Java I/O Streams: A Comprehensive Example

Stream Classification & Demonstration

Introduction

In Java, streams are classified into two main categories based on the type of data they handle:

- **Byte Streams**: Handle I/O of raw binary data (8-bit bytes). Used for images, audio, or any binary files.
  - `FileInputStream`, `FileOutputStream`
  - `BufferedInputStream`, `BufferedOutputStream`
  - `DataInputStream`, `DataOutputStream`
  - `ObjectInputStream`, `ObjectOutputStream`
- **Character Streams**: Handle I/O of text data (16-bit Unicode). Used for human-readable text.
  - `FileReader`, `FileWriter`
  - `BufferedReader`, `BufferedWriter`
  - `PrintWriter`

Graphical Representation of Reader and Writer Hierarchy

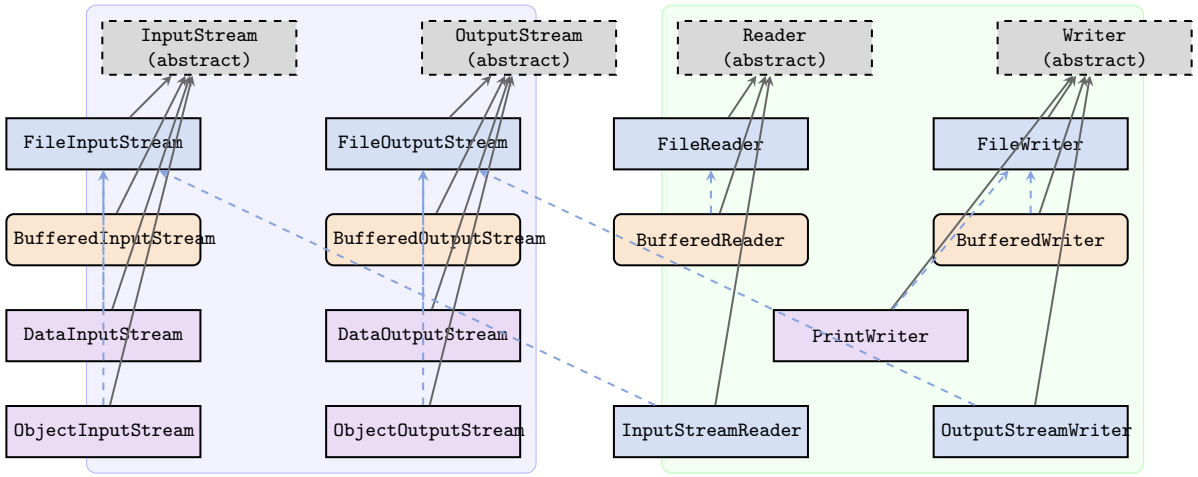


Figure 1: Java I/O Stream Hierarchy (Compact View)

Understanding the Diagram

- **Solid Arrows**: Represent inheritance (is-a relationship)
- **Dashed Arrows**: Represent wrapping/decorator pattern (has-a relationship)
- **Blue Background**: Byte Streams (handle 8-bit bytes)
- **Green Background**: Character Streams (handle 16-bit Unicode)

Why Different Types of Readers and Writers?

Stream Type	Purpose	Why Use It?
FileReader/FileWriter	Basic text I/O	Direct character-level access to files
BufferedReader/BufferedWriter	Efficient text I/O	Reduces I/O operations using buffering; supports `readLine()`
PrintWriter	Formatted output	Provides `println()`, `printf()`, and auto-flush
InputStreamReader/OutputStreamWriter	Byte-char bridge	Converts byte streams to character streams and handles encoding

Table 1: Character Stream Readers and Writers

Stream Type	Purpose	Why Use It?
FileInputStream/FileOutputStream	Basic binary I/O	Direct byte-level access to files
BufferedInputStream/BufferedOutputStream	Efficient binary I/O	Reduces disk access using buffering
DataInputStream/DataOutputStream	Primitive I/O	Machine-independent reading and writing of primitive data
ObjectInputStream/ObjectOutputStream	Object serialization	Stores and retrieves complete objects including object graphs

Table 2: Byte Stream Readers and Writers

The Decorator Pattern in Java I/O

Java I/O uses the **Decorator Pattern** to add functionality dynamically:

```
// Wrapping example - adding buffering to file reading
BufferedReader br = new BufferedReader(new FileReader("file.txt"));

// Wrapping example - adding data handling to file I/O
DataOutputStream dos = new DataOutputStream(
    new BufferedOutputStream(
        new FileOutputStream("data.dat")));
```

