RAJU KUMAR

Files and I/O Stream Day-9

Introduction to Java I/O (Input/Output)

In Java, **Input/Output (I/O)** refers to the process of reading from and writing to external sources such as files, network connections, or other devices. The `java.io` package provides the necessary classes and interfaces to perform these operations.

Java I/O operations are performed through **streams**, which can be thought of as sequences of data. **Streams** represent input and output resources, like files, and provide mechanisms to read from or write to these resources.

There are two types of streams in Java:

- **Byte Streams**: Deal with raw binary data, such as reading and writing bytes to a file (e.g., `FileInputStream` and `FileOutputStream`).
- **Character Streams**: Deal with text data and handle character encoding automatically (e.g., `FileReader` and `FileWriter`).

What is `FileReader`?

`FileReader` is a **class** in Java that is part of the **character stream** category. It is used to read the contents of a file, interpreting the file as a stream of characters. It extends the `InputStreamReader` class, which makes it a subclass of the broader `Reader` class in Java.

- `FileReader` reads files in the form of **characters** and is designed for text files (as opposed to binary files).
- `InputStreamReader` is a bridge from byte streams to character streams. It reads bytes from the underlying input stream (like a `FileInputStream`) and decodes them into characters using a specified character encoding.
- `Reader` is an abstract class that defines basic methods for reading characters, arrays of characters, and lines from a character-based input stream.

FileReader - Class or Interface?

`FileReader` is a **class**, not an interface. It implements the `Readable` interface, which means that it is a concrete class providing a specific implementation to read characters from a file.

```
public class FileReader extends InputStreamReader {
    public FileReader(String fileName) throws FileNotFoundException {
        super(new FileInputStream(fileName));  // Creates a FileInputStream for reading bytes from t
he file
    }

    public FileReader(File file) throws FileNotFoundException {
        super(new FileInputStream(file));  // Creates a FileInputStream for reading bytes from a File
    }

    // The read() method is inherited from InputStreamReader
}
```

Where Did `FileReader` Come From?

To understand where `FileReader` fits, let's look at the class hierarchy:

1. `Object`: All Java classes inherit from the `Object` class. It provides basic methods like `toString()`, `equals()`, etc.
2. `InputStreamReader` (extends `Reader`): `InputStreamReader` is a bridge from byte streams to character streams, meaning it reads bytes from an underlying byte stream (like `FileInputStream`) and converts them into characters using a character encoding.
    - **Why?** Because Java has both byte-based and character-based streams, and this class allows for reading byte data and converting it to characters (text data).
3. `FileReader` (extends `InputStreamReader`): `FileReader` specifically deals with **reading files as a stream of characters**. It is essentially a specialized version of `InputStreamReader` tailored for reading text from files.

In summary, `FileReader` is **derived from** `InputStreamReader`, which itself is derived from the abstract `Reader` class.

Class Hierarchy

```
java.lang.Object
  └── java.io.Reader (abstract class)
      └── java.io.InputStreamReader (class)
          └── java.io.FileReader (class)
```

- `Reader`: The abstract class representing the basic operations for reading characters.
- `InputStreamReader`: A bridge between byte streams (`InputStream`) and character streams (`Reader`).
- `FileReader`: A concrete class specifically designed for reading characters from files.

Key Features of `FileReader`:

- **Purpose**: `FileReader` is used to read **character-based** data (text) from files. It is not suitable for reading binary data (like images, audio, or any non-text data).

- **Constructor**: You can create a `FileReader` object by passing either a **file path** (String) or a `File` **object**.
  - Example: `FileReader fr = new FileReader("path/to/file.txt");`
- `read()` **Method**: This is the core method for reading characters from the file. It reads one character at a time and returns the Unicode value of that character. If the end of the file is reached, it returns `-1`.
  - Example: `int data = fr.read();`
- **Efficient for Text Files**: Since it reads files as characters, it handles different encodings (like UTF-8) and works efficiently for text data.

Using `FileReader`

Here's a simple example demonstrating how to use `FileReader` to read a text file:

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

public class FileReaderExample {
    public static void main(String[] args) {
        try {
            // Create a FileReader to read from a text file
            FileReader fr = new FileReader("testfile.txt");

            // Wrap it with BufferedReader to read efficiently
            BufferedReader br = new BufferedReader(fr);

            String line;
            while ((line = br.readLine()) != null) {
                System.out.println(line);  // Print each line from the file
            }

            // Close the BufferedReader (which also closes the FileReader)
            br.close();
        } catch (IOException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

Where Does `FileReader` Fit in Java I/O?

