Lab 12 :- To implement demonstration of dialog box and toolbar

Aim:

Procedure:

Designing Dialog box

The Dialog class is the base class for dialogs, but you should avoid instantiating Dialog directly. Instead, use one of the following subclasses:

AlertDialog:A dialog that can show a title, up to three buttons, a list of selectable items, or a custom layout.

DatePickerDialog or TimePickerDialog: A dialog with a pre-defined UI that allows the user to select a date or time.

Creating a Dialog Fragment
You can accomplish a wide variety of dialog designs—including custom layouts and those described in the Dialogs design guide—by extending DialogFragment and creating an AlertDialog in the onCreateDialog() callback method.
For example, here's a basic AlertDialog that's managed within a DialogFragment:

```
class StartGameDialogFragment : DialogFragment() {

  override fun onCreateDialog(savedInstanceState: Bundle): Dialog {
  return activity?.let {
  // Use the Builder class for convenient dialog construction
   val builder = AlertDialog.Builder(it)
       builder.setMessage(R.string.dialog_start_game)
       .setPositiveButton(R.string.start,
       DialogInterface.OnClickListener { dialog, id  ->
        // START THE GAME!
       })
       .setNegativeButton(R.string.cancel,
       DialogInterface.OnClickListener { dialog, id ->
        // User cancelled the dialog
        })
         // Create the AlertDialog object and return it
         builder.create()
        } ?: throw IllegalStateException("Activity cannot be null")
    }
}
```

Figure 1. A dialog with a message and two action buttons.

Now, when you create an instance of this class and call show() on that object, the dialog appears as shown in figure 1.

Building an Alert Dialog

The AlertDialog class allows you to build a variety of dialog designs and is often the only dialog class you'll need. As shown in figure 2, there are three regions of an alert dialog:

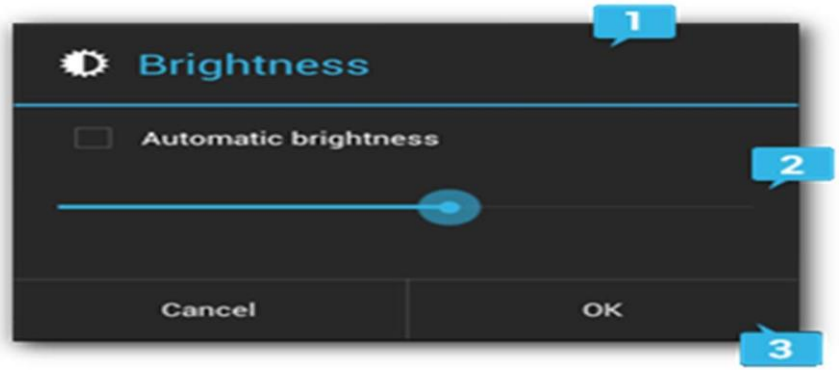


Figure 2. The layout of a dialog.

1. Title
2. Content area
3. Action buttons

The AlertDialog.Builder class provides APIs that allow you to create an AlertDialog with these kinds of content, including a custom layout.

To build an AlertDialog:

// 1. Instantiate an <code>

<a
href="/reference/android/app/AlertDialog.Builder.html">AlertDialog.Builder</
a></code> with its constructor

```
val builder: AlertDialog.Builder? = activity?.let {

   AlertDialog.Builder(it)

}


// 2. Chain together various setter methods to set the dialog characteristics

builder?.setMessage(R.string.dialog_message)

        .setTitle(R.string.dialog_title)


// 3. Get the <code><a
href="/reference/android/app/AlertDialog.html">AlertDialog</a></code>  from
<code><a
href="/reference/android/app/AlertDialog.Builder.html#create()">create()</a><
/code>

val dialog: AlertDialog? = builder?.create()
```

Adding buttons

To add action buttons like those in figure 2, call the setPositiveButton() and setNegativeButton() methods:

```
val alertDialog: AlertDialog? = activity?.let {

    val builder = AlertDialog.Builder(it)

    builder.apply {

        setPositiveButton(R.string.ok,

                DialogInterface.OnClickListener { dialog, id ->

                    // User clicked OK button

                })

        setNegativeButton(R.string.cancel,

                DialogInterface.OnClickListener { dialog, id ->
```

```
                // User cancelled the dialog
            })
    }
    // Set other dialog properties
    // Create the AlertDialog
    builder.create()
}
```

The set...Button() methods require a title for the button (supplied by a string resource) and a DialogInterface.OnClickListener that defines the action to take when the user presses the button.

