Java Loops & Methods

The *while* loop

Syntax:

while (condition is true **) {**
    do these statements
}

Just as it says, the statements execute while the condition is true. Once the condition becomes false, execution continues with the statements that appear after the loop.

Example:

```
int count = 1;
while (count <= 10) {
    out.println(count);
    count = count + 1;
}
```

This loop prints out the numbers from 1 through 10 on separate lines. How does it work?

Output:

```
1
2
3
4
5
6
7
8
9
10
```

This is an example of a *counting* loop. They are called this because the loop is just counting through the values of the *loop control variable* (LCV), which in this case is called count. It executes a definite number of times.

while loops can also be used as *indefinite* loops – when you can't pre-determine how many times the loop will execute. For example, let's take this problem.

A colony of 800 puffins is increasing at the rate of 4% annually. When will their population first exceed 1200?

Without doing the calculations, can we determine how many times this loop will run? No! We'll use a while loop to solve it, though:

```java
import components.simplewriter.SimpleWriter;
import components.simplewriter.SimpleWriter1L;

/**
 * A colony of 800 Puffins is increasing at the rate of 4% annually. When will
 * their population first exceed 1200?
 *
 * @author Chris Kiel
 *
 */
public final class PuffinColony {

    /**
     * Private constructor so this utility class cannot be instantiated.
     */
    private PuffinColony() {
    }

    /**
     * Main method.
     *
     * @param args
     *            the command line arguments
     */
    public static void main(String[] args) {
        final double GROWTH_RATE = 0.04;
        SimpleWriter out = new SimpleWriter1L();
        int population, year;

        //Initializations
        population = 800;
        year = 0; //zero years from now; i.e., the present time
        while (population <= 1200) {
            //we do not want fractional Puffins!
            population = (int) (population * (1 + GROWTH_RATE));
            year++;
        }

        //Output results
        out.println("The population first goes over 1200 after " + year
                + " years.");
        out.println("After that time, the population is " + population + ".");

        /*
         * Close output stream
         */
        out.close();
    }
}
```

```
/* Output
 * The population first goes over 1200 after 11 years.
 * After that time, the population is 1225.
 */
```

Infinite Loops

A loop is called an *infinite loop* if it never ends. Obviously, we don't want to have these!
Examples:

```
int count = 1;
while (count <= 10) {
     out.println(count);
     count = count – 1;
}
```

Why is this an infinite loop? count starts at 1, then goes to 0, -1, -2, etc., never getting greater than 10, which is needed to make the condition false! So this loop never ends.

```
int count = 1;
while (count != 10) {
     out.println(count);
     count = count + 2;
}
```

Why is this one an infinite loop? count starts at 1, then goes to 3, 5, 7, 9, 11, etc., never equalling 10, so the loop is infinite.

```
int count = 1;
while (count <= 10); {
     out.println(count);
     count = count + 1;
}
```

Did you see what makes this an infinite loop? Look after the condition. See the semicolon? Remember that the ; is a statement terminator. What the compiler determines is this:

```
int count = 1;
while (count != 10) {
     //do nothing
}
```

Since nothing is taking place inside the loop – which is an empty statement followed by a ; - the condition never becomes false and the loop is infinite.

Important while loop rules:
1. The condition must eventually become false.
2. Never put a semicolon after the condition unless you want an infinite loop.

So what if you accidentally write an infinite loop in your program? Everyone does at some point! Often your only clue is lines of output streaming by on the Console or you see nothing at all. All you have to do is either press <Ctrl-C> or click the red square in the Console bar to stop the program.

The *while* loop is the generic loop; every loop can be written as a *while* loop.

The *for* loop

A **for** loop is a counting loop.

for (*initializer* **;** *loop entry condition* **;** *updater* **) {**
        do these statements
}

The initializer gives an initial value to the loop control variable (LCV)
The loop entry condition must be true for the loop to execute
The updater helps the LCV to reach a state when the condition will eventually become false

Examples:
```
for (int count = 1; count <= 100; count++){
     out.println("Hello");
}
```
Semantics:
count initialized to one
1. condition checked; is true
2. println executed
3. count incremented
4. repeat steps 1-3 until condition is false (count > 100), then continue to next statement after loop

Note: if you have a one-statement loop, you don't need the curly braces. However, while you are learning how to write these structures, I recommend you use them!

