



Chapter 5

Writing a program

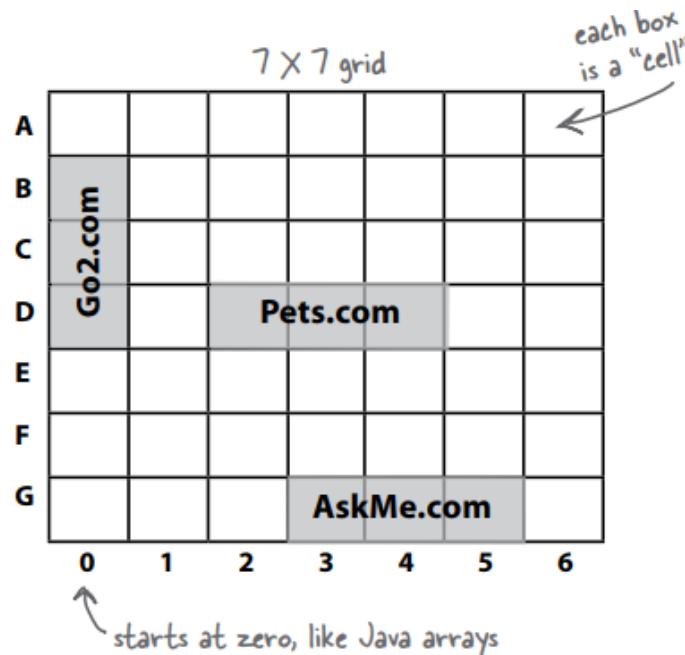
- We already learned to make objects, declare instance variables, write some methods, and encapsulation.
- We can employ more tools to strengthen our program, such as loops, operators, type conversions.
- Let's integrate these tools to make a game - battleship.





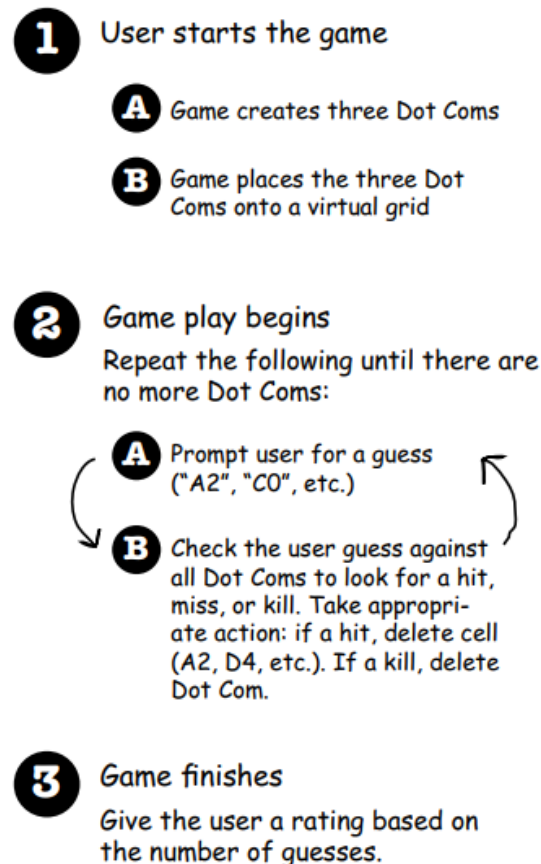
- It's you against the computer, but unlike the real Battleship game, in this one you don't place any ships of your own. Instead, your job is to sink the computer's ships in the fewest number of guesses. We called it "Sink a Dot Com"
- Goal: Sink all of the computer's Dot Coms in the fewest number of 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

