Code Documentation

Socket Programing code using Java Documentation:

```
import java.android.*;
import java.io.*;
import java.okhttp3.*;
import org.w3c.dom.Document;
```

Import statements for an Android application using OkHttp and XML parsing

Imports classes from the java.io package for input and output operations, such as IOException.

Imports classes from the okhttp3 package for making HTTP requests.

Imports the Document interface from the org.w3c.dom package, which represents an XML document in the DOM API.

```
public class MainActivity extends AppCompatActivity
```

MainActivity is the entry point of the application, inheriting from AppCompatActivity. This class sets up the main user interface and initializes necessary components when the application starts.

```
 EditText receivePortEditText, targetPortEditText, messageEditText, targetIPEditText,
encrypKeyEditText;
    RelativeLayout firstLayout, thirdLayout;
    Button connectBtn, getIPBtn;
    ImageButton sendButton, voiceMsgOn, attachmentBtn;
    TextView clickHereBtn;

    MenuItem changeBG, saveChat, disconnect, removeAllChat,voiceMode, resetLayout;

    ServerClass serverClass;
    ClientClass clientClass;
    SendReceive sendReceive;

    ScrollView conversations;
    LinearLayout conversationLayout;

    boolean need = true, voice = false;

    private Toolbar mToolbar;

    String filePath = null, ip = null;


    int shift = 0;


    static final int MESSAGE_READ=1;
```

```
    static final String TAG = "trap";
```

The given code snippet declares various UI components and member variables for an Android activity. It includes `EditText` fields for user input (`receivePortEditText`, `targetPortEditText`, `messageEditText`, `targetIPEditText`, `encrypKeyEditText`), `RelativeLayout` instances for layout management (`firstLayout`, `thirdLayout`), and various buttons (`connectBtn`, `getIPBtn`, `sendButton`, `voiceMsgOn`, `attachmentBtn`). A `TextView` (`clickHereBtn`) is also declared for displaying clickable text. Menu items (`changeBG`, `saveChat`, `disconnect`, `removeAllChat`, `voiceMode`, `resetLayout`) provide options for various actions. Networking classes (`ServerClass`, `ClientClass`, `SendReceive`) facilitate server-client communication. A `ScrollView` (`conversations`) and a `LinearLayout` (`conversationLayout`) handle chat conversation display. Boolean flags (`need`, `voice`) and integers (`shift`) manage state and configuration. Additionally, a `Toolbar` (`mToolbar`) is used for the app's action bar, and strings (`filePath`, `ip`) store file paths and IP addresses. Constants (`MESSAGE_READ`, `TAG`) are used for message handling and logging, respectively.

```
    private static final int REQUEST_EXTERNAL_STORAGE = 1;
    private static String[] PERMISSIONS_STORAGE = {
            Manifest.permission.READ_EXTERNAL_STORAGE,
            Manifest.permission.WRITE_EXTERNAL_STORAGE
    };



Handler handler=new Handler(new Handler.Callback() {
        @Override
        public boolean handleMessage(Message msg) {

            if (msg.what == MESSAGE_READ) {
                byte[] readBuff = (byte[]) msg.obj;
                String tempMsg = new String(readBuff, 0, msg.arg1);
                addMessage(Color.parseColor("#F42FFF"), tempMsg);
            }
            return true;
        }
    });
```

The given code snippet defines a request code for external storage permissions (`REQUEST_EXTERNAL_STORAGE = 1`) and an array of required storage permissions (`PERMISSIONS_STORAGE`), including read and write access. A `Handler` object (`handler`) is created with a callback to handle messages. When a message is received (`handleMessage` method), it checks if the message type (`msg.what`) matches `MESSAGE_READ`. If it does, the message's byte array payload (`msg.obj`) is converted to a string (`tempMsg`) using its byte array and length (`msg.arg1`). The method `addMessage` is then called to display the message, applying a specific color (`#F42FFF`). The handler

returns `true` to indicate the message was handled. This setup is crucial for managing asynchronous message handling, particularly for displaying received messages in a user interface.

```
@Override
  protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);

      initialization();

  }
  @Override
  private void initialization() {


      mToolbar = findViewById(R.id.main_page_toolbar);


      setSupportActionBar(mToolbar);
      getSupportActionBar().setTitle("Messenging App");


      receivePortEditText = findViewById(R.id.receiveEditText);
      targetPortEditText = findViewById(R.id.targetPortEditText);
      messageEditText = findViewById(R.id.messageEditText);
      targetIPEditText = findViewById(R.id.targetIPEditText);
      encrypKeyEditText = findViewById(R.id.encrypEditText);
      firstLayout = findViewById(R.id.firstLayout);
      thirdLayout = findViewById(R.id.third_layout);
      sendButton = findViewById(R.id.send_message_btn);
      connectBtn = findViewById(R.id.connectButton);
      getIPBtn = findViewById(R.id.getIPButton);
      clickHereBtn = findViewById(R.id.click_here);
      voiceMsgOn = findViewById(R.id.voice_btn);
      attachmentBtn = findViewById(R.id.send_files_and_voice_btn);

      conversations = findViewById(R.id.conversations);
      conversationLayout = findViewById(R.id.scroll_view_linear_layout);
  }
```

The provided code defines the `onCreate` method and an `initialization` method for an Android activity. In `onCreate`, the activity's layout is set to `activity_main`, and the `initialization` method is called. The `initialization` method sets up the activity's user interface components. It first configures the toolbar (`mToolbar`) and sets its title to "Messaging App". It then initializes various UI elements using `findViewById` to link them to their corresponding views in the layout, including `EditText` fields (`receivePortEditText`, `targetPortEditText`, `messageEditText`, `targetIPEditText`, `encrypKeyEditText`), `RelativeLayout` containers (`firstLayout`, `thirdLayout`), buttons (`sendButton`, `connectBtn`, `getIPBtn`), an `ImageButton` (`voiceMsgOn`, `attachmentBtn`), a

`TextView` (`clickHereBtn`), a `ScrollView` (`conversations`), and a `LinearLayout` (`conversationLayout`). This setup organizes the UI components for user interaction in the messaging application.

```
@Override
    public boolean onCreateOptionsMenu(Menu menu) {

        super.onCreateOptionsMenu(menu);
        getMenuInflater().inflate(R.menu.options_menu, menu);

        changeBG = menu.findItem(R.id.main_change_bg_option);
        saveChat = menu.findItem(R.id.main_save_chat_option);
        disconnect = menu.findItem(R.id.main_disconnect_option);
        removeAllChat = menu.findItem(R.id.main_remove_chat_option);
        voiceMode = menu.findItem(R.id.main_voice_mode);
        resetLayout = menu.findItem(R.id.main_reset_layout);

        changeBG.setEnabled(false);
        saveChat.setEnabled(false);
        disconnect.setEnabled(false);
        removeAllChat.setEnabled(false);
        voiceMode.setEnabled(false);
        resetLayout.setEnabled(false);

        return true;
    }


    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        super.onOptionsItemSelected(item);


        if(item.getItemId() == R.id.main_change_bg_option)
            openChangeBGDialogBox();

        if(item.getItemId() == R.id.main_save_chat_option)
            openSaveChatDialogBoxForHim();

        if(item.getItemId() == R.id.main_disconnect_option)
            openDisconnectAlertDialogBox();

        if(item.getItemId() == R.id.main_remove_chat_option)
            openRemoveAllChatAlertDialogBox();

        if(item.getItemId() == R.id.main_voice_mode)
            voiceModeOperation();

        if(item.getItemId() == R.id.main_reset_layout)
            resetLayoutAlertDialog();
        return true;
    }
```

The provided code snippet defines two overridden methods for handling the options menu in an Android activity.

onCreateOptionsMenu

The `onCreateOptionsMenu` method is responsible for creating the options menu when the activity is created. It inflates the menu layout from `R.menu.options_menu` and initializes several `MenuItem` objects (`changeBG`, `saveChat`, `disconnect`, `removeAllChat`, `voiceMode`, `resetLayout`) by finding them by their IDs. These menu items are then disabled by calling `setEnabled(false)` on each one. Finally, the method returns `true` to indicate that the menu has been created successfully.

onOptionsItemSelected

The `onOptionsItemSelected` method handles the actions to be performed when a menu item is selected. It checks the ID of the selected item and calls the corresponding method:

- `openChangeBGDialogBox()` for the "Change Background" option.
- `openSaveChatDialogBoxForHim()` for the "Save Chat" option.
- `openDisconnectAlertDialogBox()` for the "Disconnect" option.
- `openRemoveAllChatAlertDialogBox()` for the "Remove All Chat" option.
- `voiceModeOperation()` for the "Voice Mode" option.
- `resetLayoutAlertDialog()` for the "Reset Layout" option.

