10 File I/O and Serialization

Overview

- Understanding File I/O in Java
- Random access files
- Serialization
- The Serializable interface and why we won't use it

File I/O in Java

So far, we have handled applications that only operatoe on data stored in main memory (RAM). Data in RAM is super-fast to access compared to data stored on disk or on other storage devices. Differences in data read and write times can be orders or magnitude slower in disk-based applications compared to memory-bound applications.

In previous projects and courses, you've had some experience reading from and writing to files. Most of the time we use language libraries for this interaction with files without thinking too much about it. For example, Java provides the `Scanner` API for reading data from files.

You all used the `Scanner` API to implement your TUIs in a previous project and lab. There were no files involved there, because the "source" of input was `System.in`, or the "standard input stream".

```
Scanner scan = new Scanner(System.in);
```

If you wanted to read from a file instead, you would initialise the `Scanner` like so:

```
// You need to wrap the scanner in a try-catch if you want to read a file
try (Scanner scan = new Scanner(new File("my text file.txt"))) {
  // do stuff with the file's contents
  // the same Scanner API applies: scan.nextLine(), scan.next(), etc.
} catch (FileNotFoundException e) {
  // handle the scenario where the file doesn't exist
}
```

Similarly, Java provides an API for writing to files:

```
try (FileWriter writer = new FileWriter(new File("some text file.txt"))) {
  // write data to the file
  // API calls like writer.write("some text...");
} catch (IOException e) {
  // handle the exception
}
```

Random access

- **Sector** or **block**: The basic amount of data that will be read or written at one time by disk drive hardware. This is typically 512 bytes.
- **Seek**: Since data is read a block at a time, large pieces of data that cover sequential blocks tend to be faster to read. This is called *sequential access*. *Random access* involves accessing data from all over the disk drive (or from anywhere in a file) instead of in sequential blocks (or instead of start-to-finish in a file). So far, most of the file processing you've done has likely been sequential access.
  - The difference between sequential vs. random access is becoming smaller as SSDs and Flash storage devices (compared to traditional hard disk drives) become mainstream.

The `Scanner` API only allows sequential acccess to a Java file. You can't start reading from anywhere in the file. You haev to start at the beginning and move to the end, one unit at a time (using methods like `next()` (which gets you the next "word") or `nextLine()` (which gets you the next line)).

In addition to being quicker, these methods provide an abstracted view of the file. Instead of a lump of bits and bytes, we look at the file in terms that make sense to us (words, ints, lines, etc.).

However, sometimes we need *random* access ability to a file.

Java provides the RandomAccessFile API to allow this. Now if you access a file at some random position, there's no guarantee that you're getting a nice logically meaningful chunk of data. That's why this file's API is in terms of *bytes*.

I'll use "RAF" to mean to `RandomAccessFile` below.

Some key concepts:

- The RAF keeps track of a *file-pointer offset* position. That is, the position in the file at which the next read or write will occur.
- You use the `seek(long pos)` method to tell the RAF to move its pointer to the given offset (measured from the beginning of the file).
- You can use `readByte()` to read a single byte from the RAF (starting at wherever its offset currently is).
- You can use `read(byte[] b, int off, int len)` to read `len` bytes, starting at `off` and place them into the array `b`.
- Similarly, you can use `write`, `writeByte`, `writeBytes` to write to the RAF.

Hurray! You now have the ability read and write to arbitrary locations in the file. But you're dealing with bytes.

You'll notice that the RAF also provides methods to `readDouble`, `readChar`, etc. These methods are kind of doing what the `Scanner` does: they read the requisite number of bytes (e.g., 4 bytes for `int`, 8 bytes for `double`) and treat those bytes as the appropriate data type. It's up to you to know that, e.g., the next 4 bytes actually contain a meaningful integer.

Serialization

This leads us into our next topic.

Serialization is the conversion of an object (or some piece of data) to a byte stream. *Deserialization* is the process of turning a byte stream back into the object (or the original piece of data).

They are sometimes called *marshalling* and *unmarshalling*.

There are many reasons why we might want to serialize data:

- To transmit over the wire
- To enable interoperability between different systems. For example, you might want to "export" a Java object so that it can be "imported" into a Python program
- To persist data so that it "survives" the termination of a program

Of course, there exist multiple structured data formats for exporting data to files, like JSON, XML, and YAML. Those are certainly much more friendly and human-readable than writing out raw byte streams. However, those exports tend to be much larger, since those formats use plain text to represent the data, and tend to include "extraneous" data (like colons `:`, braces `{ } [ ]`, whitespace ` `, and plain text as opposed to raw bytes). So data written out in formats like JSON tend to be more human-readable while still being structured enough to be parsed by programs, but they tend to have a larger memory footprint as well.

Hence, we sometimes opt to serialize our data into raw byte streams. For example, suppose we are trying to serialize a `short` (an integer data type in Java that takes up two bytes of memory).

First recognise that the short is an *abstraction*. The computer doesn't know what an integer or a short is; all it knows is how to read bits and bytes. We (humans) decide that in certain contexts, certain sequences of bits and bytes mean certain human-sensible things (like integers, booleans, or characters).

So to serialise this short, our first task is become "less abstract" — we're going from the abstract human-friendly representation (the number 31543) to a less abstract representation (a byte array). We can use the `ByteBuffer` class to help with this.

```
short shortNum = 31543;
ByteBuffer bb = ByteBuffer.allocate(2);
bb.putShort(shortNum);
byte[] asArray = bb.array();
```

We can now use the `RandomAccessFile` to write out this byte array.

```
// assuming we initialised the RAF

randomAccessFile.write(asArray);
```

The entire `asArray` byte array has been written to the random access file. Note that this moves the pointer forwards two bytes! So any future reads will happen from that point onward. If you wanted to `read` the two bytes back into memmory, you would need to move the cursor back first.