Java-Networking

Java Networking

Definition
Java Networking is a set of **APIs in Java** that allows applications to **communicate over networks** using protocols like **TCP/IP, UDP, HTTP**, etc.

- Enables **data exchange between computers, servers, or devices.**

Why Java Networking is Needed

- Build **client-server applications**
- Access **resources over the Internet**
- Enable **distributed systems**
- Supports protocols like **HTTP, FTP, TCP, UDP**

Key Java Networking Classes

Class	Purpose
java.net.Socket	Represents **client socket** for TCP connection
java.net.ServerSocket	Represents **server socket** to listen for clients
java.net.URL	Represents **web resource** URL
java.net.HttpURLConnection	Connect and communicate over **HTTP**
InetAddress	Represents **IP address** of a host

Networking Concepts in Java
A. TCP (Reliable, Connection-Oriented)

- Uses Socket (client) & ServerSocket (server)
- Guarantees **ordered and error-free delivery**

Example (Server):

```
ServerSocket server = new ServerSocket(5000);
Socket client = server.accept();
BufferedReader in = new BufferedReader(new
InputStreamReader(client.getInputStream()));
System.out.println("Message: " + in.readLine());
```

Example (Client):

```
Socket socket = new Socket("localhost", 5000);
PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
out.println("Hello Server");
```

B. UDP (Connectionless, Faster)

- Uses DatagramSocket and DatagramPacket
- No guarantee of delivery; lightweight

```
DatagramSocket socket = new DatagramSocket();
String msg = "Hello UDP";
byte[] buffer = msg.getBytes();
InetAddress address = InetAddress.getByName("localhost");
DatagramPacket packet = new DatagramPacket(buffer, buffer.length, address, 5000);
socket.send(packet);
socket.close();
```

C. URL & HTTP

```
URL url = new URL("https://www.example.com");
HttpURLConnection con = (HttpURLConnection) url.openConnection();
```

```
con.setRequestMethod("GET");
BufferedReader in = new BufferedReader(new InputStreamReader(con.getInputStream()));
String line;
while((line = in.readLine()) != null) {
    System.out.println(line);
}
in.close();
```

Real-Life Analogy

- **TCP Socket** → Phone call (reliable, connected)
- **UDP Socket** → Postcard (fast, may be lost)
- **URL/HTTP** → Visiting a website

Advantages

- Enables **client-server communication**
- Supports multiple **protocols**
- Cross-platform & integrated into Java API

Best Practices

- Always **close sockets** to avoid resource leaks
- Use **try-with-resources** for streams
- Handle **network exceptions** (IOException, SocketException)
- Use **multithreading** for server handling multiple clients

---

Common Interview Questions (Cognizant Level)

Q1. Difference between TCP and UDP?

- TCP → Reliable, connection-oriented
- UDP → Fast, connectionless, may lose data

Q2. Which Java classes are used for TCP communication?

A. Socket and ServerSocket

Q3. How to send data over UDP in Java?

A. Using DatagramSocket and DatagramPacket

Q4. How to read a webpage in Java?

A. Using URL and HttpURLConnection

Q5. What is the role of InetAddress?

A. Represents host IP address and allows hostname resolution

One-Line Summary (Quick Revision)

Java Networking provides APIs like Socket, ServerSocket, URL, and DatagramSocket to build client-server, HTTP, and UDP/TCP-based applications.

Java Socket Programming

Definition:

Socket Programming in Java allows **two-way communication between programs over a network** using **TCP/IP protocols.**

- A **Socket** is an endpoint for sending/receiving data between **client and server.**

Why Socket Programming is Needed

- Build **client-server applications**
- Exchange **real-time data** (chat apps, multiplayer games)

- Enables **network-based distributed systems**
- Works for **TCP (reliable)** and **UDP (fast)** communication

---

Key Components

Component	Description
Socket	Represents a **client connection**
ServerSocket	Listens for **incoming client requests**
InputStream / OutputStream	Send/receive **data streams**
InetAddress	Represents **IP address** of host

TCP Socket Programming Example

A. Server Code

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

public class TCPServer {
    public static void main(String[] args) throws IOException {
        ServerSocket server = new ServerSocket(5000);
        System.out.println("Server is running...");
        Socket client = server.accept();
        System.out.println("Client connected!");

        BufferedReader in = new BufferedReader(new
InputStreamReader(client.getInputStream()));
        PrintWriter out = new PrintWriter(client.getOutputStream(), true);

        String msg = in.readLine();
        System.out.println("Client says: " + msg);

        out.println("Hello Client!");
        client.close();
        server.close();
    }
}
```

B. Client Code

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

public class TCPClient {
    public static void main(String[] args) throws IOException {
        Socket socket = new Socket("localhost", 5000);
        PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
        BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));

        out.println("Hello Server!");
        System.out.println("Server says: " + in.readLine());

        socket.close();
    }
}
```

UDP Socket Programming Example

```
DatagramSocket socket = new DatagramSocket();
```

```
String msg = "Hello UDP";
byte[] buffer = msg.getBytes();
InetAddress address = InetAddress.getByName("localhost");
DatagramPacket packet = new DatagramPacket(buffer, buffer.length, address, 5000);
socket.send(packet);
socket.close();
```

Real-Life Analogy

- **TCP Socket** → Phone call (reliable, continuous)
- **UDP Socket** → Postcard (fast, may be lost)
- **ServerSocket** → Receptionist, waiting for clients

Advantages

- Enables **real-time communication**
- Works over **LAN & Internet**
- Supports **multithreading for multiple clients**
- Flexible for **TCP & UDP protocols**

Best Practices

- Always **close sockets and streams**
- Handle **exceptions** (IOException, SocketException)
- Use **multithreading** on server for multiple clients
- Avoid **blocking calls** on client if server is slow

Common Interview Questions (Cognizant Level)

Q1. What is Socket in Java?

A. Endpoint for communication between client and server.

Q2. Difference between TCP and UDP Sockets?

- TCP → Reliable, connection-oriented
- UDP → Fast, connectionless

Q3. What is ServerSocket?

A. Listens for client connections and creates a Socket per client.

Q4. How to send and receive data via sockets?

A. Using InputStream and OutputStream (BufferedReader / PrintWriter).

Q5. How to handle multiple clients?

A. Use **multithreaded server**, each client in a separate thread.

One-Line Summary (Quick Revision)

Socket Programming in Java enables two-way communication between client and server over TCP/UDP using Socket and ServerSocket APIs.

---

java.net.ServerSocket

Definition:

ServerSocket is a **Java class** used to create a **server application** that listens for **incoming client connections** on a specified **port number.**

- When a client connects, it returns a **Socket object** to communicate with that client.

Why ServerSocket is Needed

- Enables **server-side network programming**
- Accepts **multiple client connections**

- Acts as a **gateway for TCP-based communication**
- Provides **reliable connection-oriented communication**

Key Constructors

Constructor	Description
ServerSocket(int port)	Creates server listening on specified port
ServerSocket(int port, int backlog)	Backlog = max queued clients
ServerSocket(int port, int backlog, InetAddress bindAddr)	Bind to specific IP address

---

Key Methods

Method	Purpose
accept()	Waits for **client connection**, returns a Socket
close()	Closes the server socket
getInetAddress()	Returns server IP address
getLocalPort()	Returns the port number server is listening on
setSoTimeout(int timeout)	Sets timeout for accept()

Simple Example: ServerSocket

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

public class MyServer {
    public static void main(String[] args) throws IOException {
        ServerSocket server = new ServerSocket(5000);
        System.out.println("Server is listening on port 5000...");

        Socket client = server.accept(); // Wait for client
        System.out.println("Client connected: " + client.getInetAddress());

        BufferedReader in = new BufferedReader(new
InputStreamReader(client.getInputStream()));
        PrintWriter out = new PrintWriter(client.getOutputStream(), true);

