BACHELOR OF COMPUTER APPLICATIONS

SEMESTER 6

DCA3201

MOBILE APPLICATION DEVELOPMENT

Unit 4

Multimedia in Android

Table of Contents

SL No	Topic		Fig No / Table / Graph	SAQ / Activity	Page No
1	Introduction		-	-	3 - 4
	1.1	Learning Objectives	-	-	
2	Audio and Video Playback		-	1	5 - 29
	2.1	MediaPlayer API	-	-	
	2.2	ExoPlayer for Advanced Playback:	-	-	
	2.3	Media Streaming	-	-	
3	Recording Audio and Video		-	2	30 - 51
	3.1	Audio Recording with MediaRecorder	-	-	
	3.2	Video Recording with Camera2 API:	-	-	
	3.3	Permissions and Best Practices	-	-	
4	Summary		-	-	52 - 53
5	Answers		-	-	54
6	Terminal Questions		-	-	55
7	Terminal Answers		-	-	55

1. INTRODUCTION

Multimedia plays an integral role in the modern mobile application development landscape, and when it comes to the Android platform, it becomes a pivotal component that enriches user experiences. Android, an open-source operating system developed by Google, boasts a vast user base across the globe, and multimedia elements are at the forefront of what makes Android applications engaging and interactive. This introduction explores the significance of multimedia in Android, shedding light on how it enhances user engagement and the key considerations in its integration.

In the realm of mobile applications, multimedia encompasses a wide range of elements, including images, audio, video, and animations. These elements are fundamental in crafting visually appealing and immersive user interfaces, fostering user engagement, and conveying information effectively. In Android, developers leverage the platform's rich set of tools, libraries, and APIs to harness the power of multimedia. Whether it's building entertainment apps with vibrant graphics and high-quality audio or educational apps featuring video tutorials, the Android ecosystem provides a versatile environment for multimedia development.

Multimedia in Android is not limited to passive content consumption. It extends to interactive applications, such as augmented reality (AR) and virtual reality (VR), which push the boundaries of user engagement. Additionally, with the rise of social media and user-generated content, multimedia plays a pivotal role in enabling users to create, edit, and share content seamlessly through Android apps. As we delve deeper into the world of multimedia in Android, we will explore the tools and techniques that developers employ to harness the potential of these elements, ensuring that mobile applications are not only functional but also visually captivating and user-centric.

1.1 Learning Objectives:

At the end of this topic, students should be able to:

- *Explain the purpose of the MediaPlayer class for audio and video playback.*
- *Describe how media files are loaded and played using MediaPlayer.*
- *Describe the process of recording audio and video using Android's built-in tools.*

2. AUDIO AND VIDEO PLAYBACK

In the world of mobile application development, the integration of audio and video playback features has become not only commonplace but also a crucial aspect of creating engaging and interactive mobile apps. As we increasingly rely on smartphones and tablets for our entertainment, communication, and information needs, the ability to seamlessly play audio and video content is a must for developers.

Audio Playback in Mobile Apps:

Audio playback in mobile applications encompasses a wide range of functionalities. It includes the ability to play music tracks, podcasts, audiobooks, and even sound effects within games. For instance, music streaming apps like Spotify or Apple Music allow users to stream and listen to their favorite songs on the go. In contrast, meditation and mindfulness apps provide users with soothing audio content to help with relaxation and stress management.

Developers leverage various audio formats, such as MP3, AAC, and FLAC, and APIs to control playback, adjust volume, and manage playlists. They can also tap into features like offline downloads for uninterrupted listening experiences, or they might incorporate equalizer settings to enhance the audio quality.

Video Playback in Mobile Apps:

Video playback is equally pivotal in mobile app development. From social media platforms like YouTube and TikTok to video streaming giants like Netflix and Amazon Prime Video, these apps deliver seamless video content to users. Moreover, education and e-learning applications use video playback to provide instructional materials, while video conferencing apps facilitate real-time communication through video.

Developers work with video codecs, such as H.264 and VP9, to ensure compatibility across a wide range of devices. They enable features like adaptive streaming, which adjusts video quality based on the user's internet connection, and implement controls for play, pause, seek, and full-screen mode.

Example:

Let's consider a fitness mobile application that combines both audio and video playback. In this app, users can access a library of workout routines. When they select a workout, the app plays a video that demonstrates the exercises, accompanied by audio instructions for timing and guidance. Users can pause, rewind, or skip through the video, adjusting the volume as needed. Additionally, they can listen to their favorite workout playlists during the session.

2.1 MediaPlayer API

The MediaPlayer API is a critical component in mobile application development, particularly in the context of Android development, where it's widely used for audio and video playback. This API offers several characteristics and properties that make it essential for building multimedia-rich mobile applications.

Characteristics of the MediaPlayer API:

- ***Versatility:*** The MediaPlayer API supports a wide range of media formats, making it versatile for playing audio and video content in various formats.
- ***Simple API:*** The API offers a relatively straightforward and easy-to-use interface for controlling media playback, making it accessible to developers of all skill levels.
- ***Playback Control:*** It provides methods for controlling media playback, including play, pause, stop, seek, volume control, and looping.
- ***Event Handling:*** Developers can register event listeners to receive notifications about playback events like completion, errors, buffering, and seek completion.
- ***Media Streaming:*** The API allows for streaming media content from a remote server, making it suitable for applications that require streaming capabilities, such as music and video streaming apps.
- ***Audio Focus:*** In Android, the MediaPlayer API integrates with the audio focus system, allowing apps to request and manage audio focus, ensuring a seamless user experience in scenarios like background audio playback.

2.1.1.1 Introduction to the `MediaPlayer` class for audio and video playback.

The MediaPlayer class in Java is a fundamental component for audio and video playback in mobile application development. It provides a versatile way to play both audio and video files on Android devices. The class is part of the Android framework and offers a straightforward API for controlling playback, including features like play, pause, stop, and seek. It allows you to play media files from local storage or online sources.

How to use the MediaPlayer class:

- ***Initialization:*** To use the MediaPlayer, you need to create an instance of the class.
- ***Data Source:*** Set the data source (the file or URL to your audio or video content) using setDataSource().
- ***Preparation:*** Prepare the MediaPlayer using prepare(), which initializes the player and buffers the data source.
- ***Playback Control:*** You can control playback with methods like start() (play), pause(), stop(), and seekTo().
- ***Event Handling:*** Implement listeners to handle events like completion or errors.
- ***Release:*** Always release the MediaPlayer resources with release() when you're done to free up system resources.

Example:

```
 // Initialize MediaPlayer
MediaPlayer mediaPlayer = new MediaPlayer();

// Set the data source (file path or URL)
mediaPlayer.setDataSource("path_to_media_file.mp3");

// Prepare and start playback
mediaPlayer.prepare();
mediaPlayer.start();
```

2.1.1.2 Load and play media files using `MediaPlayer`

To load and play media files in a mobile application using Java, you typically use the Android platform's MediaPlayer class. This class allows you to load and play audio and video files. The below Java code is an example of loading and playing both audio and video files in an Android mobile application.

