**Unit 5**

# UI Fragments, Menus and Dialogs

## The need for UI flexibility

Flexible User Interfaces is a key part of proper Android app development. We need to ensure that our apps work across a wide variety of devices, versions, orientations and locales. Thankfully, Android has several powerful constructs that make responsive, flexible user interfaces fairly easy.

This is typically used for creating robust applications that work across different conditions. This guide will cover how to define alternative resources, how to have different interfaces for tablets vs phones, how to manage localization and much more.

## Introduction to fragments

**Android Fragment** is the part of activity, it is also known as sub-activity. There can be more than one fragment in an activity. Fragments represent multiple screens inside one activity. Android fragment lifecycle is affected by activity lifecycle because fragments are included in activity.

Each fragment has its own life cycle methods that is affected by activity life cycle because fragments are embedded in activity. The **FragmentManager** class is responsible to make interaction between fragment objects.

Unlike activities, fragments are usable. This means that they can be used to showcase different messages or functions to the user. These functionalities or features allow us to develop more interactive applications. Fragments also have their logic and can thus, accept and handle different events. Fragments are beneficial since they allow code to be divided into smaller and more manageable chunks.

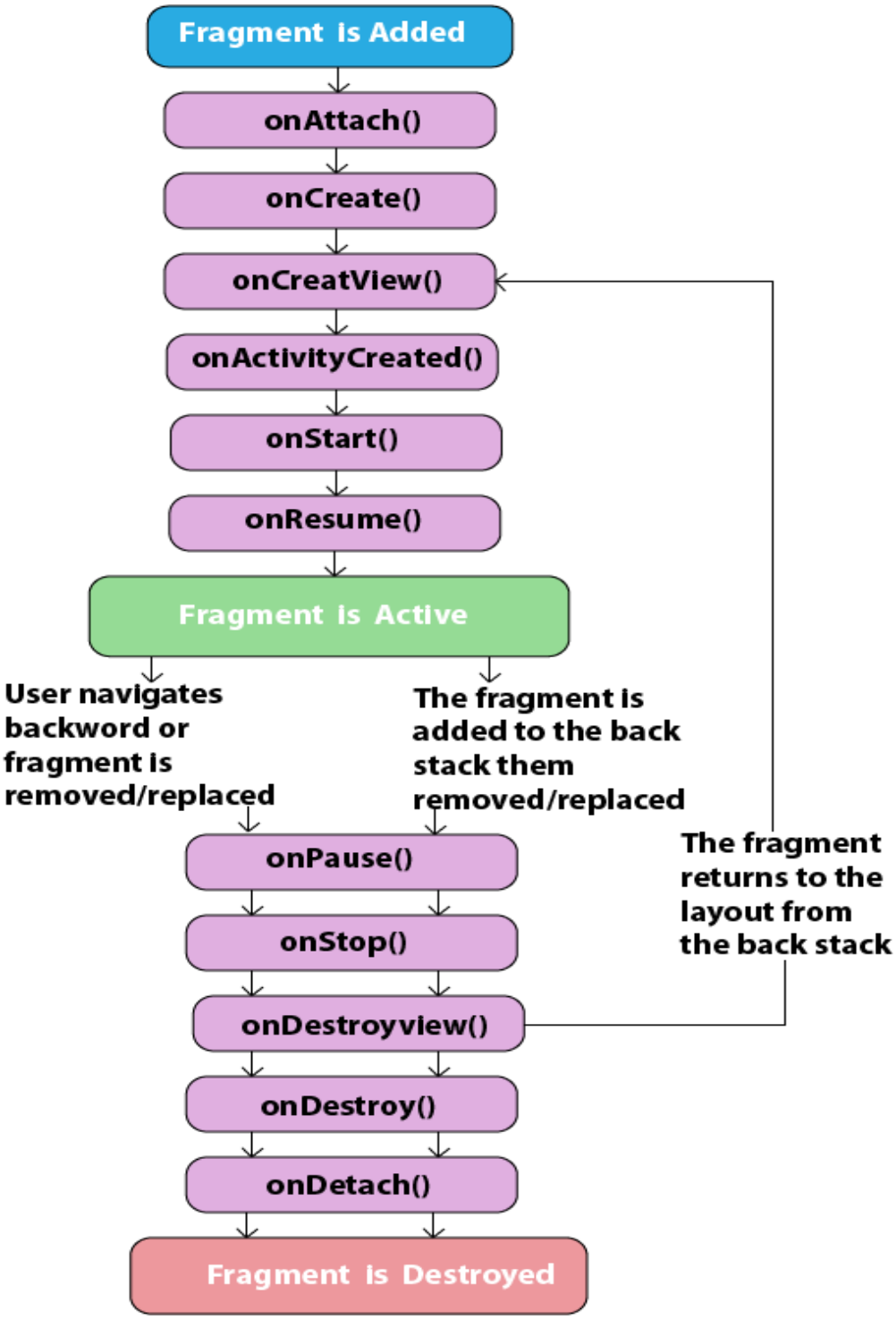
**Lifecycle of fragment**

Just like activities, fragments also have a lifecycle. This component tracks the fragments in all aspects of their lifecycle. This includes when they are initialized, started, created, resumed, and destroyed. A LifecycleObserver allows the developer to detect when a specific fragment is active. As a result, certain actions can be executed. For instance, an app can display a Snack bar or Toast message.

Alternatively, callback methods can also help in the management of the fragment's lifecycle. These functions include onCreate, onStart, onResume, onPause, onStop, and onDestroy. The callback methods are called depending on the fragment's state. Such as listed below:

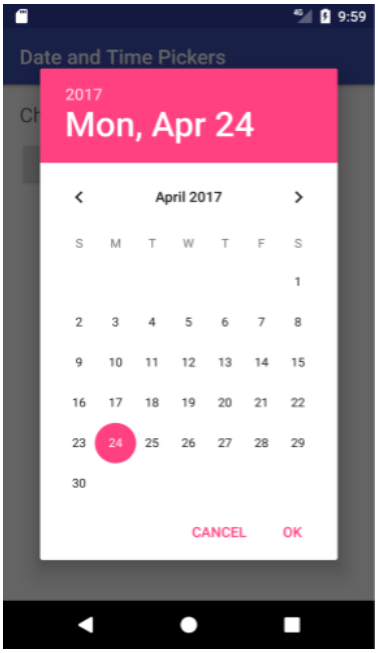
| No. | Method | Description |
|---|---|---|
| 1) | onAttach(Activity) | it is called only once when it is attached with activity. |
