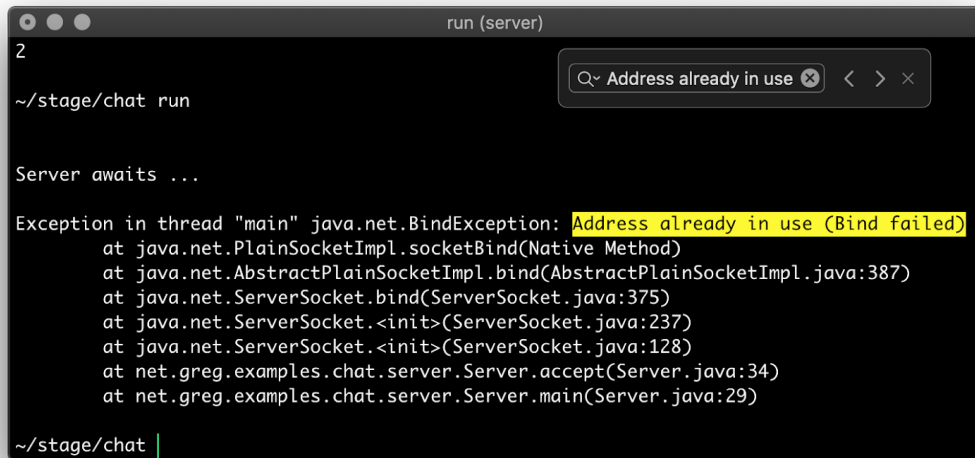


Should the Server hang, or if it is mistakenly dismissed via CTRL-Z instead of CTRL-C, the servers' process continues to claim the port; successive attempts to launch another image of the that server reports a port-bind fail:

A terminal window titled "run (server)" with a dark background. The prompt is "~ /stage/chat". The user has entered "run", and the terminal shows "Server awaits ...". Then, an exception is thrown: "Exception in thread \"main\" java.net.BindException: Address already in use (Bind failed)". The stack trace follows, showing the path from the main method to the bind operation. A search bar at the top right of the terminal shows the text "Address already in use".

```
run (server)
2
~/stage/chat run

Server awaits ...

Exception in thread "main" java.net.BindException: Address already in use (Bind failed)
    at java.net.PlainSocketImpl.socketBind(Native Method)
    at java.net.AbstractPlainSocketImpl.bind(AbstractPlainSocketImpl.java:387)
    at java.net.ServerSocket.bind(ServerSocket.java:375)
    at java.net.ServerSocket.<init>(ServerSocket.java:237)
    at java.net.ServerSocket.<init>(ServerSocket.java:128)
    at net.greg.examples.chat.server.Server.accept(Server.java:34)
    at net.greg.examples.chat.server.Server.main(Server.java:29)

~/stage/chat |
```

The Bash function shown on the next page - named 'cullany' - kills the errant process (provided you supply a unique identifier to the functi

```
run (server)
696
697 cullany() {
698
699 | ps auxx | grep "$1" > "$1"
700
701 while IFS=' ' read -r line || [[ -n "$line" ]]; do
702
703     export PID=2
704     PID="$(echo $line | cut -d " " -f $PID)"
705
706     sudo kill -9 "${PID}"
707
708 done < "$1"
709
710 rm $1
711 }
712
713
```

Line 699 redirects the output of the canonical ps utility into a readout file named after the grepped-for term (the unique descriptor for the process that you want killed).

Lines 701-8 are a while loop that finds in the readout file, the PID of (potentially many matches found by the ps utility. Currently, the PID is the second space-delimited column on a MAC's reporting of the ps utility.

