CS Android Programming: Homework 3 Music player

Submission. All submissions should be done via git. Refer to the git setup and submission documents for the correct procedure. Please fill in the README file that you will find inside your Android Studio project directory (at the top level).

There is NO code collaboration for homework. Each student must do their own coding and they must do all of their own coding. You can talk to other students about the problem, you can talk to the instructor or TA. If you discuss the homework deeply with someone, please note that in your README. If you use AI, please note it in the README and include your prompts either in the README or in your code (in comments). Please fill in the README that we provide.

No posting code. For all homeworks, PLEASE do NOT post code publicly. If you are having a specific problem with your code, you can always make a private post.

Overview

> Without music to decorate it, time is just a bunch of boring production deadlines or dates by which bills must be paid.
> –Frank Zappa

> Sometimes you have to play a long time to be able to play like yourself.
> —Miles Davis

> There's always a dissonance between what you wish was happening and what is actually happening. That's the nature of creativity, that there's a certain level of disappointment in there.
> —Jerry Garcia

> I remember one time - it might have been a couple times - at the Fillmore East in 1970, I was opening for this sorry-ass cat named Steve Miller. Steve Miller didn't have his shit going for him, so I'm pissed because I got to open for this non-playing motherfucker just because he had one or two sorry-ass records out. So I would come late and he would have to go on first and then we got there we smoked the motherfucking place, everybody dug it.
> —Miles Davis

For this assignment, you will be implementing a simple music player.
On the next page are some screenshots of an example app.

Figure 1: Basic layout of music player

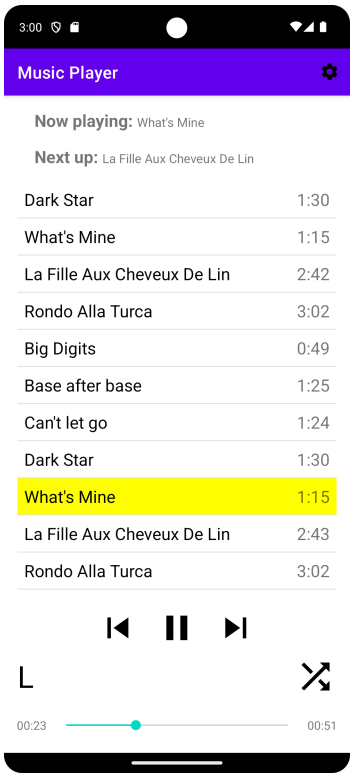

Figure 2: Layout while playing music

Figure 1 shows the initial layout of the player. The first song is selected, but it is not playing. The user can scroll the list of songs and interact with the player's controls.

Figure 2 shows the player while playing. The play/pause icon is set to pause and the seek bar displays the 23s of progress. The text views on the sides of the seek bar show 23s elapsed and 51s remaining.

The song list consists of the same songs repeated twice. That is on purpose. I only want to put so many mp3 files in the project to keep its size down. But I want you to have enough songs so that you can scroll in the recycler view. So I repeat the same songs.

Once you start getting into the details you will ask questions like, "what controls the current song?" I used a string that held the song name at one point. But if we have two songs with the same name, a string is a bad choice. The code you have contains an integer which is an index into the songResources list. But because entries in this list can be logically identical, I add a unique identifier so each entry is distinct.

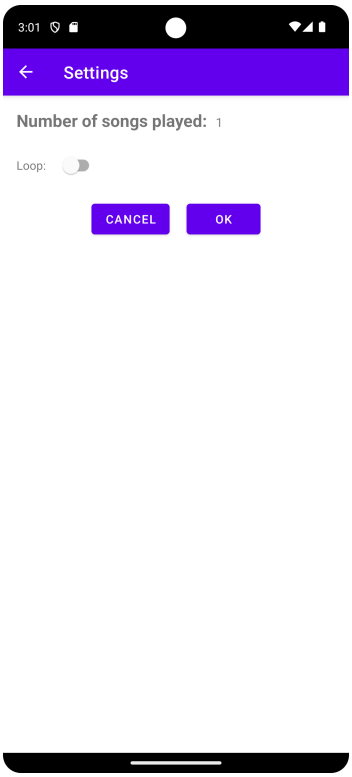

Figure 3: Settings fragment

Figure 3 shows the settings fragment which the user arrives at by pressing the gear icon in the action bar. It displays the number of songs played, and allows the user to change the loop setting for songs. You can confirm the change "OK" or cancel. You can also press the phone back button or the Up button to return to the player. If the player was playing, it continues while the settings are displayed. It should also properly transition to the next song.