| 2) | onCreate(Bundle) | It is used to initialize the fragment. |
| 3) | onCreateView(LayoutInflater, ViewGroup, Bundle) | creates and returns view hierarchy. |
| 4) | onActivityCreated(Bundle) | It is invoked after the completion of onCreate() method. |
| 5) | onViewStateRestored(Bundle) | It provides information to the fragment that all the saved state of fragment view hierarchy has been restored. |
| 6) | onStart() | makes the fragment visible. |
| 7) | onResume() | makes the fragment interactive. |
| 8) | onPause() | is called when fragment is no longer interactive. |
| 9) | onStop() | is called when fragment is no longer visible. |
| 10) | onDestroyView() | allows the fragment to clean up resources. |

| | | |
|---|---|---|
| 11) | onDestroy() | allows the fragment to do final clean up of fragment state. |
| 12) | onDetach() | It is called immediately prior to the fragment no longer being associated with its activity. |


Fragment is Added
onAttach()
onCreate()
onCreatView()
onActivityCreated()
onStart()
onResume()
Fragment is Active
User navigates backword or fragment is removed/replaced
The fragment is added to the back stack them removed/replaced
onPause()
onStop()
onDestroyview()
onDestroy()
onDetach()
Fragment is Destroyed
The fragment returns to the layout from the back stack

## Creating a UI fragment

A Fragment is a self-contained component with its own user interface (UI) and lifecycle that can be reused in different parts of an app's UI. (A Fragment can also be used without a UI, in order to retain values across configuration changes, but this lesson does not cover that usage. A Fragment can be a static part of the UI of an Activity, which means that the Fragment remains on the screen during the entire lifecycle of the Activity. However, the UI of an Activity may be more effective if it adds or removes the Fragment dynamically while the Activity is running.One example of a dynamic Fragment is the DatePicker object, which is an instance of DialogFragment, a subclass of Fragment. The date picker displays a dialog window floating on top of its Activity window when a user taps a button or an action occurs. The user can click OK or Cancel to close the Fragment.



.

### Creating a fragment class

To create a fragment, extend the Fragment class, then override key lifecycle methods to insert our app logic, similar to the way we would with an Activity class.

