Mini project-Language Learning Tool

Batch Details:

Muralisekar Janissha -3122235001058

Mariya Joevita.V-3122235001077

Mokitha.S- 3122235001083

a) Problem Statement

Develop a language learning tool that provides a structured, level-based approach to language acquisition. Each level offers four distinct methods of learning, designed to enhance vocabulary, comprehension, grammar, and conversational skills. The tool should facilitate progression through levels, encouraging consistent practice and mastery of each learning mode for effective language acquisition.

b) Motivation for the problem statement

The motivation for this language learning tool is to create an accessible, engaging, and adaptive way to master a new language. Traditional methods often lack variety and fail to maintain learners' interest, so this tool offers a level-based, interactive approach. Each level includes four unique methods to reinforce vocabulary, comprehension, grammar, and conversation, making learning enjoyable and effective. Additionally, this approach allows learners to progress at their own pace, building confidence and enabling consistent practice in daily life.

c) Scope and Limitations:

Scope:

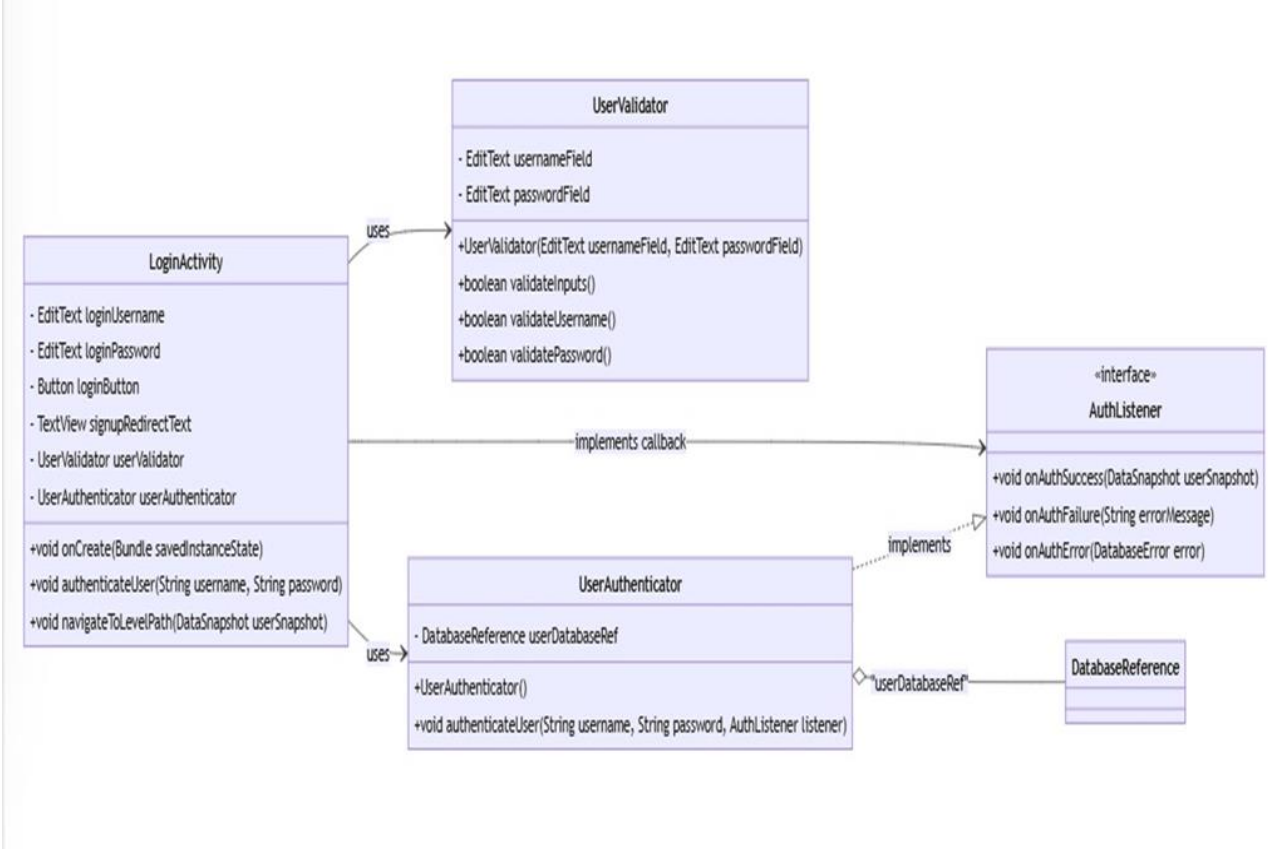
- Provides a structured, multi-level framework to enhance language skills progressively.
- Each level includes four distinct learning methods to improve vocabulary, grammar, comprehension, and conversational skills.
- Adaptable for various languages, with additional levels, learning styles, and exercises.
- Supports self-paced learning and is accessible across mobile platforms.

Limitations:

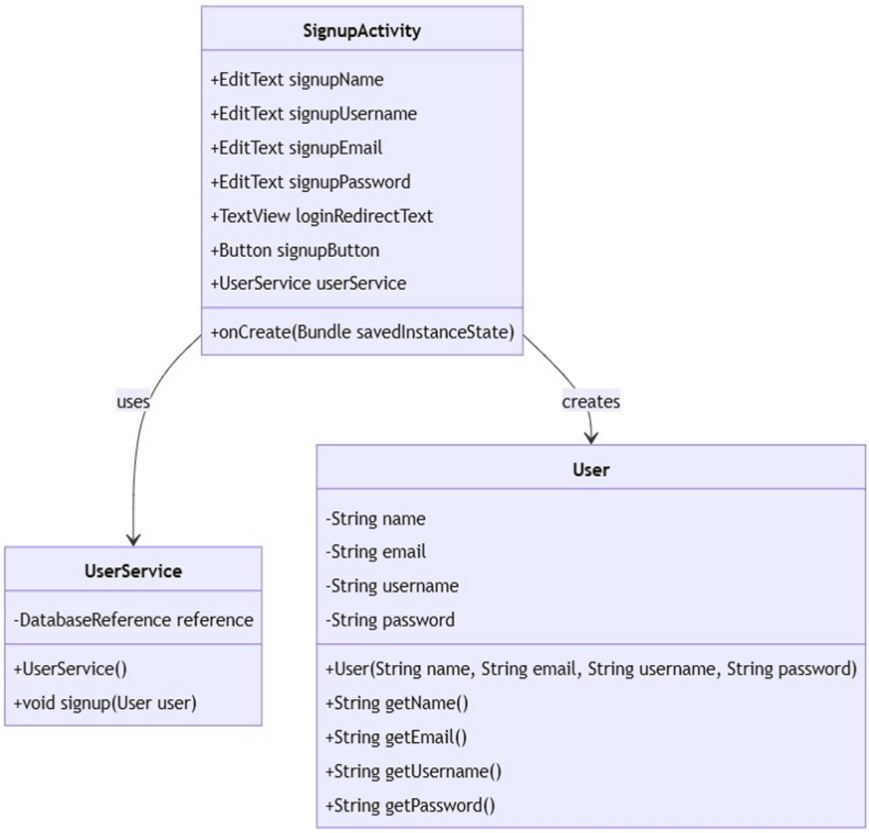
- Requires frequent updates to maintain engagement, needing dedicated resources.
- May be less effective for users who prefer in-person practice or immersive experiences.
- Covers foundational skills, but advanced learners might need supplementary resources for complex language nuances.
- No real time notifications to alert the users about their current progress or reminders.

d) Class Diagrams:

