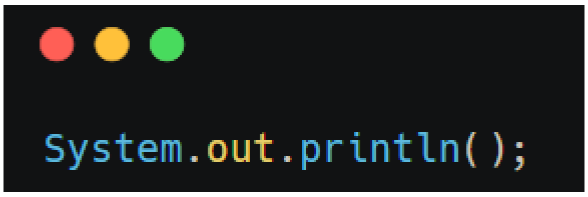
11.8-User Input using BufferedReader and Scanner

Introduction:

In Java, when we want to print output to the console, we use the statement:



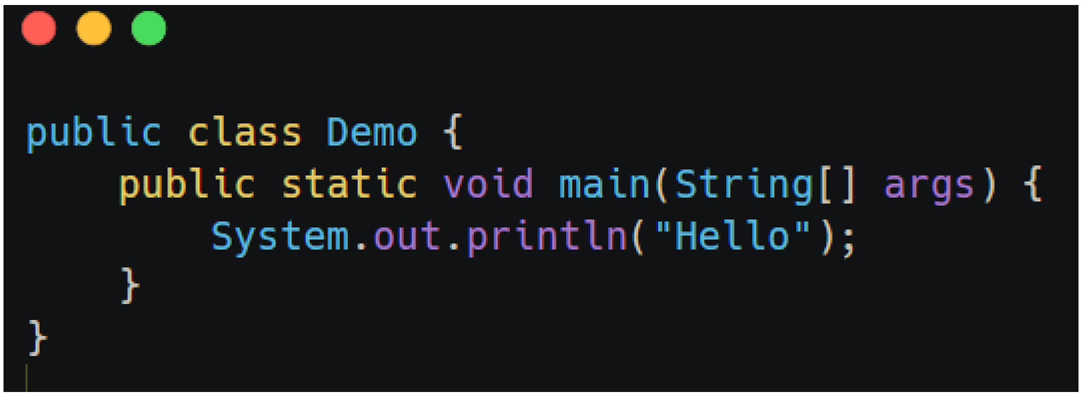
```java
System.out.println();
```

In this statement, **println()** is a method from the **PrintStream** class, while out is a static and final object of the **System** class.

Breaking it down:

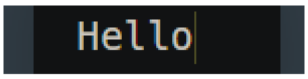
- System is the class.
- out is the static object.
- println is the method from the PrintStream class.

Example:



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

Output:



```
Hello
```

Explanation:

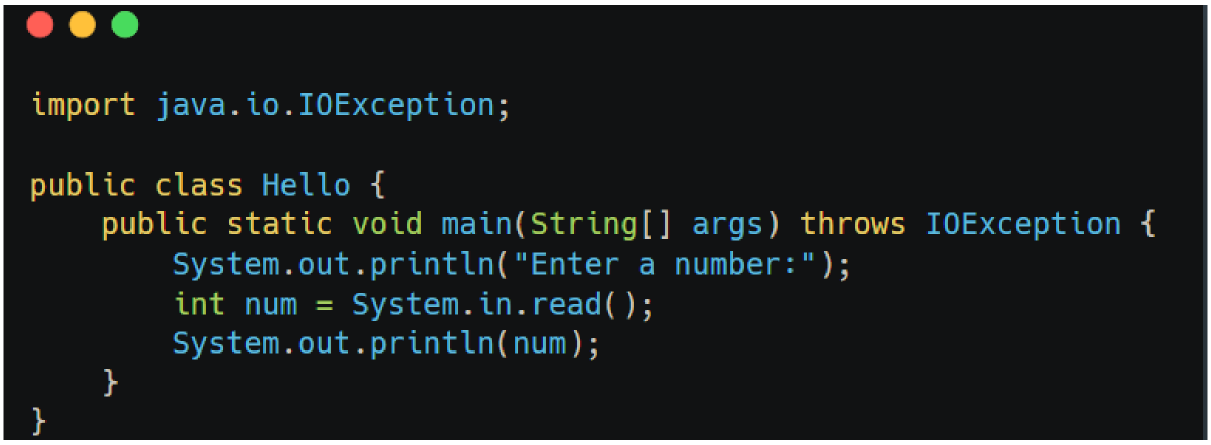
Here, we use the out object to display the output on the console. The **System.out.println()** statement executes and prints "Hello" to the console.