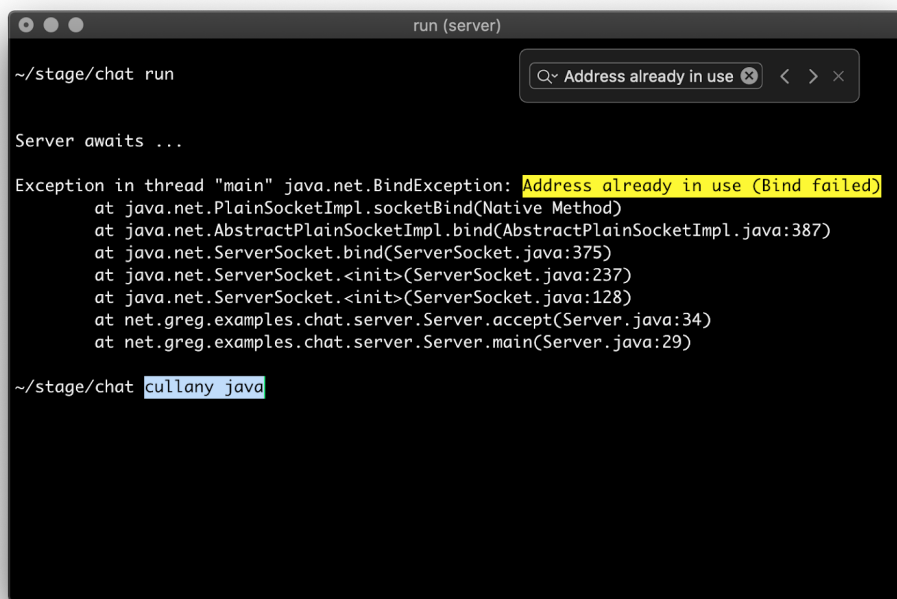
The while loop uses the cut utility to excise the PID value(s) to kill (again, located in column two).

Line 708 injects the eponymously-named file into the looping logic. Recall that the file is after the search term you specify in a CLI call to the cullany function.

Line 706 leverages the protected 'kill' utility that needs temporary sudo escalation, hence a prompt for admin rights is mandated.

Line 710 cleans-up.

The following example shows a common use case. You've start the server a second time, without first stopping the first image - along with the remedy: Kill all processes that contain the descriptor 'java' (clients included, as they are useless w/o the server).



