

# Mobile Computing

## Practice # 2c

### Android Applications – Interface (in Kotlin)

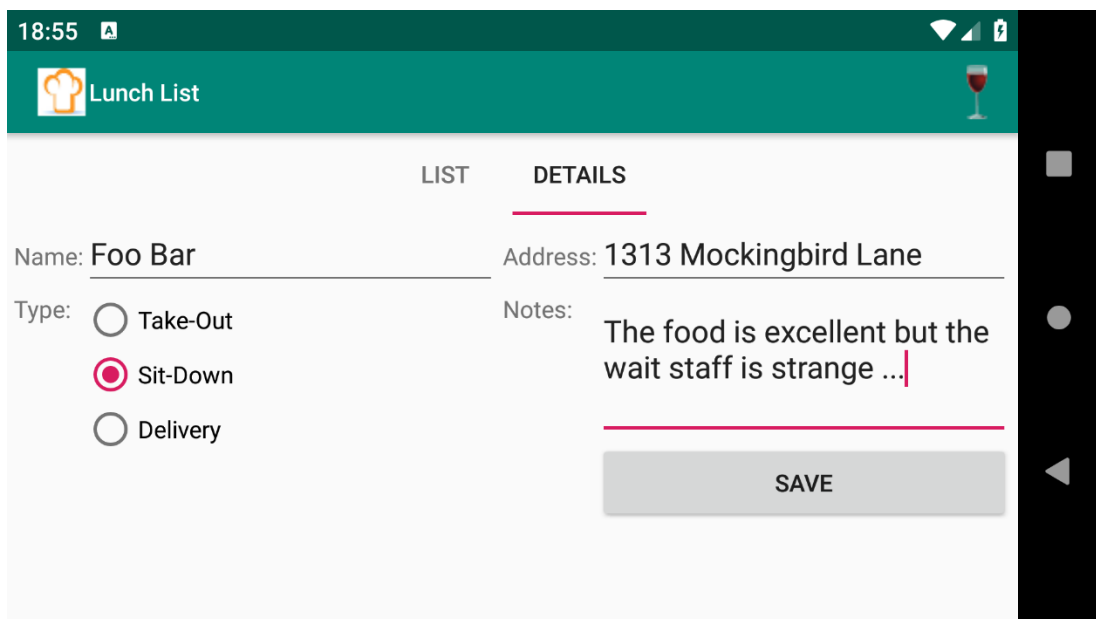
One more step in the restaurants application.

1. Design an alternative layout for showing up in landscape mode.

Our current layout is not very good in landscape orientation. You can improve it putting a **ScrollView** around the **Details Tab**, but it is better to design a new layout just for landscape mode (you should not have ever Views outside of the available space).

- a. First, create a new layout file with the same name (*activity\_main.xml*), but with an orientation (landscape) qualifier. This will create a new resource directory (*LunchList/res/layout-land/*) and put the file inside (use File->New->Layout resource file)
- b. Then, copy the contents of the original layout file and change it to have the following characteristics:
  - Use a table of four columns, with columns #1 and #3 as stretchable
  - Put the name and address labels and edits on the same row
  - Put the type, notes, and Save button on the same row, with the notes and Save button stacked via a **LinearLayout**.
  - Make the notes three lines instead of two, since we have the room
  - **[Optionally]** fix the maximum width of the **EditText** widgets to 130 scaled pixels (sp), so they do not automatically grow outlandishly large if we type a lot]
  - Add a bit of padding in places to make the placement of the labels and fields look a bit better

The Details tab of the new design should look like the following. The List tab can remain the same.



Note that we did not create a landscape version of our row layout (*row.xml*). Android, upon not finding one in *LunchList/res/layout-land/*, will fall back to the one in *LunchList/res/layout/*. Since we do not really need our row to change, we can leave it as is.

Note also that when you change the screen orientation, your existing restaurants will vanish. That is because we are not persisting them anywhere, and rotating the screen, by default, destroys and recreates the activity. Android automatically saves, before destroying an activity in a configuration change, the state of some interface controls that are not based on other variables (for instance Lists are based on external arrays). The values of internal activity variables are lost because those are not automatically saved.

