UNIT - III

CHAPTER 6

DESIGNING USER INTERFACE (SUDOKU GAME APP)

CONTENTS

6.1 Introduction

Chapter 1 introduced us to Android development with a simple "Hello, Android" program. This set the stage for our journey. In Chapter 2, we explored the core structure and layout of Android apps. This helped build a solid foundation. Chapter 3 covered essential components like activities, intents, and fragments. These concepts gave us a deeper understanding of Android's building blocks. Chapters 4 and 5 expanded our knowledge by exploring user interfaces and various views in the Android ecosystem.

Now, in Chapter 6, we dive into a practical project by designing a user interface for a Sudoku game. This chapter bridges theory and application. It allows us to use what we've learned to create a visually appealing and user-friendly interface for the game. Through this hands-on project, we gain practical experience. This helps deepen our understanding of key elements in Android development.

The Sudoku game serves as a practical example. It helps us learn how to design interfaces that are intuitive, interactive, and visually pleasing. This exercise lets us grasp the intricacies of UI design within the context of a real-world application. It makes the learning process both engaging and effective. Let's design the UI for the Sudoku game.

6.2 Understanding Sudoku Game

The Sudoku game is a popular logic-based puzzle that involves filling a 9x9 grid with numbers so that each row, each column, and each of the nine 3x3 subgrids contain all the numbers from 1 to 9 without repetition. Let us understand the rules and gameplay of Sudoku:

	7		5	8	3		2	
	5	9	2			3		
3	4				6	5		7
7	9	5				6	3	2
		3	6	9	7	1		
6	8				2	7		
9	1	4	8	3	5		7	6
	3		7		1	4	9	5
5	6	7	4	2	9		1	3

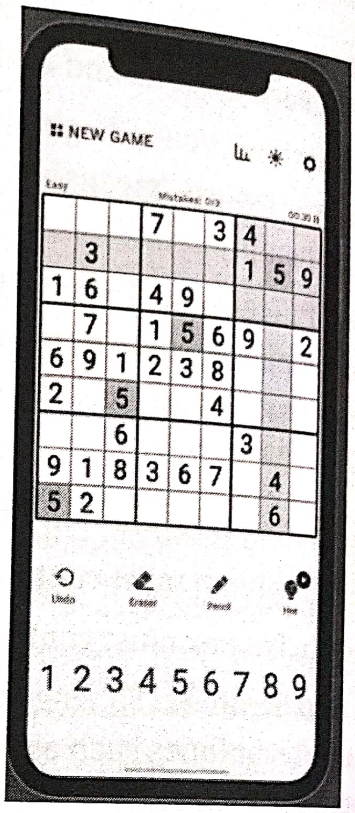
1. **Grid Structure:** The Sudoku grid consists of 81 cells, organized into a 9x9 layout. The grid is further divided into nine 3x3 subgrids.

2. **Objective:** The goal of Sudoku is to fill in the grid with numbers from 1 to 9 in such a way that each row, each column, and each 3x3 subgrid contains all the numbers exactly once.
3. **Given Numbers:** At the beginning of the game, some numbers are already provided in the grid. These are known as "givens" or "clues." The player's task is to complete the grid by filling in the remaining empty cells.
4. **Number Placement:** To solve the puzzle, players must use logic and deduction to determine the correct placement of numbers in each cell. The numbers must adhere to the rules of Sudoku to ensure that no row, column, or subgrid contains repeated numbers.
5. **Rules :**
   - Each row must contain the numbers 1 to 9, without repetition.
   - Each column must contain the numbers 1 to 9, without repetition.
   - Each 3x3 box must contain the numbers 1 to 9, without repetition.
6. **Logic and Strategy:** Solving Sudoku requires logical thinking and strategic planning. Players often use techniques such as scanning, cross-hatching, and elimination to deduce the correct numbers for each cell.
7. **Unique Solution:** A valid Sudoku puzzle has only one unique solution. This means that there is only one way to correctly fill in the grid while following the rules of the game.
8. **Difficulty Levels :** Sudoku puzzles come in varying levels of difficulty, ranging from easy to hard. The complexity of the puzzle is determined by the number and placement of initial givens.
9. **Error Avoidance:** In Sudoku, making a mistake can lead to an unsolvable puzzle. Therefore, players must be careful and methodical in their number placements to avoid errors.

The solution to the above sudoku is given below

1	7	6	5	8	3	9	2	4
8	5	9	2	7	4	3	6	1
3	4	2	9	1	6	5	8	7
7	9	5	1	4	8	6	3	2
4	2	3	6	9	7	1	5	8
6	8	1	3	5	2	7	4	9
9	1	4	8	3	5	2	7	6
2	3	8	7	6	1	4	9	5
5	6	7	4	2	9	8	1	3

Traditionally, Sudoku is played using pencil and paper, but digital versions have gained popularity. In the paper format, errors can lead to significant rework, requiring the erasure of previous entries. In contrast, the mobile app version offers the advantage of easily modifying entries without the hassle of erasing mistakes manually. The Sudoku example program for Android is shown in the figure.



6.3 Designing by Declaration

In the context of designing user interfaces, two main approaches are commonly used: **procedural** and **declarative**.

1. **Procedural Design:** Procedural design involves creating user interfaces using code. For example, in a Swing application, developers write Java code to define and manipulate UI components like JFrame and JButton. This approach is known for its programmatic nature, where the logic and structure of the UI are defined through code.
2. **Declarative Design:** Declarative design, on the other hand, focuses on describing the UI in a markup language without the need for extensive code. For example, when designing a web page, HTML is used to specify the layout and content presentation without detailing the implementation logic. XML is another markup language commonly used for declarative UI design.

Android's Approach

- Android provides flexibility by supporting both procedural (Java code) and declarative (XML) approaches for creating user interfaces. Developers can choose to work predominantly in Java or XML to define the UI components and their attributes.
- Android documentation typically includes information on both the Java APIs and the corresponding XML attributes for UI components, allowing developers to select their preferred method.

Google Recommendation :

Google suggests utilizing declarative XML for designing Android user interfaces whenever possible. XML code is often more concise, easier to comprehend, and less prone to changes in future Android versions compared to Java code.

What is Designing by Declaration?

Designing by declaration refers to the approach of creating user interfaces by defining their structure and appearance using declarative markup languages such as XML, rather than writing procedural code. In this method, developers specify the layout, properties, and behavior of UI components through descriptive markup to focus on what should be displayed rather than how it should be implemented.

In the context of Android app development, designing by declaration involves using XML files to define the visual elements and attributes of the user interface. By separating the UI design from the application logic, developers can achieve a more modular and maintainable codebase. This approach allows for a clear separation of concerns, making it easier to manage and update the UI independently of the underlying functionality.

Advantages of Declarative Design

1. **Clarity and Readability:** XML markup is often more readable and intuitive than complex procedural code for developers to understand and modify the UI layout in easy manner.
2. **Ease of Maintenance:** Changes to the UI can be made more efficiently by editing the declarative markup, without the need to modify the application logic.
3. **Consistency:** Declarative design promotes consistency in UI elements and layouts across the application to enhance the overall user experience.
4. **Collaboration:** Designers and developers can collaborate effectively by working on separate aspects of the UI, with designers focusing on the visual presentation using XML files.

6.4 Creating the Opening Screen

An opening screen. also known as a splash screen is the starting page of an app. The opening screen serves as the entry point to the application and often provides essential information, options, or actions for users to interact with. This screen is crucial as it provides the first impression and navigational options for the user.

Key Characteristics and Components of an Opening Screen

The opening screen sets the tone for the user experience and serves as a gateway to the application's features and functionalities. It is designed to capture users' attention, provide a seamless entry point into the application, and guide users towards their intended actions or destinations within the app. The key characteristics and components of an openning screen are listed below.

- **Introduction:** The opening screen may introduce the application, provide a brief overview of its purpose, or display the application's logo or branding elements.
- **Navigation:** It typically offers navigation options for users to move to different sections of the application such as starting a new task, accessing settings, viewing tutorials, or exploring features.
- **Call to Action:** The opening screen may include prominent buttons or prompts that encourage users to take specific actions such as signing in, creating an account, or starting a new activity.

- **Visual Design:** The design of the opening screen plays a crucial role in creating a positive first impression on users. It often incorporates visually appealing elements, colors, and graphics to engage users.
- **Functionality:** Depending on the application's purpose, the opening screen may include interactive elements, animations, or multimedia content to enhance user engagement.
- **Loading Indicators:** In cases where the application requires time to load or initialize, the opening screen may display loading indicators or progress bars to inform users about the process.

The opening screen should be visually appealing, user-friendly, and offer clear navigation to ensure users can easily access the main features of the app.

UI Requirements for the Sudoku Opening Screen

For a Sudoku game, the opening screen serves as the gateway to the various functionalities of the game. The specific requirements include:

1. **Starting a New Game :** A prominent button that allows users to start a new Sudoku game session.
2. **Continuing a Previous Game :** A button that enables users to resume a Sudoku game they had previously saved.
3. **Viewing Information About Sudoku :** A button to display instructions or information about how to play Sudoku, including rules and tips.
4. **Exiting the App :** A button to close the Sudoku application.

The Sudoku opening screen should be designed with the following considerations:

- **Layout :** Use a simple and intuitive layout that guides the user to their desired action without confusion.
- **Design :** Incorporate a visually appealing design that matches the theme of Sudoku, possibly using calm and neutral colors to reflect the puzzle nature of the game.
- **User Experience :** Ensure the screen is responsive and accessible, with buttons that are easy to read and interact with.

By focusing on these aspects, the opening screen will not only serve its functional purposes but also enhance the overall user experience, making the app inviting and easy to use.

Step-by-Step Process for Creating the Sudoku Opening Screen

Step 1. Project Setup:

The first step in creating any Android application is to set up the project. This involves defining the project name, package name, language and minimum SDK version, among other settings. This ensures that the application is correctly configured and compatible with the desired range of Android devices.

1. Open Android Studio.
2. Start a New Android Studio Project.
3. Select "Empty View Activity" Template.

4. Name Your Project: 'Sudoku'.
5. Set Language: Java.
6. Set the minimum SDK to "API 21: Android 5.0 (Lollipop)".
7. Finish Project Setup

ep 2. Define the Layout in XML:

Layouts in Android define the structure of the user interface. XML files are used to create a declarative layout, which is preferred over procedural layouts due to its simplicity and readability. The layout file specifies how UI elements are arranged on the screen. The below code create and define the layout for the opening screen of the Sudoku game using XML.

Code activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/background"
    android:padding="30dp">

    <TextView
        android:id="@+id/main_title"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/main_title"
        android:textSize="24sp"
        android:layout_gravity="center"
        android:layout_marginTop="100dp"
        android:layout_marginBottom="25dp"
        android:textColor="@color/text_color" />

    <Button
        android:id="@+id/continue_button"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/continue_label" />

    <Button
        android:id="@+id/new_game_button"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/new_game_label" />
```

```
    <Button
        android:id="@+id/about_button"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/about_label" />

    <Button
        android:id="@+id/exit_button"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/exit_label" />
</LinearLayout>
```

Explanation

This layout XML defines a simple and clean opening screen for the Sudoku game, with a centered title and four buttons arranged vertically for different actions.

1. **LinearLayout Container:**
   - The root element of the layout is a '<LinearLayout>'. This container arranges its child views in a single column (vertical orientation).
   - Attributes such as 'android:layout_width="match_parent"' and 'android:layout_height="match_parent"' ensure that the layout fills the entire screen. The background color is set using a reference to a color resource ('@color/background'), and padding is applied ('android:padding="30dp"') to provide space around the edges.
2. **TextView for Title:**
   - Inside the 'LinearLayout', a '<TextView>' is used to display the main title of the app. The 'android:id="@+id/main_title"' attribute assigns a unique ID to this 'TextView' so it can be referenced in the code.
   - The text to be displayed is referenced from a string resource ('@string/main_title'). Other attributes include 'android:textSize="24sp"' for text size, 'android:layout_gravity="center"' to center the title horizontally, 'android:layout_marginBottom="25dp"' for spacing below the title, and 'android:textColor="@android:color/white"' to set the text color to white.
3. **Continue Button:**
   - A '<Button>' is defined for continuing a previous game. The 'android:id="@+id/continue_button"' attribute assigns a unique ID to this button.
   - The button text is set using a reference to a string resource ('@string/continue_label'). The 'android:layout_width="match_parent"' attribute makes the button span the full width of the parent container, while 'android:layout_height="wrap_content"' allows the button height to adjust based on its content.
4. New Game Button:
   - Another '<Button>' is defined for starting a new game. It has the ID '@+id/new_game_button' and the text is set from a string resource ('@string/new_game_label').
   - Similar to the continue button, the layout width and height are set to 'match_parent' and 'wrap_content', respectively, ensuring consistent button sizing.

5. About and Exit Buttons:

- Two more '<Button>' elements are defined for displaying information about the game ('@+id/about_button' with text '@string/about_label') and for exiting the app ('@+id/exit_button' with text '@string/exit_label').
- Both buttons follow the same layout attributes as the previous buttons, ensuring they fill the parent container's width and adjust their height based on content.

ep 3. Define Strings in strings.xml:

String resources allow text strings to be defined separately from the code and hence it is easier to manage and localize text in the application. The below code define the text strings used in the layout in the 'strings.xml' file.

Code res/values/strings.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">Sudoku</string>
    <string name="main_title">Android Sudoku</string>
    <string name="continue_label">Continue</string>
    <string name="new_game_label">New Game</string>
    <string name="about_label">About</string>
    <string name="exit_label">Exit</string>
</resources>
```

Explanation

- **<resources>:** The root element for defining resources.
- **<string name="...">:** Defines a string resource with a specific name and value.

For example, '<string name="app_name">Sudoku</string>' defines a string resource named 'app_name' with the value "Sudoku".

These string resources are referenced in the layout XML file using the '@string/' syntax, allowing for easy updates and localization.

tep 4. Define Colors in colors.xml:

Color resources allow colors to be defined and reused throughout the application to ensure a consistent color scheme. The below code define the background color used in the layout in the 'colors.xml' file.

Code res/values/colors.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="background">#FFFFFF</color>
    <color name="text_color">#000000</color>
</resources>
```

Explanation

- **<color name="...">:** Defines a color resource with a specific name and hex color value. For example, '<color name="background">#FFFFFF</color>' defines a color resource named 'background' with the hex value '#FFFFFF'. (White color).

This color resource is referenced in the layout XML file using the '@color/' syntax to set the background color or text_color of the 'LinearLayout'.

Step 5. Define the Activity in Java:

Activities are the entry points for interacting with the user. They provide the window in which the app draws its UI. The 'onCreate' method is where the activity's initial setup is done, such as creating the layout and setting up event listeners. The below code define the main activity for the Sudoku app in Java.

Code MainActivity.java

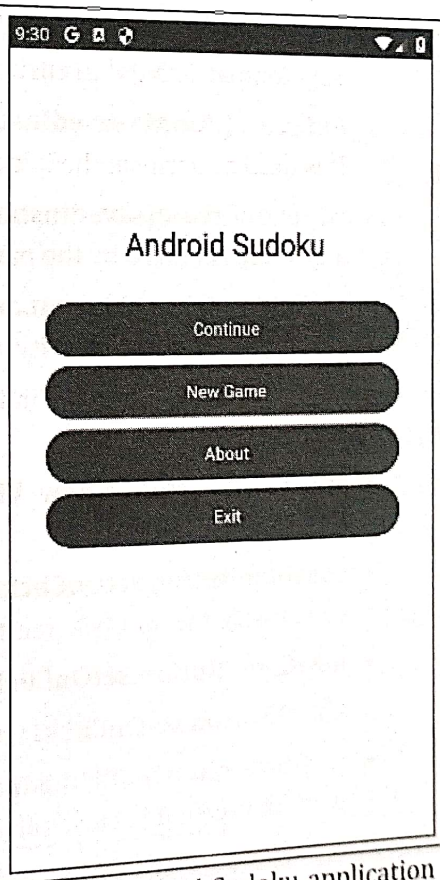
```
package com.example.sudoku;

import android.os.Bundle;
import androidx.appcompat.app.AppCompatActivity;
import android.view.View;
import android.widget.Button;

public class MainActivity extends AppCompatActivity {

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

        Button continueButton = findViewById(R.id.continue_button);
        Button newGameButton = findViewById(R.id.new_game_button);
        Button aboutButton = findViewById(R.id.about_button);
        Button exitButton = findViewById(R.id.exit_button);

        continueButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                // Handle continue button click
            }
        });

        newGameButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                // Handle new game button click
            }
        });
```

```
        aboutButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                // Handle about button click
            }
        });

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

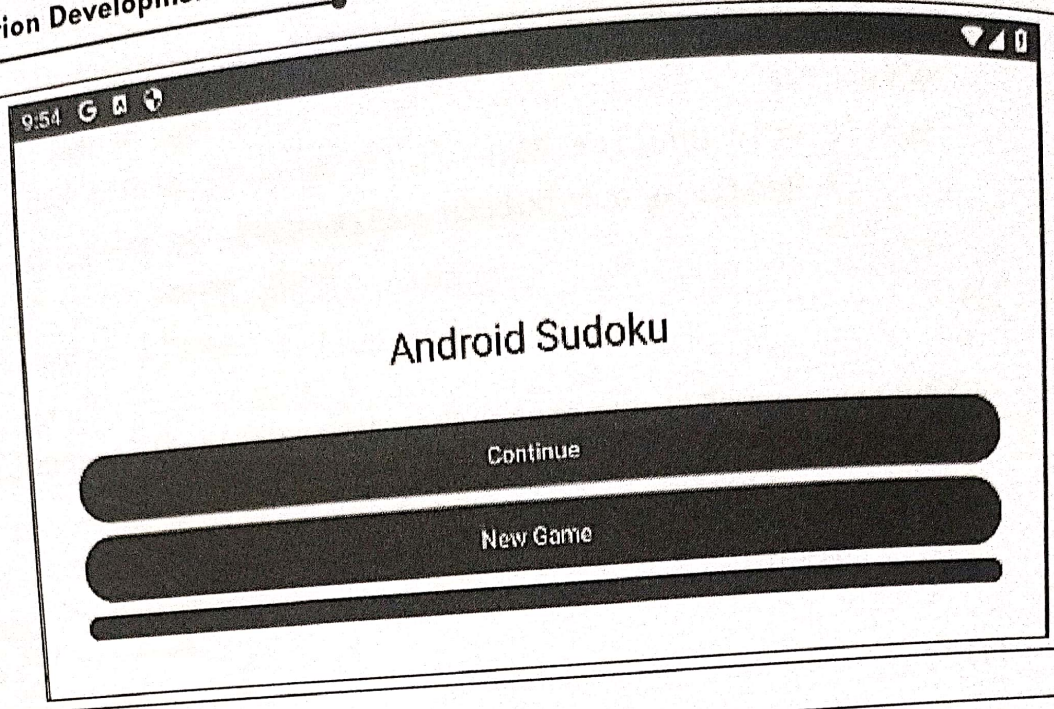
Output

The below displayed output in portrait mode features the opening screen for the Android Sudoku application. The screen showcases a clean and minimalistic design with a light background, to ensure readability and a pleasant user experience. At the top center of the screen, a TextView displays the title "Android Sudoku" in black text to provide a clear introduction to the app.

Below the title, four buttons are arranged vertically, each labeled with a different function: "Continue," "New Game," "About," and "Exit."



In landscape mode, the output shows the opening screen for the Android Sudoku application with a layout issue where not all buttons are displayed. We can observe that only two buttons, "Continue" and "New Game," are visible, and the "About" and "Exit" buttons are missing. This issue occurs because the vertical space available in landscape mode is reduced compared to portrait mode. The LinearLayout is set to vertical orientation, meaning all buttons are stacked vertically. When the screen is rotated to landscape mode, there isn't enough vertical space to display all four buttons due to the constraints of the device's screen height.



Explanation

- **package org.example.sudoku;:** Declares the package for the class.
- **import statements:** Import necessary classes from the Android and AndroidX libraries.
- **MainActivity extends AppCompatActivity:** Defines the main activity class, inheriting from 'AppCompatActivity' to ensure compatibility with older Android versions.
- **onCreate(Bundle savedInstanceState):** This method is called when the activity is first created. It is used to perform the initial setup for the activity.
- **super.onCreate(savedInstanceState):** Calls the superclass's 'onCreate' method to perform any setup required by the parent class.
- **setContentView(R.layout.activity_main):** Sets the layout for the activity using the layout resource defined in 'activity_main.xml'.
- **findViewById(R.id....):** Finds views by their ID and assigns them to variables for further manipulation.
- **setOnClickListener(new View.OnClickListener()):** Sets click listeners for the buttons to handle user interactions.
- **continueButton.setOnClickListener(...):** Sets a click listener for the "Continue" button. The code inside the 'onClick' method will be executed when the button is clicked.
- **newGameButton.setOnClickListener(...):** Sets a click listener for the "New Game" button.
- **aboutButton.setOnClickListener(...):** Sets a click listener for the "About" button.
- **exitButton.setOnClickListener(...):** Sets a click listener for the "Exit" button. The 'finish()' method is called to close the activity when the button is clicked.

6.5 Using Alternate Resources

Switching the emulator or physical device to landscape mode may result in a few buttons running off the bottom of the screen. This issue can be observed in the landscape mode output shown in the previous section. Adjusting the layout to work with all orientations can be challenging and often leads to odd-looking screens. When this occurs, it is necessary to create a different layout specifically for landscape mode or use a 'ScrollView'. These approaches ensure that the application maintains a user-friendly and visually appealing interface in both orientations.

6.5.1 Using ScrollView

A ScrollView is a layout container that enables scrolling for a single child view. It is useful when the content is too large to fit on the screen. By wrapping the existing layout in a ScrollView, the user can scroll through the content to ensure all buttons are accessible regardless of the screen orientation.

Code **activity_main.xml**

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/background"
    android:padding="30dp">

    <LinearLayout
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="@color/background"
        android:padding="30dp">

        <TextView
            android:id="@+id/main_title"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/main_title"
            android:textSize="24sp"
            android:layout_gravity="center"
            android:layout_marginTop="100dp"
            android:layout_marginBottom="25dp"
            android:textColor="@color/text_color" />

        <Button
            android:id="@+id/continue_button"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="@string/continue_label" />

        <Button
            android:id="@+id/new_game_button"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="@string/new_game_label" />
```

```
        <Button
            android:id="@+id/about_button"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="@string/about_label" />

        <Button
            android:id="@+id/exit_button"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="@string/exit_label" />
    </LinearLayout>
</ScrollView>
```

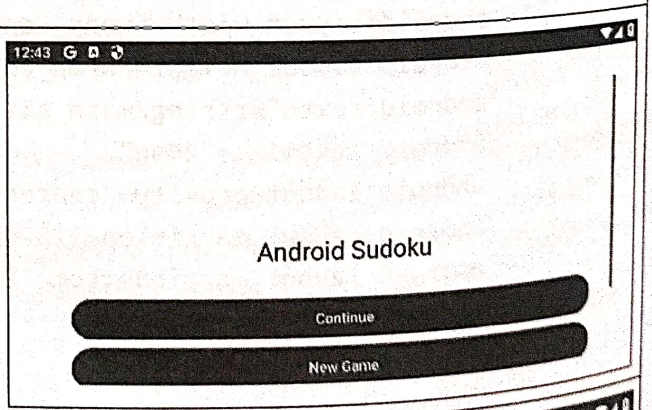
Explanation

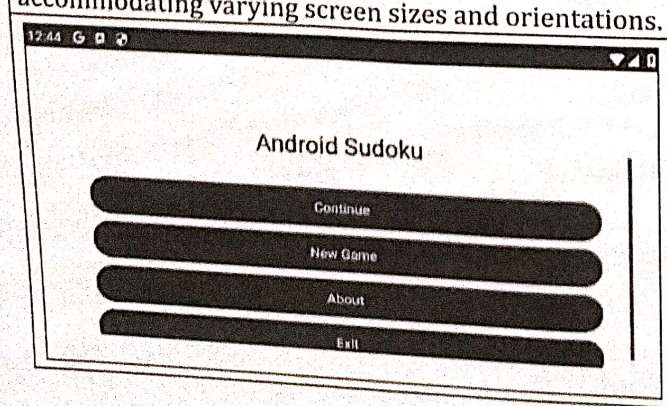
- **ScrollView:** The root element is a 'ScrollView', which allows the entire content to be scrollable vertically. This ensures that all buttons are accessible, even if they don't fit within the available vertical space on the screen.
- **LinearLayout:** The 'LinearLayout' is placed inside the 'ScrollView'. Its height is set to 'match_parent', allowing it to expand only as much as needed to fit its content.
- **TextView and Buttons:** The 'TextView' and 'Button' elements are nested within the 'LinearLayout', to ensure that they are displayed in a vertical stack.

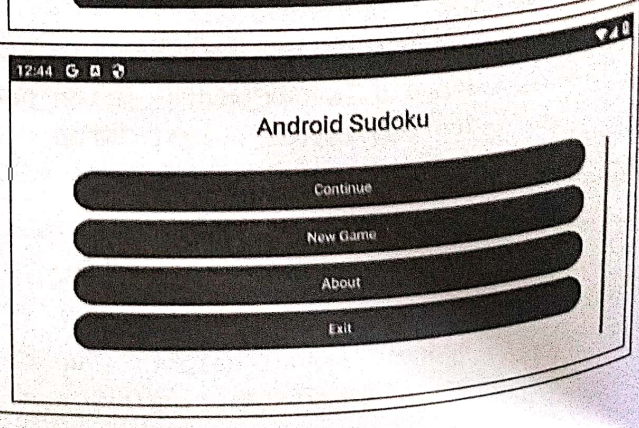
By using a 'ScrollView', the layout can adapt to different screen sizes and orientations, ensuring all UI elements are accessible.

Output

The below output demonstrate the use of a ScrollView in both portrait and landscape modes for the Android Sudoku application's opening screen. This ensures all buttons—Continue, New Game, About, and Exit—are accessible to the user. The scroll bar remains always visible for users to navigate vertically through the options seamlessly, regardless of screen orientation. This design choice enhances the user experience by maintaining a consistent and clean interface while accommodating varying screen sizes and orientations.







6.5.2 Using Different Layout for Landscape Mode

Creating a separate layout file for landscape mode allows for better optimization of the UI for different orientations. This approach involves defining a distinct layout that rearranges or resizes elements to fit appropriately within the horizontal space available in landscape mode.

Code **res/layout-land/activity_main.xml:**

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/background"
    android:padding="30dp"
    android:gravity="center">

    <LinearLayout
        android:orientation="vertical"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:gravity="center">

        <TextView
            android:id="@+id/main_title"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/main_title"
            android:textSize="24sp"
            android:layout_gravity="center"
            android:layout_marginBottom="25dp"
            android:textColor="@color/text_color" />

        <Button
            android:id="@+id/continue_button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/continue_label"/>

        <Button
            android:id="@+id/new_game_button"
            android:layout_width="wrap_content"
```

```
            android:layout_height="wrap_content"
            android:text="@string/new_game_label"/>

        <Button
            android:id="@+id/about_button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/about_label"/>

        <Button
            android:id="@+id/exit_button"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/exit_label"/>
    </LinearLayout>

</LinearLayout>
```

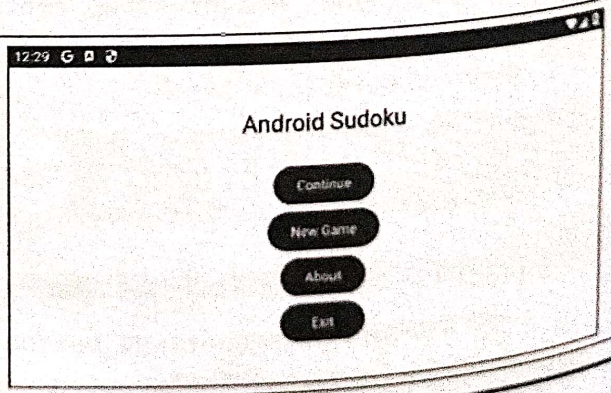
Explanation

- **Separate Layout File:** The layout file is placed in the 'res/layout-land' directory, which is automatically used by the Android system when the device is in landscape mode.
- **Horizontal LinearLayout:** The root element is a 'LinearLayout' with horizontal orientation, allowing the buttons and title to be arranged side by side, optimizing the use of horizontal space.
- **Vertical Inner LinearLayout:** Inside the horizontal 'LinearLayout', a vertical 'LinearLayout' is used to stack the title and buttons vertically. The 'android:layout_weight="1"' attribute ensures that the inner layout expands to fill the available vertical space, centering the elements.

By using a different layout for landscape mode, the UI can be tailored to better fit the available screen space, ensuring all elements are visible and accessible.

Output

The output image shows the application's opening screen in landscape mode with vertically centered buttons and a title. This layout ensures all UI components are visible and accessible.



6.6 Implementing an About Box

The About box is an essential feature that provides users with information about the application. An About box provides users with information about the application to enhance the overall user experience by giving insights into what the app does, its purpose, or any other relevant information. In the context of the Sudoku app, the About box will display a brief description of Sudoku, how it's

played, and its objectives. When the user selects the About button, the app will display a new screen (activity) containing the information about Sudoku. The user can read through the information and then press the Back button to return to the main screen.

The Steps to Implement an About Box

To create an About box that provides information about the Sudoku game when the user selects the About button, follow these steps. This process involves adding an activity and setting it up to display information about the game.

1. **Define a New Activity:** This activity will display the About information.
2. **Create a Layout File for the Activity:** The layout will include a ScrollView containing a TextView for displaying the content.
3. **Add String Resources:** These resources will contain the title and content for the About box.
4. **Create the Activity Class:** This class will set up the layout and handle the display of the content.
5. **Wire Up the About Button:** In the main activity, set up a click listener for the About button to start the new activity.
6. **Update AndroidManifest.xml:** Declare the new activity in the manifest file so the system recognizes it.

Step-by-Step Implementation of an About Box in Sudoku App

Step 1: Define the Layout for the About Activity:

Create a new XML layout file 'about.xml' in the 'res/layout' directory.

Code 'res/layout/about.xml'

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="10dp">
    <TextView
        android:id="@+id/about_content"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
            android:textSize="10pt"
        android:text="@string/about_text" />
</ScrollView>
```

Explanation

- **ScrollView:** Makes the content scrollable if it exceeds the screen size.
- **TextView:** Displays the About text, referenced from a string resource.

Step 2: Define Strings in 'strings.xml':

Add the following strings to the 'res/values/strings.xml' file.

Code 'res/values/strings.xml'

```
<string name="about_title">About Android Sudoku</string>
<string name="about_text">
    Sudoku is a logic-based number placement puzzle.
    Starting with a partially completed 9x9 grid, the
    objective is to fill the grid so that each
    row, each column, and each of the 3x3 boxes
    (also called <i>blocks</i>) contains the digits
    1 to 9 exactly once.
</string>
```

Explanation

- **about_title:** Title for the About activity.
- **about_text:** Text content for the About activity, providing information about Sudoku.

Step 3: Create the About Activity:

Create a new Java class 'About.java' in the 'src/main/java/com/example/sudoku' directory.

Code 'About.java'

```
package com.example.sudoku;

import android.app.Activity;
import android.os.Bundle;

public class About extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.about);
    }
}
```

Explanation

- **About Class:** Defines a new activity that extends 'Activity'.
- **onCreate Method:** Sets the content view to 'about.xml' when the activity is created.

Step 4: Modify MainActivity to Wire Up the About Button:

Update 'MainActivity.java' to handle the About button click and start the About activity.

- **Changes to 'MainActivity.java':**
  1. **Add imports:**

Code

```
import android.content.Intent;
```

2. **Set click listener for the About button in the 'onCreate' method:**

Code

```
Button aboutButton = findViewById(R.id.about_button);
aboutButton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        Intent intent = new Intent(MainActivity.this, About.class);
        startActivity(intent);
    }
});
```

Explanation

- **Intent to Start About Activity:** In the 'aboutButton.setOnClickListener' method, an 'Intent' is created to start the 'About' activity.
- **startActivity Method:** This method is called with the 'Intent' to launch the About activity.

Step 5: Declare the New Activity in AndroidManifest.xml:

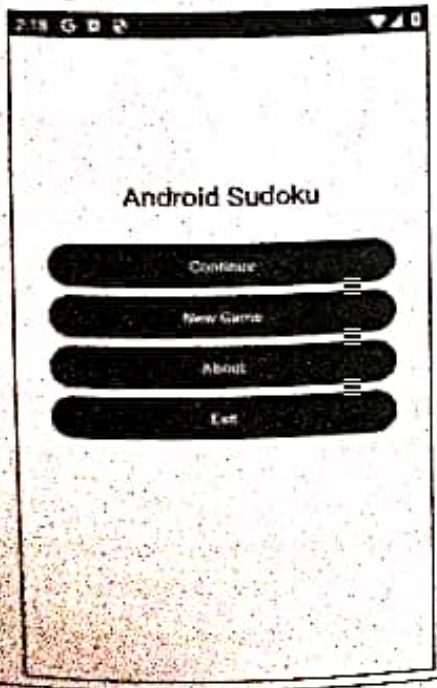
Add the following activity declaration to the 'AndroidManifest.xml' file.

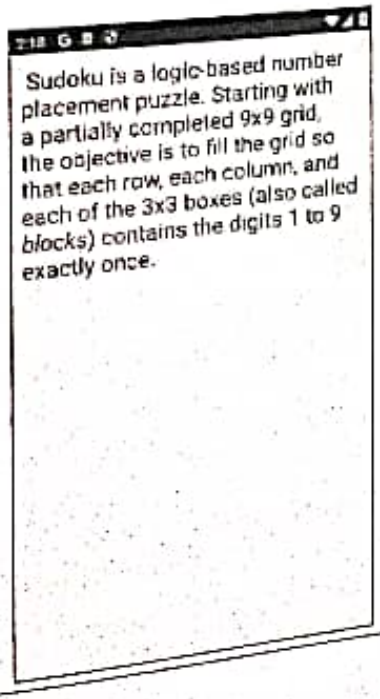
Code 'AndroidManifest.xml'

```
<activity android:name=".About"
    android:label="@string/about_title">
</activity>
```

Output

When users click the About button in the Sudoku application's main screen, the About activity is launched, displaying information about Sudoku in a scrollable view. The Back button can be used to dismiss the window and return to the main screen. This implementation provides users with detailed information about the game.





Explanation

- **Activity Declaration:** Declares the About activity so it can be launched by the application.

6.7 Applying a Theme

A theme in Android is a collection of style attributes that define the look and feel of an application or activity. Themes are similar to Cascading Style Sheets (CSS) used in web development, as they separate the content from the presentation. This separation allows developers to apply a consistent look and feel across the entire app or specific components without modifying the underlying functionality.

Importance and Benefits of Applying a Theme

1. **Consistency:** Themes ensure a consistent user interface throughout the application. By defining a set of styles and applying them globally, all UI components adhere to the same visual standards.
2. **Reusability:** Themes allow for the reuse of style attributes. Instead of setting styles individually for each component, a theme can apply a uniform style across multiple activities or views.
3. **Maintainability:** With themes, updating the look and feel of the app is straightforward. Changes to the theme are reflected throughout the app, making it easier to maintain and update the design.
4. **Customization:** Themes provide the flexibility to customize and extend default styles. Developers can create their own themes or extend existing ones to tailor the UI to their specific needs.
5. **User Experience:** A well-designed theme enhances the user experience by providing a visually appealing and coherent interface. It makes the app look professional and polished.

Step-by-Step Implementation of Applying a Theme

Step 1: Applying a Default Theme:

The first step is to modify the 'AndroidManifest.xml' file to apply a predefined theme to the About activity.

▲ **Modify the About Activity Declaration**

Update the About activity declaration to include the 'android:theme' property.

Code 'AndroidManifest.xml'

```
<activity android:name=".About"
    android:label="@string/about_title"
    android:theme="@android:style/Theme.Dialog">
</activity>
```

Explanation

- **android:theme:** This attribute applies the specified theme to the activity. The '@android:style/Theme.Dialog' reference points to a predefined theme that styles the activity as a dialog.
- **@android:** prefix: Indicates that the referenced resource is defined by Android, not by the application.

Step 2: Applying a Custom Theme:

If a predefined theme doesn't meet the requirements, a custom theme can be created by defining styles in 'res/values/styles.xml'.

▲ **Define a Custom Theme:**

Code 'res/values/styles.xml'

```
<resources>
    <style name="CustomDialogTheme" parent="Theme.AppCompat.Dialog">
        <item name="android:windowBackground">@color/background</item>
        <item name="android:textColor">@color/text_color</item>
        <!-- Add more style attributes as needed -->
    </style>
</resources>
```

Explanation

- **CustomDialogTheme:** A custom theme that extends an existing theme ('Theme.AppCompat.Dialog'). Additional style attributes can be defined to customize the look and feel of the dialog.

▲ **Apply the Custom Theme:**

Update the About activity declaration to include the custom theme in the 'AndroidManifest.xml'.

Code 'AndroidManifest.xml'

```
<activity android:name=".About"
    android:label="@string/about_title"
    android:theme="@style/CustomDialogTheme">
</activity>
```

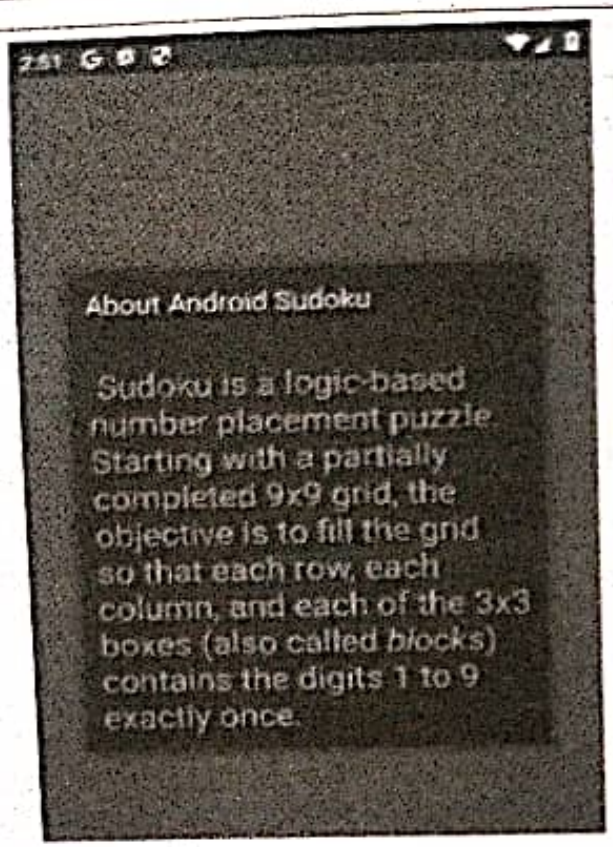
Explanation

- **android:theme:** Applies the custom theme to the activity. The '@style/CustomDialogTheme' reference points to the custom theme defined in 'styles.xml'.

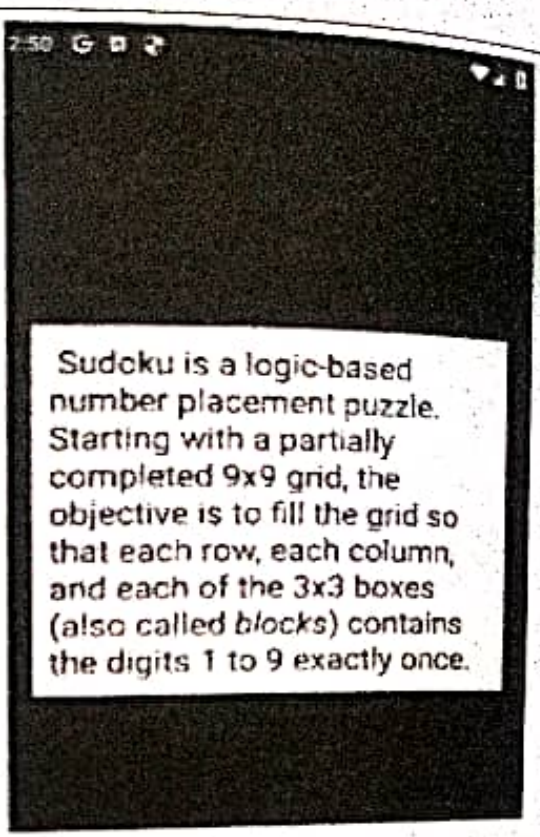
Output

The About box will now appear as a dialog, making it visually distinct from the rest of the application. This style change helps in emphasizing the informational nature of the About box and makes it look more integrated with the overall app design.

When users click the About button, the About activity is displayed as a dialog. This provides a clear and focused presentation of the information about Sudoku, with the ability to scroll through the text if needed. The dialog can be dismissed by pressing the Back button, returning the user to the main screen.



Default Theme



Custom Theme

Different between Styles vs Themes

Feature	Style	Theme
Definition	A set of attributes that can be applied to a specific view or view group.	A set of attributes that can be applied to an entire activity or application.
Application	Applied directly to views in layout XML files or programmatically.	Applied to activities or the entire application in the AndroidManifest.xml or programmatically.
Inheritance	Can inherit styles from other styles using the parent attribute.	Can inherit themes from other themes using the parent attribute.
Scope	Applied only to the view or view group to which it is assigned.	Applied globally to all views or activities that use the theme.
Attribute Override	Can override specific attributes of a view, such as text color or background.	Can override theme attributes for specific views or view groups using theme overlays.
Reusability	Can be reused across different views or view groups within the same app.	Can be reused across different activities or the entire app.
Use Cases	Used for defining view-specific appearance, such as text color or background.	Used for defining app-wide appearance, such as color scheme or font family.

6.8 Adding a Menu

Android supports two kinds of menus: the **options menu**, which appears when the user presses the physical Menu button or taps the overflow menu icon (three vertical dots) on modern devices, and

he **context menu**, which pops up when the user presses and holds on the screen. Let us discuss on ow to add an options menu that opens when the user interacts with the the overflow menu icon to ccess to settings and other options.

Step-by-Step Implementation

Step 1: Change the Theme to Include the ActionBar:

This step is to ensure the application theme includes an ActionBar to display the menu. The ActionBar provides a familiar navigation and menu interaction for users.

We need to modify the 'styles.xml' file to extend a theme that includes the ActionBar. This involves setting the base theme to 'Theme.Material3.DayNight', which includes ActionBar support.

Code 'res/values/styles.xml'

```
<resources xmlns:tools="http://schemas.android.com/tools">
    <!-- Base application theme. -->
    <style name="Theme.Sudoku" parent="Theme.Material3.DayNight" />
    <style name="CustomDialogTheme" parent="Theme.AppCompat.Dialog">
            <item name="android:windowBackground">@color/background</item>
            <item name="android:textColor">@color/text_color</item>
    </style>
</resources>
```

Explanation

- **parent="Theme.Material3.DayNight":** Sets the base theme to one that includes ActionBar support.
- **Theme.Sudoku:** Defines the primary theme for the Sudoku application.
- **CustomDialogTheme:** Defines a custom theme for dialog windows with specified background and text colors.

Step 2: Define Strings for Menu Items:

We need to define the text labels for the menu items to ensure they are easily configurable and maintainable. Add the necessary string resources in 'strings.xml' to define labels for the menu items.

Code 'res/values/strings.xml'

```
    <string name="settings_label">Settings...</string>
    <string name="hints_title">Show Hints</string>
    <string name="hints_summary">Show Past Results</string>
```

Explanation

- **settings_label:** Defines the label for the Settings menu item.
- **show_hints:** Defines the label for the Show Hints option.
- **show_past_results:** Defines the label for the Show Past Results option

Step 3: Define the Menu in XML:

We need to create a menu resource file that defines the structure and content of the options menu. Create a new XML file 'menu.xml' in the 'res/menu' directory to define the menu items.

Code 'res/menu/menu.xml'

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/settings"
        android:title="@string/settings_label" />
    <item android:id="@+id/show_hints"
        android:title="@string/show_hints" />
    <item android:id="@+id/show_past_results"
        android:title="@string/show_past_results" />
</menu>
```

Explanation

- **menu:** The root element for defining a menu.
- **item:** Defines a single menu item. The android:id attribute uniquely identifies the item, while android:title sets the label.

Step 4: Modify MainActivity to Handle the Menu:

We need to inflate the menu defined in XML and handle menu item selections within the MainActivity. Update 'MainActivity.java' to include the necessary methods to inflate the menu and respond to menu item selections.

1. **Add the necessary imports to 'MainActivity.java':**

Code

```
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
```

Explanation

- **Menu:** Represents the options menu.
- **MenuInflater:** Used to instantiate menu XML files into Menu objects.
- **MenuItem:** Represents a single item in a menu.

2. **Override the 'onCreateOptionsMenu' method to inflate the menu:**

Code

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    super.onCreateOptionsMenu(menu);
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.menu, menu);
    return true;
}
```

Explanation

- **onCreateOptionsMenu:** This method is called when the options menu is first created.
- **super.onCreateOptionsMenu(menu):** Calls the superclass implementation.
- **MenuInflater inflater = getMenuInflater():** Gets a MenuInflater to instantiate menu XML files.
- **inflater.inflate(R.menu.menu, menu):** Inflates the menu from the menu.xml resource file into the provided Menu object.

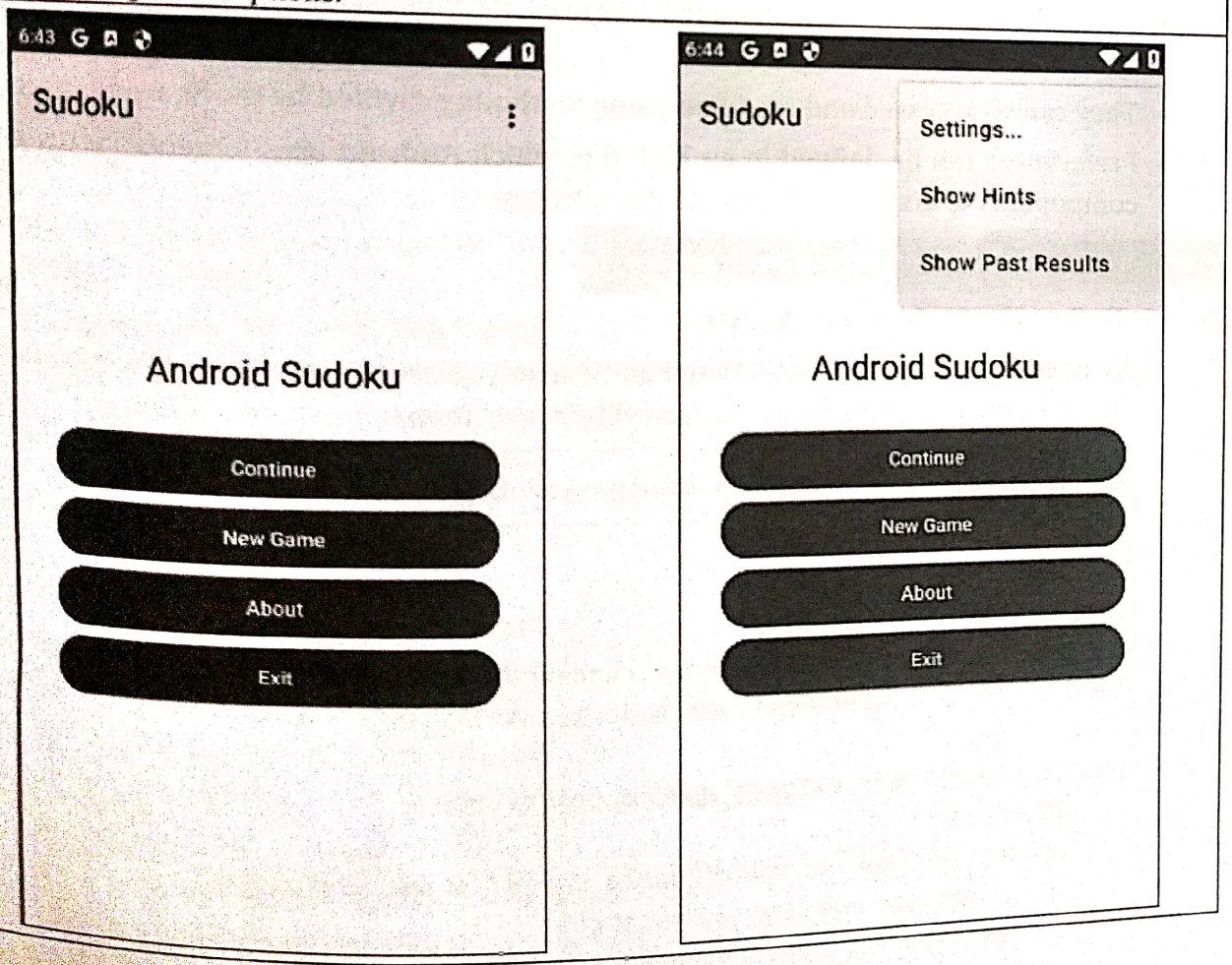
3. **Override the 'onOptionsItemSelected' method to handle menu item selections:**

Code

```
@Override
  public boolean onOptionsItemSelected(MenuItem item) {
        //  We will implement the actions in next section
        return false;
    }
```

Output

When the Menu button is pressed or the overflow menu icon is tapped, a menu with options for Settings, Show Hints, and Show Past Results appears. The selection of these menu items can be further implemented to perform their respective actions. This enhances the user experience by providing easy access to configurable options.



Explanation

- **onOptionsItemSelected:** This method is called whenever an item in the options menu is selected.
- **MenuItem item:** Represents the selected menu item.
- **return false:** Indicates that the menu item selection has not been handled yet. Actions will be implemented in the next section.

6.9 Adding Settings

We have already added the menu with menu items Settings, Show Hints, and Show Past Results. Now, by clicking on Settings, let's create preferences. Preferences allow users to configure settings that control the behavior and appearance of the application.

Understanding Preferences

Preferences are used to save and retrieve key-value pairs of primitive data types. They provide a simple way for apps to store user preferences such as settings and retrieve them when needed. Preferences are essential for maintaining user-specific configurations to improve user experience by allowing personalization and to ensure that settings persist across app sessions.

How Preferences Work

- Preferences are stored in a SharedPreferences file, which is essentially a key-value pair storage system.
- They can be accessed and modified using methods provided by the SharedPreferences class.
- Preferences can be defined in an XML file, which Android uses to generate the necessary UI components to display and manage the settings.

Step-by-Step Implementation of Preferences

Step 1: Define the Preferences Activity:

We need to create a new activity to handle user preferences for users to configure settings for the app. Let us create a new Java class 'Prefs.java' to manage the preferences.

Code 'src/main/java/com/example/sudoku/Prefs.java'

```
package com.example.sudoku;

import android.os.Bundle;
import androidx.preference.PreferenceFragmentCompat;
import androidx.appcompat.app.AppCompatActivity;

public class Prefs extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        getSupportFragmentManager().beginTransaction()
```

```
                .replace(android.R.id.content, new PrefsFragment())
                .commit();
    }

    public static class PrefsFragment extends PreferenceFragmentCompat {
        @Override
        public void onCreatePreferences(Bundle savedInstanceState,
                                        String rootKey) {
            setPreferencesFromResource(R.xml.preferences, rootKey);
        }
    }
}
```

Explanation

- **Prefs Class:** Extends 'AppCompatActivity' to provide a container for the preferences fragment.
- **onCreate:** Sets up the fragment transaction to display the 'PrefsFragment'.
- **PrefsFragment Class:** Extends 'PreferenceFragmentCompat' to display the preferences defined in the 'preferences.xml' file.
- **onCreatePreferences:** Loads the preferences from the XML resource.

Step 2: Define the Preferences in XML:

We need to define the preferences options available for users to configure. Create a new XML file 'preferences.xml' to define the preferences.

Code

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
    <ListPreference
        android:key="difficulty_level"
        android:title="Difficulty Level"
        android:summary="Select the difficulty level of the game"
        android:entries="@array/difficulty_levels"
        android:entryValues="@array/difficulty_values"
        android:defaultValue="medium" />
    <SwitchPreferenceCompat
        android:key="download_data"
        android:title="Download Data"
        android:summary="Enable or disable downloading additional data"
        android:defaultValue="false" />
</PreferenceScreen>
```

Explanation

- **PreferenceScreen:** The root element for defining a screen of preferences.
- **ListPreference:** Represents a preference with a list of entries to choose from.
- **android:key:** Unique identifier for the preference.

- **android:title:** Title displayed for the preference.
- **android:summary:** Summary text providing additional information about the preference.
- **android:entries:** The displayed names for the list items.
- **android:entryValues:** The values corresponding to each entry.
- **android:defaultValue:** Default value for the preference.
- **SwitchPreferenceCompat:** Represents a preference with a switch that can be toggled on and off.
- **android:key:** Unique identifier for the preference.
- **android:title:** Title displayed for the preference.
- **android:summary:** Summary text providing additional information about the preference.
- **android:defaultValue:** Default value for the preference.

Step 3: Define Preference Entries in Arrays XML:

We need to provide the entries and values for the ListPreference. Create a new XML file 'arrays.xml' to define the entries and values for the difficulty level preference.

Code 'res/values/arrays.xml'

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array name="difficulty_levels">
        <item>Easy</item>
        <item>Medium</item>
        <item>Hard</item>
    </string-array>
    <string-array name="difficulty_values">
        <item>easy</item>
        <item>medium</item>
        <item>hard</item>
    </string-array>
</resources>
```

Explanation

- **string-array:** Defines an array of strings.
- **name:** Unique identifier for the array.
- **item:** Represents each item in the array.

Step 4: Update MainActivity:

To connect the settings menu item to the preferences activity, the 'onOptionsItemSelected' method in 'MainActivity' needs to be updated.

Code

```
@Override
    public boolean onOptionsItemSelected(MenuItem item) {
        int id = item.getItemId();
        if (id == R.id.settings) {
```

```
        startActivity(new Intent(this, Prefs.class));
        return true;

    }
    // More items can be added here if needed
    return false;

}
```

Step 5: Register the Preferences Activity in the Manifest:

We need to ensure that the preferences activity is registered in the AndroidManifest.xml so it can be launched. Add the preferences activity to the 'AndroidManifest.xml' file.

Code 'AndroidManifest.xml'

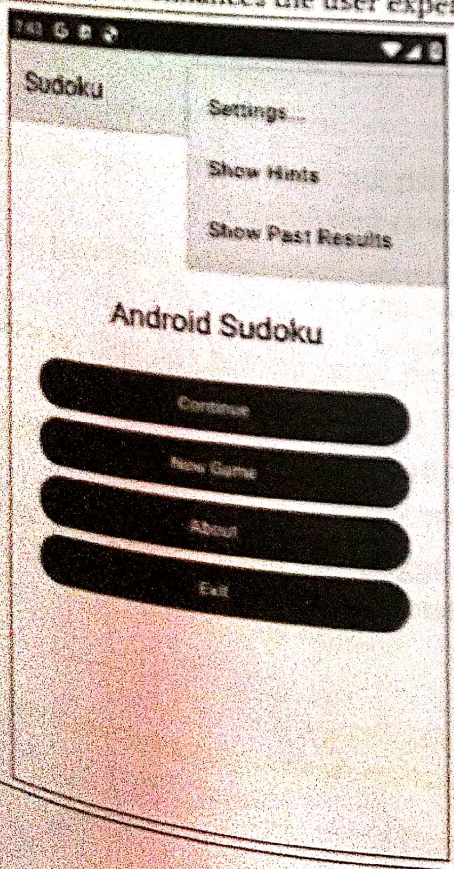
```
<activity android:name=".Prefs"
    android:label="@string/settings_title" />
```

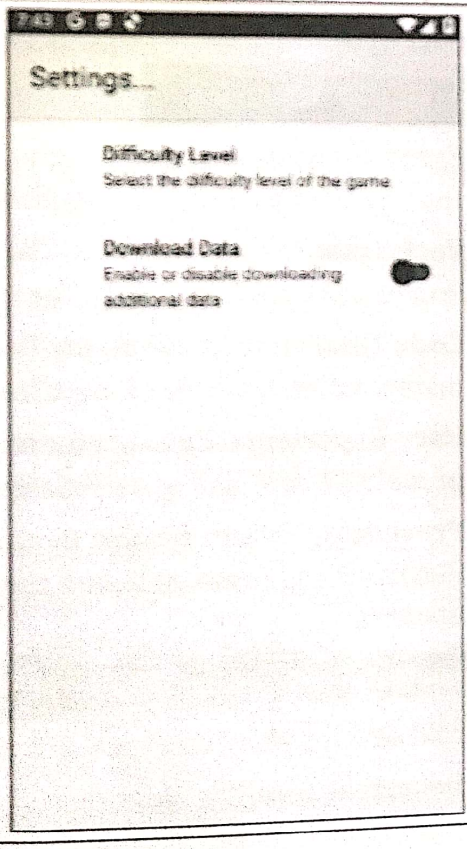
Explanation

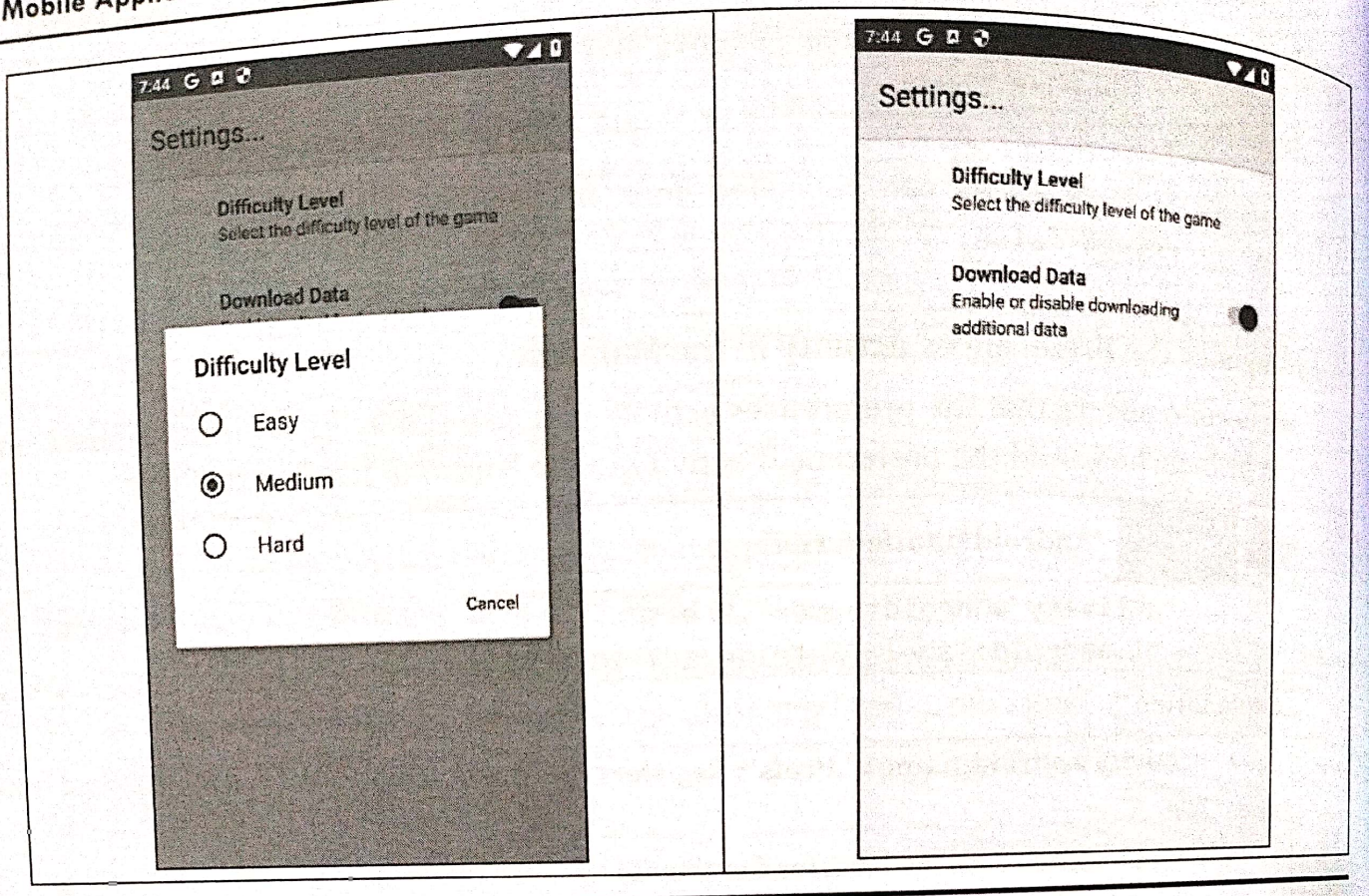
- **<activity android:name=".Prefs">:** Registers the 'Prefs' activity so it can be launched from the app.
- **android:label:** Sets the label for the activity.
- **android:theme:** Specifies the theme for the activity (if needed).

Output

When the Menu button is pressed, a menu with a Settings option appears. Selecting the Settings option opens a preferences screen where the user can toggle settings such as "Download Data" and select the difficulty level. This enhances the user experience by providing easy access to configurable options.







6.10 Debugging

Debugging is the process of identifying, analyzing, and fixing bugs or issues in software. It is an essential part of the software development lifecycle to ensure that applications run smoothly and meet the desired functionality and performance standards. Android provides various tools and techniques to facilitate the debugging process and ensure the quality and performance of their apps.

Importance of Debugging

1. **Error Identification:** Debugging helps in identifying errors or bugs in the code, which can then be fixed to ensure the application functions as intended.
2. **Performance Optimization:** The inefficiencies in the code can be identified and optimized to improve the performance of the application.
3. **Code Quality Improvement:** Regular debugging ensures that the codebase remains clean, maintainable, and free of bugs, leading to higher quality software.
4. **User Experience Enhancement:** Debugging ensures that the application runs smoothly to provide a better and more reliable experience for users.
5. **Preventing Future Issues:** By thoroughly debugging the application, potential issues can be identified and resolved before they cause problems. It reduces the likelihood of bugs in future updates.

Approaches to Debugging in Android

In Android development, there are two primary approaches to debugging applications:

1. Debugging with Log Messages
2. Debugging with the Debugger

6.10.1 Debugging with Log Messages

Log messages play a crucial role in tracking the flow of execution and diagnosing issues in Android applications. The Android Log class offers several static methods that developers can utilize to print messages of different severity levels to the system log. By incorporating log messages strategically throughout the codebase, developers can gain insights into the application's behavior, identify potential issues, and facilitate the debugging process effectively.

In addition to printing messages of various severity levels, the Android Log class provides developers with the flexibility to customize log messages based on the context and importance of the information being logged. By using different log levels such as Error, Warning, Information, Debugging, and Verbose, developers can categorize and prioritize log messages to focus on critical areas of the codebase or specific functionalities that require attention.

The log messages can also include relevant contextual information, variable values, method calls, and timestamps to provide a comprehensive view of the application's runtime behavior. Developers can leverage log messages not only for debugging purposes but also for performance monitoring, tracking user interactions, and analyzing system events to enhance the overall quality and reliability of the Android application.

Log Levels and Methods

The following table summarizes the different log levels, their methods, and detailed descriptions with example usage:

Method	Description	Example Usage
Log.e	Used to log error messages, indicating serious issues that may cause the application to malfunction.	Log.e(TAG, "Error occurred while processing the request")
Log.w	Used to log warning messages, indicating potential issues that should be looked into.	Log.w(TAG, "Potential issue detected in the configuration")
Log.i	Used to log informational messages, providing useful information about the application's normal operations.	Log.i(TAG, "Application started successfully")
Log.d	Used to log debugging messages, which are useful during development to understand the flow and state of the application.	Log.d(TAG, "Variable x value is: " + x)
Log.v	Used to log verbose messages, providing detailed information for diagnostic purposes.	Log.v(TAG, "Entering method calculateResults()")
Log.wtf	Used to log critical failure messages, indicating serious errors that should never happen.	Log.wtf(TAG, "Unexpected condition occurred!")

Example Code with Log Messages

Code MainActivity.java

```
package com.example.sudoku;

import android.content.Intent;
import android.os.Bundle;
import android.util.Log;
import androidx.appcompat.app.AppCompatActivity;
import android.view.View;
import android.widget.Button;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;

public class MainActivity extends AppCompatActivity {
    private static final String TAG = "SudokuApp";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Log.i(TAG, "MainActivity created");  // Informational message

        Button continueButton = findViewById(R.id.continue_button);
        Button newGameButton = findViewById(R.id.new_game_button);
        Button aboutButton = findViewById(R.id.about_button);
        Button exitButton = findViewById(R.id.exit_button);

        continueButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Log.d(TAG, "Continue button clicked");
                // Handle continue button click
            }
        });

        newGameButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Log.d(TAG, "New Game button clicked");
                // Handle new game button click
            }
        });
```

```
    aboutButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            Log.d(TAG, "About button clicked");
            Intent intent = new Intent(MainActivity.this, About.class);
            startActivity(intent);
        }
    });

    exitButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            Log.d(TAG, "Exit button clicked");
            finish();
        }
    });
}

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    super.onCreateOptionsMenu(menu);
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.menu, menu);
    Log.i(TAG, "Options menu created");  // Informational message
    return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    int id = item.getItemId();
    if (id == R.id.settings) {
        Log.d(TAG, "Settings menu item selected");
        startActivity(new Intent(this, Prefs.class));
        return true;
    }
    Log.w(TAG, "Unhandled menu item selected: " + id);  // Warning message
    // More items can be added here if needed
    return false;
}
```

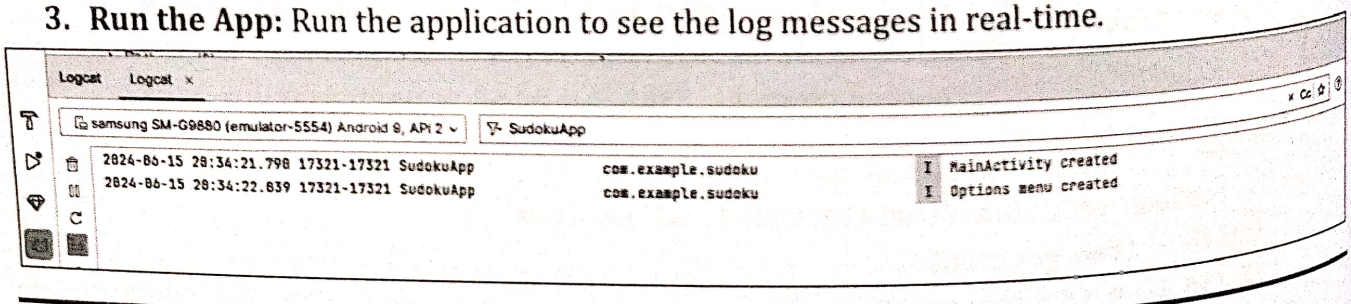
Explanation

- The log messages added in the 'MainActivity.java' code help track key events and actions. When the 'MainActivity' is created, an informational message is logged with 'Log.i(TAG, "MainActivity created")'.
- For button clicks, debugging messages such as 'Log.d(TAG, "Continue button clicked")', 'Log.d(TAG, "New Game button clicked")', 'Log.d(TAG, "About button clicked")', and 'Log.d(TAG, "Exit button clicked")' are logged to track user interactions with the continue, new game, about, and exit buttons, respectively.
- When the options menu is created, another informational message is logged with 'Log.i(TAG, "Options menu created")'.
- Finally, menu item selections are tracked with 'Log.d(TAG, "Settings menu item selected")' for the settings item, and a warning message 'Log.w(TAG, "Unhandled menu item selected: " + id)' is logged for any unhandled menu items.
- The 'TAG' is a constant string used in log messages to identify the source of the log message. It is particularly useful in filtering the logs based on the tag name. In the example code, 'TAG' is set to '"SudokuApp"'. This allows developers to filter and view only the log messages related to the Sudoku app in Logcat or when using 'adb logcat'. This filtering capability makes it easier to track and diagnose issues specific to the application without being overwhelmed by logs from other parts of the system.

Viewing Log Messages

Using Logcat in Android Studio:

1. **Open Logcat:** In Android Studio, open the Logcat window by selecting 'View > Tool Windows > Logcat'.
2. **Filter by Tag:** Use the filter option to search for specific tags or severity levels. We can filter by the TAG used in log messages, such as "SudokuApp".
3. **Run the App:** Run the application to see the log messages in real-time.



6.10.2 Debugging with Debugger

Debugging with a debugger is an essential approach for identifying and resolving issues in Android applications. A debugger allows developers to pause program execution, inspect variables, step through code line-by-line, and monitor the application's state to understand the root cause of any issues. This method is particularly useful for diagnosing complex problems that are not easily traceable with log messages.

Importance of Debugging with a Debugger

- **Real-time Analysis:** Provides the ability to inspect the current state of an application, including variable values, memory usage, and thread states.

- **Step-by-Step Execution:** Allows developers to execute code line-by-line, making it easier to identify the exact point where an error occurs.
- **Conditional Breakpoints:** Enables setting conditions for breakpoints to pause execution only when certain conditions are met to reduce the noise of unnecessary breaks.
- **Watch Variables:** Helps in tracking the value changes of specific variables during execution.
- **Call Stack Inspection:** Provides insight into the sequence of method calls leading to a particular point in the code. It helps in understanding the flow and dependencies.

Setting Up the Debugger in Android Studio

1. **Enable Debuggable Mode:**

   Ensure the application is in debuggable mode. This can be done by setting the 'android:debuggable' attribute to 'true' in the 'AndroidManifest.xml' file during development.

Code AndroidManifest.xml

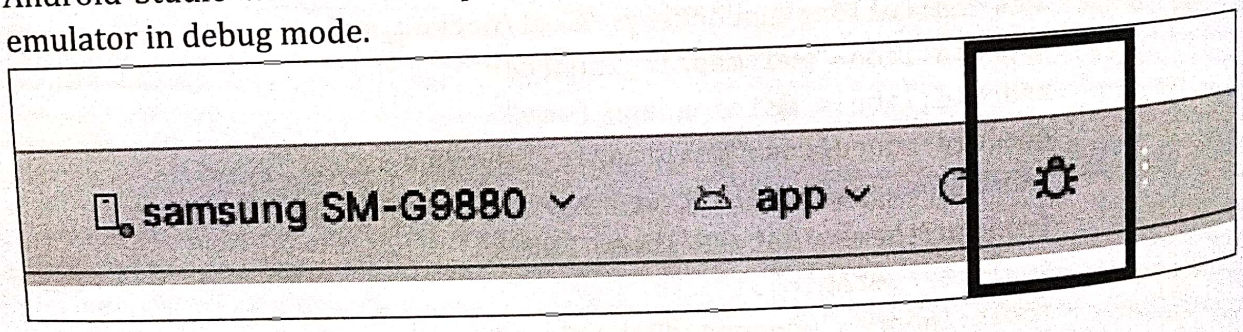
```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    <application
        android:debuggable="true"
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.Sudoku"
        tools:targetApi="31"
        tools:ignore="HardcodedDebugMode">
        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".About"
            android:label="@string/about_title"
            android:theme="@android:style/Theme.Dialog">
```

```
        </activity>
        <activity android:name=".Prefs"
            android:label="@string/settings_label" />

    </application>

</manifest>
```

2. **Set Breakpoints:**
   - Open the Java file in Android Studio.
   - Click in the left margin next to the line number where we want to set a breakpoint. A red dot will appear, indicating the breakpoint.

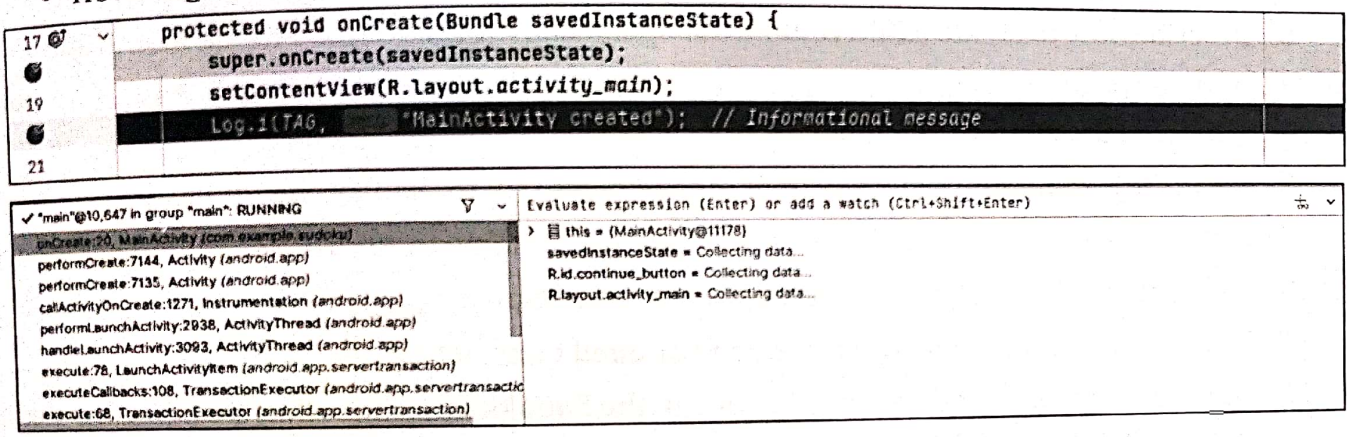
```
16      @Override
17      protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_main);
19          Log.i(TAG,  msg: "MainActivity created");  // Informational message
21
```

3. **Attach the Debugger:**
   - Click on the 'Debug' button (green bug icon) in the toolbar or right-click the project and select 'Debug 'app''.
   - Android Studio will build the project and deploy the app to the connected device or emulator in debug mode.



4. **Debugging Controls:**
   - **Resume Program (F9):** Continues execution until the next breakpoint or the end of the program.
   - **Pause Program:** Pauses the execution at the current line.
   - **Step Over (F8):** Executes the current line and moves to the next line.
   - **Step Into (F7):** If the current line contains a method call, steps into that method to debug its code.
   - **Step Out (Shift+F8):** Steps out of the current method and returns to the caller method.
   - **Evaluate Expression (Alt+F8):** Allows evaluation of expressions and variables at the current execution point.

5. **Inspect Variables:**
   - When the program is paused at a breakpoint, the 'Variables' pane in Android Studio shows the current values of all variables in the current scope.
   - Hovering over a variable in the editor window also shows its current value.



6. **Call Stack:**

   The 'Call Stack' pane shows the sequence of method calls that led to the current execution point. This helps in understanding the flow of execution and dependencies.

7. **Watch Expressions:**

   Add specific variables or expressions to the 'Watches' pane to monitor their values throughout the execution of the program.

6.11 Review Questions

Section - A

1. What is procedural design in the context of UI design?
2. What is declarative design, and how is it typically implemented in Android development?
3. Why does Google recommend using declarative XML for designing Android user interfaces?
4. Describe the advantages of declarative design over procedural design.
5. What does "designing by declaration" refer to in Android development?
6. What is the purpose of an About Box in an app?
7. What are the two kinds of menus supported in Android?
8. What are the different approaches of debugging an Android App?
9. What is an Importance of Debugging with a Debugger?

Section - B

1. Explain the key characteristics and components of an opening screen for an app.
2. What are the UI requirements for the Sudoku opening screen?