Tutorials & Code Camps

Lesson 1: Socket Communications

Lesson 1: Socket Communications

[<<BACK] [CONTENTS] [NEXT>>]

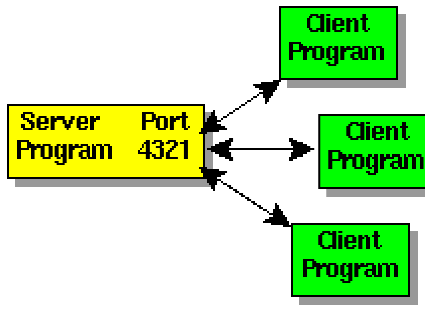
Java Programming Language Basics, Part 1, finished with a simple network communications example using the Remote Method Invocation (RMI) application programming interface (API). The RMI example allows multiple client programs to communicate with the same server program without any explicit code to do this because the RMI API is built on sockets and threads.

This lesson presents a simple sockets-based program to introduce the concepts of sockets and multi-threaded programming. A multi-threaded program performs multiple tasks at one time such as fielding simultaneous requests from many client programs.

> **Note:** See Creating a Threaded Slide Show Applet for another example of how multiple threads can be used in a program.

What are Sockets and Threads?

A socket is a software endpoint that establishes bidirectional communication between a server program and one or more client programs. The socket associates the server program with a specific hardware port on the machine where it runs so any client program anywhere in the network with a socket associated with that same port can communicate with the server program.



A server program typically provides resources to a network of client programs. Client programs send requests to the server program, and the server program responds to the request.

One way to handle requests from more than one client is to make the server program multi-threaded. A multi-threaded server creates a thread for each communication it accepts from a client. A thread is a sequence of instructions that run independently of the program and of any other threads.

Using threads, a multi-threaded server program can accept a connection from a client, start a thread for that communication, and continue listening for requests from other clients.

About the Examples

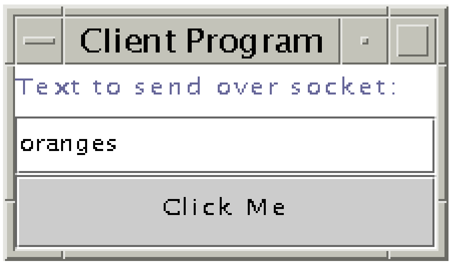
The examples for this lesson consist of two versions of the client and server program pair adapted from the FileIO.java application presented in Part 1, Lesson 6: File Access and Permissions.

Example 1 sets up a client and server communication between one server program and one client program. The server program is not multi-threaded and cannot handle requests from more than one client.

Example 2 converts the server program to a multi-threaded version so it can handle requests from more than one client.

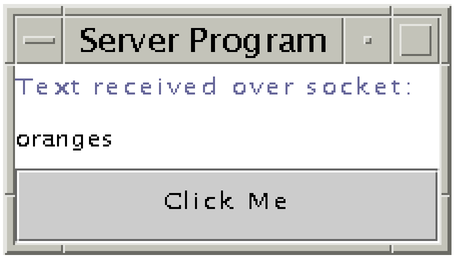
Example 1: Client-Side Behavior

The client program presents a simple user interface and prompts for text input. When you click the `Click Me` button, the text is sent to the server program. The client program expects an echo from the server and prints the echo it receives on its standard output.



Example 1: Server-Side Behavior

The server program presents a simple user interface, and when you click the `Click Me` button, the text received from the client is displayed. The server echoes the text it receives whether or not you click the `Click Me` button.



Example 1: Compile and Run

To run the example programs, start the server program first. If you do not, the client program cannot establish the socket connection. Here are the compiler and interpreter commands to compile and run the example.

```
javac SocketServer.java
javac SocketClient.java

java SocketServer
java SocketClient
```

Example 1: Server-Side Program

The server program establishes a socket connection on Port 4321 in its `listenSocket` method. It reads data sent to it and sends that same data back to the server in its `actionPerformed` method.

listenSocket Method

The `listenSocket` method creates a `ServerSocket` object with the port number on which the server program is going to listen for client communications. The port number must be an available port, which means the number cannot be reserved or already in use. For example, Unix systems reserve ports 1 through 1023 for administrative functions leaving port numbers greater than 1024 available for use.

```
public void listenSocket(){
  try{
    server = new ServerSocket(4321);
  } catch (IOException e) {
    System.out.println("Could not listen on port 4321");
    System.exit(-1);
  }
```

Next, the `listenSocket` method creates a `Socket` connection for the requesting client. This code executes when a client starts up and requests the connection on the host and port where this server program is running. When the connection is successfully established, the `server.accept` method returns a new `Socket` object.

```
  try{
    client = server.accept();
  } catch (IOException e) {
    System.out.println("Accept failed: 4321");
    System.exit(-1);
  }
```

Then, the `listenSocket` method creates a `BufferedReader` object to read the data sent over the socket connection from the `client` program. It also creates a `PrintWriter` object to send the data received from the client back to the server.

```
    try{
     in = new BufferedReader(new InputStreamReader(
                               client.getInputStream()));
     out = new PrintWriter(client.getOutputStream(),
                        true);
    } catch (IOException e) {
      System.out.println("Read failed");
      System.exit(-1);
    }
  }
```

Lastly, the `listenSocket` method loops on the input stream to read data as it comes in from the client and writes to the output stream to send the data back.

```
      while(true){
        try{
          line = in.readLine();
  //Send data back to client
          out.println(line);
        } catch (IOException e) {
          System.out.println("Read failed");
          System.exit(-1);
        }
      }
```

actionPerformed Method

The `actionPerformed` method is called by the Java platform for action events such as button clicks. This `actionPerformed` method uses the text stored in the `line` object to initialize the `textArea` object so the retrieved text can be displayed to the end user.

```
  public void actionPerformed(ActionEvent event) {
     Object source = event.getSource();

     if(source == button){
         textArea.setText(line);
     }
  }
```

Example 1: Client-Side Program

The client program establishes a connection to the server program on a particular host and port number in its `listenSocket` method, and sends the data entered by the end user to the server program in its `actionPerformed` method. The `actionPerformed` method also receives the data back from the server and prints it to the command line.

listenSocket Method

The `listenSocket` method first creates a `Socket` object with the computer name (`kq6py`) and port number (4321) where the server program is listening for client connection requests. Next, it creates a `PrintWriter` object to send data over the socket connection to the server program. It also creates a `BufferedReader` object to read the text sent by the server back to the client.

```
  public void listenSocket(){
  //Create socket connection
     try{
       socket = new Socket("kq6py", 4321);
       out = new PrintWriter(socket.getOutputStream(),
                   true);
       in = new BufferedReader(new InputStreamReader(
                   socket.getInputStream()));
     } catch (UnknownHostException e) {
       System.out.println("Unknown host: kq6py");
       System.exit(1);
     } catch  (IOException e) {
       System.out.println("No I/O");
       System.exit(1);
     }
  }
```

actionPerformed Method

The `actionPerformed` method is called by the Java platform for action events such as button clicks. This actionPerformed method code gets the text in the Textfield object and passes it to the PrintWriter object, which then sends it over the socket connection to the server program.

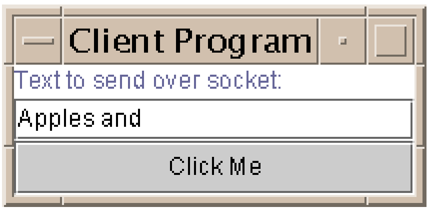
The actionPerformed method then makes the Textfield object blank so it is ready for more end user input. Lastly, it receives the text sent back to it by the server and prints the text out.

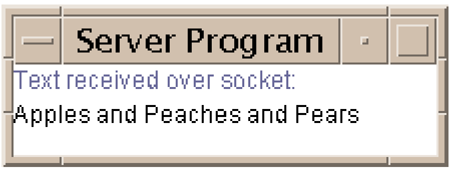
```
public void actionPerformed(ActionEvent event){
   Object source = event.getSource();

   if(source == button){
//Send data over socket
      String text = textField.getText();
      out.println(text);
      textField.setText(new String(""));
      out.println(text);
   }
//Receive text from server
   try{
     String line = in.readLine();
     System.out.println("Text received: " + line);
   } catch (IOException e){
     System.out.println("Read failed");
     System.exit(1);
   }
}
```

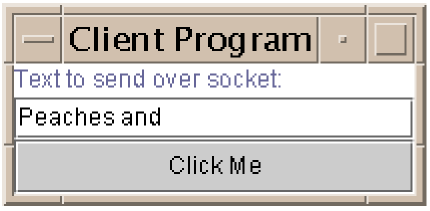
Example 2: Multithreaded Server Example

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 program.

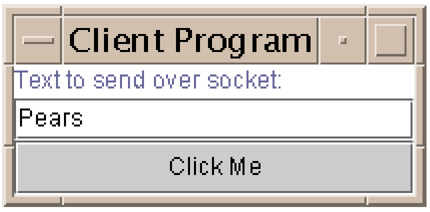

Client Program
Text to send over socket:
Apples and
Click Me


Server Program
Text received over socket:
Apples and Peaches and Pears

First Client


Client Program
Text to send over socket:
Peaches and
Click Me

Second Client


Client Program
Text to send over socket:
Pears
Click Me

Third Client

The multithreaded server program creates a new thread for every client request. This way each client has its own connection to the server for passing data back and forth. When running multiple threads, you have to be sure that one thread cannot interfere with the data in another thread.

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`. The `run` method executes independently in each thread. If three clients request connections, three `ClientWorker` instances are created, a thread is started for each `ClientWorker` instance, and the `run` 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);
       }
    }
  }
}
```

The `JTextArea.append` method is thread safe, which means its implementation includes code that allows one thread to finish its append operation before another thread can start an append operation. This prevents one thread from overwriting all or part of a string of appended text and corrupting the output. If the `JTextArea.append` method were not thread safe, you would need to wrap the call to `textArea.append(line)` in a `synchronized` method and replace the `run` method call to `textArea.append(line)` with a call to `appendText(line)`:

```
 public synchronized void appendText(line){
     textArea.append(line);
   }
```

The `synchronized` keyword means this thread has a lock on the `textArea` and no other thread can change the `textArea` until this thread finishes its append operation.

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);
      }
    }
```

More Information

You can find more information on sockets in the All About Sockets section in The Java Tutorial.

[TOP]