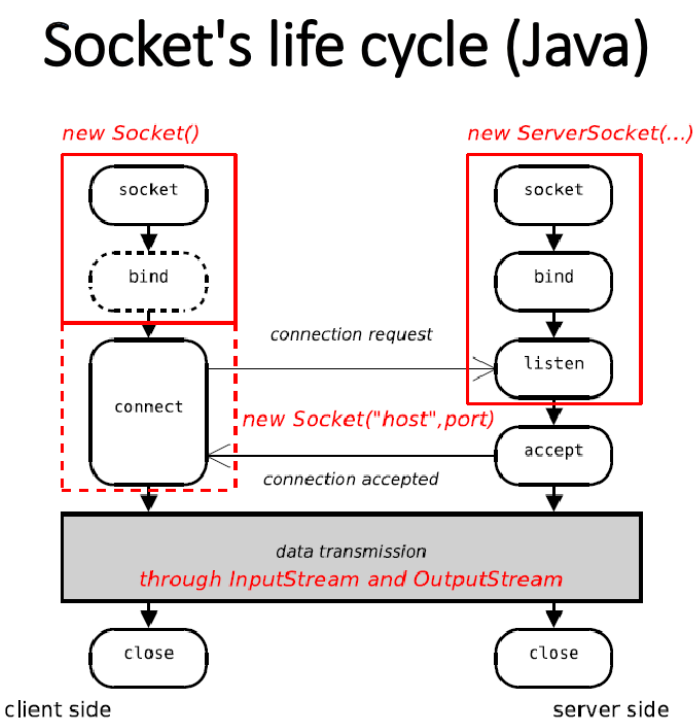
UNIT-7 Sockets for Servers

Sockets are bound to the **port numbers** and when we run any server it just listens on the socket and waits for client requests. For example, **tomcat server** running on **port** 8080 waits for client requests and once it gets any client request, it responds to them.

ServerSocket is a **java.net** class that provides a system-independent implementation of the server side of a client/server socket connection. The constructor for **ServerSocket** throws an exception if it can't listen on the specified port (for example, the port is already being used).

In Java, the basic life cycle of a server program is this:

- A new `ServerSocket` is created on a port using a `ServerSocket()` constructor.
- `ServerSocket` listens on the port using the `accept()` method.
- Uses `getInputStream()` and/or `getOutputStream()`.
- Server and client interact using an agreed-upon communication protocol.
- Either the server or the client (or both) close the connection.
- The server goes back to listening with the `accept()` method.



Implementing your own daytime server is easy. First, create a server socket that listens on port 13:

```
ServerSocket server = new ServerSocket(13);
```

Next, accept a connection:

```
Socket connection = server.accept();
```

When a client does connect, the `accept()` method returns a `Socket` object.

Example:

```
import java.net.*;
import java.io.*;
public class App {
    public static void main(String[] args) throws Exception {
        ServerSocket ss=new ServerSocket(4999);
        System.out.println("Server Port is Listening");

        Socket s=ss.accept();
        System.out.println("Client Connected");
        InputStreamReader in=new InputStreamReader(s.getInputStream());
        BufferedReader bf=new BufferedReader(in);
        String str=bf.readLine();
        System.out.println("Client:"+str);
         s.close();
        ss.close();

    }
}
```

Output: Server Port is Listening

Serving Binary Data

Sending binary, nontext data is not significantly harder. You just use an `Output Stream` that writes a `byte` array rather than a `Writer` that writes a `String`.

`Example:`

```
MyServer.java
import java.io.*;
import java.net.*;
public class MyServer {
    public static void main(String[] args) throws IOException{
```

```
        //create server socket
        ServerSocket serverSocket=new ServerSocket(4444);
        //accept client connection
        Socket socket=serverSocket.accept();

        //Read the data from the socket into a byte array
        InputStream inputStream=socket.getInputStream();
        ByteArrayOutputStream byteArrayOutputStream=new ByteArrayOutputStream();
        byte[] data=new byte[1024];
        int bytesRead;
        while((bytesRead=inputStream.read(data))>0){
            byteArrayOutputStream.write(data, 0, bytesRead);
        }
        //write the data to a file
        FileOutputStream fileOutputStream=new
FileOutputStream("E:\\NetworkProgramming\\WriteFile.txt");
        fileOutputStream.write(byteArrayOutputStream.toByteArray());
        //close the streams and socket
        fileOutputStream.close();
        inputStream.close();
        serverSocket.close();



    }
}
```

