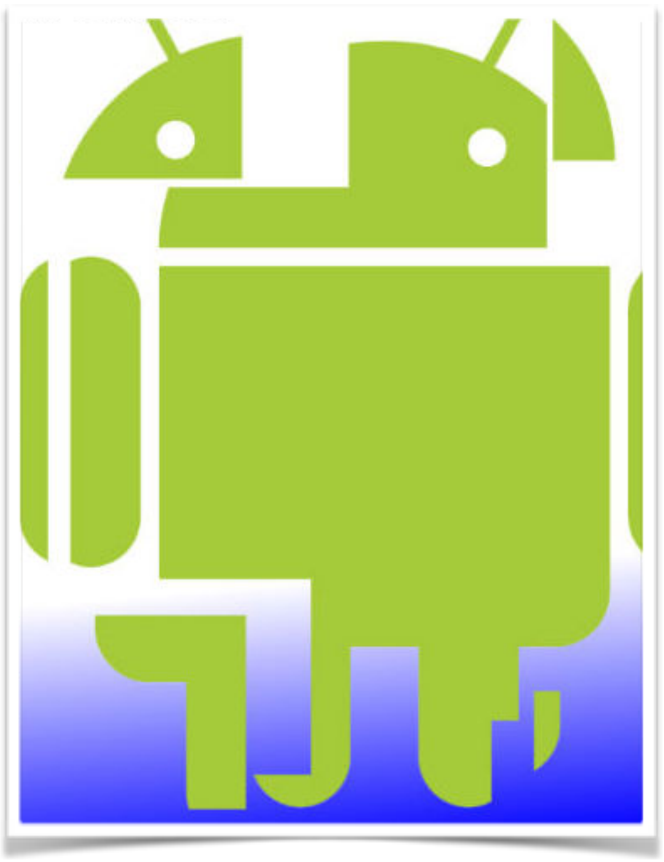
CS425 - Web and Mobile Software Engineering
Dr. Lilia Sfaxi
27 avril 2017

MedTech

Advanced Graphical Elements: Fragments and Menus

Android Lab 03



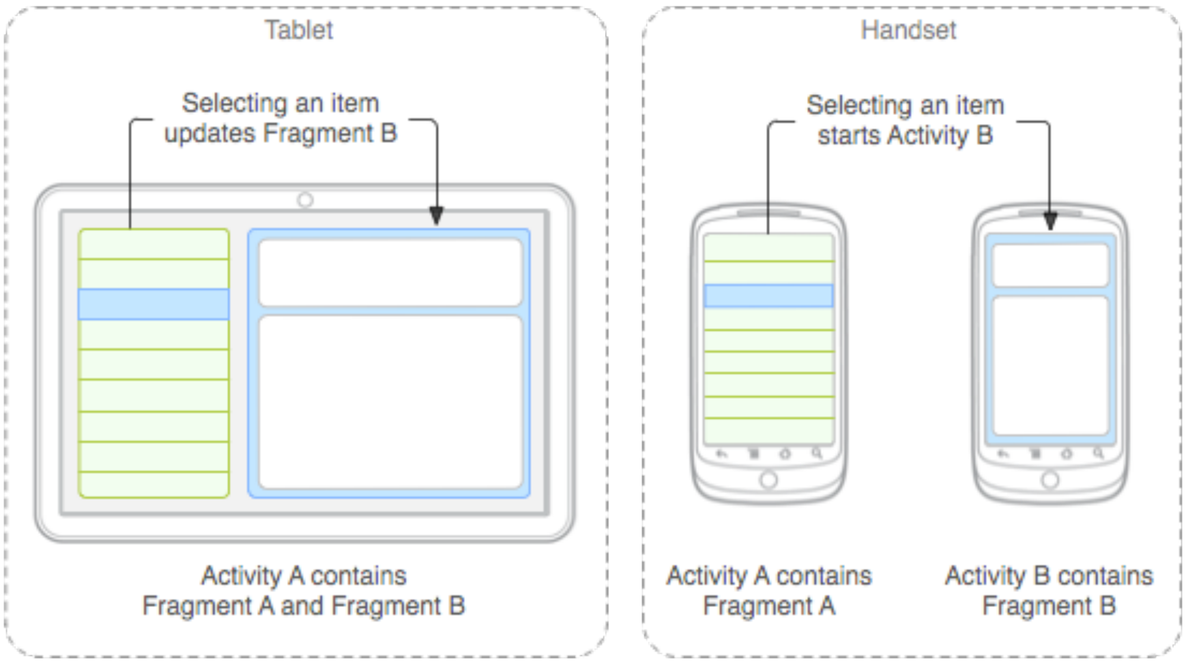
I. Fragments

1. Definition and Usage

A fragment represents a behaviour or a part of a graphical interface in an activity. It is possible to combine several fragments in a single activity to create an interface with multiple panels, and reuse a fragment in multiple activities.

