Lec 42 File IO P1:

File Input/Output (File IO) in Java allows you to store data permanently on your computer, instead of just keeping it temporarily in memory. You can create, read, write, and delete files using Java programs, which helps you manage data that needs to be saved even after the computer is turned off.

Comparator and Collection Recap

The difference between Comparable and Comparator interfaces is explained. Comparable is used for default sorting of objects, while Comparator allows custom sorting, such as sorting by different object fields.

Introduction to File Input/Output (IO)

File IO is introduced as a way to achieve permanent data storage, unlike variables, arrays, or collections, which only store data temporarily in RAM.

Temporary vs. Permanent Storage

Variables, arrays, and collections keep data only as long as the program runs; permanent storage means saving data to a hard disk so it remains after shutdown.

Real-life Example: Downloading Files

When you download a file, it moves from a server to your hard disk, making it available even after restarting your computer. This demonstrates data persistency.

Purpose of File IO in Java

File IO is used to achieve data persistence, allowing programs to create, write, read, and delete files and folders on the disk.

Creating Files Programmatically

Files can be created not just through the operating system, but also directly from Java programs, allowing automation and more control over file management.

Steps to Create a File in Java

To create a file, you specify the file path, use the File class from java.io, and call the createNewFile() method. Paths must be accurate, and double slashes are used as separators in Java strings (First put “ ”, then paste the path inside it so that it automatically adds the double slashes).

Creating Multiple Files Using Loops

You can use loops in Java to create many files at once by changing the file name in each iteration, which is useful for bulk operations.

Deleting Files in Java

Files can be deleted programmatically using the delete() method of the File class. If deleted via code, the file does not go to the recycle bin, unlike manual deletion.

Custom Recycle Bin Logic

To mimic recycle bin behavior in code, you must first copy the file to a backup location before deleting it, as Java's delete() does not handle this automatically.

