Week 11: Network Programming - Clients & Servers

Introduction

Network programming in Java enables communication between computers connected over a network. This allows the development of distributed systems, real-time applications, chat applications, multiplayer games, and much more. In this week's topic, we will delve into building **client-server applications** using Java's java.net package, which offers the classes and methods necessary for establishing network connections and data exchange.

Overview of Network Programming

At its core, network programming involves **communication between a client and a server**. The **client** is a program that initiates communication, typically by sending a request for a service. The **server** is a program that listens for requests from clients and responds accordingly, providing data or services.

Client-Server Architecture

This architecture forms the basis of network programming. Here's how it works:

- **Client**: Sends a request to the server and waits for a response. For example, a web browser (client) makes requests to a web server to fetch and display web pages.
- **Server**: Listens for client requests and responds appropriately. A server can handle multiple client requests simultaneously, depending on its implementation.

Key Concepts

1. **Sockets**: A **Socket** represents a connection between a client and a server. It enables data exchange by providing input and output streams.
2. **ServerSockets**: A **ServerSocket** is used on the server side to listen for incoming client connection requests. Once a client connects, a Socket object is created for communication with that client.

Practical Examples Using NetBeans IDE

1. Building a Simple Server-Client Communication System

Step 1: Create the Server Application

In NetBeans, create a new Java project called **SimpleServer** and add the following code:

Code for Server (SimpleServer.java):

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

public class SimpleServer {
    public static void main(String[] args) {
        try (ServerSocket serverSocket = new ServerSocket(12345)) { // Listen on port 12345
            System.out.println("Server is listening on port 12345...");
```

```
            // Accept a client connection
            Socket socket = serverSocket.accept();
            System.out.println("Client connected!");

            // Set up input and output streams for communication
            BufferedReader input = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            PrintWriter output = new PrintWriter(socket.getOutputStream(), true);

            // Read message from the client
            String clientMessage = input.readLine();
            System.out.println("Client says: " + clientMessage);

            // Respond to the client
            output.println("Hello Client! You said: " + clientMessage);

            // Close the socket
            socket.close();
        } catch (IOException ex) {
            System.out.println("Server exception: " + ex.getMessage());
            ex.printStackTrace();
        }
    }
}
```

Explanation:

- The **ServerSocket** is created to listen on port 12345.
- When a client connects, the server accepts the connection and creates a **Socket** for communication.
- The server reads a message sent by the client and responds with a simple acknowledgment.

Step 2: Create the Client Application

In NetBeans, create a new Java project called **SimpleClient** and add the following code:

Code for Client (SimpleClient.java):

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

public class SimpleClient {
    public static void main(String[] args) {
        try (Socket socket = new Socket("localhost", 12345)) { // Connect to server on localhost
            // Set up input and output streams for communication
            BufferedReader input = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            PrintWriter output = new PrintWriter(socket.getOutputStream(), true);
```

```
            // Send a message to the server
            output.println("Hello Server!");

            // Read and display the server's response
            String serverMessage = input.readLine();
            System.out.println("Server says: " + serverMessage);
        } catch (IOException ex) {
            System.out.println("Client exception: " + ex.getMessage());
            ex.printStackTrace();
        }
    }
}
```

Explanation:

- The **Socket** connects to the server on localhost at port 12345.
- The client sends a message to the server and reads the server's response.

Running the Example:

1. Start the **SimpleServer** program. It will wait for a client connection.
2. Run the **SimpleClient** program. It will connect to the server, send a message, and print the server's response.

2. Sockets and ServerSockets in Detail

Sockets:

- **Definition**: A socket is an endpoint for communication between two machines.
- **Usage**: Sockets are used to establish connections between clients and servers and enable data transfer.
- **Key Methods**
    - getInputStream(): Returns an input stream to read data from the socket.
    - getOutputStream(): Returns an output stream to write data to the socket.

ServerSockets:

- **Definition**: A ServerSocket is used by a server application to listen for client connection requests.
- **Usage**: When a client connects, the ServerSocket creates a Socket for communication with the client.
- **Key Methods**:
    - accept(): Waits for a client to connect and returns a Socket when a connection is established.