One difference when creating a Fragment is that we must use the onCreateView() callback to define the layout. In fact, this is the only callback we need in order to get a fragment running. For example, here's a simple fragment that specifies its own layout:

```
import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.ViewGroup;

public class FragmentDemo extends Fragment {
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        // Inflate the layout for this fragment
        return inflater.inflate(R.layout.article_view, container, false);
    }
}
```

Just like an activity, a fragment should implement other lifecycle callbacks that allow us to manage its state as it is added or removed from the activity and as the activity transitions between its lifecycle states. For instance, when the activity's onPause() method is called, any fragments in the activity also receive a call to onPause().

**Android Fragment Example**

## activity_main.xml
## File: activity_main.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto"
            xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
       tools:context=".fragmentexample.MainActivity">

   <fragment
      android:id="@+id/fragment1"
      android:name="fragmentexample.Fragment1"
      android:layout_width="0px"
      android:layout_height="match_parent"
      android:layout_weight="1"
      />

   <fragment
      android:id="@+id/fragment2"
      android:name=" fragmentexample.Fragment2"
      android:layout_width="0px"
      android:layout_height="match_parent"
      android:layout_weight="1"
      />

</LinearLayout>
```

File: frgment_fragment1.xml

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#F5F5DC"
    tools:context=".fragmentexample.Fragment1">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="@string/hello_blank_fragment" />

</FrameLayout>
```

File: frgment_fragment2.xml

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#F0FFFF"
    tools:context=".fragmentexample.Fragment2">

    <!-- TODO: Update blank fragment layout -->
    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="@string/hello_blank_fragment" />
```

```
</FrameLayout>
```

## MainActivity class

## Fil:.java

```
package  fragmentexample;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;

public class MainActivity extends AppCompatActivity {

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

---

```
package example.javatpoint.com.fragmentexample;
import android.os.Bundle;
        import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class Fragment1 extends Fragment {
```

```
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
Bundle savedInstanceState) {
        // Inflate the layout for this fragment
        return inflater.inflate(R.layout.fragment_fragment1, container, false);    }   }
```

File: Fragment2.java

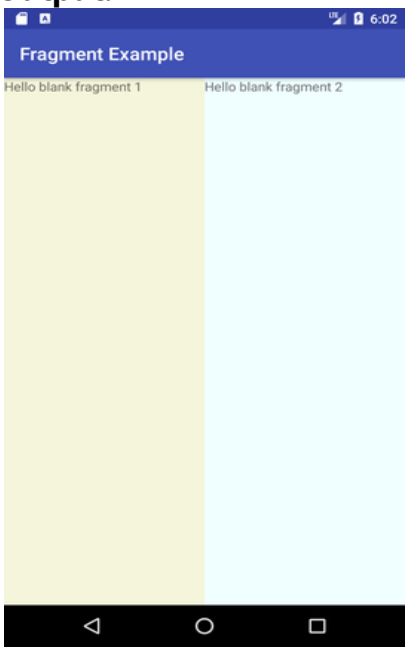
```
package fragmentexample;
import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class Fragment2 extends Fragment {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                        Bundle savedInstanceState) {
        // Inflate the layout for this fragment
        return inflater.inflate(R.layout.fragment_fragment2, container, false);
```

} }

**Output:**




## Introduction to fragment manager

The Fragment Manager is responsible for all runtime management of fragments including adding, removing, hiding, showing, or otherwise navigating between fragments. The Fragment Manager manages the fragment back stack. At runtime, the Fragment Manager can perform back stack operations like adding or removing fragments in response to user interactions. Each set of changes are committed together as a single unit called a Fragment Transaction. For a more in-depth discussion about fragment transactions, see the fragment transactions guide.

When the user presses the Back button on their device, or when we call FragmentManager.popBackStack(), the top-most fragment transaction is popped off of the stack. In other words, the transaction is reversed. If there are no more fragment transactions on the stack, and if you aren't using child fragments, the back event bubbles up to the activity. If you are using child fragments, see special considerations for child and sibling fragments.

When you call addToBackStack() on a transaction, note that the transaction can include any number of operations, such as adding multiple fragments, replacing fragments in multiple

containers, and so on. When the back stack is popped, all of these operations are reversed as a single atomic action. If we've committed additional transactions prior to the popBackStack () call, and if we did not use addToBackStack () for the transaction, these operations are not reversed. Therefore, within a single Fragment Transaction, avoid interleaving transactions that affect the back stack with those that do not.

