latex input: mmd-article-header Title: Java I/O Notes Author: Ethan C. Petuchowski Base Header Level: 1 latex mode: memoir Keywords: Java, programming language, syntax, fundamentals CSS: http://fletcherpenney.net/css/document.css xhtml header: copyright: 2014 Ethan Petuchowski latex input: mmd-natbib-plain latex input: mmd-article-begin-doc latex footer: mmd-memoir-footer

Refs

- Oracle's Java Tuts
- More Oracle Tuts
- *Java I/O* 2nd ed., Elliotte Rusty Harold, O'Reilly, 2010.

Intro

- I/O Streams handle byte-oriented I/O
    - Streams are *not* intended for reading or writing text
- Readers & writers handle character I/O
    - This *are* intended for reading and writing text
- You can redirect `System.in|out|err` using `System.set|In|Out|Err(Stream)`

I/O Streams

- A stream is an ordered sequence of bytes of indeterminate length
- **An object from/to which we can read/write bytes** (one item at a time)
    - E.g. a file, network, device, other program, or block of memory
- There are **Input** and **Output** Streams
- Could be any *kind* of data (bytes, primitives, objects, etc.)
- Streams can manipulate & transform the data
- There are 60+ stream types
- You need to compose stream filters together to get the type of reads you want
    - E.g. you want to do buffered reads of type `double` from a file

```
DataInputStream din =
        new DataInputStream(
            new BufferedInputStream(
                new FileInputStream("filename.dbls")));
```

This will allow you to use `readDouble()` which is read from the file through a black-box buffering mechanism that you can expect will speed things up.

- To **write text to a file** use a `PrintWriter`

```
PrintWriter out = new PrintWriter("file.txt");
out.println(aLine);
out.printf("%s%n", more);
out.close();
```

Ints and Bytes and streams

- When a stream's method says it returns or accepts `int`s, it's probably assuming they're all in the range $[0, 255]$ and internally converting them to/from `byte`s
  - e.g. `InputStream.read()` returns an `int` which is `-1` if the end of the stream is reached; but really it's reading the next `byte` of data
  - e.g. `OutputStream.write(int)` expects an `int` *within* $[0, 255]$
    - If you pass in an `int` outside the range it ignores the top 24 high-order bits

InputStream

- The `abstract` "superclass of *all* classes representing an input stream of bytes".
- Subclassers *must* define a method for returning the next byte
- You probably don't want to operate on a `byte` level, so use a subclass that does what you want
- `skip(long)` is often faster than `read`ing and then discarding

Interface

```
interface AutoCloseable {
    void close() throws Exception
}

interface Closeable extends AutoCloseable {
    void close() throws IOException // close & release system resou
}

abstract class InputStream implements Closeable {
    int  available()    // minimal est. of #bytes available w/o blo
    void close()        // from Closeable
    abstract int read() // read next byte, block till read
    int  read(byte[])   // read into buffer, return #bytes received
    int  read(byte[], offset, len) // read ≤ len bytes
    long skip(long n)   // skip & discard next `n` bytes

    /* not always supported */
    void mark(int)      // mark position for reset()
    void reset()        // go to last mark()
}
```

Reads can block

- if you're trying hard not to block, you can try

  ```
  int bytesAvailable = in.available();
  if (bytesAvailable > 0) {
      byte[] data = new byte[bytesAvailable];
      in.read(data);
  }
  ```

  - Alternatively, you could delegate blocking calls to another thread
  - Also, I think Java's newer NIO library is for that purpose, though I know nothing about it.

OutputStream

- Analogous to `InputStream` above
- Using `write(byte[] arr)` instead of `write(int b)` will be *WAY* faster
- To subclass it, you *have* to implement `write(int b)`, but you'll probably also *want* to implement `write(byte[] data, int offset, int length)` to improve performance

Interface

```
/** "a destination for data that can be flushed" */
interface Flushable {
    void flush() // writes buffered output to underlying stream
}

abstract class OutputStream implements Closeable, Flushable {
    void close() // from Closeable, also flushes
    void flush() // from Flushable

    /* write byte to strm, block till written */
    void write(byte[])
    void write(byte[], offset, len)

    // It's an `int` rather than `byte` because it's unsigned and Ja
    // doesn't have unsigned types. Don't write values > 255 because
    // all libraries handle that the same way.
    abstract void write(int aByte)
}
```

FilterOutputStreams

- `ProgressMonitorInputStream` automatically pops up a `javax.swing` dialog box with a progress meter if the operation is taking too long
- `PushbackInputStream` allows you to "`unread`" data *into* the stream, which will then be the first data you `read` back out

Network streams

URL

- You create a `URL(...)` with parameters
    1. `String url`
    2. `String protocol, String host, String file`
    3. `URL context, String url` (retrieves relative `url` to the given `context`)
- Now you can get `InputStream in = url.openStream(); in.read();` etc.
    - You probably want to buffer it though
- Now you can just go ahead and read the page, because Java takes care of sending the `GET` request etc.
    - This is *unlike* just using a `Socket`

Data streams

- Read/write Strings, ints, doubles, and other primitive types

Streams in Memory

- `SequenceInputStream` --- chain input streams together so they appear as a single stream. Each is `closed` as its end is reached
- `ByteArray[In|Out]putStream` --- read/write from/to a byte array
- `Piped[In|Out]putStream` --- you create both and connect them to each other, then you can write from one and read from the other, potentially in different threads, in a produce/consumer-type arrangement