Additional Practical Examples

Example 1: Handling Multiple Client Connections Using Threads

Multi-Threaded Server (MultiThreadedServer.java):

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

public class MultiThreadedServer {
    public static void main(String[] args) {
        try (ServerSocket serverSocket = new ServerSocket(12345)) {
            System.out.println("Multi-threaded Server is running on port 12345...");

            while (true) {
                // Accept client connections and create a new thread for each client
                Socket clientSocket = serverSocket.accept();
                new ClientHandler(clientSocket).start();
            }
        } catch (IOException ex) {
            System.out.println("Server exception: " + ex.getMessage());
            ex.printStackTrace();
        }
    }
}
class ClientHandler extends Thread {
    private Socket clientSocket;

    public ClientHandler(Socket socket) {
        this.clientSocket = socket;
    }

    @Override
    public void run() {
        try (BufferedReader input = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
             PrintWriter output = new PrintWriter(clientSocket.getOutputStream(), true)) {

            String clientMessage;
            while ((clientMessage = input.readLine()) != null) {
                System.out.println("Client says: " + clientMessage);
                output.println("Echo: " + clientMessage);
            }
        } catch (IOException ex) {
            System.out.println("Client handler exception: " + ex.getMessage());
            ex.printStackTrace();
        }
    }
```

}

Explanation:

- The server handles multiple clients by creating a new ClientHandler thread for each incoming connection.
- This allows concurrent communication with multiple clients.

Practical Example: Building a Simple Server-Client Communication System

1. Server Code:

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

public class SimpleServer {
    public static void main(String[] args) {
        try (ServerSocket serverSocket = new ServerSocket(12345)) {
            System.out.println("Server is listening on port 12345...");
            Socket socket = serverSocket.accept(); // Accept a client connection
            System.out.println("Client connected!");

            // Set up input and output streams
            BufferedReader input = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            PrintWriter output = new PrintWriter(socket.getOutputStream(), true);

            // Read data from the client and respond
            String clientMessage = input.readLine();
            System.out.println("Client says: " + clientMessage);
            output.println("Hello Client! You said: " + clientMessage);

            // Close the socket
            socket.close();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

2. Client Code:

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

public class SimpleClient {
    public static void main(String[] args) {
        try (Socket socket = new Socket("localhost", 12345)) {
            // Set up input and output streams
```

```
            BufferedReader input = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            PrintWriter output = new PrintWriter(socket.getOutputStream(), true);

            // Send a message to the server
            output.println("Hello Server!");

            // Receive the server's response
            String serverMessage = input.readLine();
            System.out.println("Server says: " + serverMessage);
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

Exercises

1. **Exercise 1**: Modify the server to handle multiple messages from a client before closing the connection.
2. **Exercise 2**: Create a client-server application where the server performs basic arithmetic operations based on the client's requests.

Overview

This section introduces two important concepts in network programming using Java: **concurrent servers** and **URL/URLConnection classes**. These concepts enable the development of robust, efficient, and scalable networked applications that can handle multiple client requests and interact with web resources.

1. Concurrent Servers

A **concurrent server** is a server capable of handling multiple client requests at the same time. Instead of processing one client connection at a time, a concurrent server creates a new thread for each client that connects to it. This allows the server to continue listening for and accepting new connections while serving existing clients in parallel. Concurrent servers are essential for building scalable applications, such as web servers, chat servers, and multiplayer game servers.

Why Use Concurrent Servers?

- **Scalability**: Without concurrency, a server would have to process client requests one at a time, which leads to poor performance and potential bottlenecks as client numbers grow.
- **Responsiveness**: Concurrent servers ensure that each client receives attention without waiting for other clients' interactions to complete, providing a better user experience.
- **Real-World Example**: Consider a web server that handles requests to load web pages. With a concurrent server, multiple users can browse and load pages simultaneously without waiting for each other.

How Concurrent Servers Work in Java

In Java, creating a concurrent server typically involves:

- Using a ServerSocket to listen for incoming client connections.
- Creating a new thread (or using thread pools) for each client that connects. This thread handles all communication with the client, while the main server thread continues to listen for new connections.

Code Example:

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

public class MultiThreadedServer {
   public static void main(String[] args) {
      try (ServerSocket serverSocket = new ServerSocket(12345)) {
         System.out.println("Server is running on port 12345...");

         while (true) {
            // Accept incoming client connections
            Socket clientSocket = serverSocket.accept();
```

```
            // Create and start a new thread to handle the client
            new ClientHandler(clientSocket).start();
        }
    } catch (IOException ex) {
        ex.printStackTrace();
    }
  }
}
class ClientHandler extends Thread {
  private Socket clientSocket;

  public ClientHandler(Socket socket) {
    this.clientSocket = socket;
  }

  @Override
  public void run() {
    try (BufferedReader input = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
         PrintWriter output = new PrintWriter(clientSocket.getOutputStream(), true)) {

        String message;
        while ((message = input.readLine()) != null) {
          System.out.println("Client says: " + message);
          output.println("Echo: " + message);
        }
    } catch (IOException ex) {
        ex.printStackTrace();
    }
  }
}
```