        String message = in.readLine();
        System.out.println("Client says: " + message);

        out.println("Hello Client, message received!");

        client.close();
        server.close();
    }
}
```

Real-Life Analogy

- **ServerSocket** → Receptionist at office waiting for visitors
- **accept()** → Receptionist greets and provides access
- **Socket** → Conversation channel with visitor

Advantages

- Handles **incoming client connections reliably**
- Supports **multiple clients using multithreading**

- Works with **TCP connections** (reliable, ordered data)

Best Practices

- Always **close ServerSocket** after use
- Use **try-with-resources** to manage resources
- Use **multithreading** to handle multiple clients
- Set **timeouts** to avoid indefinite blocking on accept()

Common Interview Questions (Cognizant Level)

Q1. What is ServerSocket?
A. Class in Java used to create a server that listens for client connections.

Q2. How to accept client connections?
A. Using accept() method which returns a Socket object.

Q3. Can ServerSocket handle multiple clients?
A. Yes, by using **multithreading**, each client gets a separate Socket.

Q4. What is the difference between ServerSocket and Socket?

- ServerSocket → Listens for clients
- Socket → Communicates with client

Q5. How to prevent ServerSocket from blocking forever?
A. Use setSoTimeout(int millis) to set a timeout.

One-Line Summary (Quick Revision)

ServerSocket is a Java class that listens for client connections on a specified port and creates a Socket to communicate with each client.

Implementing Socket Client in Java

Definition

A **Socket Client** in Java is a program that **connects to a server using TCP/IP** via a **Socket object** and communicates by **sending/receiving data.**

Why Socket Client is Needed

- To **request services** from a server
- Enables **two-way communication**
- Works with **ServerSocket** for distributed applications
- Ideal for **chat apps, file transfer, real-time systems**

Key Steps to Implement a Socket Client

1. **Create a Socket** → Connect to server IP and port
2. **Get Input/Output Streams** → Send/receive data
3. **Send/Receive Data** → Using PrintWriter, BufferedReader, or streams
4. **Close Connection** → Release resources

Java Socket Client Example

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

public class SocketClient {
    public static void main(String[] args) {
        try {
            // Step 1: Connect to server on localhost:5000
            Socket socket = new Socket("localhost", 5000);
```

```
            System.out.println("Connected to server!");

            // Step 2: Create input and output streams
            PrintWriter out = new PrintWriter(socket.getOutputStream(), true); // send
data
            BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream())); // receive data
            BufferedReader userInput = new BufferedReader(new
InputStreamReader(System.in)); // keyboard input

            // Step 3: Send message to server
            System.out.print("Enter message to server: ");
            String msg = userInput.readLine();
            out.println(msg);

            // Step 4: Receive response from server
            String response = in.readLine();
            System.out.println("Server says: " + response);

            // Step 5: Close resources
            socket.close();
            in.close();
            out.close();
            userInput.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Real-Life Analogy

- **Client** → Person calling a **receptionist**
- **ServerSocket** → Receptionist waiting for calls
- **Socket** → Telephone line
- **Streams** → Conversation through the line

Advantages

- Enables **real-time client-server communication**
- Works over **LAN or Internet**
- Flexible for **TCP communication**
- Can integrate with **multithreaded server for multiple clients**

Best Practices

- Always **close socket and streams**
- Handle **exceptions** (IOException)
- Use **try-with-resources** for safety
- Avoid blocking operations on UI threads

Common Interview Questions (Cognizant Level)

Q1. How to create a client socket in Java?

A. Use Socket socket = new Socket("host", port)

Q2. How does client communicate with server?

A. Through **InputStream** and **OutputStream**

Q3. Difference between Socket and ServerSocket?

- Socket → Client-side endpoint

- ServerSocket → Listens for incoming client connections

Q4. How to send data from client to server?

A. Using PrintWriter or OutputStream

Q5. How to receive data from server?

A. Using BufferedReader or InputStream

One-Line Summary (Quick Revision)

A Socket Client in Java connects to a server via a Socket, sends and receives data using streams, and closes the connection after communication.
