UNIT-7

Unit 7: Socket for Servers **5 Hrs.**

7.1 Using ServerSockets: Serving Binary Data, Multithreaded Servers, Writing to Servers with Sockets and Closing Server Sockets
7.2 Logging: What to Log and How to Log
7.3 Constructing Server Sockets: Constructing Without Binding
7.4 Getting Information about Server Socket
7.5 Socket Options: SO_TIMEOUT, SO_RSUMEADDR, SO_RCVBUF and Class of Service
7.6 HTTP Servers: A Single File Server, A Redirector and A Full-Fledged HTTP Server

SERVER SOCKETS

- ServerSockets is a Java class that represents a server socket, which is a mechanism for listening to incoming network requests on a specified port number and IP address. It is part of the Java standard library and is commonly used in network programming.

The most commonly used parameters are:

- Port number: The port number on which the server socket listens for incoming requests. This is an integer value between 0 and 65535.
- Backlog: The maximum number of pending connections that can be queued up before the server starts rejecting new connections. This is an integer value and the default value is usually 50.
- InetAddress: The IP address on which the server socket listens for incoming requests. This can be set to null to listen on all available network interfaces or to a specific IP address.
- Timeout: The maximum amount of time that the server socket will wait for a client to connect before timing out. This is an integer value in milliseconds and the default value is usually 0, which means that the socket will wait indefinitely.
- Receive buffer size: The size of the buffer used to receive data from the client. This is an integer value and the default value is usually 8192 bytes.

SERVER SOCKETS

```
import java.net.*;
public class MyServer {
    public static void main(String[] args) throws Exception {
        // create a server socket on port 12345
        ServerSocket serverSocket = new ServerSocket(12345);
        // set the backlog to 100 pending connections
        serverSocket.setBacklog(100);
        // set the timeout to 10 seconds
        serverSocket.setSoTimeout(10000);
        // accept incoming connections
        while (true) {
            Socket clientSocket = serverSocket.accept();
            // handle the client socket
        }
    }
}
```

CONSTRUCTING SERVER SOCKETS

1. public ServerSocket(int port) throws BindException, IOException: This constructor creates a new server socket that listens on the specified port number. The queue length for incoming connections is set to the default value.
2. public ServerSocket(int port, int queueLength) throws BindException, IOException: This constructor creates a new server socket that listens on the specified port number, and sets the queue length for incoming connections to the specified value.
3. public ServerSocket(int port, int queueLength, InetAddress bindAddress) throws IOException: This constructor creates a new server socket that listens on the specified port number, and sets the queue length for incoming connections to the specified value. It also binds the server socket to the specified network interface address.
4. public ServerSocket() throws IOException: This constructor creates a new server socket with an unspecified port number. The server socket must be bound to a specific port and address using the bind() method before it can be used.

CONSTRUCTING SERVER SOCKETS WITHOUT BINDING

```java
import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.ServerSocket;
public class ConsWithOutBind {
    Run | Debug
    public static void main(String[] args) {
        try {
            // create a server socket without binding it to an address or port
            ServerSocket serverSocket = new ServerSocket();
            // bind the server socket to a specific address and port with the default queue length
            InetSocketAddress endpoint1 = new InetSocketAddress(hostname:"localhost", port:8080);
            serverSocket.bind(endpoint1);
            System.out.println("Server socket bound to " + endpoint1);
            // bind the server socket to another address and port with a queue length of 50
            InetSocketAddress endpoint2 = new InetSocketAddress(hostname:"192.168.0.10", port:9090);
            serverSocket.bind(endpoint2, backlog:50);
            System.out.println("Server socket bound to " + endpoint2 + " with queue length 50");
            // handle incoming connections...

            // close the server socket when done
```

Steps of Server sockets

1. **Create a ServerSocket object**: This is done using the ServerSocket class, which represents a server socket that listens for incoming connections from clients. You can specify the port number and other settings when creating the ServerSocket object.
2. **Listen for incoming connections**: Once you have created the ServerSocket object, you can use its accept() method to listen for incoming connections from clients. This method blocks until a client connects to the server.
3. **Accept a client connection**: When a client connects to the server, the accept() method returns a Socket object that represents the client connection. You can use this object to communicate with the client.
4. **Handle the client connection**: Once you have accepted the client connection, you can handle the client request. This involves reading data from the client using the InputStream of the Socket object, processing the request, and sending a response to the client using the OutputStream of the Socket object.
5. **Close the client connection**: After you have finished processing the client request, you should close the client connection by calling the close() method on the Socket object.
6. **Repeat steps 2-5 for additional client connections**: If the server is designed to handle multiple client connections simultaneously, you can repeat steps 2-5 for each new client connection.
7. **Close the ServerSocket object**: When the server program is finished listening for incoming connections, you should close the ServerSocket object by calling its close() method.

