**Lesson: All About Sockets**

URLs and URLConnections provide a relatively high-level mechanism for accessing resources on the Internet. Sometimes your programs require lower-level network communication, for example, when you want to write a client-server application.

In client-server applications, the server provides some service, such as processing database queries or sending out current stock prices. The client uses the service provided by the server, either displaying database query results to the user or making stock purchase recommendations to an investor. The communication that occurs between the client and the server must be reliable. That is, no data can be dropped and it must arrive on the client side in the same order in which the server sent it.

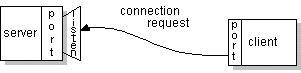
TCP provides a reliable, point-to-point communication channel that client-server applications on the Internet use to communicate with each other. To communicate over TCP, a client program and a server program establish a connection to one another. Each program binds a socket to its end of the connection. To communicate, the client and the server each reads from and writes to the socket bound to the connection.

[**What Is a Socket?**](http://docs.oracle.com/javase/tutorial/networking/sockets/definition.html)

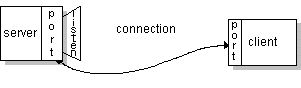
A socket is one end-point of a two-way communication link between two programs running on the network. Socket classes are used to represent the connection between a client program and a server program. The java.net package provides two classes--Socket and ServerSocket--that implement the client side of the connection and the server side of the connection, respectively.

Normally, a server runs on a specific computer and has a socket that is bound to a specific port number. The server just waits, listening to the socket for a client to make a connection request.

On the client-side: The client knows the hostname of the machine on which the server is running and the port number on which the server is listening. To make a connection request, the client tries to rendezvous with the server on the server's machine and port. The client also needs to identify itself to the server so it binds to a local port number that it will use during this connection. This is usually assigned by the system.



If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket bound to the same local port and also has its remote endpoint set to the address and port of the client. It needs a new socket so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client.



On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server.

The client and server can now communicate by writing to or reading from their sockets.

**Definition:**

A *socket* is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent to.

An endpoint is a combination of an IP address and a port number. Every TCP connection can be uniquely identified by its two endpoints. That way you can have multiple connections between your host and the server.

The java.net package in the Java platform provides a class, Socket, that implements one side of a two-way connection between your Java program and another program on the network. The Socket class sits on top of a platform-dependent implementation, hiding the details of any particular system from your Java program. By using the java.net.Socket class instead of relying on native code, your Java programs can communicate over the network in a platform-independent fashion.

Additionally, java.net includes the ServerSocket class, which implements a socket that servers can use to listen for and accept connections to clients. This lesson shows you how to use the Socket and ServerSocket classes.

If you are trying to connect to the Web, the URL class and related classes (URLConnection, URLEncoder) are probably more appropriate than the socket classes. In fact, URLs are a relatively high-level connection to the Web and use sockets as part of the underlying implementation. See [Working with URLs](http://docs.oracle.com/javase/tutorial/networking/urls/index.html) for information about connecting to the Web via URLs.

# 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](http://docs.oracle.com/javase/tutorial/networking/sockets/examples/EchoClient.java" \t "_blank), that connects to an echo server. The echo server receives data from its client and echoes it back. The example [EchoServer](http://docs.oracle.com/javase/tutorial/networking/sockets/examples/EchoServer.java" \t "_blank) implements an echo server. (Alternatively, the client can connect to any host that supports the [Echo Protocol](http://tools.ietf.org/html/rfc862).)

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](http://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html) statement in the EchoClient example are critical. These lines establish the socket connection between the client and the server and open a [PrintWriter](https://docs.oracle.com/javase/8/docs/api/java/io/PrintWriter.html" \t "_blank) and a[BufferedReader](https://docs.oracle.com/javase/8/docs/api/java/io/BufferedReader.html) 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](https://docs.oracle.com/javase/8/docs/api/java/net/Socket.html) 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](http://docs.oracle.com/javase/tutorial/essential/environment/cmdLineArgs.html) 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](http://docs.oracle.com/javase/tutorial/essential/io/index.html).

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.

**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](http://docs.oracle.com/javase/tutorial/networking/sockets/examples/KnockKnockClient.java" \t "_blank), and is very similar to the [EchoClient](http://docs.oracle.com/javase/tutorial/networking/sockets/examples/EchoClient.java" \t "_blank) example from the previous section. The server program is implemented by two classes: [KnockKnockServer](http://docs.oracle.com/javase/tutorial/networking/sockets/examples/KnockKnockServer.java) and [KnockKnockProtocol](http://docs.oracle.com/javase/tutorial/networking/sockets/examples/KnockKnockProtocol.java). KnockKnockServer, which is similar to [EchoServer](http://docs.oracle.com/javase/tutorial/networking/sockets/examples/EchoServer.java" \t "_blank), 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](http://docs.oracle.com/javase/tutorial/networking/sockets/examples/KnockKnockProtocol.java" \t "_blank) 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](http://docs.oracle.com/javase/tutorial/networking/sockets/examples/KnockKnockServer.java" \t "_blank).

