**Introduction to SOCKET**

A socket is a software component that provides an abstraction of a network communication endpoint. It enables a program to establish a connection with another program over a network and exchange data in real-time. Sockets are a fundamental concept in network programming and are used to build all kinds of network applications. Sockets are identified by a combination of an IP address and a port number. An IP address is a unique identifier assigned to a network interface, while a port number is an identifier that specifies a specific application running on that interface. When a program wants to establish a connection with another program over a network, it needs to know the IP address and port number of the remote endpoint it wants to communicate with.

Sockets can be used for both TCP and UDP communication protocols. TCP (Transmission Control Protocol) provides a reliable, connection-oriented stream of data, while UDP (User Datagram Protocol) provides an unreliable, connectionless datagram service. There are several types of sockets, including stream sockets, datagram sockets, raw sockets, and multicast sockets. Stream sockets provide a reliable, byte-stream communication channel, while datagram sockets provide a connectionless, message-oriented communication channel. Raw sockets allow low-level access to the network interface, while multicast sockets allow communication with multiple hosts simultaneously. Sockets are used in a wide variety of network applications, including web browsing, email, file transfer, and online gaming. For example, when you browse a website, your web browser establishes a connection to the web server using a socket. The browser sends a request to the server, and the server responds with the requested web page. Similarly, when you send an email, your email client uses a socket to establish a connection to the email server and send the message. Here are the seven basic operations performed by sockets:-

1. Creating a Socket: The first step in using a socket is to create it. This is typically done using the socket() system call, which returns a file descriptor that can be used for subsequent socket operations. The socket function takes three arguments: the address family, the socket type, and the protocol. The address family specifies whether the socket will use IPv4 or IPv6 addresses, while the socket type determines whether it will be a stream or datagram socket. The protocol parameter is typically set to 0, which causes the system to select the appropriate protocol automatically.
2. Binding a Socket: Once a socket has been created, it must be bound to a specific address and port number. This is done using the bind() system call, which takes the socket file descriptor, a sockaddr structure containing the local address and port number, and the length of the sockaddr structure as arguments. Binding a socket allows other processes on the network to connect to it and send data.
3. Listening for Connections: If the socket is intended to act as a server, it must be set to listen for incoming connections using the listen() system call. This prepares the socket to accept incoming connections from clients. The listen() function takes the socket file descriptor and a backlog parameter as arguments. The backlog parameter specifies the maximum number of pending connections that can be queued by the socket.
4. Connecting to a Socket: If the socket is intended to act as a client, it must connect to a remote socket before it can send and receive data. This is done using the connect() system call, which takes the socket file descriptor, a sockaddr structure containing the remote address and port number, and the length of the sockaddr structure as arguments. Once the connection has been established, data can be sent and received using the send() and recv() functions.
5. Accepting Connections: When a server socket receives an incoming connection request from a client, it must accept the connection using the accept() system call. The accept() function takes the socket file descriptor, a sockaddr structure that will be filled with the client's address and port number, and a pointer to the length of the sockaddr structure as arguments. The accept() function returns a new socket file descriptor that can be used to communicate with the client.
6. Sending Data: Data can be sent over a socket using the send() function, which takes the socket file descriptor, a pointer to the data to be sent, the size of the data in bytes, and optional flags as arguments. The send() function returns the number of bytes sent, which may be less than the size of the data if the socket buffer is full.
7. Receiving Data: Data can be received from a socket using the recv() function, which takes the socket file descriptor, a pointer to a buffer that will receive the data, the size of the buffer in bytes, and optional flags as arguments. The recv() function returns the number of bytes received, which may be less than the size of the buffer if there is no more data to be read.

**Using Socket: Investigating Protocols With TELNET**

Telnet is a protocol that enables remote access to command-line interface (CLI) of a remote computer or server. It allows users to execute commands on the remote system and communicate with it in a textual manner. Socket programming in Java is a way of creating network applications where two different processes can communicate with each other. By using sockets, we can establish a connection between a client and a server over the network, and transfer data between them.