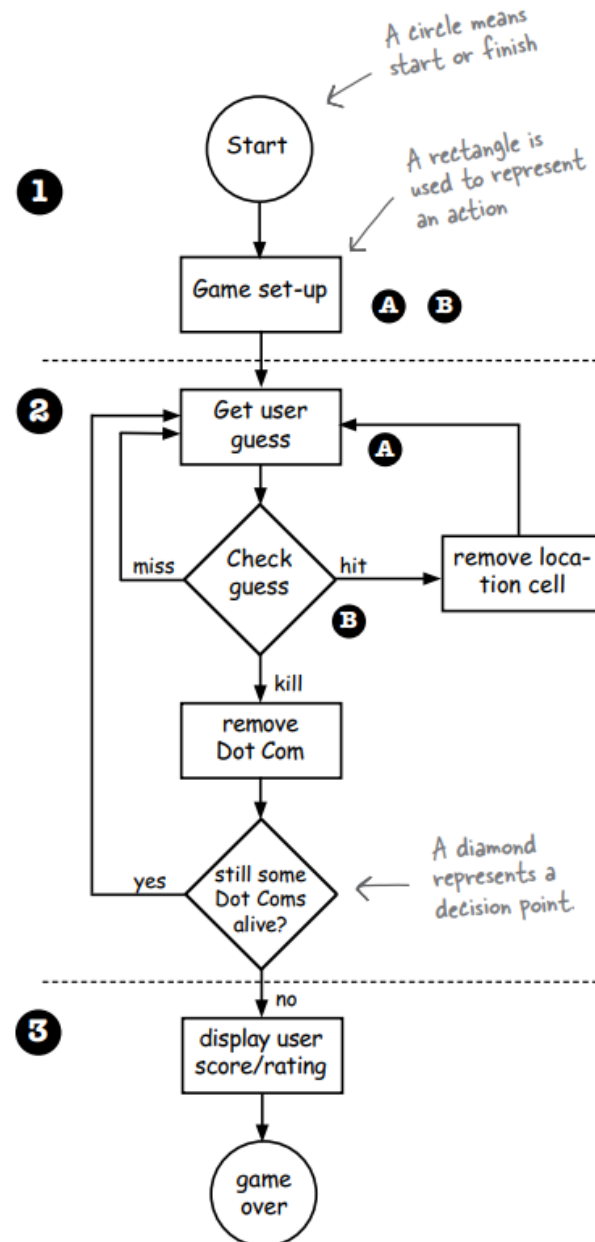




- First, a high-level design
- We need more information about what the game should do. Then we can design objects and methods to accomplish it.
- We can start from writing down a step-by-step instructions:

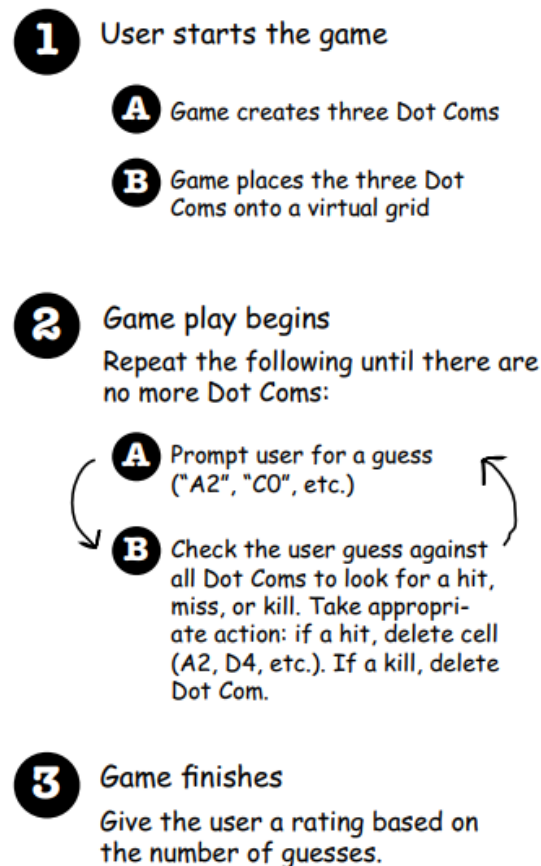


- Then, convert these steps to a flowchart.