Important available methods are outlined below:

| **Method** | **Description** |
|---|---|
| addOnBackStackChangedListener | Add a new listener for changes to the fragment back stack. |
| beginTransaction() | Creates a new transaction to change fragments at runtime. |
| findFragmentById(int id) | Finds a fragment by id usually inflated from activity XML layout. |
| findFragmentByTag(String tag) | Finds a fragment by tag usually for a runtime added fragment. |
| popBackStack() | Remove a fragment from the backstack. |
| executePendingTransactions() | Forces committed transactions to be applied. |

**Difference between Activity and Fragments**

| **Activity** | **Fragment** |
|---|---|
| Activity is an application component that gives a user interface where the user can interact. | The fragment is only part of an activity, it basically contributes its UI to that activity. |
| Activity is not dependent on fragment | Fragment is dependent on activity. It can't exist independently. |
| we need to mention all activity it in the manifest.xml file | Fragment is not required to mention in the manifest file |
| We can't create multi-screen UI without using fragment in an activity, | After using multiple fragments in a single activity, we can create a multi-screen UI. |
| Activity can exist without a Fragment | Fragment cannot be used without an Activity. |

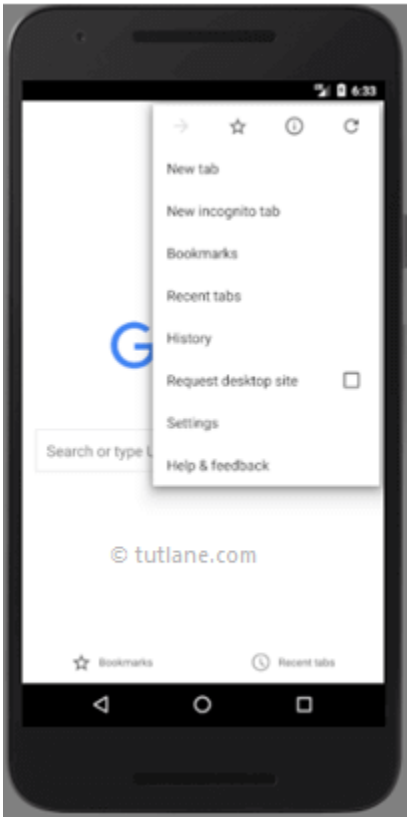
| | |
|---|---|
| Creating a project using only Activity then it's difficult to manage | While Using fragments in the project, the project structure will be good and we can handle it easily. |
| Lifecycle methods are hosted by OS. The activity has its own life cycle. | Lifecycle methods in fragments are hosted by hosting the activity. |
| Activity is not lite weight. | The fragment is the lite weight. |

**Menu**

In android, **Menu** is a part of the user interface (UI) component which is used to handle some common functionality around the application. By using Menus in our applications, we can provide better and consistent user experience throughout the application.

We can use Menu APIs to represent user actions and other options in our android application activities.

Following is the pictorial representation of using menus in the android application.



In android, we can define a Menu in separate XML file and use that file in our activities or fragments based on our requirements.

## Define an Android Menu in XML File

For all menu types, Android provides a standard XML format to define menu items. Instead of building a menu in our activity's code, we should define a menu and all its items in an XML menu resource and load menu resource as a Menu object in our activity or fragment.

In android, to define menu, we need to create a new folder **menu** inside of our project resource directory (**res/menu/**) and add a new XML file to build the menu with the following elements.

| Element | Description |
|---|---|
| <menu> | It's a root element to define a Menu in XML file and it will hold one or more and elements. |
| <item> | It is used to create a menu item and it represents a single item on the menu. This element may contain a nested <menu> element in order to create a submenu. |
| <group> | It's an optional and invisible for <item> elements. It is used to categorize the menu items so they share properties such as active state and visibility. |

Following is the example of defining a menu in an XML file (**menu_example.xml**).

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
   <item android:id="@+id/mail"
      android:icon="@drawable/ic_mail"
      android:title="@string/mail" />
   <item android:id="@+id/upload"
```

```
    android:icon="@drawable/ic_upload"
    android:title="@string/upload"
    android:showAsAction="ifRoom" />
  <item android:id="@+id/share"
    android:icon="@drawable/ic_share"
    android:title="@string/share" />
</menu>
```

The **<item>** element in **menu** supports different type of attributes to define item's behaviour and appearance. Following are the some of commonly used **<item>** attributes in android applications.

| Attribute | Description |
|---|---|
| android: id | It is used to uniquely identify an element in the application. |
| android:icon | It is used to set the item's icon from drawable folder. |
| android: title | It is used to set the item's title |
| android:showAsAction | It is used to specify how the item should appear as an action item in the app bar. |

In case if we want to add **submenu** in **menu** item, then we need to add a **<menu>** element as the child of an **<item>**. Following is the example of defining a submenu in menu item.

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
   <item android:id="@+id/file"
      android:title="@string/file" >
      <!-- "file" submenu -->
      <menu>
         <item android:id="@+id/create_new"
            android:title="@string/create_new" />
         <item android:id="@+id/open"
            android:title="@string/open" />
      </menu>
   </item>
</menu>
```

**Load Android Menu from an Activity**

Once we are done with creation of menu, we need to load the menu resource from our activity using **MenuInflater.inflate()** like as shown below.

