Andhra Pradesh State Skill Development Corporation

Skill AP
APSSDC

eSkill AP
Learn Anytime Anywhere
Andhra Pradesh State Skill Development Corporation

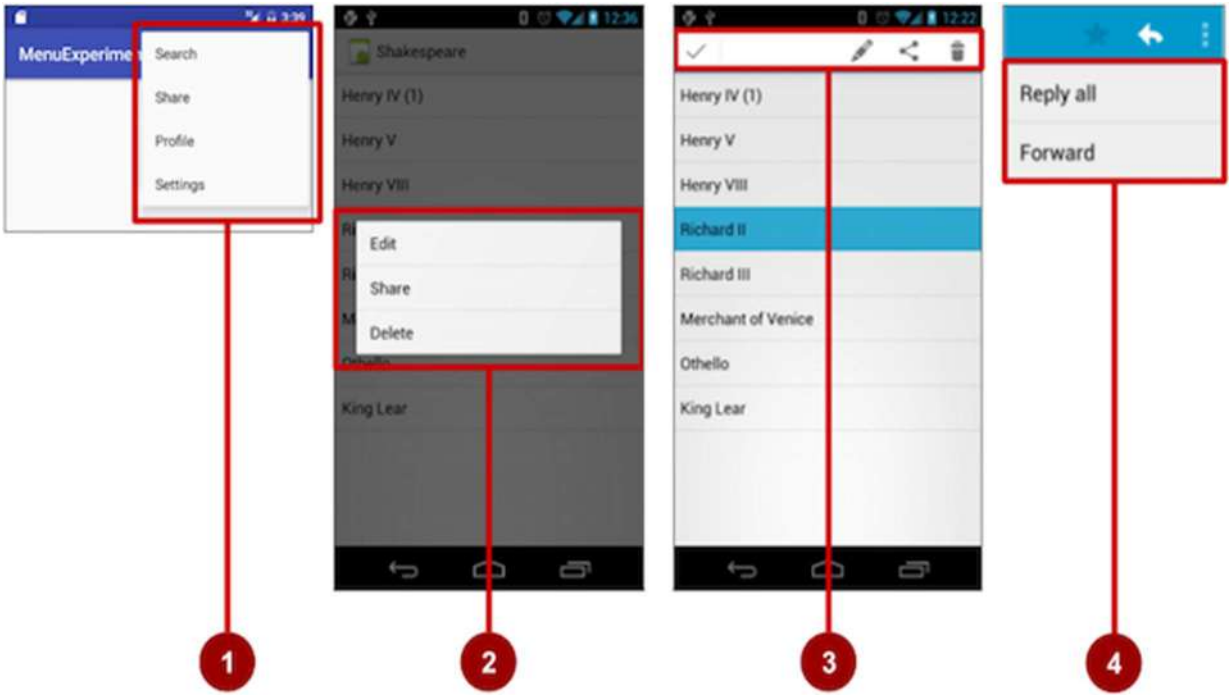
ANDROID APPLICATION DEVELOPMENT

MENUS

Menus

Types of Menus

A menu is a set of options. The user can select from a menu to perform a function, for example searching for information, saving information, editing information, or navigating to a screen. The figure below shows the types of menus that the Android system offers.



1. **App bar with Options menu:** Appears in the app bar and provides the primary options that affect use of the app itself.
   - Examples of menu options:
     - Search to perform a search
     - Share to share a link
     - Settings** to navigate to a Settings *Activity*.
2. **Contextual menu:** Appears as a floating list of choices when the user performs a long tap on an element on the screen.
   - Examples of menu options:
     - Edit to edit the element
     - Delete to delete it
     - Share to share it over social media.
3. **Contextual action bar:** Appears at the top of the screen overlaying the app bar, with action items that affect the selected element or elements.
   - Examples of menu options:
     - Edit, Share, and Delete for one or more selected elements.
4. **Popup menu:** Appears anchored to a View such as an ImageButton, and provides an overflow of actions or the second part of a two-part command.
   - Example of a popup menu:

– the Gmail app anchors a popup menu to the app bar for the message view with Reply, Reply All, and Forward.

App bar with options menu

- The app bar (also called the action bar) is a dedicated space at the top of each ***Activity*** screen. When you create an ***Activity*** from a template (such as Empty Template), an app bar is automatically included for the ***Activity***.
- The app bar by default shows the app title, or the name defined in ***AndroidManifest.xml*** by the ***android:label*** attribute for the ***Activity***. The app bar may also include the Up button for navigating up to the parent activity. Up navigation is described in the chapter on using the app bar for navigation.
- The options menu in the app bar usually provides navigation to other screens in the app, or options that affect using the app itself. (The options menu should not include options that act on an element on the screen. For that you use a contextual menu, described later in this chapter.)
- For example, your options menu might let the user navigate to another activity to place an order. Or your options menu might let the user change settings or profile information, or do other actions that have a global impact on the app.
- The options menu appears in the right corner of the app bar. The app bar is split into four functional areas that apply to most apps, as shown in the figure below.

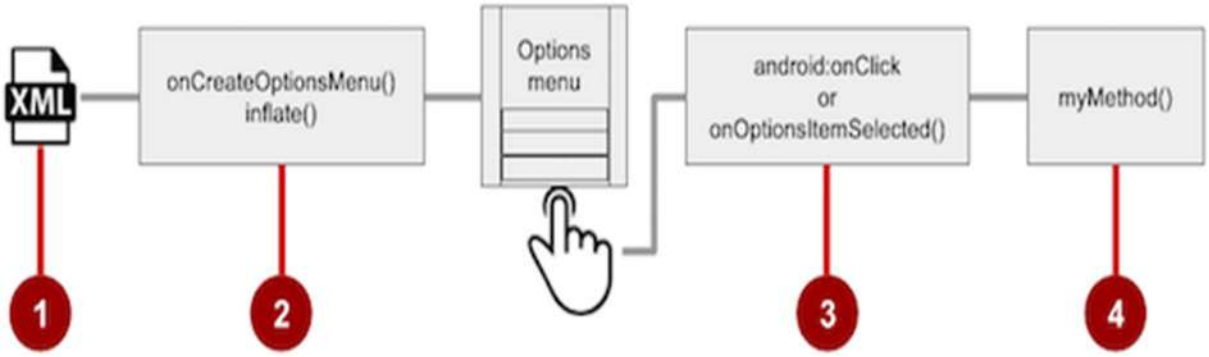
In the figure above:

1. ***Navigation button or Up button:*** Use a navigation button in this space to open a navigation drawer, or use an Up button for navigating up through your app's screen hierarchy to the parent activity. Both are described in the next chapter.
2. ***Title:*** The title in the app bar is the app title, or the name defined in ***AndroidManifest.xml*** by the **android:label** attribute for the activity.

3. ***Action icons for the options menu:*** Each action icon appears in the app bar and represents one of the options menu's most frequently used items. Less frequently used options menu items appear in the overflow options menu.
4. ***Overflow options menu:*** The overflow icon opens a popup with option menu items that are not shown as icons in the app bar.

