

# Android tutorial: Coffee ordering app

## For who?

- Students who take the *Programming Techniques* 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?

- IDE set up guide
- Introduction to Android studio, Android app development concepts and Event-driven paradigm
- Step-by-step guide to create a coffee ordering app – “Java Bean” and communicate with a web service
- Small exercises to brush up your coding skill

## What are the prerequisites?

- Java programming language
- Object oriented concept
- Relational database concepts

## Android Studio version?

- Originally created for Android Studio version 3.5.2;
- updated for version 4.1.2 (March, 2021);
- updated for Bumblebee|2021.1.1 Patch 2 (Feb 2022);
- updated for Dolphin|2021.3.1 Patch 1 (Dec 2022);
- updated for Electric Eel|2022.1.1 Patch 1 (Jan 2023)

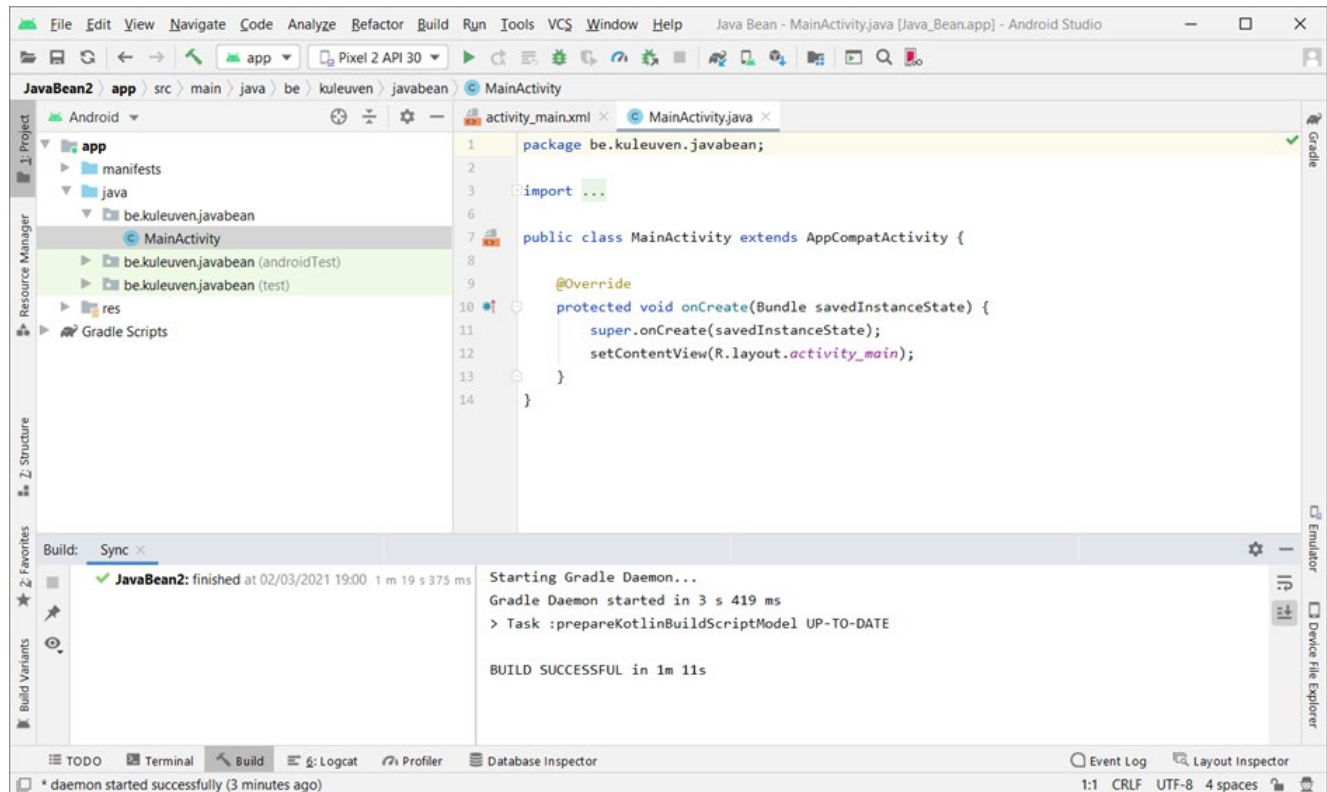
A screenshot of a mobile application titled "Java Bean". The interface includes a status bar at the top showing the time 19:24 and various icons. Below the title bar, there is a form with the following elements: a text input field for "Customer name"; a dropdown menu currently showing "Cappuccino"; a section for "Toppings" with two checkboxes, "Sugar" and "Whip Cream", both of which are unchecked; a "Quantity" section with three buttons: a minus button, a central display showing the number "1", and a plus button; and a large "SUBMIT" button at the bottom of the form.

## TABLE OF CONTENTS

<b>1. Setting Up a Project .....</b>	<b>3</b>
<b>2. Setting Up a Run Environment .....</b>	<b>4</b>
2.1. Setting up an Android emulator .....	4
2.2. Configuring an Android phone as a development device .....	4
2.3. Running the app .....	4
<b>3. Getting Started .....</b>	<b>5</b>
3.1. Navigating Android Studio .....	5
3.2. Understanding the Android app development concept .....	5
3.3. Event-driven system .....	7
<b>4. Hands-on .....</b>	<b>8</b>
4.1. User interface design .....	8
4.2. Building our first interface in design mode .....	9
<b>5. Event handling &amp; View manipulation .....</b>	<b>12</b>
5.1. Callback functions for buttons .....	12
5.2. Domain class: Coffee order .....	13
5.3. Intents .....	14
5.4. Passing data between activities .....	14
<b>6. Working with external data sources .....</b>	<b>18</b>
6.1. Adding Volley to the project .....	18
6.2. Back-end .....	18
6.3. Sending data .....	19
6.4. Retrieving data .....	22
6.5. Show the orders in a RecyclerView .....	24
<b>Appendix A: Coupling Layout to Code .....</b>	<b>30</b>
<b>Appendix B: Lifecycle callback methods .....</b>	<b>31</b>
<b>Appendix C: JSON .....</b>	<b>32</b>
JSON Objects .....	32
JSON Array .....	32
Creating JSON Data in Java .....	33
Retrieving JSON Data in Java .....	34
<b>Appendix D: XML .....</b>	<b>36</b>
XML .....	36

## 1. SETTING UP A PROJECT

Developing an Android app can be done entirely on Android studio. Before a project can be set up, you need to download and install Android studio from <https://developer.android.com/studio> or install it by using the [JetBrains Toolbox App](#). Once installed, follow the steps below to set up your first project:




1. Start Android Studio, and you will be welcomed with a wizard screen. Choose **New Project**
2. Select a Project Template and choose how the first screen of your app looks like. We will choose **Phone and Tablet > Empty Activity** as an example in this tutorial. Then click **Next**
3. On the **Configure Your Project** screen, you can set the name of your app and other information. In this tutorial, we will leave the settings with their default values, but we set the following information:
  - Name: Java Bean
  - Package name: be.kuleuven.gt.javabeen
  - Language: Java
  - Minimum API level: API 23: Android 6.0 (Marshmallow) - to ensure that our app is compatible with 97.2% of all android devices (at the time of writing this tutorial). For more information about API level distribution click **help me choose**.
4. Click **Finish**. If the SDK for your selected API level is not installed yet, Android studio will automatically download and install it. Once this is done, click **Finish**.
5. The project setup wizard is completed. You will see a window as shown above. Note that during the first startup of your project, Android studio will automatically run through a configuration process and install a number of required libraries. Because of this, the initial build process might take several minutes to complete. Subsequent builds will complete much faster.

## 2. SETTING UP A RUN ENVIRONMENT

The app that we are developing is not meant to be run on a computer, but on a smartphone. Hence, you may need an Android device to test/run your app. If you own a physical Android device, you may skip the following section and go to Configuring an Android phone as a development device.

### 2.1. Setting up an Android emulator

To test/run Android apps on your computer without the need for a real Android device, you can create an Android Virtual Device (AVD). Open the Device Manager  under the Tools menu or by using the Device Manager button in the “tool window bar” around the outside of the IDE window or by clicking the appropriate icon in the Navigation Bar. Select Virtual, use the Create device button and choose an appropriate device: a Phone or Tablet, with a preferred size and resolution (for instance Pixel 5) and choose a system image (click the arrow to download this image first, and you will have to agree; the download will take a while), for instance API 26.

This device appears in the device list and is selectable in the toolbar at the top (“available devices”).

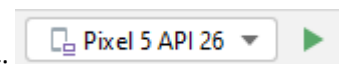
### 2.2. Configuring an Android phone as a development device

If you have a physical Android device, you may also connect it to your computer and use this as test environment. This will always be faster and smoother than any emulator. More info on how to set up a physical device can be found here: <https://developer.android.com/studio/run/device>.

The most important step is to enable the **Developer Options** on the **Settings** of your device by tapping 7 times (no, it is no joke!) the **build number** (**Settings** -> **About Phone**). Finally, go to **Developer Options** on the **Settings** of your phone and turn on **USB Debugging**.

### 2.3. Running the app

Once the app is finished, you can run it with the green arrow key in the toolbar:

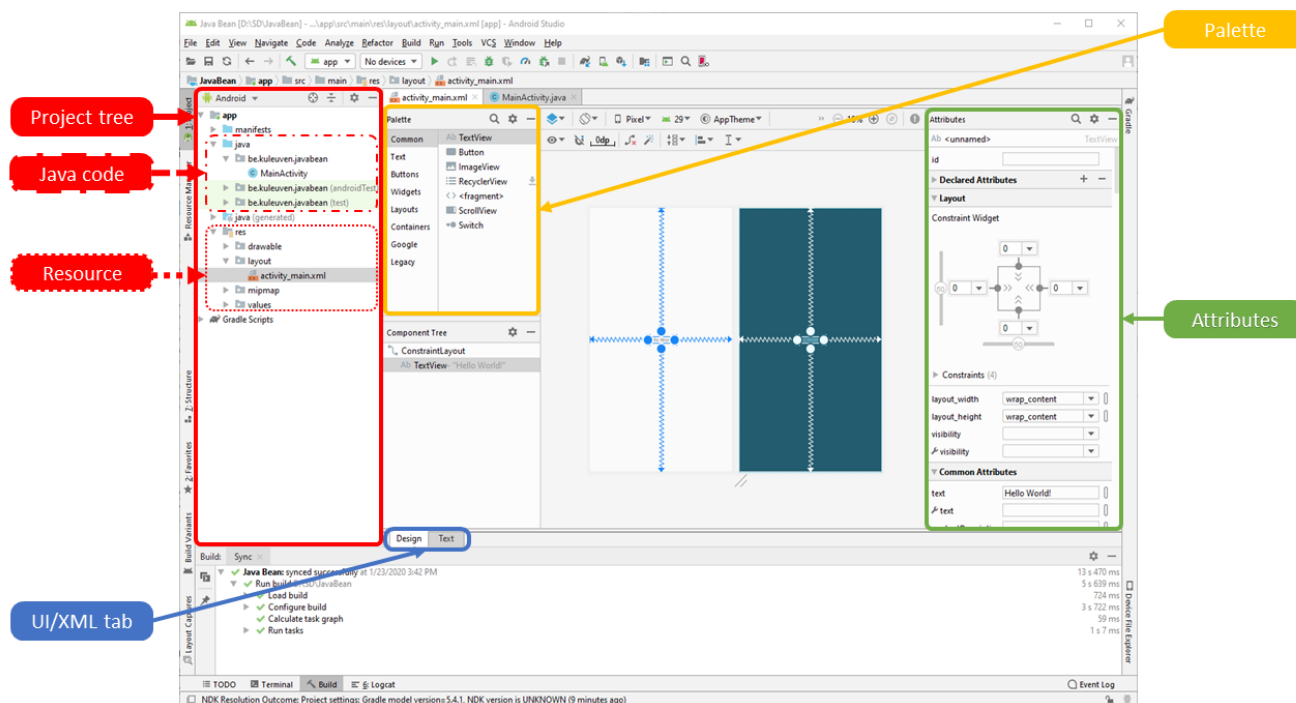


In the dropdown menu you can select whether to run the app on an emulator or on your phone - if it is connected and set up properly. After a short build process (which will be longer the first time you run the project) your app will launch and show the text "Hello world!" in the center of the screen.

## 3. GETTING STARTED

### 3.1. Navigating Android Studio

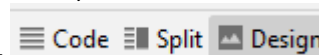
The window of Android studio contains several important areas as shown below:



The area in the **red box** is the navigation tree of all project files. In Android app development, user interface (UI) and code are separated into different files (java for code, and xml or other files for UI/resources). The highlighted **java** folder contains all java code of your project whereas the **res** (so-called resource folder) contains the UI of your activities, static strings, icons, images and other non-code objects of your projects. The **layout** folder contains all your activities' user interface in xml format.

Areas in **orange**, **green**, and **blue** provide necessary tools for UI design, and are only visible once you have opened an xml file of your activity user interface. To create a user interface you can use the design view; it has a rendered view (left) and a blueprint view (right) with only the outlines. You can choose to work with one of the two or with both ("Select Design Surface": Blueprint, Design, both or Color Blind Modes). With these buttons in the top-

right corner of the tab you can split between Code, Design or both (Split):



### 3.2. Understanding the Android app development concept

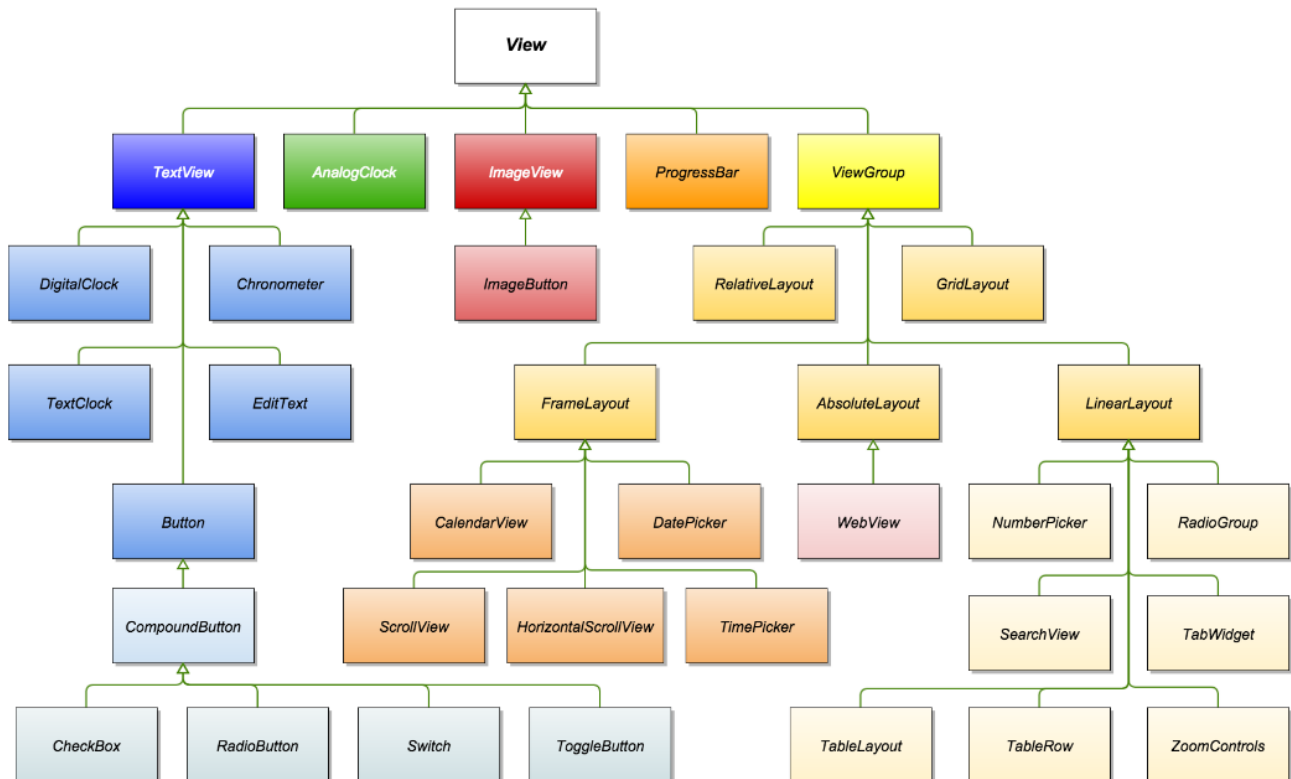
#### 3.2.1. Project structure

An Android project comprises of many folders and files, all of which can be overwhelming at the first glance. To make sense of this complex project structure, we first need to grasp the philosophy behind this organizational structure; an Android project is structured according the View-Controller architectural pattern, which separates user interfaces from the application logic (the code that produces dynamic behavior of the application at runtime) and data models. This is the reason behind the separation of the **java** folder and the **res** folder in the project tree. The idea is not only to ensure maintainability of your project, but also to allow user interface designers to

work independently from programmers. For example, you can give a makeover to your app without changing your java code by just styling the user interface via the drag-drop design tab or the xml text tab (area in [blue](#)).

### 3.2.2. View Hierarchy

## The Android View Class



In an Android app, everything that you can see on the screen (i.e. all user interface components) inherits from the class **View**. View is not only limited to the user interface controls that you can see and interact with (e.g. a button or text), but also the components that group and organize other Views. A good metaphor is to think of the way you organize physical objects in everyday life. You may have individual objects laying around; you can also organize them by putting those individual objects into boxes and you can also organize those boxes by putting them into larger boxes. Both individual objects and boxes are physical objects. Similar to View, there are individual or “simple” Views (e.g., Button, Progress bar) but there is also a ViewGroup which organizes other Views just like those boxes. The figure above shows an overview of the View hierarchy in an Android app.

In this tutorial, we will stick with the terminology View for individual user interface components and ViewGroup for components that organize other views.

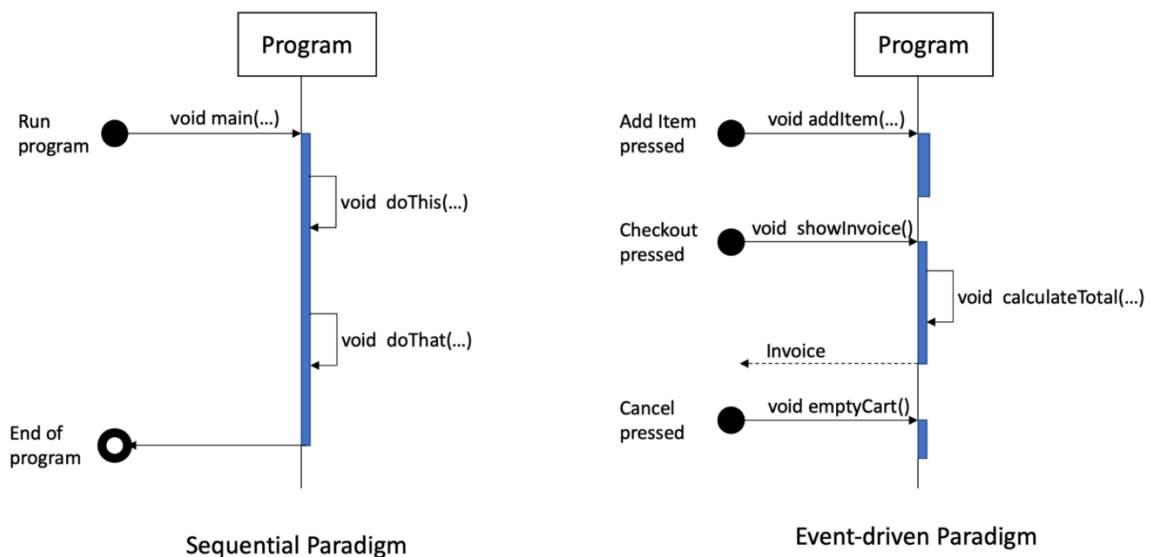
### 3.3. Event-driven system

In an Android app, your code will be executed in an event-driven manner. In other words, your code will only be executed when there is a trigger. For example, if you press a button, a dialog box shows up. The dialog box would not show up until the button is pressed. This means the button pressing action is a trigger, a so called **event**. The event invokes a piece of code (e.g. a method) – a *callback* method - that does some computations and then shows the dialog box. This piece of code or method is called **event handler**.

In the event-driven paradigm, you can consider that each event triggers a small program (or more accurately a method) to execute. For instance, if the user presses the 'Buy Now' button in a shopping app, a piece of code or program or function that calculates the total price and charges the user's credit card will be executed. After all, a method or function behaves like a small program for a specific job.

An event handler can be assigned to an event statically on the XML layout (as shown in the [Hands-on](#) section of this tutorial) or dynamically in your java code (see [Appendix A: Coupling Layout to Code](#))

The figure below illustrates the similarities and differences between the sequential and the event-driven paradigm through an example on two UML sequence diagrams.