Java's I/O system has two main categories of streams:

1. **Byte Streams**: Used for reading and writing binary data (e.g., `FileInputStream`, `FileOutputStream`).
2. **Character Streams**: Used for reading and writing text data (e.g., `FileReader`, `FileWriter`).

- `FileInputStream` and `FileOutputStream` are used for binary data, while `FileReader` and `FileWriter` are used for character data.

- The `InputStreamReader` class serves as a bridge between these two categories, allowing byte streams to be converted into character streams.

Java I/O Stream Hierarchy - Focus on Character Streams

Here's the hierarchy for **character streams** in Java:

```
java.io.Reader (abstract class)
   ├── java.io.FileReader (class)        ← Reads character data from files
   └── java.io.BufferedReader (class)    ← Wraps Reader for efficient reading
```

The Flow of Character Streams

1. `FileReader`: Reads raw characters from the file, dealing directly with character-based data.
2. `BufferedReader`: Often used alongside `FileReader` to read large files efficiently by buffering data in memory, reducing disk access.

Conclusion

- `FileReader` is a **class** in Java used for reading character-based data from files.
- It is a part of the `java.io` **package**, extending the `InputStreamReader` class and implementing the `Reader` interface.
- It's typically used to read text files efficiently and is often combined with `BufferedReader` to optimize reading performance.
- It is part of the character stream category, which is ideal for text data, as opposed to byte streams, which handle binary data.

I hope this breakdown clarifies the purpose of `FileReader`, its place in Java's I/O system, and how it interacts with other classes.

File Handling in Java: Key Concepts and Differences

1. Creating a File in Java

- In Java, a file is represented by the `File` class (`java.io.File`).
- The method `file.createNewFile()` **creates a new empty file** if it does not exist.
- Example:

```
File file = new File("R:\\File\\HeyThere.txt");
if (file.createNewFile()) {
    System.out.println("File created successfully.");
} else {
    System.out.println("File already exists.");
}
```

- **Checking File Existence:**
  - `file.exists()` → Returns `true` if the file is present on disk.
  - `file.isFile()` → Returns `true` if the path points to a valid file (not a directory).

File Methods in Java: A Quick Overview

Method	Description	Return Type
`createNewFile()`	Creates a new file if it does not already exist. If the file already exists, it returns `false`.	`boolean`
`exists()`	Checks if the file or directory exists.	`boolean`
`isFile()`	Checks if the path points to a file (not a directory).	`boolean`
`isDirectory()`	Checks if the path points to a directory.	`boolean`
`delete()`	Deletes the file or directory. If the file or directory does not exist, returns `false`. If it does exist and is deleted, returns `true`.	`boolean`
`renameTo(File dest)`	Renames or moves the file to the specified destination path.	`boolean`
`length()`	Returns the size of the file in bytes.	`long`
`lastModified()`	Returns the last modified time of the file in milliseconds.	`long`
`setReadable(boolean readable)`	Sets the readability of the file. Can be used to modify file permissions.	`boolean`
`setWritable(boolean writable)`	Sets the writability of the file. Can be used to modify file permissions.	`boolean`
`setExecutable(boolean executable)`	Sets the executability of the file. Can be used to modify file permissions.	`boolean`
`mkdir()`	Creates a **single** directory. Returns `true` if the directory was created successfully. Returns `false` if the directory already exists or if the parent directory doesn't exist.	`boolean`
`mkdirs()`	Creates the **directory and any necessary parent directories**. If the directory already exists, it returns `false`.	`boolean`

Explanation of `mkdir()` vs `mkdirs()`

- `mkdir()`:
  - Creates **only one directory**.
  - If the parent directories do not exist, it will **not create them**, and the method will return `false`.
  - **Example:**

```
File dir = new File("R:\\File\\newDir");
if (dir.mkdir()) {
    System.out.println("Directory created successfully.");
} else {
    System.out.println("Failed to create directory.");
}
```

- `mkdirs()`:
  - Creates **the directory and any necessary parent directories**.
  - If any parent directory in the path does not exist, `mkdirs()` will create them.

