Working with Sockets

23

Outcomes:

By the end of this chapter you should be able to:

- *explain the terms* ***client***, ***server***, ***host*** *and* ***port***;
- *describe the* ***client–server*** *model*;
- *explain the function of a* ***socket***;
- *distinguish between the Java* `Socket` *and* `ServerSocket` *classes and explain their function*;
- *write a simple client–server application using sockets*;
- *write a server application that supports multiple clients*;
- *write multi-threaded client–server applications*;
- *create client and server applications that utilise a JavaFX interface.*

23.1 Introduction

In this chapter we are going to explore the way in which Java can be used to write programs that communicate over a network. We should say from the outset that here we are dealing only with communication over a local area network (LAN). Communication over wide networks, and in particular the Internet, involve issues that are beyond the scope of this text. In particular, communication over the Internet is now fraught with security issues, and we are not able to address such concerns here. In previous editions of this book, we covered applets, which are java programs that run in a browser. We are no longer going to deal with applets as browsers have by and large stopped supporting them because of the enormous security questions that they can pose.

Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_23

Network programs rely very much on the concept of a **client** and a **server**. A server program provides some sort of service for other programs—clients—normally located on a different machine. The service it provides could be one of many things—it could send some files to the client; it could send web pages to the client; it could read some data from the local machine and send that across, maybe having done some processing first; it could perform a complex calculation; it could print some material on a local printer. The possibilities are endless.

It should be noted that the distinction between a client and server can become blurred: a program acting as a client in one situation could also act as a server in another, and vice versa. It is also important to note that it is often the case that a machine, rather than a program, is referred to as a server. This usually happens when a machine is dedicated to running a particular server program—typically a file server—and does very little else. Strictly speaking we should refer to the machine on which a server runs as the **host**.

In general, communication between a client and a server could be over a local area network, a wide area network, or over the Internet. Server programs that offer a service via the Internet have to obey a particular set of rules or protocols to ensure that the client and server are "speaking the same language". Common examples are File Transfer Protocol (FTP) for servers that send files, and Hypertext Transfer Protocol (HTTP) for services that send web pages to a client.

However, as we have stated above, here we will be dealing here only with local area networks which communicate by implementing **sockets**—special programs that allow data to pass between two applications running on different computers.

23.2 Sockets

In Chap. 18 you were introduced to the idea of a *stream*—a channel of communication between the computer's main memory and some external device such as a disk (not to be confused with streams for processing collections, that we covered in Chap. 22). In that chapter you were shown how Java provides high-level classes that hide the programmer from the low-level details of how data is stored on a disk or other device. Just as the external storage of data is a complicated business, so too is the transmission of data across a network.

A **socket** is a software mechanism that is able to hide the programmer from the detail of how data is actually transmitted, in a not dissimilar way to that in which the high-level file handling classes protect the programmer from the details of external storage. Sockets were originally developed for the Unix™ operating system and they enabled the programmer to treat a network connection as just another stream to which data can be written, and from which it can be read. Sockets have since been developed for other operating systems such as Windows™, and fortunately for Java.

In order to understand sockets it is also necessary to understand the concept of a **port**. A machine on a network is referred to by its IP (Internet Protocol) address. However, any particular host can perform a number of different functions, and therefore needs to be able to distinguish between different types of request, such as email requests, file transfer requests, requests for web pages and so on. This is accomplished by assigning each type of request a special number known as a port. Many port numbers are now internationally recognized, and so all computers will agree on their meaning. For example, a request on port 80 will always be expected to be an HTTP request; port 21 is for FTP (File Transfer Protocol) requests. A client program can therefore assume that server programs will be using these ports for those particular services.

All sockets must be capable of doing the following:

- connect to a remote machine;
- send data;
- receive data;
- close a connection.

A socket which is to be used for a server must additionally be able to:

- bind to a port (that is to associate the server with a port number);
- listen for incoming data;
- accept connections from a remote server on the bound port.

The Java `Socket` class has methods that correspond to the first four of the above; the `ServerSocket` class provides methods for the last three.

23.3 A Simple Server Application

The server we are going to build is going to offer a very simple service to a client; it will wait to receive two integers, and then it will send back the sum of those two integers. Clearly this would not in reality be a very useful server—a real-world server would be offering a far more complex service—perhaps performing some very complicated processing, or retrieving data from a database running on the same machine, or maybe printing on a printer local to the server. However, our simple addition server demonstrates the principles of a client–server protocol very nicely.

A program such as this would typically be launched from a command line, or perhaps launched as a service on startup—many services run in the background, and it is often the case that the user has little awareness of their existence. In order to monitor the behaviour of our server we have organised it so that it reports its behaviour to a console.

The `AdditionServer` class is presented below—have a look at it and then we'll take you through it.

AdditionServer

```
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.ServerSocket;
import java.net.Socket;

public class AdditionServer
{
    public static void main(String[] args)
    {
        final int port = 8901;

        // declare a "general" socket and a server socket
        Socket connection;
        ServerSocket listenSocket;

        // declare low level and high level objects for input
        InputStream inStream;
        DataInputStream inDataStream;

        // declare low level and high level objects for output
        OutputStream outStream;
        DataOutputStream outDataStream;

        // declare other variables
        String client;
        int first, second, sum;
        boolean connected;
        while(true)
        {
            try
            {
                // create a server socket
                listenSocket = new ServerSocket(port);
                System.out.println("Listening on port " + port);

                // listen for a connection from the client
                connection = listenSocket.accept();
                connected = true;
                System.out.println("Connection established");

                // create an input stream from the client
                inStream = connection.getInputStream();
                inDataStream = new DataInputStream(inStream);

                // create an output stream to the client
                outStream = connection.getOutputStream ();
                outDataStream = new DataOutputStream(outStream);

                // wait for a string from the client
                client = inDataStream.readUTF();
                System.out.println("Address of client: " + client);
                while(connected)
                {
                    // read an integer from the client
                    first = inDataStream.readInt();
                    System.out.println("First number received: " + first);

                    // read an integer from the client
                    second = inDataStream.readInt();
                    System.out.println("Second number received: " + second);

                    sum = first + second;
                    System.out.println("Sum returned: " + sum);

                    // send the sum to the client
                    outDataStream.writeInt(sum);
                }
            }

            catch (IOException e)
            {
                connected = false;
            }
        }
    }
}
```

For convenience we have hard-coded the port number (8901) into the program and have declared a constant for this purpose; the client will need to be made aware of this port number.