Adding the options menu

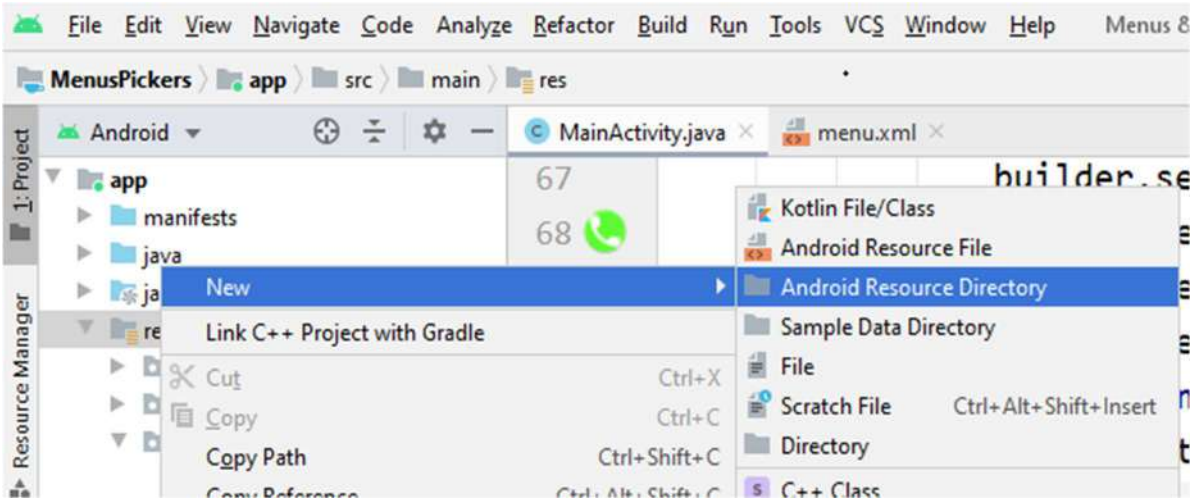
- Android provides a standard XML format to define options menu items. Instead of building the menu in your ***Activity*** code, you can define the menu and all its items in an XML menu resource. A menu resource defines an application menu (options menu, context menu, or popup menu) that can be inflated with MenuInflater, which loads the resource as a Menu object in your Activity.
- If you start an app project using the Basic Activity template, the template adds the menu resource for you and inflates the options menu with ***MenuInflater***, so you can skip this step and go right to "Defining how menu items appear".
- If you are not using the Basic Activity template, use the resource-inflate design pattern, which makes it easy to create an options menu. Follow these steps (refer to the figure below):



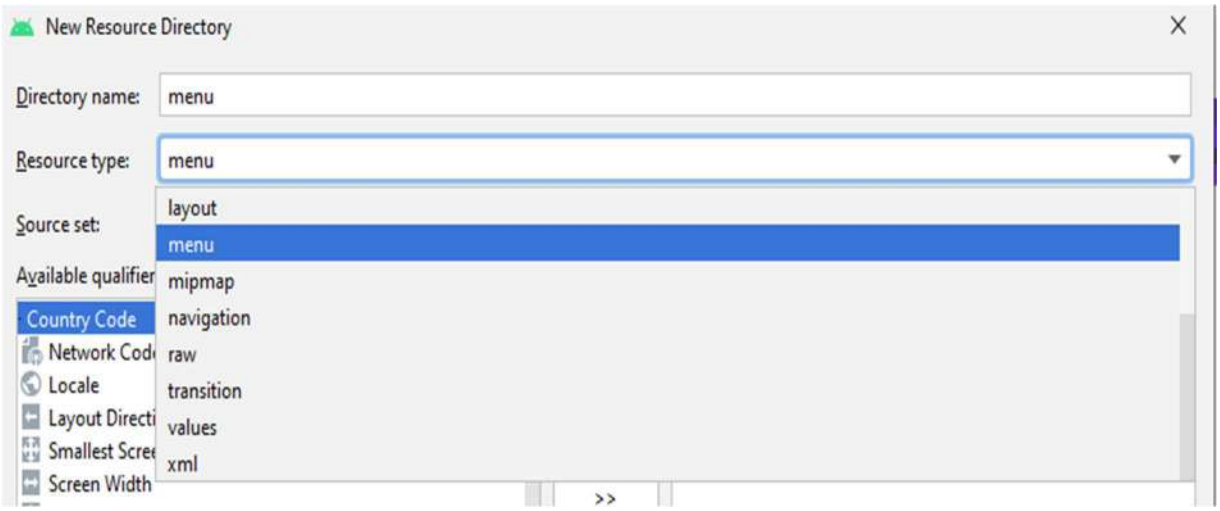
1. **XML menu resource.** Create an XML menu resource file for the menu items, and assign appearance and position attributes as described in the next section.
2. **Inflating the menu.** Override the onCreateOptionsMenu() method in your ***Activity*** to inflate the menu.
3. **Handling menu-item clicks.** Menu items are View elements, so you can use the ***android:onClick*** attribute for each menu item. However, the onOptionsItemSelected() method can handle all the menu-item clicks in one place and determine which menu item the user clicked, which makes your code easier to understand.
4. **Performing actions.** Create a method to perform an action for each options menu item.

Steps To Create App bar with options menu

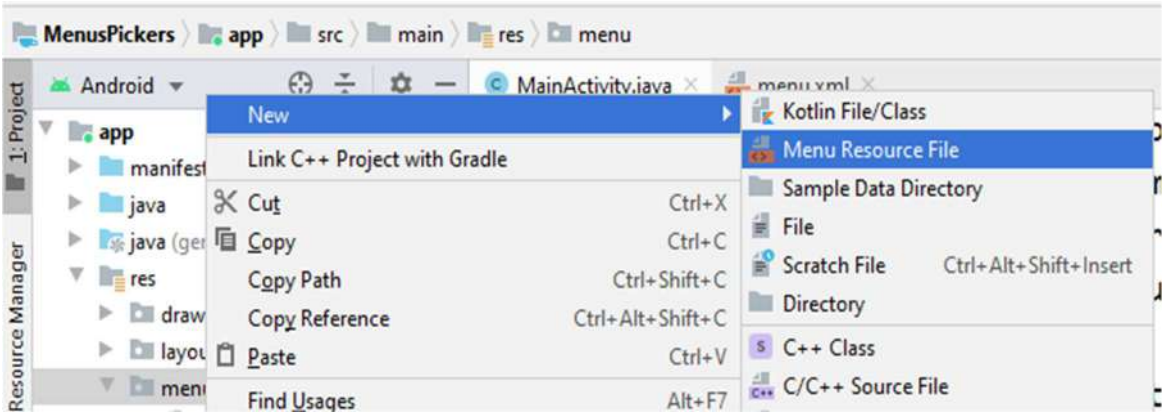
1. Create AndroidResourse directory
   - Select the **res** folder in the **Project > Android** pane and choose **File > New > Android resource directory.**



- Choose **menu** in the **Resource** type drop-down menu, and click **OK.**



2. XML menu resource (filename.xml)
   - Select the new **menu** folder, and choose **File > New > Menu resource file.**



- Enter the name, such as **Ex: menu_main**, and click **OK**. The new ***menu*main.xml_** file now resides within the **menu** folder.