Server program Read From client and write to server

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

public class Server {
    Run | Debug
    public static void main(String[] args) throws IOException {
        int port = 8080;
        try (ServerSocket serverSocket = new ServerSocket(port)) {
            System.out.println("Server started on port " + port);
            while (true) {
                Socket clientSocket = serverSocket.accept();
                System.out.println("New client connected: " + clientSocket.getInetAddress().getHostAddress());
                DataInputStream in = new DataInputStream(clientSocket.getInputStream());
                String message = in.readUTF();
                System.out.println("Received message from client: " + message);
                DataOutputStream out = new DataOutputStream(clientSocket.getOutputStream());
                out.writeUTF("Server received message: " + message);
                clientSocket.close();
            }
        }
    }
}
```

SERVING BINARY DATA

- In network programming, serving binary data involves sending data in its raw binary format over the network

```java
import java.io.*;
import java.net.*;
import java.nio.ByteBuffer;
import java.util.Base64;

public class BinaryServer {
    Run | Debug
    public static void main(String[] args) throws IOException {
        int port = Integer.parseInt(args[0]);

        ServerSocket serverSocket = new ServerSocket(port);
        System.out.println("Server started on port " + port);

        while (true) {
            Socket clnt = serverSocket.accept();
            System.out.println("Client connected: " + clnt);
            long currentTime = System.currentTimeMillis();
            ByteBuffer buffer = ByteBuffer.allocate(Long.BYTES);
            buffer.putLong(currentTime);
            byte[] timeBytes = buffer.array();
            String encodedTime = Base64.getEncoder().encodeToString(timeBytes);
            DataOutputStream out = new DataOutputStream(clnt.getOutputStream());
            out.writeUTF(encodedTime);
            clnt.close();
        }
    }
}
```

```
AAABh68DLqk=

Connection to host lost.

Press any key to continue...
```

A MULTITHREADED SERVER

- A multithreaded server is a type of server that **uses multiple threads of execution** to handle **multiple client connections concurrently**. When a new client connection is established, the server **creates a new thread to handle that connection**, allowing multiple clients to be served simultaneously.
- The main characteristics of a multithreaded server include:
- **Scalability**: Multithreaded servers can handle multiple client connections concurrently, allowing them to scale up to handle a large number of clients.
- **Responsiveness**: By using multiple threads, a multithreaded server can be more **responsive to client requests**, as each thread can handle a **single request without blocking other clients**.
- **Modularity**: Multithreaded servers are often modular in design, with each thread **handling a specific client connection**. This can make the server **easier to maintain and debug**.

Advantages

- Improved performance
- Increased reliability
- Better resource utilization
- Enhanced user experience
- Support for diverse clients

A MULTITHREADED PROGRAM

```
class RunThread implements Runnable {
    public void run() {
        System.out.println(x:"I am at Runnable at run function:");
    }
}
public class RunableThread {
    Run | Debug
    public static void main(String[] args) {
        RunThread rt = new RunThread();
        Thread t1 = new Thread(rt);
        t1.start();
    }
}
```

A MULTITHREADED SERVER

```java
import java.io.*;
import java.net.*;
import java.util.*;
public class ThreadServer {
    Run | Debug
    public static void main(String[] args) throws IOException {
        int port = Integer.parseInt(args[0]);
        try (ServerSocket serverSocket = new ServerSocket(port)) {
            System.out.println(x:"Daytime server started.");
            while (true) {
                Socket clientSocket = serverSocket.accept();
                System.out.println("Accepted connection from " + clientSocket.getInetAddress());
                // Create a new thread to handle the client connection
                DaytimeThread dt= new DaytimeThread(clientSocket);
                Thread thread = new Thread(dt);
                thread.start();
            }
        }
    }
}
```

```java
class DaytimeThread implements Runnable {
    private final Socket clientSocket;
    public DaytimeThread(Socket clientSocket) {
        this.clientSocket = clientSocket;
    }
    public void run() {
        try {
            DataOutputStream out = new DataOutputStream(clientSocket.getOutputStream());
            String dateTime = new Date().toString();
            out.writeUTF(dateTime);
            out.close();
            clientSocket.close();
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

