Lab 2: Introduction to layouts

This class will teach you how to build a graphical user interface for your application, in both your tablet and your smartwatch. It will be the first step towards building a complete sport tracking application, which you will develop step by step during the upcoming lab sessions.

1 Introduction to Android User Interface

The graphical user interface (UI) for an Android app is built using a hierarchy of **`View`** and **`ViewGroup`** objects (Fig. 1). **`View`** objects are usually UI widgets such as **`TextView`**, **`ImageView`**, and **`Button`**. **`ViewGroup`** objects are invisible view containers that define how the child **`View`**s are laid out, such as in a grid or a vertical list. Android provides an XML vocabulary that corresponds to the subclasses of **`View`** and **`ViewGroup`** so you can define your UI in XML using a hierarchy of UI elements.

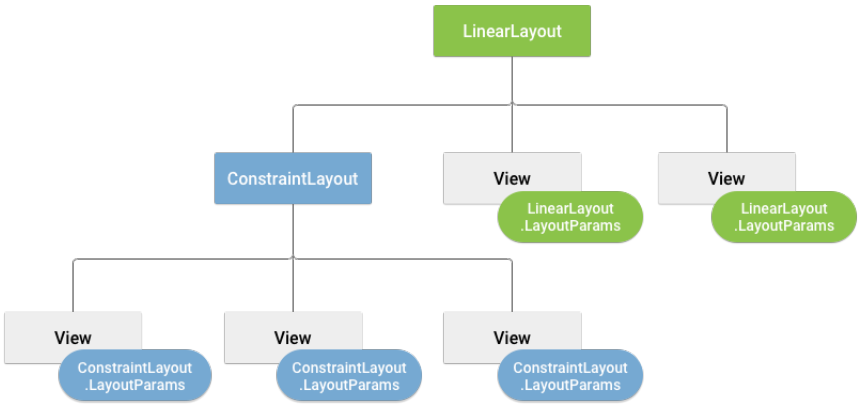


Figure 1: **`View`** hierarchy with layout parameters

In this lab session, you will create the layout for a **login `Activity`**. You will design the layout in XML including several **`View`** elements (also called *widgets*). Additionally, you will implement the execution of a code whenever a button is tapped. Finally, you will learn some particularities of Android Wear.

During the next lab sessions, you will gradually add more features to this initial application, in order to build a fully-functional **sport tracking application**.

2 Building your first Android Layout

You will start by building the application that will run on your tablet. To do so you will create a new application, design the layout of the login **`Activity`** (which at this point will be your main **`Activity`**), and implement the execution of a special code when tapping a particular button. Let's get started!

2.1 Create a new project

The very first thing to do is to check for any possible Android Studio update. Be sure you have the build tools for the latest API level (28) installed from the SDK Manager. Then, create a new "Phone and Tablet" (API level 15) and "Wear" project (API level 23), choosing an empty activity for both.

There is a small fix to manually apply in order to get the *mobile* module of the project to compile without warnings. Basically, there are conflicting versions of libraries. Therefore, you need to add the following lines in the *mobile*'s module **`build.gradle`** file:

```
dependencies {
    ...
    implementation 'com.android.support:support-media-compat:28.0.0'
    implementation 'com.android.support:support-v4:28.0.0'
    ...
}
```

You can optionally apply the recommended fix given by Android Studio about giving a specific version number to **`play-services-wearable`** library:

```
dependencies {
    ...
    implementation 'com.google.android.gms:play-services-wearable:16.0.0'
    ...
}
```

2.2 Create a `LinearLayout`

There are several kinds of layouts, each with a different purpose. Identical results can be achieved using different layouts, but their performance differs. Particularly, it might be much lower with deeply nested structures containing a large amount of widgets. We present two of the available layouts to you:

1. **LinearLayout** organizes its children into a single horizontal or vertical row. It creates a scrollbar if the length of the window exceeds the length of the screen (Fig. 2a).
2. **RelativeLayout** allows you to specify the location of child objects relative to each other, or to the parent (Fig. 2b).

(a) Linear Layout (b) Relative Layout

Figure 2: Examples of Layouts

There are other available layouts: **ConstraintLayout**, **GridLayout**, **FrameLayout**, **TableLayout** and **TableRow**. We encourage you to read about them and perhaps try to implement them.

