

Class

16

CSC 240 Computer Science III

Advanced File I/O: Binary Files, Random Access & Serialization

Dr. Si Chen (schen@wcupa.edu)



Tue / Thu · 9:30 – 10:45 AM · Anderson Hall 315
Fall 2026 · Week 10 · Thursday, October 29, 2026
wcu.ninja/csc240

Today's plan

Warm-up (3 min): one try-with-resources question from Tuesday.

Part 1, binary files: why a file of numbers does not have to be text; `DataOutputStream` / `DataInputStream`.

Part 2, random access: jump straight to record `*i*` with `RandomAccessFile` and `seek`, and change one record in place.

Part 3, serialization: save whole objects (even a whole `ArrayList`) with one call.

In-class (20 min): save the Pokémon team both ways.

HW3 Q5 (Persistence) uses exactly these three tools. Code: Class 16 zip on `wcu.ninja`. Read Gaddis 11.3.

Warm-up: what goes wrong here?

```
static int countLines(String file) {
    Scanner in = null;
    int n = 0;
    try {
        in = new Scanner(new File(file));
        while (in.hasNextLine()) {
            in.nextLine();
            n++;
        }
    } catch (FileNotFoundException e) {
        System.out.println("cannot open " + file);
    } finally {
        in.close();
    }
    return n;
}
```

1. Call it with a file that does **not** exist. What happens, step by step?

2. Rewrite it with try-with-resources.

Answer 1: `new File(...)` is fine, `new Scanner(...)` throws, the catch prints, then `finally` calls `in.close()` on **null**: `NullPointerException`, and the method crashes anyway.

Answer 2: `try (Scanner in = new Scanner(new File(file))) { ... } catch (FileNotFoundException e) { ... }` and delete the `finally` and the `null`. Java only closes a resource that was actually opened.

Every file example today uses try-with-resources.

PART 1

Text Files vs. Binary Files

Everything so far (Scanner, PrintWriter, BufferedReader) wrote characters a person can read. A binary file stores values the way the computer keeps them in memory: compact, fast, and unreadable in Notepad.

The same five ints, two ways

```
int[] scores = {1234567, 2026, -40, 31415926, 1000000};

// text: characters + newlines
try (PrintWriter out = new PrintWriter("scores.txt")) {
    for (int s : scores) out.println(s);
}
// binary: 4 bytes per int, no separators
try (DataOutputStream out = new DataOutputStream(
    new FileOutputStream("scores.dat"))) {
    for (int s : scores) out.writeInt(s);
}

// scores.txt: 34 bytes
// scores.dat: 20 bytes

// xxd scores.txt    (hex dump)
// 31 32 33 34 35 36 37 0a 32 30 32 36 0a ...
// 1  2  3  4  5  6  7 \n  2  0  2  6 \n
// xxd scores.dat
// 00 12 d6 87 00 00 07 ea ff ff ff d8 01 df 5e 76 ...
// 1234567      2026      -40      31415926
```

Text: each digit is a character (1 byte in ASCII). `31415926` takes 9 bytes with its newline; `-40` takes 4.

Binary: every `int` is **exactly 4 bytes**, whatever its value. A `double` is 8, a `long` 8, a `boolean` 1.

No parsing when you read it back (no `Integer.parseInt`), so it is faster, and there is no `NumberFormatException`.

The price: you cannot read or edit it in a text editor, and the reader must know **exactly** what was written.

Open `scores.dat` in VS Code or Notepad and look.

Class 16 zip → 01_data_streams/ (part 1 of main)

DataOutputStream and DataInputStream

Writing: wrap a FileOutputStream

```
try (DataOutputStream out = new DataOutputStream(
    new FileOutputStream("team.dat"))) {
    out.writeInt(2);           // COUNT first
    out.writeUTF("Pikachu");   // String
    out.writeInt(25);          // level
    out.writeDouble(6.0);      // weight
    out.writeUTF("Snorlax");
    out.writeInt(30);
    out.writeDouble(460.0);
}

// also: writeBoolean, writeChar, writeLong ...
// append instead of overwrite:
//   new FileOutputStream("team.dat", true)
```

Reading: same methods, same order

```
try (DataInputStream in = new DataInputStream(
    new FileInputStream("team.dat"))) {
    int n = in.readInt();      // how many?
    for (int i = 0; i < n; i++) {
        String name = in.readUTF();
        int level = in.readInt();
        double weight = in.readDouble();
        System.out.println(name + " Lv." + level
            + " " + weight + " kg");
    }
}

// Pikachu Lv.25 6.0 kg
// Snorlax Lv.30 460.0 kg
```

No lines, no commas: the **order of the write calls is the file format**. The reader repeats it exactly.