The settings layout contains the count of songs played. This will not change, even if the settings fragment is active and the song transitions to the next. If you close the settings fragment and return, the proper count should display.

The settings fragment can display the correct count when it is created, but it does not update. We will learn about live data which will allow us to update data when it changes, but not for this homework.

Specifications

Your app should contain the following elements.

- Text showing the name of the current song that is playing.
- Text showing the next song that will be played.
  - If a song is selected from the list, the names of the current and next song update.
- A RecyclerView containing a list of songs, and their total time.
  - When a song in this list is clicked by a user, that song is highlighted in yellow (the background of the text view is set to yellow).
  - When a song in this list is clicked by a user the song starts playing.
- A play/pause button.

  - Pressing this button plays/pauses the music.
  - The button switches between the play and the pause symbol each time it is clicked.
- Skip forward and skip backwards buttons.
  - Pressing skip forward causes the next song to be played. The song after the last song is the first song.
  - Pressing the skip backwards button causes the previous song to be played, even if the current song is well on its way. The previous button never restarts a song. The song before the first song is the last song.
  - Skip forward/backward does not change the play/pause state of the player, neither does clicking on a song.
  - We provide a function called `setBackgroundDrawable`. Please use that to set the background of an image button. It also sets a tag so we can do automatic testing.
- Loop indicator (far left) and shuffle button (far right)
  - The L for Loop indicator indicates if the current song will restart once it finishes. If the loop condition is true, the background of the L is red and the current song will loop when it finishes. Otherwise, the background of the L is white.
  - When the shuffle button is pressed, the songs are randomly permuted. The current song stays the same, but the next song can change as well as the overall song order. The recyclerview should update to show the new order, and the next song text indicator should also update.
- A slider that shows the progress through the song (hint: use a SeekBar).
  - The slider should update as the song plays.
  - If the user drags the slider then the song should skip to the appropriate position
  - As the user holds the seek bar down to move it, the song continues playing normally. The song position does not change as the user holds down the seek bar.
  - When a user clicks on a song or goes forward/back the song starts at the beginning.
  - You need to use the atomic Boolean userModifyingSeekBar to make sure that while the user is changing the seekbar, your displayTime coroutine is not also updating the seek bar. If you both try to update it, it will visually jitter between the two values.
- Text (to the left of the slider) showing how long a song has been playing in the format "MM:SS" (see convertTime)
- Text (to the right of the slider) showing how long until a song finishes in the format "MM:SS"
- There is a gear icon in the action bar. When pressed this opens the settings fragment.
  - There is a switch for "loop" mode, and there are two buttons, cancel and ok.
  - The values displayed for loop mode and the count of played songs should accurately reflect the current state when the fragment is displayed.
  - The Cancel button.
    - Pressing this goes back to the main activity, discarding any modifications.
  - Returning to the player via the Up button (upper left of action bar) discards user modifications to the loop mode.
  - Returning to the player via the phone back button discards user modifications to the loop mode.
  - The OK button.
    - Pressing this goes back to the main activity and uses any changes the user made to the loop mode.
  - If loop mode is enabled, press forward/backward button to play the next/previous song and then loop on that song.
- If a song is playing and the settings fragment is invoked, play should continue and follow the

normal rules of succession. By the normal rules, I mean the song after the current one starts when the current one finishes unless the player is in loop mode in which case the same song restarts.

- You will need controls for your app. We recommend Android's built-in vector graphics, which are very versatile, look good at any resolution and are easy to add to your project. You can see the course video on icons and images for more information. Once you add them to your project, they will have names like @drawable/ic_play_arrow_black_24dp and @drawable/ic_baseline_skip_previous_24dp.
- The music files are in res/raw directory.

Initialization

When the app starts it should not play music, but it should select the first song in the list.

When a song finishes playing the next song in the list should be played (unless loop is enabled). If the last song in the list is played then wrap around to the beginning and play the first song.

Layout

- You can use LinearLayouts or ConstraintLayouts.
- I use 8dp of margin for most layout elements.
- Center the (back/play/forward) controls and separate them by 20dp. Separate the buttons on the settings layout by 20dp.
- All drawable buttons are 50dp by 50dp, except the L which is width 30dp and height of 50dp. The L is 36sp.
- "Now playing", "Next Up", and "Number of songs played" are all bold, 20sp. The text following them is not bold and 15sp.
- Song titles are black and 20sp. Times are default color and 20sp.
- TextView time displays use the default font size and are 50dp in width.