## 4. HANDS-ON

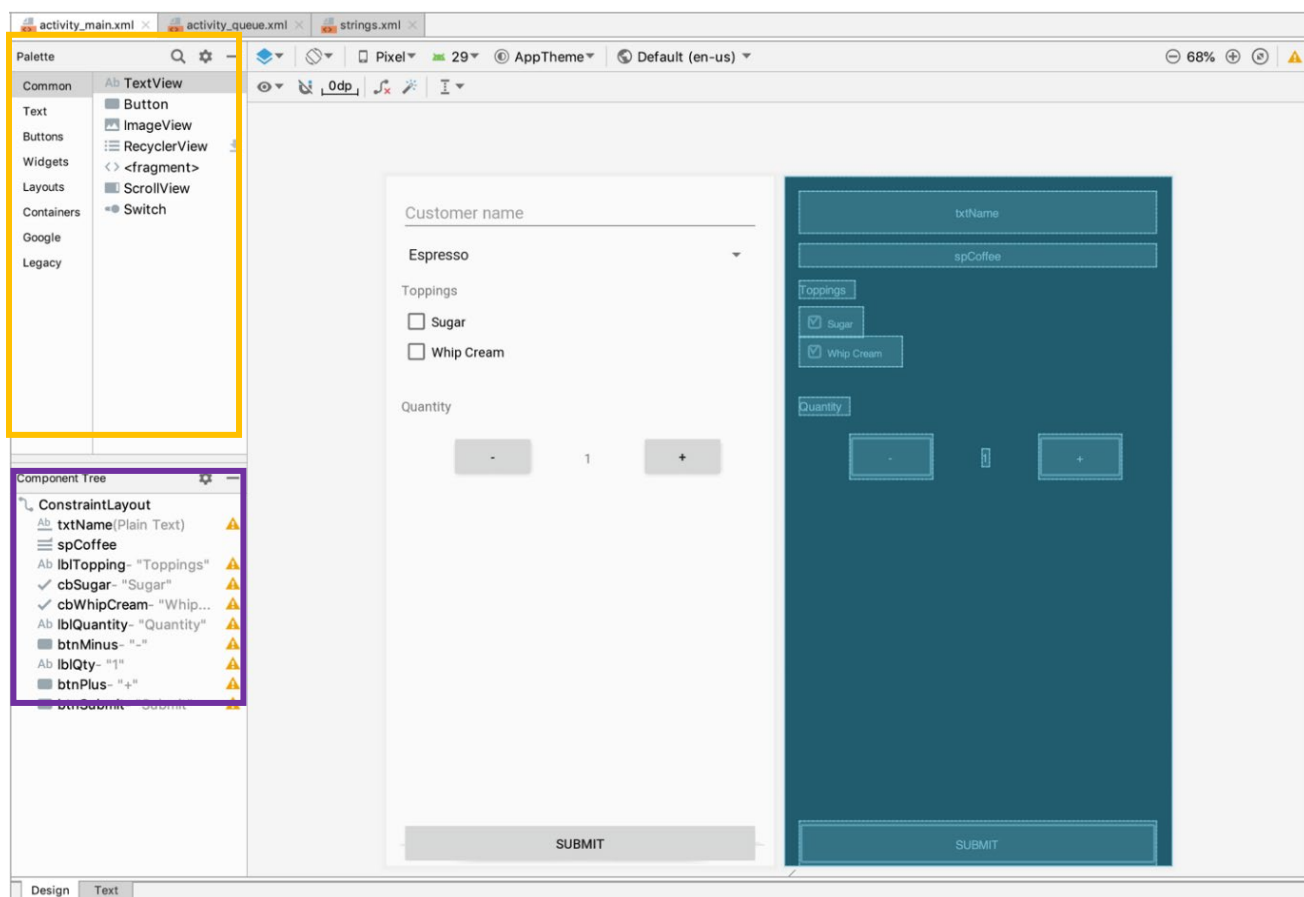
In this Java Bean tutorial, we will develop a simple app that allows users to order coffee. We will first start with designing a menu and proceed further to accessing two web-services to order some coffees and to show users when their order will be ready.

### 4.1. User interface design

Each screen of the app is a so-called **Activity**; recall that you have chosen **Empty Activity** as the first screen of your app when setting up the project.

Firstly, we will develop the user interface of the main activity. In the project tree (the left menu, **red** in overview image), look for the `activity_main.xml` file. If you have chosen the “Android” view of the project (check the top of the project tree, this is the default view), this file should be in the `app/res/layout` folder. Opening this file will show the design view editor. At the top-right corner you can switch between Code, Split and Design view. Though code (xml) view gives you more direct control, it's best to start off in design view to get an understanding of how interfaces are built. While using design view, it's a good idea to frequently check the Code tab to learn how the XML-code changes. Only use Code view if you understand what you are doing! For more info about XML, check [Appendix D: XML](#).

#### 4.1.1. Palette window





In the Palette window (area in orange) you can see all kinds of user interface (UI) components or “Views” (subclasses of android.view.View). Some of the common UI components are

- Widgets and Text: visible elements in the layout (buttons, labels, all kinds of text fields, ...)
- Containers: elements that can contain widgets and other containers (tool bar, scroll view, ...)
- Layouts: elements to lay out the different widgets in a container

#### 4.1.2. Attributes

When adding UI Components to the design at least two properties are required for all: android:layout\_width and android:layout\_height. (look in the attributes window on the right – area in Green in the overview screenshot). These two properties determine the dimensions of each view. For example, some of the possible values are:

- “match\_parent”: fill the entire parent element
- “wrap\_content”: as big as necessary to show the content

Every widget (so-called view) needs a unique identifier or an “id”, to address this widget from within the Java code. Provide for every widget you add a meaningful id, in the format “@+id/<your id>”. Step-by-step, try to create the design as shown in the picture on the right - keep an eye on the XML “Component Tree”. But first remove the default HelloWorld! Textfield from the component tree (area in purple).

## 4.2. Building our first interface in design mode

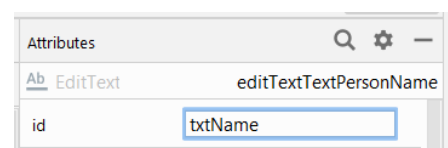
Let’s build our first activity.

### a) ConstraintLayout

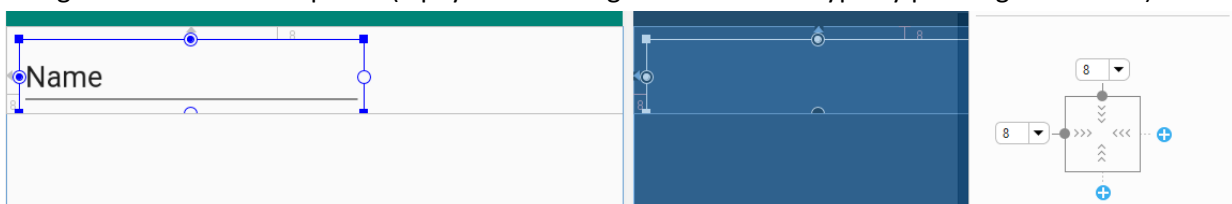
the activity should already contain a constraint layout. Child elements of a constraint layout can be organized by defining relationships between them, or between elements and the screen edges. You need to constrain every view at least one time horizontally and one time vertically.

### b) Plain Text (from palette category Text)

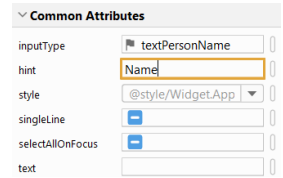
Add a Plain Text field (= editable textfield) to the layout to allow the users to type their name. This field will have an id to allow us to get a reference in the code. Whenever we first declare an id in XML mode or Code mode, we need to write “@+id/nameOfTheID” (note the + sign); in Design mode you do not have to type the “@+id/” sequence, because Android Studio will add it. Give it the id txtName (Android Studio will start the Rename refactoring). If you get an error icon next to the text field in the component tree, with an error saying “Touch target size too small”, scale the text field up.



Then we are going to constraint the view. Drag an arrow from the upper circle to the top border of the view to constraint it vertically. Finally, drag the left circle to the edge of the screen to constraint it horizontally. Now the view is constrained in both axes and will stay in the top left corner. You can add margins in the attributes panel. (Tip: you can change the constraint type by pressing on the >>>).



Instead of setting the text field to 'Name' we will set the hint to 'Name'. Again, look for this in the attributes panel. This way our users don't need to delete the text 'Name' to fill in their own name.

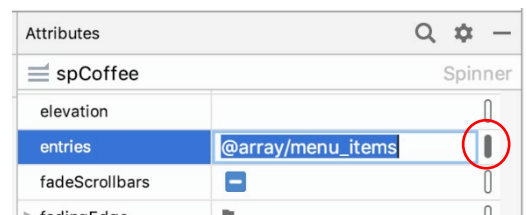


**c) Spinner (from palette category Containers)**

Add a spinner (with id spCoffee) to choose between Espresso, Cappuccino and Latte. Add menu items (string array) to res/values/strings.xml, so your string.xml looks like the following:

```
<resources>
  <string name="app_name">Java Bean</string>
  <string-array name="menu_items">
    <item>Espresso</item>
    <item>Cappuccino</item>
    <item>Latte</item>
  </string-array>
</resources>
```

Choose @array/menu\_items as entries of the spinner by changing the entries attribute in activity\_main.xml (see screenshot on the right). Add constraints to the left of the edge of the screen and to the bottom of the Name field. Note you can also select the array from a list by clicking the small button on the right of the field (marked by the red circle).



**d) TextView**

Add a TextView ("Toppings") to the layout. This is a field that is not changeable by the user, however, this field can be changed by the app. Constraint it horizontally to the left of the edge of the screen and vertically to the bottom of the Spinner. Give it the id lblTopping. You can add a reference to a value defined in strings.xml instead of the hard-coded text in the activity\_main.xml (the IDE gives a warning and a "hint" to solve it).

**e) Checkbox (two times)**

Add checkboxes (from Buttons) for 'Sugar' and 'Whip Cream' and put them below the toppings TextView. Do not forget to constraint them. Give them the id cbSugar and cbWhipCream.

**f) TextView ("Quantity"), with id lblQuantity (and do not forget the constraints).**

**g) Add the following components and drag them in an orderly fashion**

**i) Button**

Add a button to hold a minus sign and give it the id btnMinus.

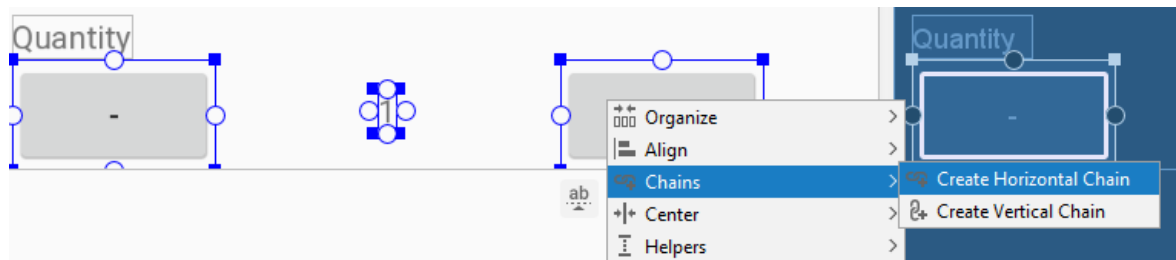
**ii) TextView**

Add a TextView to hold the current amount of coffees ordered and give it the id lblQty and value 1.

