Mobile Computing

Practice # 2a

Android Applications - Interface

1. Create an Android lunch options list app that allows the user to take note of restaurant characteristics like its name, address and type, filling, for the moment, a class (Restaurant) instance with the values.

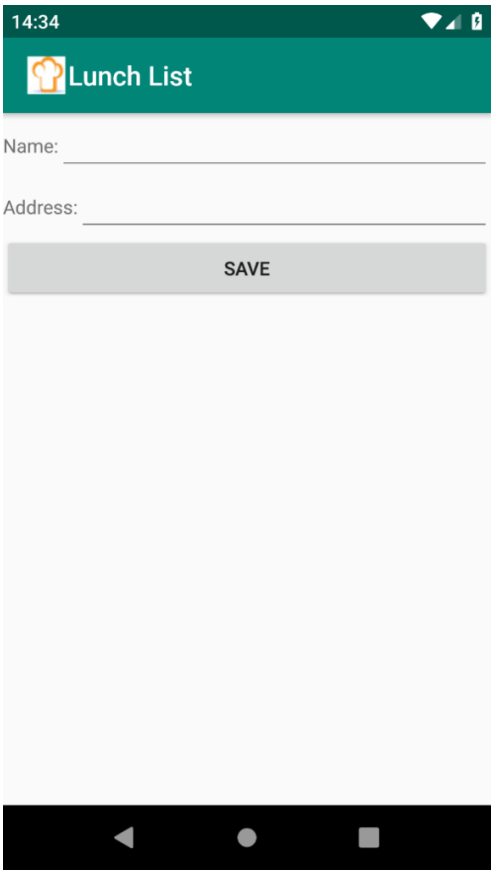
Follow these steps:

a. Create a new Android project in Android Studio. Fill the 'New Project' form with **Lunch List** for the Project name (lunchlist1 as package name) and an empty activity. Specify API 21 as the minimum. Mantain activities derived from ***AppCompatActivity***.

b. Delete the Test directories from the project sources and any library dependencies for testing. Replace the root of the main layout by a ***LinearLayout*** (delete the **Constraintlayout** library dependency if it's on the dependencies list) and design a new layout resource, containing: a label (TextView) showing "Name: " and a text box (EditText) in a first line; another label showing "Address: " followed, in the same line, with a second text box; a button, in a third row, with label "Save". Use only as containers *LinearLayouts*. The final interface should be as is shown in the next picture.
For the first *EditText* include the attribute andoid:imeOptions="actionNext" and for the last EditText include android:imeOptions="actionDone" (to dismiss the keyboard).

c. Copy the **rest_icon.png** file to the drawable resource directory. In the activity **onCreate()** method add this icon to the activity Action Bar (the top header bar):

```
ActionBar bar = getSupportActionBar();
if (bar != null) {
  bar.setIcon(R.drawable.rest_icon);
  bar.setDisplayShowHomeEnabled(true);
}
```