You will be using a **LinearLayout** for this first part of the lab. Open the layout file in **mobile/res/layout/activity_main.xml**, be sure to switch to the **Text** tab at the bottom of the editor's window, and replace all the content with 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="match_parent"
    android:orientation="vertical" />
```

The above code defines a layout taking the whole screen space. The width and height are set to **match_parent**. This value declares that the **View** (or **ViewGroup** in our case) should expand its width and height to match those of the parent **View**. The orientation of the layout is set to be **vertical**, that is to say that the child **View** and **ViewGroup** elements will be stacked one under the other.

Now let's add some elements to your **LinearLayout**!

2.3 Add elements to your `Layout`

This time, we will use the **Design** tab to edit our layout. Some people consider that it is a lazy way to generate dirty code (which is an idea coming from graphical HTML editors), but the output is actually clean and has the benefits of knowing the available widgets, display a preview of the final appearance and easily change properties. Hence, it's a good tool to save time and result in something functional.

Let's start by dropping a **Plain Text** and a **Password** element from the palette in the upper left corner of the layout designer (from the *Text* category). Then, drop a horizontal **`LinearLayout`** under the two previously added widgets (from the *Layouts* category). Now you will add two buttons inside the horizontal **`LinearLayout`** simply by dragging and dropping them there (select **Button** from the *Buttons* category). The two buttons will appear side-to-side because the alignment inside the **`LinearLayout`** was set to be horizontal.

Now let's add an **ImageView** at the top of the layout. First, Download it from Moodle (**`ic_logo.png`**). To use this custom image in your project, you need to copy it in the **`mobile/src/main/res/drawable`** directory, **or**, in Android Studio, right-click on the **`res`** folder and using the menu to add an image resource. This second solution have the benefit of generating automatically optimized variations of the image according to the different screen densities available. You probably want to check the **`adjustViewBounds`** box in the *Properties* panel on the right so the image container wraps around the image itself.

Finally, add a **`Space`** elements in the top and bottom of the layout (from the *Layouts* category). The result is shown in Fig. 3.

As we dropped elements in a vertical linear layout, the widgets have been added with a **`layout_width`** set to **`match_parent`** (i.e. match the containing layout's width, which in turn uses the full screen's width). The **`layout_height`** is set to **`wrap_content`** because in such a vertical ordering, it makes sense to pack the widget's height to their respective content. The reverse is done for the widgets inside the horizontal **`LinearLayout`**.

Now, rename the IDs of the elements. You can do that in the **Design** tab by selecting each widget and updating the **id** attributes panel that will appear on the right of the editor. Alternatively, you can do it in the **Text** tab by updating the **`android:id`** field. Use the following names for the widgets:

1. **AppIcon** for the image view
2. **Username** for the plain text
3. **Password** for the password
4. **LoginButton** for the button on the left
5. **RegisterButton** for the button on the right

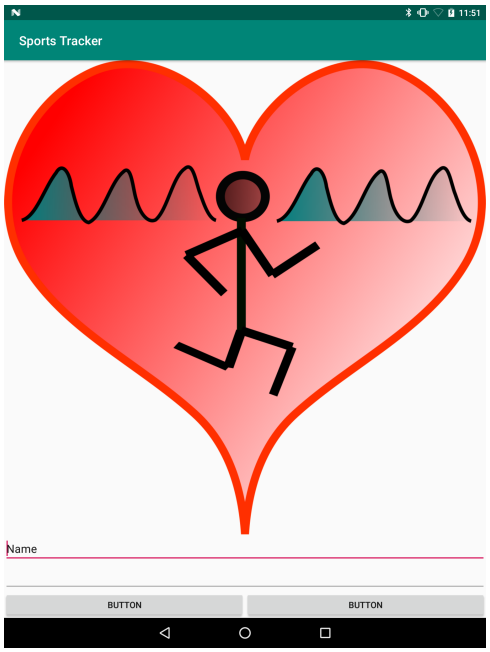


Figure 3: View of the initial Layout

2.4 Customizing the Views sizes

Relying on the **match_parent** and **wrap_content** sizes is not really flexible to create the layout we aim for. It is possible to set a given fixed size to each element in the layout, but this has the drawback of not scaling well across devices with different screen sizes. Thankfully, there is an alternative solution, which is to set the **layout_weight** of the elements. Let's try it out!

Now go switch to the **Text** tab and set the **layout_height** field to **0dp**, then set **layout_weight** to **1** for all the elements of the vertical **LinearLayout**:

```
<Space
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_weight="1"/>
```

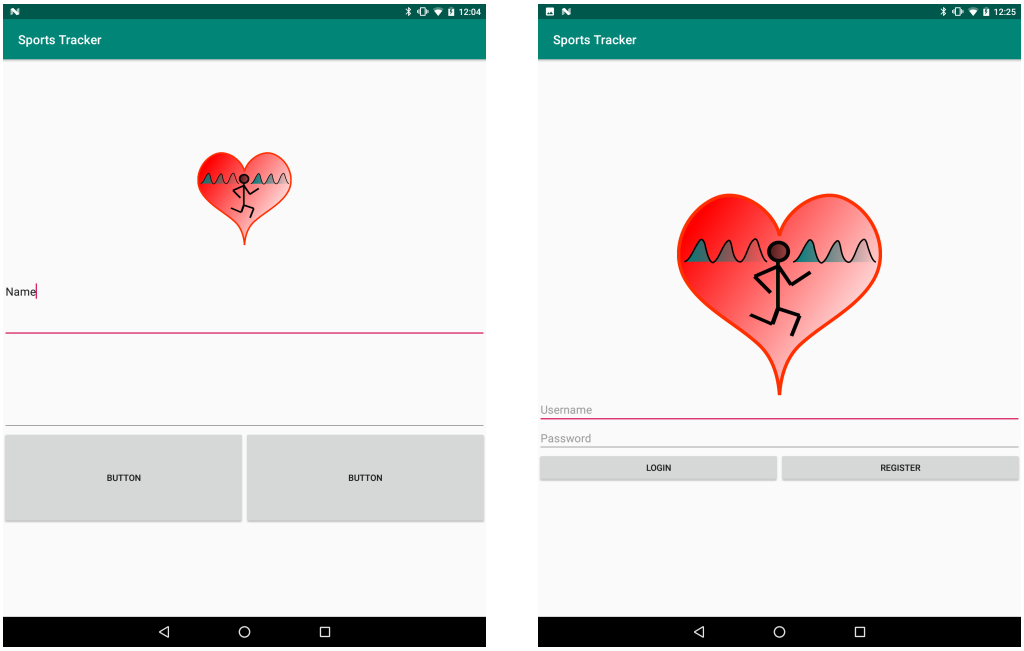
Then, for the elements in the nested horizontal **LinearLayout**, set **layout_width** to **0dp**, **layout_height** to **match_parent**, and **layout_weight** to **1**:

```
<Button
    android:id="@+id/LoginButton"
    android:layout_width="0dp"
    android:layout_height="match_parent"
```

```
android:layout_weight="1"
android:text="Button" />
```

Setting the **layout_height** to **0dp** when using weights is strongly advised because otherwise the system computes a height relative to the **View**'s content and parent, to finally use a height that is computed in a different way.

In the end, your layout will contain six widgets where each takes exactly 1/6th of the available screen's space (vertically) as shown in Fig. 4a.



(a) Layout with uniform widget weights

(b) Layout after customizing the widget sizes and resetting the text inside the buttons

Figure 4: View of the layout with uniform weights and after customizing the sizes of the widgets

Now, let's fine-tune our layout by changing the weights of each element, so as to see the effect in practice. After selecting a widget, *Ctrl+click*ing on it opens up the XML editor at the View's position. It's therefore easy to find. Try changing the **layout_weight** of some widgets. You can try setting the image's height to take three times more space than the other widgets by setting its weight to **3**, and set the weight of the space elements to **2**.

This way of setting the widgets sizes is not always optimal and we are loosing usability because of the cropped text. A compromise can be to clear all the weights except for the **Space** and the **ImageView** elements of the vertical **LinearLayout**. For all the remaining elements of the vertical **LinearLayout**, set the **layout_height** to **wrap_content**. That

way, the two spaces and the image will take all the remaining space. In this example, the remaining space is the total screen height minus the minimal height of the text inputs and the horizontal **`LinearLayout`** that contains the two buttons.

Two final adjustments involves the buttons and edit text widgets. You can re-set the ”text” inside the buttons the same way you edited the ID names of the widgets (either in the **Text** or **Design** tab). for the **`LoginButton`** you can write **”Login”** and for the **`RegisterButton`** you can write **”Register”**. For the **`TextViews`**, you can put a text that is a hint to what the user can write. You can do that by changing **`android:name`** into **`android:hint`** in the **`Username`** and inserting the latter in **`Password`**. The final resulting layout is shown in Fig. 4b.

Nesting layouts one in another, you can achieve really complex displays. It may be necessary in some cases to use a different kind of layout depending on your use case. Feel free to experiment using both the graphical editor (which is good for discoverability and feedback) and XML editor (especially for fine-tuning or removing some useless properties added while experimenting).

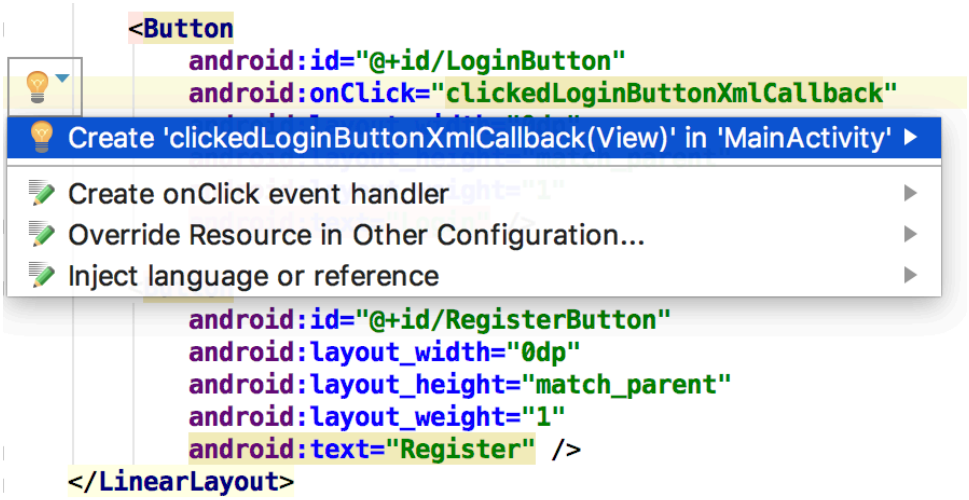
3 Reacting to taps on `Views`

There are two common ways to interact with a tap. We will use the two buttons to illustrate how both work, as both can be used depending on the situation.

You will implement functions that write some text in a new **`TextView`** (to create) between the **Password** field and the **Buttons**. Start by adding it as it was shown in the previous section. Make sure that the **`layout_width`** is set to **`match_parent`** and that **`layout_height`** is set to **`wrap_content`**. Name its ID **LoginMessage**, and leave the text empty for now.

3.1 XML callback

In the layout editor, whether it’s through the **Design** tab or the **Text** tab, it’s possible to set the function to call when the **`LoginButton`** is pressed: it’s named a ***callback***. The function will be called **`clickedLoginButtonXmlCallback`**. It should be defined in the **`onClick`** property, and Android Studio helps you not to forget to create the function itself in the Java code:



If you forget, the project will compile and run fine, until someone presses the button: the app will crash because it cannot find the called function! Let's add a very simple one for the first button. The callback function will find a reference to the **TextView** in the layout (please take care that the ID defined in the XML is the same one as in the Java code: **android:id="@+id/LoginMessage"**, and set it to a hard-coded string as follows:

```
public void clickedLoginButtonXmlCallback(View view) {
    TextView textView = findViewById(R.id.LoginMessage);
    textView.setText("We used the XML callback!");
}
```

3.2 Java callback

If your layout is more dynamic, it will require to bind some Views with their respective callback programatically. In this case, you should do that in the **onCreate(...)** method of the **Activity** after the layout inflation (before that, the button does not yet exist). Let's define the **onClick(...)** function of the second button, this function will set the **TextView** to a different hard-coded string:

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

    Button button = findViewById(R.id.RegisterButton);
    button.setOnClickListener(new View.OnClickListener() {
        @Override
```

```
        public void onClick(View view) {
            TextView textView = findViewById(R.id.LoginMessage);
            textView.setText("We used the Java callback!");
        }
    });
}
```

Testing is easy, and we now have most of the bases required for building more complex apps. But before that, we need to take care of Android Wear !

4 Building the Android Wear layout

For the Android Wear App, you will use a new type of layouts: the **constraint layout**. The constraint layout allows you to position your widgets relative to one another and to their parents in the horizontal and vertical axis. The general concept is to constrain a given side of a widget to another side of any other widget or the parent.

You will notice that Android Studio has automatically created a layout for you based on the template from the App Creation Wizard.

Delete the content of **wear/res/layout/activity_main.xml** and replace it by the following XML code to replace the previous layout with a **ConstraintLayout**:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.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:id="@+id/container"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@android:color/white"
    tools:deviceIds="wear">
</android.support.constraint.ConstraintLayout>
```

This defines a white container that takes the full available space in the screen. Next, you will add an **ImageView** and a **TextView** inside the white container. To do so, import the same **ic_logo.png** image as before in the Android Wear module.

In the **Design** tab of **activity_activity.xml** drag and drop an **ImageView** (choose the **ic_logo** as a source image). Then drag and drop a black **TextView** under the previously

added image.

We would like the image and the text to be centered in the layout, with the image at the top of the text. To achieve this, switch to the **Text** tab, and set the layout constraints as follows:

```
<ImageView
    android:id="@+id/ImageView"
    android:layout_width="50sp"
    android:layout_height="50sp"
    android:src="@mipmap/ic_logo"
    app:layout_constraintBottom_toTopOf="@id/textView"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
<TextView
    android:id="@+id/textView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="TextView"
    android:textColor="@color/black"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toBottomOf="@id/ImageView" />
```

layout_constraintLeft_toLeftOf and **layout_constraintRight_toRightOf** allow to define how the elements are positioned horizontally. In this case, you constrained them to be aligned to the parent container in the left and right side (i.e. centered).

We define how the elements are positioned vertically using similar definitions. In this case, you set the image and the text to be aligned vertically, in the center of the container:

- **layout_constraintBottom_toBottomOf**
- **layout_constraintTop_toTopOf**
- **layout_constraintBottom_toTopOf**
- **layout_constraintTop_toBottomOf**

Notice that you also set the width and height of the **ImageView** to **50sp**, this is because the image appears too big when the height and width are set to **wrap_content**.

One last thing remains to do, which is to assign a value to the **TextView**. You will do that in the **onCreate(...)** method of the **MainActivity**. You need to find a reference to the

TextView in the layout and then set it to the hard-coded string **"Hello Round World!"** (as it was done previously in the callback functions).

The resulting Android Wear layout should look like in Fig. 5.

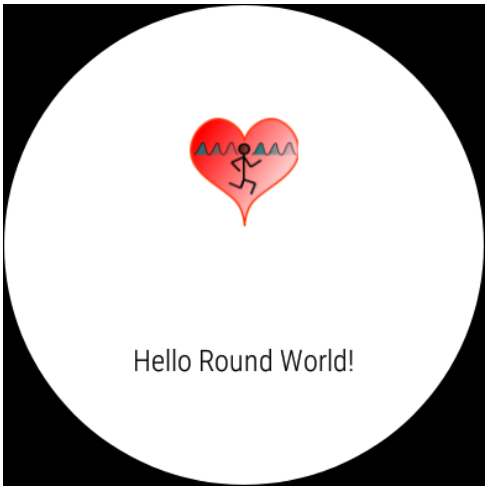


Figure 5: Android Wear Layout

4.1 Android Wear idle display

The last specific element we will see about Android Wear is the idle display. As some watches are using an OLED display, the energy consumption is coming exclusively from each pixel's brightness. That way, it is still possible to display meaningful information (such as the time) while most of the screen turns off. This is called the **Ambient mode**, which can also be used to keep the activity on the screen when it's going into low-power mode. We therefore need to update our Java code to use the ambient display:

```
public class MainActivity extends WearableActivity {

    private TextView mTextView;
    private ConstraintLayout mLayout;

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

        mTextView = (TextView) findViewById(R.id.textView);
        mTextView.setText("Hello Round World!");
```

```
        mLayout = findViewById(R.id.container);
        // Enables Always-on
        setAmbientEnabled();
    }

    @Override
    public void onEnterAmbient(Bundle ambientDetails) {
        super.onEnterAmbient(ambientDetails);
        updateDisplay();
    }
    @Override
    public void onExitAmbient() {
        super.onExitAmbient();
        updateDisplay();
    }

    private void updateDisplay() {
        if (isAmbient()) {
            mLayout.setBackgroundColor(getResources().getColor(
            android.R.color.black,
                    getTheme()));
        } else {
            mLayout.setBackgroundColor(getResources().getColor(
            android.R.color.white,
                    getTheme()));
        }
    }
}
```

The final step before having this feature enabled is to declare it in the **AndroidManifest.xml** file. Above the **<application>** tag, add the following declaration:

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

Now, run the app on the watch. After a timeout of around 15 seconds, the background goes from white to black. Internally, the CPU goes to a sleep mode, still waking up every minute. It's possible to use this wake-up by over-riding the **onUpdateAmbient()** method and update any **TextView** which might be displayed at the current time.

This is a trivial example saving just a little energy. With a more complex app, you can still

display meaningful information when the screen is idle. As a example, a runner can still see the current run time, pace and heart rate while in the idle mode. In the active mode, the app can display charts with the evolution of the measured data or a live map of the track.