To solve this last problem do the following:

Use **onSaveInstanceState()** to save the current contents of one of the activity variables (the **current** variable), and restore it in **onRestoreInstanceState()**. It could be also restored in the **onCreate()** method testing if the Bundle parameter is not null (after the screen was rotated). Complex variables should be Serializable or **Parcelable** (a more compact form of serialization) before we can save them on a Bundle. For this exercise just implement the Serializable interface for the Restaurant class.

We will, at this point, not cover the restaurant list array – you will still lose all existing restaurants on a rotation event. For solving that issue, we will later persist our restaurant list, for instance in a database.

2. We will now add support for both creating new restaurants and editing ones that were previously entered. Along the way, we will get rid of our tabs, splitting the application into two activities: one for the list, and one for the details form.
  - a. Create a new **Empty Activity** to serve as our details form. Call it **DetailsActivity.kt**. Edit its content to have only the icon in the **ActionBar** as the **MainActivity**:

```
class DetailsActivity: AppCompatActivity() {  
    override onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        ...  
    }  
}
```

- b. This Activity should be added to the project manifest inside the <application> tag. If it was created by Android Studio using the New Activity option, that should be already in place.

```
<activity android:name=".DetailsActivity">  
</activity>
```

- c. To share information between activities we can use several techniques, like top-level properties or objects. Nevertheless every Android application has a singleton object that represents the application. It's easy to create our own application class in the project and to access that object from the activities. The **Application** object remain in memory with the app process.  
We will need to share between our two activities the array list of restaurants (the model (restaurants) of the application), the adapter, for changing data in the second activity, and the current selected restaurant.

Add a new class to the project, derived from **Application**, as follows:

```
class LunchApp: Application() {  
    val rests: ArrayList<Restaurant> = ArrayList()  
    var current: Restaurant? = null  
    var adapter: RestaurantAdapter? = null  
}
```

The **RestaurantAdapter** is class that should be instantiated in the **MainActivity**, to work with the **ListView** control, displayed on that Activity layout.

The new name of the **Application** class must be registered in the manifest as an android:name property of the application tag:

```
<application  
    android:name=".LunchApp"  
    ...
```

- d. Then, we need to refactor our **MainActivity** activity, simplifying also the layout (activity\_main.xml) to have only the **ListView**. To let the user see something when the list is empty we can also define a **TextView** (not visible when the **ListView** is not empty) to show a message in that situation. We should then have as *activity\_main.xml* the following:

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <ListView
        android:id="@+id/listview"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
    <TextView
        android:id="@+id/empty_list"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/lv_empty_msg"/>
</LinearLayout>
```

The **onCreate()** callback instantiates the **Adapter** and associates it with the list, setting the **TextView** as the message to show when the list is empty:

```
private val app by lazy { application as LunchApp }

app.adapter = RestaurantAdapter(this, app.rests)

val list = findViewById<ListView>(R.id.listview)
list.adapter = app.adapter
list.emptyView = findViewById(R.id.empty_list)
list.setOnItemClickListener { _, _, pos, _ -> onRestItemClick(pos) }
```

- e. Add a new menu item to add a new restaurant.  
On the **onOptionsItemSelected()** menu handler, add the code to start the new **DetailsActivity**:

```
when (item.itemId)
    ...
    R.id.add -> {
        startActivity(Intent(this, DetailsActivity::class.java))
        true;
    }
```

- f. The **ListView** listener responding for clicks on a list item should contain:

```
app.current = app.rests[pos]
startActivity(Intent(this, DetailsActivity::class.java).putExtra(ID_EXTRA, pos))
```

When the user selects a restaurant, the **Details activity** should start, showing the restaurant details and allowing the user to edit the data. To distinguish this situation from the one where the user wants to add a new restaurant, this time the intent to invoke the **Details activity** will transport the position of the selected restaurant. The string acting as key is defined as a **const String**.

- g. To complete the **Details** functionality start by creating a new layout file, *activity\_details.xml*, and put there what was inside the Details tab, which should be by now, something like:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/details"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:padding="5dp"
    android:stretchColumns="1">
    <TableRow
        android:layout_width="match_parent"
        android:layout_height="wrap_content">
        <TextView android:text="@string/tv_name" />
        <EditText
            android:id="@+id/ed_name"
            android:inputType="text" />
    </TableRow>
    <TableRow
        android:layout_width="match_parent"
        android:layout_height="wrap_content">
        <TextView android:text="@string/tv_address" />
        <EditText
            android:id="@+id/ed_address"
            android:inputType="text" />
    </TableRow>
    <TableRow>
        <TextView android:text="@string/tv_type" />
        <RadioGroup android:id="@+id/rg_types">
            <RadioButton
                android:id="@+id/rb_take"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:checked="true"
                android:text="@string/rb_take" />
            <RadioButton
                android:id="@+id/rb_sit"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:text="@string/rb_sit" />
            <RadioButton
                android:id="@+id/rb_delivery"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:text="@string/rb_delivery" />
        </RadioGroup>
    </TableRow>
    <TableRow>
        <TextView android:text="@string/tv_notes" />
        <EditText
            android:id="@+id/ed_notes"
            android:gravity="top"
            android:inputType="textMultiLine"
            android:lines="2"
            android:maxLines="2" />
    </TableRow>
    <Button
        android:id="@+id/bt_save"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/bt_save" />
</TableLayout>
```

Have the **setContentView()** call in **onCreate()** in **Details** load this layout (*R.layout.activity\_details*). In the *res/layout-land* directory change the name of *activity\_main.xml* to *activity\_details.xml* and eliminate the **TabLayout** stuff, leaving only its contents.

- h. Add now the following data members to the **Details class**, as any other considered useful:

```
private val app by lazy { application as LunchApp }
private val rPos by lazy { intent.getIntExtra(ID_EXTRA, -1) }
```

and complete the Details **onCreate()** method with:

```
if (rPos != -1)
    load();
```

The private **load()** method should fill the form views with the application **current** Restaurant (if the intent brings a valid position, the user has clicked a restaurant in the Main activity which has filled the **LunchApp** shared **current** property).