A fragment has its own lifecycle, receives its own inputs and can be added, updated or deleted dynamically. It is always included in an activity and its lifecycle is directly affected by that of the activity containing it. As long as the container activity is executing, the fragment can be manipulated in an independent manner (added, updated or deleted). But, if the activity is destroyed, the fragment also is.

Fragments were initially created to support the design of interfaces adaptable to different screen sizes, enabling thus to different devices with wider screens (such as tablets) to use this space to combine and exchange graphical components.

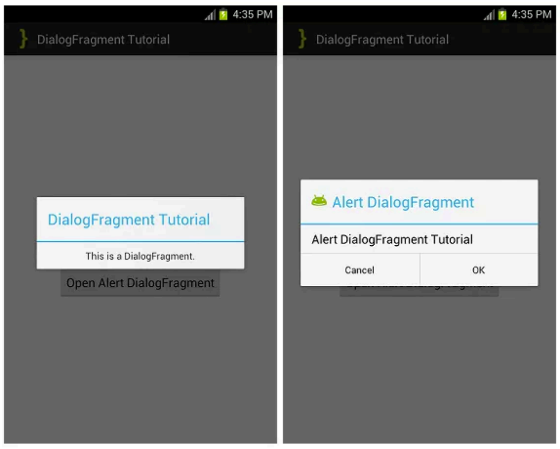

Tablet
Selecting an item updates Fragment B
Activity A contains Fragment A and Fragment B
Handset
Selecting an item starts Activity B
Activity A contains Fragment A
Activity B contains Fragment B

2. Fragment Creation

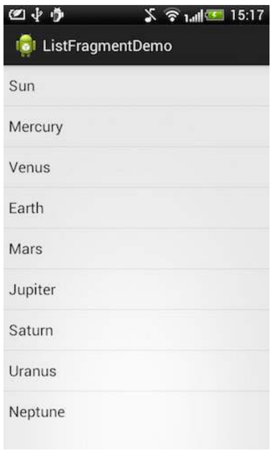
To create a fragment, you must create a class inheriting from *android.app.Fragment*, or one of its subclasses. We can cite for example:

- DialogFragment

Displays a modal dialog box instead of using methods of your activity, as it can be added to the activity's fragment stack.

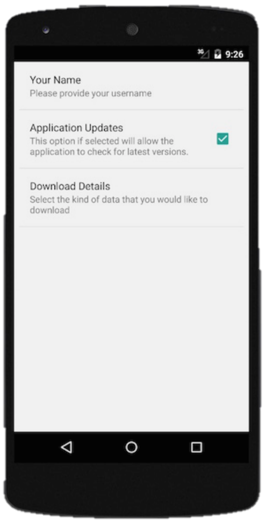


- ListFragment



Displays a list of elements managed by an adapter, and provides several methods to manage a listView.

- PreferenceFragment



9:26
Your Name
Please provide your username
Application Updates
This option if selected will allow the application to check for latest versions.
Download Details
Select the kind of data that you would like to download

Displays a hierarchy of Preference objects as a list. Creates an activity settings for your application.

To create a fragment, do the following steps:

- Create the fragment's layout

Start by defining a layout in a separate XML file. Implement then the transition method *onCreateView* to inflate the fragment's interface. It must return a View object representing the fragment's root, and it uses a LayoutInflater to build the objects' hierarchy from XML resources.

- Add the fragment to an activity

The fragment can be added statically or dynamically to the activity:

- **Statically**: declare a fragment in the XML layout of the activity, using the <fragment> element. Use the attribute *android:name* to point to the Java class of the fragment to instantiate in the layout.
- **Dynamically**: insert in the activity's code the created fragment in an existing ViewGroup container. To do this, a *Fragment Manager* is used.

