

Session 1- Facilitation Guide
Flow Control in Java
INDEX

- I. Recap from the previous session**
- II. Flow Control**
 - **if and switch Statements**
 - **if-else Branching**
 - **switch Statements**
 - **Loops and Iterators**
 - **Using while Loops**
 - **Using do Loops**
 - **Using for Loops**
- III. User Input**
 - A. Using Buffered Reader Class**
 - B. Using Scanner Class**
 - C. Using Console Class**

For (2 hrs) ILT

I. Recap from the last session

Operators: We have discussed with examples on various types of operators: arithmetic, assignment, relational, logical, and ternary (conditional) operators.

II. Flow Control

Now let us explore another important topic - flow control, which controls the flow of execution within a program. We use following control statements in Java:

- **if-else Branching**
- **switch Statements**
- **Loops**

We will show how these statements are used to control the flow of the program and

make decisions based on conditions.

Flow control statements in Java are used to control the flow of execution within a program. They allow you to make decisions, repeat actions, and control the order in which statements are executed. Flow control statements are essential for creating programs with logic and enabling different paths of execution based on certain conditions. There are three main types of flow control statements in Java:

Conditional Statements: Conditional statements in Java are control structures that allow you to make decisions based on certain conditions. They enable you to execute specific blocks of code depending on whether a given condition is true or false. Java provides two main types of conditional statements: if statement and switch statement.

Loop Statements: Loop statements in Java are control structures that allow us to repeat a block of code multiple times based on a given condition. They enable us to iterate over collections, perform repetitive tasks, and process data in a controlled and organized manner. Java provides three main types of loop statements: for loop, while loop, and do-while loop.

Branching Statements: Branching statements in Java are control flow statements that allow us to alter the normal flow of execution in your program. They are useful to make decisions and control the order in which statements are executed. Java provides three main types of branching statements: break, continue, and return.

A. if-else Branching

The if statement is the most basic form of a conditional statement in Java. It evaluates a boolean expression and executes a block of code if the expression evaluates to true.

Syntax:

```
if (boolean_expression) {  
    // Code to be executed if the boolean_expression is true  
}
```

Example:

Java

```
...  
int age = 20;  
if (age >= 18) {
```

```
        System.out.println("You are an adult.");
    }
    ...

```

The "if" statement executes a specific part of the code if the test expression evaluates to true. But if the test expression returns false, the if statement does nothing. In this case, we can use an optional "else" block. In Java, statements inside the body of the else block are executed when a test expression evaluates to false. These statements are called if-...else statements. This is the syntax of the if...else statement.

```
Java
if (condition) {
    // codes in if block
}
else {
    // codes in else block
}

```

Following examples will make the if...else syntax clear:

```
Java
public class IfElseExample {
    public static void main(String[] args) {
        int age = 10;
        // Checks if age is greater than or equal to 18
        // and execute the following block.
        if (age >= 18) {
            System.out.println("You are an adult."); //
Output from if
        }
        // Execute this block
        // if number is less than 18
    }
}

```

```

        else {
            System.out.println("You are a minor."); // Output
from else
        }

// This statement will be executed irrespective of the outcome
// of the if...else statement unless there is an Exception //
occurred. (Exception will be explained later)
        System.out.println("Statement outside if...else
block");
    }
}

```

In Java we can combine multiple if..else...if statements and it is commonly called if-else-if ladder. It is used to decide among multiple options. The if statements are executed from the top down. As soon as one of the conditions controlling the if statement is true, the statement associated with that “if” will be executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed. Here is an example of if-else-if ladder statements:

Java

```

public class IfElseIfLadder {

    public static void main(String[] args) {

        int age =18;

        //Checks if the age is below 5;
        if (age<5) {
            System.out.println("You are eligible for free
ride.");
        }
        // Checks if age is between 5 and 12
        else if (age<12) {
            System.out.println("You have to pay half the
fair.");
        }
    }
}

```

```

        // Checks any other age (above or equal to 12)
    else {
        System.out.println("Please pay the full fair.");
    }
}
}

```

Output:

Please pay the full fair.