- **Example:**

```
File dir = new File("R:\\File\\newDir\\subDir");
if (dir.mkdirs()) {
    System.out.println("Directory created successfully, including parent directories.");
} else {
    System.out.println("Failed to create directories.");
}
```

Additional Useful Methods

Method	Description	Return Type
`getAbsolutePath()`	Returns the **absolute** path of the file (including the full directory path).	`String`
`getName()`	Returns the **name** of the file or directory (without the path).	`String`
`getParent()`	Returns the **parent directory** of the file (if any), or `null` if there is no parent.	`String`
`getCanonicalPath()`	Returns the **canonical path** of the file (resolves symbolic links).	`String`
`list()`	Returns an array of **filenames** in the directory (if the file is a directory).	`String[]`
`listFiles()`	Returns an array of **File objects** representing the files in the directory (if the file is a directory).	`File[]`

When to Use `mkdir()` vs `mkdirs()`?

- Use `mkdir()` when you only need to create a **single directory**, and you are sure the parent directories already exist.
- Use `mkdirs()` when you need to create the **directory along with any missing parent directories**.

Example Code for `mkdir()` and `mkdirs()`

```
import java.io.File;

public class DirectoryCreationExample {
    public static void main(String[] args) {
        // Using mkdir() - creates only one directory
        File singleDir = new File("R:\\File\\newDir");
        if (singleDir.mkdir()) {
            System.out.println("Single directory created successfully.");
        } else {
            System.out.println("Failed to create directory.");
        }
```

```
        // Using mkdirs() - creates directory along with missing parents
        File multipleDirs = new File("R:\\File\\newDir\\subDir");
        if (multipleDirs.mkdirs()) {
            System.out.println("Directory and parent directories created successfully.");
        } else {
            System.out.println("Failed to create directories.");
        }
    }
}
```

Summary of Key Takeaways

1. **File Existence Methods:**
   - `file.exists()` – Checks if the file or directory exists.
   - `file.isFile()` – Checks if the path is a **file**, not a directory.
2. **Creating Directories:**
   - `mkdir()` – Creates a **single directory** (fails if parent directories are missing).
   - `mkdirs()` – Creates the **directory and any missing parent directories**.
3. **Other Useful File Methods**:
   - `length()` – File size in bytes.
   - `lastModified()` – Timestamp of last modification.
   - `delete()` – Deletes the file/directory.
   - `renameTo()` – Renames or moves the file.
   - `setReadable()` , `setWritable()` – Modifies file permissions.

---

2. Reading a File in Java

There are multiple ways to read files, depending on the type of data and efficiency needed.

2.1 Using `BufferedReader` (Character Stream - Text Files)

- `BufferedReader` reads **character-based text** efficiently, supporting **line-by-line reading**.
- **Uses:** Ideal for reading large text files efficiently.
- **Example:**

```
import java.io.*;

public class BufferedReaderExample {
    public static void main(String[] args) {
        try (BufferedReader br = new BufferedReader(new FileReader("R:\\File\\HeyThere.txt"))) {
            String line;
            while ((line = br.readLine()) != null) { // Reads each line
                System.out.println(line);
            }
        } catch (IOException e) {
```

```
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

✔️ **Advantages**:

✅ Efficient for large text files (uses buffering).

✅ Reads entire lines instead of single characters.

2.2 Using `FileInputStream` (Byte Stream - Binary Files)

- `FileInputStream` reads **byte-by-byte** and is ideal for **binary files** (images, videos, etc.).
- **Example:**

```
import java.io.*;

public class InputStreamExample {
    public static void main(String[] args) {
        try (FileInputStream fis = new FileInputStream("R:\\File\\HeyThere.txt")) {
            int data;
            while ((data = fis.read()) != -1) { // Reads byte by byte
                System.out.print((char) data);
            }
        } catch (IOException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

✔️ **Best for:** Binary files (images, videos, audio).

❌ **Not efficient for text files** (no line-by-line reading).