Example: Loading and Playing Audio File

```
import android.media.MediaPlayer;

import android.os.Bundle;

import android.support.v7.app.AppCompatActivity;

public class AudioPlayerActivity extends AppCompatActivity {

  private MediaPlayer mediaPlayer;

  private String audioFilePath = "your_audio_file.mp3"; // Replace with your audio file path


  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_audio_player);
    mediaPlayer = new MediaPlayer();
    try {
      mediaPlayer.setDataSource(audioFilePath);
      mediaPlayer.prepare();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public void playAudio() {
    if (!mediaPlayer.isPlaying()) {
            mediaPlayer.start();
        }
    }
    public void pauseAudio() {
        if (mediaPlayer.isPlaying()) {
            mediaPlayer.pause();
        }
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
        if (mediaPlayer != null) {
            mediaPlayer.release();
        }
    }
}
```

The above code demonstrates how to load and play an audio file. The **onCreate method** initializes the MediaPlayer, and you can use the **playAudio** and **pauseAudio** methods to control playback. Finally **release** the MediaPlayer resources when you're done.

Example: Loading and Playing Video File:

```
import android.net.Uri;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.VideoView;

public class VideoPlayerActivity extends AppCompatActivity {
    private VideoView videoView;
    private String videoFilePath = "your_video_file.mp4"; // Replace with your video file path

    @Override
```

```
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_video_player);
    videoView = findViewById(R.id.videoView);
    Uri videoUri = Uri.parse(videoFilePath);
    videoView.setVideoURI(videoUri);
    videoView.start();
  }
}
```

This code loads and plays a video file using the VideoView widget. Replace "your_video_file.mp4" with the actual path to your video file.

Note: Remember to handle exceptions and release resources properly for smooth playback and resource management.

2.1.1.3 Controlling playback (play, pause, stop, seek, Listen, Error Handling).

Controlling playback in mobile application development, particularly for audio and video, often involves using the '**MediaPlayer**' class (Android) for Android development and the **'AVPlayer'** class (iOS) for iOS development.

The below Java coding is an example for Android using the MediaPlayer class.

i. ***Play Media:***

To start playing audio or video, use the **'start()**' method of the **'MediaPlayer'** class. Here's an example:

```
mediaPlayer.start(); // Start playback
```

ii. ***Pause Media:***

The playback can be paused using the **'pause ()'** method:

```
mediaPlayer.pause(); // Pause playback
```

iii. Stop Media:

To stop playback, use the **'stop()'** method followed by resetting the MediaPlayer:

```
mediaPlayer.stop(); // Stop playback

mediaPlayer.reset(); // Reset the MediaPlayer for future use
```

iv. Seek To a Specific Position:

To seek to a specific position in the media using the **'seekTo()'** method with the desired time in milliseconds:

```
int positionInMillis = 30000; // 30 seconds

mediaPlayer.seekTo(positionInMillis); // Seek to 30 seconds
```

v. Listening to Playback Completion:

Implement an 'OnCompletionListener' to be notified when the media playback completes:

```
mediaPlayer.setOnCompletionListener(new MediaPlayer.OnCompletionListener() {

  @Override

  public void onCompletion(MediaPlayer mp) {

    // Playback has completed

  }

});
```

vi. Error Handling:

Handle errors by implementing an 'OnErrorListener'

```
mediaPlayer.setOnErrorListener(new MediaPlayer.OnErrorListener() {

  @Override
```

```
        public boolean onError(MediaPlayer mp, int what, int extra) {
            // Handle the error
            return false; // Returning false indicates that the error was not handled
        }
    });
```

This code is designed for Android mobile application development using the MediaPlayer class.

Note: Remember to initialize and release the MediaPlayer instance properly, and ensure the necessary permissions and resource management for media playback are set up in your Android application.

2.2 ExoPlayer for Advanced Playback:

ExoPlayer is an open-source media player library developed by Google that offers advanced features and capabilities for multimedia playback in mobile applications. It is a highly flexible and extensible solution that provides better control and performance compared to the standard Android **'MediaPlayer'.**

ExoPlayer is an excellent choice for applications that require advanced multimedia capabilities and a high degree of customization. It is widely used in video streaming, music, and other multimedia-rich mobile apps, and it provides a powerful foundation for creating immersive user experiences.

Here's a high-level example of using ExoPlayer for video playback in an Android app:

Java code demonstrates how to use ExoPlayer to play a video in an Android application.

```
// Create an ExoPlayer instance
SimpleExoPlayer exoPlayer = new SimpleExoPlayer.Builder(context).build();
```

In this section, we create an instance of the ExoPlayer, which is the core component responsible for video playback. We use the **'SimpleExoPlayer.Builder'** to create an ExoPlayer instance.

```
// Define the media source
MediaSource mediaSource = new ProgressiveMediaSource.Factory(dataSourceFactory)
   .createMediaSource(Uri.parse("https://example.com/video.mp4"));
```

Here, we define the media source, which is the video that ExoPlayer will play. In this code, we are creating a ProgressiveMediaSource using a dataSourceFactory, which should be configured earlier in your code. The **dataSourceFactory** is responsible for providing data from the source, and it can be customized to support various sources like local files, HTTP URLs, or other data retrieval mechanisms. In this example, we're using a remote video file hosted at "https://example.com/video.mp4".

```
// Prepare the player with the media source
exoPlayer.setMediaSource(mediaSource);
exoPlayer.prepare();
```

We set the created **mediaSource** as the media to be played by the ExoPlayer. After that, we call **prepare**() to initialize the player and start the buffering process. This prepares ExoPlayer to start playing the media.

```
// Attach the player to a view for rendering
playerView.setPlayer(exoPlayer);
```

In this section, we attach the ExoPlayer to **a playerView**. The **playerView** is a UI component that displays the video content and provides playback control widgets. By setting the player to the **playerView**, it will handle rendering the video and user interactions.

```
// Control playback
exoPlayer.setPlayWhenReady(true); // Start playback
```

exoPlayer.seekTo(0); // Seek to a specific position

Here, we control the playback of the video. We use **setPlayWhenReady(true)** to start playback. The **seekTo(0)** function allows us to seek to a specific position in the video; in this case, we're seeking to the beginning (position 0).

// Release the player when done

exoPlayer.release();

Finally, when you're finished with the video playback or when your activity or fragment is being destroyed, it's essential to release the ExoPlayer's resources to ensure proper memory management.

This code provides a basic outline of how to use ExoPlayer to play a video in an Android app. In a real application, you would typically set up event listeners, error handling, and UI controls to provide a seamless user experience.

2. 1.2.1 Features and advantages of using ExoPlayer

The Key Features of ExoPlayer are:

- Media Format Support: ExoPlayer supports a wide range of media formats, including common ones like MP4, MKV, WebM, as well as adaptive streaming formats like DASH, HLS, and SmoothStreaming.
- Adaptive Streaming: ExoPlayer excels in adaptive streaming, allowing seamless quality adjustments during playback for the best possible experience, making it suitable for video-on-demand (VoD) and live streaming applications.
- Customization: It offers a high level of customization, allowing developers to control every aspect of playback, from audio and video rendering to buffering and media source selection.
- Media Source Extension: It has a powerful extension system that enables the creation of custom media sources, ideal for integrating with various content providers or adding support for new streaming protocols.