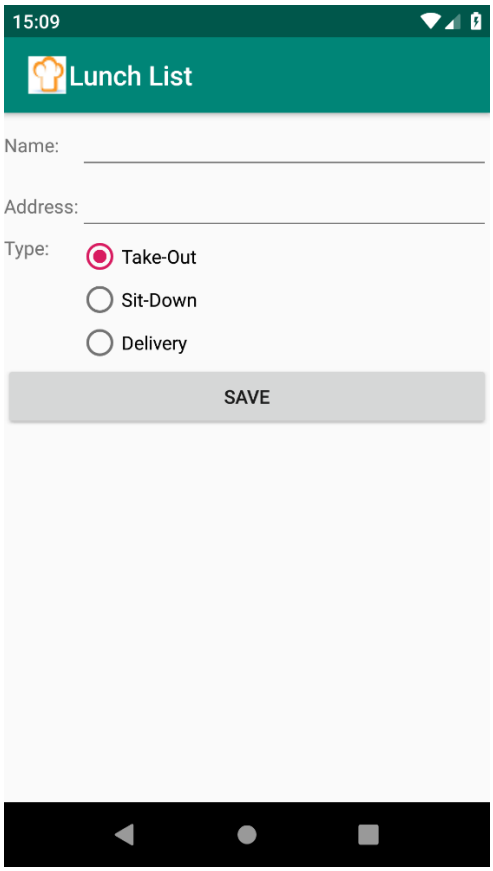


d. Create a new class (Restaurant) with string fields 'name' and 'address', and their getters and setters (Kotlin saves you the need to define getters and setters if they are simple).
e. Hook the save button (through a Listener) to save the text boxes' content to an instance of the Restaurant class, created in the listener method.
f. **[Optional** improvements (to be done later): draw a .png image with 48x48 pixels and put it in the res\drawable folder with a name 'lunch_icon.png'. Change the project manifest to include this icon (@drawable/lunch_icon) as app icon. Play with the views' fonts (color, typeface, size, bold, …).**]**

2. Modify the previous project to align the entry text boxes, using a **TableLayout**. Also add a group of **RadioButtons** to characterize the lunch place. The options should be: Take-out, Sit-down and Delivery. Also update the Restaurant class to include a new field to take note of the restaurant type.

Try to see what happens if you click save without a radio button selected. To correct, choose and pre-select a default value.

[Optional experiment: Try adding more radio buttons than there is room to display on the screen. Solve the problem that appears using a **ScrollView** container (it is a ViewGroup).**]**



3. Adding a list

Instead of using a single Restaurant instance, use now a Java ArrayList like this:

```
List<Restaurant> rests = new ArrayList<>();
```

Whenever the user clicks the Save button, a new item should be added to the ArrayList (through an **ArrayAdapter**, needed by the ListView view to display the array). Provide also a toString() method to the Restaurant class returning its name as a string.

After modifying the Restaurant class (adding toString()), follow these steps:

a. Modify the layout **adding** a RelativeLayout to its top. Inside this Layout attach the previous LinearLayout to the bottom (attribute android:layout_alignParentBottom). Use the top remaining space to attach a ListView (using the attribute android:layout_alignParentTop and

android:layout_above referring the first LinearLayout (you need an **id**)).

b. The ListView needs an ArrayAdapter to get its items from. The ArrayAdapter can be constructed wrapping an ArrayList. If the variable 'list' represents the ListView, this can be done in the Activity as:

```
adapter = new ArrayAdapter<Restaurant>(this, android.R.layout.simple_list_item_1, rests);
list.setAdapter(adapter);
```

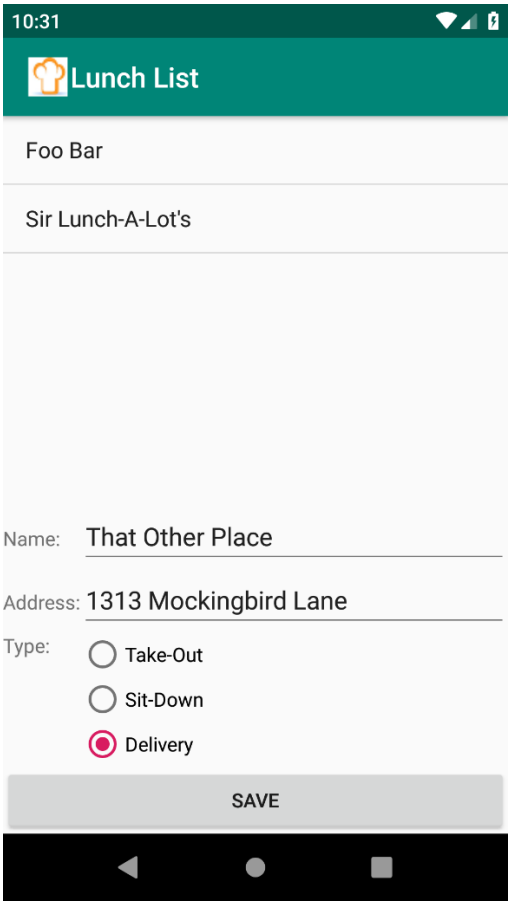
The name 'simple_list_item_1' is a stock (existing) layout for a list row (displaying only a string taken from the toString() method of each ArrayAdapter element).

c. To add a new item (Restaurant) to the list we must use the adapter (otherwise it will not show on the ListView) and its method add(), like *adapter.add(r)*. This will add the restaurant **r** also to the ArrayList.

d. When the user selects an item in the ListView fill all the details in the other controls. Use an appropriate listener (OnItemClickListener), triggered when the user clicks an item in the list.

e. **[Optional** experiment: Try to substitute the ListView with a Spinner and the address text box with an AutoCompleteTextView. See their definitions in the framework reference**].**

We can see an example of this interface in the following image.