Compressing Streams

- E.g. for un/compressing G/Zips and JAR files

Cryptographic Streams

Example using `MessageDigest`

From (Java I/O pg. 225), calculate SHA-1 hash of given webpage

```
public class URLDigest {
    public static void main(String[] args)
                throws IOException, NoSuchAlgorithmException {
        URL u = new URL(args[0]);
        InputStream in = u.openStream();

        // choose which one-way hash algorithm
        MessageDigest sha = MessageDigest.getInstance("SHA");

        // read the whole input, one chunk at a time
        byte[] data = new byte[128];
        while (true) {
            int bytesRead = in.read(data);
            if (bytesRead < 0) break;

            // feed each chunk to the hasher
            sha.update(data, 0, bytesRead);
        }

        // retrieve the resulting hash
        byte[] result = sha.digest();

        // print the hash one byte at a time
        for (int i = 0; i < result.length; i++) {
            System.out.print(result[i] + " ");
        }
        System.out.println();

        // print the hash as a giant integer
        System.out.println(new BigInteger(result));
    }
}
```

- using `Digest[In|Out]putStream` is basicaly the same, except you don't have to call `update`
- Encryption is also roughly the same

Object streams

- `Object[In|Out]putStream`
- Support I/O of objects
- Objects that support serialization
    1. implement `Serializable` (more in "[Serialization][]")
    2. have a no argument constructor
    3. mark non-zerializable fields "`transient`"
- This way of serializing is very slow
- When you serialize an object like this, it cannot be garbage collected until the stream is `reset` or `closed`

- So one should "save the entire state only when the entire state is available and then close the stream immediately." (Java I/O, 265)

E.g.

```
out.writeObject(obj);
Object obj = in.readObject();
```

Readers and Writers

- Readers take their input as bytes from streams, and convert it into `chars` according to the specified encoding format (writers do vice-versa)

Scanners

```
s = new Scanner(new BufferedReader(new FileReader("xanadu.txt")));
```

- A `Scanner` breaks input into *tokens*
- By default it uses whitespace to separate tokens
- To change it to `comma with optional following whitespace`, use:

```
s.useDelimiter(",\\s*");
```

New I/O: Buffers and Channels

- For nonblocking I/O (since 1.4), "buffers are filled with data from the channel and then drained of data by the application" (Java I/O, pg. 13).
- This API is not always helpful, but it *is* helpful for
  - Network servers with tons of simultaneous clients
  - Repeated random access to parts of large files

Buffers

- There is a buffer class for each primitive data type
- These allow you to perform I/O operations directly on the files without copying them into RAM first
- The fastest way to use these is *heavily* task and OS specific
- Buffers have a *fixed* size

Files and Paths

class File

- "An abstract representation of file and directory **pathnames**."
- Its instances are *immutable* --- i.e. once created, the abstract pathname represented by a `File` object will never change
- Remove, rename, create directory find `mtime`, file-system stuff
- Can be "*relative*" or "*absolute*"
    - Relative paths begin searching in the `cwd`
- Consists of two components
    1. A *prefix string* (e.g. "`/`" on Unix)
        - "*Relative*" pathnames have *no prefix*
    2. A *sequence* of zero or more *names*
- By default the classes in the `java.io` package always resolve *relative* pathnames against the *current user directory*. This directory is named by the *system property* `user.dir`, and is typically the directory in which the *Java virtual machine* was *invoked*.

interface Path

- Object representing the system absolute path to a resource
- You can't really *do* anything with this except manipulate the *path*, but you can do `toFile()` to turn it into a `File` object

Serialization

To implement Java's default serialization

```
public class MyClass implements Serializable { ... }
```

To exclude instance variables from serialization

```
transient private String dontSerialize;
```

The default Java serialization mechanism will turn data members into a set of bytes in a stream, which can be passed across the network (e.g. via `Sockets` or `RMI`) or stored to the local filesystem.

To implement a *custom* serialization policy, use

```
public class Myclass implements java.io.Externializable { ... }
```

RMI (Remote Method Invocation)

You tag your objects as serializable, and then you can invoke Java methods and the associated argument and return objects are passed across the network by the RMI subsystem.

Other

Memory-mapping files

- Use this to make file operations faster
- For just like sequential *reading* or something, it'd be simpler & roughly the same speed to just go with a `BufferedInputStream`

Example

```
try (FileChannel channel = new FileChannel.open(pathObject)) {
    int length = (int) channel.size();
    MappedByteBuffer buffer =
            channel.map(FileChannel.MapMode.READ_ONLY, 0, length);

    for (int byteIWant : bytesIWant)
        bytesIWanted.add( buffer.get(byteIWant) );
}
```

GSON

5/20/14

Source hosted on Google Code

- Provides simple `toJson()` and `fromJson()` methods to convert Java objects to JSON and vice-versa
- Works on "arbitrarily complex" Java objects including pre-existing objects that you don't have source-code for
- Support for generics
- Fields present in the JSON but not the object to be deserialized to are *ignored*
    - This is a "feature" because it makes things more flexible
- In general, this looks like a simple and useful tool
- You use `GsonBuilder()` instead of `Gson()` to allow more customization