Java IOStreams

In Java, **Streams** are used to perform input and output operations. They act as a channel or medium to carry data between Java applications and I/O devices (like files, console, network connections) in a continuous flow. All stream classes are defined within the `java.io` package.

Java provides two main types of streams:

1. **Byte Oriented Streams:** Handle data in the form of bytes (binary data). Each data item is typically 1 byte in size. Classes usually end with "Stream".
2. **Character Oriented Streams:** Handle data in the form of characters (textual data, using encoding like ASCII or Unicode). Each data item is typically 2 bytes in size (for Unicode characters). Classes usually end with "Reader" or "Writer".

Here is a comparison of the two stream types:

Feature	Byte Oriented Streams	Character Oriented Streams
Data Type	Bytes (Binary Data)	Characters (Textual Data)
Unit Size	1 byte	2 bytes (Unicode character)
Base Classes	`InputStream`, `OutputStream`	`Reader`, `Writer`
Naming Conv.	Class names end with "Stream"	Class names end with "Reader" or "Writer"
Use Case	Handling raw binary data (images, audio, etc.)	Handling text data

Export to Sheets

Byte Oriented Streams

- **InputStream:** Used to read data from an input source into a Java application.
  - *Examples:* `ByteArrayInputStream`, `FileInputStream`, `BufferedInputStream`, `DataInputStream`, `ObjectInputStream`.
- **OutputStream:** Used to write data from a Java application to an output destination.
  - *Examples:* `ByteArrayOutputStream`, `FileOutputStream`, `BufferedOutputStream`, `DataOutputStream`, `ObjectOutputStream`.

Character Oriented Streams

- **Reader:** Used to read character data from an input source into a Java application.
  - *Examples:* `CharArrayReader`, `FileReader`, `BufferedReader`, `InputStreamReader`.
- **Writer:** Used to write character data from a Java application to an output destination.

    - *Examples:* `CharArrayWriter`, `FileWriter`, `BufferedWriter`, `PrintWriter`.

File Operations

Java provides specific streams for performing operations on files: `FileOutputStream`, `FileInputStream`, `FileWriter`, and `FileReader`.

FileOutputStream

Used to write byte data from a Java application to a file.

Steps:

1. **Create `FileOutputStream` object:**
    - `public FileOutputStream(String fileName)`: Creates a stream that replaces the file content if it exists.
    - `public FileOutputStream(String fileName, boolean append)`: Creates a stream. If `append` is `true`, data is added to the end; if `false`, file content is replaced.
    - *Reasoning:* The constructor checks if the file exists. If not, it creates a new file.
    - *Example:* `FileOutputStream fos = new FileOutputStream("path/to/file.txt", true);`
2. **Prepare data:** Convert the data (e.g., a String) into a byte array.
    - *Example:* `String data = "Hello"; byte[] bytes = data.getBytes();`
3. **Write data:** Use the `write(byte[] bytes)` method.
    - *Example:* `fos.write(bytes);`
4. **Close the stream:** Release system resources. It's crucial to close streams. Using try-with-resources is the recommended approach for automatic closing.
    - *Example:* `fos.close();`

Code Examples (FileOutputStream):

Java

```
import java.io.FileOutputStream;

public class Main {
   public static void main(String[] args) throws Exception{ // Declaring
throws Exception is simple but less robust
         FileOutputStream fos = new
FileOutputStream("/Users/nagoorn/Documents/docs/hello.txt", true); //
Append mode
         String data = "\nWelcome to Java Programming!";
         byte[] bytes = data.getBytes(); // Convert String to bytes
         fos.write(bytes); // Write bytes to file
         System.out.printf("Data Sent to
/Users/nagoorn/Documents/docs/hello.txt file");
         fos.close(); // Manually close stream
   }
}
```

Java

```
import java.io.FileOutputStream;
```

```
public class Main {
   public static void main(String[] args) {
         FileOutputStream fos =null;
         try{ // Handling exceptions with try-catch-finally
            fos = new
FileOutputStream("/Users/nagoorn/Documents/docs/hello.txt", false); //
Replace mode
            String data ="Welcome to Java Programming!";
            byte[] bytes = data.getBytes();
            fos.write(bytes);
            System.out.printf("Data Sent to
/Users/nagoorn/Documents/docs/hello.txt file");
         } catch (Exception e) {
            e.printStackTrace();
         }finally{ // Ensure stream is closed even if exception occurs
            try {
               if(fos != null) fos.close(); // Check before closing
            }catch (Exception e){
               e.printStackTrace();
            }
         }
   }
}
```

Java

```
import java.io.FileOutputStream;

public class Main {
   public static void main(String[] args) {
         try( // Recommended: try-with-resources for automatic resource
management
               FileOutputStream fos = new
FileOutputStream("/Users/nagoorn/Documents/docs/hello.txt", false);
               ){
            String data ="Welcome to Java Programming!";
            byte[] bytes = data.getBytes();
            fos.write(bytes);
            System.out.printf("Data Sent to
/Users/nagoorn/Documents/docs/hello.txt file");
         } catch (Exception e) {
            e.printStackTrace();
         } // Stream is automatically closed here
   }
}
```

FileInputStream

Used to read byte data from a file into a Java application.

Steps:

1. **Create `FileInputStream` object:**
   - `public FileInputStream(String fileName) throws IOException`: Creates a stream from the specified file.
   - *Reasoning:* The constructor checks if the file exists. If not, it throws `FileNotFoundException`.
   - *Example:* `FileInputStream fis = new FileInputStream("path/to/file.txt");`

2. **Find data size:** Use `public int available()` to get the estimated number of bytes that can be read without blocking.
   - *Example:* `int size = fis.available();`
3. **Create byte array:** Create a byte array to hold the data read from the stream.
   - *Example:* `byte[] bytes = new byte[size];`
4. **Read data:** Use `public void read(byte[] bytes)` to read data into the byte array.
   - *Example:* `fis.read(bytes);`
5. **Convert to String (if text):** Convert the byte array back to a String.
   - *Example:* `String data = new String(bytes);` `System.out.println(data);`
6. **Close the stream:** Release resources (handled automatically by try-with-resources).
   - *Example:* `fis.close();`

Code Examples (FileInputStream):

Java

```
import java.io.FileInputStream;

public class Main {
   public static void main(String[] args) throws Exception{ // Simple
exception declaration
         FileInputStream fis = new
FileInputStream("/Users/nagoorn/Documents/docs/hello.txt");
         int size = fis.available(); // Get file size in bytes
         byte[] bytes = new byte[size]; // Create byte array of file size
         fis.read(bytes); // Read data into byte array
         String data = new String(bytes); // Convert bytes to String
         System.out.println(data);
         fis.close(); // Manually close
   }
}
```

