Java Basic Output

eazy bytes

In Java, we can use any of the following methods to send output to standard output (screen).

1. System.out.print(); // prints given input

2. System.out.println(); // prints given input & moves the cursor to the beginning of the next line.

3. System.out.printf(); // support formatting the given input

? Here you may have a question on how to provide input to the program using keyboard ? Don't worry, we are going to explore in few minutes

Let's understand System.out.println

eazy bytes

System.out.println is a Java statement used to print output to the console. Let's break down the components of this statement:

1. System: System is a class in the java.lang package. It provides access to the standard input, output, and error streams. The out field of the System class represents the standard output stream.

2. out: out is a static field in the System class of type PrintStream. It is an instance of the PrintStream class, which provides methods to print different types of data to various output streams.

3. println: println is a method of the PrintStream class. It stands for "print line" and is used to print a line of text to the output stream. After printing the specified text, it adds a newline character ("\n") to the end, which moves the cursor to the beginning of the next line.

Instead of creating the object of PrintStream every time to invoke the print related methods, we can simply rely on System.out which will behind the scenes will take care of creating the object of PrintStream

Java Basic Input

eazy bytes

Just like a System.out static final variable, we also have System.in static final variable. They represents the objects of PrintStream and InputStream respectively.

Using InputStream object, we can supply the data from input streams like keyboard input or another input source specified by the user.

By invoking the read() method using the InputStream object like shown below, we can accept the data from end user. But it has limitations like it can only accept a single char and return the output as ASCII representation of the given input char.

```
int num = System.in.read();
```

To enable efficient reading of the data from a InputStream, Java provides below options,

1) java.io.BufferedReader

2) java.util.Scanner (Since 1.5 version)

java.io.BufferedReader

eazy bytes

BufferedReader class reads text from a character-input stream, buffering characters to provide for the efficient reading of characters, arrays, and lines.

```
public class BufferedReader extends Reader {

}
```

The Java BufferedReader class inherits the abstract Reader class because it is a specialization of the Reader class, but buffering is enabled.

BufferedReader employs an internal buffer of 8 KB (8192 bytes or characters), with the option for modification. During a read operation, BufferedReader stores a chunk of characters obtained from the disk within its internal buffer.

Subsequently, it processes these internal buffer characters individually. This approach minimizes the frequency of I/O operations, which can be resource-intensive due to disk communication. Consequently, this method enhances the efficiency and speed of character reading by reducing the overall number of I/O (Input/Output) operations.

java.io.BufferedReader

eazy bytes

Below are the constructors supported by BufferedReader,

Constructor	Description
BufferedReader(Reader in)	Creates a buffering character-input stream that uses a default-sized input buffer (8192 bytes).
BufferedReader(Reader in, int sz)	Creates a buffering character-input stream that uses an input buffer of the specified size.

Read data from console using **BufferedReader**

eazy bytes

Below is the code that can be used to read data from console by InputStreamReader And BufferedReader,

```
InputStreamReader inpSReader = new InputStreamReader(System.in);
BufferedReader reader = new BufferedReader(inpSReader);

System.out.println("Type input & press enter when done: ");
String str = reader.readLine();
System.out.println("You entered '" + str + "'");
reader.close();
```

readLine() method reads a line of text where as read() method reads a single character.

To connect to an input device, like a keyboard, we establish a connection by creating an InputStreamReader object. This is achieved through a constructor that accepts an InputStream, typically using the standard input stream System.in.

Subsequently, a BufferedReader object is instantiated using the newly created InputStreamReader. The BufferedReader allows us to efficiently read the input.

We then prompt the user to enter a string via the console, reading it using the readLine() method. Finally, we display the entered string, and when the reader object is no longer needed, it is closed to manage system resources effectively.