The FragmentManager helps :

- Manipulating the existing fragments of your application thanks to the methods *frindFragmentById()* or *findFragmentByTag()*
- Manage the Back Stack, accessible via the *Back* button of the device.
  - Unstack the fragment with *popBackStack*
  - Associate a Listener to any changes in the stack, with *addOnBackStackChangedListener()*

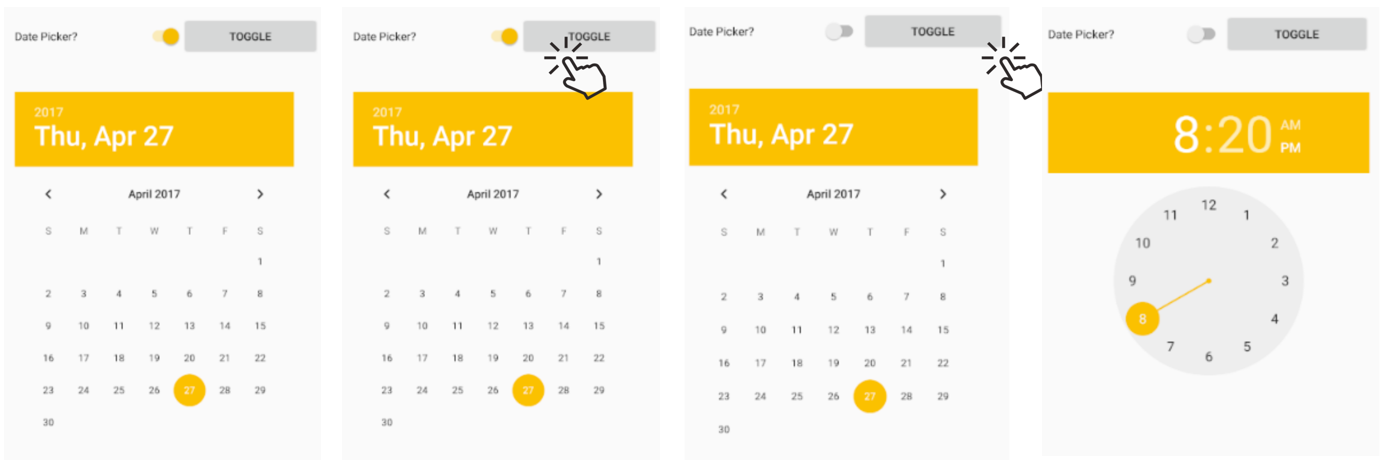
Managing the fragments with the fragment manager is possible thanks to a set of *FragmentTransactions*. A transaction is an operation on fragments (add, update, replace) in an activity, as a response to a user's action.

A transaction can be stored in the back stack of the activity, to enable the user to go back, thanks to the method *addToBackStack*.

II. Exercise 1: Fragments

1. Objective

The goal of this exercise is to create and replace fragments in an Android application. We need to obtain the following result:



2. Fragment Creation

To create the fragments:

- Create an application with a simple empty activity.
- Create the layout for the first fragment. To do this, create a file *frag_date.xml* under the directory *layout*.
- Insert an object DatePicker, using the colours of your choice.
- Repeat these steps to create a *frag_time.xml* layout containing a TimePicker object.
- Create a class: FragmentPicker extending the Fragment class.
- Implement the *onCreateView* method, where you must specify which fragment to use. To do this, use the following instruction:

```
inflater.inflate(R.layout.frag_date, container, false);
```

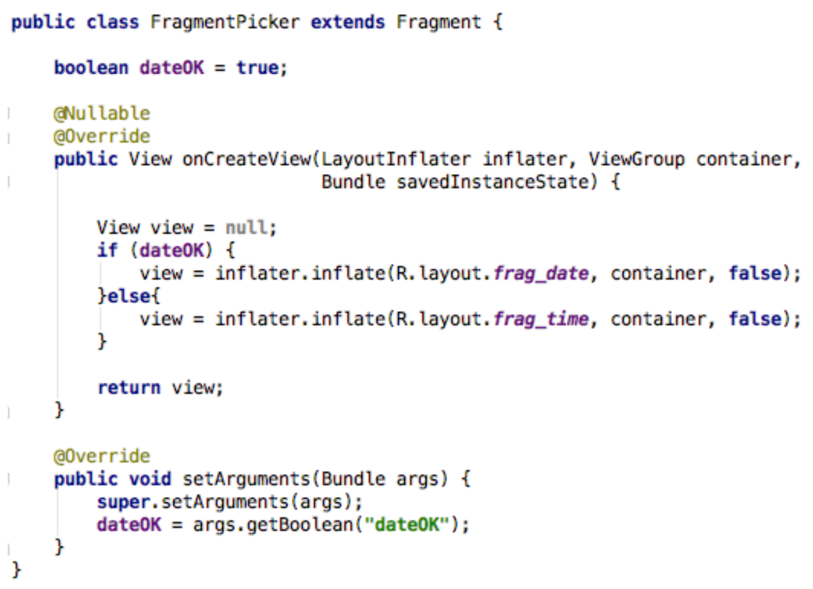
3. Main Activity Creation

To create the main activity:

- Create its layout. Insert a switch and a button one next to the other. The button's onclick method is called: *togglePicker*
- The goal is to click on the button *Toggle*, and to display the Date Picker is the switch is on, and the Time Picker if it is off. You should now implement a function *insertFragment*.

```
public void insertFragment(){
    Fragment fragment = new FragmentPicker();

    Bundle bundle = new Bundle();
    bundle.putBoolean("dateOK", s.isChecked());
    fragment.setArguments(bundle);

    FragmentManager fragmentManager = getFragmentManager();
    FragmentTransaction transaction = fragmentManager.beginTransaction();
    transaction.replace(R.id.frag_layout, fragment);
    transaction.commit();
}
```

  - The *frag_layout* represents the id of the LinearLayout that you will insert the fragment into.
  - The bundle object sends a set of key/value parameters to the fragment. In our case, we will send the state of the switch (checked or not) in a boolean parameter *dateOK*.
  - The fragment manager places the fragment in the right place, in the fragment element of the layout.
- This function will be called in **two places**: inside of the *onCreate* function, for initialisation, and in the method *togglePicker* when the button is clicked.
- To define which fragment to load, change the code of the FragmentPicker class as follows:



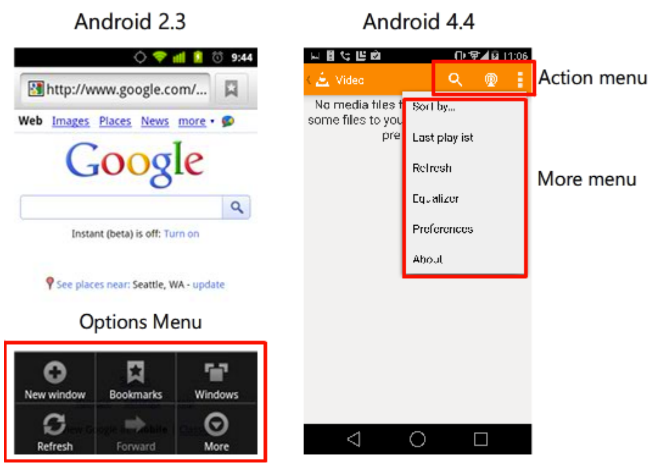
```java
public class FragmentPicker extends Fragment {

    boolean dateOK = true;

    @Nullable
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {

        View view = null;
        if (dateOK) {
            view = inflater.inflate(R.layout.frag_date, container, false);
        }else{
            view = inflater.inflate(R.layout.frag_time, container, false);
        }

        return view;
    }

    @Override
    public void setArguments(Bundle args) {
        super.setArguments(args);
        dateOK = args.getBoolean("dateOK");
    }
}
```

Activity 1: Create the application as indicated in the steps above. Launch the emulator to test your code.

III. Menus

Menu APIs were initially destined to define (among other things) the behaviour of the button menu of the telephone which, from the version 3.0, became obsolete. As a lot of terminals do not have this button anymore, it was necessary to replace these menus.

From now on, the functionalities of the options menu are defined in the Action Bar:



The Action Bar is a characteristic that identifies the location of the user in the app et provides him with actions and navigation modes. It provides a space designed to give an identity to your application and a set of important actions the user can perform, such as search or back navigation.

In the default applications created with Android Studio, an action bar is automatically added. The content of this bar is defined in the XML file found in the res/menu directory.

