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

MedTech

Basic Graphical Elements and Intents

Android Lab 01

I. Views and ViewGroups

The class View represents the main class for creating a graphical element in Android. It is the main class for all widgets, used to create interactive graphical components (buttons, text views, text inputs,…).

A view occupies a rectangular space on the screen, responsible for the management of the events initiated by the user.

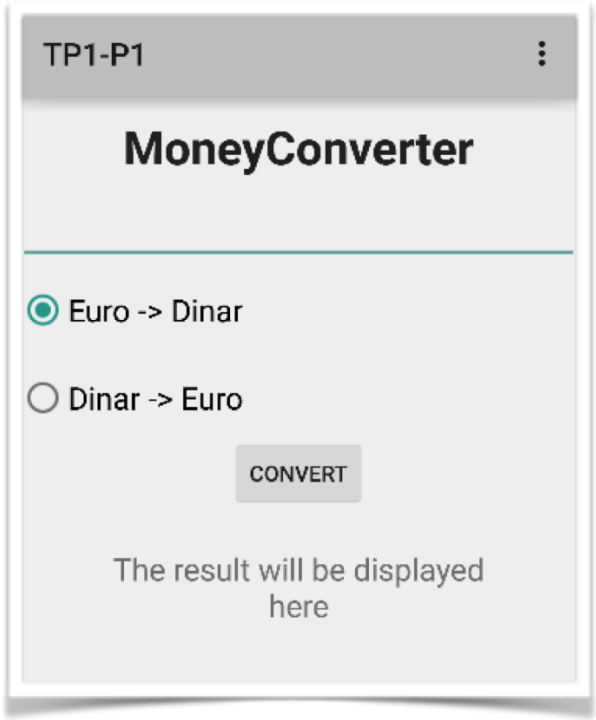
A child class ViewGroup is defined, representing the main class for all layouts, invisible containers for Views or ViewGroups, and defining their placement on the scree.

Each element in a graphical interface is either a View or a ViewGroup. The dispositions of these elements in the screen are defined with Layouts. A layout is a ViewGroup that groups a set of widgets or other layouts, enabling thus to define their dispositions in the screen. We can site for example, FrameLayout, LinearLayout, RelativeLayout,…

II. Exercise 1: Basic Graphical Components

1. Objective

This first part of the lab aims to create a simple conversion application, having an interface looking like the one below.



Activity 1: Start by creating the graphical interface. Launch the emulator and check that the result is correct.

In what follows, we define the steps to define the behaviour of this interface.

2. Behaviour of the Button

Having a button defined in the XML layout, which identifier is *b_convert*. To manage this button, three ways are available:

- Method 1: Rewriting the Click Listener method

In Java, a listener is an object enabling the programmer to react to actions of the user (mouse click, keyboard, etc.). In our case, we need a listener for a click on the button, called onClickListener. Listener is an interface, providing methods that must be implemented by the programmer for each element. For example, the onClickListener contains a method onClick, that we must implement to represent the behaviour of the button.

1. Create an attribute in your activity of type Button:

```
private Button bConvert;
```

2. In the method onCreate, initialise the new attribute bConvert by associating the graphical button defined in the XML layout:

```
this.bConvert = (Button) this.findViewById(R.id.b_convert) ;
```

3. Use the following code to define a behaviour for the click on the button bConvert. It can be done in the method onCreate, to be enabled at the creation of the activity.

```
this.bConvert.setOnClickListener(new OnClickListener() {
        @Override
        public void onClick(View v) {
                //behaviour of the button
        }
});
```

- Method 2: Associating a method to the button

An easier manner to define the behaviour of the button, is to associate a special method for the click on the button. The problem is that this solution works only for the click, but is not provided for other events, like a long click, for example.

1. In the XML layout file, as an attribute of the button element, add:

```
android:onClick="convert"
```

   This attribute indicates that a method called *convert* is defined in the activity, and represents the behaviour on the click of the button.

2. In the code of the activity, add the following method:

```
public void convert(View v){
        // behaviour of the button
}
```

The method *convert*, has a particular signature: it must be public, return void, and take as a parameter a View object. It has the same signature as the onClick of the first method. The

View object passed as a parameter represents the graphical object that was clicked, hence the button. If you use this method, it is useless to create a Button attribute in the activity, you will not need it.