Java

```
import java.io.FileInputStream;

public class Main {
   public static void main(String[] args){
         FileInputStream fis =null;
         try { // try-catch-finally for resource management
            fis = new
FileInputStream("/Users/nagoorn/Documents/docs/hello.txt");
            int size = fis.available();
            byte[] bytes = new byte[size];
            fis.read(bytes);
            String data =new String(bytes);
            System.out.println(data);
         }catch (Exception e){
            e.printStackTrace();
         }finally{ // Ensure stream is closed
            try {
               if(fis != null) fis.close();
            }catch (Exception e){
               e.printStackTrace();
            }
         }
   }
}
```

Java

```
import java.io.FileInputStream;

public class Main {
   public static void main(String[] args){
         try ( // try-with-resources
               FileInputStream fis = new
FileInputStream("/Users/nagoorn/Documents/docs/hello.txt");
               ){
            int size = fis.available();
            byte[] bytes = new byte[size];
            fis.read(bytes);
            String data =new String(bytes);
            System.out.println(data);
         }catch (Exception e){
            e.printStackTrace();
         } // Stream automatically closed
   }
}
```

Q) Write a Java program to display the content of a particular file by taking file name and location as command line argument?

Java

```
import java.io.FileInputStream;

public class Main {
   public static void main(String[] args){
         String fileName = args[0]; // Get filename from command line
arguments
         try(
               FileInputStream fis = new FileInputStream(fileName); // Use
provided filename
               ){
            int size = fis.available();
            byte[] bytes = new byte[size];
            fis.read(bytes);
            String data =new String(bytes);
            System.out.println(data);
         } catch (Exception e) {
            e.printStackTrace();
         }
   }
}
```

Command Line Execution Example:

```
D:\apps>javac Main.java
D:\apps>java Main D:\apps\Main.java
import java.io.*;
import java.sql.*;

public class Main {
   public static void main(String[] args) {
       try(
               Connection connection =
                   DriverManager.getConnection(
                       "jdbc:mysql://localhost:3306/durgadb",
                        "root",
                        "root@123"
                   );

               PreparedStatement
preparedStatement = connection.prepareStatement(
                       "insert into
WEB_APPS values(?,?)"
                   );

               ){
           preparedStatement.setString(1,
"APP01");
           File file = new
File("/Users/nagoorn/Documents/apps/FULLSTACKJAVA6PM/ADVJAVA-730/SERVLETS/
INTELLIJ-APPS/app07/src/main/webapp/WEB-INF/web.xml");
           FileReader fileReader = new
FileReader(file);
           preparedStatement.setCharacterStream(2, fileReader,
file.length());
           int rowCount =
preparedStatement.executeUpdate();
           if(rowCount == 1){
               System.out.printf("WEB
Application Inserted Successfully");
           }else{
               System.out.printf("WEB
Application Insertion Failed");
           }
       } catch (Exception e) {
           e.printStackTrace();
       }
   }
}
```

Reasoning: This code demonstrates reading the content of the `Main.java` file itself by passing its path as a command-line argument (`args[0]`).

Q) Write a java program to count the number of words over the data in a particular file and count the word "Durga" repetitions?

File Content (`Welcome.txt`):

```
Durga Software Solutions is an Organisation for IT training, Durga Software
Solutions has the branches throughout the globe, Durga Software Solutions
has a development centre at Madhapur, Durga Software Solutions is
specialised in JAVA / J2EE / PYTHON / AWS Courses Training.
```

Code:

Java

```
import java.io.FileInputStream;
```

```
public class Main {
   public static void main(String[] args){
         String fileName = "/Users/nagoorn/Documents/docs/welcome.txt"; //
File path
         try(
               FileInputStream fis = new FileInputStream(fileName);
               ){
            int size = fis.available();
            byte[] bytes = new byte[size];
            fis.read(bytes);
            String data =new String(bytes);
            String[] words= data.split(" "); // Split the content by spaces
to get words
            int wordsCount =words.length; // Count total words
            System.out.println("No Of Words  : "+wordsCount);
            int durgaCount =0;
            for (String word :words) { // Iterate through words
               if(word.equalsIgnoreCase("durga")){ // Check for "durga"
ignoring case
                  durgaCount =durgaCount + 1;
               }
            }
            System.out.println("Durga Count : "+durgaCount);
         } catch (Exception e) {
            e.printStackTrace();
         }
   }
}
```

Output:

```
No Of Words  : 42
Durga Count : 4
```

Reasoning: This program reads the file content as a single string, splits it into words using space as a delimiter, and then counts the total words and occurrences of "Durga" (case-insensitive).

Q) Write a Java program to copy the image data from one file to another file?

Java

```
import java.io.FileInputStream;
import java.io.FileOutputStream;

public class Main {
   public static void main(String[] args){
         String sourceFile = "/Users/nagoorn/Documents/Images/prabhas.jpg";
// Source image path
         String targetFile = "/Users/nagoorn/Documents/docs/prabhas.jpg";
// Destination image path
         try( // Using try-with-resources for both streams
               FileInputStream fis = new FileInputStream(sourceFile);
               FileOutputStream fos = new FileOutputStream(targetFile);
               ){
```

```
            byte[] bytes = new byte[fis.available()]; // Read the whole
file into a byte array (suitable for smaller files)
            fis.read(bytes); // Read from source
            fos.write(bytes); // Write to target
            System.out.println("Image Copied from " + sourceFile + " to " +
targetFile);
         } catch (Exception e) {
            e.printStackTrace();
         }
   }
}
```

Reasoning: Images are binary data, so Byte Oriented Streams (`FileInputStream` and `FileOutputStream`) are appropriate. Reading the entire file into memory is simple but might be inefficient for large files; reading in chunks would be more scalable.

FileWriter

Used to write character data from a Java application to a file.

Steps:

1. **Create `FileWriter` object:**
   - `FileWriter fw = new FileWriter("path/to/file.txt", true);` (true for append, false for replace).
   - *Reasoning:* Similar to `FileOutputStream`, it creates the file if it doesn't exist.
2. **Prepare data:** Convert data (String) to character array or use String directly.
   - *Example:* `String data = "Hello"; char[] chars = data.toCharArray();`
3. **Write data:** Use methods like `write(char[] chars)` or `write(String str).`
   - *Example:* `fw.write(chars);` or `fw.write(data);`
4. **Close the stream:** Use `fw.close()` or try-with-resources.