MyClient.Java

```
import java.net.*;
import java.io.*;
public class MyClient {
    public static void main(String[] args) throws IOException {
        //connect to server
        Socket socket = new Socket("localhost",4444);
        //read the file into array
        File file = new File("E:\\NetworkProgramming\\ReadFile.txt");
        FileInputStream in = new FileInputStream(file);
        // Get the size of the file
        long length = file.length();
        byte[] data = new byte[(int)length];
        in.read(data);
        //send the data over the socket
        OutputStream out = socket.getOutputStream();
        out.write(data);
       //close the socket and input stream
```

```
        out.close();
        in.close();
        socket.close();
    }

}
```

Simple SocketServer and Client Example (helps to understand Multithreaded Server)

Client.java

```
import java.net.*;
import java.io.*;
public class client {
    public static void main(String[] args) throws IOException {
        Socket s=new Socket("localhost",4999);
        PrintWriter pr=new PrintWriter(s.getOutputStream());
        pr.println("hello");
        pr.flush();
        s.close();

    }
}
```

Server.java

```
import java.net.*;
import java.io.*;
public class server {
```

```
    public static void main(String[] args) throws IOException {
        ServerSocket ss=new ServerSocket(4999);
        Socket s=ss.accept();
        System.out.println("Client Connected");
        InputStreamReader in=new InputStreamReader(s.getInputStream());
        BufferedReader bf=new BufferedReader(in);
        String str=bf.readLine();
        System.out.println("Client:"+str);
        s.close();
        ss.close();


    }
}
//output
```
Client Connected

Clinet:hello

Multithreaded Servers:

In that case there is only one client can communicate with the server. It will not allow simultaneous client connections. Try to start another client. You will see that the second client cannot be connected until the first client closes its connection. To allow **simultaneous** connections we should know multithreaded programming. Here in the following **Multithreaded Socket Programming** , you can connect more than one client connect to the server and communicate.

For each client connection, the server starts a child thread to process the request independent of any other incoming requests.

Socket serverClient=server.accept();

ServerClientThread sct = new ServerClientThread(serverClient,counter);

Example:

Server

```
MultithreadedSocketServer.java
import java.net.*;
import java.io.*;
public class MultithreadedSocketServer{
    public static void main(String[] args) throws Exception {
        try{
```

```
            ServerSocket server=new ServerSocket(8888);
            int counter=0;
            System.out.println("Server Started ....");
            while(true){
              counter++;
              Socket serverClient=server.accept();  //server accept the client
connection request
              System.out.println(" >> " + "Client No:" + counter + " started!");
              ServerClientThread sct = new
ServerClientThread(serverClient,counter); //send  the request to a separate
thread
              sct.start();
            }
          }catch(Exception e){
            System.out.println(e);
          }
    }
}
```

Client

ServerClientThread.java

```
import java.net.*;
import java.io.*;
class ServerClientThread extends Thread {
    Socket serverClient;
    int clientNo;
    int squre;
    ServerClientThread(Socket inSocket,int counter){
      serverClient = inSocket;
      clientNo=counter;
    }
    public void run(){
      try{
        DataInputStream inStream = new
DataInputStream(serverClient.getInputStream());
        DataOutputStream outStream = new
DataOutputStream(serverClient.getOutputStream());
        String clientMessage="", serverMessage="";
        while(!clientMessage.equals("bye")){
          clientMessage=inStream.readUTF();
          System.out.println("From Client-" +clientNo+ ": Number is
:"+clientMessage);
          squre = Integer.parseInt(clientMessage) *
Integer.parseInt(clientMessage);
          serverMessage="From Server to Client-" + clientNo + " Square of " +
clientMessage + " is " +squre;
```

```
          outStream.writeUTF(serverMessage);
          outStream.flush();
        }
        inStream.close();
        outStream.close();
        serverClient.close();
      }catch(Exception ex){
        System.out.println(ex);
      }finally{
        System.out.println("Client -" + clientNo + " exit!! ");
      }
    }
  }
```

Closing Server Sockets

- If you're finished with a server socket, you should close it.
- This frees up the port for other programs that may wish to use it.
- Closing a **ServerSocket** frees a port on the local host, allowing another server to bind to the port;
- The **close()** method of **ServerSocket** class is used to close this socket.

```
public static void main(String[] args) throws IOException {
        ServerSocket ss=new ServerSocket(4999);
        Socket s=ss.accept();
/ write the log entry first in case the client disconnects
auditLogger.info(now + " " + connection.getRemoteSocketAddress());
        System.out.println("Client Connected");
        InputStreamReader in=new InputStreamReader(s.getInputStream());
        BufferedReader bf=new BufferedReader(in);
        String str=bf.readLine();
        System.out.println("Client:"+str);
        s.close();
        ss.close();


    }
```

Logging:

- Java Logging API, found in the **java.util.logging** package.
- Long periods of time are spent with servers running without supervision.
- It's important to debug what happened when in a server long after the fact.
- Debugging is done on a server for a long period of time

What to Log:

There are two primary things you want to store in your logs:

• Requests Log

• Server errors Log

The **error log** contains mostly unexpected exceptions that occurred while the server was running.

