Module 11 — File I/O

ITP 120 — Java Programming I

Northern Virginia Community College | Randy Michak

> 💡 **Prerequisites**
>
> This module builds on everything from Modules 1–10. You need to be comfortable with variables, control flow, methods, arrays, classes, objects, and inheritance before jumping into File I/O. If anything feels shaky, review those modules first.

> 🎯 **Learning Objectives**
>
> By the end of this module you will be able to:
>
> - Explain why programs need to read and write files
> - Use the `File` class to inspect files and directories
> - Write data to text files using `PrintWriter`
> - Append data to existing files using `FileWriter`
> - Read data from text files using `Scanner`
> - Handle `FileNotFoundException` with try-catch
> - Use try-with-resources to close files automatically
> - Process file data: count lines, sum numbers, parse CSV
> - Build a complete file-processing program end to end

1. Why File I/O?

Every program you've written so far has one thing in common: when it stops running, all its data disappears. Variables live in RAM — and RAM is wiped clean the moment your program ends.

That's fine for small calculations, but most real programs need to **persist** data. A gradebook has to remember grades across sessions. A contact list needs to survive a reboot. A log file has to accumulate entries over time. That's what File I/O is for.

> 🧠 **Think of It This Way**
>
> RAM is a whiteboard — fast, convenient, but erased every night. A file is a notebook — slower to write in, but it's still there when you come back tomorrow.

Text Files vs. Binary Files

Type	What's stored	Examples	Human-readable?
Text files	Characters (Unicode/ ASCII)	.txt, .csv, .java, .html	Yes — open in Notepad
Binary files	Raw bytes	.jpg, .class, .exe, .zip	No — garbled in a text editor

This module focuses on **text files**. They're the easiest to work with in Java, and they're everywhere — config files, logs, data exports, and more.

2. The File Class

Before you read or write a file, Java gives you a way to represent it: the `File` class from `java.io`. A `File` object doesn't open the file — it's just a reference to a path on disk. Think of it as a pointer to a location.

```
import java.io.File;

public class FileDemo {
    public static void main(String[] args) {
        File f = new File("grades.txt");  // just a path reference

        System.out.println("Name:    " + f.getName());
        System.out.println("Path:    " + f.getPath());
        System.out.println("Exists:  " + f.exists());
        System.out.println("IsFile:  " + f.isFile());
        System.out.println("Size:    " + f.length() + " bytes");
    }
}
```

Useful File Methods

Method	Returns	What it tells you
`exists()`	boolean	Does this path exist on disk?
`getName()`	String	Just the filename, no directory path
`getPath()`	String	The path as you wrote it
`getAbsolutePath()`	String	Complete absolute path
`length()`	long	Size in bytes
`isFile()`	boolean	Is this a file (not a directory)?
`isDirectory()`	boolean	Is this a directory?
`canRead()`	boolean	Does your program have read permission?
`canWrite()`	boolean	Does your program have write permission?

> ⚠️ **Common Mistake**
>
> `new File("grades.txt")` does *not* create the file on disk. It creates a Java object representing a path. The actual file gets created when you open a `PrintWriter` or `FileWriter` on it. Always check `f.exists()` before reading.

3. Writing to Files: PrintWriter

`PrintWriter` is the easiest way to write text to a file. You already know `System.out.print()` and `System.out.println()` — `PrintWriter` works exactly the same way, just aimed at a file instead of the console.

```
import java.io.PrintWriter;
import java.io.FileNotFoundException;

public class WriteDemo {
    public static void main(String[] args) throws FileNotFoundException {
        PrintWriter pw = new PrintWriter("output.txt");

        pw.println("Alice: 92");
        pw.println("Bob: 87");
        pw.printf("Average: %.1f%n", 89.5);

        pw.close();  // IMPORTANT: flushes buffer and closes the file
        System.out.println("File written.");
    }
}
```

After running this, `output.txt` will contain:

```
Alice: 92
Bob: 87
Average: 89.5
```

PrintWriter Methods

Method	Does what
`print(value)`	Writes value, no newline
`println(value)`	Writes value + newline
`printf(format, args)`	Formatted output — same syntax as `System.out.printf`
`close()`	Flushes buffer and closes file
`flush()`	Flushes buffer without closing

> ⚠️ **Always Close Your File**
>
> If you forget to call `pw.close()`, some or all of the data may never make it to disk. Java buffers writes in memory and only flushes them on close. A file that looks empty usually means you forgot to close it.