In order for us to test you code, you need to use these IDs EXACTLY as written. Do not change the capitalization, do not change the spelling. If you do, our automated tests will fail and you will lose points and we will all be sad.

- TextView after "Now playing": `"@+id/playerCurrentSongText"`
- TextView after "Next up": `"@+id/playerNextSongText"`
- RecyclerView that holds the songs `"@+id/playerRV"`
- Player controls are ImageButtons with these IDs
  `"@+id/playerSkipBackButton"`
  `"@+id/playerPlayPauseButton"`
  `"@+id/playerSkipForwardButton"`
- TextView loop indicator: `"@+id/loopIndicator"`
- ImageButton permute button: `"@+id/playerPermuteButton"`

- TextView time passed display: `"@+id/playerTimePassedText"`
- SeekBar showing progress: `"@+id/playerSeekBar"`
- TextView time remaining display: `"@+id/playerTimeRemainingText"`

Settings

The settings fragment displays the number of songs played so far. It counts song starts while the player is playing. You don't need to listen to the whole song to increase the counter.

Settings has a switch element to control whether loop mode is on. The switch should be set correctly (i.e., if loop mode is on when we enter the settings mode, the switch will be on). The user can commit their changes by clicking ok. Whatever they set the loop mode to should be in effect. Every other way of returning to the player does not save their change. The user can return by hitting the "up" arrow in the upper left corner, they can hit the phone's back button, or they can click the cancel button. In all of these cases, the loop indicator should not change state.

Coroutines

We haven't said much about coroutines, though they are present in your first two programming assignments. We use one here for updating the time and seek bar. Coroutines are light-weight threads that are independent of the control flow in your app as they execute concurrently (at the same time). Coroutines are useful for when you need to do something like update the time display. Having code that just wakes up, sees what time it is and update the display is simple to write and to reason about.

We have included the code to launch and manage the coroutine, so no worries there. It calls the function `displayTime`, which has type "private suspend fun." The suspend qualifier says this function is callable from a coroutine.

The code in `displayTime` can read and write your programs state freely. Just go for it. You (mostly) do not need any synchronization between the main thread and the coroutine. The one exception is `userModifyingSeekBar`, which makes sure we give priority from `displayTime` to the user's action when the user moves the seek bar.

Files. Let's go over the files in the project. All locations where you need to write code are marked with `// XXX write me`. You can trust me, there won't be unmarked areas that you have to change. But you can always create helper functions.

- **AndroidManifest.xml** Do nothing here.
- **MainActivity.kt** You have to fill in some menu initialization code. This will look very similar to the demos.

  You also need to fill in `onSupportNavigateUp`, which also follows the demos closely.
- **MainViewModel.kt** Welcome to your view model! Here is where we will store most of our state. Having it here means we don't have to stress about whether the player fragment or the settings fragment is active.

  `currentIndex` holds the index of the currently playing song. The index refers to the songResources list.

  Most of the properties in the view model are public, so fragments can directly read and write them. It seemed like the most straightforward option.

  `getCopyOfSongInfo` we provide. It turns out, creating a mutable list from a list copies the list. That makes sense once you think about it for a while.

  `suffleAndReturnCopyOfSongInfo` is the only function in this class that is challenging to write. You need to make a copy of the `songResources` list and shuffle it. Then this new list has to replace `songResources`. You also need to return the new list. This function is responsible for the permute button.

  We discuss this more below, but the player has a permute button. The current song stays the same, but the entire list is randomly permuted, so there will likely be a different next and previous song after the permutation.

  The real head scratcher for permutation is dealing with `currentIndex`. Let's say our song list is `A B C`. `currentIndex` is 1, so B is the current song. Then we permute the list and now it is `A C B`. The current song is still B, but B isn't at index 1 anymore...

  The rest of the file is pretty straightforward.
- **PlayerFragment.kt**
  - `userModifyingSeekBar` will be used in your fragment code and in the coroutines, which is why it needs to be an AtomicBoolean. We will get to it later.

    Much of this code should be familiar. `setBackgroundDrawable` is a routine we provide, that you must use to set the background of an image button to a particular drawable resource identifier (such as `R.drawable.baseline_settings_24`, but that is wrong).

