Java I/O

Table of contents

Streams 1
Readers & Writers 7
`Scanner` 11
New I/O (`java.nio.file`) 12
Summary: which to use? 14

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.

`Scanner` (`java.util`, since 1.5) sits alongside these as a convenience utility — neither a stream nor a reader.

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.

- 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.

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

What the abstraction hides is threefold: the **source** (keyboard, disk file, network, …), the **destination** (terminal, disk file, network, …), and the **chunking** — whether data arrives one byte at a time (keyboard) or many at once (files).

How a program is written should not depend on whether its input is from a file or a keyboard, or whether its destination is a file, the console or the network. If the source or destination changes, the program should not need rewriting.

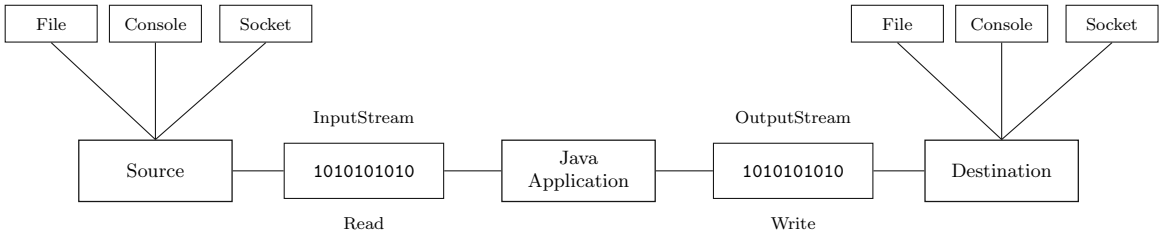


Figure 1: The stream abstraction: whatever the real source (file, console, socket), the application reads bytes through one `InputStream`, and writes them back out through one `OutputStream`, whatever the real destination.

Java has two families of streams:

- **Byte streams** (`java.io.InputStream` / `java.io.OutputStream`) — raw binary data (files, images, network data).
- **Character streams** (`Reader`/`Writer`, see below) — characters/text, handling Unicode; better suited to strings, text files and console text.

  A byte stream reads data as **bytes**, whereas a character stream reads data as **characters**.

Reference: *Java: The Complete Reference*, Eleventh Edition — Herbert Schildt.

The byte-stream hierarchy

`InputStream` and its subclasses represent a stream of bytes, drawn from different sources — `FileInputStream` gets bytes from a file, `ByteArrayInputStream` from an array, and so on. 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).

