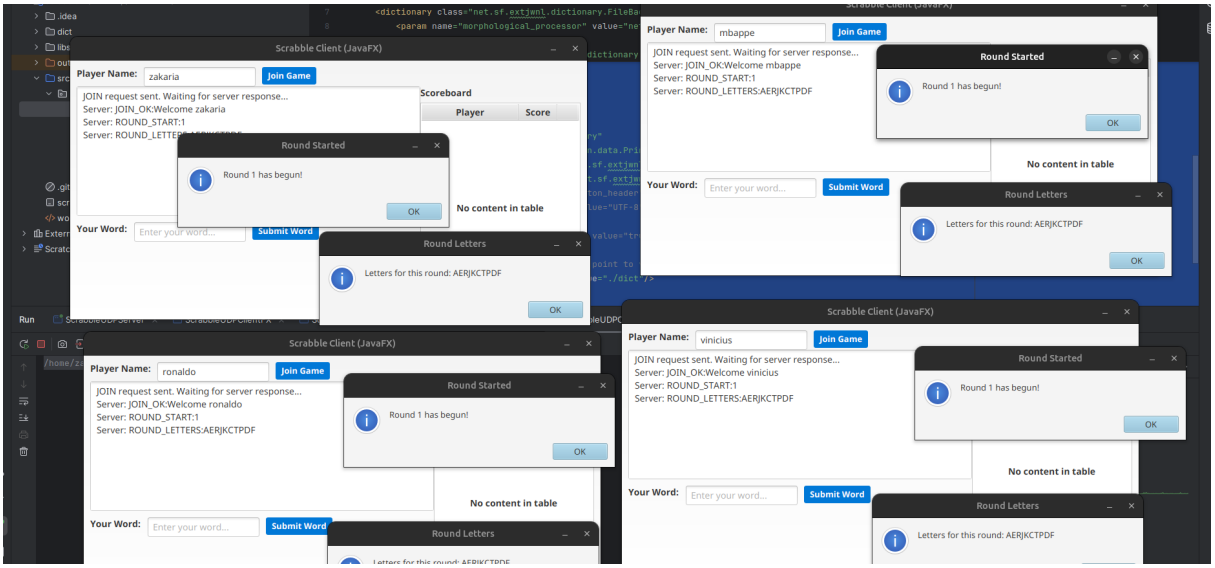

UDP Server Scrabble Game

This game is made using UDP (I know TCP is a better protocol for a game like this), requirements state clearly that this game needs to be a UDP implementation . I m going to go through code, explain why some decisions have been made, why i chose an approach rather than other ones, libraries chosen, and explain the logic in general .

Library Choices and Configuration

Why These Libraries ?

`net.sf.extjwnl` **(extJWNL - Extended Java WordNet Library):**

- **Purpose:** Facilitates interaction with WordNet, a lexical database for the English language, to validate words.
- **Why Not Alternatives?** extJWNL offers robust features and flexibility for word validation, making it suitable for a Scrabble game that requires checking the validity of player-submitted words.

`net.sf.extjwnl.dictionary.Dictionary` **and Related Classes:**

- **Purpose:** Core components of extJWNL used to load and interact with the WordNet dictionary.
- **Configuration:** Requires a properties file (`wordnet_properties.xml`) to define how the dictionary is loaded and accessed.

NOTE : if you have a maven project, the implementation of extJWNL is FAR more easier, no setup is needed, you can just instantiate the class Dictionary after downloading the “dict” needed folder, this can be done from the wordNet website.

Configuring extJWNL :

- **WordValidator** Class :

```
package scrabbleGame;

import net.sf.extjwnl.dictionary.Dictionary;
import net.sf.extjwnl.JWNLException;

import java.io.FileInputStream;
import java.io.IOException;

public class WordValidator {
    private Dictionary dictionary;

    public WordValidator() throws JWNLException, IOException {
        // Initialize extJWNL with prop file
        dictionary = Dictionary.getInstance(new FileInputStream("wordnet_properties.xml"));
    }

    // Method to validate words...
}
```

`wordnet_properties.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- extJWNL properties configuration -->
<jwnl_properties language="en">
    <!-- Configuration details -->
</jwnl_properties>
```

Explanation:

- **Initialization:**
  - **Purpose:** The `WordValidator` class initializes the WordNet dictionary using extJWNL.
  - **Process:** It reads the `wordnet_properties.xml` file, which contains configuration details like dictionary paths, morphological processors, and file managers.
- **Configuration File (`wordnet_properties.xml`):**
  - **Purpose:** Defines how extJWNL interacts with the WordNet data.
  - **Key Components:**
    - **Morphological Processor:** Handles word forms and morphological variations.
    - **Dictionary Element Factory:** Determines how dictionary elements are created and accessed.
    - **File Manager:** Manages file access, specifying paths to the WordNet data files.
- **Why Use a Properties File?**
  - **Flexibility:** Allows easy modification of dictionary settings without altering the code.
  - **Separation of Concerns:** Keeps configuration separate from application logic, enhancing maintainability.

Server-Side Implementation

Overview of `ScrabbleUDPServer`

The `ScrabbleUDPServer` class is responsible for managing the game logic, handling player connections, coordinating rounds, validating words, and maintaining scores. It leverages UDP (User Datagram Protocol) for communication, ensuring lightweight and efficient data transfer between the server and clients.

Key Components and Their Roles

1. **Constants:**
   - `SERVER_PORT`: The port on which the server listens for incoming UDP packets.
   - `MAX_PLAYERS`: Maximum number of players allowed in a game.
   - `NUM_ROUNDS`: Total number of rounds in the game.
2. **Data Structures:**
   - `players`: A synchronized `HashMap` mapping player identifiers (`"ip:port"`) to their scores.
   - `playerNames`: A synchronized `HashMap` mapping player identifiers to their chosen names.
   - `roundSubmissionMap`: A synchronized `HashMap` tracking each player's submitted word per round.
3. **Game State Variables:**
   - `gameStarted`: A `volatile` boolean indicating whether the game has commenced.
   - `currentRound`: A `volatile` integer tracking the current round number.
4. **Word Validation:**
   - `wordValidator`: An instance of `WordValidator` to check the validity of submitted words.
5. **Networking:**
   - `serverSocket`: A `DatagramSocket` used for sending and receiving UDP packets.

Detailed Breakdown :

- **Main Method and Server Start-Up :**

