Java I/O

Table of contents

Topics 1
The thread running through it 1
Set as homework 2
Applied class 2
Practical 2

See csse2002[1] for course logistics.

Topics

1. Streams (`java.io`, since Java 1.0) — byte-oriented
2. Readers & Writers (`java.io`, since Java 1.1) — character-oriented
3. Scanner (`java.util`, since Java 1.5) — a convenience wrapper, neither stream nor reader
4. New I/O (`java.nio.file`, since Java 1.7) — the file system itself

Instance Saving & Loading — saving and loading an object model to a file — is taken up in the Week 10 practical, week10-lab-java-io[2].

All content is in java-io[3].

The thread running through it

Each generation of the API is a response to what the previous one was awkward at:

[1] courses/csse2002/csse2002.pdf
[2] courses/csse2002/week10-lab-java-io.pdf
[3] courses/csse2002/java-io.pdf

- Byte streams give one uniform abstraction over any source or destination, so a method can take an `OutputStream` and not care whether it is a console, a file, or an array (see java-solid-principles[4] — Liskov Substitution).
- They were never ideal for text, so **Readers & Writers** were added to work in `chars` and Unicode. `InputStreamReader` is the bridge back to the byte world.
- Everything in `java.io` is assembled by **wrapping** one object around another — buffering, line-reading and formatting are separate classes you stack, not options on one class.
- None of it was designed for managing files and directories, so **New I/O** adds `Path` and `Files`.
- Every stream and reader must be **closed**; `try`-with-resources is the way to do it (Effective Java Item 9). This is where the topic leans hardest on java-exceptions[5].

Summary — which to use? Streams for binary data, Readers & Writers for text, New I/O when manipulating a file system or file contents.

Set as homework

- Read through `flush()`.
- Scanner vs. BufferedReader on Stack Overflow.
- Rewrite the stream and reader exercises using New I/O.

Applied class

See week8-tutorial-class-invariants[6] — protecting class invariants (a continuation of java-specification[7] and java-encapsulation[8] from earlier weeks).

Practical

See week8-lab-junit-testing[9] — Test Driven Development with JUnit, extending java-junit[10] from Week 5.

[4] courses/csse2002/java-solid-principles.pdf
[5] courses/csse2002/java-exceptions.pdf
[6] courses/csse2002/week8-tutorial-class-invariants.pdf
[7] courses/csse2002/java-specification.pdf
[8] courses/csse2002/java-encapsulation.pdf
[9] courses/csse2002/week8-lab-junit-testing.pdf
[10] courses/csse2002/java-junit.pdf