Introduction to LOOPS in Java

Imagine you are asked to print the value of $2^1, 2^2, 2^3, 2^4$ ......2^{10}. One way could be taking a variable $x = 2$ and printing and multiplying it with 2 each time.

```
public class Main {
  public static void main(String[] args) {
    int x = 2;
    /* printing value and multiplying it by 2 every time */
    System.out.print(x + " ");
    x *= 2;
    System.out.print(x + " ");
    x *= 2;
    System.out.print(x + " ");
    x *= 2;
    System.out.print(x + " ");
    x *= 2;
    System.out.print(x + " ");
    x *= 2;
    System.out.print(x + " ");
    x *= 2;
    System.out.print(x + " ");
    x *= 2;
    System.out.print(x + " ");
    x *= 2;
    System.out.print(x + " ");
    x *= 2;
    System.out.print(x + " ");

  }
}
```

Output:

```
2 4 8 16 32 64 128 256 512 1024
```

Another way is using loops. Using Looping statements the above code of **10** lines can be reduced to **3** lines.

```
public class Main {
  public static void main(String[] args) {
    int x = 2;
    for (int i = 0; i < 10; i++) {
      System.out.print(x + " ");
      x *= 2;
    }
  }
}
```

Output:

```
2 4 8 16 32 64 128 256 512 1024
```

Explanation:

Looping Constructs in Java are statements that allow a set of instructions to be performed repeatedly as long as a specified condition remains true.

Java has three types of loops i.e. the for loop, the while loop, and the do-while loop. for and while loops are **entry-controlled** loops whereas do-while loop is an **exit-controlled** loop.

A very obvious example of loops can be daily routine of programmers i.e. Eat -> Sleep -> Code -> Repeat
Introduction to LOOPS

Types of loops

There are three types of Loops. They are explained below.

The for Loop

When we know the exact number of times the loop is going to run, we use for loop. It provides a concise way of writing initialization, test condition, and increment/decrement statements in one line. Thus, it is easy to debug and also has no risk of forgetting any part of the loop, since the condition is checked before.

For loop is an **entry-controlled** loop as we check the condition first and then evaluate the body of the loop.

Syntax:

```
for(Initialization ; Test Expression ; Updation){
    // Body of the Loop (Statement(s))
}
```

Example

```
public class Main {
  public static void main(String[] args) {
    int maxNum = 5;
    /* loop will run 5 times until test condition becomes false */
    for (int i = 1; i <= maxNum; i++) {
      System.out.println(i);
    }
  }
}
```

Output:

```
1
2
3
4
5
```

Explanation:

Here, initially i is 1 which is less than maxNum so it is printed. Then i is incremented and printed until it is greater than maxNum. **Explanation:**

Iteration	Variable	Condition i<=maxNum	Action and Update Expression
1st	i=1, maxNum=5	true	Print 1 and Increment i
2nd	i=2, maxNum=5	true	Print 2 and Increment i
3rd	i=3, maxNum=5	true	Print 3 and Increment i
4th	i=4, maxNum=5	true	Print 4 and Increment i
5th	i=5, maxNum=5	true	Print 5 and Increment i
6th	i=6, maxNum=5	false	Terminate Loop

For Loop Variations

Multiple Initializations and Update Expressions

The for loop can have multiple initialization expressions and/or update expressions. Imagine you have a 2D matrix and you want to fill only diagonal places with 1. You need two variables i and j to represent the row index and column index in the matrix. Using these two values, a cell can be uniquely identified in the matrix. In this example, you can initialize two variables, check two conditions and update two variables in a single loop.

Note: Here, for initialization and update expressions we used comma but for condition we used '&&' (and) operator. Other Logical operators can also be used.

```java
public class Main {

  public static void main(String[] args) {
    int totalRows = 5, totalCols = 5;
    // Declaring a 2D array
    int[][] matrix = new int[totalRows][totalCols];

    /* for loop to fill diagonal values */
    for (
      int row = 0, col = 0;
      row < totalRows && col < totalCols;
      row++, col++
    ) {
      matrix[row][col] = 1;
    }

    // Printing matrix
    for (int i = 0; i < totalRows; i++) {
      for (int j = 0; j < totalCols; j++) {
        System.out.print(matrix[i][j] + " ");
      }
      System.out.println();
    }
  }
}
```

Output

```
1 0 0 0 0
0 1 0 0 0
0 0 1 0 0
0 0 0 1 0
0 0 0 0 1
```

The while Loop

The while loop is used when the number of iterations is not known but the terminating condition is known. Loop is executed until the given condition evaluates to false. While loop is also an **entry-controlled** loop as the condition is checked before entering the loop. The test condition is checked first and then the control goes inside the loop.

