Mobile and Ubiquitous Computing

Labsheet 3 – Layouts

In this labsheet, we'll be looking at Android's layout models. Last week, some of you might have run into problems with all of your elements ending up piled on top of each other. Or you might have found that elements did not end up where the visual building interface made it seem like they would. We're going to have a look at how you can finely control the layout of items on the screen.

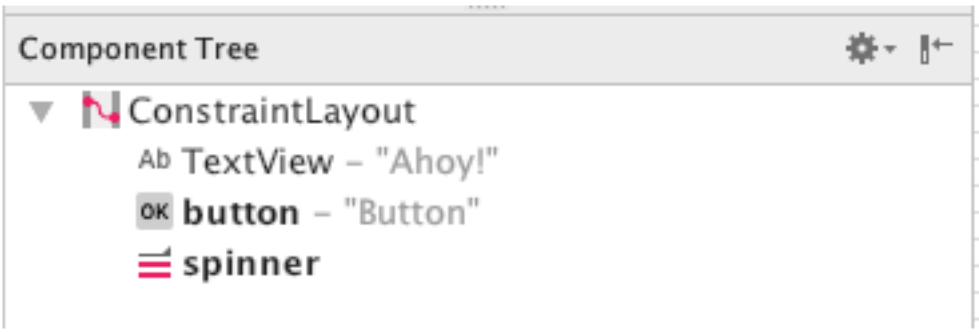
Layouts in Android

You should have encountered layouts previously in Java with JavaFX/Swing/AWT. Like all GUI widget toolkits, Android has a number of ways of specifying how widgets should be placed on the screen and how these widgets should behave as the screen expands, shrinks or rotates. There is extensive documentation on the different ways of managing layouts in Android. We don't have time to explore all of them here, but some of them should feel familiar from similar layout concepts in Java's GUI toolkits.

OK, let's go back to last week, where we had an activity that contained a Button, TextView and Spinner. First, rename your project to `week4lab` and refactor appropriately (alternatively, where you see `week4lab` in the code, replace it with `week3lab` or `week2_lab`, whatever you have at the moment.) If you created an Empty Activity then, Android will have created your activity using the *Constraint* layout. Look at the activity's text view and on the second line you will see:

```
<android.support.constraint.ConstraintLayout ...>
```

It's also visible in the design view:



LinearLayout

The advantage of using `ConstraintLayout` is that it allows for complex responsive UI designs without having a bewildering number of groupings for elements. But it's a bit tricky to get into, so let's go with something a little simpler to start, the `LinearLayout`. To do this, switch to the text view for your activity and replace `android.support.constraint.ConstraintLayout` with

`LinearLayout`. Now switch back to design view. You'll see that things have changed, and your three elements are now arranged in three columns.

In our case, it makes sense for our three elements to be arranged top to bottom, rather than left-to-right. In text view, edit the XML so that the definition includes `android:orientation="vertical"`. It should look something like this:

```xml
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:id="@+id/layout"
    tools:context="cs.bham.ac.uk.week4lab.MainActivity">
```

Currently, each of the *Views* you have in this XML activity have attributes that are specifically for constraint layouts. In this instance the attributes are not incompatible (changing layouts can often cause your app to stop compiling), but the existing attributes will stop our linear layout working properly. You will need to remove each attribute beginning `app:layout_constraint` from the *Spinner*, *TextView* and *Button* (e.g., `app:layout_constraintLeft_toLeftOf="parent"` etc). Your views should be defined like this:

```xml
<TextView
    android:id="@+id/textView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/hello_world" />

<Button
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:onClick="onClickUpdate"
    android:text="@string/updatebutton" />

<Spinner
    android:id="@+id/spinner"
    android:layout_width="276dp"
    android:layout_height="24dp"
    android:entries="@array/spinneroptions" />
```

Now, back in design view, you should see them in three neat rows. Try moving one of the elements. You will see that the new layout model determines how the elements can move. Build and run your app to see how it works. The *LinearLayout* has the advantage of being very simple, but it is quite limited. Layouts can be nested, so *LinearLayout* can sometimes be useful for structuring more complex layouts.