Having declared this constant, we have gone on to declare a number of variables:

```
Socket connection;
ServerSocket listenSocket;

InputStream inStream;
DataInputStream inDataStream;

OutputStream outStream;
DataOutputStream outDataStream;

String client;
int first, second, sum;
boolean connected;
```

The first two variables are, respectively, a `Socket` and a `ServerSocket`. As this is a server application it requires both the general functionality of the `Socket` class and the specialist functionality of the `ServerSocket` class.

Next we declare the objects that we will need to establish an input stream with the client. We have come across the classes `InputStream` and `DataInputStream` before, in Chap. 18. The former allows communication at a low level in the form of bytes; the latter allows the high-level communication in the form of strings, integers, characters and so on with which we are familiar.

After this we declare objects of `OutputStream` and `DataOutputStream` that we will need to establish the output stream. Finally we make some other declarations that we will need later on.

Now we start an infinite loop. The idea is that the server will accept a connection request from a client, and when that client is finished making requests it will be ready to receive connections from other clients; this will continue until the server is terminated. If the server was called from the command line, then closing the console window will terminate it—if you are working in an IDE, you will have to terminate the process via the IDE interface when you no longer require it.

From now everything is placed in a **`try`** block because the constructor of the `ServerSocket` class, and its `accept` method both throw `IOExceptions`.

The first instruction in the **`try`** block looks like this:

```
listenSocket = new ServerSocket(port);
```

We are creating a new `ServerSocket` object and binding it to a particular port.

In order to get the server to listen for a client requesting a connection on that port, we call the `accept` method of the `ServerSocket` class; we also place a message in the console to tell us that the server is listening for a request:

```
System.out.println("Listening on port " + port);
connection = listenSocket.accept();
```

The `accept` method returns an object of the `Socket` class, which we assign to the `connection` variable that we declared earlier.

Once the connection is established we set the **`boolean`** variable, `connected`, to **`true`** and display a message:

```
connected = true;
System.out.println("Connection established");
```

The next thing we do is call the `getInputStream` method of the `Socket` object, `connection`. This returns an object of the `InputStream` class, thus providing a stream from client to server. We then wrap this low-level `InputStream` object with a high-level `DataInputStream` `object`, in the same way as we did when handling files in Chap. 18:

```
inStream = connection.getInputStream();
inDataStream = new DataInputStream(inStream);
```

We then create an output stream in the same way:

```
OutputStream outStream;
DataOutputStream outDataStream;
```

As you will see shortly, we have designed our client to send its IP address to the server once it is connected. So our next instructions to the server are to wait to receive a string on the input stream, and then to display a message on the console.

```
client = inDataStream.readUTF();
System.out.println("Address of client: " + client);
```

Once a connection has been established we want the server to perform the addition calculation for the client as many times as the client requires. Thus we provide a **`while`** loop that continues until the connection is lost:

```
while(connected)
{
        // read an integer from the client
        first = inDataStream.readInt();
        System.out.println("First number received: " + first);

        // read an integer from the client
        second = inDataStream.readInt();
        System.out.println("Second number received: " + second);

        sum = first + second;
        System.out.println("Sum returned: " + sum);

        // send the sum to the client
        outDataStream.writeInt(sum);
}
```

You can see that we read two integers from the input stream, displaying them each time on the console. We then calculate and display the sum, which we send back to the client on the output stream.

The `accept` method of `ServerSocket` throws an `IOException` when the connection is lost. Therefore we have coded the **`catch`** block so that the connected variable that controls the inner **`while`** loop is set to **`false`**, so that when the client closes the connection the server will no longer expect to receive integers, but will return to the top of the outer **`while`** loop, and wait for another connection request:

In a moment we will show you how to create a client program that requests a

```
catch (IOException e)
{
      connected = false;
}
```

service from our server. But before we do that, take a look at Fig. 23.1, which shows the result of a typical session from the point of view of the server, running in

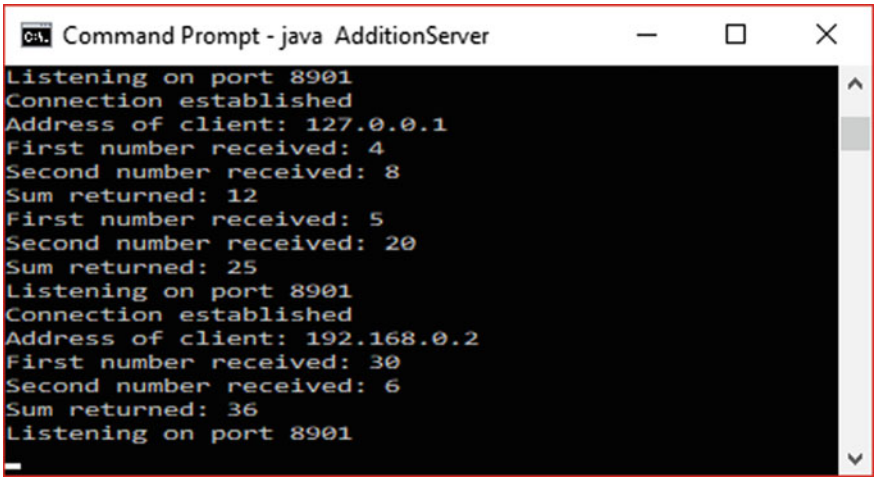

Fig. 23.1 A typical session for the addition server

a console. The server listens on port 8901; a client connects, requests two calculations and ends the session; the server waits for another connection; another client connects, requests one calculation and then ends the session, and the server once again waits for another client to connect.