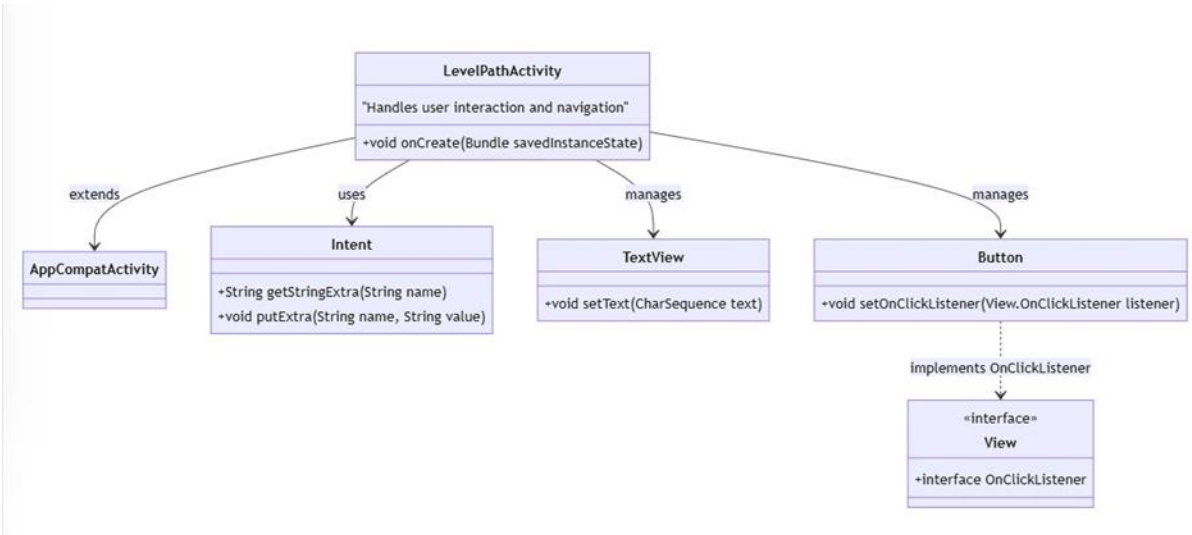
1. User Validation (Sign-up/Log in):



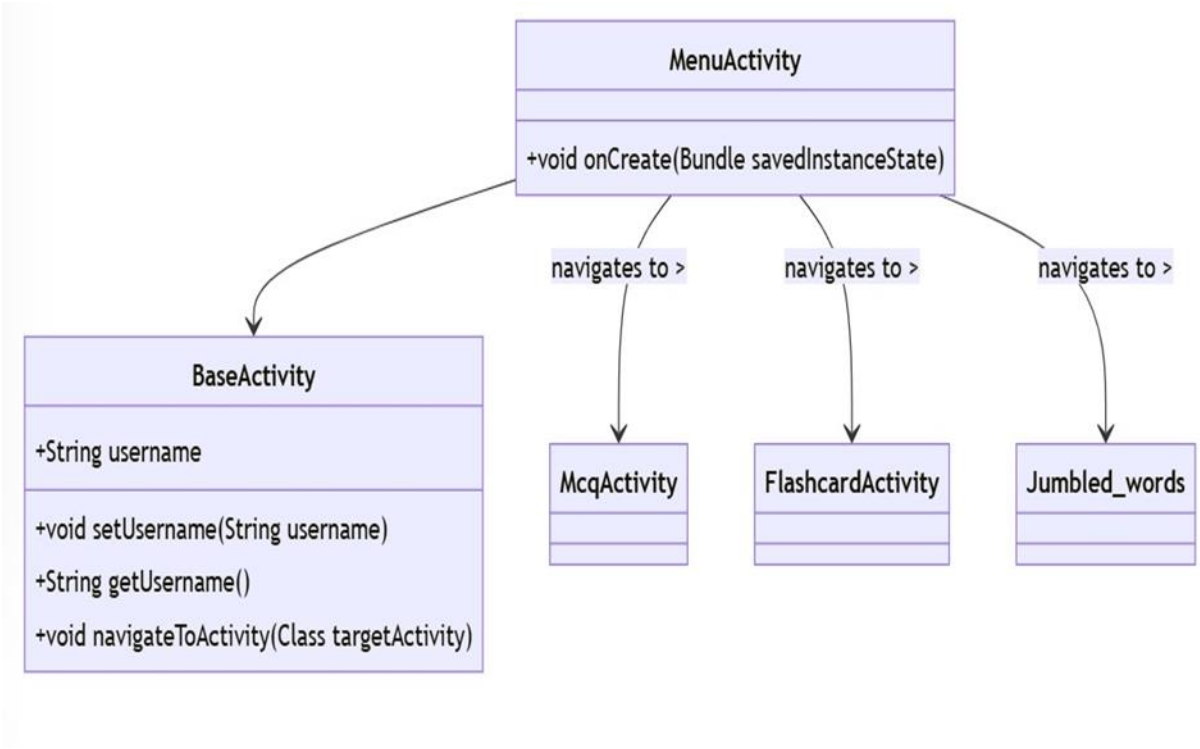
1.1. Sign-up Activity:



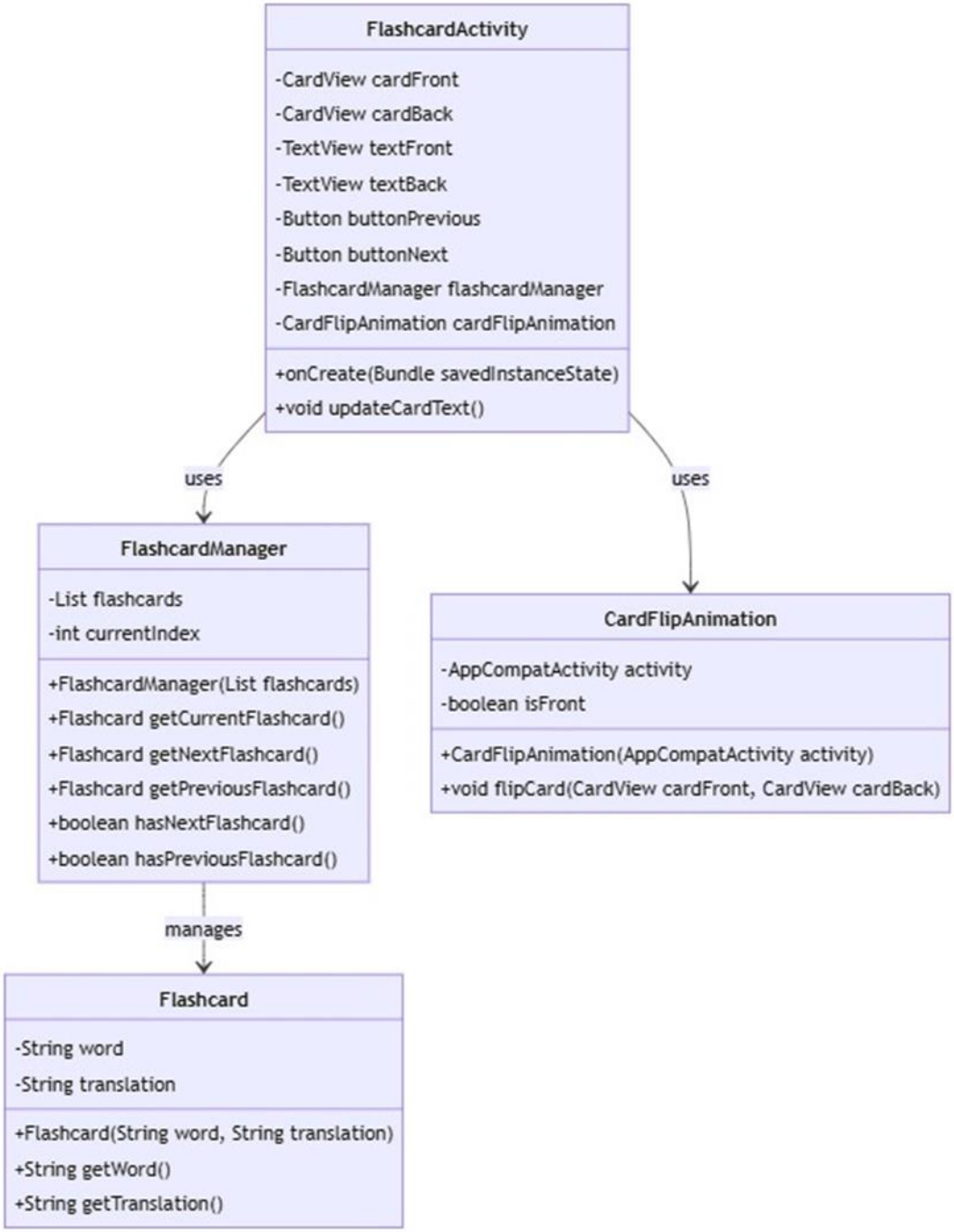
2. Level Activity:



3. Menu Activity:

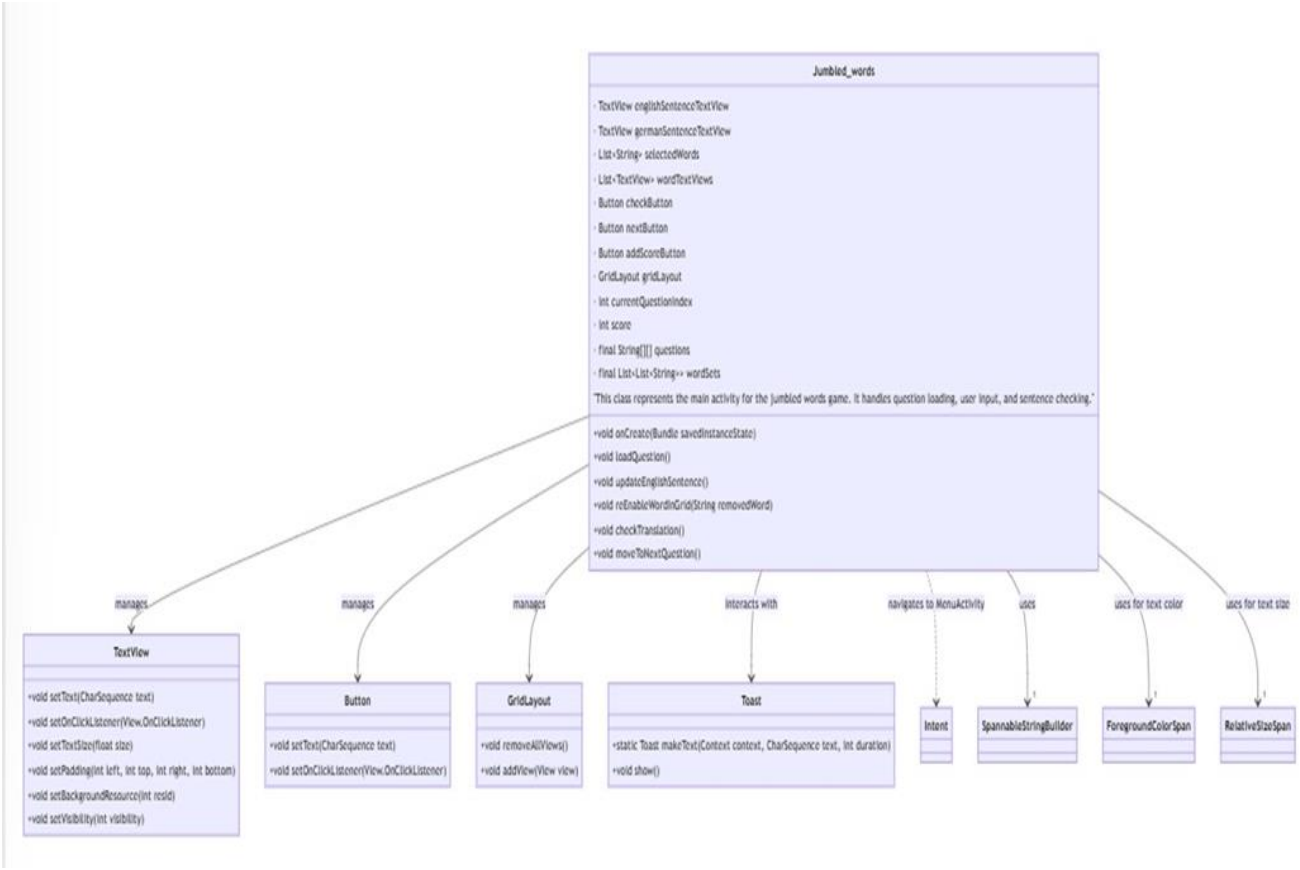


4. Flashcard Activity:



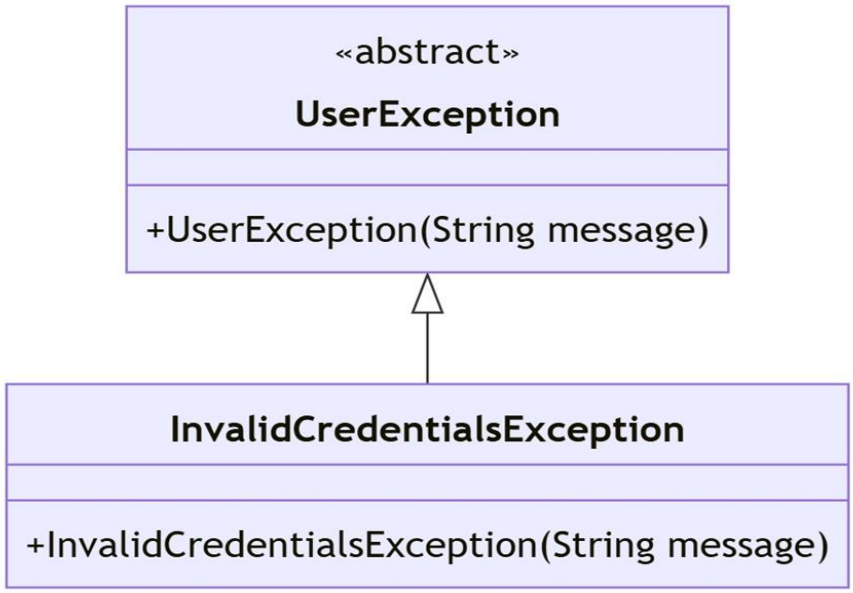
5. MCQ Activity:

6. Jumbled Words Activity:

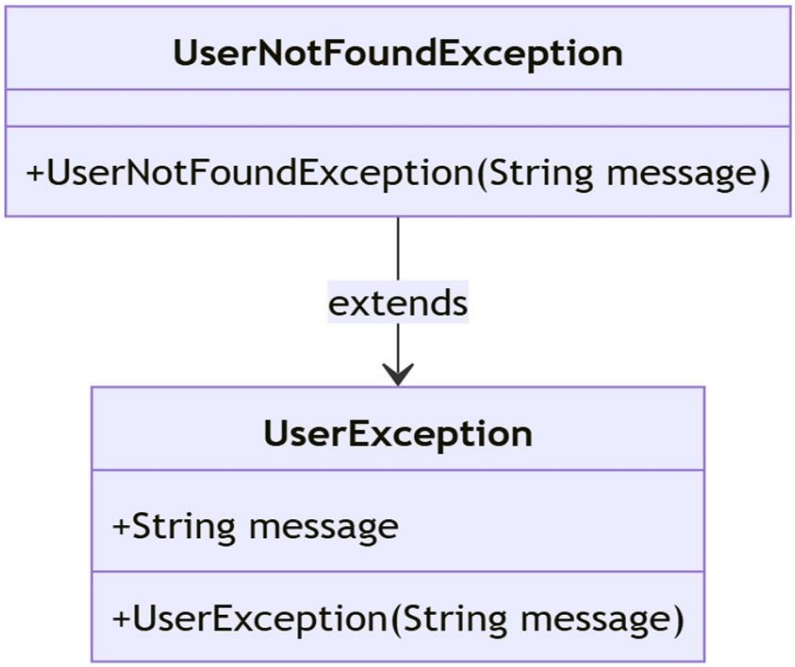


7.Exceptions:

7.1. Invalid Credentials Exception:



7.2. User Not Found Exception:



e) Modules Split Up:

1. Sign Up / Log In Module

- **Description:** This module manages user authentication. Users can create an account by signing up with their email, phone number, or social media accounts and log in securely to access their progress and personalized learning data.
- **Purpose:** It allows for user-specific data storage, such as vocabulary progress, scores, and activity history.