The “if...else” statement can be nested that means one if...else block can be inside another. Here is an example to explain the nested if...else block:

Java

```

public class VotingEligibility {
    public static void main(String[] args) {
        //age
        int age = 25;
        //Is Indian citizen?
        boolean isIndianCitizen = true;
        // Check voting eligibility
        if (isIndianCitizen) {
            if (age>=18) {
                System.out.println("You are eligible to
vote.");
            }
            else {
                System.out.println("You must be 18 years
old or above to vote.");
            }
        }
        else {
            System.out.println("Only citizens are eligible to

```

```
        vote.");  
    }  
}  
}
```

Output:

You are eligible to vote.

B. switch Statements

The switch statement is used to select one of many code blocks to be executed based on the value of an expression. It is typically used when there are multiple cases to be evaluated against the same expression.

Syntax:

```
Java  
..  
switch (expression) {  
    case value1:  
        // Code to be executed if the expression matches value1  
        break;  
    case value2:  
        // Code to be executed if the expression matches value2  
        break;  
    // More cases can be added  
    default:  
        // Code to be executed if none of the cases match the  
        expression  
        break;  
}  
...
```

Example:

Java

```
public class SwitchCaseExample {  
    public static void main(String[] args) {  
  
        int dayOfWeek = 3;  
        switch (dayOfWeek) {  
            case 1:  
                System.out.println("Monday");  
                break;  
            case 2:  
                System.out.println("Tuesday");  
                break;  
            case 3:  
                System.out.println("Wednesday");  
                break;  
            case 4:  
                System.out.println("Thursday");  
                break;  
            case 5:  
                System.out.println("Friday");  
                break;  
            case 6:  
                System.out.println("Saturday");  
                break;  
            case 7:  
                System.out.println("Sunday");  
                break;  
            default:  
                System.out.println("Invalid day");  
                break;  
        }  
    }  
}
```

Output:

```
C:\sourcecodes>javac SwitchCaseExample.java

C:\sourcecodes>java SwitchCaseExample
Wednesday

C:\sourcecodes>
```

Note that each case block should end with a break statement to exit the switch block. If a matching case is found, the code inside that case is executed, and without a break, the flow will continue to the next case, even if it doesn't match.

From Java 14, we can use a new form of switch label that allows multiple constants per case as shown below:

First let us look at the following switch case block.

Java

```
...
public void daysOfMonth(int month) {
    switch (month) {
        case 1:
        case 3:
        case 5:
        case 7:
        case 8:
        case 10:
        case 12:
            System.out.println("this month has 31 days");
            break;
        case 4:
        case 6:
        case 9:
            System.out.println("this month has 30 days");
            break;
        case 2:
            System.out.println("February can have 28 or 29
```



```

    days");
        break;
    default:
        System.out.println("invalid month");
    }
}
...

```

You can easily understand that this method prints the number of days in a given month. Some months have 31 days, some others have 30 days and February can have 28 or 29 days. The code works fine. But the code looks cumbersome and we have to use deliberate fall-through cases which makes it unnecessarily verbose.

But starting from Java 14 , we can use multiple cases at a time in a switch-case block. So above code can be simplified to:

Java

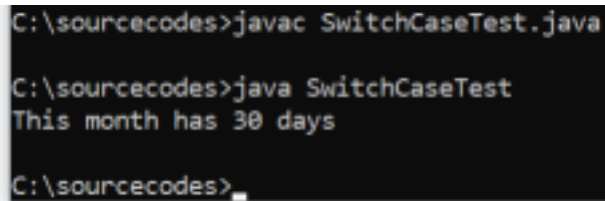
```

public class SwitchCaseTest {
    public static void main(String[] args) {
        int month = 6;
        switch (month) {
            case 1, 3, 5, 7, 8, 10, 12 : System.out.println("this
month has 31 days"); break;
            case 4, 6, 9 : System.out.println("This month has 30
days");break;
            case 2 : System.out.println("February can have 28 or
29 days");break;
            default : System.out.println("Invalid month");
        }
    }
}

```

You can easily find that code has become clearer and less verbose by combining multiple cases.