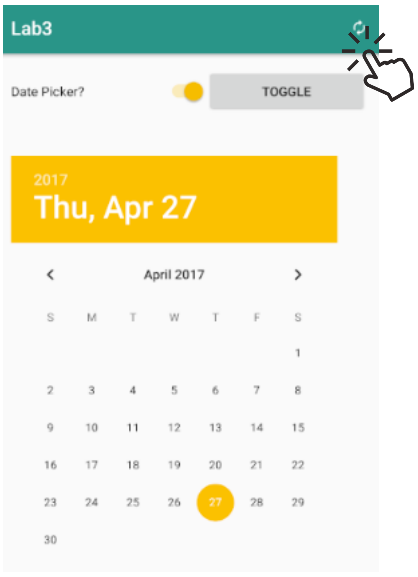
To define the behaviour of the elements in the action bar, you must implement the (automatically generated) methods:

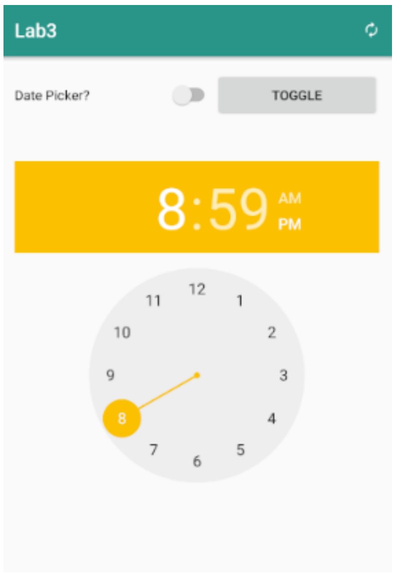
- *onCreateOptionsMenu*: builds the menu by calling the content of the XML menu.
- *onOptionsItemSelected*: defines the behaviour of each of the actions of the bar.

IV. Exercise 2: Action Bar

1. Objective

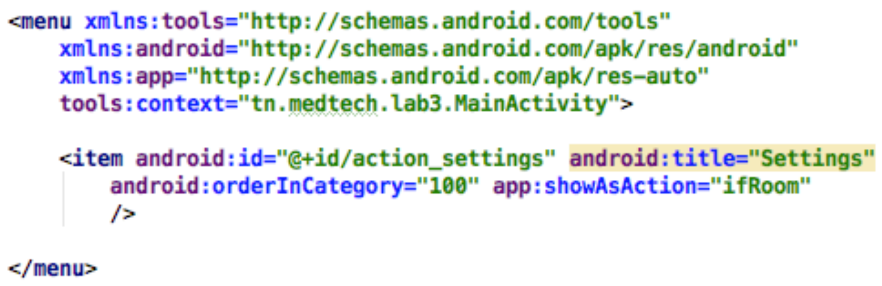
The main goal of this part is to add elements to the action bar of the preceding application. The final result should look like the following:





2. Implementation

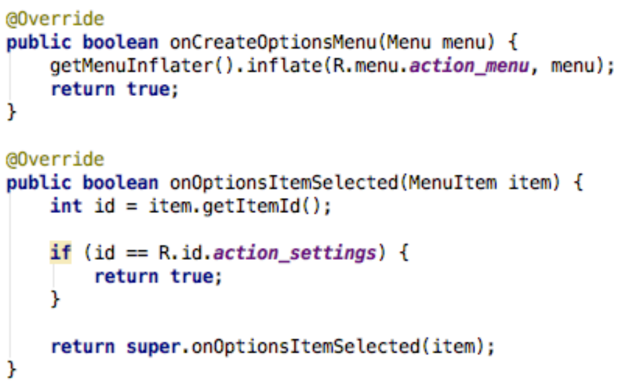
- Create a Menu Item with an icon of your choice:



```
<menu xmlns:tools="http://schemas.android.com/tools"
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    tools:context="tn.medtech.lab3.MainActivity">

    <item android:id="@+id/action_settings" android:title="Settings"
        android:orderInCategory="100" app:showAsAction="ifRoom"
        />

</menu>
```

The attribute *showAsAction* configures the appearance of the element in the bar, or in the Action Overflow ( ). Several values are possible: never, always or ifRoom.

- Implement the functions *onCreateOptionsMenu* and *onOptionsItemSelected* as follows:

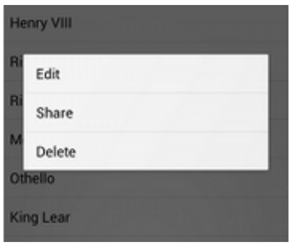


```java
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.action_menu, menu);
    return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    int id = item.getItemId();

    if (id == R.id.action_settings) {
        return true;
    }

    return super.onOptionsItemSelected(item);
}
```

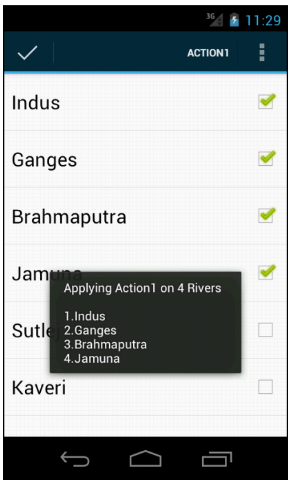
Activity 2: Create the Action Bar as explained above.