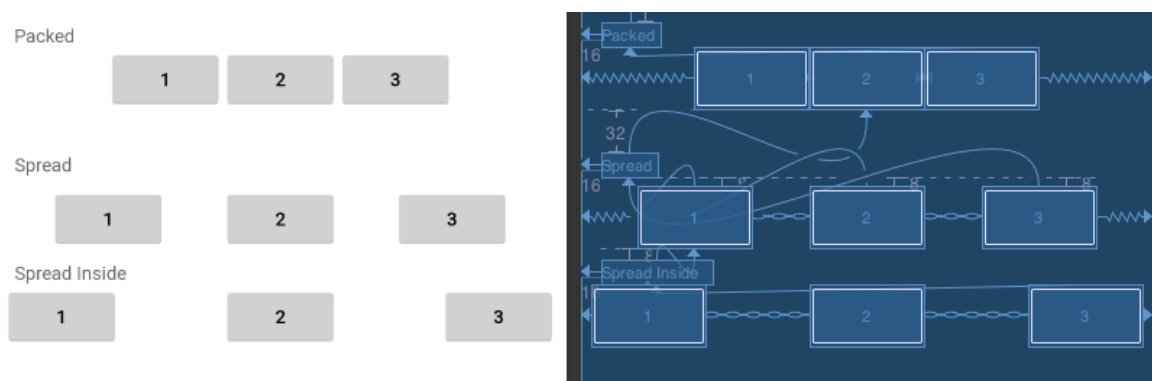
**iii) Button**

Add another button to hold the plus sign. Define vertical constraints and give it the id btnPlus.

Then select all three components while holding ctrl, right click, and choose Create Horizontal Chain.




Chains allow you to control the space between elements and how the elements use the space. It is possible to cycle through the different chain modes. There are four different modes: Packed, Spread and Spread\_inside. You can cycle by pressing the chain logo while hovering over a view object in the chain. For now, we will keep it on Spread.



#### h) **Button**

Add a button ("Submit") that will be used to submit the order and give it the id `btnSubmit`. Constrain it to the left and bottom edges of the screen.

Take your time to play around with the settings of the layout editor (take a look at the advanced properties and follow the following link to the Android Developer documentation for more information about the UI properties: <https://developer.android.com/guide/topics/ui/index.html>).

At this moment, we designed the user interface of the app, but the app has no logic yet. But first, It is time to run your JavaBean app and have a look at the UI. Click the green 'Run App' arrow .

As you can see, everything seems to work on the surface. We can type our name in the field, check the boxes, and push the buttons. But nothing happens when buttons are pushed. Next we will add functionality to these visual components.

## 5. EVENT HANDLING & VIEW MANIPULATION

### 5.1. Callback functions for buttons

Recall [3.2.1. Project structure](#) and [3.2.3. Event-driven system](#), every UI xml file has a corresponding Java file where all logic resides. When we compile our app, Android Studio will automatically search for corresponding Java files to implement said logic. To program `activity_main.xml`, we work with the `MainActivity.java` file in the Java folder. Take note that this class, like any Activity, inherits from `AppCompatActivity`.

The `MainActivity` class comes with the `onCreate(Bundle savedInstanceState)` method, which overrides a method from its superclass. This is a lifecycle callback method that is called by the Android system when an activity is first created; here it sets up the correct view. See [Appendix B](#) for the other lifecycle callback methods. The `onCreate()` method performs a similar function than a constructor would in a regular Java class (note that you never manually call the constructor of an Activity).

We will now implement `+` (`btnPlus`) and `-` (`btnMinus`) button handlers which update the quantity (`lblQty`) in `TextView` in the middle once any of the buttons are clicked.

To handle the click event of `+` (`btnPlus`), add the following code to your `MainActivity.java` file:

```
package be.kuleuven.gt.javabeen;

import ...;

public class MainActivity extends AppCompatActivity {

    private Button btnPlus;
    private TextView lblQty;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        btnPlus = (Button) findViewById(R.id.btnPlus);
        lblQty = (TextView) findViewById(R.id.lblQty);
    }

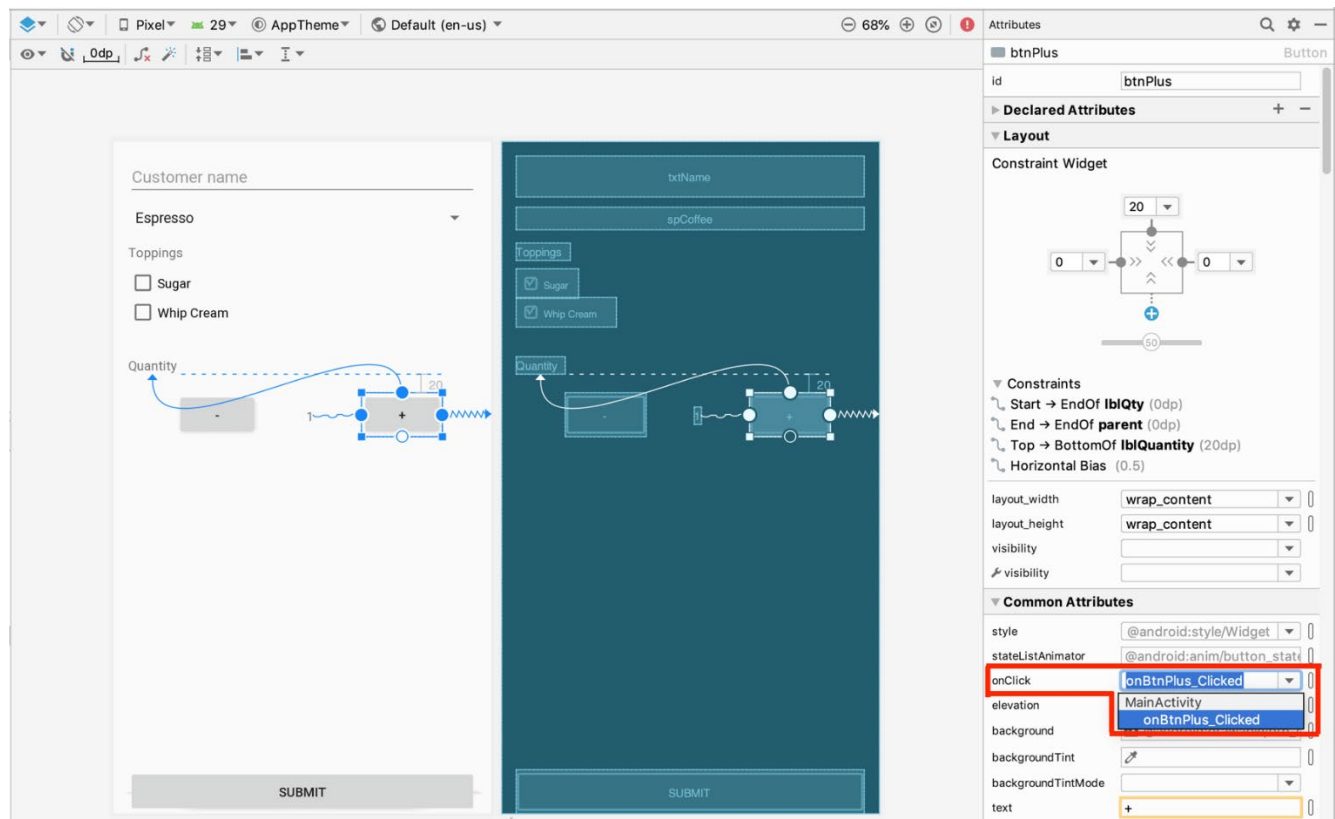
    public void onBtnPlus_Clicked(View Caller) {
        int quantity = Integer.parseInt(lblQty.getText().toString()) + 1;
        lblQty.setText(Integer.toString(quantity));
    }
}
```

Some things to note:

- `setContentView(R.layout.activity_main)` loads the interface we've just created and instantiates Java objects for all the view elements.
- `findViewById()` is the method you can use to retrieve individual view objects from the layout. Here we store both the plus button and the quantity label as attributes, so we can access them later.
- To access an id, we use the syntax `R.id.<some_id>`. You'll notice that all the ids you created during your construction of the layout are visible in the autocomplete.
- The casts of the `findViewById` results to `Button` and `TextView` are optional.
- It is crucial that you only call `findViewById` after calling `setContentView`. Beforehand, the view elements do not exist yet. Try it out, your app will crash if you change the order of these operations.
- Any method with a parameter of type `View` can be used as an event callback. The name does not matter, but it's good practice to use a descriptive name.

Make sure you understand the code before continuing.

Then on the activity\_main.xml file, click on the plus button and change the onClick property of the button to “onBtnPlus\_Clicked” as shown in the figure below (recall static event handler assignment):



Run your code and try press the + button and see the effect. Recall [3.2.3. Event-driven system](#) ; what happened is when the + button is clicked (= the event), the public void onBtnPlus\_Clicked(View caller) callback method is executed as a handler. Error messages appear under the “Run”-tab (ALT-4), detailed logs (including error messages) can be found under the Logcat-tab.

**Challenge:** Implement the onClick event of the – (minus) button. Keep in mind that the quantity should not go below 0 and if the quantity is 0, the user shall not be able to press the submit button (setEnabled method). Do not forget to couple the listener/callback method to the button. Check if the submit button is enabled/disabled as it should (in case the quantity is 0, it should be disabled); to realize this, you will have to add some code to the btnPlus listener too.

You can use “shift-F10” to re-run your app.

## 5.2. Domain class: Coffee order

This application is about ordering coffee, so it is a good OO principle to add a CoffeeOrder class to keep the necessary data of a coffee order in one object and implement some business logic. We will use a class implementation (it is also possible to start with a record implementation). To the right is the first iteration of the CoffeeOrder UML diagram.

CoffeeOrder
-name: String -coffee: String -sugar: boolean -whipCream: boolean -quantity: int
+CoffeeOrder(n: String, c: String, s: boolean, w: boolean, q: int):ctor (getters when necessary, no setters)

To organize your code, put the CoffeeOrder class in a separate package: “be.kuleuven.gt.javabean.model”.

### 5.3. Intents

*You can think of an intent as an “intent to do something”. It is a type of message that allows you to bind separate objects (such as activities) together at runtime. If one activity wants to start a second activity, it does this by sending an intent to Android. Android will start the second activity and pass it the intent.* (from: Head First Android Development, ISBN-13: 978-1449362188, O’Reilly).

