Unit-3: JDBC

Introduction:

Databases only contain data in a specific format and order. In-order to communicate with it (read, write, update, delete data, create, delete database collections, etc.), we have been using a CLI language called SQL.

But not every user must know SQL for using an RDB.

Hence, In-order to perform SQL operations on a database without the user's direct knowledge about SQL and interaction with CLI, we use high-level programming languages.

i.e., the high-level languages are used on applications to give user-friendly interface to users for interacting with an RDB.

Examples of relational databases are: MySQL, Oracle, PostgreSQL, etc.

Examples of high-level languages for RDB communication are: Java, Python, C, C++, etc.

The query commands will be pre-written in high-level languages. So, no changes/personalization can be made to the queries, by the end user.

I.e., only programmer can change the query commands

Communication with DB is achieved by the usage of APIs (Application Protocol Interfaces).

An API is a set of rules and tools that allows different software applications to communicate with each other by sharing data in specific formats.

I.e., an API defines a way for communication.

For C language, we use ODBC (Open DataBase Communication) API.

For Java language, we use JDBC (Java DataBase Connectivity) API.

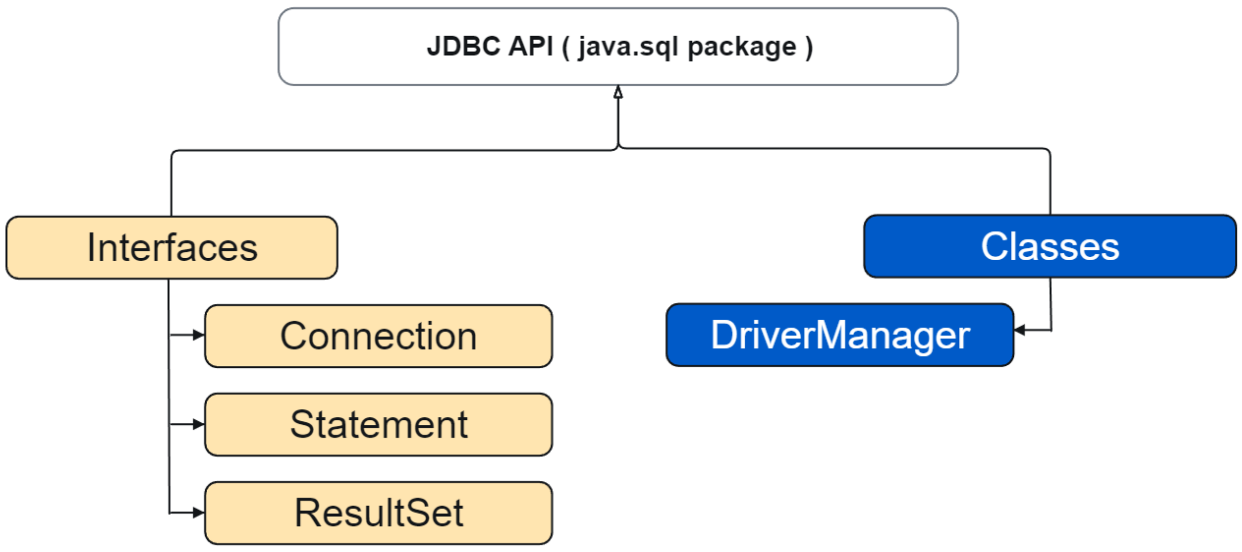
Hence, JDBC is a Java API used to connect and interact with Relational databases.

Java Application <---> JDBC API <---> SQL <---> RDB

{ JDBC API only allows to communicate with Relational DataBase (SQL) (which stores data in the form of tables) but not with non-RDB(NoSQL) (which stores date in the form of objects) (ex. MongoDB) }

JDBC API is provided by the java.sql package.

The package includes interfaces and classes that contribute to different functionalities/roles, while connecting and querying the database.

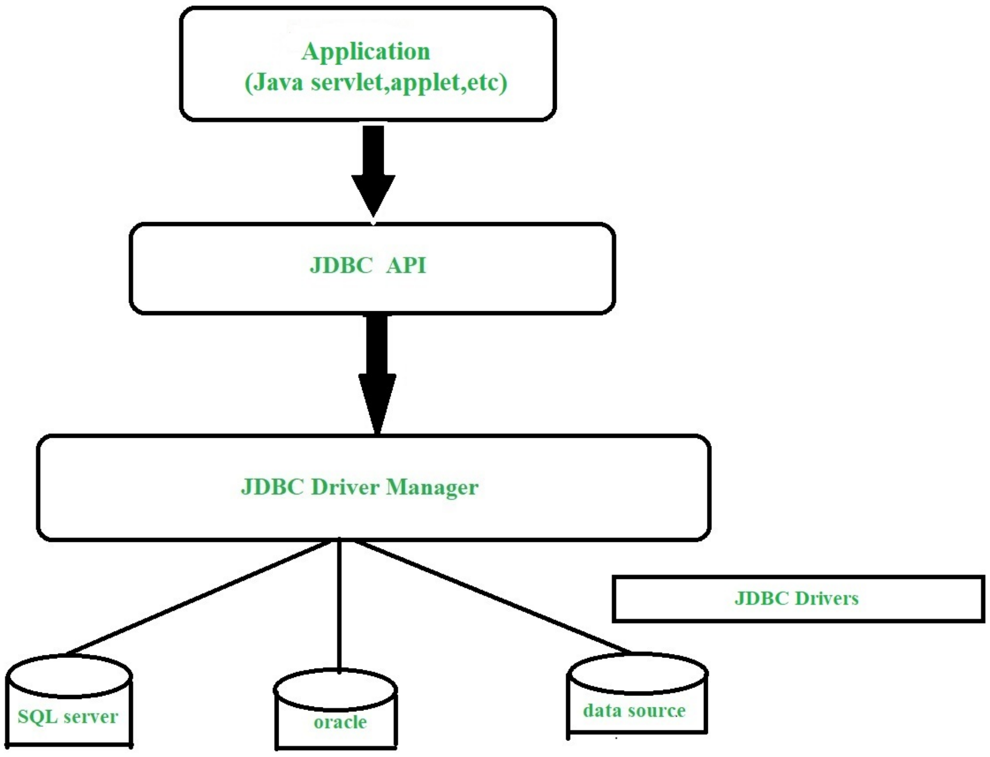


But, the API package is not enough to achieve the connection with DB. Inorder to allow the user's hardware to connect and communicate with the DB, the System require a driver, known as JDBC Driver.

Hence, JDBC Driver is a software component that enables Java applications to connect and interact with DB.

Every DB Language provides its own JDBC driver. It must be downloaded and pasted in Java application root directory.

The JDBC's DriverManager Interface is used to manage the connections and interactions with DB, through the Driver.

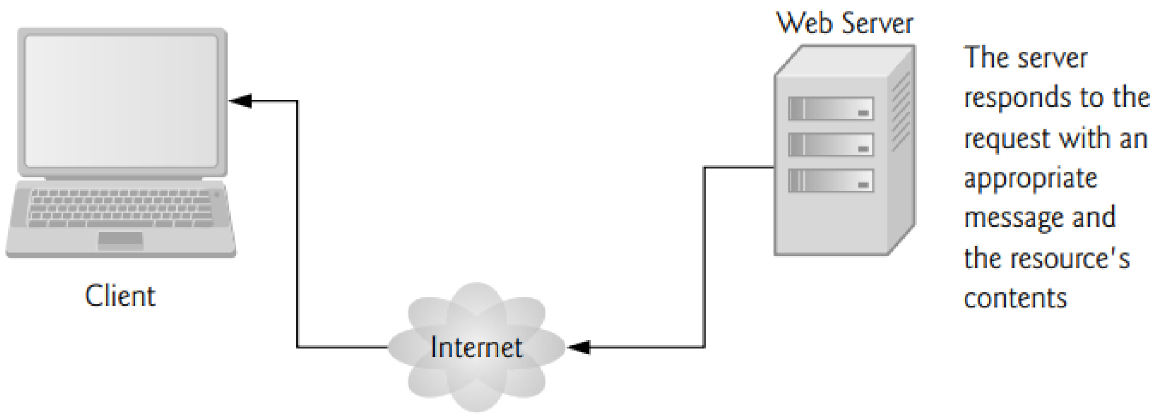


Types of JDBC Architecture

- JDBC (Java Database Connectivity) can be used in different architectures, commonly classified into 2-tier and 3-tier architectures.

