Java I/O

Table of contents

Setup 1
Airline schedule 1
Maze — designing a file format 6

Practical for Week 10 (no lecture this week). Extends java-io[1] (Week 8) with saving/loading a data model to/from files, plus a file-format design exercise.

Setup

Provided classes: `Schedule`, `Airport`, `Flight`, `DayOfWeek`, `BadScheduleException` (an airline schedule of flights between airports), plus an unrelated `Maze` class.

Airline schedule

Goal: make a `Schedule`'s data persistent by saving/loading it to/from a file.

Task 0 — implement `Schedule.toString()` per its Javadoc spec.

[1] courses/csse2002/java-io.pdf

Working

```
@Override
public String toString() {
    StringJoiner joiner = new StringJoiner(System.lineSeparator());
    joiner.add(String.valueOf(getConnectedAirports().size()));
    joiner.add(String.valueOf(this.flights.size()));
    for (Airport airport : getConnectedAirports()) {
        joiner.add(airport.toString());
    }
    for (Flight flight : this.flights) {
        joiner.add(flight.toString());
    }
    return joiner.toString();
}
```

Task 1 — implement `void save(String filename)`, propagating any exceptions rather than handling them:

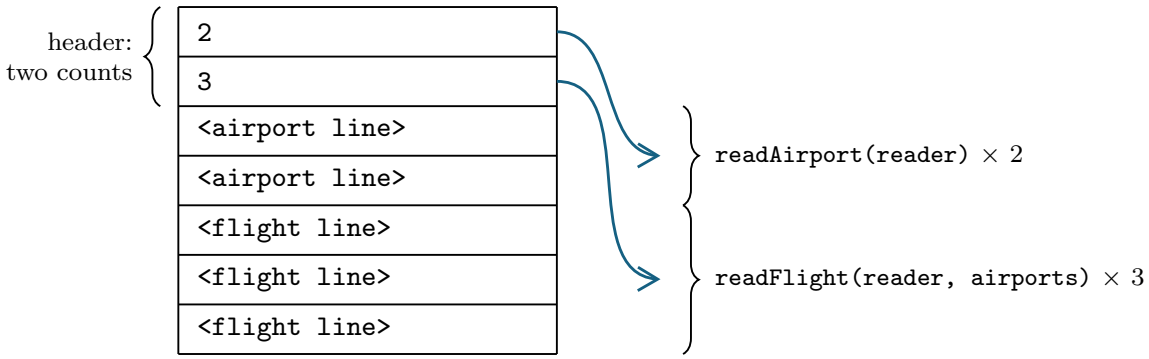
Working

```
public void save(String filename) throws IOException {
    BufferedWriter writer = new BufferedWriter(new FileWriter(filename));
    writer.write(this.toString());
    writer.close();
}
```

Between them, `toString()` and `load()` fix a file format. `toString()` emits two counts and then the airports and flights in that order; `load()` has to consume exactly the same thing, which is what makes the two counts load-bearing — they are what tells the reader how many times to go round each loop:

Every `BadScheduleException` thrown by `load()` below is one of the ways the file on disk can fail to match Figure 1: a header line missing, a count that isn't a number, a data line missing, a line with the wrong number of `|`-separated fields, or an airport code a flight refers to that never appeared in the airport block.

Task 2 — implement `static Schedule load(String filename)`, using two private helpers (`readFlight`, `readAirport`) and throwing `BadScheduleException` on a failed numeric parse or an unknown airport code:



Within a line, fields are separated by `|`:

`<airport line>`

lineParts[0]	lineParts[1]
new Airport(lineParts[0], lineParts[1])	

`<flight line>`

flight number	origin	destination	day of week
Integer.parseInt	match getCode()	match getCode()	DayOfWeek.valueOf

Figure 1: The Schedule file format fixed by toString() and load(). The two counts shown (2 airports, 3 flights) are an illustrative instance, not a fixture of the format. The teal arrows record the dependency that makes the format parseable at all: each header count governs the trip count of one read loop. The four flight fields are pinned exactly by load() - lineParts[0] goes to Integer.parseInt, [1] and [2] are matched against Airport.getCode(), [3] to DayOfWeek.valueOf - whereas the two airport fields are only ever passed straight to new Airport(lineParts[0], lineParts[1]), so which of them is the code is not settled by load() and they are left unnamed.