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

If we observe above code we are calling our menu using MenuInflater.inflate() method in the form of **R.menu.menu_file_name**. Here our xml file name is **menu_example.xml** so we used file name **menu_example**.

**Handle Android Menu Click Events**

In android, we can handle a menu item click events using **ItemSelected()** event based on the menu type. Following is the example of handling a context menu item click event using **onContextItemSelected()**.

```
@Override
public boolean onContextItemSelected(MenuItem item) {
   switch (item.getItemId()) {
      case R.id.mail:
         // do something
         return true;
      case R.id.share:
         // do something
         return true;
      default:
         return super.onContextItemSelected(item);
   }
}
```

If you observe above code, the **getItemId()** method will get the id of selected menu item based on that we can perform our actions.

**Android Different Types of Menus**

In android, we have a three fundamental type of Menus available to define a set of options and actions in our android applications.
The following are the commonly used Menus in android applications.

- Options Menu
- Context Menu
- Popup Menu

**Android Options Menu**

In android, **Options Menu** is a primary collection of menu items for an activity and it is useful to implement actions that have a global impact on the app, such as Settings, Search, etc.

## options_menu.xml

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android" >
    <item android:id="@+id/search_item"
        android:title="Search" />
    <item android:id="@+id/upload_item"
        android:title="Upload" />
    <item android:id="@+id/copy_item"
        android:title="Copy" />
    <item android:id="@+id/print_item"
        android:title="Print" />
    <item android:id="@+id/share_item"
        android:title="Share" />
        <item android:id="@+id/bookmark_item"
        android:title="BookMark" />
</menu>
```

## MainActivity.java

```
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuItem;
import android.widget.Toast;


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.options_menu, menu);
    return true;
  }
  @Override
  public boolean onOptionsItemSelected(MenuItem item) Toast.makeText(this, "Selected Item:
" +item.getTitle(),
                                Toast.LENGTH_SHORT).show();
    switch (item.getItemId()) {
      case R.id.search_item:
        // do your code
        return true;
      case R.id.upload_item:
        // do your code
        return true;
      case R.id.copy_item:
        // do your code
        return true;
      case R.id.print_item:
        // do your code
        return true;
      case R.id.share_item:
        // do your code
        return true;
      case R.id.bookmark_item:
        // do your code
        return true;
      default:
        return super.onOptionsItemSelected(item);
    }
  }
}
```

## Android Context Menu

In android, **Context Menu** is a floating menu that appears when the user performs a long click on an element and it is useful to implement actions that affect the selected content or context frame.

### activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:orientation="vertical" >
   <Button
      android:id="@+id/btnShow"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Long press me"
      android:layout_marginTop="200dp"

      android:layout_marginLeft="100dp"/>
</LinearLayout>
```

If we observe above code we created a one Button control in XML Layout file to show the context menu when we do long press on Button.

Once we are done with the creation of layout with required control, we need to load the XML layout resource from our activity **onCreate()** callback method, for that open main activity file **MainActivity.java** from **contextmenuexample** path and write the code like as shown below.

### MainActivity.java

```
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.ContextMenu;
import android.view.MenuItem;
import android.view.View;
```

```
import android.widget.Button;
import android.widget.Toast;

public class MainActivity extends AppCompatActivity {
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      Button btn = (Button) findViewById(R.id.btnShow);
      registerForContextMenu(btn);
   }
   @Override
   public void onCreateContextMenu(ContextMenu menu, View v, ContextMenu.ContextMenuInfo
menuInfo) {
      super.onCreateContextMenu(menu, v, menuInfo);
      menu.setHeaderTitle("Context Menu");
      menu.add(0, v.getId(), 0, "Upload");
      menu.add(0, v.getId(), 0, "Search");
      menu.add(0, v.getId(), 0, "Share");
      menu.add(0, v.getId(), 0, "Bookmark");
   }
   @Override
   public boolean onContextItemSelected(MenuItem item) {
      Toast.makeText(this, "Selected Item: " +item.getTitle(), Toast.LENGTH_SHORT).show();
      return true;
   }
}
```

### Android Popup Menu

In android, **Popup Menu** displays a list of items in a vertical list that's anchored to the view that invoked the menu and it's useful for providing an overflow of actions that related to specific content.