- 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*

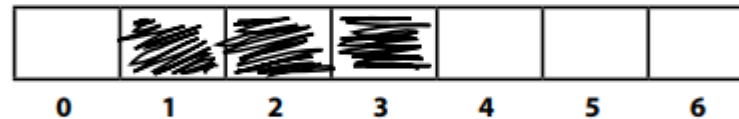


- First we create a simplified version:
 - The map is a 1x7 grid with one ship
 - Create one single DotCom instance, pick a location for it (three consecutive cells on the single virtual seven-cell row)
 - Ask the user for a guess, check the guess, and repeat until all three cells have been hit.
- Keep in mind that the virtual row is... *virtual*. 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 *the three cells where the DotCom occupies*.

- Sink Dot Com – simplified version

- 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.

- Developing a class
 1. Figure out what the class is supposed to do.
 2. List the instance variables and methods.
 3. Write pseudo code (called prep code in the textbook) for the methods.
 4. Write test code for the methods.
 5. Implement the class.
 6. Test the methods.
 7. Debug and reimplement as needed.

- 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.



- Pseudo code
- Most pseudo 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 translate into *how* when we actually write the method code (the **real code**).



- Let's try to convert this flow to prep code...

- 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.

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



- Test code
- 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.
- The goal of this class is to teach you learn OO, not software engineering, so I leave this topic for you. You can read more about XP in page 101.
- You don't have to use XP to develop your project, some people just find XP really helpful.

- We need to write test code that can make a SimpleDotCom object and run its methods
- 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
  
```




```

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);

        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)

What else should be added? What are we *not* testing in this code, that we should be testing for?

- Q: how exactly do you run a test on something that doesn't yet exist?
- A: We never said you start by **running** the test; you start by writing the test
- Q: 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.
- 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

- Real code
- The prep code 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*.

METHOD: *String checkYourself(String userGuess)*

GET the user guess as a String parameter

CONVERT the user guess to an *int*

You need **Integer.parseInt()**

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

You need a for loop

// 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



```

public String checkYourself(String stringGuess) {
    int guess = Integer.①parseInt(stringGuess); ← convert the String to an int

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

    ②
    for (int cell : locationCells) { ← repeat with each cell in the locationCells array (each cell location of the object)

        if (guess == cell) { ← compare the user guess to this element (cell) in the array

            result = "hit";
            ③ numOfHits++; ← we got a hit!

            ④ break; ← get out of the loop, no need to test the other cells
        } // end if

    } // end for

    if (numOfHits == locationCells.length) {
        result = "kill"; ← 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"
    } // end if

    System.out.println(result); ← display the result for the user
    return result; ← ("Miss", unless it was changed to "Hit" or "Kill")
} // end method ← return the result back to the calling method
    
```



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.)



- Q: What happens in `Integer.parseInt()` if the thing you pass isn't a number?
- A: If you try to parse something like "two", the code will blow up at runtime. (In official, it means the program throws an exception and crash)
- Q: What's the difference between the for loop used here and the traditional one?
- A: The loop syntax used here is called enhanced for loop. It's very convenient when your loop needs to iterate over the elements in an array (because you don't have to care about how many elements are there in that array)

- About for loops
- This is the traditional for loop

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

Initialization boolean test iteration expression post-increment operator the code to repeat goes here (the body)

- The enhanced for loop provides a simpler way to walk through all the elements in the array (or collection)

`for (String name : nameArray) { }`

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).

★ 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.



