File I/O

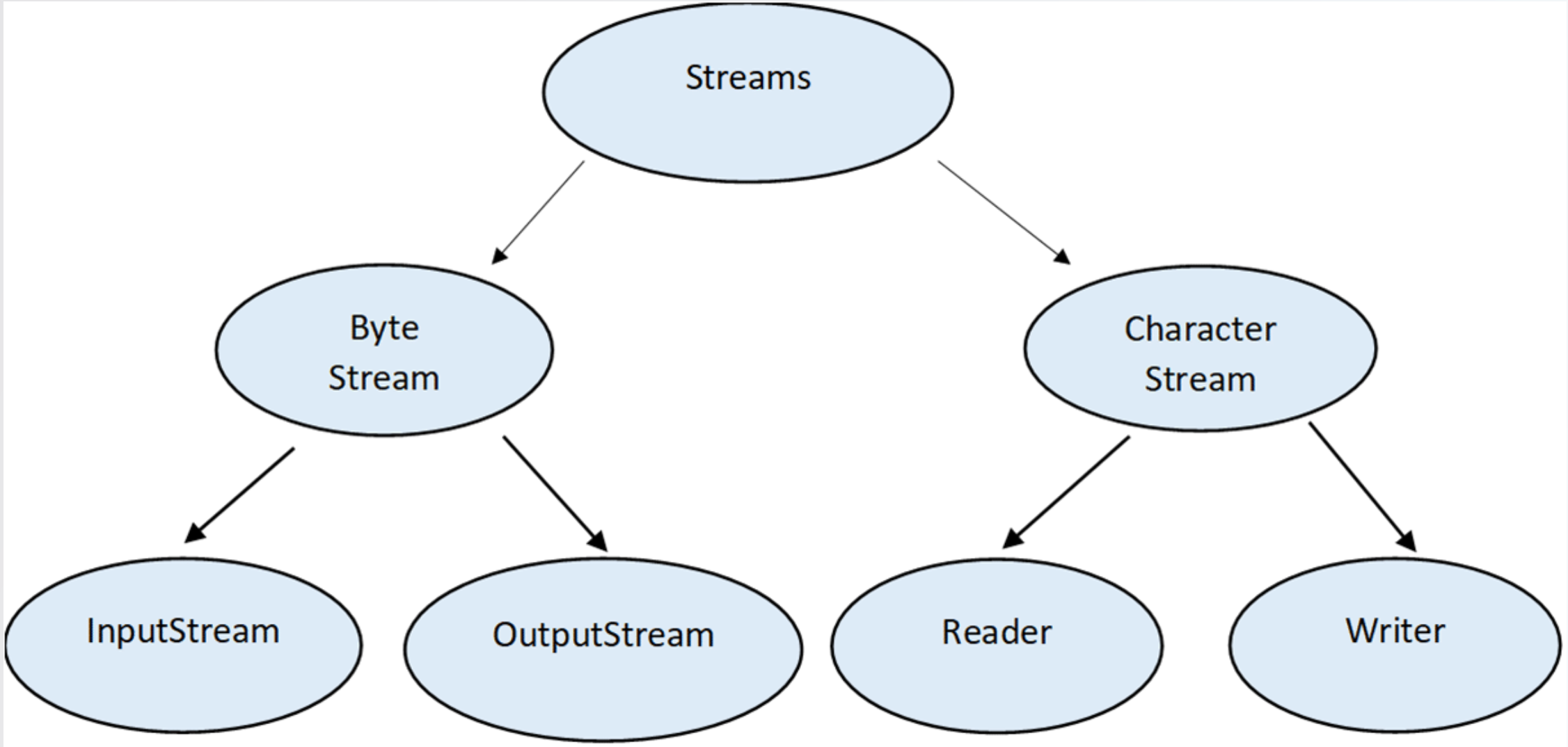
Agenda

1. I/O Streams
2. InputStream
3. OutputStream
4. Buffered Streams
5. Reader
6. Writer
7. RandomAccessFile

I/O Streams

- A **stream** is a sequence of data.
  - The stream abstraction represents a communication channel with the world outside the program
    - a file, a network connection
- An **I/O stream** represents an input source or output destination that can be read from or written to, respectively
- Information of several different types can be sent along a stream
  - bytes, primitive data types, objects
- Streams throw `IOExceptions` in Java 🚨

Class Hierarchy of I/O Streams



Reading Bytes: `InputStream`

- `InputStream` is an **abstract class** that serves as a superclass for all input streams of bytes
  - subclasses: `FileInputStream`, `ByteArrayInputStream`, `StringBufferInputStream`

Recall that **abstract classes** cannot be instantiated, but do provide a list of all methods that any subclass must be able to implement.

`InputStream` Interface

method	purpose
`void close()`	close this stream for reading
`void mark(int limit)`	specify the current location in the file to be able to return to, along with a maximum number of bytes that can be read before the mark is invalidated.
`int read()`	return the next byte (8 bits) as an `int`
`void reset()`	return to the previously marked location

Using `InputStream`

All subclasses of `InputStream` are byte streams, meaning that they return 8 bits of data from a file at a time as a `byte`.

- Useful for reading **raw data** from a file: image data, audio data, machine code
- Not so useful for dealing with text: a `char` is 16 bits, or two `bytes`.

The `byte` type is just a sequence of eight 1s and 0s that can be interpreted in a number of ways.

Example: `FileInputStream`

A `FileInputStream` is a fully-implemented subclass of the `InputStream` and can be used to read information from a file.