Code Example (FileWriter):

Java

```
import java.io.FileWriter;

public class Main {
   public static void main(String[] args){
         try( // try-with-resources
               FileWriter fw =new
FileWriter("/Users/nagoorn/Documents/docs/welcome.txt", true); // Append
mode
               ){
            String data =", It is very simple to learn"; // String data
            // char[] chars = data.toCharArray(); // Can write char array
or String directly
            fw.write(data); // Write String directly
            System.out.println("Data Send to
/Users/nagoorn/Documents/docs/welcome.txt");
         } catch (Exception e) {
            e.printStackTrace();
         }
    }
}
```

FileReader

Used to read character data from a file into a Java application.

Steps:

1. **Create `FileReader` object:**
   - `FileReader fr = new FileReader("path/to/file.txt");`
   - *Reasoning:* Throws `FileNotFoundException` if the file doesn't exist.
2. **Read data:** Read character by character using `read()` which returns the character's ASCII value (or -1 at end of file), or read into a character array.
   - *Example (character by character):*

     Java

     ```
     int val = fr.read();
     String data = "";
     while(val != -1){
          data = data + (char)val; // Convert ASCII value to char
     and append
          val = fr.read();
     }
     System.out.println(data);
     ```
3. **Close the stream:** Use `fr.close()` or try-with-resources.

Code Example (FileReader):

Java

```
import java.io.FileReader;

public class Main {
   public static void main(String[] args){
         try( // try-with-resources
               FileReader fr =new
FileReader("/Users/nagoorn/Documents/docs/welcome.txt");
               ){
            int val = fr.read(); // Read first character (as int ASCII
value)
            String data ="";
            while(val != -1){ // Loop until end of file (-1)
               data = data+ (char)val; // Append character to string
               val = fr.read(); // Read next character
            }
            System.out.println(data);
         } catch (Exception e) {
            e.printStackTrace();
         }
   }
}
```

Dynamic Input Approaches

Providing input to a Java application during its execution is called Dynamic Input. Common approaches include `BufferedReader`, `Scanner`, and `Console`.

BufferedReader

A common class for reading text from an input stream, buffering characters to provide efficient reading of characters, arrays, and lines.

- **Setup:** `BufferedReader br = new BufferedReader(new InputStreamReader(System.in));`
    - *Reasoning:*
        - `System.in`: A predefined `InputStream` object connected to the standard input (usually keyboard/console). It provides raw byte input.
        - `InputStreamReader`: Acts as a bridge, converting byte streams from `System.in` into character streams using the default character encoding. It reads individual characters inefficiently.
        - `BufferedReader`: Wraps the `InputStreamReader` and buffers the input. This allows reading larger chunks (like entire lines) efficiently, improving performance compared to reading character by character directly from `InputStreamReader`.
- **Reading Methods:**
    - `public String readLine()`: Reads a complete line of text.
    - `public int read()`: Reads a single character, returned as an integer ASCII value.

Q) What is the difference between read() method and readLine() method in BufferedReader?

Ans:

- `read()`: Reads a single character at a time, returning its integer ASCII value. Returns -1 at the end of the stream.
- `readLine()`: Reads an entire line of text from the input stream, returning it as a `String`. Returns `null` at the end of the stream.

Code Example (read() vs readLine()):

Java

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

public class Main {
   public static void main(String[] args) {
         try( // try-with-resources
               BufferedReader br = new BufferedReader(new
InputStreamReader(System.in)) // Setup BufferedReader for console input
               ){
            System.out.print("Enter Data    : ");
            String data =br.readLine(); // Read a full line
            System.out.print("Enter the same data again     : ");
            int val = br.read(); // Read only the first character

            System.out.println("readLine()  : " +data);
            System.out.println("read()      : " +val+"["+(char)val+"]"); //
Display ASCII value and corresponding character
         } catch (Exception e) {
            e.printStackTrace();
         }
   }
}
```

Output Example:

```
Enter Data    : Durga Software Solutions
Enter the same data again     : Durga Software Solutions
readLine()  : Durga Software Solutions
read()      : 68[D]
```

Reasoning: This output clearly shows that `readLine()` captured the entire input line, while `read()` only captured the first character ('D', which has ASCII value 68). Note that after `readLine()` reads the first line, the newline character remains in the buffer, which is then consumed by the subsequent `read()`.

Handling Primitive Data Input with BufferedReader:

`BufferedReader` primarily reads Strings or characters. To get primitive types (like `int`, `double`), you need a two-step process:

1. Read the input as a `String` using `readLine()`.
2. Convert the `String` to the desired primitive type using the respective wrapper class's static `parseXXX()` method (e.g., `Integer.parseInt()`, `Double.parseDouble()`).

Code Example (Reading and Parsing Primitives):

Java

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

public class Main {
   public static void main(String[] args) {
         try( // try-with-resources
               BufferedReader br = new BufferedReader(new
InputStreamReader(System.in))
               ){
            System.out.print("First Value   : ");
            String firstValue = br.readLine(); // Read as String
            System.out.print("Second Value  : ");
            String secondValue = br.readLine(); // Read as String

            System.out.println("CONCAT  : " + (firstValue + secondValue));
// String concatenation

            int fval = Integer.parseInt(firstValue); // Parse String to int
            int sval = Integer.parseInt(secondValue); // Parse String to
int

            System.out.println("ADD     : " + (fval + sval)); // Arithmetic
operations
            System.out.println("SUB     : " + (fval - sval));
            System.out.println("MUL     : " + (fval * sval));
            System.out.println("DIV     : " + (fval / sval));

         } catch (Exception e) {
            e.printStackTrace();
         }
   }
}
```

Output Example:

```
First Value   : 10
Second Value  : 5
CONCAT  : 105
ADD     : 15
SUB     : 5
MUL     : 50
DIV     : 2
```

Reasoning: This demonstrates that reading numerical input via `BufferedReader.readLine()` initially treats it as text (String). To perform mathematical operations, the Strings must be explicitly parsed into numerical types using methods like `Integer.parseInt()`.

Java Input/Output: Scanner, Console, Serialization & File System

This document covers dynamic input methods (`Scanner`, `Console`) and persistent data handling (`Serialization`, `Externalization`, `File` System).

Dynamic Input Methods

Dynamic input allows providing data to Java applications at runtime. The main approaches discussed are `BufferedReader`, `Scanner`, and `Console`. While `BufferedReader` was covered previously, this section focuses on `Scanner` and `Console`.