A terminal window titled "run (server)" with a search bar at the top right containing the text "Address already in use". The terminal shows the command `~/stage/chat run` being executed. The output indicates the server is awaiting connections, but then an exception occurs: `Exception in thread "main" java.net.BindException: Address already in use (Bind failed)`. The stack trace shows the error originates from `java.net.PlainSocketImpl.socketBind` and propagates through `java.net.ServerSocket.bind`, `java.net.ServerSocket.<init>`, and finally to the `main` method of `net.greg.examples.chat.server.Server`. At the bottom of the terminal, the command `~/stage/chat cullany java` is entered.

```
run (server)
~/stage/chat run

Server awaits ...

Exception in thread "main" java.net.BindException: Address already in use (Bind failed)
    at java.net.PlainSocketImpl.socketBind(Native Method)
    at java.net.AbstractPlainSocketImpl.bind(AbstractPlainSocketImpl.java:387)
    at java.net.ServerSocket.bind(ServerSocket.java:375)
    at java.net.ServerSocket.<init>(ServerSocket.java:237)
    at java.net.ServerSocket.<init>(ServerSocket.java:128)
    at net.greg.examples.chat.server.Server.accept(Server.java:34)
    at net.greg.examples.chat.server.Server.main(Server.java:29)

~/stage/chat cullany java
```

```
run (server)

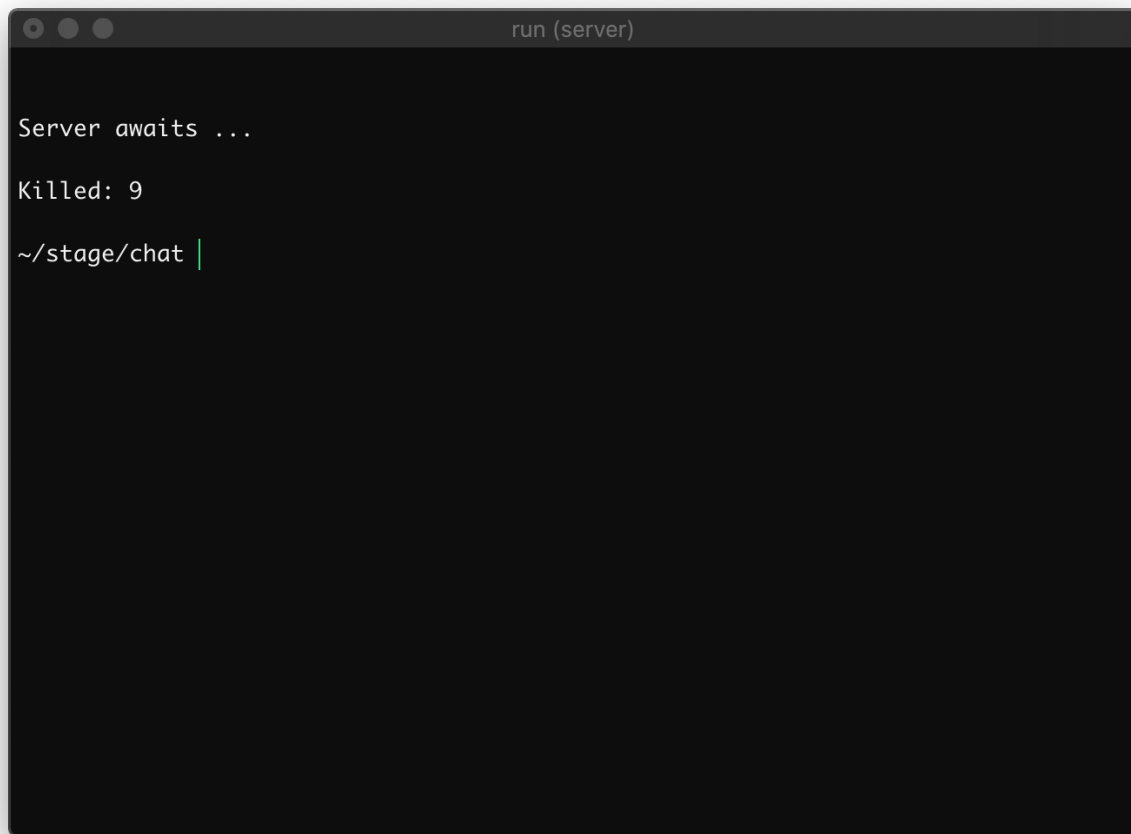
Server awaits ...

Exception in thread "main" java.net.BindException: Address already in use (Bind
failed)
    at java.net.PlainSocketImpl.socketBind(Native Method)
    at java.net.AbstractPlainSocketImpl.bind(AbstractPlainSocketImpl.java:38
7)
    at java.net.ServerSocket.bind(ServerSocket.java:375)
    at java.net.ServerSocket.<init>(ServerSocket.java:237)
    at java.net.ServerSocket.<init>(ServerSocket.java:128)
    at net.greg.examples.chat.server.Server.accept(Server.java:34)
    at net.greg.examples.chat.server.Server.main(Server.java:29)

~/stage/chat cullany java
Password:
kill: 11142: No such process

~/stage/chat
```

Any terminal w/ the already-running server process will show this message: Killed: 9



```
run (server)

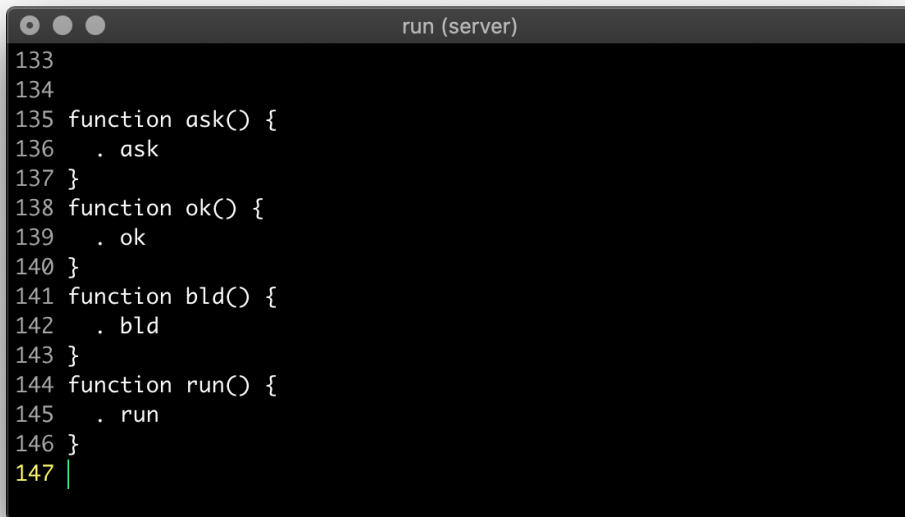
Server awaits ...
Killed: 9
~/stage/chat |
```