BufferedReader examples to identify even number

eazy bytes

Below is the code that can be used to accept a number from user by using BufferedReader and identify if it is a even number or not,

```
public class BufferedReaderEvenOddDemo {

    public static void main(String[] args) throws IOException {
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader bf = new BufferedReader(isr);
        System.out.println("Please enter a numeric value...");
        String input = bf.readLine();
        int num  = Integer.parseInt(input);
        if(num%2==0){
            System.out.println("You have entered an even number");
        }else {
            System.out.println("You have entered a odd number");
        }
        bf.close();
    }

}
```

BufferedReader examples to sum two numbers

eazy bytes

Below is the code that can be used to accept 2 numbers from user by using BufferedReader and perform sum of them,

```
public class BufferedReaderSumDemo {

    public static void main(String[] args) throws IOException {
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader bf = new BufferedReader(isr);
        System.out.println("Please enter a first numeric value...");
        String input1 = bf.readLine();
        System.out.println("Please enter a second numeric value...");
        String input2 = bf.readLine();
        int num1   = Integer.parseInt(input1);
        int num2   = Integer.parseInt(input2);
        int result = num1 + num2;
        System.out.println("The sum of two given numbers is : "+result);
        bf.close();
    }

}
```

BufferedReader examples to identify a prime number

eazy bytes

Below is the code that can be used to accept a number from user by using BufferedReader and identify if it is a prime number or not,

```
public class PrimeNumberChecker {

    public static void main(String[] args) throws IOException {
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader bf = new BufferedReader(isr);
        System.out.println("Please enter a numeric value to identify if it is
                            a prime number or not...");
        String input = bf.readLine();
        int num  = Integer.parseInt(input);
        boolean isPrime = isPrime(num);
        if(isPrime){
            System.out.println("Given number is a prime number");
        }else {
            System.out.println("Given number is not a prime number");
        }
        bf.close();
    }

    private static boolean isPrime(int num) {
        if(num <= 1) {
            return false;
        }
        for (int i=2;i <= Math.sqrt(num);i++) {
            if(num%i==0) {
                return false;
            }
        }
        return true;
    }

}
```

Read data from a file using **BufferedReader**

eazy bytes

Below is the code that can be used to read data from a file using the readLine() method of BufferedReader,

```
String filePath = "path/to/your/file.txt";
FileReader fileReader = new FileReader(filePath);
BufferedReader reader = new BufferedReader(fileReader);

String line;
while ((line = reader.readLine()) != null) {
    System.out.println(line);
}

// Close the BufferedReader outside the try-catch block
reader.close();
```

The readLine() method is employed for reading a line of text, considering a line terminated by a line feed ('\n'), a carriage return ('\r'), or a carriage return followed immediately by a linefeed.

This method returns a string containing the line's contents, excluding any line-termination characters. If the end of the stream is reached, it returns null.

In the given example, we utilized the readLine() method to sequentially read a file line by line. We create a FileReader object encapsulated within a BufferedReader object in Java by passing it through the constructor.

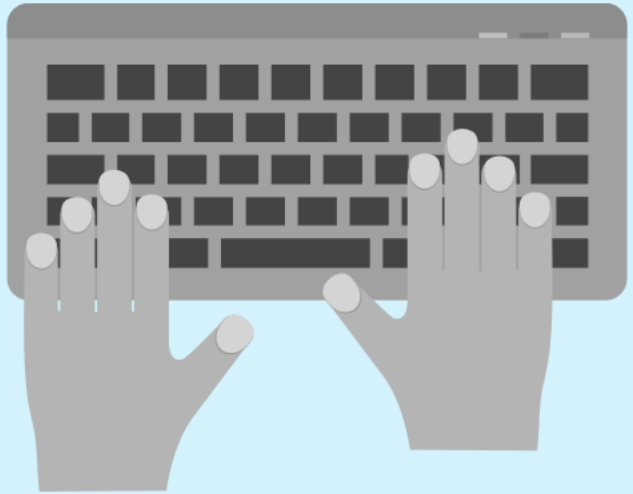
A while loop is employed to read each line of the file and print it, excluding new line characters, until a null value is encountered, indicating no further lines to read.

The reader object is closed when it is no longer required.

java.util.Scanner

eazy bytes

The **Scanner** class is part of the **java.util** package in Java and it provides a more convenient way to read input from various sources such as files and input streams such as keyboard etc. The Scanner class allows you to read input of different data types, such as integers, floating-point numbers, strings, and characters. To use the Scanner class, you need to first create an instance of the Scanner class and specify the input source. Here's an example:



```java
import java.util.Scanner;

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

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

        System.out.print("Enter your age: ");
        int age = scanner.nextInt();

        System.out.println("Hello " + name +
            ", you are " + age + " years old.");

        scanner.close();
    }
}
```

In this example, we first create an instance of the Scanner class and pass in the System.in object as the input source. We then use the next() method to read the next token of input as a string and assign it to the name variable. We also use the nextInt() method to read the next token of input as an integer and assign it to the age variable. Finally, we print out a message that includes the name and age variables.

