Android tutorial: Coffee ordering app

“refactoring”

For who?

Students who take the *Software development* course at KU Leuven Groep T Campus

For what?

To get started with Android app development

To get to know data storage solutions provided by our campus

To be used in conjunction with explanation given in the lab sessions

What is included?

Refactoring of the Android app from the Android tutorial

What are the prerequisites?

Run the Android tutorial

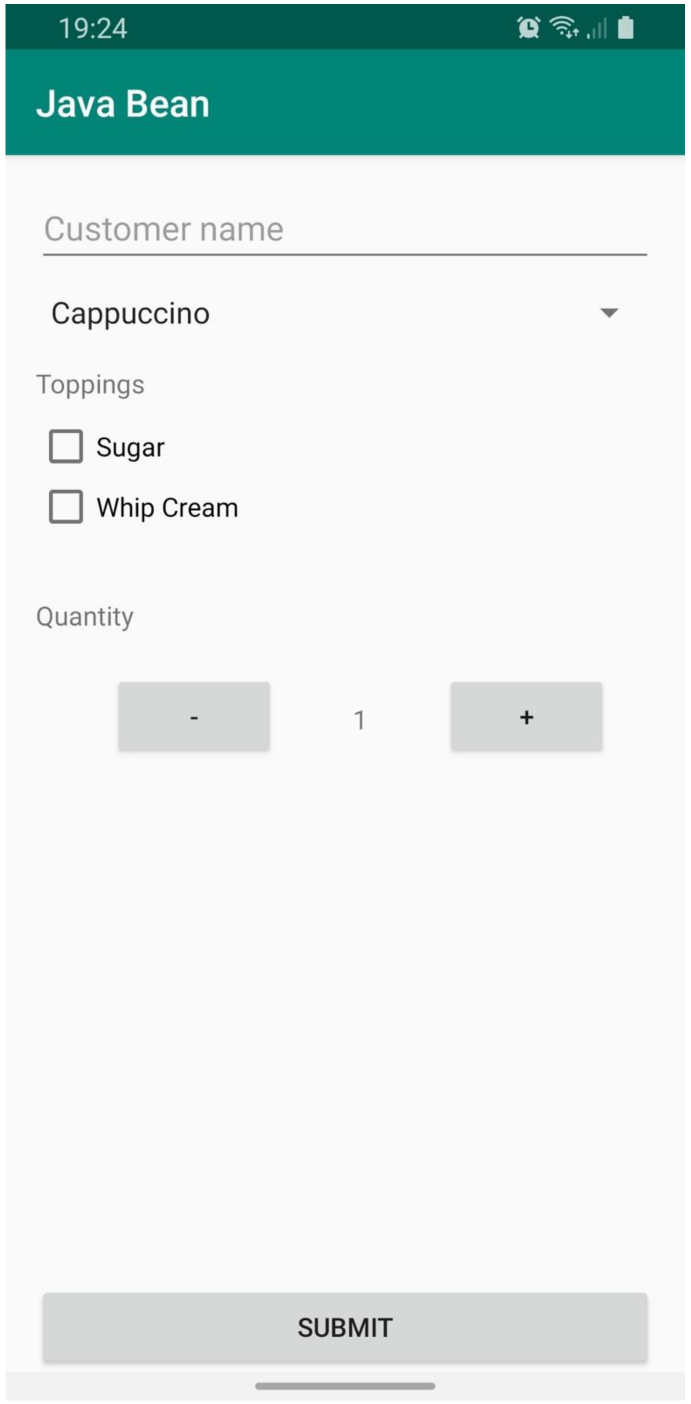


Table of Contents

Introduction: why refactor the code?

Looking at the code of the Android tutorial JavaBean, these remarks can be made:

- there are no domain objects: the different properties that belong to a coffee order are taken from the input form and piece by piece passed to the intent for the QueueActivity
- no separation of concerns, no encapsulation, no single responsibility per class,
- …

Looking at the QueueActivity, it is even worse: all the logic is packed in one big method (onCreate) of one class.

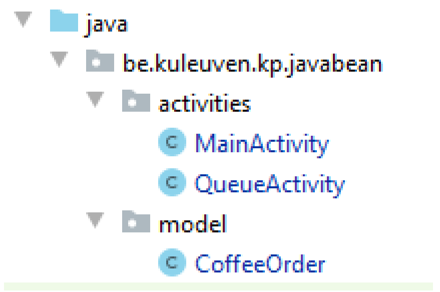
There is a lot of room to improve the code and to make it more extensible: for instance make it easier to add other coffee order related actions. We will do two refactorings on the code: introduce a domain object CoffeeOrder and refactor the QueueActivity class.

Introduce the domain object CoffeeOrder

Create class CoffeeOrder

The simplest refactoring in this example is to add a domain object, to collect the data and add behavior. The app is about ordering coffee, so it seems reasonable to add a class `CoffeeOrder`.

