Mobile Computing

Practice # 2a

Android Applications – Interface (in Kotlin)

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.

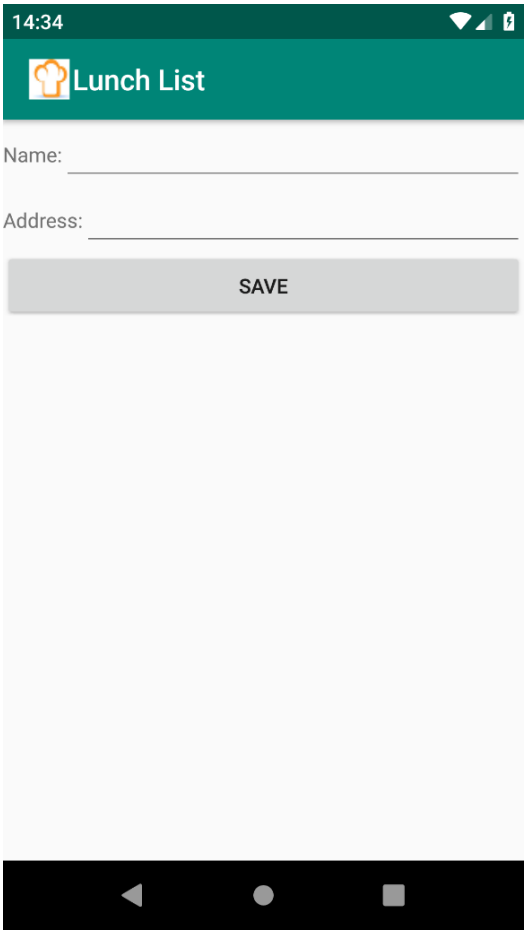
Follow these steps:

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) 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 picture.
For the first EditText include the attribute andoid:imeOptions="actionNext" and for the last EditText include android:imeOptions="actionDone" (to dismiss the virtual 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 (accessed as the property supportActionBar:

```
supportActionBar?.setIcon(R.drawable.rest_icon)
supportActionBar?.setDisplayShowHomeEnabled(true)
```

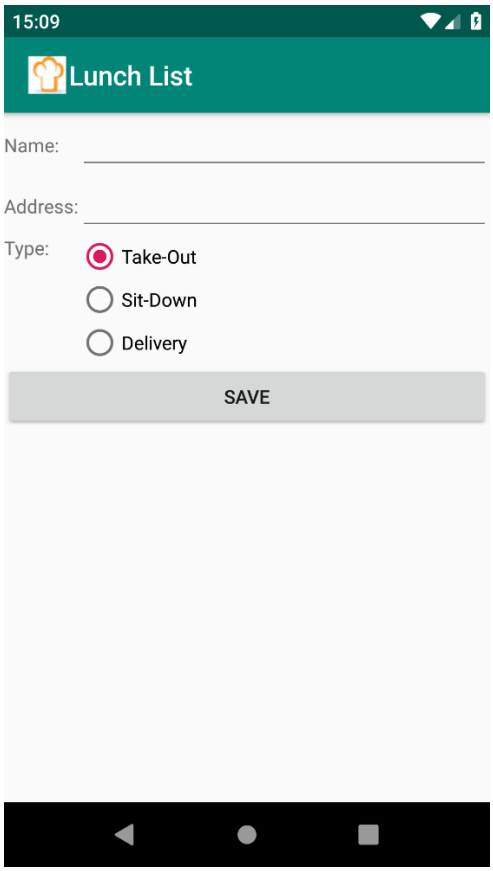


d. Create a new class (Restaurant) with string properties 'name' and 'address'.
e. Hook the save button (through a Listener) to save the text boxes' content to an instance of the Restaurant class, created in the listener method.
f. **[Optional** improvements: draw a .png image with 48x48 pixels and put 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**. Add also 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 property to take note of the restaurant type.

