Assignment 2: Android Depth Of Field Calculator

- This assignment is to be done **individually or in pairs**. Do not share your code or solution with others who are not in your group, and do not copy code found online. Ask questions on Piazza discussion forum (see webpage) or in office hours.
  - You may use code provide by the instructor, or in help guides/videos provided by the instructor (such as my YouTube videos).
  - You may follow any guides you like online as long as it is providing you information on how to solve the problem vs a solution to the assignment.
    - If receiving help from someone, they must be teaching you not writing your code.
    - You may not resubmit code you have previously submitted for any course offering except your assignment 1 (which is expected to form the basis of your assignment 2's model).
  - If you copy a more than 4-5 lines of code from a guide/tutorial, it is a good idea to cite it in your code:
    `// Code taken from xyz.com/awesome/stuff/here.html`

1. Android App Overview

1. User can add (and possibly edit or remove) lenses in the list of known lenses.
2. User selects a lens and enters information about the photo they are taking. The app computes and displays the depth of field for the photo.

To learn Android programming you can use a book on Android (see course website for recommended book) or any online tutorials. The course website links a number of tutorials recorded for this course which cover many of the necessary topics.

2. Required Application Features

Implementing these **"Required Features" can earn a *good* grade**; to get a ***great* grade, you must also complete some "Optional" features** (listed later). See marking guide for details.

2.1 General Requirements

- Create an Android application targeting the minimum SDK version API 24 or lower.
- Use your assignment 1 solution as the basis for your model for this assignment.
  - You may edit your files any way you need to support your application's needs.
  - You will not need the text UI. Place assignment 1 `.java` files which you need in a `model` package inside the normal Android App's Java folder.
- Each activity should display a meaningful title. Do this in `strings.xml`.
- Screen-shots in this document are for inspiration. As long as your application correctly implements the required features, any nice and usable UI is fine.
- None of the things listed as "hints" are required; you may choose to do them or not.
- Activity files (`.java` and `.xml`) must be well named, but need not match this document.
- Create a GitLab repo on `csil-git1.cs.surrey.sfu.ca/`
  Commit your changes often (at least every 4 hours of work).
- You do not need to handle screen rotation.
- Use the Toolbar widget to give all activities an app bar (also called the action bar) at the top of all activities. Required buttons are:
  - Add Lens activity: Back and Save
  - Calculate activity: Back

- Hint:
  - Each time you create a new activity for your project, choose “Basic Activity”. This will then always give you the same file structure. Delete the floating action bar if not needed.

2.2 Screen 1: Lens List

- This is the initial activity displayed at startup.
- Display the lenses from your model.
  - For each lens, show:
    make, focal length, and maximum aperture.
- Use your solution to assignment 1 to store the lenses.
  - You may edit your code as needed.
  - Use the singleton design pattern[1] with your lens manager class. See video on website.
- Use a Floating Action Button (FAB) to allow the user to add a new lens to the collection by launching a new activity to enter lens details.
  - Change the icon on the FAB to be a +
  - See video on course website.
- User may tap on a lens in the list to launch the Calculate Depth of Field activity.
- When your app starts up, pre-populate the lens manager with the lenses shown in the screenshot (same as assignment 1).
- You must make your lens manager class a singleton; therefore, you’ll be able to access your model (collection of lenses) with code similar to:

```
LensManager lenses =
    LensManager.getInstance();
Lens lens = lenses.get(0);
```

Figure 1: Possible look of lens list activity

Hints

- Use a `ListView` or `RecyclerView` to show the list of lenses.
  - `ListView` is easier to use; `RecyclerView` is more modern and flexible but harder to use. I recommend just using the `ListView` unless you are looking for a challenge.
  - See website for tutorial on populating the list.
- After adding a new lens you’ll need to refresh your UI’s lens list. The simplest way to do this is to override your Lens List activity’s `onStart()` method and have it either reinitialize, or call `notifyDataSetChanged()` on, the `ListView` adapter.
- For how to pass data to another activity, see hints for the other activities.
- Avoid duplicate code: Extracting duplicate code to a function can help you better understand your code structure.