> ⚠️ **PrintWriter Overwrites by Default**
>
> If `output.txt` already exists, `new PrintWriter("output.txt")` will **erase it** and start fresh. To add to an existing file, use `FileWriter` with the append flag (Section 4).

4. Appending to Files

Sometimes you don't want to overwrite — you want to add to the end of an existing file. Log files are the classic example: every time the program runs, you append a new entry without losing the old ones.

The trick is `FileWriter` with a second argument of `true`:

```
import java.io.PrintWriter;
import java.io.FileWriter;
import java.io.IOException;

public class AppendDemo {
    public static void main(String[] args) throws IOException {
        // true = open in append mode (don't overwrite)
        FileWriter fw = new FileWriter("log.txt", true);
        PrintWriter pw = new PrintWriter(fw);

        pw.println("Session started: " + new java.util.Date());
        pw.println("User logged in.");

        pw.close();
        System.out.println("Entry appended.");
    }
}
```

> 🧠 **Think of It This Way**
>
> `new PrintWriter("file.txt")` is like ripping out all the pages of a notebook and starting fresh. `new FileWriter("file.txt", true)` is like opening to the last page and writing a new line.

> 💡 **IOException vs FileNotFoundException**
>
> `FileWriter` throws `IOException` (a broader exception), not just `FileNotFoundException`. Your method needs `throws IOException` or a try-catch for `IOException`.

5. Reading from Files: Scanner

You've used `Scanner` to read keyboard input. The exact same class reads files — you just give it a `File` object instead of `System.in`.

```
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;

public class ReadDemo {
    public static void main(String[] args) throws FileNotFoundException {
        Scanner sc = new Scanner(new File("grades.txt"));

        while (sc.hasNextLine()) {
            String line = sc.nextLine();
            System.out.println(line);
        }

        sc.close();
    }
}
```

If `grades.txt` contains:

```
Alice: 92
Bob: 87
Carol: 95
```

Output:

```
Alice: 92
Bob: 87
Carol: 95
```

Scanner Methods for Files

Method	Returns	Use when
`hasNextLine()`	boolean	Checking if another full line exists
`nextLine()`	String	Reading one complete line
`hasNext()`	boolean	Checking if another token (word) exists

Method	Returns	Use when
`next()`	String	Reading one whitespace-delimited word
`hasNextInt()`	boolean	Checking if next token is an integer
`nextInt()`	int	Reading one integer
`hasNextDouble()`	boolean	Checking if next token is a double
`nextDouble()`	double	Reading one double
`close()`	void	Done reading — always close

> 💡 **hasNext() vs hasNextLine()**
>
> Use `hasNextLine()` when reading structured records — one full line at a time. Use `hasNext()` when scanning for individual tokens (words, numbers) regardless of line breaks.

> 🔧 **Try It**
>
> Create a file called `numbers.txt` with five integers, one per line. Write a program that reads each number using `Scanner` and prints the number doubled. Close the scanner when done.

6. Exception Handling for File I/O

File operations can fail in ways normal code can't. The file might not exist, permissions might block access, or the disk might be full. Java forces you to deal with this through **checked exceptions**.

The most common one for file reading is `FileNotFoundException`.

Option 1: throws (Pass It Up)

Add `throws FileNotFoundException` to your method signature. This passes responsibility to the caller:

```
public static void main(String[] args) throws FileNotFoundException {
    Scanner sc = new Scanner(new File("data.txt"));
    // ... read the file ...
}
```

Simple, but the program crashes with a stack trace if the file isn't found. Fine for learning. Not great for production.

