Android application:
Tic Tac Toe

Documentation

Dobrin Cosmin-Iulian

Content:

1. Theme of the project

A simple game of tic tac toe, that can be played alone or with a friend.

This simple tic tac toe game is a great way to entertain yourself in a boring situation (like a long trip by train or car, or in a situation where you have to wait in line for some time). It can be played alone against a robot, or against a friend close to you. The robot can be set to play in different difficulty levels (easy/medium/hard), so that it can adapt to your style. The previous can be done from the settings menu, where you can also customize the game to your likings. The app game also comes with a language menu, where you can choose from 5 different languages: English(the default one), German, Spanish, French, Italian and Romanian.

2. Technology that was used

This application was developed using the Android Studio integrated development enviroment (IDE), which is an IDE developed for Google's Android operating system, built on JetBrains' IntelliJ IDEA software and designed specifically for Android development. The application was written in the Java programming language. The reason behind the choosing of this programming language over Kotlin stayed in the fact that I was already familiar with Java and I was much more confident using it. Furthermore it is a widely used language with a big community of developers, which results in a lot of support.

3. Explained code segments

The first activity that the user interacts with is the **„MenuActivity"**. The interface is plain and simple, containing four buttons that are self explanatory. The first two buttons („TWO PLAYERS" and „SINGLE PLAYER") will take the client to a different activity called „GameActivity", where the client can enjoy the actual tic tac toe game, the third button („SETTINGS") will take the client to the „SettingsActivity", where he/she can adjust the settings of the game to his likings, while the fourth button („LANGUAGE") will pop a dialog box which lets the client choose a different language. The interface consists in a .xml file and the functionality of the buttons is given through listeners. Down below is the code for all the listeners:

```
buttonLaunchTwoPlayers.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        launchTwoPlayers();
    }
});

buttonLaunchSinglePlayer.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        launchSinglePlayer();
    }
```

```
});

buttonLanguage.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        localeManager.showChangeLanguageDialog();
    }
});

buttonSettings.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        launchSettingsActivity();
    }
});
```

As it can be observed, each listener will execute a specific method whenever its button is clicked. The simplest of these methods is „launchSettingsActivity()", in which an intent to the „SettingsActivity" is instantiated, and then the intent is passed as a parameter to the „startActivity()" method.

```
private void launchSettingsActivity() {
    Intent intent = new Intent(this, SettingsActivity.class);
    startActivity(intent);
}
```

For the first two buttons the methods executed are very similar, the only difference laying in the „extra" that is used. That „extra" is carring information about the state of the single player mode, which can be either turned on or off. Other than that, both intents are pointed to the „GameActivity" class. Below is the code for the two methods:

```
private void launchTwoPlayers() {
    Intent intent = new Intent(this, GameActivity.class);
    intent.putExtra(EXTRA_SINGLE_PLAYER_MODE,
SettingsUtility.SINGLE_PLAYER_MODE_OFF);
    startActivity(intent);
}

private void launchSinglePlayer() {
    Intent intent = new Intent(this, GameActivity.class);
    intent.putExtra(EXTRA_SINGLE_PLAYER_MODE ,
SettingsUtility.SINGLE_PLAYER_MODE_ON);
    startActivity(intent);
}
```

The method executed after pressing the „LANGUAGE" button is a method that is defined in the „LocaleManager" class and it is accesed through an instance of that class. Using an array of strings and an „AlertDialog" builder, the menu for language choosing was created. Depending on the item chosen by the user, a language is set using the „setLocale()" method, also defined in the

„LocaleManager” class. The code for the method executed when „LANGUAGE” button is pressed:

```
public void showChangeLanguageDialog() {
    final String[] listItems = {activity.getString(R.string.english),
            activity.getString(R.string.german),
activity.getString(R.string.spanish),
            activity.getString(R.string.french),
activity.getString(R.string.italian),
            activity.getString(R.string.romanian)};

    AlertDialog.Builder mBuilder = new AlertDialog.Builder(context);
    mBuilder.setTitle(R.string.choose_language);
    mBuilder.setSingleChoiceItems(listItems, -1, new
DialogInterface.OnClickListener() {
        @Override
        public void onClick(DialogInterface dialogInterface, int i) {
            if(i == 0) {
                setLocale("en");
                activity.recreate();
            } else if(i == 1) {
                setLocale("de");
                activity.recreate();
            } else if(i == 2) {
                setLocale("es");
                activity.recreate();
            } else if(i == 3) {
                setLocale("fr");
                activity.recreate();
            } else if(i == 4) {
                setLocale("it");
                activity.recreate();
            } else if(i == 5) {
                setLocale("ro");
                activity.recreate();
            }

            dialogInterface.dismiss();
        }
    });

    AlertDialog mDialog = mBuilder.create();
    mDialog.show();
}
```

These four listeners are part of a bigger method called „setUpMenu()”, which has the role to set up the functionality of the menu activity. In this method, the four buttons are instantiated:

```
Button buttonLaunchTwoPlayers = findViewById(R.id.button_start);
Button buttonLanguage = findViewById(R.id.button_language);
Button buttonSettings = findViewById(R.id.button_settings);
Button buttonLaunchSinglePlayer = findViewById(R.id.button_start_robot);
```

Then the „setOnSystemUiVisibilityChangeListener()” is defined, with the role to hide the system bars if the user chooses so from the „SettingsActivity()”.

```
decorView.setOnSystemUiVisibilityChangeListener(new
View.OnSystemUiVisibilityChangeListener() {
    @Override
    public void onSystemUiVisibilityChange(int visibility) {
        if (mHideSystemBars == SettingsUtility.HIDE_SYSTEM_BARS) {
            if (visibility == 0) {
                decorView.setSystemUiVisibility(hideSystemBars());
            }
        }
    }
});
```

Before the definition of the listeners, the method loadSettings is called, so that the system bar settings done by the user are loaded. This method is defined outside this method by creating an instance of the class „SharedPreferences” that refers to the settings done by the user. Using this instance, the flag that corresponds to the state of the system bars is fetched and used to initialize the „mHideSystemBars” member variable.

```
private void loadSettings() {
    SharedPreferences preferences =
getSharedPreferences(SettingsUtility.PREFS_FILE_SETTINGS, Context.MODE_PRIVATE);
    mHideSystemBars =
preferences.getInt(SettingsUtility.PREFS_HIDE_SYSTEM_BARS_VALUE, 0);
}
```

Besides the „setUpMenu()”, the „onWindowFocusChange()” method can be observed. This method dictates what happens when the window focus gets changed. If the system bars are hidden, unfocusing the window will result in system bars showing.

```
@Override
public void onWindowFocusChanged(boolean hasFocus) {
    super.onWindowFocusChanged(hasFocus);
    if (mHideSystemBars == SettingsUtility.HIDE_SYSTEM_BARS) {
        if (hasFocus) {
            decorView.setSystemUiVisibility(hideSystemBars());
        }
    } else {
        if (hasFocus) {
            decorView.setSystemUiVisibility(showSystemBars());
        }
    }
}
```

The **„LocaleManager”** class, in which the previously mentioned method „showChangeLanguageDialog()” was defined, also consists of two other important methods.

One of them is the „setLocale()” method, which receives as argument the string representing a language and sets it using the „Locale” and „Configuration” classes. Afterwards, the language received as parameter is saved in a shared preferences file.

```
private void setLocale(String language) {
    Locale locale = new Locale(language);
    Locale.setDefault(locale);
    Configuration config = new Configuration();
    config.locale = locale;
    activity.getBaseContext().getResources().updateConfiguration(config,
            activity.getBaseContext().getResources().getDisplayMetrics());

    SharedPreferences.Editor editor = context.getSharedPreferences("Settings",
            Context.MODE_PRIVATE).edit();
    editor.putString("My_Lang", language);
    editor.apply();
}
```

The other method is the „loadLocale()” method, which is used as a convenient way to set the locale to the one stored in the shared preferences file. It is usually used in the „onCreate()” method, so that the activity will load in the preferred language.

```
public void loadLocale() {
    SharedPreferences prefs = context.getSharedPreferences("Settings",
            Context.MODE_PRIVATE);
    String language = prefs.getString("My_Lang", "");
    setLocale(language);
}
```

The **„SettingsActivity”**, besides the system bars related methods that were already discussed, contains three other important methods.

The first one is the „saveSettings()” method. It has the role of saving the user settings in a shared preferences file. This is done by instantiating the „Editor” class that extends the „SharedPreferences” class and then, using the editor and the „SettingsUtility” (a class that contains flags), the client's choice of settings is saved in the shared preferences file in a key-value format.

```
private void saveSettings() {
    SharedPreferences.Editor editor =
this.getSharedPreferences(SettingsUtility.PREFS_FILE_SETTINGS,
Context.MODE_PRIVATE).edit();
    editor.putInt(SettingsUtility.PREFS_START_BUTTON_ID,
mRadioGroupStart.getCheckedRadioButtonId());
    editor.putInt(SettingsUtility.PREFS_DRAW_BUTTON_ID,
mRadioGroupDraw.getCheckedRadioButtonId());
    editor.putInt(SettingsUtility.PREFS_SYMBOL_BUTTON_ID,
mRadioGroupSymbol.getCheckedRadioButtonId());
    editor.putInt(SettingsUtility.PREFS_PLAYER1_BUTTON_ID,
```

```
mRadioGroupPlayer1.getCheckedRadioButtonId());
    editor.putInt(SettingsUtility.PREFS_DIFFICULTY_BUTTON_ID,
mRadioGroupDifficulty.getCheckedRadioButtonId());

    editor.putInt(SettingsUtility.PREFS_WHO_STARTS_VALUE, gameEngine.getWhoStarts());
    editor.putInt(SettingsUtility.PREFS_WHO_STARTS_DRAW_VALUE,
gameEngine.getWhoStartsDraw());
    editor.putInt(SettingsUtility.PREFS_SYMBOL_VALUE, gameEngine.getSymbolPlayer1());
    editor.putInt(SettingsUtility.PREFS_PLAYER1_VALUE, gameEngine.getWhoIsPlayer1());
    editor.putInt(SettingsUtility.PREFS_DIFFICULTY_VALUE,
gameEngine.getDifficultyLevel());
    editor.putBoolean(SettingsUtility.PREFS_SWITCH_VALUE,
mSwitchSystemBars.isChecked());
    editor.putInt(SettingsUtility.PREFS_HIDE_SYSTEM_BARS_VALUE, mHideSystemBars);
    editor.apply();
}
```