The **error log** does not contain client errors.

The error log is completely for unexpected exceptions.

These go into the **request log**.

Logging **SocketException** error is Server error.

Server Errors are: Closed socket connection, Slow network (timeout), Network firewall , Idle connection, Errors in code

How to Log:

Java 1.3 and earlier still use third-party logging libraries such as **log4j** or **Apache Commons Logging**

The `java.util.logging` package available since Java 1.4

```
private final static Logger auditLogger = Logger.getLogger("requests");
```

There are seven levels defined as named constants in `java.util.logging.Level` in descending order of seriousness:

- Level.SEVERE (highest value) SEVERE is a message level indicating a serious failure.
- Level.WARNING(WARNING is a message level indicating a potential problem)
- Level.INFO(**INFO** is a message **level** for informational messages)
- Level.CONFIG(CONFIG is a message level for static configuration messages.)
- Level.FINE
- Level.FINER

- Level.FINEST (lowest value)

  FINE is a message level providing tracing information. **FINER** indicates a fairly detailed tracing message. **FINEST** indicates a highly detailed tracing message.

```
import java.io.*;
import java.net.*;
import java.sql.Date;
import java.util.logging.*;
public class MyServer {
    private final static Logger auditLogger = Logger.getLogger("requests");
    private final static Logger errorLogger = Logger.getLogger("errors");
    public static void main(String[] args) {
        try{
        ServerSocket ss=new ServerSocket(4999);
        Socket s=ss.accept();
        // write the log entry first in case the client disconnects
        Date now = new Date(0);
        auditLogger.info(now+""+s.getRemoteSocketAddress());
        System.out.println("Client Connected");
        InputStreamReader in=new InputStreamReader(s.getInputStream());
        BufferedReader bf=new BufferedReader(in);
        String str=bf.readLine();
        System.out.println("Client:"+str);
        s.close();
        ss.close();

        } catch (IOException ex) {
        errorLogger.log(Level.SEVERE, "error", ex.getMessage());
        } catch (RuntimeException ex) {
        errorLogger.log(Level.SEVERE, "unexpected error " + ex.getMessage(), ex);
        }

    }

}
```

Constructing Server Sockets:

There are four public `ServerSocket` constructors:

```
public ServerSocket(int port) throws BindException, IOException
public ServerSocket(int port, int queueLength)throws BindException, IOException
public ServerSocket(int port, int queueLength, InetAddress bindAddress) throws
IOException
public ServerSocket() throws IOException
```

These constructors specify the **port**, the **length of the queue** used to hold incoming connection requests, and the **local network interface to bind to**.
For example, to create a server socket that would be used by an HTTP server on port 80, you would write:

```
1. ServerSocket httpd = new ServerSocket(80);
```

To create a server socket that would be used by an HTTP server on port 80 and queues up to 50 connections at a time:

```
2. ServerSocket httpd = new ServerSocket(80, 50);

3. InetAddress local = InetAddress.getByName("192.168.210.122");
ServerSocket httpd = new ServerSocket(5776, 10, local);
```

Example for looking local ports0- 1024

```
import java.io.*;
import java.net.*;

public class MyServer {

    public static void main(String[] args) {
        for (int port = 1; port <= 1024; port++) {
            try {
            ServerSocket server = new ServerSocket(port);
            } catch (IOException ex) {
            System.out.println("There is a server on port " + port + ".");
            }
            }

    }

}
Output:
There is a server on port 80.
There is a server on port 135.
There is a server on port 139.
There is a server on port 445.
```

Constructing Without Binding

The **noargs** constructor creates a **ServerSocket** object but does not actually bind it to a port, so it cannot initially accept any connections. It can be bound later using the **bind()** methods:

```
public void bind(SocketAddress endpoint) throws IOException
public void bind(SocketAddress endpoint, int queueLength) throws IOException
```

Example:

```
ServerSocket ss = new ServerSocket();
// set socket options...
SocketAddress http = new InetSocketAddress(80);
ss.bind(http);
```

Getting Information About a Server Socket:

The **ServerSocket** class provides two getter methods that tell you the **local address** and **port** occupied by the server socket.