23.4 A Simple Client Application

The client application will utilize a JavaFX interface as shown in Fig. 23.2.

Figure 23.2 should give you an idea of how the operation of the client will work. Once the address and port number of the remote host are known, the connection is established. Then the user is free to enter numbers and press the button to send these numbers to the server and display the result. The middle text box is used to display messages regarding the connection.

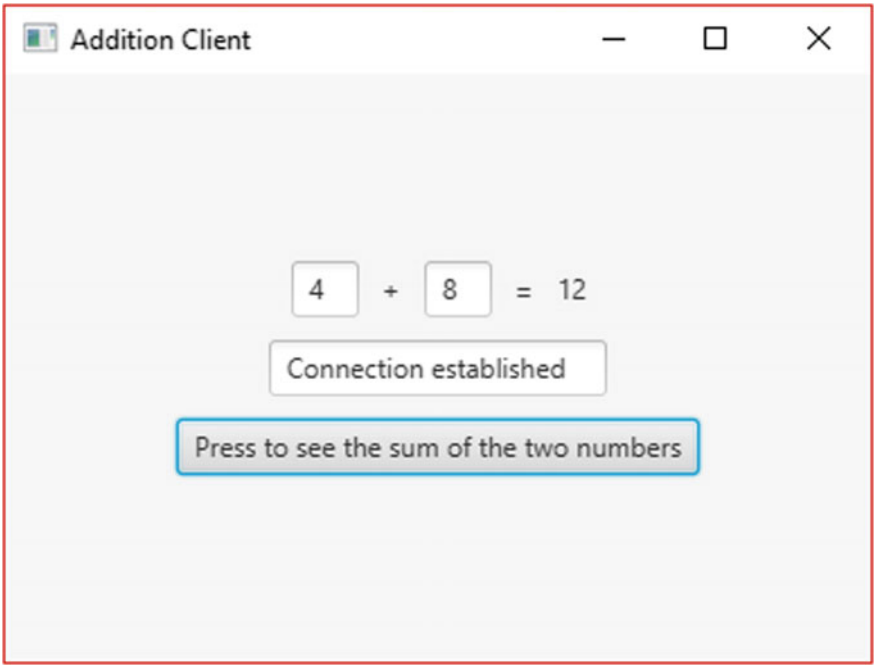

Fig. 23.2 The addition client

Here is the code for the `AdditionClient`:

AdditionClient

```
import java.io.InputStream;
import java.io.DataInputStream;
import java.io.OutputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.Optional;
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.control.TextInputDialog;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class AdditionClient extends Application
{
       private String remoteHost;
       private  int port;

       // declare low level and high level objects for input
       private InputStream inStream;
       private DataInputStream inDataStream;

       // declare low level and high level objects for output
       private  OutputStream outStream ;
       private DataOutputStream outDataStream;

       // declare a socket
       private Socket connection;

       @Override
       public void start(Stage stage)
       {
            getInfo(); // call the method that gets the information about the server

            // declare visual components
            TextField msg = new TextField();

            TextField firstNumber = new TextField();
            Label plus = new Label("+");
            TextField secondNumber = new TextField();

            Label equals = new Label("=");
            Label sum = new Label();

            Button calculateButton = new Button("Press to see the sum of the two numbers");

            // configure the scene
            msg.setMaxWidth(150);
            firstNumber.setMaxWidth(30);
            secondNumber.setMaxWidth(30);

            HBox hBox = new HBox(10);
            hBox.setAlignment(Pos.CENTER);
            hBox.getChildren().addAll(firstNumber, plus, secondNumber, equals, sum);
            VBox root = new VBox(10);

            root.setAlignment(Pos.CENTER);
            root.getChildren().addAll(hBox, msg,  calculateButton);

            Scene scene = new Scene(root, 400, 300);
            stage.setScene(scene);
            stage.setTitle("Addition Client");

            // show the stage
            stage.show();

            try
            {
```

```
                // attempt to create a connection to the server
                connection = new Socket(remoteHost, port);
                msg.setText("Connection established");

                // create an input stream from the server
                inStream = connection.getInputStream();
                inDataStream = new DataInputStream(inStream);

                // create an output stream to the server
                outStream = connection.getOutputStream();
                outDataStream = new DataOutputStream(outStream);

                // send the host IP to the server
                outDataStream.writeUTF(connection.getLocalAddress().getHostAddress());
            }

            catch (UnknownHostException e)
            {
                msg.setText("Unknown host");
            }

            catch (IOException ie)
            {
                msg.setText("Network Exception");
            }

            // specifiy the behaviour of the calculate button
            calculateButton.setOnAction(e ->
                        {
                           try
                           {
                                // send the two integers to the server
                                outDataStream.writeInt(Integer.parseInt(firstNumber.getText()));
                                outDataStream.writeInt(Integer.parseInt(secondNumber.getText()));

                                // read and display the result sent back from the server
                                int result = inDataStream.readInt();
                                sum.setText("" + result);
                            }
                            catch(IOException ie)
                            {
                            }
                        }
                                    );

        }

        private void getInfo()
        {
              Optional<String> response;

              // use the TextInputDialog class to allow the user to enter the host address
              TextInputDialog addressDialog = new TextInputDialog();
              addressDialog.setHeaderText("Enter remote host");
              addressDialog.setTitle("Addition Client");

              response = addressDialog.showAndWait();
              remoteHost = response.get();

              // use the TextInputDialog class to allow the user to enter port number
              TextInputDialog portDialog = new TextInputDialog();
              portDialog.setHeaderText("Enter port number");
              portDialog.setTitle("Addition Client");

              response = portDialog.showAndWait();
              port = Integer.valueOf(response.get());
        }

        public static void main(String[] args)
        {
                 launch(args);
        }
}
```