To investigate protocols with Telnet using socket in Java, we need to create a Telnet client that can connect to a remote Telnet server and interact with it. We can do this using the Socket class in Java. Here are the steps to implement a Telnet client using socket in Java:

1. Create a Socket object by specifying the IP address and port number of the Telnet server.
2. Create input and output streams to communicate with the Telnet server.
3. Send Telnet commands to the server by writing them to the output stream.
4. Read the server response from the input stream.
5. Repeat steps 3 and 4 as needed to execute commands and receive responses.

*import java.io.\*;*

*import java.net.\*;*

*public class TelnetClient {*

*public static void main(String[] args) {*

*String serverName = "localhost";*

*int port = 23;*

*try {*

*// Connect to the Telnet server*

*Socket socket = new Socket(serverName, port);*

*// Create input and output streams*

*InputStream in = socket.getInputStream();*

*OutputStream out = socket.getOutputStream();*

*// Send Telnet commands*

*out.write("ls\n".getBytes());*

*// Read the server response*

*BufferedReader reader = new BufferedReader(new InputStreamReader(in));*

*String line;*

*while ((line = reader.readLine()) != null) {*

*System.out.println(line);*

*} // Close the socket and streams*

*socket.close();*

*in.close();*

*out.close();*

*} catch (IOException e) {*

*e.printStackTrace();*

*}*

*}*

*}*

**Reading from Server with Socket**

To read from a server with sockets in Java, you need to create a Socket object that connects to the server using its hostname and port number, and get an InputStream from the Socket object to read data from the server. You can then read data from the InputStream using methods like read() or readLine(). Once you are done reading data, you should close the InputStream and the Socket objects using the close() method to release any resources associated with them. You can also use a try-with-resources block to ensure that the InputStream and Socket objects are properly closed even if an exception is thrown. To read from a server with sockets in Java, you need to follow these steps:

1. Create a Socket object: A Socket is an endpoint for communication between two machines over a network. To read data from a server, you need to create a Socket object by specifying the hostname and port number of the server you want to connect to. You can do this by calling the Socket() constructor and passing in the hostname and port number as parameters.
2. Get the InputStream: Once you have established a connection with the server, you need to get an InputStream from the Socket object. An InputStream is an input stream of bytes that can be read from a server. You can get the InputStream from the Socket object by calling the getInputStream() method.
3. Read data from the InputStream: Once you have the InputStream, you can start reading data from the server. You can do this by calling the read() method on the InputStream. The read() method blocks until data is available to be read, so you can put it inside a loop to continuously read data from the server.
4. Convert the InputStream to a higher-level reader: To read data from the server in a more convenient way, you can wrap the InputStream inside a higher-level reader like BufferedReader. You can do this by passing the InputStream to the InputStreamReader constructor and then passing the InputStreamReader object to the BufferedReader constructor.
5. Process the data: Once you have read data from the server, you can process it in any way you want. For example, you can print it to the console, save it to a file, or send it to another server.
6. Close the reader and the Socket: Once you are done communicating with the server, you need to close the reader and the Socket to release any resources associated with them. You can do this by calling the close() method on the reader and the Socket object.