Option 2: try-catch (Handle It Here)

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class SafeRead {
    public static void main(String[] args) {
        try {
            Scanner sc = new Scanner(new File("data.txt"));
            while (sc.hasNextLine()) {
                System.out.println(sc.nextLine());
            }
            sc.close();
        } catch (FileNotFoundException e) {
            System.out.println("Error: File not found.");
            System.out.println(e.getMessage());
        }
    }
}
```

If `data.txt` exists — output is file contents. If it doesn't:

```
Error: File not found.
data.txt (No such file or directory)
```

🧠 **Think of It This Way**

`throws` is like saying "not my problem — you deal with it." `try-catch` is like saying "I'll handle this right here." For files, try-catch is almost always the right choice in real programs because it lets you give the user a clear message.

⚠️ **Don't Swallow Exceptions**

An empty catch block hides errors silently and makes debugging a nightmare:

```
} catch (FileNotFoundException e) {
    // do nothing — BAD PRACTICE
}
```

At minimum, print the exception message so you know something went wrong.

7. Try-with-Resources

Manually calling `close()` works, but there's a cleaner approach: **try-with-resources**. Declare your file object inside the `try` parentheses, and Java automatically closes it when the block ends — even if an exception occurs.

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class TryWithResources {
    public static void main(String[] args) {
        try (Scanner sc = new Scanner(new File("data.txt"))) {
            while (sc.hasNextLine()) {
                System.out.println(sc.nextLine());
            }
        } catch (FileNotFoundException e) {
            System.out.println("File not found: " + e.getMessage());
        }
        // sc is automatically closed here — no close() needed
    }
}
```

You can open multiple resources in the same try, separated by semicolons:

```
try (Scanner sc = new Scanner(new File("input.txt"));
     PrintWriter pw = new PrintWriter("output.txt")) {

    while (sc.hasNextLine()) {
        pw.println(sc.nextLine().toUpperCase());
    }

} catch (Exception e) {
    System.out.println("Error: " + e.getMessage());
}
// Both sc and pw are automatically closed
```

> 💡 **Why Try-with-Resources Is Better**
>
> With manual `close()`, if an exception fires before you reach the close call, the file stays open. Try-with-resources closes the file no matter what happens — exception or not. Make it your default for any file work.

8. Processing File Data

Reading a file line by line is just the start. Here are three patterns you'll use constantly.

Pattern 1: Count Lines

```
try (Scanner sc = new Scanner(new File("data.txt"))) {
    int count = 0;
    while (sc.hasNextLine()) {
        sc.nextLine();  // consume the line
        count++;
    }
    System.out.println("Total lines: " + count);
} catch (FileNotFoundException e) {
    System.out.println("File not found.");
}
```

Pattern 2: Sum and Average Numbers

File `scores.txt` — one integer per line:

```
try (Scanner sc = new Scanner(new File("scores.txt"))) {
    int sum = 0;
    int count = 0;

    while (sc.hasNextInt()) {
        sum += sc.nextInt();
        count++;
    }

    if (count > 0) {
        System.out.printf("Sum:     %d%n", sum);
        System.out.printf("Average: %.2f%n", (double) sum / count);
    }
} catch (FileNotFoundException e) {
    System.out.println("File not found.");
}
```

If `scores.txt` contains 85, 92, 78, 96, 88 (one per line):

```
Sum:     439
Average: 87.80
```

Pattern 3: Parse CSV Data with split()

CSV (Comma-Separated Values) is one of the most common data formats. Each line holds fields separated by commas:

```
// students.csv:
// Alice,92,88,95
// Bob,78,82,80
// Carol,95,91,97

try (Scanner sc = new Scanner(new File("students.csv"))) {
    while (sc.hasNextLine()) {
        String line = sc.nextLine();
        String[] parts = line.split(",");

        String name  = parts[0];
        int s1 = Integer.parseInt(parts[1]);
        int s2 = Integer.parseInt(parts[2]);
        int s3 = Integer.parseInt(parts[3]);
        double avg = (s1 + s2 + s3) / 3.0;

        System.out.printf("%-10s avg: %.1f%n", name, avg);
    }
} catch (FileNotFoundException e) {
    System.out.println("File not found.");
}
```