```
public static void main(String[] args) {
    ScrabbleUDPServer server = new ScrabbleUDPServer();
    server.start();
}
```

Purpose: Entry point of the server application.

Process: Instantiates the server and invokes the `start()` method to begin operations

- **start() Method :**

```
public void start() {
    try {
        // Initialize dictionary
        wordValidator = new WordValidator();

        // Create and bind the server socket
        serverSocket = new DatagramSocket(SERVER_PORT);
```

```
        System.out.println("UDP Server started on port " + SERVER_PORT);

        // Start a thread to read console input (for manual "start" command)
        Thread consoleThread = new Thread(this::handleConsoleInput);
        consoleThread.start();

        // Start a thread to listen for UDP packets
        Thread udpListenerThread = new Thread(this::udpListenerLoop);
        udpListenerThread.start();

        // Wait until the game starts (either "start" typed or 4 players joined)
        synchronized (this) {
            while (!gameStarted) {
                wait(); // Wait until we're notified that gameStarted = true
            }
        }

        // Now run the 5 rounds
        for (int round = 1; round <= NUM_ROUNDS; round++) {
            currentRound = round;
            // Create a new submissions map for this round
            roundSubmissionMap.put(round, Collections.synchronizedMap(new HashMap<>()));

            // Generate letters
            String letters = generateRandomLetters(10);
            System.out.println("Round " + round + " letters: " + letters);

            // Broadcast to all players
            broadcastMessage("ROUND_START:" + round);
            broadcastMessage("ROUND_LETTERS:" + letters);

            // Wait for all players to submit (no timeouts).
            waitForAllSubmissions(round);

            // Validate & score
            scoreRound(round, letters);

            // Broadcast round results
            broadcastScores("ROUND_OVER:" + round);
        }

        // Announce winner
        broadcastFinalWinner();
        System.out.println("Game finished. Shutting down.");

        // Close socket
        serverSocket.close();
        System.exit(0);

    } catch (IOException | JWNLException | InterruptedException e) {
        e.printStackTrace();
    }
}
```

Explanation:

1. **Initialization:**
   - **Word Validator:** Initializes `wordValidator` to enable word validation using WordNet.
   - **Server Socket:** Binds a `DatagramSocket` to `SERVER_PORT` to listen for incoming UDP packets.
2. **Concurrency:**
   - **Console Thread:** Starts a separate thread (`consoleThread`) to handle server console inputs, allowing the admin to manually start the game by typing "start".
   - **UDP Listener Thread:** Starts another thread (`udpListenerThread`) to continuously listen for UDP packets from clients.
3. **Game Start Synchronization:**

- **Wait Mechanism:** Uses `synchronized` and `wait()` to pause the main thread until `gameStarted` becomes `true`, either via manual input or automatic when `MAX_PLAYERS` are joined.

4. **Game Rounds:**
   - **Loop:** Iterates through the defined number of rounds (`NUM_ROUNDS`).
   - **Letter Generation:** Generates 10 random letters for each round.
   - **Broadcasting:** Sends round start and letters to all players.
   - **Submission Waiting:** Blocks until all players have submitted their words.
   - **Scoring:** Validates and scores each player's submission.
   - **Round Results:** Broadcasts the updated scores to all players.
5. **Game Conclusion:**
   - **Winner Announcement:** Determines and announces the final winner or a tie.
   - **Cleanup:** Closes the server socket and terminates the application.

Interesting Points:

- **Threading:** Demonstrates handling multiple tasks concurrently (listening for packets and console input).
- **Synchronization:** Illustrates using `synchronized`, `wait()`, and `notifyAll()` to manage thread coordination.
- **Exception Handling:** Emphasizes the importance of catching and handling exceptions to prevent server crashes.

ment this organization, you would need to select the content and use AI to help restructure it through the edit menu.

- Handling Console Input (`handleConsoleInput`) :

```
private void handleConsoleInput() {
    try (BufferedReader reader = new BufferedReader(new InputStreamReader(System.in))) {
        while (!gameStarted) {
            String line = reader.readLine();
            if (line != null && line.trim().equalsIgnoreCase("start")) {
                synchronized (this) {
                    gameStarted = true;
                    notifyAll();
                }
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Explanation:

- **Purpose:** Allows the server administrator to manually start the game by typing "start" into the console.
- **Process:**

- Continuously reads lines from the console.
- If "start" is entered, sets `gameStarted` to `true` and notifies the main thread to proceed.

Interesting Points:

- **Resource Management:** Uses try-with-resources to ensure the `BufferedReader` is closed properly.
- **User Interaction:** Shows how to incorporate console-based commands for server control.

- Listening for UDP Packets (`udpListenerLoop`) :

```
private void udpListenerLoop() {
    byte[] buf = new byte[1024];

    while (true) {
        try {
            DatagramPacket packet = new DatagramPacket(buf, buf.length);
            serverSocket.receive(packet);

            String msg = new String(packet.getData(), 0, packet.getLength()).trim();
            String clientKey = packet.getAddress().getHostAddress() + ":" + packet.getPort();

            // Handle the message in another method
            handleClientMessage(msg, clientKey, packet);
        } catch (SocketException e) {
            // This happens when socket is closed. We can just break the loop.
            System.out.println("Server socket closed, UDP listener thread terminating.");
            break;
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Explanation:

- **Purpose:** Continuously listens for incoming UDP packets from clients.
- **Process:**
  - Receives a packet.
  - Extracts the message and identifies the client using their IP and port.
  - Delegates message handling to `handleClientMessage()`.

Interesting Points:

- **Networking Basics:** Introduces how UDP works (connectionless, lightweight).
- **Packet Handling:** Demonstrates receiving and parsing UDP packets.

- Handling Client Messages (`handleClientMessage`) :

```
private void handleClientMessage(String msg, String clientKey, DatagramPacket packet) {
    // If game not started, handle JOIN
    if (!gameStarted) {
        if (msg.startsWith("JOIN:")) {
            handleJoin(msg, clientKey, packet);
        }
```

```
    } else {
        // The game has started, handle "WORD:xxxx" if it's relevant to current round
        if (msg.startsWith("WORD:")) {
            handleWordSubmission(msg, clientKey, packet);
        }
    }
}
```

Explanation:

- **Purpose:** Determines the type of message received and routes it to the appropriate handler.
- **Process:**
  - If the game hasn't started, looks for "JOIN:" messages to handle new player connections.
  - If the game is in progress, looks for "WORD:" messages representing player submissions.

Interesting Points:

- **Message Protocol:** Illustrates how defining a clear message format (e.g., "JOIN:", "WORD:") aids in parsing and handling communications.
- **State Management:** Shows how the server's behavior changes based on the game's state (`gameStarted`).

- Handling Player Joins (`handleJoin`) :

```
private void handleJoin(String msg, String clientKey, DatagramPacket packet) {
    // parse out the player's name
    String playerName = msg.substring("JOIN:".length()).trim();

    synchronized (this) {
        if (players.size() < MAX_PLAYERS && !players.containsKey(clientKey)) {
            players.put(clientKey, 0);
            playerNames.put(clientKey, playerName); // Store the player's name
            System.out.println("Player joined -> " + playerName + " at " + clientKey);
            sendMessage(packet.getAddress(), packet.getPort(),
                    "JOIN_OK:Welcome " + playerName);
        } else {
            // Lobby is full or player already joined
            sendMessage(packet.getAddress(), packet.getPort(), "JOIN_REJECT:Lobby Full or Duplicate");
        }

        // If we just hit 4 players, auto-start the game
        if (!gameStarted && players.size() == MAX_PLAYERS) {
            gameStarted = true;
            notifyAll();  // wake up main thread
        }
    }
}
```

Explanation:

- **Purpose:** Manages new player connections before the game starts.
- **Process:**
  - Extracts the player's name from the message.
  - Checks if the lobby isn't full and the player isn't already joined.

- Adds the player to the `players` and `playerNames` maps.
- Sends a confirmation or rejection message to the client.
- Automatically starts the game if `MAX_PLAYERS` are reached.

Interesting Points:

- **Concurrency Control:** Uses `synchronized` to ensure thread-safe operations on shared data structures.
- **Client Identification:** Combines IP and port to uniquely identify each client.
- **Feedback Mechanism:** Sends appropriate responses to clients based on their join request status.

- Handling Word Submissions (`handleWordSubmission`) :

```
private void handleWordSubmission(String msg, String clientKey, DatagramPacket packet) {
    int round = currentRound;  // volatile read

    // If the current round is 0 (not started) or beyond NUM_ROUNDS, ignore
    if (round < 1 || round > NUM_ROUNDS) {
        return;
    }

    // Extract the word
    String submittedWord = msg.substring("WORD:".length()).trim();

    // Record the submission
    Map<String, String> submissions = roundSubmissionMap.get(round);
    if (submissions != null && !submissions.containsKey(clientKey)) {
        submissions.put(clientKey, submittedWord);
        // Acknowledge receipt
        sendMessage(packet.getAddress(), packet.getPort(), "RECEIVED:Your word for round " + round);

        // Check if that was the last submission needed
        if (submissions.size() == players.size()) {
            synchronized (this) {
                notifyAll();
            }
        }
    }
}
```

Explanation:

- **Purpose:** Processes word submissions from players during an active round.
- **Process:**
  - Validates if the current round is active.
  - Extracts the submitted word from the message.
  - Records the submission in `roundSubmissionMap` if it's the first submission from the player for that round.
  - Sends an acknowledgment to the player.
  - Checks if all players have submitted their words to proceed with scoring.

Interesting Points:

- **Data Validation:** Ensures submissions are only recorded for active rounds and that players don't submit multiple words per round.
- **Synchronization:** Notifies the main thread when all submissions are received, allowing the game to progress.

- Waiting for All Submissions (`waitForAllSubmissions`) :

```
private void waitForAllSubmissions(int round) throws InterruptedException {
    while (true) {
        synchronized (this) {
            Map<String, String> submissions = roundSubmissionMap.get(round);
            if (submissions != null && submissions.size() == players.size()) {
                break; // all players have submitted
            }
            wait(); // wait to be notified when a new submission arrives
        }
    }
}
```

Explanation:

- **Purpose:** Blocks the main thread until all players have submitted their words for the current round.
- **Process:**
  - Continuously checks if the number of submissions matches the number of players.
  - If not, waits until notified (e.g., when a new submission arrives).

Interesting Points:

- **Thread Coordination:** Demonstrates how threads can wait for certain conditions to be met before proceeding.
- **Efficiency:** Avoids busy-waiting by using `wait()` and `notifyAll()` for efficient synchronization.

- Scoring Rounds (`scoreRound`) :

```
private void scoreRound(int round, String letters) {
    Map<String, String> submissions = roundSubmissionMap.get(round);

    for (Map.Entry<String, String> entry : submissions.entrySet()) {
        String playerKey = entry.getKey();
        String submittedWord = entry.getValue();

        boolean valid = validateWord(submittedWord, letters);
        int scoreGain = valid ? submittedWord.length() : 0;

        // Update player's total score
        int oldScore = players.get(playerKey);
        players.put(playerKey, oldScore + scoreGain);
    }
}
```

Explanation:

- **Purpose:** Validates each player's submitted word and updates their scores accordingly.
- **Process:**
  - Iterates through all submissions for the round.
  - Validates each word against the available letters and the WordNet dictionary.
  - Assigns points based on word validity (length of the word if valid).
  - Updates the player's total score.

Interesting Points:

- **Business Logic:** Encapsulates the core game mechanics of validating and scoring words.
- **Modularity:** Keeps scoring logic separate, making the code easier to manage and extend.

- Word Validation (`validateWord` and `canFormWord`) :

```
private boolean validateWord(String word, String letters) {
    // Check that all letters can be formed from the pool
    if (!canFormWord(word.toUpperCase(), letters.toUpperCase())) {
        return false;
    }
    // Check dictionary via WordNet
    return wordValidator.isValidWord(word);
}

private boolean canFormWord(String word, String letters) {
    Map<Character, Integer> freq = new HashMap<>();
    for (char c : letters.toCharArray()) {
        freq.put(c, freq.getOrDefault(c, 0) + 1);
    }
    for (char c : word.toCharArray()) {
        if (!freq.containsKey(c) || freq.get(c) == 0) {
            return false;
        }
        freq.put(c, freq.get(c) - 1);
    }
    return true;
}
```

Explanation:

- `validateWord` :
  - **Purpose:** Ensures the submitted word is both constructible from the available letters and exists in the English dictionary.
  - **Process:**
    - Calls `canFormWord` to check letter availability.
    - Uses `wordValidator` to verify dictionary validity.
- `canFormWord` :
  - **Purpose:** Checks if the word can be formed using the provided letters, considering letter frequencies.
  - **Process:**
    - Creates a frequency map of available letters.

- Iterates through each character in the word, decrementing the frequency count.
- Returns `false` if any letter is insufficient.

Interesting Points:

- **Algorithm Design:** Demonstrates efficient methods for frequency counting and validation.
- **Error Handling:** Prevents invalid or impossible words from affecting game fairness.

- Generating Random Letters (`generateRandomLetters`) :

```
private String generateRandomLetters(int n) {
    Random random = new Random();
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < n; i++) {
        char c = (char) ('A' + random.nextInt(26));
        sb.append(c);
    }
    return sb.toString();
}
```

Explanation:

- **Purpose:** Generates a string of `n` random uppercase letters for each round.
- **Process:**
  - Uses `Random` to select letters uniformly from 'A' to 'Z'.
  - Constructs the string using `StringBuilder`.

Interesting Points:

- **Randomization:** Shows how to generate random data within a specific range.
- **Performance:** Highlights the use of `StringBuilder` for efficient string concatenation in loops.

- Broadcasting Messages (`broadcastMessage`, `sendMessage`) :

```
private void broadcastMessage(String message) {
    synchronized (players) {
        for (String clientKey : players.keySet()) {
            String[] parts = clientKey.split(":");
            try {
                InetAddress addr = InetAddress.getByName(parts[0]);
                int port = Integer.parseInt(parts[1]);
                sendMessage(addr, port, message);
            } catch (UnknownHostException e) {
                e.printStackTrace();
            }
        }
    }
}

private void sendMessage(InetAddress address, int port, String message) {
    byte[] data = message.getBytes();
    DatagramPacket packet = new DatagramPacket(data, data.length, address, port);
    try {
        serverSocket.send(packet);
    } catch (IOException e) {
        e.printStackTrace();
```

```
    }
}
```

Explanation:

- `broadcastMessage`:
  - **Purpose:** Sends a message to all connected players.
  - **Process:**
    - Iterates through all player keys.
    - Parses each client's IP and port.
    - Calls `sendMessage` to deliver the message.
- `sendMessage`:
  - **Purpose:** Sends a single UDP packet to a specified client.
  - **Process:**
    - Converts the message to bytes.
    - Constructs a `DatagramPacket` with the target address and port.
    - Sends the packet via `serverSocket`.

Interesting Points:

- **Networking Fundamentals:** Illustrates sending data over UDP, including addressing and packet construction.
- **Error Handling:** Handles potential `UnknownHostException` and `IOException` during message delivery.

- Broadcasting Scores and Final Results (`broadcastScores`, `broadcastFinalWinner`) :

```
private void broadcastScores(String prefix) {
    // Build scoreboard message
    StringBuilder sb = new StringBuilder();
    sb.append(prefix).append("\n");
    sb.append("Current Scores:\n");
    synchronized (players) {
        for (Map.Entry<String, Integer> entry : players.entrySet()) {
            String clientKey = entry.getKey();
            int score = entry.getValue();
            String playerName = playerNames.get(clientKey); // Get player name
            sb.append(playerName).append(" -> ").append(score).append(" points\n");
        }
    }
    broadcastMessage(sb.toString());
}

private void broadcastFinalWinner() {
    int maxScore = -1;
    List<String> winners = new ArrayList<>();
    synchronized (players) {
        for (Map.Entry<String, Integer> entry : players.entrySet()) {
            int score = entry.getValue();
            String playerName = playerNames.get(entry.getKey()); // Get player name
            if (score > maxScore) {
                maxScore = score;
                winners.clear();
```

```
                winners.add(playerName); // Use player name
            } else if (score == maxScore) {
                winners.add(playerName); // Use player name
            }
        }
    }
    String result;
    if (winners.size() == 1) {
        result = "WINNER:" + winners.get(0) + ":" + maxScore;
    } else {
        result = "TIE:" + winners + ":" + maxScore;
    }
    broadcastMessage(result);
}
```

Explanation:

- `broadcastScores`:
  - **Purpose:** Sends the current scores to all players after each round.
  - **Process:**
    - Constructs a multi-line message with player names and their scores.
    - Uses `broadcastMessage` to send the scoreboard.
- `broadcastFinalWinner`:
  - **Purpose:** Determines and announces the final winner(s) at the end of the game.
  - **Process:**
    - Finds the maximum score and identifies all players who achieved it.
    - Constructs a message indicating the winner or a tie.
    - Sends the final result to all players.

Interesting Points:

- **Data Aggregation:** Shows how to process and summarize data from multiple sources (player scores).
- **Conditional Logic:** Handles scenarios with single winners and ties.
- **User Feedback:** Emphasizes the importance of communicating game status and results to players.

Word Validation with `WordValidator`

Overview

The `WordValidator` class encapsulates the logic for verifying the validity of words submitted by players. It leverages the extJWNL library to interact with the WordNet dictionary, ensuring that only legitimate English words are accepted.

Detailed Breakdown :

```
package scrabbleGame;

import net.sf.extjwnl.dictionary.Dictionary;
import net.sf.extjwnl.data.IndexWord;
```

```java
import net.sf.extjwnl.data.POS;
import net.sf.extjwnl.JWNLException;

import java.io.FileInputStream;
import java.io.IOException;

/**
 * A class to validate words using extJWNL + WordNet.
 */
public class WordValidator {
    private Dictionary dictionary;

    public WordValidator() throws JWNLException, IOException {
        // Initialize extJWNL with prop file
        dictionary = Dictionary.getInstance(new FileInputStream("wordnet_properties.xml"));
    }

    /**
     * Checks if the given word is a valid English word.
     * We'll try NOUN and VERB
     *
     * @param word The word to validate.
     * @return true if valid, false if not.
     */
    public boolean isValidWord(String word) {
        if (word == null || word.trim().isEmpty()) {
            return false;
        }
        String lower = word.toLowerCase();
        try {
            IndexWord indexNoun = dictionary.lookupIndexWord(POS.NOUN, lower);
            if (indexNoun != null) {
                return true;
            }
            IndexWord indexVerb = dictionary.lookupIndexWord(POS.VERB, lower);
            if (indexVerb != null) {
                return true;
            }
        } catch (JWNLException e) {
            e.printStackTrace();
        }
        return false;
    }
}
```

Explanation:

- **Initialization:**
  - **Dictionary Instance:** Loads the WordNet dictionary using the provided properties file.
  - **Error Handling:** Throws exceptions if the dictionary fails to load, ensuring that the application is aware of configuration issues.
- **Word Validation (`isValidWord`):**
  - **Purpose:** Determines if a given word exists in the English language.
  - **Process:**
    - **Null or Empty Check:** Immediately rejects invalid inputs.
    - **Lower casing:** Converts the word to lowercase for consistent dictionary lookup.
    - **Lookup:** Attempts to find the word as both a noun and a verb.
    - **Result:** Returns `true` if found in either category; `false` otherwise.

Interesting Points:

- **Dictionary Lookup:** Demonstrates how to interact with WordNet to verify word existence and categories.
- **Robustness:** Ensures that only meaningful words (nouns and verbs) are accepted, preventing the use of obscure or non-standard words.
- **Error Handling:** Highlights the importance of managing exceptions during external resource interactions.

Client-Side Implementation

Overview of `ScrabbleUDPClientFX`

The `ScrabbleUDPClientFX` class represents the client-side application built using JavaFX. It provides a graphical user interface (GUI) for players to join the game, receive round information, submit words, and view scores.

Key Components and Their Roles

1. **Networking:**
   - **DatagramSocket (`socket`):** Used for sending and receiving UDP packets to/from the server.
   - **AtomicBoolean (`running`):** Manages the listener thread's running state.
2. **UI Elements:**
   - **Text Fields and Buttons:** For player name input, joining the game, and word submissions.
   - **TextArea (`serverMessagesArea`):** Displays server messages and game updates.
   - **TableView (`scoreTable`):** Shows the scoreboard with player names and scores.
3. **Concurrency:**
   - **Listener Thread:** Runs in the background to receive messages from the server without freezing the UI.
4. **Model:**
   - `PlayerScore` **Class:** Represents each player's score for display in the `TableView`.

Detailed Breakdown

Main Method and Application Start :

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

- **Purpose:** Entry point for the JavaFX application.
- **Process:** Calls `launch()`, which initializes the JavaFX runtime and invokes the `start()` method.

`start()` Method :

```
@Override
public void start(Stage primaryStage) {
    primaryStage.setTitle("Scrabble Client (JavaFX)");

    // Main layout as a BorderPane, so we can place scoreboard on the right or bottom
    BorderPane root = new BorderPane();

    // Top Section: Join box
    HBox joinBox = new HBox(10);
    Label nameLabel = new Label("Player Name:");
    nameField = new TextField();
    nameField.setPromptText("Enter your name...");
    joinButton = new Button("Join Game");
    joinButton.setOnAction(e -> handleJoin());
    joinBox.getChildren().addAll(nameLabel, nameField, joinButton);
    root.setTop(joinBox);

    // Center: TextArea for messages & Word submission
    VBox centerBox = new VBox(10);

    // Server messages
    serverMessagesArea = new TextArea();
    serverMessagesArea.setEditable(false);
    serverMessagesArea.setPrefHeight(200);
    serverMessagesArea.setWrapText(true);

    // Word submission
    HBox wordBox = new HBox(10);
    Label wordLabel = new Label("Your Word:");
    wordField = new TextField();
    wordField.setPromptText("Enter your word...");
    submitWordButton = new Button("Submit Word");
    submitWordButton.setOnAction(e -> handleSubmitWord());
    submitWordButton.setDisable(true); // disabled until we receive letters
    wordBox.getChildren().addAll(wordLabel, wordField, submitWordButton);

    centerBox.getChildren().addAll(serverMessagesArea, wordBox);
    root.setCenter(centerBox);

    // Right Side: Scoreboard
    scoreData = FXCollections.observableArrayList();
    scoreTable = createScoreTable();
    // Place scoreboard in a VBox or directly set as right node
    VBox scoreBox = new VBox(new Label("Scoreboard"), scoreTable);
    scoreBox.setSpacing(5);
    root.setRight(scoreBox);

    // Scene + load CSS
    Scene scene = new Scene(root, 800, 400);  // a bit wider to accommodate the table
    scene.getStylesheets().add(getClass().getResource("style.css").toExternalForm());
    primaryStage.setScene(scene);
    primaryStage.show();

    // Initialize UDP socket & start listener
    try {
        socket = new DatagramSocket();
        socket.setReuseAddress(true);
        running.set(true);

        Thread listenerThread = new Thread(this::listenForServer);
        listenerThread.setDaemon(true);
        listenerThread.start();
    } catch (SocketException ex) {
        showMessage("ERROR: Unable to open UDP socket: " + ex.getMessage());
        ex.printStackTrace();
    }
}
```

Explanation:

1. **UI Layout:**

- **BorderPane:** Divides the window into top, center, and right sections.
- **Top (Join Box):** Contains input fields for the player's name and a button to join the game.
- **Center:** Displays server messages and allows word submissions.
- **Right (Scoreboard):** Shows the current scores of all players in a table format.

2. **Styling:**
   - **CSS:** Applies styles from `style.css` to enhance the UI's appearance.
3. **Networking:**
   - **DatagramSocket:** Initializes a UDP socket for communication.
   - **Listener Thread:** Starts a background thread to listen for incoming server messages without blocking the UI.

Interesting Points:

- **JavaFX Layouts:** Demonstrates the use of `BorderPane`, `HBox`, and `VBox` for organizing UI components.
- **Event Handling:** Shows how to attach event listeners to buttons for interactive functionality.
- **Multithreading in UI Applications:** Highlights the importance of running network operations in separate threads to keep the UI responsive.

Creating the Scoreboard (`createScoreTable`) :

```
private TableView<PlayerScore> createScoreTable() {
    TableView<PlayerScore> table = new TableView<>(scoreData);

    TableColumn<PlayerScore, String> playerCol = new TableColumn<>("Player");
    playerCol.setCellValueFactory(new PropertyValueFactory<>("player"));
    playerCol.setPrefWidth(150);

    TableColumn<PlayerScore, Integer> scoreCol = new TableColumn<>("Score");
    scoreCol.setCellValueFactory(new PropertyValueFactory<>("score"));
    scoreCol.setPrefWidth(60);

    table.getColumns().addAll(playerCol, scoreCol);
    table.setPrefHeight(300);

    return table;
}
```

Explanation:

- **Purpose:** Constructs the `TableView` to display player names and their corresponding scores.
- **Process:**
  - Defines two columns: "Player" and "Score".
  - Uses `PropertyValueFactory` to bind table columns to the `PlayerScore` class's properties.
  - Sets preferred widths for better layout.
  - Adds the columns to the table and sets its height.

Interesting Points:

- **Data Binding:** Explains how `TableView` can dynamically display data from model classes.
- **UI Components:** Introduces students to JavaFX's `TableView` and `TableColumn` for structured data presentation.

Handling Join Requests (`handleJoin`) :

```
private void handleJoin() {
    if (joined) {
        showMessage("Already joined the game!");
        return;
    }
    String name = nameField.getText().trim();
    if (name.isEmpty()) {
        showMessage("Please enter a valid name before joining.");
        return;
    }
    sendMessage("JOIN:" + name);
    joined = true;
    showMessage("JOIN request sent. Waiting for server response...");
}
```

Explanation:

- **Purpose:** Sends a join request to the server with the player's name.
- **Process:**
  - Checks if the player has already joined to prevent duplicate joins.
  - Validates that the name field isn't empty.
  - Sends a "JOIN:" message to the server with the player's name.
  - Updates the UI to reflect the join request status.

Interesting Points:

- **Input Validation:** Ensures that the player provides necessary information before proceeding.
- **State Management:** Tracks whether a player has already joined to prevent multiple join attempts.

Handling Word Submissions (`handleSubmitWord`) :

```
private void handleSubmitWord() {
    if (!joined) {
        showMessage("You must join first!");
        return;
    }
    String word = wordField.getText().trim();
    if (word.isEmpty()) {
        showMessage("Cannot submit an empty word!");
        return;
    }
    sendMessage("WORD:" + word);
    showMessage("Submitted word: " + word);

    // Clear the field and disable until next round letters arrive
```

```
    wordField.clear();
    submitWordButton.setDisable(true);
}
```

Explanation:

- **Purpose:** Allows the player to submit a word for the current round.
- **Process:**
  - Checks if the player has joined the game.
  - Validates that the word field isn't empty.
  - Sends a "WORD:" message to the server with the submitted word.
  - Updates the UI to reflect the submission and resets the input field.

Interesting Points:

- **User Interaction:** Demonstrates how user actions (button clicks) trigger network communications.
- **UI Feedback:** Shows the importance of providing immediate feedback to users upon their actions.

Listening for Server Messages (`listenForServer`)

```
private void listenForServer() {
    byte[] buf = new byte[2048];
    while (running.get()) {
        try {
            DatagramPacket packet = new DatagramPacket(buf, buf.length);
            socket.receive(packet);

            String serverMsg = new String(packet.getData(), 0, packet.getLength()).trim();
            Platform.runLater(() -> handleServerMessage(serverMsg));

        } catch (SocketException e) {
            // Socket closed
            running.set(false);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Explanation:

- **Purpose:** Continuously listens for incoming messages from the server in a background thread.
- **Process:**
  - Receives UDP packets from the server.
  - Converts the packet data to a string message.
  - Uses `Platform.runLater` to ensure that UI updates occur on the JavaFX Application Thread.
  - Handles socket closure gracefully by stopping the listener.

Interesting Points:

- **Thread Safety:** Explains the necessity of updating UI components on the JavaFX Application Thread to prevent concurrency issues.
- **Daemon Threads:** Demonstrates the use of daemon threads, which terminate automatically when the application exits.

Handling Server Messages (`handleServerMessage`)

```
private void handleServerMessage(String msg) {
    showMessage("Server: " + msg);

    // Because we sometimes get multi-line messages (like scores),
    // let's split by newline and handle line by line.
    String[] lines = msg.split("\n");

    for (String line : lines) {
        if (line.startsWith("JOIN_OK:")) {
            // Possibly update UI state
        }
        else if (line.startsWith("JOIN_REJECT:")) {
            joined = false;
            showMessage("Join rejected by server.");
            showPopup("Join Rejected", "The server has rejected your join request.", Alert.AlertType.ERROR);
        }
        else if (line.startsWith("ROUND_START:")) {
            // Round start info (e.g., "ROUND_START:1")
            String roundNum = line.substring("ROUND_START:".length()).trim();
            showPopup("Round Started", "Round " + roundNum + " has begun!", Alert.AlertType.INFORMATION);
        }
        else if (line.startsWith("ROUND_LETTERS:")) {
            // Let user input a word now
            submitWordButton.setDisable(false);
            String letters = line.substring("ROUND_LETTERS:".length()).trim();
            showPopup("Round Letters", "Letters for this round: " + letters, Alert.AlertType.INFORMATION);
        }
        else if (line.startsWith("RECEIVED:")) {
            // Acknowledged
        }
        else if (line.startsWith("ROUND_OVER:")) {
            // Round over
            String roundNum = line.substring("ROUND_OVER:".length()).trim();
            showPopup("Round Over", "Round " + roundNum + " is over. Scores have been updated.", Alert.AlertType.INFORMATION);
        }
        else if (line.startsWith("Current Scores:")) {
            // Next lines (if any) contain scoreboard info
            // We'll handle it after we exit this for loop, because we want to parse
            // them in the overall message. Or we can just handle inline:
            // Do nothing here, we parse in the iteration below.
        }
        else if (line.startsWith("WINNER:")) {
            // "WINNER:playerKey:score"
            showPopup("Game Over", line, Alert.AlertType.INFORMATION);
            closeClient();
        }
        else if (line.startsWith("TIE:")) {
            // "TIE:[player1, player2]:score"
            showPopup("Game Over (Tie)", line, Alert.AlertType.INFORMATION);
            closeClient();
        }
        else {
            // Could be part of the "Current Scores" lines
            if (line.contains("->") && line.contains("points")) {
                // This is likely a line with "PlayerName -> X points"
                parseScoreLine(line);
            }
            else {
                // A generic broadcast or other message
            }
        }
```

```
    }
}
```

Explanation:

- **Purpose:** Parses and responds to various types of messages from the server.
- **Process:**
  - Splits multi-line messages and handles each line individually.
  - Recognizes specific message prefixes (e.g., "JOIN_OK:", "ROUND_START:") to determine the action.
  - Updates the UI by showing pop-ups, enabling buttons, and updating the scoreboard based on the message content.

Interesting Points:

- **Message Parsing:** Teaches how to interpret structured messages and trigger corresponding UI updates.
- **User Feedback:** Emphasizes providing clear and immediate feedback to users based on game events.
- **Modularity:** Demonstrates handling different message types in a clean and organized manner.

Parsing Score Lines (`parseScoreLine`) :

```
private void parseScoreLine(String line) {
    if (line.trim().isEmpty()) return;

    // Example line: "PlayerName -> 10 points"
    try {
        String[] parts = line.split("->");
        if (parts.length < 2) return;
        String playerPart = parts[0].trim();  // e.g., "PlayerName"
        String scorePart = parts[1].trim();   // e.g., "10 points"

        // Remove " points"
        scorePart = scorePart.replace(" points", "").trim();

        int score = Integer.parseInt(scorePart);

        // Update the scoreboard entry for this player
        updateScoreboard(playerPart, score);

    } catch (Exception e) {
        // ignore parse errors
    }
}
```

Explanation:

- **Purpose:** Extracts player names and scores from score lines to update the scoreboard.
- **Process:**
  - Splits the line into player name and score.

- Cleans the score string by removing the " points" suffix.
- Parses the score as an integer.
- Calls `updateScoreboard` to reflect the new score in the UI.

Interesting Points:

- **String Manipulation:** Shows practical techniques for parsing and processing strings.
- **Error Handling:** Handles potential parsing errors gracefully without disrupting the user experience.

Updating the Scoreboard (`updateScoreboard`)

```
private void updateScoreboard(String player, int score) {
    // If player already in scoreboard, update. Else add new row.
    for (PlayerScore ps : scoreData) {
        if (ps.getPlayer().equals(player)) {
            ps.setScore(score);
            scoreTable.refresh();
            return;
        }
    }
    // Not found, add new
    scoreData.add(new PlayerScore(player, score));
    scoreTable.refresh();
}
```

Explanation:

- **Purpose:** Reflects the latest scores in the `TableView`.
- **Process:**
  - Checks if the player already exists in the scoreboard.
  - Updates the score if the player is found.
  - Adds a new entry if the player isn't already listed.
  - Refreshes the table to display the latest data.

Interesting Points:

- **Data Binding:** Demonstrates how changes in data models (`scoreData`) automatically reflect in the UI (`scoreTable`).
- **Efficiency:** Ensures that only necessary updates are made, preventing unnecessary data duplication.

Displaying Pop-ups and Messages (`showPopup`, `showMessage`):

```
private void showPopup(String title, String message, Alert.AlertType type) {
    Alert alert = new Alert(type);
    alert.setTitle(title);
    alert.setHeaderText(null); // no header
    alert.setContentText(message);
    alert.showAndWait();
```

```
}

private void showMessage(String text) {
    serverMessagesArea.appendText(text + "\n");
}
```

Explanation:

- `showPopup`:
  - **Purpose:** Displays a modal dialog to inform the player of important events (e.g., round start, game over).
  - **Process:** Creates and configures an `Alert` dialog with the specified title, message, and type, then displays it.
- `showMessage`:
  - **Purpose:** Appends messages to the `TextArea` for continuous updates (e.g., server messages).
  - **Process:** Adds the message text to the `serverMessagesArea` with a newline for readability.

Interesting Points:

- **User Interface Design:** Highlights the importance of different types of feedback (modal dialogs vs. continuous logs) for user experience.
- **JavaFX Components:** Explores the use of `Alert` and `TextArea` for user notifications.

Closing the Client (`closeClient`)

```
private void closeClient() {
    running.set(false);
    if (socket != null && !socket.isClosed()) {
        socket.close();
    }
}
```

Explanation:

- **Purpose:** Gracefully shuts down the client by stopping the listener thread and closing the socket.
- **Process:**
  - Sets the `running` flag to `false` to stop the listener loop.
  - Closes the `DatagramSocket` if it's open.

Interesting Points:

- **Resource Management:** Emphasizes the importance of releasing network resources to prevent resource leaks.
- **Thread Termination:** Shows how to signal threads to stop safely.

JavaFX Application Stop (`stop`)

```
@Override
public void stop() throws Exception {
    // Called when the application is closed (e.g., window is closed).
    super.stop();
    closeClient();
}
```

Explanation:

- **Purpose:** Ensures that the client shuts down gracefully when the application window is closed.
- **Process:** Overrides the `stop()` method to call `closeClient()`, ensuring all resources are properly released.

Interesting Points:

- **Lifecycle Management:** Illustrates how to manage application lifecycle events in JavaFX.
- **Best Practices:** Reinforces the practice of cleaning up resources upon application termination.

`PlayerScore` Model Class :

```
public static class PlayerScore {
    private final String player;
    private int score;

    public PlayerScore(String player, int score) {
        this.player = player;
        this.score = score;
    }

    public String getPlayer() {
        return player;
    }

    public int getScore() {
        return score;
    }

    public void setScore(int newScore) {
        this.score = newScore;
    }
}
```

Explanation:

- **Purpose:** Represents a player's score in the scoreboard.
- **Structure:**
  - **Properties:** `player` (name) and `score`.
  - **Getters and Setters:** Facilitate data binding with the `TableView`.

Interesting Points:

- **Model Classes:** Demonstrates how to create simple data models for UI components.

- **Encapsulation:** Shows the use of private fields and public methods to manage data access.