

# Java Misc

---

## Book

Our book, *Core Web Programming*, includes lots of basic material on Java, exceptions, socket programming, and HTTP programming in particular.

Also, the online docs for Java APIs are at

<http://java.sun.com/j2se/1.4.1/docs/api/index.html>

Also, see Nick's Java Doc Fast page

<http://www.stanford.edu/class/cs108/JavaDocFast.html>

## Streams

InputStream -- read from

OutputStream -- write to

Reader/Writer variants -- use for text, so they can read/write strings and chars and take care of unicode conversion

"Layered" design -- wrap a stream around another to layer up effects.

e.g. `BufferedReader in = new BufferedReader(new FileReader( ... ));`

## Text Reading

Below is the standard incantation to read a text file.

We construct a `FileReader` object, that takes either a `File` object or a `String` filename.

We wrap that reader in a `BufferedReader`, which responds to a `readLine()` message which returns a `String`, or null if there is no more data.

`readLine()` recognizes the many different end-line conventions, and strips all the end-line chars before returning the string.

It's polite to `close()` the reader when done. This may help free up resources in the VM. However, code that forgets to `close()` will generally still work.

## Text Reading Code

```
public void readLines(String fname) {
    try {
        // Build a reader on the fname, (also works with File object)
        BufferedReader in = new BufferedReader(new FileReader(fname));

        String line;
        while ((line = in.readLine()) != null) {
            // do something with 'line'
            System.out.println(line);
        }

        in.close(); // polite (could use 'finally' clause)
    }
    catch (IOException e) {
        e.printStackTrace();
    }
}
```

```
}
}
```

## Text Writing

Writing is pretty similar.

We construct a `BufferedWriter` on a `FileWriter`

The writer responds to `print()` and `println()` messages to write strings and chars.

## Text Writing Code

```
public void writeLines(String fname) {
    try {
        // Build a writer on the fname (also works on File objects)
        BufferedWriter out = new BufferedWriter(new FileWriter(fname));

        // Send out.print(), out.println() to write chars
        for (int i=0; i<data.size(); i++) {
            out.println( ... ith data string ... );
        }

        out.close();        // polite
    }
    catch (IOException e) {
        e.printStackTrace();
    }
}
```

## Binary File Reading / Writing

Reading a binary file looks similar, but we use an `InputStream` or

`BufferedInputStream` instead of a `Reader`.

Send the `read(byte[] buff)` message that reads a number of bytes out of the stream and into an `byte[]` array.

Returns the number of bytes actually read, or -1 on EOF.

## Binary File Example

```
/*
 * Utility method that reads all of the bytes out of the given
 * file and writes them out the given output stream.
 */
private void sendFile(OutputStream out, File file) {
    try {
        FileInputStream in = new FileInputStream(file);
        byte[] buff = new byte[1000];
        int count;
        while((count = in.read(buff)) != -1) {
            out.write(buff, 0, count);
        }
        in.close();
    }
    catch (IOException e) {
        e.printStackTrace();
    }
}
```

}

## Exceptions

An exception is an error condition that arises at runtime -- it can stop the normal sequence of instructions and jump to an error handling section.

The error handling code may be in the same method, or the current method may exit immediately to jump to error handling code in an earlier method that called the current method.

Many stream operations can throw exceptions

In Java, the compiler will insist that most types of exception are caught by the code.

## throws clause

If code might throw an exception, it can be wrapped in a try/catch (below) to handle the exception locally.

Or, a method may add a "throws" clause to its prototype to indicate that it may generate an exception back to its caller.

e.g. `public void foo() throws IOException { ...}`

In that case, the caller must deal with the exception.

## try / catch

A try/catch block attempts to execute a block of statements. If an exception occurs in some method called from within the try block, then the catch() clause handles the exception.

Having intercepted the exception, the catch clause can handle it in any number of ways. The simplest strategy is to call `e.printStackTrace()` to produce some debugging output.

There can be multiple catch clauses, in which case they are searched from top to bottom, and the first that matches the exception gets it.