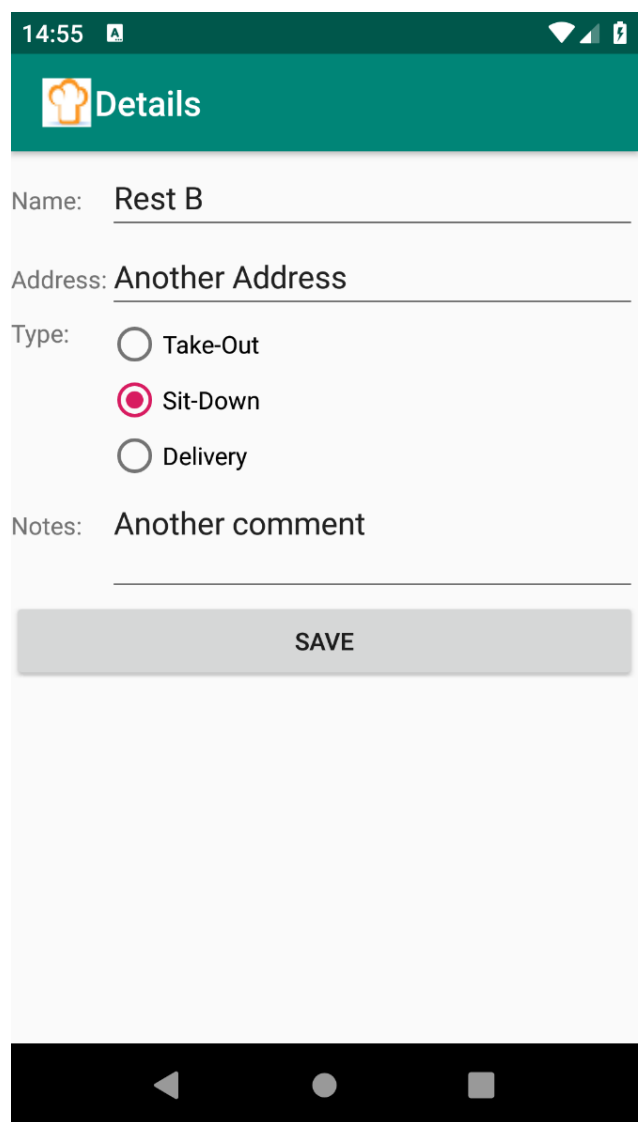
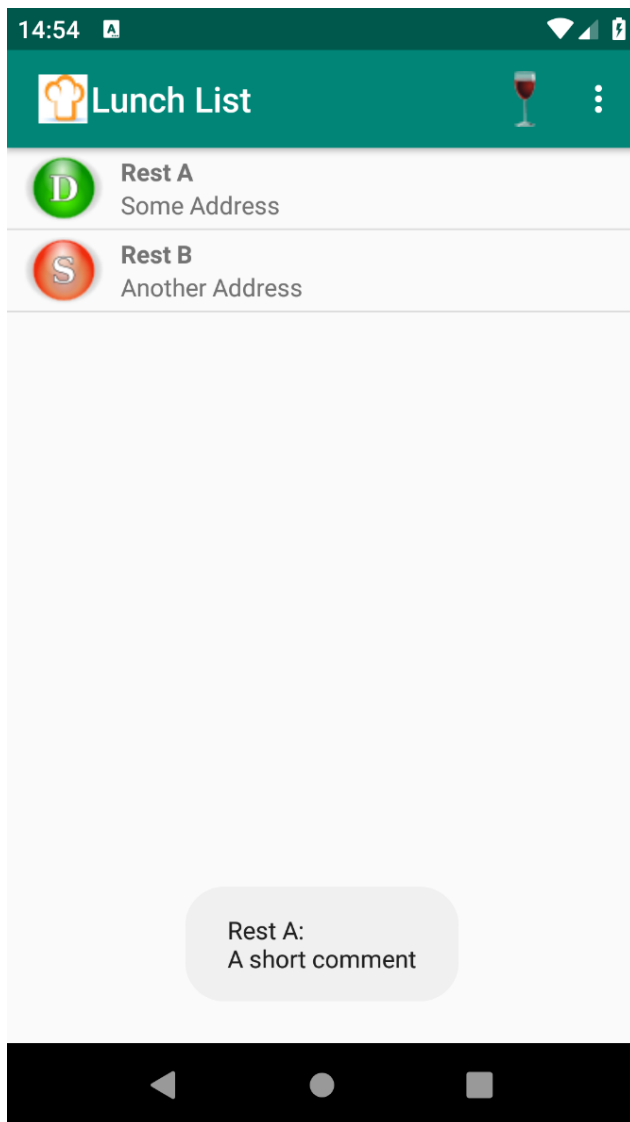
```
private fun load() {
    edName.setText(app.current?.name)
    edAddress.setText(app.current?.address)
    edNotes.setText(app.current?.notes)
    when (app.current?.type) {
        "sit" -> rgTypes.check(R.id.rb_sit)
        "take" -> rgTypes.check(R.id.rb_take)
        "delivery" -> rgTypes.check(R.id.rb_delivery)
    }
}
```

- i. Finally when the user clicks the save button, one of two things should happen, depending on if the user has filled a new restaurant form or has edited an existing one. In the last case the existing one should be removed from the adapter. In both cases a new **current** restaurant should be created and added to the **adapter** (in the same position if it is a replacement). After saving the new data, this activity should be finished, returning to the Main activity. All of this is done in the **Save button** listener:

```
void onBtSaveClick() {
    val type = when(rgTypes.checkedRadioButtonId) {
        R.id.rb_take -> "take"
        R.id.rb_sit -> "sit"
        R.id.rb_delivery -> "delivery"
        else -> ""
    }
    if (rPos != -1)
        app.adapter!!.remove(app.current)
    app.current = Restaurant(edName.text.toString(), edAddress.text.toString(), type,
                             edNotes.text.toString())
    if (rPos != -1)
        app.adapter!!.insert(app.current, rPos)
    else
        app.adapter!!.add(app.current)
    finish()
}
```

You can finish by replacing the application icon on the manifest to *rest\_icon.png*.

The appearance of the new interface is now as follows:



Notice that now the restaurants list survive rotations (and other activity destructions) because now it is stored in the application class (as long as the application (process) remains in memory).