Scanner

The `Scanner` class (`java.util.Scanner`) provides a simple way to read primitive data types and strings directly from various input sources, including the console (`System.in`). It is generally more convenient than `BufferedReader` for reading formatted input and primitive types.

Steps to use Scanner for Dynamic Input:

1. **Create `Scanner` object:**
   - `Scanner s = new Scanner(System.in);`
   - *Reasoning:* This connects the Scanner to the standard input stream (`System.in`).
2. **Read Dynamic Input:** Use appropriate methods based on the data type.
   - **Primitive Data:** Use `public xxx nextXxx()` methods (e.g., `nextInt()`, `nextFloat()`, `nextByte()`, etc.).
   - **String Data:** Use `public String nextLine()` or `public String next()`.

Q) What is the difference between `nextLine()` method and `next()` method in Scanner?

Ans:

- **`nextLine()`:** Reads the entire line of input, including spaces, until the newline character (`\n`) is encountered. The newline character is consumed.
- **`next()`:** Reads only the next word or token from the input, using whitespace as the delimiter. It stops reading at the first whitespace character and does *not* consume the rest of the line (including the newline).

Code Example (nextLine() vs next()):

Java

```java
import java.util.Scanner;

public class Main {
   public static void main(String[] args) {
         try( // try-with-resources for Scanner (since Java 7)
               Scanner scanner = new Scanner(System.in)
               ){
            System.out.print("Enter the Data  : ");
            String data1 =scanner.nextLine(); // Reads the whole line
            System.out.print("Enter the same Data Again  : ");
            String data2 =scanner.next(); // Reads only the first word

            System.out.println("nextLine()   : " +data1);
            System.out.println("next()       : " +data2);

         } catch (Exception e) {
            e.printStackTrace();
         }
   }
}
```

Output Example:

```
Enter the Data  : Durga Software Solutions
Enter the same Data Again  : Durga Software Solutions
nextLine()   : Durga Software Solutions
next()       : Durga
```

Reasoning: The output shows that `nextLine()` captured the full phrase "Durga Software Solutions", while `next()` only read "Durga" because it stopped at the first space. The remaining input " Software Solutions" and the newline are left in the buffer after the first `next()`. This can sometimes lead to unexpected behavior if `next()` is followed by `nextLine()` without consuming the leftover newline.

Code Example (Reading Primitive Data with Scanner):

Java

```
import java.util.Scanner;

public class Main {
   public static void main(String[] args) {
         try( // try-with-resources
               Scanner scanner = new Scanner(System.in)
               ){

            System.out.print("Employee Number    : ");
            int eno = scanner.nextInt(); // Reads an integer directly
            System.out.print("Employee Name      : ");
            String ename =scanner.next(); // Reads the next word (e.g.,
first name)
            System.out.print("Employee Salary  : ");
            float esal = scanner.nextFloat(); // Reads a float directly
            System.out.print("Employee Address : ");
            String eaddr =scanner.next(); // Reads the next word (e.g.,
city name)

            System.out.println("Employee Details");
            System.out.println("------------------------------");
            System.out.println("Employee Number    : "+eno);
            System.out.println("Employee Name      : "+ename);
            System.out.println("Employee Salary    : "+esal);
            System.out.println("Employee Address   : "+eaddr);

         } catch (Exception e) {
            e.printStackTrace();
         }
   }
}
```

Output Example:

```
Employee Number    : 111
Employee Name      : Durga
Employee Salary    : 50000
Employee Address   : Hyd

Employee Details
------------------------------
Employee Number    : 111
Employee Name      : Durga
Employee Salary    : 50000.0
Employee Address   : Hyd
```

Reasoning: `Scanner` directly provides methods (`nextInt`, `nextFloat`, etc.) to read input as specific primitive types, avoiding the need for manual String parsing required with `BufferedReader`.

Console

The `java.io.Console` class is primarily used for secure text input from the console, especially for passwords or sensitive information, and provides convenience for prompting the user. It is not always available, particularly in some IDE environments.

Problems with BufferedReader/Scanner addressed by Console:

1. Requires separate print/println statements for prompts before reading input.
   - `System.out.print("Prompt: ");`
   - `scanner.nextLine();`
2. Does not provide security for sensitive data like passwords, as input is displayed on the console.

Console Advantages:

- Allows displaying a prompt message and reading input in a single method call (`readLine`, `readPassword`).
- `readPassword()` method suppresses the input characters from being displayed on the console, offering security for passwords.

Steps to use Console:

1. **Get Console object:**
   - `Console c = System.console();`
   - *Reasoning:* Returns the unique `Console` object for the Java virtual machine. Returns `null` if a console is not available.
2. **Read Dynamic Input:**
   - **Read String with Prompt:** `public String readLine(String message, Object... args)`
     - Displays the formatted message and reads a line of text.
   - **Read Password with Prompt:** `public char[] readPassword(String message, Object... args)`
     - Displays the formatted message and reads a line of text, suppressing echoes. Returns a character array, which is more secure than storing passwords in a `String`.

Code Example (Console):

Java

```
import java.io.Console;

class Test {
    public static void main(String[] args) throws Exception {
       Console console = System.console(); // Get the Console instance

       if (console == null) { // Check if console is available (e.g., not
running in some IDEs)
           System.err.println("Console not available.");
           System.exit(1);
       }

       String uname = console.readLine("User Name    : "); // Prompt and
read username
```

```
        char[] pwd = console.readPassword("Password        : "); // Prompt
and read password securely
        String upwd = new String(pwd); // Convert char[] to String for
comparison (handle char[] securely if not needed as String)

        // Basic authentication check
        if (uname.equals("durga") && upwd.equals("durga")) {
            System.out.println("User Login Successful");
        } else {
            System.out.println("User Login Failed");
        }
        // Clear the password character array after use for security
        java.util.Arrays.fill(pwd, ' ');
    }
}
```

Command Line Execution Example:

```
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % javac Test.java
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % java Test
User Name   : durga
Password        :
User Login Successful
nagoorn@Nagoors-MacBook-Pro COREJAVA-830 % java Test
User Name   : durga
Password        :
User Login Failed
```

Reasoning: When running from a proper console (like a terminal), the input for "Password" is not displayed as you type, providing basic security. The output shows the login attempt results. *Note:* `Console` is not supported by all IDEs. It is designed for command-line environments.

Serialization and Deserialization

In distributed applications or when needing to save object states, we often need to transfer objects.

