# *Head First Java: Chapter 5 Notes*

*August 27, 2019*

## *Flow and Principles of Java Programming*

Here a simple Java program will be used to example the flow of writing Java programs. This program is a simplified version of the "Battleship" game called "Simple Dot Com", but in two dimension (a row), and there is only one 3-cell long target. It is recommended when writing Java code, the coder follow these steps:

1. Prep Code — This is essentially the pseudo-code for Java. Most prep code contains three parts:

   (a) Instance Variable Declarations. For example:

   ```
   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.
   ```

   (b) Method Declarations. For example:

   ```
   DECLARE a checkYourself() method that takes a String
   for the user's guess, checks it and return 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.
   ```

   (c) Method Logic. For example for the above two methods, the method logic prep code would be:

   ```
   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 his is 3
                   RETURN ''Kill'' as the result
               ELSE it was not a so 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 in int array parameter
    ASSIGN the cell locations parameter to the cell locations
        in stance variable
END METHOD
```

Based on this prep code, we can determine the *test code*.

2. Test Code — Writing the test code before the real code helps with proper code structure. What the test code should test is self-evident from the prep code. For example, for the `checkYourself()` method above:

   (a) Instantiate a `SimpleDotCom` object (this is the target and will hold the `main` method).

   (b) Assign the `SimpleDotCom` object a location (the `locationCells` instance variable), which is an array of 3 `int`.

   (c) Create a String (`userGuess`) to represent a user guess.

   (d) Invoke `checkYourself()` with fake user guess as argument.

   (e) Print out the result to see if it is correct.

   Note that because the actual program (*i.e.* the *real code*) hasn't been written yet, the test code written beforehand probably would not compile or run. But writing the test code first helps with structuring the actual program later. Given the outline above, the test code would be:

```
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);
    }
}
```

3. Real Code — Once the test code is written, that should give an idea of what is needed in the real code. Based on the scaffolding of the prep code, the actual code of the program can be written. For example, the test code included the `checkYourself();` method that check to see if the user guess hit the cell location:

```
public String checkYourself(String stringGuess) {
    int guess = Interger.parseInt(stringGuess);
    String result = ''miss'';
    for (int cell : locationCells) {
        if (guess == cell) {
            result = ''hit'';
            numOfHits++;
            break;
        }
    }
    if (numOfHits == locationCells.length) {
        result = ''kill'';
    }
    System.out.println(result);
    return result;
}
```

## *For Loops*

There are two types of `for` loops in java:

1. Regular For Loops — These are primitive for loops that manually tells Java how many times to loop:

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

   For example the above for loop tells Java to repeat code within the `{}` for 100 times. There are three parts of these regular for loops

   (a) Initialization — Declare and initialize a variable to use within the loop body (*i.e. a counter*). More than one variable can be initialized here. `int i = 0;`

   (b) Boolean Test — The condition in which the loop will end. Because Java evaluates this as a Boolean test i must resolve into `true` or `false`. `i < 100;`

   (c) Iteration Expression — At the end of each loop, whatever written here will happen. `i++;`

2. Enhanced For Loops — Only available Java 5.0+. These are conceptually similar to Python for loops that iterates over an array (Python lists, etc.):

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

   This for loop loops over each element in the array `nameArray`. There are two parts to an enhanced for loop:

   (a) Iteration Variable Declaration — Declare and initiate a variable to use within the loop body. Similar in concept with Python

loops, except the type must also be specified (usually String, but really can be any types.)

(b) Iterable Collection — The array for which to iterate over. Non-array collections can also be used here.

## *Casting Primitives and Implicit Widening*

Recall that different variable types have different bit depth. Implicit Widening refer to when a smaller bit depth variable is assigned to a larger bit depth variable or placed into an array of a larger bit depth variable. For example:

```
byte x = 4;
int y = 7;
y = x;

jshell> y ==> 4
```

Here, a x is a `byte` type variable, which is a shallower bit depth than y which is an `int` type variable. Therefore when the value of x was assigned to y, this is a legal operation because a byte "fits" into an int, due to implicit widening. The opposite is not true:

```
byte x = 4;
int y = 7;
x = y;

jshell> Error:
|  incompatible types: possible lossy conversion from int to byte
|  x = y
|      ^
```

Java raised an exception here because an int cannot "fit" into a "byte".

In order for a variable with a deeper bit depth to "fit" into a shallower one, the variable has to be explicitly *cast* into a variable type. Consider the same example:

```
byte x = 4;
int y = 7;
x = (byte) y;
jshell> x ==> 7
```

The line of code `x = (byte) y;` tells the Java compiler to *cast* the value of y (in int type) into a byte type (the type for x) by truncating the value to fit into the smaller type. This can lead to unexpected behavior if the variable value that is being cast is larger than the maximum size of the variable type being cast to. Finally, variable casting is a convenient way to truncate decimal numbers to whole numbers by casting either a double or a float to a byte/int/short/long.