Working

```
public static Schedule load(String filename) throws IOException,
 ↪  BadScheduleException {
    BufferedReader reader = new BufferedReader(new FileReader(filename));

    String airportsLine = reader.readLine();
    String flightsLine = reader.readLine();
    if (airportsLine == null || flightsLine == null) {
        throw new BadScheduleException();
    }

    int numAirports, numFlights;
    try {
        numAirports = Integer.parseInt(airportsLine);
        numFlights = Integer.parseInt(flightsLine);
    } catch (NumberFormatException e) {
        throw new BadScheduleException();
    }

    List<Airport> airports = new ArrayList<>();
    for (int i = 0; i < numAirports; ++i) {
        airports.add(readAirport(reader));
    }
    List<Flight> flights = new ArrayList<>();
    for (int i = 0; i < numFlights; ++i) {
        flights.add(readFlight(reader, airports));
    }

    return new Schedule(flights);
}

private static Flight readFlight(BufferedReader reader, List<Airport> airports)
        throws IOException, BadScheduleException {
    String line = reader.readLine();
    if (line == null) {
        throw new BadScheduleException();
    }

    String[] lineParts = line.split("\\|");
    if (lineParts.length != 4) {
        throw new BadScheduleException();
    }

    int flightNumber;
    try {
        flightNumber = Integer.parseInt(lineParts[0]);
    } catch (NumberFormatException e) {
        throw new BadScheduleException();
    }

    Airport origin = null, destination = null;
    for (Airport airport : airports) {
        if (airport.getCode().equals(lineParts[1])) {
            origin = airport;
        }
        if (airport.getCode().equals(lineParts[2])) {
            destination = airport;
        }
    }
    if (origin == null || destination == null) {
        throw new BadScheduleException();
    }
```

`readAirport` has to declare `BadScheduleException` as well as `IOException`, because its body throws it; with `throws IOException` alone the listing does not compile.
`readFlight`/`readAirport` both throw `BadScheduleException` for a malformed line — this is a form of defensive programming (see java-specification[2]) protecting the `Schedule` created by `load()` from corrupt input, rather than letting a `NullPointerException`/`ArrayIndexOutOfBoundsException` leak out from deeper in the parsing logic.

Maze — designing a file format

`Maze` maps `(row, column)` positions to tile types (start, end, wall, empty). Unlike `Schedule`, there's no existing format to follow — the task is to *design* one that's sufficient to save and reload a `Maze`'s full state.

Task 3 — design a file format for `Maze`. (Hint: the class's invariants may let you store less than a full grid dump.)

i Working

Several reasonable designs, each with different tradeoffs:

1. **Visual dump** — store the same characters `render()` would print (e.g. `#` for wall, `S`/`E` for start/end). Human-readable, but more complex to parse back than necessary.
2. **Linear encoding of every tile** — e.g. `#S#E` for a 2×2 grid means wall at (0,0), start at (0,1), wall at (1,0), end at (1,1) (with or without an outer border wall included).
3. **One line per map entry** — e.g. `1,2,#` means "wall at (1, 2)". Simple, but verbose for large mazes with many walls.
4. **Only relevant tiles** — the first two stored positions are always start/end, and every position after that is implicitly a wall; positions can be `(row, column)` pairs or a single linear index (`rows * row + column`).
5. **Start/end positions + a bitmap** — store start and end explicitly, then a bit per remaining tile (`1` = wall, `0` = empty).
6. **Start/end positions + wall runs** — store start and end, then walls as (start, end) position pairs, so a run of adjacent wall tiles can be stored as a single entry instead of one entry per tile.

Options 4-6 are the ones the hint is pointing at: each leans on a regularity in the data rather than dumping the grid verbatim — 4 and 6 store only the tiles that are not empty

[2] courses/csse2002/java-specification.pdf

(6 collapsing adjacent walls into runs), and 5 still visits every remaining tile but spends one bit on it instead of a whole character.