*import java.io.BufferedReader;*

*import java.io.IOException;*

*import java.io.InputStreamReader;*

*import java.net.Socket;*

*public class Client {*

*public static void main(String[] args) {*

*String hostName = "localhost"; // set the hostname of the server*

*int portNumber = 12345; // set the port number to connect to*

*try {*

*// create a socket object to connect to the server*

*Socket socket = new Socket(hostName, portNumber);*

*// create a BufferedReader object to read from the socket*

*BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));*

*// read the message from the server and print it to the console*

*String message = in.readLine();*

*System.out.println("Server response: " + message);*

*// close the socket and the BufferedReader*

*in.close();*

*socket.close();*

*} catch (IOException e) {*

*System.err.println("Error connecting to server: " + e.getMessage());*

*}*

*}*

*}*

**Writing to Server with Sockets**

To write to a server using sockets in Java, you need to create a Socket object to connect to the server and obtain an OutputStream object from the socket. Then, you can write data to the server using the write() method of the OutputStream, which sends the data as bytes. If you want to send text instead of bytes, you can wrap the OutputStream in a PrintWriter object, which allows you to send text using the println() method. Once you have finished writing to the server, you should close the OutputStream and the Socket to release any resources they are holding. It's important to note that the server must be able to understand the data you are sending and know how to interpret it. Here are the steps to writing to a server using sockets in Java:

1. Create a Socket object to connect to the server, specifying the hostname and port number of the server as arguments to the Socket constructor. This will create a client-side socket that is connected to the server.
2. Obtain an OutputStream object from the Socket using the getOutputStream() method. This will give you an output stream that you can use to send data to the server.
3. Write data to the output stream using the write() method. This method takes an array of bytes as an argument, so if you want to send text data, you will need to convert the text to bytes first using the getBytes() method of the String class.
4. If you are sending text data, you may want to wrap the OutputStream in a PrintWriter object, which will allow you to use the println() method to send text instead of bytes.
5. Once you have finished writing data to the server, you should close the OutputStream and the Socket using the close() method. This will release any resources they are holding and free up system resources.

*import java.io.IOException;*

*import java.io.OutputStream;*

*import java.net.Socket;*

*public class Client {*

*public static void main(String[] args) {*

*String hostName = "localhost"; // set the hostname of the server*

*int portNumber = 12345; // set the port number to connect to*

*String message = "Hello, server!"; // set the message to send to the server*

*try {*

*// create a socket object to connect to the server*

*Socket socket = new Socket(hostName, portNumber);*

*// get the output stream from the socket*

*OutputStream out = socket.getOutputStream();*

*// send the message to the server*

*out.write(message.getBytes());*

*// close the output stream and the socket*

*out.close();*

*socket.close();*

*} catch (IOException e) {*

*System.err.println("Error connecting to server: " + e.getMessage());*

*}*

*}*

*}*

**Constructing And Connecting Sockets: Basic Consturctors**

***The Socket Class***

The Socket class is a programming construct provided by various networking libraries in different programming languages that allows for communication between two programs over a network. It provides methods for creating and managing sockets, connecting to remote sockets, sending and receiving data, and handling errors and exceptions that may occur during the communication process. The Socket class typically provides a set of constructors and methods that allow programmers to customize various aspects of socket communication, such as the protocol used, the address family, and the socket options. By using the Socket class, programmers can create robust network applications that can communicate with other programs over a wide range of network topologies and protocols.

**Basic Constructors**

Each Socket constructor specifies the host and the port to connect to. Hosts may be specified as an InetAddress or a String. Remote ports are specified as int values from 1 to 65535:

**public** Socket(String host, **int** port) **throws** UnknownHostException, IOException

**public** Socket(InetAddress host, **int** port) **throws** IOException

These constructors connect the socket (i.e., before the constructor returns, an active network connection is established to the remote host). If the connection can’t be opened for some reason, the constructor throws an IOException or an UnknownHostExcep tion. For example:

**try** {  
Socket toOReilly = **new** Socket("www.oreilly.com", 80); *// send and receive data...*

} **catch** (UnknownHostException ex) { System.err.println(ex);

} **catch** (IOException ex) { System.err.println(ex);

}

In this constructor, the host argument is just a hostname expressed as a String. If the domain name server cannot resolve the hostname or is not functioning, the constructor throws an UnknownHostException. If the socket cannot be opened for some other rea‐ son, the constructor throws an IOException. There are many reasons a connection attempt might fail: the host you’re trying to reach may not accept connections on that port, the hotel WiFi service may be blocking you until you log in to its website and pay $14.95, or routing problems may be preventing your packets from reaching their destination.

