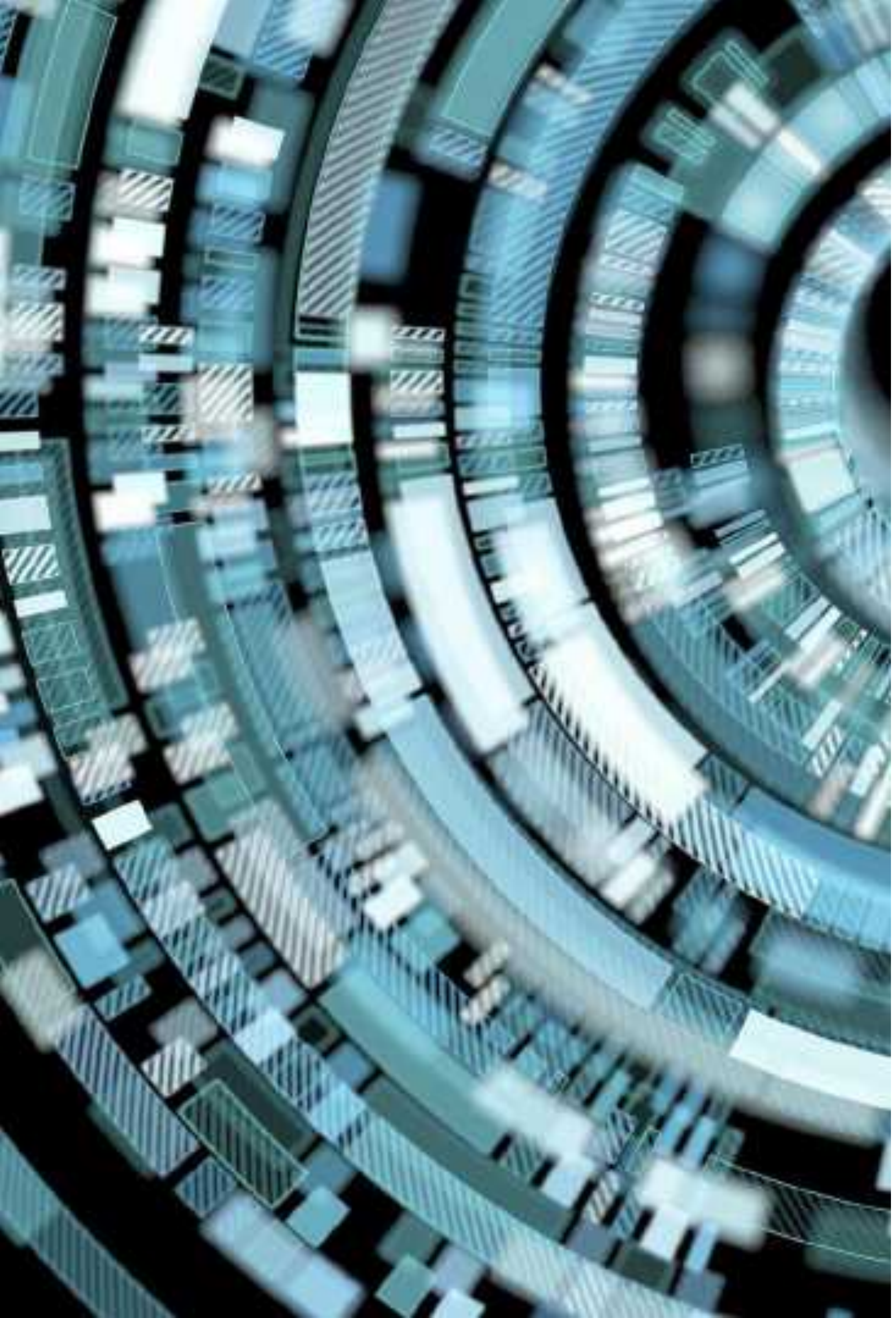
# Computer System Design & Application
# 计算机系统设计与应用A

陶伊达 （TAO Yida）
taoyd@sustech.edu.cn



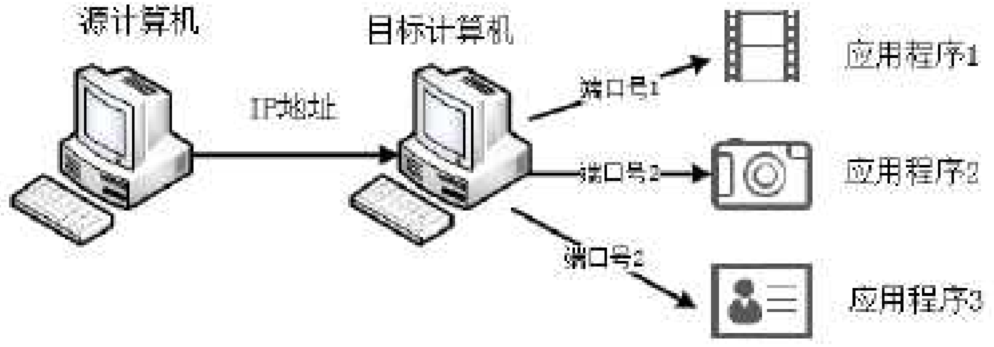
# Lecture 9

- Network Basics
- Network Protocols
- Socket Programming

## Networking

- Networking is a concept of connecting two or more computing devices together so that we can share resources
- The `java.net` package provides a powerful and flexible infrastructure for networking, providing various classes and interfaces that execute the low-level communication features

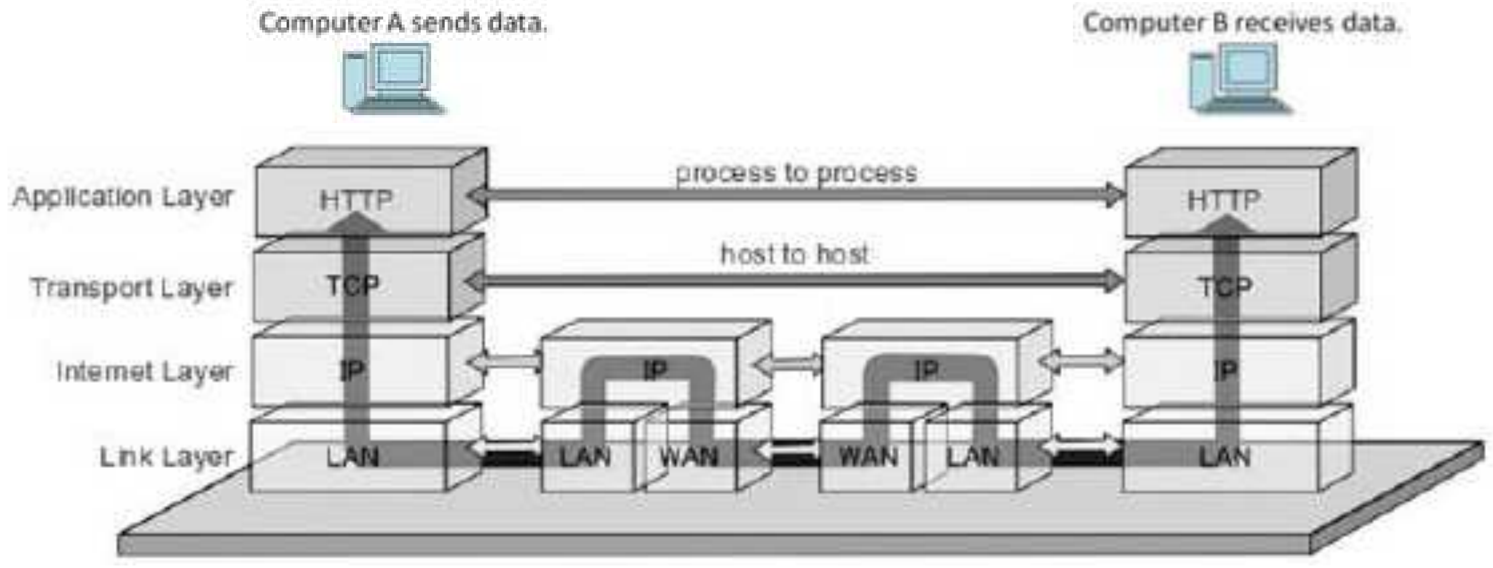
# Networking Terminology



- **IP address**: a unique address that distinguishes a device on the internet or a local network
- **Domain name**: a human-friendly version of an IP address that you enter in browser (translated by DNS)
- **Port number**: a number used to identify different applications/processes uniquely

## Network Architecture

- Network architecture refers to a network's structural and logical layout. It describes how the network devices are connected and the rules that govern data transfer between them



https://www.elprocus.com/tcp-ip-protocol-architecture-and-its-layers/

# Network Architecture

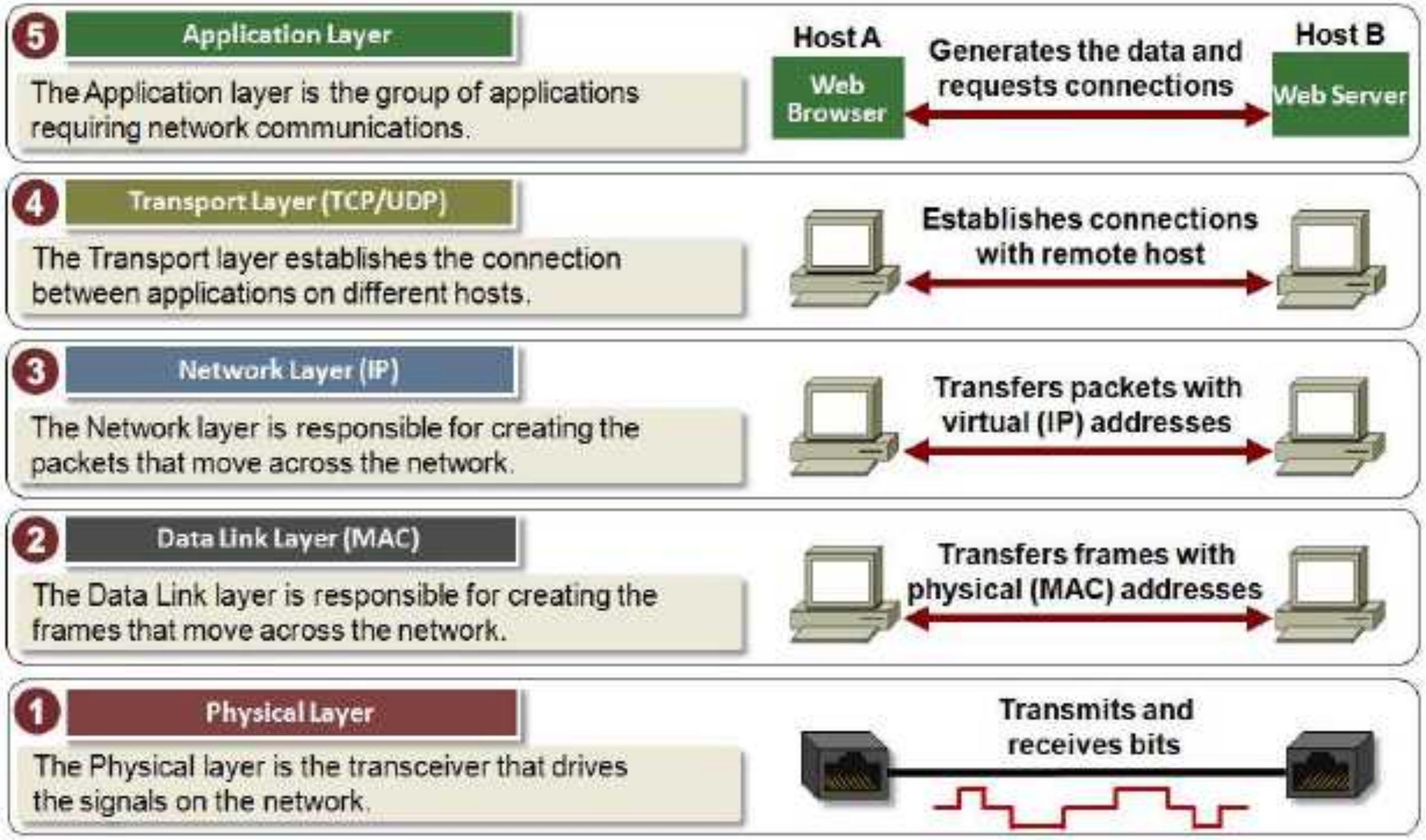
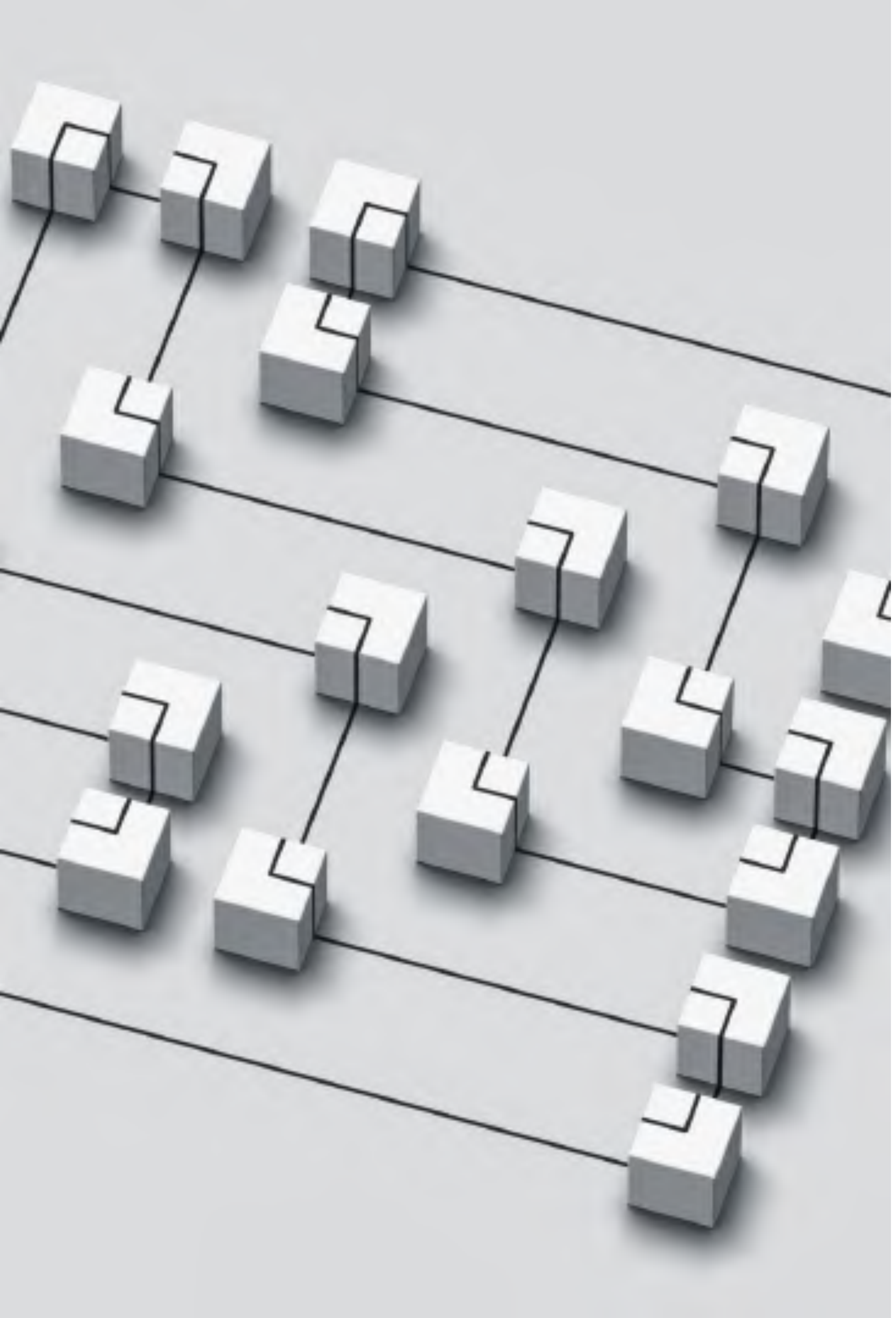

