Documentation

The Java™ Tutorials

Trail: Custom Networking
Lesson: All About Sockets

Writing the Server Side of a Socket

This section shows you how to write a server and the client that goes with it. The server in the client/server pair serves up Knock Knock jokes. Knock Knock jokes are favored by children and are usually vehicles for bad puns. They go like this:

Server: "Knock knock!"
Client: "Who's there?"
Server: "Dexter."
Client: "Dexter who?"
Server: "Dexter halls with boughs of holly."
Client: "Groan."

The example consists of two independently running Java programs: the client program and the server program. The client program is implemented by a single class, `KnockKnockClient`, and is very similar to the `EchoClient` example from the previous section. The server program is implemented by two classes: `KnockKnockServer` and `KnockKnockProtocol`. `KnockKnockServer`, which is similar to `EchoServer`, contains the `main` method for the server program and performs the work of listening to the port, establishing connections, and reading from and writing to the socket. The class `KnockKnockProtocol` serves up the jokes. It keeps track of the current joke, the current state (sent knock knock, sent clue, and so on), and returns the various text pieces of the joke depending on the current state. This object implements the protocol—the language that the client and server have agreed to use to communicate.

The following section looks in detail at each class in both the client and the server and then shows you how to run them.

The Knock Knock Server

This section walks through the code that implements the Knock Knock server program, `KnockKnockServer`.

The server program begins by creating a new `ServerSocket` object to listen on a specific port (see the statement in bold in the following code segment). When running this server, choose a port that is not already dedicated to some other service. For example, this command starts the server program `KnockKnockServer` so that it listens on port 4444:

```
java KnockKnockServer 4444
```

The server program creates the `ServerSocket` object in a `try`-with-resources statement:

```
int portNumber = Integer.parseInt(args[0]);

try (
    ServerSocket serverSocket = new ServerSocket(portNumber);
    Socket clientSocket = serverSocket.accept();
    PrintWriter out =
        new PrintWriter(clientSocket.getOutputStream(), true);
    BufferedReader in = new BufferedReader(
        new InputStreamReader(clientSocket.getInputStream()));
) {
```

`ServerSocket` is a `java.net` class that provides a system-independent implementation of the server side of a client/server socket connection. The constructor for `ServerSocket` throws an exception if it can't listen on the specified port (for example, the port is already being used). In this case, the `KnockKnockServer` has no choice but to exit.

If the server successfully binds to its port, then the `ServerSocket` object is successfully created and the server continues to the next step—accepting a connection from a client (the next statement in the `try`-with-resources statement):

```
clientSocket = serverSocket.accept();
```

The accept method waits until a client starts up and requests a connection on the host and port of this server. (Let's assume that you ran the server program KnockKnockServer on the computer named knockknockserver.example.com.) In this example, the server is running on the port number specified by the first command-line argument. When a connection is requested and successfully established, the accept method returns a new Socket object which is bound to the same local port and has its remote address and remote port set to that of the client. The server can communicate with the client over this new Socket and continue to listen for client connection requests on the original ServerSocket This particular version of the program doesn't listen for more client connection requests. However, a modified version of the program is provided in Supporting Multiple Clients.

After the server successfully establishes a connection with a client, it communicates with the client using this code:

```
    try (
        // ...
        PrintWriter out =
            new PrintWriter(clientSocket.getOutputStream(), true);
        BufferedReader in = new BufferedReader(
            new InputStreamReader(clientSocket.getInputStream()));
    ) {
        String inputLine, outputLine;

        // Initiate conversation with client
        KnockKnockProtocol kkp = new KnockKnockProtocol();
        outputLine = kkp.processInput(null);
        out.println(outputLine);

        while ((inputLine = in.readLine()) != null) {
            outputLine = kkp.processInput(inputLine);
            out.println(outputLine);
            if (outputLine.equals("Bye."))
                break;
        }
```

This code does the following:

1. Gets the socket's input and output stream and opens readers and writers on them.
2. Initiates communication with the client by writing to the socket (shown in bold).
3. Communicates with the client by reading from and writing to the socket (the while loop).