Because this constructor doesn’t just create a Socket object but also tries to connect the socket to the remote host, you can use the object to determine whether connections to a particular port are allowed.

*Find out which of the first 1024 ports seem to be hosting TCP servers on a specified host*

***import java.net.\*****;* ***import java.io.\*****;*

***public class LowPortScanner*** *{****public static void*** *main(String[] args) {*

*String host = args.length > 0 ? args[0] : "localhost";*

***for****(****int****i=1;i<1024;i++){* ***try*** *{*

*Socket s =* ***new*** *Socket(host, i);  
System.out.println("There is a server on port " + i + " of "*

*+ host);*

*s.close();*

*}* ***catch*** *(UnknownHostException ex) { System.err.println(ex);****break****;*

*}* ***catch*** *(IOException ex) {  
// must not be a server on this port*

*} }*

*} }*

**Constructing without connecting**

Constructing without connecting is a technique used in network programming to create a network socket without establishing a connection to another socket. This is useful in situations where a socket needs to be created before a connection is made, such as in a client-server model where the server socket is created in advance and waits for incoming connection requests, while the client socket is created later when the client requests a connection. By separating the construction and connection phases, the program can handle these tasks independently and with greater flexibility, allowing for more efficient network communication. This approach also makes it possible to create a pool of sockets that can be used to handle multiple connections simultaneously, making the program more scalable and efficient.

For example, let's say you are developing a client-server application where the server waits for incoming connection requests from multiple clients. In this scenario, the server needs to create a socket that listens for incoming connection requests, while the clients create sockets that connect to the server socket. To implement this, the server will first create a socket using the socket() system call, and then bind it to a specific IP address and port using the bind() system call. Finally, the server will call the listen() system call to make the socket ready for incoming connection requests. At this point, the server socket is constructed without connecting to any clients.

On the other hand, the client will create a socket using the socket() system call, and then call the connect() system call to establish a connection to the server socket. By separating the construction and connection phases, the program can handle these tasks independently and with greater flexibility.

In summary, constructing without connecting is a technique that allows network programmers to create sockets in advance, making the program more efficient and scalable. This approach is commonly used in client-server applications, where the server waits for incoming connections and clients establish connections to the server socket.

**Socket Address**

A socket address, also known as a network endpoint, is a combination of an IP address and a port number that uniquely identifies a communication endpoint in a network. It is used to establish a connection between two networked devices and to exchange data between them.

The IP address identifies the device or host, while the port number identifies a specific application running on that device. Together, they form the socket address. The port number is a 16-bit integer value ranging from 0 to 65535, and it identifies a specific process or service running on the device. The IP address is a 32-bit or 128-bit binary value that identifies the device on the network.

For example, let's say we have a server running on a computer with the IP address 192.168.0.1, and the server is listening on port number 80. This means that any incoming connection requests to the server should be sent to the IP address 192.168.0.1 on port number 80.

In this example, the socket address for the server would be represented as "192.168.0.1:80". When a client connects to this server, it would need to specify this socket address to establish a connection. Similarly, if the server needs to connect to another device on the network, it would need to specify the socket address of that device.

Socket addresses are a crucial component of network communication, allowing devices to communicate and exchange data with each other in a standardized way.