- Add menu items using the tag.

```
<item
   android:id="@+id/user"
   android:title="User"/>
```

- Adding icons for menu items

1. Right-click **drawable**
2. Choose **New > Image Asset**
3. Choose **Action Bar and Tab Items**
4. Edit the icon name
5. Click clipart image, and click icon
6. Click **Next**, then **Finish**

- Icon and appearance attributes
- Use the following attributes to govern the menu item's appearance: *android:icon:* An image to use as the menu item icon. For example, the following menu item defines ic_order_white as its icon:

```
<item
   android:id="@+id/user"
   android:title="User"
   android:icon="@mipmap/user"
      />
```

- Use the ***app:showAsAction*** attribute to show menu items as icons in the app bar, with the following values:
- ***"always":*** Always place this item in the app bar. Use this only if it's critical that the item appear in the app bar (such as a Search icon). If you set multiple items to always

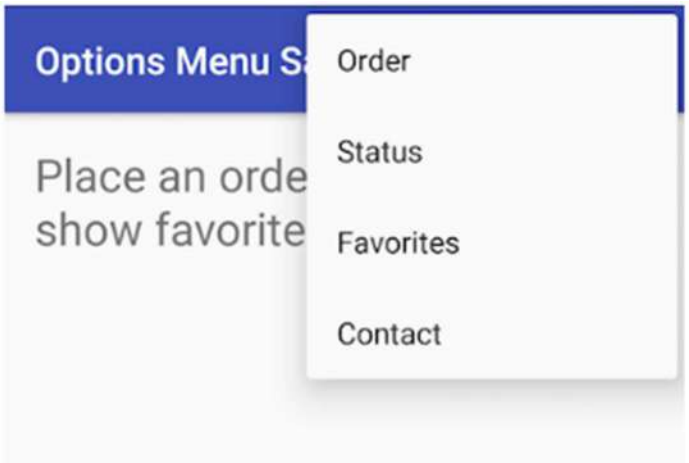
appear in the app bar, they might overlap something else in the app bar, such as the app title.

- ***"ifRoom":*** Only place this item in the app bar if there is room for it. If there is not enough room for all the items marked ***"ifRoom"***, the items with the lowest ***orderInCategory*** values are displayed in the app bar. The remaining items are displayed in the overflow menu.
- ***"never":*** Never place this item in the app bar. Instead, list the item in the app bar's overflow menu.
- ***"withText":*** Also include the title text (defined by ***android:title***) with the item. This attribute is used primarily to include the title with the icon in the app bar, because if the item appears in the overflow menu, the title text appears regardless.

Position attributes

- Use the ***android:orderInCategory*** attribute to specify the order in which the menu items appear in the menu, with the lowest number appearing higher in the menu. This is usually the order of importance of the item within the menu. For example, if you want **Order** to be first, followed by **Status, Favorites,** and **Contact**, the following table shows the priority of these items in the menu:

Menu item	orderInCategory attribute
Order	10
Status	20
Favorites	30
Contact	40




```
<item
    android:title="Call"
    android:id="@+id/call"
    android:icon="@mipmap/call"
    android:orderInCategory="40"
    app:showAsAction="always"/>
```

3. onCreateOptionsMenu() to inflate the menu

- If you start an app project using the Basic Activity template, the template adds the code for inflating the options menu with ***MenuInflater***, so you can skip this step.
- If you are not using the Basic Activity template, inflate the menu resource in your activity by overriding the onCreateOptionsMenu() method and using the getMenuInflater() method of the **Activity** class.
- The **getMenuInflater()** method returns a MenuInflater, which is a class used to instantiate menu XML files into **Menu** objects. The **MenuInflater** class provides the inflate() method, which takes two parameters:
- The resource ***id*** for an XML layout resource to load (***R.menu.menu*****main_** in the following example).
- The Menu to inflate into (*menu* in the following example)

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.menu,menu);
    return super.onCreateOptionsMenu(menu);
}
```

4. onClick attribute or onOptionsItemSelected()
   - However, the onOptionsItemSelected() method can handle all the menu-item clicks in one place and determine which menu item the user clicked. This makes your code easier to understand.

```
@Override
public boolean onOptionsItemSelected(@NonNull MenuItem item) {
    //Action
    return super.onOptionsItemSelected(item);
}
```

   - For example, you can use a ***switch case*** block to call the appropriate method (such as **Toast** message) based on the menu item's ***id*.** You retrieve the ***id*** using the getItemId() method:

```
@Override
public boolean onOptionsItemSelected(@NonNull MenuItem item) {
    switch (item.getItemId()){
        case R.id.notification:
            Toast.makeText(this, ""+item.getTitle(), Toast.LENGTH_SHORT).show();
            break;
        case R.id.dial:
            Toast.makeText(this, ""+item.getTitle(), Toast.LENGTH_SHORT).show();
            break;
        case R.id.call:
            Toast.makeText(this, ""+item.getTitle(), Toast.LENGTH_SHORT).show();
            break;
    }
    return super.onOptionsItemSelected(item);
}
```

Coding Implementation

- Create Start new android Studio Project and Select Empty Activity
- No need modify your ***activity*main.xml_**, If any requriments you can modify it.
- Create Android Resourse Directory with the name of menu
- Create menu resourse xml file
- adding items
- intialization in java file
- handling the click events

activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.and
roid.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

menu.xml

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <item
        android:id="@+id/notification"
        android:title="Search" />
    <item
        android:id="@+id/dial"
        android:title="Dail" />
    <item
        android:id="@+id/call"
        android:icon="@mipmap/call"
        android:title="Call"
        app:showAsAction="always" />
```

```
    <item
        android:id="@+id/user"
        android:title="User" />

</menu>
```