5
Application Layer
The Application layer is the group of applications requiring network communications.
Host A
Web Browser
Generates the data and requests connections
Host B
Web Server
4
Transport Layer (TCP/UDP)
The Transport layer establishes the connection between applications on different hosts.
Establishes connections with remote host
3
Network Layer (IP)
The Network layer is responsible for creating the packets that move across the network.
Transfers packets with virtual (IP) addresses
2
Data Link Layer (MAC)
The Data Link layer is responsible for creating the frames that move across the network.
Transfers frames with physical (MAC) addresses
1
Physical Layer
The Physical layer is the transceiver that drives the signals on the network.
Transmits and receives bits

Image: https://www.networxsecurity.org/members-area/glossary/i/internet-protocol.html

## How do devices communicate with each other?

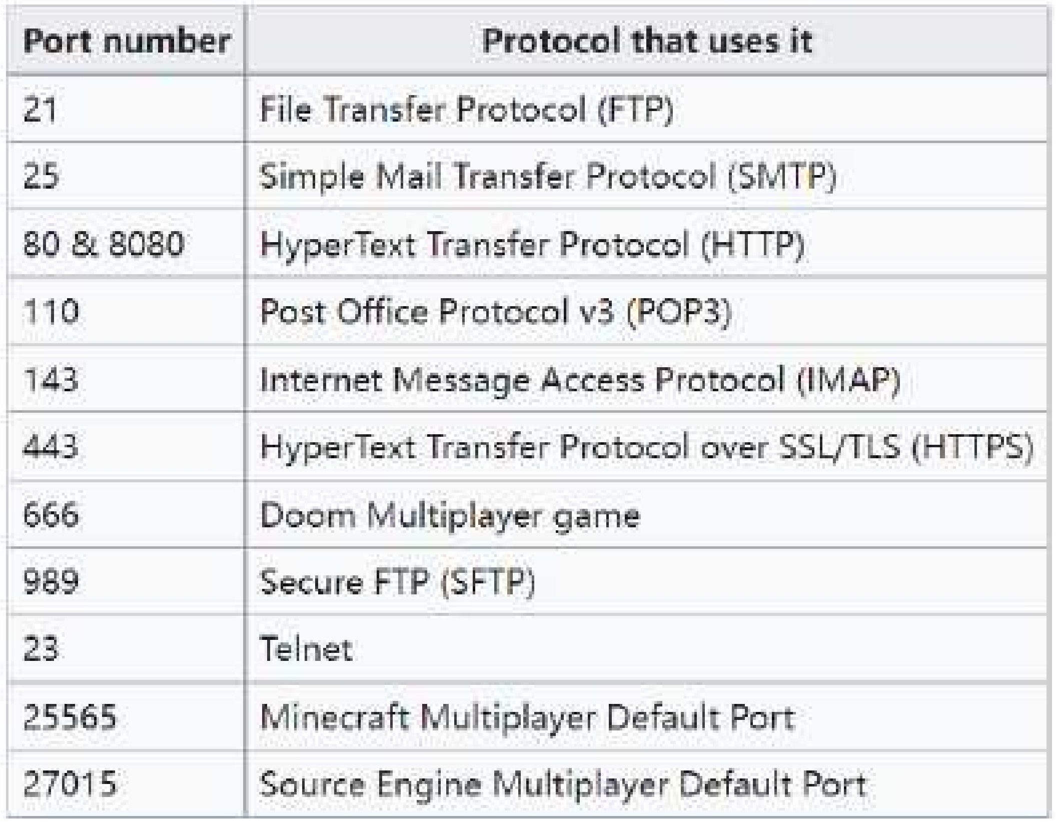


# Network Protocols

- A network protocol is a set of established rules that dictate how to format, transmit and receive data so that computer network devices can communicate, regardless of the differences in their underlying infrastructures, designs or standards.
- To successfully send and receive information, devices on both sides of a communication exchange must accept and follow protocol conventions
- Without computing protocols, computers and other devices would not know how to engage with each other.

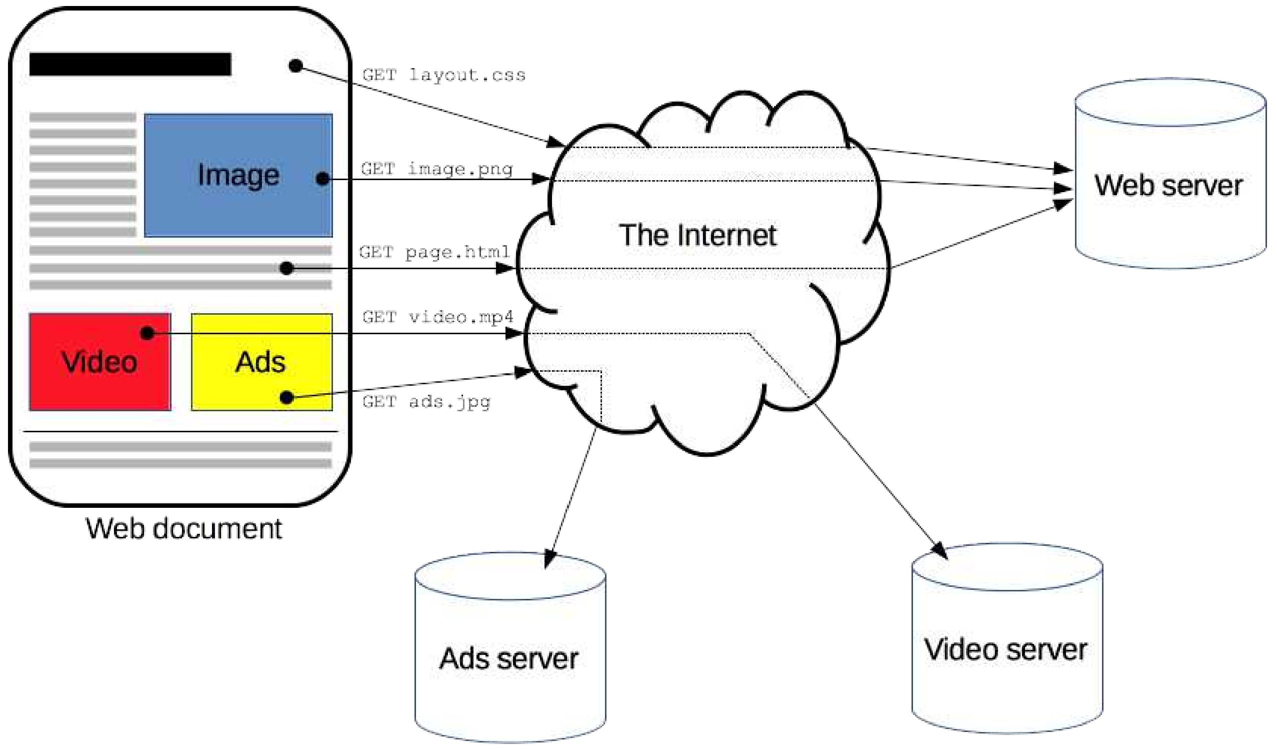
## Application Layer Protocols

- Each Internet application has a different application protocol, which describes how the data for that particular application are transmitted.
- A port number helps a computer decide which application should receive an incoming piece of data



| Port number | Protocol that uses it |
|---|---|
| 21 | File Transfer Protocol (FTP) |
| 25 | Simple Mail Transfer Protocol (SMTP) |
| 80 & 8080 | HyperText Transfer Protocol (HTTP) |
| 110 | Post Office Protocol v3 (POP3) |
| 143 | Internet Message Access Protocol (IMAP) |
| 443 | HyperText Transfer Protocol over SSL/TLS (HTTPS) |
| 666 | Doom Multiplayer game |
| 989 | Secure FTP (SFTP) |
| 23 | Telnet |
| 25565 | Minecraft Multiplayer Default Port |
| 27015 | Source Engine Multiplayer Default Port |

# HTTP (Hypertext Transfer Protocol)



https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview

- HTTP is a protocol for fetching resources such as HTML documents.
- It is the foundation of any data exchange on the Web
- It is a client-server protocol, which means requests are initiated by the recipient, usually the Web browser.

# HTTP Commands

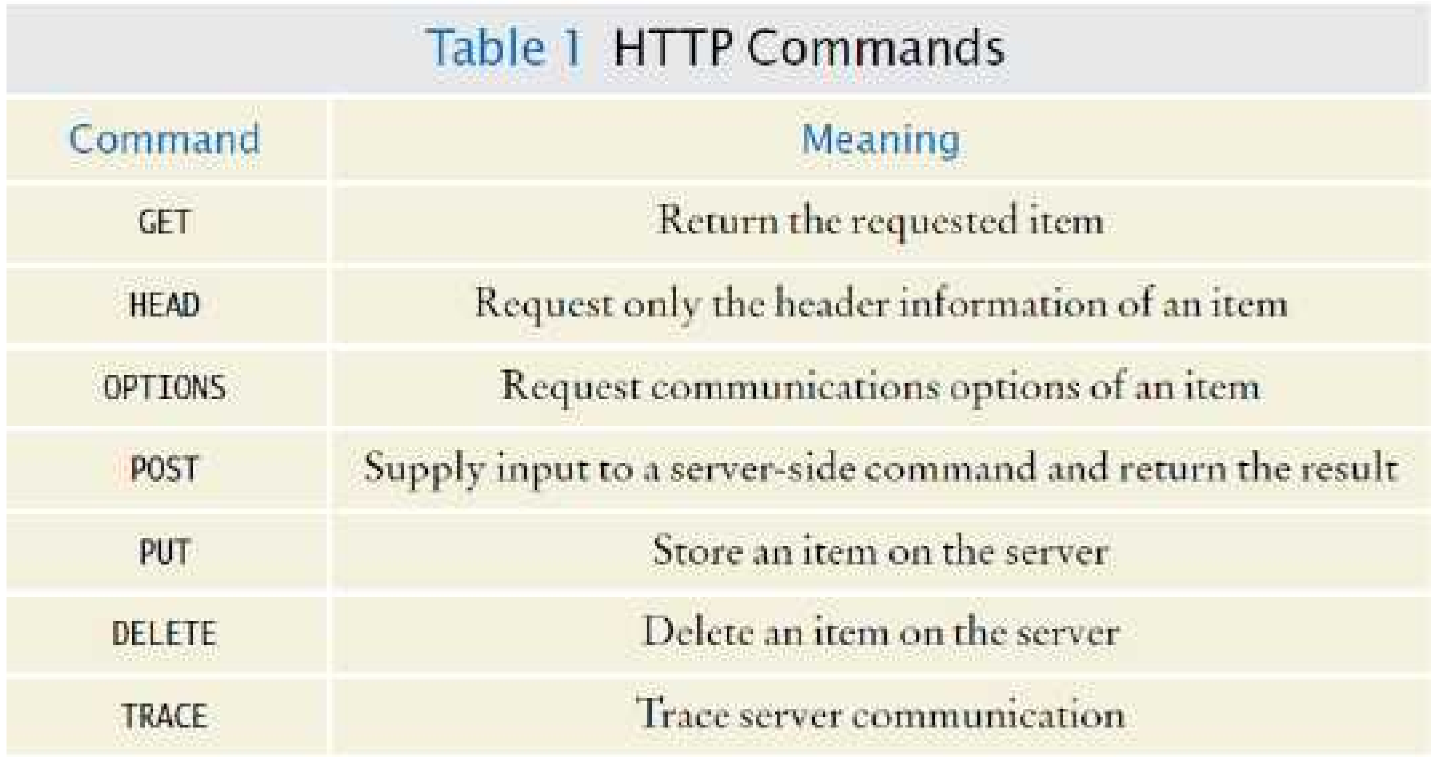
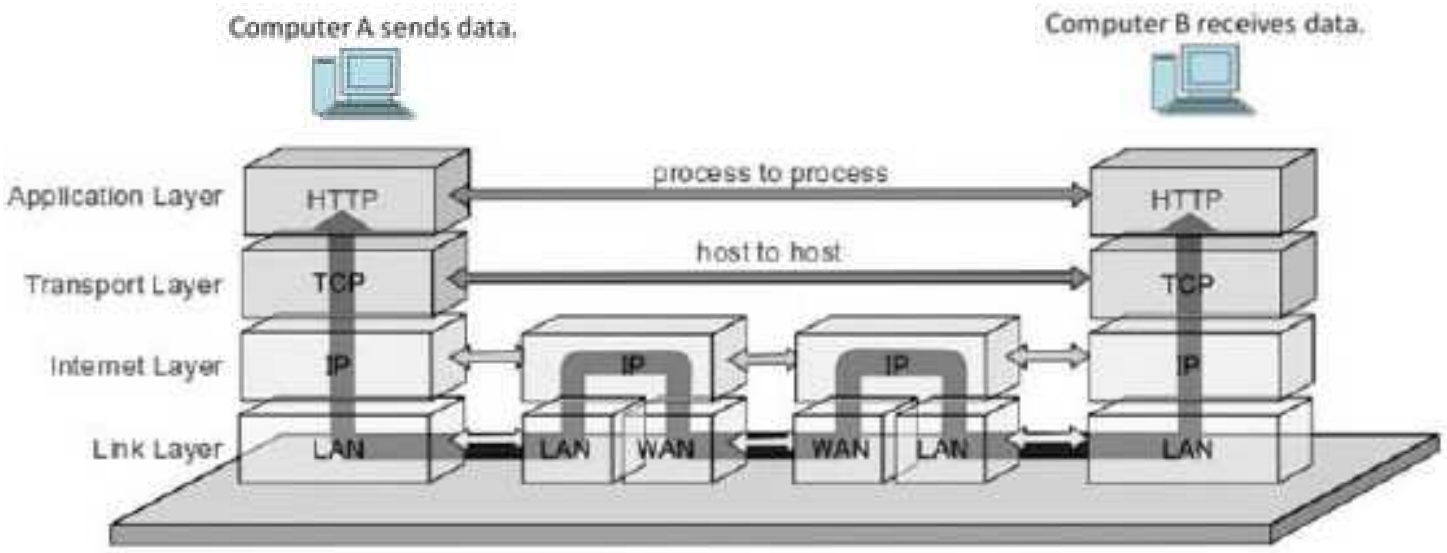