2.3 Alternative: Using `Files.readAllLines()`

- This method reads **all lines at once** into a `List<String>`.
- **Best for:** Small text files where you need quick access.
- **Example:**

```
import java.nio.file.*;
import java.io.IOException;
import java.util.List;

public class ReadFileExample {
    public static void main(String[] args) {
        try {
            List<String> lines = Files.readAllLines(Paths.get("R:\\File\\HeyThere.txt"));
            for (String line : lines) {
                System.out.println(line);
```

```
            }
        } catch (IOException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

✔️ **Advantages:**

✅ **Simple and concise** for small files.

❌ **Not memory efficient** for large files (loads everything at once).

---

3. Key Differences: `BufferedReader` vs. `InputStream`

Feature	**BufferedReader**	**InputStream**
Type	Character Stream	Byte Stream
Reads Data As	Text (Characters, Lines)	Raw Data (Bytes)
Efficiency	Uses internal buffer, faster for text	Reads byte-by-byte, slower for text
Use Case	Text files (`.txt` , `.csv`)	Binary files (`.jpg` , `.mp3`)
Example	`BufferedReader br = new BufferedReader(new FileReader("file.txt"));`	`FileInputStream fis = new FileInputStream("file.jpg");`

When to Use What?

- If working with **text files**, prefer `BufferedReader` (efficient, supports `readLine()`).
- If working with **binary files** (images, videos), use `InputStream` (`FileInputStream`).

Conclusion & Best Practices

✅ Use `BufferedReader` for reading large text files efficiently.

✅ Use `FileInputStream` when handling binary files.

✅ Always check `file.exists()` before performing operations.

✅ Prefer **try-with-resources** (`try (...) {}`) to automatically close streams.

✅ If reading a small file, `Files.readAllLines()` is a simple alternative.

Difference Between `BufferedReader` and `InputStream`

Feature	**BufferedReader**	**InputStream**
Purpose	Reads text (character stream) efficiently	Reads raw binary data (byte stream)
Class Type	Works with **character-based** streams (`Reader`)	Works with **byte-based** streams (`InputStream`)
Common Implementations	`BufferedReader` , `FileReader`	`FileInputStream` , `ByteArrayInputStream`
Reads Data As	**Lines** (via `readLine()`) or **characters** (via `read()`)	**Bytes** (via `read()`)

Efficiency	Uses an internal buffer for faster performance	Reads one byte at a time (less efficient for text files)
Best Used For	Reading text files efficiently	Reading binary files (images, audio, video)
Example Usage	`BufferedReader br = new BufferedReader(new FileReader("file.txt"));`	`FileInputStream fis = new FileInputStream("file.txt");`

Code Examples

Using `BufferedReader` (Character Stream)

Best for reading text files efficiently, line by line.

```
import java.io.*;

public class BufferedReaderExample {
    public static void main(String[] args) {
        try (BufferedReader br = new BufferedReader(new FileReader("file.txt"))) {
            String line;
            while ((line = br.readLine()) != null) { // Reads line by line
                System.out.println(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

✔ **Best for:** Text files

✔ **Reads:** Line by line (`readLine()`), character by character (`read()`)

Using `InputStream` (Byte Stream)

Best for reading binary files or raw data.

```
import java.io.*;

public class InputStreamExample {
    public static void main(String[] args) {
        try (FileInputStream fis = new FileInputStream("file.txt")) {
            int data;
            while ((data = fis.read()) != -1) { // Reads byte by byte
                System.out.print((char) data);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

✔️ **Best for:** Binary files (images, audio, etc.)

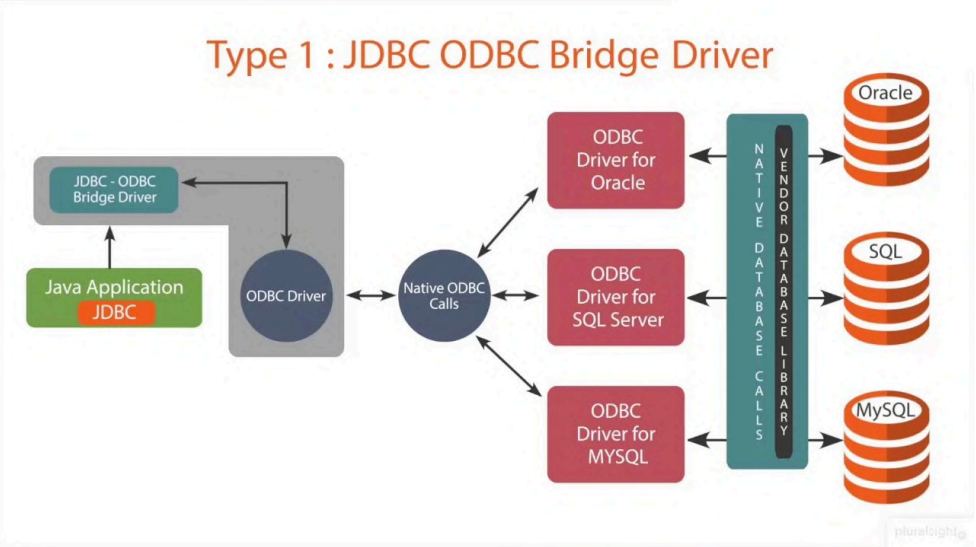
✔️ **Reads:** Byte by byte (`read()`)

Which One Should You Use?

- If **reading text files**, use `BufferedReader` (faster and more convenient).
- If **reading binary files (images, videos, etc.)**, use `InputStream`.

DML, DDL, DCL, TCL, and DQL in SQL:

Category	Full Form	Purpose	Example Commands
DML	Data Manipulation Language	Deals with data manipulation (insert, update, delete)	`INSERT`, `UPDATE`, `DELETE`, `MERGE`
DDL	Data Definition Language	Defines and modifies database structure	`CREATE`, `ALTER`, `DROP`, `TRUNCATE`, `RENAME`
DCL	Data Control Language	Controls user access and permissions	`GRANT`, `REVOKE`
TCL	Transaction Control Language	Manages transactions in the database	`COMMIT`, `ROLLBACK`, `SAVEPOINT`, `SET TRANSACTION`
DQL	Data Query Language	Retrieves data from the database	`SELECT`



JDBC (Java Database Connectivity). Based on your text, it seems like you're facing a ClassNotFoundException or connection issue with the River database management system or a related component.

Here's what you can check:

1. **Check JDBC Driver**

- Ensure that the correct JDBC driver (e.g., `river-jdbc.jar` or any related driver) is added to your classpath.
- If using Maven, include the correct dependency in `pom.xml`.

2. **Verify Database Connection**
   - Check if the **connection URL** is correctly formatted.
   - Example:

```
Connection con = DriverManager.getConnection("jdbc:river://localhost:3306/dbname", "user", "password");
```

3. **Load the Driver Correctly**
   - If required, explicitly load the driver:

```
Class.forName("com.river.jdbc.Driver");
```

4. **Check Spelling of Class Name**
   - The class might be misspelled or unavailable. Verify the exact class name from the official documentation.
5. **Ensure Database is Running**
   - The database service should be active and accessible.
6. **Check for Corrupt JAR Files**
   - If the JAR is corrupted or outdated, download the latest version.

Exceptions in JDBC

1. ClassNotFoundException

🔷 **Cause:** JDBC driver class is missing or not loaded.

🔷 **Fix:**

- Ensure you have the correct **JAR file** in your classpath.
- Use `Class.forName("com.mysql.cj.jdbc.Driver");` for MySQL (or the correct driver for your DB).
- If using **Maven**, add the correct dependency.

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.33</version>
</dependency>
```

2. SQLException

🔷 **Cause:** Issues with database connectivity, incorrect SQL queries, or authentication failures.

🔷 **Fix:**

- Check **DB URL, username, password**.
- Ensure the database **server is running**.
- Verify the **SQL syntax**.

Example:

```
try {
    Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/dbname", "use
r", "pass");
} catch (SQLException e) {
    System.out.println("SQL Error: " + e.getMessage());
}
```

---

3. SQLSyntaxErrorException

🔷 **Cause:** Incorrect SQL syntax.

🔷 **Fix:**

- Verify the **table and column names**.
- Check if any **reserved SQL keywords** are used incorrectly.

Example:

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM users WHERE id=1"); // Ensure correct syntax
```

---

4. SQLIntegrityConstraintViolationException

🔷 **Cause:** Violating constraints like **PRIMARY KEY, UNIQUE, NOT NULL, FOREIGN KEY**.

🔷 **Fix:**

- Ensure no duplicate entries for **PRIMARY KEY**.
- Check foreign key references exist in the parent table.

```
try {
    PreparedStatement ps = con.prepareStatement("INSERT INTO users (id, name) VALUES (?,
?)");
    ps.setInt(1, 1); // Duplicate ID may cause an error
    ps.setString(2, "John");
    ps.executeUpdate();
} catch (SQLIntegrityConstraintViolationException e) {
    System.out.println("Constraint Violation: " + e.getMessage());
}
```

---

5. SQLTimeoutException

🔷 **Cause:** Query execution took too long and timed out.

🔷 **Fix:**

- Optimize your **query and indexing**.
- Set a timeout using:

```
Statement stmt = con.createStatement();
stmt.setQueryTimeout(30); // Timeout in seconds
```

---

6. NullPointerException (NPE)

🔷 **Cause:** Accessing a `null` object, like an uninitialized `Connection`.

🔷 **Fix:**

- Ensure the connection is **not null** before using it.

```
if (con != null) {
    con.close();
} else {
    System.out.println("Connection is null!");
}
```

---

7. DriverNotFoundException

🔷 **Cause:** The JDBC driver is missing in the classpath.

🔷 **Fix:**

- Ensure the correct driver **JAR file** is available.
- For MySQL: `mysql-connector-java-8.0.33.jar`
- For PostgreSQL: `postgresql-42.3.1.jar`
- Use:

```
Class.forName("org.postgresql.Driver");
```