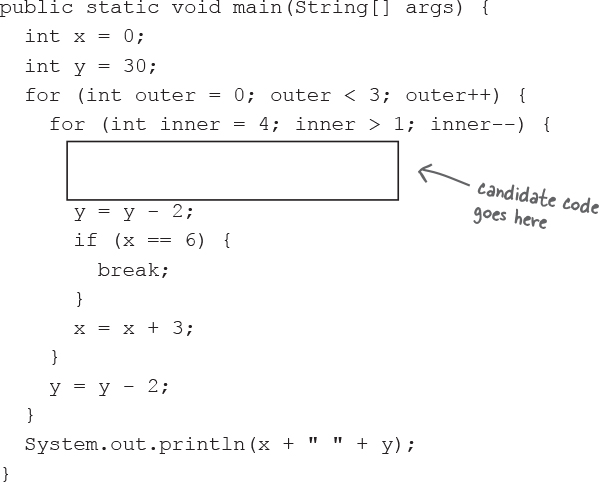
26. Compile it and \_\_\_\_

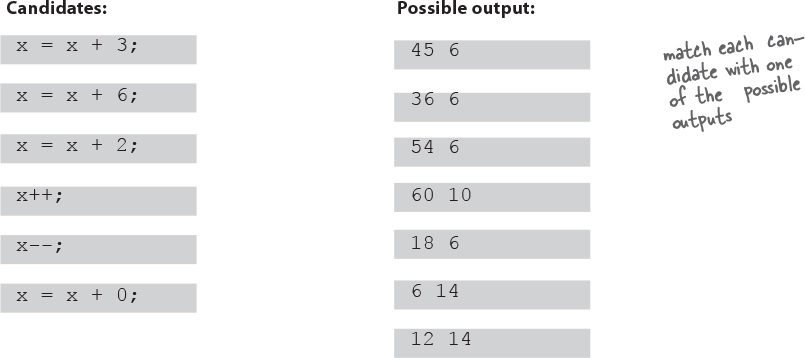
27. ++ quantity

**Mixed Messages**



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 outpu**t 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.





### Exercise Solutions



**Be the JVM:**

class Output {

public static void main(String[] args) {

Output output = new Output();

output.go();

}

void go() {

int value = 7;

for (int i = 1; i < 8; i++) {

value++;

if (i > 4) {

System.out.print(++value + " ");

}

if (value > 14) {

System.out.println(" i = " + i);

break;

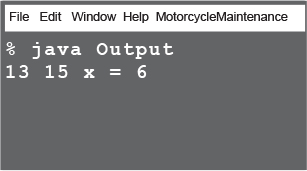
}

}

}

}

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



**Code Magnets:**

class MultiFor {

public static void main(String[] args) {

for (int i = 0; i < 4; i++) {

for (int j = 4; j > 2; j--) {

System.out.println(i + " " + j);

}

if (i == 1) {

i++;

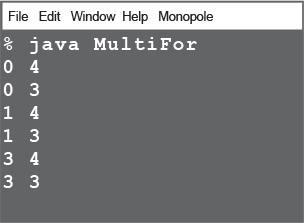
}

}

}

}

**What would happen if this code block came before the ‘j’ for loop?**



**Puzzle Solutions**