CLOSING SERVER SOCKETS

- If you're finished with a server socket, you should close it, especially if the program is going to continue to run for some time.
- This frees up the port for other programs that may wish to use it. Closing a ServerSocket should not be confused with closing a Socket.
- 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.

```
ServerSocket server = null;
try {
  server = new ServerSocket(port);
  // ... work with the server socket
} finally {
  if (server != null) {
    try {
      server.close();
    } catch (IOException ex) {
      // ignore
    }
```

WRITING TO SERVERS WITH SOCKETS

- In the examples so far, the server has only written to client sockets. It hasn't read from them. Most protocols, however, require the server to do both. This isn't hard.
- You'll accept a connection as before, but this time ask for both an InputStream and an OutputStream. Read from the client using the InputStream and write to it using the OutputStream.
- The main trick is understanding the protocol: when to write and when to read.

WRITING TO SERVERS WITH SOCKETS ECHOCLIENT

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

public class EchoClient {
    Run | Debug
    public static void main(String[] args) throws IOException {
        Socket socket = new Socket(host:"localhost", port:8000);
        System.out.println(x:"Connected to echo server.");
        DataInputStream in = new DataInputStream(socket.getInputStream());
        DataOutputStream out = new DataOutputStream(socket.getOutputStream());
        Scanner sc = new Scanner(System.in);
        while (true) {
            System.out.print(s:"Enter text: ");
            String message = sc.nextLine();
            out.writeUTF(message);
            out.flush();
            String response = in.readUTF();
            System.out.println("Echo from server: " + response);
        }

    }
}
```

WRITING TO SERVERS WITH SOCKETS ECHOSERVER

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

public class EchoServer {
    Run | Debug
    public static void main(String[] args) throws IOException {
        try (
            ServerSocket serverSocket = new ServerSocket(port:8000)) {
            System.out.println(x:"Echo server started.");
            while (true) {
                Socket clientSocket = serverSocket.accept();
                System.out.println("Accepted connection from " + clientSocket.getInetAddress());
                Thread thread = new Thread(new EchoThread(clientSocket));
                thread.start();
            }
        }
    }
}
```

```java
class EchoThread implements Runnable {
    private final Socket clientSocket;

    public EchoThread(Socket clientSocket) {
        this.clientSocket = clientSocket;
    }
    public void run() {
        try {
            DataInputStream in = new DataInputStream(clientSocket.getInputStream());
            DataOutputStream out = new DataOutputStream(clientSocket.getOutputStream());
            while (true) {
                String message = in.readUTF();
                System.out.println("Received message from " + clientSocket.getInetAddress() + ": " + message);
                out.writeUTF(message);
                out.flush();
            }
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

LOGGING IN(HOW TO AND WHEN TO)

- Logging in Java is the process of **recording messages** that provide information **about the execution of a program**. It is an important tool for developers to diagnose and fix issues in their code, and for operations teams to monitor and **troubleshoot running applications**.
- Java provides a **built-in logging framework** called **java.util.logging**, which allows developers **to create loggers, handlers, and formatters to manage and format log messages**.
- Loggers are used to **write messages** to **logs**, **handlers are used to route log messages** to different destinations (such as the console or a file), and formatters are used to **format log messages** in a particular way (such as adding a timestamp or a logging level).
- Logging in Java works by specifying **a logging level for each message**. The logging levels, in order of increasing severity, are **SEVERE, WARNING, INFO, CONFIG, FINE, FINER, and FINEST.**
- A message with a particular logging level will be recorded only if the logger's level is set to that level or higher. For example, if the logger's level is set to INFO, only messages with a logging level of INFO, WARNING, or SEVERE will be recorded.
- There are **seven levels** defined as named constants in java.util.logging. Level in descending order of seriousness:
- (Level.SEVERE (highest value), Level.WARNING, Level.INFO, Level.CONFIG, Level.FINE, Level.FINER Level.FINEST (lowest value)

LOGGING IN(HOW TO AND WHEN TO)