- **Serialization:** The process of converting an object's state into a format that can be stored (e.g., in a file) or transmitted (e.g., over a network). It involves separating data from the object and converting it to a stream of bytes.
- **Deserialization:** The reverse process of reconstructing an object from its serialized byte stream.
- **Marshalling:** Converting data from a system's internal representation to a format suitable for network transmission. (Often used synonymously with Serialization in the context of network communication).
- **Unmarshalling:** Converting data from network representation back to the system's internal representation. (Often used synonymously with Deserialization).

Java provides `java.io.ObjectOutputStream` for Serialization and `java.io.ObjectInputStream` for Deserialization. These are byte-oriented streams.

Performing Serialization

Steps:

1. **Create a Serializable Object:** The class of the object to be serialized *must* implement the **`java.io.Serializable`** marker interface. If a class does not implement `Serializable`, attempting to serialize its objects will result in a `java.io.NotSerializableException`.
   - *Example:* `public class Employee implements Serializable { ... }`
2. **Create `FileOutputStream`:** To specify the target file where the serialized data will be stored.
   - *Example:* `FileOutputStream fos = new FileOutputStream("path/to/emp.txt");`
3. **Create `ObjectOutputStream`:** Wrap the `FileOutputStream` with an `ObjectOutputStream`.
   - *Example:* `ObjectOutputStream oos = new ObjectOutputStream(fos);`
4. **Write the Object:** Use the `public void writeObject(Object obj) throws IOException` method.
   - *Example:* `oos.writeObject(employeeObject);`
   - *Reasoning:* `ObjectOutputStream` handles the process of taking the object, serializing its state, and writing the resulting bytes to the underlying `FileOutputStream`.

Performing Deserialization

Steps:

1. **Create `FileInputStream`:** To read the serialized data from the source file.
   - *Example:* `FileInputStream fis = new FileInputStream("path/to/emp.txt");`
2. **Create `ObjectInputStream`:** Wrap the `FileInputStream` with an `ObjectInputStream`.
   - *Example:* `ObjectInputStream ois = new ObjectInputStream(fis);`
   - *Reasoning:* `ObjectInputStream` reads the byte stream from the `FileInputStream`, performs the deserialization process, and reconstructs the object in memory.
3. **Read the Object:** Use the `public Object readObject() throws IOException, ClassNotFoundException` method. Cast the returned `Object` to the correct class type.
   - *Example:* `Employee emp2 = (Employee) ois.readObject();`

Code Example (Serialization and Deserialization):

(Requires an `Employee` class that implements `Serializable` with appropriate fields and a `getEmployeeDetails` method. Assuming `com.durgasoft.beans.Employee` exists and is Serializable)

Java

```
import com.durgasoft.beans.Employee; // Assuming this class exists and is
Serializable

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable; // Import Serializable interface

// Example Employee class structure (must be in com.durgasoft.beans package
for the main class)
/*
package com.durgasoft.beans;
import java.io.Serializable;
public class Employee implements Serializable {
    private static final long serialVersionUID = 1L; // Recommended for
version control
    int eno;
    String ename;
    float esal;
    String eaddr;

    public Employee(int eno, String ename, float esal, String eaddr) {
        this.eno = eno;
        this.ename = ename;
        this.esal = esal;
        this.eaddr = eaddr;
    }

    public void getEmployeeDetails() {
        System.out.println("Employee Details");
        System.out.println("------------------------");
        System.out.println("Employee Number    : " + eno);
        System.out.println("Employee Name      : " + ename);
        System.out.println("Employee Salary    : " + esal);
        System.out.println("Employee Address   : " + eaddr);
    }
}
*/

public class Main {
   public static void main(String[] args) throws Exception {
      // Serialization
      FileOutputStream fileOutputStream = new
FileOutputStream("/Users/nagoorn/Documents/docs/emp.txt");
      ObjectOutputStream objectOutputStream = new
ObjectOutputStream(fileOutputStream);
      Employee employee = new Employee(111, "Durga", 50000, "Hyd");

      System.out.println("Employee Details Before Serialization");
      employee.getEmployeeDetails();
      System.out.println("Performing Serialization..."); // Added for
clarity
      objectOutputStream.writeObject(employee);
      objectOutputStream.close(); // Close the stream
      System.out.println("Serialization Successful."); // Added for clarity

      System.out.println(); // Add a newline for separation

      // Deserialization
      FileInputStream fileInputStream = new
FileInputStream("/Users/nagoorn/Documents/docs/emp.txt");
      ObjectInputStream objectInputStream = new
ObjectInputStream(fileInputStream);
      System.out.println("Performing Deserialization..."); // Added for
clarity
```

```
      Employee employee2 = (Employee) objectInputStream.readObject(); //
Read and cast the object
      objectInputStream.close(); // Close the stream

      System.out.println("Employee Details After Deserialization");
      employee2.getEmployeeDetails();
   }
}
```

Output Example:

```
Employee Details Before Serialization
Employee Details
------------------------
Employee Number    : 111
Employee Name      : Durga
Employee Salary    : 50000.0
Employee Address   : Hyd
Performing Serialization...
Serialization Successful.

Performing Deserialization...
Employee Details After Deserialization
Employee Details
------------------------
Employee Number    : 111
Employee Name      : Durga
Employee Salary    : 50000.0
Employee Address   : Hyd
```

Reasoning: The output shows the original object's details before serialization and the details of the newly reconstructed object after deserialization, confirming the process worked.

Serialization Considerations:

- **`NotSerializableException`:** Occurs if you try to serialize an object whose class does not implement `Serializable`.

  Java

  ```
  import java.io.FileOutputStream;
  import java.io.ObjectOutputStream;
  // class A does NOT implement Serializable
  class A{
  }
  public class Main {
     public static void main(String[] args) throws Exception {
        FileOutputStream fos = new FileOutputStream("a.txt");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        A a = new A();
        oos.writeObject(a); // This line will cause the exception
        oos.close();
        fos.close();
        System.out.println("Serialization Successful"); // This line
  will not be reached
     }
  }
  ```

Output:

```
java.io.NotSerializableException: A
    ... (stack trace)
```

Reasoning: The `Serializable` interface is a marker interface, indicating to the JVM that the class is safe for serialization. Without it, the JVM doesn't know how to convert the object into a byte stream.

- **Private Variables:** Private variables are included in Serialization/Deserialization. The mechanism accesses the state, not just public fields.

