Unit 7:

Socket for server

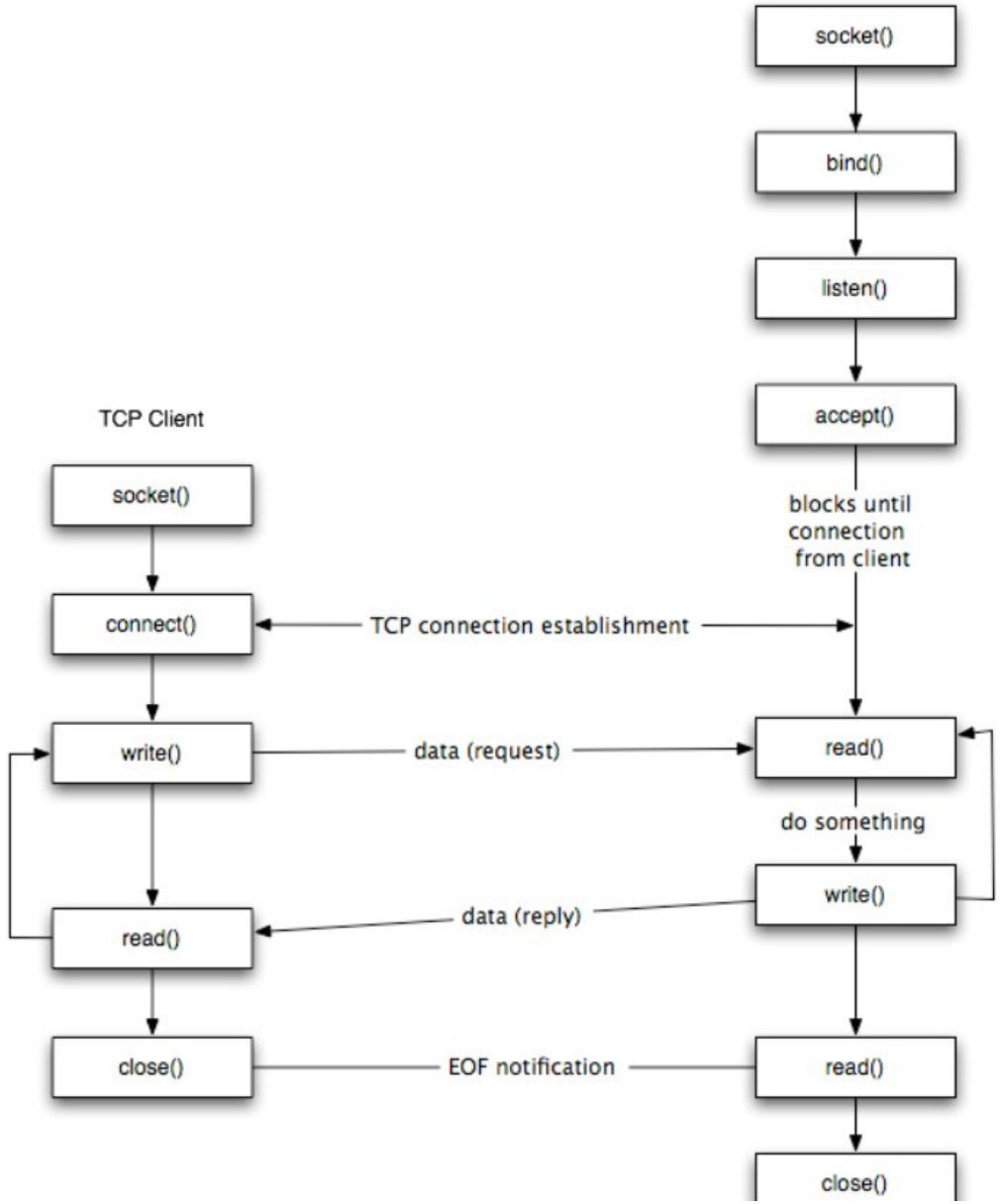
By

Khushbu Kumari Sarraf

EMBA,BE

Server Socket

- A server socket is a software endpoint that allows a server program to accept incoming network connections from client programs. In other words, it's a mechanism that enables a server application to listen for and accept connections from multiple clients.
- A server socket is created by the server program and is bound to a specific network address and port number. Once a server socket is created and bound, it enters a listening state and begins accepting incoming connection requests from clients.
- When a client program wants to connect to the server, it sends a request to the server socket. The server socket accepts the connection request, creates a new socket for the client, and establishes a two-way communication channel between the server and client sockets. The server socket can continue to listen for new incoming connections while simultaneously communicating with connected clients.