    You use `setBackgroundDrawable` when the background is a drawable, but you use `setBackgroundColor` when you want to set the background color. We also provide the local function `setBackgroundColor` which also sets the tag for our tests.
  - `updateDisplay` is where you should update most items in the view binding object. We put most of that code here because then if we are playing a song in the background, this function just returns because it has no view binding object when it is not visible.

    The only other functions that access the view binding object is `onViewCreated` and the `displayTime` coroutine.

- `onViewCreated` does the stuff you would expect, like it creates an adapter and sets on click listeners for the buttons. You need to set up the SeekBar.

  The seek bar's change listener (that you write) should set `userModifyingSeekBar` to let the rest of your code know when the user is touching the seek bar and potentially moving it. During that time (when `userModifyingSeekBar` is true), `displayTime` should not update the progress bar, because doing so will create weird looking visuals where the dot on progress bar jumps between where the user is dragging it and where it should be to mark the progress through the song. Preventing this conflicting update of the seek bar is the purpose of `userModifyingSeekBar`.

  An AtomicBoolean has `set` and `get` methods. That is all you need.

- (advanced topic) Check out how we launch the coroutine using `viewLifecycleOwner.lifecycleScope.launch`. The cool part about this is that the coroutine runs while the fragment is alive, and it is killed when the fragment gets killed. The coroutine itself only updates the view while it is active. When it is in the background, it is not active, so you don't need to worry about it accessing the view binding object.

- `displayTime`. In this routine you need to update the seek bar (I mean, update it if the user isn't modifying it), and update the time passed and time remaining.

- `convertTime` is worth writing and debugging separately to just get it right.

- I have a bunch of utility functions that do things like start and stop a media player. I also handle transitioning to the next song and replay, etc.

- Do your best to track `viewModel.songsPlayed`. There are many corner cases here. If you just click songs while the player is paused, please don't increment `songsPlayed`.

- **Repository.kt** No changes here. Our song information structure has a name, an integer resource identifier, a string indicating the time, and a unique identifier. That last one is needed to make sure that every member of the song resources list is unique.

  Maybe we should get the length of the songs from the files, but that seemed like too much complication.

- **RVDiffAdapter.kt** This is similar to the adapters we have done in the demo code. You initialize this with a view model and a `clickListener` callback function.

  To get the position in the list from the ViewHolder, just call `bindingAdapterPosition`. We will skip the `getPos` function that we have been using.

  class Diff is complete and correct.

- **SettingsFragment.kt** We give you most of this file. Check out the `initMenu` function. We clear the menu to prevent running recursive copies of the settings fragment. Groove to that funky `viewLifecycleOwner`. Adding that means the menu changes go away when the fragment dies.

- **activity_main.xml** All good, no changes.

- **player_fragment.xml** This requires the player layout. Lots of details. It might make sense to make a "rough and ready" version that doesn't adhere to all of our detailed specifications, and then refine it later. Having a functional player allows you to make progress on other portions of the assignment.

- **settings_fragment.xml** Please write the layout. I used nested linear layouts.
- **song_row.xml** Please write the layout. Let the images of the layout (seen above) be your guide.
- **menu/player_menu.xml** This is complete and correct, but you should look at it.
- **navigation/mobile_navigation.xml** This is complete and correct, but you should look at it. the player fragment is the start destination.

Hints

For the project you hand in, do not change the music in the Repository object or the order. If our music is driving you crazy, you can replace the songs during development and update the Repository.

I would suggest doing a "quick and dirty" layout that allows you to start on the more complex coding before fine-tuning the layout. But if you are good at layout, just have at it.

The MediaPlayer object has internal state and methods that can be called only in a specific state. Here are some useful URLs.
`https://developer.android.com/reference/android/media/MediaPlayer#state-diagram`
`https://developer.android.com/reference/android/media/MediaPlayer#valid-and-invalid-states`
We use it in a pretty simple way, so you shouldn't need all these details.

If you think clicking the L should change the loop mode and that going to a separate settings activity is silly, well, I have to say I agree with you. That partitioning of functionality is a bit pedagogical.

Work on one feature at a time. There are a lot of tiny elements in this project, many of them do not depend on each other.

When it comes time to "get rid" of a MediaPlayer, call reset() and then release(). Otherwise you will get errors about unhandled events.

We will press lots of buttons on your app to exercise all of its functionality.

This project will require a little bit of outside research. We do not cover SeekBars (for example) in class, nor do we cover MediaPlayers. But we cover the things you need to know to figure these things out on your own. Google, the Android documentation, Stack Overflow, and Ed discussion are your friends.