  Java

```java
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

class Student implements Serializable {
  int sno = 111;
  String sname = "Durga";
  private String semail = "durga@dss.com"; // Private variable
  // No getter for semail, but it will still be serialized
}
public class Main {
   public static void main(String[] args) throws Exception {
      FileOutputStream fos = new FileOutputStream("std.txt");
      ObjectOutputStream oos = new ObjectOutputStream(fos);
      Student s = new Student();
      oos.writeObject(s);
      oos.close();
      fos.close();
      System.out.println("Serialization Successful");
      // To verify, you'd need deserialization code and access the
private field via reflection or make it accessible.
   }
}
```

  Reasoning: Serialization saves the *state* of an object, which includes the values of all its non-static, non-transient fields, regardless of their access modifiers (`public`, `private`, `protected`, default).

- **Static Variables:** Static variables are **not** serialized. They belong to the class, not the object instance, and their data is stored in the method area, separate from object state on the heap.
- **Transient Variables:** Use the `transient` keyword to prevent specific instance variables from being serialized. This is crucial for sensitive data (like passwords) or resources that cannot be serialized. During deserialization, transient fields are initialized to their default values (0 for numeric, `false` for boolean, `null` for objects).

  Java

```java
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
// import java.io.FileInputStream; // Needed for deserialization test
// import java.io.ObjectInputStream; // Needed for deserialization
test

class User implements Serializable {
  String uname = "durga";
  transient String upwd = "durga123"; // This field will NOT be
serialized
  String uaddr = "Hyderabad";
  int uage = 22;
  // Static variables are also not serialized
  private static int MIN_AGE = 18;
  private static int MAX_AGE = 25;

  // Add a method to show values (including transient) for
verification
  public void showDetails() {
      System.out.println("Name: " + uname);
      System.out.println("Password: " + upwd); // Will be null after
deserialization
      System.out.println("Address: " + uaddr);
      System.out.println("Age: " + uage);
      System.out.println("Min Age (Static): " + MIN_AGE); // Static
retains value
  }
}

public class Main {
   public static void main(String[] args) throws Exception {
      // Serialization
      FileOutputStream fos = new FileOutputStream("user.ser"); //
Using .ser extension is common
      ObjectOutputStream oos = new ObjectOutputStream(fos);
      User user = new User();
      oos.writeObject(user);
      oos.close();
      fos.close();
      System.out.println("Serialization Successful (password is
transient).");

      // Deserialization (Add this part to verify transient field
behavior)
      /*
      FileInputStream fis = new FileInputStream("user.ser");
      ObjectInputStream ois = new ObjectInputStream(fis);
      User deserializedUser = (User) ois.readObject();
      ois.close();
      fis.close();
      System.out.println("\nAfter Deserialization:");
      deserializedUser.showDetails(); // Password will be null
      */
   }
}
```

Reasoning: The `transient` keyword marks a variable as not part of the object's persistent state, thus excluding it from the default serialization process.

- **Inheritance:** If a superclass is `Serializable`, its subclasses are automatically `Serializable`. If a superclass is *not* `Serializable`, its state will *not* be saved/restored by the subclass's serialization process unless the superclass has a no-argument constructor accessible to the subclass and the subclass handles its superclass's state explicitly (which is complex).

  Java

  ```
  import java.io.FileOutputStream;
  import java.io.ObjectOutputStream;
  import java.io.Serializable;

  class User implements Serializable { // Superclass is Serializable
      int uid = 1; // This will be serialized
  }
  class Employee extends User { // Subclass is also Serializable
      int eno = 111;
      String ename = "Durga";
  }

  public class Main {
     public static void main(String[] args) throws Exception {
        FileOutputStream fos = new FileOutputStream("emp.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        Employee e = new Employee(); // Object of subclass
        oos.writeObject(e); // Serializing subclass object
        oos.close();
        fos.close();
        System.out.println("Serialization Successful for Employee
  (inherits Serializable).");
        // Deserialization would restore both uid, eno, and ename.
     }
  }
  ```

  Reasoning: Java's default serialization mechanism traverses the object's class hierarchy. If the base class implements `Serializable`, its serializable fields are included.

- **Association (Object Graph):** When a serializable object contains references to other objects (composition or aggregation), Serialization recursively serializes the entire "object graph" of reachable, serializable objects. If any object in the graph is *not* serializable and is not marked `transient`, serialization will fail with `NotSerializableException`.

  Java

  ```
  import java.io.FileOutputStream;
  import java.io.ObjectOutputStream;
  import java.io.Serializable;

  class Bank implements Serializable { // Serializable
     int bankId = 111;
  }
  class Account implements Serializable { // Serializable
     String accNo = "abc123";
  ```

```
   Bank bank = new Bank(); // Contains a Bank object (which is
Serializable)
}
class Employee implements Serializable { // Serializable
   int eno = 1234;
   Account account = new Account(); // Contains an Account object
(which contains a Bank)
}

public class Main {
   public static void main(String[] args) throws Exception {
      FileOutputStream fos = new FileOutputStream("objgraph.ser");
      ObjectOutputStream oos = new ObjectOutputStream(fos);
      Employee e = new Employee(); // Object of the top-level class
      oos.writeObject(e); // Serializing the Employee object
      oos.close();
      fos.close();
      System.out.println("Serialization Successful for Object
Graph.");
      // Deserialization would restore Employee, Account, and Bank
objects.
   }
}
```

Reasoning: Serialization follows the object's references. As long as all objects in the chain implement `Serializable`, the entire graph is saved. If `Bank` or `Account` did *not* implement `Serializable`, this code would throw `NotSerializableException`.

Externalization

`java.io.Externalizable` is an interface that provides a way to customize the serialization process. It is a subinterface of `Serializable` but is *not* a marker interface.

Purpose of Externalization:

To have complete control over the serialization and deserialization process, allowing for:

- Applying custom encoding/decoding logic.
- Implementing data compression/decompression.
- Choosing exactly which data gets saved.
- Potentially improving performance and reducing file size compared to default serialization.

Steps to use Externalization:

1. **Create an Externalizable class:**
   - Declare a class that implements the `java.io.Externalizable` interface.
   - Implement the two methods declared in `Externalizable`:
     - `public void writeExternal(ObjectOutput out) throws IOException`: Called during serialization. Contains the logic to write the desired object data to the `ObjectOutput` stream using methods like `writeInt()`, `writeUTF()`, `writeLong()`, etc. You decide what and how to write.
     - `public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException`: Called during

deserialization. Contains the logic to read the data from the `ObjectInput` stream using methods like `readInt()`, `readUTF()`, `readLong()`, etc., in the *same order* they were written in `writeExternal`. Use this data to restore the object's state.

  - Declare a **public no-argument constructor**. This is required by the deserialization mechanism to create an instance of the class before calling `readExternal` on it. Default serialization does not require this.