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

[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.**]**



3. Adding a list

Instead of using a single Restaurant instance use now an **ArrayList** object (collection) like this:

```
val rests = arrayListOf<Restaurant>()    // creates an initial empty 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 the reastaurant name.

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

a. Modify the layout **adding** a **RelativeLayout** (or **ConstraintLayout**) 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 property 'list' represents the **ListView**, this can be done in the Activity as:

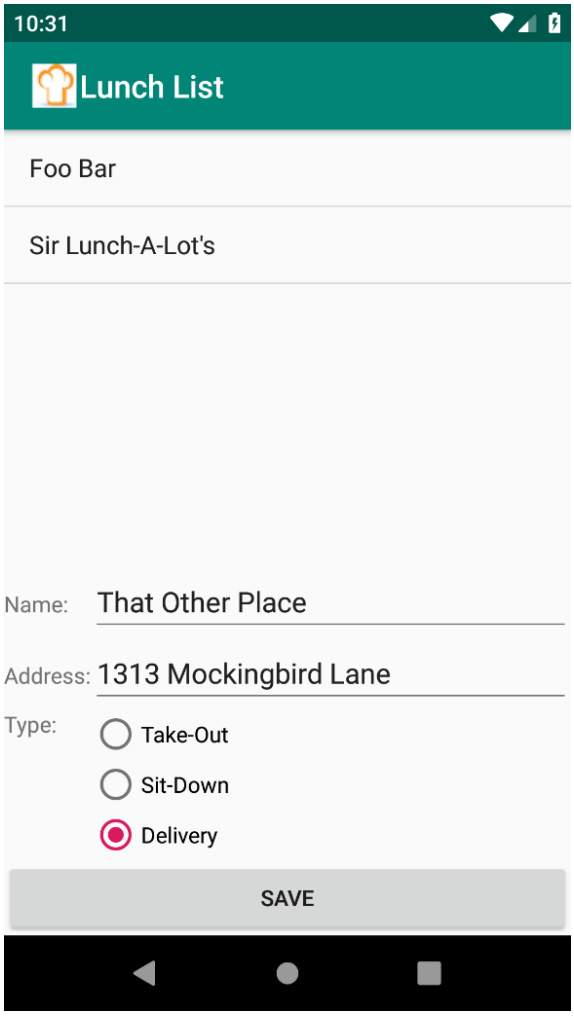
```
val adapter by lazy { ArrayAdapter(this, android.R.layout.simple_list_item_1, rests) }

...
list.adapter = adaper
```

The lazy (a predefine function, accepting another function) delegation allows a property to be initialized only on the first access in the class code, instead of being immediately. This could be needed for properties that can be initialized only when the activity is becoming active (after calling **onCreate()**)
The name 'simple_list_item_1' is a stock 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).
e. **[Optional** experiment: Try to substitute the ListView with a Spinner and the address text box with an AutoCompleteTextView.**]**

We can see an example interface in the following image.

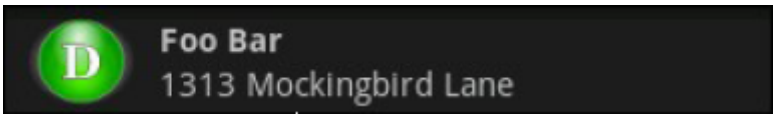


Adding more complex list items

If we want more complex list items (with accustom visualization) we need to customize the **ArrayAdapter** associated with the **ListView**. We can start by defining our own adapter class like this (e.g. as an inner Activity class):

```
inner class RestaurantAdapter(ctx: Context, rid: Int, objs: Any): ArrayAdapter<Restaurant>(ctx, rid,   objs) {
}
```

Next we need to design our list row and write a 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 available in the res/drawable folder (**ball_red.png**, **ball_yellow.png**, **ball_green.png**).

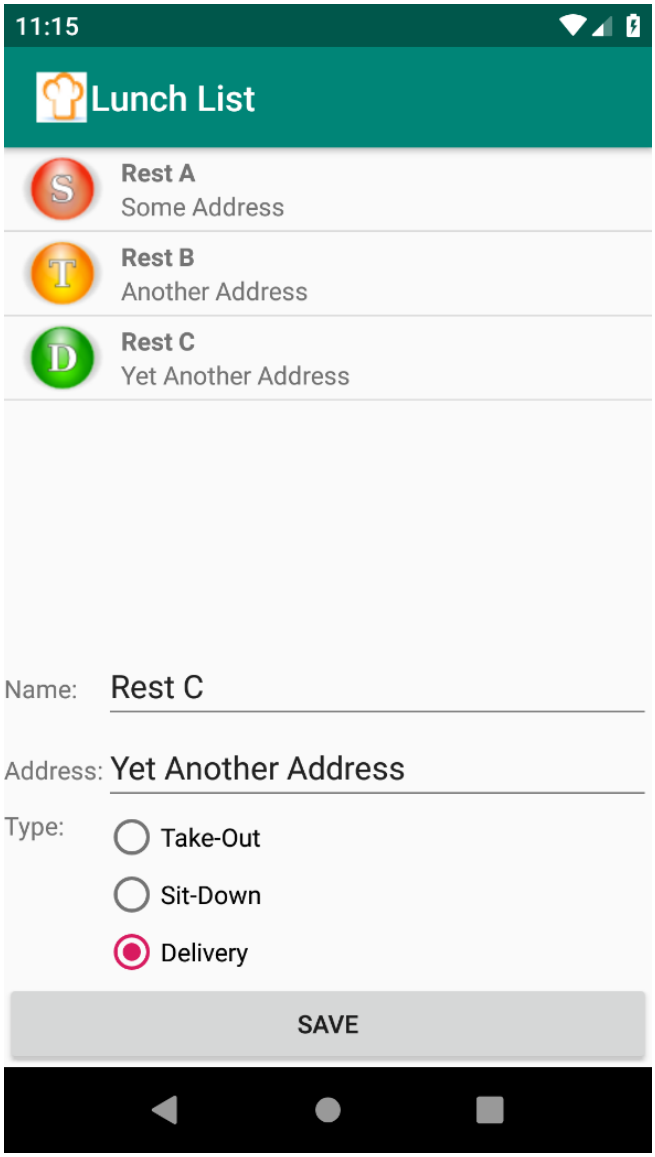


Make the displayed text always a single line (android:maxLines="1") and truncatable (android:ellipsize).

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 getView(int position, View convertView, ViewGroup parent): View {
    val row = convertView ?: layoutInflater.inflate(R.layout.list_row, parent, false)
    val r = rests[position]
    row.findViewById<TextView>(R.id.title).text = r.name
    row.findViewById<TextView>(R.id.address).text = r.address
    val symbol = row.findViewById<ImageView>(R.id.symbol)
    when (r.type) {
       "sit" -> symbol.setImageResource(R.drawable.ball_red)
       "take" -> symbol.setImageResource(R.drawable.ball_yellow)
       "delivery" -> symbol.setImageResource(R.drawable.ball_green)
    }
    return row
}
```