The following functions in the canonical `~/.bashrc` file allow you to eliminate supplying

the keyword ‘source’ or the period [.] when calling the same-named script in your local directory, no matter where it may be located. This is because of how the command interpreter for Bash works.

The command interpreter searches for a match to your command in a list of reserved words, then for aliases/functions in the canonical ~/.bashrc file before looking for your script.

All of the lifecycle commands are covered by the next few entries in the canonical ~/.bashrc file:



```
133
134
135 function ask() {
136     . ask
137 }
138 function ok() {
139     . ok
140 }
141 function bld() {
142     . bld
143 }
144 function run() {
145     . run
146 }
147 |
```

Without these entries, just prepend the keyword ‘source’ or the period to your command.

The lifecycle calls are:

bld - Maven builds the Chat Server, the Chat Client, and its Proxy.

run - Starts Chat Server (as an executable Shade JAR, sorry Spring boot get the boot).

ask - Starts one (of any number of) client processes.

The three compilation units achieve graceful cooperative interdependence by managing their respectful lifecycle situations, which imposed upon them by the nature of the protocols inherent to socket programming.

Mainly, the lifecycle oddities are handled (when both possible AND appropriate) via the use of try-catch blocks. Due to scoping issues that nested classes impose, lexical blocks (like try-catch) introduce inconsistencies that preclude their use.

Some chatter like:

”referents of inner classes must be final or effectively final”

Maybe lambdas are the solution here?

Another failed (and turns out unneeded) attempt to tame the server’s behavior to tolerate a client’s untimely departure involves the JVM’s Runtime shutdown hooks.

Two implementations are provided.

They both involve the clients’ attempt(s) to “wave goodbye” to the server by scaffolding a port and signaling the server that it has to depart. Lambda and Nested Class approaches are implemented.

The shutdown hook approach is a hack, as it puts the responsibility of the Server’s health on the Client.

Additionally, shutdown hooks must finish quickly, and may not get a chance to start, let

alone finish completely.

(at least it's an illustrative look at shutdown hooks)

Look at the Client.java class, Lines 105-60.

The Client has both a dedicated DataOutputStream, and a dedicated DataInputStream, as does the Server.

All communication between the Client and the Server happens via sockets - A Server Socket for the Server, and a plain Socket for each Client.

When the Server accepts a Client's request, it delegates communication to a Proxy for that Client, named ... ClientProxy, which is a thread (implements Runnable). This Client proxy is granted its own port and dedicated I/O Streams. Finally it is registered with the Server's internal List of registrants.

This is not highly scalable.

The messaging logic is delegated to the Proxy, appreciably lightening the Server's burden.

All three components/classes contain threads, the implementation of which all run in a tight loop. To make each component fault-tolerant, each components' canonical 'run()'

method swallows various exceptions.

Client.java

The read and write threads each swallow:

EOFException, SocketException, IOException

ClientProxy.java

The sole thread swallows:

EOFException, SocketException, IOException

Server.java

The sole thread (process) swallows:

EOFException

As the interplay of several Clients/Proxies with the Server are observed, you will see a scaffolding event per unique socket connection, followed by exchanges. The exchanges all bear a unique identifier, and there are only three exchanges per connection, after which the connection is passivated.

This is by design.

In the screenshots of Server Events (below), you'll see that each Client has a distinct port assigned to each of its requests.