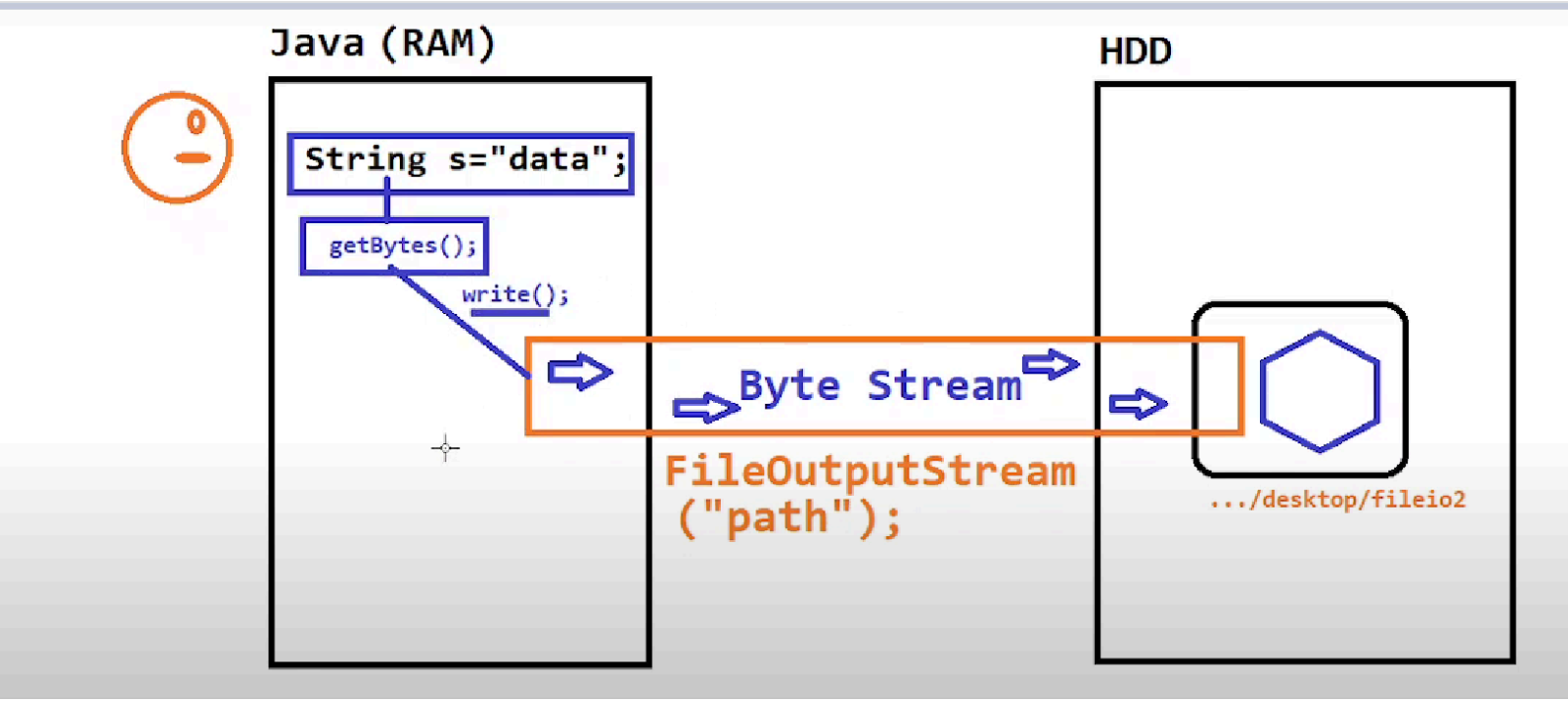
Writing Data to Files

To write data to a file, you use the FileOutputStream class, which acts as a "pipe" from RAM to the hard disk. Data is converted into bytes before being written.

FileOutputStream and Byte Streams

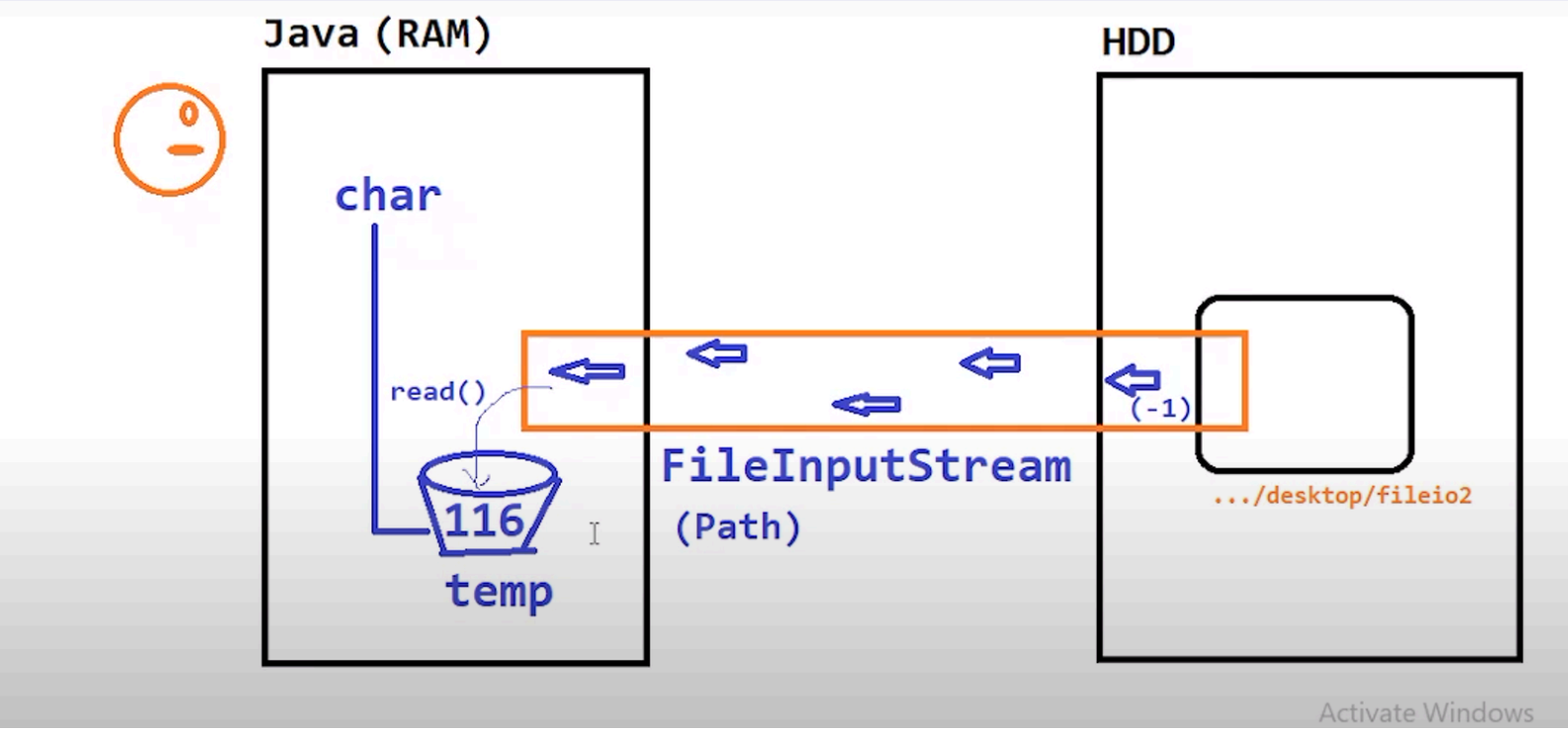
FileOutputStream sends data as a byte stream from memory to the file. Strings must be converted to byte arrays using the getBytes() method before writing.