The increment does not have to be 1:
```
        for (int count = 1; count <= 100; count+=2) {
            out.println("Hello");
        }
```

Another well-known example :

```
        int count;
        for (count = 100; count >= 0; count--) {
            out.println(count + " bottles of beer on the wall");
        }
```

What is the value of count after the end of this loop?

For loops can be skipped if initial condition is false:

```
        for (int count = 10; count < 10; count--) {
            out.print(count);
        }
        out.println();
```

Notice the use of the increment and decrement shortcut operators in these loops. They are not required, but most programmers use them.

Loop Variable Scope

If *count* is declared within the *for* loop, it cannot be used after the for statement:

```
        for (int count = 1; count <= 100; count++) {
            out.println("Hello");
        }
        out.println(" count = " + count);
        // Syntax error, count undeclared
```

If *count* is declared before the *for* statement, it can be used after the for loop:

```
        int count; // Declare the loop variable here
        for (count = 1; count <= 100; count++) {
            out.println("Hello");
        }
        out.println("count = " + count); // So it can be used here
```

Nested Loops

You can nest one structure inside of another. We have seen that you can nest one if statement inside of another; not surprisingly, we can also nest one loop inside of another.

Think of them like gears; the outer one goes slowly, the inner one goes quickly. In other words, for every execution of the outer loop, there are multiple executions of the inner loop.

Suppose you wanted to print the following table:

```
1    2    3    4    5    6    7    8    9
2    4    6    8    10   12   14   16   18
3    6    9    12   15   18   21   24   27
4    8    12   16   20   24   28   32   36
```

You could use a nested *for* loop. The *outer loop* prints the four rows and in each row, the *inner loop* prints the nine columns.

```
        // For each of 4 rows
        for (int row = 1; row <= 4; row++) {
            for (int col = 1; col <= 9; col++) {
                out.print(col * row + "\t"); //Print one row
            }
            out.println(); // Start a new row
        }
```

Make a table to show the relationship between the row and column variables needed to print the following triangular pattern:

```
#####
####
###
##
#
```

You could use the following nested *for* loop.

```
        for (int row = 1; row <= 5; row++) {
            for (int col = 1; col <= 6 - row; col++) {
                out.print('#');
            }
            out.println();
        }
```

Loop Review

Three loops: for, while, do …while

Which do you use? Depends on what you need to do:

Do you need to produce values like 1..10, 50..100, etc? ➔ counting loop ➔ for loop

Is it not determinable in advance how many times the loop will run? ➔ indefinite loop ➔ while loop

How to decide what loop to use

- If you know ahead of time exactly how many times loop should be executed, use a *for* loop. You can "know" this if you know the constant (e.g., 5) or there is a variable containing the number of times to execute the loop (e.g., numTimes). For loops are also very good for numerical calculations and counting loops, and are optimized structures in the compiler.
- If the loop might need to be skipped, use a while loop
- Remember that all loops can be written as while loops, but the reverse is not true.

Common loop errors

- empty loops

```
for (int count=1; count<=10; count++); //Notice the ;

while(count<=10);
```

Neither of these loops has any statements inside of it, so nothing happens in them!

- loop boundary problems

```
//how many times do these loops execute?
for (int count = 1; count <= 10; count++)
for (int count = 1; count < 10; count++)
for (int count = 0; count < 10; count++)
for (int count = 10; count > 10; count--)
```

Common problem: Finding max or min

- Finding the max/min requires certain things:
  - declare variables for max/min
  - initialize max/min **before** the loop
  - initialize max/min in one of two ways:
    - by setting them to the first data value
    - by setting them to the predefined constant `Integer.MIN_VALUE` or `Integer.MAX_VALUE`
  - inside the loop, check each data value, changing max/min as required
  - after the loop is the time to display the values for max/min
- We will use a sentinel loop, which stops when it checks for a *sentinel* value (some value that does not belong to data set)
- **Sentinel loop format:**

```
get data value
while (value != sentinel){
     //process value
     . . .
     get data value
}
```

```java
import components.simplereader.SimpleReader;
import components.simplereader.SimpleReader1L;
import components.simplewriter.SimpleWriter;
import components.simplewriter.SimpleWriter1L;

/**
 * Finds max & min values from ints in a file.
 *
 * @author Chris Kiel
 *
 */
public final class MinMax {

    /**
     * Private constructor so this utility class cannot be instantiated.
     */
    private MinMax() {
    }
```

```java
/**
 * Prompts for and returns int entered by user.
 *
 * @param in
 *            the input stream
 * @param out
 *            the output stream
 * @return int input from user.
 *
 */
private static int getInput(SimpleReader in, SimpleWriter out) {
    out.print("Enter a number, -1 to quit: ");
    return in.nextInteger();
}

/**
 * Calculates and returns average.
 *
 * @param sum
 *            sum of ints
 * @param count
 *            count of ints
 * @return average, zero if empty
 *
 */
private static double calcAvg(int sum, int count) {
    double avg = 0;

    if (count > 0) {
        avg = (double) sum / count;
    }

    return avg;
}
```

```java
/**
 * Prints results of calculations.
 *
 * @param out
 *            the output stream
 * @param sum
 *            sum of ints
 * @param count
 *            count of ints
 * @param max
 *            max of ints
 * @param min
 *            min of ints
 */
private static void printResults(SimpleWriter out, int sum, int count,
        int max, int min) {
    out.println("\nThere were " + count + " numbers.");
    out.println("The maximum of the numbers was " + max);
    out.println("The minimum of the numbers was " + min);
    out.println("The sum of the numbers was " + sum);
    out.println("The average of the numbers was " + calcAvg(sum, count));
}

/**
 * Main method.
 *
 * @param args
 *            the command line arguments
 */
public static void main(String[] args) {
    SimpleReader in = new SimpleReader1L();
    SimpleWriter out = new SimpleWriter1L();
    int number, sum, count;
    int max, min;

    //initializations
    sum = 0;
    count = 0;
    max = Integer.MIN_VALUE; //initialize to smallest int possible
    min = Integer.MAX_VALUE; //initialize to largest int possible
```

```
        /*
         * Use sentinel loop to get data, determine max, min, sum & count
         */

        //Priming read
        number = getInput(in, out);
        while (number != -1) {
            //check for max, min
            if (number > max) { //Why use separate ifs, not
                max = number; //  if..else?
            }
            if (number < min) {
                min = number;
            }

            //accumulate sum, count
            sum = sum + number;
            count++;

            //Input data
            number = getInput(in, out);
        }

        printResults(out, sum, count, max, min);

        /*
         * Close input and output streams
         */
        in.close();
        out.close();
    }

}
//Output
//Enter a number, -1 to quit: 2
//Enter a number, -1 to quit: 8
//Enter a number, -1 to quit: 32
//Enter a number, -1 to quit: 4
//Enter a number, -1 to quit: 5
//Enter a number, -1 to quit: 17
//Enter a number, -1 to quit: -1
//
//There were 6 numbers.
//The maximum of the numbers was 32
//The minimum of the numbers was 2
//The sum of the numbers was 68
//The average of the numbers was 11.333333333333334
```

Single-entry, Single-exit

- It is important for our control structures to have a single entry and a single exit.
- For example,
  - instead of writing a loop with one return statement inside the loop and another after the loop, write the structure with **one** return statement only
  - this applies to methods as well; good design means single entry, single exit

For example, take the `calcAvg(sum, count)` method above. We could have said

```
    private static double calcAvg(int sum, int count) {

        if (count > 0) {
            return (double) sum / count;
        } else {
            return 0;
        }
    }
```

but instead of this, we used a local **`double`** variable, set its value in the method, and had only one `return` statement:

```
    private static double calcAvg(int sum, int count) {
        double avg = 0;

        if (count > 0) {
            avg = (double) sum / count;
        }

        return avg;
    }
```

- Notice the single return statement; this is a better design
- If we were to have a logical error, it would be much easier to trace this version than the previous one.
- I am requiring you to do it this way!! (You'll thank me later!)
- This leads to some basic rules for value-returning methods.

Rules for Value-Returning Methods

In a value-returning method:

- Declare a local variable of the same type as the method.
- Use this local variable for your calculations inside the method, ultimately assigning it a value.
- Return this local variable at the end of the method.
- Do not print inside these methods; rather, call them from a printing statement.

Calling Value-Returning Methods

Three ways to call value-returning methods:

1. On the right-hand side of an assignment statement:
   ```
   double average = calcAvg(sum, count);
   ```
2. In an output statement:
   ```
   out.println("The average is " + calcAvg(sum, count));
   ```
3. In a condition:
   ```
   if (isPrime(n))
   ```