V. Contextual Mode

A contextual menu is a floating menu that shows when a user long-clicks on an element, for example. It provides actions affecting the clicked content or its container.



Since the version 3 of Android, this menu was replaced by a contextual mode, enabling to perform actions on the selected content. It displays actions that affect the selected content in a bar on top of the screen that replaces the Action Bar, and enables the user to select one or several elements.



Applying Action1 on 4 Rivers

1.Indus
2.Ganges
3.Brahmaputra
4.Jamuna

In general, we switch to the contextual mode if the user long-clicks on an element, or selects a checkbox or any equivalent view. The contextual bar takes momentarily the place of the action bar.

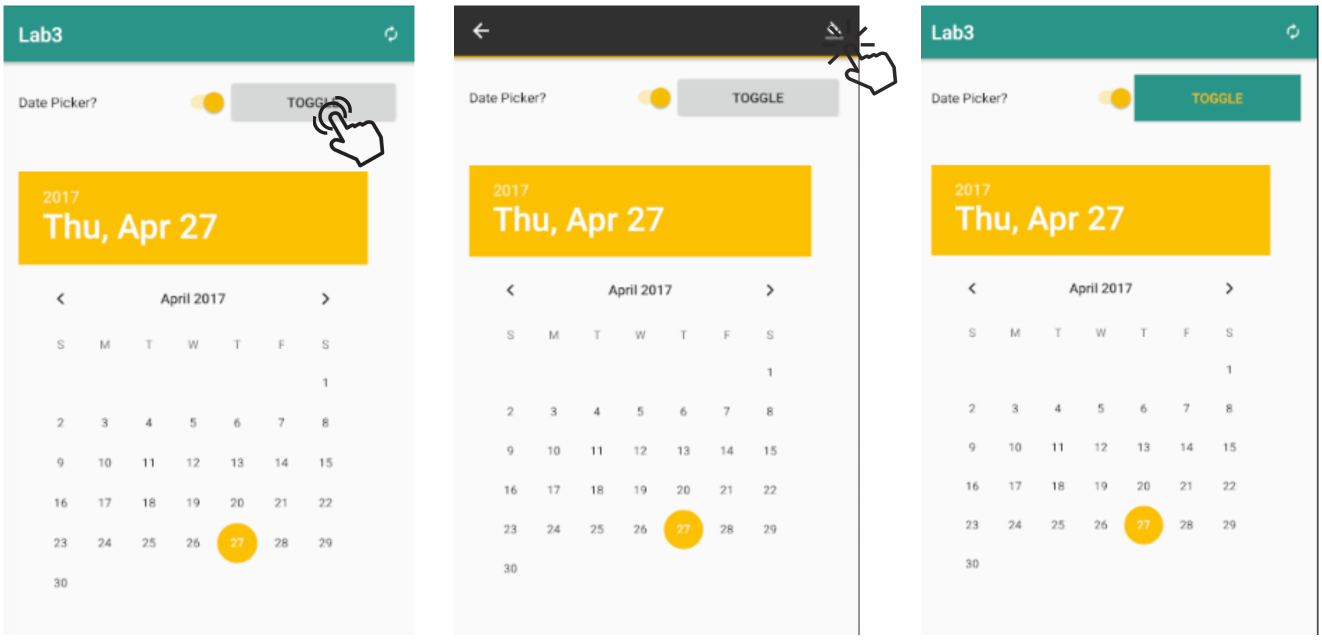
It is possible to activate this mode for an individual widget, or for a group of elements grouped in a list or a grid, for example.

VI. Exercise 3: Contextual Mode

1. Objective

The goal of this part is to activate the contextual menu if the user long-clicks on the button.

In the contextual bar, the user will be able to change the colour of the button.