Table 1 HTTP Commands

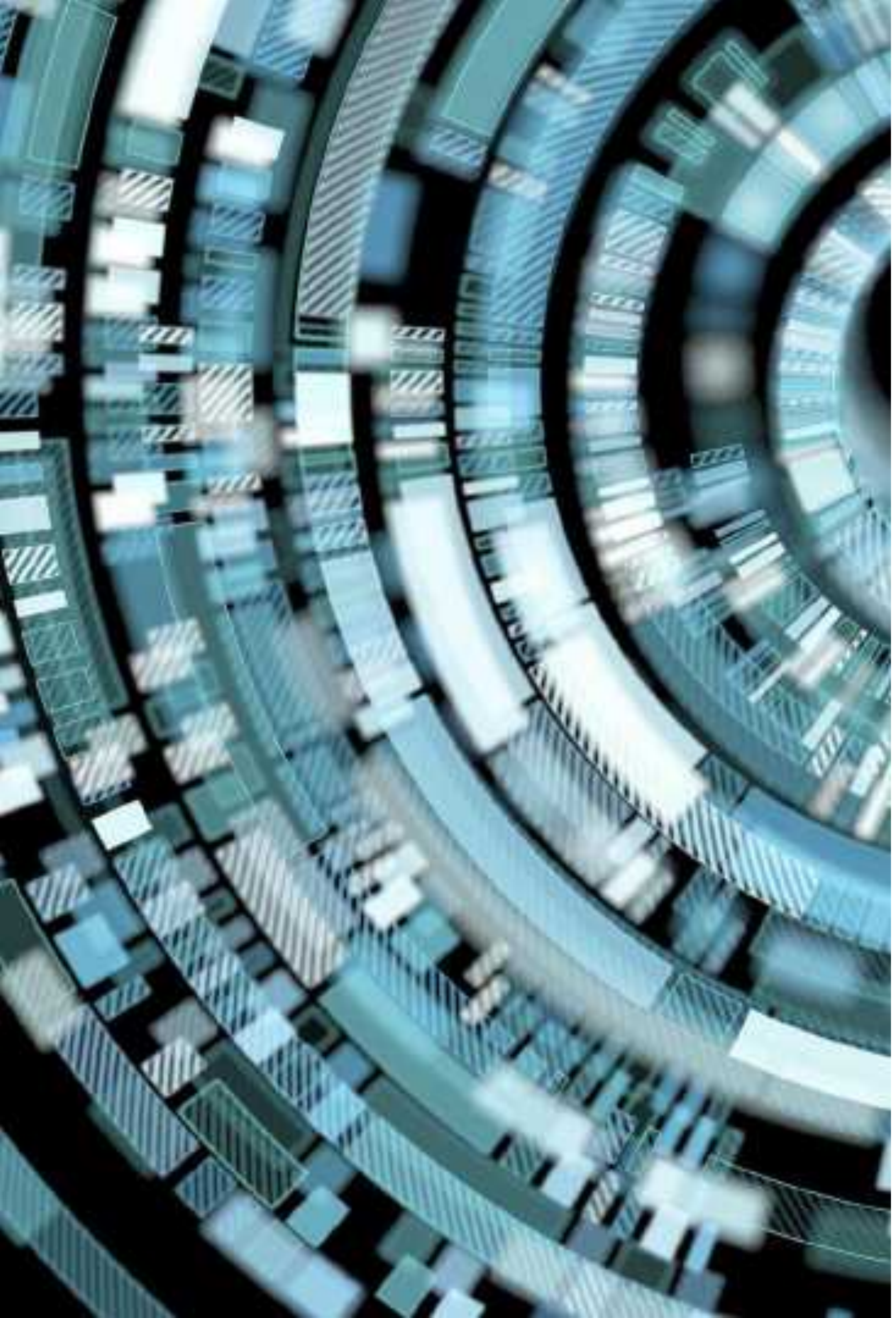
| Command | Meaning |
|---|---|
| GET | Return the requested item |
| HEAD | Request only the header information of an item |
| OPTIONS | Request communications options of an item |
| POST | Supply input to a server-side command and return the result |
| PUT | Store an item on the server |
| DELETE | Delete an item on the server |
| TRACE | Trace server communication |

# Transport Layer Protocols

- TCP (Transmission Control Protocol)
  - TCP provides a reliable, point-to-point communication channel for clients and servers to communicate over the Internet
  - TCP is the protocol used most on top of IP, we often referred to as TCP/IP
- UDP (User Datagram Protocol)
  - contains a minimum amount of communication mechanisms (no acknowledgement, unreliable)



https://www.elprocus.com/tcp-ip-protocol-architecture-and-its-layers/

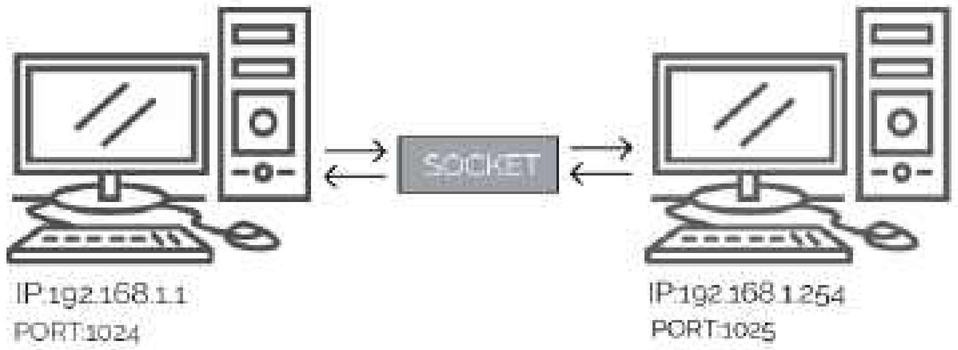


# Lecture 9

- Network Basics
- Protocols
- Socket Programming

## Socket

- To communicate over TCP, a client program and a server program establish a connection to one another
- Each program binds a socket to its end of the connection
- A socket is one endpoint of a two-way communication link between two programs running on the network.
    - Endpoint: IP address + Port number
- To communicate, the client and the server each reads from and writes to the socket bound to the connection.

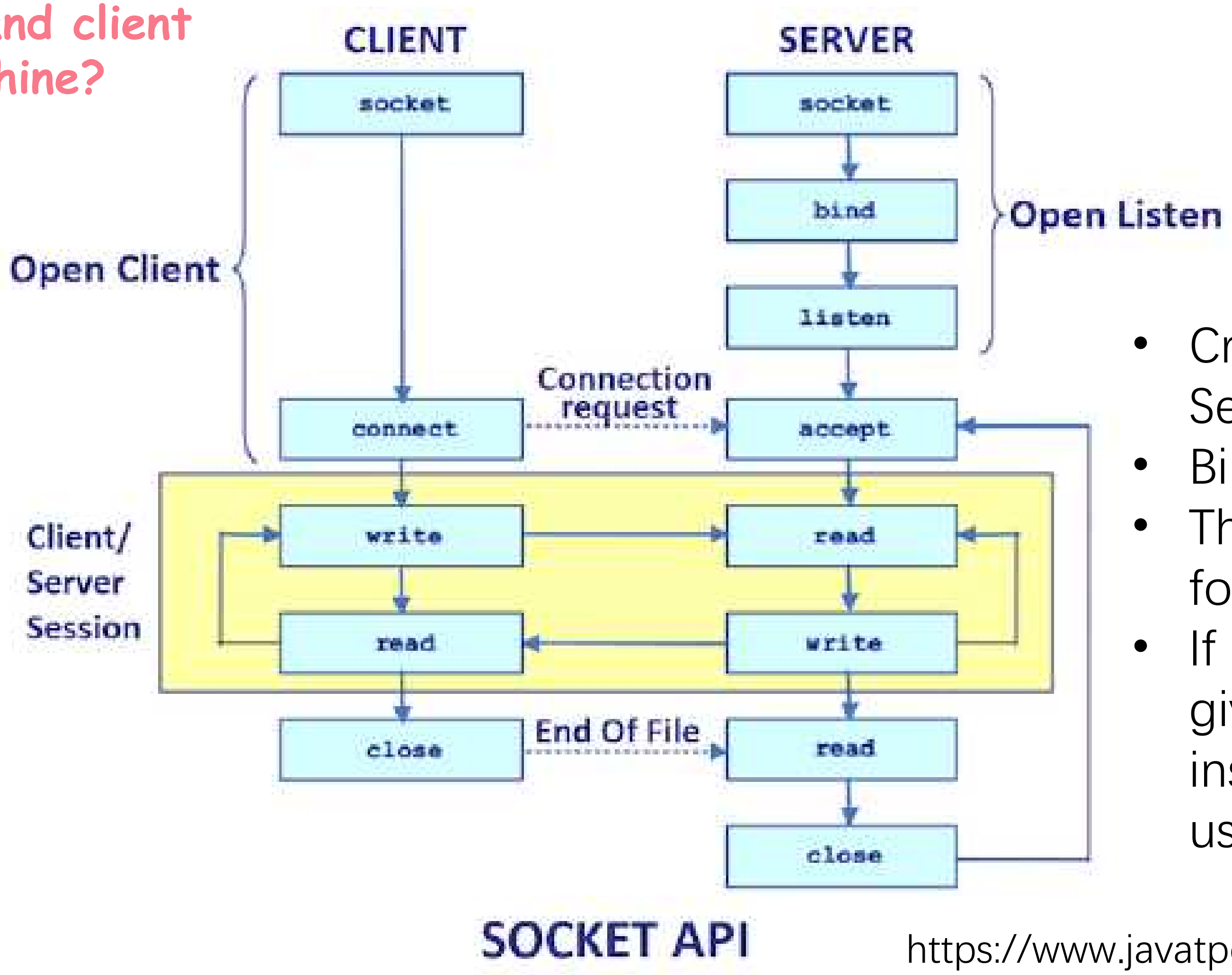


https://examradar.com/java-networking-network-basics-socket-overview/

```
Socket s = new Socket("www.serverip.com", 1234);
```

```
ServerSocket ss = new ServerSocket(1234);
Socket s = ss.accept();
```

What if the server and client run on the same machine?



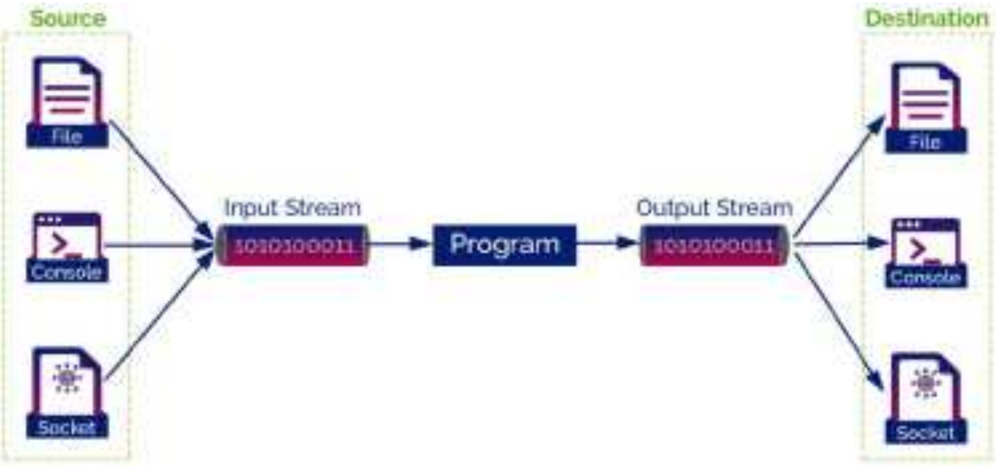
- Create an instance of Socket.
- Pass the IP address or hostname of the Server and a port number
- Establish the connection and use Socket to read and write.