This setup allows the application to present a menu to the user and perform specific actions based on the selected menu item.

```
private void resetLayoutAlertDialog() {
        final AlertDialog.Builder alert = new AlertDialog.Builder(MainActivity.this);
        View mView = getLayoutInflater().inflate(R.layout.custom_disconnect_dialog, null);

        TextView textView = mView.findViewById(R.id.custom_disconnect_dialog_textView);
        textView.setText("Do you want to reset your layout?");
        Button btn_cancel = (Button) mView.findViewById(R.id.btn_cancel);
        Button btn_yes = (Button) mView.findViewById(R.id.btn_yes);

        alert.setView(mView);

        final AlertDialog alertDialog = alert.create();
        alertDialog.setCanceledOnTouchOutside(false);
        alertDialog.setTitle("Reseting your Layout");

        btn_cancel.setOnClickListener(new View.OnClickListener() {
            @Override
```

```
            public void onClick(View v) {
                Toast.makeText(MainActivity.this, "Cancelled", Toast.LENGTH_SHORT).show();
                alertDialog.close();
            }
        });

        btn_yes.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {

                resetLayoutForHim();
                alertDialog.close();
            }
        });

        alertDialog.open();
    }

    private void resetLayoutForHim() {
        sendReceive.write(caesarCipherEncryption("bg@%@bg0",shift));
        thirdLayout.setBackgroundResource(R.drawable.background3);
        mToolbar.setBackgroundColor(Color.parseColor("#233E4E"));
    }


    private void voiceModeOperation() {
        if(voiceMode.getTitle().equals("Voice Mode : Off")) {
            voiceMode.setTitle("Voice Mode : On");
            openVoiceModeDialogBox();
            voice = true;
            attachmentBtn.setVisibility(View.INVISIBLE);
            voiceMsgOn.setVisibility(View.VISIBLE);
            saveChat.setEnabled(false);
            disconnect.setEnabled(false);
            changeBG.setEnabled(false);
            removeAllChat.setEnabled(false);
            resetLayout.setEnabled(false);
            Toast.makeText(MainActivity.this, "Voice Mode Enabled",
Toast.LENGTH_SHORT).show();


        }
        else {
            voiceMode.setTitle("Voice Mode : Off");
            voice = false;
            attachmentBtn.setVisibility(View.VISIBLE);
            voiceMsgOn.setVisibility(View.INVISIBLE);
            saveChat.setEnabled(true);
            disconnect.setEnabled(true);
            changeBG.setEnabled(true);
            removeAllChat.setEnabled(true);
            resetLayout.setEnabled(true);
            Toast.makeText(MainActivity.this, "Voice Mode Disabled",
Toast.LENGTH_SHORT).show();
```

```
        }

    }
```

The provided code snippet defines three methods for managing UI interactions and functionality in the `MainActivity` class of an Android application.

resetLayoutAlertDialog

The `resetLayoutAlertDialog` method creates and displays a custom alert dialog to confirm if the user wants to reset the layout. It inflates a custom view (`custom_disconnect_dialog`), sets the dialog text, and defines `OnClickListener`s for the "Cancel" and "Yes" buttons. The "Cancel" button dismisses the dialog with a toast message, while the "Yes" button calls `resetLayoutForHim` to reset the layout and then closes the dialog.

resetLayoutForHim

The `resetLayoutForHim` method resets the layout to a predefined state. It sends a command using `sendReceive.write` with encrypted text and updates the background of `thirdLayout` and the toolbar color.

voiceModeOperation

The `voiceModeOperation` method toggles the voice mode on and off. When enabled, it changes the `voiceMode` title to "Voice Mode : On", displays the voice message button, hides the attachment button, disables various menu items, and shows a toast message indicating voice mode is enabled. When disabled, it reverts these changes, setting the title to "Voice Mode : Off" and showing a toast message indicating voice mode is disabled.

These methods provide user interaction handling for resetting the layout and toggling voice mode, enhancing the app's functionality and user experience.

```
    private void openVoiceModeDialogBox() {
        final AlertDialog.Builder alert = new AlertDialog.Builder(MainActivity.this);
        View mView = getLayoutInflater().inflate(R.layout.custom_voice_mode_dialog, null);

        Button btn_yes = (Button) mView.findViewById(R.id.btn_yes);

        alert.setView(mView);

        final AlertDialog alertDialog = alert.create();
        alertDialog.setCanceledOnTouchOutside(true);
        alertDialog.setTitle("Voice Command Mode");

        btn_yes.setOnClickListener((v) -> {
            alertDialog.close();
        });
```

```
        alertDialog.show();
    }
```

The `openVoiceModeDialogBox` method creates and displays a custom alert dialog in the `MainActivity` class of an Android application. It uses an `AlertDialog.Builder` to construct the dialog with a custom layout (`custom_voice_mode_dialog`). Inside the dialog, it inflates the layout and retrieves a reference to the "Yes" button (`btn_yes`). An `OnClickListener` is set for the "Yes" button to close the dialog when clicked (`alertDialog.close()`). The dialog is then displayed with a title "Voice Command Mode" and allows it to be closed by touching outside of it (`alertDialog.setCanceledOnTouchOutside(true)`). This method provides a simple dialog interface for handling user interactions related to voice command mode settings within the application.

```
 private void openRemoveAllChatAlertDialogBox() {
        final AlertDialog.Builder alert = new AlertDialog.Builder(MainActivity.this);
        View mView = getLayoutInflater().inflate(R.layout.custom_remove_all_chat_dialog,
null);


        CheckBox cbRemoveForAll = mView.findViewById(R.id.remove_for_all);

        Button btn_cancel = (Button) mView.findViewById(R.id.btn_cancel);
        Button btn_clear_messages = (Button) mView.findViewById(R.id.btn_clear_messages);

        alert.setView(mView);

        final AlertDialog alertDialog = alert.create();
        alertDialog.setCanceledOnTouchOutside(false);
        alertDialog.setTitle("Clearing All your Chat");
        cbRemoveForAll.setOnCheckedChangeListener(new
CompoundButton.OnCheckedChangeListener() {
            @Override
            public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
                if(isChecked)
buttonView.setTextColor(getResources().getColor(R.color.green)); // changing color of
checkbox
                else buttonView.setTextColor(getResources().getColor(R.color.black));
            }
        });



        btn_cancel.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Toast.makeText(MainActivity.this, "Cancselled",
Toast.LENGTH_SHORT).show();
                alertDialog.dismiss();
            }
        });
```

```
        btn_clear_messages.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {

                if(cbRemoveForAll.isChecked()){
                    sendReceive.write(caesarCipherEncryption("remove@%@", shift));
                }

                removeAllChatForHim();
                Log.d(TAG, "Remove all chat Message: "
+caesarCipherEncryption("diconnect@%@d", shift));
                alertDialog.dismiss();
            }
        });

        alertDialog.show();
    }
```

This method provides a user-friendly interface for confirming the removal of all chat messages. It ensures clarity with color-coded checkbox feedback and allows cancellation of the operation. Moreover, it integrates securely by encrypting commands and notifying users of the removal process through logging.