Output:



```
C:\sourcecodes>javac SwitchCaseTest.java  
  
C:\sourcecodes>java SwitchCaseTest  
This month has 30 days  
  
C:\sourcecodes>
```

From Java 14, a new form of the switch label "case L ->" has been introduced that allows you to use multiple constants for each case, reducing the verbosity of the code. The switch block code now looks cleaner, shorter, and easier to understand. Also, you don't have to explicitly call break statements to end the case, resulting in less error prone code (no break and no fall through).

Here is the example:

Java

```
public class SwitchCaseWithMultipleConstants {  
    public static void main(String[] args) {  
        int month =14;  
        switch (month) {  
            case 1, 3, 5, 7, 8, 10, 12 -> System.out.println("this month  
has 31 days");  
            case 4, 6, 9 -> System.out.println("this month has 30  
days");  
            case 2 -> System.out.println("February can have 28 or 29  
days");  
            default -> System.out.println("invalid month");  
        }  
    }  
}
```

Output:

invalid month

Just carefully review the above code and you will notice that there are no break statements and we have used arrow (->) instead of colon(:).

From Java 14 onwards, switch case blocks can be used as an expression, which means the switch block can return a value. Following example will make the concept clear:

Java

```
public class SwitchCaseWithReturn {
    static int getDays (int month) {
        return switch (month) {
            case 1, 3, 5, 7, 8, 10, 12 -> 31;
            case 4, 6, 9 -> 30;
            case 2 -> 28;
            default -> 0;
        };
    }

    public static void main(String[] args) {

        int month =3; //March
        System.out.println("Number of days in March=" +
getDays(month));
    }
}
```

Output:

Number of days in March=31

C. Loops

A loop is a fundamental control structure in programming that is executed over and over again as long as the given condition is true. Loops are blocks of code that help you

automate repetitive tasks and iterate through collections of data. The most commonly used loops in Java are: “while loop”, “do-while loop” and “for loop”. Java also provides a useful mechanism called iterator for iterating over collections of data.

(Iterator will be discussed later along with Java collection frameworks. The iterator provides a way to iterate over elements in a collection such as ArrayList, LinkedList, etc.)

Need of Loops in Program:

Imagine a program which is required to calculate the tax of 5000 persons. We only got PAN numbers of those persons and other details will be retrieved from the server based on their PAN numbers. After that tax will be calculated according to some rules using the details available. If we do not use a loop then we have to perform the same operations 5000 times. Suppose we all know the code for calculating tasks for individuals, now we need to write the same lines 5000 times in the code. That would take up a lot of effort which is particularly just copy pasting the same sentences many times. That’s where loops come into play. Loops make it very easy to group all the code that’s needed to be repetitively processed.

Using while Loop

The while loop executes a block of code repeatedly as long as a given boolean condition is true. A while loop iterates through a set of statements till its boolean condition evaluates to false. As long as the condition given evaluates to true, the loop iterates.

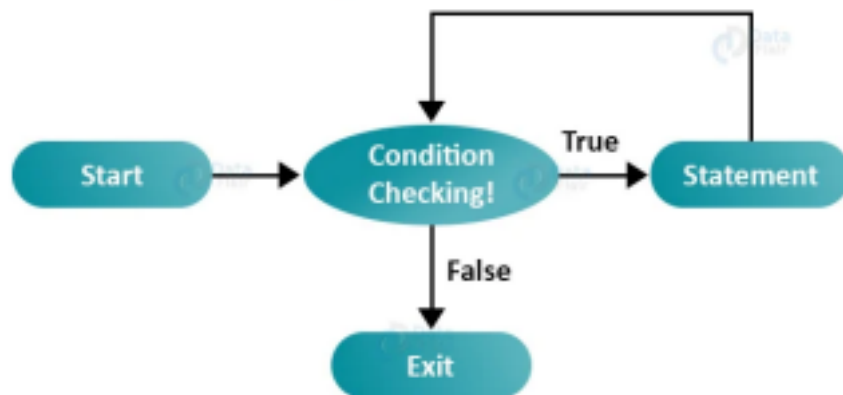
The basic syntax of Java while loop is:

```
Java
...
while(boolean condition)
{
    //statements;
}
```

