

Java Sockets

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/1.3/docs/api/index.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();
    }
}
```

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.

Many stream operations can throw exceptions

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

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 can intercept the exception stack unwind.

Having intercepted the exception, the catch clause can handle it in any number of ways (see below).

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.

If the catch clause matches the exception, it halts the stack unwind and executes the matching catch. Otherwise, the exception keeps going up the call stack. If you want a catch() clause (below) that catches **everything**, declare it to catch `Throwable`. More often, you just catch some specific exception that you care about, such as `IOException`.

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 use of `flush()` to send the output (vs buffering)

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 earlier Java handout