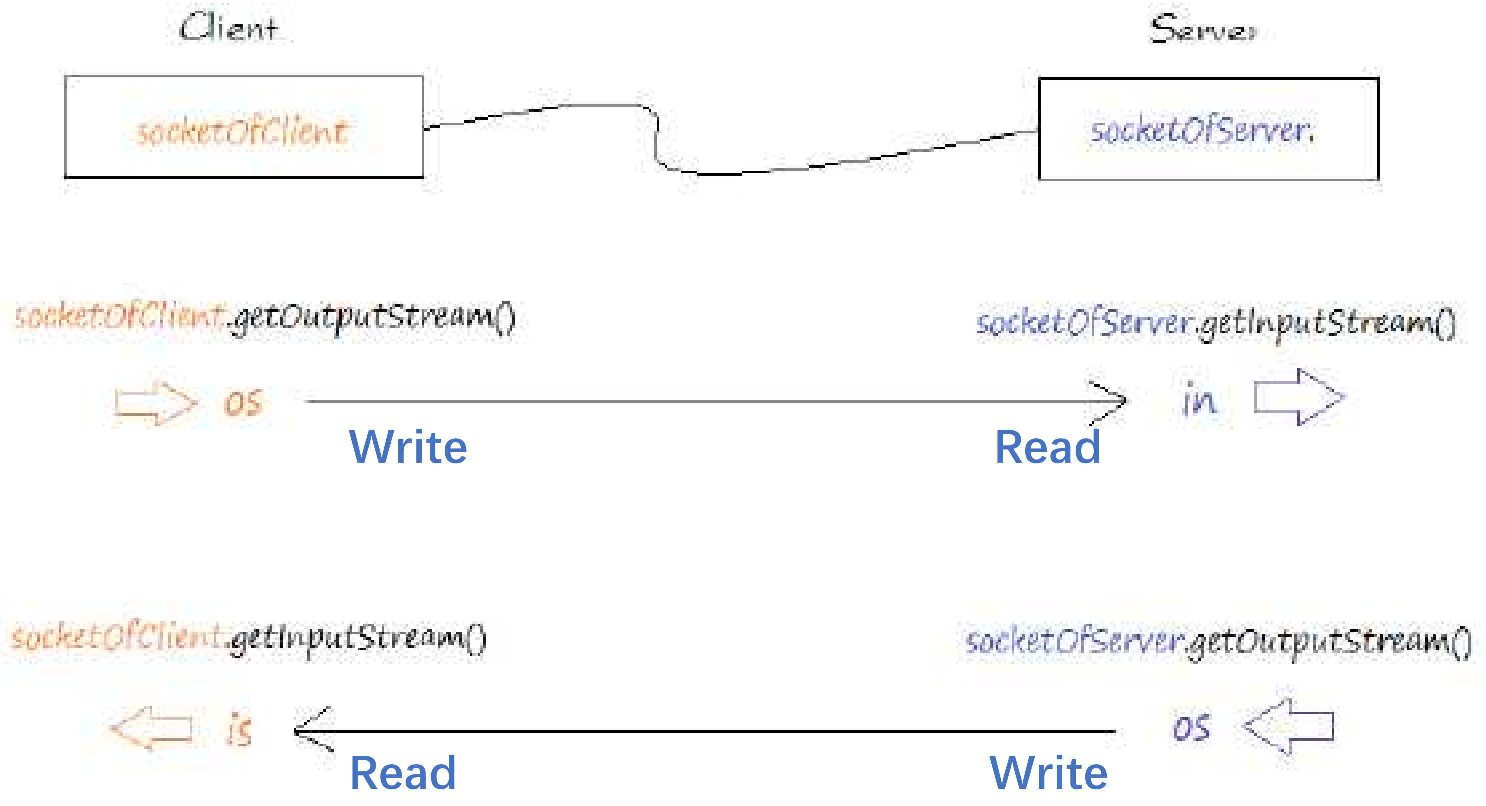
- Create an instance of ServerSocket.
- Bind to 1234 port number
- The accept() method waits for the client.
- If clients connects to the given port, return an instance of Socket that is used for reading and writing.

https://www.javatpoint.com/socket-programming

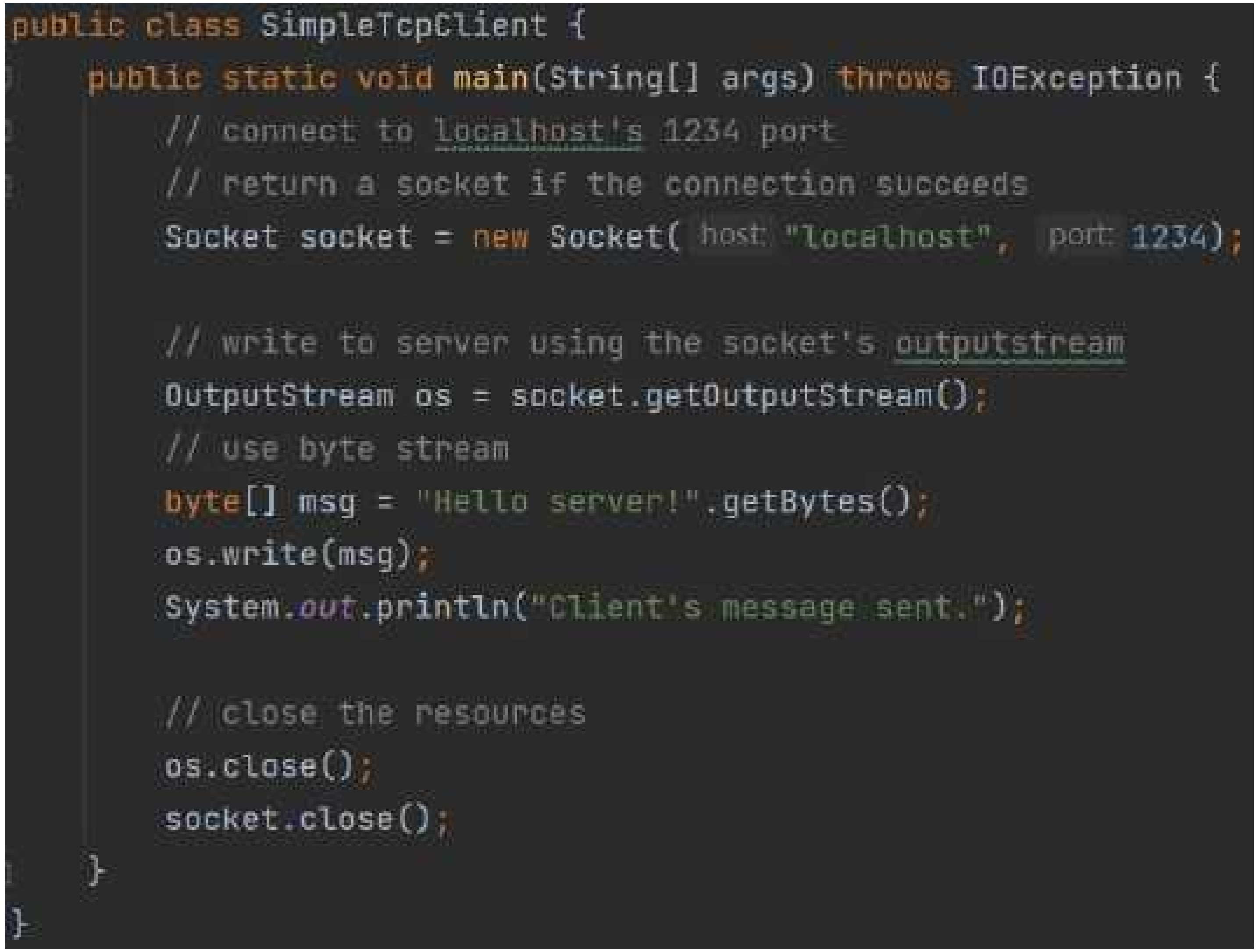
## Reading from and Writing to a Socket

- After establishing the connection, we can use socket.getInputStream() and socket.getOutputStream() for both the client and the server



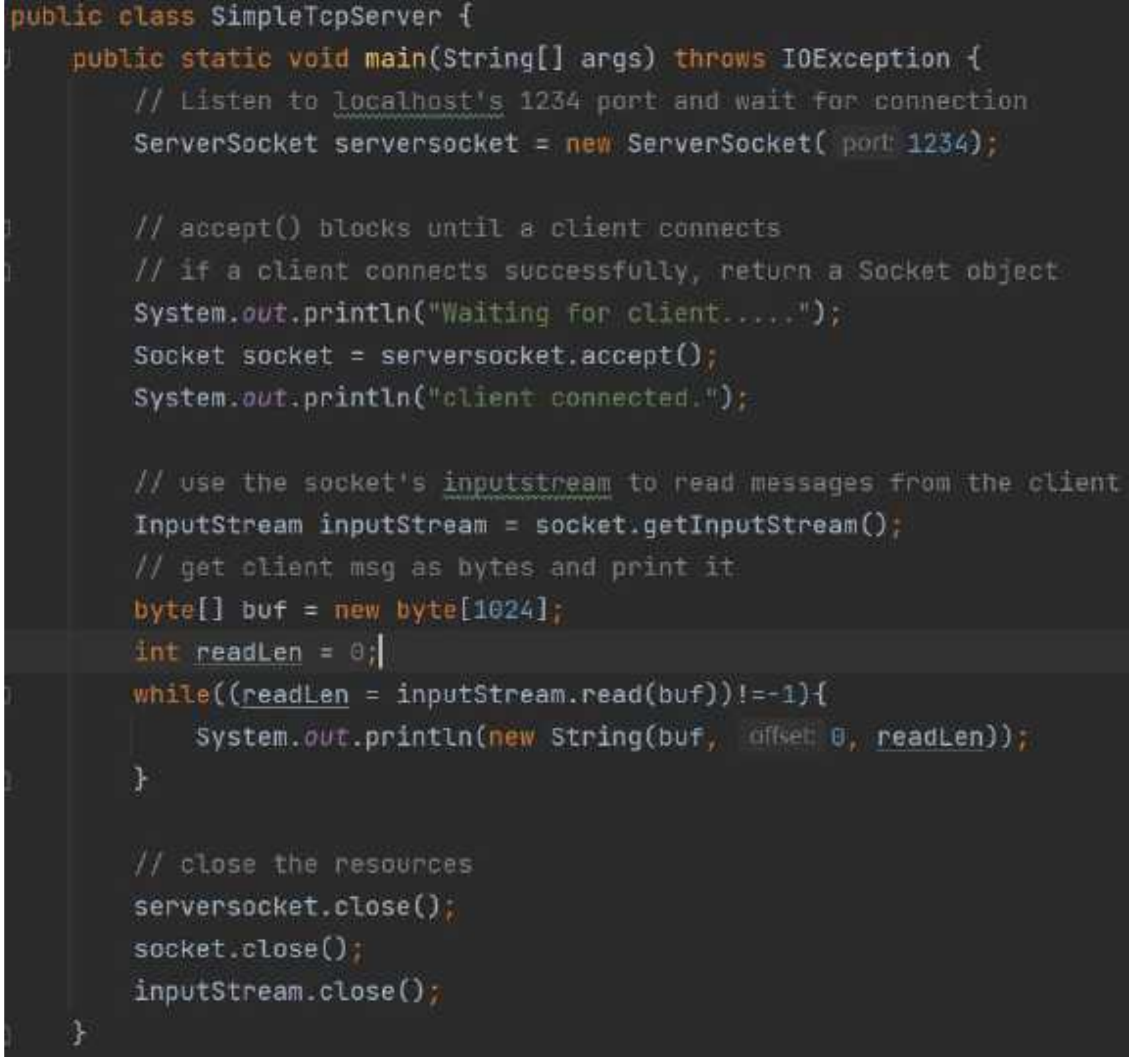

Client
Server
socketOfClient
socketOfServer.
socketOfClient.getOutputStream()
socketOfServer.getInputStream()
os
in
Write
Read
socketOfClient.getInputStream()
socketOfServer.getOutputStream()
is
os
Read
Write

## A Toy Example: Client



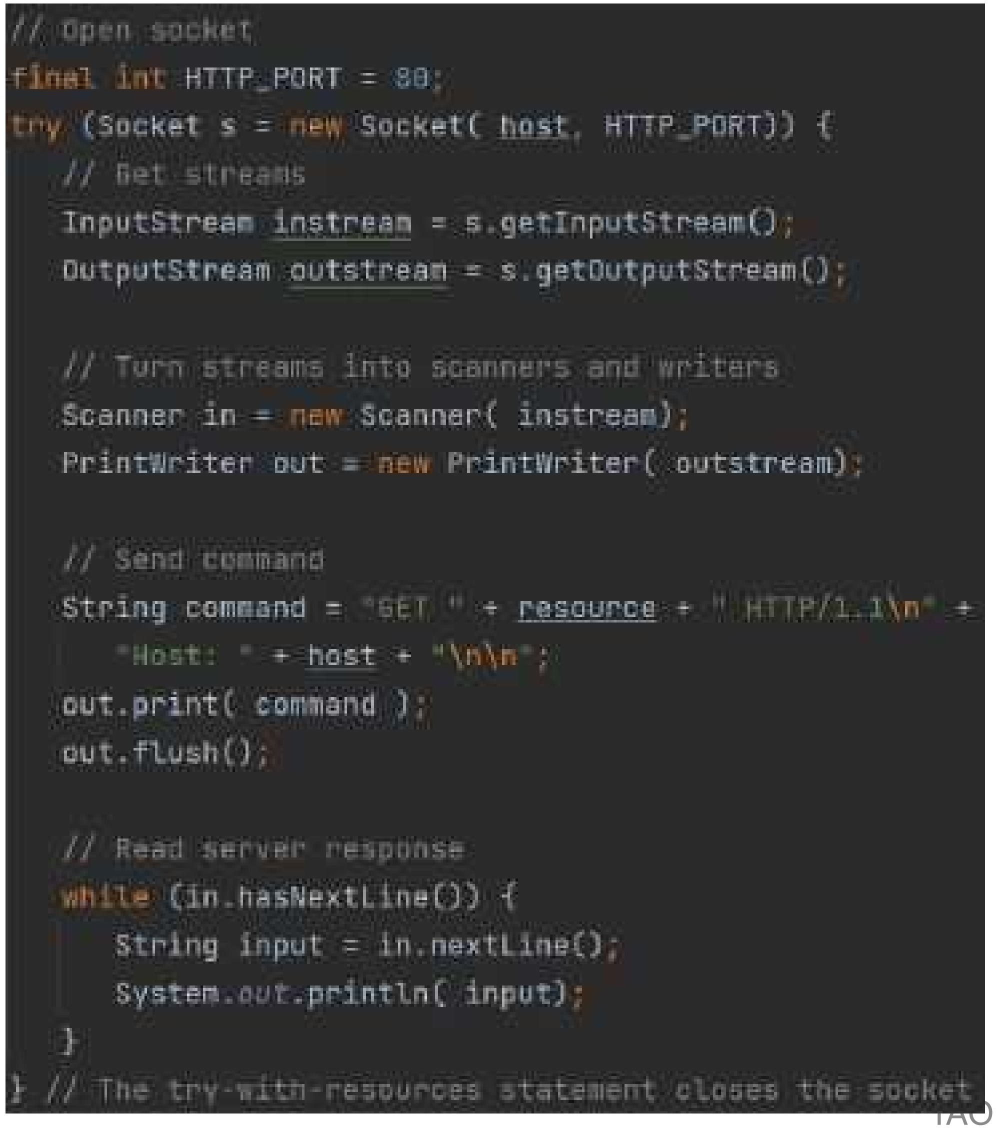
```
public class SimpleTcpClient {
    public static void main(String[] args) throws IOException {
        // connect to localhost's 1234 port
        // return a socket if the connection succeeds
        Socket socket = new Socket( host: "localhost",  port: 1234);

        // write to server using the socket's outputstream
        OutputStream os = socket.getOutputStream();
        // use byte stream
        byte[] msg = "Hello server!".getBytes();
        os.write(msg);
        System.out.println("Client's message sent.");

        // close the resources
        os.close();
        socket.close();
    }
}
```

## A Toy Example: Server



```java
public class SimpleTcpServer {
    public static void main(String[] args) throws IOException {
        // Listen to localhost's 1234 port and wait for connection
        ServerSocket serversocket = new ServerSocket( port: 1234);

        // accept() blocks until a client connects
        // if a client connects successfully, return a Socket object
        System.out.println("Waiting for client.....");
        Socket socket = serversocket.accept();
        System.out.println("client connected.");

        // use the socket's inputstream to read messages from the client
        InputStream inputStream = socket.getInputStream();
        // get client msg as bytes and print it
        byte[] buf = new byte[1024];
        int readLen = 0;
        while((readLen = inputStream.read(buf))!=-1){
            System.out.println(new String(buf,  offset: 0, readLen));
        }

        // close the resources
        serversocket.close();
        socket.close();
        inputStream.close();
    }
```

# Example: Fetching a web page



```java
// Open socket
final int HTTP_PORT = 80;
try (Socket s = new Socket( host, HTTP_PORT)) {
    // Get streams
    InputStream instream = s.getInputStream();
    OutputStream outstream = s.getOutputStream();

    // Turn streams into scanners and writers
    Scanner in = new Scanner( instream);
    PrintWriter out = new PrintWriter( outstream);

    // Send command
    String command = "GET " + resource + " HTTP/1.1\n" +
        "Host: " + host + "\n\n";
    out.print( command );
    out.flush();

    // Read server response
    while (in.hasNextLine()) {
        String input = in.nextLine();
        System.out.println( input);
    }
} // The try-with-resources statement closes the socket
```

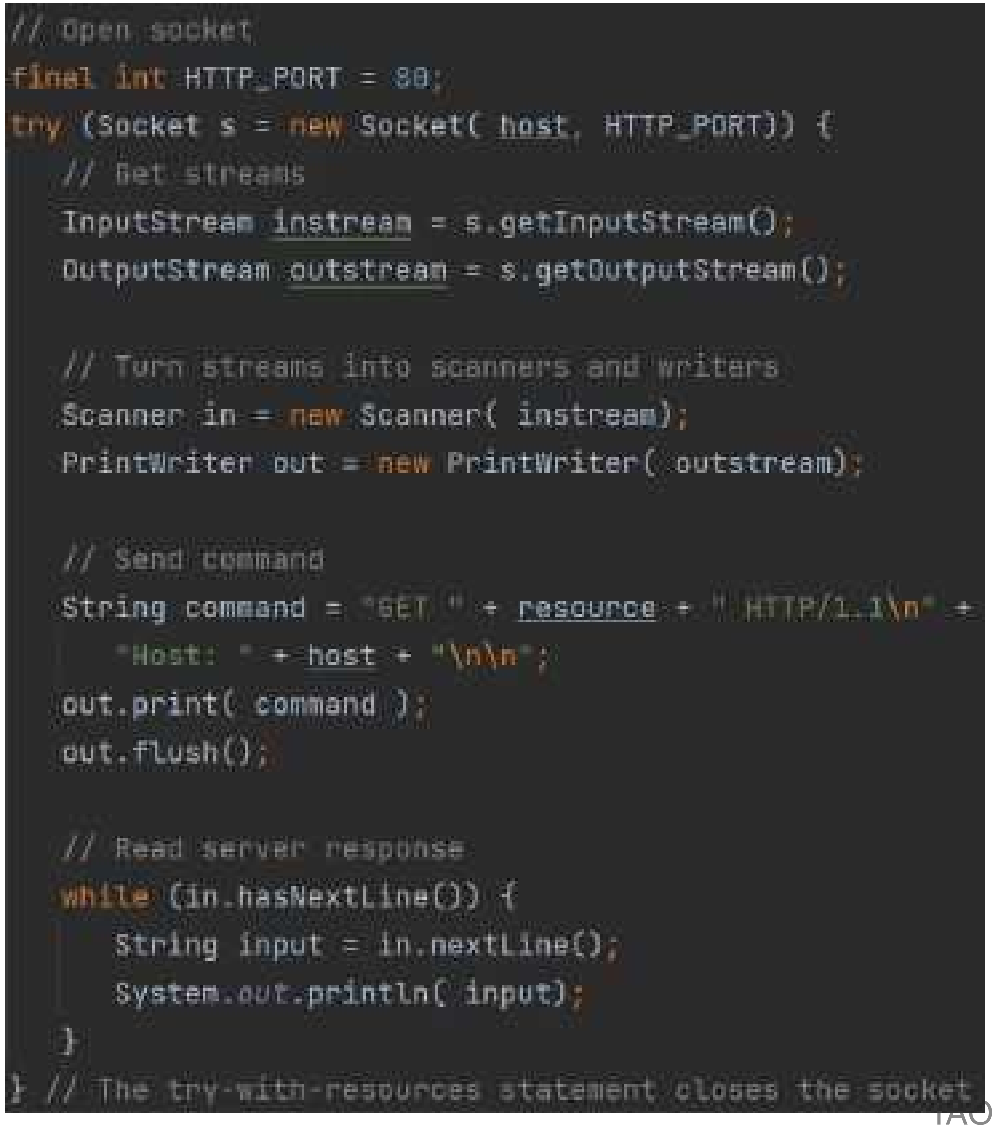
The client establish a Socket with the server. The socket constructor throws an UnknownHostException if it can't find the host.

InputStream and OutputStream classes are used for reading and writing bytes. If you want to communicate with the server by sending and receiving text, you should turn the streams into scanners and writers

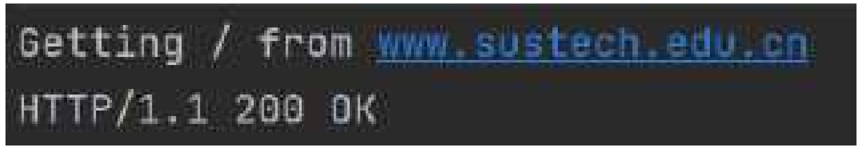
A print writer buffer characters. We need to flush the buffer manually so that the server get a complete request

Receive responses from the server

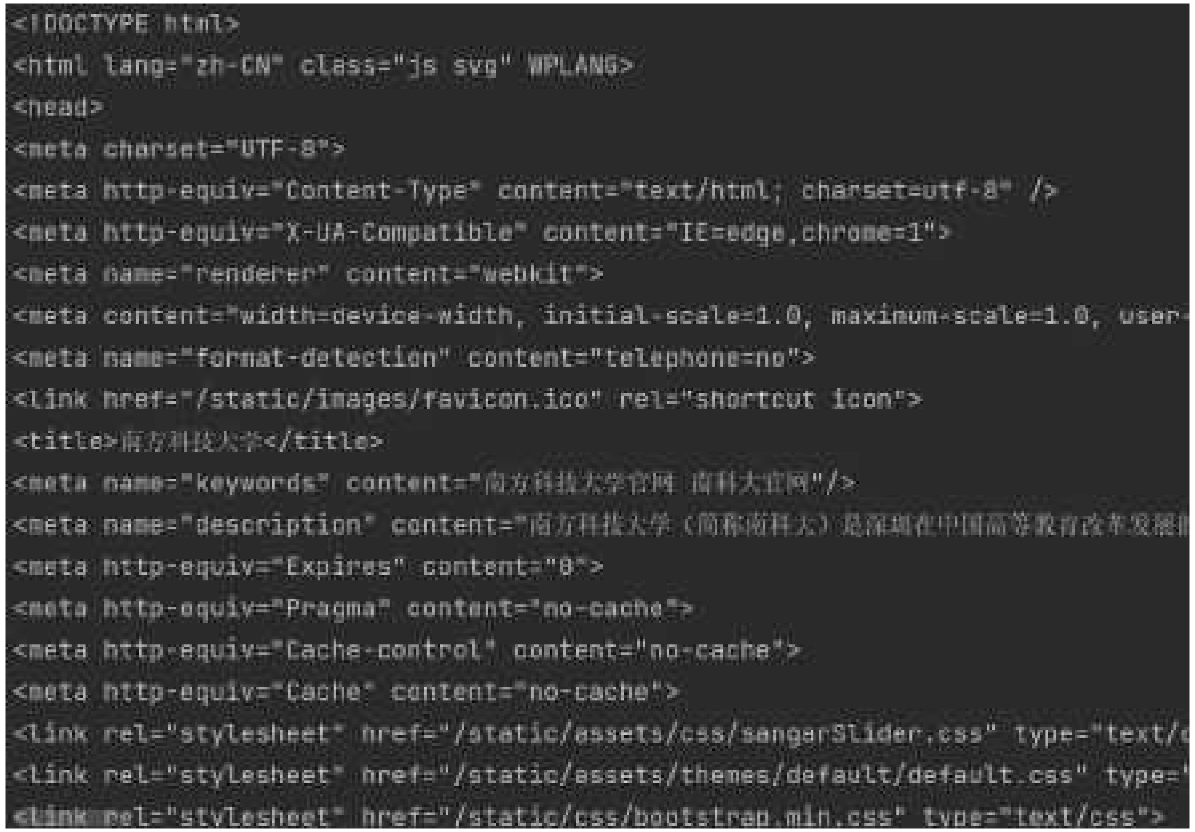
# Example: Fetching a web page



```java
// Open socket
final int HTTP_PORT = 80;
try (Socket s = new Socket( host, HTTP_PORT)) {
   // Get streams
   InputStream instream = s.getInputStream();
   OutputStream outstream = s.getOutputStream();

   // Turn streams into scanners and writers
   Scanner in = new Scanner( instream);
   PrintWriter out = new PrintWriter( outstream);

   // Send command
   String command = "GET " + resource + " HTTP/1.1\n" +
      "Host: " + host + "\n\n";
   out.print( command );
   out.flush();

   // Read server response
   while (in.hasNextLine()) {
      String input = in.nextLine();
      System.out.println( input);
   }
} // The try-with-resources statement closes the socket
```



```
Getting / from www.sustech.edu.cn
HTTP/1.1 200 OK
```



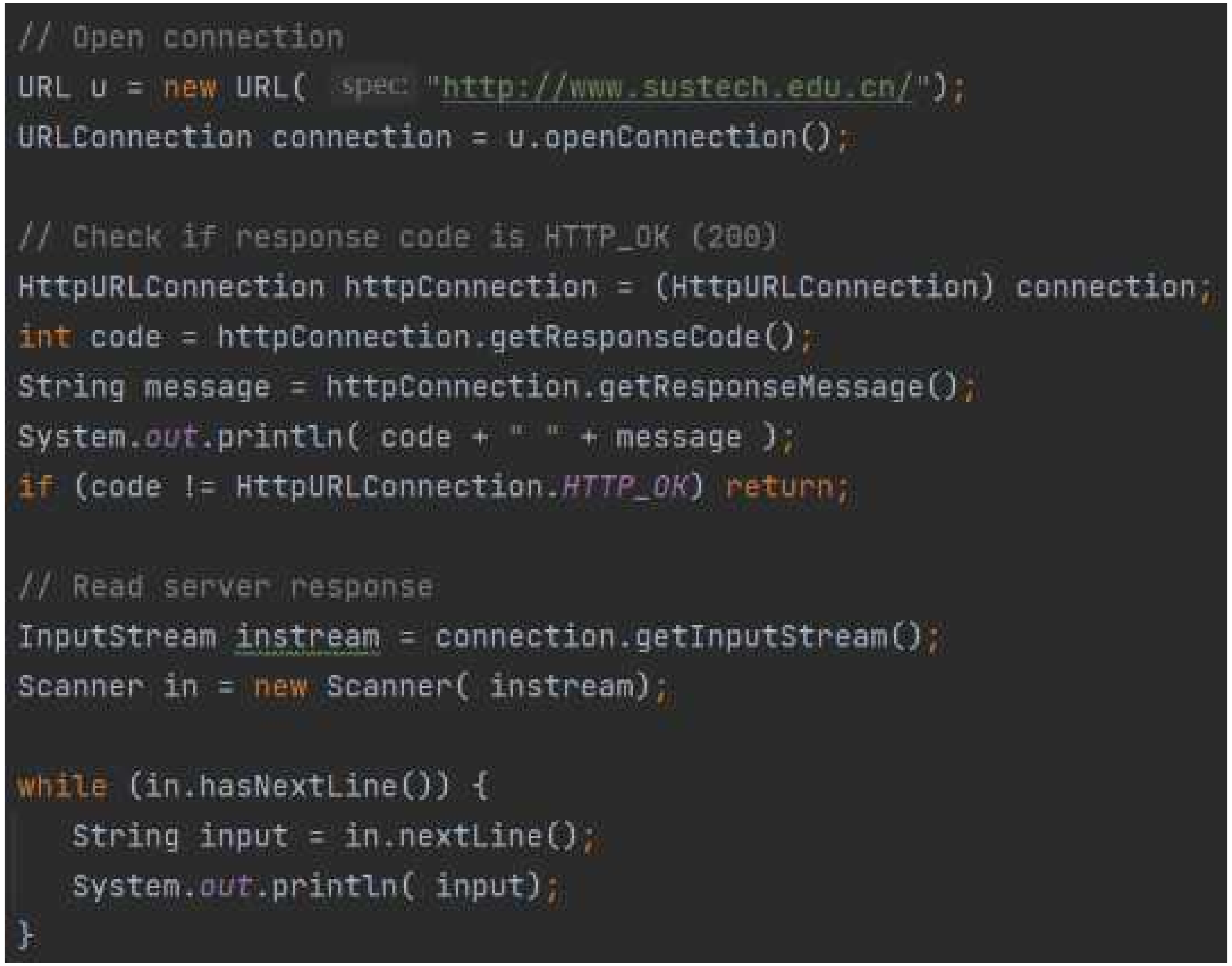
```
<!DOCTYPE html>
<html lang="zh-CN" class="js svg" WPLANG>
<head>
<meta charset="UTF-8">
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
<meta name="renderer" content="webkit">
<meta content="width=device-width, initial-scale=1.0, maximum-scale=1.0, user-
<meta name="format-detection" content="telephone=no">
<link href="/static/images/favicon.ico" rel="shortcut icon">
<title>南方科技大学</title>
<meta name="keywords" content="南方科技大学官网 南科大官网"/>
<meta name="description" content="南方科技大学（简称南科大）是深圳在中国高等教育改革发展
<meta http-equiv="Expires" content="0">
<meta http-equiv="Pragma" content="no-cache">
<meta http-equiv="Cache-control" content="no-cache">
<meta http-equiv="Cache" content="no-cache">
<link rel="stylesheet" href="/static/assets/css/sangarSlider.css" type="text/c
<link rel="stylesheet" href="/static/assets/themes/default/default.css" type="
<link rel="stylesheet" href="/static/css/bootstrap.min.css" type="text/css">
```

## URL Connections

- HTTP is such an important protocol, so Java contains a URLConnection class (java.net package), which provides convenient support for the HTTP
- The URLConnection class takes care of the socket connection, so you do not have to fuss with sockets when you want to retrieve from a web server.
- As an additional benefit, the URLConnection class can also handle FTP, the file transfer protocol.