It's important to note that when you're done using the Scanner object, you should close it using the **close()** method to release any system resources that were associated with it.

Java Basic Input

eazy bytes

Here are some important methods of the **Scanner** class:

- **next()** It is used to read a word from the user
- **nextInt()** It is used to read an int value from the user
- **nextBoolean()** It is used to read a boolean value from the user
- **nextFloat()** It is used to read a float value from the user
- **nextDouble()** It is used to read a double value from the user
- **nextLine()** It is used to read a String value from the user until a enter button pressed
- **nextLong()** It is used to read a long value from the user
- **nextShort()** It is used to read a short value from the user
- **nextByte()** It is used to read a byte value from the user

Apart from the above methods which helps to read input data, Scanner also has methods that helps to check if the next data/token available to read,

- **hasNext()** It returns true if Scanner has another token for input or to mark whether the input is ended or not
- **hasNextInt()** It is used to check if the next token or input is of int data type or not
- **hasNextFloat()** It is used to check if the next token or input is of float data type or not
- **hasNextDouble()** It is used to check if the next token or input is of double data type or not
- **hasNextLine()** It is used to check if there is another line or string in the input or not
- **hasNextLong()** It is used to check if the next token or input is of a long data type or not
- **hasNextShort()** It is used to check if the next token or input is of short data type or not
- **hasNextByte()** It is used to check if the next token or input is a byte or not

Read data from a file using **Scanner**

eazy bytes

Below is the code that can be used to read data from a file using the Scanner,

```
File file = new File("path/to/your/file.txt");
Scanner scanner = new Scanner(file);

while (scanner.hasNextLine()) {
    String line = scanner.nextLine();
    System.out.println(line);
}

scanner.close();
```

eazy bytes

BufferedReader

BufferedReader uses buffering to read a sequence of characters from a character-input stream

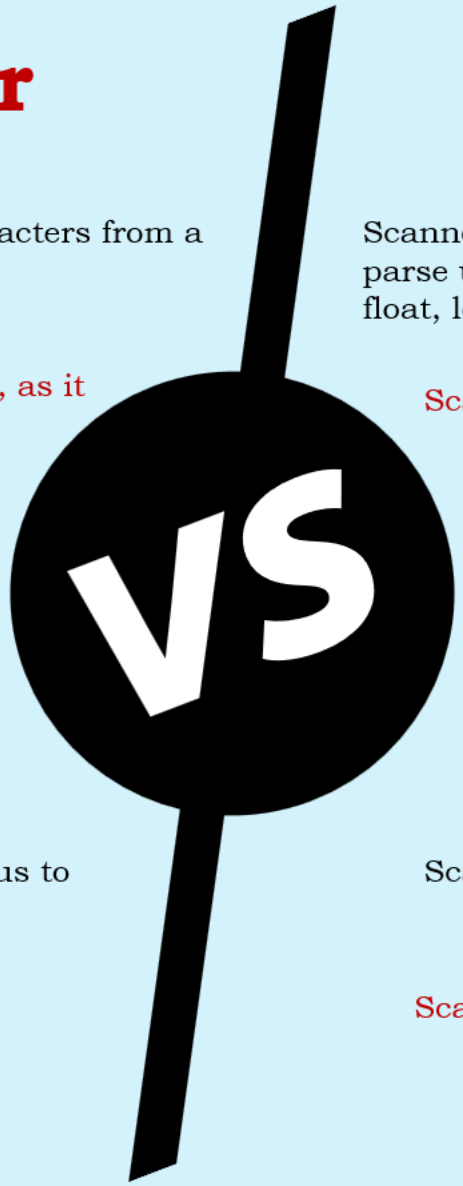
BufferedReader has been around for longer than Scanner, as it has been included in Java since the release of JDK 1.1

BufferedReader has larger buffer size of 8KB & is better suited for reading long strings from a file as it allows changing the size of the buffer

BufferedReader is a synchronized class, which means you can share a BufferedReader object between multiple threads