## try / catch example

Here, the `fileRead()` method uses a try/catch to catch the `IOException` internally and print an error message. Note that the `IOException` is caught inside `fileRead()`, so `IOException` is not declared in a "throws" in the method prototype.

```
public void fileRead(String fname) {           // NOTE no throws

    try {
        // this is the standard way to read a text file...
        FileReader reader = new FileReader(new File(fname));
        BufferedReader in = new BufferedReader(reader);

        String line;
        while ((line = in.readLine()) != null) {
            ...
            // readLine() etc. can fail in various ways with
            // an IOException      }
        }
    }
}
```

```

    // Control jumps to the catch clause on an exception
    catch (IOException e) {
        e.printStackTrace(); // on possible thing to do
        // or could ignore the exception
    }
}

```

## ClientSocket

Open a client socket

Use input/output streams

Note the manual use of flush() to send the output (vs auto-flush in perl)

We use a StringBuffer, since it does append() more efficiently, than a series of string = string + string operations.

```

Socket sock = new Socket("hostname", 80);
PrintWriter out = new PrintWriter(sock.getOutputStream());
BufferedReader in =
    new BufferedReader(new InputStreamReader(sock.getInputStream()));

out.print("GET / HTTP/1.0\r\n");
out.print("\r\n");
out.flush(); // ok, really send the output

// Read back everything into a single string
String line;
StringBuffer buff = new StringBuffer(2000);
while ((line = in.readLine()) != null) {
    buff.append(line);
    buff.append("\n");
}

String response = buff.toString();

```

## ServerSocket

Same concepts as in Perl, just different syntax

```

ServerSocket serv = new ServerSocket(port);

while (true) {
    Socket sock = serv.accept();
    BufferedReader in =
        new BufferedReader(new InputStreamReader(sock.getInputStream()));
    PrintWriter out = new PrintWriter(sock.getOutputStream());

    // .. deal with sock
    sock.close();
}

```

## URL

Built-in class

We'll use it for rel/absolute URL conversion

Can throw an exception on protocols it doesn't know

# ArrayList

See the Java section handout

## Java Regular Expressions

Regular expressions have just been added to Java with the 1.4 version. They are basically the same as Perl regular expressions, although the syntax for using them is more heavy than in Perl.

### Pattern

The Pattern class stores a regular expression pattern. (See the Pattern class docs)  
The regular expression chars work as in perl: `\w \s * +` etc. etc.

The regular expression is defined in a Java string, which has its own use of the backslash (`\`).

For that reason, each backslash needs a backslash in front of it so it goes through. To express for example, `\w`, in the Java string we have to write `"\\w"`.

Pattern of two `\w` words separated by whitespace

```
Pattern p1 = Pattern.compile("\\w+\\s+\\w+");
```

Pattern of "hello" followed by space followed by a word. Not case sensitive variant.

```
Pattern p2 = Pattern.compile("hello\\s+\\w+",
    Pattern.CASE_INSENSITIVE);
```

The Pattern.MULTILINE option allows the matching to span across lines

### Matcher

A Matcher runs a pattern on some text to find all the matching sections of text.

(See the Matcher class docs)

```
Matcher mat = pat.matcher(text);
```

Responds to `find()` to find successive matches

Responds to `group()` to return the matched text

Responds to `group(i)` to return the *i*th (..) sub-part of the matched text (like `$1`, `$2` ... in Perl)

### Example \w Extraction

Suppose we have "hello SOMEWORD" and we want to extract the word.

Could use `\S` instead of `\w` -- `\S` is any non-whitespace char

```
Pattern pat = Pattern.compile("hello\\s+(\\w+)");
```

```
Pattern pat = Pattern.compile("hello\\s+(\\S+)");    // \S alternative
```

```
Matcher mat = pat.matcher(text);
```

```
if (mat.find()) {
    return(mat.group(1));    // extract the word
}
else return("");
```

## Example .\*? Extraction

(See the Perl section handout for the discussion of .\*?)

Suppose we want to extract everything within {...}'s

We use parens ( ) to mark the text we want to extract

We \ the { and } because they have their own meaning in regular expressions

```
Pattern pat = new Pattern.compile("\\{(.*)\\}");
```

```
Matcher mat = pat.matcher(text);
```

```
while (m.find()) {
```

```
    System.out.println(mat.group(1));
```

```
}
```