There are three different action buttons you can add:

Positive

You should use this to accept and continue with the action (the "OK" action).

Negative

You should use this to cancel the action.

Neutral

You should use this when the user may not want to proceed with the action, but doesn't necessarily want to cancel. It appears between the positive and negative buttons. For example, the action might be "Remind me later."

You can add only one of each button type to an AlertDialog. That is, you cannot have more than one "positive" button.

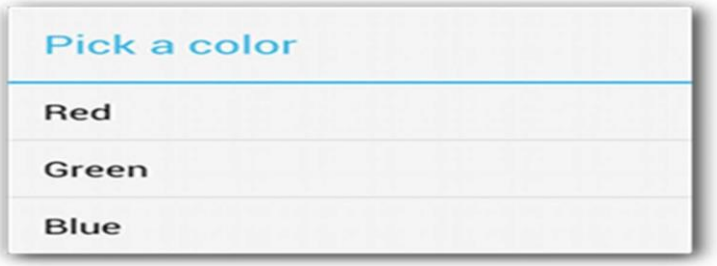


Figure 3. A dialog with a title and list.

Adding a list

There are three kinds of lists available with the AlertDialog APIs:

- A traditional single-choice list
- A persistent single-choice list (radio buttons)
- A persistent multiple-choice list (checkboxes)

To create a single-choice list like the one in figure 3, use the setItems() method:

```
override fun onCreateDialog(savedInstanceState: Bundle?): Dialog {
    return activity?.let {
        val builder = AlertDialog.Builder(it)
        builder.setTitle(R.string.pick_color)
                .setItems(R.array.colors_array,
                        DialogInterface.OnClickListener { dialog, which ->
                            // The 'which' argument contains the index position
                            // of the selected item
                        })
        builder.create()
    } ?: throw IllegalStateException("Activity cannot be null")
}
```

Because the list appears in the dialog's content area, the dialog cannot show both a message and a list and you should set a title for the dialog with setTitle(). To specify the items for the list, call setItems(), passing an array. Alternatively, you can specify a list using setAdapter(). This allows you to back the list with dynamic data (such as from a database) using a ListAdapter.

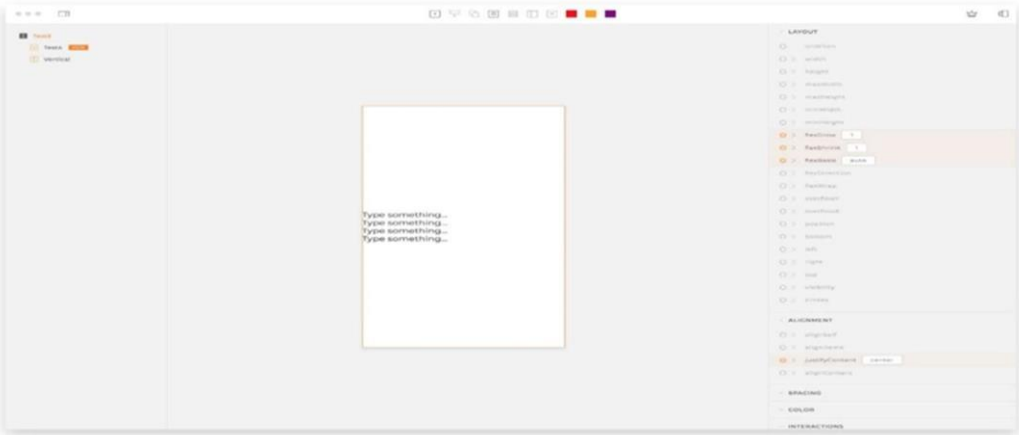
If you choose to back your list with a ListAdapter, always use a Loader so that the content loads asynchronously. This is described further in Building Layouts with an Adapter and the Loaders guide.

2. Designing Toolbar

Stage 1. Priming

Priming stands for the prime (first) version of the interface, like prototyping, but in Views Tools, the result is production quality code.
we move all the actions to the Top Bar.view component. Here's the result of our Priming work. At this point, the functionality is in with a default interface representation.