One of the important properties of a while loop is that you cannot know the length of the loop everytime you define it. For example, if you want to iterate over all the digits of a number you don't know how big the number is. In this case, you can use a while loop, which iterates through the set of statements until the boolean condition returns False. If

the boolean condition evaluates to true, the loop will continue to iterate. In a while loop, the condition of the loop is checked first, then the control goes into the structure only if it evaluates to True. This type of loop is called **entry-controlled**. Generally, the body of a while loop contains a variable that controls or modifies the boolean condition mentioned.

Flow Diagram for Whileloop



Following example will explain the while loop:

```
Java
/**
 *
 * A simple program to explain Do loop in Java
 *
 */
public class DoLoopExample {

    public static void main(String[] args) {

        int count = 0;
        while (count < 5) {
            System.out.println("Count is: " + count);
            count++;
        }
        System.out.println("The value of count after the loop is " +
count);
    }
}
```

Output:

```

C:\sourcecodes>javac WhileLoopExample.java

C:\sourcecodes>java WhileLoopExample
Count is: 0
Count is: 1
Count is: 2
Count is: 3
Count is: 4
The value of count after the loop is 5

C:\sourcecodes>

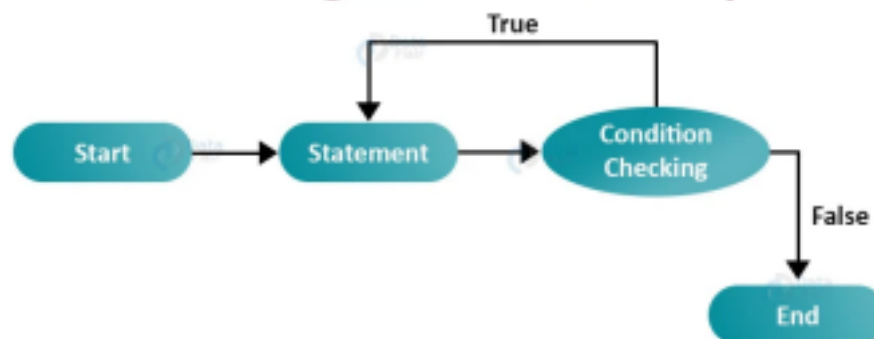
```

You can see that the value of the variable “counter” has become 5, which is the reason for terminating the loop.

Using do-while loop

The do-while loop is similar to the while loop but ensures that the block of code is executed at least once, even if the condition is false. Java do-while loop first executes the statement and then checks the condition. Otherwise, it is similar to “while” loop. The only difference is that if the condition at the beginning of the loop is false, the statements in the loop will still be executed, whereas in the while loop, the statements will not be executed. This loop is **exit-controlled** because it checks the condition only after the statements inside the loop have been executed.

Flow Diagram for Dowhileloop



Here is an typical example of do while loops:

Java

```

/*
 * Example to explain Do while loops in Java

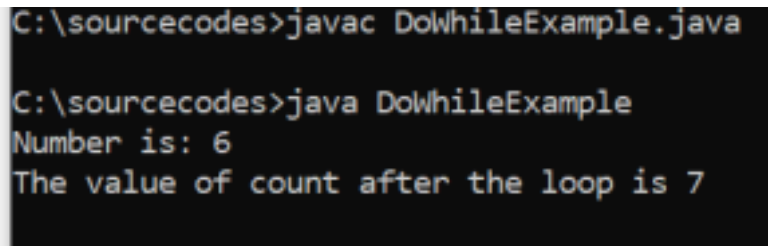
```

```

*/
public class DoWhileExample {
    public static void main(String[] args) {
        int num = 6;
        do {
            System.out.println("Number is: " + num);
            num++;
        }
        while (num <5); // Condition is false in the beginning.
        System.out.println("The value of count after the loop
is " + num);
    }
}

```

Output:



```

C:\sourcecodes>javac DoWhileExample.java

C:\sourcecodes>java DoWhileExample
Number is: 6
The value of count after the loop is 7

```

Please notice that the loop has been executed once even though the condition in the loop was false from the beginning.