The second method is „loadSettings()", which is used to load the settings done by the user from the shared preferences file. This is done by instantiating the „SharedPreferences" class, then fetching the saved identifiers and checking the corresponding radio buttons.

```
private void loadSettings() {
    SharedPreferences preferences =
this.getSharedPreferences(SettingsUtility.PREFS_FILE_SETTINGS, Context.MODE_PRIVATE);
    mRadioGroupStart.check(preferences.getInt(SettingsUtility.PREFS_START_BUTTON_ID,
R.id.radio_button_winner));
    mRadioGroupDraw.check(preferences.getInt(SettingsUtility.PREFS_DRAW_BUTTON_ID,
R.id.radio_button_draw_other));

mRadioGroupSymbol.check(preferences.getInt(SettingsUtility.PREFS_SYMBOL_BUTTON_ID,
R.id.radio_button_symbol_X));

mRadioGroupPlayer1.check(preferences.getInt(SettingsUtility.PREFS_PLAYER1_BUTTON_ID,
R.id.radio_button_you));

mRadioGroupDifficulty.check(preferences.getInt(SettingsUtility.PREFS_DIFFICULTY_BUTTO
N_ID, R.id.radio_button_easy));

    mHideSystemBars =
preferences.getInt(SettingsUtility.PREFS_HIDE_SYSTEM_BARS_VALUE, 0);

mSwitchSystemBars.setChecked(preferences.getBoolean(SettingsUtility.PREFS_SWITCH_VALU
E, false));
}
```

The last method is „setUpSettings”, in which the listeners for the radio button groups are set and afterwards the „loadSettings()” method is called, so that the radio button groups are prechecked with last saved preferences of the user. The implementations of the listeners are similar. They all override the „onCheckedChanged()” method, and implement a switch inside, which sets the values of the variables from the class „GameEngie” according to the radio button chosen by the user. An example can be seen below:

```
mRadioGroupStart.setOnCheckedChangeListener(new RadioGroup.OnCheckedChangeListener()
{
    @Override
    public void onCheckedChanged(RadioGroup group, int checkedId) {
        switch (checkedId) {
            case R.id.radio_button_winner:
                gameEngine.setWhoStarts(SettingsUtility.WINNER_STARTS);
                break;
            case R.id.radio_button_different:
                gameEngine.setWhoStarts(SettingsUtility.DIFFERENT_PLAYER_STARTS);
                break;
        }
    }
});
```

Moreover, before the implementations of the listeners, the buttons are linked to the corresponding views from the .xml file, through the „findViewById()” method:

```
mRadioGroupStart = findViewById(R.id.radio_group_who_starts);
mRadioGroupDraw = findViewById(R.id.radio_group_who_starts_case_draw);
mRadioGroupSymbol = findViewById(R.id.radio_group_symbol_chooser);
mRadioGroupPlayer1 = findViewById(R.id.radio_group_player1_chooser);
mRadioGroupDifficulty = findViewById(R.id.radio_group_difficulty_chooser);
mSwitchSystemBars = findViewById(R.id.switch_show_system_bars);
```

The activity where all the action takes place is the **„GameActivity”**. Besides the methods related to the system bars, that were already discussed, there are several important methods that will be briefly explained below.

The method „showPlayerSymbol()” has the role to update the textView dedicated to player 1 with the symbol chosen by the user for player 1, player 2 receiving the other symbol. Furthermore, player 1 will have the text color set to black, while player 2 will have the text color set to gray, to emphasise which player’s turn is.

```
private void showPlayerSymbol() {
        if (gameEngine.getSymbolPlayer1() == SettingsUtility.X) {
            textViewSymbolPlayer1.setText("X");
            textViewSymbolPlayer2.setText("O");
        } else if (gameEngine.getSymbolPlayer1() == SettingsUtility.O) {
            textViewSymbolPlayer1.setText("O");
            textViewSymbolPlayer2.setText("X");
        }
        textViewSymbolPlayer1.setTextColor(Color.BLACK);
```

```
        textViewSymbolPlayer2.setTextColor(Color.GRAY);
}
```

The method „highlightWhoStarts()" will reverse the colors of the textViews belonging to the players, with the purpose of keeping the user aware of the player that has to make a move.

```
private void highlightWhoStarts() {
    if (gameEngine.getPlayer1Turn()) {
        textViewSymbolPlayer1.setTextColor(Color.BLACK);
        textViewSymbolPlayer2.setTextColor(Color.GRAY);
    } else {
        textViewSymbolPlayer1.setTextColor(Color.GRAY);
        textViewSymbolPlayer2.setTextColor(Color.BLACK);
    }
}
```

The method „updateTextPoints()" updates the textViews corresponding to the score of each player with the current value from the „GameEngine" instance.

```
private void updatePointsText() {
    textViewPlayer1.setText(getResources().getString(R.string.player_1_pointsText,
            gameEngine.getPlayer1Points())); // String resources + placeholders
    textViewPlayer2.setText(getResources().getString(R.string.player_2_pointsText,
            gameEngine.getPlayer2Points()));
}
```

The method „loadButtonsText()" is used to fetch the text from each button and store it in an array called „field", which will then be returned.

```
private String[][] loadButtonsText() {

    String[][] field = new String[3][3];

    for(int i = 0; i < 3; i++) {
        for(int j = 0; j < 3; j++) {
            field[i][j] = buttons[i][j].getText().toString();
        }
    }

    return field;
}
```

The method „updateButtonsText(String[][] gameTable)" is used to update the text on the buttons with the current values. This method is heavily used in the „botMove()" method, to update the board after the bot make its move.

```
private void updateButtonsText(String[][] gameTable) {
    for(int i = 0; i < 3; i++) {
        for(int j = 0; j < 3; j++) {
            buttons[i][j].setText(gameTable[i][j]);
        }
    }
}
```

The method „cleanBoard()" is used to clean the board. It iterates through all the buttons and sets the text to „".

```
private void cleanBoard() {
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            buttons[i][j].setText("");
        }
    }
}
```

The method „showToast()" is used to show a toast message that indicates the result after the end of a game. It verifies if the game was either a win for player 1, a win for player 2, or a draw and shows an appropriate toast for each case.

```
private void showToast() {
    if (gameEngine.getPlayer1Wins() && (!gameEngine.getPlayer2Wins())) {
        Toast.makeText(getApplicationContext(),
                getResources().getString(R.string.player_1_wins),
Toast.LENGTH_SHORT).show();
    } else if (gameEngine.getPlayer2Wins() && (!gameEngine.getPlayer1Wins())) {
        Toast.makeText(getApplicationContext(),
                getResources().getString(R.string.player_2_wins),
Toast.LENGTH_SHORT).show();
    } else if((!gameEngine.getPlayer1Wins()) && (!gameEngine.getPlayer2Wins())) {
        Toast.makeText(getApplicationContext(),
                getResources().getString(R.string.draw), Toast.LENGTH_SHORT).show();
    }
}
```

A very important method is the „gameButtonClicked(View v)" which dictates what is going to happen when a button is pressed. Firstly, the pressed button will be checked to see if it is empty.

Secondly, a decision is taken regarding the symbol that is going to be used for writing on the button. After the decision is made, it will be decided which player's turn is and according to that information, the button will be checked with either a „X" or a „O". Afterwards, the round count will be updated and the round result will be checked through the „GameEngine" instance. Finally, the next player to move will have its textView box highlighted and then the „botMove()" will be called.

```
private void gameButtonClicked(View v) {
    if (!((Button) v).getText().toString().equals("")) {
        return;
    }

    if (gameEngine.getSymbolPlayer1() == SettingsUtility.X) {
        if (gameEngine.getPlayer1Turn()) {
            ((Button) v).setText("X");
        } else {
            ((Button) v).setText("O");
        }
    } else if (gameEngine.getSymbolPlayer1() == SettingsUtility.O) {
        if (gameEngine.getPlayer1Turn()) {
            ((Button) v).setText("O");
        } else {
            ((Button) v).setText("X");
        }
    }

    gameEngine.updateRoundCount();
    gameEngine.roundResult(loadButtonsText());
    highlightWhoStarts();

    botMove();
}
```

Another very important method is the „botMove()" method. This method is intended to interpret the state of the game and the settings done by the user, to make a move accordingly. This process is backed up by two classes that hold the logic, called „BotEngine" and „GameEngine". In this method a lot of decisions are being made, depending on the gameplay mode (in case of two players the bot will no longer be needed), followed by the difficulty level, who is player 1 and finally who's turn is it. After these ascertainments, a similar set of instructions will run. The game board will be updated with the move done by the bot through a method from the „BotEngine" class, called through the „BotEngine" instance. Afterwards, the round count will be updated and the round result will be checked through the „GameEngine" class instance. Finally the textView of the player that is next to make a move is highlighted.

```
private void botMove() {
    if (mSinglePlayerMode) {
        BotEngine botEngine = new BotEngine(gameEngine);
        if (gameEngine.getDifficultyLevel() == SettingsUtility.DIFFICULTY_LEVEL_HARD) {
            if (gameEngine.getWhoIsPlayer1() == SettingsUtility.BOT_IS_PLAYER_1) {
                    if (gameEngine.getPlayer1Turn()) {

                        updateButtonsText(botEngine.botP1HardMoves(loadButtonsText()));
                        updateButtonsText(botEngine.botP2HardMoves(loadButtonsText()));

                        gameEngine.updateRoundCount();
                        gameEngine.roundResult(loadButtonsText());
                        highlightWhoStarts();
                    }
                } else if (gameEngine.getWhoIsPlayer1() == SettingsUtility.YOU_ARE_PLAYER_1) {
                    if (!gameEngine.getPlayer1Turn()) {

                        updateButtonsText(botEngine.botP2HardMoves(loadButtonsText()));
```

```
                    updateButtonsText(botEngine.botP1HardMoves(loadButtonsText()));

                    gameEngine.updateRoundCount();
                    gameEngine.roundResult(loadButtonsText());
                    highlightWhoStarts();
                }
            }
        } else if (gameEngine.getDifficultyLevel() == SettingsUtility.DIFFICULTY_LEVEL_MEDIUM) {
            if (gameEngine.getWhoIsPlayer1() == SettingsUtility.BOT_IS_PLAYER_1) {
                if (gameEngine.getPlayer1Turn()) {

                    updateButtonsText(botEngine.botP1MediumMoves(loadButtonsText()));
                    updateButtonsText(botEngine.botP2MediumMoves(loadButtonsText()));

                    gameEngine.updateRoundCount();
                    gameEngine.roundResult(loadButtonsText());
                    highlightWhoStarts();
                }
            } else if (gameEngine.getWhoIsPlayer1() == SettingsUtility.YOU_ARE_PLAYER_1) {
                if (!gameEngine.getPlayer1Turn()) {

                    updateButtonsText(botEngine.botP2MediumMoves(loadButtonsText()));
                    updateButtonsText(botEngine.botP1MediumMoves(loadButtonsText()));

                    gameEngine.updateRoundCount();
                    gameEngine.roundResult(loadButtonsText());
                    highlightWhoStarts();
                }
            }
        } else if (gameEngine.getDifficultyLevel() == SettingsUtility.DIFFICULTY_LEVEL_EASY) {
            if (gameEngine.getWhoIsPlayer1() == SettingsUtility.BOT_IS_PLAYER_1) {
                if (gameEngine.getPlayer1Turn()) {

                    updateButtonsText(botEngine.botP1EasyMoves(loadButtonsText()));
                    updateButtonsText(botEngine.botP2EasyMoves(loadButtonsText()));

                    gameEngine.updateRoundCount();
                    gameEngine.roundResult(loadButtonsText());
                    highlightWhoStarts();
                }
            } else if (gameEngine.getWhoIsPlayer1() == SettingsUtility.YOU_ARE_PLAYER_1) {
                if (!gameEngine.getPlayer1Turn()) {

                    updateButtonsText(botEngine.botP2EasyMoves(loadButtonsText()));
                    updateButtonsText(botEngine.botP1EasyMoves(loadButtonsText()));

                    gameEngine.updateRoundCount();
                    gameEngine.roundResult(loadButtonsText());
                    highlightWhoStarts();
                }
            }
        }
    }
}
```