- How the compiler sees it (i.e., the enhanced for loop):
- 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: some programming language refer to the enhanced loop as the "for each" or "for {SOMETHING} in {ANOTHER_THING}"

• Final code for the tester

```

public class SimpleDotComTester {
    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);
        String testResult = "failed";
        if (result.equals("hit"))
        {
            testResult = "passed";
        }
        System.out.println(testResult);
    }
}
    
```

```

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
    
```

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
passed
    
```

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.



- Start your program with a high-level design
- You'll write 3 things when you create a new class: prep code, test code, and real code.
- Prep code describes what to do, not how to do it.
- Design your test code according to the prep code, write test code *before* you implement the methods
- You can use the enhanced loop if you don't need to know the indices of your array object.

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

hint: the `setLocationCell()` has to be used here.

WHILE the dot com is still alive :

GET user input from the command line

Should behave like the command line interface on the right.

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
```

Note: write down your ideas here, not the real code.

```
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

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 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

→ You need a random number generator

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

GET user input from the command line

→ You need a helper function to get user input

// 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



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



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)

① Make a random number

```
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.

- How to get the helper object?
- We provide the code but we do not explain it here because it contains too many topics

```

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;
    }
}
    
```




What's this? A *bug*?

Gasp!

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
```

- Go back to review the code. Can you spot the bug? Can you come up with any ideas to fix it?

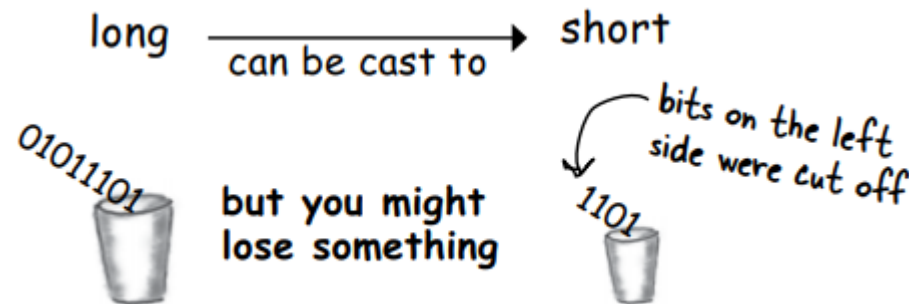
- Casting primitives
- Recall: in chapter 3 we talked about the sizes of the various primitives, and said Java doesn't allow you to shove a big thing directly into a small thing

```
long y = 42;  
int x = y;    // won't compile
```

- To force the compiler to jam the value of a bigger primitive variable into a smaller one, you can use the cast operator

```
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!
    
```


 Quiz: do you know why x is this value?

- 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 Incense
% java Output
12 14 x = 6
    
```

-or-

```

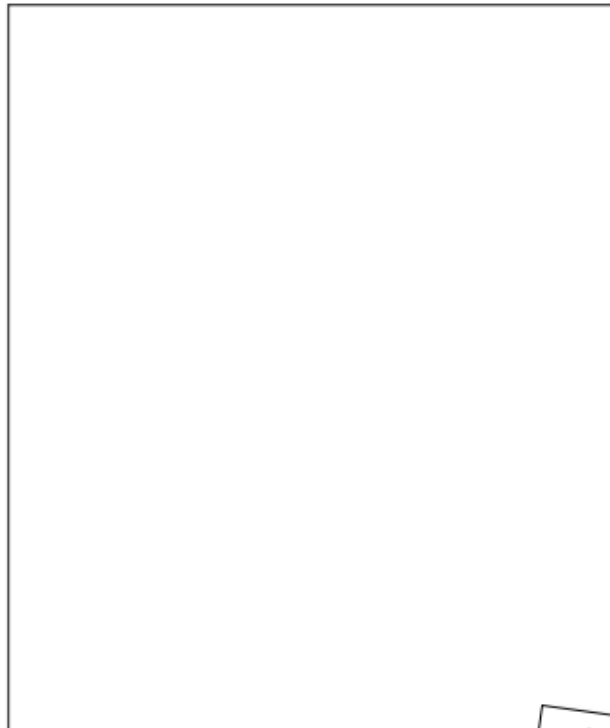
File Edit Window Help Believe
% java Output
13 15 x = 6
    
```



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 Run
% java MultiFor
0 4
0 3
1 4
1 3
3 4
3 3
```



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