In this tutorial, we want to open a new activity and pass data (a coffee order) to this new activity. Afterwards we will add code to place the order by calling a RESTful web service.

- Create a new empty activity called “OrderConfirmationActivity”. It comes with its own layout file: “activity\_order\_confirmation.xml”. Place a TextView with id txtInfo as a child in the ConstraintLayout. Later we will add another Button to this layout.
- Add the following method to your MainActivity class, and don’t forget to assign this method as the handler for the onClick event of the Submit button (btnSubmit) in main\_activity.xml file. Run your app and check that it switches to the new activity.

```
public void onBtnSubmit_Clicked(View Caller) {  
    Intent intent = new Intent(this, OrderConfirmationActivity.class);  
    startActivity(intent);  
}
```

As you can see, it creates an Intent and starts a new activity with this Intent. If you click the Submit button, your app will switch to this new Activity.

### 5.4. Passing data between activities

Next, we want to pass the coffee order data from the main screen to this new OrderConfirmationActivity. This data can be passed to the next activity as a “dataload” on the intent. You can adapt the code like this:

```
public void onBtnSubmit_Clicked(View Caller) {  
    // create CoffeeOrder from UI data  
    EditText txtName = (EditText) findViewById(R.id.txtName);  
    Spinner spCoffee = (Spinner) findViewById(R.id.spCoffee);  
    CheckBox cbSugar = (CheckBox) findViewById(R.id.cbSugar);  
    CheckBox cbWhipCream = (CheckBox) findViewById(R.id.cbWhipCream);  
  
    CoffeeOrder order = new CoffeeOrder(  
        txtName.getText().toString(),  
        spCoffee.getSelectedItem().toString(),  
        cbSugar.isChecked(),  
        cbWhipCream.isChecked(),  
        Integer.parseInt(lblQty.getText().toString())  
    );  
    Intent intent = new Intent(this, OrderConfirmationActivity.class);  
    intent.putExtra("Order", order);  
    startActivity(intent);  
}
```

A syntax error will appear on the `intent.putExtra("Order", order)` line: “Cast 2<sup>nd</sup> argument to `android.os.Parcelable`” (It might say “Serializable” or “CharSequence” instead of “Parcelable”). In essence the problem is this: passing data between activities in Android is done with a Bundle object. This object is a kind of map (key-value pairs) that can only contain the following text (String) or primitives (int, float, ...). And since CoffeeObject is neither, it cannot be added to this bundle.

The solution is to provide a way to convert a `CoffeeOrder` object to a string representation, pass it between activities, and then convert it back again. We could do this by making use of a `toString` method in `CoffeeOrder`, but this would make converting back quite unwieldy. Fortunately, Java provides some interfaces that make this simpler for us: `Serializable` or `Parcelable`. They both do essentially the same thing, but `Serializable` (the standard Java way) is quite slow. Therefore Android introduced the more efficient `Parcelable` interface, which is the one we will use.

Let the `CoffeeOrder` class implement this interface (“... implements `Parcelable`”), 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. You can use some of the provided methods of the `Parcel` class to add your fields to it, depending on their type (`writeString`, `writeBoolean`, `writeInt`, etc). Note: in some older versions of the API, there is no “`parcel.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!). If the `writeBoolean()` method is available, use it instead. The order in which you write the different values is important when loading the object again from the bundle.

To convert back from a `Parcel` to a `CoffeeOrder`, provide a **second constructor** for `CoffeeOrder` that takes a `Parcel` as parameter and restore the fields following the order and strategy of the `writeToParcel` method.

Finally, 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.” Let your IDE automatically add this field.

This is the `CoffeeOrder` implementation after implementing the `Parcelable` interface and adding the second constructor:

```
public class CoffeeOrder implements Parcelable {
    private String name;
    private String coffee;
    private boolean sugar;
    private boolean whipCream;
    private int quantity;

    public static final 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];
        }
    };

    public CoffeeOrder(String name, String coffee, boolean sugar,
        boolean whipCream, int quantity) {
        this.name = name;
        this.coffee = coffee;
        this.sugar = sugar;
        this.whipCream = whipCream;
        this.quantity = quantity;
    }

    public CoffeeOrder(Parcel in) {
        this(
            in.readString(),
            in.readString(),
            Boolean.parseBoolean(in.readString()),
            in.readByte() == 1,
            in.readInt()
        );
    }

    @Override
    public int describeContents() {
        return 0;
    }

    @Override
    public void writeToParcel(Parcel parcel, int i) {
        parcel.writeString(name);
        parcel.writeString(coffee);
        // workaround 1 for writeBoolean
        parcel.writeString(Boolean.valueOf(sugar).toString());
        // workaround 2 for writeBoolean
        parcel.writeByte(whipCream ? (byte) 1 : (byte) 0);
        parcel.writeInt(quantity);
    }
}
```

After the order has been passed to the intent and received by the OrderConfirmationActivity, we can use the information it contains inside this new activity. For now we will simply print the info in the textfield. You can build the string in the OrderConfirmationActivity's onCreate method, or better yet, you could add a toString method to the CoffeeOrder class and use that (this is the approach we have used here).



```
public class OrderConfirmationActivity extends AppCompatActivity {  
    private TextView txtInfo;  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_order_confirmation);  
        txtInfo = (TextView) findViewById(R.id.txtInfo);  
        CoffeeOrder order = (CoffeeOrder) getIntent().getParcelableExtra("Order");  
        txtInfo.setText(order.toString());  
    }  
}
```

## JavaBean 2023

```
CoffeeOrder{name='jeff', coffee='Cappuccino', sugar=true,  
whipCream=false, quantity=2}
```

You may be wondering why we need to convert the Parcel, which is essentially a text representation, back into a CoffeeOrder, since we're immediately converting this order back into a string again – this time with the toString method. The answer is that the display we are using here is simply a quick example, in reality we will likely want to use the information in the CoffeeOrder object for more complex calculations, in which case having an object to work with provides us with a lot more flexibility. The next chapter [6. Working with external data sources](#) provides an example.

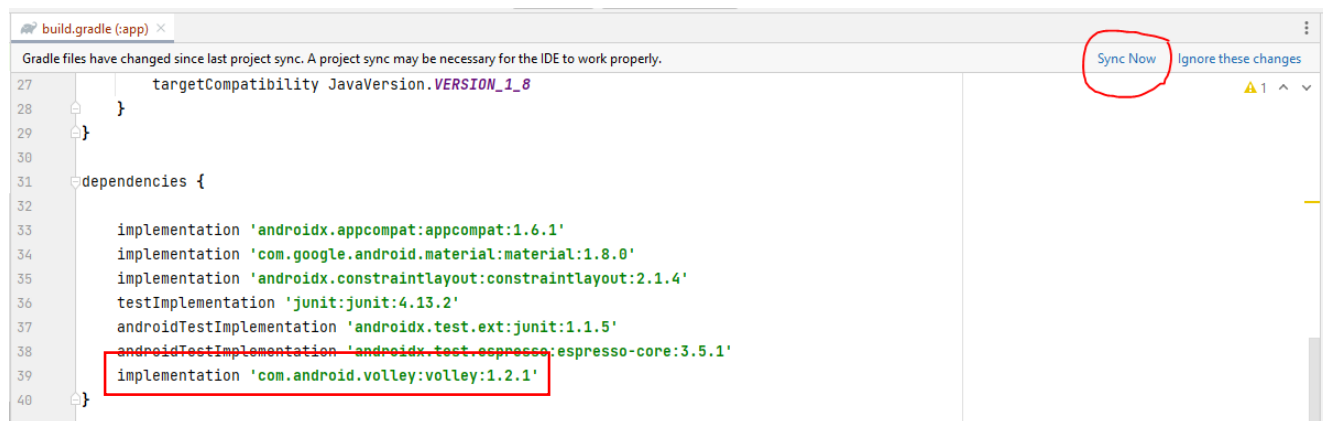
## 6. WORKING WITH EXTERNAL DATA SOURCES

Our coffee ordering app as it is now, is somehow boring and unrealistic. This is because between invocations the app does not keep any data, and the customers do not know when their coffee will be ready. In this tutorial, we will make this app communicate with a server that takes coffee orders and allows the app to inquire a coffee order list from a waiting queue. The communication with the server is done through a RESTful web service. Explanation about web service architecture will be given in the lab session and is also available in a short video on Toledo.

### 6.1. Adding Volley to the project

#### 6.1.1. build.gradle

We will use an external library “Volley” to facilitate the app-server communication through http-calls; to do so, add the implementation `'com.android.volley:volley:1.2.1'` line to the dependency of build.gradle (Module:app). You can find this file in the project tree (left menu) under Gradle Scripts. As a result, your build.gradle should look like the following:



Don't forget to click **Sync Now** to let Android Studio download volley and add it to your project.

#### 6.1.2. AndroidManifest.xml

Additionally, before you can run and test your application, you will have to give it the permission to do an INTERNET request. Add this line in the AndroidManifest.xml file, before the closing `</manifest>` tag:

```
<uses-permission android:name="android.permission.INTERNET"/>
```

### 6.2. Back-end

#### 6.2.1. Database

All information is stored in a MySQL database, which is hosted on a Groep T server. For this tutorial, a pre-made database is provided, while for your project you will have to create a database yourself. In both cases you can use MySQL Workbench to connect to the database, using the following credentials:

- Host : mysql.studev.groept.be
- Port : 3306

- Login : [YOUR\_STUDEV\_ACCOUNT]
- Password : [YOUR\_STUDEV\_PASSWORD]

**For this tutorial, both the login and password are ptdemo.** You can login with these credentials to check the table structure and content, but please don't make any changes as this will cause the tutorial to stop working for everyone.

**For your project, your login credentials will be provided during the lab sessions.**

**Important:** the database server can only be accessed while on the campus, or while using VPN. To get your MySQL workbench to work with the server, your development computer must be connected to any of the following Wifi: Campusroam, Campusroam 2.4, eduroam, eduroam 2.4 in the campus. From outside the campus you can install a [SSL VPN Pulse client](#) to create a VPN (Virtual Private Network) and connect to the B-zone.

### 6.2.2. Webservice

The web service is nothing more than a web application running on a web server which interacts with the database server. You can add custom services at [https://studev.groept.be/api/\[YOUR\\_STUDEV\\_ACCOUNT\]](https://studev.groept.be/api/[YOUR_STUDEV_ACCOUNT]), and log in with your studev account – to be provided in the class. The credentials are identical to the ones for the database server. Instructions & explanation to be provided in the lab session, and by two short videos on Toledo.

In short: you can write your own (parameterized) queries to be executed on the database, each of which will be assigned a unique URL. It is this URL that your Android app connects to by using Volley.

