**Reading from a File using FileInputStream in Java**

The **FileInputStream** class is a subclass of **InputStream** that allows a program to read bytes from a file. It is used to open a file for reading, and it provides methods for reading **bytes**of data from the file. However, it does not provide any methods for parsing the data or for reading data types such as integers or strings.

That's why we combine Scanner and FileInputStream class to be able to extract data

The **Scanner** class, on the other hand, is a utility class that provides methods for parsing different types of input, such as integers, floating-point numbers, and strings. It can read from any input stream, including a **FileInputStream**.

By combining the **FileInputStream** and **Scanner** classes, you can easily read and parse the contents of a file in your Java program.

Here is an example of how to use the **Scanner** class to read input from an input stream in Java:

1. import java.util.Scanner;
3. public class Main {
4. public static void main(String[] args) throws Exception {
5. // Open the input stream
6. FileInputStream in = new FileInputStream("input.txt");
8. // Create a Scanner object that reads from the input stream
9. Scanner sc = new Scanner(in);
11. // Read and process the input
12. while (sc.hasNextLine()) {
13. String line = sc.nextLine();
14. // Process the line of input
15. // ...
16. }
18. // Close the input stream and the Scanner object when you are done using them
19. in.close();
20. sc.close();
21. }
22. }

In this example, the **Scanner** object is created by passing it an **InputStream** object as an argument (e.g. **Scanner sc = new Scanner(in);**). The **Scanner** object can then be used to read lines of text from the input stream using the **nextLine()** method.

The example also shows how to close both the input stream and the **Scanner** object when they are no longer needed, using the **close()** method. This is important to free up system resources and prevent resource leaks.

When a resource leak occurs?

A resource leak occurs when a program consumes a resource, such as memory or a file handle, but fails to release it when it is no longer needed. This can lead to the exhaustion of available resources over time, which can cause the program to crash or become unstable.

In the case of the **Scanner** class, a resource leak can occur if the **Scanner** object is not closed when it is no longer needed. For example, if you create a **Scanner** object to read from a file and then forget to close it, the file handle will remain open and other programs will not be able to access the file.

To avoid resource leaks, it is important to properly manage resources in your program by allocating and releasing them as needed. In the case of the **Scanner** class, this means calling the **close()** method on the **Scanner** object when you are done using it.  
Here is an example of a program that consumes a resource (in this case, a file handle) and fails to release it, leading to a resource leak:

1. Copy code
2. import java.io.FileInputStream;
3. import java.util.Scanner;
5. public class Main {
6. public static void main(String[] args) throws Exception {
7. // Open the input stream
8. FileInputStream in = new FileInputStream("input.txt");
10. // Create a Scanner object that reads from the input stream
11. Scanner sc = new Scanner(in);
13. // Read and process the input
14. while (sc.hasNextLine()) {
15. String line = sc.nextLine();
16. // Process the line of input
17. // ...
18. }
20. // Forget to close the input stream and the Scanner object
21. }
22. }

In this example, the program opens an input stream to read from a file, but we intentionally forget to close the stream when it is done. As a result, the file handle remains open and other programs will not be able to access the file. This can lead to r**esource exhaustion over time**, as more and more file handles **are left open and not released**.

Course content

AI Assistant