socket()
bind()
listen()
accept()
blocks until connection from client
TCP Client
socket()
connect()
TCP connection establishment
write()
data (request)
read()
do something
write()
read()
data (reply)
close()
EOF notification
read()
close()

Server Socket

- The `ServerSocket` class is used to implement a server program. Here are the typical steps involve in developing a server program:
  - 1. Create a server socket and bind it to a specific port number
  - 2. Listen for a connection from the client and accept it. This results in a client socket is created for the connection.
  - 3. Read data from the client via an `InputStream` obtained from the client socket.
  - 4. Send data to the client via the client socket's `OutputStream`.
  - 5. Close the connection with the client.
- The steps 3 and 4 can be repeated many times depending on the protocol agreed between the server and the client.
- The steps 1 to 5 can be repeated for each new client. And each new connection should be handled by a separate thread.

Server Socket Constructor

- **Create a Server Socket:**

Create a new object of the ServerSocket class by using one of the following constructors. There are three public ServerSocket constructors

- public ServerSocket(int port)throws IOException, BindException
- public ServerSocket(int port, int queueLength) throws IOException, BindException
- public ServerSocket(int port, int queueLength, InetAddress bindAddr) throws IOException

Server Socket Constructor

- public ServerSocket(int port)throws IOException, BindException- This constructors creates a server socket on the port specified by the argument. If you pass for the port number, the system selects an available port for you. The constructor throws an IOException (specially, a BindException) if the socket cannot be created and bound to the requested port.

```
try{
    ServerSocket httpd = new ServerSocket(80);
}
catch{(IOException e){
 System.err.println(e);
}
```

Server Socket Constructor

- public ServerSocket(int port, int queueLength) throws IOException, BindException- In Java, the ServerSocket class provides a constructor that takes two integer parameters: port and queueLength.

•**port:** This parameter specifies the port number on which the ServerSocket will listen for incoming connections. Port numbers range from 0 to 65535, but only certain port numbers are reserved for specific services. For example, port 80 is typically used for HTTP traffic.

•**queueLength:** This parameter specifies the maximum length of the queue of incoming connections. When multiple clients try to connect to the same ServerSocket simultaneously, some of them may be put in a waiting queue until the server can process their requests. The queue length parameter specifies the maximum number of connections that can be queued up in this way. If the queue is full, new connection requests will be refused.

Server Socket Constructor

```
import java.io.*;
import java.net.*;
public class MyServer {
 public static void main(String[] args) throws IOException {
 int port = 1234;
int queueLength = 10;
ServerSocket serverSocket = new ServerSocket(port, queueLength);
while (true) {
Socket clientSocket = serverSocket.accept();
// Handle the client connection... } } }
```

Server Socket Constructor

ServerSocket(int port, int queueLength, InetAddress bindAddr): The ServerSocket class in Java also provides a constructor that takes three parameters: port, queueLength, and bindAddr. This constructor allows you to specify the local network interface that the ServerSocket should bind to. Here's an explanation of what each parameter does:

•port: This parameter specifies the port number on which the ServerSocket will listen for incoming connections. Port numbers range from 0 to 65535.

•queueLength: This parameter specifies the maximum length of the queue of incoming connections.

•bindAddr: This parameter specifies the local network interface that the ServerSocket should bind to. It is an instance of the InetAddress class, which represents an IP address. If bindAddr is null, the ServerSocket will bind to the wildcard address, which means it will listen on all network interfaces

Server Socket Constructor

```
import java.io.*;
 import java.net.*;
public class MyServer {
 public static void main(String[] args) throws IOException {
 int port = 1234;
int queueLength = 10;
InetAddress bindAddr = InetAddress.getByName("192.168.0.100");
ServerSocket serverSocket = new ServerSocket(port, queueLength, bindAddr);
 while (true) {
Socket clientSocket = serverSocket.accept();
// Handle the client connection... } } }
```

Listen for a connection:

Once a `ServerSocket` instance is created, call `accept()` to start listening for incoming client requests:

```
Socket socket = serverSocket.accept();
```

Note that the `accept()` method blocks the current thread until a connection is made. And the connection is represented by the returned `Socket` object.

