File Reading

The fifth topic in the Java Basics category (the reading half of File I/O): `Path` / `Files` basics, `Files.readAllLines()` / `Files.readString()`, line-by-line reading with `BufferedReader`, `Files.lines()` being lazy and Closeable, searching a file for a word, and the real difference between `NoSuchFileException` and `FileNotFoundException`.

Beginner 20 min

Last updated: October 31, 2026
Read online: https://www.learnforgex.com/en/topics/file-reading

---

File Reading

File I/O is the sixth topic in the Java Basics category -- split into two topics: this one covers READING (`File Reading`), and the next one covers WRITING (`File Writing`). This split is the reverse of the "combine short, independent sub-topics into a SINGLE topic" practice used in the `functional-interfaces-streams` and `collections` categories -- here, a single subject (File I/O) turned out to be BROAD enough that it was SPLIT into two topics instead.

What Is File I/O?

File I/O (file input/output) is a program interacting with files on disk -- reading, writing, copying, deleting. Java uses TWO different APIs together for this: the older `java.io` package (STREAM-based classes like `FileReader`, `BufferedReader`, `FileWriter`, `BufferedWriter`) and the modern `java.nio.file` package (PATH-based classes like `Path`, `Files`, which reduce most operations to single-line static methods).

Why Does It Exist?

A program that can't read/write persistent data isn't very useful in the real world -- log files, configuration files, CSV reports, user-uploaded documents, all of it requires File I/O. `java.nio.file` (NIO.2) was designed to fix some of the older `java.io`'s annoyances (the complexity of checked-exception handling, lack of symbolic link support, difficulty accessing filesystem metadata) -- but some `java.io` classes like `BufferedReader` are still widely used, especially when line-by-line processing is needed.

History

The `java.io` package has been part of Java since version 1.0 (1996) -- the classic stream-based model. `java.nio` (Non-blocking I/O) arrived in Java 1.4 (2002), but the actual filesystem API, `java.nio.file` (`Path` / `Files`, also known as "NIO.2"), was added in Java 7 (2011) -- offering a more readable API, real exception types (like `NoSuchFileException`), and static methods that directly support filesystem operations (copying, moving, symbolic links). Java 11 (2018) further simplified reading/writing an entire file as a single `String` with `Files.readString()` / `Files.writeString()`.

Path and Files Basics

`Path.of(...)` creates an object that REPRESENTS a file location -- but it does NOT touch the FILESYSTEM, it's just an "address". Methods like `Files.exists()` actually check the filesystem. `Files.readAllLines(path)` reads the entire file into memory and returns a `List<String>` with each line as an element -- the simplest way to read a small-to-medium text file.

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.List;

public class PathAndFilesBasicsExample {
    public static void main(String[] args) throws IOException {
        // Path.of() builds a platform-independent file path -- it does NOT
        // touch the filesystem by itself, it's just a representation of a
        // location.
        Path path = Path.of("demo-input.txt");

        // Files.exists() actually checks the filesystem.
        System.out.println("Exists before creating it? " + Files.exists(path));

        // Files.writeString() creates the file (see the "File Writing" lesson
        // for the full picture) -- used here just to set up something to read.
        Files.writeString(path, "Hello World\nJava is great\nPractice makes perfect\nHello
again\nJava rocks!");
        System.out.println("Exists after creating it? " + Files.exists(path));

        // Files.readAllLines() reads the ENTIRE file into memory as a
        // List<String>, one element per line -- the simplest way to read a
        // small-to-medium text file.
        List<String> lines = Files.readAllLines(path);
        System.out.println("Number of lines read: " + lines.size());
        lines.forEach(System.out::println);

        // Path carries useful metadata methods too.
        System.out.println("getFileName(): " + path.getFileName());
        System.out.println("toAbsolutePath(): " + path.toAbsolutePath());

