Android Menus & Dialogs

Menus

For each activity you have a menu xml file connected to it. It will be located in the “res/menu” folder. This is what called “Action menu”. When using the ActionBarActivity(or the support library AppCompatActivity) this is the menu that will be shown on top for each activity, and will hold the activity's name, settings and more.

To add a menu just right click on the res/menu folder and pick a name for the xml file.

Creating an item

inside this menu xml file you can add “items”.

Starts with the tag “<item>” inside this tag you must add an id , a title, orderInCategory and showAsAction..

orderInCategory - allows you to order the specific item by the size the number inside (putting “2” there will put the item second in the list), the items goes by ascending order from the right to the left.

showAsAction - this must start instead of with “Android:” with “App” and this indicates if the item will be visible in the header of the app, if not the three dots will be shown and the item will be visible by clicking on them, recommended value will be “ifRoom” which means show them only if there is a space left in the action bar. **app:showAsAction="ifRoom"**

creating actions on the added items

To do so you must go to the activity’s java file and override the Activity **onCreateOptionsMenu** function, inside this you must add a call to the static function “**getMenuInflater().Inflate(R.xmlFileName,menu)**”. This line inflates the correct xml you

Written By: Eran Katsav - All Rights Reserved ©

created earlier to the menu of this activity (the second param - menu, refers to the argument passed to the function - where to inflate the xml file to).

After this you must override the function **onOptionsItemSelected**. There you can connect each item ,using their id's to the action you want to perform. You get the id for the item passed to the function using the getItemId function of the MenuItem. Mostly this is done by using the Switch - Case mechanism, where the id is the changing variable representing the menu button clicked and taken from the xml and you decide on the action for each one.

Note After taking care of a certain press of an item you must return true to let the app know that this action has been taken care of (**consumed**). Additionally the default return should be at the bottom of the function and should be false which will cause the app to deal with the press in the default way.

Context menu

Context menu is the options menu you receive when you long click on a widget (Button, toggle,TextView etc.)
To do this you must follow the steps:

1. Create a new menu xml file(also in the res/menu folder) and add <item> tags for each option in the menu, give it an id and title.
2. Go to the chosen activity java file and get a reference to the widget you would like to add a context menu to.
3. Use the function "**registerForContextMenu**(widgetRef)" this will connect the widget to the context menu.
4. Override the functions "**onCreateContextMenu**" and the "**onContextItemSelected**".
5. inside the "**onCreateContextMenu**" use the function argument **view** get the view's id and according to it inflate using the function **getMenuInflater.inflate(xml,menu)** the proper xml file to each widget. In other words: f you have a few widgets with different context menus you must first check the view's id (the view that was clicked is passed to the function) and decide which xml to inflate.
6. Then inside the "**onContextItemSelected**()" do the same way as the "onOptionsItemSelected" function, check action's id and do the desired code.

Written By: Eran Katsav - All Rights Reserved ©

Context menu and contextual action mode

The contextual action mode displays action items that affect the selected content in a bar at the top of the screen and allows the user to select multiple items.
When a user enables this mode by selecting an item, a contextual action bar appears at the top of the screen to present actions the user can perform on the currently selected item(s). While this mode is enabled, the user can select multiple items (if you allow it), deselect items, and continue to navigate within the activity (as much as you're willing to allow). The action mode is disabled and the contextual action bar disappears when the user deselects all items, presses the BACK button, or selects the Done action on the left side of the bar.

To use this feature do the following steps:

1. Get a reference to the view and call **startActionMode**(callback) on it.
2. Pass it an **ActionMode.Callback** interface implementation.
3. In this implementation inflate the necessary xml menu in the **onCreateActionMode**() function and get the clicks in the **onActionItemClicked**() function.

Note: If you have a collection of items in a ListView or GridView and want to allow users to perform batch actions, you should:

1. Implement the AbsListView.MultiChoiceModeListener interface and set it for the view group with setMultiChoiceModeListener(). In the listener's callback methods, you can specify the actions for the contextual action bar, respond to click events on action items, and handle other callbacks inherited from the ActionMode.Callback interface.
2. Call setChoiceMode() with the CHOICE_MODE_MULTIPLE_MODAL argument on the listview instance.

Creating Popup Menu

A PopupMenu is a modal menu anchored to a View. It appears below the anchor view if there is room, or above the view otherwise. It is usefull for providing several actions related to one action menu item. If you define your menu in XML, here's how you can show the popup menu upon a view click (in the OnClickListener):

Written By: Eran Katsav - All Rights Reserved ©

1. Instantiate a PopupMenu with its constructor, which takes the current application Context and the View to which the menu should be anchored.
2. Use MenuInflater to inflate your menu resource into the Menu object returned by PopupMenu.getMenu().
3. Call PopupMenu.show().

To get the clicks events simply use the PopupMenu instance setOnMenuItemClickListener method and pass it an PopupMenu.OnMenuItemClickListener implementation, there check the id and that's it.

Dialogs

Dialogs is a form of pop up that shows a specific information that requires the user immediate attention. It is shown to the user in front of the activity and can be forced to not be cancelled until the user interact with it, for example low battery notice.

The kind of dialog we have:

- Alert Dialog(confirm/Yes No) - alerts the user with options.
- Multichoice Dialog(list or checkboxes)
- Custom Dialog
- Date Picker Dialog
- Time Picker Dialog
- Progress Dialog

Old way to create a dialog (deprecated)

Use the function “**showDialog**(int)”. The function receives a static final number (any number but must be different for every dialog) then override the function “**onCreateDialog**” in the activity. Inside it, use a “switch/case” of each dialog and create them (you could also override the onShowDialog, onDissmissDialog, etc).
In the old way, the the activity was in charge of managing the dialog. Both the showDialog and the onCreateDialog functions are deprecated, **and the new android way wants us to separate**

Written By: Eran Katsav - All Rights Reserved ©

the dialog management from the activity, this is because **Fragment** have taken their place, and instead we just need to create and show them at the same time.
It is important to understand that dialogs are not deprecated but are created, customized and shown to the user at the same time, if we need more control over the dialog we should use the DialogFragment class we will learn about later on.