Where does the file end? (and how to get garbage)

```
// A) no count: read until readInt throws EOFException
try (DataInputStream in = new DataInputStream(
    new FileInputStream("scores.dat"))) {
    while (true) {
        System.out.print(" " + in.readInt());
    }
} catch (EOFException e) {
    System.out.println(" (end of file)");
}
// 1234567 2026 -40 31415926 1000000 (end of file)

// B) write the count first (previous slide)

// C) read with the WRONG method:
try (DataInputStream in = new DataInputStream(
    new FileInputStream("scores.dat"))) {
    System.out.println(in.readDouble());
}
// 2.6197459776656243E-308      <- no exception!
```

`EOFException` (a subclass of `IOException`) is how the Data streams say "no more bytes". It is one of the few places where Java uses an exception as an end marker.

Writing the **count first** is cleaner: the loop is an ordinary `for`, and a short (damaged) file is a real error.

C is the scary one: `readDouble` happily takes the first 8 bytes (two ints) and returns nonsense. The bytes carry no type information.

So a binary format must be written down somewhere (a comment, a spec), and writer and reader must agree on it.

Class 16 zip → 01_data_streams/ (parts 2–4)

PART 2

Random Access: Jump to Record i

Every stream so far is sequential: to read the 1000th value you read the first 999. If every record has the same size, you can compute where record i starts and go there directly.

Fixed-length records: the position is arithmetic

```
// temps.dat: one int (4 bytes) per day
//
// byte:   0    4    8   12   16   20   24   28
//         +---+---+---+---+---+---+---+
//         | 61 | 64 | 58 | 70 | 73 | 69 | 55 |
//         +---+---+---+---+---+---+---+
// record:  0    1    2    3    4    5    6
//         Mon  Tue  Wed  Thu  Fri  Sat  Sun

// record i starts at byte  i * RECORD_SIZE
//   Thursday (3) -> 3 * 4 = 12

// a record with two ints (id, level):
//   RECORD_SIZE = 8
//   record i: id at 8*i, level at 8*i + 4

// number of records = file length / RECORD_SIZE
//   28 / 4 = 7
```

Same idea as an array index:

$a[i]$ is at $\text{start} + i * \text{elementSize}$ in memory.

Only works if **every record has the same size**. `int`, `double`, `long` are fixed; a `String` written with `writeUTF` is not (it depends on the length).

That is why databases store a name in a fixed-width field (e.g. always 20 chars, padded).

Record size and field order are the file format: write them as constants (`RECORD_SIZE = 4`) and use them everywhere.

RandomAccessFile: seek, read, write in place

```
try (RandomAccessFile f =
    new RandomAccessFile("temps.dat", "rw")) {
    f.setLength(0);           // empty it first
    for (int t : temps) f.writeInt(t);
    System.out.println(f.length());           // 28
}

try (RandomAccessFile f =
    new RandomAccessFile("temps.dat", "r")) {
    f.seek(3L * RECORD_SIZE); // jump to byte 12
    System.out.println(f.getFilePointer()); // 12
    System.out.println(f.readInt());        // 70
    System.out.println(f.getFilePointer()); // 16
}

try (RandomAccessFile f =
    new RandomAccessFile("temps.dat", "rw")) {
    f.seek(5L * RECORD_SIZE);
    f.writeInt(79);           // overwrite Saturday
}
// backwards: 55 79 73 70 58 64 61
```

Mode "r": read only (file must exist). Mode "rw": read and write, created if missing.

It has the same `readInt` / `writeInt` / `readDouble` ... methods as the Data streams, plus `seek(pos)`, `getFilePointer()`, `length()`, `setLength(n)`.

Every read or write **moves the pointer** forward by the bytes it used. To read and then rewrite the same record, `seek` back first.

`writeInt` in "rw" mode overwrites those 4 bytes; nothing else in the file moves.

`setLength(0)`: without it, writing a shorter list into an old file leaves old records at the end.

Reading past the end:
`EOFException`.