Output:

```
Alice      avg: 91.7
Bob        avg: 80.0
Carol      avg: 94.3
```

> **⚠️ Integer.parseInt() Can Throw NumberFormatException**
>
> If a field contains unexpected data — a blank, a header, a typo — `Integer.parseInt()` will throw `NumberFormatException`. Validate your data or wrap the parse in its own try-catch.

> **🔧 Try It**
>
> Create a file called `words.txt` with one word per line. Write a program that reads every word, prints it in uppercase, and then prints the total word count at the end.

9. Practical Example: Student Grade Processor

Let's put it all together. This program reads student records from an input file, calculates each student's average, and writes a formatted report to an output file.

Input File: students.csv

```
Alice,85,92,88
Bob,70,75,68
Carol,95,98,100
David,60,72,65
```

The Program

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.Scanner;

public class GradeProcessor {

    public static void main(String[] args) {
        String inputFile  = "students.csv";
        String outputFile = "report.txt";

        try (Scanner sc = new Scanner(new File(inputFile));
             PrintWriter pw = new PrintWriter(outputFile)) {

            pw.println("=== Grade Report ===");
            pw.println();

            while (sc.hasNextLine()) {
                String line = sc.nextLine().trim();
                if (line.isEmpty()) continue;  // skip blank lines

                String[] parts = line.split(",");
                String name  = parts[0];
                int s1 = Integer.parseInt(parts[1]);
                int s2 = Integer.parseInt(parts[2]);
                int s3 = Integer.parseInt(parts[3]);
                double avg = (s1 + s2 + s3) / 3.0;

                String grade = getLetterGrade(avg);

                pw.printf("%-10s  Scores: %d, %d, %d  |  Avg: %.1f  |  Grade: %s%n",
                          name, s1, s2, s3, avg, grade);
            }

            pw.println();
            pw.println("Report generated by GradeProcessor.java");

        } catch (FileNotFoundException e) {
            System.out.println("Input file not found: " + e.getMessage());
        } catch (Exception e) {
            System.out.println("Error processing file: " + e.getMessage());
        }
```

```
        System.out.println("Done. See " + outputFile);
    }

    public static String getLetterGrade(double avg) {
        if (avg >= 90) return "A";
        if (avg >= 80) return "B";
        if (avg >= 70) return "C";
        if (avg >= 60) return "D";
        return "F";
    }
}
```

```
  === Grade Report ===

  Alice       Scores: 85, 92, 88  |  Avg: 88.3  |  Grade: B
  Bob         Scores: 70, 75, 68  |  Avg: 71.0  |  Grade: C
  Carol       Scores: 95, 98, 100 |  Avg: 97.7  |  Grade: A
  David       Scores: 60, 72, 65  |  Avg: 65.7  |  Grade: D