getBytes() method return a array.



Reading Data from Files

To read data, use FileInputStream, which reads bytes in integer form(ASCII) from the file and converts them back to characters. Data is read one byte at a time, and a loop continues until the end-of-file marker (-1) is reached.



Buffered Streams for Efficiency

Reading or writing one byte at a time is slow; buffered streams like BufferedInputStream and BufferedOutputStream improve performance by handling data in larger chunks.

Importance of Closing Streams

Files cannot be deleted if they are still open in a stream. Always close streams after reading or writing to avoid resource leaks and errors.

Task: File Filtering Practice

A practical task is given: copy names starting with a vowel from one file to another using file streams, demonstrating reading, filtering, and writing data programmatically.

Next Session:

The session explains how file operations work in Java, focusing on reading, writing, appending, and manipulating file contents, as well as creating folders and understanding the difference between regular and buffered streams. Buffered streams are highlighted as a way to make file input and output much faster and more efficient, especially with large files.

Introduction to File Operations

File operations in Java allow reading data from files, but you cannot update a file's content in place. Instead, you read the data, manipulate it as needed, and then write it back, replacing the old content.

File Overwriting and Appending

When writing to a file using FileOutputStream, if the file exists, its old content is deleted and replaced with the new data (overwriting). To keep the old content and add new data, you must use the append mode; otherwise, the previous data is lost.

Appending Data and File Update Limitations

Appending is possible if you want to add new data to the end of a file without deleting existing data. However, you cannot remove or modify data in the middle of a file directly; this requires reading, changing, and rewriting the entire content.

Practice Task: Filtering and Moving Data Between Files

The task involves reading names from one file, extracting those that start with vowels, and writing them to another file. Data is read into a string, split into names, filtered, and then written to a new file.

Filtering Data by Criteria

Names are split by spaces, and each is checked to see if it starts with a vowel (a, e, i, o, u). Matching names are concatenated into a new string, which can be further

cleaned (e.g., trimming extra spaces) before writing to a file.

Writing Filtered Data to a File

The filtered string is written to a new file using FileOutputStream. Using the correct method ensures the data is stored as intended, and closing streams after writing is important to avoid resource leaks.

Appending Data with FileOutputStream

To append data instead of overwriting, the FileOutputStream constructor is called with a boolean flag set to true. This ensures new data is added to the end of the file.

Append:

```
package com.mainapp;
import java.io.FileOutputStream;
public class Launch {

    public static void main(String[] args) throws Exception {

        FileOutputStream fos = new FileOutputStream("C:\\Users\\codehunt\\Desktop\\f2\\pqrs.txt",true);
        fos.write(" kaju".getBytes());
        fos.close();

    }
}
```

Without Path:

```java
package com.mainapp;
import java.io.FileOutputStream;
public class Launch {

    public static void main(String[] args) throws Exception {


        FileOutputStream fos = new FileOutputStream("pqrs.txt");
        fos.write(" kaju".getBytes());
        fos.close();

    }
}
```

Creating and Managing Folders

Java can create new folders using the File class and the mkdir() method. Folder names can be dynamically set, and file/folder paths need to be constructed carefully, considering operating system differences (e.g., slashes in paths).