        Files.deleteIfExists(path);
    }
}
```

Reading Line by Line with BufferedReader

`BufferedReader` is the classic `java.io` way -- it WRAPS a `FileReader` and BUFFERS reads internally, which is much faster than reading one character at a time. `readLine()` returns `null` exactly ONCE, when there's nothing left to read -- that's the loop's natural termination condition. `BufferedReader` holds a real file handle, so it MUST be used INSIDE try-with-resources.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;

public class BufferedReaderExample {
    public static void main(String[] args) throws IOException {
        Path path = Path.of("demo-buffered.txt");
        Files.writeString(path, "line one\nline two\nline three");

        // BufferedReader is the classic java.io way to read a file line by
        // line -- it wraps a FileReader and buffers reads internally, which is
        // much faster than reading one character at a time.
        //
        // try-with-resources guarantees close() is called even if an
        // exception is thrown mid-read -- this matters because BufferedReader
        // holds a real file handle open until it's closed.
        try (BufferedReader reader = new BufferedReader(new FileReader(path.toFile()))) {
            String line;
            int lineNumber = 1;
            // readLine() returns null exactly once, when there's nothing left
            // to read -- that's the loop's natural termination condition.
            while ((line = reader.readLine()) != null) {
                System.out.println(lineNumber + ": " + line);
                lineNumber++;
            }
        }

        Files.deleteIfExists(path);
    }
}
```

Expected Output

```
1: line one
2: line two
3: line three
```

The pattern `while ((line = reader.readLine()) != null) { ... }` combines the assignment AND the comparison in a single expression -- a common, idiomatic reading loop pattern in Java.

Counting Lines: readAllLines() vs Files.lines()

There are two ways to find the number of lines in a file: `Files.readAllLines(path).size()` (loads the entire file into memory, fine for small files) or `Files.lines(path).count()` (LAZY -- returns a `Stream<String>` that reads without loading the ENTIRE file into memory at once, scalable for very large files).

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.stream.Stream;

public class FileReadingStreamAndCountExample {
    public static void main(String[] args) throws IOException {
        Path path = Path.of("demo-count.txt");
        Files.writeString(path, "one\ntwo\nthree\nfour\nfive");

        // Simplest way to count lines: read everything into a List, then ask
        // its size. Fine for small files, but it loads the WHOLE file into
        // memory just to get a count.
        long countViaReadAllLines = Files.readAllLines(path).size();
        System.out.println("Count via Files.readAllLines().size(): " +
countViaReadAllLines);

        // Files.lines() returns a LAZY Stream<String> -- it doesn't load the
        // whole file at once, it reads as the stream is consumed. This scales
        // to files far larger than available memory.
        //
        // CRITICAL: Files.lines() opens a real file handle under the hood, so
        // the Stream it returns is Closeable and MUST be used inside
        // try-with-resources -- forgetting to close it leaks a file handle,
        // exactly like forgetting to close a Scanner or BufferedReader.
        long countViaLines;
        try (Stream<String> lineStream = Files.lines(path)) {
            countViaLines = lineStream.count();
        }
        System.out.println("Count via Files.lines().count() (lazy, must be closed): " +
countViaLines);

        Files.deleteIfExists(path);
    }
}
```

Expected Output

```
Count via Files.readAllLines().size(): 5
Count via Files.lines().count() (lazy, must be closed): 5
```

> The `Stream<String>` returned by `Files.lines()` holds a real file handle underneath -- meaning it needs to be CLOSED (`close()`), just like a `Scanner` or `BufferedReader`. Using it WITHOUT try-with-resources (a one-liner like `Files.lines(path).count()`) leaks the file handle -- forgetting that `Stream` is `Closeable` is a common mistake.

Searching a File for a Word

Combining `Files.readAllLines()` with the Stream API (see the "Stream API Fundamentals" lesson) reduces searching a file for a keyword to a single line: FILTER the lines, keep only the ones containing the word.

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.List;

public class SearchWordInFileExample {
    public static void main(String[] args) throws IOException {
        Path path = Path.of("demo-search.txt");
        Files.writeString(path,
                "Hello World\nJava is great\nPractice makes perfect\nHello again\nJava
rocks!");

        // Combining Files.readAllLines() with the Stream API (see the "Stream
        // Fundamentals" lesson) makes searching a file for a keyword a
        // one-liner: filter the lines, keep only the ones that contain the
        // word.
        List<String> matches = Files.readAllLines(path).stream()
                .filter(line -> line.contains("Java"))
                .toList();

        System.out.println("Lines containing \"Java\":");
        matches.forEach(line -> System.out.println("  " + line));
        System.out.println("Match count: " + matches.size());

        // Case-insensitive search is a small variation -- lowercase both
        // sides before comparing.
        List<String> caseInsensitiveMatches = Files.readAllLines(path).stream()
                .filter(line -> line.toLowerCase().contains("hello"))
                .toList();
        System.out.println("Lines containing \"hello\" (case-insensitive): " +
caseInsensitiveMatches);

        Files.deleteIfExists(path);
    }
}
```