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

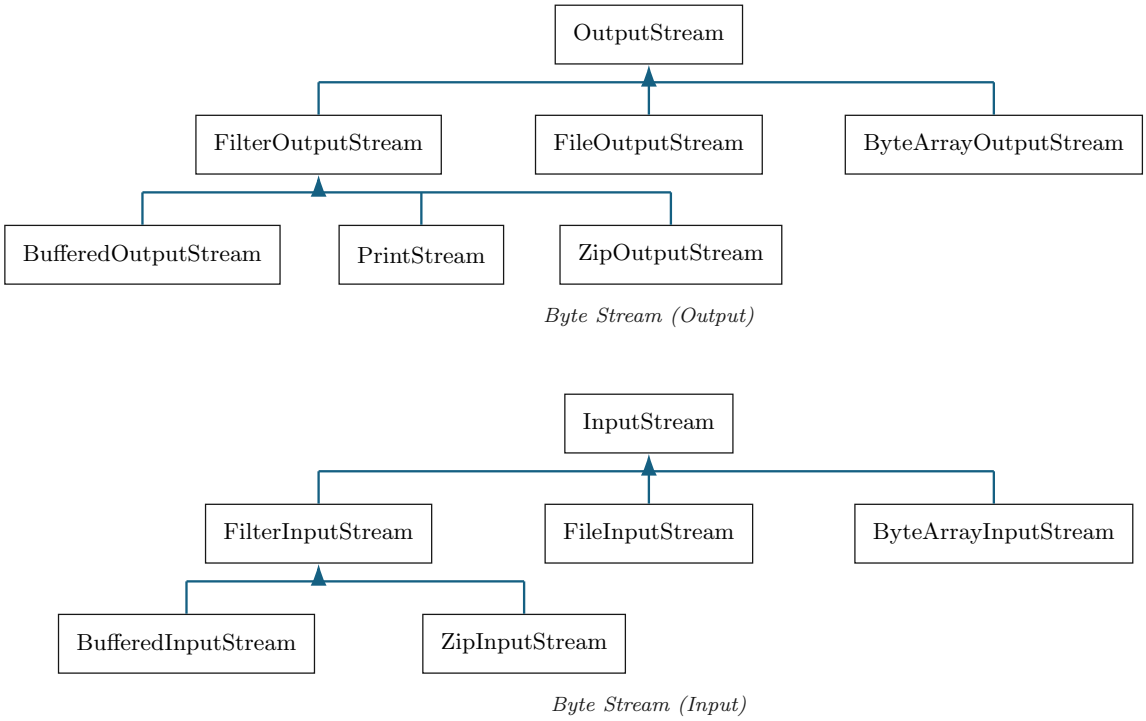


Figure 2: The byte-stream class hierarchies: `OutputStream` and `InputStream`. Each superclass sits above its subclasses, and the arrowhead points into the parent. The wrappers `BufferedInputStream` and `BufferedOutputStream` are `Filter*` streams — streams that wrap another stream.

Worked example — writing to an abstract `OutputStream`

`System.out` is an output stream already in everyday use. That makes it the first thing to hand a method that takes an abstract `OutputStream`: `sendToStream(System.out)`. The version below keeps `sendToStream` untouched and swaps the argument twice:

```
public class Foo {
    public static void main(String[] args) throws IOException {
        OutputStream fos = new FileOutputStream("output.txt");
        sendToStream(fos);

        fos = new ByteArrayOutputStream();
        sendToStream(fos);

        System.out.println(fos);
    }

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

`sendToStream` never changes: first `System.out` (straight to the console), then a `FileOutputStream`, which leaves `output.txt` holding `foo bar baz`, then a `ByteArrayOutputStream`. `main` has to declare `throws IOException`; without the declaration it does not compile. The final `System.out.println(fos)` prints the byte-array stream's accumulated contents, because `ByteArrayOutputStream.toString()` decodes the buffer it has collected.

End of file

All input streams (and everything built on them) need to consider "end of file":

```
public int read() // returns -1 on end of file
```

Reading a whole file one byte at a time therefore loops until `read()` hands back `-1`:

```
public class Foo {
    public static void main(String[] args) throws IOException {
        readAll(new FileInputStream("output.txt"));
```

```
    }

    private static void readAll(InputStream input) throws IOException {
        int in = 0;
        while (in != -1) {
            in = input.read();
            System.out.println(in);
        }
    }
}
```

Run against an `output.txt` containing `foo bar baz`, that prints the *byte values* — `102`, `111`, `111`, `32`, … The fix is to cast in the `println` inside the loop, so you see the characters instead:

```
System.out.println((char) in);
```

The loop as written tests `in` before refreshing it, so the final iteration still prints the sentinel — `-1`, or, once cast, the char with all sixteen bits set, , which is not a printable character. Reading into the loop condition, `while ((in = input.read()) != -1)`, avoids that.

Buffering

Getting information from files can be slow a `byte` or `char` at a time. `BufferedInputStream` is an input stream which **wraps around another input stream**. The buffer is an area in the main memory used to store or hold the data temporarily. This is faster because one large read from the device then serves many small `read()` calls.

Only the construction changes; `readAll` is untouched:

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

(What each layer actually contributes is drawn in Figure 3.)

On a test VM, reading a 4MB file takes **82 milliseconds with buffering** and **18,193 milliseconds without**.

Closing streams

Streams need closure. Streams and Readers have a `close()` method; systems may limit how many files you can have open at a time; when you have finished with a stream, close it.

Getting this right takes three steps. First, just catching the failure to open:

```
public static void main(String[] args) {
    try {
        readAll(new BufferedInputStream(new FileInputStream("output.txt")));
    } catch (IOException e) {
        System.out.println("Error: File Not Found " + e);
    }
}
```

That still never closes anything. Adding a `finally` block does — but the close can itself throw, so it needs its own `try`, and the whole thing 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 closed automatically:

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

> *Effective Java* (2017, Addison-Wesley) — Item 9: "Prefer try-with-resources to try-finally".

A cautionary example of what production I/O failures cost: a national passport office suspended issuance after a computer-system breakdown (reported November 2022).

Wrapping streams together

Nothing in `java.io` is one class doing everything: you assemble the behaviour you want by nesting one stream inside another. That is what the hierarchy's whole `Filter*` branch is for, and why the constructor calls look the way they do. The second chain below uses `InputStreamReader` and `BufferedReader`, which the next section introduces — it is drawn here so both chains can be compared side by side.

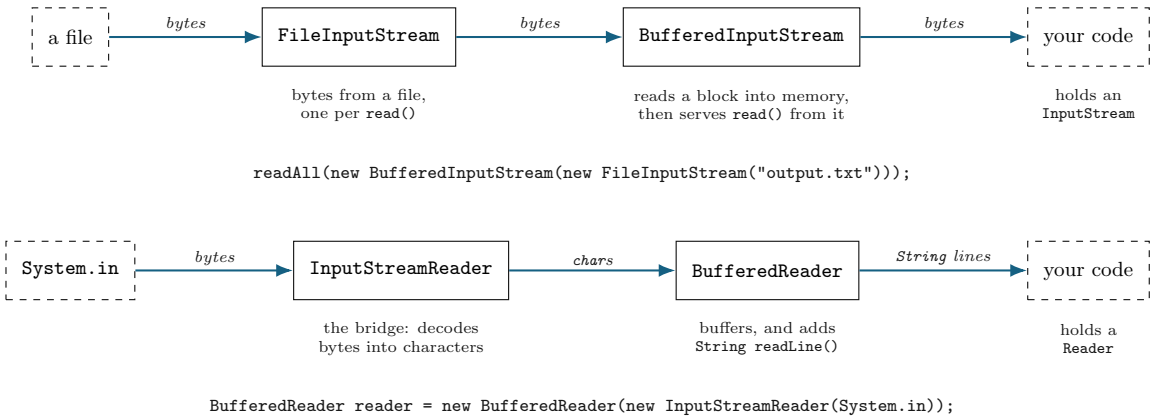


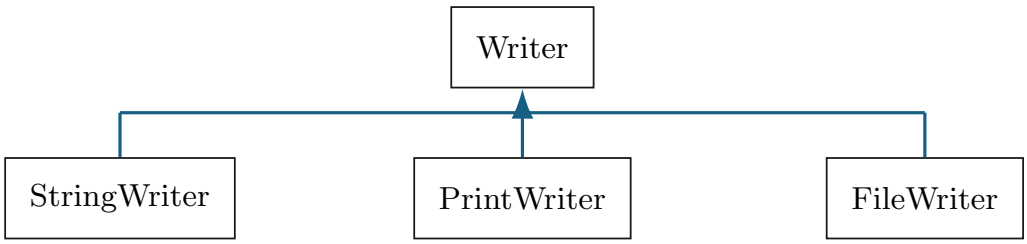
Figure 3: How the nesting in `new BufferedInputStream(new FileInputStream(...))` and `new BufferedReader(new InputStreamReader(System.in))` actually behaves — each wrapper adds one capability and passes the rest through, and your code only ever holds the outermost object. `BufferedInputStream` wraps another input stream; `InputStreamReader` acts as a bridge decoding bytes into characters; `BufferedReader` wraps another `Reader` and adds `readLine()`. The gloss beneath each box says what that stage contributes.

Readers & Writers

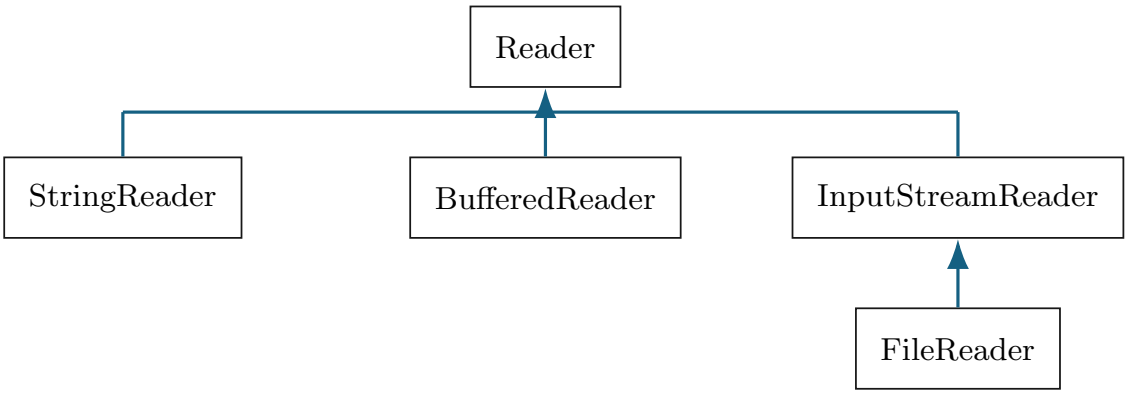
Originally Java I/O was byte-oriented. Byte streams were fine for raw binary data but not ideal for text, so character streams were added in 1.1 to handle text and Unicode characters more conveniently and efficiently.

- `java.io.Writer` — an abstraction over writing `char`s to an output.
- `java.io.Reader` — an abstraction over reading `char`s from an input source.

`InputStreamReader` is the **bridge** between the two systems: it wraps an `InputStream` (like `System.in`) and decodes its bytes into characters.



java.io.Writer — writing `chars` to an output



java.io.Reader — reading `chars` from an input source

Figure 4: The character-stream hierarchies: `Writer` and `Reader`, drawn the same way as Figure 2. The one two-level branch is the important one — `FileReader` is a kind of `InputStreamReader`, i.e. reading a text file is itself a case of decoding bytes into characters.

Worked example — read the first 10 characters of input

```
> Words, words, and more words
Words, wor
```

```java
public class ReaderEx {
    public static void main(String[] args) throws IOException {
        readAndPrint(new InputStreamReader(System.in));
    }

    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);
    }
}
```

Exercise: rewrite the program to use `FileReader`. Because `readAndPrint` takes the abstract `Reader`, only the argument in `main` has to change.

`BufferedReader`

`BufferedReader` wraps another `Reader`. As well as buffering it adds `String readLine()`:

```java
public class BufferedReaderEx {
    public static void main(String[] args) {
        BufferedReader reader = new BufferedReader(new
         ↪ FileReader("readwithbuffer.txt"));
        for (int i = 0; i < 5; i++) {
            System.out.println(reader.readLine());
        }
    }
}
```

Wrapping `System.in` instead of a file is the standard console-input idiom:

```java
BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
```

The exception handling has to be made explicit: `FileReader`'s constructor throws `FileNotFoundException`, and `readLine()` throws `IOException`, so each needs handling.

Below, the second is wrapped in an unchecked `RuntimeException` rather than being declared. See java-exceptions[3].

```
BufferedReader reader = null;
try {
    reader = new BufferedReader(new FileReader("readwithbuffer.txt"));
} catch (FileNotFoundException e) {
    System.err.println("File not found: " + e.getMessage());
}

for (int i = 0; i < 10; i++) {
    try {
        System.out.println(reader.readLine());
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```

Catching and continuing leaves `reader` `null` when the file is missing, so the loop then throws a `NullPointerException`. `try`-with-resources — as used for streams above — would both scope the reader correctly and close it.

`PrintWriter`

`System.out` is a `PrintStream`; `PrintWriter` is another (better) option for character output — it can write to many destinations, and it can write **formatted** text. You can construct one straight from `System.out`:

```
PrintWriter printWriter = new PrintWriter(System.out);
printWriter.println("I love CSSE2002");
printWriter.printf("Formatted number: %.2f%n", 123.45336);
printWriter.flush();
```

Or stack it on a `FileWriter` inside a `try`-with-resources, closing both:

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

[3] courses/csse2002/java-exceptions.pdf

```
    e.printStackTrace();
}
```

`writeroutput.txt` then holds:

```
I love CSSE2002
Formatted number: 123.45
```

`flush()`

If an `OutputStream`/`Writer` is buffered, output might not be sent immediately — `flush()` will send any pending output. This matters for:

- **"Interactive" or other situations where you expect a response** — the other end won't respond if you haven't actually sent anything yet.
- **Debugging or logging**, where you need an up-to-date view of what is happening.

`close()`-ing a stream will flush 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 and strings, as a more user-friendly alternative to `BufferedReader`. It allows reading of primitive Java types from the stream, and `regex` scanning is possible for advanced use cases.

Because it accepts either, the same logic can be handed a `Reader`:

```
public class ScannerEx {
    public static void main(String[] args) {
        readInts(new InputStreamReader(System.in));
    }

    private static void readInts(Reader reader) {
        Scanner scanner = new Scanner(reader);
        int total = 0;
        while (scanner.hasNextInt()) {
            int newInt = scanner.nextInt();
            total += newInt;
        }
        System.out.println(total);
    }
}
```

or given `System.in` directly — in which case the `Scanner` internally creates its own `InputStreamReader`:

```
Scanner scanner = new Scanner(System.in);
int total = 0;
while (scanner.hasNextInt()) {
    total += scanner.nextInt();
}
System.out.println(total);
```

Reading strings:

- `scanner.next()` — reads the next **word**, up to whitespace, skipping any leading whitespace.
- `scanner.nextLine()` — reads the **entire line**, until Enter is pressed.

```
Scanner scanner = new Scanner(System.in);
System.out.println("Enter three words:");
String word1 = scanner.next();   // reads up to first space
String word2 = scanner.next();
String word3 = scanner.next();
System.out.println("You entered: " + word1 + ", " + word2 + ", " + word3);
```

Further reading: Scanner vs. BufferedReader.

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

The problem with the older `java.io`: it focused mainly on streams (byte and char streams), because the main goal was **reading and writing data, not managing files or directories**. That leaves it with limited support for file-system operations, and makes it inconvenient for file attributes, directory traversal, and path manipulation.

`java.nio.file` (since Java 1.7) introduces two new abstractions:

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

Creating a `Path`

`Path` is the key abstraction for locating files, and can be created from path strings or from a URI:

```
Path p1 = Path.of("docs/output.txt");        // the whole path as one string
Path p2 = Path.of("docs", "output.txt");     // separate name elements, which Java
 ↪  joins into a path
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)`
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.

```
public static void main(String[] args) throws IOException {
    Path myPath = Path.of("src", "Week08", "NIO", "myfile.txt");
    Files.createFile(myPath);
}
```

Reading a whole file is then a one-liner — no stream, no wrapper, no `close()`:

```java
public static void main(String[] args) throws IOException {
    Path myPath = Path.of("src", "Week08", "NIO", "myfile.txt");

    List<String> lines = Files.readAllLines(myPath);
    for (String line : lines) {
        System.out.println(line);
    }
}
```

Exercise: rewrite the earlier stream and reader exercises using New I/O.

Summary: which to use?

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

Saving and loading a whole object model to a file is picked up later, in week10-lab-java-io[4].

[4] courses/csse2002/week10-lab-java-io.pdf