A simple look at reading one or a few `bytes` from a file:

```
FileInputStream fis = new FileInputStream("myFile.txt");
int b = fis.read(); // get one byte at a time
byte[] chunk = new byte[8];
int result = fis.read(chunk); // get a chunk of 8 bytes
// the return value is the number of bytes read; hopefully chunk.length
```

Example: `FileInputStream`

`int readUntil(byte stop, FileInputStream fis)` reads from a file until it encounters a specific byte, and returns the number of bytes read before that point.

```
public static int readUntil(byte stop, FileInputStream fis) {
  int count = 0;
  while (fis.read() != stop) {
    count++;
  }
  return count;
}
```

Writing Bytes: `OutputStream`

- `OutputStream` is an **abstract class** that serves as a superclass for all Output streams of bytes
  - subclasses: `FileOutputStream`, `ByteArrayOutputStream`, `StringBufferOutputStream`

Behaves exactly like the `InputStream` abstract class, but in reverse!

`OutputStream` Interface

method	purpose
`void close()`	close this stream for reading
`void write(byte[] b)`	writes all of the bytes in the array to the destination
`void write(int b)`	writes the first 8 bits of the int to the destination

Example: `FileOutputStream`

```
FileOutputStream fos = new FileOutputStream("myFile.txt");
fos.write(5); // write a single byte

byte[] chunk = {1, 2, 3, 4, 5, 6, 7, 8};
fos.write(chunk); // write a chunk of 8 bytes

fos.close(); // close the stream
```

Buffered Streams

Buffered streams read/write data from/to a **buffer**, which is a temporary storage area in memory.

- This means that the disk operations are only executed when the buffer is empty (reading) or full (writing)
- Gives improved performance since reading and writing to program memory is very fast compared to disk operations
    - Writing to an array: nanoseconds
    - Writing to a disk: milliseconds

`BufferedInputStream` and `BufferedOutputStream` are subclasses of `InputStream` and `OutputStream`, respectively that can be used wherever the superclass is expected.

Constructing Buffered Streams

- First, create the `InputStream` or `OutputStream` that you want to buffer.
- Then, construct the buffered version by passing a reference to the unbuffered stream to the constructor.

```
FileInputStream fis = new FileInputStream("myFile.txt");
BufferedInputStream bis = new BufferedInputStream(fis);

FileOutputStream fos = new FileOutputStream("myFile.txt");
BufferedOutputStream bos = new BufferedOutputStream(fos);
```

Writing Without a Buffer

```
fos.write(0); // write to disk, taking 1 ms
fos.write(3); // write to disk, taking 1 ms
fos.write(4); // write to disk, taking 1 ms
fos.write(7); // write to disk, taking 1 ms
```

Takes ~4ms to write 4 bytes to disk. 🐢 A disk has to literally rotate four times.

Writing With a Buffer

```
bos.write(0); // write to buffer, taking ~1 ns
bos.write(3); // write to buffer, taking ~1 ns
bos.write(4); // write to buffer, taking ~1 ns
bos.write(7); // write to buffer, taking ~1 ns

bos.flush(); // write to disk, taking 1 ms
```

Takes 1ms + 4ns (= 1.000004 ms) to write 4 bytes to disk. 🐰 A disk has to literally rotate once.

Character Streams

Whereas `InputStream` and `OutputStream` manipulate `bytes`, `Reader` and `Writer` deal with `chars`.

- `char` is a 16-bit type that can represent a single Unicode character

`Reader` and `Writer` are abstract classes that serve as superclasses for all character streams.

`Reader` & `Writer`

The `Reader` is implemented by `InputStreamReader`, `FileReader`, and `StringReader`.

The `Writer` is implemented by `BufferedWriter`, `FileWriter`, and `StringWriter`.

Examples

This example uses a StringReader, but similar code would work with a FileReader for example.

```
@Test
void test() throws IOException {
    Reader r = new StringReader("one two three four five");
    char c = (char) r.read();
    assertEquals('o', c);
    c = (char) r.read();
    assertEquals('n', c);
}
```

The characters are still read as `ints`, so you need to remember to cast them to `chars`. *Why?*

Examples

This example uses a StringWriter, but similar code would work with a FileWriter for example.

```
@Test
void test() throws IOException {
    Writer w = new StringWriter();
    w.write('ê');
    assertEquals("ê", w.toString());
}
```

Why is the type of `w` `Writer` and not `StringWriter`?

Random Access Files

- Streams (and Readers/Writers, and Scanners) have a **sequential** nature
  - You can only read from the beginning of a file to the end, sometimes resetting backwards to a fixed position.
- A **random access file** allows you to read from or write to any position in the file more easily
  - Behaves like a large array of bytes that you can freely index into
  - Provides a *file pointer*, which marks the current position in the file and can be reset to any position

RandomAccessFile.java

method	purpose
`RandomAccessFile(String name, String mode)`	constructs a new random access file with the given name and mode (reading or writing or both)
`int read(byte[] b)`	Read some bytes from the current position in the file. The current position moves forward as the bytes are read.
`int readInt()`	Reads four bytes from the file and returns them as an `int`
`String readLine()`	Reads bytes from the file until a `\n` character is read and returns them as a `String`
`void seek(long pos)`	Sets the file pointer to the given position
`void write(byte[] b)`	Writes the given bytes to the file at the current position.

RandomAccessFile.java

Check out the JavaDocs for the full API. The `RandomAccessFile` has several additional methods for reading and writing that make dealing with more structured data (primitives, Strings, objects) easier than with Streams, Readers, and Writers.

Live Coding

1. Read the contents of a file using a Reader and an Iterator following the design process:
    i. Understand the problem
    ii. Formalize the interface
    iii. Write tests
    iv. Implement the behavior
2. RandomAccessFile