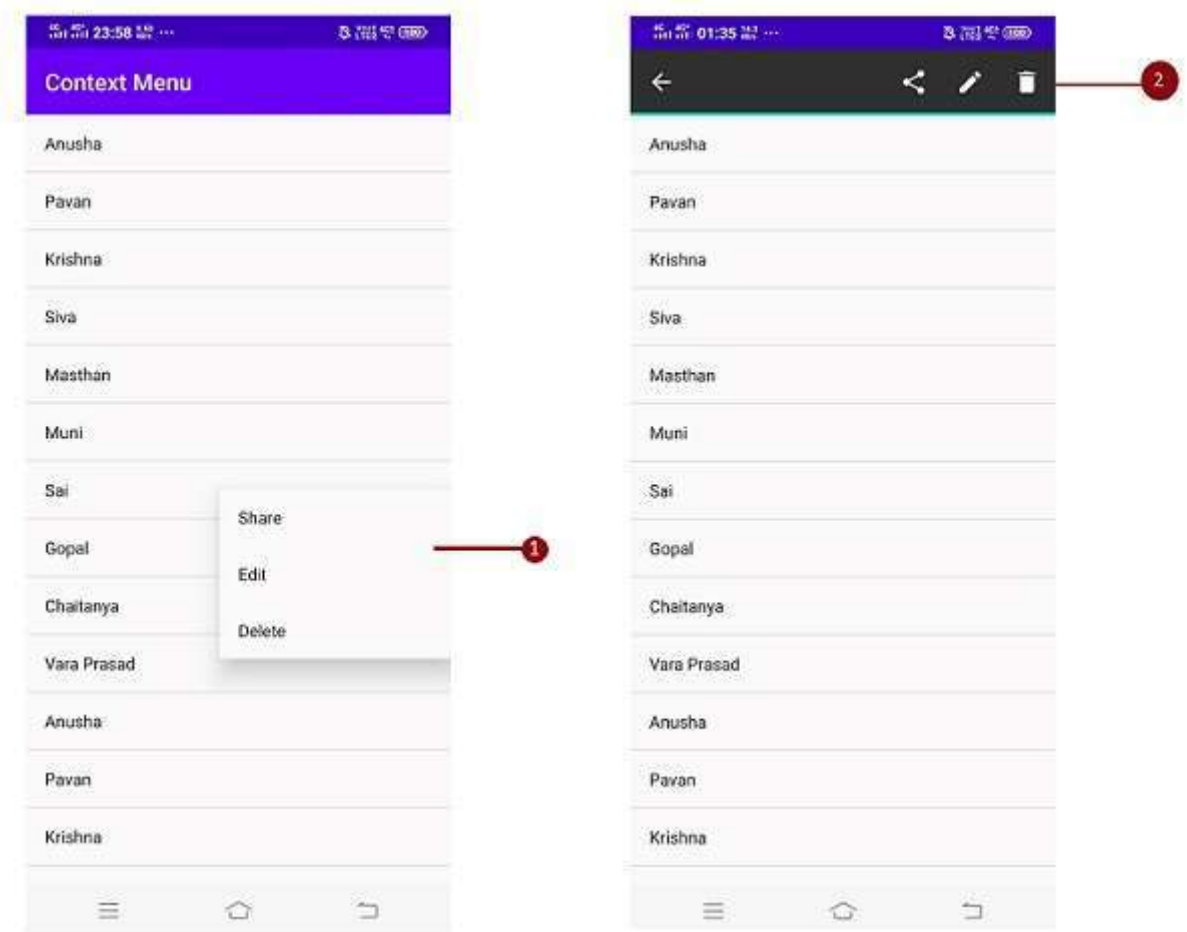
MainActivity.java

```
package com.example.menuspickers;

import android.os.Bundle;
import android.view.Menu;
import android.view.MenuItem;
import android.widget.Toast;
import androidx.annotation.NonNull;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {

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

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.menu, menu);
        return super.onCreateOptionsMenu(menu);
    }

    @Override
    public boolean onOptionsItemSelected(@NonNull MenuItem item) {

        switch (item.getItemId()) {
            case R.id.notification:
                // we are getting menu item title here
                Toast.makeText(this, ""+item.getTitle(), Toast.LENGTH_SHORT).show();
                break;
            case R.id.dial:
                Toast.makeText(this, ""+item.getTitle(), Toast.LENGTH_SHORT).show();
                break;
            case R.id.call:
                Toast.makeText(this, ""+item.getTitle(), Toast.LENGTH_SHORT).show();
                break;
        }
        return super.onOptionsItemSelected(item);
    }
}
```

Contextual Menus

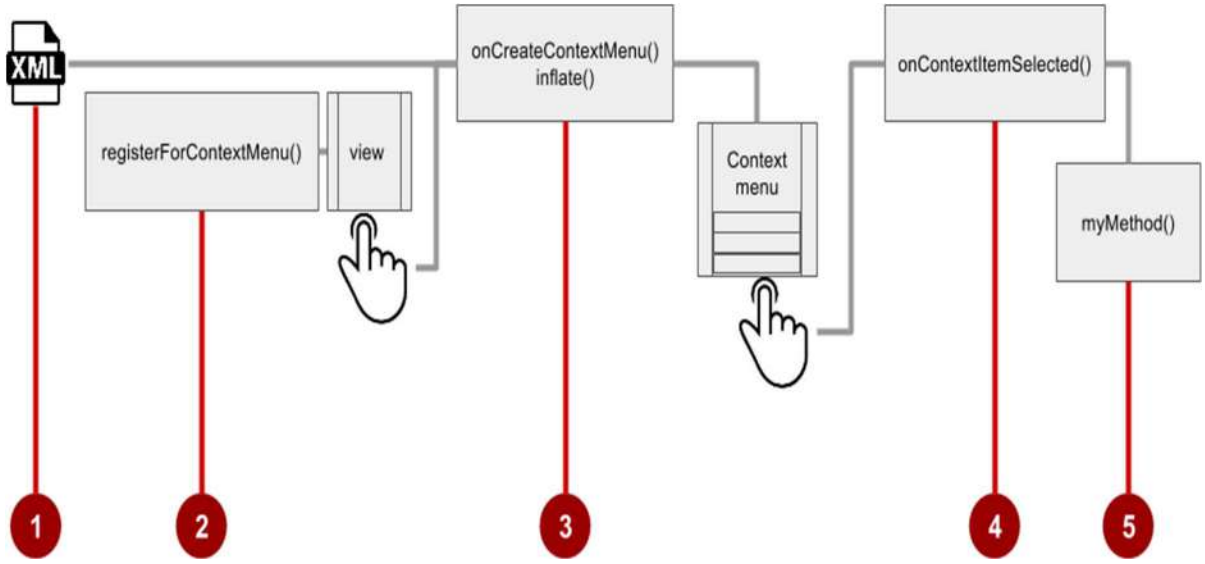
Use a contextual menu to allow users to take an action on a selected View. Contextual menus are most often used for items in a ListView, RecyclerView, GridView, or other view collection in which the user can perform direct actions on each item.

- Android provides two kinds of contextual menus:
  - A ***context menu***, shown on the left side in the figure below, appears as a floating list of menu items when the user performs a long tap on a *View*. It is typically used to modify the View or use it in some fashion.
  - For example, a context menu might include
  - **Edit** to edit the contents of a *View*,
  - **Delete** to delete a *View*,
  - **Share** to share a *View* over social media. Users can perform a contextual action on one selected View at a time.
  - A ***contextual action bar***, shown on the right side of the figure below, appears at the top of the screen in place of the app bar or underneath the app bar, with action items that affect one or more selected *View* elements. Users can perform an action on multiple *View* elements at once, if your app allows it.



Floating context menu

The familiar resource-inflate design pattern is used to create a context menu, modified to include registering (associating) the context menu with a *View*. The pattern consists of the steps shown in the figure below.