- Method 3: Implementing the OnClickListener Interface

It is possible to use inheritance to implement the onClick method, without calling setOnClickListener each time. This method can be useful if you want to group the behaviours of a set of elements in the same class.

1. Your activity must implement the OnClickListener interface.

```
public class MoneyConverter extends Activity implements OnClickListener {…}
```

2. Create the attribute *bConvert* and associate if the XML element *b_convert* as shown in the first method.

3. Define the current activity as the listener to this button.

```
bConvert.setOnClickListener(this);
```

4. Override the onClick method directly in your activity:

```
public void onClick(View v) {
        if (v.getId()==R.id.b_convert){
                //behaviour of the button
        }
}
```

Be careful, this onClick method will be common to all the graphical elements that use the current activity as an OnClickListener. To distinguish which element was clicked, test the id of the view (as shown in the code above).

3. Behaviour of an EditText

An EditText is a graphic object that enables the application to receive input from the user.

1. Create an attribute in your activity :

```
private EditText eEntry;
```

2. In the onCreate method, initialise the attribute:

```
this.eEntry = (EditText) this.findViewById(R.id.e_entry) ;
```

3. Define its behaviour. For an input element, the main functionalities are:

   - Reading the content:

```
String s = eEntry.getText().toString();
```

   - Modifying the content:

```
eEntry.setText("New Text »);
```

4. Behaviour of a TextView

A TextView is a graphical objet that displays an immutable string. It is used by the application in the same manner as the EditText.

5. Behaviour of a Radio Button

A radio button is a box with two states: checked or unchecked. Radio buttons are in general used in a RadioGroup, where only one button can be checked.

1. Create an attribute RadioButton in your activity and associate it to the appropriate XML element.

2. To test the state of you button, call the method *isChecked()*.

```
if (rDinarEuro.isChecked() ){
        // code
}
```

3. To dynamically change the state of a radio button, use *setChecked(state)*:

```
radio1.setChecked(true);
radio2.setChecked(false);
```

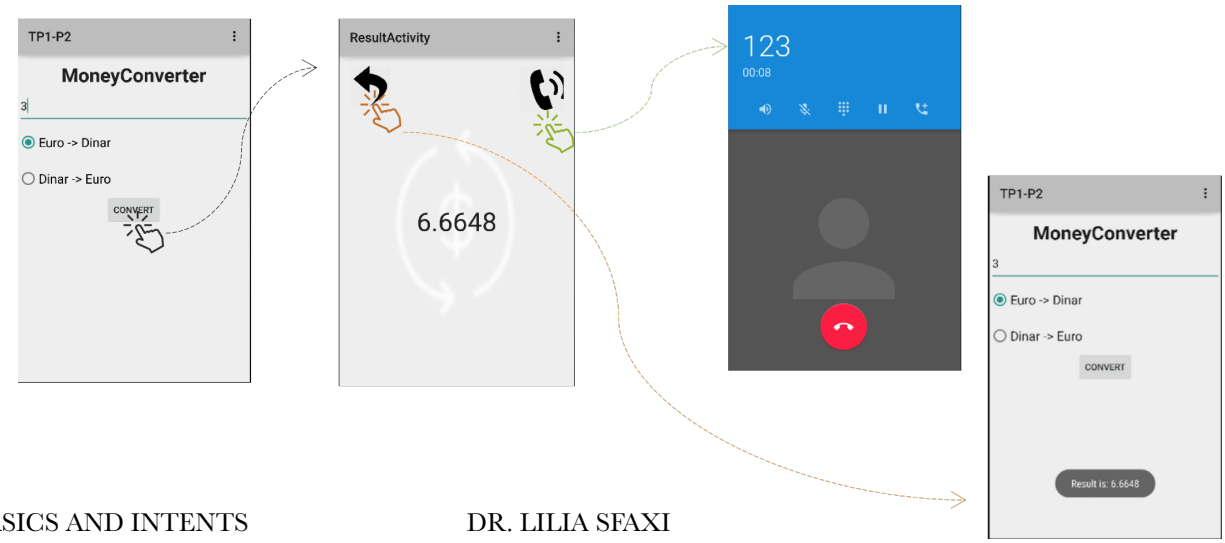
There is no need to create a Java object for the RadioGroup. If radio1 and radio2 of the previous example are in the same RadioGroup, it is unnecessary to change both their states: if you check one of them, the other will be automatically unchecked.