The method „setUpGame()”, as the name suggests is used to set up the game activity. This is done by assigning the views from the .xml file to the attributes from the „GameActivity” class through the identifiers.

```
textViewPlayer1 = findViewById(R.id.text_view_p2);
textViewPlayer2 = findViewById(R.id.text_view_p1);
textViewSymbolPlayer1 = findViewById(R.id.text_view_symbol_1);
textViewSymbolPlayer2 = findViewById(R.id.text_view_symbol_2);
```

```
buttons = new Button[3][3];
Button buttonReset = findViewById(R.id.button_reset);
```

An „onClickListener” is set for the reset button, calling the „resetGame()” method through the „GameEngine” instance and the „highlightWhoStarts()” method after.

```
buttonReset.setOnClickListener( new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        gameEngine.resetGame();
        highlightWhoStarts();
    }
});
```

The „setOnSystemUiVisibilityChangeListener()” is also called, to hide the system bars, as explained earlier.

The buttons that represent the game board exist in the form of a matrix of three lines and three columns. To easily assign the views by identifiers to the elements of the matrix, a nested for loop is used. Inside the loop a string that represents the current button's identifier is initialized. Afterwards, the identifier is found through the „getResources.getIdentifier()” method which receives as arguments the earlier created string, the type it is looking for and the package name. The id is then stored as an integer and assigned to the current element of the „buttons” array. After that, a listener is set on the specific button, calling the „gameButtonClicked()” method when it detects a click.

```
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        String buttonId = "button_" + i + j;
        int resId = getResources().getIdentifier(buttonId, "id", getPackageName());
        buttons[i][j] = findViewById(resId);
        buttons[i][j].setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                 gameButtonClicked(v);
            }
        });

        int autoSizeMaxTextSize = getResources().getInteger(R.integer.size);

        TextViewCompat.setAutoSizeTextTypeUniformWithConfiguration(buttons[i][j],
                12, autoSizeMaxTextSize,1, 1);
    }
}
```

Besides everything else, a completion listener is set in the „setUpGame()” method through the „GameEngine” instance, calling four other methods: „updatePointsText()”, „showToast()”, „cleanBoard()”, „botMove()”. This listener will run after a game is completed, to reset the board and update the points.

```
gameEngine.setCompletionListener(new GameCompletionListener() {
    @Override
    public void onCompletion() {
        updatePointsText();
        showToast();
        cleanBoard();
        botMove();
    }
});
```

At the end of the „setUpGame()" method, four methods are called in the following order: „showPlayerSymbol()", „updatePointsText()", „highlightWhoStarts()", „botMove()". These methods are meant to prepare the scoreboard for the game and afterwards, let the bot make a move if it is its turn.

Finally, in the „onCreate()" method, the „loadLocale()" method is called through the „LocaleManager" instance, to set the proper language as chosen by the user. Then, the content view is set, the settings are loaded, the game state is loaded through the „GameEngine" instance, an intent is instantiated using the intent that was used to launch this activity and afterwards, the „extra" is fetched from the intent to later recognize the gameplay mode. In the end, the „setUpGame()" method is called. The reason for loading the game state in the „onCreate()" method is due to the fact that the activity gets destroyed and rebuilt when rotation of the screen occurs. To prevent loosing the game state, the game state is saved when the „onStop()" method is called, and then gets loaded when the „onCreate()" method is called again.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    localeManager.loadLocale();
    setContentView(R.layout.activity_game);
    decorView = getWindow().getDecorView();
    loadSettings();
    gameEngine.loadGameState();
    Intent intent = getIntent();
    mSinglePlayerMode = intent.getBooleanExtra(MenuActivity.EXTRA_SINGLE_PLAYER_MODE,
false);
    setUpGame();
}


@Override
protected void onStop() {
    super.onStop();
    gameEngine.saveGameState();
}
```

For saving and loading the game state, two classes were implemented: „GameState” and „GameStateRepository”.

The „GameState” class plays the role of a container class for the variables that need to be saved in order to preserve the game state. The class only contains the necessesary variables with the corresponding setter and getter methods.

```
public class GameState implements Serializable {

    private boolean player1Turn = true;
    private int roundCount;
    private int player1Points;
    private int player2Points;

    public void setPlayer1Turn(boolean player1Turn) {
        this.player1Turn = player1Turn;
    }

    public void setRoundCount(int roundCount) {
        this.roundCount = roundCount;
    }

    public void setPlayer1Points(int player1Points) {
        this.player1Points = player1Points;
    }

    public void setPlayer2Points(int player2Points) {
        this.player2Points = player2Points;
    }

    public boolean getPlayer1Turn() {
        return player1Turn;
    }

    public int getRoundCount()  {
        return roundCount;
    }

    public int getPlayer1Points() {
        return player1Points;
    }

    public int getPlayer2Points()   {
        return player2Points;
    }
}
```

The „GameStateRepository” class is the class where the methods for saving and loading content are defined. This class instantiates the „GameState” class and also takes its instance as a parameter for the class constructor. The „save()” method is implemented through the use of the „FileOutputStream” class instance, which is initialized and then passed as an argument to the constructor of the „ObjectOutputStream” class, that is instantiated afterwards. Using the instance

of the class „ObjectOutputStream”, the „GameState” class instnance is written and then, the output stream is closed.

```
public void save() {

    try {
        FileOutputStream fos = new FileOutputStream("GameState.ser");
        ObjectOutputStream os = new ObjectOutputStream(fos);
        os.writeObject(gameState);
        os.close();
    } catch(Exception e) {
        e.printStackTrace();
    }
}
```

For the „load()” method the process happens in reverse. Instead of the „FileOutputStream” and „ObjectOutputStream”, the „FileInputStream” and „ObjectInputStream” classes are used. Instead of writting the object, now the object is read and assigned to the „GameState” instance. In the end, the input stream is closed.

```
public void load() {

    try {
        FileInputStream fis = new FileInputStream("GameState.ser");
        ObjectInputStream is = new ObjectInputStream(fis);
        gameState = (GameState) is.readObject();
        is.close();
    } catch(Exception e) {
        e.printStackTrace();
    }
}
```

The „GameEngine” class holds most the methods related to the game logic and functionality of the application.

The method „saveGameState()” sets all the variables in the „GameState” instance to the current values, and then calls the „save()” method through the „GameStateRepository” instance to save the object.

```
public void saveGameState() {

    gameState.setPlayer1Turn(player1Turn);
    gameState.setRoundCount(roundCount);
    gameState.setPlayer1Points(player1Points);
    gameState.setPlayer2Points(player2Points);

    gameStateRepository.save();
}
```

The method „loadGameState()” calls the „load()” method through the „GameStateRepository” instance, which will bring back the „GameState” instance in the state it was before it was destroyed. Afterwards, the values of the variables are extracted and assigned to the attributes in the „GameEngine” class.

```
public void loadGameState() {

    gameStateRepository.load();

    player1Turn = gameState.getPlayer1Turn();
    roundCount = gameState.getRoundCount();
    player1Points = gameState.getPlayer1Points();
    player2Points = gameState.getPlayer2Points();
}
```

The „roundResult(String[][] gameTable)” method has the role of checking for the end of the game, either if it is through a win for player 1, a defeat, or a draw. In each case, an appropriate method will be called: „player1Wins()”, „player2Wins()”, „draw()”. These methods imply setting the boolean variables „player1Wins” and „player2Wins” to true/false depending on the case, incrementing the points if a player won, restarting the game by calling the „restartGame()” method and finally calling the „completion()” method.

```
private void player1Wins() {
    player1Wins = true;
    player2Wins = false;
    player1Points++;
    restartGame();
    completion();
}

private void player2Wins() {
    player1Wins = false;
    player2Wins = true;
    player2Points++;
    restartGame();
    completion();
}

private void draw() {
    player1Wins = false;
    player2Wins = false;
    restartGame();
    completion();
}
```

The „restartGame()” method is used to reset the round count to 0 and detemine who starts the next round by calling the „whoStarts()” method.

```
private void restartGame() {
    roundCount = 0;
    whoStarts();
}
```

The „whoStarts()" method determines who is going to start the next round. It does so, by checking the booelan variables „player1Wins" and „player2Wins" and, depending on their value and on the settings done by the user regarding who starts after a certain result, it will give the turn accordingly.

As in the settings activity there is an option to let a different player start every game, another variable called „player1StartsMatch()" is introduced. In this specific case, the variable will be set to its opposite value, and then it will be passed as an argument to the „setPlayer1Turn()" method, this way letting the other player start.

```
private void whoStarts() {

    if ((!getPlayer1Wins()) && (!getPlayer2Wins())) {

        if (getWhoStartsDraw() == SettingsUtility.DRAW_OTHER_PLAYER_STARTS) {
            setPlayer1Turn((!getPlayer1Turn()));
        } else if (getWhoStartsDraw() == SettingsUtility.DRAW_SAME_PLAYER_STARTS) {
            setPlayer1Turn(getPlayer1Turn());
        }

    } else if (getPlayer1Wins()) {

        if (getWhoStarts() == SettingsUtility.WINNER_STARTS) {
            setPlayer1Turn(true);
        } else if ((getWhoStarts() == SettingsUtility.DIFFERENT_PLAYER_STARTS) ) {
            setPlayer1StartsMatch(!getPlayer1StartsMatch());
            setPlayer1Turn(getPlayer1StartsMatch());
        }

    } else if (getPlayer2Wins()) {
        if (getWhoStarts() == SettingsUtility.WINNER_STARTS) {
            setPlayer1Turn(false);
        } else if (getWhoStarts() == SettingsUtility.DIFFERENT_PLAYER_STARTS) {
            setPlayer1StartsMatch(!getPlayer1StartsMatch());
            setPlayer1Turn(getPlayer1StartsMatch());
        }
    }
}
```

The method „checkForWin(String[][] gameTable)" receives a matrix that corresponds to the state of the game board, and checks for a streak of three same symbols on either a row, column or diagonal. It will return „true" if it finds a streak and false if it doesn't.