1 The Singleton pattern allows one instance of your model to exist the lifetime of your application. Android activities come and go depending on a number of things, such as rotating the screen. If your Activity just held onto the `LensManager` object, then it would be destroyed every time the screen rotated and be inaccessible when you switched to a new activity. The singleton allows all activities to access the same instance of your model (`LensManager`).

2.3 Screen 2: Add Lens

- Allows user to enter the lens's required values.
  - If using `EditText` widgets for data input, each must have a hint for what goes in it (such as show in UI to the right).
  - May use other data entry widgets if desired.
- For focal length entry, only allow non-negative integer values.
- For aperture entry:
  - If using a text entry box, only allow non-negative floating point values; no upper bound.
  - If using a drop-down box then check online for most common apertures.
- Add the following to the taskbar:
  - left arrow ("Up") to cancel
  - "Save" to save

Figure 2: Possible look of add lens activity.

Hints

- Convert a `String` to an `int` or `double` with:
  ```
  int x = Integer.parseInt("200");
  double y =Double.parseDouble("2.8");
  ```
  - Note that if the `String` does not contain a number it throws `NumberFormatException`.
  - You need-not handle validating the user's input. If user does not enter any values and clicks OK, your application may crash; this is fine. See Optional Features section for fixing this issue.
- When adding a new lens, you must refresh your Lens List activity's list. See tip in previous section related to `onStart()`.

Requirement on Launching and Passing Data to an activity

- Data is passed from one activity to another using an `Intent`. See video: https://www.youtube.com/watch?v=SaXYFHYGLj4
  - *Watch the video!* It shows the **required** way of creating an intent and respecting encapsulation; and it's testable!
  - For this app we can add the new lens directly into the model (via the singleton); therefore, we don't need to return a result from the activity.

2.4 Screen 3: Calculate Depth of Field

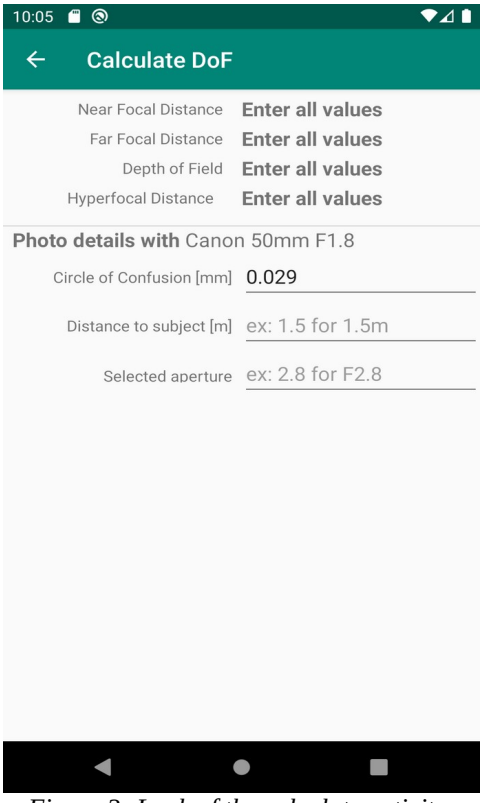

Figure 3: Look of the calculate activity.

- Display the selected lens's description.
- Allow user to enter:
  - camera's circle of confusion (pre-populated to 0.029)
  - distance to subject (in meters)
  - selected aperture (the F number)
- UI must allow only non-negative decimal values.
- Calculate and display the four depth of field values.
- Similar to Add Lens, you must use a `public static` method to encapsulate creating the `Intent` to launch this activity; see hint below.
- If the entered aperture is less than the lens's maximum aperture, display an error message such as "Invalid aperture". See screen shot on next page.
- If the user enters a circle of confusion of 0, it's OK to display "NaN" (not a number), or a message to enter all values, or a message that there is an error.
- In the taskbar, add the left arrow ("up") to leave the screen and return to the Lens List activity.
- You may either have a "Calculate" button which computes the values and updates the UI. Or, as described in the "Features you can Select to Implement" section, you can have it automatically calculate when the user changes the input data (as shown in the screen-shots).

Hints

- Pass data to the Calculate activity using an `Intent`.
  - Pass in the lens's index for accessing it via the `LensManager` (the singleton!)
  - For good encapsulation, have the Calculate activity expose a method which creates the intent to start it. Pass this function the data to be encoded into the `Intent`.

```
public static Intent makeLaunchIntent(
                                Context context, int lensIdx);
```

- If trying to display a number in a `TextView`, note that

```
myTextView.setText(42);
```

will attempt to load the `strings.xml` resource which has number 42 into the `TextView`, which likely does not exist and will crash your program. Instead, convert the into to a `String`:

```
myTextView.setText("" + 42);
```

- Here are some more screenshots showing what your UI might look like when the user has calculated all values, or entered an invalid aperture.

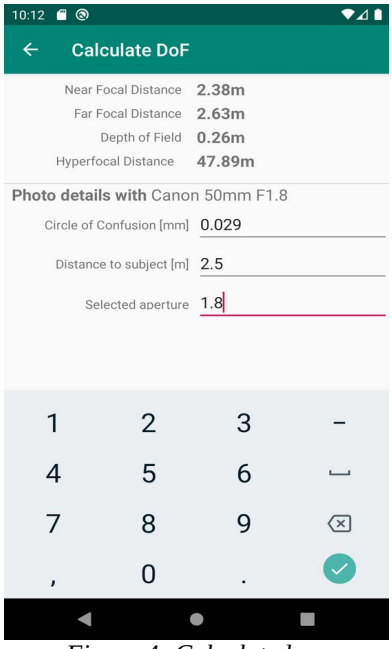

Figure 4: Calculated

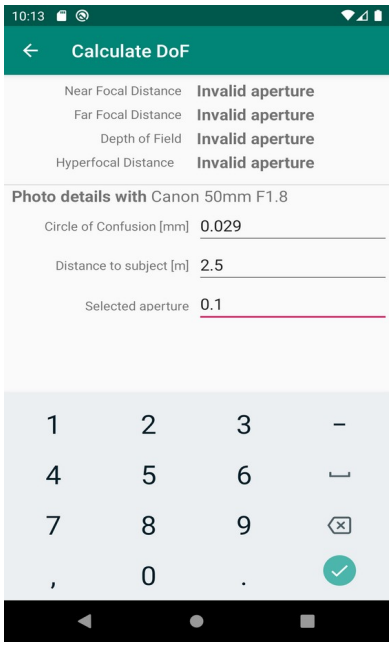

Figure 5: Invalid Aperture

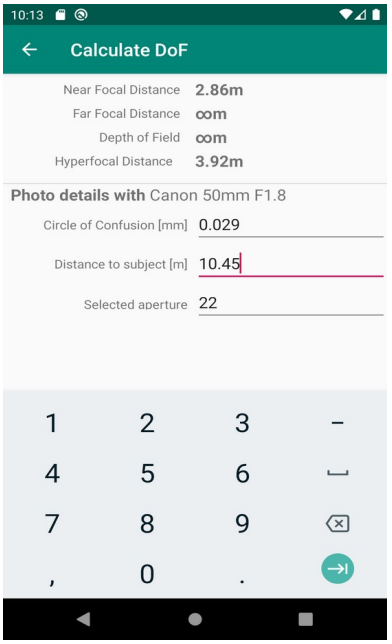

Figure 6: Infinite DoF

3. Features you can Select to Implement

By completing one or more of these features, you stand to move from "Good work" to "Great work". See marking guide.

You may only get marks for the features listed here if the required parts of the application work well (need not be perfect).

If you attempt any of these features, your Lens List activity must state the features you added.

- List the feature number and title.
  - For example, have a `TextView` at the bottom of the screen listing the optional features you completed.
  - *Hint*: Enter text into the `TextView` in one line like (`\n` for linefeed):
    `Optional Features:\n1.App Bar Buttons\n3. Error Checking Inputs\n4.Auto-recalculate`
- You may also briefly mention how to access the feature (if not clear from your UI).

You may attempt any of these features, in any order.