2. Define the Contextual Menu

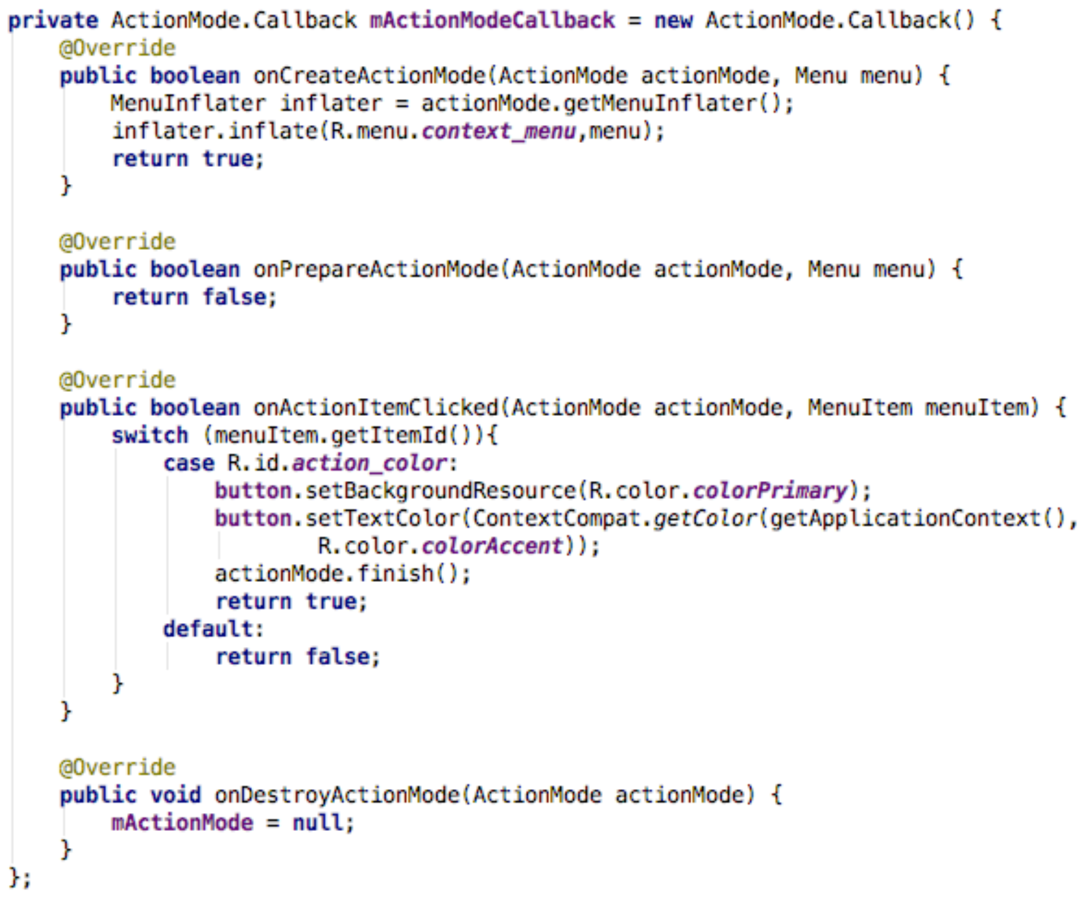
To define the content of the contextual bar, add a new XML file in the *menu* directory. We call it *context_menu.xml*.

```
<menu xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:android="http://schemas.android.com/apk/res/android"
    >

    <item android:title="Change Color"
        android:orderInCategory="100"
        app:showAsAction="always"
        android:id="@+id/action_color"
        android:icon="@drawable/ic_format_color_fill"
        />
</menu>
```

3. Define the Callback

Create the object *mActionModeCallback* that defines the behaviour at the emergence of the contextual mode in the activity. The instantiation of this object requires the implementation of several methods, such as:

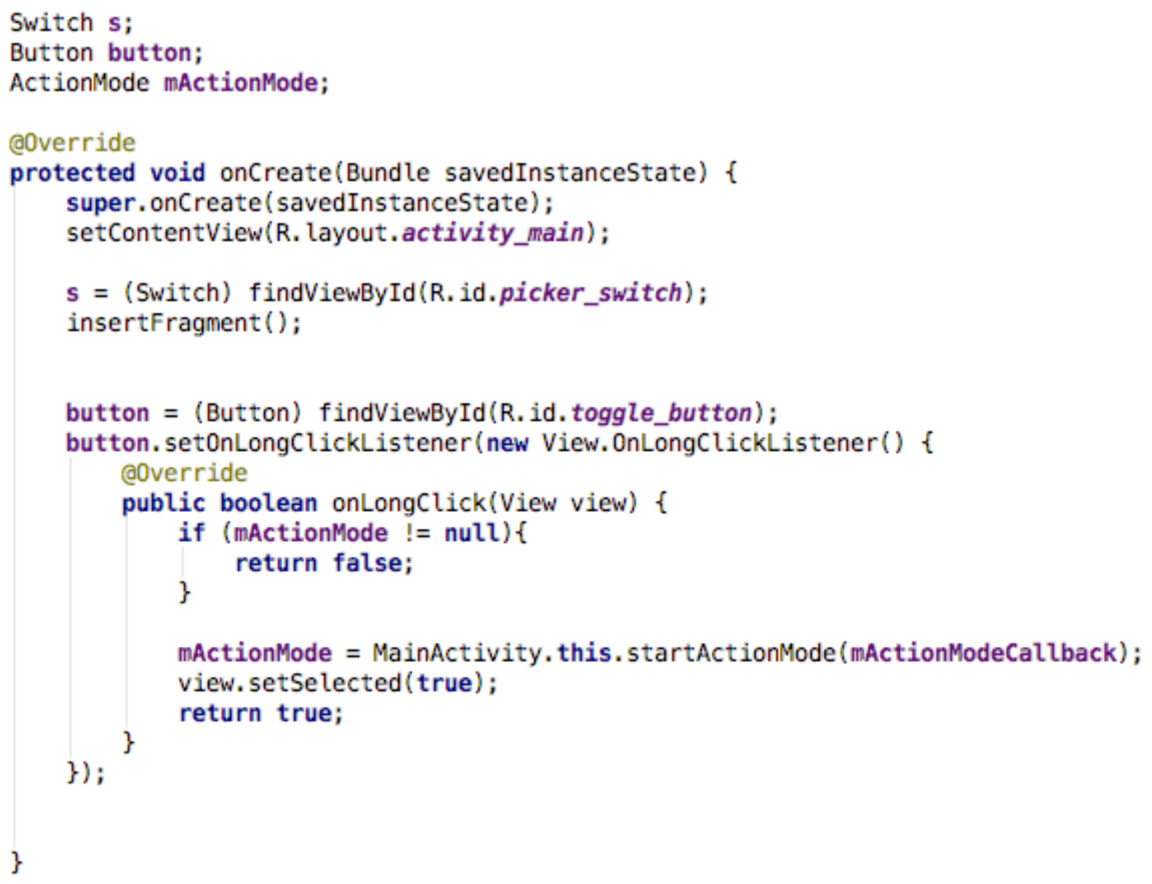
- *onCreateActionMode* : behaviour when creating the contextual menu.
- *onActionItemClicked* : behaviour when clicking on an element of the contextual bar.
- *onDestroyActionMode* : behaviour when the contextual bar disappears.



```java
private ActionMode.Callback mActionModeCallback = new ActionMode.Callback() {
    @Override
    public boolean onCreateActionMode(ActionMode actionMode, Menu menu) {
        MenuInflater inflater = actionMode.getMenuInflater();
        inflater.inflate(R.menu.context_menu,menu);
        return true;
    }

    @Override
    public boolean onPrepareActionMode(ActionMode actionMode, Menu menu) {
        return false;
    }

    @Override
    public boolean onActionItemClicked(ActionMode actionMode, MenuItem menuItem) {
        switch (menuItem.getItemId()){
            case R.id.action_color:
                button.setBackgroundResource(R.color.colorPrimary);
                button.setTextColor(ContextCompat.getColor(getApplicationContext(),
                        R.color.colorAccent));
                actionMode.finish();
                return true;
            default:
                return false;
        }
    }

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

4. Associate the Callback to an Event

Finally, you have to associate the callback to an event, which is the long-click on the button. To do this, write the following code in your activity:



```java
Switch s;
Button button;
ActionMode mActionMode;

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

    s = (Switch) findViewById(R.id.picker_switch);
    insertFragment();


    button = (Button) findViewById(R.id.toggle_button);
    button.setOnLongClickListener(new View.OnLongClickListener() {
        @Override
        public boolean onLongClick(View view) {
            if (mActionMode != null){
                return false;
            }

            mActionMode = MainActivity.this.startActionMode(mActionModeCallback);
            view.setSelected(true);
            return true;
        }
    });


}
```

Activity 3: You know what you have to do…