Read data from the client

Once a Socket object is returned, you can use its InputStream to read data sent from the client like this:

```
InputStream input = socket.getInputStream();
```

The InputStream allows you to read data at low level: read to a byte array. So if you want to read the data at higher level, wrap it in an InputStreamReader to read data as characters.

```
InputStreamReader reader = new InputStreamReader(input);
int character = reader.read();  // reads a single character
```

You can also wrap the InputStream in a BufferedReader to read data as String, for more convenient:
BufferedReader reader = new BufferedReader(new InputStreamReader(input));

```
String line = reader.readLine();    // reads a line of text
```

Send data to the client:

Use the OutputStream associated with the Socket to send data to the client, for example:

```
OutputStream output = socket.getOutputStream();
```

As the OutputStream provides only low-level methods (writing data as a byte array), you can wrap it in a PrintWriter to send data in text format, for example:

```
PrintWriter writer = new PrintWriter(output, true);
writer.println("This is a message sent to the server");
```

The argument true indicates that the writer flushes the data after each method call (auto flush).

Close the client connection:

Closing a ServerSocket frees a port on the local host, allowing another server to bind to the port; it also breaks all currently open sockets that the ServerSocket has accepted. Server sockets are closed automatically when a program dies, so it's not absolutely necessary to close them in programs that terminate shortly after the ServerSocket is no longer needed. Invoke the `close()` method on the client `Socket` to terminate the connection with the client:

```
socket.close();
```

This method also closes the socket's `InputStream` and `OutputStream`, and it can throw `IOException` if an I/O error occurs when closing the socket.

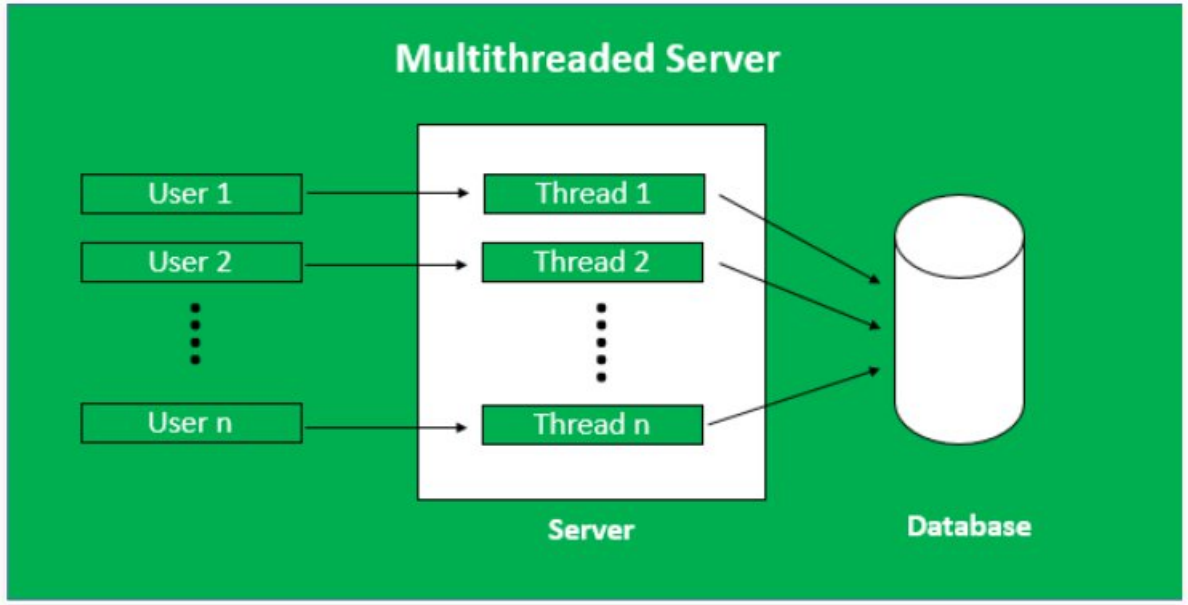
Using ServerSockets

The ServerSocket class contains everything needed to write servers in Java. It has constructors that create new ServerSocket objects, methods that listen for connections on a specified port, methods that configure the various server socket options, and the usual miscellaneous methods such as toString().In Java, the basic life cycle of a server program is this:

1. A new ServerSocket is created on a particular port using a ServerSocket() constructor.
2. The ServerSocket listens for incoming connection attempts on that port using its accept() method. accept() blocks until a client attempts to make a connection, at which point accept() returns a Socket object connecting the client and the server.
3. Depending on the type of server, either the Socket's getInputStream() method, getOutputStream() method, or both are called to get input and output streams that communicate with the client.
4. The server and the client interact according to an agreed-upon protocol until it is time to close the connection.
5. The server, the client, or both close the connection.
6. The server returns to step 2 and waits for the next connection.

Multithreaded Servers

A server having more than one thread is known as Multithreaded Server. When a client sends the request, a thread is generated through which a user can communicate with the server. We need to generate multiple threads to accept multiple requests from multiple clients at the same time.



Multithreaded Servers

A multi-threaded server is a server that uses multiple threads to handle client requests concurrently. In a multithreaded server architecture, each client request is handled by a separate thread, which allows multiple requests to be processed at the same time. This can improve the server's performance and responsiveness, especially in environments where many clients are simultaneously accessing the server.

- When a client connects to a multithreaded server, the server creates a new thread to handle the client's request. This allows the server to continue accepting new client connections without waiting for the current client request to be completed. The thread that handles the client request typically performs the necessary processing and returns the response to the client before terminating.
- One of the benefits of using a multithreaded server is that it can take advantage of modern multi-core processors, which can execute multiple threads simultaneously. This allows the server to handle multiple clients requests concurrently and can lead to significant performance improvements over a single-threaded server.

Advantages of Multithreaded Server

- **Quick and Efficient:** Multithreaded server could respond efficiently and quickly to the increasing client queries quickly.
- **Waiting time for users decreases:** In a single-threaded server, other users had to wait until the running process gets completed but in multithreaded servers, all users can get a response at a single time so no user has to wait for other processes to finish.
- **Threads are independent of each other:** There is no relation between any two threads. When a client is connected a new thread is generated every time.
- **The issue in one thread does not affect other threads:** If any error occurs in any of the threads then no other thread is disturbed, all other processes keep running normally. In a single-threaded server, every other client had to wait if any problem occurs in the thread.

Disdvantages of Multithreaded Server

- **Complicated Code:** It is difficult to write the code of the multithreaded server. These programs can not be created easily
- **Debugging is difficult:** Analyzing the main reason and origin of the error is difficult.

Example 9-3. A multithreaded daytime server

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

public class MultithreadedDaytimeServer {

  public final static int PORT = 13;

  public static void main(String[] args) {
   try (ServerSocket server = new ServerSocket(PORT)) {
     while (true) {
       try {
         Socket connection = server.accept();
         Thread task = new DaytimeThread(connection);
         task.start();
       } catch (IOException ex) {}
     }
    } catch (IOException ex) {
      System.err.println("Couldn't start server");
    }
  }

  private static class DaytimeThread extends Thread {
```

```
    private Socket connection;