We have declared a number of attributes, the first of which will hold values for the address of the remote host and the port number, and the remaining attributes are concerned with input and output streams, and the socket:

```
private String remoteHost;
private int port;

private InputStream inStream;
private DataInputStream inDataStream;

private  OutputStream outStream ;
private DataOutputStream outDataStream;

private Socket connection;
```

The first thing that we do inside the `start` method is to call a helper method `getInfo`. This will prompt the user to enter the address of the host machine on which the server is running, and the port number; the user of course must be aware of this information in order for the client to make the connection. So let's begin by briefly looking at the `getInfo` method:

```
private void getInfo()
{
      Optional<String> response;

      // use the TextInputDialog class to allow the user to enter the host address
      TextInputDialog addressDialog = new TextInputDialog();
      addressDialog.setHeaderText("Enter remote host");
      addressDialog.setTitle("Addition Client");

      response = addressDialog.showAndWait();
      remoteHost = response.get();

      // use the TextInputDialog class to allow the user to enter port number
      TextInputDialog portDialog = new TextInputDialog();
      portDialog.setHeaderText("Enter port number");
      portDialog.setTitle("Addition Client");

      response = portDialog.showAndWait();
      port = Integer.valueOf(response.get());
}
```

Here we are using the `TextInput Dialog` class that we introduced in Chap. 17 to get the host address and then to get the port number, as shown in Fig. 23.3.

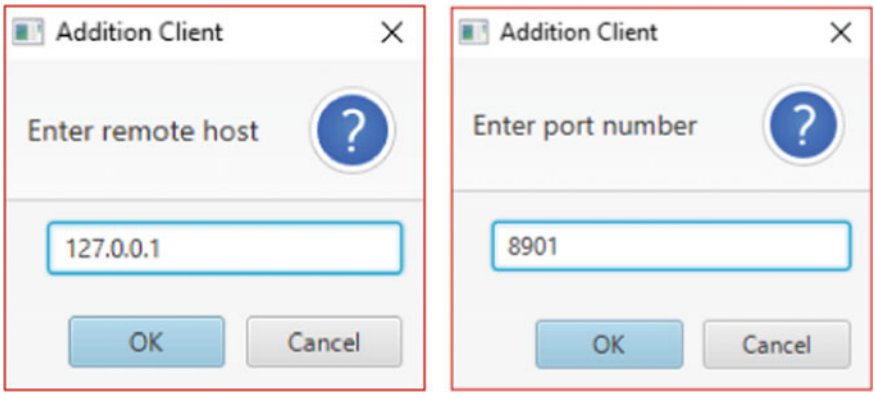

Fig. 23.3 Getting the remote host and port number from the user

Having called `getInfo` we go on to declare and configure the visual elements. Once we have shown the stage, we attempt to make the connection:

```
try
{
        // attempt to create a connection to the server
        connection = new Socket(remoteHost, port);
        msg.setText("Connection established");

        // create an input stream from the server
        inStream = connection.getInputStream();
        inDataStream = new DataInputStream(inStream);

        // create an output stream to the server
        outStream = connection.getOutputStream();
        outDataStream = new DataOutputStream(outStream);

        // send the host IP to the server
        outDataStream.writeUTF(connection.getLocalAddress().getHostAddress());
}

catch(UnknownHostException e)
{
        msg.setText("Unknown host");
}

catch(IOException ie)
{
        msg.setText("Network Exception");
}
```

The code must be placed within a **`try`** block. This is necessary because the constructor of the `Socket` class potentially throws two exceptions. As you can see it is called with two arguments, the name or IP address of the host machine (a `String`) and the port number (an **`int`**).

Creating a new `Socket` in this way transmits a message requesting a response from the remote machine specified, listening on the port in question. If the connection is established, and no exception is therefore thrown, the constructor goes on to display the message "Connection established" in the message area, and then to initialize the input and output streams. It finishes with this instruction:

```
outDataStream.writeUTF(connection.getLocalAddress().getHostAddress());
```

You will recall that we programmed the server so that the first thing it did after the connection was established was to wait for a string from the client. Here you can see how the client sends its address to the server on the output stream. It calls the `getLocalAddress` method of the `Socket` class. This returns an object of the `InetAddress` class. The `InetAddress` class holds a representation of an IP address and enables us to obtain the host name, or the IP address (as a `String`), with the methods `getHostName` and `getHostAddress` respectively.

Now we have to catch the exceptions that can be thrown by the constructor. As you can see there are two **`catch`** blocks. The first handles an `UnknownHostException` which will be thrown if the host we are trying to connect to is unknown. As you can see from the code, an appropriate message is placed in the message area.

If there is another network error (perhaps no server is running on the specified host), then an `IOException` is thrown and the message "Network Exception" is displayed.

Finally we need to provide the code that determines what happens when we press the button that gets the server to perform the addition for us:

```
calculateButton.setOnAction(e ->
                  {
                           try
                           {
                                // send the two integers to the server
                                outDataStream.writeInt(Integer.parseInt(firstNumber.getText()));
                                outDataStream.writeInt(Integer.parseInt(secondNumber.getText()));

                                // read and display the result sent back from the server
                                int result = inDataStream.readInt();
                                sum.setText("" + result);
                           }
                           catch(IOException ie)
                           {
                           }
                  }
                           );
```

This is pretty straightforward: we send the two numbers to the server and read the response. We enclose everything in a **`try...catch`** block so that the exceptions thrown by the `readInt` and `writeInt` methods are handled.

The socket example here is clearly rather elementary. Java provides a very wide range of possibilities for communication via sockets, for example secure sockets and sockets for multicasting. This is beyond the scope of this book, but it is hoped that we have given you a flavour for what is available so that those of you who want to develop your skills in this area are able to move forward. To help you do that we are going to provide two more examples. Firstly we will show you how to extend our addition server so that it accepts multiple clients at the same time. Secondly we are going to develop a rather more complex example, namely a chat application.