2. **Perform Serialization and Deserialization:** Use `ObjectOutputStream.writeObject()` and `ObjectInputStream.readObject()` just as with `Serializable`. The JVM will detect that the object is `Externalizable` and call `writeExternal` and `readExternal` instead of the default serialization/deserialization logic.

Code Example (Externalization):

Java

```
import java.io.*; // Import all necessary classes

class Student implements Externalizable { // Implements Externalizable

    private String sid;
    private String sname;
    private String semail;
    private String smobile;

    // Public no-argument constructor IS REQUIRED for Externalization
    public Student() {
        System.out.println("Student: No-argument constructor called"); //
Trace
    }

    // Constructor to create objects
    public Student(String sid, String sname, String semail, String smobile)
{
        this.sid = sid;
        this.sname = sname;
        this.semail = semail;
        this.smobile = smobile;
    }

    // Helper method to display details
    public void getStudentDetails() {
        System.out.println("Student Details");
        System.out.println("-------------------");
        System.out.println("Student ID     : " + sid);
        System.out.println("Student Name   : " + sname);
        System.out.println("Student Email  : " + semail);
        System.out.println("Student Mobile :  " + smobile);
    }

    @Override
    public void writeExternal(ObjectOutput out) throws IOException {
        System.out.println("from writeExternal() method....."); // Trace
        // Custom logic to manipulate data before writing
        // Example: Extracting parts of the data
        String[] items1 = sid.split("-"); // "DSS-111" -> "111"
        int sno = Integer.parseInt(items1[1]);
        String[] items2 = semail.split("@"); // "durga@dss.com" -> "durga"
        String emailId = items2[0];
        String[] items3 = smobile.split("-"); // "91-9988776655" ->
"9988776655"
        long mobileNo = Long.parseLong(items3[1]);

        // Write the manipulated data to the stream
        out.writeInt(sno);
        out.writeUTF(sname); // writeUTF for Strings
        out.writeUTF(emailId);
        out.writeLong(mobileNo);
    }

    @Override
    public void readExternal(ObjectInput in) throws IOException,
ClassNotFoundException {
        System.out.println("from readExternal() method....."); // Trace
        // Custom logic to reconstruct data while reading
        // Example: Rebuilding the original format
        sid = "DSS-" + in.readInt(); // Read int and prepend "DSS-"
        sname = in.readUTF(); // Read String
        semail = in.readUTF() + "@dss.com"; // Read String and append
"@dss.com"
        smobile = "91-" + in.readLong(); // Read long and prepend "91-"
    }
}

public class Main {
    public static void main(String[] args) throws Exception {
        // Serialization
        FileOutputStream fos = new
FileOutputStream("/Users/nagoorn/Documents/docs/std_ext.ser"); // Unique
filename
        ObjectOutputStream oos = new ObjectOutputStream(fos);

        Student std1 = new Student("DSS-111", "Durga", "durga@dss.com",
"91-9988776655");
        System.out.println("Student Details before Serialization");
        std1.getStudentDetails();
        System.out.println("Before Performing Serialization");
        oos.writeObject(std1); // Triggers writeExternal()
        oos.close();
        fos.close();
        System.out.println("After Performing Serialization");
        System.out.println();

        // Deserialization
        FileInputStream fin = new
FileInputStream("/Users/nagoorn/Documents/docs/std_ext.ser");
        ObjectInputStream ois = new ObjectInputStream(fin);
        System.out.println("Before Performing Deserialization");
        Student std2 = (Student) ois.readObject(); // Triggers no-arg
constructor and then readExternal()
        ois.close();
        fin.close();
        System.out.println("After Performing Deserialization");
        System.out.println("Student Details After Deserialization");
        std2.getStudentDetails();
    }
}
```

Output Example:

```
Student Details before Serialization
Student Details
-------------------
Student ID     : DSS-111
Student Name   : Durga
Student Email  : durga@dss.com
Student Mobile :  91-9988776655
Before Performing Serialization
from writeExternal() method.....
After Performing Serialization

Before Performing Deserialization
Student: No-argument constructor called // Constructor called first
from readExternal() method.....      // Then readExternal is called
After Performing Deserialization
Student Details After Deserialization
Student Details
-------------------
Student ID     : DSS-111
Student Name   : Durga
Student Email  : durga@dss.com
Student Mobile :  91-9988776655
```

Reasoning: The output traces show that `writeExternal` was called during serialization and the no-argument constructor followed by `readExternal` were called during deserialization. This confirms the custom logic within these methods was executed, successfully transforming and restoring the data.

Note: `java.io.Externalizable` extends `java.io.Serializable`. Implementing `Externalizable` means the class is also `Serializable`, but it opts for custom control over the default serialization provided by `Serializable`.

File System In Java

The `java.io.File` class represents a file or directory path in the file system. It is **not** used for reading/writing data to files directly, but for managing file/directory properties, creation, deletion, navigation, etc.

Types of Files (Conceptual):

- **Sequential Files:** Data is read/written in a linear order from beginning to end. (Handled by streams like `FileInputStream`/`FileOutputStream`, `FileReader`/`FileWriter`).
- **Random Access Files:** Allows reading and writing data at arbitrary positions within the file. (Handled by `RandomAccessFile`).

Using `java.io.File`

- **Creating a `File` object:**
  - `public File(String filePath)`: Creates a `File` object representing the path. **This does not create the physical file or directory on disk.**

        - *Example:* `File file = new File("E:/abc/xyz/emp.txt");` (Only creates the object in memory).

- **Creating Physical File/Directory:**
    - `public boolean createNewFile() throws IOException`: Creates the file represented by the `File` object on disk. Returns `true` if the file was created, `false` if it already existed.
    - `public boolean mkdir()`: Creates the directory represented by the `File` object. Returns `true` if created, `false` if it already existed or creation failed.
    - `public boolean mkdirs()`: Creates the directory, including any necessary but nonexistent parent directories.

- **Getting File/Directory Details:**
    - `public String getName()`: Returns the name of the file or directory.
    - `public String getParent()`: Returns the path of the parent directory, or `null` if the path is at the root.
    - `public String getAbsolutePath()`: Returns the absolute path string of the file or directory.

- **Checking File/Directory Type and Existence:**
    - `public boolean isFile()`: Returns `true` if the `File` object represents an existing file, `false` otherwise.
    - `public boolean isDirectory()`: Returns `true` if the `File` object represents an existing directory, `false` otherwise.
    - `public boolean exists()`: Returns `true` if the file or directory exists in the file system.