Explanation:

- The server listens for client connections on port 12345.
- For each client connection, a new ClientHandler thread is created to manage communication with that client.
- This structure allows multiple clients to connect and communicate with the server concurrently.

2. URL and URLConnection Classes

The **URL** and **URLConnection** classes in Java are part of the java.net package. These classes make it easy to interact with web resources, such as retrieving data from web pages, sending requests, and reading responses.

What is a URL?

URL (Uniform Resource Locator) is a reference (an address) to a resource on the internet. A typical URL might look like https://www.example.com, specifying the protocol (https), domain (www.example.com), and potentially a path to a specific resource (e.g., /index.html).

The URL Class

The URL class represents a URL and provides methods to access various parts of the URL (such as the protocol, host, file path, etc.) and open connections to the resource it points to.

Example:

```
import java.net.*;

public class URLExample {
   public static void main(String[] args) {
      try {
         URL url = new URL("https://www.example.com");
         System.out.println("Protocol: " + url.getProtocol());
         System.out.println("Host: " + url.getHost());
         System.out.println("Port: " + url.getPort());
         System.out.println("File: " + url.getFile());
      } catch (MalformedURLException ex) {
         ex.printStackTrace();
      }
   }
}
```

Explanation:

- The URL object is created with a specified web address.
- Various methods, such as getProtocol(), getHost(), and getFile(), allow you to extract parts of the URL.

The URLConnection Class

URLConnection is an abstract class that represents a communication link between the application and a URL. It provides methods for sending requests to and reading responses from web servers.

Example:

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

public class URLConnectionExample {
   public static void main(String[] args) {
      try {
         URL url = new URL("https://www.example.com");
         URLConnection connection = url.openConnection();

         // Reading data from the URL
```

```
            BufferedReader input = new BufferedReader(new
InputStreamReader(connection.getInputStream()));
            String line;
            while ((line = input.readLine()) != null) {
                System.out.println(line);
            }
            input.close();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

Explanation:

- The URLConnection object establishes a connection to the specified URL.
- Using input streams, you can read data returned by the server.

Practical Example: Creating a Multi-Threaded Server

1. Multi-Threaded Server Code:

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

public class MultiThreadedServer {
    public static void main(String[] args) {
        try (ServerSocket serverSocket = new ServerSocket(12345)) {
            System.out.println("Multi-threaded Server is running on port 12345...");

            while (true) {
                Socket clientSocket = serverSocket.accept();
                new ClientHandler(clientSocket).start(); // Handle each client in a separate thread
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}

class ClientHandler extends Thread {
    private Socket clientSocket;

    public ClientHandler(Socket socket) {
        this.clientSocket = socket;
    }

    @Override
```

```
    public void run() {
        try (BufferedReader input = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
             PrintWriter output = new PrintWriter(clientSocket.getOutputStream(), true)) {

