**Socket Programming:**

Sockets provide the communication mechanism between two computers using TCP. A client program creates a socket on its end of the communication and attempts to connect that socket to a server.

When the connection is made, the server creates a socket object on its end of the communication. The client and server can now communicate by writing to and reading from the socket.

The java.net.Socket class represents a socket, and the java.net.ServerSocket class provides a mechanism for the server program to listen for clients and establish connections with them.

The following steps occur when establishing a TCP connection between two computers using sockets:

* The server instantiates a ServerSocket object, denoting which port number communication is to occur on.
* The server invokes the accept() method of the ServerSocket class. This method waits until a client connects to the server on the given port.
* After the server is waiting, a client instantiates a Socket object, specifying the server name and port number to connect to.
* The constructor of the Socket class attempts to connect the client to the specified server and port number. If communication is established, the client now has a Socket object capable of communicating with the server.
* On the server side, the accept() method returns a reference to a new socket on the server that is connected to the client's socket.

**ServerSocket Class Methods:**

The **java.net.ServerSocket** class is used by server applications to obtain a port and listen for client requests

The ServerSocket class has four constructors:

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | **public ServerSocket(int port) throws IOException** Attempts to create a server socket bound to the specified port. An exception occurs if the port is already bound by another application. |
| 2 | **public ServerSocket(int port, int backlog) throws IOException** Similar to the previous constructor, the backlog parameter specifies how many incoming clients to store in a wait queue. |
| 3 | **public ServerSocket(int port, int backlog, InetAddress address) throws IOException** Similar to the previous constructor, the InetAddress parameter specifies the local IP address to bind to. The InetAddress is used for servers that may have multiple IP addresses, allowing the server to specify which of its IP addresses to accept client requests on |
| 4 | **public ServerSocket() throws IOException** Creates an unbound server socket. When using this constructor, use the bind() method when you are ready to bind the server socket |

If the ServerSocket constructor does not throw an exception, it means that your application has successfully bound to the specified port and is ready for client requests.

Here are some of the common methods of the ServerSocket class:

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | **public int getLocalPort()** Returns the port that the server socket is listening on. This method is useful if you passed in 0 as the port number in a constructor and let the server find a port for you. |
| 2 | **public Socket accept() throws IOException** Waits for an incoming client. This method blocks until either a client connects to the server on the specified port or the socket times out, assuming that the time-out value has been set using the setSoTimeout() method. Otherwise, this method blocks indefinitely |
| 3 | **public void setSoTimeout(int timeout)** Sets the time-out value for how long the server socket waits for a client during the accept(). |
| 4 | **public void bind(SocketAddress host, int backlog)** Binds the socket to the specified server and port in the SocketAddress object. Use this method if you instantiated the ServerSocket using the no-argument constructor. |

When the ServerSocket invokes accept(), the method does not return until a client connects. After a client does connect, the ServerSocket creates a new Socket on an unspecified port and returns a reference to this new Socket. A TCP connection now exists between the client and server, and communication can begin.

**Socket Class Methods:**

The **java.net.Socket** class represents the socket that both the client and server use to communicate with each other. The client obtains a Socket object by instantiating one, whereas the server obtains a Socket object from the return value of the accept() method.

The Socket class has five constructors that a client uses to connect to a server:

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | **public Socket(String host, int port) throws UnknownHostException, IOException.** This method attempts to connect to the specified server at the specified port. If this constructor does not throw an exception, the connection is successful and the client is connected to the server. |
| 2 | **public Socket(InetAddress host, int port) throws IOException** This method is identical to the previous constructor, except that the host is denoted by an InetAddress object. |
| 3 | **public Socket(String host, int port, InetAddress localAddress, int localPort) throws IOException.** Connects to the specified host and port, creating a socket on the local host at the specified address and port. |
| 4 | **public Socket(InetAddress host, int port, InetAddress localAddress, int localPort) throws IOException.** This method is identical to the previous constructor, except that the host is denoted by an InetAddress object instead of a String |
| 5 | **public Socket()** Creates an unconnected socket. Use the connect() method to connect this socket to a server. |

When the Socket constructor returns, it does not simply instantiate a Socket object but it actually attempts to connect to the specified server and port.

Some methods of interest in the Socket class are listed here. Notice that both the client and server have a Socket object, so these methods can be invoked by both the client and server.