But let us first put the classes in packages: `activities` for the Activity classes (you can drag-and-drop the two Activity classes to this new package; the IDE will ask about Refactoring these) and `model` for the domain objects, the "data model" (see screendump at the right).


```
java
  be.kuleuven.kp.javabean
    activities
      MainActivity
      QueueActivity
    model
      CoffeeOrder
```

Create a class `CoffeeOrder`, with the fields you can discover from the MainActivity form: name (String), coffee (String), sugar (boolean), whipCream (boolean), quantity (int). You can provide a constructor with these 5 fields. As long as no getter/setters or other method are needed, you can leave the code like that.

Update MainActivity

Replace the code that passes the 5 values separately to the intent in the method `onBtnSubmit_Clicked` , with a CoffeeOrder object:

```
…
CoffeeOrder order = new CoffeeOrder(txtName.getText().toString(),
        spCoffee.getSelectedItem().toString(),
        cbSugar.isChecked(),
        cbWhipCream.isChecked(),
        Integer.parseInt(lblQty.getText().toString()));

Intent intent = new Intent(this, QueueActivity.class);
intent.putExtra("Order", order);
…
```

Implement the Parcelable interface

A syntax error will appear on the intent.putExtra("Order", order) line: "Cast 2nd parameter to android.os.Parcelable". To allow passing the order object to the intent, the CoffeeOrder class should implement the "Parcelable" interface. A profound discussion of this interface is beyond the scope this tutorial, but we can solve this error in a simple way.

Passing data between objects in Android is done with a Bundle object. This object is a kind of Map (key-value pairs) that can only contain the following data: String, primitives (int, long, …), Serializable and Parcelable objects (Serializable and Parcelable are interfaces). The default serialization (Serializable) in Java is slow; in Android they therefor introduced a faster way to serialize an object and pass it to a Bundle object: the Parcelable interface.

Let the CoffeeOrder class implement this interface, and provide an implementation for the two methods of this interface: `describeContents` (may return 0) and `writeToParcel`. WriteToParcel is used to serialize the different values of the object. In the version of the API used for this tutorial, there is no "dest.writeBoolean()" method, yet, so we have to use a workaround with String values (see sugar) or byte values (see whipCream) or something else. Using two different styles is only done for pedagogical reasons here! Make your own code as consistent as possible and use the same option for both boolean values. The order in which you write the different values is important when loading the object again from the bundle.

```
@Override
public void writeToParcel(Parcel dest, int flags) {
    dest.writeString(name);
    dest.writeString(coffee);
    dest.writeString(Boolean.valueOf(sugar).toString()); //
writeBoolean not yet available in this version of the API
   //
dest.writeString(Boolean.valueOf(whipCream).toString());
    dest.writeByte(whipCream ? (byte) 1 : (byte) 0);
    dest.writeInt(quantity);
}
```

To make this possible, provide a **second constructor** that takes a `Parcel` as parameter and restore the fields following the order and strategy of the writeToParcel method.

```
public CoffeeOrder(Parcel in) {
    name = in.readString();
    coffee = in.readString();
    sugar = Boolean.parseBoolean(in.readString());
    //whipCream = Boolean.parseBoolean(in.readString());
    whipCream = (in.readByte() == 1) ? true : false;
    quantity = in.readInt();
}
```

From the Android documentation: "Classes implementing the Parcelable interface must also have a non-null static field called `CREATOR` of a type that implements the `Parcelable.Creator` interface." This is the code:

```
public static final Parcelable.Creator<CoffeeOrder>
                      CREATOR = new Creator<CoffeeOrder>() {
    @Override
    public CoffeeOrder createFromParcel(Parcel in) {
        return new CoffeeOrder(in);
    }

    @Override
    public CoffeeOrder[] newArray(int size) {
        return new CoffeeOrder[size];
    }
};
```

Update QueueActivity

At the "receiver" side, the onCreate method in the QueueActivity class, you can get the CoffeeOrder object out of the Bundle with the "getParcelable()" method:

```
CoffeeOrder order = (CoffeeOrder) getIntent().getExtras().getParcelable("Order");
```

To finish this part of the refactoring, you can implement the "Challenges" from the basic tutorial: Format the information as follows: Show the name of the customer and the ordered coffee *n* times, where *n* is the quantity. E.g., if "John" orders 2 cappuccinos with sugar, show:

```
John
      Cappuccino + sugar
      Cappuccino + sugar
```

In which class will you implement this? Where are the data you want to show? Show the result in the “txtInfo” field of the QueueActivity UI (do not start the volley-calls yet, because we will refactor them soon).

Further improvements: constructors

Imagine that the constructor of the CoffeeOrder class contains some business logic: for instance the maximum number of coffees in one order can be 5. You should enforce this limit in your UI, but it is a good idea to check for this any time you assign a value to the quantity field in a CoffeeOrder object. Implement this in the CoffeeOrder constructor:

```
private final int MAX_NUMBER_OF_COFFEES = 5;

…

public CoffeeOrder(String name, String coffee, boolean sugar, boolean whipCream, int quantity) {
    this.name = name;
    this.coffee = coffee;
    this.sugar = sugar;
    this.whipCream = whipCream;
    if (quantity > MAX_NUMBER_OF_COFFEES) { // limit to 5 coffees
        quantity = MAX_NUMBER_OF_COFFEES;
    }
    this.quantity = quantity;
}
```

And how about the other constructor(s)? Is it a good idea to implement the same logic – although very simple in this case – over and over again in every constructor?

Solution: use the this() call to call this “main” constructor from the other constructor(s).

```
public CoffeeOrder(Parcel in) {
    this (
            in.readString(),
            in.readString(),
            Boolean.parseBoolean(in.readString()),
            (in.readByte() == 1) ? true : false,
            in.readInt()
    );
}
```

CoffeeOrder from JSON document

In the QueuActivity class you will find this line of code:

```
info += o.get("customer") + ": " + o.get("coffee") + " x " + o.get("quantity") + " " + o.get("toppings") + " ready in " +
o.get("time_left") + " minutes\n";
```

This can be improved by introducing another constructor in the CoffeeOrder class, based on a JSON object:

```
public CoffeeOrder(JSONObject o) {
    try {
        name = o.getString("customer");
        coffee = o.getString("coffee");
        String toppings = o.getString("toppings");
        sugar = toppings.contains("sugar");
        whipCream = toppings.contains("cream");
        quantity = o.getInt("quantity");
        if (quantity > MAX_NUMBER_OF_COFFEES) {
            quantity = MAX_NUMBER_OF_COFFEES;
        }
    } catch (JSONException e) {
        e.printStackTrace();
    }
}
```

Unfortunately we cannot use the this() call, because it has to be the first line in the constructor and in the case of parsing a JSON-object, this is not possible:

And then you can use this constructor in the QueueActivity class:

```
o = response.getJSONObject(i);
CoffeeOrder order = new CoffeeOrder(o);
info += order.toString() + "\n" + "will be ready at " + o.get("due-date") + "\n";
```

Further improvement: responsibility in correct class

Looking at the code in the QueueActivity class of the tutorial, there is another piece of code that makes use of the fields of a CoffeeOrder:

```
String toppings = (extras.getBoolean("Sugar") ? "+sugar" : "") +
        (extras.getBoolean("WhipCream") ? "+cream" : "");
String requestURL = SUBMIT_URL + extras.get("Name") + "/" +
      extras.get("Coffee") + "/" + ((toppings.length() == 0) ?
          "-" : toppings) + "/" + extras.get("Quantity");
```

This logic should be implemented in the CoffeeOrder class in a method: getRequestURL():

```java
public String getRequestURL(){
    StringBuilder sb = new StringBuilder(name);
    sb.append("/" + coffee);
    sb.append("/" + getToppingsURL());
    sb.append("/" + quantity);
    return sb.toString();
}

private String getToppingsURL() {
    if (hasToppings()) {
        return (sugar ? "+sugar" : "") + (whipCream ? "+cream" : "");
    } else {
        return "-";
    }
}

private boolean hasToppings() {
    return sugar || whipCream;
}
```

In the QueueActivity class the code can become more self-explaining and readable:

```java
String requestURL = SUBMIT_URL + order.getRequestURL();
```

Summary

To refactor the code in a "more object oriented" way and model the data that "kept hanging around together" in a domain class, we improved the design of the project. CoffeeOrder related code ("business logic") is now implemented at the right spot: the CoffeeOrder class.

The more complex part is making your domain object ready to be passed into a Bundle by implementing the Parcelable interface.

Refactor QueueActivity: Extract Method

For the last refactoring, you can use the built-in refactoring tool: Extract Method.

Select the code:

```
String info = "";
for (int i = 0; i < response.length(); i++) {
    JSONObject o = null;
    try {
        o = response.getJSONObject(i);
        CoffeeOrder order = new CoffeeOrder(o);
        info += order.toString() + "\n" + " will be ready at " + o.get("due_date") + "\n";
    } catch (JSONException e) {
        e.printStackTrace();
    }
}
txtInfo.setText(txtInfo.getText().toString() + info);
```

Choose for Refactor > Extract > Method (CTRL-ALT-M) and choose a destination (inside the QueueActivity class) and a method name.

The aim is to make your code more readable and split it up in methods, each at their correct level of abstraction. Take the time to refactor your own project code from time to time: after adding a new feature, after a long day or night of work, … every time you feel it is necessary.

“Clean code” reads much faster, is more understandable and more easy to extend.