23.5 Connections from Multiple Clients

In our previous example the server could accept connections from only one client at a time; it wasn't able to listen for other connections until the first connection had disconnected. If we want the server to accept multiple connections at the same time, then each connection has to run in its own thread.

We are going to create a class that extends `Thread` which we will call `AdditionServerThread`. But in order to understand the logic, let's first look at the new version of the addition server (`AdditionServerMultiple`) which is going to use this class:

AdditionServerMultiple

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class AdditionServerMultiple
{

    public static void main(String[] args)
    {
        final int port = 8901;
        AdditionServerThread thread;
        Socket socket;

        System.out.println("Listening for connections on port: " + port);
        try
        {
            ServerSocket listenSocket = new ServerSocket(port);

            while(true) // continuously listen for connections
            {
                socket = listenSocket.accept();
                thread = new AdditionServerThread(socket);
                thread.start();
            }
        }

        catch(IOException e)
        {
        }
    }
}
```

The logic is quite straightforward. Once the new `ServerSocket` is created we enter an infinite loop that continuously listens for connections. Once a connection is made, a new thread is created and started; a reference to the socket is sent as an argument. In this way, the server is able to support multiple connections on the same port.

Now we can think about the `AdditionServerThread` class. All the functionality that we previously saw in the `AdditionServer` will be placed in the `run` method of this class. There will also be one extra feature: each client that connects will generate its own id, an integer based on the number of connections that have been made so far. The server will be able to report in the console which client is making a particular request for an addition calculation. To illustrate this, a sample session is shown in Fig. 23.4.

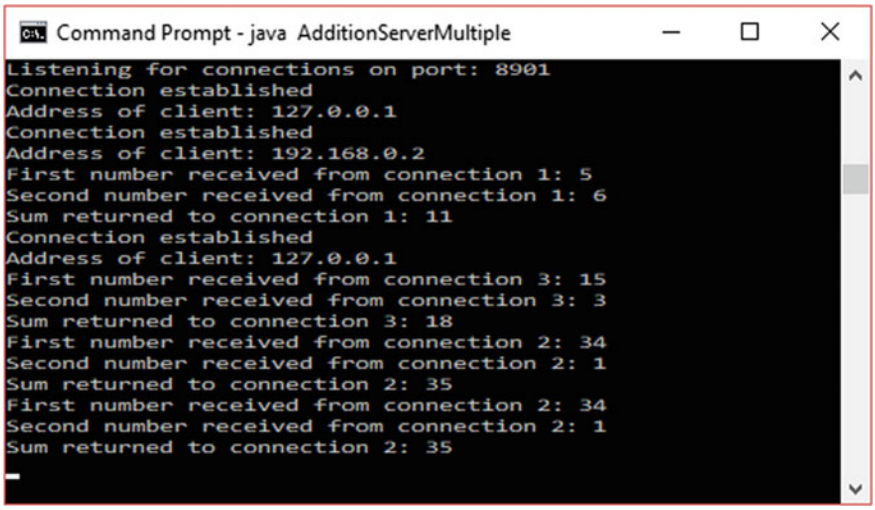

Fig. 23.4 The addition server making connections with multiple clients and responding to requests

So let's take a look at the `AdditionServerThread` class:

AdditionServerThread

```
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;

public class AdditionServerThread extends Thread
{
    private int id;
    private static int totalConnections;
    private final int port = 8901;

    // declare a "general" socket
    private final  Socket connection;

    // declare low level and high level objects for input
    private InputStream inStream;
    private DataInputStream inDataStream;

    // declare low level and high level objects for output
    private OutputStream outStream;
    DataOutputStream outDataStream;

    // declare other variables
    private String client;
    private int first, second, sum;
    private boolean connected;

    public AdditionServerThread(Socket socketIn)
    {
         connection = socketIn;
    }

    @Override
    public void run()
    {
        try
        {
            connected = true;
            System.out.println("Connection established");
```

```
            totalConnections++;     // increase the total number of connections
            id = totalConnections;  // assign an id

            // create an input stream from the client
            inStream = connection.getInputStream();
            inDataStream = new DataInputStream(inStream);

            // create an output stream to the client
            outStream = connection.getOutputStream ();
            outDataStream = new DataOutputStream(outStream);

            // wait for a string from the client
            client = inDataStream.readUTF();
            System.out.println("Address of client: " + client);

            while(connected)
            {
                // read an integer from the client
                first = inDataStream.readInt();
                System.out.println("First number received from connection " +  id + ": " + first);

                // read an integer from the client
                second = inDataStream.readInt();
                System.out.println("Second number received from connection " + id + ": " + second);

                sum = first + second;
                System.out.println("Sum returned to connection " + id + ": " + sum);

                // send the sum to the client
                outDataStream.writeInt(sum);
            }
        }