- Multiple Media Tracks: ExoPlayer supports multiple audio, video, and text tracks, making it ideal for applications with multilingual or accessibility requirements.
- Smooth Transitions: It ensures smooth transitions between different media sources, such as ads or switching between different video qualities.
- Seamless Error Handling: ExoPlayer provides comprehensive error handling and recovery mechanisms, ensuring uninterrupted playback even in challenging network conditions.
- Rich API: ExoPlayer has a well-documented API with a wealth of features, which simplifies tasks like media control, subtitles, DRM integration, and more.
- Community and Ecosystem: With an active developer community, ExoPlayer is continuously evolving and well-supported with a range of extensions and plugins for additional features.
- Easy Integration: ExoPlayer can be easily integrated into Android applications using simple APIs, making it accessible for developers with various skill levels.

Example:

```
import android.net.Uri;
import com.google.android.exoplayer2.SimpleExoPlayer;
import com.google.android.exoplayer2.source.MediaSource;
import com.google.android.exoplayer2.source.ProgressiveMediaSource;
import com.google.android.exoplayer2.ui.PlayerView;
import com.google.android.exoplayer2.upstream.DefaultDataSourceFactory;
import com.google.android.exoplayer2.util.Util;

public class VideoPlayerActivity extends AppCompatActivity {
  private SimpleExoPlayer player;
  private PlayerView playerView;

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

```
        playerView = findViewById(R.id.player_view);
        player = new SimpleExoPlayer.Builder(this).build();
        playerView.setPlayer(player);

        Uri videoUri = Uri.parse("https://example.com/sample.mp4");
        MediaSource mediaSource = buildMediaSource(videoUri);
        player.setMediaSource(mediaSource);
        player.prepare();
        player.setPlayWhenReady(true);
    }

    private MediaSource buildMediaSource(Uri uri) {
        DefaultDataSourceFactory dataSourceFactory = new DefaultDataSourceFactory(this,
                Util.getUserAgent(this, "YourApp"));
        return new ProgressiveMediaSource.Factory(dataSourceFactory)
                .createMediaSource(uri);
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        player.release();
    }
}
```

2.1.2.2 How to integrate ExoPlayer into your app.

Integrating ExoPlayer into your Android mobile application is a powerful way to handle media playback. ExoPlayer is an open-source media player library that provides a flexible, customizable, and reliable solution for playing audio and video content.

The steps to integrating ExoPlayer into your Android app using Android Studio:

Step 1: Set Up Your Android Project

Open Android Studio and create a new Android project or open an existing one.

Step 2: Add ExoPlayer Dependency

Open your app-level build.gradle file and add the ExoPlayer dependency in the dependencies section:

```
implementation 'com.google.android.exoplayer:exoplayer:2.X.X' // Use the latest version
```

Step 3: XML Layout

Create the layout for your media player in an XML file. For example, you can add a **SimpleExoPlayerView** to your layout file:

```
<com.google.android.exoplayer2.ui.SimpleExoPlayerView

   android:id="@+id/exoPlayerView"

   android:layout_width="match_parent"

   android:layout_height="wrap_content"

/>
```

Step 4: Initialize ExoPlayer in Your Activity/Fragment

In Java code (e.g., MainActivity.java), initialize and set up ExoPlayer:

```
import android.net.Uri;

import android.os.Bundle;

import androidx.appcompat.app.AppCompatActivity;

import com.google.android.exoplayer2.ExoPlayer;

import com.google.android.exoplayer2.SimpleExoPlayer;
import com.google.android.exoplayer2.source.MediaSource;

import com.google.android.exoplayer2.source.ProgressiveMediaSource;

import com.google.android.exoplayer2.trackselection.DefaultTrackSelector;

import com.google.android.exoplayer2.trackselection.TrackSelector;

import com.google.android.exoplayer2.ui.SimpleExoPlayerView;

import com.google.android.exoplayer2.upstream.DefaultDataSourceFactory;

import com.google.android.exoplayer2.upstream.DefaultHttpDataSourceFactory;

import com.google.android.exoplayer2.util.Util;


public class MainActivity extends AppCompatActivity {
  private SimpleExoPlayer exoPlayer;
  private SimpleExoPlayerView exoPlayerView;

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

    exoPlayerView = findViewById(R.id.exoPlayerView);

    // Create a TrackSelector
    TrackSelector trackSelector = new DefaultTrackSelector(this);

    // Create an ExoPlayer instance
    exoPlayer = new SimpleExoPlayer.Builder(this)
      .setTrackSelector(trackSelector)
      .build();
```

```
        // Bind ExoPlayer to the view
        exoPlayerView.setPlayer(exoPlayer);

        // Define media source (e.g., from a URL)
        String mediaUri = "https://example.com/sample.mp4";
        Uri mediaUriUri = Uri.parse(mediaUri);
        MediaSource mediaSource = new ProgressiveMediaSource.Factory(
            new DefaultHttpDataSourceFactory("userAgent"))
            .createMediaSource(mediaUriUri);

        // Prepare the player with the source
        exoPlayer.prepare(mediaSource);
        exoPlayer.setPlayWhenReady(true);
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        // Release the ExoPlayer when your activity is destroyed
        exoPlayer.release();
    }
}
```

Step 5: Permissions

Make sure to request the necessary permissions in your app's manifest file if you are accessing media files locally or from the internet.

Note: The above code demonstrates how to set up a basic ExoPlayer instance in your Android app. You can customize it further to add features like playback controls, captions, and handling various media formats. Ensure you have internet permissions if you're streaming from the web and error handling for network-related issues.

2.3 Media Streaming:

Media streaming in the context of mobile application development involves the real-time delivery of multimedia content, such as audio or video, over the internet to mobile devices. It enables users to access and consume media content without having to download it entirely, making it suitable for applications like music and video streaming services, live broadcasts, and online gaming.

The Key considerations regarding media streaming in mobile app development:

1. Streaming Protocols:

   - HTTP Live Streaming (HLS): A widely used adaptive streaming protocol for iOS and Android platforms.

   - Dynamic Adaptive Streaming over HTTP (DASH): An alternative to HLS for adaptive streaming.

   - Real-Time Messaging Protocol (RTMP): Commonly used for live video streaming.

   - WebRTC: Used for real-time peer-to-peer communication, ideal for video conferencing and live streaming.

2. Adaptive Streaming: Adaptive streaming adjusts the quality of media based on the viewer's network conditions, ensuring a smooth playback experience. It is crucial for handling varying network speeds in mobile environments.
3. Latency: Low latency is essential for live streaming applications to minimize the delay between the event and when users see it. Technologies like WebRTC or WebSockets can help reduce latency.
4. Content Delivery Network (CDN): CDNs are essential for efficient content delivery. They store and distribute media content across multiple servers worldwide, reducing latency and ensuring high availability.
5. Mobile Platform Considerations: Each mobile platform (iOS and Android) may have specific requirements and recommendations for streaming media. Developers need to be aware of platform-specific APIs and guidelines.

6. Playback Components: Mobile applications often use media player libraries or frameworks (e.g., ExoPlayer for Android and AVPlayer for iOS) for seamless playback.
7. Security: Protecting content from unauthorized access and piracy is crucial. Encryption and authentication mechanisms are commonly used.
8. Monetization: Many streaming apps use subscription models or ads for revenue. Integrating monetization methods is a key consideration.
9. User Experience: A user-friendly interface and features like pause, rewind, and volume control are important for a positive user experience.
10. Offline Access: Some streaming apps allow users to download content for offline access, which can be a significant feature for users with limited connectivity.
11. Analytics: Implementing analytics tools helps monitor user behavior, content popularity, and performance, enabling data-driven decisions for app improvements.

In mobile app development, media streaming apps can be complex, and developers need to consider various factors related to performance, user experience, and scalability to deliver high-quality content to users. Proper implementation of streaming technology is critical for the success of applications in this domain.

3.1.3.1 Concepts of adaptive streaming

Adaptive streaming is a technique used to deliver multimedia content to mobile applications while adapting to varying network conditions and device capabilities.

Two common adaptive streaming protocols are:

- HTTP Live Streaming (HLS) and
- Dynamic Adaptive Streaming over HTTP (DASH).

They work by breaking media content into small segments and dynamically switching between different quality levels based on network conditions. This ensures a smooth viewing experience for users.

HTTP Live Streaming (HLS):

HLS is developed by Apple and widely used for streaming video and audio on iOS devices. HLS uses playlists and media segments to adapt to changing conditions. To implement HLS in Android, you can use the ExoPlayer library.

Example:

```
import com.google.android.exoplayer2.ExoPlayer;

import com.google.android.exoplayer2.SimpleExoPlayer;

import com.google.android.exoplayer2.source.hls.HlsMediaSource;

import com.google.android.exoplayer2.upstream.DefaultHttpDataSourceFactory;

import com.google.android.exoplayer2.util.Util;

// Create an ExoPlayer instance

SimpleExoPlayer player = new SimpleExoPlayer.Builder(context).build();

// Create a data source factory

DefaultHttpDataSourceFactory dataSourceFactory = new
DefaultHttpDataSourceFactory(Util.getUserAgent(context, "YourAppName"));

// Create an HLS media source

HlsMediaSource hlsMediaSource = new HlsMediaSource.Factory(dataSourceFactory)

   .createMediaSource(Uri.parse("http://example.com/your-hls-stream.m3u8"));

// Set the media source to the player

player.setMediaSource(hlsMediaSource);

// Prepare and start the player

player.prepare();

player.setPlayWhenReady(true);
```

DASH (Dynamic Adaptive Streaming over HTTP):

DASH is an industry-standard adaptive streaming protocol that works on various devices and platforms, including Android. It uses manifest files and media segments, allowing seamless switching between different bitrates and resolutions. ExoPlayer also supports DASH.

In both cases, we are leveraging **ExoPlayer** to handle adaptive streaming for us, making it easier to deliver high-quality multimedia content that adapts to the viewer's network conditions and device capabilities.

Example:

```
import com.google.android.exoplayer2.source.dash.DashMediaSource;

import com.google.android.exoplayer2.source.dash.DefaultDashChunkSource;

import com.google.android.exoplayer2.upstream.DefaultHttpDataSourceFactory;

// Create a DASH media source

DashMediaSource dashMediaSource = new DashMediaSource.Factory(new DefaultDashChunkSource.Factory(new DefaultHttpDataSourceFactory("YourAppName")))

   .createMediaSource(Uri.parse("http://example.com/your-dash-stream.mpd"));

// Set the media source to the player

player.prepare(dashMediaSource);

// Prepare and start the player

player.setPlayWhenReady(true);
```

3.1.3.2 Using libraries like ExoPlayer for seamless streaming.

Using libraries like ExoPlayer in mobile application development is a great way to achieve seamless audio and video streaming. ExoPlayer is a popular open-source media player library for Android that provides advanced features for media playback, such as adaptive streaming, extensive format support, and customization options. Below, I'll provide a simple Java code example to demonstrate how to use ExoPlayer for video streaming in an Android application using Android Studio.

Step 1: Setting Up ExoPlayer in Android Studio

To use ExoPlayer in your Android project, make sure to add the necessary dependencies to your **build.gradle** file:

```
implementation 'com.google.android.exoplayer:exoplayer:2.X.X'
```

Note: Replace '2.X.X' with the latest version of ExoPlayer available at the time you implement it.

Step 2: Implement ExoPlayer in Your Android Application

Here's a basic example of how to use ExoPlayer to play a video in an Android application:

```
import android.net.Uri;

import android.os.Bundle;

import android.support.v7.app.AppCompatActivity;

import com.google.android.exoplayer2.DefaultLoadControl;

import com.google.android.exoplayer2.DefaultRenderersFactory;

import com.google.android.exoplayer2.ExoPlayerFactory;

import com.google.android.exoplayer2.SimpleExoPlayer;

import com.google.android.exoplayer2.extractor.DefaultExtractorsFactory;

import com.google.android.exoplayer2.source.ExtractorMediaSource;

import com.google.android.exoplayer2.trackselection.DefaultTrackSelector;

import com.google.android.exoplayer2.ui.SimpleExoPlayerView;
import com.google.android.exoplayer2.upstream.DefaultHttpDataSourceFactory;

import com.google.android.exoplayer2.util.Util;


public class VideoPlayerActivity extends AppCompatActivity {

    private SimpleExoPlayer player;
    private SimpleExoPlayerView playerView;

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

        playerView = findViewById(R.id.player_view);

        // Create a simple ExoPlayer instance
        player = ExoPlayerFactory.newSimpleInstance(
                new DefaultRenderersFactory(this),
                new DefaultTrackSelector(),
                new DefaultLoadControl());

        // Bind the player to the view
        playerView.setPlayer(player);

        // Define the video URL or file path
        String videoUrl = "https://www.example.com/sample.mp4";
        Uri videoUri = Uri.parse(videoUrl);

        // Create a media source for the player
        ExtractorMediaSource mediaSource = new ExtractorMediaSource(
                videoUri,
                new DefaultHttpDataSourceFactory("user-agent"),
                new DefaultExtractorsFactory(),
                null,
                null);

        // Prepare the player with the media source
        player.prepare(mediaSource);
    }

    @Override
    protected void onStart() {
        super.onStart();
        if (Util.SDK_INT > 23) {
            player.setPlayWhenReady(true);
        }
    }

    @Override
    protected void onStop() {
        super.onStop();
        if (Util.SDK_INT > 23) {
            player.setPlayWhenReady(false);
        }
    }

    @Override
    protected void onResume() {
        super.onResume();
        if ((Util.SDK_INT <= 23 || player == null)) {
            player.setPlayWhenReady(true);
        }
    }