Ways to Read Input from the Console in Java:

1. Approach 1: Using System.in.read()

Just like we use out to print output to the console, we can use in (another object of the System class) to take input from the user. The (in) object is associated with the **InputStream** class, we use the **read()** method, which returns an int value and throws a **IOException** (checked exception). Thus, we must handle this exception.

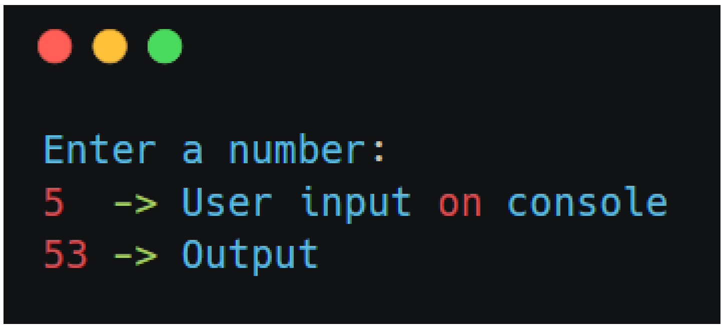
Example:



```java
import java.io.IOException;

public class Hello {
    public static void main(String[] args) throws IOException {
        System.out.println("Enter a number:");
        int num = System.in.read();
        System.out.println(num);
    }
}
```

Output:



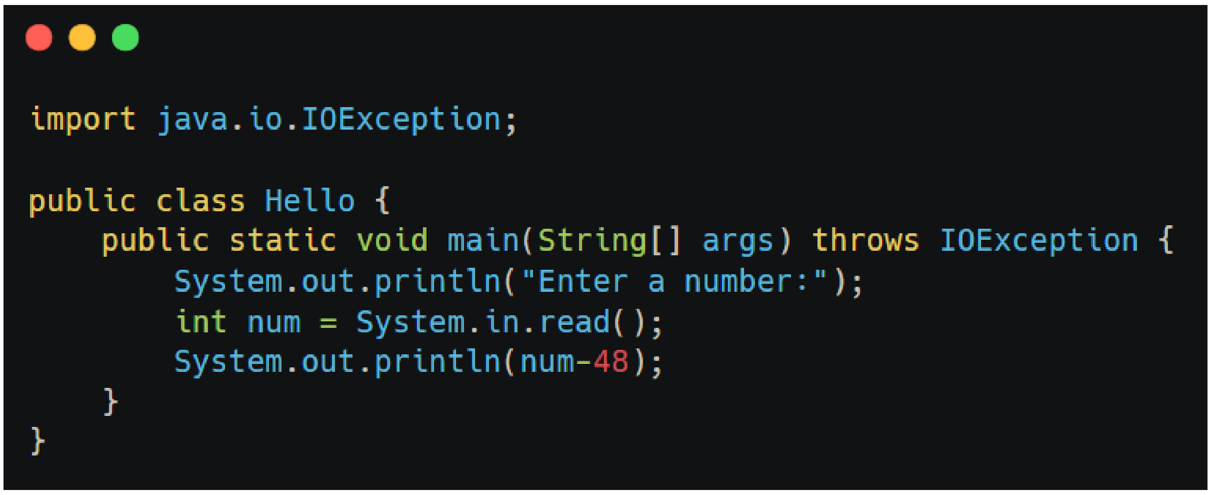
```
Enter a number:
5  -> User input on console
53 -> Output
```

Explanation:

When we as a user gives input '5', the output is 53 because the **System.in.read()** method returns the **ASCII** (American Standard Code for Information Interchange) value of the input. The ASCII value of '5' is 53.

- **Example:** If the user inputs 'a', the output will be 97, as it is the ASCII value of 'a'.

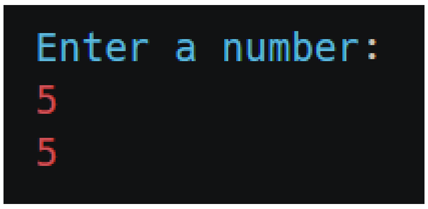
Example for Character Input:



```java
import java.io.IOException;

public class Hello {
    public static void main(String[] args) throws IOException {
        System.out.println("Enter a number:");
        int num = System.in.read();
        System.out.println(num-48);
    }
}
```

Output:



```
Enter a number:
5
5
```

To get the actual number entered, we subtract the ASCII value of '0' (which is 48). For example:

num - 48 = actual number

Thus, if the input is 53 (for '5'), we do:

53 - 48 = 5

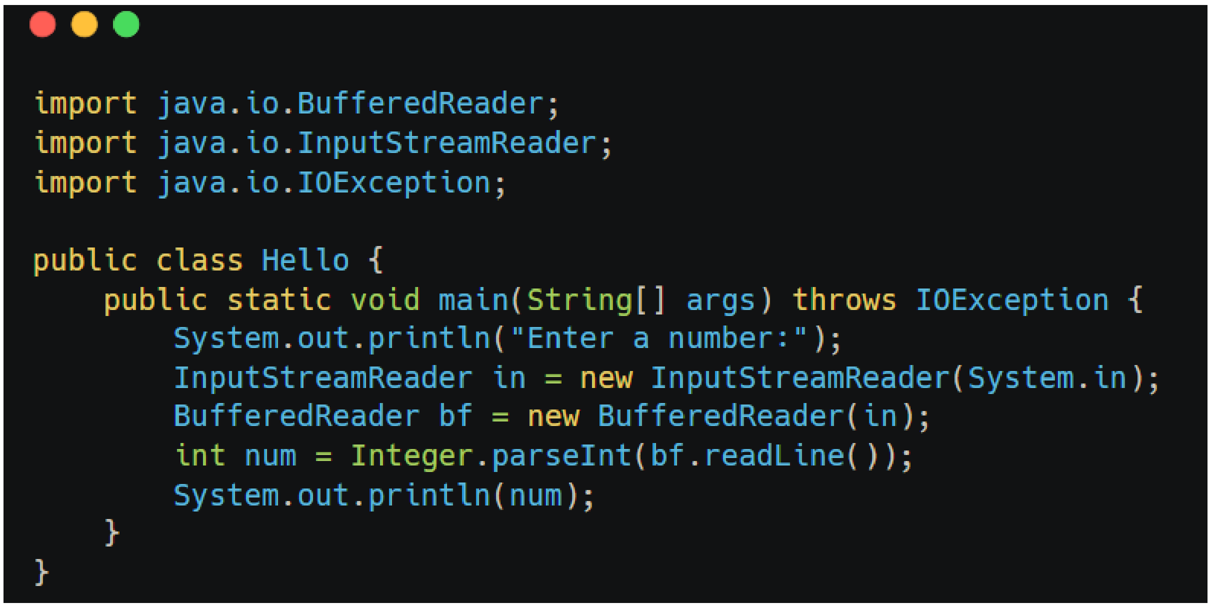
This approach reads one character at a time, returning its ASCII value.

---

2. Approach 2: Using BufferedReader

To read multiple characters or entire lines of input, we use the **BufferedReader** class from the *java.io package*. This class provides the readLine() method, which reads a complete line of text and returns it as a string.

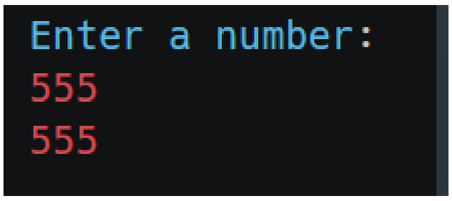
Example:



```java
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class Hello {
    public static void main(String[] args) throws IOException {
        System.out.println("Enter a number:");
        InputStreamReader in = new InputStreamReader(System.in);
        BufferedReader bf = new BufferedReader(in);
        int num = Integer.parseInt(bf.readLine());
        System.out.println(num);
    }
}
```

Output:



```
Enter a number:
555
555
```

->user input

->output

Explanation:

Here, we create an **InputStreamReader** object to read from System.in, which is passed to the **BufferedReader** constructor. The readLine() method returns a string, and since we want an integer, we convert the input using Integer.parseInt() which will parse the string into integer.

