

JAVA - NETWORKING

The term *network programming* refers to writing programs that execute across multiple devices *computers*, in which the devices are all connected to each other using a network.

The `java.net` package of the J2SE APIs contains a collection of classes and interfaces that provide the low-level communication details, allowing you to write programs that focus on solving the problem at hand.

The `java.net` package provides support for the two common network protocols:

- **TCP:** TCP stands for Transmission Control Protocol, which allows for reliable communication between two applications. TCP is typically used over the Internet Protocol, which is referred to as TCP/IP.
- **UDP:** UDP stands for User Datagram Protocol, a connection-less protocol that allows for packets of data to be transmitted between applications.

This tutorial gives good understanding on the following two subjects:

- **Socket Programming:** This is most widely used concept in Networking and it has been explained in very detail.
- **URL Processing:** This would be covered separately. Click here to learn about [URL Processing](#) in Java language.

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.

After the connections are established, communication can occur using I/O streams. Each socket has both an `OutputStream` and an `InputStream`. The client's `OutputStream` is connected to the server's `InputStream`, and the client's `InputStream` is connected to the server's `OutputStream`.

TCP is a twoway communication protocol, so data can be sent across both streams at the same time. There are following usefull classes providing complete set of methods to implement sockets.

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(intport) 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(intport, intbacklog) throws IOException**

Similar to the previous constructor, the backlog parameter specifies how many incoming clients to store in a wait queue.

3 **public ServerSocket(intport, intbacklog, InetAddressaddress) 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(inttimeout)**

Sets the time-out value for how long the server socket waits for a client during the accept.

4 **public void bind(SocketAddresshost, intbacklog)**

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 getByAddressbyte[]addr**

Returns an InetAddress object given the raw IP address .

2 **static InetAddress getByAddressStringhost, byte[]addr**

Create an InetAddress based on the provided host name and IP address.

3 **static InetAddress getByNameStringhost**

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!

```

Loading [MathJax]/jax/output/HTML-CSS/jax.js