- These architectures refer to the way in which the components responsible for database access are organized within an application.

-> 2-Tier Architecture:
In a 2-tier architecture, the JDBC components are organized into two layers: the client layer and the database layer.


Web Server
The server responds to the request with an appropriate message and the resource's contents
Client
Internet

1) Client Layer:

- The client layer includes the user interface (presentation) and the business logic.
- The client directly communicates with the database using JDBC.
- JDBC components (like **`Connection`**, **`Statement`**, **`ResultSet`**) are part of the client application.
- This architecture is also known as the "client-server" architecture.

2) Database Layer:

- The database layer includes the database itself.
- The application layer communicates with the database using JDBC.

Advantages:

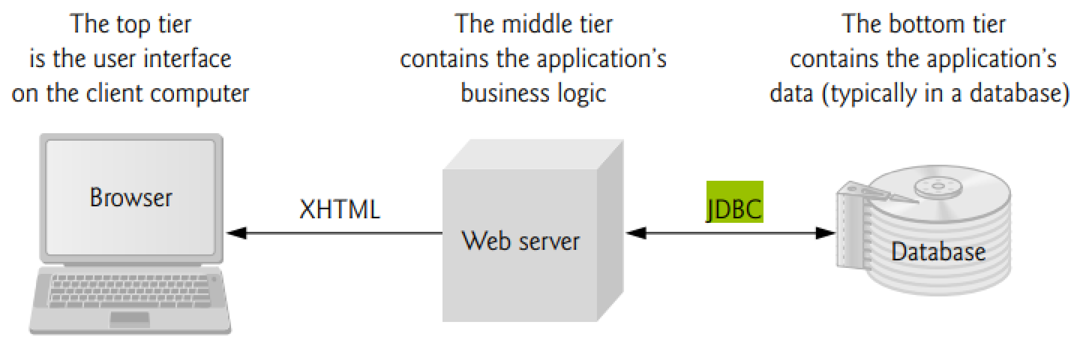
- Simple architecture.
- Suitable for small-scale applications with moderate database access requirements.

Disadvantages:

- Limited scalability: As the application grows, it can lead to increased network traffic and performance issues.
- Business logic and database access are tightly coupled.

-> 3-Tier Architecture:

In a 3-tier architecture, the components are organized into three layers: the client layer, the application layer (middleware or server-side logic), and the database layer.


The top tier is the user interface on the client computer
The middle tier contains the application's business logic
The bottom tier contains the application's data (typically in a database)
Browser
XHTML
Web server
JDBC
Database

1) Client Layer:

- Similar to the 2-tier architecture, the client layer includes the user interface and possibly some business logic.
- The client communicates with the application layer.

2) Application (Middle) Layer:

- This layer contains the business logic, application processing, and potentially presentation logic.
- JDBC components are part of this layer.

- It acts as an intermediary between the client and the database.

3) Database Layer:

- The database layer includes the database itself.
- The application layer communicates with the database using JDBC.

Advantages:

- Improved scalability: The separation of business logic and database access allows for better scalability and maintainability.
- Clear separation of concerns: Business logic is separate from database access logic.
- Suitable for large-scale applications with complex business logic.

Disadvantages:

- Increased complexity: Implementing a 3-tier architecture is more complex than a 2-tier architecture.
- Potential for increased latency due to communication between layers.

JDBC Implementation:

The steps to implement JDBC are:

1) Include JDBC Driver

2) Import Necessary Packages:

3) Establish Connection

4) Create Statement

5) Execute the Query Statement

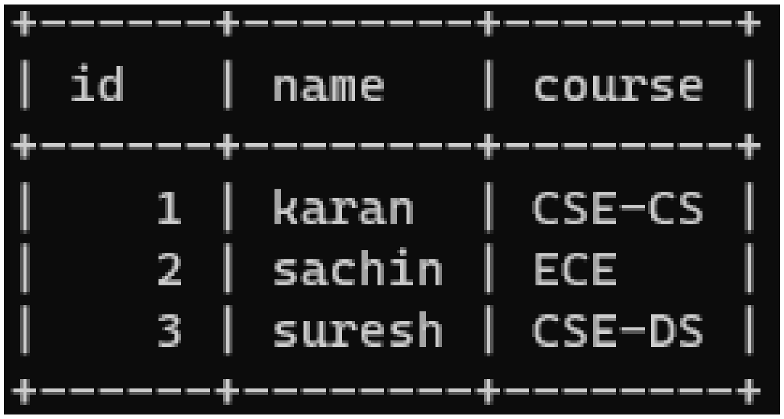
6) Catch and Store the Results

-> At the end, it is important to close the DB connection (end the session) with close() method and release the DB, to let other processes access the DB.

Example:

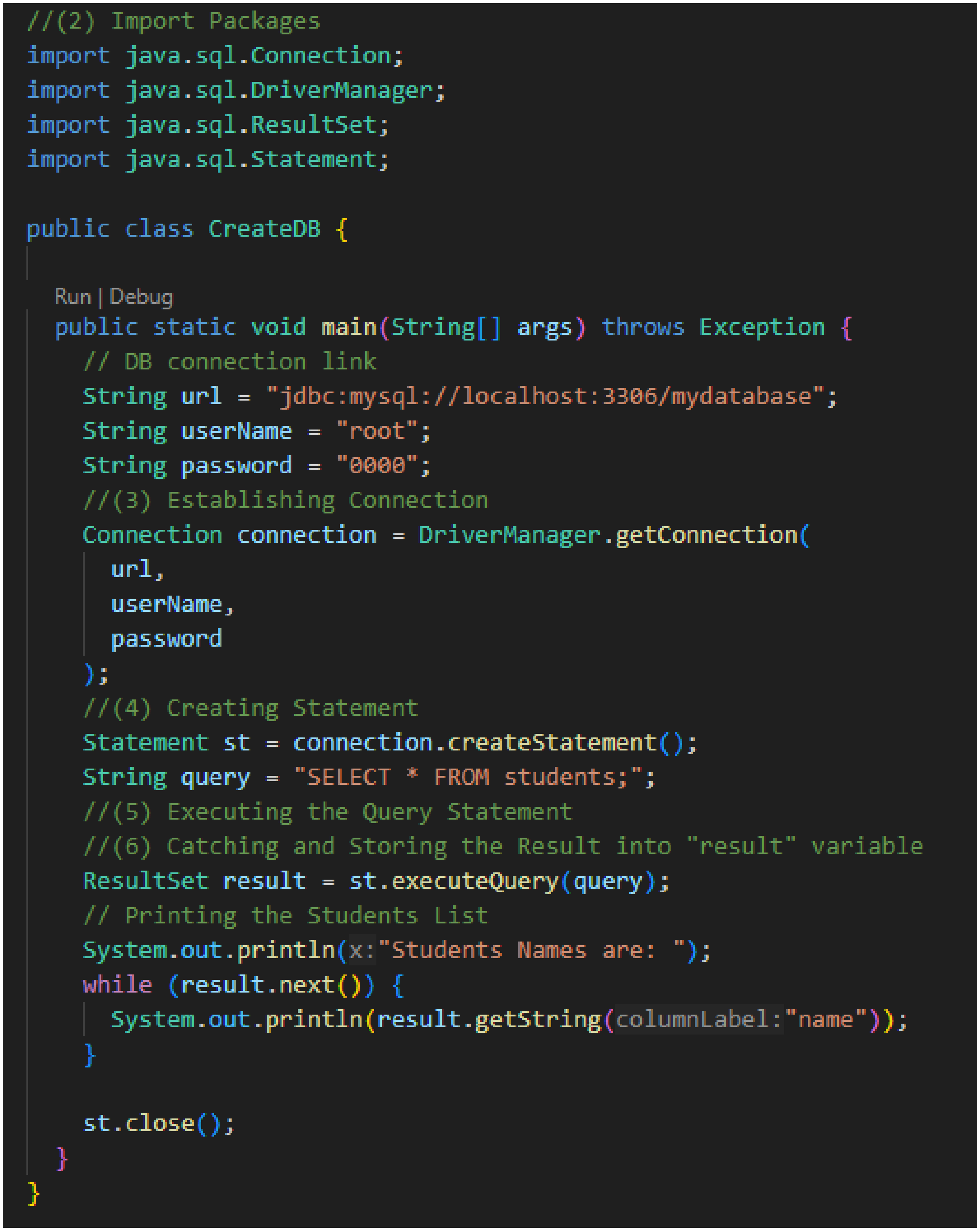
=>Get the students list from “students” table in “mydatabase” DB and print the list into terminal.