## Example: Fetching a web page (easier way)

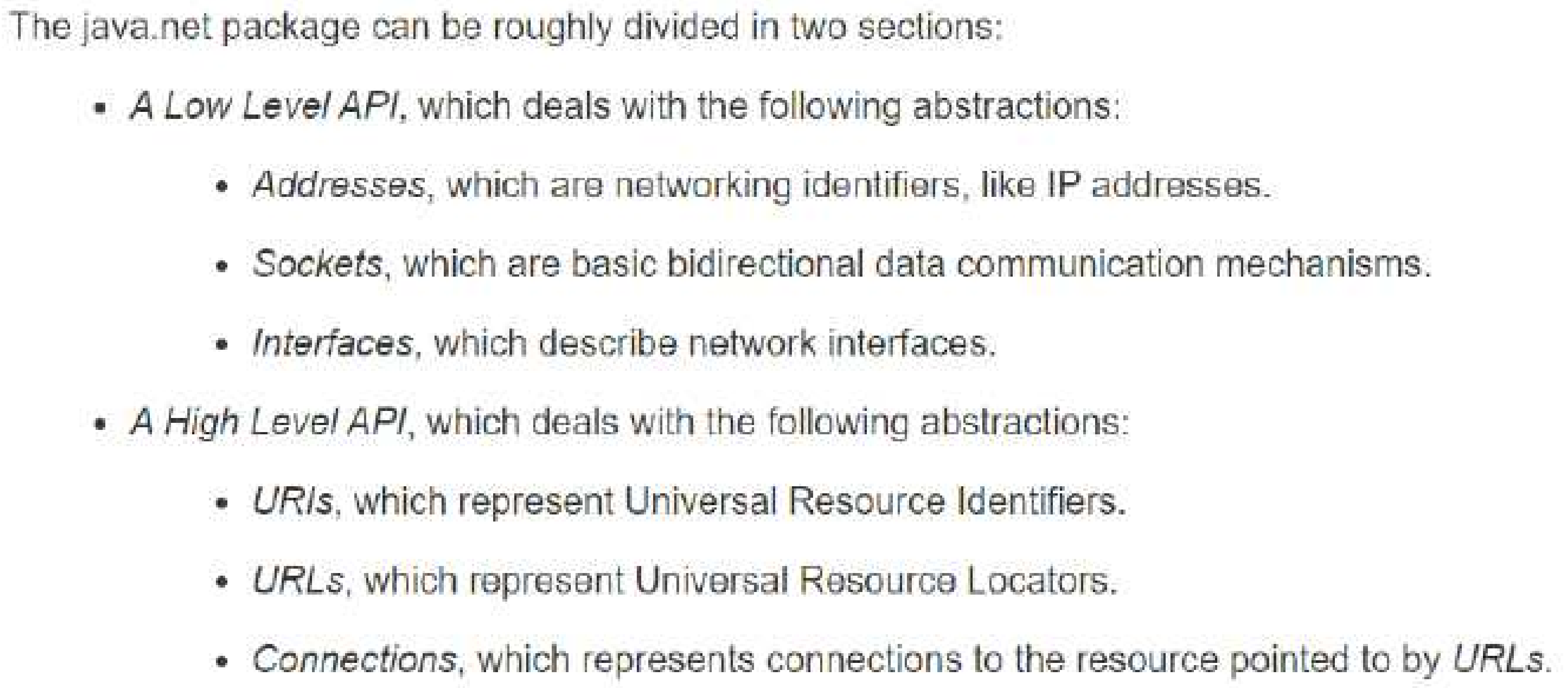
**Your may review the slides for RESTful APIs**



```
// Open connection
URL u = new URL( spec: "http://www.sustech.edu.cn/");
URLConnection connection = u.openConnection();

// Check if response code is HTTP_OK (200)
HttpURLConnection httpConnection = (HttpURLConnection) connection;
int code = httpConnection.getResponseCode();
String message = httpConnection.getResponseMessage();
System.out.println( code + " " + message );
if (code != HttpURLConnection.HTTP_OK) return;

// Read server response
InputStream instream = connection.getInputStream();
Scanner in = new Scanner( instream);

while (in.hasNextLine()) {
    String input = in.nextLine();
    System.out.println( input);
}
```

# java.net package

Provides the classes for implementing networking applications.

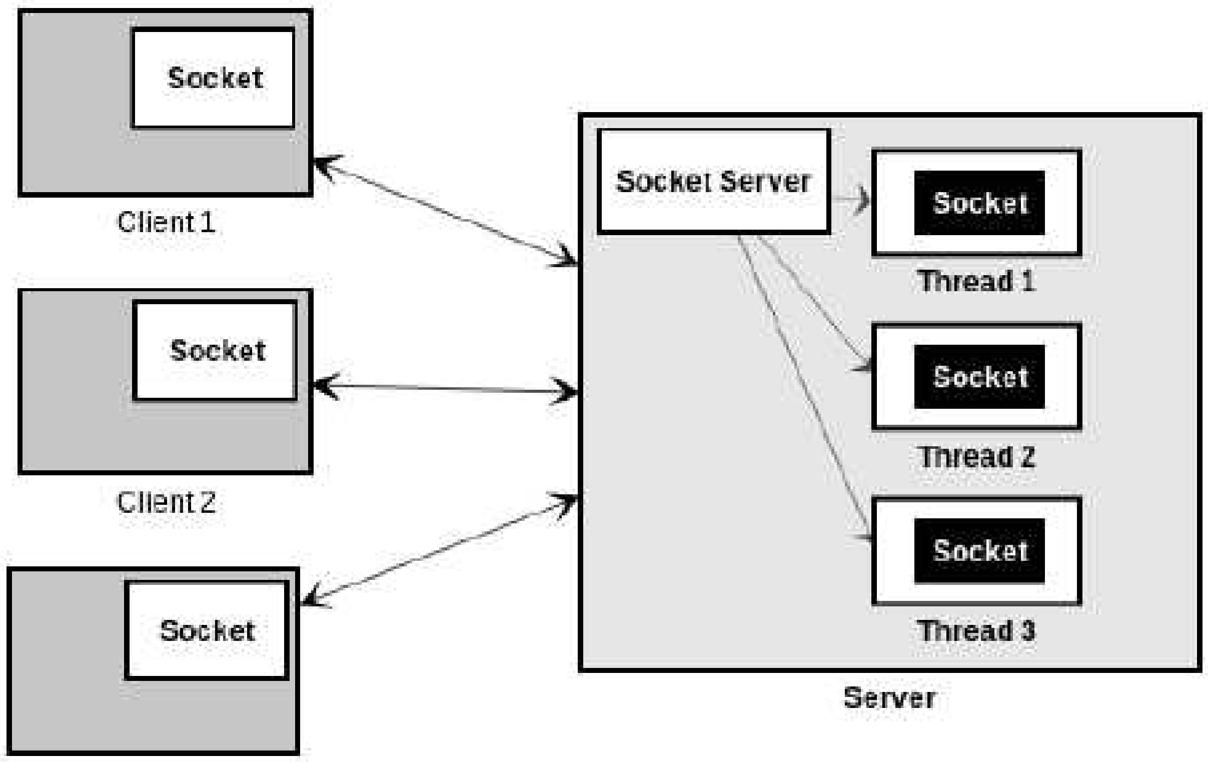


The java.net package can be roughly divided in two sections:

- *A Low Level API*, which deals with the following abstractions:
    - *Addresses*, which are networking identifiers, like IP addresses.
    - *Sockets*, which are basic bidirectional data communication mechanisms.
    - *Interfaces*, which describe network interfaces.
- *A High Level API*, which deals with the following abstractions:
    - *URIs*, which represent Universal Resource Identifiers.
    - *URLs*, which represent Universal Resource Locators.
    - *Connections*, which represents connections to the resource pointed to by *URLs*.

https://docs.oracle.com/javase/7/docs/api/java/net/package-summary.html#package_description

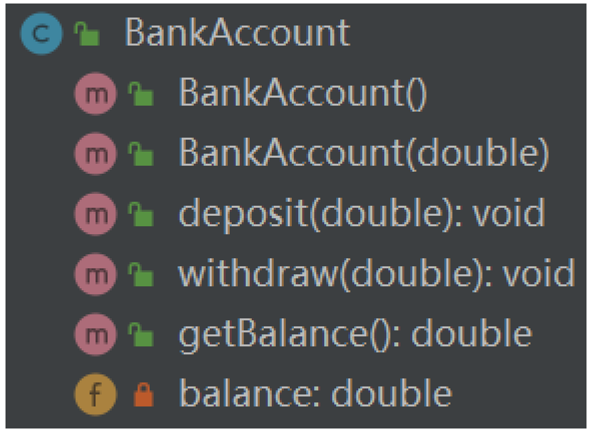
## Multiple Clients



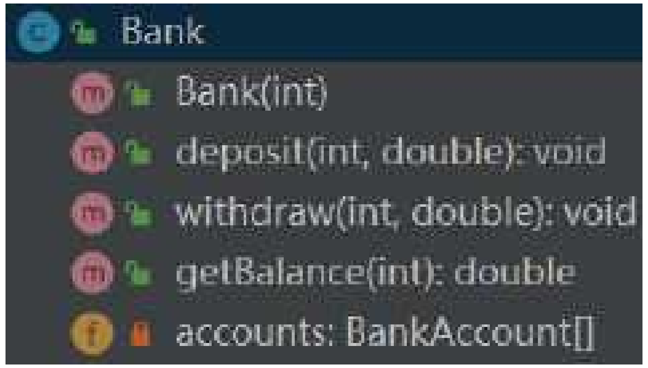
- We want our client-server architecture to support multiple clients at the same time
- We could use threads on server side: whenever a client request comes, a separate thread is assigned for handling each request

# Case Study: Banking Service

- A bank has multiple bank accounts
- A bank server provides the banking service
- A client could use the banking service to deposit, withdraw, and get balance from a specific account



```
BankAccount
    BankAccount()
    BankAccount(double)
    deposit(double): void
    withdraw(double): void
    getBalance(): double
    balance: double
```



```
Bank
    Bank(int)
    deposit(int, double): void
    withdraw(int, double): void
    getBalance(int): double
    accounts: BankAccount[]
```

deposit() and withdraw() are properly synchronized

# Banking Service Protocol

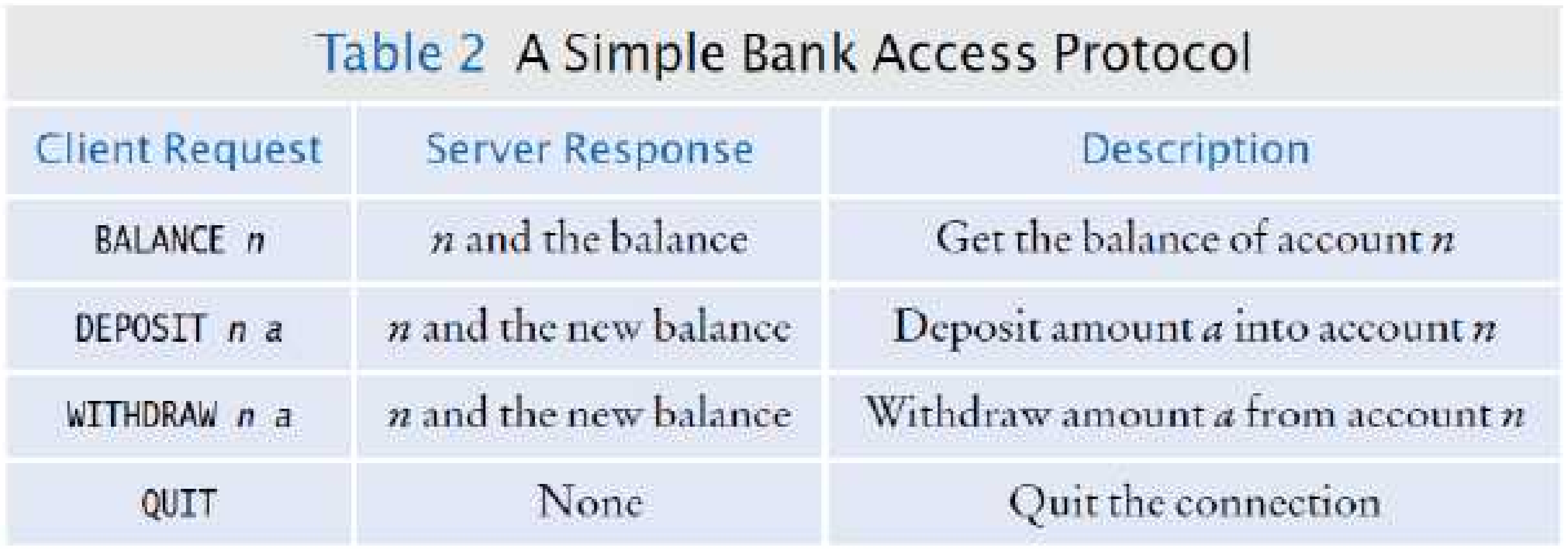
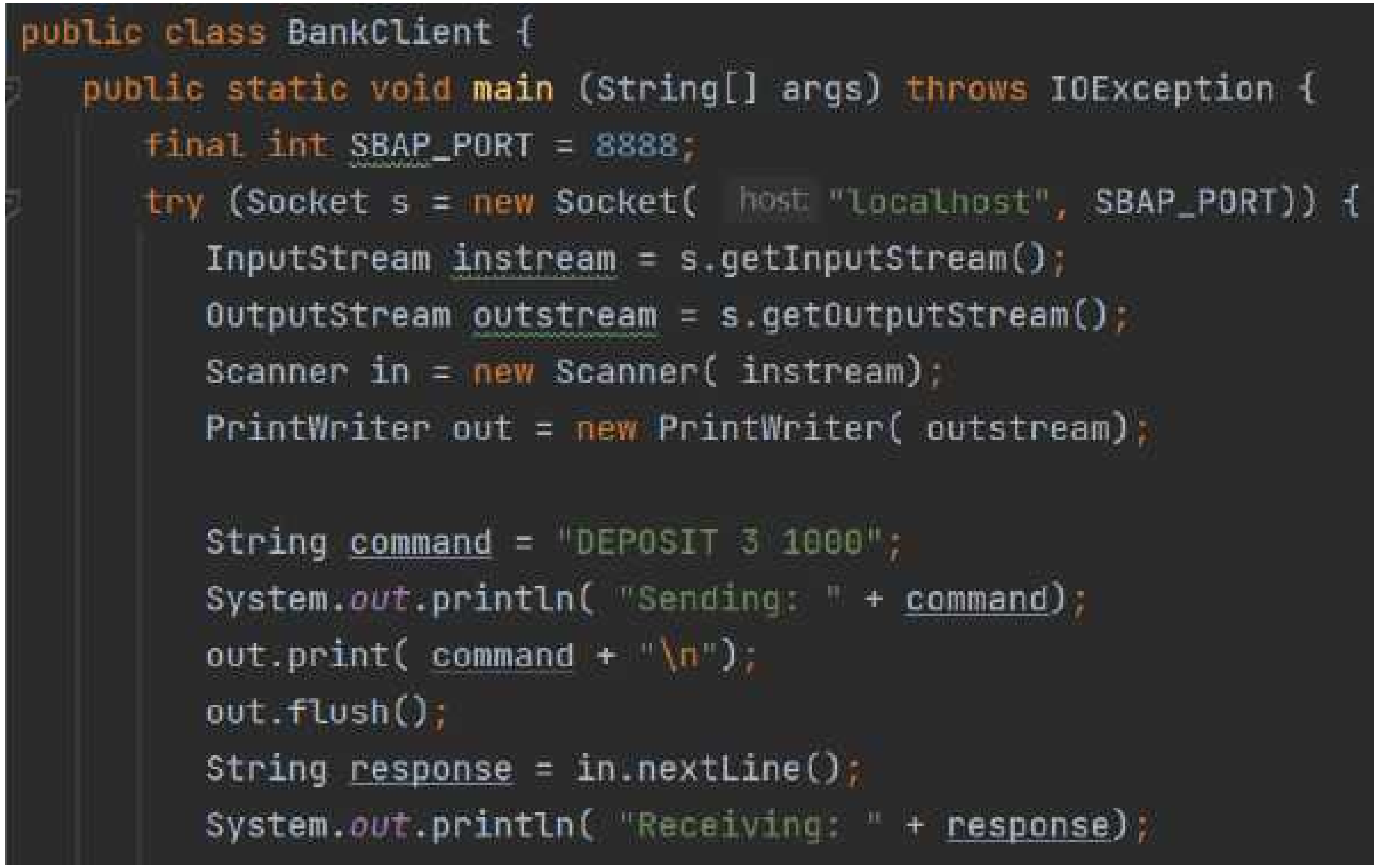


Table 2 A Simple Bank Access Protocol

| Client Request | Server Response | Description |
|---|---|---|
| BALANCE *n* | *n* and the balance | Get the balance of account *n* |
| DEPOSIT *n a* | *n* and the new balance | Deposit amount *a* into account *n* |
| WITHDRAW *n a* | *n* and the new balance | Withdraw amount *a* from account *n* |
| QUIT | None | Quit the connection |

Whenever you develop a server application, you need to specify some application-level protocol that clients can use to interact with the server

# Bank Client

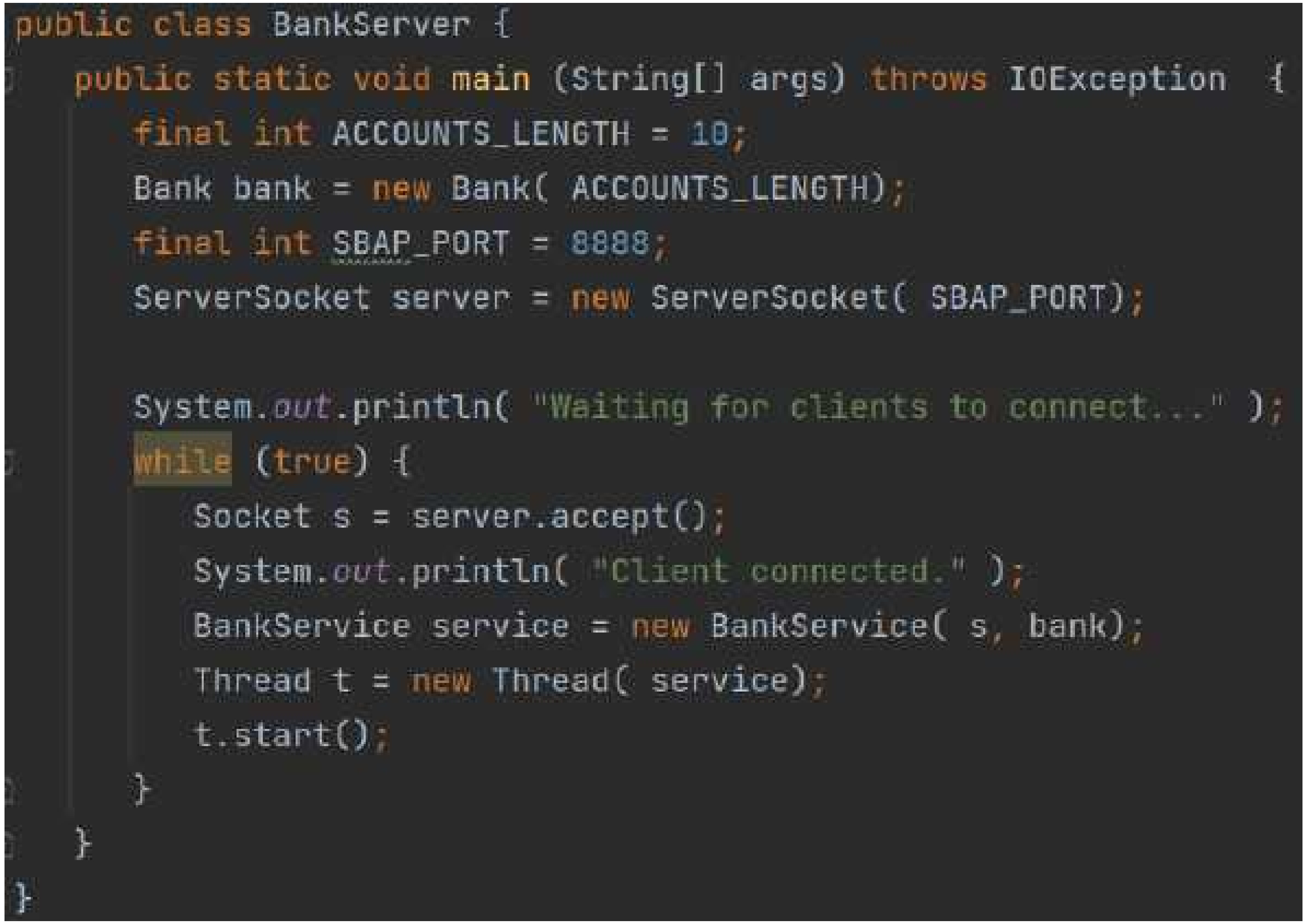


```
public class BankClient {
    public static void main (String[] args) throws IOException {
        final int SBAP_PORT = 8888;
        try (Socket s = new Socket( host: "localhost", SBAP_PORT)) {
            InputStream instream = s.getInputStream();
            OutputStream outstream = s.getOutputStream();
            Scanner in = new Scanner( instream);
            PrintWriter out = new PrintWriter( outstream);

            String command = "DEPOSIT 3 1000";
            System.out.println( "Sending: " + command);
            out.print( command + "\n");
            out.flush();
            String response = in.nextLine();
            System.out.println( "Receiving: " + response);
```

**To communicate with the server by sending and receiving text, you could turn the streams into scanners and writers**

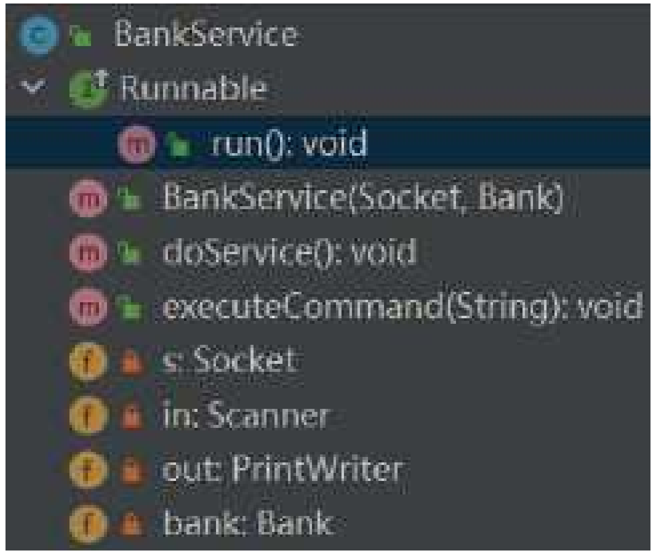
**The flush method empties the buffer and forwards all waiting characters to the destination.**

## Bank Server

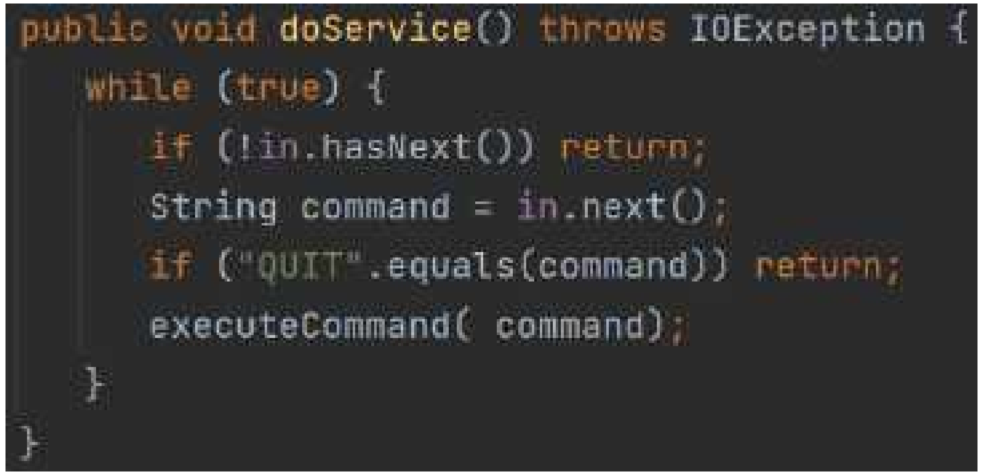


```java
public class BankServer {
    public static void main (String[] args) throws IOException  {
        final int ACCOUNTS_LENGTH = 10;
        Bank bank = new Bank( ACCOUNTS_LENGTH);
        final int SBAP_PORT = 8888;
        ServerSocket server = new ServerSocket( SBAP_PORT);

        System.out.println( "Waiting for clients to connect..." );
        while (true) {
            Socket s = server.accept();
            System.out.println( "Client connected." );
            BankService service = new BankService( s, bank);
            Thread t = new Thread( service);
            t.start();
        }
    }
}
```

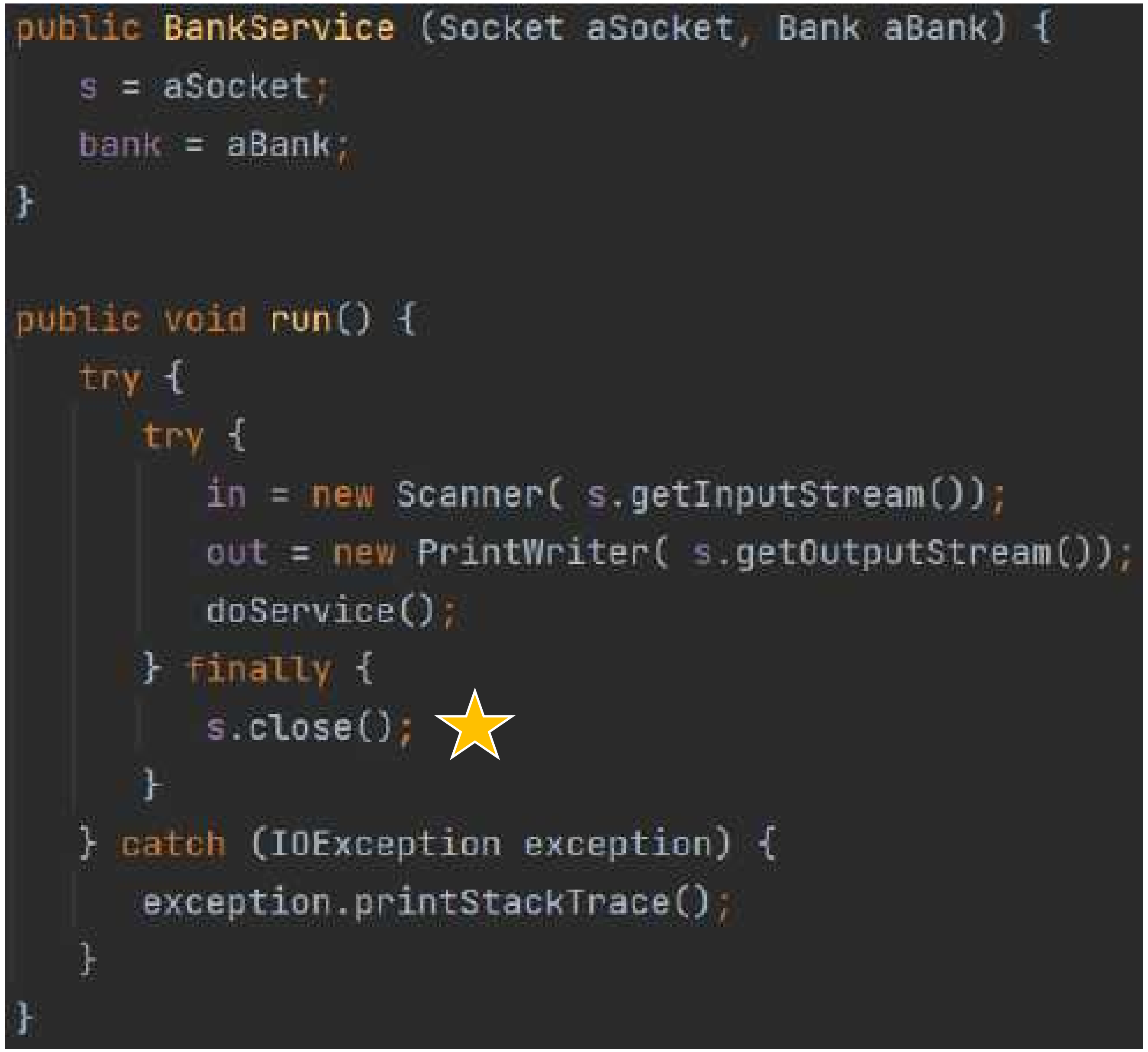
## Bank Service



```
BankService
  Runnable
    run(): void
  BankService(Socket, Bank)
  doService(): void
  executeCommand(String): void
  s: Socket
  in: Scanner
  out: PrintWriter
  bank: Bank
```



```java
public void doService() throws IOException {
    while (true) {
        if (!in.hasNext()) return;
        String command = in.next();
        if ("QUIT".equals(command)) return;
        executeCommand( command);
    }
}
```



```java
public BankService (Socket aSocket, Bank aBank) {
    s = aSocket;
    bank = aBank;
}

public void run() {
    try {
        try {
            in = new Scanner( s.getInputStream());
            out = new PrintWriter( s.getOutputStream());
            doService();
        } finally {
            s.close(); ★
        }
    } catch (IOException exception) {
        exception.printStackTrace();
    }
}
```

## Bank Service

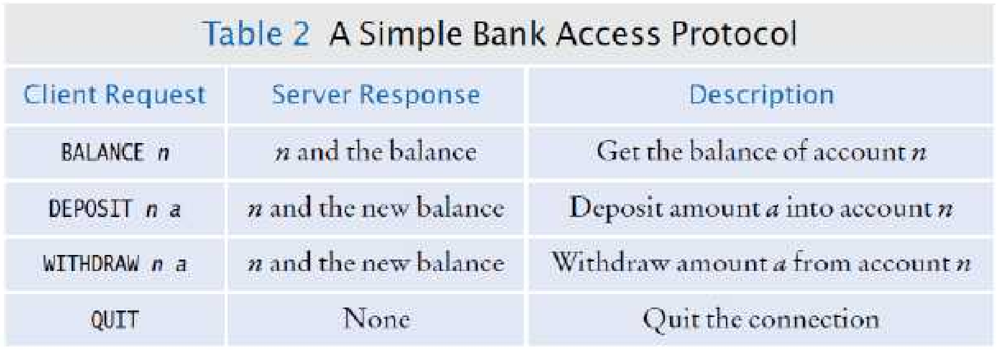
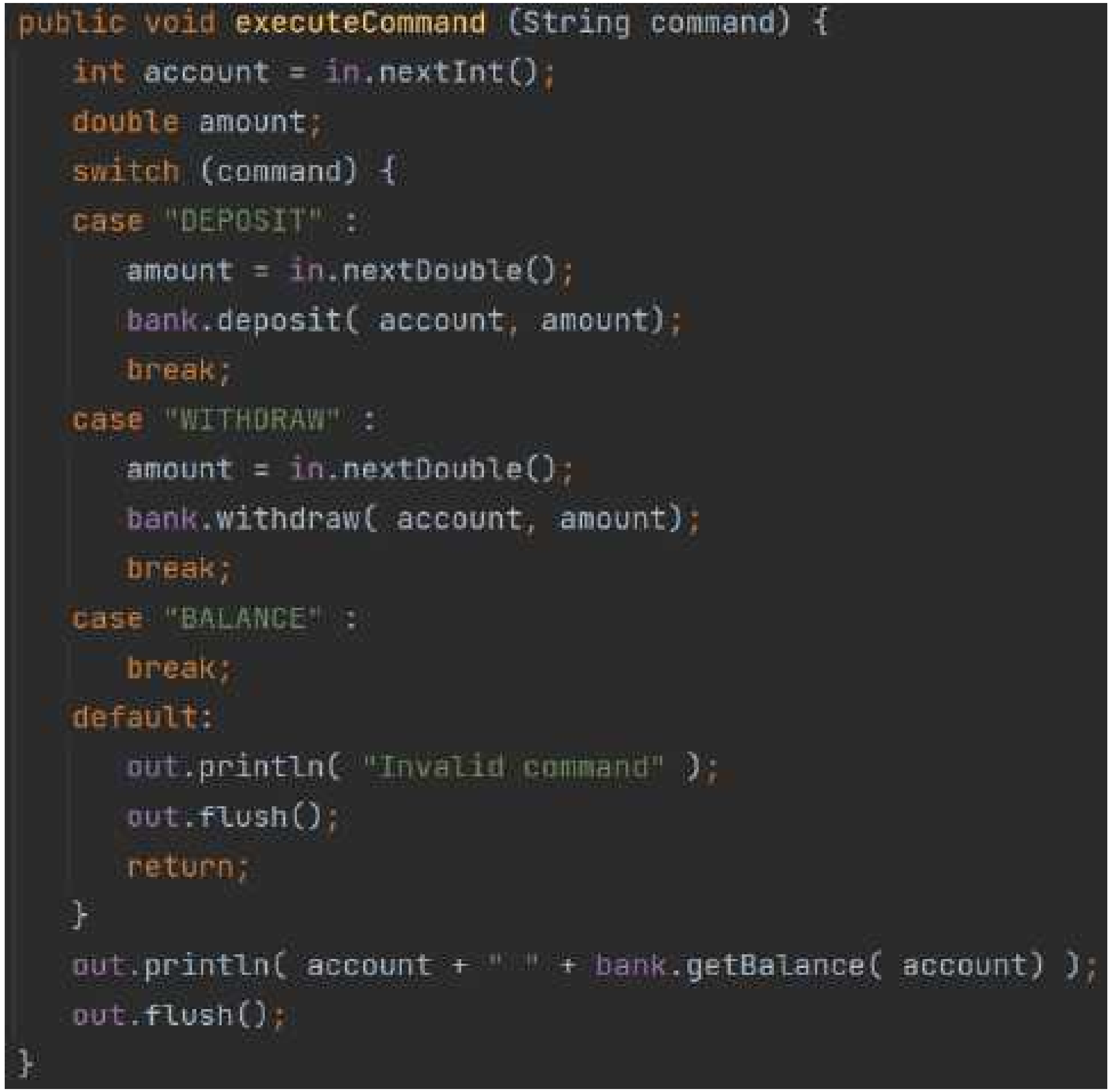


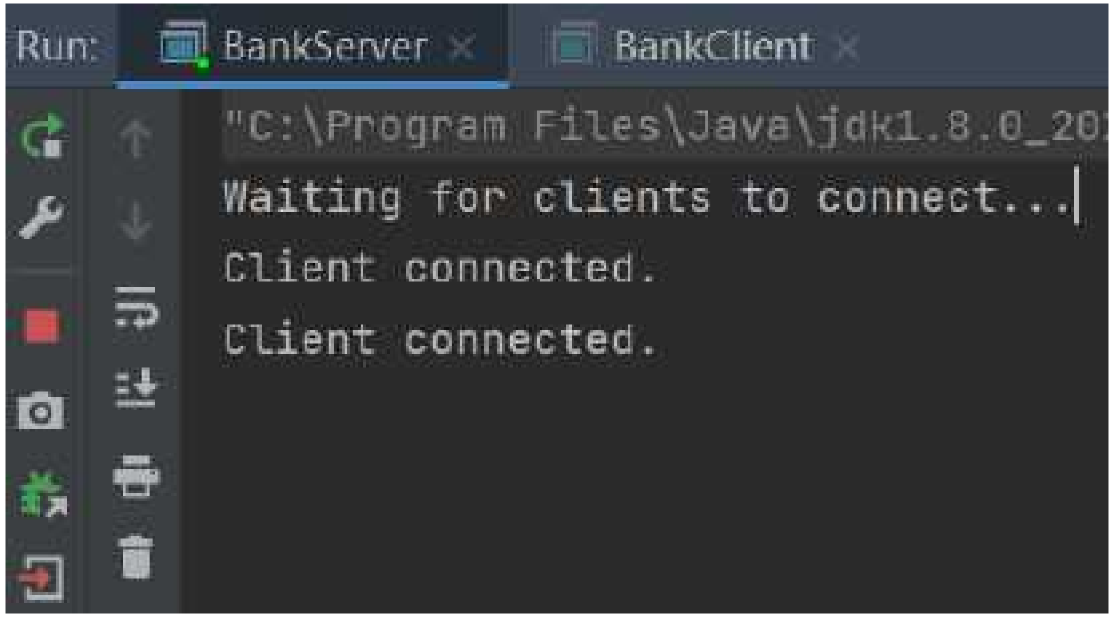
Table 2 A Simple Bank Access Protocol

| Client Request | Server Response | Description |
|---|---|---|
| `BALANCE` *n* | *n* and the balance | Get the balance of account *n* |
| `DEPOSIT` *n a* | *n* and the new balance | Deposit amount *a* into account *n* |
| `WITHDRAW` *n a* | *n* and the new balance | Withdraw amount *a* from account *n* |
| `QUIT` | None | Quit the connection |



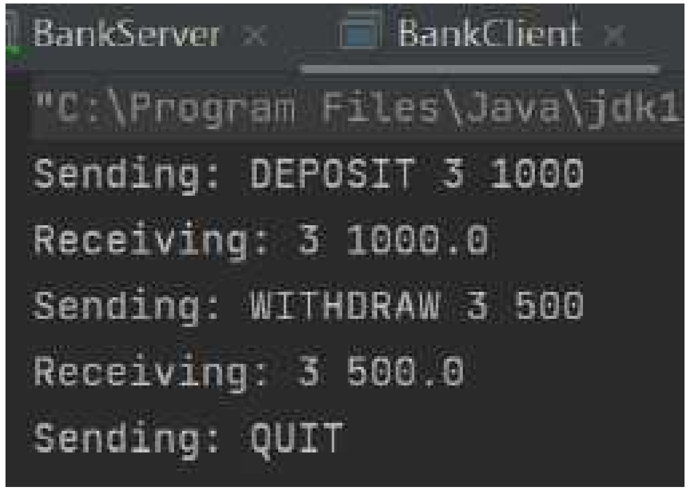
```
public void executeCommand (String command) {
    int account = in.nextInt();
    double amount;
    switch (command) {
    case "DEPOSIT" :
        amount = in.nextDouble();
        bank.deposit( account, amount);
        break;
    case "WITHDRAW" :
        amount = in.nextDouble();
        bank.withdraw( account, amount);
        break;
    case "BALANCE" :
        break;
    default:
        out.println( "Invalid command" );
        out.flush();
        return;
    }
    out.println( account + " " + bank.getBalance( account) );
    out.flush();
}
```

## Case Study

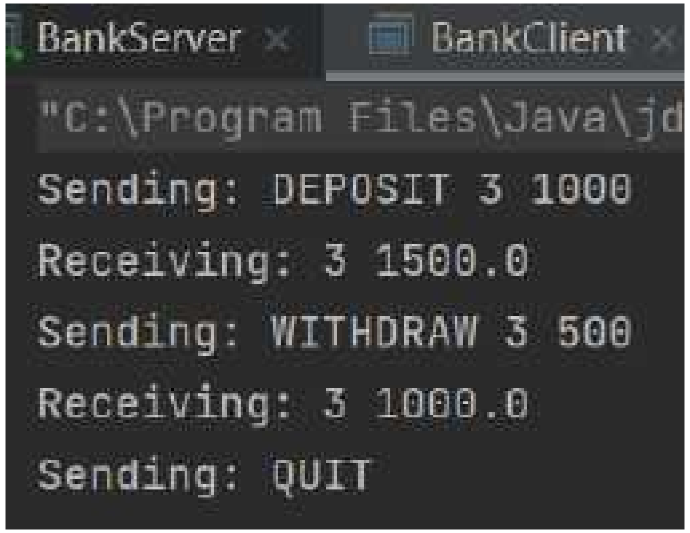


```
"C:\Program Files\Java\jdk1.8.0_20
Waiting for clients to connect...
Client connected.
Client connected.
```

Server keeps running ……



```
"C:\Program Files\Java\jdk1
Sending: DEPOSIT 3 1000
Receiving: 3 1000.0
Sending: WITHDRAW 3 500
Receiving: 3 500.0
Sending: QUIT
```



```
"C:\Program Files\Java\jd
Sending: DEPOSIT 3 1000
Receiving: 3 1500.0
Sending: WITHDRAW 3 500
Receiving: 3 1000.0
Sending: QUIT
```

## Next Lecture

- Reflection
- Annotation
- JUnit