Expected Output

```
Lines containing "Java":
  Java is great
  Java rocks!
Match count: 2
Lines containing "hello" (case-insensitive): [Hello World, Hello again]
```

Reading the Entire File as a String

`Files.readString()` (Java 11+) reads the ENTIRE file into a single `String` -- line separators INCLUDED. Unlike `Files.readAllLines()` (which strips line separators and returns a `List`), `readString()` is the better fit when you need the raw text itself (for example, to pass to a JSON parser or display as-is).

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;

public class ReadFileAsStringExample {
    public static void main(String[] args) throws IOException {
        Path path = Path.of("demo-whole-file.txt");
        Files.writeString(path, "Line A\nLine B\nLine C");

        // Files.readString() (Java 11+) reads the ENTIRE file into a single
        // String, newlines and all -- the simplest option when you need the
        // raw text as one value (for example, to pass to a JSON parser or
        // display as-is), rather than a List<String> of separate lines.
        String wholeFile = Files.readString(path);
        System.out.println("Whole file as one String:");
        System.out.println(wholeFile);
        System.out.println("Total length: " + wholeFile.length() + " characters");

        // Files.readAllLines() vs Files.readString(): readAllLines() strips
        // the line separators and gives you a List; readString() keeps them
        // and gives you one String. Pick based on whether you need to work
        // line by line or need the raw text.
        System.out.println();
        System.out.println("Contains newline characters? " + wholeFile.contains("\n"));

        Files.deleteIfExists(path);
    }
}
```

Expected Output

```
Whole file as one String:
Line A
Line B
Line C
Total length: 20 characters

Contains newline characters? true
```

Exception Handling: NoSuchFileException vs FileNotFoundException

In the modern `java.nio.file` API (`Files.readString()`, `Files.readAllLines()`, etc.), a missing file throws `NoSuchFileException` -- NOT the classic `java.io`'s `FileNotFoundException`. The two are UNRELATED sibling exception classes, even though both extend `IOException`. `FileNotFoundException` comes from classic `java.io` classes like `FileReader`/`FileInputStream`.

```
import java.io.FileNotFoundException;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.NoSuchFileException;
import java.nio.file.Path;
```

```
public class FileReadingExceptionHandlingExample {
    public static void main(String[] args) {
        Path missing = Path.of("this-file-does-not-exist.txt");

        // SURPRISE: for the modern java.nio.file API (Files.readString(),
        // Files.readAllLines(), etc.), a missing file throws
        // NoSuchFileException, NOT the classic java.io FileNotFoundException.
        // The two are UNRELATED sibling exceptions (both extend IOException),
        // even though they mean the same thing in practice.
        try {
            Files.readString(missing);
        } catch (NoSuchFileException e) {
            System.out.println("Caught NoSuchFileException (the java.nio.file exception): "
+ e.getFile());
        } catch (IOException e) {
            System.out.println("unreachable for this case: " + e.getClass().
getSimpleName());
        }

        // A multi-catch of BOTH exception types still compiles and is
        // harmless -- but for a Files.* call, the FileNotFoundException
        // branch will never actually fire, because Files.* never throws it.
        try {
            String content = Files.readString(missing);
            System.out.println("unreachable: " + content);
        } catch (FileNotFoundException e) {
            System.out.println("unreachable: FileNotFoundException never comes from Files.
*");
        } catch (NoSuchFileException e) {
            System.out.println("This branch fires instead, for the SAME missing-file case");
        } catch (IOException e) {
            System.out.println("unreachable: " + e.getClass().getSimpleName());
        }