Students Table:



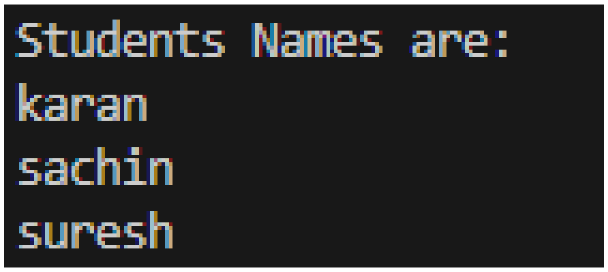
```
+------+--------+--------+
| id   | name   | course |
+------+--------+--------+
|    1 | karan  | CSE-CS |
|    2 | sachin | ECE    |
|    3 | suresh | CSE-DS |
+------+--------+--------+
```

Code logic:



```
//(2) Import Packages
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

public class CreateDB {

  Run | Debug
  public static void main(String[] args) throws Exception {
    // DB connection link
    String url = "jdbc:mysql://localhost:3306/mydatabase";
    String userName = "root";
    String password = "0000";
    //(3) Establishing Connection
    Connection connection = DriverManager.getConnection(
      url,
      userName,
      password
    );
    //(4) Creating Statement
    Statement st = connection.createStatement();
    String query = "SELECT * FROM students;";
    //(5) Executing the Query Statement
    //(6) Catching and Storing the Result into "result" variable
    ResultSet result = st.executeQuery(query);
    // Printing the Students List
    System.out.println(x:"Students Names are: ");
    while (result.next()) {
      System.out.println(result.getString(columnLabel:"name"));
    }

    st.close();
  }
}
```

Output:



```
Students Names are:
karan
sachin
suresh
```

Connection Class:

A connection represents is a session b/w a Java application and a database.

The “Connection” Interface is the one which provides the methods for:

- Establishing a connection.
- Creating statements.
- Managing transactions.
- Obtaining meta-data about the database.

Some commonly used methods of Connection interface are:

1) public Statement createStatement(): creates a statement object that can be used to execute SQL queries.

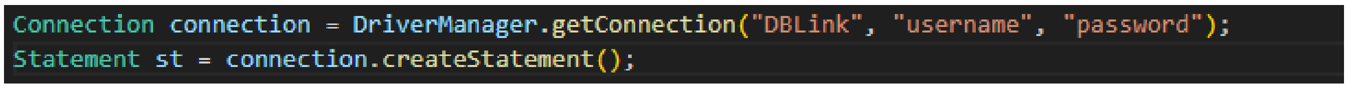
2) public void setAutoCommit(boolean status): is used to set the commit status. By default, it is true.

3) public void commit(): saves the changes made since the previous commit/rollback is permanent.

4) public void rollback(): Drops all changes made since the previous commit/rollback.

5) public void close(): closes the connection and Releases a JDBC resources immediately.

Ex:



```java
Connection connection = DriverManager.getConnection("DBLink", "username", "password");
Statement st = connection.createStatement();
```

Statements:

Statements are nothing but the executable commands/queries.

These commands/queries are same as we write in CLI of a DB to perform operations on database, such as, CRUD (create DB/table, read, update and delete).

These statements are written in String format and passed to a method of "Statement" interface to execute it.

The "Statement" interface provides various methods to execute the provided queries in the database, such as:

1) public ResultSet executeQuery(String sql): is used to execute SELECT query. It returns the object of ResultSet.

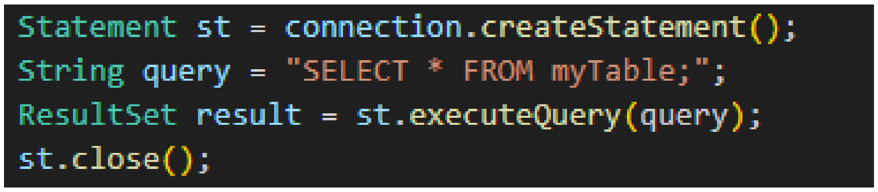
2) public int executeUpdate(String sql): is used to execute specified query, it may be create, drop, insert, update, delete etc.

3) public boolean execute(String sql): is used to execute queries that may return multiple results.

4) public int[ ] executeBatch(): is used to execute batch of commands.

This interface also provides a method to close the connection and end the session b/w database and client, which is close( ).

Ex:



```
Statement st = connection.createStatement();
String query = "SELECT * FROM myTable;";
ResultSet result = st.executeQuery(query);
st.close();
```

Catching Database Results:

When the query statement is executed, the method returns the result in the form of a table.

Hence the returned resultant table must be catched and stored in a variable.

The "ResultSet" Interface allows us to do that.

“ResultSet” Interface allows to catch the results of executed query and provides methods for navigating through the result.

The whole table cannot be printed at once. But, by using next( ) method (provided by ResultSet interface), we can traverse through the records of the result table.

Then use getString( ) or getInt( ) (based on the type of column data), to get the value in a specific column of the record.

Some commonly used methods of ResultSet interface are:

1) public boolean next(): is used to move the cursor to the one row next from the current position.

2) public boolean previous(): is used to move the cursor to the one row previous from the current position.

3) public boolean first(): is used to move the cursor to the first row in result set object.

4) public boolean last(): is used to move the cursor to the last row in result set object.

5) public boolean absolute(int row): is used to move the cursor to the specified row number in the ResultSet object.

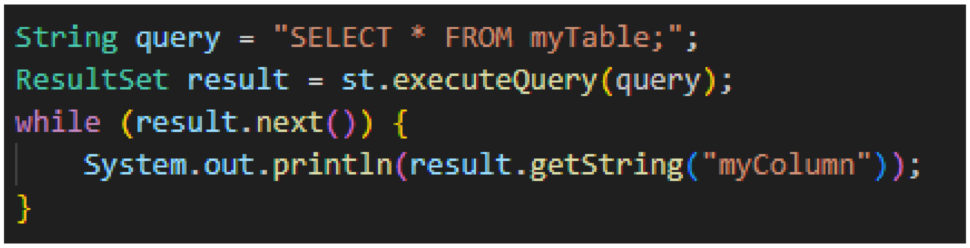
6) public int getInt(int columnIndex): is used to return the data of specified column index of the current row as int.

7) public int getInt(String columnName): is used to return the data of specified column name of the current row as int.

8) public String getString(int columnIndex): is used to return the data of specified column index of the current row as String.

9) public String getString(String columnName): is used to return the data of specified column name of the current row as String.

Ex:



```
String query = "SELECT * FROM myTable;";
ResultSet result = st.executeQuery(query);
while (result.next()) {
    System.out.println(result.getString("myColumn"));
}
```

Handling Database Queries:

..................[JDBC Implementation all 6 points]..................

The most important task while handling the database queries is "exception handling".

The queries executed in database by JDBC driver, will return a result set (table) if the request query is valid.

A query is said to be valid and will be successfully executed if:

- It is Syntactically correct.
- The request Database, Table, Records Exists.

If any of the above cases fails, the query statements fail to execute and throws an Exception into the terminal.

(An Exception is a runtime error which stops the further execution of a program.)

Any operation, such as establishing connection, creating statements, executing/updating statements and closing connection can throw an exception.

Hence, in order to make the application error-free and versatile, the exceptions must be handled.