These are useful if you've opened a server socket on an **anonymous port** and/or an **unspecified network interface**.

Two Getter Methods

1. `public InetAddress getInetAddress();`

   This method tells you **which remote host the socket is connected to** or, if the connection is now closed, *Which host the socket was connected to when it was connected.*

   Example:

```
ServerSocket httpd = new ServerSocket(8000);
InetAddress ia = httpd.getInetAddress();
```

2. `public int getLocalPort()`

   The **ServerSocket** constructors allow you to listen on an unspecified port by passing 0 for the port number.

   Example:

```
public static void main(String[] args) throws Exception {
  try {
      ServerSocket server = new ServerSocket(0);
      System.out.println("This server runs on port "
      + server.getLocalPort());
      } catch (IOException ex) {
      System.err.println(ex);
            }
       }
```

```
Output: This server runs on port 58936
```

Socket Options:

SO_TIMEOUT

The only socket option supported for server socket is ***SO_TIMEOUT***. ***SO_TIMEOUT*** is the amount of time, in milliseconds, that ***accept()*** waits for an incoming connection before throwing a **java.io.InterruptedIOException**. If **SO_TIMEOUT** is 0, then ***accept()*** will never time out. The default is to never time out.

If you want to change this, the **setSoTimeout()** method sets the **SO_TIMEOUT** field for this server socket object:

```
        public void setSoTimeout(int timeout) throws SocketException

        public int getSoTimeout() throws IOException
```

Example:

```
import java.io.IOException;
import java.net.*;
public class App {
    public static void main(String[] args) throws Exception {
        try (ServerSocket server = new ServerSocket(8000)) {
            server.setSoTimeout(30000); // block for no more than 30 seconds
            try {
            Socket s = server.accept();
            // handle the connection
            // ...
            } catch (SocketTimeoutException ex) {
            System.err.println("No connection within 30 seconds");
            }
            } catch (IOException ex) {
            System.err.println("Unexpected IOException: " + ex);
            }
    }
}
```

After 30 sec

Output: No connection within 30 seconds

SO_REUSEADDR

The SO_REUSEADDR option for server sockets is very similar to the same option for client sockets.

It determines whether a new socket will be allowed to bind to a previously used port while there might still be data passing through the network addressed to the old socket.

There are **two** methods to get and set this option:

```
public boolean getReuseAddress() throws SocketException
public void setReuseAddress(boolean on) throws SocketException
```

Example:

```
ServerSocket ss = new ServerSocket(10240);
System.out.println("Reusable: " + ss.getReuseAddress());
```

SO_RCVBUF

The SO_RCVBUF option sets the default receive buffer size for client sockets accepted by the server socket. It's read and written by these two methods:

```
public int getReceiveBufferSize() throws SocketException
public void setReceiveBufferSize(int size) throws SocketException
```

Setting SO_RCVBUF on a server socket is like calling setReceiveBufferSize() on each individual socket returned by accept() (except that you can't change the receive buffer size after the socket has been accepted)

Example:

```
ServerSocket ss = new ServerSocket();
int receiveBufferSize = ss.getReceiveBufferSize();
if (receiveBufferSize < 131072) {
ss.setReceiveBufferSize(131072);
}
ss.bind(new InetSocketAddress(8000));
//...
```

Class of Service:

Different types of Internet services have different performance needs. For instance, live streaming video of sports needs relatively high bandwidth. On the other hand, a movie might still need high bandwidth but be able to tolerate more delay and latency. Email can be passed over low-bandwidth connections

Four general traffic classes are defined for TCP:

• Low cost

• High reliability

• Maximum throughput

• Minimum delay

The **setPerformancePreferences()** method expresses the relative preferences given to connection time, latency, and bandwidth for sockets accepted on this server:

```
public void setPerformancePreferences(int connectionTime, int latency, int bandwidth)
```

Parameters:

connectionTime - An int expressing the relative importance of a short connection time

latency - An int expressing the relative importance of low latency*(process a very high volume of data messages with minimal delay)*

bandwidth - An int expressing the relative importance of high bandwidth

For example, then it could invoke this method with the values (1, 0, 0). If the application prefers high bandwidth above low latency, and low latency above short connection time, then it could invoke this method with the values (0, 1, 2).