The Java NIO Classes

This document provides a brief overview of some Java classes and methods which will be useful for working with files in the IN2011 coursework.

Paths

(See Tutorial: http://docs.oracle.com/javase/tutorial/essential/io/pathClass.html).

A path locates a file within a file system. You need to take care to distinguish between *relative* and *absolute* paths. An absolute path is one which starts at the root of the file system. The root of a Unix file system is `/`. Windows file systems can have multiple roots, distinguished by different drive letters; an example would be `C:\`. A relative path is one which does *not* start from the root.

java.nio.file.Path

This is the key class (actually, it's an interface) for locating files. To create a new Path object you can use one of the static get methods provided by java.nio.file.Paths. For example, the following code creates a Path object representing the relative path `fred`:

```
Path myPath = Paths.get("fred");
```

At some point you may need to convert a relative path to an absolute path. Here is a good way to do that:

```
myPath = myPath.toAbsolutePath().normalize();
```

This not only converts the path to an absolute path but also converts it into "normal form". (This is useful because different path expressions may actually denote the same location in a file system. For example, `/home/troy` and `/home/./troy/../troy`.)

Note that such a conversion only makes sense if there is a well-defined notion of "the current directory" (since the meaning of a relative path is "continue from the current directory"). When you run a Java program at the command line, the current directory (also known as the *working* directory) is inherited from the shell in which you are working. IDE's have their own ways of configuring the working directory.

The Path interface also provides a number of other methods you may find useful: consult the API documentation for details.

Reading, Writing and Creating Files

(See Tutorial: http://docs.oracle.com/javase/tutorial/essential/io/file.html).

A Path is essentially just a way of *naming* a file, it is not the file itself. In fact, a Path may name a file which doesn't even exist. To read and write data to a file you can create java.io.InputStream and java.io.OutputStream objects. These objects provide raw byte streams: you should *not* use Readers and Writers, since your http server needs to work with arbitrary file types, not just text files. (You

could also use NIO Channels and ByteBuffers instead of streams, if you prefer, but that is not covered in this document. See the above Tutorial link if you are interested.)

To read data from a file, create a java.io.InputStream object using the static newInputStream method provided by java.nio.file.Files. For example:

```
InputStream is = Files.newInputStream(myPath);
```

(See **Reading and Writing with Streams** below for tips on reading data from the stream.)

To write data to a file (not needed for the coursework), create a java.io.OutputStream object:

```
OutputStream os = Files.newOutputStream(myPath);
```

This will *create* a new file if the file named by `myPath` does not already exist. The newOutputStream method also accepts additional "OpenOptions" parameters which control (for example) what happens if the file *does* already exist (see the above Tutorial link for details).

Reading and Writing with Byte Streams

In its basic form, reading data from an InputStream is quite straightforward. For example, a loop such as this will read all bytes in sequence from the input stream `is`:

```
int b = is.read();
while (b != -1) {
   // do something with byte b here
   b = is.read();
}
```

However this is not very efficient. It is better to read multiple bytes at a time, using a different read method which has this signature:

```
public int read(byte[] b) throws IOException
```

See the java.io.InputStream API documentation for details.

Writing a byte `b` to an output stream `os` is also straightforward:

```
os.write(b);
```

As you might guess, there are also alternative write methods which allow a whole array of bytes to be written in one go. See the java.io.OutputStream API documentation for details.

Take care to ensure that your code closes any streams it creates for reading and writing to files when it has finished with them.

Note: to simplify the descriptions, exceptions have been ignored above. You will need to catch and handle exceptions as appropriate. See comments in Section 1.1 of the Part 2 coursework document (subsection **Robustness and Error Reporting**).