Lecture Notes

Loops

Welcome to the session on Loops.

In the previous session, you learnt how to implement conditional statements. In most of the cases you saw in the previous session, we took an input and based on the input we checked the conditions, and the program ended. But, what if you need to run the same logic for multiple inputs?

Let's say you had to print 'Hello World' 16 times. To do this, a brute force program would look like this:

```
public class HelloWorld {

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

If you were required to print 'Hello World' 100 times using the same method, the code would be long and exhaustive.

For such repetitive tasks, you learnt to implement loops. A loop is a piece of code that repeats itself until a certain condition is met. You learnt how to print 'Hello World' multiple times using a while loop.

So, if you had to write 'Hello World' 100 times, a simple program using a while loop would look like —

```
public class LearningLoops {
    public static void main(String[] args) {
        int counter = 0;
        while (counter < 100) {
            System.out.println("Hello World");
            counter = counter + 1;
        }
    }
}
```

There are some important elements in defining a loop:

1. Starting and ending point
2. Loop continuation condition
3. Increment-decrement condition, which ensures that a loop runs through iterations

In the above code, you did the following:

1. Initialized the counter, set it equal to 0, this was the starting point for the loop.
2. Set the loop-continuation condition so that the loop doesn't run more than 100 times. The 100th iteration was the stop point for the loop.
3. Ensured that during each iteration of the loop, statements were executed only once.
4. At the end of each iteration, you increased the counter by 1, to keep a track of how many times the loop ran

You applied the same loop logic to print 100 numbers from 1 to 100

```
public class PrintHundredNumbers {
    public static void main(String args[]) {
        int number = 1;
        while (number <= 100) {
            System.out.println(number);
            number++;
        }
    }
}
```

The loop structure remained the same:

1. The loop started with number = 1 , ended with number = 100
2. In each iteration of the loop, you incremented the value of the number variable by 1.
3. The loop continued iterations until 100 iterations were complete, i.e. the value of number reached 100

You extended the logic of loops with strings, where you printed out the characters of a string. The characters of a string are stored as indexes, from 0 to the length of the string minus one.

So, the steps you did were

1. Identified the length of an input string
2. Then started the loop from the string index 0, till the string index reached length - 1
3. You print out the character in the String at the index
4. In each iteration, you incremented the index by 1

So, your pseudocode looked like —

```
Read inputString
Set position = 0
WHILE position < inputString.length()
 Display - inputString.charAt(position)
 position = position + 1
END WHILE
```

And your code was –

```
import java.util.Scanner;

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

        Scanner scan = new Scanner(System.in);
        System.out.println("Enter the string  ");
        String inputString = scan.next();
        int position = 0;

        while (position < inputString.length()) {
            System.out.println(inputString.charAt(position));
            position = position + 1;
        }
    }
}
```

Then, you looked at some examples where the ending/stopping point was not clearly defined. You wanted to run a loop until some condition was met, but it was not clear how many iterations it would take to meet the condition. In that case, you used a loop control variable, which determined whether a loop should run or stop.

So, for this purpose, you made a Boolean variable, which started with a true value, and this served as the loop continuation condition. As long as this variable held true, the loop should run.

The code for this would have looked like –

```
boolean loopContinue = true;
while(loopContinue == true){
    if(condition is met){
        loopContinue = false;
    } else {
        Do nothing
    }
}
```

So, here the loop control was specified by the **loopContinue** variable. Until the value of the loop-continue variable was true, the loop kept running iterations. As soon as the loopContinue variable's value was **false**, the loop ended.

So, you made a password checker and you wanted the user to be able to try as many times as he/she wanted. We used the above logic to create a pseudocode:

```
Set password
Read input
Set Boolean loopContinue = true
WHILE loopContinue == true
 IF input == password THEN
   Print - Correct Password
   loopContinue = false
 ELSE
   Print - Try again
   loopContinue = Read (true/false)
 END IF
END WHILE
```

So, the code for your password checker program looked like —

```
import java.util.Scanner;

public class Password {

    public static void main(String args[]) {
        String password = "UpGrad";
        Scanner scan = new Scanner(System.in);

