Documentation

The Java™ Tutorials

Trail: Custom Networking
Lesson: All About Sockets

Reading from and Writing to a Socket

Let's look at a simple example that illustrates how a program can establish a connection to a server program using the `Socket` class and then, how the client can send data to and receive data from the server through the socket.

The example program implements a client, `EchoClient`, that connects to an echo server. The echo server receives data from its client and echoes it back. The example `EchoServer` implements an echo server. (Alternatively, the client can connect to any host that supports the Echo Protocol.)

The `EchoClient` example creates a socket, thereby getting a connection to the echo server. It reads input from the user on the standard input stream, and then forwards that text to the echo server by writing the text to the socket. The server echoes the input back through the socket to the client. The client program reads and displays the data passed back to it from the server.

Note that the `EchoClient` example both writes to and reads from its socket, thereby sending data to and receiving data from the echo server.

Let's walk through the program and investigate the interesting parts. The following statements in the `try`-with-resources statement in the `EchoClient` example are critical. These lines establish the socket connection between the client and the server and open a `PrintWriter` and a `BufferedReader` on the socket:

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

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

The first statement in the `try`-with resources statement creates a new `Socket` object and names it `echoSocket`. The `Socket` constructor used here requires the name of the computer and the port number to which you want to connect. The example program uses the first command-line argument as the name of the computer (the host name) and the second command line argument as the port number. When you run this program on your computer, make sure that the host name you use is the fully qualified IP name of the computer to which you want to connect. For example, if your echo server is running on the computer `echoserver.example.com` and it is listening on port number 7, first run the following command from the computer `echoserver.example.com` if you want to use the `EchoServer` example as your echo server:

```
java EchoServer 7
```

Afterward, run the `EchoClient` example with the following command:

```
java EchoClient echoserver.example.com 7
```

The second statement in the `try`-with resources statement gets the socket's output stream and opens a `PrintWriter` on it. Similarly, the third statement gets the socket's input stream and opens a `BufferedReader` on it. The example uses readers and writers so that it can write Unicode characters over the socket.

To send data through the socket to the server, the `EchoClient` example needs to write to the `PrintWriter`. To get the server's response, `EchoClient` reads from the `BufferedReader` object `stdIn`, which is created in the fourth statement in the `try`-with resources statement. If you are not yet familiar with the Java platform's I/O classes, you may wish to read Basic I/O.

The next interesting part of the program is the `while` loop. The loop reads a line at a time from the standard input stream and immediately sends it to the server by writing it to the `PrintWriter` connected to the socket:

```
String userInput;
while ((userInput = stdIn.readLine()) != null) {
    out.println(userInput);
    System.out.println("echo: " + in.readLine());
}
```

The last statement in the `while` loop reads a line of information from the `BufferedReader` connected to the socket. The `readLine` method waits until the server echoes the information back to `EchoClient`. When `readline` returns, `EchoClient` prints the information to the standard output.

The `while` loop continues until the user types an end-of-input character. That is, the `EchoClient` example reads input from the user, sends it to the Echo server, gets a response from the server, and displays it, until it reaches the end-of-input. (You can type an end-of-input character by pressing **Ctrl-C**.) The `while` loop then terminates, and the Java runtime automatically closes the readers and writers connected to the socket and to the standard input stream, and it closes the socket connection to the server. The Java runtime closes these resources automatically because they were created in the `try`-with-resources statement. The Java runtime closes these resources in reverse order that they were created. (This is good because streams connected to a socket should be closed before the socket itself is closed.)

This client program is straightforward and simple because the echo server implements a simple protocol. The client sends text to the server, and the server echoes it back. When your client programs are talking to a more complicated server such as an HTTP server, your client program will also be more complicated. However, the basics are much the same as they are in this program:

1. Open a socket.
2. Open an input stream and output stream to the socket.
3. Read from and write to the stream according to the server's protocol.
4. Close the streams.
5. Close the socket.

Only step 3 differs from client to client, depending on the server. The other steps remain largely the same.

Previous page: What Is a Socket?
Next page: Writing the Server Side of a Socket