        // FileNotFoundException DOES come from the legacy java.io classes,
        // like the FileReader used in the "BufferedReader" section.
        try {
            new java.io.FileReader(missing.toFile());
        } catch (FileNotFoundException e) {
            System.out.println("Caught FileNotFoundException (the classic java.io
exception, from FileReader)");
        }

        // The safe, general pattern: catch IOException as a fallback for
        // anything that isn't specifically handled above.
        String result = readSafely(missing);
        System.out.println("Safe helper result: " + result);
    }

    private static String readSafely(Path path) {
        try {
            return Files.readString(path);
        } catch (NoSuchFileException e) {
            return "File not found: " + path;
        } catch (IOException e) {
            return "Error reading file: " + e.getMessage();
        }
```

```
    }
}
```

> Writing `catch (FileNotFoundException | NoSuchFileException e)` for `Files.readString()` / `Files.readAllLines()` COMPILES, but the `FileNotFoundException` branch never actually FIRES for that call -- because `Files.*` methods never throw it, only `NoSuchFileException`. It matters to catch the CORRECT exception type based on which API (java.io or java.nio.file) you're using; when in doubt, catching the general `IOException` is always safe.

Best Practices

- **Use `Files.readAllLines()` / `Files.readString()` for small-to-medium files** -- they offer a simple, readable, one-line API; prefer `Files.lines()` (INSIDE try-with-resources) for very large files.
- **Use every resource that holds a file handle (`BufferedReader`, `Files.lines()`, etc.) INSIDE try-with-resources** -- forgetting to close it leads to a resource leak.
- **Catch `NoSuchFileException` when using the `java.nio.file` API, NOT `FileNotFoundException`** -- catching the wrong exception type leads to a silent bug, since that branch never fires.
- **Use the `Files.readAllLines().stream().filter(...)` pattern when searching /filtering a file** -- readable, and leverages the power of the Stream API.

Common Mistakes

- **Catching `FileNotFoundException` for `Files.readString()` / `Files.readAllLines()` and not noticing it never fires.** These APIs throw `NoSuchFileException` -- the correct exception type needs to be caught.
- **Using `Files.lines()` without try-with-resources.** The `Stream` it returns holds a real file handle underneath -- if not closed, the resource leaks.
- **Trying to read a very large file with `Files.readAllLines()` and running into an out-of-memory situation.** `Files.lines()` (lazy) should be preferred for large files.
- **Forgetting to close a `BufferedReader`.** Without try-with-resources, its underlying file handle stays open.

Summary, Cheat Sheet, and Glossary

Java has two APIs for reading files: the classic `java.io` (`BufferedReader` + `FileReader`, for line-by-line reading) and the modern `java.nio.file` (`Path` + `Files`, reducing most operations to a single line with `readAllLines()` / `readString()` / `lines()`). `Files.lines()` is

LAZY and `Closeable` -- it requires try-with-resources. For missing files, the modern API throws `NoSuchFileException`, the classic API throws `FileNotFoundException` -- these are UNRELATED classes.

Quick reference:

```
Path path = Path.of("data.txt");                            // build a path (doesn't touch
the file)
List<String> lines = Files.readAllLines(path);                 // read the whole file as a
List
String content = Files.readString(path);                         // read the whole file as
one String

try (Stream<String> s = Files.lines(path)) {                       // LAZY, try-with-
resources REQUIRED
    long count = s.count();
}

try (BufferedReader r = new BufferedReader(new FileReader(path.toFile()))) {  // classic
line-by-line reading
    String line;
    while ((line = r.readLine()) != null) { ... }
}

try { Files.readString(path); }
catch (NoSuchFileException e) { ... }                            // java.nio.file --
CORRECT exception
```

Glossary

Path — An object that represents a file location without touching the filesystem (`java.nio.file.Path`).

Files — The `java.nio.file` package's class offering static helper methods for file operations.

BufferedReader — A classic `java.io` class that wraps a reading source (e.g. `FileReader`) and buffers reads.

NoSuchFileException — The exception the `java.nio.file` API throws for a missing file (a DIFFERENT class from the classic `FileNotFoundException`).

Try-with-Resources — Java syntax that guarantees a resource (like a file handle) is automatically closed at the end of a block.