    @Override
    protected void onPause() {
        super.onPause();
        if (Util.SDK_INT <= 23) {
            player.setPlayWhenReady(false);
        }
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        releasePlayer();
    }

    private void releasePlayer() {
        if (player != null) {
            player.release();
            player = null;
        }
    }
}
```

In this example, we create an instance of ExoPlayer and use it to play a video from a URL. Make sure to adjust the video URL and view elements (e.g., player_view) according to your application's requirements and layout. Also, ensure that you handle the player's lifecycle events properly to prevent resource leaks.

Self-Assessment Questions -1

1. What types of audio content can be played in mobile applications?

   a. Music tracks and podcasts
   b. Video streams
   c. Text messages
   d. GPS coordinates
2. Which audio format is commonly used for music playback in mobile apps?
   a. MP4
   b. MP3
   c. GIF
   d. TXT
3. ____________ is one of the main features of audio playback in mobile apps?
   a. Adaptive streaming
   b. GPS tracking
   c. Video recording
   d. File encryption
4. Which mobile app category relies heavily on video playback?
   a. Weather apps
   b. Fitness apps
   c. Shopping apps
   d. Alarm clock apps
5. What is a common video codec used for mobile app development?
   a. H.264
   b. FLAC
   c. MP3
   d. BMP
6. Which of the following features does ExoPlayer excel at?
   a. Image recognition
   b. Adaptive streaming
   c. Contact management
   d. Location tracking
7. What does HLS stand for in the context of media streaming?
   a. High-Level Security

b. HyperLink Streaming
c. HTTP Live Streaming
d. Hybrid Language Syntax

8. Which mobile platform is the MediaPlayer API widely used for?
    a. iOS
    b. Android
    c. Windows Mobile
    d. Blackberry
9. How do you release resources when done using the MediaPlayer class?
    a. Call the close() method
    b. Use the dispose() function
    c. Invoke the release() method
    d. Delete the instance manually
10. Which organization developed ExoPlayer?
    a. Facebook
    b. Google
    c. Apple
    d. Microsoft
11. Which streaming protocol was developed by Apple and is commonly used for iOS devices?
    a. DASH
    b. WebRTC
    c. RTMP
    d. HLS
12. Which media format is NOT supported by ExoPlayer?
    a. MP4
    b. MKV
    c. GIF
    d. WebM

3. RECORDING AUDIO AND VIDEO

The integration of audio and video recording capabilities in mobile applications has revolutionized how we capture and share our experiences. Mobile devices are equipped with high-quality cameras and microphones, making it easier than ever to create multimedia content. From social media platforms to professional recording apps, the ability to record audio and video has become a fundamental feature in mobile app development.

One of the most prominent examples of audio and video recording in mobile apps is found in social media applications like Instagram and TikTok. These platforms empower users to capture, edit, and share short videos, often accompanied by music or voiceovers. The seamless experience provided by these apps has transformed content creation into a global phenomenon.

Audio and video recording features are not limited to entertainment or social networking. Mobile applications in various domains utilise this functionality to enhance user experiences. For instance, fitness apps may offer video recording to enable users to record and review their workouts. Educational apps can employ audio recording to facilitate language learning or note-taking. Additionally, businesses can use mobile apps to create video content for marketing and customer support.

Mobile app developers must consider various factors when implementing audio and video recording. They must handle media file formats, permissions for accessing device hardware, data storage, and user-friendly interfaces for recording and playback. Ensuring privacy and data security is also important.

The capabilities of modern smartphones, combined with powerful development frameworks, make it relatively straightforward to incorporate audio and video recording in mobile applications. Developers can utilise platform-specific APIs and third-party libraries to streamline the process.

3.1 Audio Recording with MediaRecorder

Audio recording in mobile application development for Android is commonly achieved using the **MediaRecorder** class. It provides a straightforward way to capture audio from a device's microphone and save it as a file for various purposes, such as voice memos, audio notes, or even creating a voice messaging app.

Steps to use **MediaRecorder** to record audio in an Android application:

1. **Initialize MediaRecorder:** First, you must create and configure an instance of MediaRecorder. This involves setting the audio source, output format, encoder, and output file location.

   Example:

   ```
   MediaRecorder mediaRecorder = new MediaRecorder();

   mediaRecorder.setAudioSource(MediaRecorder.AudioSource.MIC);

   mediaRecorder.setOutputFormat(MediaRecorder.OutputFormat.MPEG_4);

   mediaRecorder.setAudioEncoder(MediaRecorder.AudioEncoder.AAC);

   mediaRecorder.setOutputFile(outputFilePath);
   ```

   Configuration: Set the audio source, output format, audio encoder, and the output file using appropriate methods (‘setAudioSource()’, ‘setOutputFormat()’, ‘setAudioEncoder()’, and ‘setOutputFile()’).

2. **Prepare ‘MediaRecorder’**

   After initialising, you must call prepare() to prepare the MediaRecorder for recording:

   ```
   try {

     mediaRecorder.prepare();

   } catch (IOException e) {
   ```

```
    e.printStackTrace();
  }
```

3. **Start Recording:** Use the start() method to begin audio recording.

   ```
   mediaRecorder.start();
   ```

4. **Stop Recording:** When you're done recording, use the stop() method to stop the recording process:

   ```
   mediaRecorder.stop();
   ```

5. **Release Resources:** To properly release the MediaRecorder resources, call release():

   ```
   mediaRecorder.release();
   ```

6. **Exception Handling:** It's important to handle exceptions appropriately, especially when dealing with file I/O and audio recording operations. Make sure to catch and handle exceptions to prevent crashes in your application.
7. **Permissions:** Don't forget to request and handle the necessary permissions in your AndroidManifest.xml and request them dynamically in your code if you're targeting Android 6.0 (API level 23) or higher. For audio recording, you need the RECORD_AUDIO and WRITE_EXTERNAL_STORAGE permissions.
8. **Output File:** Specify the path for the output file, where the recorded audio will be saved. Ensure you have the appropriate permissions to write to the specified file location.

Example:

```
import android.media.MediaRecorder;

import android.os.Environment;

import java.io.IOException;


public class AudioRecorder {
```

```
  private MediaRecorder mediaRecorder;
  private String outputFile;

  public AudioRecorder() {
    mediaRecorder = new MediaRecorder();
  }

  public void startRecording() {
    outputFile = Environment.getExternalStorageDirectory().getAbsolutePath() +
"/myaudio.3gp";

    mediaRecorder.reset();
    mediaRecorder.setAudioSource(MediaRecorder.AudioSource.MIC);
    mediaRecorder.setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP);
    mediaRecorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);
    mediaRecorder.setOutputFile(outputFile);

    try {
      mediaRecorder.prepare();
    } catch (IOException e) {
      e.printStackTrace();
    }

    mediaRecorder.start();
    }
    public void stopRecording() {
        mediaRecorder.stop();
        mediaRecorder.release();
    }
}
```

3. 2.1.1 Configuring audio source, format, and output file.

In mobile application development, particularly for Android, configuring audio source, format, and output files is crucial when working with audio. You might need to record audio, specify the audio format, and store it in the desired output file.

i. **Configuring Audio Source:**

You can set the audio source for recording using Android's **MediaRecorder** class. Here are some common audio sources:

- **MediaRecorder.AudioSource.MIC:** The microphone.
- **MediaRecorder.AudioSource.CAMCORDER:** The device's camcorder microphone.
- **MediaRecorder.AudioSource.VOICE_RECOGNITION:** The microphone optimized for voice recognition.

ii. **Configuring Audio Format:**

You can specify the audio format for recording, such as encoding type and bit rate. For example, you can use **MediaRecorder.OutputFormat.THREE_GPP for 3GPP** format, and **MediaRecorder.AudioEncoder.AMR_NB for AMR-NB** encoding.

iii. **Configuring Output File:**

Set the output file where the recorded audio will be saved. This can be an absolute file path or a **File** object.

Example:

```
import android.media.MediaRecorder;
import java.io.IOException;

public class AudioRecorder {
  private MediaRecorder mediaRecorder;
  private String outputFile;

  public AudioRecorder(String outputPath) {
    outputFile = outputPath;
  }

  public void startRecording() {
    mediaRecorder = new MediaRecorder();
    mediaRecorder.setAudioSource(MediaRecorder.AudioSource.MIC);
    mediaRecorder.setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP);
    mediaRecorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);
    mediaRecorder.setOutputFile(outputFile);

    try {
      mediaRecorder.prepare();
      mediaRecorder.start();
    } catch (IOException e) {
      e.printStackTrace();
```

```
        }
    }

    public void stopRecording() {
        if (mediaRecorder != null) {
            mediaRecorder.stop();
            mediaRecorder.release();
            mediaRecorder = null;
        }
    }
}
```

This code initialises an **AudioRecorder** class, sets the audio source, format, and output file, and provides methods to start and stop audio recording.

3.2.1.2 Starting, pausing, and stopping audio recording.

Starting, pausing, and stopping audio recording is an essential aspect of mobile application development, particularly in Android Studio. To achieve this, you'll use the Android MediaRecorder class, which provides a straightforward API for recording audio. Below, I'll explain how to start, pause, and stop audio recording with Java code examples.

Starting Audio Recording:

To start audio recording, you need to configure the MediaRecorder with the appropriate settings and then call start() to begin recording.

Example:

```
import android.media.MediaRecorder;
public class AudioRecorder {
  private MediaRecorder mediaRecorder;

  private String outputFile;

  public AudioRecorder(String outputPath) {

    mediaRecorder = new MediaRecorder();

    outputFile = outputPath;

  }

  public void startRecording() {

    try {

      mediaRecorder.setAudioSource(MediaRecorder.AudioSource.MIC);

      mediaRecorder.setOutputFormat(MediaRecorder.OutputFormat.MPEG_4);

      mediaRecorder.setAudioEncoder(MediaRecorder.AudioEncoder.AAC);

      mediaRecorder.setOutputFile(outputFile);


      mediaRecorder.prepare();

      mediaRecorder.start();

    } catch (Exception e) {

      e.printStackTrace();

    }

  }

}
```

Pausing Audio Recording:

Pausing audio recording can be a bit more complex because Android's MediaRecorder doesn't have a built-in pause/resume feature. Instead, you need to create a new file for each recorded segment when you pause, then merge them later.

Example:

```
public void pauseRecording() {
    if (mediaRecorder != null) {
        mediaRecorder.stop();
        mediaRecorder.reset();
        mediaRecorder.release();
        mediaRecorder = null;
    }
}
```

Stopping Audio Recording:

To stop audio recording, you'll need to stop the MediaRecorder, release its resources, and potentially do any required post-processing like merging segments.

```
public void stopRecording() {
    if (mediaRecorder != null) {
        mediaRecorder.stop();
        mediaRecorder.release();
        mediaRecorder = null;
    }
}
```

3.2 Video Recording with Camera2 API

Video recording with the Camera2 API is a fundamental aspect of mobile application development, especially for Android apps. It allows you to access the device's camera to capture and record videos

Steps to use the Camera2 API for video recording in Android Studio:

i. **Camera2 API Setup:** First, ensure you have the necessary permissions in your AndroidManifest.xml for camera and audio recording.

```
<uses-feature android:name="android.hardware.camera" />

<uses-feature android:name="android.hardware.camera.autofocus" />

<uses-permission android:name="android.permission.CAMERA" />

<uses-permission android:name="android.permission.RECORD_AUDIO" />

<uses-permission    android:name="android.permission.WRITE_EXTERNAL_STORAGE"
/>
```

ii. **Initialize the Camera2 API:** In your Java code, initialize the Camera2 API and set up the camera capture session, as well as the media recorder.

```
import android.hardware.camera2.*;

import android.media.MediaRecorder;

public class VideoRecorder {

  private CameraCaptureSession captureSession;

  private MediaRecorder mediaRecorder;

  public void setupCamera() {

    CameraManager        cameraManager        =        (CameraManager)
getSystemService(Context.CAMERA_SERVICE);

    String cameraId = getCameraId(); // Get the appropriate camera ID

    mediaRecorder = new MediaRecorder();

    mediaRecorder.setAudioSource(MediaRecorder.AudioSource.MIC);

    mediaRecorder.setVideoSource(MediaRecorder.VideoSource.SURFACE);
```

```
        mediaRecorder.setOutputFormat(MediaRecorder.OutputFormat.MPEG_4);

        mediaRecorder.setAudioEncoder(MediaRecorder.AudioEncoder.AAC);

        mediaRecorder.setVideoEncoder(MediaRecorder.VideoEncoder.H264);

        mediaRecorder.setOutputFile(getOutputFile()); // Set the output file path

    }

    public void startRecording() {

        try {

            mediaRecorder.prepare();

            Surface recordingSurface = getRecordingSurface();

            captureSession.setRepeatingRequest(getCaptureRequest(recordingSurface),
  null, null);

            mediaRecorder.start();

        } catch (Exception e) {

            e.printStackTrace();

        }

    }


    public void stopRecording() {

        mediaRecorder.stop();

        mediaRecorder.reset();

        closeCaptureSession();
    }

    // Other methods for camera setup, configuration, and handling events

  }
```

iii. **Permissions and UI Integration:** Request runtime permissions for the camera and audio, and integrate the video recording functionality into your app's user interface (e.g., start and stop recording buttons).

iv. **Error Handling and Event Handling:** Implement error handling and event listeners to respond to changes in camera state and recording status.

3.2.2.1 Setting up the camera for video recording.

Setting up the camera for video recording in an Android application involves using the Camera2 API, which is the recommended way of handling camera operations.

Steps to set up the camera for video recording in Android Studio.

i. Permissions and Features

In your AndroidManifest.xml, make sure to add the necessary permissions and features:

```
<uses-permission android:name="android.permission.CAMERA" />

<uses-permission android:name="android.permission.RECORD_AUDIO" />

<uses-feature android:name="android.hardware.camera" />

<uses-feature android:name="android.hardware.camera.autofocus" />
```

ii. Create a Camera Preview Layout

Design a layout for the camera preview in your activity's XML file.

For example, you can add a TextureView to display the camera preview:

```
<TextureView

  android:id="@+id/textureView"
```

```
    android:layout_width="match_parent"

    android:layout_height="match_parent" />
```

ii. Initialize Camera2 API

In Java code, initialize the Camera2 API by creating a CameraCaptureSession, a CameraDevice, and setting up the camera preview:

Example:

```
private CameraManager cameraManager;

private CameraDevice cameraDevice;

private CameraCaptureSession cameraCaptureSession;

private CaptureRequest.Builder captureRequestBuilder;

private TextureView textureView;

...

cameraManager = (CameraManager) getSystemService(Context.CAMERA_SERVICE);

textureView = findViewById(R.id.textureView);

...

private void openCamera() {

  try {

      String cameraId = cameraManager.getCameraIdList()[0]; // Use the first available
camera

      // Open the camera

      if (ActivityCompat.checkSelfPermission(this, Manifest.permission.CAMERA) ==
PackageManager.PERMISSION_GRANTED) {
```

```
            cameraManager.openCamera(cameraId, cameraStateCallback, null);

          }

        } catch (CameraAccessException e) {

          e.printStackTrace();

        }

    }
```

iii. Prepare for Video Recording

Users need to set up the media recorder to capture video

```
private MediaRecorder mediaRecorder;
private boolean isRecording = false;

...

private void startRecording() {
    try {
        setupMediaRecorder();
        SurfaceTexture surfaceTexture = textureView.getSurfaceTexture();
        surfaceTexture.setDefaultBufferSize(videoWidth, videoHeight);
        Surface previewSurface = new Surface(surfaceTexture);

        captureRequestBuilder                                          =
cameraDevice.createCaptureRequest(CameraDevice.TEMPLATE_RECORD);
        captureRequestBuilder.addTarget(previewSurface);
        captureRequestBuilder.addTarget(mediaRecorder.getSurface());

        cameraDevice.createCaptureSession(Arrays.asList(previewSurface,
mediaRecorder.getSurface()), cameraCaptureSessionCallback, null);
```

```
            } catch (Exception e) {
                e.printStackTrace();
            }
        }

        private void setupMediaRecorder() {
            mediaRecorder = new MediaRecorder();
            mediaRecorder.setAudioSource(MediaRecorder.AudioSource.MIC);
            mediaRecorder.setVideoSource(MediaRecorder.VideoSource.SURFACE);
            mediaRecorder.setOutputFormat(MediaRecorder.OutputFormat.MPEG_4);
            mediaRecorder.setVideoEncoder(MediaRecorder.VideoEncoder.H264);
            mediaRecorder.setAudioEncoder(MediaRecorder.AudioEncoder.AAC);
            mediaRecorder.setOutputFile(outputFilePath);
            mediaRecorder.setVideoSize(videoWidth, videoHeight);
            mediaRecorder.setVideoFrameRate(videoFrameRate);
            mediaRecorder.setVideoEncodingBitRate(videoBitRate);
            mediaRecorder.setOrientationHint(totalRotation);
            try {
                mediaRecorder.prepare();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
```

v. Start/Stop Video Recording

Users can use a button or any other trigger to start and stop video recording:

```
private void startRecording() {
    // ...
    mediaRecorder.start();

    isRecording = true;

}


private void stopRecording() {

    mediaRecorder.stop();

    mediaRecorder.reset();

    isRecording = false;

}
```

3.2.2.2 Handling video recording states and file output

To handle video recording states and file output in a mobile application developed with Android Studio, you can use the MediaRecorder class, which provides an easy way to capture video. Here's an explanation of the key steps involved, followed by a Java code example.

Steps to Handle Video Recording States and File Output:

- Set Up Permissions: Ensure that you have the necessary permissions in your AndroidManifest.xml file to access the camera and storage, as video recording requires these permissions.
- Initialize MediaRecorder: Create an instance of the **MediaRecorder** class to configure and control the video recording process.
- Configure Settings: Set up the video source, output format, encoding parameters, and output file location. You can specify settings like video resolution, frame rate, and audio source.
- Prepare and Start Recording: Prepare the **MediaRecorder** with **prepare() and** start recording using **start().**

- Manage Recording State: Handle recording state transitions, such as starting, pausing, resuming, and stopping recording, by appropriately calling **start(), pause(), resume(), and stop()** methods.
- Release Resources: Properly release the **MediaRecorder** instance when you're done to free up resources.

Example: The below Java code demonstrates video recording using the **MediaRecorder** class in an Android Studio project:

```
import android.media.MediaRecorder;

import android.os.Environment;
public class VideoRecorder {
  private MediaRecorder mediaRecorder;
  private String outputFile;


  public VideoRecorder() {
    mediaRecorder = new MediaRecorder();
    outputFile = Environment.getExternalStorageDirectory().getPath() + "/myvideo.mp4";
// Set the output file path
  }


  public void startRecording() {
    if (mediaRecorder != null) {
      mediaRecorder.setAudioSource(MediaRecorder.AudioSource.CAMCORDER);
      mediaRecorder.setVideoSource(MediaRecorder.VideoSource.CAMERA);
      mediaRecorder.setOutputFormat(MediaRecorder.OutputFormat.MPEG_4);
      mediaRecorder.setVideoEncoder(MediaRecorder.VideoEncoder.H.264);
```

```
        mediaRecorder.setAudioEncoder(MediaRecorder.AudioEncoder.AAC);
        mediaRecorder.setOutputFile(outputFile);

        try {
            mediaRecorder.prepare();
            mediaRecorder.start();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

public void stopRecording() {
    if (mediaRecorder != null) {
        mediaRecorder.stop();
        mediaRecorder.release();
        mediaRecorder = null;
    }
}
}
```

This code initializes the MediaRecorder, sets up the output file path, starts and stops recording.

3.3 Permissions and Best Practices for audio and video recording

When it comes to audio and video recording in mobile application development using Android Studio, there are important permissions and best practices that must be implemented to ensure security, user privacy, and a seamless user experience.

Permissions:

1. **Camera Permission:** For video recording, you must request the `**CAMERA**` permission in your AndroidManifest.xml and ask for runtime permission from the user.
2. **Audio Permission:** To record audio, you need the **`RECORD_AUDIO`** permission, which also requires runtime permission.

Best Practices:

i. Permission Request: Request permissions at runtime, and ensure you handle both the allowed and denied scenarios.

ii. Runtime Permission Request Example:

```
// Check if the CAMERA and RECORD_AUDIO permissions are granted
if (ContextCompat.checkSelfPermission(this, Manifest.permission.CAMERA) !=
PackageManager.PERMISSION_GRANTED ||
  ContextCompat.checkSelfPermission(this, Manifest.permission.RECORD_AUDIO) !=
PackageManager.PERMISSION_GRANTED) {
// Request permissions
ActivityCompat.requestPermissions(this, new
String[]{Manifest.permission.CAMERA, Manifest.permission.RECORD_AUDIO},
REQUEST_PERMISSIONS);

}
```

iii. Camera and Audio Initialization:

Use `Camera2` API for camera access and `MediaRecorder` for audio and video recording. Initialize them properly.

iv. **Capture Quality Settings:** Configure video and audio settings (resolution, bit rate, format) according to your app's requirements.

v. **Recording Start and Stop:** Start and stop video recording using the `MediaRecorder` class.