2. Level Modules

- **Description:** This section organizes the vocabulary learning process into different levels of difficulty or themes. Levels could range from beginner to advanced.
- **Purpose:** Levels help guide learners through a structured progression, providing an appropriate learning curve as they advance in language proficiency.

3.Menu Activity:

- **Description:** This is the main menu screen displayed after users log in. It lists the available activities in a visually organized format with icons or buttons for each module (e.g., Flashcards, Jumbled Words, MCQ, etc.). Tapping an option takes the user directly to that specific activity.
- **Purpose:** The Menu Activity provides quick, intuitive access to all the app's learning features. It acts as a user-friendly interface to guide learners in choosing different exercises based on their preferences and needs.

4. Flashcard Activity

- **Description:** Flashcards display vocabulary words in the source language on one side, with their translations on the other side. Users can flip the card to reveal the answer and move to the next card.
- **Purpose:** This activity uses repetition to aid vocabulary retention, allowing users to build and strengthen their vocabulary through a familiar and effective learning format.

5. Jumbled Sentence Module

- **Description:** In this module, users are given sentences with words in random order. They must rearrange these words to form grammatically correct sentences in the target language.
- **Purpose:** This activity focuses on sentence structure and syntax, helping users to understand the correct order of words and improve their language comprehension and construction skills.

6. MCQ (Multiple Choice Questions) Activity

- **Description:** This section presents questions with multiple answer options. Users must select the correct option, and upon answering, the system verifies their choice and calculates a score.
- **Purpose:** MCQs provide a quick and interactive way to assess the user's understanding of vocabulary and grammar concepts, reinforcing learning and allowing users to track their progress.

f) Implementation Specifics

FlashcardActivity.java

The application is designed to create a flashcard experience where users can view a word and its translation, flipping between the two sides. It incorporates animation for a more engaging user experience and utilizes Firebase Firestore for potential data storage.

Components

1. **Flashcard Class**:

o **Purpose**: Represents an individual flashcard that contains a word and its translation.

o **Attributes**:

- word: The text in the source language.
- translation: The corresponding translation in the target language.

o **Methods**: Includes getters for retrieving the word and translation.

2. **FlashcardManager Class**:

- **Purpose**: Manages a collection of flashcards and tracks the current index.
- **Attributes**:
  - flashcards: A list containing multiple Flashcard objects.
  - currentIndex: An integer tracking the currently displayed flashcard.
- **Methods**:
  - Retrieve the current flashcard.
  - Navigate to the next or previous flashcard while checking if more flashcards are available.

3. **CardFlipAnimation Class**:

- **Purpose**: Handles the animation effects when flipping the flashcards.
- **Attributes**:
  - activity: The context of the activity where the animation is applied.
  - isFront: A boolean to determine which side of the card is currently visible.
- **Methods**:
  - Manages the flipping animation, triggering animations for both the front and back sides of the card, and updates visibility accordingly.

4. **FlashcardActivity Class**:

- **Purpose**: The main activity that ties together UI components and functionality.
- **Attributes**:
  - UI components such as CardView for front and back cards, TextView for displaying text, and Button for navigation.
  - Instances of FlashcardManager and CardFlipAnimation to manage flashcard logic and animations.
- **Lifecycle Method**:
  - **onCreate()**: Initializes the user interface, sets up flashcards, binds UI elements, and implements button click listeners for navigation and flipping the card.

o **Methods**:

- Updates the text displayed on the front and back of the flashcards based on the current index in the FlashcardManager.

User Interaction

- **Navigation**: Users can navigate through the flashcards using "Previous" and "Next" buttons, with feedback given via toast messages when no more cards are available in either direction.
- **Flipping Cards**: Tapping on the card triggers an animation that flips it to reveal the translation, enhancing engagement with a dynamic visual effect.

Animation

- The flipping animation is handled using AnimatorSet, which allows for coordinated animation sequences.
- Animations for flipping out the front card and flipping in the back card (and vice versa) are defined in separate animator resource files.

Data Initialization

- The flashcards integrated with Firebase Firestore for dynamic data retrieval.

Layout

- The application layout includes:
  - Two card views: one for the front (word) and one for the back (translation).
  - Text views to display the word and translation.
  - Navigation buttons to move between cards.
  - A main card layout to capture click events for flipping the cards.

Jumbled_words.java

The Jumbled_words class implements a jumbled words game where users are presented with a French sentence and must arrange its English translation by selecting words. The app tracks user scores and allows navigation to a main menu.

Components

1. **Class Declaration and Inheritance**:

o The Jumbled_words class extends AppCompatActivity, allowing it to function as an activity within the Android application.

2. **Member Variables**:

o **TextView Elements**:

- englishSentenceTextView: Displays the user-formed English translation.
- frenchSentenceTextView: Shows the given French sentence

o **List Variables**:

- selectedWords: A list to store the words selected by the user.
- wordTextViews: A list to keep references to the TextViews displaying individual words.

o **Button Elements**:

- checkButton: A button for users to check if the formed sentence is correct.
- nextButton: A button to proceed to the next question.
- addScoreButton: Displays the user's current score.

o **GridLayout**:

- gridLayout: A layout that organizes the selectable words in a grid format.

3. **Game State Variables**:

o currentQuestionIndex: An integer tracking the index of the current question.

- score: An integer to keep track of the user's score.

4. **Questions and Words**:

o **Questions Array**: A 2D array of strings containing pairs of sentences: the French sentence and its corresponding English translation.

o **Word Sets**: A list of lists where each inner list includes the words from the corresponding English translation of the French sentence, which will be shuffled for user interaction.

Activity Lifecycle

- **onCreate(Bundle savedInstanceState)**:
  - Initializes the activity, sets the content view, and binds UI components.
  - Sets up the selectedWords and wordTextViews lists.
  - Loads the first question and initializes the score display.

- Configures click listeners for the navigation and action buttons.

Methods

1. **loadQuestion()**:

o Clears previous selections and resets the grid layout.

o Displays the current French sentence.

o Shuffles the words of the corresponding English translation and dynamically creates TextView elements for each word.

- Sets up click listeners on these TextView elements to allow word selection.

2. **updateEnglishSentence()**:

o Updates the englishSentenceTextView with the selected words formatted with a larger size and a specific color.

- Sets a click listener on the displayed English sentence to allow users to remove the last selected word.

3. **reEnableWordInGrid(String removedWord)**:

o Iterates through the wordTextViews to find the TextView corresponding to the removed word and makes it visible again.

4. **checkTranslation()**:

o Compares the user-formed sentence with the correct English translation.

- Displays a toast message indicating whether the translation was correct and updates the score accordingly.

5. **moveToNextQuestion()**:

o Checks if there are more questions to load. If so, it increments the currentQuestionIndex, loads the next question, and hides the nextButton if it's the last question.

User Interaction

- Users interact with the app by clicking on words displayed in the grid. Each selection adds the word to the formed English sentence. Clicking the sentence allows them to remove the last selected word.
- Feedback is provided via toast messages for correct or incorrect translations.

Navigation

- A button to navigate back to the main menu (MenuActivity) is included, using an Intent to start the new activity.

Layout Design

- The layout is defined in XML and includes:
    - Two TextViews for displaying the French and English sentences.
    - A GridLayout for organizing the words dynamically.
    - Buttons for checking translations, moving to the next question, and displaying the score.

LevelPathActivity.java

The LevelPathActivity class serves as a screen in the Android application where the user's username is displayed. It allows users to navigate back to a main menu, facilitating a seamless user experience.

Components

1. **Class Structure**:

o The class is a part of the Android activity lifecycle, extending from AppCompatActivity, which is a base class for activities that use the support library action bar features.

2. **Activity Lifecycle**:

o The onCreate method is invoked when the activity is first created. This is where the layout is set, and UI elements are initialized.

Implementation Details

1. **User Interface Setup**:

o The activity hides the action bar for a full-screen view, providing an immersive experience.

- The layout is set to a specific XML file, which contains the visual elements of the activity.

2. **Data Retrieval**:

o The activity retrieves the username from the intent that started it. This allows for personalized interactions based on the user's identity.

3. **UI Elements**:

o **TextView**: A TextView displays the username, allowing users to see their identity on the screen.

- **Button**: A button is provided for navigation. It is styled to be circular and acts as the primary method for user interaction.

4. **User Interaction**:

o When the user clicks the navigation button, an intent is created to move back to the main menu activity. This intent also passes the username, ensuring that the menu can continue to provide a personalized experience.

User Experience Considerations

- The primary interaction is centered around the button, which is designed to facilitate quick navigation.
- Feedback mechanisms, such as displaying the username prominently, help users feel more connected to the app.
- The activity maintains a clean and straightforward layout, making it easy for users to understand and navigate.

LoginActivity.java

The LoginActivity class is responsible for managing user login functionality within the app. It validates user inputs, authenticates against a Firebase database, and handles navigation to the next screen based on authentication results.

Components

1. **Class Structure**:

o LoginActivity extends AppCompatActivity, making it an activity that can utilize support library features.

2. **UI Elements**:

o **EditText Fields**: Two fields are provided for user input: one for the username and one for the password.

o **Button**: A button is designated for submitting the login credentials.

- **TextView**: A text view that redirects users to the signup activity for creating a new account.

3. **Validators**:

o **UserValidator Class**: This internal class is responsible for validating user inputs. It checks whether the username and password fields are filled. If empty, it sets an error message on the respective field.

4. **Authentication**:

o **UserAuthenticator Class**: This internal class handles the Firebase authentication process. It interacts with the Firebase Realtime Database to verify the provided credentials against stored user data.

Implementation Details

1. **Activity Lifecycle**:

o The onCreate method is overridden to perform initialization tasks when the activity is created. This includes hiding the action bar for a clean interface and setting the content view to the corresponding XML layout.

2. **UI Initialization**:

o UI elements (EditText, Button, TextView) are linked to their respective views defined in the layout XML file.

3. **Input Validation**:

o Upon clicking the login button, the app first validates the inputs using the UserValidator class. If both fields are filled, it retrieves the values and proceeds to authentication.

4. **User Authentication**:

o The UserAuthenticator class is called to check if the username exists in the Firebase database. The app listens for a response from Firebase:

- If the username exists, it checks the password against the stored value.
- Appropriate callback methods (onAuthSuccess, onAuthFailure, onAuthError) are implemented to handle the results of the authentication process.

5. **Navigation**:

o On successful authentication, user data (name, email, and username) is retrieved from the database. An Intent is created to navigate to the LevelPathActivity, passing the retrieved data as extras for personalized user experience.

6. **Error Handling**:

o In case of authentication failure, the app provides feedback by highlighting the problematic field (username or password) and setting an error message accordingly.

o There is an optional error handling method for database errors, which could include logging or displaying an error message.

McqActivity.java

The McqActivity is an Android activity that presents multiple-choice questions to the user. It fetches questions from a Firestore database, displays them, allows the user to select answers, checks those answers, and tracks the user's score.

Key Components

1. **Question Class**:

o Represents a single question, including its text, options, and the correct answer.

- Provides methods to retrieve question details.

2. **McqActivity Class**:

o Inherits from AppCompatActivity, representing the main activity for the multiple-choice quiz.

- Contains methods for UI setup, data retrieval, question management, and user interaction.

3. **Firebase Firestore**:

o Used to retrieve questions and answers stored in the Firestore database.

Flow and Functionality

1. **Activity Initialization (onCreate Method)**:

o The activity sets up the user interface by hiding the action bar and inflating the layout from mcq_activity.xml.

o Initializes UI elements such as TextView, RadioButton, Button, and RadioGroup.

- Establishes a connection to Firestore and calls the method to load questions.

2. **UI Elements Initialization (initializeUIElements Method)**:

o Binds UI components to their respective IDs in the layout.

- Sets click listeners for navigation and answer checking buttons.

3. **Loading Questions from Firestore (loadQuestionsFromFirestore Method)**:

o Queries the Firestore collection named "Mcq".

o On successful retrieval, iterates through each document, extracts question details, and populates the questionList.

- Calls displayQuestion() to show the first question.

4. **Displaying Questions (displayQuestion Method)**:

o Displays the current question and its options based on the currentQuestionIndex.

- Resets the colors of the answer options and clears any selected answer.

5. **Answer Checking (checkAnswer Method)**:

o Checks if the user has selected an answer.

o Compares the selected answer with the correct answer for the current question.

o Updates the score and changes the text color of the selected answer based on correctness.

- Calls highlightCorrectAnswer() to indicate the correct answer.

6. **Highlighting the Correct Answer (highlightCorrectAnswer Method)**:

o Changes the text color of the correct answer to green after an answer is checked, allowing the user to see which answer was correct.

7. **Navigating Through Questions**:

o **Next Question (loadNextQuestion Method)**:

- Increments the currentQuestionIndex and displays the next question if not at the end of the list.

o **Previous Question (loadPreviousQuestion Method)**:

- Decrements the currentQuestionIndex and displays the previous question if not at the beginning of the list.
- Displays a toast message if the user tries to navigate beyond available questions.

8. **Score Tracking**:

o The score is updated and displayed in the scoreButton when the user answers correctly.

9. **User Feedback**:

o Toast messages are used to inform the user about actions such as reaching the end of questions, not selecting an answer, or providing incorrect answers.

User Interface

- **Layout**: The activity uses a layout defined in mcq_activity.xml, which contains:
    - A TextView for the question text.
    - Four RadioButton options for answering.
    - Buttons for navigating questions and submitting answers.

- A score display button.

Error Handling

- The code includes basic error handling for:
  - Failed retrieval of questions from Firestore.
  - Empty selections for answers.
  - Attempting to navigate beyond available questions.

MenuActivity.java

MenuActivity serves as a menu interface for the user, allowing navigation to different game or quiz activities in the application, such as multiple-choice questions (MCQs), flashcards, and a jumbled words game. It extends a BaseActivity that encapsulates common functionality, particularly for handling the username.

Key Components

1. **BaseActivity Class**:

o An abstract class extending AppCompatActivity.

o Maintains a username variable and provides methods to set/get it.

- Includes a navigateToActivity method for transitioning to other activities, passing the username via an Intent.

2. **MenuActivity Class**:

o Inherits from BaseActivity, gaining access to the username management and activity navigation functionalities.

o Sets up the user interface, retrieves the username, and assigns actions to buttons for navigation.

Flow and Functionality

1. **Activity Initialization (onCreate Method)**:

o The activity begins by calling super.onCreate() to initialize the base activity.

o The action bar is hidden using getSupportActionBar().hide().

- The layout is set to activity_menu, which contains the UI components.

2. **Username Retrieval**:

o The username is extracted from the incoming Intent using getIntent().getStringExtra("username").

o The username is displayed in a TextView to inform the user of their logged-in status or identity.

3. **Button Initialization and Click Listeners**:

o Three buttons are initialized by finding their respective views using findViewById():

- levelButton1 for navigating to the MCQ activity (McqActivity).
- flash_button for navigating to the flashcard activity (FlashcardActivity).
- levelButton3 for navigating to the jumbled words activity (Jumbled_words).

o Each button has a click listener set up using lambda expressions that call the navigateToActivity method to transition to the specified activity while passing the username.

User Interface

- The activity utilizes a layout defined in activity_menu.xml, which is assumed to contain:
    - A TextView to display the username.
    - Buttons for navigating to different activities, each labeled accordingly.

Interaction and Navigation

- The navigateToActivity method in BaseActivity handles the navigation:
    - Creates an Intent to start the target activity.
    - Adds the username as an extra to the Intent.
    - Calls startActivity(intent) to initiate the transition.

Error Handling and User Experience

- **User Feedback**: The UI includes a username display, providing immediate feedback to the user regarding their identity.
- **Design Considerations**: The layout should be user-friendly and visually appealing, with clear labeling of buttons for easy navigation.

SignupActivity.java

User Class

- **Purpose**: Represents a user in the application, encapsulating their details such as name, email, username, and password.
- **Attributes**:
    - name: The full name of the user.
    - email: The email address of the user.
    - username: A unique identifier for the user.

- password: The user's password.

- **Methods**:
  - Constructor to initialize the user attributes.
  - Getter methods for each attribute to allow access without exposing the data directly.

2. UserService Class

- **Purpose**: Handles operations related to user management, particularly user registration and data storage.
- **Attributes**:
  - reference: A reference to the Firebase Realtime Database under the "users" node, allowing interaction with user data.
- **Methods**:
  - Constructor that initializes the DatabaseReference to the Firebase database.
  - signup(User user): Accepts a User object and stores its data in the Firebase database under the username as the key.

3. SignupActivity Class

- **Purpose**: Provides the user interface for new users to sign up for the application.
- **Key Components**:
  - **UI Elements**:
    - Four EditText fields for user input: name, email, username, and password.
    - A Button for submitting the sign-up form.
    - A TextView that acts as a link to redirect users to the login page.
  - **User Interaction**:
    - Listeners are set up on the sign-up button and the login redirect text to handle user actions.

User Flow and Functionality

1. **User Input**:

o When the SignupActivity is created, the user sees input fields for their name, email, username, and password.

- Each field allows users to enter their respective information.

2. **Sign-Up Process**:

o Upon clicking the sign-up button:

- The application retrieves the entered values from the EditText fields.
- A new User object is created with the provided information.
- The UserService instance is used to call the signup method, which stores the user's details in the Firebase database.

3. **Feedback**:

o After successful sign-up, a Toast message informs the user that they have signed up successfully.

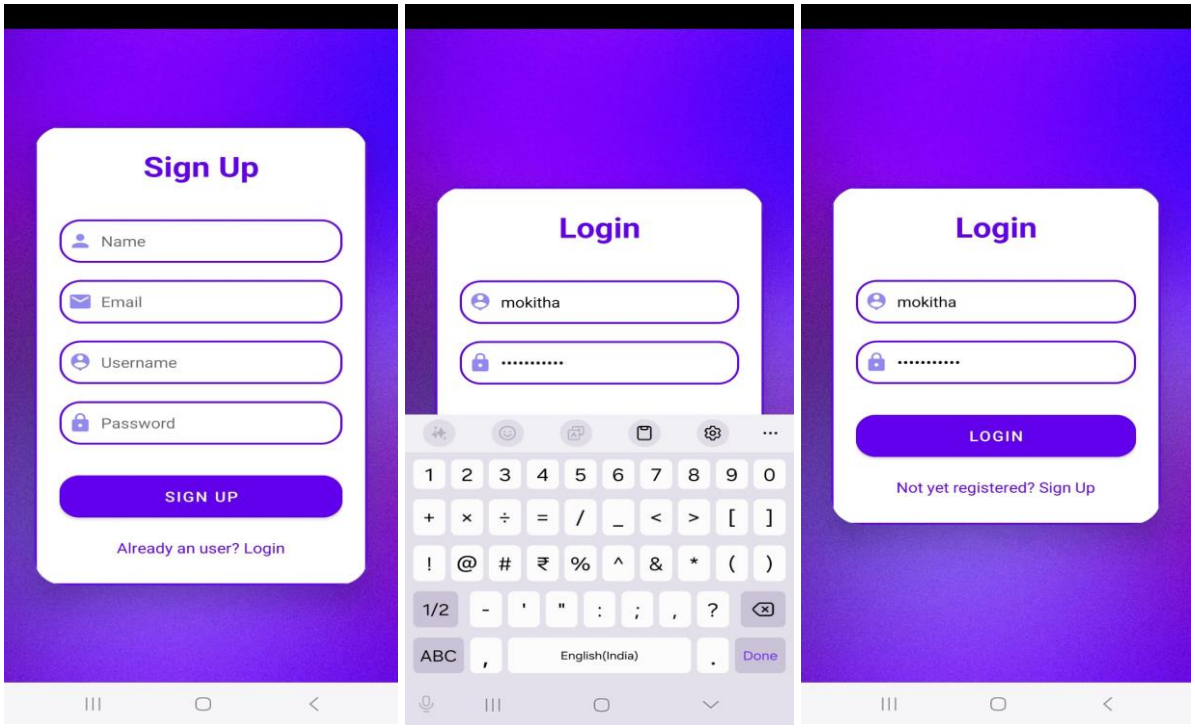
- The activity then transitions to the LoginActivity, allowing users to log in with their newly created account.

4. **Redirect to Login**:

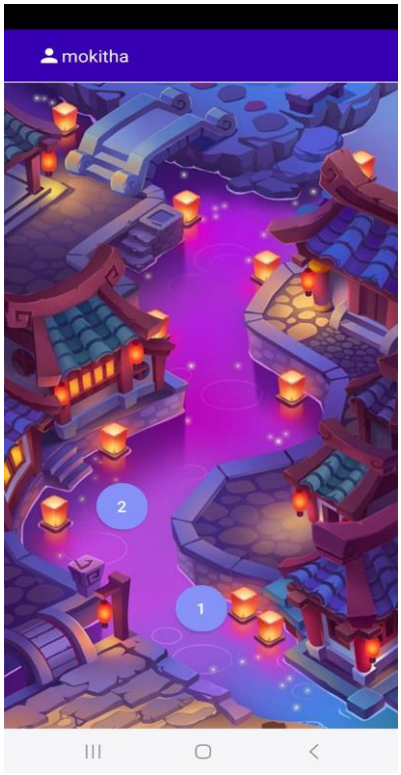
o If the user clicks on the redirect text (loginRedirectText), the application starts the LoginActivity without performing any sign-up operations.

g) Output Screenshots:

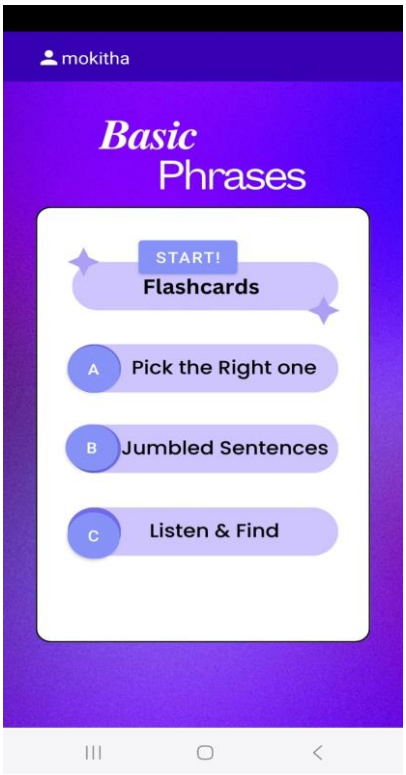
Signup and Login:



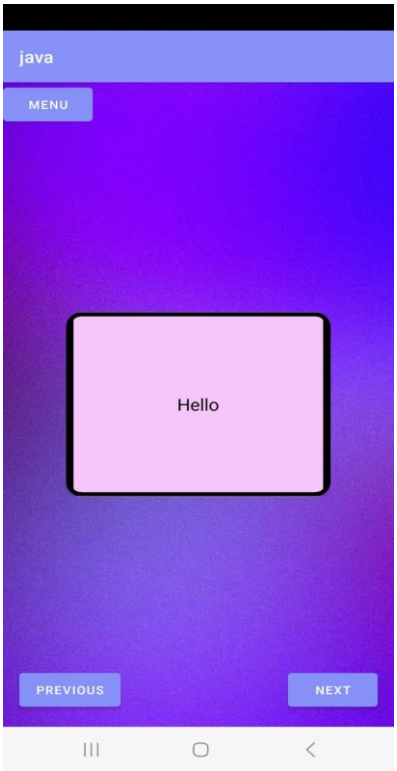
Level Path:



Menu



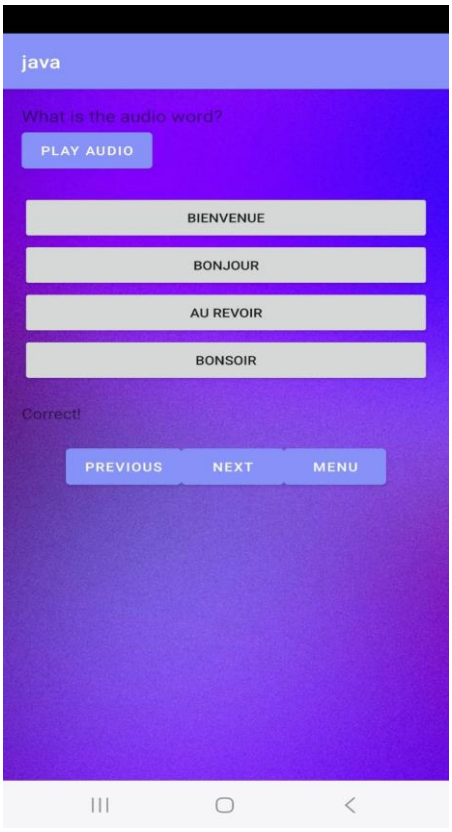
Flash cards



Pick the Right one

Jumbled Senteces

Listen and Find(Audio Question)



Badge

h) Object oriented features used:

Packages are used to store each class separately this is used to improve modularity thus achieving improved code reusability and efficiency.

Sign up Activity:

1. Encapsulation:

The `User` and `UserService` classes encapsulate user data and service logic, respectively. The `User` class contains private fields for user attributes like name, email, username, and password, along with getter methods, encapsulating the data within the class and controlling access to it. The `UserService` class encapsulates Firebase operations, keeping database interactions centralized and accessible only through the defined methods.

2. Abstraction:

The code abstracts complex details of user signup and Firebase database operations. By separating data handling into `UserService` and storing user data in the `User` class, this structure hides intricate Firebase logic from `SignupActivity`, making the signup process straightforward and maintainable.

3. Inheritance:

The `SignupActivity` class inherits from `AppCompatActivity`, benefiting from Android's activity lifecycle management and UI control without needing to implement basic activity functionality manually. This inheritance allows `SignupActivity` to function as an activity with minimal setup.

4. Polymorphism:

The `OnClickListener` interface enables polymorphism by allowing `signupButton` and `loginRedirectText` to handle click events in different ways. Each button implements the `onClick` method but performs distinct actions, demonstrating polymorphism through behavioral variation based on the component being clicked.

5. Exception Handling:

User-defined exceptions are included to ensure that the user trying to login is already in the system i.e stored in firebase.

Menu Activity:

1.Encapsulation:

The `username` field is encapsulated within `BaseActivity`, accessible only through `getUsername` and `setUsername` methods. This ensures that the `username` is controlled and managed through specific methods, maintaining data integrity within the class and restricting direct access to the variable itself.

2. Abstraction:

`BaseActivity` provides an abstraction for common activity functionality, such as handling the `username` and the `navigateToActivity` method. This hides the details of starting a new activity with the username from `MenuActivity`, allowing `MenuActivity` to focus only on setting up buttons and interactions.

3. Inheritance:

`MenuActivity` inherits from `BaseActivity`, enabling it to reuse the `username` handling and `navigateToActivity` logic. This simplifies `MenuActivity` by allowing it to inherit these properties and methods rather than reimplementing them. Additionally, this inheritance structure makes it easy to apply these common methods to other activities that could extend `BaseActivity`.

4. Polymorphism:

Polymorphism is present in the `navigateToActivity` method. Although it's defined in `BaseActivity`, it can be called in any subclass (e.g., `MenuActivity`) with different target activities (`McqActivity`, `FlashcardActivity`, `Jumbled_words`), which means it dynamically adapts to different target classes based on the specific button clicked.

Level Path Activity:

1. Encapsulation:

The `User` class encapsulates user-related data (name, email, username, password) and provides methods to access this data. This prevents direct access to the internal state of the `User` object, promoting data hiding.

2. Abstraction:

The `UserService` class abstracts the operations related to user management, such as signing up a user. Users of this class do not need to know the underlying implementation details of how users are stored in the Firebase database.

3. Inheritance:

The `MenuActivity` class inherits from `BaseActivity`, allowing it to reuse common functionality (such as username handling and activity navigation). This promotes code reuse and establishes a hierarchical relationship between classes.

4. Polymorphism:

The use of method overriding (e.g., `onCreate` method in both `MenuActivity` and `LevelPathActivity`) allows subclasses to provide specific implementations of methods defined in the base class. Additionally, the `navigateToActivity` method uses the `Class<?>` type, allowing it to accept any class type that extends `AppCompatActivity`.

Login Activity:

1. Encapsulation:

UserValidator and UserAuthenticator Classes:

These classes encapsulate their respective functionalities. The `UserValidator` class manages input validation for the login fields, while the `UserAuthenticator` handles authentication logic, including querying the Firebase database. Both classes keep their internal workings hidden from the `LoginActivity`, allowing for cleaner code and easier maintenance.

Private Fields:

Fields like `loginUsername`, `loginPassword`, and `userAuthenticator` are kept private to restrict direct access from outside the class, promoting data hiding.

2. Abstraction:

Abstract Interfaces:

The `AuthListener` interface in `UserAuthenticator` abstracts the authentication process's callback mechanism. This allows the `LoginActivity` to respond to authentication events without needing to understand the underlying details of how authentication is performed.

Separation of Concerns:

Each class has a distinct responsibility, with `UserValidator` focusing on validation and `UserAuthenticator` on authentication. This separation simplifies understanding and maintaining the code.

3. Inheritance:

Class Hierarchy:

Although your provided classes do not directly show inheritance, you could potentially create a base class for activities (like a `BaseActivity`) that contains common functionality for all activities. This would illustrate inheritance in your application design.

4. Polymorphism:

Interface Implementation:

The `AuthListener` interface allows different implementations of authentication handling methods (`onAuthSuccess`, `onAuthFailure`, `onAuthError`). This means you can create different classes that implement this interface, providing varied behaviors for user authentication without changing the `UserAuthenticator` class.

Flash Card Activity:

1. Encapsulation

Encapsulation involves bundling data (attributes) and methods (functions) within classes and restricting direct access to some components.

The `Flashcard` class encapsulates the `word` and `translation` properties by keeping them private and provides public getter methods (`getWord()` and `getTranslation()`) to access this data. This hides the internal state of a flashcard from outside interference and ensures that only controlled access is provided.

Similarly, the `FlashcardManager` and `CardFlipAnimation` classes encapsulate the data and methods related to managing flashcards and animations, respectively, which keeps the code organized and manageable.

2. Abstraction

Abstraction involves creating classes that expose only necessary functionalities and hide complex implementation details.

The `FlashcardManager` class abstracts the logic for navigating through flashcards, providing methods like `getCurrentFlashcard()`, `getNextFlashcard()`, and `getPreviousFlashcard()`, so that the `FlashcardActivity` doesn't need to know the specific details of how flashcards are managed.

The `CardFlipAnimation` class abstracts the animation logic for flipping the cards. The `flipCard` method provides an easy-to-use interface to flip the card without requiring the activity to handle the animation details directly.

3. Inheritance (from Android SDK)

The `FlashcardActivity` class inherits from `AppCompatActivity`, which is part of the Android SDK. This demonstrates **inheritance**, as `FlashcardActivity` extends `AppCompatActivity`, allowing it to inherit methods and properties for Android activities and lifecycle management.

This inheritance allows `FlashcardActivity` to use methods like `onCreate` and access context-based methods (e.g., `findViewById`, `startActivity`) necessary for Android activities.

4. Polymorphism

Polymorphism isn't directly demonstrated in this code, but there's an implied polymorphic behavior in the use of `Animator.AnimatorListener`. The code assigns an anonymous `AnimatorListener` implementation to `setOut.addListener`, which allows different behaviors (overriding `onAnimationStart`, `onAnimationEnd`, etc.) during the animation events. This enables custom behavior each time an animation event is triggered.

Polymorphism is also seen in the Android SDK itself, where methods like `findViewById` work on various types of views (e.g., `Button`, `CardView`) due to polymorphic behavior in the view hierarchy.

MCQ Activity:

1. Encapsulation:

The code utilizes encapsulation by bundling data and the methods that operate on that data within the `Question` class. This keeps internal details hidden from other parts of the program and restricts direct access, which enhances security and modularity.

2. Abstraction:

Abstraction is implemented by designing the `McqActivity` class to handle complex operations like loading questions, managing the UI, and verifying answers. By organizing functionality into distinct methods, the program simplifies interaction with the Firestore database, user input handling, and display updates, allowing users to focus on the essentials without delving into implementation specifics.

3. Inheritance:

The code leverages inheritance by having `McqActivity` extend the `AppCompatActivity` class, which enables access to Android-specific methods and properties for activity management. This promotes code reusability and integration with the Android framework, eliminating the need to create basic activity functions from scratch.

4. Polymorphism:

Polymorphism is exhibited through the way RadioButtons are treated interchangeably within the RadioGroup. The program can dynamically access, evaluate, and update different answer options at runtime, allowing for flexibility and extensibility in user input handling without modifying core logic.

Jumbled Sentence Activity:

1.Encapsulation:

The variables like `selectedWords`, `wordTextViews`, and `currentQuestionIndex` are encapsulated within the `Jumbled_words` class, along with methods like `loadQuestion()`, `updateEnglishSentence()`, and `checkTranslation()`, providing organized access and control over the class's data and functionality.

2. Abstraction:

Complex functionalities, such as loading questions, checking translations, updating the displayed sentence, and managing the grid layout of words, are abstracted into separate methods. This makes the code easier to understand and maintain by hiding the details of each function's inner workings.

3. Inheritance:

The `Jumbled_words` class extends `AppCompatActivity`, inheriting essential Android lifecycle behaviors and properties. This reuse of the base `AppCompatActivity` functionality allows `Jumbled_words` to operate as an Android activity with minimal additional code.

4. Polymorphism:

Polymorphism is utilized with the `View.OnClickListener` interface, allowing different behaviors to be assigned to UI elements like the `checkButton`, `nextButton`, and `englishSentenceTextView` click listeners. Each button's action is unique but uses the same `OnClickListener` interface, showcasing method behavior variations based on context.

Text To Speech Activity:

1. Encapsulation:

Encapsulation is achieved by using private fields in the `Word` class (`text`, `audioUrl`, and `options`). This restricts direct access to these fields, ensuring that they are only accessible through public getter methods (`getText()`, `getAudioUrl()`, and `getOptions()`).

The `Word` class encapsulates the properties and behavior of a quiz word item, making it self-contained and reusable.

2. Abstraction:

The `Word` class abstracts the concept of a "word" in the quiz context, which includes its text, audio, and answer options. This allows you to work with a single `Word` object without needing to manage each attribute separately.

The `QuizActivity` class uses abstracted methods like `fetchWordsFromFirebase()`, `loadNextQuestion()`, `displayOptions()`, and `checkAnswer()` to break down the functionality. This structure makes the code more modular and manageable by hiding complex implementation details.

3. Inheritance:

Inheritance is used in your code as `QuizActivity` extends `AppCompatActivity`, which is part of the Android framework. By inheriting from `AppCompatActivity`, `QuizActivity` gains all the functionality and lifecycle management provided by Android's activity class, allowing it to function within the Android environment and respond to events like `onCreate()`, `onDestroy()`, etc.

Additionally, `QuizActivity` implements the `TextToSpeech.OnInitListener` interface to respond to the `TextToSpeech` engine's initialization event.

4. Polymorphism:

Polymorphism is demonstrated in two key areas:

Interface Polymorphism:

`QuizActivity` implements the `TextToSpeech.OnInitListener` interface. This allows it to use the `onInit` method in a polymorphic way, responding to the initialization status of `TextToSpeech` based on the outcome.

Method Overriding:

The `onCreate()` and `onDestroy()` methods are overridden in `QuizActivity`. These methods are part of the lifecycle of an Android activity, but they're given specific functionality for this quiz app.

5) Collections:

ArrayList and hash map were used to store the words obtained from firebase.

i) Inference and future Extension:

Inference:

This project offered an invaluable experience in learning and applying Object-Oriented Programming (OOP) principles within a functional Android application. Initially, we were unfamiliar with Android Studio's interface, file organization, and tools, which felt complex. However, as we progressed, we became adept at navigating the environment and leveraging essential Android components. Working directly with Android Studio allowed us to implement OOP concepts like encapsulation, inheritance, and polymorphism within a mobile app context, enhancing our understanding of how these principles create modular, maintainable code. Additionally, integrating Firebase and Text-to-Speech (TTS) functionality, along with designing interactive UI elements, further developed our skills in Android development and user-centric design.

The quiz application we built supports language learning through a variety of activities designed to reinforce vocabulary and comprehension. The app retrieves words from Firebase and incorporates multiple-choice questions, flashcards, and jumbled sentence activities to create a diverse learning experience. Users can listen to words via TTS, engage in jumbled sentence challenges, and test vocabulary with flashcards and MCQs, each providing instant feedback. This combination of visual and auditory elements, along with random question ordering, keeps users engaged while strengthening their language skills. Overall, this project successfully created an interactive and multifaceted learning experience, igniting our interest in mobile app development.

Future Extensions:

1. Increasing Levels: Currently, the app has a basic structure for question levels. In future iterations, we aim to add multiple levels that gradually increase in difficulty. By introducing progressive learning stages, the app will cater to users with varying language proficiency, from beginners to advanced learners. This feature will offer a more customized learning journey and incentivize users to reach higher levels.

2. Expanding Modules: Alongside vocabulary quizzes, we plan to introduce additional modules, such as sentence translation, to diversify the app's learning approach. Each level could incorporate exercises beyond simple word identification, challenging users to translate full sentences or recognize proper grammar structures. This expansion would make the app a more comprehensive language-learning tool, fostering not only vocabulary growth but also a deeper understanding of sentence structure and practical usage.

3. Real-Time Notifications: To enhance user engagement, we intend to integrate real-time notifications that remind users to practice regularly. Push notifications could be used to encourage users to revisit the app, complete daily quizzes, or inform them about new content updates. By implementing this feature, the app will be more likely to become a part of the user's daily routine, promoting consistent language learning and improved retention.