Food for thought:

Where do you think the while and do-while loop can be used in a real-world scenario?

Using for loop

In Java, the for loop is a control flow statement that allows you to iterate over a range of values or elements in an array or collection. It is widely used when you know the number of iterations in advance or when you need to repeat a block of code a specific number of times.

Java for loop consists of 3 parts which define the loop itself. These are the initialization statement, a testing condition, an increment or decrement part for incrementing/decrementing the control variable.

The syntax of the for loop is as follows:

```

...
for (initialization; condition; update) {
    // Code to be executed in each iteration called Loop Body
}
...

```

Here's a breakdown of the components of the for loop:

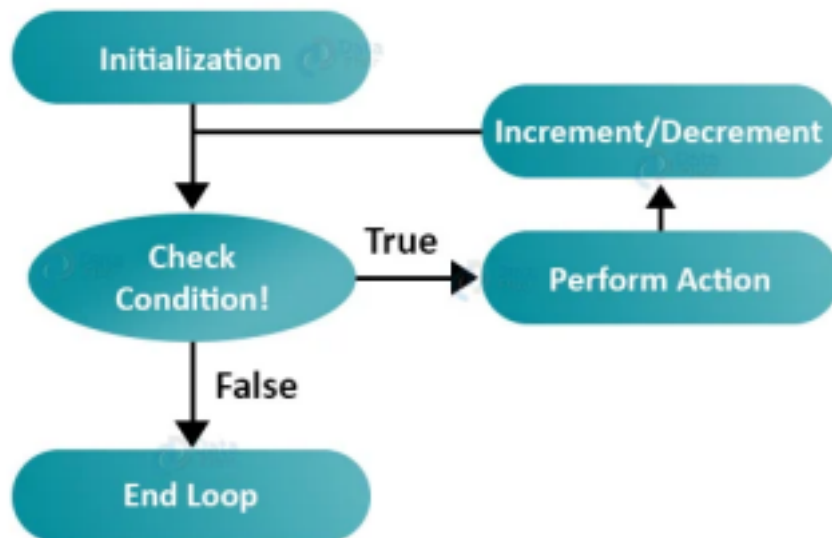
Initialization: This part is executed once at the beginning before the loop starts. It is generally used to initialize a loop control variable, typically a counter. This variable will be used to keep track of the iteration.

Condition: This part is a boolean expression that is evaluated before each iteration. If the condition is true, the loop's body is executed; otherwise, if the condition is false, the loop terminates, and the program continues with the code after the loop.

Update: This part is executed after each iteration, just before the next evaluation of the condition. It typically increments/decrements or updates the loop control variable.

Loop Body: The code inside the loop that is executed in each iteration if the condition is true.

Flow Diagram for Forloop



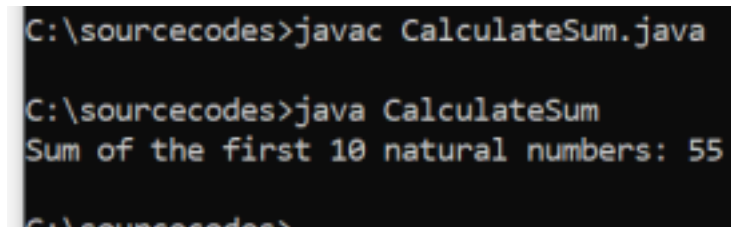
Let us now look at an example to calculate the sum of n numbers using a for loop.


```

public class CalculateSum {
    // Calculating the sum of the first n natural numbers:
    static int calculateSum (int n) {
        int sum = 0;
        // For loop to calculate sum
        for (int i = 1; i <= 10; i++) {
            sum += i;
        }
        return sum;
    }
    public static void main(String[] args) {
        System.out.println("Sum of the first 10 natural numbers: " +
        calculateSum(10));
    }
}

```

Output:



```

C:\sourcecodes>javac CalculateSum.java

C:\sourcecodes>java CalculateSum
Sum of the first 10 natural numbers: 55
C:\sourcecodes>

```

Java for loop can be nested also. A nested for loop in Java is a loop structure where one for loop is placed inside another for loop. In the following example we will create a multiplication table using a nested for loop.