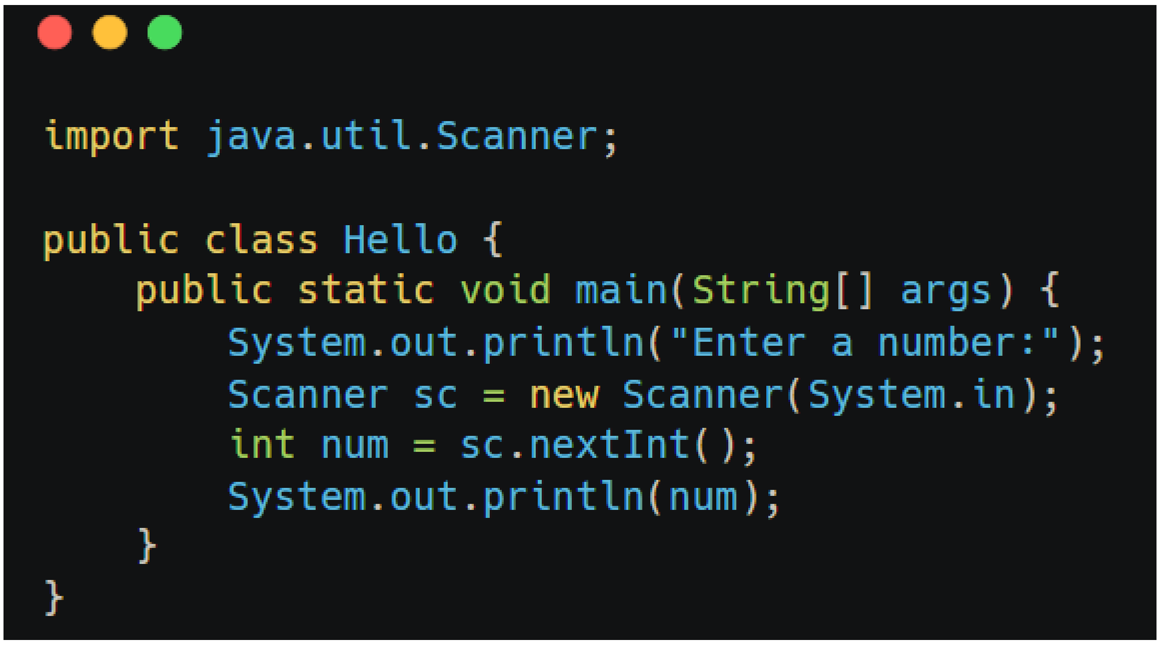
Closing Resources:

When using BufferedReader (or any resource that reads files or data from external sources), it is good practice to close the resource after use to avoid potential memory leaks. Even though the compiler may not give you errors , failing to close resources can lead to issues such as file locks or security vulnerabilities.

3. Approach 3: Using Scanner

In **Java 1.5**, the **Scanner** class was introduced, which became the preferred way to read user input. It is simpler and offers convenient methods to parse primitive types (e.g., nextInt(), nextFloat()) as well as strings.

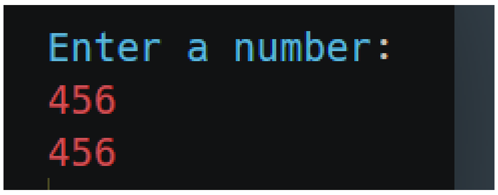
Example:



```java
import java.util.Scanner;

public class Hello {
    public static void main(String[] args) {
        System.out.println("Enter a number:");
        Scanner sc = new Scanner(System.in);
        int num = sc.nextInt();
        System.out.println(num);
    }
}
```

Output:



```
Enter a number:
456
456
```

->user Input

->output

Explanation:

- We create a Scanner object and use the **nextInt()** method to retrieve the integer input. The Scanner class automatically handles converting the input to the desired type (integer, string, etc.).
- This is currently the most popular approach for reading input in Java.

Comparison of BufferedReader and Scanner:

Thread Safety:

- **BufferedReader:** Synchronized (thread-safe).
- **Scanner:** Not synchronized (not thread-safe).

If you’re working in a **multi-threaded** environment, BufferedReader is the better option.

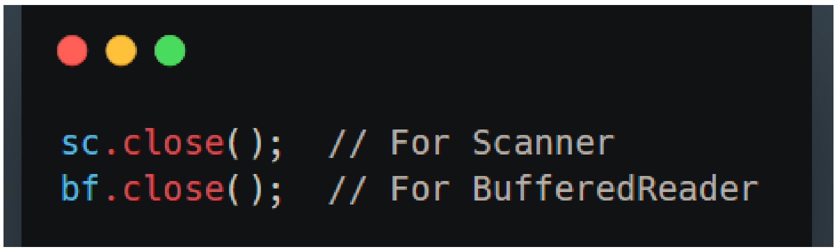
Buffer Size:

- **BufferedReader:** Default buffer size is 8 KB.
- **Scanner:** Default buffer size is 1 KB.

BufferedReader offers better performance for reading large amounts of data or long strings. You can also specify the buffer size when creating a BufferedReader.

Closing Input Streams:

Always remember to close the BufferedReader or Scanner object after use to prevent memory leaks. Example:



```
sc.close();   // For Scanner
bf.close();   // For BufferedReader
```