```java
import java.io.*;
import java.net.*;
import java.util.Date;
import java.util.logging.*;
public class LogingProgram1 {
    private static final Logger logger = Logger.getLogger(LogingProgram1.class.getName());
    Run | Debug
    public static void main(String[] args) {
        logger.info(msg:"Starting daytime server...");
        try (ServerSocket serverSocket = new ServerSocket(port:13)) {
            while (true) {
                try (Socket clientSocket = serverSocket.accept()) {
                    logger.info("Accepted client connection: " + clientSocket.getInetAddress());
                    Date now = new Date();
                    String response = now.toString() + "\r\n";
                    OutputStream outputStream = clientSocket.getOutputStream();
                    outputStream.write(response.getBytes());
                    logger.info("Sent response to client: " + response);
                } catch (IOException e) {
                    logger.log(Level.SEVERE, "Error handling client request: " + e.getMessage(), e);
                }
            }
        } catch (IOException e) {
            logger.log(Level.SEVERE, "Error starting daytime server: " + e.getMessage(), e);
        }
    }
}
```

GETTING INFORMATION ABOUT A SERVER SOCKET

Getting information about a Server Socket in Java involves retrieving various properties and attributes of a Server Socket object, which represents a server endpoint that listens for incoming client connections.

1. Local port number: This is the port number that the server socket is listening on for incoming connections. It can be obtained using the getLocalPort() method of the ServerSocket class.
2. Timeout value: This is the amount of time in milliseconds that the server socket will block waiting for incoming connections. It can be obtained using the getSoTimeout() method of the ServerSocket class.
3. Reuse address flag: This flag determines whether the server socket can reuse the address it is bound to. It can be obtained using the getReuseAddress() method of the ServerSocket class.
4. Bound address: This is the IP address to which the server socket is bound. It can be obtained using the getInetAddress() method of the ServerSocket class.
5. Number of clients waiting: This is the number of clients that are waiting to be accepted by the server socket. It can be obtained using the getQueueLength() method of the ServerSocket class.
6. Socket options: Socket options are various parameters that can be set on a server socket to control its behavior. They can be obtained using the getOption() method of the ServerSocket class.

GETTING INFORMATION ABOUT A SERVER SOCKET

```java
import java.net.*;
public class ServerSocInfo {
    Run | Debug
    public static void main(String[] args) {
        try {
            ServerSocket srvSoc = new ServerSocket(port:8080);
            System.out.println("Server socket created. Local port: " + srvSoc.getLocalPort());
            System.out.println("Server socket timeout: " + srvSoc.getSoTimeout());
            System.out.println("Server socket reuse address: " + srvSoc.getReuseAddress());
            System.out.println("Server socket bound address: " + srvSoc.getInetAddress());
            srvSoc.close();
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

SERVER SOCKET OPTIONS

Socket options specify how the native sockets on which the ServerSocket class relies send and receive data. Java supports three options: • SO_TIMEOUT • SO_REUSEADDR • SO_RCVBUF

SO_TIMEOUT:

- This option sets the maximum amount of time that a server socket or socket will block waiting for incoming data.
- When this timeout is exceeded, an exception will be thrown. This option can be set using the **setSoTimeout()** method of the ServerSocket or Socket class.

SO_REUSEADDR:

- This option determines whether a server socket or socket can reuse the local address it is bound to. This is useful when you need to quickly rebind a socket to the same local address after it has been closed.
- This option can be set using the **setReuseAddress**() method of the ServerSocket or Socket class.

SO_RCVBUF:

- This option sets the size of the receive buffer used by a server socket or socket for incoming data. Increasing the buffer size can improve performance when receiving large amounts of data.
- This option can be set using the **setReceiveBufferSize()** method of the ServerSocket or Socket class.

SET SOCKET OPTION

```java
import java.net.ServerSocket;
import java.io.IOException;
public class ServerSocOpt {
    Run | Debug
    public static void main(String[] args) throws IOException {
        ServerSocket srvSoc = new ServerSocket(port:8080);
        // set the socket options
        srvSoc.setSoTimeout(timeout:10000);
        srvSoc.setReuseAddress(on:true); // enable reuse of local address
        srvSoc.setReceiveBufferSize(1024*1024); // set receive buffer size to 1 MB

        // retrieve and display the socket options
        System.out.println("Timeout: " + srvSoc.getSoTimeout() + " ms");
        System.out.println("Reuse address: " + srvSoc.getReuseAddress());
        System.out.println("Receive buffer size: " + srvSoc.getReceiveBufferSize() + " bytes");
        srvSoc.close();
    }
}
```