Java

```

public class MultiplicationTable {
    static void createMultiplicationTable (int n, int m ) {
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= m; j++) {
                System.out.print(i * j + "\t");
            }
            System.out.println();
        }
    }

    public static void main(String[] args) {

```

```

        createMultiplicationTable (5, 10);
    }
}

```

Output:

```

C:\sourcecodes>javac MultiplicationTable.java

C:\sourcecodes>java MultiplicationTable
1      2      3      4      5      6      7      8      9      10
2      4      6      8      10     12     14     16     18     20
3      6      9      12     15     18     21     24     27     30
4      8      12     16     20     24     28     32     36     40
5      10     15     20     25     30     35     40     45     50

```

In this

example, the outer loop runs from 1 to 5, and for each value of i, the inner loop runs from 1 to 10 as well, printing the product of i and j along with a tab character.

Note: Java also supports enhanced for loop which is similar to a for loop except that it has some enhanced features. It can be used to iterate over arrays or the elements of a collection without knowing the index of each element. It will be explained after completing the sessions on arrays and collections.

III. User Input

In Java, there are several ways to read input from the console. Here are the most common methods:

- Using Command line argument
- Using Scanner Class
- Using Console Class
- Using BufferedReader Class

Using Command line argument

In Java, you can use command-line arguments by providing them as input when running a Java program from the command line. These arguments are passed to the main method of your Java program as an array of strings. Here's how you can use command-line arguments in Java:

In command line arguments we can add arguments when running the java command.

```
java CommandLineArgumentsExample arg1 arg2 arg3
```

In the above command, arg1, arg2, and arg3 are the command-line arguments that will be passed to the main method as elements of the args array.

We will explain how to access and use the command-line arguments in your Java program:

In the main method of your Java program, you can access the command-line arguments through the args parameter, which is an array of strings (Arrays will be explained later, for the time being, consider that it contains multiple values). Each element of the args array represents one of the command-line arguments provided when running the program.

Example of using command-line arguments:

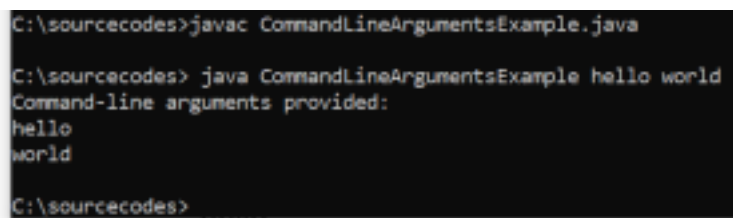
Java

```
public class CommandLineArgumentsExample {  
    public static void main(String[] args) {  
        if (args.length > 0) {  
            System.out.println("Command-line arguments provided:");  
            for (String arg : args) { //enhanced for loop  
                System.out.println(arg);  
            }  
        } else {  
            System.out.println("No command-line arguments provided.");  
        }  
    }  
}
```

If you run the above program with the command

```
java CommandLineArgumentsExample hello world
```

Output:



```
C:\sourcecodes>javac CommandLineArgumentsExample.java  
C:\sourcecodes> java CommandLineArgumentsExample hello world  
Command-line arguments provided:  
hello  
world  
C:\sourcecodes>
```

If you run the program without any command-line arguments (java CommandLineArgumentsExample), the output will be:

Output:

```
C:\sourcecodes> java CommandLineArgumentsExample hello world
Command-line arguments provided:
hello
world

C:\sourcecodes> java CommandLineArgumentsExample
No command-line arguments provided.

C:\sourcecodes>
```

In this way, you can use command-line arguments to pass input data to your Java program and modify its behavior based on the provided arguments.