At this point, let's return to testing to check that our code is doing the things it is supposed to. You can add these tests to the existing `SpinnerSelectTest` class, but logically it makes sense to create a new class file `LayoutTest` to handle this. One way to do this is to duplicate and edit the class file we already have. Another is to get Android to create one for you using the Espresso recorder. This lets you build tests by interacting with your app. To do this, got to `Run → Record Espresso Test`. This will start up your app and you should be able to see that as you take actions it records them in the pop-up list. This is a great way of building, or at least getting a structure for, complex UI tests. You could just click the update button and leave it at that. It'll ask you what you want to call the new test class, call it `LayoutTest`. You'll see a bunch of boilerplate code, including a new implementation of `Matcher`. Have a look at the code, see the variety of things it's checking and some of the methods that will be new to you (e.g., `allOf()`).

Here is the test that you should check passes for your linear layout:

```
public void isLinearLayoutAndIsVertical() {
    onView(withId(R.id.layout)).check(matches(instanceOf(LinearLayout.class)));
    onView(withId(R.id.layout)).check(matches(hasVerticalOrientation()));
}

Matcher<View> hasVerticalOrientation(){
    return new BoundedMatcher<View,LinearLayout>(LinearLayout.class) {
        @Override
        public void describeTo(Description description) {
            description.appendText("LinearLayout has vertical orientation?");
        }

        @Override
        public boolean matchesSafely(LinearLayout lo) {
                return lo.getOrientation() == LinearLayout.VERTICAL;
        }
    };
}
```

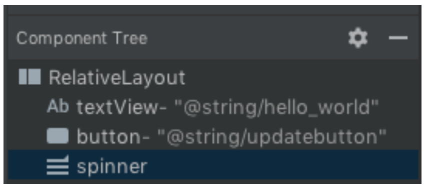
The `isLinearLayaoutAndIsVertical()`, you will not be surprised to learn, tests for two things; it tests whether the resource at `R.id.layout` is actually of type `LinearLayout` (that's the `instanceOf` check). Then it checks whether it `hasVerticalOrientation()`. There's no instance of `Matcher<View>` that can do this, so we add our own implementation. Our Matcher has to return a `BoundedMatcher` instance and this requires the implementation of two methods, but the important one is `matchesSafely`. This is the method that actually does the work of seeing if or `LinearLayout` does indeed have a `VERTICAL` orientation. Note that we use a `BoundedMatcher` rather than a `TypeSafeMatcher`, because this lets us specify that we are **only** interested in instances of `LinearLayout` (which implements `View`), and not any other `View` instances. This saves us a casting check in `matchesSafely`. It also makes the first check in `isLinearLayoutAndIsVertical` redundant because the vertical check only passes if a `LinearLayout` is supplied. This is all using Java's bounded wildcards to work: `matches()` expects to be passed an instance of something that implements `Matcher<? super View>`, i.e., a View or a class that implements `View`. The wildcard means we can be a bit more restrictive about the exact class, so long as it does implement `View`. In this `case, LinearLayout`. It's an interesting language feature that you might not have used before.

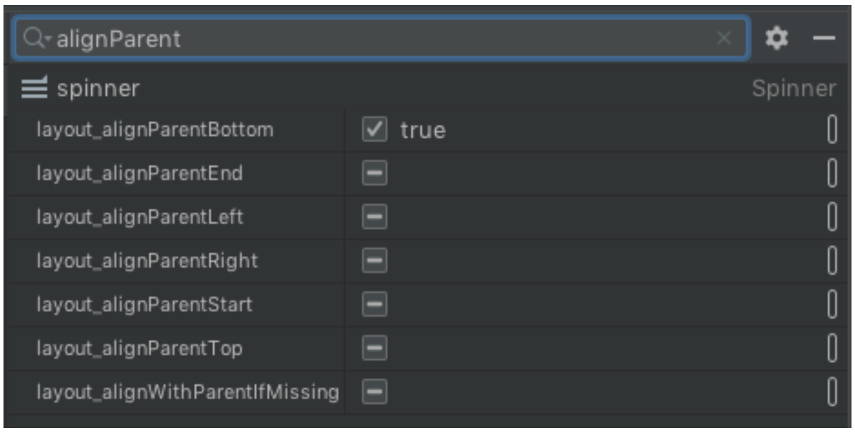
Generally, I won't be asking you to write tests, and I won't be asking you to write tests in your assessment. However! You still need to be able to read tests so that you can write code that will pass them, so it's important that you get used to reading them.

RelativeLayout

Another layout model is *RelativeLayout*. Repeat what you just did, but this time instead of having *LinearLayout* in your definition, change it to RelativeLayout. Switch back to design view. Try moving elements around. You can't – they keep snapping back. This is because in *RelativeLayout* the location of a given element (our three *Views* in this case) is set relative to other aspects of the activity. Currently we have not told any of the elements what they should be positioned relative to, so they just all sit bunched in a corner of the design view.

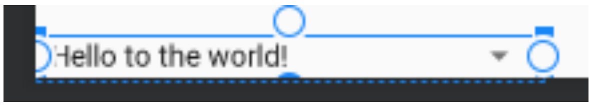
We want the *TextView* to sit at the top of the activity. To do this add the `android:layout_alignParentTop="true"` attribute to the *TextView* tag. We want the *Button* to sit in the middle of the screen, so add the `android:layout_centerInParent="true"` attribute to the *Button* tag. Finally, add the `android:layout_alignParentBottom="true"` attribute to the *Spinner* tag so that it appears at the bottom of the screen. You can also add these from the Design view by selecting each *View* from the Component Tree and then ticking the appropriate option by searching the attributes pane for the appropriate option to tick:


Component Tree
RelativeLayout
Ab textView- "@string/hello_world"
button- "@string/updatebutton"
spinner

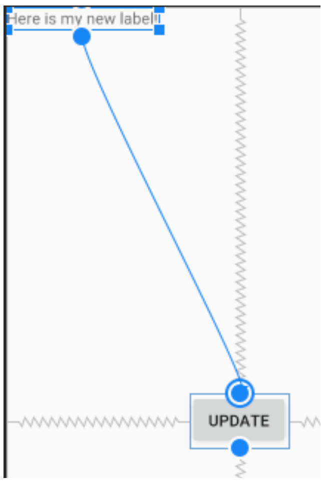

alignParent
spinner Spinner

layout_alignParentBottom	✓ true
layout_alignParentEnd	−
layout_alignParentLeft	−
layout_alignParentRight	−
layout_alignParentStart	−
layout_alignParentTop	−
layout_alignWithParentIfMissing	−

Now you can position elements relative to the whole activity and relative to one another. Note that in Design view, if you select one of the components 'handles' will appear on it:



You can click and drag from the handles of one component to the handles of another component (have a look at the image on the next page). This will set the position of components relative to one another. How do you think this way of laying things out might work with increasing numbers of elements, or when widgets are, for instance, being dynamically added and removed? Have a play around with moving the components around and creating different relative distances between the components.

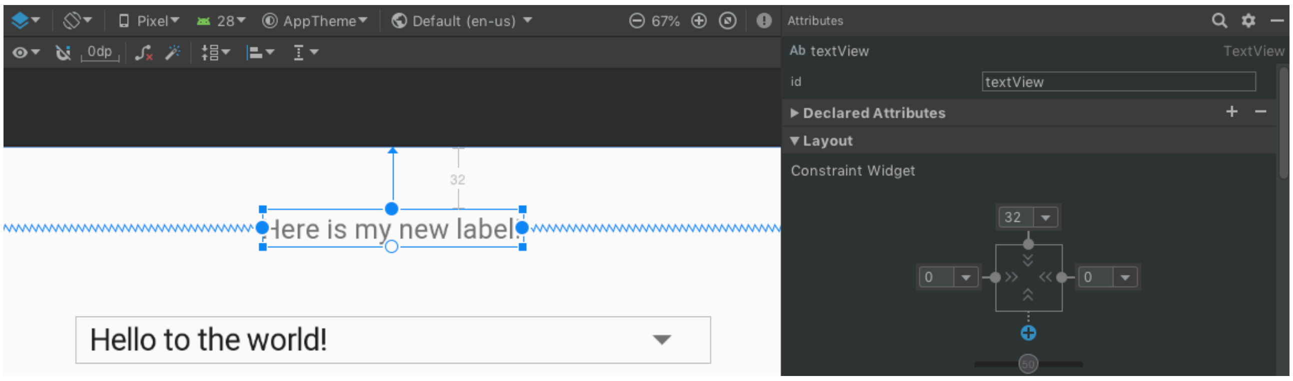


ConstraintLayout

When you're done playing with that, head back into text view and change it back to `ConstraintLayout`. To do this, remove the layout attributes you added for each of the components in the *RelativeLayout (*e.g., remove `android:layout_centerInParent="true"` from the *Button etc etc*) that you added on the previous pages of this labsheet. Then, replace the `RelativeLayout` tag with `androidx.constraintlayout.widget.ConstraintLayout`. (See the end of the labsheet for info on what this `androidx` thing is.) Now we're back to a *ConstraintLayout*. It's probably the model that's most worth persevering with; Google recommends *ConstraintLayout* because it provides *"better tooling performance and tooling support".*

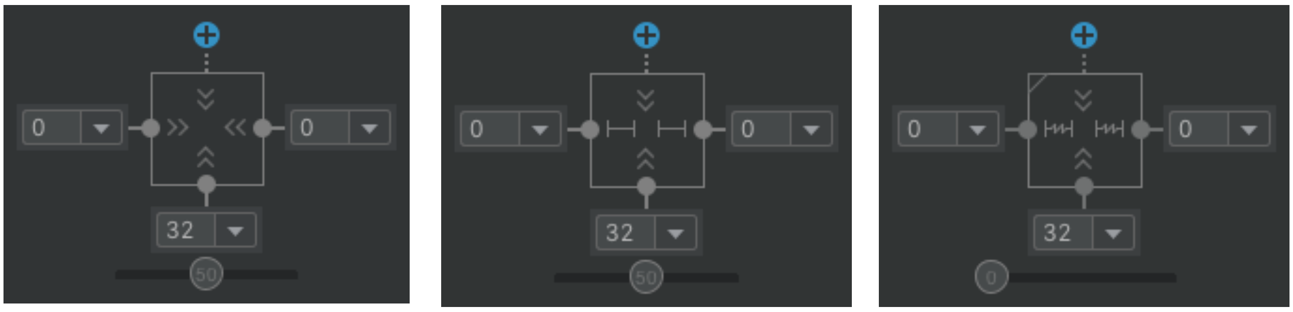
In design view, click on a widget. You'll notice that there are again a variety of handles on each side of the widgets (in this case, three are connected to other widgets). They operate in a similar manner to those for *RelativeLayout*, but they afford more complex set-ups. Have a go at moving around the components and creating constraints between elements and between the element and the activity. You **must** specify one horizontal and one vertical constraint for each component. After that, Android will take care of positioning things. *ConstraintLayout* effectively lets you model other forms of layout. For example, setting strict integer constraints on all sides of all components would yield something like the now-deprecated *AbsoluteLayout*.

Let's see what *ConstraintLayout* can do! First, select your *TextView*. Click and drag from the left-hand handle of the widget to the edge of the activity view. This should generate a zig-zag line line connecting the widget with the edge of the activity. This is an *anchor.* Now do the same for the right hand side. Now there are constraints one *both sides* and the horizontal bias is neutral (look at the Constraint Widget on in the attributes pane), this means that the label will float in the middle of the screen. Create another anchor from the top of the label to the top of the activity. In the Constraint Widget, set the *margin* of top constraint to 32 (See the figure!). This means the widget will track the top edge of the activity with a gap of 32 'density independent pixels'. These pixels avoid the problem of, say, maintaining the visual size of the margin on different displays. (On a cheap low density display, 32 normal pixels could look quite big but on an expensive high resolution display it could look very small.)

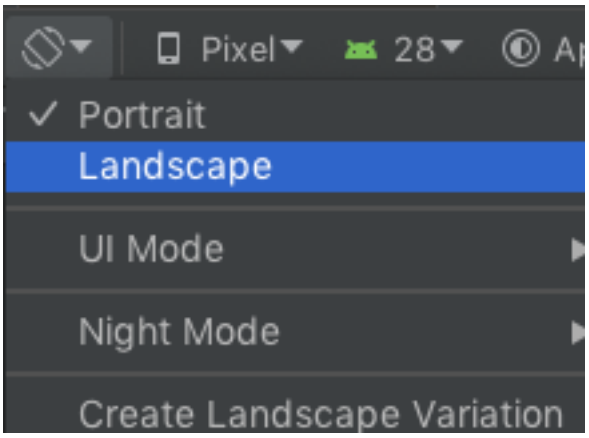


Now, click your *Spinner*. We're going to draw exactly the same constraints for this one as for the *TextView,* except for the top constraint, rather than dragging it to the top of the activity, attach it to the bottom of the TextView. Set the margin for this top constraint to 32 as well.

To finish with this layout we will do something a little different with the button. First, create left and right constraints to the very edge of the activity. Then create a bottom constraint to the bottom of the activity with a 32dp margin.



So far, so normal. But take another look at that Constraint Widget note that there are symbols *inside* the bounding box. These control how the view is resized. The chevrons/arrows, tell Android to wrap the content. So the button will grow in all directions to match the size of the text inside it. If you click these chevrons they will turn into H-shaped symbols. This is for a fixed size. So you could fix a button as being 80dp wide no matter what text is in it. Click it again and you'll see a mini-sized version of those zig-zag anchors. This is the 'match constraint' mode. It means the space will be filled up to the point of the constraint anchor. You should see that this means the button now completely fills the width of the screen. The nice thing about the *ConstraintLayout* is it dynamically reacts to the screen.



If you change the rotation of the Design view from the tool bar, you'll see that in horizontal view the button still fills the width of the screen. You have now completed a constraint-based layout. Have a play around with the handles and anchors to see what kinds of reactive layouts you can build in this way.

A constraint layout can help you produce really high-quality layouts. After this lab, I recommend working through Google's own documentation on using constraint layouts.

Finally, let's try and run a couple of tests to make sure that things are working as they should be. First, just a simple one to make sure that the three views actually appear on the screen and that they are 'stacked' in the right order: textView, then spinner below and then the button below that. Your code should pass this:

```
@Test
public void stackingOrder() {
    onView(withId(R.id.textView)).check(matches(isDisplayed()));
    onView(withId(R.id.spinner)).check(matches(isDisplayed()));
    onView(withId(R.id.button)).check(matches(isDisplayed()));
    onView(withId(R.id.spinner)).check(isCompletelyAbove(withId(R.id.button)));
   onView(withId(R.id.textView)).check(isCompletelyAbove(withId(R.id.spinner)));
}
```

That's easy enough. How about a test to make sure that our constraints have been set correctly? If you've followed the instructions correctly, this test of the constraints set on `textView` should pass:

```
@Test
public void textViewConstraints() {
onView(withId(R.id.textView)).check(matches(hasCorrectTextViewConstraints()));
}

Matcher<View> hasCorrectTextViewConstraints(){
    return new BoundedMatcher<View, TextView>(TextView.class) {
        @Override
        public void describeTo(Description description) {
            description.appendText("Check if this TextView has the correct
constraints set.");
        }

        @Override
        public boolean matchesSafely(TextView tv) {
            ConstraintLayout.LayoutParams params =
(ConstraintLayout.LayoutParams) tv.getLayoutParams();
            DisplayMetrics displayMetrics =
tv.getContext().getResources().getDisplayMetrics();
            int dpSize = (int) ((params.topMargin/displayMetrics.density)+0.5);
            return dpSize == 32;
        }
    };
}
```

What's happening here then? Once again we're not checking for something that Android will give us for 'free' (like `isDisplayed`) so we write our own matcher. This time, we just want to allow `TextView` instances so we again use a `BoundedMatcher`. The `matchesSafely` method collects the layout parameters for the supplied `TextView`, and then converts this to display points. Annoyingly, although our design is set up in display points, when we get the layout parameters from the app, they are back into good old pixels. We need to use some utility methods to find out what the conversion ratio is. That top margin should be 32dp and the final line checks this is the case. If your code isn't passing, check you've set this margin correctly.

What is Android X?

You might see in some places libraries prepended with `androidx`. What is this? When we're doing Android development we make frequent use of Android support libraries. These libraries allow us to write code targeting modern versions of Android in a way that retains backward compatibility with older versions of Android. This means you can make use new features of Android without breaking compatibility with older devices still in use. The original support library was just called 'Android Support Library'. This original library is no longer maintained.

AndroidX is Google's effort to solve some of the issues associated with the old Support Library. AndroidX brings new features, but it is also separately maintained and updated from the Android platform itself. This means that updates can be made more frequently, meaning people can make use of new features and fixes more rapidly. Old code using the `android.support` namespace is deprecated: it will continue to function for now, but all new development should make use of the `androidx.*` namespace instead.

You can read more about AndroidX in Google's AndroidX documentation.