BufferedReader explicitly throws IOException and forces us to handle it

BufferedReader is faster and efficient



Scanner

Scanner is a more versatile tool as it can not only read strings but also parse user input and extract various data types including int, short, byte, float, long, and double

Scanner is introduced in the JDK 1.5 release

Scanner's buffer size of 1KB and is more suitable with short input or input other than strings as it has a fixed buffer size

Scanner is not a synchronized class & you can not share a Scanner object between multiple threads

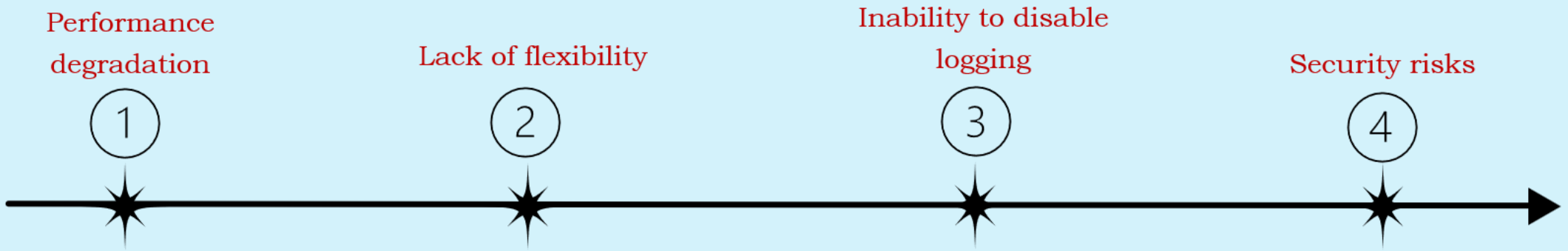
Scanner hides IOException

Scanner is slower and less efficient

Don't use System.out.println in Production code

eazy bytes

Using System.out.println in production code is generally not recommended & it should be used only during learning Java, debugging, testing. Because it can cause a number of drawbacks, such as:



Performance degradation

Printing to the console can be slow and can impact the performance of your application, especially if you are printing a large number of messages or if your application runs for a long time.

Lack of flexibility

When you print messages to the console, you have limited control over how they are formatted and displayed. This can make it difficult to read and understand the output, especially if there are a large number of messages or if they contain complex information.

Inability to disable logging

If you use System.out.println to print messages, there is no easy way to disable the logging without modifying the code. This can make it difficult to debug or troubleshoot issues in production.

Security risks

If you are logging sensitive information, such as passwords or personal data, printing it to the console can pose a security risk, as anyone with access to the console can see the information.

Instead of using System.out.println, it is recommended to use a logging framework, such as java.util.logging, Log4j, or SLF4J, that provides more advanced logging features, such as configurable logging levels, log rotation, log aggregation, and more. A logging framework can help you manage the logging output and format the log messages in a way that is more suitable for your needs. Additionally, a logging framework provides a way to enable and disable logging at runtime without modifying the code, which can be useful in production environments.

java.util.logging

eazy bytes

java.util.logging is a built-in Java package that provides a logging framework for Java applications. It is designed to provide a simple, flexible, and powerful logging mechanism that can be used to record various events and activities that occur during the execution of a Java application. Below is a sample program that leverages logging to print messages,

```java
public class LoggerDemo {

    private static Logger logger = Logger.getLogger(LoggerDemo.class.getName());

    public static void main(String[] args) {

        // set log level to SEVERE, default level info
        logger.setLevel(Level.SEVERE);

        logger.info("This is info level logging");
        logger.log(Level.WARNING, "This is warning level logging");
        logger.log(Level.SEVERE, "This is severe level logging");

        System.out.println("Hello using System.out.println");
    }

}
```

There are 7 logging levels in java.util.logging package :

SEVERE (highest severity)
WARNING
INFO (default)
CONFIG
FINE
FINER
FINEST (lowest severity)

If we configure the logging level to SEVERE, it will log messages at or above the SEVERE level. Given that SEVERE is the most critical logging level, only messages with a SEVERE level will be logged. Where as with INFO, it will log info, warning, severe levels.

While learning or using Java, java.util.logging should be enough. But for larger applications that leverages Java frameworks like Spring, SpringBoot etc, it is recommended to use the logging frameworks like Log4j, or SLF4J, that provides more advanced logging features.