        Boolean loopcontinue = true;  // Loop Control
        while (loopcontinue == true) {
            System.out.println("Enter the password ");
            String input = scan.next();
            if (input.equals(password)) {
                System.out.println("Correct password");
                loopcontinue = false;
            } else {
                System.out.println("Password is incorrect.");
            }
            System.out.println("Do you wish to try again, true or false ");
            loopcontinue = scan.nextBoolean();
        }
    }
}
```

Do While Loops

Then you learnt another kind of loop called the **do-while** loop, which looks quite similar to the while loop. The do-while loop is written as —

```
counter = 0;
 DO{
  //code to be executed
  counter++;
} WHILE(loop continuation condition);
```

So, the pseudocode for the do-while loop to print 100 numbers looks like –

```
Set number = 1;
DO
Print - number
number++
WHILE number <= 100
END DO WHILE
```

The code looked like –

```
public class DoWhile {
    public static void main(String args[]) {
        int number = 1;
        do {
            System.out.println(number);
            number = number++;
        } while (number <= 100);
    }
}
```

You also learned the major difference between a while and a do-while loop. A do-while loop checks for the loop continuation condition after an iteration is done. Thus, a do while loop ends up completing one iteration even if it does not satisfy the loop continuation condition in the first place. However, this is not possible in a simple while loop, where the loop continuation is checked before the iterations begin.

```
public class WhileLoop {

   public static void main(String args[]) {
       int number = 101;
       while (number <= 100) {
           number = number*2;
           number++;
       }
       System.out.println(number);
   }
}
```

Output: 101

```
public class Dowhile {
   public static void main(String args[]) {
       int number = 101;
       do {
           Number = number*2;
           number++;
       } while (number <= 100);
       System.out.println(number);
   }
}
```

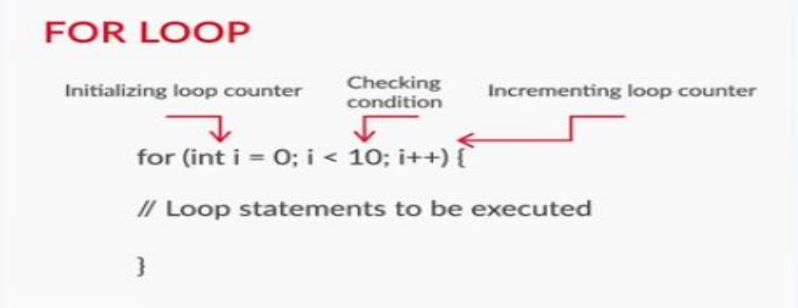
Output: 203

FOR LOOPS

Then you looked at the For loop that had the same elements of a while loop:

- A counter
- A loop-continuation statement
- An increment statement

A simple for loop looks like this:



Here, a counter is initialised, a loop-continuation statement is set, and a counter increment statement is specified. The loop should continue until the checking statement is TRUE — which would be 10 times in this case; and the counter, also called a loop control variable, is incremented by 1 after each iteration.

Here is the comparison between the structure of a for loop and a while loop.

```
public class ForLoop {
    public static void main(String args[]) {
        for(int number = 1; number <= 100; number++){
            System.out.println(number);
        }
    }
}
```

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

        int number = 1;
        while(number <= 100) {
            System.out.println(number);
            number++;
        }
    }
}
```

You learnt that both the code have counter initialisers, loop continuation conditions, and increment operators.

While these elements are spread out in a while loop, they're in a single place in the for loop.

The difference between these two loops is that for loops require you to know how many loop iterations you need in advance. But with while loops, you don't need to know the number of iterations ahead of time. Therefore, for loops should be used when you know how many times a loop should run. While loops offer more flexibility as you saw in the password checker program, where you could run the loop any number of times.

Then you looked at a program to calculate the factorial of a number using a for loop. A factorial of a number 'n' is the product of all the positive integers less than or equal to n. Thus, a factorial of 5 would be 5*4*3*2*1.

