File Writing

The sixth topic in the Java Basics category (the writing half of File I/O): `Files.writeString()` / `Files.write()`, appending with `StandardOpenOption.APPEND`, writing with `BufferedWriter`, copying files with `Files.copy()`, `Files.createDirectories()`, the standard pattern for deleting a directory tree, and writing a CSV file.

Beginner 20 min

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

---

File Writing

File Writing is the seventh topic in the Java Basics category, the WRITING half of File I/O -- a direct continuation of the "File Reading" lesson. That lesson covered the two ways to read files (`java.nio.file.Files` and the classic `java.io.BufferedReader`); here we cover WRITING to files, COPYING files, and managing directories with the same two API families.

What Is File Writing?

Writing to a file is the mirror image of the reading APIs from the "File Reading" lesson: `Files.writeString()` / `Files.write()` (modern, `java.nio.file`) and `BufferedWriter` + `FileWriter` (classic, `java.io`, with try-with-resources). This lesson also goes beyond writing to a single file, covering file COPYING (`Files.copy()`) and directory CREATION/DELETION (`Files.createDirectories()`, `Files.walk()`).

Why Does It Exist?

A program making its results PERSISTENT -- producing a report, keeping a log file, saving a configuration, writing a user's upload to disk -- requires File Writing. The modern `Files.writeString()` / `Files.write()` methods reduce the older `java.io` ritual of "open a stream → write → close the stream" to a SINGLE static method call; but classic classes like `BufferedWriter` are still preferred, especially when many small write operations need to be accumulated EFFICIENTLY.

History

The SAME history from the "File Reading" lesson applies here: `java.io` (`FileWriter`/`BufferedWriter`) has existed since Java 1.0 (1996), `java.nio.file` (`Path`/`Files`) arrived in Java 7 (2011). `Files.writeString()` was added in Java 11 (2018) -- before that, writing a single `String` to a file required a conversion step, like `Files.write(path, content.getBytes())`.

Files.writeString(): Creating and Overwriting

`Files.writeString(path, content)` is the simplest way to write -- it CREATES the file if it doesn't exist, and OVERWRITES it entirely if it does (overwriting, not accumulating). Calling it twice in a row on the same file leaves only the SECOND call's content.

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

public class WriteStringExample {
    public static void main(String[] args) throws IOException {
        Path path = Path.of("demo-write.txt");

        // Files.writeString() is the simplest way to write text to a file --
        // it CREATES the file if it doesn't exist, or OVERWRITES it entirely
        // if it does. There is no accumulation here: calling it twice in a row
        // leaves only the SECOND call's content.
        Files.writeString(path, "Hello, File!");
        System.out.println("After first write: " + Files.readString(path));

        Files.writeString(path, "This REPLACES the previous content entirely.");
        System.out.println("After second write (overwritten): " + Files.readString(path));

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

Expected Output

```
After first write: Hello, File!
After second write (overwritten): This REPLACES the previous content entirely.
```

Appending to a File

To add to the END of existing content instead of the default overwriting behavior, `StandardOpenOption.APPEND` is passed. A critical point: `APPEND` ALONE assumes the file ALREADY EXISTS -- trying to write with only `APPEND` to a file that doesn't exist yet throws `NoSuchFileException`; if you want to both create the file AND append, `StandardOpenOption.CREATE` must be passed TOGETHER with it.

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

public class AppendToFileExample {
    public static void main(String[] args) throws IOException {
        Path path = Path.of("demo-append.txt");
        Files.writeString(path, "First line");

        // By default Files.writeString() OVERWRITES -- passing
        // StandardOpenOption.APPEND changes that behavior to add onto the
        // END of the existing content instead of replacing it.
        Files.writeString(path, "\nSecond line", StandardOpenOption.APPEND);
        Files.writeString(path, "\nThird line", StandardOpenOption.APPEND);

        System.out.println("Final content after two appends:");
        System.out.println(Files.readString(path));

        // Trying to append to a file that doesn't exist yet fails, UNLESS you
        // also pass StandardOpenOption.CREATE -- APPEND alone assumes the file
        // is already there.
        Path newPath = Path.of("demo-append-new.txt");
        Files.writeString(newPath, "Created via CREATE + APPEND",
                StandardOpenOption.CREATE, StandardOpenOption.APPEND);
        System.out.println("New file via CREATE + APPEND: " + Files.readString(newPath));

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

Expected Output

```
Final content after two appends:
First line
Second line
Third line
New file via CREATE + APPEND: Created via CREATE + APPEND
```

Calling `Files.writeString(path, content, StandardOpenOption.APPEND)` for a file that doesn't exist yet throws `NoSuchFileException` -- `APPEND` does NOT AUTOMATICALLY include `CREATE`. Passing both together (`StandardOpenOption.CREATE, StandardOpenOption.APPEND`) works safely whether the file exists or not.

Writing a List Line by Line

`Files.write(path, list)` takes a `List<String>` and writes each element on ITS OWN LINE -- no need to manually add line separators (`String.join("\n", list)`).