Although for loop is easy to use and implement, there may be situations where the programmer is unaware of the number of iterations, it may depend on the user or the system. Thus, when the only iterating and/or terminating condition is known, while loop is to be used.

While loop is much like **repeating if statement**.

Syntax:

```
Initialization;
while(Test Expression){
    // Body of the Loop (Statement(s))
    Updation;
}
```

Example Let us look at an example where we find the factorial of a number. So, until i is not equal to 0, while loop will be executed and i will be multiplied to variable fact.

```
public class Main {

  public static void main(String args[]) {
    int factorial = 1, number = 5, tempNum = 0;
    tempNum = number; // Initialization;
    while (tempNum != 0) { // Test Expression
      factorial = factorial * tempNum;
      --tempNum; //Updation;
    }
    System.out.println("The factorial of " + number + " is: " + factorial);
  }
}
```

```
The factorial of 5 is: 120
```

The factorial of a number n is given as: $n = n * (n-1) * (n-2)..... * 1$. Here, until the number is not zero, the factorial variable is multiplied with the number and decremented.

Explaination

i	fact	Condition	Action
5	1	True	fact = 5, i = 4
4	5	True	fact = 20, i = 3
3	20	True	fact = 60, i = 2
2	60	True	fact = 120, i = 1
1	120	True	fact = 120, i = 0
0	120	False	Loop Terminated

Working of a while loop

Firstly, test expression is checked. If it evaluates to true, the statement is executed and again condition is checked. If it evaluates to false the loop is terminated.

While Loop Variations

Empty while Loop

The empty while loop doesn't contain any statement in its body. **It behaves as an infinite loop if the initial condition is satisfied.**

```
public class Main {

  public static void main(String args[]) {
    int x = 1;
    while (x < 10) {}
  }
}
```

Explanation:

The loop runs infinitely as the initial condition is satisfied and there is no update operation.

Infinite while Loop

An infinite while loop is a very common error made by the programmers where often they forget update statements and the loop runs infinite times.

```
public class Main {

  public static void main(String[] args) {
    int j = 10;
    while (j >= 0) {
      System.out.println(j);
    }
  }
}
```

In the above example, the value of j is not decremented and thus the loop is executed infinite times.

Also sometimes while loop is made to run infinite times as per the requirement of the program by giving the condition to be true as shown in the example.

```
public class Main {

  public static void main(String[] args) {
    while (true) {
      System.out.println("Hello World");
    }
  }
}
```

The do-while Loop

The do-while loop is like the while loop except that the condition is checked after evaluation of the body of the loop. Thus, the do-while loop is an example of an **exit-controlled loop**.

The loop is executed once, and then the condition is checked for further iterations. This loop runs at least once irrespective of the test condition, and at most as many times the test condition evaluates to true. It is the only loop that has a **semicolon(;)**.

Need for do-While Loop: The instructions inside for and while loops are not evaluated if the condition is false. In some cases, it is wanted that the loop-body executes at least once irrespective of the initial state of test-expression. This is where the do-while loop is used.

Syntax:

```
Initialization;
do {
  // Body of the loop (Statement(s))
  // Updation;
}
while(Test Expression);
```

```
public class Main {

  public static void main(String args[]) {
    Scanner sc = new Scanner(System.in);
    do {
      // Take numbers as input
      System.out.print("Enter First Number: ");
      int first = sc.nextInt();
      System.out.print("Enter Second Number: ");
      int second = sc.nextInt();

      // Print the choice list
      System.out.println(
        "Enter your choice \n1 Add\n2 Subtract\n3 Multiply\n4 Divide"
      );

      // Perform the required operation
      switch (sc.nextInt()) {
        case 1:
          first += second;
          break;
        case 2:
          first -= second;
          break;
        case 3:
          first *= second;
          break;
        case 4:
          first /= second;
          break;
      }

      // Print the output
      System.out.println("Result is " + first + ".");
      System.out.println("To continue enter 1, to exit enter 0");
    } while (1 == sc.nextInt()); // test Expression
  }
}
```

Output:

```
Enter First Number: 10
Enter Second Number: 5
Enter your choice
1 Add
2 Subtract
3 Multiply
4 Divide
4
Result is 2.
To continue enter 1, to exit enter 0
1
Enter First Number: 4
Enter Second Number: 5
Enter your choice
1 Add
2 Subtract
3 Multiply
4 Divide
1
Result is 9.
To continue enter 1, to exit enter 0
0
```

Explanation:

The above program takes two numbers from the user and performs operations on it. The program runs at least once and at max as many times, the user wants to continue.

It first takes a number initially. It then asks for another number and operation to be performed with them. It prints the result and asks the user if they want to continue.

If they enter 1, the body of the do-while loop executes again.

Nested Loop in Java

In this tutorial, we will learn about nested loops in Java with the help of examples.

If a loop exists inside the body of another loop, it's called a nested loop. Here's an example of the nested `for` loop.

```
// outer loop
for (int i = 1; i <= 5; ++i) {
  // codes

  // inner loop
  for(int j = 1; j <=2; ++j) {
    // codes
  }
..
}
```

Here, we are using a `for` loop inside another `for` loop.

We can use the nested loop to iterate through each day of a week for 3 weeks.

In this case, we can create a loop to iterate three times (3 weeks). And, inside the loop, we can create another loop to iterate 7 times (7 days).

Example 1: Java Nested for Loop

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

    int weeks = 3;
    int days = 7;

    // outer loop prints weeks
    for (int i = 1; i <= weeks; ++i) {
      System.out.println("Week: " + i);

      // inner loop prints days
      for (int j = 1; j <= days; ++j) {
        System.out.println("  Day: " + j);
      }
    }
  }
}
```

Output

```
Week: 1
  Day: 1
  Day: 2
  Day: 3
    .....  ..  ....
Week: 2
  Day: 1
  Day: 2
  Day: 3
  ....  ..  ....
....  .. ....
```

In the above example, the outer loop iterates 3 times and prints 3 weeks. And, the inner loop iterates 7 times and prints the 7 days.

We can also create nested loops with while and do...while in a similar way.

> **Note:** It is possible to use one type of loop inside the body of another loop. For example, we can put a `for` loop inside the `while` loop.

Example 2: Java for loop inside the while loop

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

    int weeks = 3;
    int days = 7;
    int i = 1;

    // outer loop
    while (i <= weeks) {
      System.out.println("Week: " + i);

      // inner loop
      for (int j = 1; j <= days; ++j) {
        System.out.println("  Days: " + j);
      }
      ++i;
    }
  }
}
```

Output:

```
Week: 1
  Day: 1
  Day: 2
  Day: 3
  .... .. ....
Week: 2
  Day: 1
  Day: 2
  Day: 3
  .... .. ....
.... .. ....
```

Here you can see that the output of both **Example 1** and **Example 2** is the same.

Example 3: Java nested loops to create a pattern

We can use the nested loop in Java to create patterns like full pyramid, half pyramid, inverted pyramid, and so on.

Here is a program to create a half pyramid pattern using nested loops.

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

    int rows = 5;

    // outer loop
    for (int i = 1; i <= rows; ++i) {

      // inner loop to print the numbers
      for (int j = 1; j <= i; ++j) {
        System.out.print(j + " ");
      }
      System.out.println("");
    }
  }
}
```

Output

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

break and continue Inside Nested Loops

When we use a `break` statement inside the inner loop, it terminates the inner loop but not the outer loop. For example,

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

    int weeks = 3;
    int days = 7;

    // outer loop
    for(int i = 1; i <= weeks; ++i) {
      System.out.println("Week: " + i);

      // inner loop
      for(int j = 1; j <= days; ++j) {

        // break inside the inner loop
        if(i == 2) {
          break;
        }
        System.out.println("  Days: " + j);
      }
    }
  }
}
```

Output

```
Week: 1
  Day: 1
  Day: 2
  .... .. ....
Week: 2
Week: 3
  Day: 1
  Day: 2
  .... .. ....
.... .. ....
```

In the above example, we have used the break statement inside the inner `for` loop. Here, the program skips the loop when `i` is **2**.

Hence, days for **week 2** are not printed. However, the outer loop that prints week is unaffected.

Similarly, when we use a `continue` statement inside the inner loop, it skips the current iteration of the inner loop only. The outer loop is unaffected. For example,

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

    int weeks = 3;
    int days = 7;

    // outer loop
    for(int i = 1; i <= weeks; ++i) {
      System.out.println("Week: " + i);

      // inner loop
      for(int j = 1; j <= days; ++j) {

        // continue inside the inner loop
        if(j % 2 != 0) {
          continue;
        }
        System.out.println("  Days: " + j);
      }
    }
  }
}
```

Output

```
Week: 1
  Days: 2
  Days: 4
  Days: 6
Week: 2
  Days: 2
  Days: 4
  Days: 6
Week: 3
  Days: 2
  Days: 4
  Days: 6
```

In the above example, we have used the continue statement inside the inner for loop. Notice the code,

```java
if(j % 2 != 0) {
  continue;
}
```

Here, the `continue` statement is executed when the value of `j` is odd. Hence, the program only prints those days that are even.

We can see the `continue` statement has affected only the inner loop. The outer loop is working without any problem.