Lab 3 Instructions

Author: Junwen Shen <junwen5@ualberta.ca>

Preparation

1. Download the `ListyCity` start code from the eClass page
2. Extract the folder
3. Open `AndroidStudio`. Go to File, choose Open, and select the `ListyCity` folder

Part 1 - Customize `ListView`

Goal

Display the city and its province simultaneously.

Modify the View design

1. Open `content.xml`
2. Add a horizontal-oriented `LinearLayout` and move the original `TextView` into the `LinearLayout` we just added.
3. Add another `TextView` for displaying the province

The `content.xml` looks like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">
    <TextView
        android:id="@+id/city_text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:padding="20dp"
        android:textSize="30sp"
        android:text="City" />
    <TextView
        android:id="@+id/province_text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:padding="20dp"
        android:textSize="30sp"
        android:text="Province" />
</LinearLayout>
```

Note:

- The ID for the first/original `TextView` is changed to **city_text** to make it meaningful. Similarly, we set ID for the second one to **province_text**
- `AndroidStudio` will generate warns on `android:text="..."`, and suggest we should use `@string` resource instead of hard-coding. In general, you should put all texts in the string resource file, but in lab we hardcode them for simplicity. Read more about it here: string-resource.

Create a new class for cities

A class can provide more features than a string literal.

1. In the project panel, right click on the directory (package) containing `MainActivity.java` and select New > Java Class. Name the class `City` and then press `enter`.
2. Add two **private** attributes, `name` and `province`, for the `City` class.
3. Add a constructor with two parameters and initialize the class attributes we just declared inside it.
4. Add getters for the attributes.

The `City.java` looks like the following:

```
public class City {
    private String name;
    private String province;

    public City(String name, String province) {
        this.name = name;
        this.province = province;
    }

    public String getName() {
        return name;
    }

    public String getProvince() {
        return province;
    }
}
```

Note:

`AndroidStudio` will generate warns on the attributes and suggest the field may be **final**, simply because we only initialize them in the constructor and don't modify them after. If we declare a variable with the **final** keyword, we can't change its value again. Thus, we can consider a final variable as a constant, as the final variable acts like a constant whose values cannot be changed. If we attempt to change the value of the final variable, then we will get a compilation error. Read more about it: java-final-keyword.

Customize the adapter

Since now we want use objects of the `City` class to represent our data, the original `ArrayAdapter` cannot meet our requirements for displaying extra information we have.

According to the official android UI Guide about the [`ListView`]:

> To customize the appearance of each item you can override the `toString()` method for the objects in your array. Or, to create a view for each item that's something other than a `TextView` (for example, if you want an `ImageView` for each array item), extend the `ArrayAdapter` class and override `getView()` to return the type of view you want for each item.

1. In the project panel, right click on the directory (package) containing `MainActivity.java` and select New > Java Class. Name the class `CityArrayAdapter` and then press `enter`.
2. Let the the `CityArrayAdapter` class extend the `ArrayAdapter` class, and make sure that the `ArrayAdapter` class has a type of `<City>`. After Step 2, you should have a class similar to

```
public class CityArrayAdapter extends ArrayAdapter<City> {
}
```

`AndroidStudio` will generate an error, because we don't have any constructor yet.

3. Add the following constructor to the class so the adapter can be properly initialized and the parent class can hold the context and all cities for us. The **0** is a placeholder for the Layout Resource because we will override the `getView()` method in the next step. Therefore, the parent doesn't need to know the actual layout we will use.

```
public class CityArrayAdapter extends ArrayAdapter<City> {
    public CityArrayAdapter(Context context, ArrayList<City> cities) {
        super(context, 0, cities);
    }
}
```

4. Override `getView()` method. While typing `getView()` the autocomplete should suggest you with the correct method. It is extremely important that you understand what each parameter represents. First we remove the auto-generated return statement, then we create our own `View` object and return it.

```
public class CityArrayAdapter extends ArrayAdapter<City> {

    public CityArrayAdapter(Context context, ArrayList<City> cities) {
        super(context, 0, cities);
    }

    @NonNull
    @Override
    public View getView(int position, @Nullable View convertView, @NonNull ViewGroup
parent) {
        View view;
```

```
        if (convertView == null) {
            view = LayoutInflater.from(getContext()).inflate(R.layout.content,
parent, false);
        } else {
            view = convertView;
        }

        City city = getItem(position);
        TextView cityName = view.findViewById(R.id.city_text);
        TextView provinceName = view.findViewById(R.id.province_text);

        cityName.setText(city.getName());
        provinceName.setText(city.getProvince());

        return view;
    }
}
```

Note:

- The `convertView` object is a way to recycle old views inside the `ListView` ultimately increasing the performance of the `ListView`. If the `convertView` object holds nothing, then we inflate the 'content.xml' and assign it to our `view`. Otherwise, we reuse the `convertView` as our `view`.
- The next step is to get the city from the `cities` list and the set the name of the city and the province to each `TextView`. The `getItem()` is a method in `ArrayAdapter` (parent class) that takes an integer parameter and returns the item at that position. In our case, it will be our city object for rendering.

Update `MainActivity`

Now the design part is ready, we just need some data and use a correct way to setup the activity.

1. Go to `onCreate()` method, create/modify two lists of strings, one for the name of the city and the other for the name of the province.

```
String[] cities = { "Edmonton", "Vancouver", "Toronto" };
String[] provinces = { "AB", "BC", "ON" };
```

2. Change the type of the `dataList` attribute to `ArrayList<City>`. Create `City` objects as our data according to the two lists of strings and put them into `dataList`.

```
dataList = new ArrayList<City>();
for (int i = 0; i < cities.length; i++) {
    dataList.add(new City(cities[i], provinces[i]));
}
```

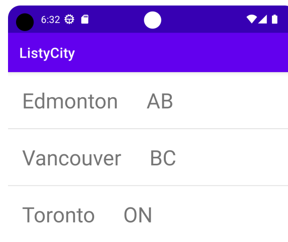
4. Change the type of the `cityAdapter` attribute to our `CityArrayAdapter` and initialize it.

```
cityAdapter = new CityArrayAdapter(this, dataList);
```

5. Let the adapter of `cityList` be our `CityArrayAdapter`.

```
cityList.setAdapter(cityAdapter);
```

Now we just reach the end of Part 1. You can run your app and see if it looks similar to this.



Part 2 - Fragments

Goal

Have the ability to receive the user input for adding more cities by using fragments.

For simplicity, we use `DialogFragment`.

Create a new Layout for the fragment

1. In the project panel, right click on the `res/layout` directory which contains `activity_main.xml` and select New > Layout Resource File. Name it `fragment_add_city`, and use `LinearLayout` as its Root element. Click `OK` to create.
2. Add two `EditText` for accepting text input (You can use the Design panel or just edit the xml file). Set the ID for the first to `edit_text_city_text` and the other to `edit_text_province_text`.

The `fragment_add_city.xml` now looks like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <EditText
        android:id="@+id/edit_text_city_text"
        android:layout_width="match_parent"
```

```
            android:layout_height="wrap_content"
            android:ems="10"
            android:inputType="text"
            android:text="City" />
        <EditText
            android:id="@+id/edit_text_province_text"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:ems="10"
            android:inputType="text"
            android:text="Province" />
</LinearLayout>
```

Note:

We don't need to add buttons, because `AlertDialog` will provide them.

Create the Fragment

1. In the project panel, right click on the directory (package) containing `MainActivity.java` and select New > Java Class. Name the class `AddCityFragment` and then press `enter`.
2. Let the the `AddCityFragment` class extend the `DialogFragment` class, make sure the imported class is `androidx.fragment.app.DialogFragment` instead of `android.app.DialogFragment`
3. Define the `AddCityDialogListener` interface inside the class, and add a private `AddCityDialogListener` attribute named `listener`. After this step, your class look similar to the following:

```
public class AddCityFragment extends DialogFragment {

    interface AddCityDialogListener {
        void addCity(City city);
    }

    private AddCityDialogListener listener;
}
```

4. Type `onAttach`, use auto-completion to override this method. The `listener` provides the reusability of the fragment.

```
public class AddCityFragment extends DialogFragment {

    interface AddCityDialogListener {
        void addCity(City city);
    }

    private AddCityDialogListener listener;

    @Override
    public void onAttach(@NonNull Context context) {
        super.onAttach(context);
```

```
        if (context instanceof AddCityDialogListener) {
            listener = (AddCityDialogListener) context;
        } else {
            throw new RuntimeException(context + " must implement
AddCityDialogListener");
        }
    }
}
```