This implementation improves user experience by confirming sensitive actions and ensuring comprehensive messaging management within the application.

```
 private void removeAllChatForHim() {

        conversationLayout.removeAllViews();
        Toast.makeText(this, "All chat has been removed!", Toast.LENGTH_SHORT).show();

    }


    private void openChangeBGDialogBox() {
        final AlertDialog.Builder alert = new AlertDialog.Builder(MainActivity.this);
        View mView = getLayoutInflater().inflate(R.layout.custom_background_change_dialog,
null);

        Button layout1 =  mView.findViewById(R.id.btn_bg1);
        Button layout2 =  mView.findViewById(R.id.btn_bg2);
        Button layout3 =  mView.findViewById(R.id.btn_bg3);
        Button layout4 =  mView.findViewById(R.id.btn_bg4);
        Button layout5 =  mView.findViewById(R.id.btn_bg5);
        Button layout6 =  mView.findViewById(R.id.btn_bg6);
        Button layout7 =  mView.findViewById(R.id.btn_bg7);
        Button layout8 =  mView.findViewById(R.id.btn_bg8);
        Button layout9 =  mView.findViewById(R.id.btn_bg9);

        alert.setView(mView);
```

```
        final AlertDialog alertDialog = alert.create();
        alertDialog.setCanceledOnTouchOutside(true);
        alertDialog.setTitle("Changing Background");


        layout1.setOnClickListener((v) -> {
            String msg =  "bg@%@bg1";
            sendReceive.write(caesarCipherEncryption(msg, shift));
            Toast.makeText(MainActivity.this, "background is selected as LAYOUT 1",
Toast.LENGTH_SHORT).show();
            alertDialog.dismiss();
        });

        layout2.setOnClickListener((v) -> {
            String msg =  "bg@%@bg2";
            sendReceive.write(caesarCipherEncryption(msg,shift));
            Toast.makeText(MainActivity.this, "background is selected as LAYOUT 2",
Toast.LENGTH_SHORT).show();
            alertDialog.dismiss();
        });

        layout3.setOnClickListener((v) -> {
            String msg =  "bg@%@bg3";
            sendReceive.write(caesarCipherEncryption(msg,shift));
            Toast.makeText(MainActivity.this, "background is selected as LAYOUT 3",
Toast.LENGTH_SHORT).show();
            alertDialog.dismiss();
        });

        layout4.setOnClickListener((v) -> {
            String msg =  "bg@%@bg4";
            sendReceive.write(caesarCipherEncryption(msg, shift));
            Toast.makeText(MainActivity.this, "background is selected as LAYOUT 4",
Toast.LENGTH_SHORT).show();
            alertDialog.dismiss();
        });
        layout5.setOnClickListener((v)-> {
            String msg =  "bg@%@bg5";
            sendReceive.write(caesarCipherEncryption(msg, shift));
            Toast.makeText(MainActivity.this, "background is selected as LAYOUT 5",
Toast.LENGTH_SHORT).show();
            alertDialog.dismiss();
        });

        layout6.setOnClickListener((v) -> {
            String msg =  "bg@%@bg6";
            sendReceive.write(caesarCipherEncryption(msg, shift));
            Toast.makeText(MainActivity.this, "background is selected as LAYOUT 6",
Toast.LENGTH_SHORT).show();
            alertDialog.dismiss();
        });

        layout7.setOnClickListener((v) -> {
            String msg =  "bg@%@bg7";
```

```
            sendReceive.write(caesarCipherEncryption(msg, shift));
            Toast.makeText(MainActivity.this, "background is selected as LAYOUT 7",
Toast.LENGTH_SHORT).show();
            alertDialog.dismiss();
        });

        layout8.setOnClickListener((v) -> {
            String msg =  "bg@%@bg8";
            sendReceive.write(caesarCipherEncryption(msg, shift));
            Toast.makeText(MainActivity.this, "background is selected as LAYOUT 8",
Toast.LENGTH_SHORT).show();
            alertDialog.dismiss();
        });

        layout9.setOnClickListener((v) -> {
            String msg =  "bg@%@bg9";
            sendReceive.write(caesarCipherEncryption(msg, shift));
            Toast.makeText(MainActivity.this, "background is selected as LAYOUT 9",
Toast.LENGTH_SHORT).show();
            alertDialog.dismiss();
        });

        alertDialog.show();

    }


    private void changeBGforHim(String msg) {

        if(msg.equals("bg@%@bg1")){
            thirdLayout.setBackgroundResource(R.drawable.background1);
            mToolbar.setBackgroundColor(Color.parseColor("#8B0000"));
        }

        else if(msg.equals("bg@%@bg2")){
            thirdLayout.setBackgroundResource(R.drawable.background2);
            mToolbar.setBackgroundColor(Color.parseColor("#461D3D"));
        }

        else if(msg.equals("bg@%@bg3")){
            thirdLayout.setBackgroundResource(R.drawable.background10);
            mToolbar.setBackgroundColor(Color.parseColor("#B0CA4590"));
        }

        else if(msg.equals("bg@%@bg4")){
            thirdLayout.setBackgroundResource(R.drawable.background4);
            mToolbar.setBackgroundColor(Color.parseColor("#30504C"));
        }

        else if(msg.equals("bg@%@bg5")){
            thirdLayout.setBackgroundResource(R.drawable.background5);
            mToolbar.setBackgroundColor(Color.parseColor("#30504C"));
        }
```

```
        else if(msg.equals("bg@%@bg6")){
            thirdLayout.setBackgroundResource(R.drawable.background6);
            mToolbar.setBackgroundColor(Color.parseColor("#461D3D"));
        }
        else if(msg.equals("bg@%@bg7")){
            thirdLayout.setBackgroundResource(R.drawable.background7);
            mToolbar.setBackgroundColor(Color.parseColor("#2E3F3B"));
        }
        else if(msg.equals("bg@%@bg8")){
            thirdLayout.setBackgroundResource(R.drawable.background8);
            mToolbar.setBackgroundColor(Color.parseColor("#DBAC30"));
        }
        else if(msg.equals("bg@%@bg9")){
            thirdLayout.setBackgroundResource(R.drawable.background9);
            mToolbar.setBackgroundColor(Color.parseColor("#095061"));
        }
        else if(msg.equals("bg@%@bg0")){
            thirdLayout.setBackgroundResource(R.drawable.background3);
            mToolbar.setBackgroundColor(Color.parseColor("#233E4E"));
        }
    }
```

The provided code includes two methods related to changing the background layout in the `MainActivity` class of an Android application.

`openChangeBGDialogBox` Method

The `openChangeBGDialogBox` method creates and displays an alert dialog for selecting different background layouts. It uses an `AlertDialog.Builder` to construct the dialog with a custom layout (`custom_background_change_dialog.xml`). Inside this dialog, it retrieves references to multiple buttons (`layout1` to `layout9`) corresponding to different background options. Each button is assigned an `OnClickListener` that triggers when clicked, sending an encrypted message via `sendReceive.write` and updating the background of `thirdLayout` and the toolbar (`mToolbar`) based on the selected background. A toast message confirms the background change, and the dialog is dismissed after selection.

`changeBGforHim` Method

The `changeBGforHim` method updates the UI to reflect the selected background (`msg`). Depending on the received message (`msg`), it sets the background resource of `thirdLayout` to different drawable resources (`R.drawable.background1` to `R.drawable.background9`). Additionally, it adjusts the color of the toolbar (`mToolbar`) accordingly using `Color.parseColor`. Each background option corresponds to a specific visual theme defined by its drawable resource and toolbar color.