### 6.3. Sending data

Let's write our first web request code. It will take the information stored in a CoffeeOrder, and insert it in the database. Modify the OrderConfirmationActivity class with the code below.

Because HTTP requests can take some time, the Volley library executes them in a **asynchronous** way: once the request is submitted, the program does not wait until the response arrives, but **makes use of a callback** method that is called when the response arrives. In the meantime the rest of the code is being executed.

The most important part of this implementation is the POST request. A full explanation of a volley request is given in a couple of short videos on Toledo, but after the code snippet you can find a **quick explanation** of the most important parts of the code.

```

public class OrderConfirmationActivity extends AppCompatActivity {
    private TextView txtInfo;
    private static final String POST_URL = "https://studev.groept.be/api/ptdemo/order/";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_order_confirmation);
        txtInfo = (TextView) findViewById(R.id.txtInfo);
        CoffeeOrder order = (CoffeeOrder) getIntent().getParcelableExtra("Order");
        txtInfo.setText(order.toString());

        ProgressDialog progressDialog = new ProgressDialog(OrderConfirmationActivity.this);
        progressDialog.setMessage("Uploading, please wait...");

        RequestQueue requestQueue = Volley.newRequestQueue(this);
        StringRequest submitRequest = new StringRequest(
            Request.Method.POST,
            POST_URL,
            new Response.Listener<String>() {
                @Override
                public void onResponse(String response) {
                    progressDialog.dismiss();
                    Toast.makeText(
                        OrderConfirmationActivity.this,
                        "Post request executed",
                        Toast.LENGTH_SHORT).show();
                }
            },
            new Response.ErrorListener() {
                @Override
                public void onErrorResponse(VolleyError error) {
                    progressDialog.dismiss();
                    Toast.makeText(
                        OrderConfirmationActivity.this,
                        "Unable to place the order" + error,
                        Toast.LENGTH_LONG).show();
                }
            }
        ) { //NOTE THIS PART: here we are passing the POST parameters to the webservice
            @Override
            protected Map<String, String> getParams() {
                /* Map<String, String> with key value pairs as data load */
                return order.getPostParameters();
            }
        };

        progressDialog.show();
        requestQueue.add(submitRequest);
    }
}
    
```

### 6.3.1. Defining the submitRequest

These lines of code initialize the submitRequest variable by calling its constructor with 4 parameters (to be clear: this is the **initialization** of the “submitRequest” object of type StringRequest; nothing is executed yet):

1. the http-request method to be used (GET or POST; in this case we use POST)
2. the request URL to be used (make sure this URL does not contain whitespace characters)

3. the callback method to be called in case of a normal response, defined by an anonymous inner class that implements the `Response.Listener` interface (here we show a “toast” message that everything went well)
4. the callback method to be called in case of an error, defined by an anonymous inner class that implements the `Response.ErrorListener` interface (here we show a “toast” message that something went wrong, the order could not be placed)

This request is **not** submitted to the server yet, it only calls the constructor of `StringRequest`!

### 6.3.2. Sending the `submitRequest`

The method call

```
requestQueue.add(submitRequest);
```

will send the `submitRequest` and starts the execution of this request.

### 6.3.3. Passing the POST parameters

In a POST request, the parameter key/value pairs are not part of the URL (as with a GET request), but are sent as a separate data part. In Volley – this feature is not well documented – this is done with the extra “`getParams()`” method that returns a `Map<String,String>` with the keys and the values. Because the `CoffeeOrder` objects has all data needed, we implemented this in the “`getPostParameters()`” method in the `CoffeeOrder` class. Make sure the keys use the correct names:

```
public Map<String, String> getPostParameters() {
    Map<String, String> params = new HashMap<>();
    params.put("customer", name);
    params.put("coffee", coffee);
    params.put("toppings", getToppingsURL());
    params.put("quantity", String.valueOf(quantity));
    return params;
}

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

### 6.3.4. POST vs GET

There are two important types of web requests to distinguish: POST and GET. You can choose which one to use for each call. A full explanation falls outside the scope of this tutorial, but broadly speaking it’s best to use **POST** requests for calls that **insert or update data in** the database (as is the case here), while **GET** requests are best for calls that **retrieve data** from the database (see [6.4. Show the queue of coffee requests with `ShowQueueActivity`](#) for an example).

In Android, POST and GET requests **differ in the way that parameters** are passed to the request. For **POST** requests, you need to define the **`getParams` method** as explained in the previous section. For **GET** requests, parameters **are appended to the URL**.

GET is slightly simpler to use, but POST provides additional **safety** over a GET request.

### 6.3.5. Showing progress

We added a simple progress dialog, but since the request goes so fast, it is hardly visible. We left the code in as a demonstration of how to do this in case your app needs to run requests that take more time. Do not forget to hide the dialog when the request has finished, either in the normal way, either with an error message.

## 6.4. Retrieving data

Each order request we send with the above code will save a record in the database. Upon inserting a record, a “date\_due” will be calculated by adding a random nr of minutes between 4 and 9 to the date and time the order was placed. This functionality is built into the database on the server, [see if you can find where!](#)

We will now add a new activity to our application which shows a queue of each request which is currently being processed (i.e. the due data is in the future). We’ve already added a webservice which retrieves this queue. You can check its output by going to <https://studev.groept.be/api/ptdemo/queue> in your web browser of choice, and you can log in to the web service to check the query that this service is connected to.

Note that the data is retrieved in JSON format, so we will have to parse this. If you don’t know what JSON is, read [Appendix C](#) first.

### 6.4.1. Confirm order button

First we need to make sure we can reach the order queue. Add a button (“Show queue”) to the activity-order-confirmation.xml file (Design or Code view, your choice). Give this button the id: “btnQueue”.

Next, create an empty activity “ShowQueueActivity” and define the following method in the OrderConfirmationActivity, to switch to this activity, and couple it with the “Show queue” button:

```
public void onBtnShowQueue_Clicked(View caller) {  
    Intent intent = new Intent(this, ShowQueueActivity.class);  
    startActivity(intent);  
}
```

### 6.4.2. Defining the queueRequest

First we will get the JSON data from the REST service. Afterwards we will show them in a scrollable view.

As first iteration of this step, you can show the JSON data in a simple TextView. Add a (temporary) TextView to the ShowQueueActivity. Choose an id and layout yourself.

Then add the following code.

```
public class ShowQueueActivity extends AppCompatActivity {

    private static final String QUEUE_URL = "https://studev.groept.be/api/ptdemo/queue";

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

    private void requestCoffeeOrderqueue() {
        RequestQueue requestQueue = Volley.newRequestQueue(this);

        JsonRequest queueRequest = new JsonRequest(
            Request.Method.GET,
            QUEUE_URL,
            null,
            new Response.Listener<JSONArray>() {
                @Override
                public void onResponse(JSONArray response) {
                    // iteration 1
                    TextView txtView = (TextView) findViewById(R.id.tempText);
                    txtView.setText(response.toString());
                }
            },
            new Response.ErrorListener() {
                @Override
                public void onErrorResponse(VolleyError error) {
                    Toast.makeText(
                        ShowQueueActivity.this,
                        "Unable to communicate with the server",
                        Toast.LENGTH_LONG).show();
                }
            }
        );
        requestQueue.add(queueRequest);
    }
}
```

These lines of code initialize the queueRequest as a JsonRequest (you can compare it with the submitRequest initialization), that returns information about the queue of coffee orders. The arguments are:

1. the HTTP-request method to be used (GET or POST; because we want to query information, GET is the appropriate method to use here)
2. the request URL to be used
3. null (this parameter is not used for this request)
4. the callback method to be called in case of a normal response; the JSON data are shown as a simple String in the temporary TextView; in the following step we will parse the JSON array, create a collection of CoffeeOrder objects and show them in a Scrollable view
5. the callback method to be called in case of an error (shows a “toast” message)

Test your code. If the order queue is not empty, you should see its JSON string representation displayed in the ShowQueueActivity.

### 6.4.3. CoffeeOrder from JSON object

To make the code more readable, we will first add another constructor to the CoffeeOrder class: it takes a JSON object as argument and we will add another field of type String: dateDue, that is part of the returned 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");
        dateDue = o.getString("date_due"); // yyyy-mm-dd hh:mm
    } catch (JSONException e) {
        e.printStackTrace();
    }
}
```

#### 6.4.4. Create the collection of CoffeeOrders

Add an extra field “orders” of type List<CoffeeOrder> in the ShowQueueActivity class:

```
private List<CoffeeOrder> orders = new ArrayList<>();
```

Next, replace the content of the onResponse part: call the method “processJSONResponse(response)”, to fill the orders collection. We’ll show the length of the collection in the text view for now, to check that it is indeed getting filled.

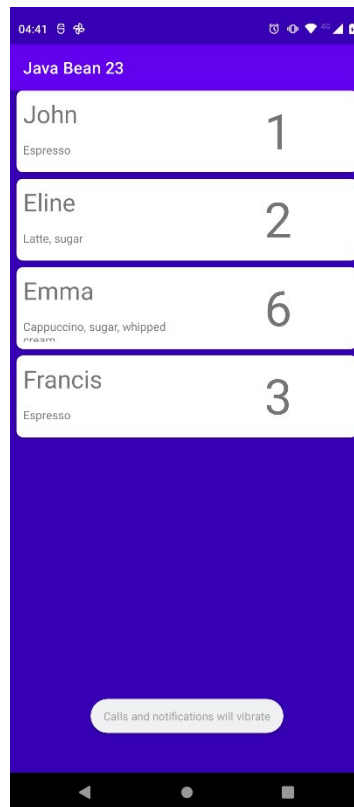
```
new Response.Listener<JSONArray>() {
    @Override
    public void onResponse(JSONArray response) {
        // iteration 2
        processJSONResponse(response);
        TextView txtView = (TextView) findViewById(R.id.tempText);
        txtView.setText(Integer.toString(orders.size));
    }
}
```

```
private void processJSONResponse(JSONArray response) {
    for (int i = 0; i < response.length(); i++) {
        try {
            CoffeeOrder order = new CoffeeOrder(response.getJSONObject(i));
            orders.add(order);
        } catch (JSONException e) {
            e.printStackTrace();
        }
    }
}
```

### 6.5. Show the orders in a RecyclerView

We’ve successfully managed to retrieve data and store it in a list, but we’re not showing it in our app yet. In this final part of the tutorial, the order list will be shown in a scrollable view. Below is a screenshot of what our activity will look like.