        catch(IOException e)
        {
            connected = false;
        }
    }
}
```

As we have said, the functionality now resides in the `run` method. There is nothing very new here, except for the fact that an id is assigned to the new connection, and this value is reported whenever this thread requests a calculation. You will see that a **`static`** attribute, `totalConnections`, has been declared, and within the `run` method this attribute is incremented each time a new connection is made. The current value is then assigned to the id of this connection.

You should also note that the socket on which the client is connected is received as a parameter and assigned in the constructor.

23.6 A Client–Server Chat Application

The final application that we are going to develop is a chat application. Figure 23.5 shows the sort of thing we are talking about.

The first thing to point out is that the only difference between the client and the server is the fact that initially the server waits for the client to initiate a connection —once the connection is established the behaviour is the same.

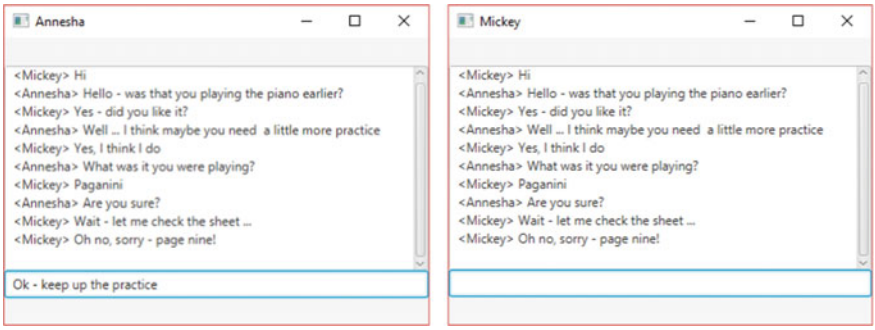

Fig. 23.5 A client–server chat application

Both the client and the server have to be able to listen for connections, and at the same time be capable of sending messages entered by the user. They will therefore need to be multi-threaded. The main thread will allow the user to enter messages which it will send to the remote program. The other thread will listen for messages from the remote application and display them in the text area.

When the thread is created it will need to receive a reference to the text area where the messages are to be displayed, and a reference to the socket connection. It will need to create an input stream which must be associated with this connection. We will be building JavaFX applications, so the threads will require the creation of `Tasks`. The `call` method of each task will be written so that the thread continuously waits for messages on the input stream and then displays them in the text area. Both the client and the server classes will need to create an object of this thread and start the thread running.

We have designed our application so that rather than having a button that has to be pressed, the message is sent and echoed in the text area when the <Enter> key is pressed. As you will see in a moment, in order to achieve this the class must provide code for the `setOnKeyReleased` method of `TextField`. This method will check whether the <Enter> key was pressed.

We'll begin by looking at the code for the server:

ChatServer

```
import java.io.DataOutputStream;
import java.io.IOException;
import java.io.OutputStream;
```

```
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Optional;
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Alert;
import javafx.scene.control.Alert.AlertType;
import javafx.scene.control.TextArea;
import javafx.scene.control.TextField;
import javafx.scene.control.TextInputDialog;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class ChatServer extends Application
{
    // declare and initialise the text display area
    private TextArea textWindow = new TextArea();


    private OutputStream outStream; // for low level output
    private DataOutputStream outDataStream; // for high level output

    private ListenerTask listener; // required for the server thread

    private final int port = 8901;
    private String name;

    @Override
    public void start(Stage stage)
    {
        getInfo(); // call method that gets user name
        startServerThread();  // start the sever thread

        TextField inputWindow = new TextField();

        // configure the behaviour of the input window
        inputWindow.setOnKeyReleased(e ->
                    {
                        String text;

                        if(e.getCode().getName().equals("Enter"))  // if the <Enter> key was pressed
                        {
                            text = "<" + name + "> " +  inputWindow.getText() + "\n";
                            textWindow.appendText(text); // echo the text
                            inputWindow.setText(""); // clear the input window

                            try
                            {
                                outDataStream.writeUTF(text); // transmit the text
                            }

                            catch(IOException ie)
                            {
                            }
                        }
                    }
                                );

        // configure the visual components
        textWindow.setEditable(false);
        textWindow.setWrapText(true);
        VBox root = new  VBox();
        root.setAlignment(Pos.CENTER);
        root.getChildren().addAll(textWindow, inputWindow);
        Scene scene = new Scene(root, 500, 300);
        stage.setScene(scene);
        stage.setTitle(name);
        stage.show();
    }

    private void startServerThread()
    {
         Socket connection; // declare a "general" socket
         ServerSocket listenSocket; // declare a server socket

         try
```

```
        {
            // create a server socket
            listenSocket = new ServerSocket(port);

            // listen for a connection from the client
            connection = listenSocket.accept();

            // create an output stream to the connection
            outStream = connection.getOutputStream ();
            outDataStream = new DataOutputStream(outStream );

            // create a thread to listen for messages
            listener = new ListenerTask(textWindow, connection);

            Thread thread = new Thread(listener);
            thread.start(); // start the thread
        }
        catch (IOException e)
        {
            textWindow.setText("An error has occured");
        }
    }

    // method to get information from user
    private void getInfo()
    {
         Optional<String> response;

        // get user name
        TextInputDialog textDialog = new TextInputDialog();
        textDialog.setHeaderText("Enter user name");
        textDialog.setTitle("Chat Server");
        response = textDialog.showAndWait();
        name =  response.get();

        // provide information to the user before starting the server thread
        Alert alert = new Alert(AlertType.INFORMATION);
        alert.setTitle("Chat Server");
        alert.setHeaderText
               ("Press OK to start server. The dialogue window will appear when a client connects.");
        alert.showAndWait();
    }

    @Override
    public void stop()
    {
        System.exit(0); // terminate application when the window is closed
    }

    public static void main(String[] args)
    {
        launch(args);
    }
}
```

We start by declaring the attributes:

```
private TextArea textWindow = new TextArea();

private OutputStream outStream;
private DataOutputStream outDataStream;

private ListenerTask listener;

private final int port = 8901;
private String name;
```

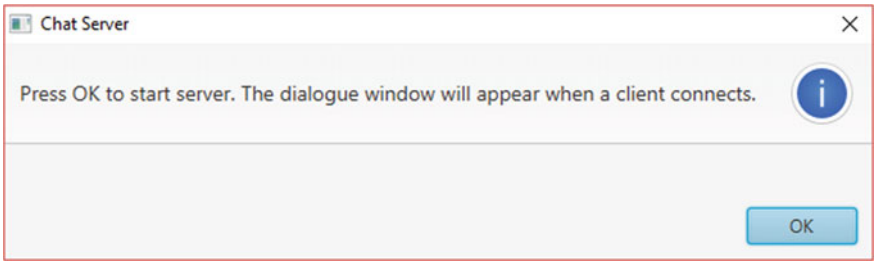

Fig. 23.6 An information alert for the chat server

We have declared and initialised a `TextArea` object, which is where the messages will be displayed. This needs to be an attribute of the class because it will later be passed to the task that listens for and displays the client messages.

Next we have declared a variable of type `ListenerTask`. This is the task that is required for the thread that listens for messages from the remote application, in this case the client; as you will see later the client will also make use of this class in order that it can receive messages from the server. You will see the code for the `ListenerTask` in a moment.

Next we declare a constant for the port number, which, for convenience, we have hard-coded, and finally we have declared a variable to hold the user name.

Now for the `start` method. We begin by calling a helper method, `getInfo`, which will prompt the user to enter a chat name; this uses the `TextInputDialog` class in the same way as you saw in the addition server example. Once the name is entered, we go on to create an information alert, in the way that we explained in Chap. 17:

```
Alert alert = new Alert(AlertType.INFORMATION);
alert.setTitle("Chat Server");
alert.setHeaderText
     ("Press OK to start server. The dialogue window will appear when a client connects.");
```

This causes the following dialogue to appear (Fig. 23.6).

Once the user has acknowledged the message by pressing the OK button, a helper method, `startServerThead` is called. As we shall see in a moment, this method begins by waiting for a connection, and for this reason it is important that we call this method before showing the scene graphic. In a JavaFX application, once the stage is shown, any routine that effectively runs in the background should be placed in a separate thread. To avoid having to create an additional thread for this purpose we have waited for the connection to be established before creating and showing the scene graphic.

```
private void startServerThread()
{
    Socket connection;
    ServerSocket listenSocket;

    try
    {

        listenSocket = new ServerSocket(port);
        connection = listenSocket.accept();

        outStream = connection.getOutputStream ();
        outDataStream = new DataOutputStream(outStream );

        listener = new ListenerTask(textWindow, connection);

        Thread thread = new Thread(listener);
        thread.start();
    }

    catch (IOException e)
    {
        textWindow.setText("An error has occured");
   }
}
```

So let's now take a closer look at the `startServerThread` method.

There is nothing here that is particularly new—you have already seen how we create a server socket and listen for a connection; and you have seen how we associate a data stream with that connection. Notice, however, the last three lines of the **`try`** block. Here we create an instance of `ListenerTask`, which we need in order to create the thread that will listen for remote messages. Notice that we send a reference to the text window and a reference to the connection. We then go on to create and start the thread.

Once the client has connected, the application goes on to deal with declaring and configuring the visual components before finally showing the scene graphic. We should draw your attention to the code for specifying the behaviour of the input window when the user types a message:

```
inputWindow.setOnKeyReleased(e ->
                    {
                        String text;

                        if(e.getCode().getName().equals("Enter"))
                        {
                            text = "<" + name + "> " +  inputWindow.getText() + "\n";
                            textWindow.appendText(text);
                            inputWindow.setText("");

                            try
                            {
                                outDataStream.writeUTF(text);
                            }

                            catch(IOException ie)
                            {
                            }
                        }
                    }
                            );
```

Each time a key is pressed and then released we check whether the key released was the <Enter> key by invoking the `getCode` method of `KeyEvent`. This returns a `KeyCode` object; the name of the key is then retrieved using the

getName method of KeyCode. If the key pressed was the <Enter> key then a string is created from the user name (in angle brackets) plus the text entered, followed by a newline character ('\n'). This string is then appended to the text area, and the input window is blanked, ready for more input. As well as echoing the user's message on the server screen it must, of course be transmitted to the client via the output stream.

We want the program to terminate when the window is closed, and in this case we have done this by implementing the stop method of the JavaFX application:

```
public void stop()
{
       System.exit(0);
}
```

The instruction System.exit(0) will terminate the system normally.

Now let's look at the code for the ListenerTask class, which forms the basis of the thread that handles messages from the remote user:

ListenerTask

```
import java.io.DataInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.Socket;
import javafx.concurrent.Task;
import javafx.scene.control.TextArea;

public class ListenerTask extends Task
{
    private InputStream inputStream; // for low level input
    private DataInputStream dataInputStream; // for high level input
    private TextArea window; // a reference to the text area where the message will be displayed
    private Socket connection; // a reference to the connection

    // constructor receives references to the text area and the connection
    public ListenerTask(TextArea windowIn, Socket connectionIn)
    {
        window = windowIn;
        connection = connectionIn;

        try
        {
            // create an input stream from the remote machine
            inputStream = connection.getInputStream();
            dataInputStream = new DataInputStream(inputStream);
        }

        catch(IOException e)
        {
        }
    }

    @Override
    protected Void call()
    {
        String msg;
        while(true)
        {
            try
            {
                msg = dataInputStream.readUTF(); // read the incoming message
                window.appendText(msg); // display the message
            }

            catch(IOException e)
            {
            }
        }
    }
}
```

As you can see, the attribute declarations include references to the objects that will be needed for the input stream, as well as a reference to a `TextArea` and a `Socket`. The constructor receives a `TextArea` object and a `Socket` object, and these are assigned to the relevant attributes. A `ListenerTask` object will therefore have access to the text window and the connection associated with the parent object. The constructor then goes on to establish the input stream:

```
public ListenerTask(TextArea windowIn, Socket connectionIn)
{
     window = windowIn;
     connection = connectionIn;

     try
     {
          inputStream = connection.getInputStream();
          dataInputStream = new DataInputStream(inputStream);
     }

     catch(IOException e)
     {
     }
}
```

Now the `call` method:

```
protected Void call()
{
    String msg;
    while(true)
    {
        try
        {
            msg = dataInputStream.readUTF();
             window.appendText(msg);
        }

        catch(IOException e)
        {
        }
    }
}
```

You can see that once the corresponding thread is started, an infinite loop is implemented so that it continuously reads messages from the data stream, and then displays the message in the text area associated with the server or client program that created the thread.

Now we can look at the client application:

ChatClient