5. Type `onCreateDialog`, use auto-completion to override this method. This is where we customize the dialog and bind the views.

```
public class AddCityFragment extends DialogFragment {

    interface AddCityDialogListener {
        void addCity(City city);
    }

    private AddCityDialogListener listener;

    @Override
    public void onAttach(@NonNull Context context) {
        super.onAttach(context);
        if (context instanceof AddCityDialogListener) {
            listener = (AddCityDialogListener) context;
        } else {
            throw new RuntimeException(context + " must implement
AddCityDialogListener");
        }
    }

    @NonNull
    @Override
    public Dialog onCreateDialog(@Nullable Bundle savedInstanceState) {
        View view =
LayoutInflater.from(getContext()).inflate(R.layout.fragment_add_city, null);
        EditText editCityName = view.findViewById(R.id.edit_text_city_text);
        EditText editProvinceName = view.findViewById(R.id.edit_text_province_text);
        AlertDialog.Builder builder = new AlertDialog.Builder(getContext());
        return builder
                .setView(view)
                .setTitle("Add a city")
                .setNegativeButton("Cancel", null)
                .setPositiveButton("Add", (dialog, which) -> {
                    String cityName = editCityName.getText().toString();
                    String provinceName = editProvinceName.getText().toString();
                    listener.addCity(new City(cityName, provinceName));
                })
                .create();
    }
}
```

6. Let `MainActivity` implement `AddCityDialogListener` so that we can add the new city into our `dataList`.

```
public class MainActivity extends AppCompatActivity implements
AddCityFragment.AddCityDialogListener {

    private ArrayList<City> dataList;
    private ListView cityList;
    private CityArrayAdapter cityAdapter;

    @Override
    public void addCity(City city) {
        cityAdapter.add(city);
        cityAdapter.notifyDataSetChanged();
    }

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

        String[] cities = { "Edmonton", "Vancouver", "Toronto" };
        String[] provinces = { "AB", "BC", "ON" };

        dataList = new ArrayList<>();
        for (int i = 0; i < cities.length; i++) {
            dataList.add(new City(cities[i], provinces[i]));
        }

        cityList = findViewById(R.id.city_list);
        cityAdapter = new CityArrayAdapter(this, dataList);
        cityList.setAdapter(cityAdapter);
    }
}
```

Update the View design

We add a `FloatingActionButton` for triggering the dialog.

1. Open `activity_main.xml`, add a `FloatingActionButton`, and set its ID to `button_add_city`.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity">
    <ListView
        android:layout_width="match_parent"
```

```xml
        android:layout_height="0dp"
        android:layout_weight="1"
        android:id="@+id/city_list" />
    <com.google.android.material.floatingactionbutton.FloatingActionButton
        android:id="@+id/button_add_city"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:clickable="true"
        app:srcCompat="@android:drawable/ic_input_add" />
</LinearLayout>
```

2. Add a on-click listener for the button in `onCreate` method.

```java
public class MainActivity extends AppCompatActivity implements
AddCityFragment.AddCityDialogListener {

    private ArrayList<City> dataList;
    private ListView cityList;
    private CityArrayAdapter cityAdapter;

    @Override
    public void addCity(City city) {
        cityAdapter.add(city);
        cityAdapter.notifyDataSetChanged();
    }

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

        String[] cities = { "Edmonton", "Vancouver", "Toronto" };
        String[] provinces = { "AB", "BC", "ON" };

        dataList = new ArrayList<>();
        for (int i = 0; i < cities.length; i++) {
            dataList.add(new City(cities[i], provinces[i]));
        }

        cityList = findViewById(R.id.city_list);
        cityAdapter = new CityArrayAdapter(this, dataList);
        cityList.setAdapter(cityAdapter);

        FloatingActionButton fab = findViewById(R.id.button_add_city);
        fab.setOnClickListener(v -> {
            new AddCityFragment().show(getSupportFragmentManager(), "Add City");
        });
    }
}
```

Note:

- We use a so-called **lambda expression** as the `onClickListener`. It provides a clear and concise way to represent one method interface using an expression. Read more about it: java-lambda-expressions.
- The more complex but ideal way to handle communication between fragments is using `Bundle` or `ViewModel`. Read more about it: fragments-communication.

Now we just reach the end of Part 2. You can run your app and see if it looks similar to this.