Steps to Implement Context Menu:

1. Create an XML menu resource file for the menu items. Assign appearance and position attributes as described in the previous section for the options menu.
2. Register a View to the context menu using the registerForContextMenu() method of the *Activity* class.
3. Implement the onCreateContextMenu() method in your *Activity* to inflate the menu.
4. Implement the onContextItemSelected() method in your *Activity* to handle menu-item clicks.
5. Create a method to perform an action for each context menu item.

Creating the XML resource file

To create the XML menu resource directory and file, follow the steps in the previous section for the options menu. However, use a different name for the file, such as menu_context. Add the context menu items within tags.

For example, the following code defines the ***Edit*** menu item:

```
<item
  android:id="@+id/context_edit"
  android:title="Edit"
  android:orderInCategory="10"/>
```

Registering a View to the context menu

To register a View to the context menu, call the registerForContextMenu() method with the View. Registering a context menu for a view sets the View.OnCreateContextMenuListener on the View to this activity, so that onCreateContextMenu() is called when it's time to show the context

menu. (You implement onCreateContextMenu in the next section.)
For example, in the *onCreate()* method for the *Activity*, you would add *registerForContextMenu()*:

```
// Registering the context menu to the TextView of the article.
ListView names_list = findViewById(R.id.list);

// Create String Array
String[] s ={"Anusha","Pavan","Krishna","Siva","Masthan","Muni","Sai","Gopal","Chaitanya","
Vara Prasad",
            "Anusha","Pavan","Krishna","Siva","Masthan","Muni","Sai","Gopal","Chaitanya","Va
ra Prasad"};

// For Displaying String Array Data in ListView, For That we need Create ArrayAdapter Like Bel
ow
ArrayAdapter<String> adapter=new ArrayAdapter<>(this,android.R.layout.simple_list_item_1,s
);

// set Adapter to ListView, Then Only we can display the data in ListView
lv.setAdapter(adapter);
registerForContextMenu(names_list);
```

Multiple views can be registered to the same context menu. If you want each item in a TextView or ListView or GridView to provide the same context menu, register all items for a context menu by passing the TextView or ListView or GridView to registerForContextMenu().

Implementing the *onCreateContextMenu()* method

When the registered View receives a long-click event, the system calls the onCreateContextMenu() method, which you can override in your *Activity*. (Long-click events are also called touch & hold events and *long-press* events.)

The *onCreateContextMenu()* method is where you define the menu items, usually by inflating a menu resource.

For example:

```
    @Override
   public void onCreateContextMenu(ContextMenu menu, View v,
ContextMenu.ContextMenuInfo menuInfo) {
      MenuInflater inflater = getMenuInflater();
      inflater.inflate(R.menu.menu,menu);
      super.onCreateContextMenu(menu, v, menuInfo);
   }
```

In the code above:

- The *menu* parameter for *onCreateContextMenu()* is the context menu to be built.
- The *v* parameter is the *View* registered for the context menu.
- The *menuInfo* parameter is extra information about the *View* registered for the context menu. This information varies depending on the class of the *v* parameter, which could be a *RecyclerView* or a *GridView*.

If you are registering a *RecyclerView* or a *GridView*, you instantiate a ContextMenu.ContextMenuInfo object to provide information about the item selected, and pass it as menuInfo, such as the row id, position, or child *View*.

The MenuInflater class provides the inflate() method, which takes two parameters:

- The resource *id* for an XML layout resource to load. In the example above, the *id* is *menu*context_.
- The Menu to inflate into. In the example above, the *Menu* is *menu*.

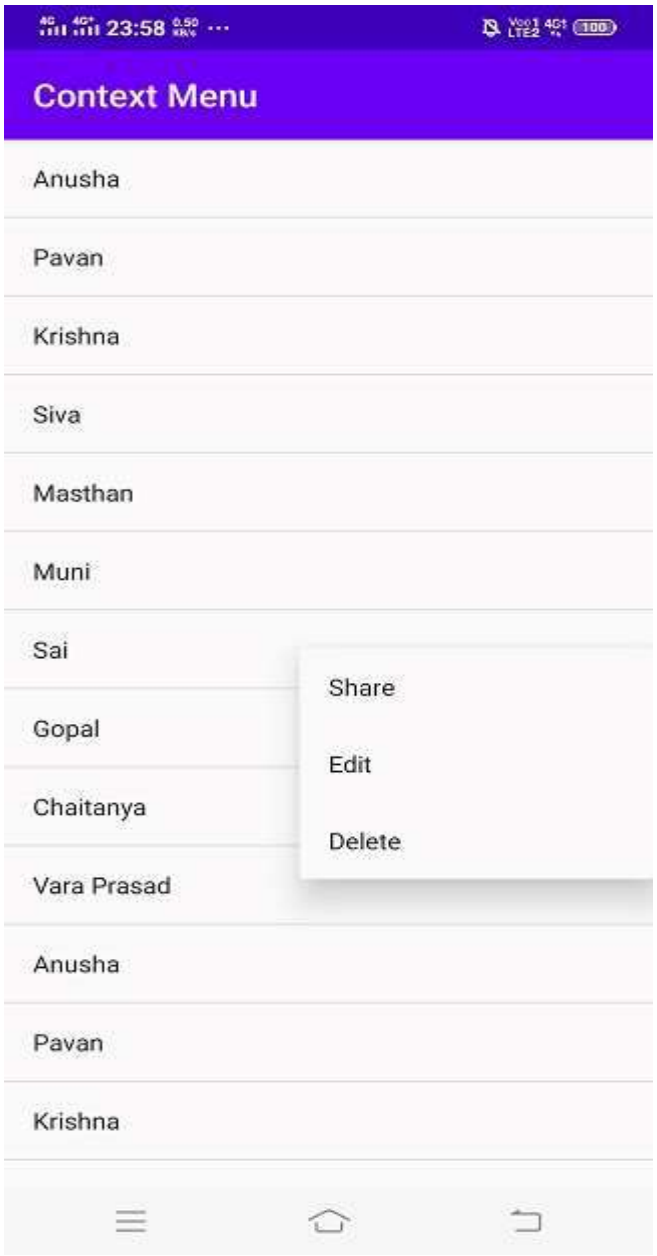
Implementing the onContextItemSelected() method

When the user clicks on a menu item, the system calls the onContextItemSelected() method. You override this method in your *Activity* in order to determine which menu item was clicked, and for which view the menu is appearing. You also use it to implement the appropriate action for the menu items, such as *editNote()* , *shareNote()* and *deleteNote()* in the following code snippet for the ***Edit*** , ***Share*** and ***Delete*** menu items:

```
@Override
public boolean onContextItemSelected(@NonNull MenuItem item) {
    switch (item.getItemId()){
        case R.id.share:
            // You can Write your requirement code here
            Toast.makeText(this, item.getTitle(), Toast.LENGTH_SHORT).show();
            break;
        case R.id.edit:
            // You can Write your requirement code here
            Toast.makeText(this, item.getTitle(), Toast.LENGTH_SHORT).show();
            break;
        case R.id.delete:
            // You can Write your requirement code here
            Toast.makeText(this, item.getTitle(), Toast.LENGTH_SHORT).show();
            break;
    }
    return super.onContextItemSelected(item);
}
```

The above code snippet uses the getItemId() method to get the *id* for the selected menu item, and uses it in a *switch case* block to determine which action to take. The *id* is the *android:id* attribute assigned to the menu item in the XML menu resource file.