Create a new android application using android studio and give names as **PopupMenuExample**. In case if you are not aware of creating an app in android studio check this article Android Hello World App.

### activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:orientation="vertical" >

   <Button
      android:id="@+id/btnShow"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Show Popup Menu"
      android:layout_marginTop="200dp" android:layout_marginLeft="100dp"/>
</LinearLayout>
```

If we observe above code we created a one Button control in XML Layout file to show the popup menu when we click on Button.

In android, to define the popup menu, we need to create a new folder menu inside of our project resource directory (res/menu/) and add a new XML (popup_menu.xml) file to build the menu.

Now open newly created xml (popup_menu.xml) file and write the code like as shown below.

## popup_menu.xml

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android" >
   <item android:id="@+id/search_item"
      android:title="Search" />
```

```
    <item android:id="@+id/upload_item"
        android:title="Upload" />
    <item android:id="@+id/copy_item"
        android:title="Copy" />
    <item android:id="@+id/print_item"
        android:title="Print" />
    <item android:id="@+id/share_item"
        android:title="Share" />
    <item android:id="@+id/bookmark_item"
        android:title="BookMark" />
</menu>
```

## MainActivity.java

```
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.MenuItem;
import android.view.View;
import android.widget.Button;
import android.widget.PopupMenu;
import android.widget.Toast;

public
class MainActivity extends AppCompatActivity implements PopupMenu.OnMenuItemClickList
ener {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button btn = (Button) findViewById(R.id.btnShow);
```

```
    btn.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            PopupMenu popup = new PopupMenu(MainActivity.this, v);
            popup.setOnMenuItemClickListener(MainActivity.this);
            popup.inflate(R.menu.popup_menu);
            popup.show();
        }
    });
}
@Override
public boolean onMenuItemClick(MenuItem item) {
    Toast.makeText(this, "Selected Item: " +item.getTitle(), Toast.LENGTH_SHORT).show();
    switch (item.getItemId()) {
        case R.id.search_item:
            // do your code
            return true;
        case R.id.upload_item:
            // do your code
            return true;
        case R.id.copy_item:
            // do your code
            return true;
        case R.id.print_item:
            // do your code
            return true;
        case R.id.share_item:
            // do your code
            return true;
        case R.id.bookmark_item:
            // do your code
```

```
            return true;
        default:
            return false;
    } } }
```

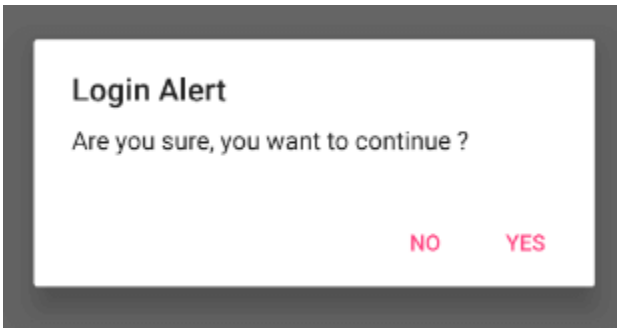
If we observe above code we are trying to show popup menu on Button click, loaded defined menu resource using Popup.inflate() and implement popup menu items click event.

Generally, during the launch of our activity, the onCreate() callback method will be called by the android framework to get the required layout for an activity.

# Dialogs

Dialog is a small window that prompt messages to the user to make a decision or enter additional details. Generally, the Dialogs are used with modals event and these useful to prompt users to perform a particular action to proceed further in the application.

Following is the pictorial representation of using dialogs in android applications.