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | **public void connect(SocketAddress host, int timeout) throws IOException** This method connects the socket to the specified host. This method is needed only when you instantiated the Socket using the no-argument constructor. |
| 2 | **public InetAddress getInetAddress()** This method returns the address of the other computer that this socket is connected to. |
| 3 | **public int getPort()** Returns the port the socket is bound to on the remote machine. |
| 4 | **public int getLocalPort()** Returns the port the socket is bound to on the local machine. |
| 5 | **public SocketAddress getRemoteSocketAddress()** Returns the address of the remote socket. |
| 6 | **public InputStream getInputStream() throws IOException** Returns the input stream of the socket. The input stream is connected to the output stream of the remote socket. |
| 7 | **public OutputStream getOutputStream() throws IOException** Returns the output stream of the socket. The output stream is connected to the input stream of the remote socket |
| 8 | **public void close() throws IOException** Closes the socket, which makes this Socket object no longer capable of connecting again to any server |

**InetAddress Class Methods:**

This class represents an Internet Protocol (IP) address. Here are following usefull methods which you would need while doing socket programming:

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | **static InetAddress getByAddress(byte[] addr)** Returns an InetAddress object given the raw IP address . |
| 2 | **static InetAddress getByAddress(String host, byte[] addr)** Create an InetAddress based on the provided host name and IP address. |
| 3 | **static InetAddress getByName(String host)** Determines the IP address of a host, given the host's name. |
| 4 | **String getHostAddress()**  Returns the IP address string in textual presentation. |
| 5 | **String getHostName()**  Gets the host name for this IP address. |
| 6 | **static InetAddress InetAddress getLocalHost()** Returns the local host. |
| 7 | **String toString()** Converts this IP address to a String. |

**Socket Client Example:**

The following GreetingClient is a client program that connects to a server by using a socket and sends a greeting, and then waits for a response.

// File Name GreetingClient.java

import java.net.\*;

import java.io.\*;

public class GreetingClient

{

public static void main(String [] args)

{

String serverName = args[0];

int port = Integer.parseInt(args[1]);

try

{

System.out.println("Connecting to " + serverName

+ " on port " + port);

Socket client = new Socket(serverName, port);

System.out.println("Just connected to "

+ client.getRemoteSocketAddress());

OutputStream outToServer = client.getOutputStream();

DataOutputStream out =

new DataOutputStream(outToServer);

out.writeUTF("Hello from "

+ client.getLocalSocketAddress());

InputStream inFromServer = client.getInputStream();

DataInputStream in =

new DataInputStream(inFromServer);

System.out.println("Server says " + in.readUTF());

client.close();

}catch(IOException e)

{

e.printStackTrace();

}

}

}

**Socket Server Example:**

The following GreetingServer program is an example of a server application that uses the Socket class to listen for clients on a port number specified by a command-line argument:

// File Name GreetingServer.java

import java.net.\*;

import java.io.\*;

public class GreetingServer extends Thread

{

private ServerSocket serverSocket;

public GreetingServer(int port) throws IOException

{

serverSocket = new ServerSocket(port);

serverSocket.setSoTimeout(10000);

}

public void run()

{

while(true)

{

try

{

System.out.println("Waiting for client on port " +

serverSocket.getLocalPort() + "...");

Socket server = serverSocket.accept();

System.out.println("Just connected to "

+ server.getRemoteSocketAddress());

DataInputStream in =

new DataInputStream(server.getInputStream());

System.out.println(in.readUTF());

DataOutputStream out =

new DataOutputStream(server.getOutputStream());

out.writeUTF("Thank you for connecting to "

+ server.getLocalSocketAddress() + "\nGoodbye!");

server.close();

}catch(SocketTimeoutException s)

{

System.out.println("Socket timed out!");

break;

}catch(IOException e)

{

e.printStackTrace();

break;

}

}

}

public static void main(String [] args)

{

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

try

{

Thread t = new GreetingServer(port);

t.start();

}catch(IOException e)

{

e.printStackTrace();

}

}

}

Compile client and server and then start server as follows:

$ java GreetingServer 6066

Waiting for client on port 6066...

Check client program as follows:

$ java GreetingClient localhost 6066

Connecting to localhost on port 6066

Just connected to localhost/127.0.0.1:6066

Server says Thank you for connecting to /127.0.0.1:6066

Goodbye!

### A bit of history

The Unix input/output (I/O) system follows a paradigm usually referred to as Open-Read-Write-Close. Before a user process can perform I/O operations, it calls Open to specify and obtain permissions for the file or device to be used. Once an object has been opened, the user process makes one or more calls to Read or Write data. Read reads data from the object and transfers it to the user process, while Write transfers data from the user process to the object. After all transfer operations are complete, the user process calls Close to inform the operating system that it has finished using that object.

### More about Java I/O

Learn how to squeeze maximum performance out of Java's nonblocking I/O classes with these how-tos:

* [Master Merlin's new I/O classes](http://www.javaworld.com/article/2075575/core-java/master-merlin-s-new-i-o-classes.html)
* [Five ways to maximize Java NIO and NIO.2](http://www.javaworld.com/article/2078654/java-se/five-ways-to-maximize-java-nio-and-nio-2.html)

When facilities for InterProcess Communication (IPC) and networking were added to Unix, the idea was to make the interface to IPC similar to that of file I/O. In Unix, a process has a set of I/O descriptors that one reads from and writes to. These descriptors may refer to files, devices, or communication channels (sockets). The lifetime of a descriptor is made up of three phases: creation (open socket), reading and writing (receive and send to socket), and destruction (close socket).

The IPC interface in BSD-like versions of Unix is implemented as a layer over the network TCP and UDP protocols. Message destinations are specified as socket addresses; each socket address is a communication identifier that consists of a port number and an Internet address.

The IPC operations are based on socket pairs, one belonging to a communication process. IPC is done by exchanging some data through transmitting that data in a message between a socket in one process and another socket in another process. When messages are sent, the messages are queued at the sending socket until the underlying network protocol has transmitted them. When they arrive, the messages are queued at the receiving socket until the receiving process makes the necessary calls to receive them.

### TCP/IP and UDP/IP communications

There are two communication protocols that one can use for socket programming: datagram communication and stream communication.

**Datagram communication:**

The datagram communication protocol, known as UDP (user datagram protocol), is a connectionless protocol, meaning that each time you send datagrams, you also need to send the local socket descriptor and the receiving socket's address. As you can tell, additional data must be sent each time a communication is made.

**Stream communication:**

The stream communication protocol is known as TCP (transfer control protocol). Unlike UDP, TCP is a connection-oriented protocol. In order to do communication over the TCP protocol, a connection must first be established between the pair of sockets. While one of the sockets listens for a connection request (server), the other asks for a connection (client). Once two sockets have been connected, they can be used to transmit data in both (or either one of the) directions.

Now, you might ask what protocol you should use -- UDP or TCP? This depends on the client/server application you are writing. The following discussion shows the differences between the UDP and TCP protocols; this might help you decide which protocol you should use.

In UDP, as you have read above, every time you send a datagram, you have to send the local descriptor and the socket address of the receiving socket along with it. Since TCP is a connection-oriented protocol, on the other hand, a connection must be established before communications between the pair of sockets start. So there is a connection setup time in TCP.

Popular on JavaWorld

* 

[Java programming with lambda expressions](http://www.javaworld.com/article/2092260/java-se/java-programming-with-lambda-expressions.html)

* [Hello, OSGi, Part 1: Bundles for beginners](http://www.javaworld.com/article/2077837/java-se/hello--osgi--part-1--bundles-for-beginners.html)
* 

[Review: 10 JavaScript editors and IDEs put to the test](http://www.javaworld.com/article/2094847/enterprise-java/review-10-javascript-editors-and-ides-put-to-the-test.html)

In UDP, there is a size limit of 64 kilobytes on datagrams you can send to a specified location, while in TCP there is no limit. Once a connection is established, the pair of sockets behaves like streams: All available data are read immediately in the same order in which they are received.

UDP is an unreliable protocol -- there is no guarantee that the datagrams you have sent will be received in the same order by the receiving socket. On the other hand, TCP is a reliable protocol; it is guaranteed that the packets you send will be received in the order in which they were sent.

In short, TCP is useful for implementing network services -- such as remote login (rlogin, telnet) and file transfer (FTP) -- which require data of indefinite length to be transferred. UDP is less complex and incurs fewer overheads. It is often used in implementing client/server applications in distributed systems built over local area networks.

### Programming sockets in Java

In this section we will answer the most frequently asked questions about programming sockets in Java. Then we will show some examples of how to write client and server applications.

**Note:** In this tutorial we will show how to program sockets in Java using the TCP/IP protocol only since it is more widely used than UDP/IP. ***Also:*** All the classes related to sockets are in the java.net package, so make sure to import that package when you program sockets.

How do I open a socket?

If you are programming a client, then you would open a socket like this:

Recent Java How-Tos

* [Autoboxing, Unboxing, and NoSuchMethodError](http://www.javaworld.com/article/2597545/java-language/autoboxing-unboxing-and-nosuchmethoderror.html)
* 

[Shipilёv on writing good benchmarks -- Java vs Scala](http://www.javaworld.com/article/2465421/core-java/shipil-v-on-writing-good-benchmarks-java-vs-scala.html)

* 

[Open source Java projects: Spring Batch](http://www.javaworld.com/article/2458888/spring-framework/open-source-java-projects-spring-batch.html)

Socket MyClient;

MyClient = new Socket("Machine name", PortNumber);

Where Machine name is the machine you are trying to open a connection to, and PortNumber is the port (a number) on which the server you are trying to connect to is running. When selecting a port number, you should note that port numbers between 0 and 1,023 are reserved for privileged users (that is, super user or root). These port numbers are reserved for standard services, such as email, FTP, and HTTP. When selecting a port number for your server, select one that is greater than 1,023!

In the example above, we didn't make use of exception handling, however, it is a good idea to handle exceptions. (From now on, all our code will handle exceptions!) The above can be written as:

Socket MyClient;

try {

MyClient = new Socket("Machine name", PortNumber);

}

catch (IOException e) {

System.out.println(e);

}

If you are programming a server, then this is how you open a socket:

ServerSocket MyService;

try {

MyServerice = new ServerSocket(PortNumber);

}

catch (IOException e) {

System.out.println(e);

}

When implementing a server you also need to create a socket object from the ServerSocket in order to listen for and accept connections from clients.

Socket clientSocket = null;

try {

serviceSocket = MyService.accept();

}

catch (IOException e) {

System.out.println(e);

}

How do I create an input stream?

On the client side, you can use the DataInputStream class to create an input stream to receive response from the server:

DataInputStream input;

try {

input = new DataInputStream(MyClient.getInputStream());

}

catch (IOException e) {

System.out.println(e);

}

The class DataInputStream allows you to read lines of text and Java primitive data types in a portable way. It has methods such as read, readChar, readInt, readDouble, and readLine,. Use whichever function you think suits your needs depending on the type of data that you receive from the server.

On the server side, you can use DataInputStream to receive input from the client:

DataInputStream input;

try {

input = new DataInputStream(serviceSocket.getInputStream());

}

catch (IOException e) {

System.out.println(e);

}

How do I create an output stream?

On the client side, you can create an output stream to send information to the server socket using the class PrintStream or DataOutputStream of java.io:

PrintStream output;

try {

output = new PrintStream(MyClient.getOutputStream());

}

catch (IOException e) {

System.out.println(e);

}

The class PrintStream has methods for displaying textual representation of Java primitive data types. Its Write and println methods are important here. Also, you may want to use the DataOutputStream:

DataOutputStream output;

try {

output = new DataOutputStream(MyClient.getOutputStream());

}

catch (IOException e) {

System.out.println(e);

}

The class DataOutputStream allows you to write Java primitive data types; many of its methods write a single Java primitive type to the output stream. The method writeBytes is a useful one.

On the server side, you can use the class PrintStream to send information to the client.

PrintStream output;

try {

output = new PrintStream(serviceSocket.getOutputStream());

}

catch (IOException e) {

System.out.println(e);

}

Note: You can use the class *DataOutputStream* as mentioned above.

How do I close sockets?

You should always close the output and input stream before you close the socket.

On the client side:

try {

output.close();

input.close();

MyClient.close();

}

catch (IOException e) {

System.out.println(e);

}

[Popular Resources](http://www.javaworld.com/resources)

* 

White Paper

[Troubleshooting Common Issues in VoIP](http://resources.javaworld.com/ccd/assets/62297/detail)

* 

White Paper

[Global Growing Pains: Tapping into B2B Integration Services to Overcome Global Expansion Challenges](http://resources.javaworld.com/ccd/assets/62240/detail)

[See All](http://www.javaworld.com/resources)

Top of Form



Bottom of Form

On the server side:

try {

output.close();

input.close();

serviceSocket.close();

MyService.close();

}

catch (IOException e) {

System.out.println(e);

}

### Examples

In this section we will write two applications: a simple SMTP (simple mail transfer protocol) client, and a simple echo server.

1. SMTP client

Let's write an SMTP (simple mail transfer protocol) client -- one so simple that we have all the data encapsulated within the program. You may change the code around to suit your needs. An interesting modification would be to change it so that you accept the data from the command-line argument and also get the input (the body of the message) from standard input. Try to modify it so that it behaves the same as the mail program that comes with Unix.

import java.io.\*;

import java.net.\*;

public class smtpClient {

public static void main(String[] args) {

// declaration section:

// smtpClient: our client socket

// os: output stream

// is: input stream

Socket smtpSocket = null;

DataOutputStream os = null;

DataInputStream is = null;

// Initialization section:

// Try to open a socket on port 25

// Try to open input and output streams

try {

smtpSocket = new Socket("hostname", 25);

os = new DataOutputStream(smtpSocket.getOutputStream());

is = new DataInputStream(smtpSocket.getInputStream());

} catch (UnknownHostException e) {

System.err.println("Don't know about host: hostname");

} catch (IOException e) {

System.err.println("Couldn't get I/O for the connection to: hostname");

}

// If everything has been initialized then we want to write some data

// to the socket we have opened a connection to on port 25

if (smtpSocket != null && os != null && is != null) {

try {

// The capital string before each colon has a special meaning to SMTP

// you may want to read the SMTP specification, RFC1822/3

os.writeBytes("HELO\n");

os.writeBytes("MAIL From: k3is@fundy.csd.unbsj.ca\n");

os.writeBytes("RCPT To: k3is@fundy.csd.unbsj.ca\n");

os.writeBytes("DATA\n");

os.writeBytes("From: k3is@fundy.csd.unbsj.ca\n");

os.writeBytes("Subject: testing\n");

os.writeBytes("Hi there\n"); // message body

os.writeBytes("\n.\n");

os.writeBytes("QUIT");

// keep on reading from/to the socket till we receive the "Ok" from SMTP,

// once we received that then we want to break.

String responseLine;

while ((responseLine = is.readLine()) != null) {

System.out.println("Server: " + responseLine);

if (responseLine.indexOf("Ok") != -1) {

break;

}

}

// clean up:

// close the output stream

// close the input stream

// close the socket

os.close();

is.close();

smtpSocket.close();

} catch (UnknownHostException e) {

System.err.println("Trying to connect to unknown host: " + e);

} catch (IOException e) {

System.err.println("IOException: " + e);

}

}

}

}

When programming a client, you must follow these four steps:

* Open a socket.
* Open an input and output stream to the socket.
* Read from and write to the socket according to the server's protocol.
* Clean up.

These steps are pretty much the same for all clients. The only step that varies is step three, since it depends on the server you are talking to.

2. Echo server

Now let's write a server. This server is very similar to the echo server running on port 7. Basically, the echo server receives text from the client and then sends that exact text back to the client. This is just about the simplest server you can write. Note that this server handles only one client. Try to modify it to handle multiple clients using threads.

import java.io.\*;

import java.net.\*;

public class echo3 {

public static void main(String args[]) {

// declaration section:

// declare a server socket and a client socket for the server

// declare an input and an output stream

ServerSocket echoServer = null;

String line;

DataInputStream is;

PrintStream os;

Socket clientSocket = null;

// Try to open a server socket on port 9999

// Note that we can't choose a port less than 1023 if we are not

// privileged users (root)

try {

echoServer = new ServerSocket(9999);

}

catch (IOException e) {

System.out.println(e);

}

// Create a socket object from the ServerSocket to listen and accept

// connections.

// Open input and output streams

try {

clientSocket = echoServer.accept();

is = new DataInputStream(clientSocket.getInputStream());

os = new PrintStream(clientSocket.getOutputStream());

// As long as we receive data, echo that data back to the client.

while (true) {

line = is.readLine();

os.println(line);

}

}

catch (IOException e) {

System.out.println(e);

}

}

}

Featured Resource



Presented by Citrix Systems

[10 essential elements for a secure enterprise mobility strategy](http://resources.javaworld.com/ccd/assets/61678/detail)

Best practices for protecting sensitive business information while making people productive from

[Learn More](http://resources.javaworld.com/ccd/assets/61678/detail)

### Conclusion

Programming client/server applications is challenging and fun, and programming this kind of application in Java is easier than doing it in other languages, such as C. Socket programming in Java is seamless.

The java.net package provides a powerful and flexibile infrastructure for network programming, so you are encouraged to refer to that package if you would like to know the classes that are provided.

Sun.\* packages have some good classes for networking, however you are not encouraged to use those classes at the moment because they may change in the next release. Also, some of the classes are not portable across all platforms.