- **Connecting `File` objects to Streams:** You can pass a `File` object to the constructors of file stream classes:
    - `public FileOutputStream(File file)`
    - `public FileWriter(File file)`
    - `public FileInputStream(File file)`
    - `public FileReader(File file)`

Code Example (File Operations):

Java

```
import java.io.*; // Import necessary classes

public class Main {
   public static void main(String[] args) throws Exception {
      File file = new File("/Users/nagoorn/Documents/docs/hello_file.txt");
// Create File object (in memory)

      System.out.println("File Created? (before createNewFile): " +
file.isFile()); // Checks if *physical* file exists

      boolean fileCreated = file.createNewFile(); // Create the physical
file
      System.out.println("File Created? (after createNewFile): " +
file.isFile()); // Check again

      System.out.println("File Name      : " + file.getName());
      System.out.println("File Parent    : " + file.getParent());
      System.out.println("Absolute Path  : " + file.getAbsolutePath());

      // Writing data using FileOutputStream with File object
      FileOutputStream fos = new FileOutputStream(file);
      String data = "Hello User! Writing to File object.";
      fos.write(data.getBytes());
      fos.close();
      System.out.println("Data written to file.");

      // Reading data using FileInputStream with File object
      FileInputStream fis = new FileInputStream(file);
      byte[] bytes = new byte[fis.available()]; // Note: available() can be
unreliable for large files/network streams
      fis.read(bytes);
      String data1 = new String(bytes);
      System.out.println("Data read from file: " + data1);
      fis.close();

      // To clean up: file.delete(); // Deletes the physical file
   }
}
```

Output Example:

```
File Created? (before createNewFile): false
File Created? (after createNewFile): true
File Name      : hello_file.txt
File Parent    : /Users/nagoorn/Documents/docs
Absolute Path  : /Users/nagoorn/Documents/docs/hello_file.txt
Data written to file.
Data read from file: Hello User! Writing to File object.
```

Reasoning: This demonstrates that creating a `File` object doesn't create the file on disk (`isFile()` is false initially), but `createNewFile()` does (`isFile()` becomes true). It also shows how to use a `File` object when creating file streams.

Code Example (Directory Operations):

Java

```
import java.io.*;

public class Main {
   public static void main(String[] args) throws Exception {
      File dir = new File("/Users/nagoorn/Documents/docs/student_dir"); //
Create File object for directory

      System.out.println("Directory Created? (before mkdir): " +
dir.isDirectory()); // Checks if physical directory exists

      boolean dirCreated = dir.mkdir(); // Create the physical directory
      System.out.println("Directory Created? (after mkdir): " +
dir.isDirectory()); // Check again

      System.out.println("Directory Name    : " + dir.getName());
      System.out.println("Directory Parent  : " + dir.getParent());
      System.out.println("Absolute Path     : " + dir.getAbsolutePath());
```

```
        // To clean up: dir.delete(); // Deletes the physical directory (must
be empty)
    }
}
```

Output Example:

```
Directory Created? (before mkdir): false
Directory Created? (after mkdir): true
Directory Name    : student_dir
Directory Parent  : /Users/nagoorn/Documents/docs
Absolute Path     : /Users/nagoorn/Documents/docs/student_dir
```

Reasoning: Similar to files, creating a `File` object for a directory doesn't create it on disk (`isDirectory()` is false). `mkdir()` creates the directory (`isDirectory()` becomes true).

RandomAccessFile

`java.io.RandomAccessFile` allows reading and writing data at specific byte positions within a file. It differs from streams like `FileInputStream`/`FileOutputStream` which primarily support sequential access.

- **Creating `RandomAccessFile` object:**
    - `public RandomAccessFile(String fileName, String accessPrivileges) throws FileNotFoundException`: Creates a random access file stream.
        - `accessPrivileges`: Specifies the mode - `"r"` (read only) or `"rw"` (read and write).
        - *Example:* `RandomAccessFile file = new RandomAccessFile("path/to/hello.txt", "rw");`
    - *Reasoning:* Opening in `"r"` mode prevents any write operations. Attempting to write will cause an `IOException: Access denied`.
- **Reading and Writing Data:** Provides various `readXXX()` and `writeXXX()` methods for primitive types and Strings (UTF). The file pointer advances automatically after each read/write operation.
    - `public void writeInt(int v) throws IOException`
    - `public void writeUTF(String str) throws IOException`
    - `public int readInt() throws IOException`
    - `public String readUTF() throws IOException`
    - *(and many others for byte, short, long, float, double, boolean)*
- **Moving the File Pointer:**
    - `public void seek(long pos) throws IOException`: Sets the file pointer to the specified byte position. `pos` is the number of bytes from the beginning of the file.
- **Closing the File:**
    - `public void close() throws IOException` (or use try-with-resources).

Code Example (RandomAccessFile):

Java

```
import java.io.*;

public class Main {
   public static void main(String[] args) throws Exception {
      RandomAccessFile randomAccessFile = new
RandomAccessFile("/Users/nagoorn/Documents/docs/hello_random.txt", "rw");
// Open in read/write mode

      // Write data sequentially
      randomAccessFile.writeInt(111); // Writes an int (4 bytes)
      randomAccessFile.writeUTF("Durga"); // Writes a String (UTF format -
length + bytes)
      randomAccessFile.writeFloat(50000); // Writes a float (4 bytes)
      randomAccessFile.writeUTF("Hyderabad"); // Writes another String

      // Move the file pointer back to the beginning
      randomAccessFile.seek(0);

      // Read data sequentially from the current pointer position
(beginning)
      System.out.println("Employee Details");
      System.out.println("----------------------");
      System.out.println("Employee Number    : " +
randomAccessFile.readInt()); // Reads the int
      System.out.println("Employee Name      : " +
randomAccessFile.readUTF()); // Reads the first String
      System.out.println("Employee Salary    : " +
randomAccessFile.readFloat()); // Reads the float
      System.out.println("Employee Address   : " +
randomAccessFile.readUTF()); // Reads the second String

      randomAccessFile.close(); // Close the file
   }
}
```

Output Example:

```
Employee Details
----------------------
Employee Number    : 111
Employee Name      : Durga
Employee Salary    : 50000.0
Employee Address   : Hyderabad
```

Reasoning: Data is written to the file using type-specific methods. `seek(0)` repositions the pointer to the start. Reading with the corresponding type-specific methods in the *same order* successfully retrieves the data. `RandomAccessFile` is useful for structured binary files or when needing to modify parts of a file without reading the whole thing into memory.