Class 16 zip → 02_random_access/ · 3L * RECORD_SIZE: seek takes a long (files can be > 2 GB)

PART 3

Serialization: Save the Object Itself

Writing an object field by field is tedious and easy to get out of order. Java can turn a whole object, and every object it refers to, into bytes, and rebuild it later.

Serializable + ObjectOutputStream / ObjectInputStream

```
public class Student implements Serializable {
    private static final long serialVersionUID = 1L;
    private String name;
    private int id;
    private double gpa;
    private transient String password; // not saved
    ...
}

// save: ONE call writes the list AND every Student
try (ObjectOutputStream out = new ObjectOutputStream(
    new FileOutputStream("roster.ser"))) {
    out.writeObject(roster); // ArrayList<Student>
}

// load: returns Object -> cast
try (ObjectInputStream in = new ObjectInputStream(
    new FileInputStream("roster.ser"))) {
    ArrayList<Student> loaded =
        (ArrayList<Student>) in.readObject();
} catch (IOException | ClassNotFoundException e) {
    System.out.println("could not load: " + e);
}

// loaded Ana (#1001, GPA 3.7, password null)
```

Serializable is a **marker interface**: no methods to write. It is your permission slip: "objects of this class may be saved".

serialVersionUID: a version number for the class layout. If you change the fields later, an old file is rejected with **InvalidClassException** instead of loading wrong data.

ArrayList and **String** are already **Serializable**; the list is only saveable if its **elements** are too.

readObject also throws **ClassNotFoundException** (checked): the file names a class this program does not have.

transient fields are skipped; after loading they hold the default (**null**, **0**, **false**).

Class 16 zip → 03_serialization/



When serialization fails, and what loading does NOT do

```
// 1. A class that is not Serializable:
out.writeObject(new Thread());
java.io.NotSerializableException: java.lang.Thread

// same if ONE field of your class is not Serializable:
// class Trainer implements Serializable {
//     private Scanner input;    // Scanner is not
// }    -> NotSerializableException: java.util.Scanner

// 2. Loading does NOT call your constructor:
class Pokemon implements Serializable {
    protected transient int level = 1;
    ...
}
// after readObject: level == 0, not 1

// 3. Wrap the second checked exception (Class 15):
} catch (ClassNotFoundException e) {
    throw new IOException(
        "not a team file: " + file, e);
}
```

`NotSerializableException` is an `IOException` thrown **at run time**: the compiler does not check it. Test your save code.

Every object reachable from the one you save gets saved: fields, fields of fields, list elements. One bad link breaks the whole save. (`transient` is the way out.)

Deserialization rebuilds the fields from the bytes; field initializers (= 1) and constructors of your class **do not run**. A transient field gets the type's default.

Callers prefer **one** exception type: wrapping `ClassNotFoundException` in an `IOException` with a cause is exactly HW3 Q5(a).

Which file tool when?

Text vs. Data streams vs. objects

Text (`PrintWriter`, `Scanner`, `BufferedReader`): people read or edit it, other programs (Excel, Python) read it. CSV, logs, config.

Data streams (`DataOutputStream`): compact numbers in a format you define. Must read in the same order, sequentially.

Objects (`ObjectOutputStream`): save and restore your own Java objects with one call. Only Java can read it, and only with the same class.

Random access

RandomAccessFile: fixed-length records, read or change record **i** without touching the others.

Good for: a big table of numbers, an index, a high-score table, a game save slot.

Not for: variable-length data (Strings, CSV lines) unless you pad them to a fixed width.

A random-access file is **storage**: changing record 3 does not change the object in your `ArrayList`, and vice versa (see HW3 Q5(d)).

Real programs often mix them: objects in memory, a text export for people, a binary file for speed.

In-class: save the team (objects + a level table)

```
// Part A: objects -> team.ser
static void saveTeam(ArrayList<Pokemon> team,
                    String file) throws IOException
{ /* TODO 1 */ }
static ArrayList<Pokemon> loadTeam(String file)
    throws IOException { /* TODO 3 */ }

// Part B: fixed-length records -> levels.dat
// record = int id + int level = 8 bytes
static void saveLevels(ArrayList<Pokemon> team,
                      String file) throws IOException
{ /* TODO 4 */ }
static int levelUp(String file, int index)
    throws IOException { /* TODO 5 */ }

// goal:
// saved 4 Pokemon to team.ser
// loaded #25 Pikachu (Electric) Lv.25 HP=35 ...
// ... (4 loaded lines)
// could not load: missing.ser (No such file or directory)
// levels.dat: 32 bytes, 4 records
// Eevee leveled up to 13
// #25 Lv.25 | #143 Lv.30 | #133 Lv.13 | #6 Lv.36
```

TODO 1: `saveTeam` with one `writeObject`, try-with-resources. Run: `NotSerializableException: Pokemon`. Why?