When the user performs a long-click on the article in the *ListView*, the floating context menu appears and the user can click a menu item.

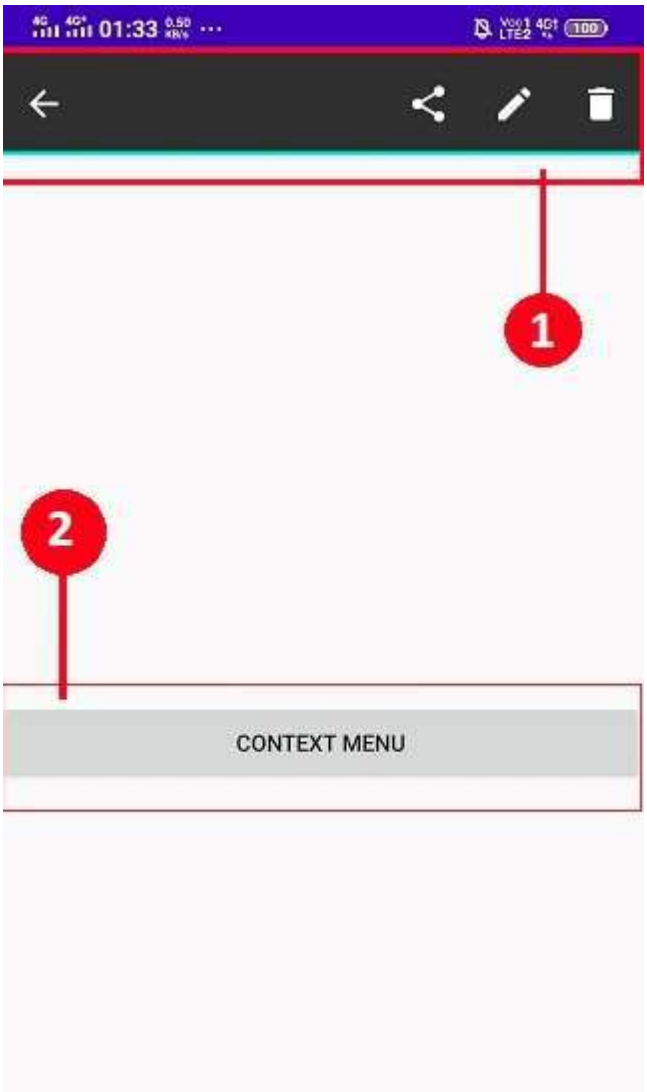


If you are using the *menuInfo* information for a *RecyclerView* or a *GridView*, you would add a statement before the *switch case* block to gather the specific information about the selected *View* (for *info*) by using AdapterView.AdapterContextMenuInfo:

```
AdapterView.AdapterContextMenuInfo info = (AdapterView.AdapterContextMenuInfo) item.
getMenuInfo();
```

Contextual action bar Menu

A ***contextual action bar*** appears at the top of the screen to present actions the user can perform on a *View* after long-clicking the *View*, as shown in the figure below.



In the above figure:

1. **Contextual action bar**. The bar offers three actions on the right side (**Edit**, **Share**, and **Delete**) and the Done button (left arrow icon) on the left side.
2. **View**. View on which a long-click triggers the contextual action bar.

The contextual action bar appears only when contextual action mode, a system implementation of ActionMode, occurs as a result of the user performing a long-click on one or more selected *View* elements.

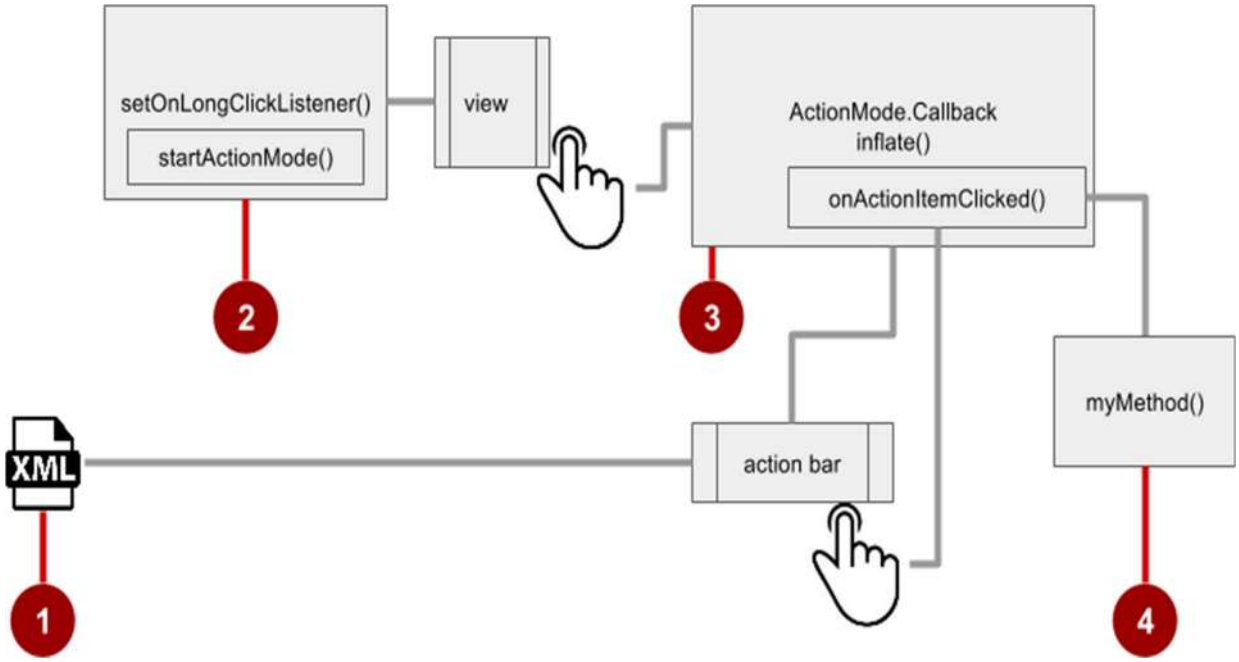
ActionMode represents UI mode for providing alternative interaction, replacing parts of the normal UI until finished. - For example, text selection is implemented as an *ActionMode*, as are contextual actions that work on a selected item on the screen. Selecting a section of text or long-clicking a view triggers *ActionMode*.