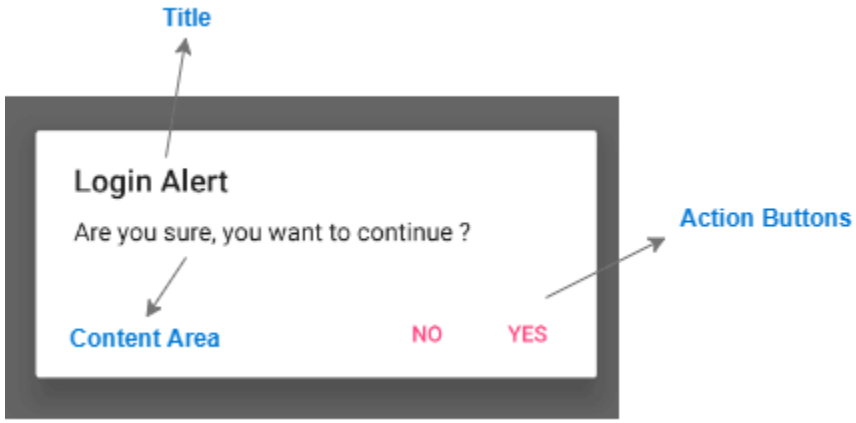

In android, we have a different type of Dialogs available, those are

| Dialog | Description |
| --- | --- |
| AlertDialog | This dialog is used to display prompt to the user with title, upto three buttons, list of selectable items or a custom layout. |
| DatePickerDialog | This dialog is a predefined UI control and it allows a user to select Date. |
| TimePickerDialog | It's a predefined UI control and it allows a user to select Time. |

## Android AlertDialog

In android, AlertDialog is used to prompt a dialog to the user with messages and buttons to perform an action to proceed further.

The AlertDialog in an android application will contain three regions like as shown below.




In android Alert Dialogs, we can show a title, up to three buttons, a list of selectable items or a custom layout based on our requirements.

| Region | Description |
|---|---|
| Title | It’s optional and it can be used to show the detailed messages based on our requirements. |
| Content Area | It is used to display a message, list or other custom layouts based on our requirements. |
| Action Buttons | It is used to display action buttons to interact with users. We can use upto 3 different action buttons in alert dialogs, such as positive, negative and neutral. |

Generally, in android we can build AlertDialog in our activity file using different dialog methods.

## Android AlertDialog Methods

Following are the some of commonly used methods related to AlertDialog control to build alert prompt in android applications.

| Method | Description |
|---|---|
| setTitle() | It is used to set the title of alertdialog and its an optional component. |
| setIcon() | It is used to set the icon before the title |
| setMessage() | It is used to set the message required message to display in alertdialog. |
| setCancelable() | It is used to allow users to cancel alertdialog by clicking on outside of dialog area by setting true/false. |
| setPositiveButton() | It is used to set the positive button for alertdialog and we can implement click event of a positive button. |

| Method | Description |
| --- | --- |
| setNegativeButton() | It is used to set the negative button for alertdialog and we can implement click event of a negative button. |
| setNeutralButton() | It is used to set the neutral button for alertdialog and we can implement click event of a neutral button. |

**Android AlertDialog Example**

Following is the example of defining a one Button control in RelativeLayout to show the AlertDialog and get the action that was performed by a user on Button click in the android application.

Create a new android application using android studio and give names as AlertDialogExample. In case if you are not aware of creating an app in android studio check this article Android Hello World App.

Now open an activity_main.xml file from \res\layout path and write the code like as shown below

## activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent" android:layout_height="match_parent">
     <Button
       android:id="@+id/getBtn"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:layout_marginLeft="150dp"
       android:layout_marginTop="200dp"
       android:text="Show Alert" />
</RelativeLayout>
```

Once we are done with the creation of layout with required controls, we need to load the XML layout resource from our activity onCreate() callback method, for that open main activity file MainActivity.java from \java\com.tutlane.alertdialogexample path and write the code like as shown below.

## MainActivity.java

```
import android.content.DialogInterface;
import android.support.v7.app.AlertDialog;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.Toast;

public class MainActivity extends AppCompatActivity {
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      Button btn = (Button)findViewById(R.id.getBtn);
      btn.setOnClickListener(new View.OnClickListener() {
         @Override
         public void onClick(View v) {
            AlertDialog.Builder builder = new AlertDialog.Builder(MainActivity.this);
            builder.setTitle("Login Alert")
                  .setMessage("Are you sure, you want to continue ?")
                  .setCancelable(false)
                  .setPositiveButton("Yes", new DialogInterface.OnClickListener() {
                     @Override
```

