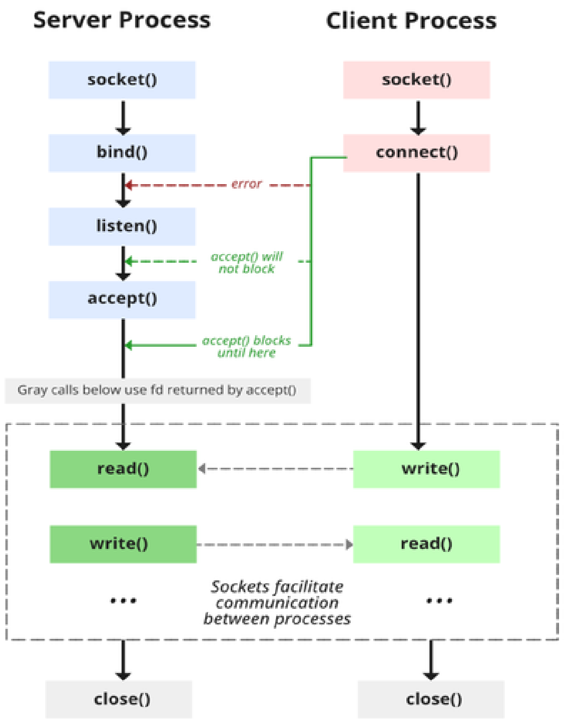
**5COSC022W.2 Client-Server Architectures**

# Tutorial Week 03: Socket Programming

Socket programming establishes network connections, sends and receives data over a network, and communicates between devices or processes. It is based on the concept of sockets, which are endpoints for communication. Sockets can be either connection-oriented or connection-less, depending on the protocol used.

Connection-oriented sockets use the **Transmission Control Protocol (TCP)**, which guarantees reliable and ordered delivery of data. TCP sockets require a **handshake** between the client and the server before exchanging data. To create a TCP socket in , you need to use the Socket class for the client and the ServerSocket class for the server. The Socket constructor takes the IP address and the port number of the server as arguments, while the ServerSocket constructor takes only the port number as an argument. The ServerSocket class has a method called accept(), which waits for a client to connect and returns a Socket object for communication. The Socket class has methods to get the input and output streams, which can be used to read and write data to and from the socket. The close() method closes the socket connection.



In this session, we will focus on Connection-oriented sockets, which utilize the Transmission Control Protocol (TCP) to ensure reliable and ordered delivery of data. You'll become familiar with implementing TCP sockets in  using the Socket class for clients and the ServerSocket class for servers. Through a handshake protocol, clients and servers establish a connection before data exchange ensues.

Throughout this exercise, you'll use the power of  to construct a basic client-server architecture. By the end, you'll have gained proficiency in initiating connections, sending messages, and processing responses using sockets.

| CHEAT SHEET FOR JAVA SOCKETS | |
|---|---|
| **Task** | **Code Snippet** |
| **Create a Server Socket** | *ServerSocket serverSocket = new ServerSocket(portNumber);* |
| **Accept Client Connection** | *Socket clientSocket = serverSocket.accept();* |
| **Create a Client Socket** | *Socket socket = new Socket("hostname", portNumber);* |
| **Read from Socket** | *BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));*<br><br>*String inputLine = in.readLine();* |
| **Write to Socket** | *PrintWriter out = new PrintWriter(socket.getOutputStream(), true);*<br><br>*out.println("message");* |
| **Close Socket** | *socket.close();* |
| **Close Server Socket** | *serverSocket.close();* |
| **Get Local Address** | *InetAddress localAddress = socket.getLocalAddress();* |
| **Get Remote Address** | *InetAddress remoteAddress = socket.getInetAddress();* |

## EXERCISE 1: BUILDING BASIC CLIENT-SERVER COMMUNICATION IN

**Objective:** The goal of this exercise is to understand the fundamentals of network programming in  by creating simple client and server classes. You will learn how to establish a connection, send messages, and receive responses using sockets.

**Overview:** In this exercise, you will be creating two separate classes - Client and Server. The Client class will connect to the Server class using a specific port. Once the connection is established, the Client will send a message to the Server, which will then respond back to the Client.

**Key Concepts:**

- **Sockets:** A socket is one endpoint of a two-way communication link between two programs running on the network. You will use the Socket class for the client and the ServerSocket class for the server.
- **Input/Output Streams:** These are used for communication between the client and server. You will use the InputStreamReader and BufferedReader classes for reading data, and the PrintWriter class for writing data.
- **Exception Handling:** Network operations can fail for various reasons, so it's important to handle exceptions properly in your code. You will use a try-catch block to catch any IOException that may occur.

**Expected Outcome:** By the end of this exercise, you should be able to create a simple client-server application in . You will gain a basic understanding of network programming, and be prepared to explore more complex topics like multi-threading and non-blocking I/O.

### STEPS:

1. **Download the Project Zip File**:
   - Log in to your Blackboard account.
   - Click on the 'Week 3' section.
   - Find the file named Tutorial-week03-exercise1-questions.zip.
   - Click on the file name to download it.
2. **Extract the Zip File**:
   - Locate the downloaded zip file on your computer. It's usually in the 'Downloads' folder unless you chose a different location. You may also you

the ZIP file to import it directly to the NetBeans without requiring extracting the ZIP file.

   - Right-click on the zip file and select ‘Extract All...’.
   - Choose a suitable location on your computer to extract the files.

3. **Open NetBeans IDE**:
   - Launch the NetBeans IDE 18 or above on your computer.
4. **Import the Project**:
   - In NetBeans, click on ‘File’ in the menu bar, then select ‘Open Project’.
   - In the file chooser window, navigate to the location where you extracted the project files.
   - You should see a folder with the same name as the zip file. Click on it to select it.
   - Click on the ‘Open Project’ button.
5. **Explore the Project**:
   - The project should now be visible in the ‘Projects’ tab on the left side of the IDE.
   - Click on the project name to expand it and explore its structure.
6. **Implement Your Code**:
   - Now, you can start implementing your code under each comment in the provided classes.

## CORE CONCEPTS FOR EXERCISE 2: STREAMS AND BYTE ARRAYS

1. **Streams:** In Java, a stream is a sequence of data. You can think of it like a flow of information that you can read from or write to. Streams are a fundamental way to handle input and output (I/O) operations.

2. **byte[] (Byte Array):** A byte array is simply an array that holds a sequence of bytes. In Java, a byte is an 8-bit signed integer. Byte arrays are often used to store binary data (like the raw content of a file, image data, etc.) that you read from or write to streams.

## INPUTSTREAM

- **Purpose:** An abstract class representing an input stream of bytes. It's the superclass of all classes representing an input stream. Think of it as a source from which you can read a sequence of bytes.
- **Key Methods:**
    - *read():* Reads the next byte of data from the input stream.
    - *read(byte[] b):* Reads a number of bytes from the input stream and stores them into the buffer array b.
    - *read(byte[] b, int off, int len):* Reads up to len bytes of data from the input stream into an array of bytes, starting at offset off.
    - *close():* Closes the input stream and releases any system resources associated with it.

## OUTPUTSTREAM

- **Purpose:** An abstract class representing an output stream of bytes. It's the superclass of all classes representing an output stream. Think of it as a destination to which you can write a sequence of bytes.
- **Key Methods:**
    - *write(int b):* Writes the specified byte to this output stream.
    - *write(byte[] b):* Writes b.length bytes from the specified byte array to this output stream.
    - *write(byte[] b, int off, int len):* Writes len bytes from the specified byte array starting at offset off to this output stream.
    - *flush():* Flushes this output stream and forces any buffered output bytes to be written out.
    - *close():* Closes this output stream and releases any system resources associated with the stream.

## BYTE[] (BYTE ARRAY)

- **Purpose:** An array to hold a sequence of bytes. It's a fundamental data structure for representing raw binary data in Java.
- **Declaration:** *byte[] myByteArray;*
- **Initialization:**

- *byte[] myByteArray = new byte[10];* // Creates an array of 10 bytes, initialized to 0.
- *byte[] myByteArray = {10, 20, 30, 40, 50};* // Initializes with specific byte values.

## INPUTSTREAM AND BUFFERED READER KEY DIFFERENCES

| **Feature** | **InputStream** | **BufferedReader** |
|---|---|---|
| **Type** | Abstract class (base for byte streams) | Concrete class (for character streams) |
| **Data** | Bytes (byte, byte[]) | Characters (char, String) |
| **Level** | Low-level | Higher-level |
| **Purpose** | Read raw binary data | Read text data efficiently |
| **Buffering** | No built-in buffering (can use BufferedInputStream) | Built-in buffering |
| **Encoding** | No inherent character encoding | Handles character encoding (via the underlying Reader) |
| **Key Methods** | read() (bytes), close() | read() (chars), readLine(), close() |
| **Typical Usage** | Binary files, network streams, raw data | Text files, user input, text processing |

## OUTPUTSTREAM AND PRINTWRITER KEY DIFFERENCES

| **Feature** | **OutputStream** | **PrintWriter** |
|---|---|---|
| **Type** | Abstract class (base for byte output streams) | Concrete class (for formatted text output) |
| **Data** | Bytes (byte, byte[]) | Characters, formatted strings, various data types (converted to text) |
| **Level** | Low-level | Higher-level |
| **Purpose** | Write raw binary data | Write formatted text data |
| **Formatting** | No built-in formatting | Built-in formatting capabilities (printf, format) |
| **Encoding** | No inherent character encoding | Handles character encoding (usually platform's default or a specified one) |
| **Key Methods** | write() (bytes), flush(), close() | print(), println(), printf(), format(), append(), flush(), close() |
| **Typical Usage** | Binary files, network streams, raw data output | Text files, console output, formatted reports |

## EXERCISE 2: BUILDING A SIMPLE CHAT APPLICATION IN

**Objective:** The goal of this exercise is to understand the fundamentals of network programming in  by creating simple chat client and server classes. You will learn how to establish a connection, send messages, and receive responses using sockets.

**Overview:** In this exercise, you will be creating two separate classes - SimpleChatClient and SimpleChatServer. The SimpleChatClient class will connect to the SimpleChatServer class using a specific port. Once the connection is established, the SimpleChatClient will send a message to the SimpleChatServer, which will then respond back to the SimpleChatClient.

**Key Concepts:**

- **Sockets:** A socket is one endpoint of a two-way communication link between two programs running on the network. You will use the Socket class for the client and the ServerSocket class for the server.
- **Input/Output Streams:** These are used for communication between the client and server. You will use the InputStream and OutputStream classes for reading data and writing data.
- **Exception Handling:** Network operations can fail for various reasons, so it’s important to handle exceptions properly in your code. You will use a try-catch block to catch any IOException that may occur.

**Expected Outcome:** By the end of this exercise, you should be able to create a simple chat application in . You will gain a basic understanding of network programming, and be prepared to explore more complex topics like multi-threading and non-blocking I/O.

## STEPS:

1. **Download the Project Zip File**:
   a. Log in to your Blackboard account.
   b. Click on the ‘Week 3’ section.
   c. Find the file named Tutorial-week03-exercise2-questions.zip.
   d. Click on the file name to download it.
2. **Extract the Zip File**:
   a. Locate the downloaded zip file on your computer. It’s usually in the ‘Downloads’ folder unless you chose a different location. You may also you

the ZIP file to import it directly to the NetBeans without requiring extracting the ZIP file.

   b. Right-click on the zip file and select ‘Extract All...’.
   c. Choose a suitable location on your computer to extract the files.

3. **Open NetBeans IDE**:
   a. Launch the NetBeans IDE on your computer.
4. **Import the Project**:
   a. In NetBeans, click on ‘File’ in the menu bar, then select ‘Open Project’.
   b. In the file chooser window, navigate to the location where you extracted the project files.
   c. You should see a folder with the same name as the zip file. Click on it to select it.
   d. Click on the ‘Open Project’ button.
5. **Explore the Project**:
   a. The project should now be visible in the ‘Projects’ tab on the left side of the IDE.
   b. Click on the project name to expand it and explore its structure.
6. **Implement Your Code**:
   a. Now, you can start implementing your code under each comment in the provided classes.