While this mode is enabled, the user can select multiple items, if your app allows it. The user can also deselect items, and continue to navigate within the activity. *ActionMode* is disabled when one of the following things occur:

- The user deselects all items.
- The user presses the Back button.
- The user taps **Done** (the left-arrow icon) on the left side of the action bar.

When *ActionMode* is disabled, the contextual action bar disappears.

Follow these steps to create a contextual action bar, as shown in the figure below:



Steps To Create ContextActionBar Menus:

1. Create an XML menu resource file for the menu items, and assign an icon to each one (as described in a previous section).
2. Set the long-click listener using setOnLongClickListener() to the View that should trigger the contextual action bar. Call startActionMode() within the *setOnLongClickListener()* method when the user performs a long tap on the View.
3. Implement the ActionMode.Callback interface to handle the *ActionMode* lifecycle. Include in this interface the action for responding to a menu-item click in the onActionItemClicked() callback method.
4. Create a method to perform an action for each context menu item.

Creating the XML resource file

Create the XML menu resource directory and file by following the steps in the previous section on the options menu. Use a suitable name for the file, such as *menu*context_. Add icons for the context menu items. For example, the **Edit** menu item would have these attributes:

```
<item
  android:id="@+id/context_edit"
  android:orderInCategory="10"
```

```
android:icon="@drawable/ic_action_edit_white"
android:title="Edit" />
```

The standard contextual action bar has a dark background. Use a light or white color for the icons. If you are using clip art icons, choose **HOLO_DARK** for the **Theme** drop-down menu when creating the new image asset.

Setting the long-click listener

Use setOnLongClickListener() to set a long-click listener to the *View* that should trigger the contextual action bar. Add the code to set the long-click listener to the *Activity* using the *onCreate()* method.

Follow these steps:

1. Declare the member variable mActionMode:

```
private ActionMode mActionMode;
```

You will call startActionMode() to enable ActionMode, which returns the *ActionMode* created. By saving this in a member variable *(mActionMode)*, you can make changes to the contextual action bar in response to other events.

2. Set up the contextual action bar listener in the *onCreate()* method, using *View* as the type in order to use the *setOnLongClickListener* like below syantax:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  // ... The rest of the onCreate code.
  Button b1 = findViewById(R.id.button); // Instead of Button you can write your own view
  b1.setOnLongClickListener(new View.OnLongClickListener()
  {
    // Start ActionMode after long-click.
  });
}
```

Implementing the ActionMode.Callback interface

Before you can add the code to *onCreate()* to start *ActionMode*, you must implement the ActionMode.Callback interface to manage the *ActionMode* lifecycle. In its callback methods, you can specify the actions for the contextual action bar, and respond to clicks on action items.

1. Add the following method to the *Activity* to implement the interface:

```
public ActionMode.Callback mActionModeCallback = new ActionMode.Callback() {
  // ... Code to create ActionMode.
}
```

2. Add the *onCreateActionMode()* code within the brackets of the above method to create ActionMode:

```
@Override
```

```
public boolean onCreateActionMode(ActionMode mode, Menu menu) {
    // Inflate a menu resource providing context menu items
    MenuInflater inflater = mode.getMenuInflater();
    inflater.inflate(R.menu.menu_context, menu);
    return true;
}
```

The onCreateActionMode() method inflates the menu using the same pattern used for a floating context menu. But this inflation occurs only when *ActionMode* is created, which is when the user performs a long-click. The MenuInflater class provides the inflate() method, which takes as a parameter the resource *id* for an XML layout resource to load (*menu*context_ in the above example), and the Menu to inflate into (*menu* in the above example).

3. Add the onActionItemClicked() method with your handlers for each menu item:

```
@Override
  public boolean onActionItemClicked(ActionMode mode, MenuItem item) {
     switch (item.getItemId()){
        case R.id.share:
           Toast.makeText(ContextActionBarActivity.this, "You are Selected"+item.getTi
tle(), Toast.LENGTH_SHORT).show();
           mode.finish();
           break;
        case R.id.edit:
           mode.finish();
           break;
        case R.id.delete:
           mode.finish();
           break;
     }
     return false;
  }
```

The above code above uses the getItemId() method to get the *id* for the selected menu item, and uses it in a *switch case* block to determine which action to take. The *id* in each *case* statement is the *android:id* attribute assigned to the menu item in the XML menu resource file.

The actions shown are the *editNote()* and *shareNote()* methods, which you create in the Activity. After the action is picked, you use the mode.finish() method to close the contextual action bar.

4. Add the onPrepareActionMode() and onDestroyActionMode() methods, which manage the ActionMode lifecycle:

```java
@Override public boolean onPrepareActionMode(ActionMode mode, Menu menu) { return false; // Return false if nothing is done. }
```

The *onPrepareActionMode()* method shown above is called each time *ActionMode* occurs, and is always called after *onCreateActionMode()*.

```
@Override
public void onDestroyActionMode(ActionMode mode) {
  mActionMode = null;
}
```

The *onDestroyActionMode()* method shown above is called when the user exits *ActionMode* by clicking **Done** in the contextual action bar, or clicking on a different view.
The following is the full code for the ActionMode.Callback interface implementation:

```
private ActionMode.Callback callback = new ActionMode.Callback() {
      @Override
      public boolean onCreateActionMode(ActionMode mode, Menu menu) {
         mode.getMenuInflater().inflate(R.menu.menu, menu);
         return true;
      }
      // Called each time ActionMode is shown. Always called after onCreateActionMode.
      @Override
      public boolean onPrepareActionMode(ActionMode mode, Menu menu) {
         return false;
      }
      // Called when the user selects a contextual menu item
      @Override
      public boolean onActionItemClicked(ActionMode mode, MenuItem item) {
         switch (item.getItemId()) {
            case R.id.share:
               Toast.makeText(ContextActionBarActivity.this, "You are Selected" + item.getT
itle(), Toast.LENGTH_SHORT).show();
               mode.finish();
               break;
            case R.id.edit:
               mode.finish();
               break;
            case R.id.delete:
               mode.finish();
               break;
         }
         return false;
      }
      // Called when the user exits the action mode
      @Override
      public void onDestroyActionMode(ActionMode mode) {
         mActionMode = null;
      }
   };
```

Starting ActionMode

You use startActionMode() to start *ActionMode* after the user performs a long-click.
To start *ActionMode*, add the *onLongClick()* method within the brackets of the

setOnLongClickListener method in *onCreate()* below syntax:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  // ... Rest of onCreate code
  b1.setOnLongClickListener(new View.OnLongClickListener() {
    // Called when the user long-clicks on articleView
    public boolean onLongClick(View view) {
      if (mActionMode != null) return false;
      // Start the contextual action bar
      // using the ActionMode.Callback.
      mActionMode =
          MainActivity.this.startActionMode(callback);
      view.setSelected(true);
      return true;
    }
  });
}
```

The above code first ensures that the *ActionMode* instance is not recreated if it's already active by checking whether *mActionMode* is *null* before starting the action mode:

```
if (mActionMode != null) return false;
```