The server program begins by creating a new [ServerSocket](https://docs.oracle.com/javase/8/docs/api/java/net/ServerSocket.html" \t "_blank) 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](https://docs.oracle.com/javase/8/docs/api/java/net/package-frame.html) 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](https://docs.oracle.com/javase/8/docs/api/java/net/ServerSocket.html#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](https://docs.oracle.com/javase/8/docs/api/java/net/Socket.html) 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](http://docs.oracle.com/javase/tutorial/networking/sockets/clientServer.html#later).

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](https://docs.oracle.com/javase/8/docs/api/java/io/PrintWriter.html) 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](http://docs.oracle.com/javase/tutorial/networking/sockets/examples/KnockKnockProtocol.java" \t "_blank) 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](http://docs.oracle.com/javase/tutorial/networking/sockets/examples/KnockKnockClient.java" \t "_blank) 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](http://docs.oracle.com/javase/tutorial/networking/sockets/readingWriting.html) 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](http://docs.oracle.com/javase/tutorial/networking/sockets/clientServer.html#later), 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](http://docs.oracle.com/javase/tutorial/networking/sockets/examples/KKMultiServer.java) and [KKMultiServerThread](http://docs.oracle.com/javase/tutorial/networking/sockets/examples/KKMultiServerThread.java). 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. TheKKMultiServerThread 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.

### Multithreaded Server Example

The example in its current state works between the server program and one client program only. To allow multiple client connections, the server program has to be converted to a [multithreaded server](http://www.oracle.com/technetwork/java/socket-140484.html)program.

|  |  |
| --- | --- |
| http://www.oracle.com/ocom/groups/public/@otn/documents/digitalasset/145391.gif  First Client  http://www.oracle.com/ocom/groups/public/@otn/documents/digitalasset/148446.gif  Second Client  http://www.oracle.com/ocom/groups/public/@otn/documents/digitalasset/145418.gif  Third Client | http://www.oracle.com/ocom/groups/public/@otn/documents/digitalasset/146376.gif |

In this example the listenSocket method loops on the server.accept call waiting for client connections and creates an instance of the ClientWorker class for each client connection it accepts. The textArea component that displays the text received from the client connection is passed to the ClientWorker instance with the accepted client connection.

|  |
| --- |
| public void listenSocket(){  try{  server = new ServerSocket(4444);  } catch (IOException e) {  System.out.println("Could not listen on port 4444");  System.exit(-1);  }  while(true){  ClientWorker w;  try{  //server.accept returns a client connection  w = new ClientWorker(server.accept(), textArea);  Thread t = new Thread(w);  t.start();  } catch (IOException e) {  System.out.println("Accept failed: 4444");  System.exit(-1);  }  }  } |

The important changes in this version of the server program over the non-threaded server program are the line and client variables are no longer instance variables of the server class, but are handled inside the ClientWorker class.

The ClientWorker class implements the Runnable interface, which has one method, run. Therun method executes independently in each thread. If three clients request connections, threeClientWorker instances are created, a thread is started for each ClientWorker instance, and therun method executes for each thread.

In this example, the run method creates the input buffer and output writer, loops on the input stream waiting for input from the client, sends the data it receives back to the client, and sets the text in the text area.

|  |
| --- |
| class ClientWorker implements Runnable {  private Socket client;  private JTextArea textArea;  //Constructor  ClientWorker(Socket client, JTextArea textArea) {  this.client = client;  this.textArea = textArea;  }  public void run(){  String line;  BufferedReader in = null;  PrintWriter out = null;  try{  in = new BufferedReader(new  InputStreamReader(client.getInputStream()));  out = new  PrintWriter(client.getOutputStream(), true);  } catch (IOException e) {  System.out.println("in or out failed");  System.exit(-1);  }  while(true){  try{  line = in.readLine();  //Send data back to client  out.println(line);  //Append data to text area  textArea.append(line);  }catch (IOException e) {  System.out.println("Read failed");  System.exit(-1);  }  }  }  } |

JTextArea.appendJTextArea.appendtextArea.append(line)synchronizedruntextArea.append(line)appendText(line)

public synchronized void appendText(line){

textArea.append(line);

}

synchronizedtextAreatextArea

The finalize() method is called by the Java virtual machine (JVM)\* before the program exits to give the program a chance to clean up and release resources. Multi-threaded programs should close all Files and Sockets they use before exiting so they do not face resource starvation. The call to server.close() in the finalize() method closes the Socket connection used by each thread in this program.

|  |
| --- |
| protected void finalize(){  //Objects created in run method are finalized when  //program terminates and thread exits  try{  server.close();  } catch (IOException e) {  System.out.println("Could not close socket");  System.exit(-1);  }  } |