```
                public void onClick(DialogInterface dialog, int which) {
                    Toast.makeText(MainActivity.this,"Selected Option:
                                                YES",Toast.LENGTH_SHORT).show();
                }
            })
            .setNegativeButton("No", new DialogInterface.OnClickListener() {
                @Override
                public void onClick(DialogInterface dialog, int which) {
                    Toast.makeText(MainActivity.this,"Selected Option:
                                                No",Toast.LENGTH_SHORT).show();
                }
            });
        //Creating dialog box
        AlertDialog dialog  = builder.create();
        dialog.show();
      }
    });
  }
}
```

If we observe above code we are calling our layout using setContentView method in the form of R.layout.layout_file_name in our activity file. Here our xml file name is activity_main.xml so we used file name activity_main and we are trying to show the AlertDialog on Button click.

Generally, during the launch of our activity, onCreate() callback method will be called by android framework to get the required layout for an activity.

## Output

When we run the above example using the android virtual device (AVD) we will get a result like as shown below.


This is how we can use AlertDialog control in android applications to show the alert dialog in android applications based on our requirements.

**Android AlertDialog Add Items List**

In android, we can show the list of items in AlertDialog based on our requirements like as shown below.


There are three different kinds of lists available with AlertDialogs in android, those are

- Single Choice List
- Single Choice List with Radio Buttons
- Single Choice List with Checkboxes

Now we will see how to use Single Choice List with Checkboxes in android application to show the list of items with checkboxes in AlertDialog and get selected item values with examples.

**Android AlertDialog Setmultichoiceitems Example**

Create a new android application using android studio and give names as AlertDialogExample. In case if we are not aware of creating an app in android studio check this article Android Hello World App.

Now open an activity_main.xml file from \res\layout path and write the code like as shown below

## activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent">
      <Button
        android:id="@+id/getBtn"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
```

```
    android:layout_marginLeft="150dp"
    android:layout_marginTop="200dp"
    android:text="Show Alert" />
</RelativeLayout>
```

If we observe above code we defined a one Button control in RelativeLayout to show the alert dialog on Button click in XML layout file.

Once we are done with the creation of layout with required controls, we need to load the XML layout resource from our activity onCreate() callback method, for that open main activity file MainActivity.java from \java\com.tutlane.alertdialogexample path and write the code like as shown below.

## MainActivity.java

```
import android.content.DialogInterface;
import android.support.v7.app.AlertDialog;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.Toast;
import java.util.ArrayList;

public class MainActivity extends AppCompatActivity {
   final CharSequence[] colors = { "Pink", "Red", "Yellow", "Blue" };
   ArrayList<Integer> slist = new ArrayList();
   boolean icount[] = new boolean[colors.length];
   String msg ="";
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
```

```
        setContentView(R.layout.activity_main);
        Button btn = (Button)findViewById(R.id.getBtn);
        btn.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                AlertDialog.Builder builder = new AlertDialog.Builder(MainActivity.this);
                builder.setTitle("Choose Colors")
                .setMultiChoiceItems(colors,icount, new DialogInterface.OnMultiChoiceClickListener
() {
                    @Override
                    public void onClick(DialogInterface arg0, int arg1, boolean arg2) {
                      if (arg2) {
                          // If user select a item then add it in selected items
                         slist.add(arg1);
                       } else if (slist.contains(arg1)) {
                          // if the item is already selected then remove it
                         slist.remove(Integer.valueOf(arg1));
                       }
                    }
                })     .setCancelable(false)
                       .setPositiveButton("Yes", new DialogInterface.OnClickListener() {
                          @Override
                          public void onClick(DialogInterface dialog, int which) {
                             msg = "";
                             for (int i = 0; i < slist.size(); i++) {
                                msg = msg + "\n" + (i + 1) + " : " + colors[slist.get(i)];
                             }
                             Toast.makeText(getApplicationContext(), "Total " + slist.size() + " Items
Selected.\n" + msg, Toast.LENGTH_SHORT).show();
                          }
                       })
```

```
                .setNegativeButton("No", new DialogInterface.OnClickListener() {
                    @Override
                    public void onClick(DialogInterface dialog, int which) {
                        Toast.makeText(MainActivity.this,"No Option
Selected",Toast.LENGTH_SHORT).show();
                    }
                });
            //Creating dialog box
            AlertDialog dialog  = builder.create();
            dialog.show();
        }
    });
  }
}
```

If we observe above code we are calling our layout using setContentView method in the form of R.layout.layout_file_name in our activity file. Here our xml file name is activity_main.xml so we used file name activity_main and we are trying to show the list of items in AlertDialog on Button click.

Generally, during the launch of our activity, the onCreate() callback method will be called by the android framework to get the required layout for an activity.

**Output**

When we run the above example using an android virtual device (AVD) we will get a result like as shown below.


This is how we can use AlertDialog control in android applications to show the list items with checkboxes in alert dialog based on our requirements in android applications.