When the user performs a long-click, the call is made to *startActionMode()* using the *ActionMode.Callback* interface, and the contextual action bar appears at the top of the display. The setSelected() method changes the state of this *View* to selected (set to *true*). The following is the code for the *onCreate()* method in the *Activity*, which now includes *setOnLongClickListener()* and *startActionMode()*:

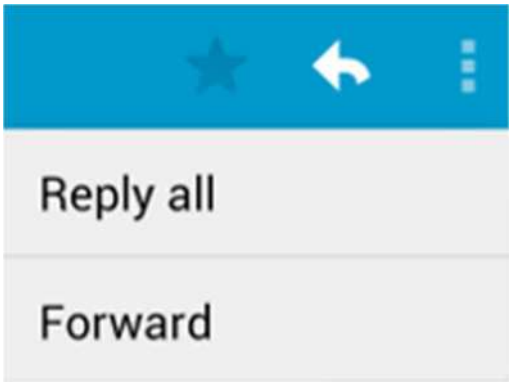
```
@Override
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.design);
   Button b1 = findViewById(R.id.button);
   b1.setOnLongClickListener(new View.OnLongClickListener() {
      @Override
      public boolean onLongClick(View v) {
         if (mActionMode != null) {
            return false;
         }
         // Start the contextual action bar using the ActionMode.Callback.
         mActionMode = ContextActionBarActivity.this.startActionMode(callback);
         return true;
      }
   });
}
```

Popup menu

A PopupMenu is a vertical list of items anchored to a View. It appears below the anchor *View* if there is room, or above the *View* otherwise.

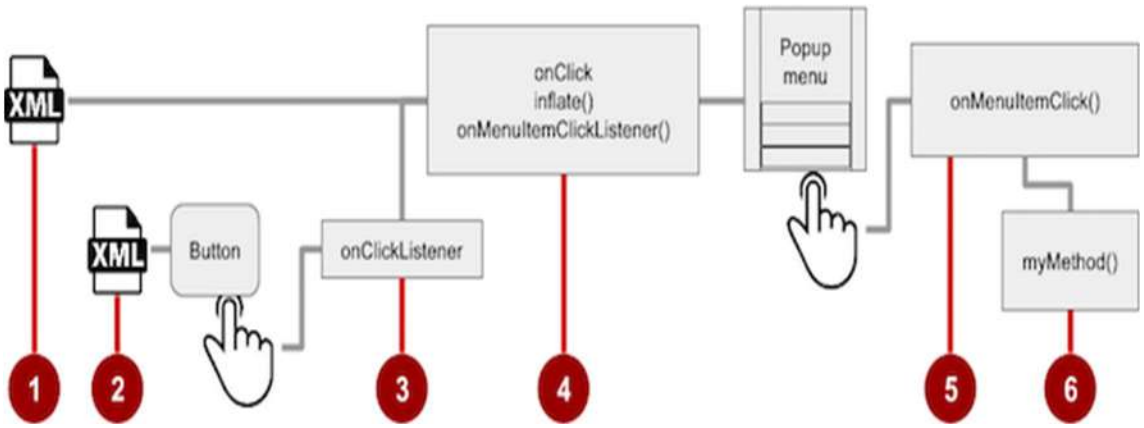
A popup menu is typically used to provide an overflow of actions (similar to the overflow action icon for the options menu) or the second part of a two-part command. Use a popup menu for extended actions that relate to regions of content in your *Activity*. Unlike a context menu, a popup menu is anchored to a *Button*, is always available, and its actions generally do not affect the content of the *View*.

For example, the Gmail app uses a popup menu anchored to the overflow icon in the app bar when showing an email message. The popup menu items **Reply** , **Reply All**, and **Forward** are related to the email message, but don't affect or act on the message. Actions in a popup menu should not directly affect the corresponding content (use a contextual menu to directly affect selected content). As shown below, a popup can be anchored to the overflow action button in the app bar.



Creating a popup menu

Follow these steps to create a popup menu (refer to figure below):



Steps to Implement PopUp Menu:

1. Create an XML menu resource file for the popup menu items, and assign appearance and position attributes (as described in a previous section).
2. Add an Button for the popup menu icon in the XML activity layout file.
3. Assign onClickListener() to the *Button*.
4. Override the *onClick()* method to inflate the popup menu and register it with PopupMenu.OnMenuItemClickListener.
5. Implement the onMenuItemClick() method.
6. Create a method to perform an action for each popup menu item.

Create the XML menu resource directory and file by following the steps in a previous section. Use a suitable name for the file, such as *menu*popup_.

Adding an ImageButton for the icon to click

Use an Button in the *Activity* layout for the icon that triggers the popup menu. Popup menus are anchored to a *View* in the *Activity*, such as an ImageButton. The user clicks it to see the menu.

```
<Button
  android:id="@+id/button"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:text="Pop Up Menu"/>
```

Assigning onClickListener to the button

1. Create a member variable (mButton) in the Activity class definition:

```
public class MainActivity extends AppCompatActivity {
private Button mButton;
// ... Rest of Activity code
}
```

2. In the *onCreate()* method for the same *Activity*, assign *onClickListener()* to the Button:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_popup_menu);
  b1= findViewById(R.id.button);
  b1.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
      // Implement PopMenu Here
    }
  });
}
```

Inflating the popup menu

As part of the *setOnClickListener()* method within *onCreate()*, add the onClick() method to inflate the popup menu and register it with PopupMenu.OnMenuItemClickListener:

```
    b1.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            PopupMenu popupMenu=new PopupMenu(PopupMenuActivity.this,b1);
            popupMenu.getMenuInflater().inflate(R.menu.menu,popupMenu.getMenu());

            popupMenu.show();
        }
    });
}
```

The method instantiates a PopupMenu object, which is *popup* in the example above. Then the method uses the MenuInflater class and its inflate() method.

The *inflate()* method takes the following parameters:

- The resource *id* for an XML layout resource to load, which is *menu*popup_ in the example above.
- The Menu to inflate into, which is *popup.getMenu()* in the example above.

The code then registers the popup with the listener, PopupMenu.OnMenuItemClickListener.

Implementing onMenuItemClick

To perform an action when the user selects a popup menu item, implement the onMenuItemClick() callback within the above *setOnClickListener()* method. Finish the method with *popup.show* to show the popup menu:

```
    b1.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            PopupMenu popupMenu=new PopupMenu(PopupMenuActivity.this,b1);
            popupMenu.getMenuInflater().inflate(R.menu.menu,popupMenu.getMenu());
            popupMenu.setOnMenuItemClickListener(new PopupMenu.OnMenuItemClickListene
r() {
                @Override
                public boolean onMenuItemClick(MenuItem item) {
                    switch (item.getItemId()){
                        case R.id.share:
                            Toast.makeText(PopupMenuActivity.this, "Share", Toast.LENGTH_SHORT
).show();
                            break;
                    }
```

```
                return false;
            }
        });
        popupMenu.show();
    }
});
```

The PopupMenu Output Shown Below: