I/O

streams and scanner

Java provides a number of I/O libraries for the programmer ranging from classes to represent various types of streams to classes representing files and paths.

Streams

Java's I/O classes are based on the abstract notion of a *stream*. A stream represents a physical device which allows for a flow of data. Data can be read from a stream or written to a stream. Java's use of this abstract notion in the design of its I/O classes allows for a very flexible library. For instance, a stream can be used to read from a keyboard, a file, a network, etc. without concern for the differences underlying these operations. The same is true for writing to a stream. Java's provided stream classes can be separated into two types: **byte streams** and **character streams**.

Byte streams are generally used in scenarios where the data read or written does not need to be encoded or decoded in any way. That is, during the transmission of data, there is no interest in what the data "says". Each byte is simply either read or written without concern for how the byte should be interpreted. An example of this would be reading or writing binary data to a file. The most generalized streams for dealing with bytes are `InputStream` and `OutpuStream`.

Character streams are generally used in scenarios where the data read or written is enconded and decoded in a particular way. Where byte streams read and write bytes, character streams read and write characters. Of course, underlying a character stream is some sort of mechanism for reading and writing bytes, but these bytes are interpreted automatically by the character stream. A character stream may be thought of as a layer on top of a byte stream for reading and writing textual data. The most generalized streams for dealing with characters are `Reader` and `Writer`.

In both byte streams and character streams the more generalized classes provide functionality for **unbuffered** reading and writing. This means that one byte or character is read or written at a time using the `read()` or `write()` methods, respectively. In some scenarios, this functionality is useful; however, often times a program will see gains in efficiency by using **buffered** streams where a token – i.e. a word – or an entire line may be read or written to the stream in a single abstract operation. However, when a stream is buffered, it is necessary to *flush* the buffer at the appropriate times. For instance, with some sort of buffered output stream, one could write to the buffer, make changes, etc. And then, when ready, would flush the buffer finally writing all the data in the buffer to the stream.

Buffered I/O

Buffered stream classes in Java may be used to wrap unbuffered stream classes. There are many classes which provide the functionality for buffered I/O. Altogether, there are close to 50 streams classes. For the proceeding examples, we will concern ourselves with character streams and particularly those which are considered most convenient for reading and writing textual output to and from the console and files.

When writing textual output to the console, we are used to seeing the following statement:

```
System.out.println("Some text to the screen.");
```

`System.out` is actually an object of type `PrintStream` which inherits from `OutputStream`. From above, we know that an `OutputStream` deals with bytes, not characters. Though, the `PrintStream` class does the conversion of bytes to characters. This can be seen when we make a call to `System.out` and characters are displayed on the screen, not bit patterns. This is a convenience class for dealing with bytes that was used before the introduction of character streams to the Java language. The preferred class for writing character output since the introduction of character streams is the `PrintWriter` class.

```
    public static void main(String [] args) {
        PrintWriter screen = new PrintWriter(System.out);
        screen.println("Some text to the screen.");
        screen.close();
    }
```

Above, we see the declaration of a `PrintWriter` object which takes as an argument the `PrintStream` object: `System.out`. The `PrintWriter` object `screen` now writes its output to the standard output (the screen) as do our previous calls to `System.out`. Note that it is important that you always close a stream after using it. Not doing so can leak system memory. The same is true when reading from a stream.

The `System` class provides an `InputStream` to represent the standard input (the keyboard) which is accessed with a call to `System.in`. However, as mentioned in the last section, an `InputStream` reads unbuffered bytes of data. In general, when reading from the standard input, we want to read characters as opposed to bytes. And, furthermore, we generally don't want to read one character at a time but read in a buffered fashion – i.e. a word or line at a time. So, first we must wrap `System.in` in a class that converts the byte stream to a character stream. This can be done with a class called `InputStreamReader` which is a type of `Reader`:

```
   InputStreamReader stdin = new InputStreamReader(System.in);
```

However, the `InputStreamReader` is not buffered, so we then wrap our object, `stdin` in a `BufferedReader` so that we may read more than a character at a time.

```
   BufferedReader keyboard = new BufferedReader(stdin);
```

This gives us our new `BufferedReader` object which allows us to read an entire line at a time from the keyboard. Also, the creation of the `BufferedReader` can be written in shorthand:

```
    public static void main(String [] args) throws IOException {
        PrintWriter screen = new PrintWriter(System.out);
        BufferedReader keyboard = new BufferedReader(new InputStreamReader(System.in));

        screen.println("Enter some text: ");
        String s = keyboard.readLine();
        screen.println("You entered: " + s);

        screen.close();
        keyboard.close();
    }
```

The first thing to note about the code above is the `throws` statement following the parameter list. This statement (which should be somewhat familiar) declares that `main()` is performing an action that could result in an error. Namely, the action in question is: reading from the standard input stream – i.e. the keyboard – and closing that stream:

```
   keyboard.readLine();
    ...
   keyboard.close();
```