These methods collectively provide functionality for dynamically changing the background layout and toolbar color in response to user interactions within the application. `openChangeBGDialogBox` facilitates user selection through a dialog interface, while `changeBGforHim` applies the selected changes to the UI. This enhances the visual

customization options available to users and improves the overall user experience by allowing them to personalize the app's appearance.

```
    private void openSaveChatDialogBoxForHim() {
        final AlertDialog.Builder alert = new AlertDialog.Builder(MainActivity.this);
        View mView = getLayoutInflater().inflate(R.layout.custom_disconnect_dialog, null);

        TextView textView = mView.findViewById(R.id.custom_disconnect_dialog_textView);
        textView.setText("Are you sure you want to save your conversations? It will be
saved on android/data/com.example.p2pmessenger/");
        Button btn_cancel = (Button) mView.findViewById(R.id.btn_cancel);
        Button btn_yes = (Button) mView.findViewById(R.id.btn_yes);

        alert.setView(mView);

        final AlertDialog alertDialog = alert.create();
        alertDialog.setCanceledOnTouchOutside(false);
        alertDialog.setTitle("Saving Conversations");

        btn_cancel.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Toast.makeText(MainActivity.this, "Cancelled", Toast.LENGTH_SHORT).show();
                alertDialog.dismiss();
            }
        });

        btn_yes.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {

                saveChatForHim();
                alertDialog.dismiss();
            }
        });

        alertDialog.show();
    }
```

The `openSaveChatDialogBoxForHim` method creates and displays an alert dialog in the `MainActivity` class of an Android application, specifically for confirming the user's intention to save conversations. Here's a breakdown of its functionality:

Method Description

The method initializes an `AlertDialog.Builder` (`alert`) using the current activity context (`MainActivity.this`). It inflates a custom view (`custom_disconnect_dialog.xml`) to customize the dialog's appearance and functionality. Inside this view, it retrieves references to UI elements:

TextView (`textView`): Displays a confirmation message informing the user about the save location of conversations.

Buttons (`btn_cancel`, `btn_yes`): Offer options to cancel or proceed with saving the conversations.

Dialog Setup

Setting View: The inflated view (`mView`) is set as the content of the alert dialog using `alert.setView(mView)`.

Title and Touch Behavior: The dialog title is set to "Saving Conversations", and it is configured to not dismiss when touched outside (`alertDialog.setCanceledOnTouchOutside(false)`).

Event Handling

Cancel Button: `btn_cancel` displays a toast message indicating cancellation and dismisses the dialog (`alertDialog.dismiss()`).

Save Button: `btn_yes` calls `saveChatForHim()` to initiate the saving process and then dismisses the dialog (`alertDialog.dismiss()`).

This method provides a clear and interactive interface for users to confirm their intent to save conversations. It ensures clarity by displaying informative text and handles user interactions effectively through button actions. By using a dialog, it enhances user experience by prompting for confirmation before executing potentially irreversible actions, such as saving data.

```
    private void saveChatForHim() {
        String allMessage = " ";
        int count = conversationLayout.getChildCount();
        TextView children;
        for (int i = 0; i < count; i++) {

            if(conversationLayout.getChildAt(i) instanceof ImageView)
                continue;
            else{
                children = (TextView) conversationLayout.getChildAt(i);

                if(children.getText().toString().contains(".txt has been received and
downloaded on android/data/com.example.p2p/")){
                    allMessage += "CLIENT: " + children.getText().toString() + "\n\n";
                }
                else if(children.getText().toString().contains(".txt has been sent")){
                    allMessage += "ME: " + children.getText().toString() + "\n\n";
                }

                else if(children.getText().toString().contains("am") ||
children.getText().toString().contains("pm")){

                    if (children.getCurrentTextColor() == Color.parseColor("#FCE4EC")) {
                        allMessage += " " + children.getText().toString() + "\n\n";
                    } else {
                        allMessage += " " + children.getText().toString() + "\n\n";
                    }
```

```
                }

                else if(children.getText().toString().equals("") ||
children.getText().toString().equals(null)){

                }
                else if(!children.getText().toString().equals("Background has been
changed") ){

                    if (children.getCurrentTextColor() == Color.parseColor("#FCE4EC")) {
                        allMessage += "ME: " + children.getText().toString() + "\n";
                    } else {
                        allMessage += "CLIENT: " + children.getText().toString() +"\n";
                    }
                }

            }

        }

        writeToFile(allMessage, true, "Chat History");
    }
```

This method effectively gathers and categorizes chat messages for saving, ensuring only relevant content is included based on predefined conditions. By iterating through child views of conversationLayout, it dynamically processes messages displayed in the UI, providing a streamlined approach to saving chat history. This enhances user interaction by offering a straightforward method to preserve chat records within the application.

```
    private void writeToFile(String data, boolean timeStamp, String name) {
        String time = "";
        if(timeStamp){

            time = getTime(true);
        }


        File path = this.getExternalFilesDir(null);
        filePath = path.toString();

        File file;

        if(time.equals(""))
            file = new File(path, name);

        else
            file = new File(path, name + "(" + time + ")" + System.lineSeparator() +
".txt");

        FileOutputStream stream;
        try {
            stream = new FileOutputStream(file, false);
            stream.write(data.getBytes());
```

```
            stream.close();
            Toast.makeText(this, "File Succcessfully Saved!", Toast.LENGTH_SHORT).show();
            Log.d(TAG, data);
        } catch (FileNotFoundException e) {
            Log.d(TAG, e.toString());
        } catch (IOException e) {
            Log.d(TAG, e.toString());
        }
    }
```

This method provides robust functionality for saving data to a file on Android external storage. It ensures file integrity by handling exceptions and provides user feedback through toast messages, enhancing usability. The optional timestamp feature supports organizing saved files chronologically, facilitating easier management of saved data.

```
 private String getTime(boolean need) {
        int minute, hour, second;
        String zone = "am";
        String time = "";

        Calendar calendar = Calendar.getInstance();
        minute = calendar.get(Calendar.MINUTE);
        hour = calendar.get(Calendar.HOUR_OF_DAY);
        if (hour >= 12) {
            zone = "pm";
        }
        if (hour > 12) {
            hour = hour % 12;
        }
        second = calendar.get(Calendar.SECOND);

        if(need)
            time = hour + ":" + minute + ":" + second + " " + zone;
        else
            time = hour + ":" + minute + " " + zone;

        return time;
    }
```

This method provides a convenient way to retrieve and format the current time for various application needs, such as timestamping messages or file names. It supports both 12-hour clock formatting and includes optional seconds for precise time representation, enhancing functionality and user experience within the application.

```
public void onSendFilesAndVoiceClicked(View view) {
        openFileandVoiceShareAlertDialog();

    }

    private void openFileandVoiceShareAlertDialog() {
        final AlertDialog.Builder alert = new AlertDialog.Builder(MainActivity.this);
        View mView = getLayoutInflater().inflate(R.layout.custom_file_voice_change_dialog,
null);
```

```
        Button btn_file = (Button) mView.findViewById(R.id.btn_file);
        Button btn_voice = (Button) mView.findViewById(R.id.btn_voice);

        alert.setView(mView);

        final AlertDialog alertDialog = alert.create();
        alertDialog.setCanceledOnTouchOutside(true);
        alertDialog.setTitle("File and Voice Sharing");


        btn_file.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                openStorage();
                alertDialog.dismiss();
            }
        });

        btn_voice.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                takeVoiceForHim();
                alertDialog.dismiss();
            }
        });

        alertDialog.show();
    }

    private void openStorage() {

        Intent intent = new
Intent().setType("text/plain").setAction(Intent.ACTION_GET_CONTENT);

        startActivityForResult(Intent.createChooser(intent, "Select a TXT file"), 123);

    }
```

The `onSendFilesAndVoiceClicked` method in the `MainActivity` class is triggered when the corresponding button is clicked in the UI. It invokes the `openFileandVoiceShareAlertDialog` method, which displays an alert dialog allowing the user to choose between file sharing and voice messaging options. Here's an overview of its functionality:

Method Descriptions

`onSendFilesAndVoiceClicked

This method serves as an event handler for the button click (`View view` parameter). It simply calls `openFileandVoiceShareAlertDialog()` to initiate the process of displaying a dialog box for file and voice sharing options.

`openFileandVoiceShareAlertDialog`

This private method constructs an `AlertDialog.Builder` instance (`alert`) using the current activity context (`MainActivity.this`). It inflates a custom layout (`custom_file_voice_change_dialog.xml`) containing two buttons:

File Button (`btn_file`): Opens device storage to select a TXT file.

Voice Button (`btn_voice`): Initiates voice messaging functionality.

The dialog's title is set to "File and Voice Sharing", and clicking either button triggers specific actions:

File Button Click: Calls `openStorage()` to prompt the user to select a TXT file from storage via `Intent.ACTION_GET_CONTENT`.

Voice Button Click: Executes `takeVoiceForHim()` to initiate voice messaging functionality.

`openStorage`

This method creates an `Intent` to prompt the user to select a file (`text/plain` type) using `Intent.ACTION_GET_CONTENT`. It launches a file chooser dialog using `startActivityForResult`, displaying options to select a TXT file.

Usage and Interaction

Together, these methods provide a user-friendly interface for initiating either file sharing or voice messaging within the application. They streamline user interaction by presenting clear options through an alert dialog and seamlessly integrate Android's file selection capabilities (`Intent.ACTION_GET_CONTENT`) for selecting TXT files. This enhances the application's functionality by enabling users to easily share files or initiate voice messages directly from the app interface, thereby improving usability and user experience.

```
 private void takeVoiceForHim() {
        Intent intent = new Intent(RecognizerIntent.ACTION_RECOGNIZE_SPEECH);
        intent.putExtra(RecognizerIntent.EXTRA_LANGUAGE_MODEL,
RecognizerIntent.LANGUAGE_MODEL_FREE_FORM);
        intent.putExtra(RecognizerIntent.EXTRA_LANGUAGE, Locale.getDefault());

        if (intent.resolveActivity(getPackageManager()) != null) {
            startActivityForResult(intent, 10);
        } else {
            Toast.makeText(this, "Your Device Doesn't Support Speech Input",
Toast.LENGTH_SHORT).show();
        }
    }
```

```
    @Override
    public void onActivityResult(int requestCode, int resultCode, Intent intent) {
        super.onActivityResult(requestCode, resultCode, intent);

        if(requestCode==123 && resultCode==RESULT_OK) {
            Uri uri = intent.getData();
            String path = getFilePathFromUri(uri);
            File file = new File(path);
            if(file.exists())
                Log.d(TAG, "Selected file already exists");


            String fileText = readTextFile(uri);
            Log.d(TAG, "text inside file: "+fileText);
            String writeMsg = "file@%@"+file.getName()+"@%@"+fileText;
            sendFilesALertDialog(writeMsg);
        }
```

These methods together provide comprehensive functionality for both voice input and file selection within the application, enhancing usability and user experience by integrating essential device features seamlessly.

```
    private void sendFilesALertDialog(String writeMsg) {
        final AlertDialog.Builder alert = new AlertDialog.Builder(MainActivity.this);
        View mView = getLayoutInflater().inflate(R.layout.custom_disconnect_dialog, null);

        TextView textView = mView.findViewById(R.id.custom_disconnect_dialog_textView);
        textView.setText("Are you sure you want to send file?");
        Button btn_cancel = (Button) mView.findViewById(R.id.btn_cancel);
        Button btn_yes = (Button) mView.findViewById(R.id.btn_yes);

        alert.setView(mView);

        final AlertDialog alertDialog = alert.create();
        alertDialog.setCanceledOnTouchOutside(false);
        alertDialog.setTitle("Sending File");


        btn_cancel.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Toast.makeText(MainActivity.this, "Cancelled", Toast.LENGTH_SHORT).show();
                alertDialog.dismiss();
            }
        });


        btn_yes.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                sendReceive.write(caesarCipherEncryption(writeMsg, shift));
                alertDialog.dismiss();
            }
```

```
        });

        alertDialog.show();
    }
```

Usage

This method provides a user-friendly confirmation dialog for sending files within the application. It ensures user consent before proceeding with potentially critical operations, enhancing user control and preventing accidental actions. By integrating this dialog, the application facilitates clear and secure communication of files between users, promoting usability and reliability.

Integration with Encryption

The method also demonstrates integration with encryption (caesarCipherEncryption(writeMsg, shift)), ensuring that sensitive data (like file content) is securely transmitted over the communication channel, maintaining confidentiality and integrity in file sharing operations.

```
 private String readTextFile(Uri uri){
        BufferedReader reader = null;
        StringBuilder builder = new StringBuilder();
        try {
            reader = new BufferedReader(new
InputStreamReader(getContentResolver().openInputStream(uri)));
            String line = "";


            while ((line = reader.readLine()) != null) {
                builder.append("\n" + line);
            }

        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (reader != null){
                try {
                    reader.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
        return builder.toString();
    }
```

Usage

This method is crucial for retrieving the textual content of selected files, enabling the application to process and manipulate file data as required. It supports various file operations

within the application, such as displaying file contents, sending file data over networks, or saving file contents locally.

Considerations

- **Efficiency**: Uses buffered reading (BufferedReader) for optimized performance when reading potentially large files.
- **Error Handling**: Implements robust error handling to manage exceptions during file reading, ensuring the application remains stable and resilient.
- **Integration**: Typically used in conjunction with file selection mechanisms (Intent.ACTION_GET_CONTENT) to handle user-selected files for processing within the application.

Overall, the readTextFile method facilitates seamless file handling functionality, enhancing the application's capability to manage and utilize textual data from user-selected files effectively.

```
 private String getFilePathFromUri(Uri uri){
        String path = uri.getPathSegments().get(1);
        path = Environment.getExternalStorageDirectory().getPath()+"/"+path.split(":")[1];
        return path;
    }

    private void openDisconnectAlertDialogBox() {
        final AlertDialog.Builder alert = new AlertDialog.Builder(MainActivity.this);
        View mView = getLayoutInflater().inflate(R.layout.custom_disconnect_dialog, null);

        Button btn_cancel = (Button) mView.findViewById(R.id.btn_cancel);
        Button btn_yes = (Button) mView.findViewById(R.id.btn_yes);

        alert.setView(mView);

        final AlertDialog alertDialog = alert.create();
        alertDialog.setCanceledOnTouchOutside(false);
        alertDialog.setTitle("Disconnecting");

        btn_cancel.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Toast.makeText(MainActivity.this, "Cancelled", Toast.LENGTH_SHORT).show();
                alertDialog.dismiss();
            }
        });

        btn_yes.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                sendReceive.write(caesarCipherEncryption("diconnect@%@d", shift));
                Log.d(TAG, "Disconnect Msg: " +caesarCipherEncryption("diconnect@%@d",
shift));
                disconnectHim();
                alertDialog.dismiss();
            }
```

```
        });

        alertDialog.show();
    }
```

Usage

This method provides a user-friendly confirmation dialog before executing critical operations like disconnecting from a network or terminating a communication session. It ensures that users have an opportunity to confirm their actions, preventing accidental disconnections and enhancing application usability.

Integration with Encryption

The method demonstrates integration with encryption (caesarCipherEncryption(...)) to secure the disconnect message transmitted over the network or communication channel. This ensures that sensitive commands or data related to disconnection are protected from unauthorized access or tampering.

Considerations

- **User Experience**: Enhances user interaction by prompting for confirmation before executing potentially irreversible actions.
- **Security**: Incorporates encryption to safeguard communication integrity and confidentiality during critical operations.
- **Feedback**: Provides visual feedback (toast messages) and logging (Log.d(...)) to inform users and developers about the progress and outcome of the disconnection process.

In summary, openDisconnectAlertDialogBox contributes to a robust and user-friendly interface for managing network operations within the application, prioritizing both security and usability aspects.

```
public void disconnectHim() {
        try {
            clientClass.socket.close();
            serverClass.socket.close();


            thirdLayout.setVisibility(View.GONE);
            mToolbar.setBackgroundColor(Color.parseColor("#233E4E"));
            thirdLayout.setBackgroundResource(R.drawable.background3);
            changeBG.setEnabled(false);
            saveChat.setEnabled(false);
            disconnect.setEnabled(false);
            getSupportActionBar().setTitle("Chit Chat");
            firstLayout.setVisibility(View.VISIBLE);

            targetIPEditText.setVisibility(View.INVISIBLE);
            targetPortEditText.setVisibility(View.INVISIBLE);
            connectBtn.setVisibility(View.INVISIBLE);
            clickHereBtn.setVisibility(View.INVISIBLE);
```

```
            getIPBtn.setVisibility(View.INVISIBLE);
            encrypKeyEditText.setVisibility(View.INVISIBLE);

            Toast.makeText(MainActivity.this, "chat is disconnected",
Toast.LENGTH_SHORT).show();
        } catch (IOException e) {
            e.printStackTrace();
            Log.d(TAG, "ERROR/n"+e);
        }
    }
```

Usage

This method is typically called after the user confirms a disconnection action through an alert dialog or similar user interaction. It ensures that network resources are properly released and UI elements are adjusted to reflect the disconnected state. By providing visual feedback and adjusting UI visibility, it enhances user experience by clearly indicating the status of the communication session.

Error Handling

The method includes a try-catch block to handle IOException, which may occur if there are issues with socket closure. In case of an exception, it logs the error (Log.d(TAG, "ERROR/n"+e)) and prints the stack trace (e.printStackTrace()), helping developers diagnose and troubleshoot connectivity issues.

Considerations

- **Resource Management**: Ensures proper closure of network resources (sockets) to free up system resources and prevent memory leaks.
- **User Interface**: Adjusts UI visibility and state to provide clear visual cues of the disconnected status to the user.
- **Error Handling**: Implements error handling to manage potential exceptions that may arise during the disconnection process, maintaining application stability.

In summary, disconnectHim plays a crucial role in managing the application's network lifecycle, ensuring a smooth transition to a disconnected state while maintaining a responsive and informative user interface.

```
public void onStartServerClicked(View v){
        String port = receivePortEditText.getText().toString();
        if(TextUtils.isEmpty(port)){
            receivePortEditText.requestFocus();
            receivePortEditText.setError("Please write your receive port first");
        }


        else{
            try{
                serverClass = new ServerClass(Integer.parseInt(port));
                serverClass.start();
```

```
                Toast.makeText(this, "Server has been started..",
Toast.LENGTH_SHORT).show();

                targetIPEditText.setVisibility(View.VISIBLE);
                targetPortEditText.setVisibility(View.VISIBLE);
                connectBtn.setVisibility(View.VISIBLE);
                clickHereBtn.setVisibility(View.VISIBLE);
                getIPBtn.setVisibility(View.VISIBLE);
                encrypKeyEditText.setVisibility(View.VISIBLE);

            }catch (Exception e){
                Toast.makeText(MainActivity.this, "Can't start server, please check the
port number first", Toast.LENGTH_SHORT).show();
            }
        }

    }
```

onStartServerClicked encapsulates the logic to start a server, validate input, provide feedback to the user, and adjust the UI accordingly. It forms a critical part of the application's functionality for establishing server connections and ensuring smooth user interaction throughout the process.

```
  public void onConnectClicked(View v){

        String port = targetPortEditText.getText().toString();
        String tergetIP = targetIPEditText.getText().toString();
        String encrypKey = encrypKeyEditText.getText().toString();
        shift = Integer.parseInt(encrypKey);

        if(TextUtils.isEmpty(port)){
            targetPortEditText.requestFocus();
            targetPortEditText.setError("Please write your target port first");
        }


        else if(tergetIP.equals(ip)){
            targetIPEditText.requestFocus();
            targetIPEditText.setError("This is your self IP, please change it");
        }

        else{
            try{
                clientClass = new ClientClass(tergetIP, Integer.parseInt(port));
                clientClass.start();
                Toast.makeText(MainActivity.this, "your sending port and listening port
has been set successfully", Toast.LENGTH_SHORT).show();
            }catch (Exception e){
                Log.d(TAG, "ERROR: "+e);
                Toast.makeText(MainActivity.this, "Can't connect with server, please check
all the requirements", Toast.LENGTH_SHORT).show();
            }
        }
```

```
    }
```

The `onConnectClicked` method is triggered when a connection button is clicked in the app's user interface. It retrieves the user input for the target port, target IP, and encryption key from their respective EditText fields. The encryption key is converted into an integer for later use. The method checks if the port field is empty and prompts the user to fill it if necessary. It also ensures that the target IP is not the same as the device's own IP to avoid self-connection, providing an error message if this is the case. If both checks pass, the method attempts to create and start a `ClientClass` instance using the provided IP and port. If successful, it displays a success message; if an exception occurs, it logs the error and shows a failure message to the user.

```
public void onSendClicked(View v){
        String msg = messageEditText.getText().toString().trim();
        String msgTime = "@%@" + getTime(false);
        String msgWithTime = msg + msgTime;


        if(TextUtils.isEmpty(msg)){
            messageEditText.requestFocus();
            messageEditText.setError("Please write your message first");
        }
        else
        {

            sendReceive.write(caesarCipherEncryption(msgWithTime, shift));
        }

    }


    public void onGetIPClicked(View view) {

        ip = Utils.getIPAddress(true);
        openIPAlertDialog();
    }

    private void openIPAlertDialog() {
        final AlertDialog.Builder alert = new AlertDialog.Builder(MainActivity.this);
        View mView = getLayoutInflater().inflate(R.layout.custom_disconnect_dialog, null);

        TextView textView = mView.findViewById(R.id.custom_disconnect_dialog_textView);
        if( ip != null && ip != "")
            textView.setText("Your IP Add is : "+ ip);
        else
            textView.setText("Can't get your IP address, please check your connectionn");
        Button btn_cancel = (Button) mView.findViewById(R.id.btn_cancel);
        Button btn_yes = (Button) mView.findViewById(R.id.btn_yes);
        btn_cancel.setVisibility(View.GONE);
        btn_yes.setVisibility(View.GONE);
        alert.setView(mView);
```

```
        final AlertDialog alertDialog = alert.create();
        alertDialog.setCanceledOnTouchOutside(true);
        alertDialog.setTitle("IP Address");

        alertDialog.show();
    }

    public void onSVoiceIconClicked(View view) {
        takeVoiceForHim();
    }


    private class SendReceive extends Thread {
        private Socket socket;
        private InputStream inputStream;
        private OutputStream outputStream;

        public SendReceive(Socket skt) {
            socket = skt;
            try {
                inputStream = socket.getInputStream();
                outputStream = socket.getOutputStream();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

        @Override
        public void run() {
            byte[] buffer = new byte[1024];
            int bytes;

            while (socket != null) {
                try {
                    bytes = inputStream.read(buffer);
                    if (bytes > 0) {
                        handler.obtainMessage(MESSAGE_READ, bytes, -1,
buffer).sendToTarget();
                    }
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }

        public void write(String msg) {
            new Thread(() -> {
                try {
                    outputStream.write(msg.getBytes());
                    addMessage(Color.parseColor("#FCE4EC"), msg);
                    runOnUiThread(() ->
                            messageEditText.setText("")
                    );
```

```
                } catch (IOException e) {
                    Log.d(TAG, "Can't send message: " + e);
                } catch (Exception e) {
                    Log.d(TAG, "Error: " + e);
                }
            }).start();

        }
    }
```

The code provided includes methods for handling various button click events and a nested `SendReceive` class for managing socket communication.

1. `onSendClicked(View v)`: This method is triggered when the send button is clicked. It retrieves the message from the input field, appends a timestamp, and encrypts the message using a Caesar cipher before sending it. If the message field is empty, it prompts the user to enter a message.

2. `onGetIPClicked(View view)`: This method is called when the user clicks the button to get their IP address. It retrieves the device's IP address using a utility function and displays it in an alert dialog.

3. `openIPAlertDialog()`: This private method creates and displays an alert dialog showing the device's IP address or an error message if the IP address cannot be retrieved. It hides cancel and yes buttons for simplicity.

4. `onSVoiceIconClicked(View view)`: This method triggers voice input functionality, though its implementation is not provided in the given code.

5. `SendReceive` class: This nested class extends `Thread` and handles socket communication. It initializes input and output streams for the given socket. The `run()` method continuously reads data from the input stream and sends it to a handler if any data is received. The `write(String msg)` method sends an encrypted message through the output stream and updates the UI to clear the message input field after sending. If an error occurs during sending, it logs the error.

Overall, these methods and class manage message sending, IP address retrieval, and socket communication within the application.

```
public class ServerClass extends Thread {

        Socket socket;
        ServerSocket serverSocket;
        int port;

        public ServerClass(int port) {
            this.port = port;
        }

        @Override
        public void run() {
            try {


                serverSocket = new ServerSocket(port);
                Looper.prepare();
                showToast("Server Started. Waiting for client...");
                Log.d(TAG, "Waiting for client...");
                socket = serverSocket.accept();
                Log.d(TAG, "Connection established from server");
                sendReceive = new SendReceive(socket);
                sendReceive.start();
            } catch (IOException e) {
                e.printStackTrace();
                Log.d(TAG, "ERROR: " + e);
            } catch (Exception e) {
                Log.d(TAG, "ERROR: " + e);
            }
        }
    }


    public class ClientClass extends Thread {
        Socket socket;
        String hostAdd;
        int port;

        public ClientClass(String hostAddress, int port) {
            this.port = port;
            this.hostAdd = hostAddress;
        }
         Socket socket = null;
        try {
            socket = new Socket(SERVER_ADDRESS, SERVER_PORT);
            new ReadThread(socket).start();
            new WriteThread(socket).start();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private static class ReadThread extends Thread {
        private BufferedReader in;
```

```
        public ReadThread(Socket socket) throws IOException {
            in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        }

        public void run() {
            String message;
            try {
                while ((message = in.readLine()) != null) {
                    System.out.println(message);
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

The provided code includes two classes, `ServerClass` and `ClientClass`, which handle server and client operations for socket communication, along with a nested `ReadThread` class for reading data from the socket.

1. `ServerClass`: This class extends `Thread` and is responsible for setting up the server. It initializes the server on a specified port and waits for client connections. When a client connects, it creates an instance of the `SendReceive` class to manage communication. The `run()` method handles the core server logic, including starting the server socket, waiting for client connections, and logging relevant messages.

2. `ClientClass`: This class extends `Thread` and manages client-side operations. It connects to the server using the provided host address and port. The `run()` method (not fully provided in the code snippet) is supposed to establish a socket connection to the server and start separate threads for reading from and writing to the socket. The code snippet initializes a socket and starts `ReadThread` and `WriteThread` instances for handling input and output operations.

3. `ReadThread`: This nested class extends `Thread` and is responsible for reading messages from the socket's input stream. It continuously reads lines of text from the input stream and prints them. The constructor initializes a `BufferedReader` for the socket's input stream, and the `run()` method manages the reading loop, handling any IOExceptions that occur.

Overall, these classes manage the server-client connection and communication, with separate threads handling specific tasks like reading and writing data through the socket.

```
        @Override
        public void run() {
            try {
                socket = new Socket(hostAdd, port);

                sendReceive = new SendReceive(socket);
                sendReceive.start();
                showToast("Connected to other device. You can now exchange messages.");

                Log.d(TAG, "Client is connected to server");


                enableComponent();
            } catch (IOException e) {
                e.printStackTrace();
                Log.d(TAG, "Can't connect to server. Check the IP address and Port number
and try again: " + e);
            } catch (Exception e) {
                Log.d(TAG, "ERROR: " + e);
            }
        }
    }


    private void addMessage(int color, String message) {

        runOnUiThread(() -> {
                    TextView textView = new TextView(this);
                    TextView msgTime = new TextView(this);

                    if (color == Color.parseColor("#FCE4EC")
                            && !(caesarCipherDecryption(message,
shift).contains("bg@%@bg"))
                            && !(caesarCipherDecryption(message,
shift).contains("diconnect@%@d"))
                            && !(caesarCipherDecryption(message,
shift).contains("file@%@"))
                            && !(caesarCipherDecryption(message,
shift).contains("remove@%@"))) {
                        textView.setPadding(200, 20, 10, 10);

                        textView.setGravity(Gravity.RIGHT);
                        textView.setBackgroundResource(R.drawable.sender_messages_layout);
                        textView.setTextIsSelectable(true);

                        LinearLayout.LayoutParams lp1 = new
LinearLayout.LayoutParams(LinearLayout.LayoutParams.MATCH_PARENT,
LinearLayout.LayoutParams.WRAP_CONTENT);

                        lp1.leftMargin = 200;

                        textView.setLayoutParams(lp1);

                        msgTime.setPadding(0,0,0,0);
```

```
                    msgTime.setTextSize(14);
                    msgTime.setTextColor(Color.parseColor("#FCE4EC"));
                    msgTime.setTypeface(textView.getTypeface(), Typeface.ITALIC );
                    conversationLayout.setGravity(View.TEXT_ALIGNMENT_CENTER);
                    msgTime.setGravity(Gravity.LEFT);

                    LinearLayout.LayoutParams lp4 = new
LinearLayout.LayoutParams(LinearLayout.LayoutParams.MATCH_PARENT,
LinearLayout.LayoutParams.WRAP_CONTENT);

                    lp4.leftMargin = 200;
                    msgTime.setLayoutParams(lp4);


                }

                else if(!(caesarCipherDecryption(message, shift).contains("bg@%@bg"))
                        && !(caesarCipherDecryption(message,
shift).contains("diconnect@%@d"))
                        && !(caesarCipherDecryption(message,
shift).contains("file@%@"))
                        && !(caesarCipherDecryption(message,
shift).contains("remove@%@"))) {
                    textView.setPadding(10, 20, 200, 10);

                    textView.setGravity(Gravity.LEFT);
                    textView.setBackgroundResource(R.drawable.receiver_messages_layout
);
                    textView.setTextIsSelectable(true);

                    LinearLayout.LayoutParams lp2 = new
LinearLayout.LayoutParams(LinearLayout.LayoutParams.MATCH_PARENT,
LinearLayout.LayoutParams.WRAP_CONTENT);

                    lp2.rightMargin = 200;
                    textView.setLayoutParams(lp2);

                    msgTime.setTextSize(14);
                    msgTime.setTextColor(Color.parseColor("#FFFFFF"));
                    msgTime.setTypeface(textView.getTypeface(), Typeface.ITALIC);
                    conversationLayout.setGravity(View.TEXT_ALIGNMENT_CENTER);
                    LinearLayout.LayoutParams lp3 = new
LinearLayout.LayoutParams(LinearLayout.LayoutParams.MATCH_PARENT,
LinearLayout.LayoutParams.WRAP_CONTENT);

                    lp3.rightMargin = 200;
                    msgTime.setGravity(Gravity.RIGHT);
                    msgTime.setLayoutParams(lp3);


                }
                textView.setTextColor(color);
                Log.d(TAG, "encrypted msg: " + message);
```

```
                    String actualMessage = caesarCipherDecryption(message, shift);
                    Log.d(TAG, "decrypted msg: " + actualMessage);


                    String[]  messages = actualMessage.split("@%@", 0);


                    if(messages[0].equals("file")){
                        textView.setPadding(0,0,0,0);

                        textView.setTextSize(15);
                        textView.setTypeface(textView.getTypeface(), Typeface.BOLD);
                        conversationLayout.setGravity(View.TEXT_ALIGNMENT_CENTER);
                        textView.setGravity(Gravity.CENTER);


                        Log.d(TAG, "File Name: "+messages[1]);
                        Log.d(TAG, "File Contains:\n "+messages[2]);
                        if(color == Color.parseColor("#FCE4EC"))
                            textView.setText(messages[1]+" has been sent");
                        else{
                            textView.setText(messages[1]+" has been received and
downloaded on android/data/com.example.p2p/");
                            writeToFile(messages[2], false, messages[1]);
                        }
                    }

    }
}
```

The provided code includes updates to the `ClientClass` and `addMessage` methods for handling client connections and message display within a chat application.

1. **`ClientClass`**:

   - The `run()` method attempts to connect to a server using the specified host address and port. If successful, it initializes the `SendReceive` instance to handle message communication, starts the `SendReceive` thread, displays a toast message indicating the connection is established, and logs a success message. If an IOException or any other exception occurs, it logs the error and provides an appropriate message.

2. **`addMessage(int color, String message)`**:

   - This method is responsible for adding messages to the conversation layout on the UI. It runs on the UI thread to ensure thread safety for UI operations.

- Based on the message type and content, it sets the padding, gravity, background, and layout parameters for the message `TextView`.

- It uses a Caesar cipher decryption method to check the message content and determine its type (e.g., regular message, file transfer).

- Regular messages are displayed with different styling for sent and received messages.

- If the message is a file, it updates the `TextView` to indicate whether the file was sent or received. For received files, it saves the file content to the device.

- It also handles messages that do not include specific keywords like "bg@%@bg", "disconnect@%@d", "file@%@", or "remove@%@" to avoid processing these special messages.

Overall, these methods manage client connections, enable message exchange between devices, and display messages appropriately in the chat interface.

React Component Code Documentation:

```
import React from 'react';
import './ProjectComponent.css';

const ProjectComponent = ({ image, name, building, type, project, description }) => {
  return (
    <div className="project-component">
      <img src={image} alt={name} className="project-image" />
      <div className="project-details">
        <label>
          Name: <input type="text" value={name} readOnly />
        </label>
        <label>
          Building: <input type="text" value={building} readOnly />
        </label>
        <label>
          Type: <input type="text" value={type} readOnly />
        </label>
        <label>
          Project: <input type="text" value={project} readOnly />
        </label>
      </div>
      <p className="project-description">{description}</p>
    </div>
  );
};

export default ProjectComponent;
```

The `ProjectComponent` is a React functional component designed to display detailed information about a project. It takes in several props: `image`, `name`, `building`, `type`,

`project`, and `description`, and uses them to render the project's information in a structured format.

1. **Component Structure**:

   - The component is wrapped in a `div` with the class name `project-component`.

   - An `img` tag displays the project's image, sourced from the `image` prop, with an `alt` attribute set to the project's name.

   - A `div` with the class name `project-details` contains four labels, each paired with an `input` element that displays the respective prop values (`name`, `building`, `type`, and `project`). All inputs are set to read-only to prevent user modification.

   - A `p` tag with the class name `project-description` displays the project's description.

2. **Styling**:

   - The component relies on an external CSS file, `ProjectComponent.css`, for styling. The CSS classes (`project-component`, `project-image`, `project-details`, and `project-description`) are used to apply specific styles to the component's elements.

3. **Usage**:

   - To use this component, you need to import it and provide the necessary props. For example:

     ```jsx
     <ProjectComponent
       image="path/to/image.jpg"
       name="Project Name"
       building="Building Name"
       type="Project Type"
       project="Project ID"
       description="Project Description"
     />
     ```

```

Overall, the `ProjectComponent` neatly organizes and displays project-related information, making it a reusable and easy-to-integrate component in a React application.

```
import axios from 'axios';

const API_URL = 'https://jsonplaceholder.typicode.com/photos';

// Fetch project data
export const fetchProjects = async () => {
  try {
    const response = await axios.get(API_URL);
    return response.data;
  } catch (error) {
    console.error('Error fetching projects:', error);
    return [];
  }
};

// Create a new project
export const createProject = async (project) => {
  try {
    const response = await axios.post(API_URL, project);
    return response.data;
  } catch (error) {
    console.error('Error creating project:', error);
    return null;
  }
};

// Update a project
export const updateProject = async (id, project) => {
  try {
    const response = await axios.put(`${API_URL}/${id}`, project);
    return response.data;
  } catch (error) {
    console.error('Error updating project:', error);
    return null;
  }
};

// Delete a project
export const deleteProject = async (id) => {
  try {
    await axios.delete(`${API_URL}/${id}`);
    return true;
  } catch (error) {
    console.error('Error deleting project:', error);
    return false;
  }
```
```

```
};
```

The provided code defines a set of asynchronous functions for interacting with an API endpoint (`https://jsonplaceholder.typicode.com/photos`) to perform CRUD (Create, Read, Update, Delete) operations on project data. The operations are handled using the `axios` library to make HTTP requests.

1. **Fetch Project Data** (`fetchProjects`):

   - This function makes a GET request to the API URL to retrieve a list of projects.

   - If successful, it returns the project data.

   - If an error occurs, it logs the error to the console and returns an empty array.

2. **Create a New Project** (`createProject`):

   - This function accepts a `project` object as an argument and makes a POST request to the API URL to create a new project.

   - If successful, it returns the newly created project's data.

   - If an error occurs, it logs the error to the console and returns `null`.

3. **Update a Project** (`updateProject`):

   - This function accepts a project `id` and a `project` object as arguments and makes a PUT request to the API URL to update the specified project.

   - If successful, it returns the updated project's data.

   - If an error occurs, it logs the error to the console and returns `null`.

4. **Delete a Project** (`deleteProject`):

   - This function accepts a project `id` as an argument and makes a DELETE request to the API URL to delete the specified project.

   - If successful, it returns `true`.

   - If an error occurs, it logs the error to the console and returns `false`.

These functions provide a straightforward interface for performing CRUD operations on project data using the `axios` library to handle HTTP requests and responses.

ChatBots Java Code Documentation:

```
import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;
import java.util.regex.Pattern;

public class SimpleChatbot {

    private static Map<Pattern, String> patterns = new HashMap<>();

    static {
        patterns.put(Pattern.compile(".*\\b(hi|hello|hey)\\b.*",
Pattern.CASE_INSENSITIVE), "Hello! How can I help you?");
        patterns.put(Pattern.compile(".*\\b(how are you|how's it going)\\b.*",
Pattern.CASE_INSENSITIVE), "I'm just a bunch of code, but I'm doing great! How about
you?");
        patterns.put(Pattern.compile(".*\\b(what is your name|who are you)\\b.*",
Pattern.CASE_INSENSITIVE), "I'm SimpleChatbot, your virtual assistant.");
        patterns.put(Pattern.compile(".*\\b(bye|exit|goodbye)\\b.*",
Pattern.CASE_INSENSITIVE), "Goodbye! Have a great day!");
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        boolean keepChatting = true;

        System.out.println("Hello! I'm a simple chatbot. How can I assist you today?");

        while (keepChatting) {
            System.out.print("You: ");
            String userInput = scanner.nextLine();

            boolean matched = false;
            for (Map.Entry<Pattern, String> entry : patterns.entrySet()) {
                if (entry.getKey().matcher(userInput).find()) {
                    System.out.println("Chatbot: " + entry.getValue());
                    matched = true;
                    if (entry.getValue().equals("Goodbye! Have a great day!")) {
                        keepChatting = false;
                    }
                    break;
                }
            }

            if (!matched) {
                System.out.println("Chatbot: I'm sorry, I don't understand that. Can you
rephrase?");
```

```
            }
        }

        scanner.close();
    }
}
```

The `SimpleChatbot` class is a basic chatbot program written in Java. It uses regular expressions to match user input against predefined patterns and provides appropriate responses. The chatbot can recognize common greetings, inquiries about its well-being, questions about its identity, and farewells. Here's a summary of its functionality:

1. **Patterns and Responses**:

   - A `HashMap` named `patterns` stores pairs of `Pattern` objects (regular expressions) and corresponding response strings. The `Pattern` objects are created using predefined regular expressions that match specific keywords or phrases in user input.

   - The `patterns` map is initialized in a static block, ensuring it is populated when the class is loaded.

2. **Main Method**:

   - The `main` method starts by creating a `Scanner` object to read user input from the console.

   - A `while` loop runs until the user decides to exit by entering a farewell phrase. Inside the loop:

     - The chatbot prompts the user for input.

     - It then checks the input against each pattern in the `patterns` map.

     - If a match is found, the corresponding response is printed. If the response is a farewell message, the loop terminates.

     - If no patterns match, the chatbot replies with a default message indicating it didn't understand the input.

3. **Closing Resources**:

   - After the loop exits, the `Scanner` object is closed to release the associated resources.

Overall, the `SimpleChatbot` provides a basic framework for creating a text-based conversational agent that can respond to user inputs based on pattern matching.