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

public class WriteLinesExample {
    public static void main(String[] args) throws IOException {
        Path path = Path.of("demo-lines.txt");

        // Files.write(path, list) takes a List<String> and writes each
        // element as its own line -- the line-separator handling is done for
        // you, unlike manually joining with "\n".
        List<String> fruits = List.of("Apple", "Banana", "Cherry");
        Files.write(path, fruits);

        System.out.println("File content:");
        System.out.println(Files.readString(path));

        System.out.println("Read back as a List:");
        Files.readAllLines(path).forEach(line -> System.out.println("  " + line));

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

Expected Output

```
File content:
Apple
Banana
Cherry

Read back as a List:
  Apple
  Banana
  Cherry
```

Writing with BufferedWriter

`BufferedWriter` is the writing counterpart to `BufferedReader` (see the "File Reading" lesson) -- it wraps a `FileWriter`, buffering writes internally. `write()` does NOT add a line separator BY ITSELF -- `newLine()` must be called explicitly for that (it uses the platform's correct separator: `\n` on Linux/macOS, `\r\n` on Windows).

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

public class BufferedWriterExample {
    public static void main(String[] args) throws IOException {
        Path path = Path.of("demo-buffered-write.txt");

        // BufferedWriter is the classic java.io counterpart to
        // BufferedReader -- it wraps a FileWriter and buffers writes
        // internally. write() does NOT add a line separator by itself;
        // newLine() does that explicitly (and uses the platform's correct
        // separator, "\n" on Linux/macOS, "\r\n" on Windows).
        try (BufferedWriter writer = new BufferedWriter(new FileWriter(path.toFile()))) {
            writer.write("Line 1");
            writer.newLine();
            writer.write("Line 2");
            writer.newLine();
            writer.write("Line 3");
            // No trailing newLine() here -- the file ends right after "Line 3".
        }

        System.out.println("File content:");
        System.out.println(Files.readString(path));

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

Expected Output

```
File content:
Line 1
Line 2
Line 3
```

Whether to call `newLine()` after the last line is YOUR decision -- in the example above, `newLine()` was NOT called after the third line, so the file ends right after `"Line 3"`, without a trailing line separator.

File Copying and Directory Management

`Files.createDirectories()` creates a directory AND any missing parent directories along the way (like `mkdir -p`) -- it does NOT fail if the directory already exists. `Files.copy()` copies a file's content in a single call -- but by DEFAULT it throws `FileAlreadyExistsException` if the destination already exists; `StandardCopyOption.REPLACE_EXISTING` makes it overwrite instead. To DELETE an entire directory tree, you first need to WALK it (`Files.walk()`) and delete starting from the DEEPEST entries (files, then subdirectories) -- a non-empty directory can't be deleted directly.

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

public class CopyAndDirectoryExample {
    public static void main(String[] args) throws IOException {
        // Files.createDirectories() creates a directory AND any missing
        // parent directories along the way (like "mkdir -p") -- it does
        // NOT fail if the directory already exists.
        Path baseDir = Files.createDirectories(Path.of("demo-output"));
        System.out.println("Created directory: " + baseDir.getFileName());

        Path original = baseDir.resolve("original.txt");
        Files.writeString(original, "Content to be copied.");

        Path copy = baseDir.resolve("copy.txt");
        // Files.copy() copies a file's content in one call -- but by default
        // it throws FileAlreadyExistsException if the destination already
        // exists. StandardCopyOption.REPLACE_EXISTING makes it overwrite
        // instead.
        Files.copy(original, copy, StandardCopyOption.REPLACE_EXISTING);
        System.out.println("Copied content: " + Files.readString(copy));

        // Copying again to the SAME destination without REPLACE_EXISTING
        // would throw -- demonstrating why the option matters.
        try {
            Files.copy(original, copy); // no REPLACE_EXISTING this time
            System.out.println("unreachable");
        } catch (java.nio.file.FileAlreadyExistsException e) {
            System.out.println("Caught FileAlreadyExistsException without
REPLACE_EXISTING");
        }