3.1 Edit and Delete Lens

- Support editing a lens stored in the list of lenses.
- Support removing a lens from the list of lenses.
- Must have needed buttons in the Toolbar.
- Hints:
  - From the Calculate activity, add a button to edit the selected lens.
  - Re-use the Add Lens activity and just pass it extra data (via an Intent) of which lens is to be edited.
  - From the Calculate activity, add a button to delete the selected lens.
  - Make sure that you update the Lens List activity's list view when the model changes.

3.2 Error Checking Input

- Enforce at least the following constraints on user input:
  - Add Lens activity (and edit if supported):
    - Make's string length is > 0
    - Focal length > 0
    - Aperture >= 1.0 and aperture <= 22
  - Calculate activity:
    - Circle of Confusion must be > 0
    - Distance to subject > 0
    - Selected aperture within range listed above
- When you detect an error:
  - Display a good error message. If appropriate, you may use a toast.
  - In the case of Add Lens (or edit), prevent saving the lens until all values are valid.
  - In the case of Calculate, only calculate once all values are valid.

3.3 Auto-recalculate

- On the Calculate activity, automatically recalculate all depth of field values when the user changes any of the input fields.
- Remove any redundant buttons from the UI.
- Hint:
  - To recompute while the user is entering data, you'll need to pass a `TextWatcher` object to the `addTextChangedListener(...)` method of each of the input `EditText`.
  - In this `TextWatcher.afterTextChanged(...)` call your code to recompute the depth of field values; other methods in `TextWatcher` can be left untouched.
  - You have three input `EditText`s: you can create one `TextWatcher` object and then pass it to each of the `EditText`s to reduce code duplication.

3.4 Save Data

- Save all the lenses between executions of your application.
- When your app starts up, if there are no stored lenses add the sample lenses from Assignment 1. However, if there are saved lenses then don't add these sample lenses.
- Hints:
  - You may want to use `SharedPreferences`.
  - You may want to edit your lens manager class to support working with a `SharedPreference`.
  - You may use external serialization libraries if you like (Gson/JSON/....).

3.5 Lens Icons

- Significantly enhance the user interface by allowing the user to set an icons or images for each lens. You may, for example, have 5 built-in icons the user can choose between.
- Change the lens list activity to use a complex layout with the lens's icon/image and text.
- See video on course website related to making complex list views.

3.6 Empty State

- When your application has no lens to show, display a nice looking message on the main screen instead of the list of lens.
- The message must give the user some directions on how to start creating a new lens.
- Add an image, a nice multi element layout, … (not just one `TextView`)
- If you do this, you must also have the ability to delete lenses, otherwise since the list must be pre-populated the TA will be unable to test it.

4. Deliverables

To CourSys (https://courses.cs.sfu.ca/) you must submit:

1. **ZIP file of your project, as per directions on course website.**

   If you worked individually, you will need to create a group in CourSys which consists of just you before you can submit your ZIP file.

   If you worked in a pair, you will need to create a group in CourSys and invite your partner. Please ensure your partner accepts the invitation so that everyone gets credit for the work.

2. **URL and Tag for your Git repository:**
   1. Add the TA for the course as a "Developer" member of your repo:
      - Goto csil-git1.cs.surrey.sfu.ca and select your project
      - On the left hand side, click the cog-wheel drop-down ('Settings')
      - Select "Members"
      - Add both TA to your repo as a **Developer**. TA SFU IDs = **srchauha cprabhu**

   2. Create a tag for your submission as follows:
      - In Android Studio, go to VCS --> Git --> Tag...
      - Enter a name for your tag, such as: `final_submission`
      - Leave Commit and Message blank.
      - Click Create Tag
      - Push changes to remote repo. On "Push Commits" dialog, select **"Push Tags: All"**.
      - You can check the tag was pushed correctly in GitLab online.
      - (If you resubmit, create a new tag as above and submit the new tag via CourSys).

   3. Submit the git@... URL and tag name to CourSys
      - Find Git URL on csil-git1.cs.surrey.sfu.ca/. Should be similar to: git@csil-git1.cs.surrey.sfu.ca:yourid/myProjName.git
      - The "tag" is the name you used above, such as "`final_submission`"

Please remember that all submissions will automatically be compared for unexplainable similar submissions. Everyone's submissions will be quite similar, given the nature of this assignment, but please make sure you do your own original work; we will still be checking.