Activity 2: As shown above, define the behaviour of the elements of your interface, using the method of your choice.

III. Exercise 2: Intents

1. Objective

The objective of this part is to change the behaviour of the previous application to obtain the following result:



2. Intents: Definition

An Android application can contain a lot of activities. An activity uses the method setContentView to be associated to a graphical layout. Basically, activities are independent from one another, but they can collaborate to exchange data and actions.

Typically, one of the activities is defined as a launcher activity, the one that is displayed at the bootstrapping of the application.

All the activities interact in an asynchronous mode. To switch from an activity to another, the first creates an intent. It is a message that can be used to trigger an action from another component of the application. It can invoke activities, broadcast receivers or services. The methods used to do so are:

- startActivity(intent) : starts an activity
- sendBroadcast(intent) : sends an intent to all the concerned Broadcast receivers
- startService(intent) ou bindService(intent, …) : launches a service

An intent contains informations used by the Android system:

- The name of the component to start
- The action to realise: ACTION-VIEW, ACTION_SEND…
- The data: a URI referencing the data on which the action will be performed
- The category: additional information about the type of the components that handle the intent: CATEGORY-BROWSABLE, CATEGORY-LAUNCHER…
- The extras: key-value pairs containing information to be sent to the target component
- The flags: Metadata class for this intent, showing for example how to launch an activity and what to do once it is launched.

There are mainly two types of intents: Explicit and Implicit intents

3. Explicit Intents

Explicit intents specify the component to start by name (complete name of the class). They enable starting a component of your own application, as the name is known. For example: start another activity as a response to an action of the user.

The main arguments of an explicit intent are:

- The context starting the intent. In general, it is the activity from which it was launched.
- The target component.

It is typically called like the following:

```
Intent myActivityIntent =
        new Intent (StartClass.this, EndClass.class) ;
 startActivity (myActivityIntent) ;
```

The data shared between activities can be sent as extras. An extra is a key-value element, sent to the target component. It is created as follows:

```
myActivityIntent.putExtra("clef","valeur");
```

Of course, all extras must be attached to the intent before calling startActivity.

It is also possible to start an activity with a result, by establishing a bi-directional link between activities. To receive a result from another activity, call *startActivityForResult* instead of *startActivity*.

Destination activity must of course be designed to send back a result once the operation performed, and this is done using an intent. The main activity will receive it and will hand it using the predefined method *onActivityResult*.

This is an example of an intent with a result:

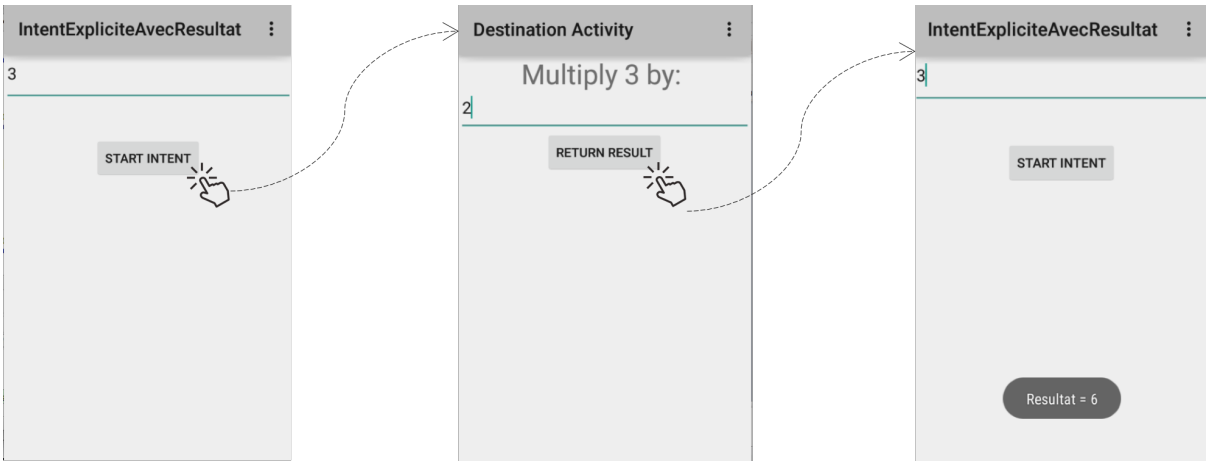
Activity1:

```
public void start(View v){
    Intent i = new Intent(this,DestinationActivity.class);
    i.putExtra("val",Integer.valueOf(texte.getText().toString()));
    startActivityForResult(i, REQUEST_CODE);
}

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent intent) {

    if (requestCode == REQUEST_CODE) {
        if(resultCode == RESULT_OK){
            Toast.makeText(this,"Resultat = "+intent.getIntExtra("resultat",0),Toast.LENGTH_LONG).show();
        }
        if (resultCode == RESULT_CANCELED) {
            Toast.makeText(this,"Pas de Resultat !",Toast.LENGTH_LONG).show();
        }
    }
}
```

Activity2:

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_destination);

    tv = (TextView)findViewById(R.id.show);
    mult = (EditText) findViewById(R.id.multEdit);

    valueReceived = getIntent().getIntExtra("val",0);
    tv.setText("Multiply "+valueReceived+" by: ");

}


public void retour (View v){
    Intent returnIntent = new Intent();
    if (mult.getText()!= null) {
        int result = valueReceived * Integer.valueOf(mult.getText().toString());
        returnIntent.putExtra("resultat", result);
        setResult(RESULT_OK, returnIntent);
    }else{
        setResult(RESULT_CANCELED, returnIntent);
    }
    finish();
}
```

The result displayed by this code is the following:



Activity 3: Start by creating the interface of the second activity. Implement then the code of the back button, that enables to come back to the previous activity to display the result in a toast.

4. Implicit Intents

Implicit intents don't call specific components, but declare an action to realise. They enable a component of an application to call another component, even though it is in another application.

Example: show a position in a map.

The main arguments of an implicit intent are :

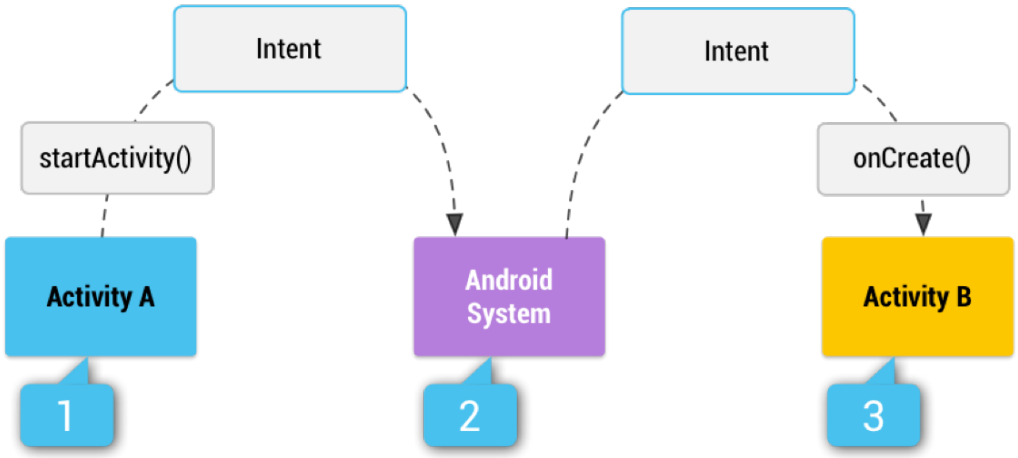
- The action to realise, which can be predefined (ACTION-VIEW, ACTION_EDIT…) or created by the user.
- The data: the main data we are sending, like the phone number to call.

It is typically called as follows:

```
Intent myActivityIntent = new Intent (<action>, <donnee>) ;
startActivity(myActivityIntent) ;
```

An implicit intent works as follows:

1. The activity A creates an intent with an action and sends it as a parameter of startActivity.
2. The Android system looks for all the applications to find an intent filter corresponding to this intent.
3. When a correspondance is found, the system starts the activity B, by invoking its onCreate method and sending the intent to it.



Implicit intents use the notion of intent filter to find the target activity to start. An intent filter is an expression in the manifest file of an application, which specifies the type of intent that the component wants to receive. It enables to other activities to launch directly your activity by using a certain intent.

If you don't declare intent filters in your activity, it can't be started unless you use an explicit intent. It is nonetheless recommended not to declare intent filters for services, for security reasons.

This is an example of an intent filter:

```
<activity
    android:name=".PopupActivity"
    android:label="Popup" >
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

These are some examples of commonly used actions :

Action	Data	Description
ACTION_DIAL	tel:123	Displays the telephone dialer with the number 123
ACTION_VIEW	http://www.google.com	Displays the Google page in a browser
ACTION_EDIT	content://contacts/people/2	Edits the information on the person of your contacts book with the id 2
ACTION_VIEW	content://contacts/people/2	Displays an activity that displays the information of the contact with id 2
ACTION_VIEW	content://contacts/people	Displays the contact list. The selection of a contact can display the details in a new intent

Most of the actions need permissions to add in the manifest file. For example, to authorise your activity to start a call, add the following line in the manifest:

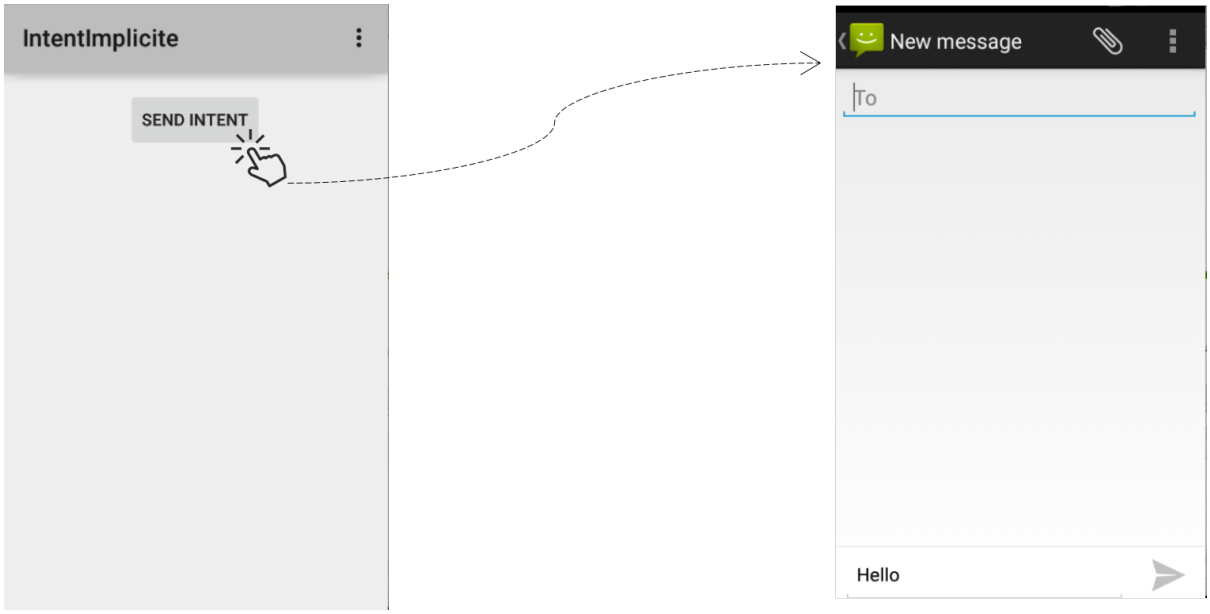
```
<uses-permission android:name="android.permission.CALL_PHONE">
</uses-permission>
```

An example of an implicit intent enabling to send a text message:

```
public void send(View v){
    // Create the text message with a string
    Intent sendIntent = new Intent();
    sendIntent.setAction(Intent.ACTION_SEND);
    sendIntent.putExtra(Intent.EXTRA_TEXT, "Hello");
    sendIntent.setType("text/plain");

    // Verify that the intent will resolve to an activity
    if (sendIntent.resolveActivity(getPackageManager()) != null) {
        startActivity(sendIntent);
    }else{
        Toast.makeText(this,"The send action could not be performed!",Toast.LENGTH_SHORT).show();
    }
}
```

The *resolveActivity* method helps avoiding a crash of the application if the called activity does not exist. It is a kind of exception handling.

The result will be as follows:



Activity 4: Implement the code of the call button, that enables to start a phone call to a predefined number.