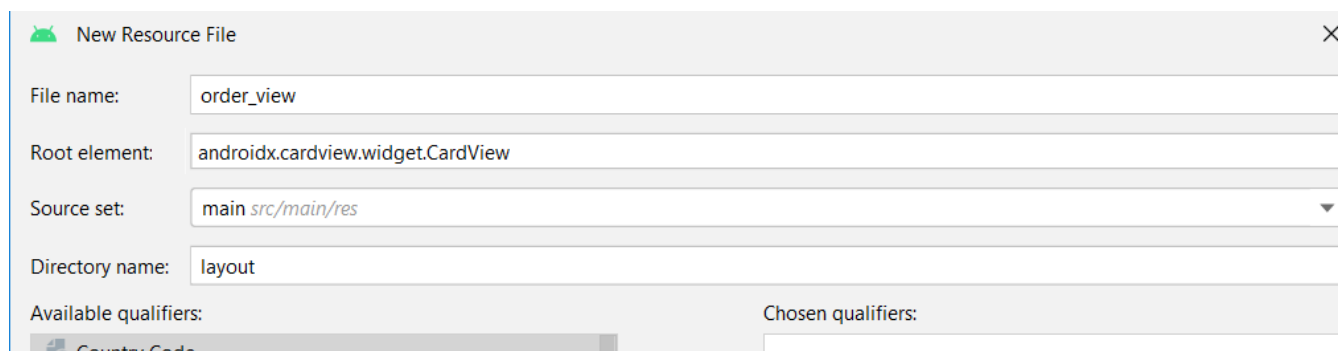




The best view to display large sets of data in Android is a RecyclerView. In [“Create dynamic lists with RecyclerView”](#) you can find more details (it is part of the [Android Jetpack](#) libraries). In summary you provide the data (in this example the list of CoffeeOrders) and a view holder (extends RecyclerView.ViewHolder) that defines how each item should be rendered. The RecyclerView combines both with a RecyclerView.Adapter.

### 6.5.1. Layout for individual orders

We need a separate layout to lay out the individual CoffeeOrders in the RecyclerView. You can use a CardView. Create a new layout file, “order\_view.xml” (the extension will be added automatically by your IDE), in the layout folder, like this:



Feel free to lay out this card as you want, here is an example layout file (order\_view.xml), with three text fields for the name, a description and the quantity of coffees ordered:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.cardview.widget.CardView xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="100dp"
    android:padding="4dp"
    android:layout_marginBottom="8dp"
    android:layout_marginLeft="8dp"
    android:layout_marginRight="8dp"
    app:cardCornerRadius="8dp">

    <androidx.constraintlayout.widget.ConstraintLayout
        android:id="@+id/cardContentLayout"
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <TextView
            android:id="@+id/orderName"
            android:layout_width="209dp"
            android:layout_height="47dp"
            android:layout_marginStart="8dp"
            android:layout_marginTop="8dp"
            android:textSize="30sp"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toTopOf="parent" />

        <TextView
            android:id="@+id/orderDetails"
            android:layout_width="209dp"
            android:layout_height="28dp"
            android:layout_marginStart="8dp"
            android:layout_marginTop="8dp"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toBottomOf="@+id/orderName" />

        <TextView
            android:id="@+id/orderQuantity"
            android:layout_width="155dp"
            android:layout_height="75dp"
            android:layout_marginStart="20dp"
            android:layout_marginEnd="8dp"
            android:layout_marginBottom="8dp"
            android:textAlignment="center"
            android:textSize="60sp"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintHorizontal_bias="0.272"
            app:layout_constraintStart_toEndOf="@+id/orderName"
            app:layout_constraintTop_toTopOf="parent" />
    </androidx.constraintlayout.widget.ConstraintLayout>
</androidx.cardview.widget.CardView>
```

### 6.5.2. Add the RecyclerView

Replace the temporary TextView in the activity\_show\_queue.xml file by a RecyclerView (id: orderQueueView). Let it fill the complete screen. You can give it a background color:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
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"
tools:context=".ShowQueueActivity">

<!-- <TextView
    android:id="@+id/tempText"
    android:layout_width="411dp"
    android:layout_height="239dp"
    android:background="#FFE57F"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />-->

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/orderQueueView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="@color/purple_700"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

### 6.5.3. Create a custom adapter

To put an object as an element in a RecyclerView, an “adapter” is needed. In this example we will create a CoffeeOrderAdapter, that makes it possible to render CoffeeOrder objects in the RecyclerView.

Let this class extend “RecyclerView.Adapter<CoffeeOrderAdapter.ViewHolder>” and override the methods:

- onCreateViewHolder: to inflate the order\_view.xml
- onBindViewHolder: to set the data in the rows of the view

It should also contain the collection of coffee orders to render. We will not explain every detail of the code, but it uses an internal class (ViewHolder) to render every element in the view. This is the definition of the CoffeeOrderAdapter class:

```
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.TextView;
import androidx.recyclerview.widget.RecyclerView;
import java.util.List;
import be.kuleuven.gt.kp.javabeen2023v2.R;

public class CoffeeOrderAdapter extends RecyclerView.Adapter<CoffeeOrderAdapter.ViewHolder> {
    private List<CoffeeOrder> coffeeOrderList;

    public CoffeeOrderAdapter(List<CoffeeOrder> coffeeOrderList) {
        this.coffeeOrderList = coffeeOrderList;
    }

    @Override
    public CoffeeOrderAdapter.ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
        LayoutInflater inflater = LayoutInflater.from(parent.getContext());
        View orderView = inflater.inflate(R.layout.order_view, parent, false);
        ViewHolder myViewHolder = new ViewHolder(orderView);
        return myViewHolder;
    }

    @Override
    public void onBindViewHolder(CoffeeOrderAdapter.ViewHolder holder, int position) {
        CoffeeOrder coffeeOrder = coffeeOrderList.get(position);
        ((TextView) holder.order.findViewById(R.id.orderName))
            .setText(coffeeOrder.getName());
        ((TextView) holder.order.findViewById(R.id.orderDetails))
            .setText(coffeeOrder.getDescription());
        ((TextView) holder.order.findViewById(R.id.orderQuantity))
            .setText(Integer.toString(coffeeOrder.getQuantity()));
    }

    @Override
    public int getItemCount() {
        return coffeeOrderList.size();
    }

    static class ViewHolder extends RecyclerView.ViewHolder {

        public View order;

        public ViewHolder(View coffeeorderView) {
            super(coffeeorderView);
            order = (View) coffeeorderView;
        }
    }
}
```

In this adapter class we use 3 getters to show CoffeeOrder fields:

- String getName()
- String getDescription()
- int getQuantity()

Please implement them in the CoffeeOrder class if you don't already have them. getName() and getQuantity() should be self-explanatory. For getDescription() you can use your imagination, in our case we simply printed a string of coffee type and selected toppings, separated by commas.

#### 6.5.4. Integration

The final step is to integrate everything into our ShowQueueActivity. First, add the RecyclerView as a field, since we need to access it in two places:

```
private RecyclerView orderQueueView;
```

Most of the work is done in the onCreate() method. Here we retrieve the recycler view, create an adapter that's linked to our list of orders, connect this adapter to the recycler view, and then add a layout manager (this will make sure that the orders are vertically stacked).

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_show_queue);
    orderQueueView = findViewById( R.id.orderQueueView );
    CoffeeOrderAdapter adapter = new CoffeeOrderAdapter( orders );
    orderQueueView.setAdapter( adapter );
    orderQueueView.setLayoutManager( new LinearLayoutManager( this ));
    requestCoffeeOrderqueue();
}
```

Finally, update the onResponse listener one more time. After the new data has been retrieved, we need to notify the adapter that its data set has changed. This will trigger it to update the RecyclerView. Don't forget to remove the TextView code, since we deleted the TextView in favor of the RecyclerView.

```
new Response.Listener<JSONArray>() {
    @Override
    public void onResponse(JSONArray response) {
        // iteration 3
        processJSONResponse(response);
        orderQueueView.getAdapter().notifyDataSetChanged();
    }
}
```

Finally, run your app. You will be able to place coffee orders and have a look at the queue of placed coffee orders by you and your classmates.

Congratulations, you've made it through the tutorial. Time to put what you've learned into practice and start working on your own app!

## APPENDIX A: COUPLING LAYOUT TO CODE

There are two ways to couple actions (= Java code) with the user interface (for instance to couple a method to a button click). This can be done

- in the XML layout file
- using listeners in Java code

- a) In the XML layout file, you can link a listener/method to a button by using the `onClick` property (“increment” method):

The method should have this signature: `public void <method name>(View view)`

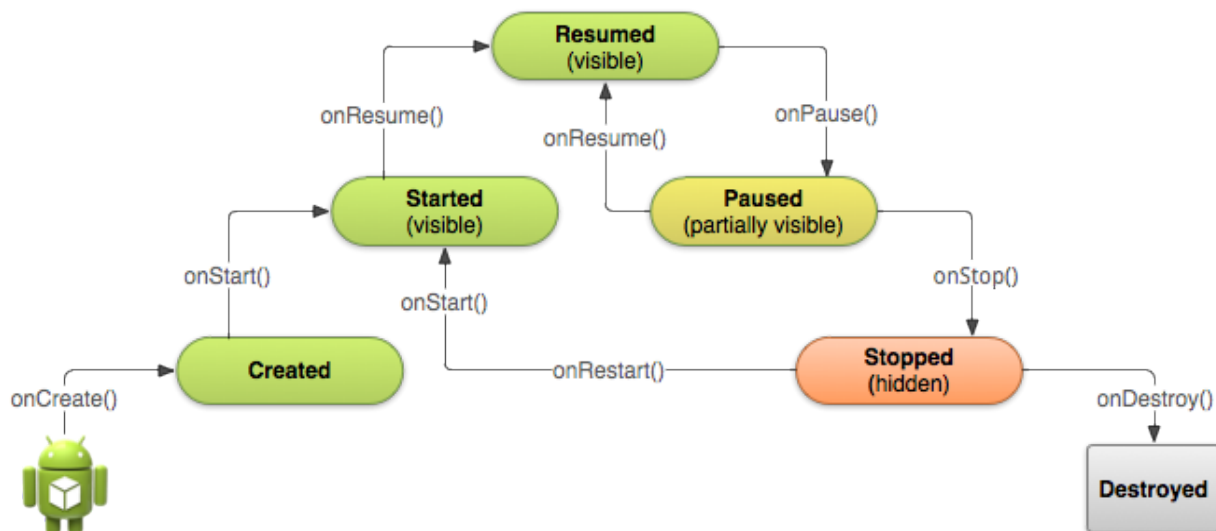
```
<Button
    android:id="@+id/increaseButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="8dp"
    android:text="@string/increase"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintStart_toEndOf="@+id/quantity"
    app:layout_constraintTop_toBottomOf="@+id/quantityText"
    android:onClick="increment"/>
```

Note that this is the approach we have used in the tutorial, except we used design view instead of XML view.