  Report generated by GradeProcessor.java
```

Notice how this program combines everything from this module: reading CSV data with Scan

🔧 **Try It — Extend the Program**

Modify the grade processor to also calculate and write the class average at the bot

📖 Key Terms

Term	Definition
`File`	A Java class representing a path to a file or directory — do
`PrintWriter`	A class for writing formatted text to a file. Supports print
`FileWriter`	Lower-level writer; used with the append flag (true) to add
`Scanner (file)`	The same Scanner you know — pointed at a File object instead
`FileNotFoundException`	Thrown when you try to open a file that doesn't exist or can
`IOException`	Broader I/O exception; parent of FileNotFoundException. Used
`Try-with-resources`	A try block that automatically closes declared resources whe
`Append mode`	Opening a file for writing without erasing its existing cont
`CSV`	Comma-Separated Values — a plain text format where fields ar
`split(",")`	String method that splits a string on a delimiter and return

📋 Module Summary

- **File I/O** lets programs persist data beyond a single run — RAM is temporary, files
- The **File class** represents a path; it doesn't open or create anything on its own.
- **PrintWriter** writes text to a file using the same methods as System.out; always cal
- **FileWriter with true** opens a file in append mode — existing content is preserved.
- **Scanner(new File(...))** reads a text file the same way you read keyboard input.
- **FileNotFoundException** is a checked exception — handle it with try-catch or declare
- **Try-with-resources** automatically closes files — use it by default.
- Common processing patterns: count lines, sum/average numbers, parse CSV with split

Module 11 Quiz — File I/O

Name: _____ **Date:** _____ **Score:** _____ / 10

Circle the best answer for each question.

1.

What does new File("grades.txt") do in Java?

A) Creates a Java object representing the path grades.txt

B) Opens grades.txt for reading immediately

C) Creates an empty file called grades.txt on disk

D) Deletes grades.txt if it already exists

2.

Which statement about RAM vs. files is correct?

A) Files are faster than RAM for storing data

B) RAM retains data when the program ends; files do not

C) Files retain data when the program ends; RAM does not

D) Both RAM and files are erased when the program ends

3.

What is the output of the following code if output.txt already contains the line "Hell

```
PrintWriter pw = new PrintWriter("output.txt");
pw.println("World");
pw.close();
```

A) The file contains "Hello" then "World" on separate lines

B) The file contains only "World"

C) The file contains "HelloWorld" on one line

D) A FileNotFoundException is thrown

4.

Which constructor opens a file so that new data is added to the *end* without erasing ex

A) `new PrintWriter("log.txt")`

B) `new Scanner(new File("log.txt"))`

C) `new FileWriter("log.txt", false)`

D) `new FileWriter("log.txt", true)`

5.

A file named data.txt contains:

```
10
20
30
```

What does this code print?

```
Scanner sc = new Scanner(new File("data.txt"));
int sum = 0;
while (sc.hasNextInt()) {
    sum += sc.nextInt();
}
sc.close();
System.out.println(sum);
```

A) 60

B) 10

C) A FileNotFoundException is thrown

D) 30

6.

Which exception must be handled when using new Scanner(new File("f.txt"))?

A) IOException

B) NullPointerException

C) NumberFormatException

D) FileNotFoundException

7.

What is the main advantage of try-with-resources over manually calling close()?

A) It automatically closes the file even if an exception occurs

B) It reads files faster than a regular try block

C) It prevents FileNotFoundException from being thrown

D) It allows you to open binary files without extra imports

8.

Given the line String line = "Alice,92,88,95";, what does line.split(",")[1] return?

A) "Alice"

B) "92"

C) "88"

D) 92 (as an int)

9.

Which File method returns true if the path exists and is a regular file (not a directo

A) exists()

B) canRead()

C) isDirectory()

D) isFile()

10.

You want to read a file and write a transformed version to another file. Which approac

A) `try { Scanner sc = ...; PrintWriter pw = ...; } catch (...) { }`

B) Both A and C are correct and produce identical behavior

C) `try (Scanner sc = ...; PrintWriter pw = ...) { }`

D) `try (Scanner sc = ...) { try (PrintWriter pw = ...) { } }`

✅ Answer Key

Question	Answer	Explanation
1	A	The File class represents a file or directory path.
2	C	A Scanner can read from a file by passing a File object to its con
3	B	PrintWriter writes formatted text to a file.
4	D	A FileNotFoundException occurs when the specified file does not ex
5	A	hasNextLine() returns true if there is another line to read.
6	D	The throws clause declares that a method may throw a checked excep
7	A	try-with-resources automatically closes files when the block finis
8	B	Appending to a file uses new FileWriter("file.txt", true) with the
9	D	A BufferedReader reads text efficiently by buffering chunks of dat
10	C	The exists() method checks whether a file or directory exists befo

ITP 120 — Java Programming I — Northern Virginia Community College

Author: Randy Michak |

Contribute on GitHub