The pseudocode for the program looks like —

```
Read input
Set factorial = 1
FOR i = 1 to i = input
 factorial = factorial * i
 i++
END FOR
Print - factorial
```

Then, you wrote the code for this program:

```
import java.util.Scanner;

public class FactorialModified {
    public static void main(String arg[]) {
        int factorial = 1;
        System.out.println("Enter the number  ");
        Scanner scan = new Scanner(System.in);
        int number = scan.nextInt();

        if (number > 0) {
            for (int i = 1; i <= number; i++) {
                factorial = factorial * i;
                System.out.println(factorial);
            }
        } else if (number == 0) {
            factorial = 1;
            System.out.println(factorial);
        } else {
            System.out.println("There is no factorial for a negative number");
        }

    }
}
```

You also saw that if you entered the loop conditions incorrectly, the loop may go into infinite iterations. It is therefore, desired that you define the loop conditions correctly.

ARRAYS

You then looked at how to initialise and fill elements in an array. An array index starts from 0 and ends at the array length minus one.

Here is the pseudocode for your quick recap —

```
Read arraySize
Initialise array[arraySize]
FOR index = 0 to arraysize-1
  array[index] = Read value
index++
END FOR
```

After this, you saw how to find the element with the maximum value in an array. You need to look at the first element and compare it with the second. Then, you would compare the largest of the two with the third, and then, the larger of these three with the fourth, and so on, until all the elements of the array are checked. Then logically, you came up with a variable called max, which is able take the value of the larger of the two numbers compared. In this way, you could directly compare the max variable's value with the next element in the array. If max variable's value is larger than the next number, that's okay. But if it's smaller than the new number, max takes the value of the new number. Thus, max remains the largest of the numbers considered.

So, the pseudocode for this looks like —

```
Initialise array
Set max = array[0];
FOR index = 0 to arraylength-1
  IF max < array[index] THEN
    max = array[index]
  END IF
  index++
END FOR
Print - max
```

Then, you wrote the code based on this logic –

```
import java.util.Scanner;

public class ArrayMax {
    public static void main(String[] args) {
        int num, max;
        Scanner scan = new Scanner(System.in);
        System.out.print("Enter number of elements in the array:");
        num = scan.nextInt();
        int array[] = new int[num];
        System.out.println("Enter elements of array: ");
        for (int index = 0; index < num; index++) {
            array[index] = scan.nextInt();
```

```
        }
        max = array[0];
        for (int index = 0; index < num; index++) {
            if (max < array[index]) {
                max = array[index];
            }
        }
        System.out.println("Maximum value:" + max);
    }
}
```

Break and Continue Statements

After the loops, you learnt about the break and continue statements.

While a break statement is used to break out of a loop and continue with the rest of the code, a continue statement is used to skip the statements in an iteration.

To understand this, consider this code:

```
public class BreakNContinue {
    public static void main(String args[]) {
        for( int 2 = 0; i< 10; i++) {
            if (i==2) {
                continue;
            } else if (i==4) {
                break;
            } else {
                System.out.print(i + " ");
            }
        }
    }
}
```

The output for the code above is 1 and 3.

When the increment variable i was 1, the loop printed 1. Then, when i was 2, the loop encountered a continue statement and skipped the rest of the statements in that iteration. Again, when ***i*** was 3, it printed 3. Then, when I was 4, the code encountered a break statement, and thus, broke out of the loop at that very instant.

You framed the logic for finding a prime number using the break statement.
Given a number, you continued checking if the number was divisible by any number from 2 to the square root of the number. If in any iterations it was divisible, then it was not a prime number, otherwise it was.

The pseudocode for this program was —

```
Read inputnumber
Set primeCheck = true

FOR i = 2 to Math.sqrt(inputnumber)
  IF number%i == 0 THEN
    Display - Input Number not prime
    primeCheck = false
  break
  END IF
  i++
END FOR

IF primeCheck == true THEN
  Print - Input number is Prime
END IF
```

So, when an input number was collected, your code started a loop starting from 2 up to the square root of the number. In each iteration of the loop, it was checked if the input number was divisible by any of the numbers. If it was, then you set the value of the primeCheck variable to **false** and break out of the loop, as the number was clearly not a prime number. If at the end of the loop, the value of the primeCheck was still **true** , it meant that the number was a prime number.