And, in fact, this line could be put in a `try-catch` statement and handled in whatever way is appropriate. In a case where the `Exception` is handled, the `throws` statement could be eliminated:

```
public static void main(String [] args) {
        ...
    screen.print("Enter some text: ");
    try {
        String s = keyboard.readLine();
        screen.println("You entered: " + s);
    } catch(IOException) {
        System.err.println("There was an error reading from the keyboard.");
    } finally {
        keyboard.close();
    }
        ...
}
```

The `finally` statement after the `catch` statement forces the program to try to close the stream even if an `Exception` is thrown. This will be covered in more detail after covering `Exception`s more generally.

All things considered, it is generally wise to make calls in a `try-catch` block and `close()` in a `finally` when writing a program that invokes methods that could potentially throw `IOException`s. However, this is not always convenient, in which case the `throws` declaration may be acceptable.

At any rate, if the `IOException` is not handled or declared we see the following compiler error:

```
Foo.java:12: error: unreported exception IOException; must be caught
or declared to be thrown
        keyboard.readLine();
                         ^
Foo.java:17: error: unreported exception IOException; must be caught
or declared to be thrown
        keyboard.close();
                      ^
2 errors
```

Another thing to notice is that the code above generates an unexpected output. Namely, the prompt is displayed after reading a line from the keyboard:

```
[user@notnotbc]$ java SomeClass
Hello
Enter text:
You entered: Hello
[user@notnotbc]$
```

The reason is because the `PrintWriter` object, `screen`, is buffered and it is at the discrepancy of the JVM to decide when to *flush* the buffer – i.e. actually send the text to the screen. This can be overridden by a call to the `flush()` method which is common to all streams.

```
    ...
screen.println("Enter some text: ");
screen.flush();
String s = keyboard.readLine();
screen.println("You entered: " + s);
screen.flush();
    ...
```

With the change above, we get the following output:

```
[user@notnotbc]$ java SomeClass
Enter text:
Hello
You entered: Hello
[user@notnotbc]$
```

Of course, it may become annoying to constantly make explicit calls to `flush()`. In this case, one can simply make the `PrintWriter` auto-flushable by passing the appropriate parameter to the constructor. This way, the buffer is flushed with each call to `println()`, though not flushed automatically with calls to `print()`. (Note that `System.out` is auto-flushed).

```
// second argument triggers auto-flushing
PrintWriter screen = new PrintWriter(System.out, true);
```

Keep in mind that the JVM ultimately has the ability to optimize code as it sees fit. This means that input and output coming from different streams may, at times, be flushed in an unexpected order. In most situations, however, making explicit calls to `flush()` or having an auto-flushing stream will suffice.

Since Java's stream classes are very generalized, these same classes and methods can be used for file I/O. Simply pass in a `File` object as an argument to the constructor, as opposed to say, `System.out` or `System.in`.

```
public static void main(String [] args) throws FileNotFoundException {
    File f = new File("output.txt");
    PrintWriter outfile = new PrintWriter(f);
    outfile.println("Writing to a file!");
    outfile.close();
}
```

The output from the code above can be seen by displaying the contents of the file `output.txt`:

```
[user@notnotbc]$ cat output.txt
Writing to a file!
[user@notnotbc]$
```

Notice that opening a file to write to it may throw a `FileNotFoundException`. When we created a `PrintWriter` with `System.in`, there was no concern for `Exceptions`. (Check the initial code above). The code above automatically creates the file `output.txt` if it does not exist. That is, as soon as one writes to a stream that is associated with a non-existent file, the file is created and the data is written to that file. This begs the question: why throw a `FileNotFoundException` if it simply creates the file anyway? From the Java 8 API:

```
FileNotFoundException - If the given file object does not denote an existing,
writable regular file and a new regular file of that name cannot be created,
or if some other error occurs while opening or creating the file
```

In the case above, if the file already exists it will be overwritten, though streams may also be directed to append data to the file, as opposed to overwriting it.

This is not the case when opening a file for input. If the file does not exist, one cannot read it. Whereas, if a file does not exist, one can still write to it by first creating it, as mentioned above.

```
    public static void main(String [] args) throws IOException {
        File out = new File("output.txt");
        PrintWriter outfile = new PrintWriter(out);

        File in = new File("input.txt");
        BufferedReader infile = new BufferedReader(new FileReader(in));

        outfile.println("Writing to a file!");
        String s = infile.readLine();
        outfile.println("Read from input.txt: " + s);

        outfile.close();
        infile.close();
    }
```