        // Cleaning up an entire directory tree requires walking it and
        // deleting from the DEEPEST entries first -- you can't delete a
        // non-empty directory. Files.walk() + Comparator.reverseOrder()
        // (files/subdirectories before their parent) is the standard pattern
        // for this.
        try (var paths = Files.walk(baseDir)) {
            paths.sorted(Comparator.reverseOrder())
                    .forEach(p -> {
                        try {
                            Files.delete(p);
                        } catch (IOException e) {
                            System.out.println("Failed to delete: " + p);
                        }
                    });
        }
        System.out.println("Directory exists after cleanup? " + Files.exists(baseDir));
    }
}
```

> The pattern `Files.walk(dir).sorted(Comparator.reverseOrder()).forEach(Files::delete)` is the STANDARD way to delete a directory tree -- `reverseOrder()` is CRITICAL, because a directory can't be deleted before the files/subdirectories INSIDE it are deleted. Trying to delete without `Comparator.reverseOrder()` (in natural, shallow-to-deep order) throws an exception for non-empty directories.

Writing a CSV File

Writing a structured text file (like CSV) combines string-joining basics (see the "String" lesson) with File I/O: `String.join(",", array)` builds each line, all lines are gathered in a `StringBuilder`, and the whole thing is written to disk with a SINGLE `Files.writeString()` call -- more EFFICIENT than making a separate write call for every line.

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

public class WriteCsvExample {
    public static void main(String[] args) throws IOException {
        Path path = Path.of("demo-data.csv");

        String[] headers = {"Name", "Age", "City"};
        String[][] rows = {
                {"Alice", "30", "London"},
                {"Bob", "25", "Paris"},
                {"Charlie", "35", "Tokyo"}
        };

        writeCsv(path, headers, rows);

        System.out.println("CSV file content:");
        System.out.println(Files.readString(path));

        Files.deleteIfExists(path);
    }

    private static void writeCsv(Path path, String[] headers, String[][] rows) throws
IOException {
        // String.join(",", array) turns an array into a single comma-joined
        // line -- the same helper used for the header row and every data row.
        StringBuilder csv = new StringBuilder();
        csv.append(String.join(",", headers)).append('\n');
        for (String[] row : rows) {
            csv.append(String.join(",", row)).append('\n');
        }

        // Writing the fully-built content in ONE Files.writeString() call is
        // more efficient than writing line by line -- fewer individual I/O
        // operations against the filesystem.
        Files.writeString(path, csv.toString());
    }
}
```

Expected Output

```
CSV file content:
Name,Age,City
Alice,30,London
Bob,25,Paris
Charlie,35,Tokyo
```

Best Practices

- **Use `Files.writeString()` / `Files.write()` for one-off, simple writes** -- short and readable; prefer `BufferedWriter` if you're accumulating many small writes.
- **When using `APPEND` and you're not sure whether the file exists, pass `StandardOpenOption.CREATE` together with it** -- `APPEND` alone throws `NoSuchFileException` if the file doesn't exist.
- **Always add `StandardCopyOption.REPLACE_EXISTING` when calling `Files.copy()` if you want the destination overwritten** -- otherwise you get an exception if the destination already exists.
- **Use the `Files.walk().sorted(Comparator.reverseOrder())` pattern when deleting a directory tree** -- trying to delete in natural order fails on non-empty directories.

Common Mistakes

- **Using `StandardOpenOption.APPEND` on a file that doesn't exist yet and getting `NoSuchFileException`.** `CREATE` needs to be passed together with it.
- **Calling `Files.copy()` without `REPLACE_EXISTING` and getting `FileAlreadyExistsException` when the destination already exists.** This option should be added if overwriting is intended.
- **Trying to delete a directory tree in natural order (shallow-to-deep) and getting a non-empty-directory error.** Deletion should go from deep to shallow with `Comparator.reverseOrder()`.
- **Assuming `BufferedWriter.write()` automatically adds a line separator.** `write()` only writes the text -- `newLine()` must be called separately for a line break.

Summary, Cheat Sheet, and Glossary

There are also two APIs for writing files: the modern `Files.writeString()` / `Files.write()` (one line, customizable behavior via `StandardOpenOption`) and the classic `BufferedWriter` + `FileWriter` (line-by-line writing, explicit line-break control with `newLine()`). `Files.copy()` copies files (may need `REPLACE_EXISTING`), `Files.createDirectories()` creates directories, and deleting a directory tree requires `Files.walk()` plus reverse sorting.

Quick reference:

```
Files.writeString(path, "content");                                  // create/overwrite
Files.writeString(path, "\nmore", StandardOpenOption.APPEND);         // append (file must
EXIST)
Files.writeString(path, "content",
        StandardOpenOption.CREATE, StandardOpenOption.APPEND);          // create if
missing + append
Files.write(path, listOfStrings);                                        // List<String> -
> line by line

try (BufferedWriter w = new BufferedWriter(new FileWriter(path.toFile()))) {  // classic
writing
    w.write("line");
    w.newLine();                                                                 // EXPLICIT
line break
}

Files.createDirectories(dirPath);                                              // directory
(+ parents)
Files.copy(source, dest, StandardCopyOption.REPLACE_EXISTING);                 // copy,
overwrite

Files.walk(dirPath)
    .sorted(Comparator.reverseOrder())                                          // deep to
shallow
    .forEach(p -> { try { Files.delete(p); } catch (IOException e) {} });         // delete
the tree
```

Glossary

BufferedWriter — A classic `java.io` class that wraps a writing destination (e.g. `FileWriter`) and buffers writes.

StandardOpenOption — An enum that customizes `Files.writeString()` / `Files.write()` 's behavior (overwrite, append, create) (`APPEND` , `CREATE` , etc.).

StandardCopyOption — An enum that customizes `Files.copy()` 's behavior (e.g. `REPLACE_EXISTING`).

Files.walk() — A method that walks ALL files/subdirectories in a directory tree, returning a `Stream<Path>` .

FileAlreadyExistsException — The exception thrown when a method like `Files.copy()` tries to write to a destination that ALREADY EXISTS without permission to overwrite.