```
private boolean checkForWin(String[][] gameTable) {
    // check row streaks
    for (int i = 0; i < 3; i++) {
        if (gameTable[i][0].equals(gameTable[i][1])
                && gameTable[i][0].equals(gameTable[i][2])
                && !gameTable[i][0].equals("")) {
            return true;
        }
    }
    // check column streaks
    for (int i = 0; i < 3; i++) {
        if (gameTable[0][i].equals(gameTable[1][i])
                && gameTable[0][i].equals(gameTable[2][i])
                && !gameTable[0][i].equals("")) {
            return true;
        }
    }
    // check for a streak on principal diagonal
    if (gameTable[0][0].equals(gameTable[1][1])
            && gameTable[0][0].equals(gameTable[2][2])
            && !gameTable[0][0].equals("")) {
        return true;
    }
    // check for a streak on secondary diagonal
    if (gameTable[0][2].equals(gameTable[1][1])
            && gameTable[0][2].equals(gameTable[2][0])
            && !gameTable[0][2].equals("")) {
        return true;
    }

    return false;
}
```

The „resetGame()” is called when the reset button is pressed, and its role is to reset the players points and the round count to 0, and call the „completion()” method in order to update the text views and clean the board. The variables „player1Wins”, „player2Wins” and „player1Turn” are set to true so that the „showToast()” method from the „completion()” method won’t show any toast, as the result of the game cannot be decided, and also the new game will be started by player 1, as it should.

```
public void resetGame() {
    player1Points = 0;
    player2Points = 0;
    roundCount = 0;
    player1Wins = true;
    player2Wins = true;
    player1Turn = true;
    completion();
}
```

The method „check(int row, int column, String[][] gameTable)" is fundamental for the single player gameplay mode, as this is the method used to write on buttons when the bot turn comes. The method receives three arguments that refer to the indexes of the row and column to be checked and also the matrix corresponding to the current state of the game table.
Inside, the method will check if the specified button is already checked, and if it isn't it assigns the chosen symbol to the corresponding matrix element, depending on the settings done by the user.

```
public void check(int row, int column, String[][] gameTable) {
    if (isChecked(row, column, gameTable)) {
        return;
    }

    if (getWhoIsPlayer1() == SettingsUtility.BOT_IS_PLAYER_1) {
        if (mSymbolPlayer1 == SettingsUtility.X) {
            gameTable[row][column] = "X";
        } else if (mSymbolPlayer1 == SettingsUtility.O) {
            gameTable[row][column] = "O";
        }
    } else if (getWhoIsPlayer1() == SettingsUtility.YOU_ARE_PLAYER_1) {
        if (mSymbolPlayer1 == SettingsUtility.X) {
            gameTable[row][column] = "O";
        } else if (mSymbolPlayer1 == SettingsUtility.O) {
            gameTable[row][column] = "X";
        }
    }
}
```

The method „isChecked(int row, int column, String[][] gameTable)" takes three arguments that refer to the indexes of the row and column to be checked and also the matrix corresponding to the current state of the game table. It verifies if a certain element of the matrix is checked or empty and returns true if it is checked and false if it isn't.

```
public boolean isChecked(int row, int column, String[][] gameTable) {
    if (!(gameTable[row][column].equals(""))) {
        return true;
    } else {
        return false;
    }
}
```

The method „getSymbolAt(int row, int column, String[][] gameTable)" works just like a getter method, providing the symbol from a specified location in the matrix corresponding to the game board.

```
private String getSymbolAt(int row, int column, String[][] gameTable) {
    return gameTable[row][column];
}
```

The method „match(int row1, int row2, int column1, int column2, String[][] gameTable)" has the role to verify if two cells of the matrix corresponding to the game board have the same value. I returns true if the values match and false if the values don't match.

```
private boolean match(int row1, int column1, int row2, int column2, String[][]
gameTable) {
    if
(getSymbolAt(row1,column1,gameTable).equals(getSymbolAt(row2,column2,gameTable))) {
        return true;
    } else {
        return false;
    }
}
```

The method „isEnemySymbol(int row, int column, String[][] gameTable)" verifies if the symbol from the specified cell of the matrix corresponding to the game board belongs to the enemy(the user). It returns true if an enemy symbol is found or false if a different symbol is found.

```
public boolean isEnemySymbol(int row, int column, String[][] gameTable) {
    if (getWhoIsPlayer1() == SettingsUtility.BOT_IS_PLAYER_1) {
        if (getSymbolPlayer1() == SettingsUtility.X) {
            return getSymbolAt(row, column, gameTable).equals("O");
        } else if (getSymbolPlayer1() == SettingsUtility.O) {
            return getSymbolAt(row, column, gameTable).equals("X");
        }
    } else if (getWhoIsPlayer1() == SettingsUtility.YOU_ARE_PLAYER_1) {
        if (getSymbolPlayer1() == SettingsUtility.X) {
            return getSymbolAt(row, column, gameTable).equals("X");
        } else if (getSymbolPlayer1() == SettingsUtility.O) {
            return getSymbolAt(row, column, gameTable).equals("O");
        }
    }
    return false;
}
```

The method „isOwnSymbol(int row, int column, String[][] gameTable)" verifies if the symbol from the specified cell of the matrix corresponding to the game board belongs to the bot. It returns true if the symbol used by the bot is found or false if a different symbol is found.

```
public boolean isOwnSymbol(int row, int column, String[][] gameTable) {
    if (getWhoIsPlayer1() == SettingsUtility.BOT_IS_PLAYER_1) {
        if (getSymbolPlayer1() == SettingsUtility.X) {
            return getSymbolAt(row, column, gameTable).equals("X");
        } else if (getSymbolPlayer1() == SettingsUtility.O) {
            return getSymbolAt(row, column, gameTable).equals("O");
        }
    } else if (getWhoIsPlayer1() == SettingsUtility.YOU_ARE_PLAYER_1) {
        if (getSymbolPlayer1() == SettingsUtility.X) {
            return getSymbolAt(row, column, gameTable).equals("O");
        } else if (getSymbolPlayer1() == SettingsUtility.O) {
            return getSymbolAt(row, column, gameTable).equals("X");
        }
    }
    return false;
}
```

The method „checkEdge(String[][] gameTable)" is used to verify that two opposite edges are unchecked and then check one of them.

```
public void checkEdge(String[][] gameTable) {
    if ((!isChecked(0,1, gameTable)) && (!isChecked(2,1, gameTable))) {
        check(0,1, gameTable);
    } else if ((!isChecked(1,0, gameTable)) && (!isChecked(1,2, gameTable))) {
        check(1,0, gameTable);
    } else if ((!isChecked(2,1, gameTable)) && (!isChecked(0,1, gameTable))) {
        check(2,1, gameTable);
    } else if ((!isChecked(1,2, gameTable)) && (!isChecked(1,0, gameTable))) {
        check(1,2, gameTable);
    }
}
```

The method „checkCorner(String[][] gameTable)" verifies each corner, and when it finds an empty cell it checks it.

```
public void checkCorner(String[][] gameTable) {
    if (!isChecked(0,0, gameTable)) {
        check(0,0, gameTable);
    } else if (!isChecked(0,2, gameTable)) {
        check(0,2, gameTable);
    } else if (!isChecked(2,1, gameTable)) {
        check(2,0, gameTable);
    } else if (!isChecked(1,2, gameTable)) {
        check(2,2, gameTable);
    }
}
```

The method „findSymbolsOnOppositeEdges(String[][] gameTable)" verifies if there are two of the same symbol on the opposite edges of the matrix.

```
public boolean findSymbolsOnOppositeEdges(String[][] gameTable) {
    if ((isChecked(0,1, gameTable)) && (isChecked(2,1, gameTable))) {
        return true;
    } else if ((isChecked(1,0, gameTable)) && (isChecked(1,2, gameTable))) {
        return true;
    } else {
        return false;
    }
}
```

The method „checkWhatIsLeft(String[][] gameTable)" iterates through all the elements of the matrix until it finds one that is not checked and then, it will check that element.

```
public void checkWhatIsLeft(String[][] gameTable) {
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (!isChecked(i, j, gameTable)) {
                check(i, j, gameTable);
                return;
            }
        }
    }
}
```

The method „checkCornerCloseToEnemyEdge(String[][] gameTable)” searches for the first enemy symbol on the edge that has an empty corner besides it and then, it checks that corner.

```
public void checkCornerCloseToEnemyEdge(String[][] gameTable) {

    if (isEnemySymbol(0, 1, gameTable)) {

        if ((isEnemySymbol(2, 0, gameTable)) &&
                (!isChecked(0, 0, gameTable))) {
            check(0, 0, gameTable);

        } else if ((isEnemySymbol(2, 2, gameTable)) &&
                (!isChecked(0, 2, gameTable))) {
            check(0, 2, gameTable);
        }


    } else if (isEnemySymbol(1, 0, gameTable)) {

        if ((isEnemySymbol(0, 2, gameTable)) &&
                (!isChecked(0, 0, gameTable))) {
            check(0, 0, gameTable);

        } else if ((isEnemySymbol(2, 2, gameTable)) &&
                (!isChecked(2, 2, gameTable))) {
            check(2, 2, gameTable);
        }

    } else if (isEnemySymbol(1, 2, gameTable)) {

        if ((isEnemySymbol(0, 0, gameTable)) &&
                (!isChecked(0, 2, gameTable))) {
            check(0, 2, gameTable);

        } else if ((isEnemySymbol(2, 0, gameTable)) &&
                (!isChecked(2, 2, gameTable))) {
            check(2, 2, gameTable);
        }

    } else if (isEnemySymbol(2, 1, gameTable)) {

        if ((isEnemySymbol(0, 0, gameTable)) &&
                (!isChecked(2, 0, gameTable))) {
            check(2, 0, gameTable);

        } else if ((isEnemySymbol(0, 2, gameTable)) &&
                (!isChecked(2, 2, gameTable))) {
            check(2, 2, gameTable);
        }

    }
}
```

The method „goForWin(String[][] gameTable)” searches for a streak of two bot symbols and then completes the streak by checking the third cell in a row.

```
public void goForWin(String[][] gameTable) {
    boolean stop;

    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {

            if (isOwnSymbol(i, j, gameTable)) {
                stop = checkVertically(gameTable, i, j);
                if (stop) return;
                stop = checkHorizontally(gameTable, i, j);
                if (stop) return;
                stop = checkPrincipalDiagonal(gameTable, i, j);
                if (stop) return;
                stop = checkSecondaryDiagonal(gameTable, i, j);
                if (stop) return;
            }
        }
    }
}
```

The method „canWin(String[][] gameTable)” verifies if there is a posibility to win, by scanning all the possible lines for a streak of two bot symbols and returns true if a win posibility exists or false if it doesn’t.