    DaytimeThread(Socket connection) {
      this.connection = connection;
    }

    @Override
    public void run() {
      try {
        Writer out = new OutputStreamWriter(connection.getOutputStream());
        Date now = new Date();
        out.write(now.toString() +"\r\n");
        out.flush();
      } catch (IOException ex) {
        System.err.println(ex);
      } finally {
        try {
          connection.close();
        } catch (IOException e) {
          // ignore;
        }
      }
    }
  }
}
```

Logging

Logging is an important tool for debugging and monitoring network programming applications. In network programming, logging can be used to track the flow of data between network components, monitor network connections and errors, and provide insight into the performance of the application. Logging is an important feature that helps developers to trace errors. It provides a logging API that was introduced in Java 1.4 version. It provides the ability to capture the log file. When an application generates the logging call, the Logger records the event in the LogRecord. After that, it sends it to the corresponding handlers or appenders. Before sending it to the console or file, the appenders format that logs record by using formatted or layouts.

Importance of logging

- Tracing information of the application
- Recording unusual circumstances or errors that may occur in the program.
- Getting the info about whats going in the application.

What to log

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

- Requests
- Server errors

The server often keeps two different logfiles for these two different items.

The log usually contains one entry (requests) for each connection made to the server. Servers that perform multiple operations per connection may have one entry per operation instead.

The error log contains mostly unexpected exceptions that occurred while the server was running. The error doesn’t contain client error. Client error goes into the request log.

How to log(Logging Levels)

- The log levels control the logging details. Each level is associated with a numeric value. In order to log messages in network programming, you can use a logging library or framework. These libraries provide a set of logging levels that can be used to categorize the severity of the messages being logged. The most common logging levels are:

Level	Value	Used For
SEVERE	1000	Indicates some serious failure
WARNING	900	Potential Problem
INFO	800	General Info
CONFIG	700	Configuration Info
FINE	500	General developer info
FINER	400	Detailed developer info
FINEST	300	Specialized developer Info

Logging Level Detail

- Note : Read from book (page -188)

Getting Information about 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. This would be the case,

for one example, in the data connection of an FTP session:

```
public InetAddress getInetAddress()
```

This method returns the address being used by the server (the local host). If the local host has a single IP address (as most do), this is the address returned by InetAddress.getLocalHost(). If the local host has more than one IP address, the specific address returned is one of the host's IP addresses. You can't predict which address you will get. For example:

```
ServerSocket httpd = new ServerSocket(80);
InetAddress ia = httpd.getInetAddress();
```

If the ServerSocket has not yet bound to a network interface, this method returns null:

```
public int getLocalPort()
```

Socket Option

Socket options specify how the native sockets on which the ServerSocket class relies send and receive data. For server sockets, Java supports three options:

- SO_TIMEOUT
- SO_REUSEADDR
- SO_RCVBUF

It also allows you to set performance preferences for the socket’s packets.

Socket Option

• 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, 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

The countdown starts when accept() is invoked. When the timeout expires, accept() throws a SocketTimeoutException, a subclass of IOException. You need to set this option before calling accept(); you cannot change the timeout value while accept() is waiting for a connection. The timeout argument must be greater than or equalto zero; if it isn't, the method throws an IllegalArgumentException. The getSoTimeout() method returns this server socket's current SO_TIMEOUT value.

Socket Option

• SO_REUSEADDR-The SO_REUSEADDR option for server sockets is very similar to the same option for

client sockets, discussed in the previous chapter. It determines whether a new socket will be allowed to bind to a previously used port while there might still be data traversing the network addressed to the old socket. As you probably expect, there are two methods to get and set this option:

```
public boolean getReuseAddress() throws SocketException
```

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

The default value is platform dependent. This code fragment determines the default value by creating a new ServerSocket and then calling getReuseAddress():

Socket Option

• 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).

Http Servers

HTTP is a large protocol. A full-featured HTTP server must respond to requests for files, convert URLs into filenames on the local system, respond to POST and GET requests, handle requests for files that don’t exist, interpret MIME types and much, much more. However, many HTTP servers don’t need all of these features. For example, many sites simply display an “under construction” message.

Single file Servers

- A Single-File Server Our investigation of HTTP servers begins with a server that always sends out the same file, no matter what the request. It’s called Single File HTTP Server. The filename, local port, and content encoding are read from the command line. If the port is omitted, port 80 is assumed. If the encoding is omitted, ASCII is assumed.
- The SingleFileHTTPServer class holds the content to send, the header to send, and the port to bind to. The start() method creates a ServerSocket on the specified port, then enters an infinite loop that continually accepts connections and processes them.

Full-Fledged Servers

Enough special-purpose HTTP servers. This next section develops a full-blown HTTP server, called JHTTP, that can serve an entire document tree, including images, applets, HTML files, text files, and more. It will be very similar to the SingleFileHTTPServer, except that it pays attention to the GET requests. This server is still fairly lightweight because this server may have to read and serve large files from the filesystem over potentially slow network connections. Rather than processing each request as it arrives in the main thread of execution, you'll place incoming connections in a pool.