If the file `input.txt` does not exist, the code outputs that following:

```
    Exception in thread "main" java.io.FileNotFoundException: input.txt (No such file or directory)
        at java.io.FileInputStream.open0(Native Method)
        at java.io.FileInputStream.open(Unknown Source)
        at java.io.FileInputStream.<init>(Unknown Source)
        at java.io.FileReader.<init>(Unknown Source)
        at SomeClass.main(Foo.java:10)
```

However, if `input.txt` does exist, we do not get the error above.

```
    [user@notnotbc]$ cat input.txt
    This is in a file!
    [user@notnotbc]$
```

If the contents of the `input.txt` are as shown above, the corresponding contents of `output.txt` are as shown below after executing the code above.

```
    [user@notnotbc]$ cat output.txt
    Writing to a file!
    Read fom input.txt: This is in a file!
    [user@notnotbc]$
```

Notice that since we are using `readLine()` we must now handle or declare an `IOException`. Since `FileNotFoundException` is a type of `IOException`, we can simply declare the `IOException` which covers all subclasses.

Finally, the code above is merely a sample to illustrate how streams work. As mentioned previously, there are many different types of streams, each with their own benefits. Also, robust code will take measures to avoid and recover from various `IOException`s by checking for file existence and nesting code in `try-catch` blocks.

Furthermore, steps should always be taken to be sure streams are being closed properly. In small programs, it is hard to see the consequences of keeping streams open, as all streams are automatically closed when a program terminates. However, imagine a very long-running program – i.e. server software – which may be constantly opening streams as it receives requests, but never closing them. This could lead to dramatic memory leaks.

Scanner

Another useful utility for reading from an input source is the `Scanner` class. `Scanner` provides another layer of abstraction on top of a character stream. Where a buffered character stream can read single characters or a line at a time, `Scanner` can read single tokens – i.e. a word – or read up until any delimiter among other things.

The code below opens a file for output using a `PrintWriter` and a file for input using a `Scanner`. The `Scanner` reads from the input file and the `PrintWriter` writes the corresponding data to the output file.

```
import java.io.*;
import java.util.*;

class Foo {
    public static void main(String [] args) throws FileNotFoundException {
        File out = new File("output.txt");
        PrintWriter outfile = new PrintWriter(out);

        File in = new File("input.txt");
        Scanner infile = new Scanner(in);

        if(infile.hasNextLine()) {
            String s = infile.nextLine();
            outfile.println("Read line from input.txt: " + s);
        }

        if(infile.hasNextInt()) {
            int i = infile.nextInt();
            outfile.println("Read int from input.txt: " + i);
        }

        outfile.close();
        infile.close();
    }
}
```

The `Scanner` class has a set of methods that allow a program to check if a particular type of data is available before attempting to read that data. This can be seen in the statements:

```
infile.hasNextLine()
 ...
infile.hasNextInt()
```

These methods are *state-independent*. This means that these methods will return valid information regardless of the state of the underlying stream or the data available. The complementary methods are:

```
infile.nextLine()
 ...
infile.nextInt()
```

These methods are *state-dependent*. This means that they may or may not throw an `Exception` depending on the state of the underlying stream or the data available. State-dependent methods are designed, in part, to reduce the clutter of exception-handling and validation. (A similar design pattern can be seen with `Iterators`). In fact, the entire `Scanner` class seriously reduces the need to handle `IOExceptions` as the class itself handles all of these underlying details “behind the scenes”. This can be seen, in part, in the `throws` declaration above. The only thing declared is a `FileNotFoundException` which is only relevant to the construction of the `Scanner`.

The set of *state-dependent* methods allow the `Scanner` to parse certain types of data from a `String` automatically. For instance, the method `nextInt()` reads a `String` from the stream and then attempts to convert it to an `int` value. If the data cannot be converted, an `InputMismatchException` is thrown. However, the complementary *state-independent* method provides an input validation resource to prevent this from happening.

Consider the following input file, `input.txt`:

```
[user@notnotbc]$ cat input.txt
This is a string!
One more string...
[user@notnotbc]$
```

After running the code above, the output file, `output.txt` contains the following:

```
[user@notnotbc]$ cat output.txt
Read line from input.txt: This is a string!
[user@notnotbc]$
```

As can be seen, only the first line was read from `input.txt` because the second attempt to read required a `String` containing a parseable `int` value. If the second line of `input.txt` is changed to an appropriate value, this will be mirrored in the output file.

```
[user@notnotbc]$ cat input.txt
This is a string!
12345
[user@notnotbc]$
```

After running the code above, the output file, `output.txt` contains the following:

```
[user@notnotbc]$ cat output.txt
Read line from input.txt: This is a string!
Read int from intput.txt: 12345
[user@notnotbc]$
```

Now we see the that the `int` value was read and written.

Finally, notice that `java.util.*` had to be imported. This is because the `Scanner` class does not really add much functionality in terms of I/O, it's simply a convenience class for parsing information from some input stream. For this reason, it's packaged in `java.util`.