Similarly, in the screenshots of Client/Proxy Events (below), you'll see that each Client has a distinct port assigned to it, and that because there are two Clients, their three requests interleave, with the ServerSocket repeating/reporting each Client/Proxy socket identifier three times, once per (3) requests, per client socket.

This is because the Clients' respective Proxies are threads, the allotted runtime slots of which are subject to the whims of the JVM's Thread Scheduler.

```
run (server)

~/stage/chat run

Server awaits ...

Happens Only ONCE Per Client Connection: Socket[addr=/127.0.0.1,port=51405,localport=1234]

requests.size(): 1

Happens Only ONCE Per Client Connection: Socket[addr=/127.0.0.1,port=51406,localport=1234]

requests.size(): 2

    server socket: Socket[addr=/127.0.0.1,port=51405,localport=1234]
client socket meta: 51405
    payload: 2020.02.12.23.38.17

    server socket: Socket[addr=/127.0.0.1,port=51406,localport=1234]
client socket meta: 51406
```

```
run (server)

    server socket: Socket[addr=/127.0.0.1,port=51406,localport=1234]
client socket meta: 51406
    payload: 2020.02.12.23.38.18

    server socket: Socket[addr=/127.0.0.1,port=51405,localport=1234]
client socket meta: 51405
    payload: 2020.02.12.23.38.19

    server socket: Socket[addr=/127.0.0.1,port=51406,localport=1234]
client socket meta: 51406
    payload: 2020.02.12.23.38.20

    server socket: Socket[addr=/127.0.0.1,port=51405,localport=1234]
client socket meta: 51405
    payload: 2020.02.12.23.38.21

    server socket: Socket[addr=/127.0.0.1,port=51406,localport=1234]
client socket meta: 51406
    payload: 2020.02.12.23.38.22

Happens Only ONCE Per Client Connection: Socket[addr=/127.0.0.1,port=51408,localport=1234]
```

```
run (server)

Happens Only ONCE Per Client Connection: Socket[addr=/127.0.0.1,port=51407,localport=1234]
requests.size(): 3

Happens Only ONCE Per Client Connection: Socket[addr=/127.0.0.1,port=51409,localport=1234]
requests.size(): 4

Happens Only ONCE Per Client Connection: Socket[addr=/127.0.0.1,port=51410,localport=1234]
requests.size(): 5

Happens Only ONCE Per Client Connection: Socket[addr=/127.0.0.1,port=51411,localport=1234]
requests.size(): 6
^C
~/stage/chat
```

```
client one

~/stage/chat ask

Client reads a response:
ClientProxy::
Reports Server echos to Client client 0
which originates at port 51405:
2020.02.12.23.38.17

Client reads a response:
ClientProxy::
Reports Server echos to Client client 1
which originates at port 51406:
2020.02.12.23.38.18

Client reads a response:
ClientProxy::
Reports Server echos to Client client 0
which originates at port 51405:
2020.02.12.23.38.19

Client reads a response:
ClientProxy::
Reports Server echos to Client client 1
which originates at port 51406:
2020.02.12.23.38.20

Client reads a response:
ClientProxy::
Reports Server echos to Client client 0
which originates at port 51405:
2020.02.12.23.38.21

Client reads a response:
ClientProxy::
Reports Server echos to Client client 1
```

```
client one
2020.02.12.23.38.18
Client reads a response:
ClientProxy::
Reports Server echos to Client client 0
which originates at port 51405:
2020.02.12.23.38.19

Client reads a response:
ClientProxy::
Reports Server echos to Client client 1
which originates at port 51406:
2020.02.12.23.38.20

Client reads a response:
ClientProxy::
Reports Server echos to Client client 0
which originates at port 51405:
2020.02.12.23.38.21

Client reads a response:
ClientProxy::
Reports Server echos to Client client 1
which originates at port 51406:
2020.02.12.23.38.22

Client Shutdown Hook via Lambda expression starts

Client Shutdown Hook Nested Thread starts

Client Shutdown Hook Nested Thread finishes

Client Shutdown Hook via Lambda expression finishes
/Users/greg/stage/chat
~/stage/chat |
```