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



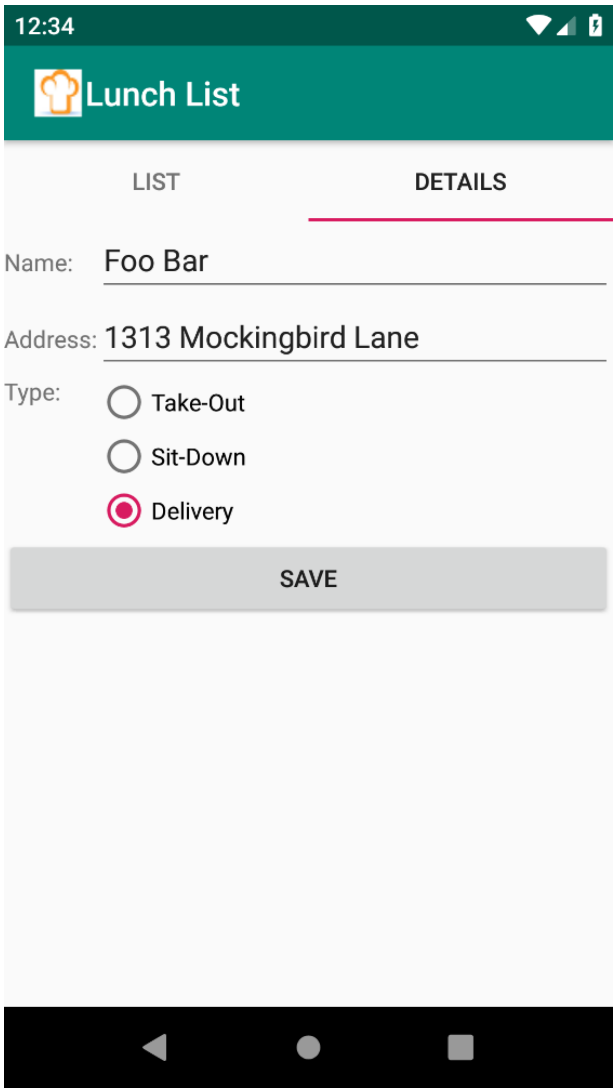
4. Using a tab view

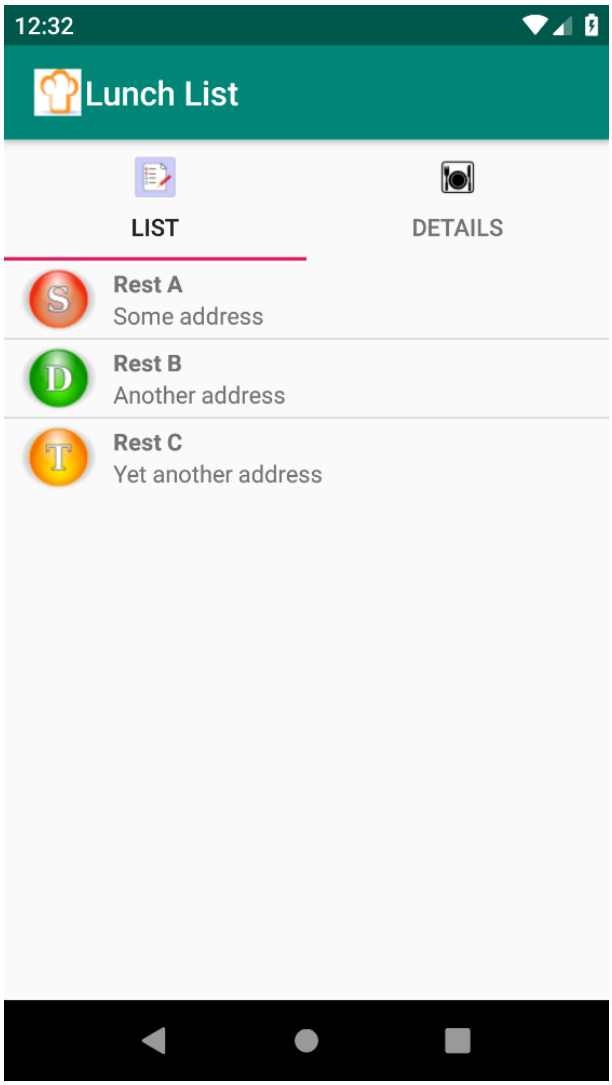
Because the space to observe the restaurant list and to have the details of each in a single screen could be short, it is convenient to separate the two visualizations. For that we can build the same functionality using a tabbed view (originally, a combination of a **TabHost**, **TabWidget** and a **FrameLayout**), 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 old 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 replaced 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.kt* to incorporate the two needed tabs adding a **TabLayout**.

Examples:





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