Java I/O

Table of contents

Introduced in 2026-04-23-java-io[1] (Lecture, Week 8).

Java spreads its I/O functionality across many more classes than most other languages, split roughly into three generations of API:

1. **Streams** (`java.io`, since 1.0) — byte-oriented.
2. **Readers & Writers** (`java.io`, since 1.1) — character-oriented, added because byte streams weren't ideal for text/Unicode.
3. **New I/O** (`java.nio.file`, since 1.7) — file-system operations (paths, directories, attributes), which `java.io` was never designed for.

Streams

A **stream** is an abstraction that either produces or consumes information, letting Java perform I/O uniformly regardless of whether data comes from a file, keyboard, console, network socket, or elsewhere. We often don't want to rewrite our program just because the source or destination changed.

- An **output stream** is an abstract destination to be written to.
- An **input stream** is an abstract source of input that can be read without concern for how it's supplied.

Java has two families of streams:

[1] courses/csse2002/2026-04-23-java-io.pdf

- **Byte streams** (`InputStream`/`OutputStream`) — raw binary data (files, images, network data).
- **Character streams** (`Reader`/`Writer`, see below) — characters/text, handling Unicode.

  A byte stream reads data as bytes, whereas a character stream reads data as characters. (Schildt, *Java: The Complete Reference*, 11th ed.)

`InputStream` / `OutputStream`

`InputStream` and its subclasses represent a stream of bytes, drawn from different sources (`FileInputStream` from a file, `ByteArrayInputStream` from an array, …). Methods that use streams should accept the **superclass** type as a parameter, so any concrete stream can be substituted (see java-solid-principles[2] — Liskov Substitution):

```
private static void sendToStream(OutputStream stream) throws IOException {
    String output = "foo bar baz";
    for (char letter : output.toCharArray()) {
        stream.write(letter);
    }
    stream.flush();
}
// sendToStream(System.out);
// sendToStream(new FileOutputStream("output.txt"));
// sendToStream(new ByteArrayOutputStream());
```

End of file: all input streams need to consider "end of file" — `read()` returns `-1` at EOF, so a loop reading one byte at a time typically continues `while (in != -1)`.

Buffering

Reading a file a byte at a time is slow. `BufferedInputStream` wraps another input stream and buffers reads (the buffer is an area of main memory used to temporarily hold data) — on one test VM, reading a 4MB file took 82ms buffered vs. 18,193ms unbuffered:

```
readAll(new BufferedInputStream(new FileInputStream("output.txt")));
```

[2] courses/csse2002/java-solid-principles.pdf

Closing streams

Streams (and Readers) need closure — systems may limit how many files can be open at once, so always `close()` a stream when finished with it. Wrapping cleanup in `try/catch/finally` gets verbose fast:

```
BufferedInputStream input = null;
try {
    input = new BufferedInputStream(new FileInputStream("output.txt"));
    readAll(input);
} catch (IOException e) {
    System.out.println("Error: File Not Found " + e);
} finally {
    try {
        input.close();
    } catch (IOException e) {
        System.out.println("Error closing the stream: " + e);
    }
}
```

`try-with-resources` is much cleaner — any resource declared in the `try(...)` parentheses is automatically closed (Effective Java, 3rd ed., Item 9: "Prefer try-with-resources to try-finally"):

```
try (BufferedInputStream input = new BufferedInputStream(new
 ↪  FileInputStream("output.txt"))) {
    readAll(input);
} catch (IOException e) {
    System.out.println("Error: File Not Found " + e);
}
```

Readers & Writers

`Reader/Writer` are the character-based counterparts of `InputStream/OutputStream`. `InputStreamReader` bridges the two: it wraps an `InputStream` (like `System.in`) and decodes its bytes into characters.

```
private static void readAndPrint(Reader reader) throws IOException {
    char[] letters = new char[10];
    for (int i = 0; i < 10; i++) {
        letters[i] = (char) reader.read();
    }
    System.out.println(letters);
}
```

```
// readAndPrint(new InputStreamReader(System.in));
// readAndPrint(new FileReader("myfile.txt"));
```

`BufferedReader`

Wraps another `Reader`; as well as buffering, it adds `String readLine()`:

```
BufferedReader reader = new BufferedReader(new FileReader("readwithbuffer.txt"));
for (int i = 0; i < 5; i++) {
    System.out.println(reader.readLine());
}
```

`PrintWriter`

`System.out` is actually a `PrintStream`; `PrintWriter` is a better option for character output — it can write to many destinations and supports formatted text (`printf`):

```
try (FileWriter fileWriter = new FileWriter("writeroutput.txt");
     PrintWriter printWriter = new PrintWriter(fileWriter)) {
    printWriter.println("I love CSSE2002");
    printWriter.printf("Formatted number: %.2f%n", 123.45336);
} catch (IOException e) {
    e.printStackTrace();
}
```

`flush()`

If an `OutputStream`/`Writer` is buffered, output might not be sent immediately — `flush()` forces any pending output out. This matters for interactive situations (the other end won't respond if nothing's actually been sent yet) and for debugging/logging (an up-to-date view of what's happening). `close()`-ing a stream flushes it as well.

`Scanner`

`Scanner` (`java.util`, since 1.5) is neither a stream nor a reader — it's a **utility class** that wraps an existing `InputStream` or `Reader`, added to simplify reading user input and parsing primitive types/strings (a friendlier alternative to `BufferedReader`). It also supports regex-based scanning for advanced use cases.

```java
Scanner scanner = new Scanner(System.in); // internally wraps its own
 ↪  InputStreamReader
int total = 0;
while (scanner.hasNextInt()) {
    total += scanner.nextInt();
}
System.out.println(total);
```

Reading strings: `scanner.next()` reads the next word (skipping leading whitespace, stopping at whitespace); `scanner.nextLine()` reads the entire line up to Enter.

New I/O (`java.nio.file`)

`java.io` is mainly stream-oriented — its goal was reading/writing data, not managing files or directories, so it's awkward for file attributes, directory traversal, and path manipulation. `java.nio.file` (since Java 1.7) adds two key abstractions:

- `Path` — represents a file location.
- `Files` — utility methods for file operations.

Creating a `Path`

```java
Path p1 = Path.of("docs/output.txt");        // whole path as one string
Path p2 = Path.of("docs", "output.txt");     // separate name elements, joined by
 ↪  Java
Path p3 = Path.of(new URI("file:///docs/output.txt"));
```

`Files` operations

Category	Methods
Create/delete	`Files.createFile(Path)`, `Files.createDirectory(Path)`, `Files.delete(Path)`, `Files.deleteIfExists(Path)`
Query	`Files.exists(Path)`, `Files.isDirectory(Path)`, `Files.isRegularFile(Path)`
Manipulate	`Files.copy(Path, Path)`, `Files.move(Path, Path)`

Category	Methods
Read/write	`Files.readAllLines(Path)`, `Files.readString(Path)`, `Files.write(Path, Iterable<? extends CharSequence>)`, `Files.writeString(Path, CharSequence)`

`CharSequence` is an interface representing a read-only sequence of characters; `String`, `StringBuilder`, `StringBuffer`, and `CharBuffer` all implement it.

```
Path myPath = Path.of("src", "Week08", "NIO", "myfile.txt");
List<String> lines = Files.readAllLines(myPath);
for (String line : lines) {
    System.out.println(line);
}
```

Summary: which to use?

- **Streams** — good for binary data, a byte at a time.
- **Readers & Writers** — best for text data.
- **New I/O** — when manipulating a file system or file contents (paths, directories, copying/moving files).