```
import java.io.DataOutputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.Optional;
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.TextField;
import javafx.scene.control.TextArea;
import javafx.scene.control.TextInputDialog;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class ChatClient extends Application
{
    // declare and initialize the text display area
    private TextArea textWindow = new TextArea();

    private OutputStream outStream; // for low level output
    private DataOutputStream outDataStream; // for high level output

    private ListenerTask listener; // required for the cleint thread

    private int port; // to hold the port number of the server
    private  String remoteMachine; // to hold the name chosen by the user

    private String name;


    @Override
    public void start(Stage stage)
    {
        getInfo(); // call method that gets user name and server details
        startClientThread(); // start the client thread

        TextField inputWindow = new TextField();

        // configure the behaviour of the input window
        inputWindow.setOnKeyReleased(e ->
                    {
                        String text;

                        if(e.getCode().getName().equals("Enter")) // if the <Enter> key was pressed
                        {
                            text = "<" + name + "> " +  inputWindow.getText() + "\n";
                            textWindow.appendText(text); // echo the text
                            inputWindow.setText(""); // clear the input window

                            try
                            {
                                outDataStream.writeUTF(text); // transmit the text
                            }
                            catch(IOException ie)
                            {
                            }
                        }
                    }
                            );

        // configure the visual components
        textWindow.setWrapText(true);
        textWindow.setEditable(false);
        VBox root = new VBox();
        root.setAlignment(Pos.CENTER);
        root.getChildren().addAll(textWindow, inputWindow);
        Scene scene = new Scene(root, 500, 300);
        stage.setScene(scene);
        stage.setTitle(name);
        stage.show();
```

```
    }

    private void startClientThread()
    {
        Socket connection;  // declare a "general" socket

        try
        {
            // create a connection to the server
            connection = new Socket(remoteMachine, port);

            // create output stream to the connection
            outStream = connection.getOutputStream();
            outDataStream = new DataOutputStream (outStream);

            // create a thread to listen for messages
            listener = new ListenerTask(textWindow, connection);
            Thread thread = new Thread(listener);
            thread.start(); // start the thread
        }

       catch(UnknownHostException e)
       {
            textWindow.setText("Unknown host");
       }

       catch (IOException e)
       {
           textWindow.setText("An error has occured");
       }
   }

   // method to get information from user
   private void getInfo()
   {
       Optional<String> response;

       // get address of host
       TextInputDialog textDialog1 = new TextInputDialog();
       textDialog1.setHeaderText("Enter remote host");
       textDialog1.setTitle("Chat Client");
       response = textDialog1.showAndWait();
       remoteMachine = response.get();

       // get port number
       TextInputDialog textDialog2 = new TextInputDialog();
       textDialog2.setHeaderText("Enter port number");
       textDialog2.setTitle("Chat Client");
       response = textDialog2.showAndWait();
       port = Integer.valueOf(response.get());

       // get user name
       TextInputDialog textDialog3 = new TextInputDialog();
       textDialog3.setHeaderText("Enter user name");
       textDialog3.setTitle("Chat Client");
       response = textDialog3.showAndWait();
       name =  response.get();

   }

   @Override
   public void stop()
   {
       System.exit(0); // terminate application when the window is closed
   }

   public static void main(String[] args)
   {
       launch(args);
   }
}
```

As you can see there is not a great deal of difference between the client and the server. The only significant differences are:

- The client needs to know the address of the host that is running the server, so there is an additional attribute, a string, to hold this address; the `port` attribute is not assigned a value when it is declared, but instead is given a value by the

user. Thus, in addition to asking the user to choose a name, the `getInfo` method now also requests information about the host machine and the port number on which the server is listening.
- In the `startClientThread` method there is no need for a `ServerSocket`; instead the socket is created by establishing the connection with the remote machine:

```
connection = new Socket(remoteMachine, port);
```

You are now in a position to test out our chat application—you will need to know the name or local IP address of the machine running the server. If you don't have access to two machines, then you can run both programs on the same machine—although this rather takes away the point! In the end of chapter exercises you are given some help with how to do this.

23.7 Self-test Questions

1. Explain what is meant by each of the following terms:

   (a) client;
   (b) server;
   (c) host;
   (d) port;
   (e) socket.

2. Explain the principles of *client–server* architecture, and describe how this is implemented in Java.
3. Which functions are provided by the Java `Socket` class?
4. Which additional functions are provided by the Java `ServerSocket` class?

23.8 Programming Exercises

1. Implement the `AdditionServer` and `AdditionClient` programs from this chapter. If you have more than one computer running on the same network you can run these programs on different machines. The client will need to be supplied with either the name or the local IP address of the host machine.

If both client and server are running on the same machine, you can use "localhost" as the name, or you can use the IP address 127.0.0.1, which references the local machine.

2. Implement the version of the addition server that accepts multiple clients, and test this out by connecting a number of clients. These can be on the same machine as the server, on remote machines, or a combination.

3. Implement and test out the chat application from this chapter. Again you can run both client and server on the same machine, but it is, of course, more fun to run them from different computers. Just a note, that if you are running them on the same machine, the applications will appear on top of one another, so you will need to move one out of the way to see the other one.

4. Write a server application that tells jokes to the client, and lets the client respond. A good example would be a classic "Knock Knock" joke. The client would receive the message "Knock Knock" from the server, and would be expected to reply "Who's there?" and so on.
   You might be able to think of variations to this program. You could adapt it, for example, so that the a different joke is told each time a client connects (that is, if you actually know that many "Knock Knock" jokes!). Or perhaps a series of jokes could be told. You might also want to try allowing multiple clients to connect.

5 Try to devise a two- (or even more) player game that could be played across a network. An example might be noughts-and-crosses. The best approach would be to create a server that can deal with multiple connections—take a look at Sect. 23.5 to help with ideas for implementation.