```
// Example to start recording
mediaRecorder = new MediaRecorder();
camera = Camera.open(cameraId);

// Set camera parameters
camera.unlock();
mediaRecorder.setCamera(camera);

// Set audio source and output format
mediaRecorder.setAudioSource(MediaRecorder.AudioSource.CAMCORDER);
mediaRecorder.setVideoSource(MediaRecorder.VideoSource.CAMERA);

// Set output file, encoding format, etc.

// Start recording
mediaRecorder.prepare();
mediaRecorder.start();

// To stop recording
mediaRecorder.stop();
mediaRecorder.reset();

mediaRecorder.release();

camera.lock();
```

vi. **File Storage and Management:** Save recorded files in appropriate locations, and consider implementing logic for file naming and storage management.

vii. **Error Handling:** Implement error handling to manage exceptions during recording gracefully.

viii. **UI and User Feedback:** Provide user feedback during recording, such as showing recording indicators and progress.

ix. **Permission and Feature Checks:** Verify that the device has a camera and audio recording capabilities before attempting to record.

x. **Testing and Emulator Usage:** Test on real devices as emulator camera and audio capabilities may differ.

Self-Assessment Questions - 2

13. What is the primary class used for audio recording in Android mobile app development?
    a. AudioRecorder
    b. MediaSource
    c. AudioPlayer
    d. MediaRecorder
14. What is the purpose of the prepare() method in the MediaRecorder class?
    a. Start audio recording
    b. Pause audio recording
    c. Stop audio recording
    d. Prepare the MediaRecorder for recording
15. Which permission is required for audio recording in Android applications?
    a. CAMERA
    b. INTERNET
    c. RECORD_AUDIO

d. LOCATION

16. Which audio source is used to capture audio from the device's microphone?
    a. MediaRecorder.AudioSource.CAMCORDER
    b. MediaRecorder.AudioSource.MIC
    c. MediaRecorder.AudioSource.VOICE_RECOGNITION
    d. MediaRecorder.AudioSource.SPEAKER
17. Which class is commonly used for video recording with the Camera2 API in Android?
    a. VideoRecorder
    b. CameraRecorder
    c. VideoCapture
    d. MediaRecorder
18. Which method is used to start audio recording with the MediaRecorder?
    a. initialize()
    b. record()
    c. start()
    d. play()
19. Which of the following is not a commonly used audio source for recording?
    a. MediaRecorder.AudioSource.MIC
    b. MediaRecorder.AudioSource.CAMCORDER
    c. MediaRecorder.AudioSource.SCREEN_CAPTURE
    d. MediaRecorder.AudioSource.VOICE_RECOGNITION
20. What is the main purpose of handling permissions in Android applications for audio and video recording?
    a. Enhancing app performance
    b. Ensuring user privacy and security
    c. Adding new features to the app
    d. Streamlining the user interface

4. SUMMARY

In mobile application development, the capability to play audio and video is a fundamental feature, and Android provides developers with powerful tools to achieve this. Two primary APIs stand out: the **MediaPlayer** API and ExoPlayer. These APIs enable seamless audio and video playback within Android apps, and they each have their unique features and advantages.

The **MediaPlayer** class is a pivotal component for audio and video playback in Android. It offers a versatile platform for playing audio and video files. By initialising a **MediaPlayer** instance, setting a data source, preparing it, and controlling playback, developers can create compelling multimedia experiences.

Developers load media files for playback by setting the data source, which can be a local file, remote URL, or even a content URI. The media source is then prepared and played. This class allows developers to stream and play media files with ease.

MediaPlayer provides methods to control playback, including play, pause, stop, and seek. It also allows handling events such as completion and errors. Properly managing these controls enhances the user experience.

ExoPlayer is a sophisticated media player library developed by Google. It offers a plethora of features, including adaptive streaming, support for various media formats, and robust customization options. ExoPlayer is a great choice for those looking to create feature-rich and high-performance media applications.

Integrating ExoPlayer into an Android app involves adding its dependencies and configuring it to load media. It supports adaptive streaming, which allows the player to adjust the quality of the media stream based on the user's network conditions. ExoPlayer's modularity and flexibility make it suitable for diverse media playback needs.

Adaptive streaming is a technique that optimizes media playback quality based on the viewer's network conditions. Two common adaptive streaming formats are HTTP Live Streaming (HLS) and Dynamic Adaptive Streaming over HTTP (DASH). These formats break media into small chunks and adapt the quality by switching between different versions of the content.

Libraries like ExoPlayer excel at handling adaptive streaming formats. They automatically adapt to the user's network conditions, ensuring seamless playback without the need for manual quality adjustments. ExoPlayer's adaptive streaming support, along with other features, makes it an excellent choice for building media streaming apps.

Audio and video recording are vital for various mobile applications, including social media, camera apps, and voice recorders. Android provides APIs for both audio and video recording.

For audio recording, Android developers utilize the **MediaRecorder** class. It allows you to configure the audio source, format, and specify the output file where the recorded audio will be stored.

MediaRecorder provides methods to start, pause, and stop audio recording. It is important to correctly handle these states to create a user-friendly recording experience.

To record video, Android developers often use the Camera2 API. This API allows developers to configure and control camera parameters for video recording, such as resolution, frame rate, and focus. Properly managing video recording states, such as recording, pausing, and stopping, is crucial for creating a smooth video recording experience. Additionally, you need to specify the output file where the recorded video will be saved.

Ensuring user privacy and security is paramount when implementing audio and video recording features in mobile applications.

Developers must request and handle permissions for camera access, audio recording, and file storage. Proper runtime permission handling is essential to ensure that the app has the necessary permissions to access these sensitive resources.

Adherence to best practices includes configuring media capture settings for quality and performance, error handling, and user feedback during the recording process. Properly managing resources and ensuring graceful handling of exceptions is vital.

5. ANSWERS

1. Music tracks and podcasts
2. MP3
3. Adaptive streaming
4. Fitness apps
5. H.264
6. Adaptive streaming
7. HTTP Live Streaming
8. Android
9. Invoke the release() method
10. Google
11. HLS
12. GIF
13. MediaRecorder
14. Prepare the MediaRecorder for recording
15. RECORD_AUDIO
16. MediaRecorder.AudioSource.MIC
17. MediaRecorder
18. start()
19. MediaRecorder.AudioSource.SCREEN_CAPTURE
20. Ensuring user privacy and security

6. TERMINAL QUESTIONS

1. Explain the characteristics of the media player API
2. Explicate how to use the MediaPlayer class with example.
3. Write a code to load and play audio and video file.
4. Write a short note on ExopPlayer for Advanced Playback.
5. Elucidate the common adaptive streaming protocols. with example.
6. Explain Audio Recording with MediaRecorder.
7. Explain how to use the Camera2 API for video recording in Android Studio.
8. Explicate the handling video recording states and file output.

7. TERMINAL ANSWERS

1. Refer to Section 4.1.1
2. Refer to Section 4.1.1.1
3. Refer to Section 4.1.1.2
4. Refer to Section 4.1.2
5. Refer to Section 4.1.3.1
6. Refer to Section 4.2.1
7. Refer to Section 4.2.2
8. Refer to Section 4.2.2.2