```
public boolean canWin(String[][] gameTable) {
    boolean stop;

    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {

            if (isOwnSymbol(i, j, gameTable)) {
                stop = scanVertically(gameTable, i, j);
                if (stop) return true;
                stop = scanHorizontally(gameTable, i, j);
                if (stop) return true;
                stop = scanPrincipalDiagonal(gameTable, i, j);
                if (stop) return true;
                stop = scanSecondaryDiagonal(gameTable, i, j);
                if (stop) return true;
            }
        }
    }
    return false;
}
```

The method „block(String[][] gameTable)” searches for a strak of two enemy symbols and completes the line with a bot symbol to block the streak.

```
public void block(String[][] gameTable) {
    boolean stop;

    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {

            if (isEnemySymbol(i, j, gameTable)) {
                stop = checkVertically(gameTable, i, j);
                if (stop) return;
                stop = checkHorizontally(gameTable, i, j);
                if (stop) return;
                stop = checkPrincipalDiagonal(gameTable, i, j);
                if (stop) return;
                stop = checkSecondaryDiagonal(gameTable, i, j);
                if (stop) return;
            }
        }
    }
}
```

The method „canBlock(String[][] gameTable)” searches for a streak of two enemy symbols and returns true if it finds it or false if it doesn’t.

```
public boolean canBlock(String[][] gameTable) {
    boolean stop;

    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {

            if (isEnemySymbol(i, j, gameTable)) {
                stop = scanVertically(gameTable, i, j);
                if (stop) return true;
                stop = scanHorizontally(gameTable, i, j);
                if (stop) return true;
                stop = scanPrincipalDiagonal(gameTable, i, j);
                if (stop) return true;
                stop = scanSecondaryDiagonal(gameTable, i, j);
                if (stop) return true;
            }
        }
    }
    return false;
}
```

The method „checkVertically(String[][] gameTable, int i, int j)” verifies if two bot symbols are on the same vertical line and if the third cell is empty, case in which it checks that third cell to complete the streak and returns true. If it doesn’t find any posibility to complete a vertical streak, it returns false.

```
private boolean checkVertically(String[][] gameTable, int i, int j) {

        if ((i == 0) || (i == 1)) {
            if (match(i, j, i + 1, j, gameTable)) {
                if ((i == 0) && (!isChecked(i + 2, j, gameTable))) {
                    check(i + 2, j, gameTable);
                    return true;
                } else if ((i == 1) && (!isChecked(i - 1, j, gameTable))) {
                    check(i - 1, j, gameTable);
                    return true;
                }
            } else if (i == 0) {
                if (match(i, j, i + 2, j, gameTable) &&
                        (!isChecked(i + 1, j, gameTable))) {
                    check(i + 1, j, gameTable);
                    return true;
                }
            }
        }
    return false;
}
```

The method „checkHorizontally(String[][] gameTable, int i, int j)" verifies if two bot symbols are on the same horizontal line and if the third cell is empty, case in which it checks that third cell to complete the streak and returns true. If it doesn't find any posibility to complete a horizontal streak, it returns false.

```
private boolean checkHorizontally(String[][] gameTable, int i, int j) {

        if ((j == 0) || (j == 1)) {
            if (match(i, j, i, j + 1, gameTable)) {
                if ((j == 0) && (!isChecked(i, j + 2, gameTable))) {
                    check(i, j + 2, gameTable);
                    return true;
                } else if ((j == 1) && (!isChecked(i, j - 1, gameTable))) {
                    check(i, j - 1, gameTable);
                    return true;
                }
            } else if (j == 0) {
                if (match(i, j, i, j + 2, gameTable) &&
                        (!isChecked(i, j + 1, gameTable))) {

                    check(i, j + 1, gameTable);
                    return true;
                }
            }
        }
    return false;
}
```

The method „checkPrincipalDiagonal(String[][] gameTable, int i, int j)" verifies if two bot symbols are on the principal diagonal and if the third cell is empty, case in which it checks that third cell to complete the streak and returns true. If it doesn't find any posibility to complete the diagonal streak, it returns false.

```
private boolean checkPrincipalDiagonal(String[][] gameTable, int i, int j) {

        if (((i == 0) && (j == 0)) || ((i == 1) && (j == 1))) {
            if (match(i, j, i + 1, j + 1, gameTable)) {
                if ((i == 0) && (!isChecked(i + 2, j + 2, gameTable))) {
                    check(i + 2, j + 2, gameTable);
                    return true;
                } else if ((i == 1) && (!isChecked(i - 1, j - 1, gameTable))) {
                    check(i - 1, j - 1, gameTable);
                    return true;
                }
            } else if (i == 0) {
                if (match(i, j, i + 2, j + 2, gameTable) &&
                        (!isChecked(i + 1, j + 1, gameTable))) {

                    check(i + 1, j + 1, gameTable);
                    return true;
                }
            }
        }
    return false;
}
```

The method „checkSecondaryDiagonal(String[][] gameTable, int i, int j)" verifies if two bot symbols are on the secondary diagonal and if the third cell is empty, case in which it checks that third cell to complete the streak and returns true. If it doesn't find any posibility to complete the diagonal streak, it returns false.

```
private boolean checkSecondaryDiagonal(String[][] gameTable, int i, int j) {

        if (((i == 0) && (j == 2)) || ((i == 1) && (j == 1))) {
            if (match(i, j, i + 1, j - 1, gameTable)) {
                if ((i == 0) && (!isChecked(i + 2, j - 2, gameTable))) {
                    check(i + 2, j - 2, gameTable);
                    return true;
                } else if ((i == 1) && (!isChecked(i - 1, j + 1, gameTable))) {
                    check(i - 1, j + 1, gameTable);
                    return true;
                }
            } else if (i == 0) {
                if (match(i, j, i + 2, j - 2, gameTable) &&
                        (!isChecked(i + 1, j - 1, gameTable))) {

                    check(i + 1, j - 1, gameTable);
                    return true;
                }
            }
        }
    return false;
}
```

The method „scanVertically(String[][] gameTable, int i, int j)” verifies if two bot symbols are on the same vertical line and if the third cell is empty, case in which it returns true. If it doesn’t find any posibility to complete a vertical streak, it returns false.

```
private boolean scanVertically(String[][] gameTable, int i, int j) {

    if ((i == 0) || (i == 1)) {
        if (match(i, j, i + 1, j, gameTable)) {
            if ((i == 0) && (!isChecked(i + 2, j, gameTable))) {
                return true;
            } else if ((i == 1) && (!isChecked(i - 1, j, gameTable))) {
                return true;
            }
        } else if (i == 0) {
            if (match(i, j, i + 2, j, gameTable) &&
                    (!isChecked(i + 1, j, gameTable))) {
                return true;
            }
        }
    }
    return false;
}
```

The method „scanHorizontally(String[][] gameTable, int i, int j)” verifies if two bot symbols are on the same horizontal line and if the third cell is empty, case in which it returns true. If it doesn’t find any posibility to complete a horizontal streak, it returns false.

```
private boolean scanHorizontally(String[][] gameTable, int i, int j) {

    if ((j == 0) || (j == 1)) {
        if (match(i, j, i, j + 1, gameTable)) {
            if ((j == 0) && (!isChecked(i, j + 2, gameTable))) {
                return true;
            } else if ((j == 1) && (!isChecked(i, j - 1, gameTable))) {
                return true;
            }
        } else if (j == 0) {
            if (match(i, j, i, j + 2, gameTable) &&
                    (!isChecked(i, j + 1, gameTable))) {

                return true;
            }
        }
    }
    return false;
}
```

The method „scanPrincipalDiagonal(String[][] gameTable, int i, int j)” verifies if two bot symbols are on the principal diagonal and if the third cell is empty, case in which it returns true. If it doesn’t find any posibility to complete the diagonal streak, it returns false.

```
private boolean scanPrincipalDiagonal(String[][] gameTable, int i, int j) {

    if (((i == 0) && (j == 0)) || ((i == 1) && (j == 1))) {
        if (match(i, j, i + 1, j + 1, gameTable)) {
            if ((i == 0) && (!isChecked(i + 2, j + 2, gameTable))) {
                return true;
            } else if ((i == 1) && (!isChecked(i - 1, j - 1, gameTable))) {
                return true;
            }
        } else if (i == 0) {
            if (match(i, j, i + 2, j + 2, gameTable) &&
                    (!isChecked(i + 1, j + 1, gameTable))) {

                return true;
            }
        }
    }
    return false;
}
```

The method „scanSecondaryDiagonal(String[][] gameTable, int i, int j)" verifies if two bot symbols are on the secondary diagonal and if the third cell is empty, case in which it returns true. If it doesn't find any posibility to complete the diagonal streak, it returns false.

```
private boolean scanSecondaryDiagonal(String[][] gameTable, int i, int j) {

    if (((i == 0) && (j == 2)) || ((i == 1) && (j == 1))) {
        if (match(i, j, i + 1, j - 1, gameTable)) {
            if ((i == 0) && (!isChecked(i + 2, j - 2, gameTable))) {
                return true;
            } else if ((i == 1) && (!isChecked(i - 1, j + 1, gameTable))) {
                return true;
            }
        } else if (i == 0) {
            if (match(i, j, i + 2, j - 2, gameTable) &&
                    (!isChecked(i + 1, j - 1, gameTable))) {

                return true;
            }
        }
    }
    return false;
}
```

The method „scanEdgeOppositeToCorner(String[][] gameTable)" searches for two enemy symbols: one on an edge and the other on an opposite corner. If it finds a pair like that, it returns true and if it doesn't, it returns false.

```
public boolean scanEdgeOppositeToCorner(String[][] gameTable) {

    if (isEnemySymbol(0, 1, gameTable)) {

        if (isEnemySymbol(2, 0, gameTable)) {
            return true;
        } else if (isEnemySymbol(2, 2, gameTable)) {
            return true;
        }
    } else if (isEnemySymbol(1, 0, gameTable)) {

        if (isEnemySymbol(0, 2, gameTable)) {
            return true;
        } else if (isEnemySymbol(2, 2, gameTable)) {
            return true;
        }
    } else if (isEnemySymbol(1, 2, gameTable)) {

        if (isEnemySymbol(0, 0, gameTable)) {
            return true;
        } else if (isEnemySymbol(2, 0, gameTable)) {
            return true;
        }
    } else if (isEnemySymbol(2, 1, gameTable)) {

        if (isEnemySymbol(0, 0, gameTable)) {
            return true;
        } else if (isEnemySymbol(0, 2, gameTable)) {
            return true;
        }
    }

    return false;
}
```

The method „scanEnemyCorners(String[][] gameTable)" searches for the first checked enemy corner and returns true if it finds it, or false if it doesn't find it.