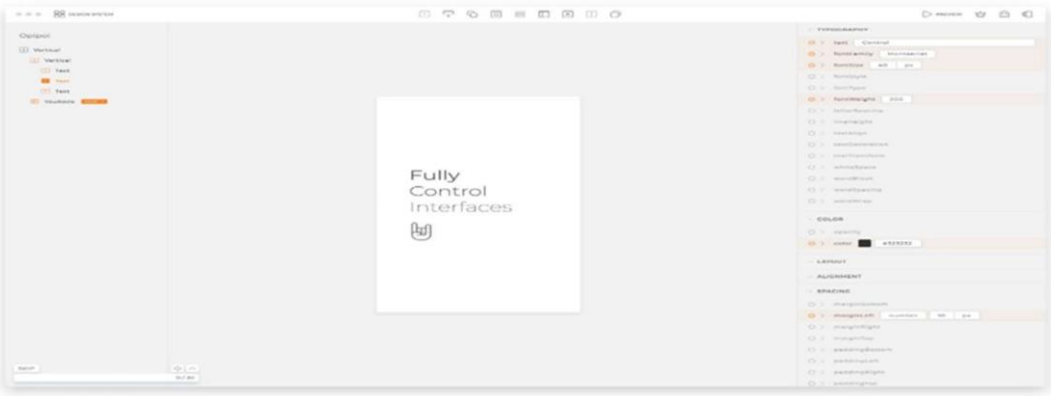
The result after Priming stage
Stage 2. Composition
We need to move buttons around, put back the window controls where they belong, center the block related tools, and remove the label from the Share Feedback button to make everything clean and consistent.



The result after playing with composition

Stage 3. Styling

On to the beauty! First, we need to replace the Views logo with a bit more obvious icon symbolizing the Design System window. There are few icons we need to make from scratch.



The result after playing with styles

Result:-

Lab 13 :- To demonstrate games and customizing controller buttons

Aim:

Procedure :

Demonstration of games:

Game UI designs refer to the user interface that game players first see when they enter a web or mobile app game. They work as the bridge to connect the game and the players together, making it easy for players to quickly understand the gameplay mechanics, find the right information, and start playing as soon as possible.

The better the design your game has, including better storytelling, high-quality animations, characters graphics, gameplay mechanics and user experience, the longer players are likely to play. The more immersive your game's UI is, the more likely players will make a purchase.

In total, a game's UI is a major factor for a potential customer and the better the UI, the more likely your revenue will increase.

10 of the best practices that can make your game UI designs shine:

i. Make players understand your game naturally

ii. Guide players using the right method at the right time

iii. Allow players to skip things

iv. Make your rewards and in-app purchases stand out

v. Never clutter game UI

vi. Create a clear visual hierarchy

vii. Make accessible navigation, like sidebar

viii. Make your game responsive

ix. Enable gamers to give feedback

x. Test your game UIs

2. Customizing controller buttons:

Customizing game's buttons is an important feature that can make gaming sessions more comfortable, make you a better gamer, and open the experience to new players.

i. In the toolbar, select Button.

ii. Drag the button images onto the Editor Canvas.

iii. Prepare the canvas (optional): Change the Grid size:

- Deselect any elements in the tree.
- In the Properties section, Canvas panel, adjust the Grid size to 10 pixels

iv. Prepare the canvas (optional): Set the Zoom to 200%.

v. Next, move the buttons into an arrangement in the canvas

vi. Once they are arranged, you might find it handy to be able to move them all at once. This is the Container's job. Draw a Container around all the buttons:

- Select the Container button in the Toolbar.
- In the canvas, draw around the buttons, until all have a red border.
- The tree will have a container with all the buttons a child of it.
- Select the container in the tree and then move the buttons in the canvas and it will move the buttons as one.

vii. Anchor the Container to the bottom center of the window (or wherever you plan for your buttons to be positioned on the skin):

- With the Container selected in the Tree, open on Position panel in the Properties section.
- Select the Anchor position.

viii. Add a second button state or more

ix. Add the appropriate action to each button image:

- Select a button in the Tree or Editor.
- In the Properties section, click on Actions.

- Double-click or click on the plus to the right of the table, to open the Action Settings.
- For a Tilt Down button, Choose Mouse Click for the Source and Tilt Down for the Action. Click, OK, when finished.
- Repeat this process for each button.

x. Test the skin with the Live Preview.

xi. Finally, click Close and save the skin (to the Skins Directory).

Output:-