Step 1 is already familiar. Step 2 is shown in bold and is worth a few comments. The bold statements in the code segment above initiate the conversation with the client. The code creates a KnockKnockProtocol object—the object that keeps track of the current joke, the current state within the joke, and so on.

After the KnockKnockProtocol is created, the code calls KnockKnockProtocol's processInput method to get the first message that the server sends to the client. For this example, the first thing that the server says is "Knock! Knock!" Next, the server writes the information to the PrintWriter connected to the client socket, thereby sending the message to the client.

Step 3 is encoded in the while loop. As long as the client and server still have something to say to each other, the server reads from and writes to the socket, sending messages back and forth between the client and the server.

The server initiated the conversation with a "Knock! Knock!" so afterwards the server must wait for the client to say "Who's there?" As a result, the while loop iterates on a read from the input stream. The readLine method waits until the client responds by writing something to its output stream (the server's input stream). When the client responds, the server passes the client's response to the KnockKnockProtocol object and asks the KnockKnockProtocol object for a suitable reply. The server immediately sends the reply to the client via the output stream connected to the socket, using a call to println. If the server's response generated from the KnockKnockServer object is "Bye." this indicates that the client doesn't want any more jokes and the loop quits.

The Java runtime automatically closes the input and output streams, the client socket, and the server socket because they have been created in the try-with-resources statement.

The Knock Knock Protocol

The KnockKnockProtocol class implements the protocol that the client and server use to communicate. This class keeps track of where the client and the server are in their conversation and serves up the server's response to the client's statements. The KnockKnockProtocol object contains the text of all the jokes and makes sure that the client gives the proper response to the server's statements. It wouldn't do to have the client say "Dexter who?" when the server says "Knock! Knock!"

All client/server pairs must have some protocol by which they speak to each other; otherwise, the data that passes back and forth would be meaningless. The protocol that your own clients and servers use depends entirely on the communication required by them to accomplish the task.

The Knock Knock Client

The `KnockKnockClient` class implements the client program that speaks to the `KnockKnockServer`. `KnockKnockClient` is based on the `EchoClient` program in the previous section, Reading from and Writing to a Socket and should be somewhat familiar to you. But we'll go over the program anyway and look at what's happening in the client in the context of what's going on in the server.

When you start the client program, the server should already be running and listening to the port, waiting for a client to request a connection. So, the first thing the client program does is to open a socket that is connected to the server running on the specified host name and port:

```
String hostName = args[0];
int portNumber = Integer.parseInt(args[1]);

try (
    Socket kkSocket = new Socket(hostName, portNumber);
    PrintWriter out = new PrintWriter(kkSocket.getOutputStream(), true);
    BufferedReader in = new BufferedReader(
        new InputStreamReader(kkSocket.getInputStream()));
)
```

When creating its socket, the `KnockKnockClient` example uses the host name of the first command-line argument, the name of the computer on your network that is running the server program `KnockKnockServer`.

The `KnockKnockClient` example uses the second command-line argument as the port number when creating its socket. This is a *remote port number*—the number of a port on the server computer—and is the port to which `KnockKnockServer` is listening. For example, the following command runs the `KnockKnockClient` example with `knockknockserver.example.com` as the name of the computer that is running the server program `KnockKnockServer` and 4444 as the remote port number:

```
java KnockKnockClient knockknockserver.example.com 4444
```

The client's socket is bound to any available *local port*—a port on the client computer. Remember that the server gets a new socket as well. If you run the `KnockKnockClient` example with the command-line arguments in the previous example, then this socket is bound to local port number 4444 on the computer from which you ran the `KnockKnockClient` example. The server's socket and the client's socket are connected.

Next comes the `while` loop that implements the communication between the client and the server. The server speaks first, so the client must listen first. The client does this by reading from the input stream attached to the socket. If the server does speak, it says "Bye." and the client exits the loop. Otherwise, the client displays the text to the standard output and then reads the response from the user, who types into the standard input. After the user types a carriage return, the client sends the text to the server through the output stream attached to the socket.

```
while ((fromServer = in.readLine()) != null) {
    System.out.println("Server: " + fromServer);
    if (fromServer.equals("Bye."))
        break;

    fromUser = stdIn.readLine();
    if (fromUser != null) {
        System.out.println("Client: " + fromUser);
        out.println(fromUser);
    }
}
```

The communication ends when the server asks if the client wishes to hear another joke, the client says no, and the server says "Bye."

The client automatically closes its input and output streams and the socket because they were created in the `try`-with-resources statement.

Running the Programs

You must start the server program first. To do this, run the server program using the Java interpreter, just as you would any other Java application. Specify as a command-line argument the port number on which the server program listens:

```
java KnockKnockServer 4444
```

Next, run the client program. Note that you can run the client on any computer on your network; it does not have to run on the same computer as the server. Specify as command-line arguments the host name and the port number of the computer running the `KnockKnockServer` server program:

```
java KnockKnockClient knockknockserver.example.com 4444
```

If you are too quick, you might start the client before the server has a chance to initialize itself and begin listening on the port. If this happens, you will see a stack trace from the client. If this happens, just restart the client.

If you try to start a second client while the first client is connected to the server, the second client just hangs. The next section, Supporting Multiple Clients, talks about supporting multiple clients.

When you successfully get a connection between the client and server, you will see the following text displayed on your screen:

```
Server: Knock! Knock!
```

Now, you must respond with:

```
Who's there?
```

The client echoes what you type and sends the text to the server. The server responds with the first line of one of the many Knock Knock jokes in its repertoire. Now your screen should contain this (the text you typed is in bold):

```
Server: Knock! Knock!
Who's there?
Client: Who's there?
Server: Turnip
```

Now, you respond with:

```
Turnip who?
```

Again, the client echoes what you type and sends the text to the server. The server responds with the punch line. Now your screen should contain this:

```
Server: Knock! Knock!
Who's there?
Client: Who's there?
Server: Turnip
Turnip who?
Client: Turnip who?
Server: Turnip the heat, it's cold in here! Want another? (y/n)
```

If you want to hear another joke, type **y**; if not, type **n**. If you type **y**, the server begins again with "Knock! Knock!" If you type **n**, the server says "Bye." thus causing both the client and the server to exit.

If at any point you make a typing mistake, the `KnockKnockServer` object catches it and the server responds with a message similar to this:

```
Server: You're supposed to say "Who's there?"!
```

The server then starts the joke over again:

```
Server: Try again. Knock! Knock!
```

Note that the `KnockKnockProtocol` object is particular about spelling and punctuation but not about capitalization.

Supporting Multiple Clients

To keep the `KnockKnockServer` example simple, we designed it to listen for and handle a single connection request. However, multiple client requests can come into the same port and, consequently, into the same `ServerSocket`. Client connection requests are queued at the port, so the server must accept the connections sequentially. However, the server can service them simultaneously through the use of threads—one thread per each client connection.

The basic flow of logic in such a server is this:

```
while (true) {
    accept a connection;
    create a thread to deal with the client;
}
```

The thread reads from and writes to the client connection as necessary.

Try This:

Modify the `KnockKnockServer` so that it can service multiple clients at the same time. Two classes compose our solution: `KKMultiServer` and `KKMultiServerThread`. `KKMultiServer` loops forever, listening for client connection requests on a

`ServerSocket`. When a request comes in, `KKMultiServer` accepts the connection, creates a new `KKMultiServerThread` object to process it, hands it the socket returned from accept, and starts the thread. Then the server goes back to listening for connection requests. The `KKMultiServerThread` object communicates to the client by reading from and writing to the socket. Run the new Knock Knock server `KKMultiServer` and then run several clients in succession.

Previous page: Reading from and Writing to a Socket
Next page: All About Datagrams