TODO 2: fix `Pokemon.java` (an `implements`, a `serialVersionUID`, an import).

TODO 3: `loadTeam`: `readObject`, cast; wrap `ClassNotFoundException` in an `IOException` with a cause.

TODO 4: `saveLevels`: `"rw"`, `setLength(0)`, `writeInt(id)`, `writeInt(level)` per Pokémon.

TODO 5: `levelUp`: the level of record `*i` is at byte `8*i + 4`. Seek, read, **seek back**, write level + 1.

TODO 6 (think): make `level` transient. What do the loaded Pokémon show, and why not 1?

Class 16 zip → 04_pokemon_save_starter/ (goal output at the top of Main.java)

Check yourself

1. A file holds 250 `double` values written with `writeDouble`. How many bytes is it? Where does value number 100 (counting from 0) start?
2. A program writes `writeInt(age)` then `writeUTF(name)`. The reader calls `readUTF()` then `readInt()`. What happens?
3. Why can't you use `seek` to jump to the 50th line of a CSV file?
4. `class Team implements Serializable { ArrayList<Pokemon> members; }` and `Pokemon` is **not** `Serializable`. Does it compile? What happens on `writeObject(team)`?
5. You read record 2 with `seek(8) + readInt()` and then call `writeInt(99)` without another `seek`. Which record did you just change?
6. Name the two checked exceptions `readObject` can throw. Which one does `missing.ser` cause?

Check yourself: answers

1. $250 \times 8 = \mathbf{2000 \text{ bytes}}$. Value 100 starts at byte $100 \times 8 = \mathbf{800}$.
2. No compile error, no exception. With age 19 and name "Ana": `readUTF` takes the first 2 bytes of the int (0) as the length and returns "", then `readInt` returns **1245187**. Silent garbage: writer and reader must agree on the order.
3. Lines have different lengths, so line 50's position cannot be computed; you have to read the 49 lines before it.
4. It **compiles** (the check is at run time). `writeObject` throws `NotSerializableException: Pokemon`, because every object reachable from `team` must be Serializable.
5. Record **3**: `readInt` moved the pointer from 8 to 12. To change record 2, `seek(8)` again before writing.
6. `IOException` and `ClassNotFoundException`. A missing file is a `FileNotFoundException` (an `IOException`), thrown by `new FileInputStream(...)` before `readObject` even runs.

Wrap-up & what's next

Binary files store values, not characters: an `int` is always 4 bytes. `DataOutputStream` / `DataInputStream`, same methods in the same order; the end is a count or an `EOFException`.

Random access: fixed-length records, $\text{position} = \text{index} * \text{RECORD_SIZE}$, `seek` then read/write; `"rw"` changes a record in place. Every read/write moves the pointer.

Serialization: implements `Serializable` + `serialVersionUID`, `writeObject` / `readObject` + `cast`, `transient` to skip a field; everything reachable must be `Serializable`.

Practice: finish `04_pokemon_save_starter` (TODO 1–6).

HW3 (all of Chapter 11, including Q5 Persistence) is due **Tue Nov 10, 11:59 PM**. Start Q5 this weekend while today is fresh.

Next week: see the schedule on wcu.ninja/csc240.

Sources

Gaddis, **Starting Out with Java**, Ch. 11.3: binary files (`DataOutputStream`, `DataInputStream`, `EOFException`), random access files (`RandomAccessFile`, `seek`, `getFilePointer`), and object serialization (`Serializable`, `ObjectOutputStream`, `ObjectInputStream`, `transient`).

Java SE 17 API: `java.io.DataOutput` / `DataInput` (sizes of `writeInt`, `writeDouble`, `writeUTF`), `RandomAccessFile`, `ObjectOutputStream`, `ObjectInputStream`, `Serializable`, `NotSerializableException`, `InvalidClassException`. Java Object Serialization Specification §1–3.

`Pokemon` and `InvalidLevelException` are the same as Classes 13–15; `Student` is new.

All outputs, byte sizes and hex dumps shown were checked with OpenJDK 17.