Create, customize and show the dialog(AlertDialog)

For each dialog you must first create a builder and pass the context (“AlertDialog.Builder builder = new AlertBuilder(this)”). After that you can use the builder's function to populate the dialog with a title (“builder.setTitle()”), a message (builder.setMessage()”), or buttons listeners (setPositiveButton/negative,neutral).
After all this you use the “.create” on the builder (“AlertDialog dialog = builder.create()”), and “.show()” on the newly created AlertDialog. We can also add a listeners to the dialogDissmiss event. Each function call return a reference to the dialog so we can chain the calls like this:

```
new AlertDialog.Builder(MainActivity.this).setTitle("Confirm delete").setMessage("Are you sure you want to delete the file?").setPositiveButton("Yes I am sure", new MyListener())
.setNegativeButton("NO I don't want", new MyListener()).setNeutralButton("????", new MyListener()).setCancelable(false).create().show();
```

In this way we are creating functions on the builder (and creating it ‘on the fly’).
Note: **setCancelable**(false) means that the user must interact with the dialog and cannot dismiss it by pressing outside of it.
MyListener is a **DialogInterface.OnClickListener** implementation which is the special listener for dialog clicks, inside you must override the onClick function.

MultiChoice Dialog

Instead of the **setMessage()** function we use the **setItems**(string[], **Listener**) function.
Where the first parameter is an array of String - the choices, and second parameter is a

Written By: Eran Katsav - All Rights Reserved ©

listener for the items click. When an item will be clicked the listeners onClick function will be called with the position(the index in the array) as a parameter.

Note: You can also use the **setMultiChoiceItems** which accepts three params: the first is the String array of options, the second (optional) is a boolean array of the checked items and then third is a listener like before, this will show a multichoice dialog with checkbox next to each item and the initial marking (which is marked and which isn't) is determined by the boolean array you passed(second arg).

Note: you can add String array in the string.xml file with the <string-array> tag and add <item> into it. then get the String array reference with the getResources().getStringArray() functions and pass the name of the <string-array>

AlertDialog with Custom view

To use a custom view follow these steps:

1. Create an xml file with the desired layout and widgets - the dialog's custom view.
2. inflate the view (xml file) using: View v = getLayoutInflater().inflate(R.layoutFileName, null); (the null arg is because we get the inflated view ref as a returned value)
3. Get the desired references for the widgets with the v.findViewById function. You can use it for example to get the text from an EditText object in the custom view.
4. Create a builder like before (AlertDialog.Builder builder = new AlertDialog.Builder(context);
5. Then to set the custom view use the "**setView**()" on the builder with the inflated view ("builder.setView(v)")
6. Do the necessary customization, create the dialog and show it.

Note! Additionally you can add buttons to these dialogs with the "setPositiveButton" (positive, negative etc.) so in this case you can add ".setPositiveButton("button name", new DialogInterface.OnClickListener() {}"), like before, meaning that you can use a custom view with the regular dialog's buttons.

Written By: Eran Katsav - All Rights Reserved ©

Time \ Date pickers dialogs

To create the regular android system date and time pickers we need to create an instance of the **DatePickerDialog**,**TimePickerDialog** classes. both the constructors get 3 args: The context, An implementation of the **DatePickerDialog.OnDateSetListener** or the **TimePickerDialog.OnTimeSetListener** which override the **onDateSet** or **onTimeSet** functions and are called when the user picked the date or time, and the last parameter to the constructors are the current date or time to be shown the the user when the dialog first populate. After creating the instance just call **show**().

Progress Dialog

The progress dialog designed to show an action in progress to the user and give him a progress indication. You can create two styles of progress bars, the simplest one is the STYLE_SPINNER intended for processes who cannot be measured (indeterminate) - it show a spinning wheel with no progress indication(the wheel can spin forever) and the second one is the STYLE_HORIZONTAL which is a progress bar to show the exact progress indication for determinate processes. You can define the progress style with "**setProgressStyle**()", and can set initial progress for the "STYLE_HORIZONTAL" with "**setProgress**(int)" and the maximum number - the end of the progress with "**setMax**(int)". After doing this you can update the progress during process with "**incrementProgressBy**(int)"

Updating UI from background thread - Handler

IMPORTANT: Any change in the app UI has to be done in the **main thread**. So if the process is done on a background thread, the ".incrementProgressBy(int)" will need to be sent to the main thread of the app. Each Handler instance is associated with a single thread and that thread's message queue. When you create a new Handler, it is bound to the thread that creating it -- from that point on, it will deliver runnables (pieces of code) to that thread and execute them.

Written By: Eran Katsav - All Rights Reserved ©

There are two main uses for a Handler: (1) to schedule messages and runnables to be executed as some point in the future; and (2) to enqueue an action to be performed on a different thread than your own - This is our porpuse. **To do that follow these steps:**

1. Create a handler (“Handler handler = new Handler();”) in the main thread.
2. When you want something done on the main thread, you can use the function “**handler.post()**” inside a background thread to send messages to the main thread.
3. The handler’s post function receive a Runnable implementation and executes the override run() method on the thread where that handler was created.
4. you also have the **postDelayed** handler’s function which gets a Runnable instance and also a time in milliseconds to execute it (the first handlers use mention above).

More on That in the Background tasks with UI update on AsyncTask lesson.

Reference

Android menus in the Android developer guide

Written By: Eran Katsav - All Rights Reserved ©