Using Scanner class

This is probably the most preferred method to take input. Scanner class is available in the java.util package. To use it, you need to create an instance of the Scanner class and then call its various methods to read different types of input. The main purpose of the Scanner class is to parse primitive types and strings using regular expressions, however, it also can be used to read input from the user in the command line. Convenient methods for parsing primitives (nextInt (), nextFloat (), ...) from the tokenized input. Regular expressions can be used to find tokens.

Here are some examples of using the Scanner class in different scenarios: Reading Integers from the Console:

Java

```
import java.util.Scanner;
public class IntegerInputExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter an integer: ");
        int num = scanner.nextInt();

        System.out.println("You entered: " + num);

        scanner.close();
    }
}
```

```
}
```

Here you need to enter an integer on the console. **Output:**

```
C:\sourcecodes>javac IntegerInputExample.java

C:\sourcecodes>java IntegerInputExample
Enter an integer: 12345
You entered: 12345

C:\sourcecodes>
```

Reading Floating-Point Numbers from the Console:

Java

```
import java.util.Scanner;

public class DoubleInputExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter a floating-point number: ");
        double num = scanner.nextDouble();

        System.out.println("You entered: " + num);

        scanner.close();
    }
}
```

Output:

```
C:\sourcecodes>java DoubleInputExample
Enter a floating-point number: 2.58
You entered: 2.58

C:\sourcecodes>
```

Reading Strings from the Console:

Java

```
import java.util.Scanner;
```

```

public class StringInputExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

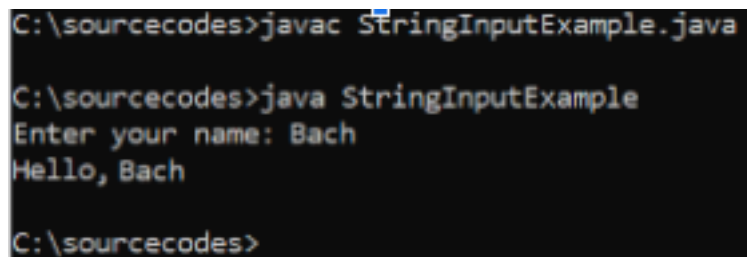
        System.out.print("Enter your name: ");
        String name = scanner.nextLine();

        System.out.println("Hello, " + name + "!");

        scanner.close();
    }
}

```

Output:



```

C:\sourcecodes>javac StringInputExample.java

C:\sourcecodes>java StringInputExample
Enter your name: Bach
Hello, Bach

C:\sourcecodes>

```

Reading Multiple Inputs in a Loop:

Java

```

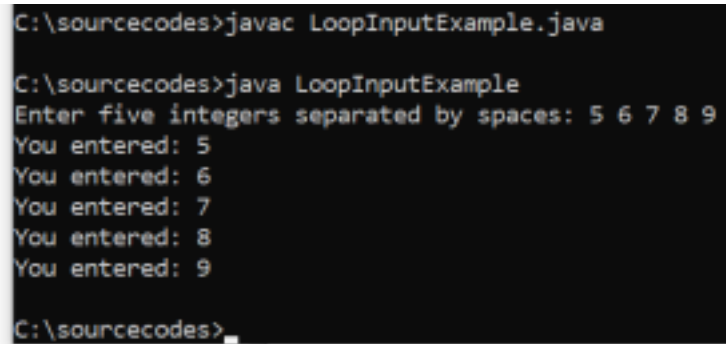
import java.util.Scanner;

public class LoopInputExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter five integers separated by spaces: ");
        for (int i = 0; i < 5; i++) {
            int num = scanner.nextInt();
            System.out.println("You entered: " + num);
        }
        scanner.close();
    }
}

```

Output:

A screenshot of a Windows command prompt window showing the execution of a Java program. The prompt is 'C:\sourcecodes>'. The user enters 'javac LoopInputExample.java'. The prompt returns 'C:\sourcecodes>'. The user enters 'java LoopInputExample'. The program outputs 'Enter five integers separated by spaces: 5 6 7 8 9'. The user enters '5', and the program outputs 'You entered: 5'. The user enters '6', and the program outputs 'You entered: 6'. The user enters '7', and the program outputs 'You entered: 7'. The user enters '8', and the program outputs 'You entered: 8'. The user enters '9', and the program outputs 'You entered: 9'. The prompt returns 'C:\sourcecodes>'.

```
C:\sourcecodes>javac LoopInputExample.java

C:\sourcecodes>java LoopInputExample
Enter five integers separated by spaces: 5 6 7 8 9
You entered: 5
You entered: 6
You entered: 7
You entered: 8
You entered: 9
C:\sourcecodes>
```

Remember to always close the Scanner after you are done reading inputs to release system resources properly. The `scanner.close();` statement ensures that the Scanner object is closed, freeing up resources.

The Scanner class is versatile and can be used to read different data types, perform error handling, and implement more complex input handling logic. It is a powerful tool for interactive console applications and reading data from various sources.

To know further about Scanner class please follow this link:

https://www.w3schools.com/java/java_user_input.asp

Using Console class

To read input from the console using the Console class in Java, you can follow these steps:

Get a reference to the Console object: You need to obtain the Console object, which represents the console of the running Java application. Note that the Console class is available only if the application is running in a console (command-line) environment.

Use Console methods to read input: Once you have the Console object, you can use its methods to read input from the user.

Here's an example of how to read a string from the console using the Console class:

Java

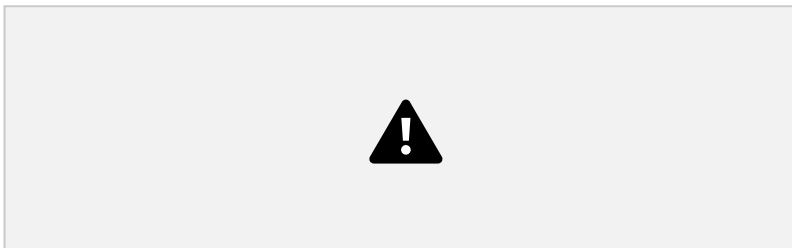
```
import java.io.Console;

public class ConsoleInputExample {
    public static void main(String[] args) {
        Console console = System.console();

        if (console == null) {
            System.out.println("Console not available. Please run this program
from a console environment.");
            return;
        }

        String name = console.readLine("Enter your name: ");
        System.out.println("Hello, " + name + "!");
    }
}
```

Output:



In this example, the `System.console()` method is used to obtain the `Console` object. If `System.console()` returns `null`, it means that the application is not running in a console environment (e.g., running from an IDE or an integrated terminal). Therefore, we display an error message and exit the program.

If the `Console` object is successfully obtained, you can use its `readLine()` method to read a line of text from the user. The `readLine()` method takes an optional prompt message that is displayed to the user before reading the input.

When running the program from the console, the user will see the prompt message ("Enter your name: "), and after entering their name and pressing Enter, the program will print "Hello, [name]!" to the console.

Keep in mind that the Console class is more suitable for simple console input scenarios. For more advanced input handling and error checking, using the Scanner class (as shown in the previous examples) might be more appropriate. Additionally, the Console class might not be available in certain environments, so it's essential to handle such cases gracefully in your code.

Using BufferedReader Class

This is the Java classic method to take input, introduced in JDK1.0. This method is used by wrapping the System.in (standard input stream) in an InputStreamReader which is wrapped in a BufferedReader, we can read input from the user in the command line.

Note: You will learn more about streams in upcoming sessions.

The input is buffered for efficient reading. Here is an example of BuferedReader class:

Java

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class ConsoleInputExample {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
        System.out.print("Enter your name: ");
        String name = br.readLine();
        System.out.print("Enter your age: ");
        int age = Integer.parseInt(br.readLine());
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        br.close();
    }
}
```

Output:



After the 2 hr ILT, the student has to do a 1 hour live lab. Please refer to the Session 1 Lab doc in the LMS.