We can handle the exceptions by either using a try-catch blocks (or) by specifying "throws" keyword at the signature of the method, indicating that "this method might throw an exception, so handle it!".

When usage of DB is completed, close the connection by using close( ) method to release the database and allowing other DBs to access the DB.

NETWORKING:

Two or more interconnected computers/electronic devices which communicate with each other by sharing messages/data/resources, is known as a Network, and the process of communicating is known as networking.

Commonly used Network Model Architectures are: OSI (and) TCP/IP.

To achieve communication successfully, there must be specific procedure with specific rules and regulations to be followed.

These procedures/rules & regulations are provided by Protocols.

The commonly used networking protocols are:

- TCP (Transfer Control Protocol)
- UDP (User Datagram Protocol)

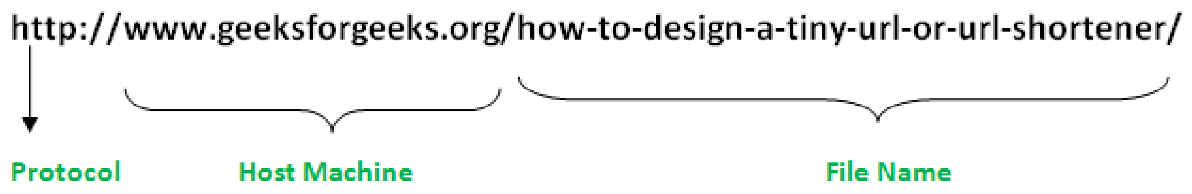
The java.net package provides all networking related functionalities in the form of classes.

Ex: URL, URLConnection, ServerSocket, Socket, DatagramSocket, DatagramPacket, InetAddress, etc.

URL Class:

A URL (Uniform Resource Locator) is a simple string in specific format, which uniquely identifies a Server location on a network followed by File (resource) location in the Server.

It includes (4) components:


http://www.geeksforgeeks.org/how-to-design-a-tiny-url-or-url-shortener/
Protocol
Host Machine
File Name

1. **Protocol:** (Rules to be followed for successful transfer of request) HTTP is the protocol here.
2. **Hostname:** Name of the machine on the WWW(World Wide Web), on which the resource (files) exists. Hostaddress (which contains IP and MAC

addresses) is converted to unique name on the network, known as Hostname.

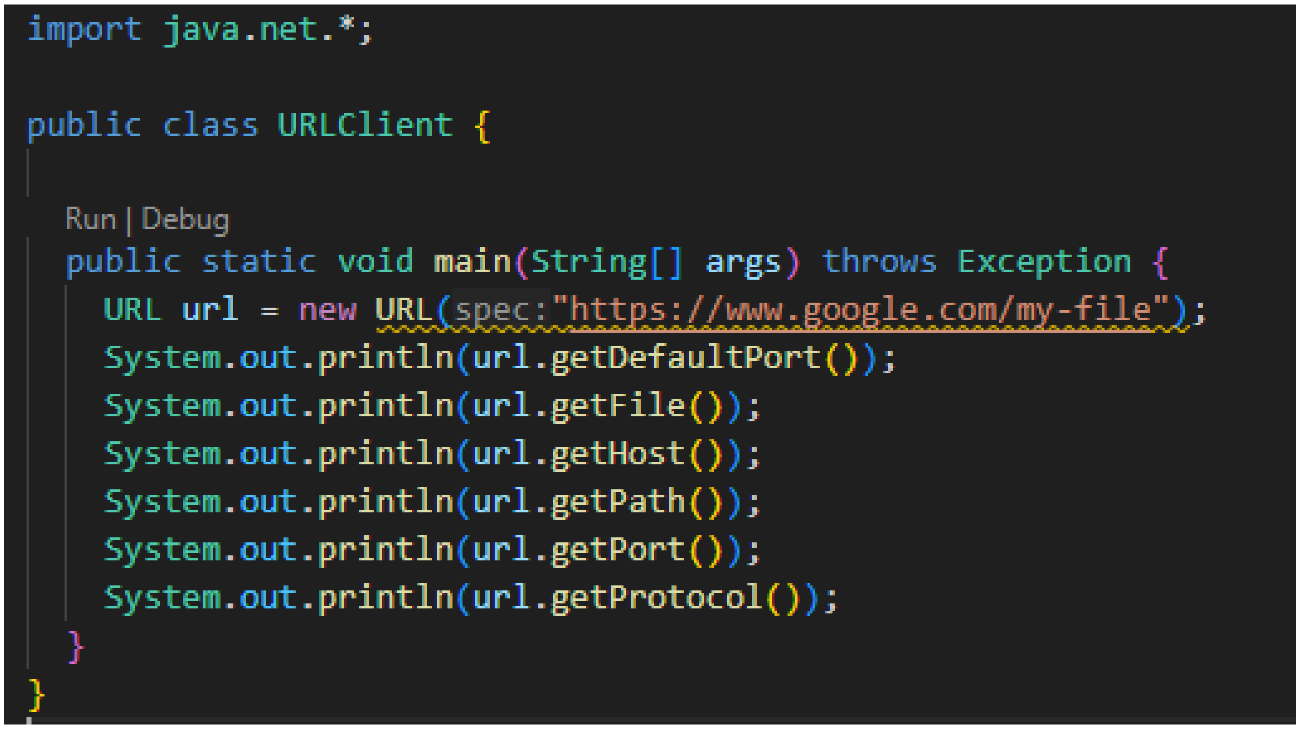
3. **File Name:** The pathname to the file on the machine.
4. **Port Number:** (ConnectionType Identification Number) It is optional because web browser identifies port number based on the protocol. The default port for HTTP is (80) and for HTTPS is (443).

URL is a form of a request to the server, from the client.

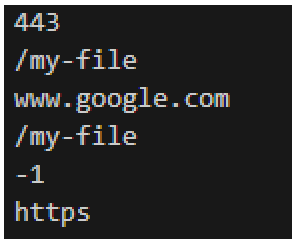
URL class is used in Java Networking Applications for identifying a specified URL string.

It Provides methods to get various components of the provided URL, such as:

Method	**Action Performed**
getDefaultPort()	Returns the default port used
getFile()	Returns the file name.
getHost()	Return the hostname of the URL in IPv6 format
getPath()	Returns the path of the URL, or null if empty
getPort()	Returns the port associated with the protocol specified by the URL (-1 if absent)
getProtocol()	Returns the protocol used by the URL



```
import java.net.*;

public class URLClient {

  Run | Debug
  public static void main(String[] args) throws Exception {
    URL url = new URL(spec:"https://www.google.com/my-file");
    System.out.println(url.getDefaultPort());
    System.out.println(url.getFile());
    System.out.println(url.getHost());
    System.out.println(url.getPath());
    System.out.println(url.getPort());
    System.out.println(url.getProtocol());
  }
}
```

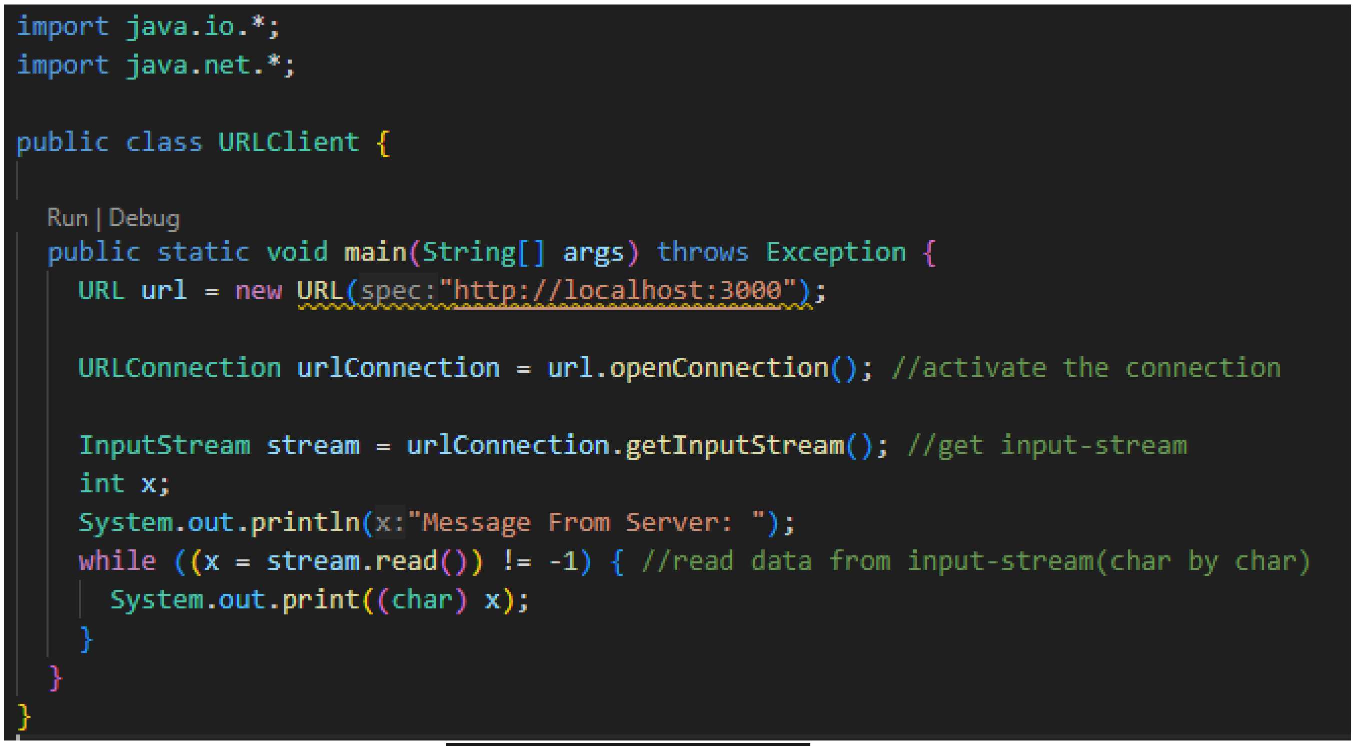


```
443
/my-file
www.google.com
/my-file
-1
https
```

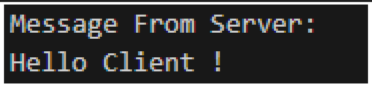
---Extra--

URL class is just to identify a URL string. While URLConnection is an abstract class used to establish a connection from the client's device to the specified URL of a Server.

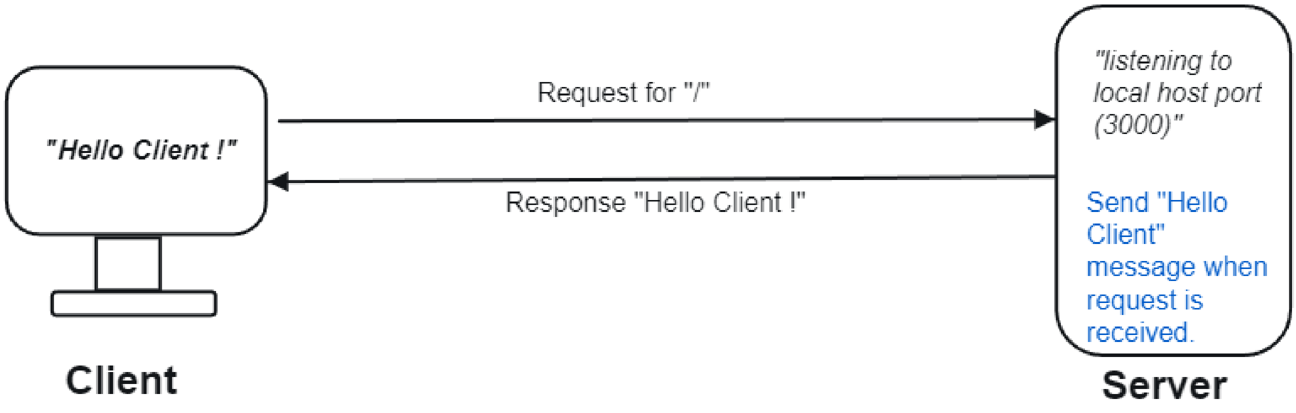
OpenConnection( ) method, provided by URLConnection abstract class, is used to establish a connection.



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

public class URLClient {

  Run | Debug
  public static void main(String[] args) throws Exception {
    URL url = new URL(spec:"http://localhost:3000");

    URLConnection urlConnection = url.openConnection(); //activate the connection

    InputStream stream = urlConnection.getInputStream(); //get input-stream
    int x;
    System.out.println(x:"Message From Server: ");
    while ((x = stream.read()) != -1) { //read data from input-stream(char by char)
      System.out.print((char) x);
    }
  }
}
```



```
Message From Server:
Hello Client !
```


"Hello Client !"
Request for "/"
Response "Hello Client !"
"listening to local host port (3000)"
Send "Hello Client" message when request is received.
Client
Server

Web Sockets:

Web-Socket is a communication protocol that provides full-duplex communication channels over a single, long-lived connection between clients and servers.

(Full-duplex communication implies a bidirectional line that can move data in both directions simultaneously)

Unlike traditional unidirectional request-response-based communication (as seen in HTTP), where the client sends a request and the server responds, Web-Sockets allow for real-time bidirectional communication.

This means that both the client and the server can send messages to each other at any time without the need for the client to request data.

Web-Sockets connection ends when the connection session ends.

Web-Sockets use a transport layer protocol to establish a communication channel between clients and servers.

While the Web-Socket protocol itself operates at a higher level, it relies on lower-level transport protocols to manage the actual data transmission.

The two most used transport layer protocols for Web-Sockets are:

- TCP (Transmission Control Protocol)
- UDP (User Datagram Protocol).

TCP protocol is a connection-based protocol.

i.e., original data is divided into segments and transmitted in an order. Hence, the connection must be persistent till all the segments transmit.

UDP protocol is a connectionless-based protocol.

i.e. original data is transmitted at a time, in the form of a packet. Hence, no persistent connection is required to transmit the data.

TCP Sockets:

TCP Socket is a connection-based communication protocol that provides bidirectional communication channels over a single, long-lived connection between clients and servers.

The data to be sent is divided into multiple segments and sent over an input-stream in an order.

These are used in applications that require a reliable and ordered data exchange, such as messaging apps, web-browsing, Email transfer, File Transfer, Database communication, etc.

The ServerSocket class is used for creating a server-based socket (and) Socket class is used for creating a client-based socket.

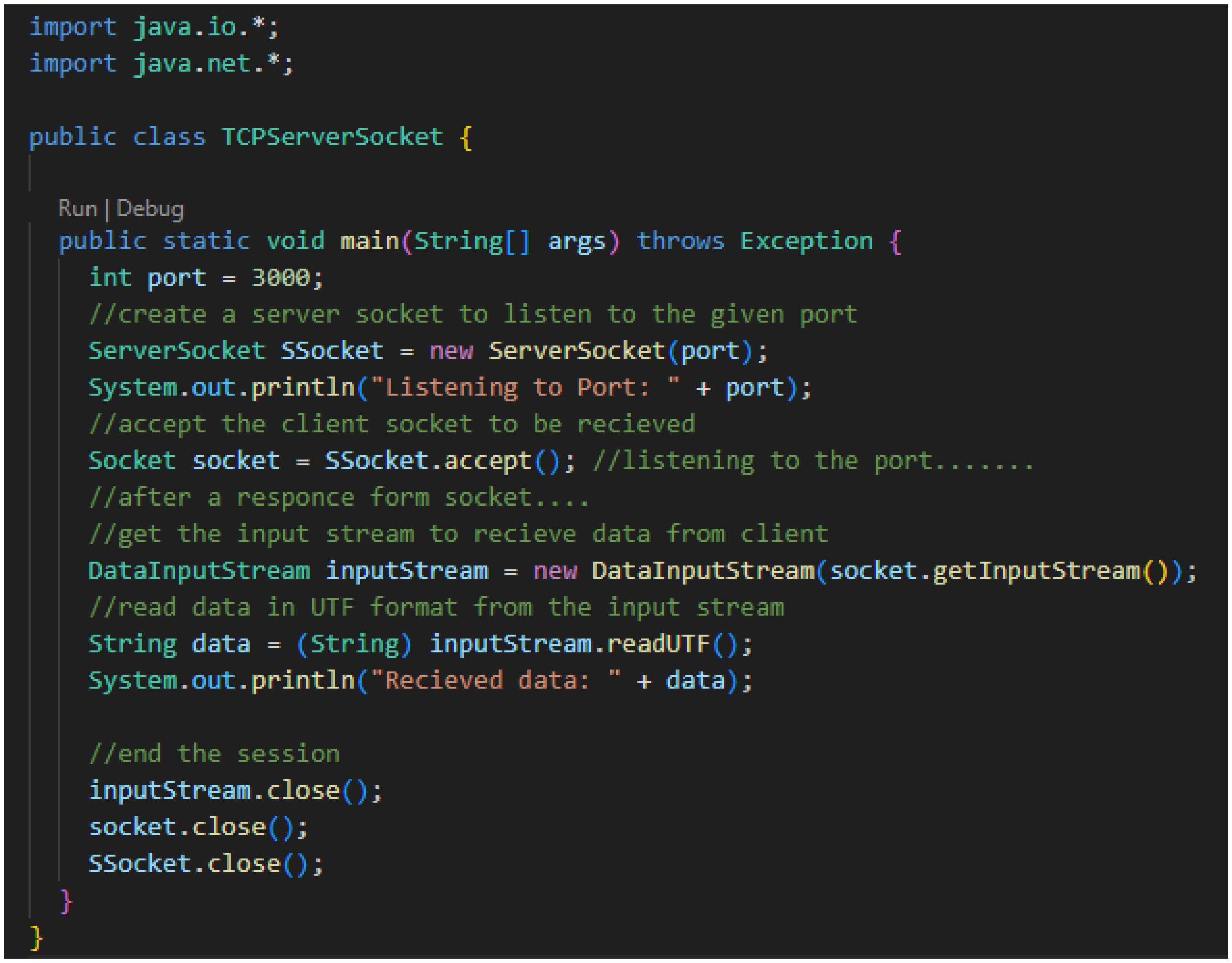
The data is sent and received through the output and input streams respectively, provided by java.io package.

Methods provided by I/O stream classes to send/receive data, are:

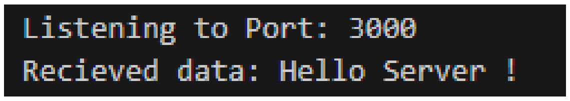
- readUTF( ) - read data from input stream in UTF format
- writeUTF( ) - write data into output stream in UTF format
- readInt( ) - read int-type data from input stream
- writeInt( ) - write int-type data into output stream
- readByte( ) - read byte-type data from input stream
- wirteByte( ) - write byte-type data into output stream

Program:

- Server Socket code:

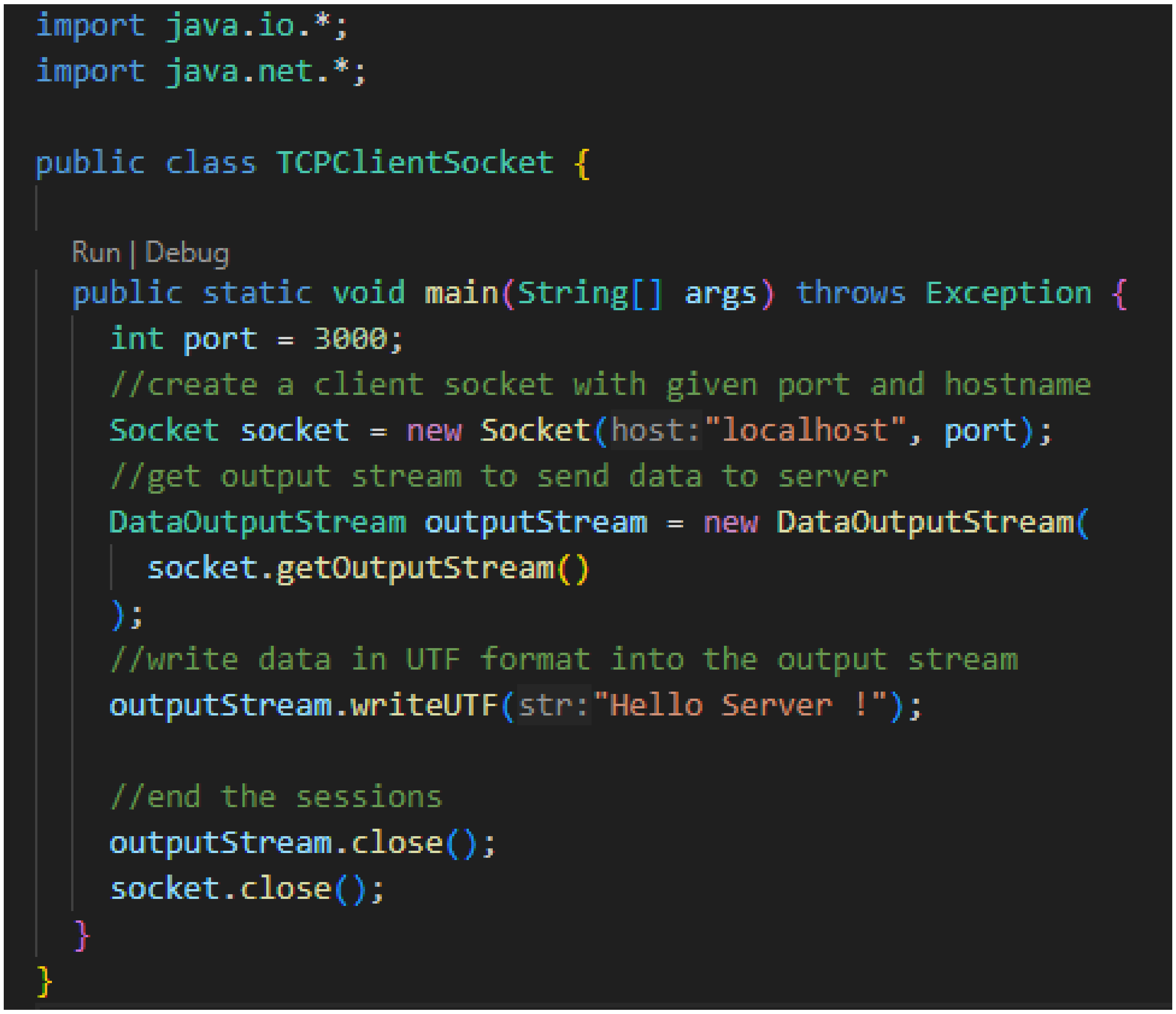


```
import java.io.*;
import java.net.*;

public class TCPServerSocket {

    Run | Debug
    public static void main(String[] args) throws Exception {
        int port = 3000;
        //create a server socket to listen to the given port
        ServerSocket SSocket = new ServerSocket(port);
        System.out.println("Listening to Port: " + port);
        //accept the client socket to be recieved
        Socket socket = SSocket.accept(); //listening to the port.......
        //after a responce form socket....
        //get the input stream to recieve data from client
        DataInputStream inputStream = new DataInputStream(socket.getInputStream());
        //read data in UTF format from the input stream
        String data = (String) inputStream.readUTF();
        System.out.println("Recieved data: " + data);

        //end the session
        inputStream.close();
        socket.close();
        SSocket.close();
    }
}
```



```
Listening to Port: 3000
Recieved data: Hello Server !
```

- Client Socket code:



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

public class TCPClientSocket {

  Run | Debug
  public static void main(String[] args) throws Exception {
    int port = 3000;
    //create a client socket with given port and hostname
    Socket socket = new Socket(host:"localhost", port);
    //get output stream to send data to server
    DataOutputStream outputStream = new DataOutputStream(
      socket.getOutputStream()
    );
    //write data in UTF format into the output stream
    outputStream.writeUTF(str:"Hello Server !");

    //end the sessions
    outputStream.close();
    socket.close();
  }
}
```

UDP Sockets:

UDP Socket is a connectionless-based communication protocol that provides bidirectional communication channels over a single, long-lived connection between clients and servers.

The data is encapsulated with the host’s(server) IP-address and Port number to form a data packet, which can be independently transmitted, at a time.

These are used for applications where it is ok for delays in data transmission and no guaranty of packet delivery, such as online games, live streaming, and video conferencing.

The DatagramSocket class is used for creating both server and client-based sockets(with and without port argument, respectively).

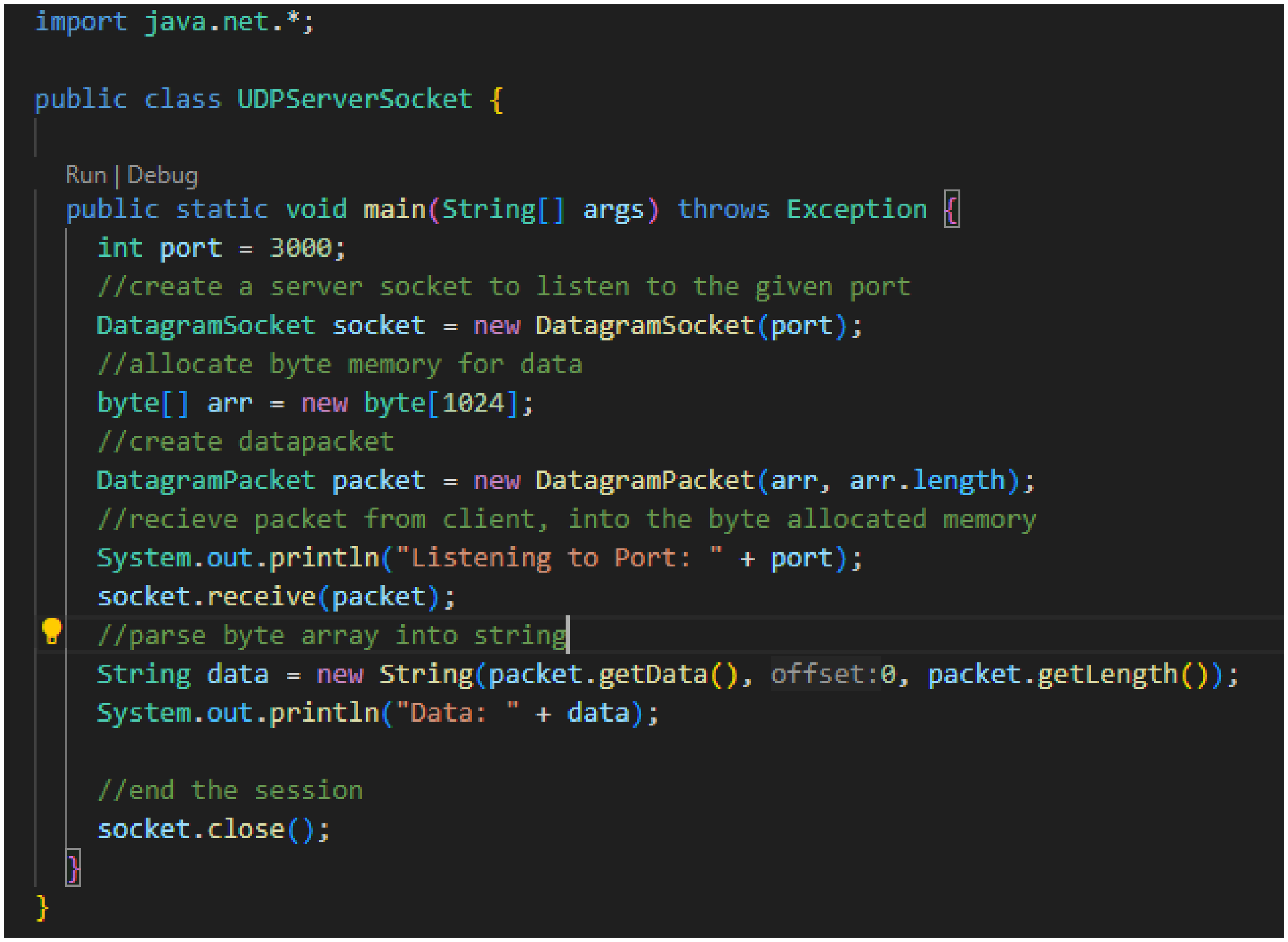
The DatagramPacket class is used to create a data packet.

Methods provided by DatagramSocket class to send and receive data are:

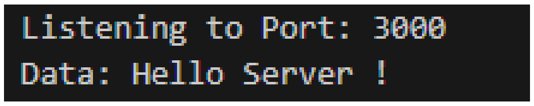
- send( ) - send byte-array.
- receive( ) - receive byte-array.

Program:

- Server Socket code:

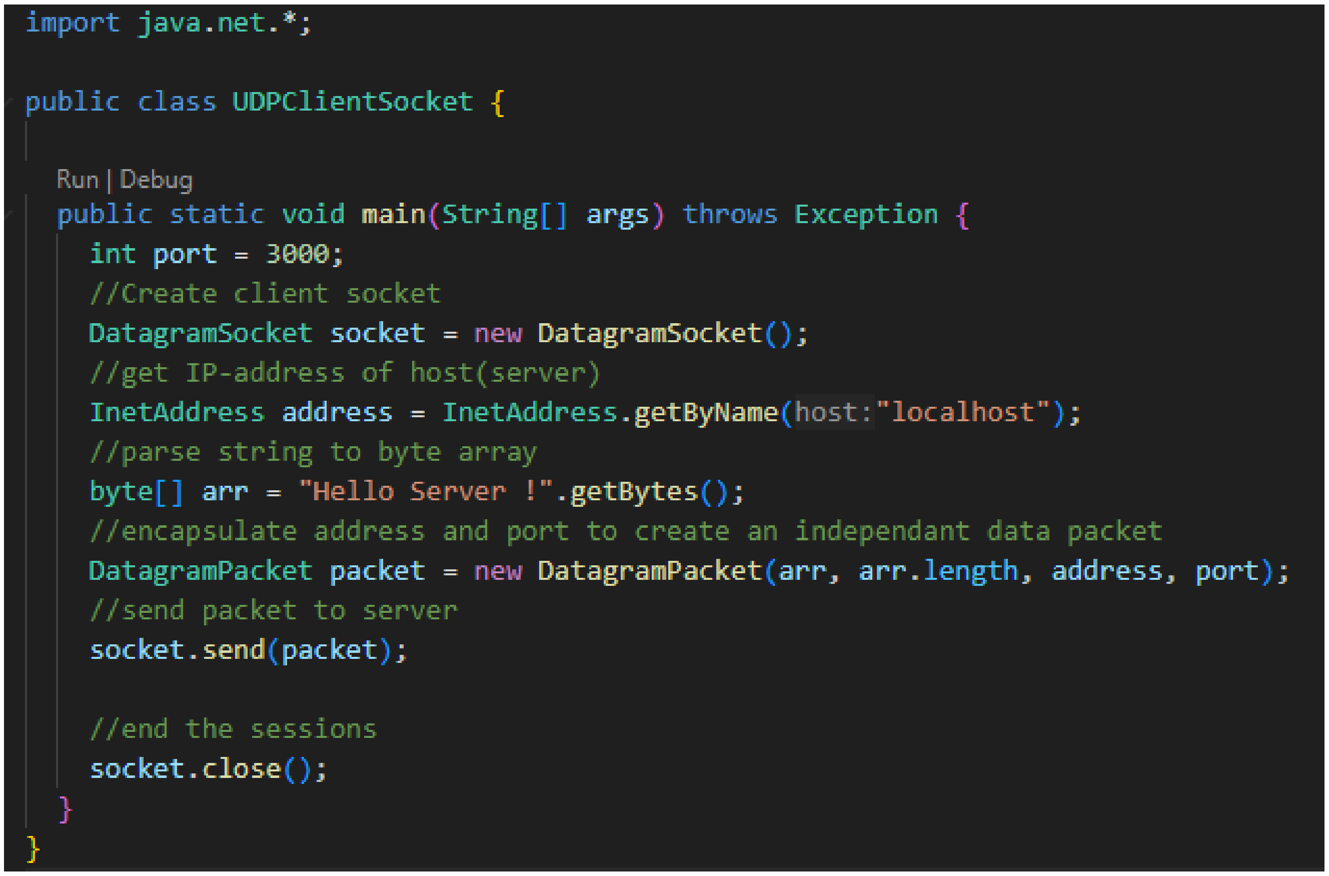


```java
import java.net.*;

public class UDPServerSocket {

  Run | Debug
  public static void main(String[] args) throws Exception {
    int port = 3000;
    //create a server socket to listen to the given port
    DatagramSocket socket = new DatagramSocket(port);
    //allocate byte memory for data
    byte[] arr = new byte[1024];
    //create datapacket
    DatagramPacket packet = new DatagramPacket(arr, arr.length);
    //recieve packet from client, into the byte allocated memory
    System.out.println("Listening to Port: " + port);
    socket.receive(packet);
    //parse byte array into string
    String data = new String(packet.getData(), offset:0, packet.getLength());
    System.out.println("Data: " + data);

    //end the session
    socket.close();
  }
}
```



```
Listening to Port: 3000
Data: Hello Server !
```

- Client Socket Code:



```java
import java.net.*;

public class UDPClientSocket {

  Run | Debug
  public static void main(String[] args) throws Exception {
    int port = 3000;
    //Create client socket
    DatagramSocket socket = new DatagramSocket();
    //get IP-address of host(server)
    InetAddress address = InetAddress.getByName(host:"localhost");
    //parse string to byte array
    byte[] arr = "Hello Server !".getBytes();
    //encapsulate address and port to create an independant data packet
    DatagramPacket packet = new DatagramPacket(arr, arr.length, address, port);
    //send packet to server
    socket.send(packet);

    //end the sessions
    socket.close();
  }
}
```

InetAddress class:

Every device(server machine or client machine) connected over a network of vast no.of other similar devices, must have a unique identifier. Hence, IP Adress serves the purpose.

An IP (internet-protocol) is a unique identification number for a device on a network of multiple devices.

Format of IP address: x : x : x : x   *[range(x) => 0-255]*

It is not possible for every client to remember the IP Address. Hence, an IP address can be assigned with a unique name on the network, known as Hostname.

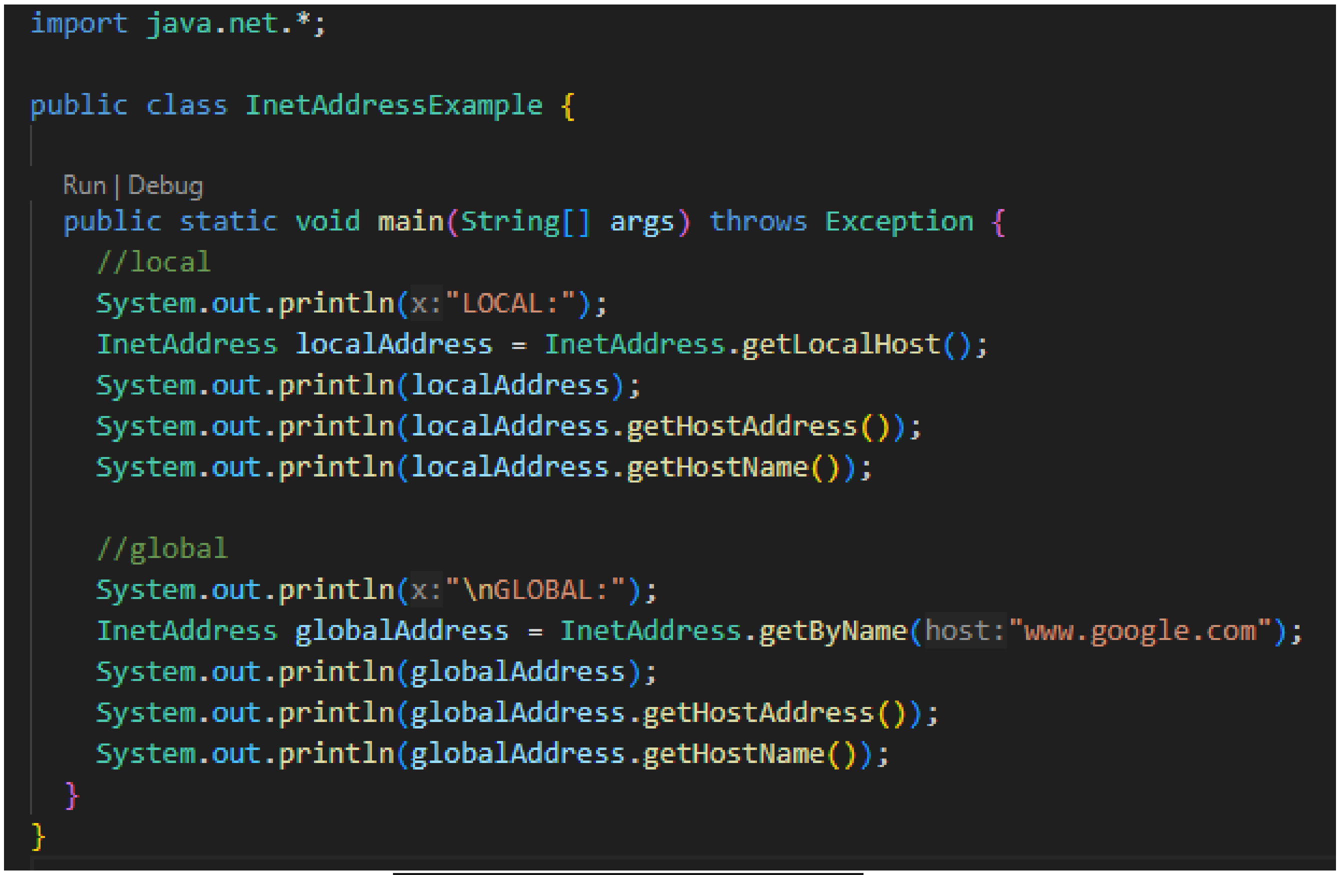
Ex: Google IP Address: 142.250.196.4

Google Host Name: www.google.com

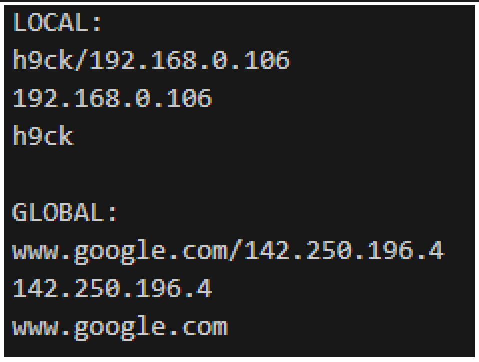
InetAddress class from java.net package is used to represent IP Addresses.

Some common methods provided by InetAddress class are:

- .getLocalHost( ) - returns local hostname and local IP address
- .getByName(<hostname>) - returns hostname and IP address of specified hostname
- .getHostAddress( ) - returns IP address
- .getHostName - returns Host Name



```
import java.net.*;

public class InetAddressExample {

  Run | Debug
  public static void main(String[] args) throws Exception {
    //local
    System.out.println(x:"LOCAL:");
    InetAddress localAddress = InetAddress.getLocalHost();
    System.out.println(localAddress);
    System.out.println(localAddress.getHostAddress());
    System.out.println(localAddress.getHostName());

    //global
    System.out.println(x:"\nGLOBAL:");
    InetAddress globalAddress = InetAddress.getByName(host:"www.google.com");
    System.out.println(globalAddress);
    System.out.println(globalAddress.getHostAddress());
    System.out.println(globalAddress.getHostName());
  }
}
```



```
LOCAL:
h9ck/192.168.0.106
192.168.0.106
h9ck

GLOBAL:
www.google.com/142.250.196.4
142.250.196.4
www.google.com
```

JavaBeans – RMI: