**Using Scanner (simple text reading)**

**Best for:** small text files, line-by-line input.

import java.io.\*;

import java.util.\*;

public class ReadUsingScanner {

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

// Create and write a sample file

File file = new File("sample.txt");

try (FileWriter writer = new FileWriter(file)) {

writer.write("Hello Java\nThis is a Scanner example");

}

// Read file using Scanner

try (Scanner sc = new Scanner(file)) {

while (sc.hasNextLine()) {

System.out.println(sc.nextLine());

}

}

}

}

✅ **Notes:**

* Scanner automatically tokenizes input by line or space.
* Slower for large files.
* No need for explicit buffering.

**✍️ 2. Using FileReader and FileWriter**

**Best for:** simple character stream reading/writing (text files).

import java.io.\*;

public class ReadUsingFileReader {

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

// Create a file and write content

try (FileWriter writer = new FileWriter("fileReader.txt")) {

writer.write("Learning FileReader and FileWriter in Java");

}

// Read using FileReader

try (FileReader reader = new FileReader("fileReader.txt")) {

int ch;

while ((ch = reader.read()) != -1) {

System.out.print((char) ch);

}

}

}

}

✅ **Notes:**

* Works with character data (UTF-16).
* Doesn’t handle binary files.
* Should usually be wrapped in BufferedReader for efficiency.

**⚡ 3. Using BufferedReader and BufferedWriter**

**Best for:** efficient reading/writing of large text files.

import java.io.\*;

public class ReadUsingBufferedReader {

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

// Write using BufferedWriter

try (BufferedWriter writer = new BufferedWriter(new FileWriter("buffered.txt"))) {

writer.write("BufferedReader is faster.\nIt uses internal buffer.");

}

// Read using BufferedReader

try (BufferedReader br = new BufferedReader(new FileReader("buffered.txt"))) {

String line;

while ((line = br.readLine()) != null) {

System.out.println(line);

}

}

}

}

✅ **Notes:**

* Uses internal buffer (default 8 KB).
* Perfect for large files.
* Reads line-by-line efficiently.

**💾 4. Using FileInputStream and FileOutputStream**

**Best for:** binary data (images, PDFs, etc.), but works with text too.

import java.io.\*;

public class ReadUsingFileInputStream {

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

// Write file using FileOutputStream

try (FileOutputStream fos = new FileOutputStream("fileStream.txt")) {

String content = "Reading with FileInputStream!";

fos.write(content.getBytes());

}

// Read file using FileInputStream

try (FileInputStream fis = new FileInputStream("fileStream.txt")) {

int b;

while ((b = fis.read()) != -1) {

System.out.print((char) b);

}

}

}

}

✅ **Notes:**

* Works at byte level.
* Best for non-text files.
* Wrap with BufferedInputStream for speed.

**🚀 5. Using BufferedInputStream and BufferedOutputStream**

**Best for:** buffered binary I/O, improves performance significantly.

import java.io.\*;

public class ReadUsingBufferedStream {

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

// Write binary data

try (BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream("bufferedStream.txt"))) {

bos.write("BufferedInputStream example".getBytes());

}

// Read binary data

try (BufferedInputStream bis = new BufferedInputStream(new FileInputStream("bufferedStream.txt"))) {

int data;

while ((data = bis.read()) != -1) {

System.out.print((char) data);

}

}

}

}

✅ **Notes:**

* Ideal for large binary or mixed I/O.
* Reads/writes chunks instead of byte-by-byte.

**⚙️ Bonus: Using Files class (Java 7+)**

import java.io.\*;

import java.nio.file.\*;

import java.util.\*;

public class ReadUsingFiles {

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

Path path = Paths.get("nio.txt");

// Write

Files.write(path, Arrays.asList("Line 1", "Line 2", "Line 3"));

// Read all lines

List<String> lines = Files.readAllLines(path);

lines.forEach(System.out::println);

}

}

✅ **Notes:**

* Easiest for modern Java.
* Uses NIO.2 API.
* Perfect for one-liner file operations.

**🧠 Summary Table**

| **Method** | **Stream Type** | **Use Case** | **Performance** |
| --- | --- | --- | --- |
| Scanner | Text | Small, simple reading | Low |
| FileReader/Writer | Character | Basic file ops | Medium |
| BufferedReader/Writer | Character | Large text files | High |
| FileInput/OutputStream | Byte | Binary data | Medium |
| BufferedInput/OutputStream | Byte | Buffered binary I/O | High |
| Files API | NIO.2 | Modern, quick ops | High |