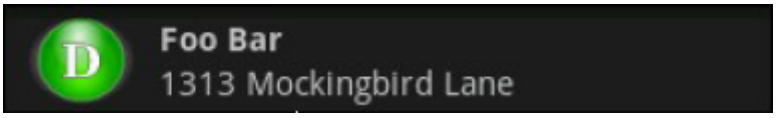
Adding more complex list items

If we want more complex list items, we need to customize the ArrayAdapter associated with the ListView. We can start by defining our own adapter class like this:

```
class RestaurantAdapter extends ArrayAdapter<Restaurant> {
  public RestaurantAdapter() {
    super(context, resourceId, objects);    // the Activity, row layout, and array of values
  }
}
```

Next, we need to design our list row on the screen and write an **XML layout** for it (for example in a file res\layout\row.xml). For a design like the next picture we need three small images, representing the

categories, and made available in the res/drawable folder (**ball_red.png**, **ball_yellow.png**, **ball_green.png**).

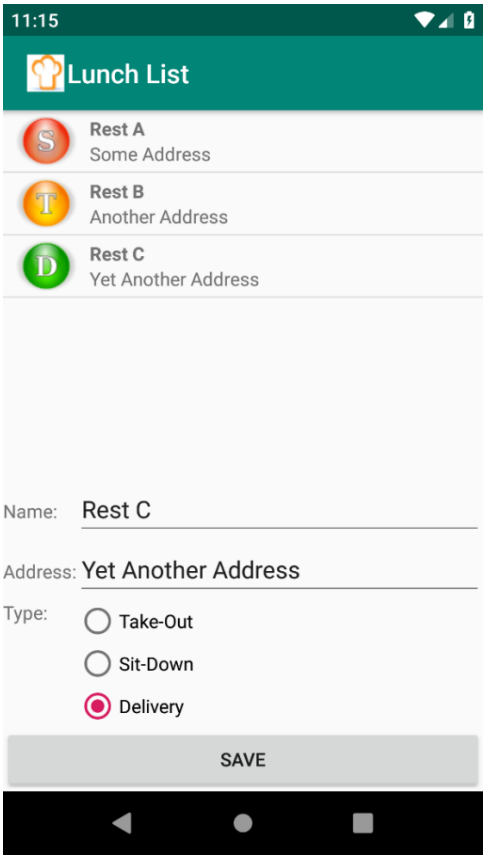


Make the displayed text always a single line (android:maxLines="1") and truncatable (android:ellipsize).

Next, we need that the adapter can build and fill each line, according to the layout and the information available in the restaurant list (List<Restaurant> rests). We can accomplish this task overriding getView() in our custom ArrayAdapter:

```
@Override
public View getView(int position, View convertView, ViewGroup parent) {
   View row = convertView;
   if (row == null) {                                    // this is a new row in the screen
      LayoutInflater inflater = getLayoutInflater();
      row = inflater.inflate(R.layout.row, null);        // get our custom layout
   }                                                     // otherwise an existing row is recycled
   Restaurant r = rests.get(position);
   ((TextView)row.findViewById(R.id.title)).setText(r.getName());          // fill restaurant name
   ((TextView)row.findViewById(R.id.address)).setText(r.getAddress());     // fill restaurant address
   ImageView symbol = (ImageView)row.findViewById(R.id.symbol);
   if (r.getType().equals("sit"))
      symbol.setImageResource(R.drawable.ball_red);                        // fill the image
   …
      // other types here
   …
   return (row);
}
```

After defining the new adapter, we can see the application like the following figure:



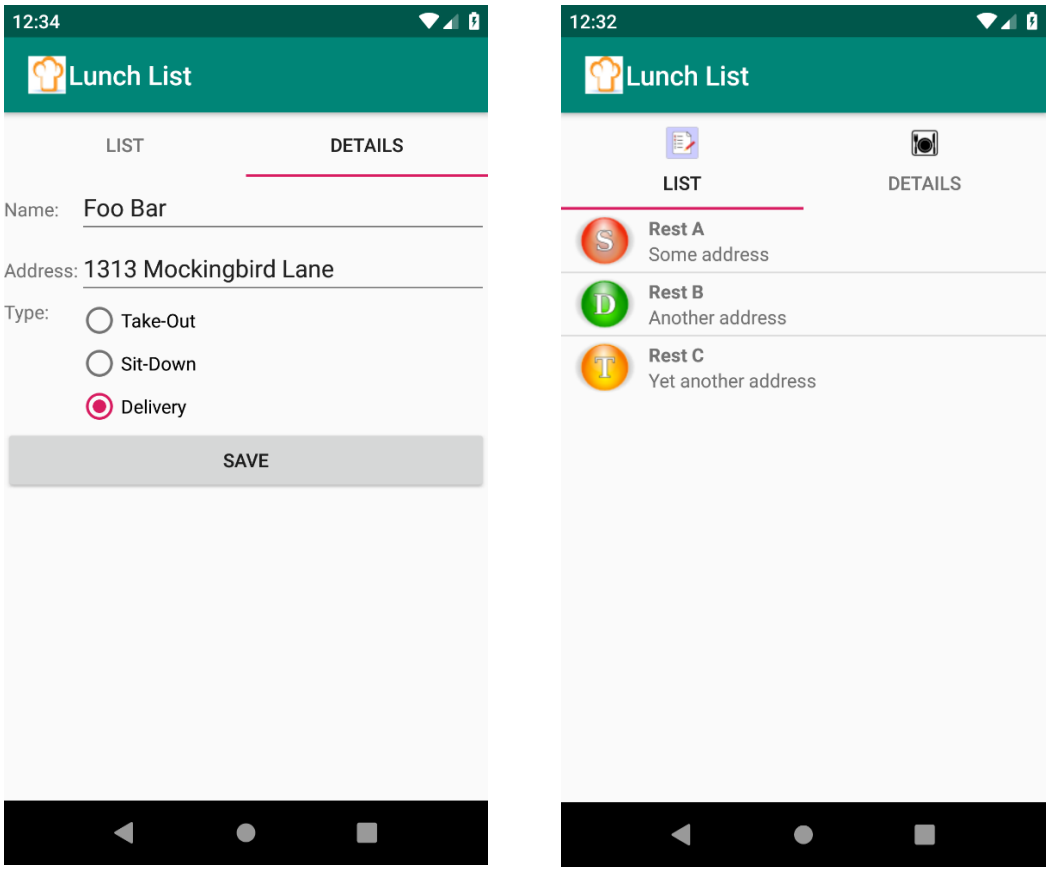
4. Using a tab view

Because the space to observe the restaurant list and to have the details of each in a single screen could be short, it is convenient to separate the two visualizations. For that we can build the same functionality using a tabbed view (originally a combination of a **TabHost**, **TabWidget** and a **FrameLayout**), by putting the list of restaurants in one tab and the details controls (and save button) on another tab (tabs are children of a **FrameLayout**). Selecting an item in the list should switch automatically to the details tab, conveniently filled with the restaurant details. In earlier Android versions the tabs controlling the corresponding views could have an icon. The icon disappeared with the introduction of the ActionBar. Still, with recent versions, we can use a style like the earlier versions (**Theme.Black** or **Theme.Light**), but we lose the **ActionBar** (it is substituted with a top bar showing the activity label, but without any functionality).

More recently the **TabHost** view was replaced by a **TabLayout** (in a support external library), but that needs to have a listener to change the active tab, using the visibility property (like **ActionBar** tabs, which were also deprecated). With the TabLayout.Tab objects, used with **TabLayout**, we can also specify an icon for the tabs.

Modify activity_main.xml, and MainActivity.java to incorporate the two needed tabs adding a **TabLayout**.

Examples:



Tabbed activity with **TabLayout** with and without icons on the tabs.