```
public boolean scanEnemyCorners(String[][] gameTable) {

    if (isEnemySymbol(0, 0, gameTable))
        return true;

    if (isEnemySymbol(0, 2, gameTable))
        return true;

    if (isEnemySymbol(2, 0, gameTable))
        return true;

    if (isEnemySymbol(2, 2, gameTable))
        return true;

    return false;
}
```

The method „randomNumber()” generates and returns a random number between 0 and 2.

```
public int randomNumber() {
    Random random = new Random();
    return random.nextInt(3);
}
```

The method „randomCheck(String[][] gameTable)” randomly checks a cell in the matrix corresponding to the game board with the symbol chosen for the bot.

```
public void randomCheck(String[][] gameTable) {
    int randomRow = randomNumber();
    int randomColumn = randomNumber();

    boolean stop = false;

    while (!stop) {

        if (!isChecked(randomRow, randomColumn, gameTable)) {
            check(randomRow, randomColumn, gameTable);
            stop = true;
        } else {
            randomRow = randomNumber();
            randomColumn = randomNumber();
        }
    }
}
```

The method „updateRoundCount()” increments the variable that keeps the count of the rounds played.

```
public void updateRoundCount() {
    roundCount++;
}
```

The method „completion()” acts as a „trigger” for the „onCompletion()” method from the „GameCompletionListener” interface. Whenever the „completion()” method is called, the „onCompletion()” method, which is overriden in the „GameActivity” class, will be called.

```
private void completion() {
    if (gameCompletionListener != null) {
        gameCompletionListener.onCompletion();
    }
}
```

The class **„BotEngine”** is the class that determines the behaviour of the bot depending on the difficulty level. This class takes an instance of the „GameEngine” class as parameter and it contains sets of methods that dictate the moves of the bot as player 1 or player 2, in the different difficulty levels.
When the difficulty level is set on „hard”, the best result a client can get is a draw, if he makes the proper moves. Otherwise, the bot will always win.
The bot, as player 1, will always check two corners for its first two moves.

```
private void botP1HardFirstMove(String[][] gameTable) {
    if (gameEngine.getRoundCount() == 0)
        gameEngine.check(0, 0, gameTable);
}

private void botP1HardSecondMove(String[][] gameTable) {
    if (gameEngine.getRoundCount() == 2) {
        if (!(gameEngine.isChecked(2, 2, gameTable))) {
            gameEngine.check(2, 2, gameTable);
        } else {
            gameEngine.check(2, 0, gameTable);
        }
    }
}
```

For his third move, it will complete the line if it can, otherwise will check another corner. In case a block move can be done, it will.

```
private void botP1HardThirdMove(String[][] gameTable) {
    if (gameEngine.getRoundCount() == 4) {

        if (gameEngine.canWin(gameTable)) {
            gameEngine.goForWin(gameTable);
        } else if (gameEngine.canBlock(gameTable)) {
            gameEngine.block(gameTable);
        } else {
            gameEngine.checkCorner(gameTable);
        }
    }
}
```

For its fourth move it will, once again, check if a line can be completed, or if it can block the enemy from completing a line. Otherwise, it will check what is left.

```
private void botP1HardFourthMove(String[][] gameTable) {
    if (gameEngine.getRoundCount() == 6) {
        if (gameEngine.canWin(gameTable)) {
            gameEngine.goForWin(gameTable);
        } else if (gameEngine.canBlock(gameTable)) {
            gameEngine.block(gameTable);
        } else {
            gameEngine.checkWhatIsLeft(gameTable);
        }
    }
}
```

For its fifth and final move, the bot will check whatever cell is left.

```
private void botP1HardFifthMove(String[][] gameTable) {
    if (gameEngine.getRoundCount() == 8)
        gameEngine.checkWhatIsLeft(gameTable);
}
```

All the methods are called sequentially in the „botP1HardMoves(String[][] gameTable)" method.

```
public String[][] botP1HardMoves(String[][] gameTable) {
    botP1HardFirstMove(gameTable);
    botP1HardSecondMove(gameTable);
    botP1HardThirdMove(gameTable);
    botP1HardFourthMove(gameTable);
    botP1HardFifthMove(gameTable);

    return gameTable;
}
```

The bot, as player 2, will always check the cell in the center if it is not checked, otherwise it will check a corner.

```
private void botP2HardFirstMove(String[][] gameTable) {
    if (gameEngine.getRoundCount() == 1) {
        if (!gameEngine.isEnemySymbol(1, 1, gameTable)) {
            gameEngine.check(1, 1, gameTable);
        } else {
            gameEngine.checkCorner(gameTable);
        }
    }
}
```

For its second move, it will block a streak if it is needed. If it's not needed and its first symbol was placed in the center, it will verify if the enemy checked an edge and its opposite corner, case in which it will check the corner close to the checked enemy edge, or if that was not the case, it will check an empty edge. If no corners were checked by the enemy, the bot will just iterate through the matrix corresponding to the game board and check the first available spot.
In the case where the center spot was checked by the enemy, if there is nothing to block, the bot will check the first available corner.

```
private void botP2HardSecondMove(String[][] gameTable) {
    if (gameEngine.getRoundCount() == 3) {
        if (gameEngine.isOwnSymbol(1,1, gameTable)) {
            if (gameEngine.scanEnemyCorners(gameTable)) {
                if (gameEngine.canBlock(gameTable)) {
                    gameEngine.block(gameTable);
                } else if (gameEngine.scanEdgeOppositeToCorner(gameTable)) {
                    gameEngine.checkCornerCloseToEnemyEdge(gameTable);
                } else {
                    gameEngine.checkEdge(gameTable);
                }
            } else {
                gameEngine.checkWhatIsLeft(gameTable);
            }
        } else if (gameEngine.isEnemySymbol(1,1, gameTable)) {
            if (gameEngine.canBlock(gameTable)) {
                gameEngine.block(gameTable);
            } else {
                gameEngine.checkCorner(gameTable);
            }
```

```
        }
    }
}
```

For the third and fourth move, the bot will complete a line if it is possible, it will block an enemy streak if it is needed, or will check whatever spot is empty.

```
private void botP2HardThirdMove(String[][] gameTable) {
    if (gameEngine.getRoundCount() == 5) {
        if (gameEngine.canWin(gameTable)) {
            gameEngine.goForWin(gameTable);
        } else if (gameEngine.canBlock(gameTable)) {
            gameEngine.block(gameTable);
        } else {
            gameEngine.checkWhatIsLeft(gameTable);
        }
    }
}

private void botP2HardFourthMove(String[][] gameTable) {
    if (gameEngine.getRoundCount() == 7) {
        if (gameEngine.canWin(gameTable)) {
            gameEngine.goForWin(gameTable);
        } else if (gameEngine.canBlock(gameTable)) {
            gameEngine.block(gameTable);
        } else {
            gameEngine.checkWhatIsLeft(gameTable);
        }
    }
}
```

All the methods are called sequentially in the „botP2HardMoves(String[][] gameTable)” method.

```
public String[][] botP2HardMoves(String[][] gameTable) {
    botP2HardFirstMove(gameTable);
    botP2HardSecondMove(gameTable);
    botP2HardThirdMove(gameTable);
    botP2HardFourthMove(gameTable);

    return gameTable;
}
```

When the difficulty level is set on „medium”, the bot, as player 1, will make two random moves at first, and then it will complete a line if it is the case, it will block a line if it is the case, or it will just check a random spot if there is no possiblity of winning or blocking.

```
private  void botP1MediumFirstMove(String[][] gameTable) {
    if (gameEngine.getRoundCount() == 0) {
        gameEngine.randomCheck(gameTable);
    }
}

private void botP1MediumSecondMove(String[][] gameTable) {
    if (gameEngine.getRoundCount() == 2) {
```

```
            gameEngine.randomCheck(gameTable);
        }
    }

    private void botP1MediumThirdMove(String[][] gameTable) {
        if (gameEngine.getRoundCount() == 4) {
            if (gameEngine.canWin(gameTable)) {
                gameEngine.goForWin(gameTable);
            } else if (gameEngine.canBlock(gameTable)) {
                gameEngine.block(gameTable);
            } else {
                gameEngine.randomCheck(gameTable);
            }
        }
    }

    private void botP1MediumFourthMove(String[][] gameTable) {
        if (gameEngine.getRoundCount() == 6) {
            if (gameEngine.canWin(gameTable)) {
                gameEngine.goForWin(gameTable);
            } else if (gameEngine.canBlock(gameTable)) {
                gameEngine.block(gameTable);
            } else {
                gameEngine.checkWhatIsLeft(gameTable);
            }
        }
    }

    private void botP1MediumFifthMove(String[][] gameTable)  {
        if (gameEngine.getRoundCount() == 8) {
            gameEngine.checkWhatIsLeft(gameTable);
        }
    }
```

All the methods are called sequentially in the „botP1MediumMoves(String[][] gameTable)” method.

```
public String[][] botP1MediumMoves(String[][] gameTable) {
    botP1MediumFirstMove(gameTable);
    botP1MediumSecondMove(gameTable);
    botP1MediumThirdMove(gameTable);
    botP1MediumFourthMove(gameTable);
    botP1MediumFifthMove(gameTable);

    return gameTable;
}
```

When the difficulty level is set on „medium”, the bot, as player 2, will make a random move at first, and then it will block a line if it is the case, or it will just check a random spot if there is no possiblity of winning or blocking. For the last two moves it will complete a line if it is possible, or it will block a streak if it is the case. If there is no possibility of winning or blocking, it will check whatever spot is left.

```
private void botP2MediumFirstMove(String[][] gameTable) {
    if (gameEngine.getRoundCount() == 1) {
        gameEngine.randomCheck(gameTable);
    }
}

private void botP2MediumSecondMove(String[][] gameTable) {
    if (gameEngine.getRoundCount() == 3) {
        if (gameEngine.canBlock(gameTable)) {
            gameEngine.block(gameTable);
        } else {
            gameEngine.randomCheck(gameTable);
        }
    }
}

private void botP2MediumThirdMove(String[][] gameTable) {
    if (gameEngine.getRoundCount() == 5) {
        if (gameEngine.canWin(gameTable)) {
            gameEngine.goForWin(gameTable);
        } else if (gameEngine.canBlock(gameTable)) {
            gameEngine.block(gameTable);
        } else {
            gameEngine.checkWhatIsLeft(gameTable);
        }
    }
}

private void botP2MediumFourthMove(String[][] gameTable) {
    if (gameEngine.getRoundCount() == 7) {
        if (gameEngine.canBlock(gameTable)) {
            gameEngine.block(gameTable);
        } else if (gameEngine.canWin(gameTable)) {
            gameEngine.goForWin(gameTable);
        } else {
            gameEngine.checkWhatIsLeft(gameTable);
        }
    }
}
```

All the methods are called sequentially in the „botP2MediumMoves(String[][] gameTable)" method.

```
public String[][] botP2MediumMoves(String[][] gameTable) {
    botP2MediumFirstMove(gameTable);
    botP2MediumSecondMove(gameTable);
    botP2MediumThirdMove(gameTable);
    botP2MediumFourthMove(gameTable);

    return gameTable;
}
```

When the difficulty level is set on „easy", all the moves are random, no matter if the bot is player 1, or player 2.

```
private void botP1EasyFirstMove(String[][] gameTable) {
    if (gameEngine.getRoundCount() == 0) {
        gameEngine.randomCheck(gameTable);
    }
}

private void botP1EasySecondMove(String[][] gameTable) {
    if (gameEngine.getRoundCount() == 2) {
        gameEngine.randomCheck(gameTable);
    }
}

private void botP1EasyThirdMove(String[][] gameTable) {
    if (gameEngine.getRoundCount() == 4) {
        gameEngine.randomCheck(gameTable);
    }
}

private void botP1EasyFourthMove(String[][] gameTable) {
    if (gameEngine.getRoundCount() == 6) {
        gameEngine.randomCheck(gameTable);
    }
}

private void botP1EasyFifthMove(String[][] gameTable) {
    if (gameEngine.getRoundCount() == 8) {
        gameEngine.randomCheck(gameTable);
    }
}

private void botP2EasyFirstMove(String[][] gameTable) {
    if (gameEngine.getRoundCount() == 1) {
        gameEngine.randomCheck(gameTable);
    }
}

private void botP2EasySecondMove(String[][] gameTable) {
    if (gameEngine.getRoundCount() == 3) {
        gameEngine.randomCheck(gameTable);
    }
}

private void botP2EasyThirdMove(String[][] gameTable) {
    if (gameEngine.getRoundCount() == 5) {
        gameEngine.randomCheck(gameTable);
    }
}

private void botP2EasyFourthMove(String[][] gameTable) {
    if (gameEngine.getRoundCount() == 7) {
        gameEngine.randomCheck(gameTable);
    }
}
```

All the methods are called sequentially in the „botP1EasyMoves(String[][] gameTable)” method and respectively, in the botP2EasyMoves(String[][] gameTable)” method.

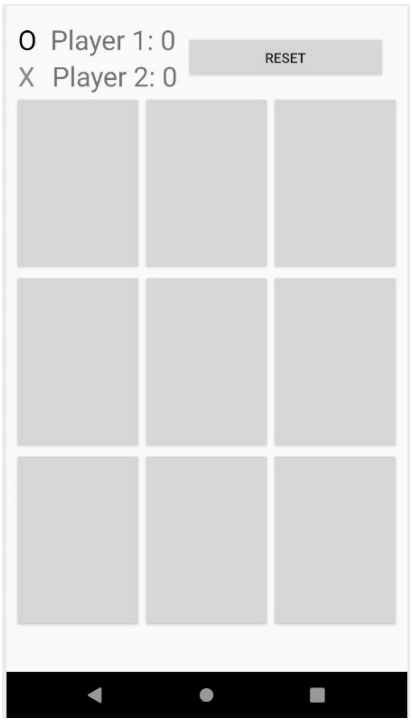
```
public String[][] botP1EasyMoves(String[][] gameTable) {
    botP1EasyFirstMove(gameTable);
    botP1EasySecondMove(gameTable);
    botP1EasyThirdMove(gameTable);
    botP1EasyFourthMove(gameTable);
    botP1EasyFifthMove(gameTable);

    return gameTable;
}

public String[][] botP2EasyMoves(String[][] gameTable) {
    botP2EasyFirstMove(gameTable);
    botP2EasySecondMove(gameTable);
    botP2EasyThirdMove(gameTable);
    botP2EasyFourthMove(gameTable);

    return gameTable;
}
```

4. A tutorial for using the application

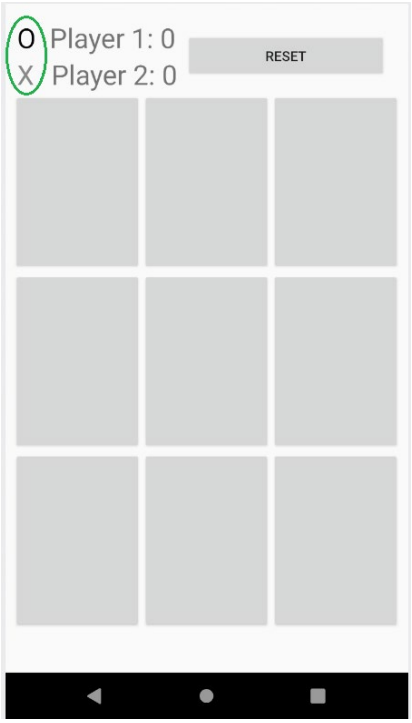
After opening the app you will arrive in a menu with four buttons.

The first button is labeled „TWO PLAYERS” and it will take you to a game that you can play with a friend.

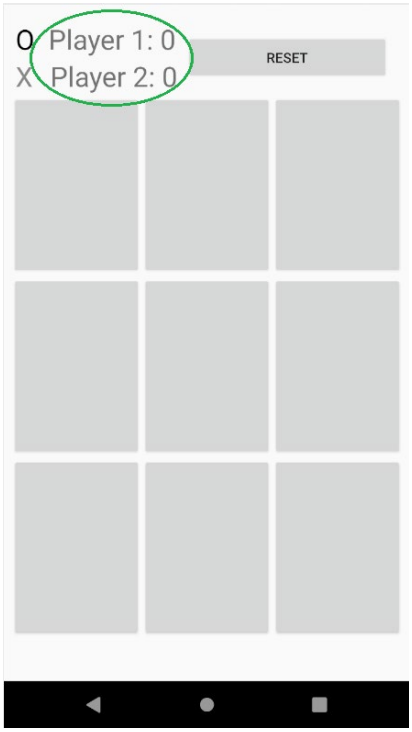
After pressing the button, you will be taken here:



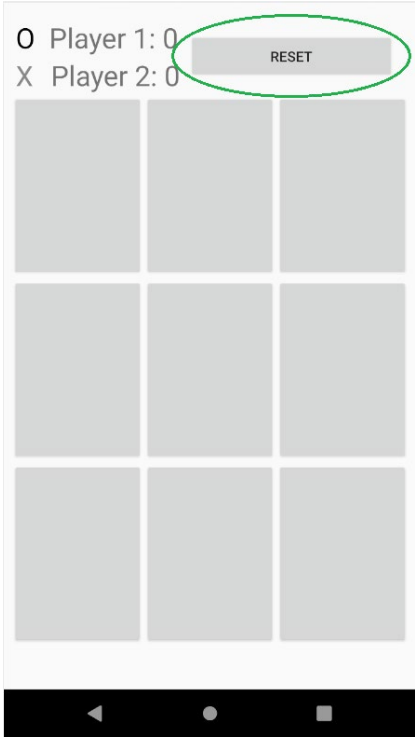
In the top left corner, two symbols can be observed, each corresponding to one of the players. They represent the symbol that the player has chosen to play with. Another thing that can be observed is that one of the symbols is highlighted (colored in black), showing which player's turn to move it is.



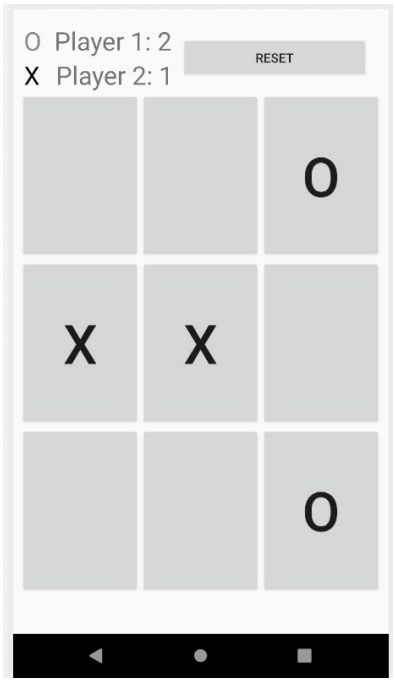
Next to the symbols, the scoreboard can be observed. It will change accordingly to the result of each game.



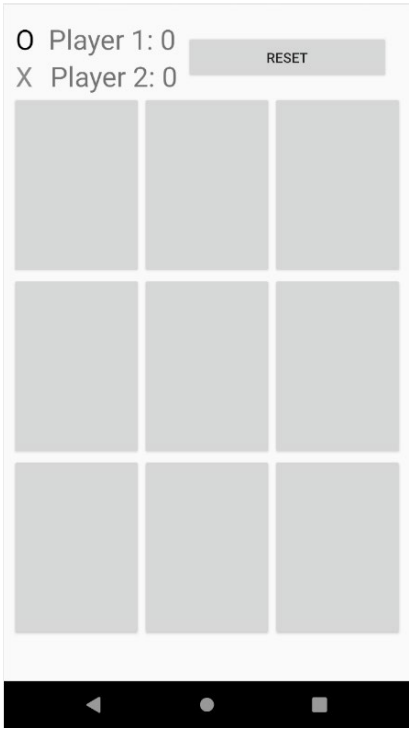
To the right of the scoreboard, the reset button can be observed. Its role is to reset the game and the score.



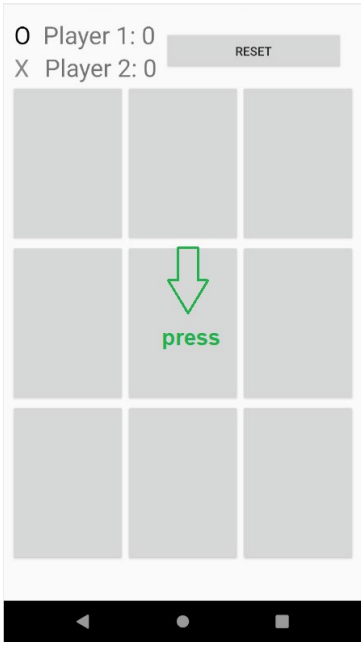
Supposing the game was in the following state:



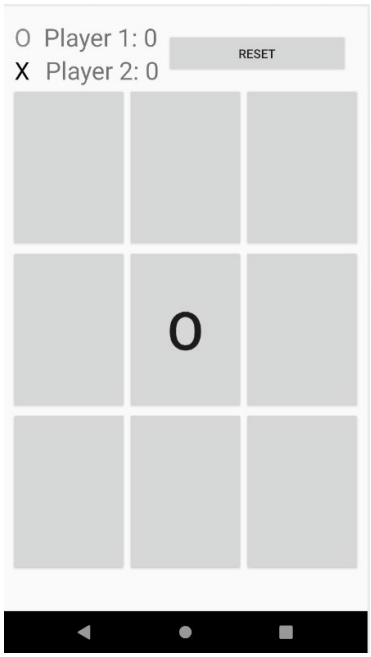
By pressing the reset button, the state of the game will now be:



To make a move, just press on one of the 9 buttons from the goame board and the symbol you have chosen to paly with will automatically be drawn:

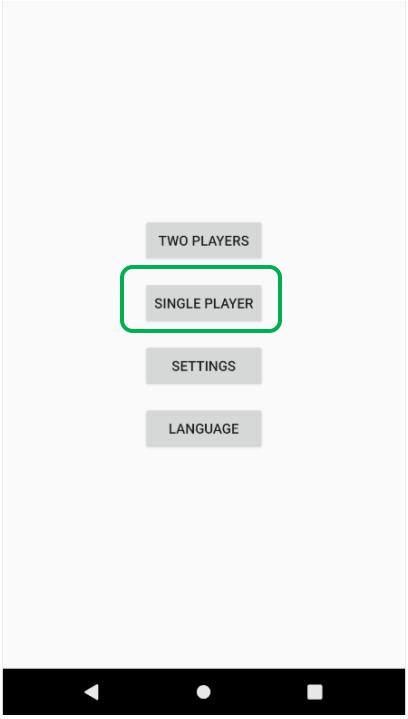


The specified spot is now marked with an „O”, and the „X” symbol is now highlighted in the top left corner, showing that it is player’s 2 turn and he has to mark a spot with the „X” symbol.

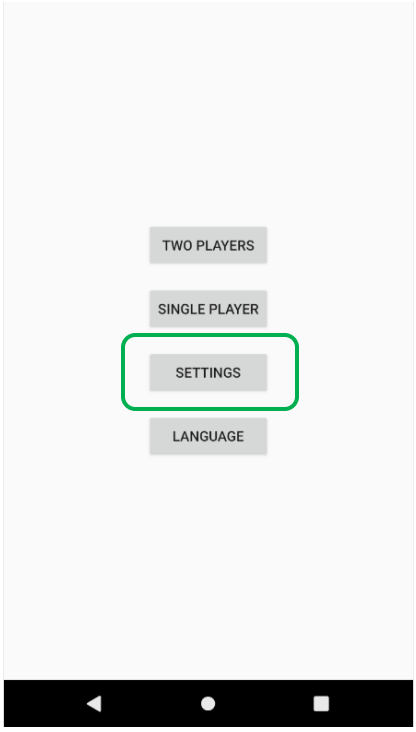


The game goes on until one of the players completes a line(vertical, horizontal, diagonal) with three same symbols, or until all the cells are marked. When a player completes a line, he wins a game and a point. If no line is completed by the end of the game, the game is a draw and nobody receives any points. After the game is completed, the game board is cleaned up, and the game starts again.

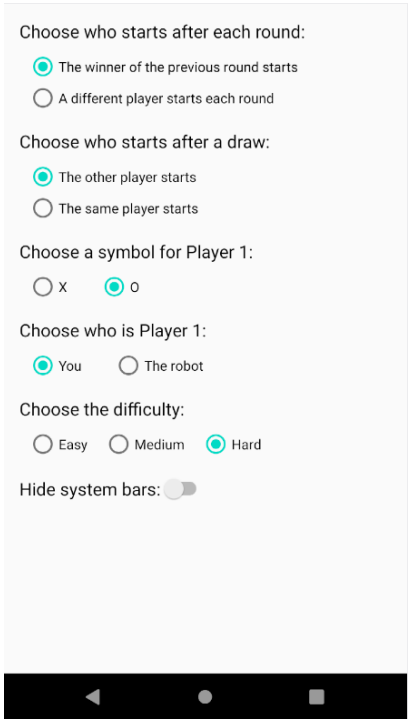
Going back to the menu, the second button, called „SINGLE PLAYER", will take you to the same game board, but this time you are competing against a bot. The game is played in the same way as before, but now you don't have to wait for your opponent to move, as the bot makes its moves instantly.



The bot will play according to a difficulty level that can be selected in the settings menu. To reach the settings menu, the „SETTINGS” button must be pressed in the main menu. The „SETTINGS” button is just below the „SINGLE PLAYER” button.

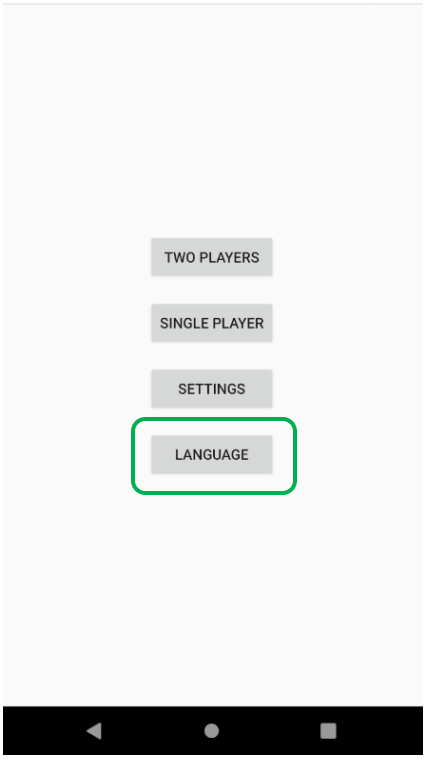


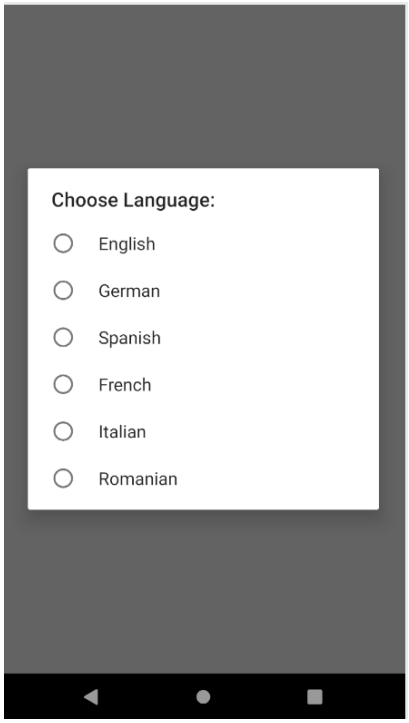
By pressing that button, the following menu will be launched:



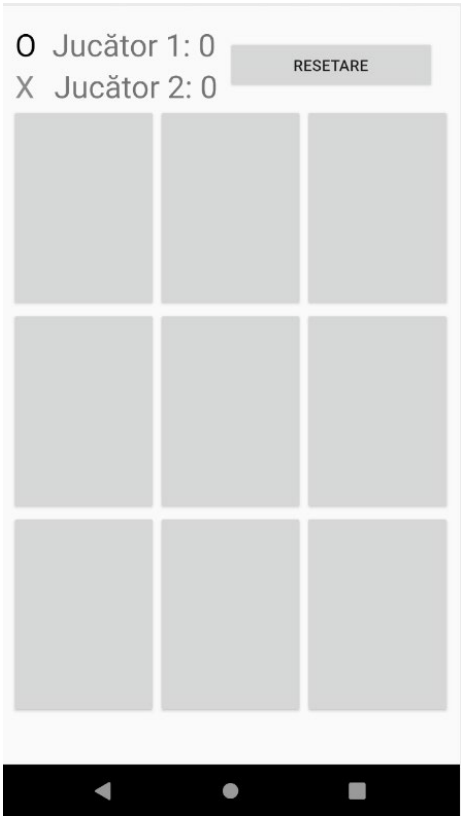
All the game settings can be adjusted to the user's likings here, including the difficulty level for the single player mode.

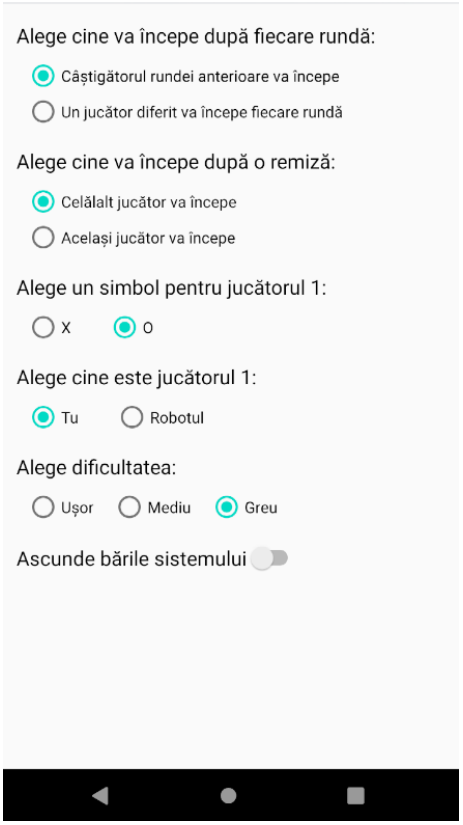
Finally, the last button in the main menu is the „LANGUAGE" button. After pressing this button, a small window will pop up, asking you to choose the language you preffer for this application.

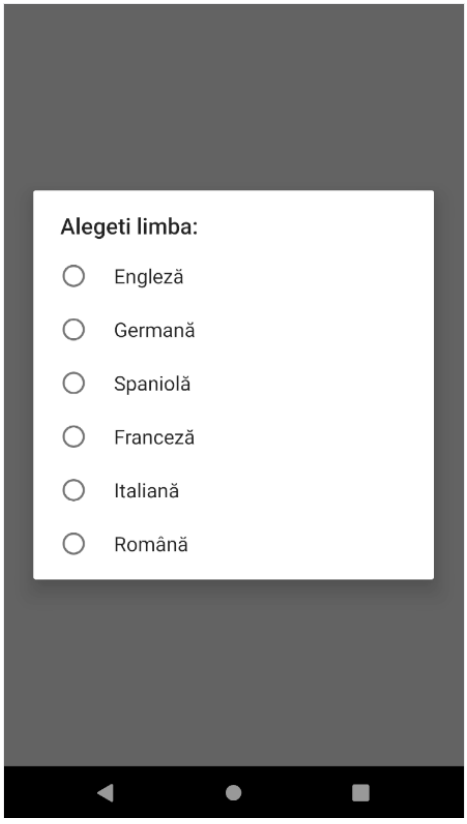




For example. By pressing on „Romanian”, this is how the application will look like:




Alege cine va începe după fiecare rundă:
Câștigătorul rundei anterioare va începe
Un jucător diferit va începe fiecare rundă
Alege cine va începe după o remiză:
Celălalt jucător va începe
Același jucător va începe
Alege un simbol pentru jucătorul 1:
X
O
Alege cine este jucătorul 1:
Tu
Robotul
Alege dificultatea:
Ușor
Mediu
Greu
Ascunde bările sistemului


Alegeti limba:
Engleză
Germană
Spaniolă
Franceză
Italiană
Română

Bibliography:

https://en.wikipedia.org/wiki/Android_Studio

https://developer.android.com/courses/fundamentals-training/toc-v2

Starting point:

https://www.youtube.com/watch?v=apDL78MFR3o

https://www.youtube.com/watch?v=9nVSYkQoV5I

https://www.youtube.com/watch?v=HTO0QmZAbTg