Practical Assessment Task

Phase 3 – Code Document

Igor Karbowy

Table of Contents

SubnetScanResult.java

```java
package com.pingpal.subnetscan;

/**
 * The {@code SubnetScanResult} class represents the result of scanning a
 * subnet, encapsulating a reachable IP address.
 * <p>
 * This class is a simple data container used to store an IP address that has
 * been found to be reachable during a subnet scan.
 * </p>
 *
 * @author Igor Karbowy
 */
public class SubnetScanResult {

    // The IP address that was found to be reachable.
    private String ipAddress;

    /**
     * Constructs a new {@code SubnetScanResult} instance with the specified IP
     * address.
     *
     * @param ipAddress the reachable IP address in standard dot-decimal
     * notation (e.g., "192.168.0.1")
     */
    public SubnetScanResult(String ipAddress) {
        this.ipAddress = ipAddress;
    }

    /**
     * Returns the reachable IP address stored in this result.
     *
     * @return a {@code String} representing the IP address in dot-decimal
     * notation
     */
    public String getIPAddress() {
        return ipAddress;
    }
}
```

SubnetScan.java

```java
package com.pingpal.subnetscan;

import static java.awt.EventQueue.invokeLater;
import java.io.IOException;
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.util.ArrayList;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;
import javax.swing.JProgressBar;
import javax.swing.JTable;
import javax.swing.table.DefaultTableModel;

/**
 * The {@code SubnetScan} class performs a scan on a given subnet range and
 * identifies reachable IP addresses by attempting to ping them.
 * <p>
 * The class uses a thread pool to concurrently scan multiple IPs. It also
 * updates a {@code JTable} with the scan results and a {@code JProgressBar} to
 * indicate progress.
 * </p>
 * <p>
 * The network range is expected to be in the form "x.x.x.x/n", where n is the
 * number of network bits. The total number of IP addresses to scan is derived
 * from this value.
 * </p>
 *
 * @author Igor Karbowy
 */
public class SubnetScan {

    // List of subnet scan results. Each result represents a reachable IP.
    private ArrayList<SubnetScanResult> subnetScanResults = new ArrayList<>();

    // The network range to scan (e.g., "192.168.0.0/24").
    private String networkRange;

    // Timeout in milliseconds for checking if an IP is reachable.
    private int timeout;

    // Total number of IPs to scan (calculated from the network range).
    private int numOfIPs;

    // Table model that will be updated with the reachable IP addresses.
    private DefaultTableModel model;

    // Progress bar to display the scanning progress.
    private JProgressBar prgSubnetScan;

    // Number of threads used for scanning.
    // It is set to the number of available processors of the machine multiplied by 32.
    private final int THREAD_COUNT = Runtime.getRuntime().availableProcessors() * 32;

    // Executor service for running scan tasks concurrently.
    private ExecutorService executorService = Executors.newFixedThreadPool(THREAD_COUNT);

    // Flag to indicate if a stop has been requested for the scan.
    private boolean stopRequested = false;

    /**
     * Constructs a new {@code SubnetScan} instance with the specified network
     * range, timeout, table for results, and progress bar.
     *
     * @param networkRange the network range to scan (format "x.x.x.x/n")
     * @param timeout the timeout (in milliseconds) to use for reachability
     * checks
     * @param tbl the {@code JTable} whose model will be updated with scan
     * results
     * @param prgSubnetScan the {@code JProgressBar} used to show scan progress
     */
    public SubnetScan(String networkRange, int timeout, JTable tbl, JProgressBar
prgSubnetScan) {
        this.networkRange = networkRange;
        this.timeout = timeout;
        // Retrieve the model from the passed JTable.
        this.model = (DefaultTableModel) tbl.getModel();
        this.prgSubnetScan = prgSubnetScan;

        // Calculate the number of IP addresses based on the network range.
        setNumOfIPs();
    }

    /**
     * Starts the subnet scan.
     * <p>
     * Clears the table and progress bar, then concurrently scans all IPs in the
     * subnet. Uses an ExecutorService to run scan tasks concurrently and
     * updates the progress bar.
     * </p>
     * <p>
     * The method executes until either all tasks complete, the timeout is
     * reached, or a stop is requested.
     * </p>
     */
    public void start() {
        // Clear current data in the table on the EDT.
        invokeLater(() -> model.setRowCount(0));
        // Reset the progress bar on the EDT.
        invokeLater(() -> prgSubnetScan.setValue(0));

        // Counter variable for IPs to scan.
        AtomicInteger ips = new AtomicInteger(0);
        // Counter variable for already scanned IPs.
        AtomicInteger scannedIps = new AtomicInteger(0);

        // Loop through each IP index in the range.
        while (ips.get() <= numOfIPs) {
            // Generate the IP to scan.
            String ip = generateIP(ips.getAndIncrement());

            // Submit a task to the executor to scan the IP.
            executorService.submit(() -> {
                InetAddress inAddress;
                try {
                    // Resolve the IP address.
                    inAddress = InetAddress.getByName(ip);
                    // Scan the IP.
                    scanIP(inAddress);
                    // Update the progress bar after scanning each IP.
                    updateProgressBar(scannedIps.getAndIncrement());
                } catch (UnknownHostException e) {
                    // If IP is unknown, ignore and continue.
                }
            });
        }

        // Initiate shutdown and wait for tasks to finish, with a maximum wait time.
        executorService.shutdown();
        try {
            // Wait until all tasks have finished, or timeout after 10 minutes.
            if (!executorService.awaitTermination(10, TimeUnit.MINUTES)) {
                // If tasks are not finished in 10 minutes, force shutdown.
                executorService.shutdownNow();
            }
        } catch (InterruptedException e) {
            // If the thread is interrupted, reset the interrupt flag and exit the loop.
            executorService.shutdownNow();
            Thread.currentThread().interrupt();
        }
    }

    /**
     * Calculates and sets the number of IP addresses to scan ({@code numOfIps}
     * variable) based on the network bits provided in the network range.
     * <p>
     * For example, for a /24 network, this computes 2^(32-24) - 1.
     * </p>
     */
    private void setNumOfIPs() {
        // Extract the network bits from the network range string (e.g., "24" from
"192.168.0.0/24")
        int networkBits = Integer.parseInt(networkRange.substring(networkRange.indexOf("/") +
1));

        // Calculate the number of IPs: 2^(32 - networkBits) - 1 (subtracting network
address)
        numOfIPs = (int) Math.pow(2, (32 - networkBits)) - 1;
    }

    /**
     * Generates an IP address from the subnet starting IP based on the given
     * index.
     *
     * @param ipNum the sequential number of the IP address to generate
     * @return the generated IP address as a {@code String} (e.g.,
     * "192.168.0.15")
     */
    private String generateIP(int ipNum) {
        // Split the network address portion (before the '/') into its segments.
        String[] segments = networkRange.substring(0,
networkRange.indexOf("/")).split("\\.");

        // Convert the IP segments to a single long value.
        long ipValue = 0;
        for (String segment : segments) {
            ipValue = (ipValue << 8) | Integer.parseInt(segment);
        }

        // Add the sequential IP number to the base IP
        long newIpValue = ipValue + ipNum;
        long mod = 1L << 32; // Total number of IPv4 addresses
        newIpValue = newIpValue % mod;
        if (newIpValue < 0) {
            newIpValue += mod;
        }

        // Extract each segment of the new IP address.
        int newSeg1 = (int) ((newIpValue >> 24) & 0xFF);
        int newSeg2 = (int) ((newIpValue >> 16) & 0xFF);
        int newSeg3 = (int) ((newIpValue >> 8) & 0xFF);
        int newSeg4 = (int) (newIpValue & 0xFF);

        // Combine the segments and return the generated IP address as a String.
        return newSeg1 + "." + newSeg2 + "." + newSeg3 + "." + newSeg4;
    }

    /**
     * Attempts to ping the single given InetAddress. If reachable, adds the IP
     * address to the results list and updates the table model.
     *
     * @param inAddress the InetAddress representing the IP to scan
     */
    private void scanIP(InetAddress inAddress) {
        try {
            // If the IP is reachable within the given timeout, consider it active.
            if (inAddress.isReachable(timeout)) {
                // Add result to the subnet scan results list.
                subnetScanResults.add(new SubnetScanResult(inAddress.getHostAddress()));
                // Update the table model on the EDT using invokeLater.
                invokeLater(() -> model.addRow(new Object[]{inAddress.getHostAddress()}));
            }
        } catch (IOException e) {
            // If an exception occurs (e.g., timeout), consider it inactive.
        }
    }

    /**
     * Updates the progress bar based on the number of IPs scanned.
     *
     * @param ipNum the number of IPs scanned so far
     */
    private void updateProgressBar(int ipNum) {
        // Calculate the percentage of scanned IPs, and update the progress bar on the EDT.
        invokeLater(() -> prgSubnetScan.setValue((int) Math.round(((double) ipNum / numOfIPs)
* 100)));
    }

    /**
     * Retrieves the list of subnet scan results.
     *
     * @return an {@code ArrayList} of {@code SubnetScanResult} objects
     */
    public ArrayList<SubnetScanResult> getSubnetScanResults() {
        return subnetScanResults;
    }

    /**
     * Sets the subnet scan results.
     * <p>
     * Used when importing results, i.e. when no subnet scan was performed to
     * have added the found devices to the subnet scan results list.
     * </p>
     *
     * @param subnetScanResults an {@code ArrayList} of {@code SubnetScanResult}
     * objects
     */
    public void setSubnetScanResults(ArrayList<SubnetScanResult> subnetScanResults) {
        this.subnetScanResults = subnetScanResults;
    }

    /**
     * Returns the network range that was scanned.
     *
     * @return the network range in string format (e.g., "192.168.0.0/24") as a
     * {@code String}
     */
    public String getNetworkRange() {
        return networkRange;
    }

    /**
     * Returns the timeout used for each IP scan.
     *
     * @return the timeout in milliseconds as an {@code int}
     */
    public int getTimeout() {
        return timeout;
    }

    /**
     * Forcefully shuts down the executor service, stopping any running tasks.
     */
    public void shutDownExecutorService() {
        executorService.shutdownNow();
    }

    /**
     * Requests the subnet scan to stop.
     * <p>
     * The {@code stopRequested} flag is set to true, so that ongoing tasks in
     * {@code start} may check this flag and terminate early.
     * </p>
     */
    public void requestStop() {
        stopRequested = true;
    }

    /**
     * Checks whether a stop has been requested for the scan.
     *
     * @return {@code true} if a stop has been requested; {@code false}
     * otherwise.320
     */
    public boolean isStopRequested() {
        return stopRequested;
    }
}
```

DevicePingResult.java

```java
package com.pingpal.deviceping;

/**
 * The {@code DevicePingResult} class represents the result of a ping operation
 * performed on a device.
 * <p>
 * This class encapsulates three pieces of information obtained from a device
 * ping:
 * </p>
 * <ul>
 * <li>The round-trip time (RTT) measured in milliseconds.</li>
 * <li>A flag indicating whether the ping was successful.</li>
 * <li>The packet loss percentage observed during the ping operation.</li>
 * </ul>
 */
public class DevicePingResult {

    // The round-trip time in milliseconds.
    private int roundTripTime;

    // Flag which indicates whether the ping was successful.
    private boolean successfulPing;

    // The percentage of packet loss measured during the ping.
    private double packetLoss;

    /**
     * Constructs a new {@code DevicePingResult} instance with the specified
     * round-trip time, successful ping flag, and packet loss percentage.
     *
     * @param roundTripTime the round-trip time in milliseconds
     * @param successfulPing {@code true} if the ping was successful;
     * {@code false} otherwise
     * @param packetLoss the percentage of packet loss (as a double, e.g. 0.0
     * for no loss, 100.0 for complete loss)
     */
    public DevicePingResult(int roundTripTime, boolean successfulPing, double packetLoss) {
        this.roundTripTime = roundTripTime;
        this.successfulPing = successfulPing;
        this.packetLoss = packetLoss;
    }

    /**
     * Returns the round-trip time (RTT) for the ping operation.
     *
     * @return the round-trip time in milliseconds as an {@code int}
     */
    public int getRoundTripTime() {
        return roundTripTime;
    }

    /**
     * Returns whether the ping was successful.
     *
     * @return {@code true} if the ping was successful; {@code false} otherwise
     */
    public boolean isSuccessfulPing() {
        return successfulPing;
    }

    /**
     * Returns the packet loss percentage observed during the ping.
     *
     * @return the packet loss percentage as a {@code double}
     */
    public double getPacketLoss() {
        return packetLoss;
    }
}
```

DevicePing.java

```java
package com.pingpal.deviceping;

import static java.awt.EventQueue.invokeLater;
import java.io.IOException;
import java.net.InetAddress;
import java.util.ArrayList;
import javax.swing.JTable;
import javax.swing.table.DefaultTableModel;

/**
 * The {@code DevicePing} class performs a series of ping operations against a
 * given IP address. It collects metrics such as round-trip times, whether each
 * ping was successful, and calculates the packet loss percentage.
 * <p>
 * The results are displayed in three different {@code JTable} components:
 * <ul>
 * <li>One for individual ping results (IP, round-trip time, success, packet
 * loss)</li>
 * <li>One summarizing response results (min, max, average round-trip
 * times)</li>
 * <li>One summarizing packet loss results (total pings, successful pings,
 * failed pings, packet loss percentage)</li>
 * </ul>
 * <p>
 * The class supports both a fixed number of pings or continuous pinging based
 * on the provided parameters.
 * </p>
 */
public class DevicePing {

    // List of results from each ping operation.
    private ArrayList<DevicePingResult> devicePingResults;

    // The IP address to ping.
    private String ipAddress;

    // The interval (in milliseconds) between successive pings.
    private int pingInterval;

    // The total number of pings to attempt if not in continuous mode.
    private int numOfPings;

    // If true, the ping operations continue indefinitely.
    private boolean continuousPinging;

    // Counter for the total number of pings attempted.
    private int pingCount;

    // Counter for the number of successful pings.
    private int successfulPings;

    // Table model for displaying individual ping results.
    private DefaultTableModel devicePingTableModel;

    // Table model for displaying ping response summaries (min, max, avg).
    private DefaultTableModel devicePingResponseResultsTableModel;

    // Table model for displaying packet loss summary results.
    private DefaultTableModel devicePingPacketResultsTableModel;

    // Flag indicating if a stop has been requested.
    private boolean stopRequested = false;

    /**
     * Constructs a new {@code DevicePing} instance with the specified IP
     * address, ping interval, number of pings, continuous pinging flag, and
     * tables for results.
     *
     * @param ipAddress the target IP address to ping
     * @param pingInterval the interval in milliseconds between pings
     * @param numOfPings the number of pings to perform (if not continuous)
     * @param continuousPinging {@code true} for continuous pinging;
     * {@code false} for a fixed number of pings
     * @param tblDevicePing the JTable for displaying individual ping results
     * @param tblDevicePingResponseResults the JTable for displaying response
     * summary (min, max, average round-trip times)
     * @param tblDevicePingPacketResults the JTable for displaying packet loss
     * summary (total, successful, failed, packet loss)
     */
    public DevicePing(String ipAddress, int pingInterval, int numOfPings, boolean
continuousPinging, JTable tblDevicePing, JTable tblDevicePingResponseResults, JTable
tblDevicePingPacketResults) {
        devicePingResults = new ArrayList<>();
        this.ipAddress = ipAddress;
        this.pingInterval = pingInterval;
        this.numOfPings = numOfPings;
        this.continuousPinging = continuousPinging;

        // Initialize counters
        this.pingCount = 0;
        this.successfulPings = 0;

        // Retrieve and store table models from the passed JTables.
        this.devicePingTableModel = (DefaultTableModel) tblDevicePing.getModel();
        this.devicePingResponseResultsTableModel = (DefaultTableModel)
tblDevicePingResponseResults.getModel();
        this.devicePingPacketResultsTableModel = (DefaultTableModel)
tblDevicePingPacketResults.getModel();
    }

    /**
     * Starts the pinging process. This method clears the UI tables, then
     * repeatedly pings the target IP address according to the ping interval,
     * until the specified number of pings has been reached (or indefinitely if
     * continuous). After pinging is complete, it populates summary result
     * tables.
     * <p>
     * Note: This method runs on the calling thread. It is expected that you run
     * it in a background thread to avoid blocking the UI.
     * </p>
     */
    public void start() {
        // Clear current data in the result tables on the EDT
        invokeLater(() -> {
            devicePingTableModel.setRowCount(0);
            devicePingResponseResultsTableModel.setRowCount(0);
            devicePingPacketResultsTableModel.setRowCount(0);
        });

        // Calculate the time (in nanoseconds) when the next ping should be performed.
        double nextPingTime = System.nanoTime() + (pingInterval * 1_000_000);

        // Loop until the program reaches the specified number of pings or continuous mode,
and no stop has been requested.
        while ((pingCount < numOfPings || continuousPinging) && !stopRequested) {
            // Call the ping IP method
            pingIP();

            try {
                // Calculate remaining time (in milliseconds) before the next ping.
                double remainingTime = (nextPingTime - System.nanoTime()) / 1_000_000;

                // If pinging the IP took longer than the ping interval, reset the remaining
time to 0 to ensure no negative time.
                if (remainingTime < 0) {
                    remainingTime = 0;
                }

                // Sleep until it's time for the next ping.
                Thread.sleep((long) remainingTime);

                // Update the time when the next ping should be performed.
                nextPingTime += (pingInterval * 1_000_000);

            } catch (InterruptedException e) {
                // If the thread is interrupted, reset the interrupt flag and exit the loop.
                Thread.currentThread().interrupt();
            }
        }

        // After pinging, update the UI with the summary results.
        populateResultsTables();
    }

    /**
     * Performs a single ping to the target IP address.
     * <p>
     * This method attempts to ping the specified IP address using
     * {@code InetAddress.isReachable()}. If the IP is reachable, it records the
     * round-trip time and considers the ping successful. If not, it records a
     * failed ping with a round-trip time equal to the ping interval. The result
     * is stored in the {@code devicePingResults} list, and the UI table is
     * updated.
     * </p>
     */
    private void pingIP() {
        try {
            // Resolve the IP address.
            InetAddress inAddress = InetAddress.getByName(ipAddress);

            // Record the time before pinging.
            long timeBeforePing = System.nanoTime();

            // Check if the IP is reachable within the ping interval.
            if (inAddress.isReachable(pingInterval)) {
                // Calculate round-trip time in milliseconds.
                int roundTripTime = (int) (System.nanoTime() - timeBeforePing) / 1_000_000;

                // Clamp roundTripTime to the pingInterval if necessary.
                roundTripTime = Math.min(roundTripTime, pingInterval);

                // Update counter variables
                pingCount++;
                successfulPings++;

                // Add a new successful ping result.
                devicePingResults.add(new DevicePingResult(roundTripTime, true,
getPacketLoss()));

                // Update the UI table with the successful ping result.
                invokeLater(() -> {
                    devicePingTableModel.addRow(new Object[]{
                        ipAddress,
                        devicePingResults.getLast().getRoundTripTime(),
                        true,
                        devicePingResults.getLast().getPacketLoss()
                    });
                });

                // Ping failed.
            } else {
                // Update counter variables.
                pingCount++;

                // Add a new unsuccessful ping result.
                devicePingResults.add(new DevicePingResult(pingInterval, false,
getPacketLoss()));

                // Update the UI table with the failed ping result.
                invokeLater(() -> {
                    devicePingTableModel.addRow(new Object[]{
                        ipAddress,
                        pingInterval,
                        false,
                        devicePingResults.getLast().getPacketLoss()
                    });
                });

            }
        } catch (IOException e) {
            // In case of an I/O error, the exception is ignored as it does not occurr given
the IP address is guaranteed to be in correct format.
        }
    }

    /**
     * Populates the summary result tables with the calculated ping statistics.
     * <p>
     * This method updates two tables:
     * </p>
     * <ul>
     * <li>The response results table with minimum, maximum, and average
     * round-trip times.</li>
     * <li>The packet results table with the total number of pings, successful
     * pings, failed pings, and the packet loss percentage.</li>
     * </ul>
     * <p>
     * The updates are performed on the EDT using {@code invokeLater()}.
     * </p>
     */
    public void populateResultsTables() {
        invokeLater(() -> {
            // Populate the response results table.
            devicePingResponseResultsTableModel.addRow(new Object[]{
                getMinimumRoundTripTime(),
                getMaximumRoundTripTime(),
                getAverageRoundTripTime()

            });

            // Populate the packet results table.
            devicePingPacketResultsTableModel.addRow(new Object[]{
                pingCount,
                successfulPings,
                pingCount - successfulPings,
                getPacketLoss()
            });
        });
    }

    /**
     * Calculates the packet loss percentage.
     *
     * @return the packet loss percentage as a double, rounded to two decimal
     * places
     */
    private double getPacketLoss() {
        double packetLoss = (1.0 - ((double) successfulPings / (double) pingCount)) *
10_000.0;
        packetLoss = Math.round(packetLoss);
        packetLoss /= 100.0;
        return packetLoss;
    }

    /**
     * Computes the minimum round-trip time among all successful ping results.
     *
     * @return the minimum round-trip time in milliseconds
     */
    private int getMinimumRoundTripTime() {
        // Assume the first result is the minimum initially.
        int minimum = devicePingResults.getFirst().getRoundTripTime();

        for (DevicePingResult result : devicePingResults) {
            // For each result, check if it is smaller than the current minimum.
            if (result.getRoundTripTime() < minimum) {
                // Update the minimum if it is the case.
                minimum = result.getRoundTripTime();
            }
        }

        return minimum;
    }

    /**
     * Computes the maximum round-trip time among all ping results.
     *
     * @return the maximum round-trip time in milliseconds
     */
    private int getMaximumRoundTripTime() {
        // Assume the first result is the maximum initially.
        int maximum = devicePingResults.getFirst().getRoundTripTime();

        for (DevicePingResult result : devicePingResults) {
            // For each result, check if it is larger than the current maximum
            if (result.getRoundTripTime() > maximum) {
                // Update the maximum if it is the case
                maximum = result.getRoundTripTime();
            }
        }

        return maximum;
    }

    /**
     * Computes the average round-trip time among all successful ping results.
     *
     * @return the average round-trip time in milliseconds, rounded to two
     * decimal places
     */
    private double getAverageRoundTripTime() {
        // Create a temporary holder variable to store the total round trip time of all
successful pings.
        double totalRoundTripTime = 0;

        for (DevicePingResult result : devicePingResults) {
            // For each result, check if it is a successful result.
            if (result.isSuccessfulPing()) {
                // Add the round-trip time if it is the case
                totalRoundTripTime += result.getRoundTripTime();
            }
        }

        // Calculate the average given the total round trip time of all successful pings.
        double avgRoundTripTime = totalRoundTripTime / successfulPings * 100.0;
        avgRoundTripTime = Math.round(avgRoundTripTime);
        avgRoundTripTime /= 100.0;

        return avgRoundTripTime;
    }

    /**
     * Requests that the pinging process stop.
     * <p>
     * This sets the {@code stopRequested} flag to true so that the ping loop in
     * {@code start()} will terminate early.
     * </p>
     */
    public void requestStop() {
        stopRequested = true;
    }

    /**
     * Checks whether a stop request has been made.
     *
     * @return {@code true} if a stop has been requested; {@code false}
     * otherwise
     */
    public boolean isStopRequested() {
        return stopRequested;
    }

    /**
     * Returns the list of individual ping results.
     *
     * @return an {@code ArrayList} of {@code DevicePingResult} objects
     */
    public ArrayList<DevicePingResult> getDevicePingResults() {
        return devicePingResults;
    }

    /**
     * Sets the list of ping results.
     * <p>
     * Used when importing results, i.e. when no device ping was performed to
     * have added the successful pings to the device ping results list.
     * </p>
     *
     * @param devicePingResults an {@code ArrayList} of {@code DevicePingResult}
     * objects
     */
    public void setDevicePingResults(ArrayList<DevicePingResult> devicePingResults) {
        this.devicePingResults = devicePingResults;
    }

    /**
     * Returns the target IP address being pinged.
     *
     * @return the IP address as a {@code String}
     */
    public String getIpAddress() {
        return ipAddress;
    }

    /**
     * Returns the ping interval in milliseconds.
     *
     * @return the ping interval as an {@code int}
     */
    public int getPingInterval() {
        return pingInterval;
    }

    /**
     * Returns the total number of pings configured.
     *
     * @return the number of pings as an {@code int}
     */
    public int getNumOfPings() {
        return numOfPings;
    }

    /**
     * Indicates whether the ping process is running continuously.
     *
     * @return {@code true} if continuous pinging is enables; {@code false}
     * otherwise
     */
    public boolean isContinuousPinging() {
        return continuousPinging;
    }

    /**
     * Returns the number of successful pings.
     *
     * @return the number of successful pings as an {@code int}
     */
    public int getSuccessfulPings() {
        return successfulPings;
    }
    /**
     * Recalculates the number of successful pings based on the current results.
     * <p>
     * This iterates over all results and increments the successful ping counter
     * for every result that indicates a successful ping.
     * </p>
     * <p>
     * Used when importing results, i.e. when no device ping was performed to
     * count the number of successful pings.
     * </p>
     */
    public void setSuccessfulPings() {
        for (DevicePingResult devicePingResult : devicePingResults) {
            successfulPings = devicePingResult.isSuccessfulPing() ? successfulPings + 1 :
successfulPings;
        }
    }

    /**
     * Updates the ping counter to reflect the total number of pings performed.
     * <p>
     * This simply sets the pingCount to the size of the
     * {@code devicePingResults} list.
     * </p>
     * <p>
     * Used when importing results, i.e. when no device ping was performed to
     * count the number of pings.
     * </p>
     */
    public void setPingCount() {
        pingCount = devicePingResults.size();
    }
}
```

PortScanResult.java

```java
package com.pingpal.portscan;

/**
 * Represents a single result from a port scan.
 * <p>
 * This class encapsulates the details of a port scan result, including the port
 * number that was found to be open, and the associated protocol determined for
 * that port.
 * </p>
 */
public class PortScanResult {

    // The port number that was found to be open.
    private int portNumber;

    // The protocol associated with the port number. For example, "http", "https".
    private String protocol;

    /**
     * Constructs a new {@code PortScanResult} with the specified port number
     * and protocol.
     *
     * @param portNumber the port number that was found to be open
     * @param protocol the protocol associated with the port (e.g., "http",
     * "ftp")
     */
    public PortScanResult(int portNumber, String protocol) {
        this.portNumber = portNumber;
        this.protocol = protocol;
    }

    /**
     * Returns the port number associated with this scan result.
     *
     * @return the open port number as an {@code int}
     */
    public int getPortNumber() {
        return portNumber;
    }

    /**
     * Returns the protocol associated with this port scan result.
     *
     * @return the protocol name as a {@code String}
     */
    public String getProtocol() {
        return protocol;
    }
}
```

PortScan.java

```java
package com.pingpal.portscan;

import static java.awt.EventQueue.invokeLater;
import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.Socket;
import java.util.ArrayList;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;
import javax.swing.JProgressBar;
import javax.swing.JTable;
import javax.swing.table.DefaultTableModel;

/**
 * The {@code PortScan} class performs a TCP port scan on a specified IP
 * address.
 * <p>
 * This class scans a range of ports (from {@code bottomRangePort} to
 * {@code topRangePort}) on the target IP address. For each port, it attempts to
 * connect using a specified timeout. If the connection is successful, the port
 * is assumed to be open, and the protocol for the port is determined using the
 * {@code Protocols} class. The results are stored in an {@code ArrayList} and
 * displayed in a {@code JTable} while the progress is updated via a
 * {@code JProgressBar}.
 * </p>
 */
public class PortScan {

    // A list of port scan results.
    private ArrayList<PortScanResult> portScanResults = new ArrayList<>();

    // The target IP address to scan.
    private String ipAddress;

    // The starting port number of the scan range.
    private int bottomRangePort;

    // The ending port number of the scan range.
    private int topRangePort;

    // The timeout (in milliseconds) for attempting to connect to each port.
    private int timeout;

    // The table model used to update the UI with the scan results.
    private DefaultTableModel model;

    // The progress bar used to display scan progress.
    private JProgressBar prgPortScan;

    // Number of threads used for scanning.
    // It is set to the number of available processors of the machine multiplied by 32.
    private final int THREAD_COUNT = Runtime.getRuntime().availableProcessors() * 32;

    // Executor service for running scan tasks concurrently.
    private ExecutorService executorService = Executors.newFixedThreadPool(THREAD_COUNT);
    // A flag indicating whether a stop has been requested.
    private boolean stopRequested = false;

    /**
     * Constructs a new {@code PortScan} instance with the specified IP address,
     * bottom range port, top range port, timeout, table for results, and
     * progress bar.
     *
     * @param ipAddress the target IP address to scan
     * @param bottomRangePort the starting port number of the scan range
     * @param topRangePort the ending port number of the scan range
     * @param timeout the connection timeout (in milliseconds) for each port
     * @param tbl the {@code JTable} whose model will be updated with scan
     * results
     * @param prgPortScan the {@code JProgressBar} that displays the scanning
     * progress
     */
     public PortScan(String ipAddress, int bottomRangePort, int topRangePort, int timeout,
JTable tbl, JProgressBar prgPortScan) {
        this.ipAddress = ipAddress;
        this.bottomRangePort = bottomRangePort;
        this.topRangePort = topRangePort;
        this.timeout = timeout;
        this.model = (DefaultTableModel) tbl.getModel();
        this.prgPortScan = prgPortScan;
    }

    /**
     * Attempts to scan a single port on the target IP address.
     * <p>
     * The method creates a new socket, attempts to connect to the given port
     * with the specified timeout, and then closes the socket. If the connection
     * is successful, it retrieves the associated protocol and updates the scan
     * results list and the UI table.
     * </p>
     *
     * @param port the port number to scan
     */
    private void scanPort(int port) {
        try {
            // Create new socket object.
            Socket socket = new Socket();
            // Attempt to connect within a given timeout.
            socket.connect(new InetSocketAddress(ipAddress, port), timeout);
            socket.close();

            // Retrieve protocol for the given port.
            String protocol = Protocols.getProtocolForPort(port);

            // Record the scan result in the port scan results list.
            portScanResults.add(new PortScanResult(port, protocol));

            // Update UI results table on the EDT
            invokeLater(() -> model.addRow(new Object[]{port, protocol}));

        } catch (IOException e) {
             // Exception is ignored if the port is not reachable, and the port is considered
closed.
        }
    }

    /**
```

```
     * Starts the port scan.
     * <p>
     * This method clears any existing data from the UI components, then
     * iterates over the port range, submitting tasks to scan each port
     * concurrently. The progress bar is updated as ports are scanned.
     * </p>
     * <p>
     * The method executes until either all tasks complete, the timeout is
     * reached, or a stop is requested.
     * </p>
     */
    public void start() {
        // Clear current data being displayed in the UI table via EDT.
        invokeLater(() -> model.setRowCount(0));

        // Reset progress bar via EDT.
        invokeLater(() -> prgPortScan.setValue(0));

        // Atomic counter to generate port numbers from bottomRangePort to topRangePort.
        AtomicInteger ports = new AtomicInteger(bottomRangePort);
        // Counter to track how many ports have been scanned so far.
        AtomicInteger scannedPorts = new AtomicInteger(0);

        // Loop through each port in the range.
        while (ports.get() <= topRangePort) {
            // Generate the port to scan.
            int port = ports.getAndIncrement();

            // Submit a task to the executor to scan the port.
            executorService.submit(() -> {
                // Scan the port.
                scanPort(port);

                // Update the progress bar after scanning each port.
                updateProgressBar(scannedPorts.getAndIncrement());
            });
        }

        // Initiate shutdown and wait for tasks to finish, with a maximum wait time.
        executorService.shutdown();
        try {
            // Wait until all tasks have finished, or timeout after 10 minutes.
            if (!executorService.awaitTermination(10, TimeUnit.MINUTES)) {
                // If tasks are not finished in 10 minutes, force shutdown.
                executorService.shutdownNow();
            }
        } catch (InterruptedException e) {
            // If the thread is interrupted, reset the interrupt flag and exit the loop.
            executorService.shutdownNow();
            Thread.currentThread().interrupt();
        }
    }

    /**
     * Updates the progress bar based on the number of ports scanned.
     * <p>
     * The progress is calculated as a percentage of the total number of ports
     * scanned.
     * </p>
     *
     * @param portNum the number of ports scanned so far
     */
    private void updateProgressBar(int portNum) {
            invokeLater(()  -> prgPortScan.setValue((int) Math.round(((double) portNum /
(topRangePort - bottomRangePort + 1)) * 100)));
    }

    /**
     * Forcefully shuts down the executor service, stopping any running tasks.
     */
    public void shutDownExecutorService() {
        executorService.shutdownNow();
    }

    /**
     * Requests the port scan to stop.
     * <p>
     * The {@code stopRequested} flag is set to true, so that ongoing tasks in
     * {@code start} may check this flag and terminate early.
     * </p>
     */
    public void requestStop() {
        stopRequested = true;
    }

    /**
     * Checks whether a stop has been requested for the scan.
     *
     * @return {@code true} if a stop has been requested; {@code false}
     * otherwise.
     */
    public boolean isStopRequested() {
        return stopRequested;
    }

    /**
     * Retrieves the list of port scan results.
     *
     * @return an {@code ArrayList} of {@code PortScanResult} objects
     */
    public ArrayList<PortScanResult> getPortScanResults() {
        return portScanResults;
    }

    /**
     * Sets the port scan results.
     * <p>
     * Used when importing results, i.e. when no port scan was performed to have
     * added the open ports to the port scan results list.
     * </p>
     *
     * @param portScanResults an {@code ArrayList} of {@code PortScanResult}
     * objects
     */
    public void setPortScanResults(ArrayList<PortScanResult> portScanResults) {
        this.portScanResults = portScanResults;
    }

    /**
     * Returns the target IP address.
     *
     * @return the IP address as a {@code String}
     */
    public String getIpAddress() {
        return ipAddress;
    }

    /**
     * Returns the starting port number of the scan range.
     *
     * @return the bottom range port as an {@code int}
     */
    public int getBottomRangePort() {
        return bottomRangePort;
    }

    /**
     * Returns the ending port number of the scan range.
     *
     * @return the top range port as an {@code int}
     */
    public int getTopRangePort() {
        return topRangePort;
    }

    /**
     * Returns the timeout used for each port connection attempt.
     *
     * @return the timeout in milliseconds as an {@code int}
     */
    public int getTimeout() {
        return timeout;
    }
}
```

Protocols.java

```java
package com.pingpal.portscan;

import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.nio.charset.StandardCharsets;
import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;
import javax.swing.JOptionPane;

/**
 * The {@code Protocols} class loads and stores a mapping between TCP/UDP port
 * numbers and their corresponding protocol names from a CSV file.
 * <p>
 * The CSV file is expected to be located at:
 * <code>./src/com/pingpal/resources/databases/port_list.csv</code> and must
 * contain a header line followed by lines where the first field is a port
 * number and the second field is the protocol name.
 * </p>
 * <p>
 * If the file is not found, the class displays an error message using a
 * JOptionPane.
 * </p>
 */
public class Protocols {

    /**
     * A mapping from port numbers (as {@code Integer}) to protocol names (as
     * {@code String}).
     */
    private static Map<Integer, String> portProtocolMap = new HashMap<>();

    // Resource path inside the JAR / classpath.
    private static final String RESOURCE_PATH =
"/com/pingpal/resources/databases/port_list.csv";

    // Private constructor to prevent instantiation.
    private Protocols() {

    }

    // Static initialiser to load the data from the classpath.
    static {
        loadFromClasspath();
    }

    /**
     * Populates the port number to protocol tool.
     * <p>
     * This method reads the port list from a CSV file and populates the
     * {@code portProtocolMap}. It skips the first line assuming it is a header.
     * </p>
     * <p>
     * If the CSV file is not found, an error dialog is shown.
     * </p>
     */
    private static void loadFromClasspath() {
        // Load the resource file.
        try (InputStream isFile = Protocols.class.getResourceAsStream(RESOURCE_PATH)) {
            // Check whether the file exists.
            if (isFile == null) {
                // Show an error message if the file cannot be found.
                JOptionPane.showMessageDialog(null, "Port list not found.", "File Not Found
Error", JOptionPane.ERROR_MESSAGE);
                return;
            }

            // Open scanner for reading.
            try (Scanner scFile = new Scanner(new InputStreamReader(isFile,
StandardCharsets.UTF_8))) {
                // Skip the header line.
                scFile.nextLine();
                // Read each subsequent line in the CSV file.
                while (scFile.hasNextLine()) {
                    // Split the line on commas
                    String[] line = scFile.nextLine().split(",");
                    // Parse the port number (first column).
                    int port_number = Integer.parseInt(line[0].trim());
                    // Get the protocol name (second column).
                    String protocol = line[1].trim();
                    // Add the port-to-protocol mapping.
                    portProtocolMap.put(port_number, protocol);
                }
            }

            // Catch general IO exceptions.
        } catch (IOException e) {
            JOptionPane.showMessageDialog(null, "Error reading port list resource.",
                    "File Read Error", JOptionPane.ERROR_MESSAGE);
        }
    }

    /**
     * Retrieves the protocol name associated with the specified port number.
     * <p>
     * If no protocol is found for the given port, a default message is
     * returned.
     * </p>
     *
     * @param portNumber the port number for which to retrieve the protocol
     * @return the protocol name corresponding to the port number, or "No
     * specific protocol associated with this port." if the port is not mapped
     */
    public static String getProtocolForPort(int portNumber) {
        return portProtocolMap.getOrDefault(portNumber, "No specific protocol associated with
this port.");
    }
}
```

TCPMessageListen.java

```java
package com.pingpal.tcpmessage.listen;

import java.awt.Color;
import static java.awt.EventQueue.invokeLater;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.BindException;
import java.net.InetAddress;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketTimeoutException;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import javax.swing.JOptionPane;
import javax.swing.JTextPane;
import javax.swing.text.BadLocationException;
import javax.swing.text.Style;
import javax.swing.text.StyleConstants;
import javax.swing.text.StyledDocument;

/**
 * Handles incoming TCP connections, receives and sends messages, and formats
 * the communication in a styled text pane.
 * <p>
 * This class starts a TCP server on a specified port, listens for client
 * connections, and enables message exchange using a JTextPane.
 * </p>
 */
public class TCPMessageListen {

    // Color constants for different message types.
    private final Color SUCCESS_COLOR = new Color(0, 204, 0);
    private final Color ERROR_COLOR = new Color(255, 51, 0);
    private final Color MESSAGE_COLOR = new Color(45, 45, 45);
    private final Color DATE_TIME_COLOR = new Color(26, 39, 107);
    private final Color HOSTNAME_COLOR = new Color(113, 89, 138);

    // Text styling for message formatting.
    private Style dateTimeStyle;
    private Style hostnameStyle;
    private Style messageStyle;
    private Style errorStyle;
    private Style successStyle;

    // UI components.
    private JTextPane txpTCPMessageListen;
    private StyledDocument doc;

    // Flag to indicate if a stop has been requested for the scan.
    private boolean stopRequested = false;

    // Socket variables.
    private ServerSocket serverSocket;
    private Socket clientSocket;

    // Reader and writer to handle sending and receiving messages.
    private BufferedReader in;
    private PrintWriter out;

    // The port on which the server will listen for connections.
    private int port;

    /**
     * Constructs a new {@code TCPMessageListen} instance with the specified
     * port, and text pane for messages.
     *
     * @param port the port on which to listen for connections
     * @param txpTCPMessageListen the JTextPane used to display the messages
     */
    public TCPMessageListen(int port, JTextPane txpTCPMessageListen) {
        this.port = port;
        this.txpTCPMessageListen = txpTCPMessageListen;

        // Extract the styled document from the text pane.
        doc = txpTCPMessageListen.getStyledDocument();
        // Initialise the different message styles.
        setStyles();
    }

    /**
     * Starts the server socket, waits (up to 60s) for a client, and then
     * continuously receives and displays messages.
     * <p>
     * If no client connects within 60 seconds, a timeout occurs, the socket is
     * closed, and an error message is displayed.
     * </p>
     */
    public void start() {
        try {
            // Clear any data that is currently displayed in the text pane.
            doc.remove(0, doc.getLength());

            // Initialise the server socket, and begin listening on the specified port.
            serverSocket = new ServerSocket(port);
            // Set 60000ms (60s) accept timeout.
            serverSocket.setSoTimeout(60_000);
            // Write to the text pane to indicate the program is waitng for a client
connection.
            updateTextPane("Waiting for client connection on " +
InetAddress.getLocalHost().getHostAddress() + ":" + port + ".\n", messageStyle);

            // Accept device to connect and initialise the client socket.
            clientSocket = serverSocket.accept();

            // Get and format client IP address.
            String clientIP = clientSocket.getInetAddress().getHostAddress();
            // Write to the text pane to indicate a client has connected.
            updateTextPane("Client connected: " + clientIP + "\n", successStyle);

            // Initialise the persistent reader and writer.
            in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
            out = new PrintWriter(clientSocket.getOutputStream(), true);

            // Enter receive loop.
            receiveMessages();

            // Throw an error if no clinet connected within 60s.
        } catch (SocketTimeoutException e) {
            // Stop the program.
            requestStop();
            // Write to the text pane to indicate no client connected within 60s.
            updateTextPane("No client connected within 60 seconds. Closing server socket.\n",
errorStyle);

            // Throw an error if the specified port is already open and being used.
        } catch (BindException e) {
            // Write to the text pane to indicate that the program is already used.
            updateTextPane("Port alread in use.\n", errorStyle);

            // General I/O error, but its most common use case is when the client
disconnects.
        } catch (IOException e) {
            // Write to the text pane to indicate the connection closed.
            updateTextPane("Connection closed.\n", errorStyle);

            // Catch exceptions that may occur when trying to append a message to the text
pane.
        } catch (BadLocationException ex) {
            // Display a JOptionPane to indicate such.
            JOptionPane.showMessageDialog(txpTCPMessageListen.getParent(), "Error occurred
during text pane update process.", "Text Pane Write Error", JOptionPane.ERROR_MESSAGE);
        }
    }

    /**
     * Sends a message to the connected client and echoes it locally with
     * timestamp and hostname formatting.
     *
     * @param message the text to send
     */
    public void sendMessage(String message) {
        try {
            // Get and format the current date and time.
            LocalDateTime now = LocalDateTime.now();
            String formattedDateTime = now.format(DateTimeFormatter.ofPattern("dd-MM-yy
HH:mm:ss"));

            // Get the host name of the local device.
            String hostname = InetAddress.getLocalHost().getHostName();

            // Write the formatted message to the text pane.
            updateTextPane(formattedDateTime + " ", dateTimeStyle);
            updateTextPane("[" + hostname + "] ", hostnameStyle);
            updateTextPane("> " + message + "\n", messageStyle);

            // Print the message for the connected device.
            out.println(formattedDateTime + " [" + hostname + "] > " + message);

            // General I/O error, but its most common use case is when the client
disconnects.
        } catch (IOException e) {
            // Write to the text pane to indicate the client disconnected.
            updateTextPane("Client disconnected.", errorStyle);
        }

    }

    /**
     * Continuously receives incoming messages from the client, formats them,
     * and appends them to the text pane.
     */
    private void receiveMessages() {
        // Loop until the client disconnects or the user chooses to stop listening for
messages and disconnect themself.
        try {
            while (!stopRequested) {
                // Declare the variable to hold the message the client sends.
                String message;
                // Loop to keep checking whether the client has sent a message.
                while ((message = in.readLine()) != null) {
                    // Check whether the message came via a PingPal connection, by checking
format.
                    if (message.contains("[") && message.contains("]")
                            && message.contains(">")) {
                        // Write the formatted message to the text pane.
                        updateTextPane(message.substring(0,
                                message.indexOf("[")), dateTimeStyle);
                        updateTextPane(message.substring(message.indexOf("["),
                                message.indexOf(">")), hostnameStyle);
                        updateTextPane(message.substring(message.indexOf(">"))
                                + "\n", messageStyle);
                    } else {
                        // Indicate that the message does not come from a PingPal connection,
however, still display it.
                        updateTextPane("Not connected to a device via PingPal. However, the
message reads:\n", errorStyle);
                        updateTextPane(message + "\n", messageStyle);
                    }
                }
            }

            // General I/O error, but it's most common use case is when the client
disconnects.
        } catch (IOException e) {
            // Write to the text pane to indicate the client disconnected.
            updateTextPane("Client disconnected.", errorStyle);
        }
    }

    /**
     * Appends a styled message to the JTextPane safely on the Event Dispatch
     * Thread.
     *
     * @param message the message text to append
     * @param style the Style to apply to the message
     */
    private void updateTextPane(String message, Style style) {
        invokeLater(() -> {
            try {
                // Append the passed message in the passed style.
                doc.insertString(doc.getLength(), message, style);

                // Catch exceptions that may occur when trying to append a message to the
text pane.
            } catch (BadLocationException e) {
                JOptionPane.showMessageDialog(txpTCPMessageListen.getParent(), "Error
occurred during text pane update process.", "Text Pane Write Error",
JOptionPane.ERROR_MESSAGE);
            }
        });
    }
    /**
     * Defines and registers custom styles used for formatting different types
     * of messages in the document.
     */
    private void setStyles() {
        // Initialise and register the date & time style.
        dateTimeStyle = doc.addStyle("dateTimeStyle", null);
        StyleConstants.setForeground(dateTimeStyle, DATE_TIME_COLOR);

        // Initialise and register the hostname style.
        hostnameStyle = doc.addStyle("hostnameStyle", null);
        StyleConstants.setForeground(hostnameStyle, HOSTNAME_COLOR);

        // Initialise and register the message style.
        messageStyle = doc.addStyle("messageStyle", null);
        StyleConstants.setForeground(messageStyle, MESSAGE_COLOR);

        // Initialise and register the error style.
        errorStyle = doc.addStyle("errorStyle", null);
        StyleConstants.setForeground(errorStyle, ERROR_COLOR);

        // Initialise and register the success style.
        successStyle = doc.addStyle("successStyle", null);
        StyleConstants.setForeground(successStyle, SUCCESS_COLOR);

        // Set the styled document of the text pane.
        txpTCPMessageListen.setStyledDocument(doc);
    }

    /**
     * Requests that the pinging process stop.
     * <p>
     * This sets the {@code stopRequested} flag to true, so that the ping loop
     * in {@code start()} will terminate early, updates the text pane to
     * indicate the connection is being closed, and close both the client socket
     * and server socket.
     * </p>
     */
    public void requestStop() {
        // Set the stopRequested flag to true.
        stopRequested = true;

        // Write to the text pane to indicate that the sockets are being closed.
        updateTextPane("Exiting TCP Message and closing sockets.\n", errorStyle);
        try {
            // Close the client socket.
            if (clientSocket != null && !clientSocket.isClosed()) {
                clientSocket.close();
            }
            // Close the server socket.
            if (serverSocket != null && !serverSocket.isClosed()) {
                serverSocket.close();
            }

            // General I/O error, but it's most common use case is when the sockets are
already closed.
        } catch (IOException e) {
            updateTextPane("Sockets closed already.\n", errorStyle);
        }
    }

    /**
```

```java
     * Checks whether a client is currently connected.
     *
     * @return {@code true} if the client socket is connected; otherwise
     * {@code false}
     */
    public boolean isDeviceConnected() {
        return clientSocket != null && clientSocket.isConnected();
    }

    /**
     * Retrieves the full contents of the text pane's document.
     *
     * @return a {@code String} containing the entire text pane output
     * @throws BadLocationException if the text cannot be accessed
     */
    public String getTextPaneContents() throws BadLocationException {
        return doc.getText(0, doc.getLength());
    }
}
```

TCPMessageConnect.java

```java
package com.pingpal.tcpmessage.connect;

import java.awt.Color;
import static java.awt.EventQueue.invokeLater;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ConnectException;
import java.net.InetAddress;
import java.net.Socket;
import java.net.SocketTimeoutException;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import javax.swing.JOptionPane;
import javax.swing.JTextPane;
import javax.swing.text.BadLocationException;
import javax.swing.text.Style;
import javax.swing.text.StyleConstants;
import javax.swing.text.StyledDocument;

/**
 * Establishes a TCP connection to a server, receives and sends messages, and
 * formats the communication in a styled text pane.
 * <p>
 * This class establishes a connection to a TCP server on a specified port, and
 * enables message exchange using a JTextPane.
 * </p>
 */
public class TCPMessageConnect {

    // Color constants for different message types.
    private final Color SUCCESS_COLOR = new Color(0, 204, 0);
    private final Color ERROR_COLOR = new Color(255, 51, 0);
    private final Color MESSAGE_COLOR = new Color(45, 45, 45);
    private final Color DATE_TIME_COLOR = new Color(26, 39, 107);
    private final Color HOSTNAME_COLOR = new Color(113, 89, 138);

    // Text styling for message formatting.
    private Style dateTimeStyle;
    private Style hostnameStyle;
    private Style messageStyle;
    private Style errorStyle;
    private Style successStyle;

    // UI components.
    private JTextPane txpTCPMessageConnect;
    private StyledDocument doc;

    // Flag to indicate if a stop has been requested for the scan.
    private boolean stopRequested = false;

    // Socket variables.
    private Socket socket;

    // Reader and writer to handle sending and receiving messages.
    private BufferedReader in;
    private PrintWriter out;
    // The server IP address the program will try to connect to.
    private String ipAddress;

    // The server port the socket will try to connect to.
    private int port;

    /**
     * Constructs a new {@code TCPMessageConnect} instance with the specified IP
     * address, port, and text pane for messages.
     *
     * @param ipAddress the IP address to which the socket will attempt to
     * connect to
     * @param port the port on which to listen for connections
     * @param txpTCPMessageConnect the JTextPane used to display the messages
     */
    public TCPMessageConnect(String ipAddress, int port, JTextPane txpTCPMessageConnect) {
        this.ipAddress = ipAddress;
        this.port = port;
        this.txpTCPMessageConnect = txpTCPMessageConnect;

        // Extract the styled document from the text pane.
        doc = txpTCPMessageConnect.getStyledDocument();
        // Initialise the different message styles.
        setStyles();
    }

    /**
     * Starts the client socket, attempts to connect to a server socket, and
     * then continuously receives and displays messages.
     * <p>
     * If no client connects within 60 seconds, a timeout occurs, the socket is
     * closed, and an error message is displayed.
     * </p>
     */
    public void start() {
        try {
            // Clear any data that is currently displayed in the text pane.
            doc.remove(0, doc.getLength());

            // Write to the text pane to indicate the program is trying to establish a
connection.
            updateTextPane("Trying to establish a connection to " + ipAddress + ":" + port +
".\n", messageStyle);

            // Attempt to connect to a device on the specified IP and port.
            socket = new Socket(ipAddress, port);
            // Set 60000ms (60s) accept timeout.
            socket.setSoTimeout(60_000);
            // Write to the text pane to indicate that connection to server is successful.
            updateTextPane("Connected to chat server at " + ipAddress + ":" + port + ".\n",
successStyle);

            // Initialize the persistent reader and writer.
            in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            out = new PrintWriter(socket.getOutputStream(), true);

            // Enter receive loop.
            receiveMessages();

            // Throw an error if the server refuses connection.
        } catch (ConnectException e) {
            // Stop the program.
            requestStop();
            // Write to the pane that the server has refused the connection request.
            updateTextPane("Server refused connection. Closing socket.\n", errorStyle);

            // Throw an error if no clinet connected within 60s.
        } catch (SocketTimeoutException e) {
            // Stop the program.
            requestStop();
            // Write to the text pane to indicate no client connected within 60s.
            updateTextPane("No client connected within 60 seconds. Closing socket.\n",
errorStyle);

            // General I/O error, but its most common use case is when the client
disconnects.
        } catch (IOException e) {
            // Write to the text pane to indicate the connection closed.
            updateTextPane("Connection closed.\n", errorStyle);

            // Catch exceptions that may occur when trying to append a message to the text
pane.
        } catch (BadLocationException ex) {
            // Display a JOptionPane to indicate such.
            JOptionPane.showMessageDialog(txpTCPMessageConnect.getParent(), "Error occurred
during text pane update process.", "Text Pane Write Error", JOptionPane.ERROR_MESSAGE);
        }
    }

    /**
     * Sends a message to the connected client and echoes it locally with
     * timestamp and hostname formatting.
     *
     * @param message the text to send
     */
    public void sendMessage(String message) {
        try {
            // Get and format the current date and time.
            LocalDateTime now = LocalDateTime.now();
            DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd-MM-yy HH:mm:ss");
            String formattedDateTime = now.format(formatter);

            // Get the host name of the local device.
            String hostname = InetAddress.getLocalHost().getHostName();

            // Write the formatted message to the text pane.
            updateTextPane(formattedDateTime + " ", dateTimeStyle);
            updateTextPane("[" + hostname + "] ", hostnameStyle);
            updateTextPane("> " + message + "\n", messageStyle);

            // Print the message for the connected device.
            out.println(formattedDateTime + " [" + hostname + "] > " + message);

            // General I/O error, but its most common use case is when the server
disconnects.
        } catch (IOException e) {
            // Write to the text pane to indicate the server disconnected.
            updateTextPane("Server disconnected.\n", errorStyle);
        }
    }

    /**
     * Continuously receives incoming messages from the client, formats them,
     * and appends them to the text pane.
     */
    private void receiveMessages() {
        try {
            // Loop until the server disconnects or the user chooses to stop listening for
messages and disconnect themself.
            while (!stopRequested) {
                // Declare the variable to hold the message the server sends.
                String message;
                // Loop to keep checking whether the server has sent a message.
                while ((message = in.readLine()) != null) {
                    // Check whether the message came via a PingPal connection, by checking
format.
                    if (message.contains("[") && message.contains("]")
                            && message.contains(">")) {
                        // Write the formatted message to the text pane.
                        updateTextPane(message.substring(0,
                                message.indexOf("[")), dateTimeStyle);
                        updateTextPane(message.substring(message.indexOf("["),
                                message.indexOf(">")), hostnameStyle);
                        updateTextPane(message.substring(message.indexOf(">"))
                                + "\n", messageStyle);
                    } else {
                        // Indicate that the message does not come from a PingPal connection,
however, still display it.
                        updateTextPane("Not connected to a device via PingPal. However, the
message reads:\n", errorStyle);
                        updateTextPane(message + "\n", messageStyle);
                    }
                }
            }

            // General I/O error, but it's most common use case is when the server
disconnects.
        } catch (IOException e) {
            // Write to the text pane to indicate the server disconnected.
            updateTextPane("Server disconnected.\n", errorStyle);
        }
    }

    /**
     * Appends a styled message to the JTextPane safely on the Event Dispatch
     * Thread.
     *
     * @param message the message text to append
     * @param style the Style to apply to the message
     */
    private void updateTextPane(String message, Style style) {
        invokeLater(() -> {
            try {
                // Append the passed message in the passed style.
                doc.insertString(doc.getLength(), message, style);

                // Catch exceptions that may occur when trying to append a message to the
text pane.
            } catch (BadLocationException e) {
                JOptionPane.showMessageDialog(txpTCPMessageConnect.getParent(), "Error
occurred during text pane update process.", "Text Pane Write Error",
JOptionPane.ERROR_MESSAGE);
            }
        });
    }
    /**
     * Defines and registers custom styles used for formatting different types
     * of messages in the document.
     */
    private void setStyles() {
        // Initialise and register the date & time style.
        dateTimeStyle = doc.addStyle("dateTimeStyle", null);
        StyleConstants.setForeground(dateTimeStyle, DATE_TIME_COLOR);

        // Initialise and register the hostname style.
        hostnameStyle = doc.addStyle("hostnameStyle", null);
        StyleConstants.setForeground(hostnameStyle, HOSTNAME_COLOR);

        // Initialise and register the message style.
        messageStyle = doc.addStyle("messageStyle", null);
        StyleConstants.setForeground(messageStyle, MESSAGE_COLOR);

        // Initialise and register the error style.
        errorStyle = doc.addStyle("errorStyle", null);
        StyleConstants.setForeground(errorStyle, ERROR_COLOR);

        // Initialise and register the success style.
        successStyle = doc.addStyle("successStyle", null);
        StyleConstants.setForeground(successStyle, SUCCESS_COLOR);

        // Set the styled document of the text pane.
        txpTCPMessageConnect.setStyledDocument(doc);
    }

    /**
     * Requests that the pinging process stop.
     * <p>
     * This sets the {@code stopRequested} flag to true, so that the ping loop
     * in {@code start()} will terminate early, updates the text pane to
     * indicate the connection is being closed, and close both the client socket
     * and server socket.
     * </p>
     */
    public void requestStop() {
        // Set the stopRequested flag to true.
        stopRequested = true;

        // Write to the text pane to indicate that the sockets are being closed.
        updateTextPane("Exiting TCP Message and closing socket.\n", errorStyle);
        try {
            // Close the socket.
            if (socket != null && !socket.isClosed()) {
                socket.close();
            }

            // General I/O error, but it's most common use case is when the socket is already
closed.
        } catch (IOException e) {
            updateTextPane("Socket closed already.\n", errorStyle);
        }
    }

    /**
     * Checks whether a client is currently connected.
     *
     * @return {@code true} if the client socket is connected; otherwise
     * {@code false}
     */
    public boolean isDeviceConnected() {
        return socket != null && socket.isConnected();
    }

    /**
     * Retrieves the full contents of the text pane's document.
     *
     * @return a {@code String} containing the entire text pane output
     * @throws BadLocationException if the text cannot be accessed
     */
    public String getTextPaneContents() throws BadLocationException {
        return doc.getText(0, doc.getLength());
    }
}
```

ExportResults.java

```java
package com.pingpal.exports;

import com.pingpal.deviceping.DevicePing;
import com.pingpal.deviceping.DevicePingResult;
import com.pingpal.portscan.PortScan;
import com.pingpal.portscan.PortScanResult;
import com.pingpal.subnetscan.SubnetScan;
import com.pingpal.subnetscan.SubnetScanResult;
import com.pingpal.tcpmessage.connect.TCPMessageConnect;
import com.pingpal.tcpmessage.listen.TCPMessageListen;
import java.io.FileWriter;
import java.io.IOException;
import java.nio.file.Path;
import javax.swing.JFileChooser;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.text.BadLocationException;
import org.json.JSONArray;
import org.json.JSONObject;

/**
 * Handles exporting of various scan and chat results to JSON or text files.
 * <p>
 * Prompts the user for a directory and file name, then writes the results of
 * SubnetScan, DevicePing, PortScan, or TCPMessage sessions to disk in the
 * respective format.
 * </p>
 */
public class ExportResults {

    // Parent UI panel for dialogueues.
    private JPanel panel;

    // File chooser for directory selection.
    private JFileChooser fchDirectoryChooser;

    // Path of the directory selected for export.
    private Path exportResultsPath;

    // Base name for the output file (no extension).
    private String fileName;

    /**
     *
     * Constructs a new {@code ExportResults} instance with the specified panel,
     * immediately asking the user to choose a directory, then a file name.
     *
     * @param panel the Swing panel used as parent for dialogue
     */
    public ExportResults(JPanel panel) {
        this.panel = panel;

        // Configure chooser to pick directories only.
        fchDirectoryChooser = new JFileChooser();
        fchDirectoryChooser.setDialogTitle("Select a directory");
        fchDirectoryChooser.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);

        // Prompt for directory.
        setExportResultsPath();

        // If successful, then prompt for file name.
        if (exportResultsPath != null) {
            setFileName();
        }
    }

    /**
     * Shows the directory chooser and stores the chosen path. If the user
     * cancels, exportResultsPath remains null.
     */
    private void setExportResultsPath() {
        // Displays the directory chooser.
        int returnVal = fchDirectoryChooser.showOpenDialog(panel);
        if (returnVal == JFileChooser.APPROVE_OPTION) {
            // Convert the selected File to Path.
            exportResultsPath = fchDirectoryChooser.getSelectedFile().toPath();
        }
    }

    /**
     * Repeatedly prompts the user for a valid file name (no blanks, periods, or
     * slashes) until one is entered.
     */
    private void setFileName() {
        // Loop until user provides acceptable name.
        while (fileName == null) {
            // Input dialogueue returns null if cancelled.
            String input = "" + JOptionPane.showInputDialog(panel, "Enter a file name:",
"File Name Input", JOptionPane.QUESTION_MESSAGE);

            // Guard against cancel or blank input.
            if (input.isBlank() || input.equals("null")) {
                // Display corresponding error message in a message dialogueue.
                JOptionPane.showMessageDialog(panel, "File name cannot be blank.", "Blank
File Name Error", JOptionPane.ERROR_MESSAGE);

                // Guard against a file name that contains a period.
            } else if (input.contains(".")) {
                // Display corresponding error message in a message dialogueue.
                JOptionPane.showMessageDialog(panel, "File name cannot contain a fullstop.",
"Invalid Format Error", JOptionPane.ERROR_MESSAGE);

                // Guard against a file name that contains a forward slash and/or back slash.
            } else if (input.contains("\\") || input.contains("/")) {
                // Display corresponding error message in a message dialogueue.
                JOptionPane.showMessageDialog(panel, "File name cannot contain a slash.",
"Invalid Format Error", JOptionPane.ERROR_MESSAGE);

                // If the name is valid, set the file name to the user input, and exit the
loop.
            } else {
                fileName = input;
            }
        }
    }

    /**
     * Writes a {@link JSONObject} to a .json file in the chosen directory,
     * showing a success or error dialogue upon completion.
     *
     * @param output the JSON object to write
     */
    private void writeToJSONFile(JSONObject output) {
        // Construct full file path with .json extension.
        // Try-with-resources to ensure FileWriter is closed.
        try (FileWriter file = new FileWriter(exportResultsPath + "\\" + fileName + ".json"))
{
            // Use toString(4) for pretty printing with 4 space indentation.
            file.write(output.toString(4));

            // If successful, display a success message in a message dialogue.
            JOptionPane.showMessageDialog(panel, "Successfully exported results to \"" +
fileName + ".json\"", "Successful Export", JOptionPane.INFORMATION_MESSAGE);

            // General I/O error.
        } catch (IOException e) {
            // If unsuccessful, display an error message in a message dialogue.
            JOptionPane.showMessageDialog(panel, "Error occured during file writing
process.", "File Writing Error", JOptionPane.ERROR_MESSAGE);
        }
    }

    /**
     * Writes plain text to a .txt file in the chosen directory, showing a
     * success or error dialogue upon completion.
     *
     * @param txt the text content to write
     */
    private void writeToTextFile(String txt) {
        // Construct full file path with .txt extension.
        // Try-with-resources to ensure FileWriter is closed.
        try (FileWriter file = new FileWriter(exportResultsPath + "\\" + fileName + ".txt"))
{
            // Write to the file.
            file.write(txt);

            // If successful, display a success message in a message dialogue.
            JOptionPane.showMessageDialog(panel, "Successfully exported results to \"" +
fileName + ".txt\"", "Successful Export", JOptionPane.INFORMATION_MESSAGE);

            // General I/O error.
        } catch (IOException e) {
            // If unsuccessful, display an error message in a message dialogue.
            JOptionPane.showMessageDialog(panel, "Error occured during file writing
process.", "File Writing Error", JOptionPane.ERROR_MESSAGE);
        }
    }

    /**
     * Exports the results of a {@link SubnetScan} to JSON.
     * <p>
     * This overload uses {@link SubnetScan} as the scan type.
     * </p>
     *
     * @param subnetScan the scan whose results to export
     */
    public void exportResults(SubnetScan subnetScan) {
        // Guard against a blank subnet scan, i.e. if no scan has been performed.
        if (subnetScan == null) {
            // Display corresponding error message in a message dialogue.
            JOptionPane.showMessageDialog(panel, "Cannot export results as no subnet scan has
been performed.", "Null Subnet Scan Error", JOptionPane.ERROR_MESSAGE);
```

```
            return;
        }

        // Guard against a blank file path, i.e. if the user has not selected a path.
        if (exportResultsPath == null) {
            // Display corresponding error message in a message dialogue.
            JOptionPane.showMessageDialog(panel, "Results not exported as no directory was
selected.", "File Path Error", JOptionPane.ERROR_MESSAGE);
            return;
        }

        // Create a blank JSON array to hold all subnet scan results.
        JSONArray resultsArray = new JSONArray();

        // Loop through each subnet scan result.
        for (SubnetScanResult result : subnetScan.getSubnetScanResults()) {
            // Create a blank JSON object to hold the individual subnet scan results.
            JSONObject resultObj = new JSONObject();

            // Append the data from the subnet scan result to the JSON object.
            resultObj.put("ipAddress", result.getIPAddress());

            // Append the JSON object to the JSON array.
            resultsArray.put(resultObj);
        }

        // Create a top-level JSON object.
        JSONObject output = new JSONObject();

        // Wrap the metadata and array in the top-level JSON object.
        output.put("networkRange", subnetScan.getNetworkRange());
        output.put("timeout", subnetScan.getTimeout());
        output.put("subnetScanResults", resultsArray);

        // Write the data to the file.
        writeToJSONFile(output);
    }

    /**
     * Exports the results of a {@link DevicePing} to JSON.
     * <p>
     * This overload uses {@link DevicePing} as the scan type.
     * </p>
     *
     * @param devicePing the ping sessions whose results to export
     */
    public void exportResults(DevicePing devicePing) {
        // Guard against a blank device ping, i.e. if no ping has been performed.
        if (devicePing == null) {
            // Display corresponding error message in a message dialogue.
            JOptionPane.showMessageDialog(panel, "Cannot export results as no device ping has
been performed.", "Null Device Ping Error", JOptionPane.ERROR_MESSAGE);
            return;
        }

        // Guard against a blank file path, i.e. if the user has not selected a path.
        if (exportResultsPath == null) {
            // Display corresponding error message in a message dialogue.
            JOptionPane.showMessageDialog(panel, "Results not exported as no directory was
selected.", "File Path Error", JOptionPane.ERROR_MESSAGE);
            return;
        }
        // Create a blank JSON array to hold all device ping results.
        JSONArray resultsArray = new JSONArray();

        // Loop through each device ping result.
        for (DevicePingResult result : devicePing.getDevicePingResults()) {
            // Create a blank JSON object to hold the individual device ping results.
            JSONObject resultObj = new JSONObject();

            // Append the data from the subnet scan result to the JSON object.
            resultObj.put("roundTripTime", result.getRoundTripTime());
            resultObj.put("successfulPing", result.isSuccessfulPing());
            resultObj.put("packetLoss", result.getPacketLoss());

            // Append the JSON object to the JSON array.
            resultsArray.put(resultObj);
        }

        // Create a top-level JSON object.
        JSONObject output = new JSONObject();

        // Wrap the metadata and array in the top-level JSON object.
        output.put("ipAddress", devicePing.getIpAddress());
        output.put("pingInterval", devicePing.getPingInterval());
        output.put("numOfPings", devicePing.getNumOfPings());
        output.put("continuousPinging", devicePing.isContinuousPinging());
        output.put("devicePingResults", resultsArray);

        // Write the data to the file.
        writeToJSONFile(output);
    }

    /**
     * Exports the results of a {@link PortScan} to JSON.
     * <p>
     * This overload uses {@link PortScan} as the scan type.
     * </p>
     *
     * @param portScan the port scan whose results to export
     */
    public void exportResults(PortScan portScan) {
        // Guard against a blank port scan, i.e. if no scan has been performed.
        if (portScan == null) {
            // Display corresponding error message in a message dialogue.
            JOptionPane.showMessageDialog(panel, "Cannot export results as no port scan has
been performed.", "Null Port Scan Error", JOptionPane.ERROR_MESSAGE);
            return;
        }

        // Guard against a blank file path, i.e. if the user has not selected a path.
        if (exportResultsPath == null) {
            // Display corresponding error message in a message dialogue.
            JOptionPane.showMessageDialog(panel, "Results not exported as no directory was
selected.", "File Path Error", JOptionPane.ERROR_MESSAGE);
            return;
        }

        // Create a JSON array to hold all port scan results.
        JSONArray resultsArray = new JSONArray();

        // Loop through each port scan result.
        for (PortScanResult result : portScan.getPortScanResults()) {
            // Create a blank JSON object to hold the individual port scan results.
            JSONObject resultObj = new JSONObject();

            // Append the data from the port scan result to the JSON object.
            resultObj.put("portNumber", result.getPortNumber());
            resultObj.put("protocol", result.getProtocol());

            // Append the JSON object to the JSON array.
            resultsArray.put(resultObj);
        }

        // Create a top-level JSON object.
        JSONObject output = new JSONObject();

        // Wrap the metadata and array in the top-level JSON object.
        output.put("ipAddress", portScan.getIpAddress());
        output.put("bottomRangePort", portScan.getBottomRangePort());
        output.put("topRangePort", portScan.getTopRangePort());
        output.put("timeout", portScan.getTimeout());
        output.put("portScanResults", resultsArray);

        // Write the data to the file.
        writeToJSONFile(output);

    }

    /**
     * Exports the results of a {@link TCPMessageListen} to a text file.
     * <p>
     * This overload uses {@link TCPMessageListen} as the scan type.
     * </p>
     *
     * @param tcpMessageListen the server side messages to export
     */
    public void exportResults(TCPMessageListen tcpMessageListen) {
        // Guard against a blank TCP message listen, i.e. if no messages have been exchanged.
        if (tcpMessageListen == null) {
            // Display corresponding error message in a message dialogue.
            JOptionPane.showMessageDialog(panel, "Cannot export results as no TCP message 
listen has been performed.", "Null TCP Message Listen Error", JOptionPane.ERROR_MESSAGE);
            return;

        }

        // Guard against a blank file path, i.e. if the user has not selected a path.
        if (exportResultsPath == null) {
            // Display corresponding error message in a message dialogue.
            JOptionPane.showMessageDialog(panel, "Results not exported as no directory was 
selected.", "File Path Error", JOptionPane.ERROR_MESSAGE);
            return;
        }

        try {
            // Write the data to the file.
            writeToTextFile(tcpMessageListen.getTextPaneContents());
        } catch (BadLocationException e) {
            // Display an error message in a message dialogue if an error occurs.
            JOptionPane.showMessageDialog(panel, "Error occured during file writing 
process.", "File Writing Error", JOptionPane.ERROR_MESSAGE);
        }

    }
    /**
     * Exports the results of a {@link TCPMessageConnect} to a text file.
     * <p>
     * This overload uses {@link TCPMessageConnect} as the scan type.
     * </p>
     *
     * @param tcpMessageConnect the server side messages to export
     */
    public void exportResults(TCPMessageConnect tcpMessageConnect) {
        // Guard against a blank TCP message connect, i.e. if no messages have been
exchanged.
        if (tcpMessageConnect == null) {
            // Display corresponding error message in a message dialogue.
            JOptionPane.showMessageDialog(panel, "Cannot export results as no TCP message
connect has been performed.", "Null TCP Message Connect Error", JOptionPane.ERROR_MESSAGE);
            return;
        }

        // Guard against a blank file path, i.e. if the user has not selected a path.
        if (exportResultsPath == null) {
            // Display corresponding error message in a message dialogue.
            JOptionPane.showMessageDialog(panel, "Results not exported as no directory was
selected.", "File Path Error", JOptionPane.ERROR_MESSAGE);
            return;
        }

        try {
            // Write the data to the file.
            writeToTextFile(tcpMessageConnect.getTextPaneContents());
        } catch (BadLocationException e) {
            // Display an error message in a message dialogue if an error occurs.
            JOptionPane.showMessageDialog(panel, "Error occured during file writing
process.", "File Writing Error", JOptionPane.ERROR_MESSAGE);
        }

    }
}
```

ImportResults.java

```java
package com.pingpal.imports;

import com.pingpal.datavalidation.ValidationUtils;
import com.pingpal.deviceping.DevicePingResult;
import com.pingpal.exceptions.imports.InvalidNumOfPingsException;
import com.pingpal.exceptions.imports.InvalidPacketLossRangeException;
import com.pingpal.exceptions.imports.InvalidPingIntervalRangeException;
import com.pingpal.exceptions.imports.InvalidPortNumberRangeException;
import com.pingpal.exceptions.imports.InvalidPortProtocolRelationshipException;
import com.pingpal.exceptions.imports.InvalidRoundTripTimeException;
import com.pingpal.exceptions.imports.InvalidScanTypeException;
import com.pingpal.exceptions.imports.InvalidSuccessfulPingException;
import com.pingpal.exceptions.imports.InvalidTimeoutRangeException;
import com.pingpal.exceptions.imports.InvalidVariableInstanceException;
import com.pingpal.exceptions.imports.MissingRequiredKeysException;
import com.pingpal.exceptions.ui.BlankFieldException;
import com.pingpal.exceptions.ui.InvalidIPAddressException;
import com.pingpal.exceptions.ui.InvalidNetworkRangeException;
import com.pingpal.exceptions.ui.InvalidPortRangeException;
import com.pingpal.portscan.PortScanResult;
import com.pingpal.subnetscan.SubnetScanResult;
import java.awt.HeadlessException;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.util.ArrayList;
import javax.swing.JFileChooser;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.filechooser.FileNameExtensionFilter;
import org.json.JSONArray;
import org.json.JSONException;
import org.json.JSONObject;
import org.json.JSONTokener;

/**
 * The {@code ImportResults} class handles the import of JSON-formatted scan
 * results from a file.
 * <p>
 * This class is responsible for selecting a file (via a JFileChooser), reading
 * its JSON contents, validating the data for different types of scan results
 * (subnet scan, device ping, and port scan), and then passing the parsed data
 * to a registered {@code ImportResultsListener}.
 * </p>
 * <p>
 * If any errors occur (e.g. missing fields, invalid data types, or JSON parsing
 * errors), the class displays a relevant error message using a JOptionPane.
 * </p>
 */
public class ImportResults {

    // The JPanel that is used as the parent for dialogues.
    private JPanel panel;

    // A JFileChooser configured to select only JSON files.
    private JFileChooser fchFileChooser;

    // The file containing the imported JSON scan results.
    private File importResultsFile;

    // The JSONObject parsed from the selected file.
    private JSONObject fileData;

    // A listener which will be notified when scan results have been successfully imported.
    private ImportResultsListener listener;

    /**
     * Constructs a new ImportResults instance with the specified panel and
     * listener.
     *
     * @param panel the JPanel used for displaying file chooser dialogues and
     * error messages
     * @param listener the listener to receive imported scan data
     */
    public ImportResults(JPanel panel, ImportResultsListener listener) {
        this.panel = panel;
        this.listener = listener;
    }

    /**
     * Opens a file chooser dialogue for the user to select a JSON file.
     * <p>
     * The file chooser is configured to only accept files with a ".json"
     * extension.
     * </p>
     */
    public void setImportResultsPath() {
        // Initialise the file chooser.
        fchFileChooser = new JFileChooser();

        // Disallow the selection of all files.
        fchFileChooser.setAcceptAllFileFilterUsed(false);
        // Restrict the accepted file type to only JSON files.
        fchFileChooser.setFileFilter(new FileNameExtensionFilter("JSON FILES", "json",
"json"));

        // Set the title of the file chooser.
        fchFileChooser.setDialogTitle("Select a JSON file");

        // If a file is selected, initialise the results file variable.
        int returnVal = fchFileChooser.showOpenDialog(panel);
        if (returnVal == JFileChooser.APPROVE_OPTION) {
            importResultsFile = fchFileChooser.getSelectedFile();
        }
    }

    /**
     * Reads and parses JSON data from the selected file.
     *
     * @return a JSONObject representing the contents of the file
     * @throws FileNotFoundException if the file does not exist
     * @throws JSONException if an error occurs during JSON parsing
     */
    private JSONObject readFileData() throws FileNotFoundException, JSONException {
        return new JSONObject(new JSONTokener(new FileReader(importResultsFile)));
    }

    /**
     * Determines the type of scan results contained in the imported JSON file,
     * calls helper methods to validate the data, then calls the appropriate
     * import method.
     * <p>
     * The method checks whether the JSON file contains subnet scan, device
     * ping, or port scan results based on the presence of specific keys, and
     * then validates the data using dedicated validation methods. If a scan
     * type is unrecognized, an InvalidScanTypeException is thrown.
     * </p>
     *
     * @throws InvalidScanTypeException if the scan type in the JSON file is
     * unknown
     */
    public void determineScanType() throws InvalidScanTypeException {
        try {
            // Parse JSON data from the selected file.
            fileData = readFileData();

            // Determine if the data is the results of a subnet scan.
            if (fileData.has("subnetScanResults")) {
                // Validate the data in the file.
                if (!validateSubnetScanData()) {
                    return;
                }
                // If all checks are successful, import the results.
                importSubnetScanData();

                // Determine if the data is the result of a device ping.
            } else if (fileData.has("devicePingResults")) {
                // Validate the data in the file.
                if (!validateDevicePingData()) {
                    return;
                }
                // If all checks successful, import the results.
                importDevicePingData();

                // Determine if the data is the result of a port scan.
            } else if (fileData.has("portScanResults")) {
                // Validate the data in the file.
                if (!validatePortScanData()) {
                    return;
                }
                // If all checks successful, import the results.
                importPortScanData();

                // If the data is not the results of a PingPal scan, throw and
InvaldScanTypeExceptioin.
            } else {
                throw new InvalidScanTypeException("Unkown scan type in file.");
            }

            // Catch any exceptions that may occur during the process of reading the data
from the file.
            // Display approppriate error messages.
        } catch (NullPointerException e) {
            JOptionPane.showMessageDialog(panel, "Results not imported as no file was
selected.", "File Path Error", JOptionPane.ERROR_MESSAGE);

        } catch (HeadlessException e) {
            JOptionPane.showMessageDialog(panel, "Error occurred during file reading
process.", "File Reading Error", JOptionPane.ERROR_MESSAGE);

        } catch (FileNotFoundException e) {
            JOptionPane.showMessageDialog(panel, "This file does not exist.", "File Reading
Error", JOptionPane.ERROR_MESSAGE);

        } catch (JSONException e) {
            JOptionPane.showMessageDialog(panel, "JSON parsing error occurred during file
reading process.", "File Reading Error", JOptionPane.ERROR_MESSAGE);
        }
    }

    /**
     * Validates that the JSON data for a subnet scan contains the required
     * fields and that their types and values are correct.
     *
     * @return {@code true} if subnet scan data is valid; {@code false}
     * otherwise
     */
    private boolean validateSubnetScanData() {
        // Check for required top-level keys.
        try {
            ValidationUtils.validateRequiredKeys(new String[]{"networkRange", "timeout",
"subnetScanResults"}, fileData);
        } catch (MissingRequiredKeysException e) {
            JOptionPane.showMessageDialog(panel, "Missing one or more required top-level
fields.", "Missing Field Error", JOptionPane.ERROR_MESSAGE);
            return false;
        }

        // Validate networkRange is a string.
        try {
            ValidationUtils.validateInstanceString(fileData.get("networkRange"));
        } catch (InvalidVariableInstanceException e) {
            JOptionPane.showMessageDialog(panel, "Network range field is not a string.",
"Data Field Error", JOptionPane.ERROR_MESSAGE);
            return false;
        }

        // Validate networkRange presence.
        try {
            ValidationUtils.validateFieldPresence(fileData.getString("networkRange"));
        } catch (BlankFieldException e) {
            JOptionPane.showMessageDialog(panel, "Network range field is blank.", "Data Field
Error", JOptionPane.ERROR_MESSAGE);
            return false;
        }

        // Validate networkRange format.
        try {
            ValidationUtils.validateNetworkRange(fileData.getString("networkRange"));
        } catch (InvalidNetworkRangeException e) {
            JOptionPane.showMessageDialog(panel, "IP range field is not in correct format.",
"Data Field Error", JOptionPane.ERROR_MESSAGE);
            return false;
        }

        // Validate timeout is an integer.
        try {
            ValidationUtils.validateInstanceInteger(fileData.get("timeout"));
        } catch (InvalidVariableInstanceException e) {
            JOptionPane.showMessageDialog(panel, "Timeout field is not an integer.", "Data
Field Error", JOptionPane.ERROR_MESSAGE);
            return false;
        }
        // Validate timeout range.
        try {
            ValidationUtils.validateTimeoutRange(fileData.getInt("timeout"));
        } catch (InvalidTimeoutRangeException e) {
            JOptionPane.showMessageDialog(panel, "Timeout value falls out of acceptable
range.", "Data Field Error", JOptionPane.ERROR_MESSAGE);
            return false;
        }

        // Validate subnetScanResults is an array.
        try {
            ValidationUtils.validateInstanceJSONArray(fileData.get("subnetScanResults"));
        } catch (InvalidVariableInstanceException e) {
            JOptionPane.showMessageDialog(panel, "Subnet scan results field is not an
array.", "Data Field Error", JOptionPane.ERROR_MESSAGE);
            return false;
        }

        // Validate each result from subnetScanResults array.
        JSONArray jsonSubnetScanResults = fileData.getJSONArray("subnetScanResults");

        for (int i = 0; i < jsonSubnetScanResults.length(); i++) {
            JSONObject jsonSubnetScanResult = jsonSubnetScanResults.getJSONObject(i);

            // Check for required keys in each object.
            try {
                ValidationUtils.validateRequiredKeys(new String[]{"ipAddress"},
jsonSubnetScanResult);
            } catch (MissingRequiredKeysException e) {
                JOptionPane.showMessageDialog(panel, "Missing required field in subnet scan
results array at index " + i + ".", "Missing Field Error", JOptionPane.ERROR_MESSAGE);
                return false;
            }

            // Validate IP address is a string.
            try {

ValidationUtils.validateInstanceString(jsonSubnetScanResult.get("ipAddress"));
            } catch (InvalidVariableInstanceException e) {
                JOptionPane.showMessageDialog(panel, "IP address field is not a string at
index " + i + ".", "Data Field Error", JOptionPane.ERROR_MESSAGE);
                return false;
            }

            // Validate IP address presence.
            try {

ValidationUtils.validateFieldPresence(jsonSubnetScanResult.getString("ipAddress"));
            } catch (BlankFieldException e) {
                JOptionPane.showMessageDialog(panel, "IP is blank at index " + i + ".", "Data
Field Error", JOptionPane.ERROR_MESSAGE);
                return false;
            }

            // Validate IP address format.
            try {

ValidationUtils.validateIPAddress(jsonSubnetScanResult.getString("ipAddress"));
            } catch (InvalidIPAddressException e) {
                JOptionPane.showMessageDialog(panel, "IP address field is not in correct
format at index " + i + ".", "Data Field Error", JOptionPane.ERROR_MESSAGE);
                return false;
            }
        }

        // All checks successful.
        return true;
    }

    /**
     * Validates that the JSON data for a device ping scan contains required
     * fields and that their values are valid.
     *
     * @return {@code true} if the device ping data is valid; {@code false}
     * otherwise
     */
    public boolean validateDevicePingData() {
        // Check for required top-level keys.
        try {
            ValidationUtils.validateRequiredKeys(new String[]{"ipAddress", "pingInterval",
"numOfPings", "continuousPinging", "devicePingResults"}, fileData);
        } catch (MissingRequiredKeysException e) {
            JOptionPane.showMessageDialog(panel, "Missing one or more required top-level
fields.", "Missing Field Error", JOptionPane.ERROR_MESSAGE);
            return false;
        }

        // Validate ipAddress is a string.
        try {
            ValidationUtils.validateInstanceString(fileData.get("ipAddress"));
        } catch (InvalidVariableInstanceException e) {
            JOptionPane.showMessageDialog(panel, "IP address field is not a string.", "Data
Field Error", JOptionPane.ERROR_MESSAGE);
            return false;
        }

        // Validate ipAddress presence.
        try {
            ValidationUtils.validateFieldPresence(fileData.getString("ipAddress"));
        } catch (BlankFieldException e) {
            JOptionPane.showMessageDialog(panel, "IP address field is blank.", "Data Field
Error", JOptionPane.ERROR_MESSAGE);
            return false;
        }

        // Validate ipAddress format.
        try {
            ValidationUtils.validateIPAddress(fileData.getString("ipAddress"));
        } catch (InvalidIPAddressException e) {
            JOptionPane.showMessageDialog(panel, "IP address field is not in correct
format.", "Data Field Error", JOptionPane.ERROR_MESSAGE);
            return false;
        }

        // Validate pingInterval is an integer.
        try {
            ValidationUtils.validateInstanceInteger(fileData.get("pingInterval"));
        } catch (InvalidVariableInstanceException e) {
            JOptionPane.showMessageDialog(panel, "Ping interval field is not an integer.",
"Data Field Error", JOptionPane.ERROR_MESSAGE);
            return false;
        }

        // Validate pingInterval range.
        try {
            ValidationUtils.validatePingInterval(fileData.getInt("pingInterval"));
        } catch (InvalidPingIntervalRangeException e) {
            JOptionPane.showMessageDialog(panel, "Ping interval value falls out of acceptable
range.", "Data Field Error", JOptionPane.ERROR_MESSAGE);
            return false;
        }

        // Validate numOfPings is an integer.
        try {
            ValidationUtils.validateInstanceInteger(fileData.get("numOfPings"));
        } catch (InvalidVariableInstanceException e) {
            JOptionPane.showMessageDialog(panel, "Number of pings field is not an integer.",
"Data Field Error", JOptionPane.ERROR_MESSAGE);
            return false;
        }

        // Validate numOfPings range.
        try {
            ValidationUtils.validateNumOfPingsRange(fileData.getInt("numOfPings"));
        } catch (InvalidNumOfPingsException e) {
            JOptionPane.showMessageDialog(panel, "Number of pings value falls out of
acceptable range.", "Data Field Error", JOptionPane.ERROR_MESSAGE);
            return false;
        }

        // Validate continuousPinging is a boolean.
        try {
            ValidationUtils.validateInstanceBoolean(fileData.get("continuousPinging"));
        } catch (InvalidVariableInstanceException e) {
            JOptionPane.showMessageDialog(panel, "Continuous pinging field is not a
boolean.", "Data Field Error", JOptionPane.ERROR_MESSAGE);
            return false;
        }

        // Validate devicePingResults is an array.
        try {
            ValidationUtils.validateInstanceJSONArray(fileData.get("devicePingResults"));
        } catch (InvalidVariableInstanceException e) {
            JOptionPane.showMessageDialog(panel, "Device ping results field is not an
array.", "Data Field Error", JOptionPane.ERROR_MESSAGE);
            return false;
        }

        // Validate each result from devicePingResults array.
        JSONArray jsonDevicePingResults = fileData.getJSONArray("devicePingResults");

        for (int i = 0; i < jsonDevicePingResults.length(); i++) {
            JSONObject jsonDevicePingResult = jsonDevicePingResults.getJSONObject(i);

            // Check for required keys in each object.
            try {
                ValidationUtils.validateRequiredKeys(new String[]{"roundTripTime",
"successfulPing", "packetLoss"}, jsonDevicePingResult);
            } catch (MissingRequiredKeysException e) {
                JOptionPane.showMessageDialog(panel, "Missing required field(s) in device
ping results array at index " + i + ".", "Missing Field Error", JOptionPane.ERROR_MESSAGE);
                return false;
            }

            // Validate roundTripTime is an integer.
            try {
```

```
ValidationUtils.validateInstanceInteger(jsonDevicePingResult.get("roundTripTime"));
            } catch (InvalidVariableInstanceException e) {
                JOptionPane.showMessageDialog(panel, "Round trip time field is not an integer
at index " + i + ".", "Data Field Error", JOptionPane.ERROR_MESSAGE);
                return false;
            }

            // Validate roundTripTime range.
            try {

ValidationUtils.validateRoundTripTime(jsonDevicePingResult.getInt("roundTripTime"),
fileData.getInt("pingInterval"));
            } catch (InvalidRoundTripTimeException e) {
                JOptionPane.showMessageDialog(panel, "Round trip time value falls out of
acceptable range at index " + i + ".", "Data Field Error", JOptionPane.ERROR_MESSAGE);
                return false;
            }

            // Validate successfulPing is a boolean.
            try {

ValidationUtils.validateInstanceBoolean(jsonDevicePingResult.get("successfulPing"));
            } catch (InvalidVariableInstanceException e) {
                JOptionPane.showMessageDialog(panel, "Successful ping field is not boolean at
index " + i + ".", "Data Field Error", JOptionPane.ERROR_MESSAGE);
                return false;
            }

            // Validate successfulPing logic.
            try {

ValidationUtils.validateSuccessfulPingLogic(jsonDevicePingResult.getBoolean("successfulPing")
, jsonDevicePingResult.getInt("roundTripTime"), fileData.getInt("pingInterval"));
            } catch (InvalidSuccessfulPingException e) {
                JOptionPane.showMessageDialog(panel, "Round trip time and successful ping
results do not not match at index " + i + ".", "Data Field Error",
JOptionPane.ERROR_MESSAGE);
                return false;
            }

            // Validate packetLoss is a double.
            try {

ValidationUtils.validateInstanceNumber(jsonDevicePingResult.get("packetLoss"));
            } catch (InvalidVariableInstanceException e) {
                JOptionPane.showMessageDialog(panel, "Packet loss field is not a double at
index " + i + ".", "Data Field Error", JOptionPane.ERROR_MESSAGE);
                return false;
            }

            // Validate packetLoss range.
            try {

ValidationUtils.validatePacketLossRange(jsonDevicePingResult.getDouble("packetLoss"));
            } catch (InvalidPacketLossRangeException e) {
                JOptionPane.showMessageDialog(panel, "Packet loss value falls out of
acceptable range at index " + i + ".", "Data Field Error", JOptionPane.ERROR_MESSAGE);
                return false;
            }
        }
        // All checks successful.
        return true;
    }

    /**
     * Validates that the JSON data for a port scan contains the required fields
     * and that each field is valid.
     *
     * @return {@code true} if the port scan data is valid; {@code false}
     * otherwise
     */
    public boolean validatePortScanData() {
        // Check for required top-level keys.
        try {
            ValidationUtils.validateRequiredKeys(new String[]{"ipAddress", "bottomRangePort",
"topRangePort", "timeout", "portScanResults"}, fileData);
        } catch (MissingRequiredKeysException e) {
            JOptionPane.showMessageDialog(panel, "Missing one or more required top-level
fields.", "Missing Field Error", JOptionPane.ERROR_MESSAGE);
            return false;
        }

        // Validate ipAddress is a string.
        try {
            ValidationUtils.validateInstanceString(fileData.get("ipAddress"));
        } catch (InvalidVariableInstanceException e) {
            JOptionPane.showMessageDialog(panel, "IP address field is not a string.", "Data
Field Error", JOptionPane.ERROR_MESSAGE);
            return false;
        }

        // Validate ipAddress presence.
        try {
            ValidationUtils.validateFieldPresence(fileData.getString("ipAddress"));
        } catch (BlankFieldException e) {
            JOptionPane.showMessageDialog(panel, "IP address field is blank.", "Data Field
Error", JOptionPane.ERROR_MESSAGE);
            return false;
        }

        // Validate ipAddress format.
        try {
            ValidationUtils.validateIPAddress(fileData.getString("ipAddress"));
        } catch (InvalidIPAddressException e) {
            JOptionPane.showMessageDialog(panel, "IP address field is not in correct
format.", "Data Field Error", JOptionPane.ERROR_MESSAGE);
            return false;
        }

        // Validate bottomRangePort is an integer.
        try {
            ValidationUtils.validateInstanceInteger(fileData.get("bottomRangePort"));
        } catch (InvalidVariableInstanceException e) {
            JOptionPane.showMessageDialog(panel, "Bottom range port field is not an
integer.", "Data Field Error", JOptionPane.ERROR_MESSAGE);
            return false;
        }

        // Validate bottomRangePort range.
        try {
            ValidationUtils.validatePortNumberRange(fileData.getInt("bottomRangePort"));
        } catch (InvalidPortNumberRangeException e) {
            JOptionPane.showMessageDialog(panel, "Bottom port value falls out of acceptable
range.", "Data Field Error", JOptionPane.ERROR_MESSAGE);
            return false;
        }

        // Validate topRangePort is an integer.
        try {
            ValidationUtils.validateInstanceInteger(fileData.get("topRangePort"));
        } catch (InvalidVariableInstanceException e) {
            JOptionPane.showMessageDialog(panel, "Top range port field is not an integer.",
"Data Field Error", JOptionPane.ERROR_MESSAGE);
            return false;
        }

        // Validate topRangePort range.
        try {
            ValidationUtils.validatePortNumberRange(fileData.getInt("topRangePort"));
        } catch (InvalidPortNumberRangeException e) {
            JOptionPane.showMessageDialog(panel, "Top port value falls out of acceptable
range.", "Data Field Error", JOptionPane.ERROR_MESSAGE);
            return false;
        }

        // Validate port range logic.
        try {
            ValidationUtils.validatePortRange(fileData.getInt("bottomRangePort"),
fileData.getInt("topRangePort"));
        } catch (InvalidPortRangeException e) {
            JOptionPane.showMessageDialog(panel, "Bottom port value is greater than top port
value.", "Data Field Error", JOptionPane.ERROR_MESSAGE);
            return false;
        }

        // Validate timeout is an integer.
        try {
            ValidationUtils.validateInstanceInteger(fileData.get("timeout"));
        } catch (InvalidVariableInstanceException e) {
            JOptionPane.showMessageDialog(panel, "Timeout field is not an integer.", "Data
Field Error", JOptionPane.ERROR_MESSAGE);
            return false;
        }

        // Validate timeout range.
        try {
            ValidationUtils.validateTimeoutRange(fileData.getInt("timeout"));
        } catch (InvalidTimeoutRangeException e) {
            JOptionPane.showMessageDialog(panel, "Timeout value falls out of acceptable
range.", "Data Field Error", JOptionPane.ERROR_MESSAGE);
            return false;
        }

        // Validate portScanResults is an array.
        try {
            ValidationUtils.validateInstanceJSONArray(fileData.get("portScanResults"));
        } catch (InvalidVariableInstanceException e) {
            JOptionPane.showMessageDialog(panel, "Port scan results field is not an array.",
"Data Field Error", JOptionPane.ERROR_MESSAGE);
            return false;
        }

        // Validate each result from portScanResults array.
        JSONArray jsonPortScanResults = fileData.getJSONArray("portScanResults");
        for (int i = 0; i < jsonPortScanResults.length(); i++) {
            JSONObject jsonPortScanResult = jsonPortScanResults.getJSONObject(i);

            // Check for required keys in each object.
            try {
                ValidationUtils.validateRequiredKeys(new String[]{"portNumber", "protocol"},
jsonPortScanResult);
            } catch (MissingRequiredKeysException e) {
                JOptionPane.showMessageDialog(panel, "Missing required field(s) in port scan
results array at index " + i + ".", "Missing Field Error", JOptionPane.ERROR_MESSAGE);
                return false;
            }

            // Validate portNumber is an integer.
            try {

ValidationUtils.validateInstanceInteger(jsonPortScanResult.get("portNumber"));
            } catch (InvalidVariableInstanceException e) {
                JOptionPane.showMessageDialog(panel, "Port number field is not an integer at
index " + i + ".", "Data Field Error", JOptionPane.ERROR_MESSAGE);
                return false;
            }

            // Validate portNumber range.
            try {

ValidationUtils.validatePortNumberRange(jsonPortScanResult.getInt("portNumber"));
            } catch (InvalidPortNumberRangeException e) {
                JOptionPane.showMessageDialog(panel, "Port number value falls out of
acceptable range at index " + i + ".", "Data Field Error", JOptionPane.ERROR_MESSAGE);
                return false;
            }

            // Validate protocol is a string.
            try {
                ValidationUtils.validateInstanceString(jsonPortScanResult.get("protocol"));
            } catch (InvalidVariableInstanceException e) {
                JOptionPane.showMessageDialog(panel, "Protocol field is not a string at index
" + i + ".", "Data Field Error", JOptionPane.ERROR_MESSAGE);
                return false;
            }

            // Validate protocol presence.
            try {

ValidationUtils.validateFieldPresence(jsonPortScanResult.getString("protocol"));
            } catch (BlankFieldException e) {
                JOptionPane.showMessageDialog(panel, "Protocol field is blank at index " + i
+ ".", "Data Field Error", JOptionPane.ERROR_MESSAGE);
                return false;
            }

            // Validate protocol corresponds to correct port number.
            try {

ValidationUtils.validatePortCorrespondsToProtocol(jsonPortScanResult.getInt("portNumber"),
jsonPortScanResult.getString("protocol"));
            } catch (InvalidPortProtocolRelationshipException e) {
                JOptionPane.showMessageDialog(panel, "Protocol does not correspond to the
port number at index " + i + ".", "Data Field Error", JOptionPane.ERROR_MESSAGE);
                return false;
            }
        }

        // All checks successful.
        return true;
    }

    /**
     * Parses subnet scan results from the JSON array and converts them into a
     * list of {@code SubnetScanResult} objects.
     *
     * @return an {@code ArrayList} of {@code SubnetScanResult} objects parsed
     * from the JSON data
     */
    private ArrayList<SubnetScanResult> parseSubnetScanResultsArray() {
        // Create the ArrayList of SubnetScanResult objects.
        ArrayList<SubnetScanResult> subnetScanResults = new ArrayList<>();

        // Parse the data from the file to a JSONArray.
        JSONArray jsonSubnetScanResults = fileData.getJSONArray("subnetScanResults");

        // Loop through each JSONObject in the JSONArray.
        for (int i = 0; i < jsonSubnetScanResults.length(); i++) {
            JSONObject jsonSubnetScanResult = jsonSubnetScanResults.getJSONObject(i);

            // Parse the individual data values from the JSONObject, and use them to create a
new SubnetScanResult object.
            SubnetScanResult subnetScanResult = new SubnetScanResult(
                    jsonSubnetScanResult.getString("ipAddress")
            );
            // Append the SubnetScanResult to the list.
            subnetScanResults.add(subnetScanResult);
        }

        return subnetScanResults;
    }

    /**
     * Imports subnet scan results by extracting the network range, timeout, and
     * a list of subnet scan results from the JSON data, then passes the data to
     * the listener.
     */
    private void importSubnetScanData() {
        // Create the temporary holder variables.
        String networkRange = fileData.getString("networkRange");
        int timeout = fileData.getInt("timeout");
        ArrayList<SubnetScanResult> subnetScanResults = parseSubnetScanResultsArray();

        // Call the listener to indicate that a subnet scan has been imported.
        listener.onSubnetScanResultsImported(networkRange, timeout, subnetScanResults);
    }

    /**
     * Parses device ping results from the JSON array and converts them into a
     * list of {@code DevicePingResult} objects.
     *
     * @return an {@code ArrayList} of {@code DevicePingResult} objects parsed
     * from the JSON data
     */
    private ArrayList<DevicePingResult> parseDevicePingResults() {
        // Create the ArrayList of DevicePingResult objects.
        ArrayList<DevicePingResult> devicePingResults = new ArrayList<>();
        // Parse the data from the file to a JSONArray.
        JSONArray jsonDevicePingResults = fileData.getJSONArray("devicePingResults");

        // Loop through each JSONObject in the JSONArray.
        for (int i = 0; i < jsonDevicePingResults.length(); i++) {
            JSONObject jsonDevicePingResult = jsonDevicePingResults.getJSONObject(i);

            // Parse the individual data values from the JSONObject, and use them to create a
new DevicePingResult object.
            DevicePingResult devicePingResult = new DevicePingResult(
                    jsonDevicePingResult.getInt("roundTripTime"),
                    jsonDevicePingResult.getBoolean("successfulPing"),
                    jsonDevicePingResult.getDouble("packetLoss")
            );
            // Append the SubnetScanResult to the list.
            devicePingResults.add(devicePingResult);
        }

        return devicePingResults;
    }

    /**
     * Imports device ping scan results by extracting the target IP address,
     * ping interval, number of pings, continuous pinging flag, and the list of
     * device ping results from the JSON data, then passes the data to the
     * listener.
     */
    private void importDevicePingData() {
        // Create the temporary holder variables.
        String ipAddress = fileData.getString("ipAddress");
        int pingInterval = fileData.getInt("pingInterval");
        int numOfPings = fileData.getInt("numOfPings");
        boolean continuousPinging = fileData.getBoolean("continuousPinging");
        ArrayList<DevicePingResult> devicePingResults = parseDevicePingResults();

        // Call the listener to indicate that a device ping has been imported.
        listener.onDevicePingResultsImported(ipAddress, pingInterval, numOfPings,
continuousPinging, devicePingResults);
    }

    /**
     * Parses port scan results from the JSON array and converts them into a
     * list of {@code PortScanResult} objects.
     *
     * @return an {@code ArrayList} of {@code PortScanResult} objects parsed
     * from the JSON data
     */
    private ArrayList<PortScanResult> parsePortScanResults() {
        // Create the ArrayList of PortScanResult objects.
        ArrayList<PortScanResult> portScanResults = new ArrayList<>();

        // Parse the data from the file to a JSONArray.
        JSONArray jsonPortScanResults = fileData.getJSONArray("portScanResults");

        // Loop through each JSONObject in the JSONArray.
        for (int i = 0; i < jsonPortScanResults.length(); i++) {
            JSONObject jsonPortScanResult = jsonPortScanResults.getJSONObject(i);

            // Parse the individual data values from the JSONObject, and use them to create a
new PortScanResult object.
            PortScanResult portScanResult = new PortScanResult(
```

```java
                    jsonPortScanResult.getInt("portNumber"),
                    jsonPortScanResult.getString("protocol")
            );
            // Append the SubnetScanResult to the list.
            portScanResults.add(portScanResult);
        }

        return portScanResults;
    }

    /**
     * Imports port scan scan results by extracting the target IP address, port
     * range, timeout, and the list of port scan results from the JSON data,
     * then passes the data to the listener.
     */
    private void importPortScanData() {
        // Create the temporary holder variables.
        String ipAddress = fileData.getString("ipAddress");
        int bottomRangePort = fileData.getInt("bottomRangePort");
        int topRangePort = fileData.getInt("topRangePort");
        int timeout = fileData.getInt("timeout");
        ArrayList<PortScanResult> portScanResults = parsePortScanResults();

        // Call the listener to indicate that a port scan has been imported.
        listener.onPortScanResultsImported(ipAddress, bottomRangePort, topRangePort, timeout,
portScanResults);
    }
}
```

ImportResultsListener.java

```java
package com.pingpal.imports;

import com.pingpal.deviceping.DevicePingResult;
import com.pingpal.portscan.PortScanResult;
import com.pingpal.subnetscan.SubnetScanResult;
import java.util.ArrayList;

/**
 * The {@code ImportResultsListener} interface defines callback methods for
 * handling the imported scan results from various types of network scans.
 * <p>
 * Implementers of this interface will receive notifications when scan results
 * have been successfully imported from a JSON file. There are separate
 * callbacks for subnet scans, device pings, and port scans, providing all
 * necessary details for further processing or updating the user interface.
 * </p>
 */
public interface ImportResultsListener {

    /**
     * Called when subnet scan results have been successfully imported.
     *
     * @param networkRange the network range that was scanned (e.g.,
     * "192.168.0.0/24")
     * @param timeout the timeout (in milliseconds) used during the scan
     * @param subnetScanResults a list of {@code SubnetScanResult} objects
     * representing the reachable IP addresses found in the scan
     */
    void onSubnetScanResultsImported(String networkRange, int timeout,
ArrayList<SubnetScanResult> subnetScanResults);

    /**
     * Called when device ping results have been successfully imported.
     *
     * @param ipAddress the IP address that was pinged
     * @param pingInterval the interval (in milliseconds) between successive
     * pings
     * @param numOfPings the total number of pings performed
     * @param continuousPinging a boolean indicating if the pinging was
     * continuous (true) or a fixed number of pings (false)
     * @param devicePingResults a list of {@code DevicePingResult} objects
     * containing the results of the ping operations (e.g., round-trip times,
     * packet loss)
     */
    void onDevicePingResultsImported(String ipAddress, int pingInterval, int numOfPings,
boolean continuousPinging, ArrayList<DevicePingResult> devicePingResults);

    /**
     * Called when port scan results have been successfully imported.
     *
     * @param ipAddress the IP address that was scanned
     * @param bottomRangePort the starting port number of the scan range
     * @param topRangePort the ending port number of the scan range
     * @param timeout the timeout (in milliseconds) used during the scan
     * @param portScanResults a list of {@code PortScanResult} objects
     * representing the results of the port scan (i.e., open ports and
     * associated protocols)
     */
    void onPortScanResultsImported(String ipAddress, int bottomRangePort, int topRangePort,
int timeout, ArrayList<PortScanResult> portScanResults);
}
```

ValidationUtils.java

```java
package com.pingpal.datavalidation;

import com.pingpal.exceptions.imports.InvalidNumOfPingsException;
import com.pingpal.exceptions.imports.InvalidPacketLossRangeException;
import com.pingpal.exceptions.imports.InvalidPingIntervalRangeException;
import com.pingpal.exceptions.imports.InvalidPortNumberRangeException;
import com.pingpal.exceptions.imports.InvalidPortProtocolRelationshipException;
import com.pingpal.exceptions.imports.InvalidRoundTripTimeException;
import com.pingpal.exceptions.imports.InvalidSuccessfulPingException;
import com.pingpal.exceptions.imports.InvalidTimeoutRangeException;
import com.pingpal.exceptions.imports.InvalidVariableInstanceException;
import com.pingpal.exceptions.imports.MissingRequiredKeysException;
import com.pingpal.exceptions.ui.BlankFieldException;
import com.pingpal.exceptions.ui.InvalidIPAddressException;
import com.pingpal.exceptions.ui.InvalidNetworkRangeException;
import com.pingpal.exceptions.ui.InvalidPortRangeException;
import com.pingpal.portscan.Protocols;
import java.awt.Color;
import org.json.JSONArray;
import org.json.JSONObject;

/**
 * The {@code ValidationUtils} class provides a set of static methods to perform
 * data validation for network scanning operations. It includes methods for
 * checking field presence, format and type validation of network ranges, IP
 * addresses, port numbers, ping intervals, timeouts, and relationships between
 * ports and protocols.
 * <p>
 * In addition, this class holds several constants such as color codes and
 * regular expression patterns, as well as minimum/maximum acceptable values for
 * various parameters.
 * </p>
 */
public class ValidationUtils {

    // Color constants used for UI validation feedback.
    public final static Color ERROR_COLOR = new Color(250, 200, 200);
    public final static Color GRAYED_OUT_COLOR = new Color(184, 184, 184);
    public final static Color NORMAL_TEXT_COLOR = new Color(233, 247, 249);
    public final static Color SUCCESSFUL_SCAN_COLOR = new Color(0, 204, 0);
    public final static Color INTERRUPTED_SCAN_COLOR = new Color(255, 51, 0);

    // Regular expression patterns.
    public final static String NETWORK_RANGE_PATTERN =
"^(?:(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9][0-9]|[0-9])\\.){3}(?:25[0-5]|2[0-4][0-9]|1[0-9]
[0-9]|[1-9][0-9]|[0-9])\\/(?:[1-9]|[12]\\d|3[0-2])$";
    public final static String IP_ADDRESS_PATTERN =
"^(?:(?:25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9][0-9]|[0-9])\\.){3}(?:25[0-5]|2[0-4][0-9]|1[0-9]
[0-9]|[1-9][0-9]|[0-9])$";

    // Range constants for validations.
    public final static int MIN_TIMEOUT = 100;
    public final static int MAX_TIMEOUT = 10_000;

    public final static int MIN_PING_INTERVAL = 100;
    public final static int MAX_PING_INTERVAL = 10_000;

    public final static int MIN_PINGS = 1;
    public final static int MAX_PINGS = 100;

    public final static int MIN_PORT = 1;
    public final static int MAX_PORT = 65_535;

    // Private constructor to prevent instantiation.
    private ValidationUtils() {

    }

    /**
     * Validates that the provided object is present (i.e. not null), and if it
     * is a {@code String}, is not blank.
     *
     * @param obj the object to validate for presence
     * @throws BlankFieldException if the object is null, or if a
     * {@code String}, is blank
     */
    public static void validateFieldPresence(Object obj) throws BlankFieldException {
        if (obj == null) {
            throw new BlankFieldException("Field is blank.");
        }

        if (obj instanceof String && ((String) obj).isBlank()) {
            throw new BlankFieldException("Field is blank.");
        }
    }

    /**
     * Validates that the provided network range string matches the required
     * format.
     *
     * @param networkRange the network range string to validate (e.g.,
     * "192.168.0.0/24")
     * @throws InvalidNetworkRangeException if the network range does not match
     * the required pattern
     */
    public static void validateNetworkRange(String networkRange) throws
InvalidNetworkRangeException {
        if (!networkRange.matches(NETWORK_RANGE_PATTERN)) {
            throw new InvalidNetworkRangeException("Invalid network range format.");

        }
    }

    /**
     * Validates that the provided IP address string matches the required
     * format.
     *
     * @param ipAddress the IP address string to validate (e.g., "192.168.0.1")
     * @throws InvalidIPAddressException if the IP address does not match the
     * required pattern
     */
    public static void validateIPAddress(String ipAddress) throws InvalidIPAddressException {
        if (!ipAddress.matches(IP_ADDRESS_PATTERN)) {
            throw new InvalidIPAddressException("Invalid IP address format.");

        }
    }

    /**
     * Validates that the bottom range port is not greater than the top range
     * port.
     *
     * @param bottomRange the starting port number of the range
     * @param topRange the ending port number of the range
     * @throws InvalidPortRangeException if the bottom range port is greater
     * than the top range port
     */
    public static void validatePortRange(int bottomRange, int topRange) throws
InvalidPortRangeException {
        if (bottomRange > topRange) {
            throw new InvalidPortRangeException("Invalid port range.");

        }
    }

    /**
     * Validates that the provided JSON object contains all required top-level
     * keys.
     *
     * @param topLevelKeys an array of keys that must be present in the JSON
     * object
     * @param fileData the JSON object to validate
     * @throws MissingRequiredKeysException if any required key is missing from
     * the JSON object
     */
    public static void validateRequiredKeys(String[] topLevelKeys, JSONObject fileData)
throws MissingRequiredKeysException {
        for (String topLevelKey : topLevelKeys) {
            if (!fileData.has(topLevelKey)) {
                throw new MissingRequiredKeysException("Missing one or more required keys");
            }
        }
    }

    /**
     * Validates that the provided object is an instance of {@code String}.
     *
     * @param obj the object to validate
     * @throws InvalidVariableInstanceException if the object is not a String
     */
    public static void validateInstanceString(Object obj) throws
InvalidVariableInstanceException {
        if (!(obj instanceof String)) {
            throw new InvalidVariableInstanceException("Field is not a string.");
        }
    }

    /**
     * Validates that the provided object is an instance of {@code Integer}.
     *
     * @param obj the object to validate
     * @throws InvalidVariableInstanceException if the object is not an Integer
     */
    public static void validateInstanceInteger(Object obj) throws
InvalidVariableInstanceException {
        if (!(obj instanceof Integer)) {
            throw new InvalidVariableInstanceException("Field is not an integer.");
        }
    }

    /**
     * Validates that the provided object is an instance of {@code Number}.
     *
     * @param obj the object to validate
     * @throws InvalidVariableInstanceException if the object is not a Number
     */
    public static void validateInstanceNumber(Object obj) throws
InvalidVariableInstanceException {
        if (!(obj instanceof Number)) {
            throw new InvalidVariableInstanceException("Field is not a number.");
        }
    }

    /**
     * Validates that the provided object is an instance of {@code Boolean}.
     *
     * @param obj the object to validate
     * @throws InvalidVariableInstanceException if the object is not a Boolean
     */
    public static void validateInstanceBoolean(Object obj) throws
InvalidVariableInstanceException {
        if (!(obj instanceof Boolean)) {
            throw new InvalidVariableInstanceException("Field is not a boolean.");
        }
    }

    /**
     * Validates that the provided object is an instance of {@code JSONArray}.
     *
     * @param obj the object to validate
     * @throws InvalidVariableInstanceException if the object is not a JSONArray
     */
    public static void validateInstanceJSONArray(Object obj) throws
InvalidVariableInstanceException {
        if (!(obj instanceof JSONArray)) {
            throw new InvalidVariableInstanceException("Field is not a JSON array.");
        }
    }

    /**
     * Validates that the timeout value falls within the acceptable range.
     *
     * @param timeout the timeout value in milliseconds to validate
     * @throws InvalidTimeoutRangeException if the timeout is less than
     * {@code MIN_TIMEOUT} or greater than {@code MAX_TIMEOUT}
     */
    public static void validateTimeoutRange(int timeout) throws InvalidTimeoutRangeException
{
        if (timeout < MIN_TIMEOUT || timeout > MAX_TIMEOUT) {
            throw new InvalidTimeoutRangeException("Invalid timeout range.");
        }
    }

    /**
     * Validates that the ping interval falls within the acceptable range.
     *
     * @param pingInterval the ping interval in milliseconds to validate
     * @throws InvalidPingIntervalRangeException if the ping interval is less
     * than {@code MIN_PING_INTERVAL} or greater than {@code MAX_PING_INTERVAL}
     */
    public static void validatePingInterval(int pingInterval) throws
InvalidPingIntervalRangeException {
        if (pingInterval < MIN_PING_INTERVAL || pingInterval > MAX_PING_INTERVAL) {
            throw new InvalidPingIntervalRangeException("Invalid ping interval range.");
        }
    }

    /**
     * Validates that the number of pings falls within the acceptable range.
     *
     * @param numOfPings the number of pings to validate
     * @throws InvalidNumOfPingsException if the number of pings is less than
     * {@code MIN_PINGS} or greater than {@code MAX_PINGS}
     */
    public static void validateNumOfPingsRange(int numOfPings) throws
InvalidNumOfPingsException {
        if (numOfPings < MIN_PINGS || numOfPings > MAX_PINGS) {
            throw new InvalidNumOfPingsException("Invalid number of pings.");
        }
    }

    /**
     * Validates that the round-trip time is within acceptable bounds.
     *
     * @param roundTripTime the measured round-trip time in milliseconds
     * @param pingInterval the ping interval used during measurement
     * @throws InvalidRoundTripTimeException if the round-trip time is negative
     * or exceeds the ping interval
     */
    public static void validateRoundTripTime(int roundTripTime, int pingInterval) throws
InvalidRoundTripTimeException {
        if (roundTripTime < 0 || roundTripTime > pingInterval) {
            throw new InvalidRoundTripTimeException("Invalid round trip time.");
        }
    }

    /**
     * Validates the logic of a successful ping.
     * <p>
     * This method ensures that if the round-trip time is less than the ping
     * interval, then the ping must be marked as successful.
     * </p>
     *
     * @param successfulPing a boolean indicating if the ping was successful
     * @param roundTripTime the round-trip time measured in milliseconds
     * @param pingInterval the ping interval in milliseconds
     * @throws InvalidSuccessfulPingException if the logic is inconsistent
     */
    public static void validateSuccessfulPingLogic(boolean successfulPing, int roundTripTime,
int pingInterval) throws InvalidSuccessfulPingException {
        if (roundTripTime < pingInterval && successfulPing == false) {
            throw new InvalidSuccessfulPingException("Invalid successful ping logic.");
        }
    }

    /**
     * Validates that the packet loss percentage is within the acceptable range.
     *
     * @param packetLoss the packet loss percentage to validate
     * @throws InvalidPacketLossRangeException if the packet loss is less than 0
     * or greater than 100
     */
    public static void validatePacketLossRange(double packetLoss) throws
InvalidPacketLossRangeException {
        if (packetLoss < 0 || packetLoss > 100) {
            throw new InvalidPacketLossRangeException("Invalid packet loss range.");
        }
    }

    /**
     * Validates that the given port number falls within the acceptable range.
     *
     * @param portNumber the port number to validate
     * @throws InvalidPortNumberRangeException if the port number is less than
     * {@code MIN_PORT} or greater than {@code MAX_PORT}
     */
    public static void validatePortNumberRange(int portNumber) throws
InvalidPortNumberRangeException {
        if (portNumber < MIN_PORT || portNumber > MAX_PORT) {
            throw new InvalidPortNumberRangeException("Invalid port number range.");
        }
    }

    /**
     * Validates that the specified protocol corresponds to the given port
     * number.
     * <p>
     * This method uses the {@code Protocols} instance to retrieve the expected
     * protocol for the port number. If the retrieved protocol does not match
     * the provided protocol, an exception is thrown.
     * </p>
     *
     * @param portNumber the port number for which to validate the protocol
     * @param protocol the protocol provided to validate
     * @throws InvalidPortProtocolRelationshipException if the protocol does not
     * match the expected value
     */
    public static void validatePortCorrespondsToProtocol(int portNumber, String protocol)
throws InvalidPortProtocolRelationshipException {
        if (!Protocols.getProtocolForPort(portNumber).equals(protocol)) {
            throw new InvalidPortProtocolRelationshipException("Port does not correspond to
the protocol.");
        }
    }
}
```

InvalidNumOfPingsException.java

```java
package com.pingpal.exceptions.imports;

/**
 * Thrown to indicate that the number of pings provided is outside of the
 * acceptable range.
 * <p>
 * This exception is typically raised when a value for the number of pings is
 * less than the minimum allowed or greater than the maximum allowed value.
 * </p>
 */
public class InvalidNumOfPingsException extends Exception {

    /**
     * Constructs a new InvalidNumOfPingsException with the specified detail
     * message.
     *
     * @param message the detail message explaining the reason for the exception
     */
    public InvalidNumOfPingsException(String message) {
        super(message);
    }

    /**
     * Constructs a new InvalidNumOfPingsException with the specified detail
     * message and cause.
     *
     * @param message the detail message explaining the reason for the exception
     * @param cause the cause (which is saved for later retrieval by the
     * {@link #getCause()} method)
     */
    public InvalidNumOfPingsException(String message, Throwable cause) {
        super(message, cause);
    }

}
```

InvalidPacketLossRangeException.java

```java
package com.pingpal.exceptions.imports;

/**
 * Thrown to indicate that the packet loss percentage provided is outside of the
 * acceptable range.
 * <p>
 * This exception is typically raised when a value for the packet loss
 * percentage is less than 0% or greater than 100%.
 * </p>
 */
public class InvalidPacketLossRangeException extends Exception {

    /**
     * Constructs a new InvalidPacketLossRangeException with the specified
     * detail message.
     *
     * @param message the detail message explaining the reason for the exception
     */
    public InvalidPacketLossRangeException(String message) {
        super(message);
    }

    /**
     * Constructs a new InvalidPacketLossRangeException with the specified
     * detail message and cause.
     *
     * @param message the detail message explaining the reason for the exception
     * @param cause the cause (which is saved for later retrieval by the
     * {@link #getCause()} method)
     */
    public InvalidPacketLossRangeException(String message, Throwable cause) {
        super(message, cause);
    }

}
```

InvalidPingIntervalRangeException.java

```java
package com.pingpal.exceptions.imports;

/**
 * Thrown to indicate that the ping interval provided is outside of the
 * acceptable range.
 * <p>
 * This exception is typically raised when a value for the ping interval
 * is less than 100 or greater than 10000.
 * </p>
 */
public class InvalidPingIntervalRangeException extends Exception {

    /**
     * Constructs a new InvalidPingIntervalRangeException with the specified
     * detail message.
     *
     * @param message the detail message explaining the reason for the exception
     */
    public InvalidPingIntervalRangeException(String message) {
        super(message);
    }

    /**
     * Constructs a new InvalidPingIntervalRangeException with the specified
     * detail message and cause.
     *
     * @param message the detail message explaining the reason for the exception
     * @param cause the cause (which is saved for later retrieval by the
     * {@link #getCause()} method)
     */
    public InvalidPingIntervalRangeException(String message, Throwable cause) {
        super(message, cause);
    }

}
```

InvalidPortNumberRangeException.java

```java
package com.pingpal.exceptions.imports;

/**
 * Thrown to indicate that the port number provided is outside of the
 * acceptable range.
 * <p>
 * This exception is typically raised when a value for the packet loss
 * percentage is less than 1 or greater than 65535.
 * </p>
 */
public class InvalidPortNumberRangeException extends Exception {

    /**
     * Constructs a new InvalidPortNumberRangeException with the specified
     * detail message.
     *
     * @param message the detail message explaining the reason for the exception
     */
    public InvalidPortNumberRangeException(String message) {
        super(message);
    }

    /**
     * Constructs a new InvalidPortNumberRangeException with the specified
     * detail message and cause.
     *
     * @param message the detail message explaining the reason for the exception
     * @param cause the cause (which is saved for later retrieval by the
     * {@link #getCause()} method)
     */
    public InvalidPortNumberRangeException(String message, Throwable cause) {
        super(message, cause);
    }

}
```

InvalidPortProtocolRelationshipException.java

```java
package com.pingpal.exceptions.imports;

/**
 * Thrown to indicate that the protocol provided does not match the
 * corresponding protocol to the port number provided.
 * <p>
 * This exception is typically raised when the protocol does not match the port
 * number provided in the
 * {@code .\src\com\pingpal\resources\databases\port_list.csv} file.
 * </p>
 */
public class InvalidPortProtocolRelationshipException extends Exception {

    /**
     * Constructs a new InvalidPortProtocolRelationshipException with the
     * specified detail message.
     *
     * @param message the detail message explaining the reason for the exception
     */
    public InvalidPortProtocolRelationshipException(String message) {
        super(message);
    }

    /**
     * Constructs a new InvalidPortProtocolRelationshipException with the
     * specified detail message and cause.
     *
     * @param message the detail message explaining the reason for the exception
     * @param cause the cause (which is saved for later retrieval by the
     * {@link #getCause()} method)
     */
    public InvalidPortProtocolRelationshipException(String message, Throwable cause) {
        super(message, cause);
    }

}
```

InvalidRoundTripTimeException.java

```java
package com.pingpal.exceptions.imports;

/**
 * Thrown to indicate that the round trip time provided is outside of the
 * acceptable range.
 * <p>
 * This exception is typically raised when a value for the round trip time is
 * less than 0 or greater than the provided ping interval.
 * </p>
 */
public class InvalidRoundTripTimeException extends Exception {

    /**
     * Constructs a new InvalidRoundTripTimeException with the specified detail
     * message.
     *
     * @param message the detail message explaining the reason for the exception
     */
    public InvalidRoundTripTimeException(String message) {
        super(message);
    }

    /**
     * Constructs a new InvalidRoundTripTimeException with the specified detail
     * message and cause.
     *
     * @param message the detail message explaining the reason for the exception
     * @param cause the cause (which is saved for later retrieval by the
     * {@link #getCause()} method)
     */
    public InvalidRoundTripTimeException(String message, Throwable cause) {
        super(message, cause);
    }

}
```

InvalidScanTypeException.java

```java
package com.pingpal.exceptions.imports;

/**
 * Thrown to indicate that the data from the imported file does not contain the
 * results of a scan provided by the functionality of PingPal.
 * <p>
 * This exception is typically raised when the data in the file does not match
 * the format of the results from either a {@code Subnet Scan},
 * {@code Device Ping}, or {@code Port Scan}.
 * </p>
 */
public class InvalidScanTypeException extends Exception {

    /**
     * Constructs a new InvalidScanTypeException with the specified detail
     * message.
     *
     * @param message the detail message explaining the reason for the exception
     */
    public InvalidScanTypeException(String message) {
        super(message);
    }

    /**
     * Constructs a new InvalidScanTypeException with the specified detail
     * message and cause.
     *
     * @param message the detail message explaining the reason for the exception
     * @param cause the cause (which is saved for later retrieval by the
     * {@link #getCause()} method)
     */
    public InvalidScanTypeException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

InvalidSuccessfulPingException.java

```java
package com.pingpal.exceptions.imports;

/**
 * Thrown to signal that the logical relationship between the round-trip time
 * and the successful-ping flag is invalid.
 * <p>
 * For example, if a ping reports a round-trip time less than the ping interval
 * but marks successfulPing as false, this exception should be thrown to
 * indicate inconsistent ping data.
 * </p>
 */
public class InvalidSuccessfulPingException extends Exception {

    /**
     * Constructs a new InvalidScanTypeException with the specified detail
     * message.
     *
     * @param message the detail message explaining the reason for the exception
     */
    public InvalidSuccessfulPingException(String message) {
        super(message);
    }

    /**
     * Constructs a new InvalidScanTypeException with the specified detail
     * message and cause.
     *
     * @param message the detail message explaining the reason for the exception
     * @param cause the cause (which is saved for later retrieval by the
     * {@link #getCause()} method)
     */
    public InvalidSuccessfulPingException(String message, Throwable cause) {
        super(message, cause);
    }

}
```

InvalidTimeoutRangeException.java

```java
package com.pingpal.exceptions.imports;

/**
 * Thrown to indicate that a timeout value falls outside the allowed range.
 * <p>
 * Use this exception when validating timeout parameters to enforce
 * {@code MIN_TIMEOUT <= timeout <= MAX_TIMEOUT}.
 * </p>
 */
public class InvalidTimeoutRangeException extends Exception {

    /**
     * Constructs a new InvalidNumOfPingsException with the specified detail
     * message.
     *
     * @param message the detail message explaining the reason for the exception
     */
    public InvalidTimeoutRangeException(String message) {
        super(message);
    }

    /**
     * Constructs a new InvalidNumOfPingsException with the specified detail
     * message and cause.
     *
     * @param message the detail message explaining the reason for the exception
     * @param cause the cause (which is saved for later retrieval by the
     * {@link #getCause()} method)
     */
    public InvalidTimeoutRangeException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

InvalidVariableInstanceException.java

```java
package com.pingpal.exceptions.imports;

/**
 * Thrown when a variable’s runtime type does not match the expected type during
 * JSON or data validation.
 * <p>
 * Use this exception in validation utilities to signal that a field was
 * expected to be one type (e.g., Integer) but was another.
 * </p>
 */
public class InvalidVariableInstanceException extends Exception {

    /**
     * Constructs a new InvalidVariableInstanceException with the specified
     * detail message.
     *
     * @param message the detail message explaining the reason for the exception
     */
    public InvalidVariableInstanceException(String message) {
        super(message);
    }

    /**
     * Constructs a new InvalidVariableInstanceException with the specified
     * detail message and cause.
     *
     * @param message the detail message explaining the reason for the exception
     * @param cause the cause (which is saved for later retrieval by the
     * {@link #getCause()} method)
     */
    public InvalidVariableInstanceException(String message, Throwable cause) {
        super(message, cause);
    }

}
```

MissingRequiredKeysException.java

```java
package com.pingpal.exceptions.imports;

/**
 * Thrown when expected top-level keys are missing from a JSON object during
 * import validation.
 * <p>
 * Indicates that one or more required field names (keys) were not present in
 * the JSON being validated.
 * </p>
 */
public class MissingRequiredKeysException extends Exception {

    /**
     * Constructs a new MissingRequiredKeysException with the specified detail
     * message.
     *
     * @param message the detail message explaining the reason for the exception
     */
    public MissingRequiredKeysException(String message) {
        super(message);
    }

    /**
     * Constructs a new MissingRequiredKeysException with the specified detail
     * message and cause.
     *
     * @param message the detail message explaining the reason for the exception
     * @param cause the cause (which is saved for later retrieval by the
     * {@link #getCause()} method)
     */
    public MissingRequiredKeysException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

BlankFieldException.java

```java
package com.pingpal.exceptions.ui;

/**
 * Thrown to indicate that a required UI input field is blank or null.
 * <p>
 * Use this exception during UI validation when a user fails to provide any
 * content for a mandatory text field.
 * </p>
 */
public class BlankFieldException extends Exception {

    /**
     * Constructs a new BlankFieldException with the specified
     * detail message.
     *
     * @param message the detail message explaining the reason for the exception
     */
    public BlankFieldException(String message) {
        super(message);
    }

    /**
     * Constructs a new BlankFieldException with the specified
     * detail message and cause.
     *
     * @param message the detail message explaining the reason for the exception
     * @param cause the cause (which is saved for later retrieval by the
     * {@link #getCause()} method)
     */
    public BlankFieldException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

InvalidIPAddressException.java

```java
package com.pingpal.exceptions.ui;

/**
 * Thrown to indicate that a provided IP address string does not match the
 * expected IPv4 format.
 * <p>
 * Use this exception during UI validation when the user enters an IP address
 * that fails the regex or format checks.
 * </p>
 */
public class InvalidIPAddressException extends Exception {

    /**
     * Constructs a new InvalidIPAddressException with the specified
     * detail message.
     *
     * @param message the detail message explaining the reason for the exception
     */
    public InvalidIPAddressException(String message) {
        super(message);
    }

    /**
     * Constructs a new InvalidIPAddressException with the specified
     * detail message and cause.
     *
     * @param message the detail message explaining the reason for the exception
     * @param cause the cause (which is saved for later retrieval by the
     * {@link #getCause()} method)
     */
    public InvalidIPAddressException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

InvalidNetworkRangeException.java

```java
package com.pingpal.exceptions.ui;

/**
 * Thrown to indicate that a provided network range (CIDR notation) does not
 * conform to the expected format (e.g., "192.168.0.0/24").
 * <p>
 * Use this exception during UI validation when the user-entered network range
 * fails the regex or format checks.
 * </p>
 */
public class InvalidNetworkRangeException extends Exception {

    /**
     * Constructs a new InvalidNetworkRangeException with the specified
     * detail message.
     *
     * @param message the detail message explaining the reason for the exception
     */
    public InvalidNetworkRangeException(String message) {
        super(message);
    }

    /**
     * Constructs a new InvalidNetworkRangeException with the specified
     * detail message and cause.
     *
     * @param message the detail message explaining the reason for the exception
     * @param cause the cause (which is saved for later retrieval by the
     * {@link #getCause()} method)
     */
    public InvalidNetworkRangeException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

InvalidPortRangeException.java

```
package com.pingpal.exceptions.ui;

/**
 * Thrown to indicate that a provided port range is invalid, for example when
 * the lower bound is greater than the upper bound.
 * <p>
 * Use this exception during UI validation when users enter a port range that
 * does not satisfy {@code minPort <= bottomRange <= topRange <= maxPort}.
 * </p>
 */
public class InvalidPortRangeException extends Exception {

    /**
     * Constructs a new InvalidPortRangeException with the specified
     * detail message.
     *
     * @param message the detail message explaining the reason for the exception
     */
    public InvalidPortRangeException(String message) {
        super(message);
    }

    /**
     * Constructs a new InvalidPortRangeException with the specified
     * detail message and cause.
     *
     * @param message the detail message explaining the reason for the exception
     * @param cause the cause (which is saved for later retrieval by the
     * {@link #getCause()} method)
     */
    public InvalidPortRangeException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

Main.java

```java
package com.pingpal;

import com.pingpal.ui.HomePage;
import static java.awt.EventQueue.invokeLater;
import java.io.IOException;
import javax.swing.JOptionPane;
import javax.swing.JPanel;

/**
 * The {@code Main} class is the fundamental class which is responsible for
 * loading and executing the entire program.
 */
public class Main {

    public static void main(String[] args) throws IOException, InterruptedException {

        // Create a Home Page object instance.
        HomePage homePage = new HomePage();

        try {
            // Display the form.
            invokeLater(() -> {
                homePage.setVisible(true);
            });
        } catch (Exception e) {
            // Catch any error that may occur and display a pane to inform the user of the
error.
            JOptionPane.showMessageDialog(new JPanel(), "Encountered error when launching
PingPal.", "Runtime Error", JOptionPane.ERROR_MESSAGE);
        }
    }
}
```

HomePage.java

```java
package com.pingpal.ui;

import com.pingpal.datavalidation.ValidationUtils;
import com.pingpal.deviceping.DevicePing;
import com.pingpal.deviceping.DevicePingResult;
import com.pingpal.exceptions.imports.InvalidScanTypeException;
import com.pingpal.exceptions.ui.BlankFieldException;
import com.pingpal.exceptions.ui.InvalidIPAddressException;
import com.pingpal.exceptions.ui.InvalidNetworkRangeException;
import com.pingpal.exceptions.ui.InvalidPortRangeException;
import com.pingpal.exports.ExportResults;
import com.pingpal.imports.ImportResults;
import com.pingpal.imports.ImportResultsListener;
import com.pingpal.portscan.PortScan;
import com.pingpal.portscan.PortScanResult;
import com.pingpal.subnetscan.SubnetScan;
import com.pingpal.subnetscan.SubnetScanResult;
import com.pingpal.tcpmessage.connect.TCPMessageConnect;
import com.pingpal.tcpmessage.listen.TCPMessageListen;
import java.awt.CardLayout;
import java.awt.Color;
import static java.awt.EventQueue.invokeLater;
import java.util.ArrayList;
import javax.swing.ImageIcon;
import javax.swing.JOptionPane;
import javax.swing.table.DefaultTableModel;

/**
 * The main application window.
 * <p>
 * Uses a {@code CardLayout} to switch between panels for subnet-scan,
 * device-ping, port-scan, TCP messaging, import/export, and help screens.
 * </p>
 * <p>
 * Inherits the properties of the {@code javax.swing.JFrame} class.
 * </p>
 * <p>
 * Implements the properties of the {@code ImportResultsListener} class.
 * </p>
 */
public class HomePage extends javax.swing.JFrame implements ImportResultsListener {

    // The card layout responsible for managing all of the different cards containing the
different scans.
    private static CardLayout card;

    // Flag indicating whether the TCP message funciton was used last.
    private boolean tcpMessageLast = true;
    // Flag indicating whether the TCP message listen function was used last.
    private boolean listenLast = false;

    // Network range text field default text constant.
    private final String DEFAULT_NETWORK_RANGE_TEXT = "e.g. 192.168.0.0/24";
    // IP address text field default text constant.
    private final String DEFAULT_IP_ADDRESS_TEXT = "e.g. 192.168.0.1";

    // Flag indicating if a subnect scan is in progress.
    private boolean subnetScanInProgress = false;
    // Thread used to execute a subnet scan.
    private Thread subnetScanThread;
    // Subnet scan object containing all the functionality.
    private SubnetScan subnetScan;

    // Flag indicating if a device ping is in progress.
    private boolean devicePingInProgress = false;
    // Thread used to execute a device ping.
    private Thread devicePingThread;
    // Device ping object containing all the functionality.
    private DevicePing devicePing;

    // Flag indicating if a port scan is in progress.
    private boolean portScanInProgress = false;
    // Thread used to execute a port scan.
    private Thread portScanThread;
    // Port scan object containing all of the functionality.
    private PortScan portScan;

    // Flag indicating if a TCP message listen is in progress.
    private boolean tcpMessageListenInProgress = false;
    // Thread used to execute a TCP message listen.
    private Thread tcpMessageListenThread;
    // TCP message listen object containing all of the functionality.
    private TCPMessageListen tcpMessageListen;

    // Flag indicating if a TCP message connect is in progress.
    private boolean tcpMessageConnectInProgress = false;
    // Thread used to execute a TCP message connect.
    private Thread tcpMessageConnectThread;
    // TCP message connect object containing all of the functionality.
    private TCPMessageConnect tcpMessageConnect;

    /**
     * Creates new form HomePage
     */
    public HomePage() {
        // Initialise components.
        initComponents();

        // Set window properties.
        this.setLocationRelativeTo(null);
        this.setResizable(false);

        // Set app icon.
        ImageIcon icon = new ImageIcon(HomePage.class.getResource(
                "/com/pingpal/resources/images/icon.png"));
        this.setIconImage(icon.getImage());

        // Initialise card layout.
        card = (CardLayout) pnlMainPanel.getLayout();
        card.show(pnlMainPanel, "pnlMainPage");
    }

    /**
     * This method is called from within the constructor to initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is always
     * regenerated by the Form Editor.
     */
    @SuppressWarnings("unchecked")
    <<Generated Code>>
    /**
     * Called when the user has pressed the button to switch to the subnet scan
     * card.
     *
     * @param evt the variable containing all of information regarding the
     * button press
     */
    private void btnSubnetScanActionPerformed(java.awt.event.ActionEvent evt) {
        // Show the subnet scan card.
        card.show(pnlMainPanel, "cardSubnetScan");
        // Mark the TCP message last flag false.
        tcpMessageLast = false;
    }

    /**
     * Called when the user has pressed the button to switch to the device ping
     * card.
     *
     * @param evt the variable containing all of information regarding the
     * button press
     */
    private void btnDevicePingActionPerformed(java.awt.event.ActionEvent evt) {
        // Show the device ping card.
        card.show(pnlMainPanel, "cardDevicePing");
        // Mark the TCP message last flag false.
        tcpMessageLast = false;
    }

    /**
     * Called when the user has pressed the button to switch to the port scan
     * card.
     *
     * @param evt the variable containing all of information regarding the
     * button press
     */
    private void btnPortScanActionPerformed(java.awt.event.ActionEvent evt) {
        // Show the port scan card.
        card.show(pnlMainPanel, "cardPortScan");
        // Mark the TCP message last flag false.
        tcpMessageLast = false;
    }

    /**
     * Called when the user has pressed the general button to switch to the
     * either the last used TCP Message function card or the listen function if
     * it is the first time accessing it.
     *
     * @param evt the variable containing all of information regarding the
     * button press
     */
    private void btnTCPMessageActionPerformed(java.awt.event.ActionEvent evt) {
        /**
         * If the TCP message listen card is currently being shown, cycle to the
         * TCP message connect card.
         */
        if (tcpMessageLast && listenLast) {
            // Show the TCP message connect card.
            card.show(pnlMainPanel, "cardTCPMessageConnect");
            // Mark the TCP message listen last flag false.
            listenLast = false;

            /**
```

```
             * If the TCP message connect card is currently being shown, cycle
             * to the TCP message listen card.
             */
        } else if (tcpMessageLast && !listenLast) {
            // Show the TCP message listen card.
            card.show(pnlMainPanel, "cardTCPMessageListen");
            // Mark the TCP message listen last flag true.
            listenLast = true;

            /**
             * If the TCP message card is not being shown, but the last TCP
             * message card shown was a TCP message listen card, cycle to the
             * TCP message listen card.
             */
        } else if (!tcpMessageLast && listenLast) {
            // Show the TCP message listen last card.
            card.show(pnlMainPanel, "cardTCPMessageListen");
            // Mark the TCP message listen last flag true.
            listenLast = true;

            /**
             * If the TCP message card is not being shown, but the last TCP
             * message card shown was a TCP message connect card, cycle to the
             * TCP message connect card.
             */
        } else if (!tcpMessageLast && !listenLast) {
            // Show the TCP message connect card.
            card.show(pnlMainPanel, "cardTCPMessageConnect");
            // Mark the TCP message listen last flag false.
            listenLast = false;
        }

        // Mark the TCP message last flag true.
        tcpMessageLast = true;
    }

    /**
     * Called when the user has pressed the button to switch to import results.
     *
     * @param evt the variable containing all of information regarding the
     * button press
     */
    private void btnImportResultsActionPerformed(java.awt.event.ActionEvent evt) {
        // Create a new ImportResutls object.
        ImportResults importResults = new ImportResults(pnlHomePage, this);

        // Prompt the used to select a file to import the results from.
        importResults.setImportResultsPath();

        try {
            // Determine if the scan is a PingPal scan, and if so import the results.
            importResults.determineScanType();
        } catch (InvalidScanTypeException ex) {
            // Display an error message if the contents of the file are not that of a PingPal
scan.
            JOptionPane.showMessageDialog(pnlHomePage, "This file does not contain the
results of a PingPal scan.", "Results File Error", JOptionPane.ERROR_MESSAGE);
        }
    }

    /**
     * Called when the user has ticked the checkbox to select continuous pinging
     * during a device ping.
     *
     * @param evt the variable containing all of information regarding the
     * button press
     */
    private void chkContinuousPingingActionPerformed(java.awt.event.ActionEvent evt) {
        // If the checkbox is selected.
        if (chkContinuousPinging.isSelected()) {
            // Gray out the spinner used to select the number of pings to indicate that this
field no longer applies.
            lblNumberOfPings.setForeground(ValidationUtils.GRAYED_OUT_COLOR);
            // Disable the user from being able to change the value of the spinner.
            spnNumberOfPings.setEnabled(false);

            // If the check box is unselected.
        } else {
            // Return the spinner to default colour.
            lblNumberOfPings.setForeground(ValidationUtils.NORMAL_TEXT_COLOR);
            // Enable the user to be able to change the value of the spinner.
            spnNumberOfPings.setEnabled(true);
        }
    }

    /**
     * Called when the user has pressed the button to export the results of a
     * device ping.
     *
     * @param evt the variable containing all of information regarding the
     * button press
     */
    private void btnExportResultsDevicePingActionPerformed(java.awt.event.ActionEvent evt) {
        /**
         * If the device ping thread is not null and if it is alive, and if a
         * set of results exists.
         */
        if (devicePingThread != null && devicePingThread.isAlive() && devicePing != null) {
            // Signal the ping logic to stop.
            devicePing.requestStop();
            /// Interrupt the thread to break out of blocking calls.
            devicePingThread.interrupt();
        }

        // Create a new ExportResults object and export the device ping results.
        ExportResults exportResults = new ExportResults(pnlDevicePing);
        exportResults.exportResults(devicePing);
    }

    /**
     * Handles the Start/Stop Port Scan button click.
     * <p>
     * When no scan is running, validates inputs, updates UI, and launches a
     * background thread to perform the port scan. If a scan is already running,
     * requests it to stop.
     * </p>
     *
     * @param evt the variable containing all of information regarding the
     * button press
     */
    private void btnStartPortScanActionPerformed(java.awt.event.ActionEvent evt) {
        // If no port scan is currently in progress.
        if (!portScanInProgress) {
            // Clear previous IP address error markers.
            txfIPAddressPortScan.setBackground(Color.WHITE);
            lblIPAddressErrorPortScan.setText("");

            // Clear previous port range error markers.
            spnBottomRangePort.setBackground(Color.WHITE);
            spnTopRangePort.setBackground(Color.WHITE);
            lblPortRangeError.setText("");

            // Flag to track overall input validity.
            boolean valid = true;

            // Validate IP address presence and format.
            try {
                // Check that the IP text is not blank.
                ValidationUtils.validateFieldPresence(txfIPAddressPortScan.getText());
                // Check that the IP text matches the expected pattern.
                ValidationUtils.validateIPAddress(txfIPAddressPortScan.getText());

            } catch (BlankFieldException e) {
                // Mark the IP field as erroneous due to lack of presence.
                txfIPAddressPortScan.setBackground(ValidationUtils.ERROR_COLOR);
                lblIPAddressErrorPortScan.setText("Please enter an IP Address.");
                // Set flag to indicate invalidity.
                valid = false;

            } catch (InvalidIPAddressException e) {
                // Mark the IP field as format-invalid.
                txfIPAddressPortScan.setBackground(ValidationUtils.ERROR_COLOR);
                lblIPAddressErrorPortScan.setText("Please enter an IP Address in the correct
format (e.g. 192.168.0.1).");
                // Set flag to indicate invalidity.
                valid = false;
            }

            // Validate port range.
            try {
                // Ensure bottom <= top.
                ValidationUtils.validatePortRange((int)
spnBottomRangePort.getModel().getValue(), (int) spnTopRangePort.getModel().getValue());

            } catch (InvalidPortRangeException e) {
                // Mark both spinners as erroneous.
                spnBottomRangePort.setBackground(ValidationUtils.ERROR_COLOR);
                spnTopRangePort.setBackground(ValidationUtils.ERROR_COLOR);
                lblPortRangeError.setText("Start value can't be larger than end value. Please
enter a valid range.");
                // Set flag to indicate invalidity.
                valid = false;
            }

            // If any validation failed, don't proceed.
            if (!valid) {
                return;
            }

            // If all checks successful, update UI to show port scan has started.
            portScanInProgress = true;

            // Update status label to show activity.
            lblPortScanInProgress.setForeground(ValidationUtils.SUCCESSFUL_SCAN_COLOR);
            lblPortScanInProgress.setText("Running port scan.");
            // Change button text to allow stopping.
            btnStartPortScan.setText("End Port Scan");

            // Retrieve user-entered parameters (IP Address, Port Range, and Timeout values).
            String ipAddress = txfIPAddressPortScan.getText();
            int bottomRangePort = (int) spnBottomRangePort.getModel().getValue();
            int topRangePort = (int) spnTopRangePort.getModel().getValue();
            int timeout = (int) spnTimeoutPortScan.getModel().getValue();

            // Instantiate the Port Scan logic with the UI table and progress bar.
            portScan = new PortScan(ipAddress, bottomRangePort, topRangePort, timeout,
tblPortScan, prgPortScan);

            // Create and start a new thread to run the scan asynchronously.
            portScanThread = new Thread(() -> {
                // Begin scanning.
                portScan.start();

                // Once done, reset the button label on the EDT.
                invokeLater(() -> btnStartPortScan.setText("Start Port Scan"));

                // Depending on whether the user requested the stop, update the status label.
                if (portScan.isStopRequested()) {
                    // If the used requested the stop update the UI components on the EDT to
indicate such.
                    invokeLater(() -> {

lblPortScanInProgress.setForeground(ValidationUtils.INTERRUPTED_SCAN_COLOR);
                        lblPortScanInProgress.setText("Port scan interrupted.");
                    });
                } else {
                    // If the used did not request the stop update the UI components on the
EDT to indicate such.
                    invokeLater(() -> {

lblPortScanInProgress.setForeground(ValidationUtils.SUCCESSFUL_SCAN_COLOR);
                        lblPortScanInProgress.setText("Port scan complete.");
                    });
                }

                // Mark that the scan is no longer in progress.
                portScanInProgress = false;
            });

            // Start the thread.
            portScanThread.start();

            // If the button is pressed and a port scan is running, end the current port
scan.
        } else if (portScanThread != null && portScanThread.isAlive() && portScan != null) {
            // Signal the scan logic to stop.
            portScan.requestStop();
            // Forcefully shutdown any remaining executor tasks.
            portScan.shutDownExecutorService();
            // Interrupt the thread to break out of blocking calls.
            portScanThread.interrupt();
        }
    }

    /**
     * Called when the user has pressed the button to export the results of a
     * port scan.
     *
     * @param evt the variable containing all of information regarding the
     * button press
     */
    private void btnExportResultsPortScanActionPerformed(java.awt.event.ActionEvent evt) {
        /**
         * If the port scan thread is not null and if it is alive, and if a set
         * of results exists.
         */
        if (portScanThread != null && portScanThread.isAlive() && portScan != null) {
            // Signal the scan logic to stop.
            portScan.requestStop();
            // Forcefully shutdown any remaining executor tasks.
            portScan.shutDownExecutorService();
            // Interrupt the thread to break out of blocking calls.
            portScanThread.interrupt();
        }

        // Create a new ExportResults object and export the port scan results.
        ExportResults exportResults = new ExportResults(pnlPortScan);
        exportResults.exportResults(portScan);
    }

    /**
     * Handles the Start/Stop Subnet Scan button click.
     * <p>
     * When no scan is running, validates the network-range input, updates the
     * UI, and launches a background thread to perform the subnet scan. If a
     * scan is already running, requests it to stop.
     * </p>
     *
     * @param evt the variable containing all of information regarding the
     * button press
     */
    private void btnStartSubnetScanActionPerformed(java.awt.event.ActionEvent evt) {
        // If no subnet scan is currently in progress.
        if (!subnetScanInProgress) {
            // Clear previous network range error markers.
            txfNetworkRange.setBackground(Color.WHITE);
            lblNetworkRangeError.setText("");

            // Flag to track overall input validity.
            boolean valid = true;

            // Validate network range presence and format.
            try {
                // Ensure the text field is not blank or null.
                ValidationUtils.validateFieldPresence(txfNetworkRange.getText());
                // Ensure the text matches the CIDR pattern (e.g., "192.168.0.0/24").
                ValidationUtils.validateNetworkRange(txfNetworkRange.getText());

            } catch (BlankFieldException ee) {
                // Mark the network range as erroneous due to lack of presence.
                txfNetworkRange.setBackground(ValidationUtils.ERROR_COLOR);
                lblNetworkRangeError.setText("Please enter a network range.");
                // Set flag to indicate invalidity.
                valid = false;

            } catch (InvalidNetworkRangeException e) {
                // Mark the network range as format-invalid.
                txfNetworkRange.setBackground(ValidationUtils.ERROR_COLOR);
                lblNetworkRangeError.setText("Please enter a network range in the correct
format (e.g. 192.168.0.0/24).");
                // Set flag to indicate invalidity.
                valid = false;
            }

            // If any validation failed, don't proceed.
            if (!valid) {
                return;
            }

            // If all checks successful, update UI to show subnet scan has started.
            subnetScanInProgress = true;

            // Update status label to show activity.
            lblSubnetScanInProgress.setForeground(ValidationUtils.SUCCESSFUL_SCAN_COLOR);
            lblSubnetScanInProgress.setText("Running subnet scan.");

            // Change button text to allow stopping.
            btnStartSubnetScan.setText("End Subnet Scan");

            // Retrieve user-entered parameters (Network Range and Timeout values).
            String networkRange = txfNetworkRange.getText();
            int timeout = (int) spnTimeoutSubnetScan.getModel().getValue();

            // Instantiate the Subnet Scan logic with the UI table and progress bar.
            subnetScan = new SubnetScan(networkRange, timeout, tblSubnetScan, prgSubnetScan);

            // Create and start a new thread to run the scan asynchronously.
            subnetScanThread = new Thread(() -> {
                // Begin scanning.
                subnetScan.start();

                // Once done, reset the button label on the EDT.
                invokeLater(() -> btnStartSubnetScan.setText("Start Subnet Scan"));

                // Depending on whether the user requested the stop, update the status label.
                if (subnetScan.isStopRequested()) {
                    invokeLater(() -> {
                        // If the used requested the stop update the UI components on the EDT
to indicate such.

lblSubnetScanInProgress.setForeground(ValidationUtils.INTERRUPTED_SCAN_COLOR);
                        lblSubnetScanInProgress.setText("Subnet scan interrupted.");
                    });
                } else {
                    invokeLater(() -> {
                        // If the used did not request the stop update the UI components on
the EDT to indicate such.

lblSubnetScanInProgress.setForeground(ValidationUtils.SUCCESSFUL_SCAN_COLOR);
                        lblSubnetScanInProgress.setText("Subnet scan complete.");
                    });
                }

                // Mark that the scan is no longer in progress.
                subnetScanInProgress = false;
            });

            // Start the thread.
            subnetScanThread.start();

            // If the button is pressed and a subnet scan is running, end the current subnet
```

```
scan.
        } else if (subnetScanThread != null && subnetScanThread.isAlive() && subnetScan !=
null) {
            // Signal the scan logic to stop.
            subnetScan.requestStop();
            // Forcefully shutdown any remaining executor tasks.
            subnetScan.shutDownExecutorService();
            // Interrupt the thread to break out of blocking calls.
            subnetScanThread.interrupt();
        }
    }

    /**
     * Handles the Start/Stop Device Ping button click.
     * <p>
     * When no ping is running, validates the IP address input, updates the UI,
     * and launches a background thread to perform the device ping. If a ping is
     * already running, requests it to stop.
     * </p>
     *
     * @param evt the variable containing all of information regarding the
     * button press
     */
    private void btnStartDevicePingActionPerformed(java.awt.event.ActionEvent evt) {
        // If no subnet scan is currently in progress.
        if (!devicePingInProgress) {
            // Clear previous IP address error markers.
            txfIPAddressDevicePing.setBackground(Color.WHITE);
            lblIPAddressErrorDevicePing.setText("");

            // Flag to track overall input validity.
            boolean valid = true;

            // Validate IP address presence and format.
            try {
                // Ensure the IP text is not blank or null.
                ValidationUtils.validateFieldPresence(txfIPAddressDevicePing.getText());
                // Check that the IP text matches the expected pattern.
                ValidationUtils.validateIPAddress(txfIPAddressDevicePing.getText());

            } catch (BlankFieldException e) {
                // Mark the IP address as erroneous due to lack of presence.
                txfIPAddressDevicePing.setBackground(ValidationUtils.ERROR_COLOR);
                lblIPAddressErrorDevicePing.setText("Please enter an IP Address.");
                // Set flag to indicate invalidity.
                valid = false;
            } catch (InvalidIPAddressException e) {
                // Mark the IP address as format-invalid.
                txfIPAddressDevicePing.setBackground(ValidationUtils.ERROR_COLOR);
                lblIPAddressErrorDevicePing.setText("Please enter an IP Address in the
correct format (e.g. 192.168.0.1).");
                // Set flag to indicate invalidity.
                valid = false;
            }

            // If any validation failed, don't proceed.
            if (!valid) {
                return;
            }

            // If all checks successful, update UI to show device ping has started.
            devicePingInProgress = true;
            // Update status label to show activity.
            lblDevicePingInProgress.setForeground(ValidationUtils.SUCCESSFUL_SCAN_COLOR);
            lblDevicePingInProgress.setText("Running device ping.");

            // Change button text to allow stopping.
            btnStartDevicePing.setText("End Device Ping");

            // Retrieve user-entered parameters (IP address, ping interval, numer of pings,
and continuous pinging values).
            String ipAddress = txfIPAddressDevicePing.getText();
            int pingInterval = Integer.parseInt("" + spnPingInterval.getModel().getValue());
            int numOfPings = Integer.parseInt("" + spnNumberOfPings.getModel().getValue());
            boolean continuousPinging = chkContinuousPinging.isSelected();

            // Instantiate the Device Ping logic with the UI tables.
            devicePing = new DevicePing(ipAddress, pingInterval, numOfPings,
continuousPinging, tblDevicePing, tblDevicePingResponseResults, tblDevicePingPacketResults);

            // Create and start a new thread to run the scan asynchronously.
            devicePingThread = new Thread(() -> {
                // Begin pinging.
                devicePing.start();

                // Once done, reset the button label on the EDT.
                invokeLater(() -> btnStartDevicePing.setText("Start Device Ping"));

                if (devicePing.isStopRequested()) {
                    // If the used requested the stop update the UI components on the EDT to
indicate such.
                    invokeLater(() -> {

lblDevicePingInProgress.setForeground(ValidationUtils.INTERRUPTED_SCAN_COLOR);
                        lblDevicePingInProgress.setText("Device Ping interrupted.");
                    });
                } else {
                    // If the used did not request the stop update the UI components on the
EDT to indicate such.
                    invokeLater(() -> {

lblDevicePingInProgress.setForeground(ValidationUtils.SUCCESSFUL_SCAN_COLOR);
                        lblDevicePingInProgress.setText("Device Ping complete.");
                    });
                }

                // Mark that the ping is no longer in progress.
                devicePingInProgress = false;
            });

            // Start the thread.
            devicePingThread.start();

            // If the button is pressed and a device ping is running, end the current device
ping
        } else if (devicePingThread != null & devicePingThread.isAlive() && devicePing !=
null) {
            // Signal the scan logic to stop.
            devicePing.requestStop();
            // Interrupt the thread to break out of blocking calls.
            devicePingThread.interrupt();
        }
    }
    /**
     * Called when the user has selected the network range text field.
     *
     * @param evt the variable containing all of information regarding the
     * button press
     */
    private void txfNetworkRangeFocusGained(java.awt.event.FocusEvent evt) {
        // Clear any text that is in the text box if the default text is being displayed.
        if (txfNetworkRange.getText().equals(DEFAULT_NETWORK_RANGE_TEXT)) {
            txfNetworkRange.setText("");
        }
    }

    /**
     * Called when the user has selected the device ping IP address text field.
     *
     * @param evt the variable containing all of information regarding the
     * button press
     */
    private void txfIPAddressDevicePingFocusGained(java.awt.event.FocusEvent evt) {
        // Clear any text that is in the text box if the default text is being displayed.
        if (txfIPAddressDevicePing.getText().equals(DEFAULT_IP_ADDRESS_TEXT)) {
            txfIPAddressDevicePing.setText("");
        }
    }

    /**
     * Called when the user has selected the port scan IP address text field.
     *
     * @param evt the variable containing all of information regarding the
     * button press
     */
    private void txfIPAddressPortScanFocusGained(java.awt.event.FocusEvent evt) {
        // Clear any text that is in the text box if the default text is being displayed.
        if (txfIPAddressPortScan.getText().equals(DEFAULT_IP_ADDRESS_TEXT)) {
            txfIPAddressPortScan.setText("");
        }
    }

    /**
     * Called when the user has pressed the button to export the results of a
     * subnet scan.
     *
     * @param evt the variable containing all of information regarding the
     * button press
     */
    private void btnExportResultsSubnetScanActionPerformed(java.awt.event.ActionEvent evt) {
        /**
         * If the subnet scan thread is not null and if it is alive, and if a
         * set of results exists.
         */
        if (subnetScanThread != null && subnetScanThread.isAlive() && subnetScan != null) {
            // Signal the scan logic to stop.
            subnetScan.requestStop();
            // Forcefully shutdown any remaining executor tasks.
            subnetScan.shutDownExecutorService();
            // Interrupt the thread to break out of blocking calls.
            subnetScanThread.interrupt();
        }

        // Create a new ExportResults object and export the subnet scan results.
        ExportResults exportResults = new ExportResults(pnlSubnetScan);
        exportResults.exportResults(subnetScan);
    }

    /**
     * Called when the user has pressed the button to switch to the TCP message
     * listen card.
     *
     * @param evt the variable containing all of information regarding the
     * button press
     */
    private void btnTCPMessageListenActionPerformed(java.awt.event.ActionEvent evt) {
        // Show the TCP message listen card.
        card.show(pnlMainPanel, "cardTCPMessageListen");
        // Mark the TCP message listen last flag true.
        listenLast = true;
        // Mark the TCP message last flag true.
        tcpMessageLast = true;
    }

    /**
     * Called when the user has pressed the button to switch to the TCP message
     * connect card.
     *
     * @param evt the variable containing all of information regarding the
     * button press
     */
    private void btnTCPMessageConnectActionPerformed(java.awt.event.ActionEvent evt) {
        // Show the TCP message connect card.
        card.show(pnlMainPanel, "cardTCPMessageConnect");
        // Mark the TCP message listen last flag false.
        listenLast = false;
        // Mark the TCP message last flag false.
        tcpMessageLast = true;
    }

    /**
     * Handles the action triggered when the "Start TCP Listen" button is
     * clicked.
     * <p>
     * Starts or stops a TCP message connect based on the current state. If no
     * connection is active, this method validates the input, updates the UI,
     * creates a new TCPMessageListen object, and starts a new thread to handle
     * the connection. If a connection is already active, it stops the running
     * connection.
     * </p>
     *
     * @param evt the variable containing all of information regarding the
     * button press
     */
    private void btnStartTCPMessageListenActionPerformed(java.awt.event.ActionEvent evt) {
        // Clear previous network range error markers.
        if (!tcpMessageListenInProgress) {
            // Get the entered port value.
            int port = (int) spnPortTCPMessageListen.getValue();

            // Update UI to show TCP message listen has started.
            tcpMessageListenInProgress = true;

            // Update status label to show activity.

lblTCPMessageListenInProgress.setForeground(ValidationUtils.SUCCESSFUL_SCAN_COLOR);
```

```
            lblTCPMessageListenInProgress.setText("Running TCP message listen.");

            // Change button text to allow stopping.
            btnStartTCPMessageListen.setText("End TCP Listen");

            // Create a new TCPMessageListen instance using the provided port.
            tcpMessageListen = new TCPMessageListen(port, txpTCPMessageListen);

            // Create a new thread to handle the TCP connection listnening in the background.
            tcpMessageListenThread = new Thread(() -> {
                // Begin trying to establish a connection.
                tcpMessageListen.start();

                // Once the connection finishes, update the UI back on the Swing thread to
show TCP message listen has stopped.
                invokeLater(() -> {
                    btnStartTCPMessageListen.setText("Start TCP Listen");

lblTCPMessageListenInProgress.setForeground(ValidationUtils.SUCCESSFUL_SCAN_COLOR);
                    lblTCPMessageListenInProgress.setText("TCP message listen stopped.");
                });

                // Mark the TCP message connect as no longer running.
                tcpMessageListenInProgress = false;
            });

            // Start the thread.
            tcpMessageListenThread.start();

            // If the button is pressed and a TCP message listen is running, end the current
TCP message listen.
        } else if (tcpMessageListenThread != null && tcpMessageListenThread.isAlive() &&
tcpMessageListen != null) {
            tcpMessageListen.requestStop();
            tcpMessageListenInProgress = false;
        }
    }

    /**
     * Handles the "Export Results" button click on the TCP Message Listen
     * panel.
     * <p>
     * If a live TCP connection thread is active, stops it; then prompts the
     * user for a directory and filename, and writes the current text-paned
     * contents to a file via the ExportResults utility.
     * </p>
     *
     * @param evt the variable containing all of information regarding the
     * button press
     */
    private void btnExportResultsTCPMessageListenActionPerformed(java.awt.event.ActionEvent
evt) {
        // If the TCP connection thread exists and is currently running...
        if (tcpMessageListenThread != null && tcpMessageListenThread.isAlive() &&
tcpMessageListen != null) {
            // Request the TCP connect logic to stop receiving/sending.
            tcpMessageListen.requestStop();
            // Interrupt the thread to break out of any blocking operations.
            tcpMessageListenThread.interrupt();
        }

        // Create an ExportResults helper, using the listen panel as the parent component.
        ExportResults exportResults = new ExportResults(pnlTCPMessageListen);
        // Delegate to the helper to export the contents of the TCPMessageListen text pane.
        exportResults.exportResults(tcpMessageListen);
    }

    /**
     * Handles the "Send" button click on the TCP Message Listen panel.
     * <p>
     * Sends the text from the input field over the active TCP connection, then
     * clears the input field. Only sends if the connection thread is alive and
     * the socket is currently connected.
     * </p>
     *
     * @param evt the variable containing all of information regarding the
     * button press
     */
    private void btnSendTCPMessageListenActionPerformed(java.awt.event.ActionEvent evt) {
        // Check if the TCP connection thread exists, is running, and the socket is
connected.
        if (tcpMessageListenThread != null && tcpMessageListenThread.isAlive() &&
tcpMessageListen != null && tcpMessageListen.isDeviceConnected()) {
            // Retrieve the message text from the input field and send it.
            tcpMessageListen.sendMessage(txfMessageTCPMessageListen.getText());
            // Clear the input field for the next message.
            txfMessageTCPMessageListen.setText("");
        }
    }

    /**
     * Handles the action triggered when the "Start TCP Connect" button is
     * clicked.
     * <p>
     * Starts or stops a TCP message connection based on the current state. If
     * no connection is active, this method validates the input, updates the UI,
     * creates a new TCPMessageConnect object, and starts a new thread to handle
     * the connection. If a connection is already active, it stops the running
     * connection.
     * </p>
     *
     * @param evt the variable containing all of information regarding the
     * button press
     */
    private void btnStartTCPMessageConnectActionPerformed(java.awt.event.ActionEvent evt) {
        // Clear previous network range error markers.
        if (!tcpMessageConnectInProgress) {
            // Clear previous IP address error markers.
            txfIPAddressTCPMessageConnect.setBackground(Color.WHITE);
            lblIPAddressErrorTCPMessageConnect.setText("");

            // Flag to track overall input validity.
            boolean valid = true;

            // Validate IP address presence and format.
            try {
                // Check that the IP text is not blank.

ValidationUtils.validateFieldPresence(txfIPAddressTCPMessageConnect.getText());
                // Check that the IP text matches the expected pattern.
                ValidationUtils.validateIPAddress(txfIPAddressTCPMessageConnect.getText());

            } catch (BlankFieldException e) {
                // Mark the IP field as erroneous due to lack of presence.
                txfIPAddressTCPMessageConnect.setBackground(ValidationUtils.ERROR_COLOR);
                lblIPAddressErrorTCPMessageConnect.setText("Please enter an IP Address.");
                // Set flag to indicate invalidity.
                valid = false;
            } catch (InvalidIPAddressException e) {
                // Mark the IP field as format-invalid.
                txfIPAddressTCPMessageConnect.setBackground(ValidationUtils.ERROR_COLOR);
                lblIPAddressErrorTCPMessageConnect.setText("Please enter an IP Address in the
correct format (e.g. 192.168.0.1).");
                // Set flag to indicate invalidity.
                valid = false;
            }

            // If any validation failed, don't proceed.
            if (!valid) {
                return;
            }

            // If all checks successful, update UI to show TCP message connect has started.
            tcpMessageConnectInProgress = true;

            // Update status label to show activity.

lblTCPMessageConnectInProgress.setForeground(ValidationUtils.SUCCESSFUL_SCAN_COLOR);
            lblTCPMessageConnectInProgress.setText("Running TCP message connect.");

            // Change button text to allow stopping.
            btnStartTCPMessageConnect.setText("End TCP Connect");

            // Create a new TCPMessageConnect instance using the provided IP and port.
            tcpMessageConnect = new
TCPMessageConnect(txfIPAddressTCPMessageConnect.getText(), (int)
spnPortTCPMessageConnect.getValue(), txpTCPMessageConnect);

            // Create a new thread to handle the TCP connection in the background.
            tcpMessageConnectThread = new Thread(() -> {
                // Begin trying to establish a connection.
                tcpMessageConnect.start();

                // Once the connection finishes, update the UI back on the Swing thread to
show TCP message connect has stopped.
                invokeLater(() -> {
                    btnStartTCPMessageConnect.setText("Start TCP Connect");

lblTCPMessageConnectInProgress.setForeground(ValidationUtils.SUCCESSFUL_SCAN_COLOR);
                    lblTCPMessageConnectInProgress.setText("TCP message connect stopped.");
                });

                // Mark the TCP message connect as no longer running.
                tcpMessageConnectInProgress = false;
            });

            // Start the thread.
            tcpMessageConnectThread.start();

            // If the button is pressed and a TCP message listen is running, end the current
TCP message listen.
        } else if (tcpMessageConnectThread != null && tcpMessageConnectThread.isAlive() &&
tcpMessageConnect != null) {
            tcpMessageConnect.requestStop();
            tcpMessageConnectInProgress = false;
        }
```

```java
    }

    /**
     * Called when the user has selected the TCP message connect IP address text
     * field.
     *
     * @param evt the variable containing all of information regarding the
     * button press
     */
    private void txfIPAddressTCPMessageConnectFocusGained(java.awt.event.FocusEvent evt) {
        // Clear any text that is in the text box if the default text is being displayed.
        if (txfIPAddressTCPMessageConnect.getText().equals(DEFAULT_IP_ADDRESS_TEXT)) {
            txfIPAddressTCPMessageConnect.setText("");
        }
    }

    /**
     * Handles the "Send" button click on the TCP Message Connect panel.
     * <p>
     * Sends the text from the input field over the active TCP connection, then
     * clears the input field. Only sends if the connection thread is alive and
     * the socket is currently connected.
     * </p>
     *
     * @param evt the variable containing all of information regarding the
     * button press
     */
    private void btnSendTCPMessageConnectActionPerformed(java.awt.event.ActionEvent evt) {
        // Check if the TCP connection thread exists, is running, and the socket is
connected.
        if (tcpMessageConnectThread != null && tcpMessageConnectThread.isAlive() &&
tcpMessageConnect != null && tcpMessageConnect.isDeviceConnected()) {
            // Retrieve the message text from the input field and send it.
            tcpMessageConnect.sendMessage(txfMessageTCPMessageConnect.getText());
            // Clear the input field for the next message.
            txfMessageTCPMessageConnect.setText("");
        }
    }

    /**
     * Handles the "Export Results" button click on the TCP Message Connect
     * panel.
     * <p>
     * If a live TCP connection thread is active, stops it; then prompts the
     * user for a directory and filename, and writes the current text-paned
     * contents to a file via the ExportResults utility.
     * </p>
     *
     * @param evt the variable containing all of information regarding the
     * button press
     */
    private void btnExportResultsTCPMessageConnectActionPerformed(java.awt.event.ActionEvent
evt) {
        // If the TCP connection thread exists and is currently running...
        if (tcpMessageConnectThread != null && tcpMessageConnectThread.isAlive() &&
tcpMessageConnect != null) {
            // Request the TCP connect logic to stop receiving/sending.
            tcpMessageConnect.requestStop();
            // Interrupt the thread to break out of any blocking operations.
            tcpMessageConnectThread.interrupt();
        }
        // Create an ExportResults helper, using the connect panel as the parent component.
        ExportResults exportResults = new ExportResults(pnlTCPMessageConnect);
        // Delegate to the helper to export the contents of the TCPMessageConnect text pane.
        exportResults.exportResults(tcpMessageConnect);
    }

    /**
     * Callback invoked when subnet scan results have been imported from a file.
     * <p>
     * This method stops any ongoing scan, displays the Subnet Scan panel,
     * populates the UI controls with imported parameters, loads the results
     * into the table, and resets the UI to reflect a completed import state.
     * </p>
     *
     * @param networkRange the CIDR network range used in the imported scan
     * @param timeout the timeout value (ms) used in the imported scan
     * @param subnetScanResults the list of imported SubnetScanResult objects
     */
    @Override
    public void onSubnetScanResultsImported(String networkRange, int timeout,
ArrayList<SubnetScanResult> subnetScanResults) {
        // If a subnet scan is currently running, request it to stop immediately.
        if (subnetScanThread != null && subnetScanThread.isAlive() && subnetScan != null) {
            subnetScan.requestStop();
            subnetScan.shutDownExecutorService();
            subnetScanThread.interrupt();
        }

        // Show the subnet scan card.
        card.show(pnlMainPanel, "cardSubnetScan");
        // Remember that TCP messaging was not last used.
        tcpMessageLast = false;

        // Clear previous network range error markers.
        txfNetworkRange.setBackground(Color.WHITE);
        lblNetworkRangeError.setText("");

        // Set imported network range.
        txfNetworkRange.setText(networkRange);

        // Set imported timeout.
        spnTimeoutSubnetScan.getModel().setValue(timeout);

        // Clear the table.
        DefaultTableModel model = (DefaultTableModel) tblSubnetScan.getModel();

        model.setRowCount(0);

        // Import the results from the file into the table.
        for (SubnetScanResult subnetScanResult : subnetScanResults) {
            model.addRow(new Object[]{subnetScanResult.getIPAddress()});
        }

        // Update the subnet scan object.
        subnetScan = new SubnetScan(networkRange, timeout, tblSubnetScan, prgSubnetScan);
        // Load the imported results into the new object.
        subnetScan.setSubnetScanResults(subnetScanResults);

        // Reset button text.
        btnStartSubnetScan.setText("Start Subnet Scan");
        // Set status message.
        lblSubnetScanInProgress.setText("Results imported successfully.");
        // Update progress bar.
        prgSubnetScan.setValue(100);
    }

    /**
     * Callback invoked when device ping results have been imported from a file.
     * <p>
     * This method stops any ongoing scan, displays the Device Ping panel,
     * populates the UI controls with imported parameters, loads the results
     * into the table, and resets the UI to reflect a completed import state.
     * </p>
     *
     * @param ipAddress the target IP address used in the imported scan
     * @param pingInterval the time interval between each ping in the imported
     * scan
     * @param numOfPings the number of times a device is pinged in the imported
     * scan
     * @param continuousPinging flag to determine whether a device was pinged
     * continuously in the imported scan
     * @param devicePingResults the list of imported devicePingResult objects
     */
    @Override
    public void onDevicePingResultsImported(String ipAddress, int pingInterval, int
numOfPings, boolean continuousPinging, ArrayList<DevicePingResult> devicePingResults) {
        // If a device ping is currently running, request it to stop immediately.
        if (devicePingThread != null && devicePingThread.isAlive() && devicePing != null) {
            devicePing.requestStop();
            devicePingThread.interrupt();
        }

        // Show the device ping card.
        card.show(pnlMainPanel, "cardDevicePing");
        // Remember that TCP messaging was not last used.
        tcpMessageLast = false;

        // Clear previous IP address error markers.
        txfIPAddressDevicePing.setBackground(Color.WHITE);
        lblIPAddressErrorDevicePing.setText("");
        // Set imported IP address.
        txfIPAddressDevicePing.setText(ipAddress);

        // Set imported ping interval.
        spnPingInterval.setValue(pingInterval);

        // Set imported number of pings value.
        spnNumberOfPings.setValue(numOfPings);

        // Set continuous pinging flag to imported value.
        chkContinuousPinging.setSelected(continuousPinging);

        // If the checkbox is selected.
        if (chkContinuousPinging.isSelected()) {
            // Gray out the spinner used to select the number of pings to indicate that this
field no longer applies.
            lblNumberOfPings.setForeground(ValidationUtils.GRAYED_OUT_COLOR);
            // Disable the user from being able to change the value of the spinner.
            spnNumberOfPings.setEnabled(false);

            // If the check box is unselected.
        } else {
            // Return the spinner to default colour.
            lblNumberOfPings.setForeground(ValidationUtils.NORMAL_TEXT_COLOR);
            // Enable the user to be able to change the value of the spinner.
            spnNumberOfPings.setEnabled(true);
        }

        // Clear the tables.
        DefaultTableModel devicePingTableModel = (DefaultTableModel)
tblDevicePing.getModel();
        DefaultTableModel devicePingResponseResultsTableModel = (DefaultTableModel)
tblDevicePingResponseResults.getModel();
        DefaultTableModel devicePingPacketResultsTableModel = (DefaultTableModel)
tblDevicePingPacketResults.getModel();

        devicePingTableModel.setRowCount(0);
        devicePingResponseResultsTableModel.setRowCount(0);
        devicePingPacketResultsTableModel.setRowCount(0);

        // Import the results from the file into the table.
        for (DevicePingResult devicePingResult : devicePingResults) {
            devicePingTableModel.addRow(new Object[]{
                ipAddress,
                devicePingResult.getRoundTripTime(),
                devicePingResult.isSuccessfulPing(),
                devicePingResult.getPacketLoss()
            });
        }

        // Update the device ping object
        devicePing = new DevicePing(ipAddress, pingInterval, numOfPings, continuousPinging,
tblDevicePing, tblDevicePingResponseResults, tblDevicePingPacketResults);
        // Load the imported results into the new object.
        devicePing.setDevicePingResults(devicePingResults);

        // Set counter variables.
        devicePing.setPingCount();
        devicePing.setSuccessfulPings();

        // Update results tables.
        devicePing.populateResultsTables();

        // Reset button text.
        btnStartDevicePing.setText("Start Device Ping");
        // Set status message.
        lblDevicePingInProgress.setText("Results imported successfully.");
    }

    /**
     * Callback invoked when port scan results have been imported from a file.
     * <p>
     * This method stops any ongoing scan, displays the Port Scan panel,
     * populates the UI controls with imported parameters, loads the results
     * into the table, and resets the UI to reflect a completed import state.
     * </p>
     *
     * @param ipAddress the target IP address used in the imported scan
     * @param bottomRangePort the starting port of the imported scan range
     * @param topRangePort the ending port of the imported scan range
     * @param timeout the timeout value (ms) used in the imported scan
     * @param portScanResults the list of imported PortScanResult objects
     */
    @Override
    public void onPortScanResultsImported(String ipAddress, int bottomRangePort, int
```

```java
topRangePort, int timeout, ArrayList<PortScanResult> portScanResults) {
        // If a port scan is currently running, request it to stop immediately.
        if (portScanThread != null && portScanThread.isAlive() && portScan != null) {
            portScan.requestStop();
            portScan.shutDownExecutorService();
            portScanThread.interrupt();
        }

        // Show the port scan card.
        card.show(pnlMainPanel, "cardPortScan");
        // Remember that TCP messaging was not last used.
        tcpMessageLast = false;

        // Clear previous IP address error markers.
        txfIPAddressPortScan.setBackground(Color.WHITE);
        lblIPAddressErrorPortScan.setText("");
        // Set imported IP address.
        txfIPAddressPortScan.setText(ipAddress);

        // Set imported bottom range port.
        spnBottomRangePort.setValue(bottomRangePort);

        // Set imported top range port.
        spnTopRangePort.setValue(topRangePort);

        // Clear previous port range error markers.
        lblPortRangeError.setText("");

        // Set imported timeout.
        spnTimeoutPortScan.setValue(timeout);

        // Clear the table.
        DefaultTableModel model = (DefaultTableModel) tblPortScan.getModel();
        model.setRowCount(0);

        // Import the results from the file into the table.
        for (PortScanResult portScanResult : portScanResults) {
            model.addRow(new Object[]{
                portScanResult.getPortNumber(),
                portScanResult.getProtocol()
            });
        }

        // Update the port scan object.
        portScan = new PortScan(ipAddress, bottomRangePort, topRangePort, timeout,
tblPortScan, prgPortScan);
        // Load the imported results into the new object.
        portScan.setPortScanResults(portScanResults);

        // Reset button text.
        btnStartPortScan.setText("Start Port Scan");
        // Set status message.
        lblPortScanInProgress.setText("Results imported successfully.");

        // Update progress bar.
        prgPortScan.setValue(100);
    }

    // Variables declaration - do not modify
    private javax.swing.JButton btnDevicePing;
    private javax.swing.JButton btnExportResultsDevicePing;
    private javax.swing.JButton btnExportResultsPortScan;
    private javax.swing.JButton btnExportResultsSubnetScan;
    private javax.swing.JButton btnExportResultsTCPMessageConnect;
    private javax.swing.JButton btnExportResultsTCPMessageListen;
    private javax.swing.JButton btnImportResults;
    private javax.swing.JButton btnPortScan;
    private javax.swing.JButton btnSendTCPMessageConnect;
    private javax.swing.JButton btnSendTCPMessageListen;
    private javax.swing.JButton btnStartDevicePing;
    private javax.swing.JButton btnStartPortScan;
    private javax.swing.JButton btnStartSubnetScan;
    private javax.swing.JButton btnStartTCPMessageConnect;
    private javax.swing.JButton btnStartTCPMessageListen;
    private javax.swing.JButton btnSubnetScan;
    private javax.swing.JButton btnTCPMessage;
    private javax.swing.JButton btnTCPMessageConnect;
    private javax.swing.JButton btnTCPMessageListen;
    private javax.swing.JCheckBox chkContinuousPinging;
    private javax.swing.JLabel lblContinuousPinging;
    private javax.swing.JLabel lblDash;
    private javax.swing.JLabel lblDevicePing;
    private javax.swing.JLabel lblDevicePingInProgress;
    private javax.swing.JLabel lblEnterMessageTCPMessageConnect;
    private javax.swing.JLabel lblEnterMessageTCPMessageListen;
    private javax.swing.JLabel lblIPAddressDevicePing;
    private javax.swing.JLabel lblIPAddressErrorDevicePing;
    private javax.swing.JLabel lblIPAddressErrorPortScan;
    private javax.swing.JLabel lblIPAddressErrorTCPMessageConnect;
    private javax.swing.JLabel lblIPAddressTCPMessageConnect;
    private javax.swing.JLabel lblIPAdressPortScan;
    private javax.swing.JLabel lblNetworkRange;
    private javax.swing.JLabel lblNetworkRangeError;
    private javax.swing.JLabel lblNetworking;
    private javax.swing.JLabel lblNoTabOpen;
    private javax.swing.JLabel lblNumberOfPings;
    private javax.swing.JLabel lblPingInterval;
    private javax.swing.JLabel lblPortRange;
    private javax.swing.JLabel lblPortRangeError;
    private javax.swing.JLabel lblPortScan;
    private javax.swing.JLabel lblPortScanInProgress;
    private javax.swing.JLabel lblPortTCPMessageConnect;
    private javax.swing.JLabel lblPortTCPMessageListen;
    private javax.swing.JLabel lblSubnetScan;
    private javax.swing.JLabel lblSubnetScanInProgress;
    private javax.swing.JLabel lblTCPMessageConnect;
    private javax.swing.JLabel lblTCPMessageConnectInProgress;
    private javax.swing.JLabel lblTCPMessageListen;
    private javax.swing.JLabel lblTCPMessageListenInProgress;
    private javax.swing.JLabel lblTimeoutPortScan;
    private javax.swing.JLabel lblTimeoutSubnetScan;
    private javax.swing.JLabel lblTitle;
    private javax.swing.JPanel pnlDevicePing;
    private javax.swing.JPanel pnlHomePage;
    private javax.swing.JPanel pnlLine;
    private javax.swing.JPanel pnlMainPanel;
    private javax.swing.JPanel pnlPortScan;
    private javax.swing.JPanel pnlSideBar;
    private javax.swing.JPanel pnlSubnetScan;
    private javax.swing.JPanel pnlTCPMessageConnect;
    private javax.swing.JPanel pnlTCPMessageListen;
    private javax.swing.JProgressBar prgPortScan;
    private javax.swing.JProgressBar prgSubnetScan;
    private javax.swing.JScrollPane scrDevicePing;
    private javax.swing.JScrollPane scrDevicePingPacketResults;
    private javax.swing.JScrollPane scrDevicePingResponseResults;
    private javax.swing.JScrollPane scrPortScan;
    private javax.swing.JScrollPane scrSubnetScan;
    private javax.swing.JScrollPane scrTCPMessageConnect;
    private javax.swing.JScrollPane scrTCPMessageListen;
    private javax.swing.JSpinner spnBottomRangePort;
    private javax.swing.JSpinner spnNumberOfPings;
    private javax.swing.JSpinner spnPingInterval;
    private javax.swing.JSpinner spnPortTCPMessageConnect;
    private javax.swing.JSpinner spnPortTCPMessageListen;
    private javax.swing.JSpinner spnTimeoutPortScan;
    private javax.swing.JSpinner spnTimeoutSubnetScan;
    private javax.swing.JSpinner spnTopRangePort;
    private javax.swing.JTable tblDevicePing;
    private javax.swing.JTable tblDevicePingPacketResults;
    private javax.swing.JTable tblDevicePingResponseResults;
    private javax.swing.JTable tblPortScan;
    private javax.swing.JTable tblSubnetScan;
    private javax.swing.JTextField txfIPAddressDevicePing;
    private javax.swing.JTextField txfIPAddressPortScan;
    private javax.swing.JTextField txfIPAddressTCPMessageConnect;
    private javax.swing.JTextField txfMessageTCPMessageConnect;
    private javax.swing.JTextField txfMessageTCPMessageListen;
    private javax.swing.JTextField txfNetworkRange;
    private javax.swing.JTextPane txpTCPMessageConnect;
    private javax.swing.JTextPane txpTCPMessageListen;
    // End of variables declaration
}
```

HomePage.java – Generated UI Code

```
private void initComponents() {

        pnlHomePage = new javax.swing.JPanel();
        pnlSideBar = new javax.swing.JPanel();
        lblTitle = new javax.swing.JLabel();
        pnlLine = new javax.swing.JPanel();
        btnSubnetScan = new javax.swing.JButton();
        lblNetworking = new javax.swing.JLabel();
        btnDevicePing = new javax.swing.JButton();
        btnPortScan = new javax.swing.JButton();
        btnTCPMessage = new javax.swing.JButton();
        btnImportResults = new javax.swing.JButton();
        btnTCPMessageListen = new javax.swing.JButton();
        btnTCPMessageConnect = new javax.swing.JButton();
        pnlMainPanel = new javax.swing.JPanel();
        lblNoTabOpen = new javax.swing.JLabel();
        pnlSubnetScan = new javax.swing.JPanel();
        lblSubnetScan = new javax.swing.JLabel();
        lblNetworkRange = new javax.swing.JLabel();
        txfNetworkRange = new javax.swing.JTextField();
        btnStartSubnetScan = new javax.swing.JButton();
        scrSubnetScan = new javax.swing.JScrollPane();
        tblSubnetScan = new javax.swing.JTable();
        btnExportResultsSubnetScan = new javax.swing.JButton();
        lblSubnetScanInProgress = new javax.swing.JLabel();
        lblTimeoutSubnetScan = new javax.swing.JLabel();
        spnTimeoutSubnetScan = new javax.swing.JSpinner();
        lblNetworkRangeError = new javax.swing.JLabel();
        prgSubnetScan = new javax.swing.JProgressBar();
        pnlPortScan = new javax.swing.JPanel();
        lblPortScan = new javax.swing.JLabel();
        txfIPAddressPortScan = new javax.swing.JTextField();
        btnStartPortScan = new javax.swing.JButton();
        spnBottomRangePort = new javax.swing.JSpinner();
        spnTopRangePort = new javax.swing.JSpinner();
        lblIPAdressPortScan = new javax.swing.JLabel();
        lblDash = new javax.swing.JLabel();
        lblPortRange = new javax.swing.JLabel();
        scrPortScan = new javax.swing.JScrollPane();
        tblPortScan = new javax.swing.JTable();
        btnExportResultsPortScan = new javax.swing.JButton();
        lblPortRangeError = new javax.swing.JLabel();
        lblIPAddressErrorPortScan = new javax.swing.JLabel();
        lblPortScanInProgress = new javax.swing.JLabel();
        lblTimeoutPortScan = new javax.swing.JLabel();
        spnTimeoutPortScan = new javax.swing.JSpinner();
        prgPortScan = new javax.swing.JProgressBar();
        pnlTCPMessageListen = new javax.swing.JPanel();
        lblTCPMessageListen = new javax.swing.JLabel();
        lblPortTCPMessageListen = new javax.swing.JLabel();
        spnPortTCPMessageListen = new javax.swing.JSpinner();
        btnStartTCPMessageListen = new javax.swing.JButton();
        lblTCPMessageListenInProgress = new javax.swing.JLabel();
        btnExportResultsTCPMessageListen = new javax.swing.JButton();
        btnSendTCPMessageListen = new javax.swing.JButton();
        lblEnterMessageTCPMessageListen = new javax.swing.JLabel();
        scrTCPMessageListen = new javax.swing.JScrollPane();
        txpTCPMessageListen = new javax.swing.JTextPane();
        txfMessageTCPMessageListen = new javax.swing.JTextField();
        pnlDevicePing = new javax.swing.JPanel();
        lblDevicePing = new javax.swing.JLabel();
        lblIPAddressDevicePing = new javax.swing.JLabel();
        lblPingInterval = new javax.swing.JLabel();
        spnPingInterval = new javax.swing.JSpinner();
        txfIPAddressDevicePing = new javax.swing.JTextField();
        lblContinuousPinging = new javax.swing.JLabel();
        chkContinuousPinging = new javax.swing.JCheckBox();
        lblNumberOfPings = new javax.swing.JLabel();
        spnNumberOfPings = new javax.swing.JSpinner();
        btnStartDevicePing = new javax.swing.JButton();
        scrDevicePing = new javax.swing.JScrollPane();
        tblDevicePing = new javax.swing.JTable();
        btnExportResultsDevicePing = new javax.swing.JButton();
        lblIPAddressErrorDevicePing = new javax.swing.JLabel();
        lblDevicePingInProgress = new javax.swing.JLabel();
        scrDevicePingPacketResults = new javax.swing.JScrollPane();
        tblDevicePingPacketResults = new javax.swing.JTable();
        scrDevicePingResponseResults = new javax.swing.JScrollPane();
        tblDevicePingResponseResults = new javax.swing.JTable();
        pnlTCPMessageConnect = new javax.swing.JPanel();
        lblTCPMessageConnect = new javax.swing.JLabel();
        lblPortTCPMessageConnect = new javax.swing.JLabel();
        lblIPAddressTCPMessageConnect = new javax.swing.JLabel();
        txfIPAddressTCPMessageConnect = new javax.swing.JTextField();
        spnPortTCPMessageConnect = new javax.swing.JSpinner();
        btnStartTCPMessageConnect = new javax.swing.JButton();
        lblIPAddressErrorTCPMessageConnect = new javax.swing.JLabel();
        lblTCPMessageConnectInProgress = new javax.swing.JLabel();
        scrTCPMessageConnect = new javax.swing.JScrollPane();
        txpTCPMessageConnect = new javax.swing.JTextPane();
        lblEnterMessageTCPMessageConnect = new javax.swing.JLabel();
        txfMessageTCPMessageConnect = new javax.swing.JTextField();
        btnSendTCPMessageConnect = new javax.swing.JButton();
        btnExportResultsTCPMessageConnect = new javax.swing.JButton();

        setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
        setTitle("PingPal");
        setBackground(new java.awt.Color(0, 255, 204));
        setCursor(new java.awt.Cursor(java.awt.Cursor.DEFAULT_CURSOR));
        setName("HomePage"); // NOI18N

        pnlHomePage.setBackground(new java.awt.Color(49, 49, 49));
        pnlHomePage.setForeground(new java.awt.Color(215, 215, 215));
        pnlHomePage.setName("PingPal"); // NOI18N
        pnlHomePage.setPreferredSize(new java.awt.Dimension(0, 0));
        pnlHomePage.setVerifyInputWhenFocusTarget(false);

        pnlSideBar.setBackground(new java.awt.Color(45, 45, 45));
        pnlSideBar.setPreferredSize(new java.awt.Dimension(280, 720));

        lblTitle.setFont(new java.awt.Font("Dubai Medium", 1, 64)); // NOI18N
        lblTitle.setForeground(new java.awt.Color(233, 247, 249));
        lblTitle.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);
        lblTitle.setText("PingPal");
        lblTitle.setFocusable(false);
        lblTitle.setHorizontalTextPosition(javax.swing.SwingConstants.CENTER);
        lblTitle.setMaximumSize(new java.awt.Dimension(215, 110));
        lblTitle.setMinimumSize(new java.awt.Dimension(215, 110));
        lblTitle.setPreferredSize(new java.awt.Dimension(215, 110));
        pnlLine.setBackground(new java.awt.Color(233, 247, 249));
        pnlLine.setFocusable(false);
        pnlLine.setPreferredSize(new java.awt.Dimension(270, 2));

        javax.swing.GroupLayout pnlLineLayout = new javax.swing.GroupLayout(pnlLine);
        pnlLine.setLayout(pnlLineLayout);
        pnlLineLayout.setHorizontalGroup(
            pnlLineLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addGap(0, 270, Short.MAX_VALUE)
        );
        pnlLineLayout.setVerticalGroup(
            pnlLineLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addGap(0, 2, Short.MAX_VALUE)
        );

        btnSubnetScan.setBackground(new java.awt.Color(45, 45, 45));
        btnSubnetScan.setFont(new java.awt.Font("Dubai Medium", 0, 16)); // NOI18N
        btnSubnetScan.setForeground(new java.awt.Color(233, 247, 249));
        btnSubnetScan.setText("Subnet Scan");
        btnSubnetScan.setBorder(null);
        btnSubnetScan.setBorderPainted(false);
        btnSubnetScan.setCursor(new java.awt.Cursor(java.awt.Cursor.HAND_CURSOR));
        btnSubnetScan.setFocusPainted(false);
        btnSubnetScan.setHorizontalAlignment(javax.swing.SwingConstants.LEADING);
        btnSubnetScan.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                btnSubnetScanActionPerformed(evt);
            }
        });

        lblNetworking.setFont(new java.awt.Font("Dubai Medium", 1, 18)); // NOI18N
        lblNetworking.setForeground(new java.awt.Color(233, 247, 249));
        lblNetworking.setText("Networking");
        lblNetworking.setFocusable(false);

        btnDevicePing.setBackground(new java.awt.Color(45, 45, 45));
        btnDevicePing.setFont(new java.awt.Font("Dubai Medium", 0, 16)); // NOI18N
        btnDevicePing.setForeground(new java.awt.Color(233, 247, 249));
        btnDevicePing.setText("Device Ping");
        btnDevicePing.setBorder(null);
        btnDevicePing.setBorderPainted(false);
        btnDevicePing.setCursor(new java.awt.Cursor(java.awt.Cursor.HAND_CURSOR));
        btnDevicePing.setFocusPainted(false);
        btnDevicePing.setHorizontalAlignment(javax.swing.SwingConstants.LEADING);
        btnDevicePing.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                btnDevicePingActionPerformed(evt);
            }
        });

        btnPortScan.setBackground(new java.awt.Color(45, 45, 45));
        btnPortScan.setFont(new java.awt.Font("Dubai Medium", 0, 16)); // NOI18N
        btnPortScan.setForeground(new java.awt.Color(233, 247, 249));
        btnPortScan.setText("Port Scan");
        btnPortScan.setBorder(null);
        btnPortScan.setBorderPainted(false);
        btnPortScan.setCursor(new java.awt.Cursor(java.awt.Cursor.HAND_CURSOR));
        btnPortScan.setFocusPainted(false);
        btnPortScan.setHorizontalAlignment(javax.swing.SwingConstants.LEADING);
        btnPortScan.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                btnPortScanActionPerformed(evt);
            }
        });

        btnTCPMessage.setBackground(new java.awt.Color(45, 45, 45));
        btnTCPMessage.setFont(new java.awt.Font("Dubai Medium", 0, 16)); // NOI18N
        btnTCPMessage.setForeground(new java.awt.Color(233, 247, 249));
        btnTCPMessage.setText("TCP Message");
        btnTCPMessage.setBorder(null);
        btnTCPMessage.setBorderPainted(false);
        btnTCPMessage.setCursor(new java.awt.Cursor(java.awt.Cursor.HAND_CURSOR));
        btnTCPMessage.setFocusPainted(false);
        btnTCPMessage.setHorizontalAlignment(javax.swing.SwingConstants.LEADING);
        btnTCPMessage.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                btnTCPMessageActionPerformed(evt);
            }
        });

        btnImportResults.setBackground(new java.awt.Color(45, 45, 45));
        btnImportResults.setFont(new java.awt.Font("Dubai Medium", 0, 16)); // NOI18N
        btnImportResults.setForeground(new java.awt.Color(233, 247, 249));
        btnImportResults.setText("Import Results");
        btnImportResults.setBorder(null);
        btnImportResults.setBorderPainted(false);
        btnImportResults.setCursor(new java.awt.Cursor(java.awt.Cursor.HAND_CURSOR));
        btnImportResults.setFocusPainted(false);
        btnImportResults.setHorizontalAlignment(javax.swing.SwingConstants.LEADING);
        btnImportResults.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                btnImportResultsActionPerformed(evt);
            }
        });

        btnTCPMessageListen.setBackground(new java.awt.Color(45, 45, 45));
        btnTCPMessageListen.setFont(new java.awt.Font("Dubai", 0, 14)); // NOI18N
        btnTCPMessageListen.setForeground(new java.awt.Color(233, 247, 249));
        btnTCPMessageListen.setText("-  Listen");
        btnTCPMessageListen.setBorder(null);
        btnTCPMessageListen.setBorderPainted(false);
        btnTCPMessageListen.setCursor(new java.awt.Cursor(java.awt.Cursor.HAND_CURSOR));
        btnTCPMessageListen.setFocusPainted(false);
        btnTCPMessageListen.setHorizontalAlignment(javax.swing.SwingConstants.LEADING);
        btnTCPMessageListen.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                btnTCPMessageListenActionPerformed(evt);
            }
        });

        btnTCPMessageConnect.setBackground(new java.awt.Color(45, 45, 45));
        btnTCPMessageConnect.setFont(new java.awt.Font("Dubai", 0, 14)); // NOI18N
        btnTCPMessageConnect.setForeground(new java.awt.Color(233, 247, 249));
        btnTCPMessageConnect.setText("-  Connect");
        btnTCPMessageConnect.setBorder(null);
        btnTCPMessageConnect.setBorderPainted(false);
        btnTCPMessageConnect.setCursor(new java.awt.Cursor(java.awt.Cursor.HAND_CURSOR));
        btnTCPMessageConnect.setFocusPainted(false);
        btnTCPMessageConnect.setHorizontalAlignment(javax.swing.SwingConstants.LEADING);
        btnTCPMessageConnect.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                btnTCPMessageConnectActionPerformed(evt);
            }
        });
        javax.swing.GroupLayout pnlSideBarLayout = new javax.swing.GroupLayout(pnlSideBar);
        pnlSideBar.setLayout(pnlSideBarLayout);
        pnlSideBarLayout.setHorizontalGroup(
            pnlSideBarLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addGroup(pnlSideBarLayout.createSequentialGroup()

.addGroup(pnlSideBarLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                    .addGroup(pnlSideBarLayout.createSequentialGroup()
                        .addGap(5, 5, 5)
                        .addComponent(pnlLine, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE))
                    .addGroup(pnlSideBarLayout.createSequentialGroup()
                        .addGap(30, 30, 30)

.addGroup(pnlSideBarLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                            .addComponent(lblTitle, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)

.addGroup(pnlSideBarLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.TRAILING,
false)
                                .addComponent(btnSubnetScan,
javax.swing.GroupLayout.Alignment.LEADING, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
                                .addComponent(btnDevicePing,
javax.swing.GroupLayout.Alignment.LEADING, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
                                .addComponent(btnPortScan,
javax.swing.GroupLayout.Alignment.LEADING, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
                                .addComponent(btnTCPMessage,
javax.swing.GroupLayout.Alignment.LEADING, javax.swing.GroupLayout.DEFAULT_SIZE, 126,
Short.MAX_VALUE))))
                    .addGroup(pnlSideBarLayout.createSequentialGroup()
                        .addGap(10, 10, 10)
                        .addComponent(lblNetworking))

.addGroup(pnlSideBarLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.TRAILING)
                        .addGroup(pnlSideBarLayout.createSequentialGroup()
                            .addContainerGap()
                            .addComponent(btnImportResults,
javax.swing.GroupLayout.PREFERRED_SIZE, 126, javax.swing.GroupLayout.PREFERRED_SIZE))
                        .addGroup(javax.swing.GroupLayout.Alignment.LEADING,
pnlSideBarLayout.createSequentialGroup()
                            .addGap(46, 46, 46)

.addGroup(pnlSideBarLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                                .addComponent(btnTCPMessageConnect,
javax.swing.GroupLayout.PREFERRED_SIZE, 110, javax.swing.GroupLayout.PREFERRED_SIZE)
                                .addComponent(btnTCPMessageListen,
javax.swing.GroupLayout.PREFERRED_SIZE, 110, javax.swing.GroupLayout.PREFERRED_SIZE)))))
                .addContainerGap(javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE))
        );
        pnlSideBarLayout.setVerticalGroup(
            pnlSideBarLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addGroup(pnlSideBarLayout.createSequentialGroup()
                .addContainerGap()
                .addComponent(lblTitle, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
                .addGap(30, 30, 30)
                .addComponent(lblNetworking, javax.swing.GroupLayout.PREFERRED_SIZE, 20,
javax.swing.GroupLayout.PREFERRED_SIZE)
```

```
                .addGap(1, 1, 1)
                .addComponent(pnlLine, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
                .addGap(20, 20, 20)
                .addComponent(btnSubnetScan, javax.swing.GroupLayout.PREFERRED_SIZE, 30,
javax.swing.GroupLayout.PREFERRED_SIZE)
                .addGap(12, 12, 12)
                .addComponent(btnDevicePing, javax.swing.GroupLayout.PREFERRED_SIZE, 30,
javax.swing.GroupLayout.PREFERRED_SIZE)
                .addGap(12, 12, 12)
                .addComponent(btnPortScan, javax.swing.GroupLayout.PREFERRED_SIZE, 30,
javax.swing.GroupLayout.PREFERRED_SIZE)
                .addGap(12, 12, 12)
                .addComponent(btnTCPMessage, javax.swing.GroupLayout.PREFERRED_SIZE, 30,
javax.swing.GroupLayout.PREFERRED_SIZE)
                .addGap(0, 0, 0)
                .addComponent(btnTCPMessageListen)
                .addGap(2, 2, 2)
                .addComponent(btnTCPMessageConnect)
                .addGap(12, 12, 12)
                .addComponent(btnImportResults, javax.swing.GroupLayout.PREFERRED_SIZE, 30,
javax.swing.GroupLayout.PREFERRED_SIZE)
                .addContainerGap(javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE))
        );

        pnlMainPanel.setBackground(new java.awt.Color(49, 49, 49));
        pnlMainPanel.setPreferredSize(new java.awt.Dimension(1000, 720));
        pnlMainPanel.setLayout(new java.awt.CardLayout());

        lblNoTabOpen.setFont(new java.awt.Font("Dubai Medium", 0, 36)); // NOI18N
        lblNoTabOpen.setForeground(new java.awt.Color(70, 70, 70));
        lblNoTabOpen.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);
        lblNoTabOpen.setText("No tab open");
        lblNoTabOpen.setPreferredSize(new java.awt.Dimension(270, 82));
        pnlMainPanel.add(lblNoTabOpen, "card3");

        pnlSubnetScan.setBackground(new java.awt.Color(49, 49, 49));

        lblSubnetScan.setFont(new java.awt.Font("Dubai Medium", 1, 24)); // NOI18N
        lblSubnetScan.setForeground(new java.awt.Color(233, 247, 249));
        lblSubnetScan.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);
        lblSubnetScan.setText("Subnet Scan");

        lblNetworkRange.setFont(new java.awt.Font("Dubai Medium", 0, 14)); // NOI18N
        lblNetworkRange.setForeground(new java.awt.Color(233, 247, 249));
        lblNetworkRange.setText("Enter the network range:");

        txfNetworkRange.setBackground(new java.awt.Color(255, 255, 255));
        txfNetworkRange.setFont(new java.awt.Font("Dubai", 0, 12)); // NOI18N
        txfNetworkRange.setForeground(new java.awt.Color(45, 45, 45));
        txfNetworkRange.setText(DEFAULT_NETWORK_RANGE_TEXT);
        txfNetworkRange.setToolTipText("Range of devices to scan.");
        txfNetworkRange.setBorder(javax.swing.BorderFactory.createEmptyBorder(1, 5, 1, 1));
        txfNetworkRange.addFocusListener(new java.awt.event.FocusAdapter() {
            public void focusGained(java.awt.event.FocusEvent evt) {
                txfNetworkRangeFocusGained(evt);
            }
        });

        btnStartSubnetScan.setBackground(new java.awt.Color(45, 45, 45));
        btnStartSubnetScan.setFont(new java.awt.Font("Dubai", 0, 12)); // NOI18N
        btnStartSubnetScan.setForeground(new java.awt.Color(233, 247, 249));
        btnStartSubnetScan.setText("Start Subnet Scan");

btnStartSubnetScan.setBorder(javax.swing.BorderFactory.createBevelBorder(javax.swing.border.B
evelBorder.RAISED));
        btnStartSubnetScan.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                btnStartSubnetScanActionPerformed(evt);
            }
        });

        tblSubnetScan.setBackground(new java.awt.Color(255, 255, 255));
        tblSubnetScan.setForeground(new java.awt.Color(45, 45, 45));
        tblSubnetScan.setModel(new javax.swing.table.DefaultTableModel(
            new Object [][] {

            },
            new String [] {
                "IP Address"
            }
        ) {
            boolean[] canEdit = new boolean [] {
                false
            };

            public boolean isCellEditable(int rowIndex, int columnIndex) {
                return canEdit [columnIndex];
            }
        });
        tblSubnetScan.setRowMargin(4);
        tblSubnetScan.setSelectionForeground(new java.awt.Color(233, 247, 249));
        tblSubnetScan.getTableHeader().setReorderingAllowed(false);
        scrSubnetScan.setViewportView(tblSubnetScan);

        btnExportResultsSubnetScan.setBackground(new java.awt.Color(45, 45, 45));
        btnExportResultsSubnetScan.setFont(new java.awt.Font("Dubai", 0, 12)); // NOI18N
        btnExportResultsSubnetScan.setForeground(new java.awt.Color(233, 247, 249));
        btnExportResultsSubnetScan.setText("Export Results");
        btnExportResultsSubnetScan.setToolTipText("Exports the results to a JSON file.");

btnExportResultsSubnetScan.setBorder(javax.swing.BorderFactory.createBevelBorder(javax.swing.
border.BevelBorder.RAISED));
        btnExportResultsSubnetScan.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                btnExportResultsSubnetScanActionPerformed(evt);
            }
        });

        lblSubnetScanInProgress.setFont(new java.awt.Font("Dubai Medium", 0, 14)); // NOI18N
        lblSubnetScanInProgress.setForeground(new java.awt.Color(0, 204, 0));

        lblTimeoutSubnetScan.setFont(new java.awt.Font("Dubai Medium", 0, 14)); // NOI18N
        lblTimeoutSubnetScan.setForeground(new java.awt.Color(233, 247, 249));
        lblTimeoutSubnetScan.setText("Timeout after (ms):");

        spnTimeoutSubnetScan.setFont(new java.awt.Font("Dubai", 0, 12)); // NOI18N
        spnTimeoutSubnetScan.setModel(new javax.swing.SpinnerNumberModel(500, 100, 10000,
1));
        spnTimeoutSubnetScan.setToolTipText("How long the scan waits for a response.");
        spnTimeoutSubnetScan.setBorder(javax.swing.BorderFactory.createEmptyBorder(1, 1, 1,
1));
        spnTimeoutSubnetScan.setPreferredSize(new java.awt.Dimension(104, 23));
        lblNetworkRangeError.setFont(new java.awt.Font("Dubai Medium", 0, 14)); // NOI18N
        lblNetworkRangeError.setForeground(new java.awt.Color(255, 51, 0));

        prgSubnetScan.setBackground(new java.awt.Color(255, 255, 255));
        prgSubnetScan.setFont(new java.awt.Font("Dubai", 0, 10)); // NOI18N
        prgSubnetScan.setForeground(new java.awt.Color(0, 204, 0));
        prgSubnetScan.setName(""); // NOI18N
        prgSubnetScan.setStringPainted(true);

        javax.swing.GroupLayout pnlSubnetScanLayout = new
javax.swing.GroupLayout(pnlSubnetScan);
        pnlSubnetScan.setLayout(pnlSubnetScanLayout);
        pnlSubnetScanLayout.setHorizontalGroup(

pnlSubnetScanLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addComponent(lblSubnetScan, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
            .addGroup(javax.swing.GroupLayout.Alignment.TRAILING,
pnlSubnetScanLayout.createSequentialGroup()
                .addGap(30, 30, 30)

.addGroup(pnlSubnetScanLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.TRAILING)
                    .addComponent(prgSubnetScan, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
                    .addGroup(pnlSubnetScanLayout.createSequentialGroup()
                        .addGap(0, 0, Short.MAX_VALUE)
                        .addComponent(btnExportResultsSubnetScan,
javax.swing.GroupLayout.PREFERRED_SIZE, 104, javax.swing.GroupLayout.PREFERRED_SIZE))
                    .addComponent(scrSubnetScan)
                    .addGroup(javax.swing.GroupLayout.Alignment.LEADING,
pnlSubnetScanLayout.createSequentialGroup()

.addGroup(pnlSubnetScanLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                            .addComponent(lblNetworkRange)
                            .addComponent(lblTimeoutSubnetScan))
                        .addGap(18, 18, 18)

.addGroup(pnlSubnetScanLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                            .addGroup(pnlSubnetScanLayout.createSequentialGroup()
                                .addComponent(btnStartSubnetScan,
javax.swing.GroupLayout.PREFERRED_SIZE, 140, javax.swing.GroupLayout.PREFERRED_SIZE)
                                .addGap(18, 18, 18)
                                .addComponent(lblSubnetScanInProgress,
javax.swing.GroupLayout.PREFERRED_SIZE, 614, javax.swing.GroupLayout.PREFERRED_SIZE)
                                .addGap(0, 0, Short.MAX_VALUE))
                            .addGroup(pnlSubnetScanLayout.createSequentialGroup()

.addGroup(pnlSubnetScanLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING,
false)
                                    .addComponent(txfNetworkRange,
javax.swing.GroupLayout.DEFAULT_SIZE, 140, Short.MAX_VALUE)
                                    .addComponent(spnTimeoutSubnetScan,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE))
                                .addGap(18, 18, 18)
                                .addComponent(lblNetworkRangeError,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
Short.MAX_VALUE)))))
                .addGap(30, 30, 30))
        );
        pnlSubnetScanLayout.setVerticalGroup(

pnlSubnetScanLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
```

```
            .addGroup(pnlSubnetScanLayout.createSequentialGroup()
                .addGap(36, 36, 36)
                .addComponent(lblSubnetScan)
                .addGap(18, 18, 18)

.addGroup(pnlSubnetScanLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)

.addGroup(pnlSubnetScanLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELINE)
                        .addComponent(lblNetworkRange)
                        .addComponent(txfNetworkRange,
javax.swing.GroupLayout.PREFERRED_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE))
                    .addComponent(lblNetworkRangeError,
javax.swing.GroupLayout.PREFERRED_SIZE, 24, javax.swing.GroupLayout.PREFERRED_SIZE))
                .addGap(18, 18, 18)

.addGroup(pnlSubnetScanLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                    .addComponent(lblTimeoutSubnetScan)
                    .addComponent(spnTimeoutSubnetScan,
javax.swing.GroupLayout.PREFERRED_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE))
                .addGap(18, 18, 18)

.addGroup(pnlSubnetScanLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                    .addComponent(lblSubnetScanInProgress,
javax.swing.GroupLayout.PREFERRED_SIZE, 24, javax.swing.GroupLayout.PREFERRED_SIZE)
                    .addGroup(pnlSubnetScanLayout.createSequentialGroup()
                        .addComponent(btnStartSubnetScan)
                        .addGap(30, 30, 30)
                        .addComponent(scrSubnetScan, javax.swing.GroupLayout.PREFERRED_SIZE,
349, javax.swing.GroupLayout.PREFERRED_SIZE)))
                .addGap(18, 18, 18)
                .addComponent(prgSubnetScan, javax.swing.GroupLayout.PREFERRED_SIZE, 18,
javax.swing.GroupLayout.PREFERRED_SIZE)
                .addGap(30, 30, 30)
                .addComponent(btnExportResultsSubnetScan,
javax.swing.GroupLayout.PREFERRED_SIZE, 25, javax.swing.GroupLayout.PREFERRED_SIZE)
                .addContainerGap(45, Short.MAX_VALUE))
        );

        pnlMainPanel.add(pnlSubnetScan, "cardSubnetScan");

        pnlPortScan.setBackground(new java.awt.Color(49, 49, 49));
        pnlPortScan.setMaximumSize(new java.awt.Dimension(1000, 720));
        pnlPortScan.setMinimumSize(new java.awt.Dimension(1000, 720));

        lblPortScan.setFont(new java.awt.Font("Dubai Medium", 1, 24)); // NOI18N
        lblPortScan.setForeground(new java.awt.Color(233, 247, 249));
        lblPortScan.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);
        lblPortScan.setText("Port Scan");
        lblPortScan.setMaximumSize(new java.awt.Dimension(1000, 42));
        lblPortScan.setMinimumSize(new java.awt.Dimension(1000, 42));
        lblPortScan.setPreferredSize(new java.awt.Dimension(1000, 42));

        txfIPAddressPortScan.setBackground(new java.awt.Color(255, 255, 255));
        txfIPAddressPortScan.setFont(new java.awt.Font("Dubai", 0, 12)); // NOI18N
        txfIPAddressPortScan.setForeground(new java.awt.Color(45, 45, 45));
        txfIPAddressPortScan.setText(DEFAULT_IP_ADDRESS_TEXT);
        txfIPAddressPortScan.setToolTipText("The device to scan.");
        txfIPAddressPortScan.setBorder(javax.swing.BorderFactory.createEmptyBorder(1, 5, 1,
1));
        txfIPAddressPortScan.addFocusListener(new java.awt.event.FocusAdapter() {
            public void focusGained(java.awt.event.FocusEvent evt) {
                txfIPAddressPortScanFocusGained(evt);
            }
        });

        btnStartPortScan.setBackground(new java.awt.Color(45, 45, 45));
        btnStartPortScan.setFont(new java.awt.Font("Dubai", 0, 12)); // NOI18N
        btnStartPortScan.setForeground(new java.awt.Color(233, 247, 249));
        btnStartPortScan.setText("Start Port Scan");
        btnStartPortScan.setToolTipText("");

btnStartPortScan.setBorder(javax.swing.BorderFactory.createBevelBorder(javax.swing.border.Bev
elBorder.RAISED));
        btnStartPortScan.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                btnStartPortScanActionPerformed(evt);
            }
        });

        spnBottomRangePort.setFont(new java.awt.Font("Dubai", 0, 12)); // NOI18N
        spnBottomRangePort.setModel(new javax.swing.SpinnerNumberModel(1, 1, 65535, 1));
        spnBottomRangePort.setToolTipText("The port to start scanning from.");
        spnBottomRangePort.setBorder(javax.swing.BorderFactory.createEmptyBorder(1, 1, 1,
1));
        spnBottomRangePort.setPreferredSize(new java.awt.Dimension(104, 23));

        spnTopRangePort.setFont(new java.awt.Font("Dubai", 0, 12)); // NOI18N
        spnTopRangePort.setModel(new javax.swing.SpinnerNumberModel(1023, 1, 65535, 1));
        spnTopRangePort.setToolTipText("The port to scan until.");
        spnTopRangePort.setBorder(javax.swing.BorderFactory.createEmptyBorder(1, 1, 1, 1));
        spnTopRangePort.setPreferredSize(new java.awt.Dimension(104, 23));

        lblIPAdressPortScan.setFont(new java.awt.Font("Dubai Medium", 0, 14)); // NOI18N
        lblIPAdressPortScan.setForeground(new java.awt.Color(233, 247, 249));
        lblIPAdressPortScan.setText("Enter the IP address:");

        lblDash.setFont(new java.awt.Font("Dubai Medium", 0, 14)); // NOI18N
        lblDash.setForeground(new java.awt.Color(233, 247, 249));
        lblDash.setText("-");

        lblPortRange.setFont(new java.awt.Font("Dubai Medium", 0, 14)); // NOI18N
        lblPortRange.setForeground(new java.awt.Color(233, 247, 249));
        lblPortRange.setText("Enter the range of ports:");

        tblPortScan.setBackground(new java.awt.Color(255, 255, 255));
        tblPortScan.setForeground(new java.awt.Color(45, 45, 45));
        tblPortScan.setModel(new javax.swing.table.DefaultTableModel(
            new Object [][] {

            },
            new String [] {
                "Port No.", "Port Protocol/Service"
            }
        ) {
            boolean[] canEdit = new boolean [] {
                false, false
            };

            public boolean isCellEditable(int rowIndex, int columnIndex) {
                return canEdit [columnIndex];
            }
        });
        tblPortScan.getTableHeader().setReorderingAllowed(false);
        scrPortScan.setViewportView(tblPortScan);
        if (tblPortScan.getColumnModel().getColumnCount() > 0) {
            tblPortScan.getColumnModel().getColumn(0).setResizable(false);
            tblPortScan.getColumnModel().getColumn(1).setResizable(false);
        }

        btnExportResultsPortScan.setBackground(new java.awt.Color(45, 45, 45));
        btnExportResultsPortScan.setFont(new java.awt.Font("Dubai", 0, 12)); // NOI18N
        btnExportResultsPortScan.setForeground(new java.awt.Color(233, 247, 249));
        btnExportResultsPortScan.setText("Export Results");
        btnExportResultsPortScan.setToolTipText("Exports the results to a JSON file.");

btnExportResultsPortScan.setBorder(javax.swing.BorderFactory.createBevelBorder(javax.swing.bo
rder.BevelBorder.RAISED));
        btnExportResultsPortScan.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                btnExportResultsPortScanActionPerformed(evt);
            }
        });

        lblPortRangeError.setFont(new java.awt.Font("Dubai Medium", 0, 14)); // NOI18N
        lblPortRangeError.setForeground(new java.awt.Color(255, 51, 0));

        lblIPAddressErrorPortScan.setFont(new java.awt.Font("Dubai Medium", 0, 14)); //
NOI18N
        lblIPAddressErrorPortScan.setForeground(new java.awt.Color(255, 51, 0));

        lblPortScanInProgress.setFont(new java.awt.Font("Dubai Medium", 0, 14)); // NOI18N
        lblPortScanInProgress.setForeground(new java.awt.Color(0, 204, 0));

        lblTimeoutPortScan.setFont(new java.awt.Font("Dubai Medium", 0, 14)); // NOI18N
        lblTimeoutPortScan.setForeground(new java.awt.Color(233, 247, 249));
        lblTimeoutPortScan.setText("Timeout after (ms):");

        spnTimeoutPortScan.setFont(new java.awt.Font("Dubai", 0, 12)); // NOI18N
        spnTimeoutPortScan.setModel(new javax.swing.SpinnerNumberModel(500, 100, 10000, 1));
        spnTimeoutPortScan.setToolTipText("How long the scan waits for a response.");
        spnTimeoutPortScan.setBorder(javax.swing.BorderFactory.createEmptyBorder(1, 1, 1,
1));
        spnTimeoutPortScan.setPreferredSize(new java.awt.Dimension(104, 23));

        prgPortScan.setBackground(new java.awt.Color(255, 255, 255));
        prgPortScan.setFont(new java.awt.Font("Dubai", 0, 10)); // NOI18N
        prgPortScan.setForeground(new java.awt.Color(0, 204, 0));
        prgPortScan.setName(""); // NOI18N
        prgPortScan.setStringPainted(true);

        javax.swing.GroupLayout pnlPortScanLayout = new javax.swing.GroupLayout(pnlPortScan);
        pnlPortScan.setLayout(pnlPortScanLayout);
        pnlPortScanLayout.setHorizontalGroup(
            pnlPortScanLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addGroup(pnlPortScanLayout.createSequentialGroup()
                .addComponent(lblPortScan, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
                .addGap(0, 0, Short.MAX_VALUE))
            .addGroup(javax.swing.GroupLayout.Alignment.TRAILING,
pnlPortScanLayout.createSequentialGroup()

.addGroup(pnlPortScanLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.TRAILING)
                    .addGroup(pnlPortScanLayout.createSequentialGroup()
                        .addContainerGap(javax.swing.GroupLayout.DEFAULT_SIZE,
Short.MAX_VALUE)
                        .addComponent(btnExportResultsPortScan,
javax.swing.GroupLayout.PREFERRED_SIZE, 104, javax.swing.GroupLayout.PREFERRED_SIZE))
                    .addGroup(javax.swing.GroupLayout.Alignment.LEADING,
pnlPortScanLayout.createSequentialGroup()
                        .addGap(30, 30, 30)

.addGroup(pnlPortScanLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                            .addComponent(prgPortScan,
javax.swing.GroupLayout.Alignment.TRAILING, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
                            .addGroup(pnlPortScanLayout.createSequentialGroup()

.addGroup(pnlPortScanLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                                    .addComponent(lblPortRange)
                                    .addComponent(lblIPAdressPortScan)
                                    .addComponent(lblTimeoutPortScan))
                                .addGap(18, 18, 18)

.addGroup(pnlPortScanLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING,
false)
                                    .addGroup(pnlPortScanLayout.createSequentialGroup()
                                        .addComponent(spnBottomRangePort,
javax.swing.GroupLayout.PREFERRED_SIZE, 70, javax.swing.GroupLayout.PREFERRED_SIZE)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
                                        .addComponent(lblDash)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
                                        .addComponent(spnTopRangePort,
javax.swing.GroupLayout.PREFERRED_SIZE, 70, javax.swing.GroupLayout.PREFERRED_SIZE))
                                    .addComponent(btnStartPortScan,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
                                    .addComponent(txfIPAddressPortScan)
                                    .addComponent(spnTimeoutPortScan,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE))
                                .addGap(18, 18, 18)

.addGroup(pnlPortScanLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                                    .addComponent(lblIPAddressErrorPortScan,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
                                    .addComponent(lblPortRangeError,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
                                    .addComponent(lblPortScanInProgress,
javax.swing.GroupLayout.Alignment.TRAILING, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)))
                            .addComponent(scrPortScan,
javax.swing.GroupLayout.Alignment.TRAILING))))
                .addGap(30, 30, 30))
        );
        pnlPortScanLayout.setVerticalGroup(
            pnlPortScanLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addGroup(pnlPortScanLayout.createSequentialGroup()
                .addGap(36, 36, 36)
                .addComponent(lblPortScan, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
                .addGap(18, 18, 18)

.addGroup(pnlPortScanLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.TRAILING)

.addGroup(pnlPortScanLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELINE)
                        .addComponent(txfIPAddressPortScan,
javax.swing.GroupLayout.PREFERRED_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE)
                         .addComponent(lblIPAdressPortScan))
                     .addComponent(lblIPAddressErrorPortScan,
javax.swing.GroupLayout.PREFERRED_SIZE, 24, javax.swing.GroupLayout.PREFERRED_SIZE))
                 .addGap(18, 18, 18)

.addGroup(pnlPortScanLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.TRAILING)

.addGroup(pnlPortScanLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELINE)
                         .addComponent(spnBottomRangePort,
javax.swing.GroupLayout.PREFERRED_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE)
                         .addComponent(spnTopRangePort,
javax.swing.GroupLayout.PREFERRED_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE)
                         .addComponent(lblDash)
                         .addComponent(lblPortRange))
                     .addComponent(lblPortRangeError, javax.swing.GroupLayout.PREFERRED_SIZE,
24, javax.swing.GroupLayout.PREFERRED_SIZE))
                 .addGap(19, 19, 19)

.addGroup(pnlPortScanLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELINE)
                     .addComponent(lblTimeoutPortScan)
                     .addComponent(spnTimeoutPortScan, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE))
                 .addGap(18, 18, 18)

.addGroup(pnlPortScanLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.TRAILING)
                     .addComponent(btnStartPortScan)
                     .addComponent(lblPortScanInProgress,
javax.swing.GroupLayout.PREFERRED_SIZE, 25, javax.swing.GroupLayout.PREFERRED_SIZE))
                 .addGap(30, 30, 30)
                 .addComponent(scrPortScan, javax.swing.GroupLayout.PREFERRED_SIZE, 306,
javax.swing.GroupLayout.PREFERRED_SIZE)
                 .addGap(18, 18, 18)
                 .addComponent(prgPortScan, javax.swing.GroupLayout.PREFERRED_SIZE, 18,
javax.swing.GroupLayout.PREFERRED_SIZE)
                 .addGap(30, 30, 30)
                 .addComponent(btnExportResultsPortScan)
                 .addContainerGap(45, Short.MAX_VALUE))
        );

        pnlMainPanel.add(pnlPortScan, "cardPortScan");

        pnlTCPMessageListen.setBackground(new java.awt.Color(49, 49, 49));

        lblTCPMessageListen.setFont(new java.awt.Font("Dubai Medium", 1, 24)); // NOI18N
        lblTCPMessageListen.setForeground(new java.awt.Color(233, 247, 249));
        lblTCPMessageListen.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);
        lblTCPMessageListen.setText("TCP Message - Listen");

        lblPortTCPMessageListen.setFont(new java.awt.Font("Dubai Medium", 0, 14)); // NOI18N
        lblPortTCPMessageListen.setForeground(new java.awt.Color(233, 247, 249));
        lblPortTCPMessageListen.setText("Enter the port to listen on:");

        spnPortTCPMessageListen.setFont(new java.awt.Font("Dubai", 0, 12)); // NOI18N
        spnPortTCPMessageListen.setModel(new javax.swing.SpinnerNumberModel(1234, 1, 65535,
1));
        spnPortTCPMessageListen.setToolTipText("The port to open the connection on.");
        spnPortTCPMessageListen.setBorder(javax.swing.BorderFactory.createEmptyBorder(1, 1,
1, 1));
```

```
        spnPortTCPMessageListen.setPreferredSize(new java.awt.Dimension(104, 23));

        btnStartTCPMessageListen.setBackground(new java.awt.Color(45, 45, 45));
        btnStartTCPMessageListen.setFont(new java.awt.Font("Dubai", 0, 12)); // NOI18N
        btnStartTCPMessageListen.setForeground(new java.awt.Color(233, 247, 249));
        btnStartTCPMessageListen.setText("Start TCP Listen");

btnStartTCPMessageListen.setBorder(javax.swing.BorderFactory.createBevelBorder(javax.swing.bo
rder.BevelBorder.RAISED));
        btnStartTCPMessageListen.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                btnStartTCPMessageListenActionPerformed(evt);
            }
        });

        lblTCPMessageListenInProgress.setFont(new java.awt.Font("Dubai Medium", 0, 14)); //
NOI18N
        lblTCPMessageListenInProgress.setForeground(new java.awt.Color(0, 204, 0));

        btnExportResultsTCPMessageListen.setBackground(new java.awt.Color(45, 45, 45));
        btnExportResultsTCPMessageListen.setFont(new java.awt.Font("Dubai", 0, 12)); //
NOI18N
        btnExportResultsTCPMessageListen.setForeground(new java.awt.Color(233, 247, 249));
        btnExportResultsTCPMessageListen.setText("Export Results");
        btnExportResultsTCPMessageListen.setToolTipText("Exports the results to a text
file.");

btnExportResultsTCPMessageListen.setBorder(javax.swing.BorderFactory.createBevelBorder(javax.
swing.border.BevelBorder.RAISED));
        btnExportResultsTCPMessageListen.addActionListener(new
java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                btnExportResultsTCPMessageListenActionPerformed(evt);
            }
        });

        btnSendTCPMessageListen.setBackground(new java.awt.Color(45, 45, 45));
        btnSendTCPMessageListen.setFont(new java.awt.Font("Dubai", 0, 12)); // NOI18N
        btnSendTCPMessageListen.setForeground(new java.awt.Color(233, 247, 249));
        btnSendTCPMessageListen.setText("Send");

btnSendTCPMessageListen.setBorder(javax.swing.BorderFactory.createBevelBorder(javax.swing.bor
der.BevelBorder.RAISED));
        btnSendTCPMessageListen.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                btnSendTCPMessageListenActionPerformed(evt);
            }
        });

        lblEnterMessageTCPMessageListen.setFont(new java.awt.Font("Dubai Medium", 0, 14)); //
NOI18N
        lblEnterMessageTCPMessageListen.setForeground(new java.awt.Color(233, 247, 249));
        lblEnterMessageTCPMessageListen.setText("Enter message:");

        txpTCPMessageListen.setEditable(false);
        txpTCPMessageListen.setBackground(new java.awt.Color(255, 255, 255));
        txpTCPMessageListen.setFont(new java.awt.Font("Dubai Medium", 0, 14)); // NOI18N
        txpTCPMessageListen.setForeground(new java.awt.Color(45, 45, 45));
        txpTCPMessageListen.setToolTipText("Messages will appear here");
        scrTCPMessageListen.setViewportView(txpTCPMessageListen);

        txfMessageTCPMessageListen.setBackground(new java.awt.Color(255, 255, 255));
        txfMessageTCPMessageListen.setFont(new java.awt.Font("Dubai Medium", 0, 14)); //
NOI18N
        txfMessageTCPMessageListen.setForeground(new java.awt.Color(45, 45, 45));
        txfMessageTCPMessageListen.setToolTipText("Write the message here.");

        javax.swing.GroupLayout pnlTCPMessageListenLayout = new
javax.swing.GroupLayout(pnlTCPMessageListen);
        pnlTCPMessageListen.setLayout(pnlTCPMessageListenLayout);
        pnlTCPMessageListenLayout.setHorizontalGroup(

pnlTCPMessageListenLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addComponent(lblTCPMessageListen, javax.swing.GroupLayout.DEFAULT_SIZE, 1000,
Short.MAX_VALUE)
            .addGroup(javax.swing.GroupLayout.Alignment.TRAILING,
pnlTCPMessageListenLayout.createSequentialGroup()

.addGroup(pnlTCPMessageListenLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.TRA
ILING)
                    .addGroup(pnlTCPMessageListenLayout.createSequentialGroup()
                        .addContainerGap(javax.swing.GroupLayout.DEFAULT_SIZE,
Short.MAX_VALUE)
                        .addComponent(btnExportResultsTCPMessageListen,
javax.swing.GroupLayout.PREFERRED_SIZE, 104, javax.swing.GroupLayout.PREFERRED_SIZE))
                    .addGroup(javax.swing.GroupLayout.Alignment.LEADING,
pnlTCPMessageListenLayout.createSequentialGroup()
                        .addGap(30, 30, 30)

.addGroup(pnlTCPMessageListenLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEA
DING)
                            .addGroup(javax.swing.GroupLayout.Alignment.TRAILING,
pnlTCPMessageListenLayout.createSequentialGroup()
                                .addComponent(lblPortTCPMessageListen)
                                .addGap(18, 18, 18)

.addGroup(pnlTCPMessageListenLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEA
DING, false)
                                    .addComponent(btnStartTCPMessageListen,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
                                    .addComponent(spnPortTCPMessageListen,
javax.swing.GroupLayout.DEFAULT_SIZE, 120, Short.MAX_VALUE))
                                .addGap(18, 18, 18)
                                .addComponent(lblTCPMessageListenInProgress,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE))
                            .addGroup(javax.swing.GroupLayout.Alignment.TRAILING,
pnlTCPMessageListenLayout.createSequentialGroup()
                                .addComponent(lblEnterMessageTCPMessageListen)
                                .addGap(18, 18, 18)
                                .addComponent(txfMessageTCPMessageListen)
                                .addGap(18, 18, 18)
                                .addComponent(btnSendTCPMessageListen,
javax.swing.GroupLayout.PREFERRED_SIZE, 69, javax.swing.GroupLayout.PREFERRED_SIZE))
                            .addComponent(scrTCPMessageListen))))
                .addGap(30, 30, 30))
        );
        pnlTCPMessageListenLayout.setVerticalGroup(

pnlTCPMessageListenLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addGroup(pnlTCPMessageListenLayout.createSequentialGroup()
                .addGap(36, 36, 36)
                .addComponent(lblTCPMessageListen)
                .addGap(18, 18, 18)
.addGroup(pnlTCPMessageListenLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.BAS
ELINE)
                    .addComponent(lblPortTCPMessageListen)
                    .addComponent(spnPortTCPMessageListen,
javax.swing.GroupLayout.PREFERRED_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE))
                .addGap(18, 18, 18)

.addGroup(pnlTCPMessageListenLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEA
DING)
                    .addComponent(btnStartTCPMessageListen)
                    .addComponent(lblTCPMessageListenInProgress,
javax.swing.GroupLayout.PREFERRED_SIZE, 24, javax.swing.GroupLayout.PREFERRED_SIZE))
                .addGap(30, 30, 30)
                .addComponent(scrTCPMessageListen, javax.swing.GroupLayout.DEFAULT_SIZE, 384,
Short.MAX_VALUE)
                .addGap(18, 18, 18)

.addGroup(pnlTCPMessageListenLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEA
DING, false)
                    .addComponent(btnSendTCPMessageListen,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
                    .addComponent(lblEnterMessageTCPMessageListen,
javax.swing.GroupLayout.Alignment.TRAILING, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
                    .addComponent(txfMessageTCPMessageListen,
javax.swing.GroupLayout.PREFERRED_SIZE, 0, Short.MAX_VALUE))
                .addGap(30, 30, 30)
                .addComponent(btnExportResultsTCPMessageListen)
                .addGap(45, 45, 45))
        );

        pnlMainPanel.add(pnlTCPMessageListen, "cardTCPMessageListen");

        pnlDevicePing.setBackground(new java.awt.Color(49, 49, 49));

        lblDevicePing.setFont(new java.awt.Font("Dubai Medium", 1, 24)); // NOI18N
        lblDevicePing.setForeground(new java.awt.Color(233, 247, 249));
        lblDevicePing.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);
        lblDevicePing.setText("Device Ping");

        lblIPAddressDevicePing.setFont(new java.awt.Font("Dubai Medium", 0, 14)); // NOI18N
        lblIPAddressDevicePing.setForeground(new java.awt.Color(233, 247, 249));
        lblIPAddressDevicePing.setText("Enter the IP address:");

        lblPingInterval.setFont(new java.awt.Font("Dubai Medium", 0, 14)); // NOI18N
        lblPingInterval.setForeground(new java.awt.Color(233, 247, 249));
        lblPingInterval.setText("Enter the ping interval (ms):");

        spnPingInterval.setFont(new java.awt.Font("Dubai", 0, 12)); // NOI18N
        spnPingInterval.setModel(new javax.swing.SpinnerNumberModel(100, 100, 10000, 1));
        spnPingInterval.setToolTipText("How often to send each ping.");
        spnPingInterval.setBorder(javax.swing.BorderFactory.createEmptyBorder(1, 1, 1, 1));
        spnPingInterval.setPreferredSize(new java.awt.Dimension(104, 23));

        txfIPAddressDevicePing.setBackground(new java.awt.Color(255, 255, 255));
        txfIPAddressDevicePing.setFont(new java.awt.Font("Dubai", 0, 12)); // NOI18N
        txfIPAddressDevicePing.setForeground(new java.awt.Color(45, 45, 45));
        txfIPAddressDevicePing.setText(DEFAULT_IP_ADDRESS_TEXT);
        txfIPAddressDevicePing.setToolTipText("The device to ping.");
        txfIPAddressDevicePing.setBorder(javax.swing.BorderFactory.createEmptyBorder(1, 5, 1,
1));
```

```
        txfIPAddressDevicePing.setPreferredSize(new java.awt.Dimension(104, 23));
        txfIPAddressDevicePing.addFocusListener(new java.awt.event.FocusAdapter() {
            public void focusGained(java.awt.event.FocusEvent evt) {
                txfIPAddressDevicePingFocusGained(evt);
            }
        });

        lblContinuousPinging.setFont(new java.awt.Font("Dubai Medium", 0, 14)); // NOI18N
        lblContinuousPinging.setForeground(new java.awt.Color(233, 247, 249));
        lblContinuousPinging.setText("Continuous pinging:");

        chkContinuousPinging.setBackground(new java.awt.Color(49, 49, 49));
        chkContinuousPinging.setForeground(new java.awt.Color(63, 63, 63));
        chkContinuousPinging.setToolTipText("Pings until manually stopped.");
        chkContinuousPinging.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                chkContinuousPingingActionPerformed(evt);
            }
        });

        lblNumberOfPings.setFont(new java.awt.Font("Dubai Medium", 0, 14)); // NOI18N
        lblNumberOfPings.setForeground(new java.awt.Color(233, 247, 249));
        lblNumberOfPings.setText("Enter the number of pings:");

        spnNumberOfPings.setFont(new java.awt.Font("Dubai", 0, 12)); // NOI18N
        spnNumberOfPings.setModel(new javax.swing.SpinnerNumberModel(10, 1, 100, 1));
        spnNumberOfPings.setToolTipText("How many pings to send.");
        spnNumberOfPings.setBorder(javax.swing.BorderFactory.createEmptyBorder(1, 1, 1, 1));
        spnNumberOfPings.setPreferredSize(new java.awt.Dimension(104, 23));

        btnStartDevicePing.setBackground(new java.awt.Color(45, 45, 45));
        btnStartDevicePing.setFont(new java.awt.Font("Dubai", 0, 12)); // NOI18N
        btnStartDevicePing.setForeground(new java.awt.Color(233, 247, 249));
        btnStartDevicePing.setText("Start Device Ping");

btnStartDevicePing.setBorder(javax.swing.BorderFactory.createBevelBorder(javax.swing.border.B
evelBorder.RAISED));
        btnStartDevicePing.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                btnStartDevicePingActionPerformed(evt);
            }
        });

        tblDevicePing.setBackground(new java.awt.Color(255, 255, 255));
        tblDevicePing.setForeground(new java.awt.Color(45, 45, 45));
        tblDevicePing.setModel(new javax.swing.table.DefaultTableModel(
            new Object [][] {

            },
            new String [] {
                "IP Address", "Round Trip Time (ms)", "Reachable", "Packet Loss (%)"
            }
        ) {
            boolean[] canEdit = new boolean [] {
                false, false, false, false
            };

            public boolean isCellEditable(int rowIndex, int columnIndex) {
                return canEdit [columnIndex];
            }
        });
        tblDevicePing.getTableHeader().setReorderingAllowed(false);
        scrDevicePing.setViewportView(tblDevicePing);
        if (tblDevicePing.getColumnModel().getColumnCount() > 0) {
            tblDevicePing.getColumnModel().getColumn(0).setResizable(false);
            tblDevicePing.getColumnModel().getColumn(1).setResizable(false);
            tblDevicePing.getColumnModel().getColumn(2).setResizable(false);
        }

        btnExportResultsDevicePing.setBackground(new java.awt.Color(45, 45, 45));
        btnExportResultsDevicePing.setFont(new java.awt.Font("Dubai", 0, 12)); // NOI18N
        btnExportResultsDevicePing.setForeground(new java.awt.Color(233, 247, 249));
        btnExportResultsDevicePing.setText("Export Results");
        btnExportResultsDevicePing.setToolTipText("Exports the results to a JSON file.");

btnExportResultsDevicePing.setBorder(javax.swing.BorderFactory.createBevelBorder(javax.swing.
border.BevelBorder.RAISED));
        btnExportResultsDevicePing.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                btnExportResultsDevicePingActionPerformed(evt);
            }
        });

        lblIPAddressErrorDevicePing.setFont(new java.awt.Font("Dubai Medium", 0, 14)); //
NOI18N
        lblIPAddressErrorDevicePing.setForeground(new java.awt.Color(255, 51, 0));

        lblDevicePingInProgress.setFont(new java.awt.Font("Dubai Medium", 0, 14)); // NOI18N
        lblDevicePingInProgress.setForeground(new java.awt.Color(0, 204, 0));

        tblDevicePingPacketResults.setBackground(new java.awt.Color(255, 255, 255));
        tblDevicePingPacketResults.setForeground(new java.awt.Color(45, 45, 45));
        tblDevicePingPacketResults.setModel(new javax.swing.table.DefaultTableModel(
            new Object [][] {

            },
            new String [] {
                "Pings Sent", "Successful Pings", "Unsuccessful Pings", "Packet Loss (%)"
            }
        ) {
            boolean[] canEdit = new boolean [] {
                false, false, false, false
            };

            public boolean isCellEditable(int rowIndex, int columnIndex) {
                return canEdit [columnIndex];
            }
        });
        tblDevicePingPacketResults.getTableHeader().setReorderingAllowed(false);
        scrDevicePingPacketResults.setViewportView(tblDevicePingPacketResults);
        if (tblDevicePingPacketResults.getColumnModel().getColumnCount() > 0) {
            tblDevicePingPacketResults.getColumnModel().getColumn(0).setResizable(false);
            tblDevicePingPacketResults.getColumnModel().getColumn(1).setResizable(false);
            tblDevicePingPacketResults.getColumnModel().getColumn(2).setResizable(false);
        }

        tblDevicePingResponseResults.setBackground(new java.awt.Color(255, 255, 255));
        tblDevicePingResponseResults.setForeground(new java.awt.Color(45, 45, 45));
        tblDevicePingResponseResults.setModel(new javax.swing.table.DefaultTableModel(
            new Object [][] {

            },
            new String [] {
                "Minimum Round Trip Time (ms)", "Maximum Round Trip Time (ms)", "Average
```

```
Round Trip Time (ms)"
            }
        ) {
            boolean[] canEdit = new boolean [] {
                false, false, false
            };

            public boolean isCellEditable(int rowIndex, int columnIndex) {
                return canEdit [columnIndex];
            }
        });
        tblDevicePingResponseResults.getTableHeader().setReorderingAllowed(false);
        scrDevicePingResponseResults.setViewportView(tblDevicePingResponseResults);
        if (tblDevicePingResponseResults.getColumnModel().getColumnCount() > 0) {
            tblDevicePingResponseResults.getColumnModel().getColumn(0).setResizable(false);
            tblDevicePingResponseResults.getColumnModel().getColumn(1).setResizable(false);
            tblDevicePingResponseResults.getColumnModel().getColumn(2).setResizable(false);
        }

        javax.swing.GroupLayout pnlDevicePingLayout = new
javax.swing.GroupLayout(pnlDevicePing);
        pnlDevicePing.setLayout(pnlDevicePingLayout);
        pnlDevicePingLayout.setHorizontalGroup(

pnlDevicePingLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addComponent(lblDevicePing, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
            .addGroup(pnlDevicePingLayout.createSequentialGroup()
                .addGap(30, 30, 30)

.addGroup(pnlDevicePingLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                    .addGroup(pnlDevicePingLayout.createSequentialGroup()

.addGroup(pnlDevicePingLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.TRAILING)
                            .addComponent(btnExportResultsDevicePing,
javax.swing.GroupLayout.PREFERRED_SIZE, 104, javax.swing.GroupLayout.PREFERRED_SIZE)
                            .addComponent(scrDevicePing,
javax.swing.GroupLayout.PREFERRED_SIZE, 940, javax.swing.GroupLayout.PREFERRED_SIZE)
                            .addComponent(scrDevicePingPacketResults,
javax.swing.GroupLayout.PREFERRED_SIZE, 940, javax.swing.GroupLayout.PREFERRED_SIZE)
                            .addComponent(scrDevicePingResponseResults,
javax.swing.GroupLayout.PREFERRED_SIZE, 940, javax.swing.GroupLayout.PREFERRED_SIZE))
                        .addGap(30, 30, 30))
                    .addGroup(pnlDevicePingLayout.createSequentialGroup()

.addGroup(pnlDevicePingLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                            .addComponent(lblPingInterval)
                            .addComponent(lblContinuousPinging)
                            .addComponent(lblNumberOfPings)
                            .addComponent(lblIPAddressDevicePing))
                        .addGap(18, 18, 18)

.addGroup(pnlDevicePingLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                            .addComponent(txfIPAddressDevicePing,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
                            .addComponent(spnPingInterval,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
                            .addComponent(spnNumberOfPings,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
                            .addGroup(javax.swing.GroupLayout.Alignment.TRAILING,
pnlDevicePingLayout.createSequentialGroup()
                                .addGap(0, 0, Short.MAX_VALUE)
                                .addComponent(chkContinuousPinging))
                            .addComponent(btnStartDevicePing,
javax.swing.GroupLayout.Alignment.TRAILING, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE))
                        .addGap(18, 18, 18)

.addGroup(pnlDevicePingLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                            .addComponent(lblIPAddressErrorDevicePing,
javax.swing.GroupLayout.PREFERRED_SIZE, 609, javax.swing.GroupLayout.PREFERRED_SIZE)
                            .addComponent(lblDevicePingInProgress,
javax.swing.GroupLayout.PREFERRED_SIZE, 614, javax.swing.GroupLayout.PREFERRED_SIZE))
                        .addGap(25, 25, 25))))
        );
        pnlDevicePingLayout.setVerticalGroup(

pnlDevicePingLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addGroup(pnlDevicePingLayout.createSequentialGroup()
                .addGap(36, 36, 36)

.addGroup(pnlDevicePingLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.TRAILING)
                    .addGroup(pnlDevicePingLayout.createSequentialGroup()
                        .addComponent(lblDevicePing)
                        .addGap(18, 18, 18)

.addGroup(pnlDevicePingLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELINE)
                            .addComponent(lblIPAddressDevicePing)
                            .addComponent(txfIPAddressDevicePing,
javax.swing.GroupLayout.PREFERRED_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE)))
                    .addComponent(lblIPAddressErrorDevicePing,
javax.swing.GroupLayout.PREFERRED_SIZE, 24, javax.swing.GroupLayout.PREFERRED_SIZE))
                .addGap(18, 18, 18)

.addGroup(pnlDevicePingLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELINE)
                    .addComponent(lblPingInterval)
                    .addComponent(spnPingInterval, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE))
                .addGap(18, 18, 18)

.addGroup(pnlDevicePingLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELINE)
                    .addComponent(spnNumberOfPings, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
                    .addComponent(lblNumberOfPings))
                .addGap(18, 18, 18)

.addGroup(pnlDevicePingLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING,
false)
                    .addComponent(lblContinuousPinging, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
                    .addComponent(chkContinuousPinging, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE))
                .addGap(18, 18, 18)

.addGroup(pnlDevicePingLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING,
false)
                    .addComponent(lblDevicePingInProgress,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
                    .addComponent(btnStartDevicePing, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE))
                .addGap(30, 30, 30)
                .addComponent(scrDevicePing, javax.swing.GroupLayout.PREFERRED_SIZE, 185,
javax.swing.GroupLayout.PREFERRED_SIZE)
```

```
                .addGap(18, 18, 18)
                .addComponent(scrDevicePingResponseResults, 
javax.swing.GroupLayout.PREFERRED_SIZE, 40, javax.swing.GroupLayout.PREFERRED_SIZE)
                .addGap(18, 18, 18)
                .addComponent(scrDevicePingPacketResults, 
javax.swing.GroupLayout.PREFERRED_SIZE, 40, javax.swing.GroupLayout.PREFERRED_SIZE)
                .addGap(30, 30, 30)
                .addComponent(btnExportResultsDevicePing, 
javax.swing.GroupLayout.PREFERRED_SIZE, 25, javax.swing.GroupLayout.PREFERRED_SIZE)
                .addContainerGap(45, Short.MAX_VALUE))
        );

        pnlMainPanel.add(pnlDevicePing, "cardDevicePing");

        pnlTCPMessageConnect.setBackground(new java.awt.Color(49, 49, 49));
        pnlTCPMessageConnect.setPreferredSize(new java.awt.Dimension(1000, 720));

        lblTCPMessageConnect.setFont(new java.awt.Font("Dubai Medium", 1, 24)); // NOI18N
        lblTCPMessageConnect.setForeground(new java.awt.Color(233, 247, 249));
        lblTCPMessageConnect.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);
        lblTCPMessageConnect.setText("TCP Message - Connect");

        lblPortTCPMessageConnect.setFont(new java.awt.Font("Dubai Medium", 0, 14)); // NOI18N
        lblPortTCPMessageConnect.setForeground(new java.awt.Color(233, 247, 249));
        lblPortTCPMessageConnect.setText("Enter the port to connect to:");

        lblIPAddressTCPMessageConnect.setFont(new java.awt.Font("Dubai Medium", 0, 14)); // 
NOI18N
        lblIPAddressTCPMessageConnect.setForeground(new java.awt.Color(233, 247, 249));
        lblIPAddressTCPMessageConnect.setText("Enter the IP address:");

        txfIPAddressTCPMessageConnect.setBackground(new java.awt.Color(255, 255, 255));
        txfIPAddressTCPMessageConnect.setFont(new java.awt.Font("Dubai", 0, 12)); // NOI18N
        txfIPAddressTCPMessageConnect.setForeground(new java.awt.Color(45, 45, 45));
        txfIPAddressTCPMessageConnect.setText(DEFAULT_IP_ADDRESS_TEXT);
        txfIPAddressTCPMessageConnect.setToolTipText("The device to connect to.");

txfIPAddressTCPMessageConnect.setBorder(javax.swing.BorderFactory.createEmptyBorder(1, 5, 1, 
1));
        txfIPAddressTCPMessageConnect.setPreferredSize(new java.awt.Dimension(104, 23));
        txfIPAddressTCPMessageConnect.addFocusListener(new java.awt.event.FocusAdapter() {
            public void focusGained(java.awt.event.FocusEvent evt) {
                txfIPAddressTCPMessageConnectFocusGained(evt);
            }
        });

        spnPortTCPMessageConnect.setFont(new java.awt.Font("Dubai", 0, 12)); // NOI18N
        spnPortTCPMessageConnect.setModel(new javax.swing.SpinnerNumberModel(1234, 1, 65535, 
1));
        spnPortTCPMessageConnect.setToolTipText("The port to connect on.");
        spnPortTCPMessageConnect.setBorder(javax.swing.BorderFactory.createEmptyBorder(1, 1, 
1, 1));
        spnPortTCPMessageConnect.setPreferredSize(new java.awt.Dimension(104, 23));

        btnStartTCPMessageConnect.setBackground(new java.awt.Color(45, 45, 45));
        btnStartTCPMessageConnect.setFont(new java.awt.Font("Dubai", 0, 12)); // NOI18N
        btnStartTCPMessageConnect.setForeground(new java.awt.Color(233, 247, 249));
        btnStartTCPMessageConnect.setText("Start TCP Connect");

btnStartTCPMessageConnect.setBorder(javax.swing.BorderFactory.createBevelBorder(javax.swing.b
order.BevelBorder.RAISED));
        btnStartTCPMessageConnect.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                btnStartTCPMessageConnectActionPerformed(evt);
            }
        });

        lblIPAddressErrorTCPMessageConnect.setFont(new java.awt.Font("Dubai Medium", 0, 14));
// NOI18N
        lblIPAddressErrorTCPMessageConnect.setForeground(new java.awt.Color(255, 51, 0));

        lblTCPMessageConnectInProgress.setFont(new java.awt.Font("Dubai Medium", 0, 14)); //
NOI18N
        lblTCPMessageConnectInProgress.setForeground(new java.awt.Color(0, 204, 0));

        txpTCPMessageConnect.setEditable(false);
        txpTCPMessageConnect.setBackground(new java.awt.Color(255, 255, 255));
        txpTCPMessageConnect.setFont(new java.awt.Font("Dubai Medium", 0, 14)); // NOI18N
        txpTCPMessageConnect.setForeground(new java.awt.Color(45, 45, 45));
        txpTCPMessageConnect.setToolTipText("Messages will appear here");
        scrTCPMessageConnect.setViewportView(txpTCPMessageConnect);

        lblEnterMessageTCPMessageConnect.setFont(new java.awt.Font("Dubai Medium", 0, 14));
// NOI18N
        lblEnterMessageTCPMessageConnect.setForeground(new java.awt.Color(233, 247, 249));
        lblEnterMessageTCPMessageConnect.setText("Enter message:");

        txfMessageTCPMessageConnect.setBackground(new java.awt.Color(255, 255, 255));
        txfMessageTCPMessageConnect.setFont(new java.awt.Font("Dubai Medium", 0, 14)); //
NOI18N
        txfMessageTCPMessageConnect.setForeground(new java.awt.Color(45, 45, 45));
        txfMessageTCPMessageConnect.setToolTipText("Write the message here.");

        btnSendTCPMessageConnect.setBackground(new java.awt.Color(45, 45, 45));
        btnSendTCPMessageConnect.setFont(new java.awt.Font("Dubai", 0, 12)); // NOI18N
        btnSendTCPMessageConnect.setForeground(new java.awt.Color(233, 247, 249));
        btnSendTCPMessageConnect.setText("Send");

btnSendTCPMessageConnect.setBorder(javax.swing.BorderFactory.createBevelBorder(javax.swing.bo
rder.BevelBorder.RAISED));
        btnSendTCPMessageConnect.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                btnSendTCPMessageConnectActionPerformed(evt);
            }
        });

        btnExportResultsTCPMessageConnect.setBackground(new java.awt.Color(45, 45, 45));
        btnExportResultsTCPMessageConnect.setFont(new java.awt.Font("Dubai", 0, 12)); //
NOI18N
        btnExportResultsTCPMessageConnect.setForeground(new java.awt.Color(233, 247, 249));
        btnExportResultsTCPMessageConnect.setText("Export Results");
        btnExportResultsTCPMessageConnect.setToolTipText("Exports the results to a text
file.");

btnExportResultsTCPMessageConnect.setBorder(javax.swing.BorderFactory.createBevelBorder(javax
.swing.border.BevelBorder.RAISED));
        btnExportResultsTCPMessageConnect.addActionListener(new
java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                btnExportResultsTCPMessageConnectActionPerformed(evt);
            }
        });

        javax.swing.GroupLayout pnlTCPMessageConnectLayout = new
javax.swing.GroupLayout(pnlTCPMessageConnect);
        pnlTCPMessageConnect.setLayout(pnlTCPMessageConnectLayout);
        pnlTCPMessageConnectLayout.setHorizontalGroup(

pnlTCPMessageConnectLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addComponent(lblTCPMessageConnect, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
            .addGroup(javax.swing.GroupLayout.Alignment.TRAILING,
pnlTCPMessageConnectLayout.createSequentialGroup()

.addGroup(pnlTCPMessageConnectLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.TR
AILING)
                    .addGroup(pnlTCPMessageConnectLayout.createSequentialGroup()
                        .addContainerGap(javax.swing.GroupLayout.DEFAULT_SIZE,
Short.MAX_VALUE)
                        .addComponent(btnExportResultsTCPMessageConnect,
javax.swing.GroupLayout.PREFERRED_SIZE, 104, javax.swing.GroupLayout.PREFERRED_SIZE))
                    .addGroup(pnlTCPMessageConnectLayout.createSequentialGroup()
                        .addGap(30, 30, 30)

.addGroup(pnlTCPMessageConnectLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LE
ADING)
                            .addGroup(pnlTCPMessageConnectLayout.createSequentialGroup()
                                .addComponent(lblEnterMessageTCPMessageConnect)
                                .addGap(18, 18, 18)
                                .addComponent(txfMessageTCPMessageConnect)
                                .addGap(18, 18, 18)
                                .addComponent(btnSendTCPMessageConnect,
javax.swing.GroupLayout.PREFERRED_SIZE, 69, javax.swing.GroupLayout.PREFERRED_SIZE))
                            .addComponent(scrTCPMessageConnect)
                            .addGroup(pnlTCPMessageConnectLayout.createSequentialGroup()

.addGroup(pnlTCPMessageConnectLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LE
ADING)
                                    .addComponent(lblIPAddressTCPMessageConnect)
                                    .addComponent(lblPortTCPMessageConnect))
                                .addGap(18, 18, 18)

.addGroup(pnlTCPMessageConnectLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LE
ADING, false)
                                    .addComponent(txfIPAddressTCPMessageConnect,
javax.swing.GroupLayout.DEFAULT_SIZE, 120, Short.MAX_VALUE)
                                    .addComponent(spnPortTCPMessageConnect,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
                                    .addComponent(btnStartTCPMessageConnect,
javax.swing.GroupLayout.Alignment.TRAILING, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE))
                                .addGap(18, 18, 18)

.addGroup(pnlTCPMessageConnectLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LE
ADING)
                                    .addComponent(lblIPAddressErrorTCPMessageConnect,
javax.swing.GroupLayout.DEFAULT_SIZE, 622, Short.MAX_VALUE)
                                    .addComponent(lblTCPMessageConnectInProgress,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
Short.MAX_VALUE))))))
                .addGap(30, 30, 30))
        );
        pnlTCPMessageConnectLayout.setVerticalGroup(

pnlTCPMessageConnectLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addGroup(pnlTCPMessageConnectLayout.createSequentialGroup()
                .addGap(36, 36, 36)
                .addComponent(lblTCPMessageConnect)
                .addGap(18, 18, 18)

.addGroup(pnlTCPMessageConnectLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LE
ADING, false)

.addGroup(pnlTCPMessageConnectLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.BA
SELINE)
                        .addComponent(lblIPAddressTCPMessageConnect)
                        .addComponent(txfIPAddressTCPMessageConnect,
javax.swing.GroupLayout.PREFERRED_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE))
                    .addComponent(lblIPAddressErrorTCPMessageConnect,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE))
                .addGap(18, 18, 18)

.addGroup(pnlTCPMessageConnectLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.BA
SELINE)
                    .addComponent(lblPortTCPMessageConnect)
                    .addComponent(spnPortTCPMessageConnect,
javax.swing.GroupLayout.PREFERRED_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE))
                .addGap(18, 18, 18)

.addGroup(pnlTCPMessageConnectLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LE
ADING, false)
                    .addComponent(lblTCPMessageConnectInProgress,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
                    .addComponent(btnStartTCPMessageConnect,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE))
                .addGap(30, 30, 30)
                .addComponent(scrTCPMessageConnect, javax.swing.GroupLayout.PREFERRED_SIZE,
341, javax.swing.GroupLayout.PREFERRED_SIZE)
                .addGap(18, 18, 18)

.addGroup(pnlTCPMessageConnectLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.BA
SELINE)
                    .addComponent(lblEnterMessageTCPMessageConnect)
                    .addComponent(txfMessageTCPMessageConnect,
javax.swing.GroupLayout.PREFERRED_SIZE, 26, Short.MAX_VALUE)
                    .addComponent(btnSendTCPMessageConnect,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE))
                .addGap(30, 30, 30)
                .addComponent(btnExportResultsTCPMessageConnect)
                .addContainerGap(45, Short.MAX_VALUE))
        );

        pnlMainPanel.add(pnlTCPMessageConnect, "cardTCPMessageConnect");

        javax.swing.GroupLayout pnlHomePageLayout = new javax.swing.GroupLayout(pnlHomePage);
        pnlHomePage.setLayout(pnlHomePageLayout);
        pnlHomePageLayout.setHorizontalGroup(
            pnlHomePageLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addGroup(pnlHomePageLayout.createSequentialGroup()
                .addComponent(pnlSideBar, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
                .addGap(0, 0, 0)
                .addComponent(pnlMainPanel, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
                .addGap(365, 365, 365))
        );
        pnlHomePageLayout.setVerticalGroup(
            pnlHomePageLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addComponent(pnlSideBar, javax.swing.GroupLayout.DEFAULT_SIZE, 726,
Short.MAX_VALUE)
            .addGroup(pnlHomePageLayout.createSequentialGroup()
                .addComponent(pnlMainPanel, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
                .addContainerGap(javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE))
        );

        javax.swing.GroupLayout layout = new javax.swing.GroupLayout(getContentPane());
        getContentPane().setLayout(layout);
        layout.setHorizontalGroup(
            layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addComponent(pnlHomePage, javax.swing.GroupLayout.PREFERRED_SIZE, 1280,
javax.swing.GroupLayout.PREFERRED_SIZE)
        );
        layout.setVerticalGroup(
            layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addComponent(pnlHomePage, javax.swing.GroupLayout.PREFERRED_SIZE, 720,
javax.swing.GroupLayout.PREFERRED_SIZE)
        );

        pack();
    }
```