Then, you wrote the code for this program:

```
import java.util.Scanner;
public class PrimeNumber {
    public static void main(String[] args) {
        int num;
        System.out.println("Enter the number you wish to check");
        System.out.println();
        Scanner scan = new Scanner(System.in);
        num = scan.nextInt();
        boolean primeCheck = true;
        for (int i = 2; i < num; i++) {
            if (num % i == 0) {
                System.out.println("Not a prime");
                primeCheck = false;
                break;
            }
        }
        if (primeCheck == true) {
            System.out.println(num + " is a prime number.");
        }
    }
}
```

Nested Loops

In the previous segment, you looked into simple while and for loops. But often, you would have to use more complex loops. You may even be required to use loops within a loop. These are called **nested loops**.

So, you saw the code for finding whether a given number was a prime number or not. But, what if you wanted to check and print all prime numbers from 1 to 1000. Would you apply the same logic again and again for the 1000 numbers?

The better solution, you saw, was to run a loop, from 2 to 1000 and check in each iteration if the number was prime. So, you had an outer loop to iterate through numbers 1 to 1000 and an inner loop where for each number you would check if the number is prime.

You wrote the pseudocode for this program:

```
FOR num = 2 to 1000
  Set primeCheck = true

  FOR i = 2 to sqrt(num)
    IF num%i == 0 THEN
      Display - Input Number not prime
      primeCheck = false
      break
    END IF
    i++
  END FOR

  IF primeCheck == true THEN
    PRINT – num
  END IF

  num++
END FOR
```

The code for this program was —

```
import java.util.Scanner;

public class PrintPrimeNumbers {
    public static void main(String[] args) {
        for (int num = 2; num <= 1000; num++) {
            boolean primeCheck = true;
            for (int i = 2; i <= Math.sqrt(num); i++) {
                if (num % i == 0) {
                    primeCheck = false;
                    break;
                }
```

```
            }
            if (primeCheck == true) {
                System.out.println(num);
            }
        }
    }
}
```

2-D Arrays

Then you learnt about 2D arrays. Similar to a 1D array, a 2D array has indices; but here, the index of an element is the combination of its row number and column number. So, an element in a 2D array can be accessed by array[i][j], where i is the row number, and j is the column number.

After that, you wrote a simple program to initialize a 2D array using a nested loop. You learnt that you can fill a 2D matrix either row-wise, where you fill all the indices of the first row and then you move on to the next row. Or, you can fill the 2D matrix column-wise, where you first fill all the elements in the first column and then move on to the next column, and so on.

So, the pseudocode for taking a user input and filling the 2D matrix was —

```
Read rowsize, columnsize
Initialise array[rowsize][columnsize]

FOR rowindex = 0 to rowsize-1
   FOR colindex = 0 to colsize-1
    array[rowindex][colindex] = Read value
    colindex++
   END FOR
   rowindex++
END FOR
```

In the outer for loop, you started the iteration of rows starting from row 1 till the number of all rows minus one. In each iteration of the outer loop, you filled all the elements in the columns of that row. Now, using the same logic, you can print the elements of a 2D array, but first printing we will print all the values in row 1, row 2, and so on.

The code for this program was —

```
import java.util.Scanner;

public class TwoDArray {
    public static void main(String[] args) {
        int rowsize, columnsize;
        Scanner scan = new Scanner(System.in);
        System.out.print("Enter number of rows in the array: ");
        rowsize = scan.nextInt();
        System.out.print("Enter number of columns in the array: ");
        columnsize = scan.nextInt();
```

```
        int array[][] = new int[rowsize][columnsize];
        System.out.println("Enter elements of array: ");
        for (int rowindex = 0; rowindex < rowsize; rowindex++) {
            for (int colindex = 0; colindex < columnsize; colindex++) {
                System.out.println("Enter value ");
                array[rowindex][colindex] = scan.nextInt();
            }
        }

        System.out.println("The  elements of array are: " + "\n");

        for (int rowindex = 0; rowindex < rowsize; rowindex++) {
            for (int colindex = 0; colindex < columnsize; colindex++) {
                System.out.print(array[rowindex][colindex] + " ");
            }
            System.out.println();
        }
    }
}
```