*import java.net.\*;*

*public class SocketAddressExample {*

*public static void main(String[] args) {*

*try {*

*// create a socket address with an IP address and port number*

*SocketAddress addr = new InetSocketAddress("192.168.0.1", 80);*

*// create a socket using the socket address*

*Socket socket = new Socket();*

*socket.connect(addr);*

*// print out the local and remote socket addresses*

*System.out.println("Local Socket Address: " + socket.getLocalSocketAddress());*

*System.out.println("Remote Socket Address: " + socket.getRemoteSocketAddress());*

*// close the socket*

*socket.close();*

*} catch (Exception e) {*

*e.printStackTrace();*

*}*

*}*

*}*

**Getting Information About A Socket**

Getting information about a socket involves obtaining details such as its address, port number, and protocol. The process starts by choosing a protocol, such as TCP or UDP. Then, a socket is created using the chosen protocol, and it is bound to a specific address and port number. If the socket is a server socket, it needs to listen for incoming connections, and when a client connects, the connection needs to be accepted. Finally, information about the socket can be obtained, such as its local or remote address and port number, using functions like getsockname() or getpeername(). By following these steps, the socket can be used to establish a two-way communication link between two programs running on a network. There are several methods to obtain information about a socket, depending on the programming language and operating system being used. Here are some common methods:

1. getLocalSocketAddress(): This method returns the local socket address to which the socket is bound. For example, you can call socket.getLocalSocketAddress() on a Socket object socket to get its local socket address.
2. getRemoteSocketAddress(): This method returns the remote socket address to which the socket is connected. For example, you can call socket.getRemoteSocketAddress() on a Socket object socket that is connected to a remote host to get its remote socket address.
3. getInputStream(): This method returns an input stream for reading data from the socket. For example, you can call socket.getInputStream() on a Socket object socket to get its input stream.
4. getOutputStream(): This method returns an output stream for writing data to the socket. For example, you can call socket.getOutputStream() on a Socket object socket to get its output stream.
5. getKeepAlive(): This method returns a boolean value indicating whether the socket has the keep-alive option enabled. For example, you can call socket.getKeepAlive() on a Socket object socket to check if the keep-alive option is enabled.
6. getSoTimeout(): This method returns the timeout value in milliseconds for blocking socket operations. For example, you can call socket.getSoTimeout() on a Socket object socket to get its timeout value.
7. getTcpNoDelay(): This method returns a boolean value indicating whether the Nagle algorithm is disabled. For example, you can call socket.getTcpNoDelay() on a Socket object socket to check if the Nagle algorithm is disabled.

*import java.net.Socket;*

*public class SocketInformationGetter {*

*public static void main(String[] args) {*

*String hostName = "example.com";*

*int portNumber = 80;*

*try (Socket socket = new Socket(hostName, portNumber)) {*

*// get the local address and port number of the socket*

*String localAddress = socket.getLocalAddress().getHostAddress();*

*int localPort = socket.getLocalPort();*

*System.out.println("Local address: " + localAddress);*

*System.out.println("Local port: " + localPort);*

*// get the remote address and port number of the socket*

*String remoteAddress = socket.getInetAddress().getHostAddress();*

*int remotePort = socket.getPort();*

*System.out.println("Remote address: " + remoteAddress);*

*System.out.println("Remote port: " + remotePort);*

*// get the TCP no-delay setting of the socket*

*boolean tcpNoDelay = socket.getTcpNoDelay();*

*System.out.println("TCP no-delay setting: " + tcpNoDelay);*

*// get the SO timeout setting of the socket*

*int soTimeout = socket.getSoTimeout();*

*System.out.println("SO timeout setting: " + soTimeout);*

*// get the keep-alive setting of the socket*

*boolean keepAlive = socket.getKeepAlive();*

*System.out.println("Keep-alive setting: " + keepAlive);*

*} catch (Exception e) {*

*e.printStackTrace();*

*}*

*}*

*}*

**Socket Options**

Socket options are parameters that can be set on a socket to modify its behavior or configure its properties. These options can be used to control various aspects of the socket's operation, such as its behavior when sending and receiving data, its timeout values, buffer sizes, and more. Socket options are implemented as integers that are passed as arguments to the setsockopt function, which is used to modify the socket's settings. The need for socket options arises from the fact that different applications may have different requirements and preferences for how sockets should operate. By providing a way to customize the behavior of a socket, socket options allow applications to optimize their network communication and achieve better performance, reliability, and security.

1. TCP\_NODELAY: The TCP\_NODELAY option disables the Nagle algorithm, which is used to improve network efficiency by reducing the number of small packets that are sent over the network. If this option is enabled, small packets will be sent immediately, which can improve latency but may decrease network efficiency. Here's an example:

*import java.net.Socket;*

*import java.net.SocketException;*

*public class Example {*

*public static void main(String[] args) throws SocketException {*

*Socket socket = new Socket("localhost", 8080);*

*// Enable TCP\_NODELAY option*

*socket.setTcpNoDelay(true);*

*}*

*}*

1. SO\_BINDADDR: The SO\_BINDADDR option allows you to specify the local address and port that a socket should be bound to. This is useful when you want to bind to a specific interface or port. Here's an example:

*import java.net.InetAddress;*

*import java.net.InetSocketAddress;*

*import java.net.Socket;*

*public class Example {*

*public static void main(String[] args) {*

*InetAddress addr = InetAddress.getByName("localhost");*

*int port = 8080;*

*Socket socket = new Socket();*

*socket.bind(new InetSocketAddress(addr, port));*

*}*

*}*

1. SO\_TIMEOUT: The SO\_TIMEOUT option specifies the maximum amount of time that a socket will wait for data to be received before throwing a SocketTimeoutException. Here's an example:

*import java.io.IOException;*

*import java.net.Socket;*

*import java.net.SocketTimeoutException;*

*public class Example {*

*public static void main(String[] args) throws IOException {*

*Socket socket = new Socket("localhost", 8080);*

*// Set timeout to 5 seconds*

*socket.setSoTimeout(5000);*

*// Wait for data to be received*

*try {*

*byte[] data = new byte[1024];*

*int bytesRead = socket.getInputStream().read(data);*

*} catch (SocketTimeoutException e) {*

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

*}*

*}*

*}*

1. SO\_LINGER: The SO\_LINGER option controls the behavior of a socket when it is closed. If this option is enabled with a non-zero timeout, the socket will wait for any remaining data to be sent before closing. Here's an example:

*import java.io.IOException;*

*import java.net.Socket;*

*import java.net.SocketException;*

*public class Example {*

*public static void main(String[] args) throws IOException {*

*Socket socket = new Socket("localhost", 8080);*

*// Set SO\_LINGER option with 10 second timeout*

*socket.setSoLinger(true, 10);*

*// Close the socket*

*socket.close();*

*}*

*}*

1. SO\_SNDBUF: The SO\_SNDBUF option specifies the size of the send buffer used by a socket. This can be used to optimize network performance by ensuring that enough data is buffered before sending it over the network. Here's an example:

*import java.net.Socket;*

*import java.net.SocketException;*

*public class Example {*

*public static void main(String[] args) throws SocketException {*

*Socket socket = new Socket("localhost", 8080);*

*// Set send buffer size to 64KB*

*socket.setSendBufferSize(64 \* 1024);*

*}*

*}*

1. SO\_RCVBUF: The SO\_RCVBUF option specifies the size of the receive buffer used by a socket. The receive buffer is a temporary storage area in memory where incoming data is held before it is processed. By increasing the size of the receive buffer, you can reduce the risk of data loss due to packet drops. Here's an example:

*import java.net.Socket;*

*import java.net.SocketException;*

*public class Example {*

*public static void main(String[] args) throws SocketException {*

*Socket socket = new Socket("localhost", 8080);*

*// Set receive buffer size to 64KB*

*socket.setReceiveBufferSize(64 \* 1024);*

*}*

*}*

1. SO\_KEEPALIVE: The SO\_KEEPALIVE option enables or disables the keep-alive mechanism for a socket. When enabled, the socket periodically sends a keep-alive packet to the remote end of the connection to ensure that the connection is still active. If no response is received, the connection is considered to be broken and an exception is thrown. Here's an example:

*import java.net.Socket;*

*import java.net.SocketException;*

*public class Example {*

*public static void main(String[] args) throws SocketException {*

*Socket socket = new Socket("localhost", 8080);*

*// Enable keep-alive mechanism*

*socket.setKeepAlive(true);*

*}*

*}*

**Socket In GUI Application**

Sockets can be used in GUI (Graphical User Interface) applications to enable communication between different components of the application or between the application and a remote server. Java provides classes such as Socket and ServerSocket that can be used to establish a connection and exchange data between a client and server. For instance, a chat application can be implemented using sockets where the GUI acts as a client and connects to a server using a Socket object. Sockets can also be used to implement features such as file transfers and remote procedure calls in GUI applications. With sockets, the client and server can exchange data, enabling real-time chat, file transfer, and remote procedure calls in the application.

**WHOIS AND A NETWROK CLIENT LIBRARY**

Whois is a protocol used to query databases that contain information about domain names, IP addresses, and other network-related entities. The data returned by a Whois query typically includes information about the domain name registrar, domain expiration date, and contact information for the domain owner. In Java networking, a Whois client can be implemented using the java.net.Socket class to establish a TCP connection to the Whois server and send a query. Once the query is sent, the client reads the response from the server and parses it to extract the relevant information.

A network client library in Java is a collection of classes and methods that provide a convenient way to perform common network-related tasks, such as sending HTTP requests, establishing TCP connections, and sending email. Popular network client libraries in Java include Apache HttpClient, OkHttp, and Retrofit.

These libraries typically provide a high-level interface that abstracts away the details of low-level network programming, allowing developers to focus on their application logic. For example, the Apache HttpClient library provides a HttpClient class that can be used to send HTTP requests and receive responses, without the need to worry about low-level details such as establishing a TCP connection and parsing the response. Similarly, the OkHttp library provides a simple API for sending HTTP requests and handling responses, while Retrofit provides a type-safe interface for calling RESTful web services.

*import java.awt.\*;*

*import java.awt.event.\*;*

*import javax.swing.\*;*

*import java.net.\*;*

*import java.io.\*;*

*public class WhoisClient extends JFrame implements ActionListener {*

*// Declare GUI components*

*private JTextField inputField;*

*private JTextArea outputArea;*

*private JButton lookupButton;*

*public WhoisClient() {*

*// Set window title*

*setTitle("Whois Client");*

*// Create GUI components*

*inputField = new JTextField(30);*

*outputArea = new JTextArea(15, 60);*

*lookupButton = new JButton("Lookup");*

*// Add GUI components to window*

*JPanel inputPanel = new JPanel();*

*inputPanel.add(new JLabel("Enter domain name or IP address:"));*

*inputPanel.add(inputField);*

*inputPanel.add(lookupButton);*

*add(inputPanel, BorderLayout.NORTH);*

*add(new JScrollPane(outputArea), BorderLayout.CENTER);*

*// Set event listener for lookup button*

*lookupButton.addActionListener(this);*

*}*

*public void actionPerformed(ActionEvent event) {*

*// Get user input*

*String input = inputField.getText();*

*try {*

*// Connect to whois server*

*Socket socket = new Socket("whois.internic.net", 43);*

*PrintWriter out = new PrintWriter(socket.getOutputStream(), true);*

*BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));*

*// Send query to server*

*out.println(input);*

*// Read response from server*

*String line;*

*while ((line = in.readLine()) != null) {*

*outputArea.append(line + "\n");*

*}*

*// Close connection*

*socket.close();*

*} catch (IOException ex) {*

*outputArea.setText("Error: " + ex.getMessage());*

*}*

*}*

*public static void main(String[] args) {*

*// Create and show window*

*WhoisClient client = new WhoisClient();*

*client.pack();*

*client.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);*

*client.setVisible(true);*

*}*

*}*