How to create a folder :

```java
package com.mainapp;
import java.io.File;
import java.util.Scanner;
public class Launch {

    public static void main(String[] args) throws Exception {

//          String foldername=new Scanner(System.in).next();
//
//          File file = new File("C:\\Users\\codehunt\\Desktop\\f2\\"+foldername);
//          boolean mkdir = file.mkdir();
//          System.out.println(mkdir);

          String foldername=new Scanner(System.in).next();

          String path="C:\\Users\\codehunt\\Desktop\\f2"+File.separator+foldername;

          File file = new File(path);
          boolean mkdir = file.mkdir();
          System.out.println(mkdir);

    }
}
```

Introduction to Buffered Streams

BufferedInputStream and BufferedOutputStream are wrappers around regular file streams that use an internal buffer (typically 8 KB) to read or write data in larger chunks, reducing the number of direct disk accesses and improving performance.

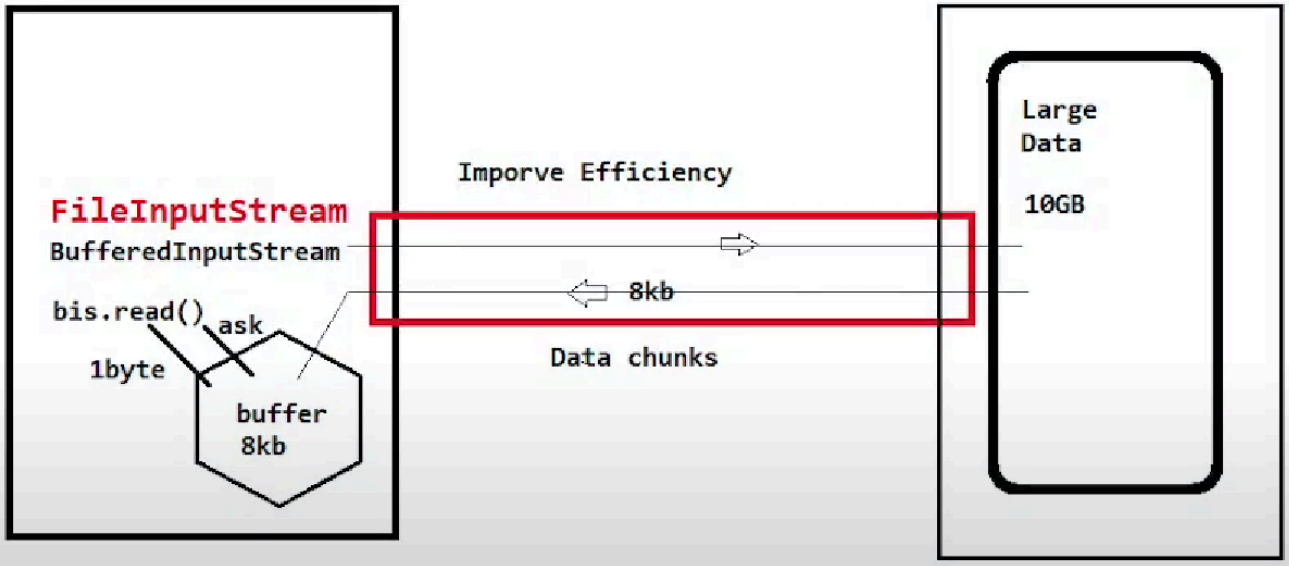
How FileInputStream Reads Data

FileInputStream reads data one byte at a time, hitting the hard disk for each byte, which is slow, especially for large files.

How BufferedInputStream Improves Efficiency

BufferedInputStream reads large chunks (e.g., 8 KB) from the disk at once and stores them in memory.

Subsequent reads fetch data from this buffer, reducing disk operations and increasing speed.



Buffer Size and Chunked Reading

The buffer size is limited (e.g., 8 KB), so for very large files, data is read in repeated chunks as the buffer empties. This avoids loading the entire file into memory at once.

Buffered Output Stream and Efficient Writing

Buffered Output Stream collects data in a buffer and writes it to the disk in chunks, making writing operations faster and more efficient than writing byte by byte.

Proper Closing of Streams

Buffered streams should be closed explicitly to ensure all buffered data is written and resources are released. Closing the buffered stream also closes the underlying file stream.

Practice and Project Preparation

The session encourages practicing these concepts and mentions a minor project involving file operations, such as creating a recycle bin, to reinforce learning.