            String clientMessage;
            while ((clientMessage = input.readLine()) != null) {
                System.out.println("Client says: " + clientMessage);
                output.println("Echo: " + clientMessage);
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

Practical Example: Using URL and URLConnection

Fetching Data from a Web Page:

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

public class URLExample {
    public static void main(String[] args) {
        try {
            URL url = new URL("https://www.example.com");
            URLConnection connection = url.openConnection();
            BufferedReader input = new BufferedReader(new
InputStreamReader(connection.getInputStream()));

            String line;
            while ((line = input.readLine()) != null) {
                System.out.println(line);
            }
            input.close();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

Exercises

1. **Exercise 1**: Modify the multi-threaded server to keep track of connected clients.
2. **Exercise 2**: Create a program that fetches and displays data from a user-specified URL.

Week 13: Input/Output Streams & Multimedia Programming

Introduction

In this week's content, we delve into the fundamental aspects of **Java's input/output (I/O) classes**, which provide essential functionality for handling data input and output operations. Using I/O streams, you can read from and write to files, manage data flows, and perform various types of data processing in Java applications. We will also explore **serialization**, a powerful feature that allows you to save and retrieve Java objects while preserving their state, making it easier to persist data or transmit it across a network. Finally, we will touch on **multimedia programming**, focusing on working with images and understanding how to display and manipulate them within Java applications. This combination of topics provides a solid foundation for building data-driven and media-rich applications in Java

1. Java I/O Classes

Java's I/O classes provide a flexible and powerful way to perform input and output operations. These classes are part of the java.io package and allow you to work with files, data streams, character streams, and more. The I/O system in Java uses **streams**, which represent an ordered sequence of data. There are two main types of streams:

- **Input Streams**: Used to read data (e.g., reading data from a file).
- **Output Streams**: Used to write data (e.g., writing data to a file).

Example of File I/O using Streams:

Reading from a File:

```
import java.io.*;
public class FileReadExample {
    public static void main(String[] args) {
        try (BufferedReader reader = new BufferedReader(new FileReader("input.txt"))) {
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

```java
import java.io.*;

public class FileReadExample {
    public static void main(String[] args) {
        try (BufferedReader reader = new BufferedReader(new FileReader("input.txt"))) {
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

Explanation:

- This example demonstrates reading data from a file called input.txt.
- The BufferedReader and FileReader classes are used to read lines of text efficiently.

Writing to a File:

```
import java.io.*;
public class FileWriteExample {
   public static void main(String[] args) {
      try (BufferedWriter writer = new BufferedWriter(new FileWriter("output.txt"))) {
         writer.write("This is an example of writing to a file.");
         writer.newLine();
         writer.write("Java makes file I/O simple.");
      } catch (IOException ex) {
         ex.printStackTrace();
      }
   }
}
```

```java
import java.io.*;

public class FileWriteExample {
    public static void main(String[] args) {
        try (BufferedWriter writer = new BufferedWriter(new FileWriter("output.txt"))) {
            writer.write("This is an example of writing to a file.");
            writer.newLine();
            writer.write("Java makes file I/O simple.");
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

Explanation:

- The BufferedWriter and FileWriter classes are used to write text to a file called output.txt.
- newLine() inserts a line separator, making it easy to structure written content.

2. Serialization

Serialization is the process of converting an object's state into a byte stream so that it can be saved to a file or transmitted over a network. This allows you to store the state of an object and later restore it, maintaining all of its data.

- **Why Use Serialization?**: Serialization is useful for persisting data, sending objects over a network, or storing objects in files.
- **Serializable Interface**: For a class to be serializable, it must implement the Serializable interface from the java.io package.

Example of Serialization and Deserialization:

```
import java.io.*;
public class Person implements Serializable {
   private String name;
   private int age;

   public Person(String name, int age) {
      this.name = name;
      this.age = age;
   }
   public static void main(String[] args) {
      // Serialize the object
      try (ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("person.ser"))) {
         Person person = new Person("Alice", 30);
         out.writeObject(person);
         System.out.println("Object serialized successfully.");
      } catch (IOException ex) {
         ex.printStackTrace();
      }
      // Deserialize the object
      try (ObjectInputStream in = new ObjectInputStream(new FileInputStream("person.ser"))) {
         Person deserializedPerson = (Person) in.readObject();
         System.out.println("Name: " + deserializedPerson.name + ", Age: " + deserializedPerson.age);
      } catch (IOException | ClassNotFoundException ex) {
         ex.printStackTrace();
      }
   }
}
```

Explanation:

- **Serialization**: The ObjectOutputStream writes the object person to a file called person.ser.
- **Deserialization**: The ObjectInputStream reads the object back from the file and restores its state.

3. Multimedia Programming

Multimedia programming in Java often involves working with images, audio, video, and other media types. For this week, we will focus on **image processing**, which allows you to load, display, and manipulate images in Java.

Example of Basic Image Handling:

Displaying an Image in a GUI Application:

```
import javax.swing.*;
import java.awt.*;
public class ImageDisplay extends JFrame {
   public ImageDisplay() {
      ImageIcon icon = new ImageIcon("example.jpg");
      JLabel label = new JLabel(icon);
      add(label);

      setTitle("Image Display Example");
      setSize(400, 300);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setVisible(true);
   }

   public static void main(String[] args) {
      new ImageDisplay();
   }
}
```

```java
import javax.swing.*;
import java.awt.*;

public class ImageDisplay extends JFrame {
    public ImageDisplay() {
        ImageIcon icon = new ImageIcon("example.jpg");
        JLabel label = new JLabel(icon);
        add(label);

        setTitle("Image Display Example");
        setSize(400, 300);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }

    public static void main(String[] args) {
        new ImageDisplay();
    }
}
```

Explanation:

- This example uses the ImageIcon class to load an image (example.jpg).
- The image is displayed in a GUI window using JLabel in a JFrame.

Manipulating Images (Basic Processing):

You can manipulate images by using libraries such as javax.imageio for reading/writing images, or third-party libraries like **Java Advanced Imaging (JAI)** for more complex processing tasks.

Practical Example: File I/O and Serialization

Reading from a File:

```
import java.io.*;

public class FileReadExample {
    public static void main(String[] args) {
        try (BufferedReader reader = new BufferedReader(new FileReader("example.txt"))) {
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

Writing to a File:

```
import java.io.*;

public class FileWriteExample {
    public static void main(String[] args) {
        try (BufferedWriter writer = new BufferedWriter(new FileWriter("output.txt"))) {
            writer.write("Hello, this is a test!");
            writer.newLine();
            writer.write("Writing to a file in Java is easy.");
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

Serialization Example:

```
import java.io.*;

public class SerializeExample implements Serializable {
    private String name;
    private int age;

    public SerializeExample(String name, int age) {
        this.name = name;
```

```
        this.age = age;
    }

    public static void main(String[] args) {
        SerializeExample person = new SerializeExample("Alice", 30);

        // Serialize the object
        try (ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("person.ser"))) {
            out.writeObject(person);
        } catch (IOException ex) {
            ex.printStackTrace();
        }

        // Deserialize the object
        try (ObjectInputStream in = new ObjectInputStream(new FileInputStream("person.ser"))) {
            SerializeExample deserializedPerson = (SerializeExample) in.readObject();
            System.out.println("Name: " + deserializedPerson.name + ", Age: " + deserializedPerson.age);
        } catch (IOException | ClassNotFoundException ex) {
            ex.printStackTrace();
        }
    }
}
```

Practical Example: Basic Image Processing

Loading and Displaying an Image:

```
import javax.swing.*;
import java.awt.*;

public class ImageExample extends JFrame {
    public ImageExample() {
        ImageIcon icon = new ImageIcon("image.jpg");
        JLabel label = new JLabel(icon);
        add(label);

        setTitle("Image Example");
        setSize(400, 300);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }

    public static void main(String[] args) {
        new ImageExample();
    }
}
```

Exercises

1. **Exercise 1**: Create a program to read and write text files.
2. **Exercise 2**: Implement a basic image manipulation tool (e.g., resizing or changing brightness).