Each wrapper adds new capabilities:

- **Buffered Streams**: Add buffering for performance
- **Data Streams**: Add primitive type handling
- **Object Streams**: Add object serialization
- **PrintWriter**: Adds formatted output methods

The following program demonstrates all major stream types with proper resource management using try-with-resources.

```
import java.io.*;
import java.util.*;

public class AllStreamsDemo {
    public static void main(String[] args) throws IOException {

        // ========== 1. BYTE STREAMS ==========
        System.out.println("\n=== Byte Streams (FileInputStream/FileOutputStream) ===");

        // Writing and reading raw bytes
        try (FileOutputStream fos = new FileOutputStream("byte_data.dat");
             FileInputStream fis = new FileInputStream("byte_data.dat")) {

            // Write some bytes
            fos.write(65);  // Writes 'A'
            byte[] bytes = {66, 67, 68};  // B, C, D
            fos.write(bytes);
```

```
18
19              // Read bytes
20              System.out.println("Available bytes: " + fis.available());
21              int byteRead;
22              System.out.print("Reading bytes: ");
23              while ((byteRead = fis.read()) != -1) {
24                  System.out.print((char) byteRead + " ");
25              }
26              System.out.println();
27          }
28
29          // ========== 2. BUFFERED BYTE STREAMS ==========
30          System.out.println("\n=== Buffered Byte Streams ===");
31
32          try (BufferedOutputStream bos = new BufferedOutputStream(
33                  new FileOutputStream("buffered_byte.dat"));
34               BufferedInputStream bis = new BufferedInputStream(
35                  new FileInputStream("buffered_byte.dat"))) {
36
37              byte[] data = "Buffered Stream Example".getBytes();
38              bos.write(data);
39              bos.flush();
40
41              byte[] buffer = new byte[1024];
42              int bytesRead = bis.read(buffer);
43              System.out.println("Read: " + new String(buffer, 0, bytesRead));
44          }
45
46          // ========== 3. DATA STREAMS ==========
47          System.out.println("\n=== Data Streams (Primitive Types) ===");
48
49          try (DataOutputStream dos = new DataOutputStream(
50                  new FileOutputStream("data.dat"));
51               DataInputStream dis = new DataInputStream(
52                  new FileInputStream("data.dat"))) {
53
54              // Write primitives
55              dos.writeInt(42);
56              dos.writeDouble(3.14159);
57              dos.writeBoolean(true);
58              dos.writeUTF("Hello Data Stream");
59
60              // Read in same order
61              int intVal = dis.readInt();
62              double doubleVal = dis.readDouble();
63              boolean boolVal = dis.readBoolean();
64              String strVal = dis.readUTF();
65
66              System.out.println("Read int: " + intVal);
67              System.out.println("Read double: " + doubleVal);
68              System.out.println("Read boolean: " + boolVal);
69              System.out.println("Read String: " + strVal);
70          }
71
72          // ========== 4. OBJECT STREAMS (Serialization) ==========
73          System.out.println("\n=== Object Streams (Serialization) ===");
74
75          // Define a simple serializable class
76          class Person implements Serializable {
77              private static final long serialVersionUID = 1L;
78              String name;
79              int age;
80              Person(String name, int age) { this.name = name; this.age = age; }
81              @Override public String toString() { return name + " (" + age + ")"; }
82          }
83
84          try (ObjectOutputStream oos = new ObjectOutputStream(
85                  new FileOutputStream("person.ser"));
86               ObjectInputStream ois = new ObjectInputStream(
87                  new FileInputStream("person.ser"))) {
88
89              Person person = new Person("Alice", 30);
90              oos.writeObject(person);
```

```
91
92              Person deserialized = (Person) ois.readObject();
93              System.out.println("Deserialized: " + deserialized);
94
95          } catch (ClassNotFoundException e) {
96              System.out.println("Class not found: " + e.getMessage());
97          }
98
99          // ========== 5. CHARACTER STREAMS ==========
100         System.out.println("\n=== Character Streams (FileReader/FileWriter) ===");
101
102         try (FileWriter fw = new FileWriter("char_data.txt");
103              FileReader fr = new FileReader("char_data.txt")) {
104
105             fw.write("Hello World\n");
106             fw.write("Character Streams");
107             fw.flush();
108
109             int ch;
110             System.out.print("Reading characters: ");
111             while ((ch = fr.read()) != -1) {
112                 System.out.print((char) ch);
113             }
114             System.out.println();
115         }
116
117         // ========== 6. BUFFERED CHARACTER STREAMS ==========
118         System.out.println("\n=== Buffered Character Streams ===");
119
120         try (BufferedWriter bw = new BufferedWriter(
121                 new FileWriter("buffered_char.txt"));
122              BufferedReader br = new BufferedReader(
123                 new FileReader("buffered_char.txt"))) {
124
125             bw.write("First line");
126             bw.newLine();
127             bw.write("Second line");
128             bw.flush();
129
130             String line;
131             System.out.println("Reading lines:");
132             while ((line = br.readLine()) != null) {
133                 System.out.println("  " + line);
134             }
135         }
136
137         // ========== 7. PRINTWRITER ==========
138         System.out.println("\n=== PrintWriter (Formatted Text Output) ===");
139
140         try (PrintWriter pw = new PrintWriter(new FileWriter("printwriter.txt"))) {
141             pw.println("Hello PrintWriter");
142             pw.printf("Formatted: %d + %.2f = %.2f%n", 10, 5.5, 15.5);
143             pw.format("Another format: %s%n", "Easy text output");
144
145             if (!pw.checkError()) {
146                 System.out.println("PrintWriter wrote successfully");
147             }
148         }
149
150         // ========== 8. CONSOLE I/O ==========
151         System.out.println("\n=== Console I/O ===");
152
153         // Reading from console (System.in)
154         try (BufferedReader consoleReader = new BufferedReader(
155                 new InputStreamReader(System.in))) {
156             System.out.print("Enter some text (or just press Enter): ");
157             String input = consoleReader.readLine();
158             System.out.println("You entered: " + input);
159         } catch (IOException e) {
160             System.err.println("Error reading from console: " + e.getMessage());
161         }
162
163         // Writing to console
```

```
164         System.out.println("Writing to System.out (standard output)");
165         System.err.println("Writing to System.err (error output)");
166 
167         System.out.println("\n=== Demo Complete ===");
168         System.out.println("Generated files: byte_data.dat, buffered_byte.dat, data.dat,
        ");
169         System.out.println("                person.ser, char_data.txt, buffered_char.txt
        , ");
170         System.out.println("                printwriter.txt");
171     }
172 }
```

Listing 1: Comprehensive Java Streams Demonstration

Code Clarifications

The clarifications below correspond to line numbers in the program.

1. **Lines 1-9**: Import statements for all necessary I/O classes.
2. **Lines 12-13**: Class declaration and main method with `throws IOException` to delegate exception handling.

Byte Streams - FileInputStream & FileOutputStream

3. **Line 15**: Comment indicating the start of byte stream operations.
4. **Lines 17-18**: Try-with-resources block that automatically closes streams.
   - `FileOutputStream("byte_data.dat")`: Creates a byte stream for writing raw bytes to a file. If file doesn't exist, it's created.
   - `FileInputStream("byte_data.dat")`: Creates a byte stream for reading raw bytes from the file.
5. **Lines 20-21**: Write operations using `write(int b)` and `write(byte[] b)`. Byte streams work with raw bytes (0-255).
6. **Line 23**: `available()` returns the number of bytes that can be read without blocking.
7. **Lines 24-27**: Reading bytes one by one using `read()`, which returns -1 at end of file.

Buffered Byte Streams

8. **Lines 31-32**: Buffered streams wrap basic streams to add buffering capability.
   - `BufferedOutputStream`: Reduces the number of writes to disk by batching data.
   - `BufferedInputStream`: Reduces the number of reads from disk by reading chunks.
9. **Line 34**: `write(byte[])` writes the entire array efficiently.
10. **Line 35**: `flush()` forces any buffered data to be written immediately.
11. **Lines 38-41**: Reading in chunks using `read(byte[])` for better performance.

Data Streams

12. **Lines 45-46**: Data streams allow reading/writing Java primitive types.
    - `DataOutputStream`: Writes primitives in a machine-independent way.
    - `DataInputStream`: Reads primitives written by `DataOutputStream`.
13. **Lines 48-52**: Writing various primitive types (`writeInt`, `writeDouble`, `writeBoolean`, `writeUTF`).
14. **Lines 54-58**: Reading back the primitives in the same order they were written.

Object Streams (Serialization)

15. **Lines 62-63**: Object streams for serializing/deserializing objects.
    - `ObjectOutputStream`: Converts objects to byte stream.
    - `ObjectInputStream`: Reconstructs objects from byte stream.
    - Objects must implement `Serializable` interface.
16. **Line 65-69**: Creating a simple serializable class as a local class.
17. **Lines 72-73**: Writing object to file using `writeObject()`.
18. **Line 76**: Reading object back using `readObject()`, which returns `Object` type.

Character Streams - FileReader & FileWriter

19. **Line 81**: Comment indicating character stream operations.
20. **Lines 83-84**: Character streams for text file handling (16-bit Unicode).
    - `FileWriter`: Writes characters to file.
    - `FileReader`: Reads characters from file.
21. **Lines 86-87**: Writing characters using `write()` method.
22. **Lines 89-92**: Reading characters one by one using `read()`.

Buffered Character Streams

23. **Lines 96-97**: Buffered character streams for efficient text processing.
    - `BufferedWriter`: Buffers character output.
    - `BufferedReader`: Buffers character input with `readLine()` method.
24. **Line 99**: `newLine()` writes platform-independent line separator.
25. **Line 100**: `flush()` ensures all buffered data is written.
26. **Lines 102-105**: Reading line by line with `readLine()`, which returns `null` at EOF.

PrintWriter

27. **Line 109**: `PrintWriter` provides convenient methods for formatted text output.
28. **Lines 111-113**: Demonstrates various output methods:
    - `println()`: Writes with line separator.
    - `printf()`: Formatted output similar to C's printf.
    - `format()`: Alternative formatting method.
29. **Line 114**: `checkError()` verifies if any error occurred during output.

Console I/O

30. **Lines 118-120**: Console I/O using `System.in` wrapped in `BufferedReader`.
31. **Line 122**: `System.out.println()` writes to console output.
32. **Lines 124-126**: `System.err` for error output (typically shown in red).

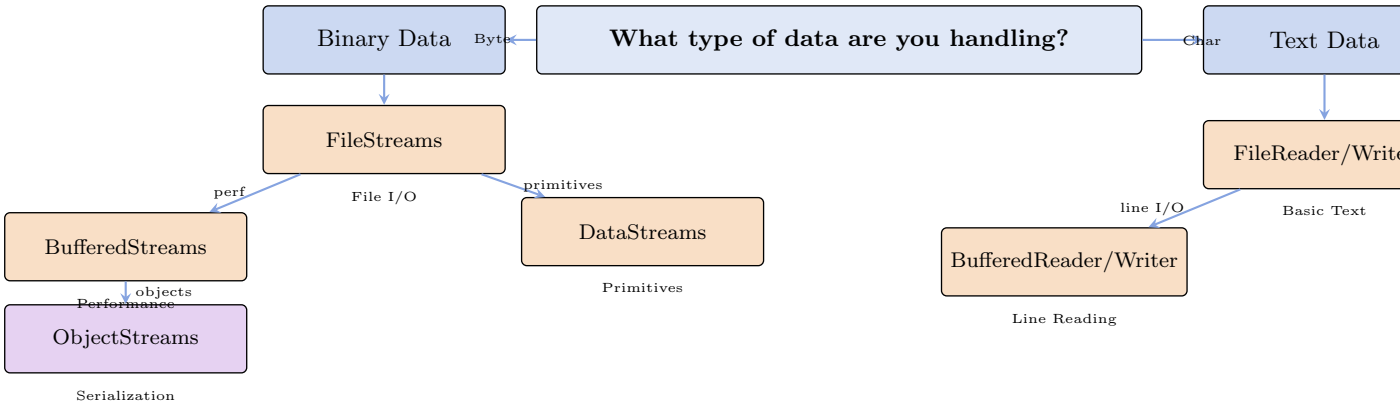
Cleanup and Execution

33. **Lines 130-133**: Informational output about generated files.
34. **Line 136**: Closing bracket for main method.

Key Takeaways

- **Byte vs Character**: Use byte streams for binary data, character streams for text.
- **Buffering**: Always wrap streams with buffered versions for better performance.
- **try-with-resources**: Automatically closes resources, preventing resource leaks.
- **Data Streams**: Ideal for reading/writing primitive types consistently.
- **Object Streams**: Enable object persistence through serialization.
- **PrintWriter**: Preferred for text output with formatting capabilities.
- **Bridge Streams**: `InputStreamReader` and `OutputStreamWriter` bridge byte and character streams, handling character encoding.

Stream Selection Guide



Use Case	Recommended Stream
Image/audio files	FileInputStream/OutputStream
Text files	FileReader/Writer + BufferedReader/Writer
Primitive data	DataInputStream/OutputStream
Object persistence	ObjectInputStream/OutputStream
Formatted text	PrintWriter
Different encoding	InputStreamReader/OutputStreamWriter

Figure 2: Stream Selection Decision Tree