- b) In the code, you can link a listener/method to a button by setting the “`OnClickListener`” (within the `onCreate` method) with a method that implements the `View.OnClickListener` interface.:

```
Button decreaseButton = (Button) findViewById(R.id.decreaseButton);
decreaseButton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        decrease();
    }
});
```

## APPENDIX B: LIFECYCLE CALLBACK METHODS



Source: <https://developer.android.com/training/basics/activity-lifecycle/starting.html>

Here are the Android activity lifecycle callbacks methods (information from: Head First Android Development, ISBN-13: 978-1449362188, O'Reilly) and when they are called:

- `onCreate()`: when the activity is first created. Use it for normal static setup, such as creating views. It also gives you a `Bundle` giving the previous saved state of the activity.
- `onRestart()`: when your activity has been stopped just before it gets started again.
- `onStart()`: when your activity is becoming visible. It is followed by `onResume()` if the activity comes into the foreground or `onStop()` if the activity is made invisible.
- `onResume()`: when your activity is in the foreground.
- `onPause()`: when your activity is no longer in the foreground because another activity is resuming. The next activity is not resumed until this method finishes, so any code in this method needs to be quick. It is followed by `onResume()` if the activity returns to the foreground or `onStop()` if it becomes invisible.
- `onStop()`: when the activity is no longer visible. This can be because another activity is covering it or because the activity is being destroyed. It is followed by `onRestart()` if the activity becomes visible again or `onDestroy()` if the activity is going to be destroyed.
- `onDestroy()`: when your activity is about to be destroyed or because the activity is finishing.

## APPENDIX C: JSON

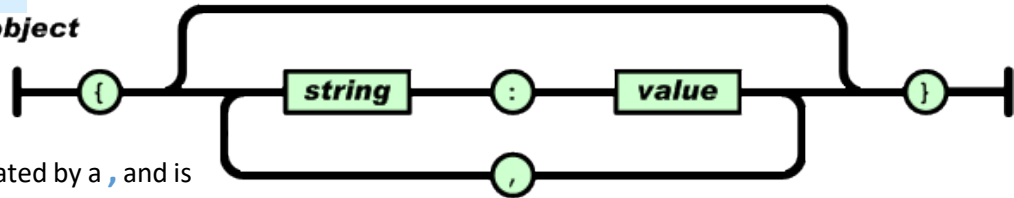
JSON (JavaScript Object Notation) is a lightweight data-interchange format. Its main advantage is that it is language independent. In other words, it can be easily used to send data from a java application to a C++ application or from a database to an android app.

There are two main structures in JSON:

### JSON Objects

Collections of key/values pairs, typically representing objects. An object starts with a `{`, followed by one or more String:Value pairs separated by a `,` and is closed with a `}`.

**object**



### JSON Object Example

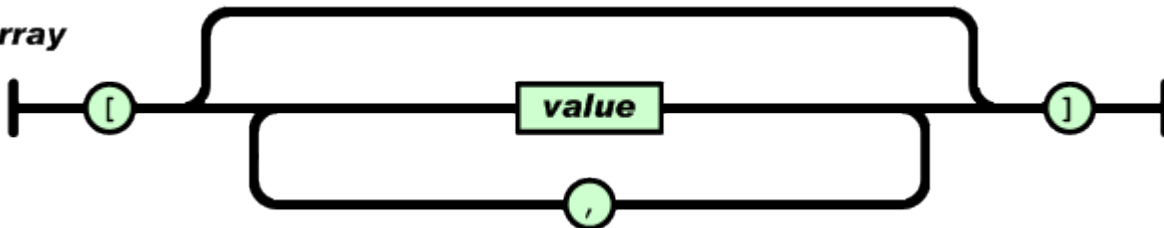
This example represents a Person object with a String name, an integer age and a String city.

```
{  
  "name": "John",  
  "age": 31,  
  "city": "New York"  
}
```

### JSON Array

An ordered list of values or JSON Objects, typically representing collections such as Lists, Arrays, Vectors, ... . An array starts with a `[` and ends with a `]`. Objects in an array are separated with a `,`.

**array**





## JSON Array Example

This JSON Array starts with a `[`, contains 3 person objects separated with a `,` and ends with a `]`.

```
[
  {
    "name": "Jonas Geuens",
    "age": 55,
    "city": "Kaapstad"
  },
  {
    "name": "Karsten Gielis",
    "age": 57,
    "city": "Brussel"
  },
  {
    "name": "Koen Pelsmaekers",
    "age": 99,
    "city": "Put-Kapelle"
  }
]
```

## Creating JSON Data in Java

This code recreates the JSONArray as found above. Please note that the first parameter of the put function represents the key and the second parameter the value with which it is associated. These key values are essential to getting your data out of JSON.

```
JSONArray jsonArray = new JSONArray();

for (Person p : list) {
    JSONObject person = new JSONObject();
    try {
        person.put( name: "title", p.getName());
        person.put( name: "age", p.getAge());
        person.put( name: "city", p.getCity());
    } catch (JSONException e) {
        e.printStackTrace();
    }
    jsonArray.put(p);
}
```

## Retrieving JSON Data in Java

Please note that in this example the JSONArray is wrapped as a value in a JSONObject (indicated with the {}) with as key "People". So to retrieve the objects, we first need to get the JSONArray from the JSONObject. Then we can iterate over all JSONObjects (Jonas, Karsten and Koen) and instantiate them as Plain Old Java Objects or "POJOs".

We assume that the JSON data is given to us as a very long String: {"People":[{"name":"Jonas Geuens", "age":55, "city":"Kaapstad" }, { "name":"Karsten Gielis", "age":57, "city":"Brussel" }, { "name":"Koen Pelsmaekers", "age":99, "city":"Put-Kapelle" }]}. JSON parsers such as <http://json.parser.online.fr/> can help us to visualize the data in a more comprehensible way. Some browsers provide add-ons to show JSON data in a pretty way.

```
{
  "People": [
    {
      "name": "Jonas Geuens",
      "age": 55,
      "city": "Kaapstad"
    },
    {
      "name": "Karsten Gielis",
      "age": 57,
      "city": "Brussel"
    },
    {
      "name": "Koen Pelsmaekers",
      "age": 99,
      "city": "Put-Kapelle"
    }
  ]
}

try {
    ArrayList<Person> list = new ArrayList<>();
    JSONObject o = new JSONObject(JSONtext);
    JSONArray array = o.getJSONArray( name: "People");
    for (int i=0; i < array.length(); i++) {
        JSONObject obj = array.getJSONObject(i);
        Person p = new Person(obj.getString( name: "name"),
                               obj.getDouble( name: "age")
                               ,obj.getString( name: "city"));
        list.add(p);
    }
} catch (JSONException e) {
    throw new RuntimeException(e);
}
```

More on JSON

<https://www.json.org/>

[https://www.w3schools.com/js/js\\_json\\_intro.asp](https://www.w3schools.com/js/js_json_intro.asp)

<http://www.vogella.com/tutorials/AndroidJSON/article.html>

## APPENDIX D: XML

XML stands for eXtensible Markup Language. It was originally designed as a comprehensible way to store and transport data (before JSON became more popular for encapsulating data). In the Android environment it is used to describe resources, for example the user interface, fixed text, animations and even icons used by your application. It decouples the design (or view) of your app from the logic behind it. This is beneficial because you can modify the user interface of your app without changing the source code. You can create XML layouts for different devices, screens and languages without duplicating logic.

Each Android XML layout typically starts with a ViewGroup. This element defines the way that child elements behave.

*Example: Two buttons (child elements) inside a linear layout (viewgroup) with horizontal orientation will be placed horizontally beside each other.*

### XML

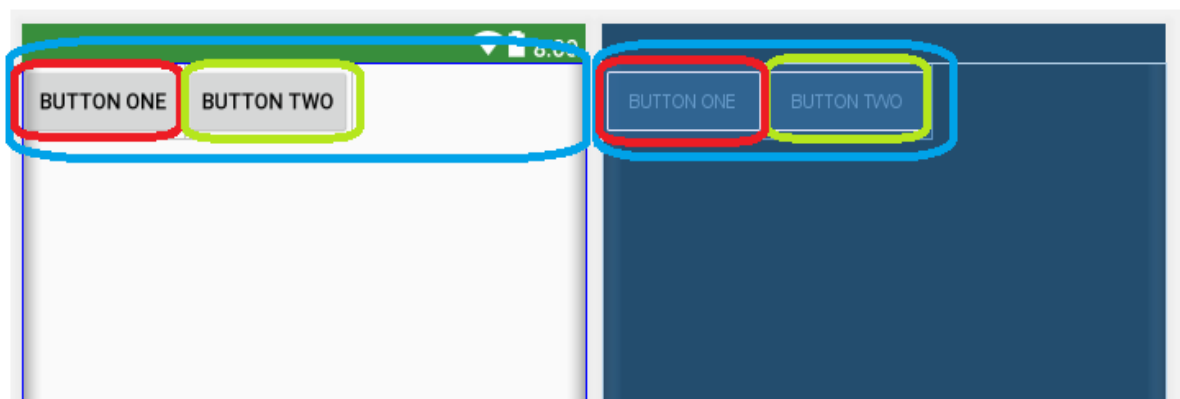
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android" android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal">

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="BUTTON ONE"/>

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="BUTTON TWO"/>

</LinearLayout>
```

### Rendered layout



Each individual layout element can have multiple properties. The android:text property, for example, defines the text of the button. Layout\_width and layout\_height define the width and height of the button. In this case this means that the button will not be bigger than needed to wrap the content.

The “android:” is used to target the namespace. What this means is that when the application is compiled, the compiler knows that it has to look in the android namespace to find, for example, the “text” field.

Please do note that the manner of defining a XML element without child nodes differs from one that has child nodes.

## Type 1: Childless XML element

The childless XML element starts with a single `<`, followed by the type of element, the properties, and ends with a `/>`.

```
<Button android:id="@+id/cancel"
        android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:layout_weight="1"
        android:text="@string/cancel" />
```

## Type 2: Child nodes

The XML element with children starts with a single `<`, followed by the type of element, the properties and a closing tag `>`. Then the child elements are defined. When all children are defined, the parent is closed with a `</type>` tag.

For each element there are multiple properties with multiple options. To style each element to your design, check the android documentation exhaustively at <https://developer.android.com/index.html>

```
<LinearLayout android:layout_width="fill_parent"
              android:layout_height="fill_parent">

    <Button
        android:id="@+id/ok"
        android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:text="@string/ok"
        android:layout_weight="1"
        />

</LinearLayout>
```

More on XML:

<https://www.w3schools.com/xml/>

<https://developer.android.com/guide/topics/ui/declaring-layout.html>

<https://developer.android.com/guide/topics/ui/declaring-layout.html#CommonLayouts>