I. Define Screens with Layout Files

An Android application is a loosely bound collection of activities, services, and other components

Android applications are packaged in an .apk file which is an archive of all components in the application. A manifest lists and configures the key application components. Android OS launches a Linux process for each application. The Android OS assigns each application a unique Linux user ID and gives privileges to each user. It launches each application in a separate Java VM. This provides a “least privilege” environment which maximizes security. Applications are normally started by the user launching an activity.

An activity is a component with which the user can interact. It is associated with a screen displaying the user interface. The contents of the screen are defined by a view which is normally defined in an XML resource file.

Applications normally consist of multiple activities, one for each “screen” in the application. Activities launch other activities by sending an intent to the application framework. When a new activity is started, it is placed on the

back stack and the previous activity is stopped. The previous activity may be reactivated when the user presses the Back button. When Back button is pressed the current top activity is removed and destroyed.

Once created, activities are in one of three main states

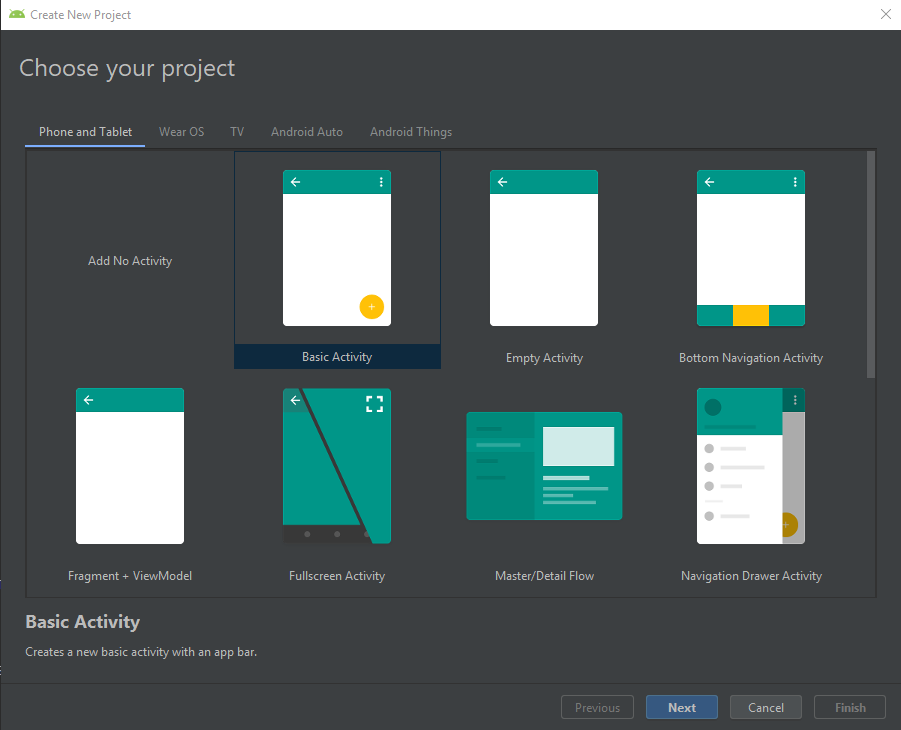
* Resumed : the activity is visible and running,
* Paused : the activity is partially obscured by another; the activity is alive but may be killed if the OS is very short on resources, and
* Stopped : the activity is now in the background—completely obscured and its state is maintained (all member variables, connections, etc.). A stopped activity can be killed if the OS needs the memory.

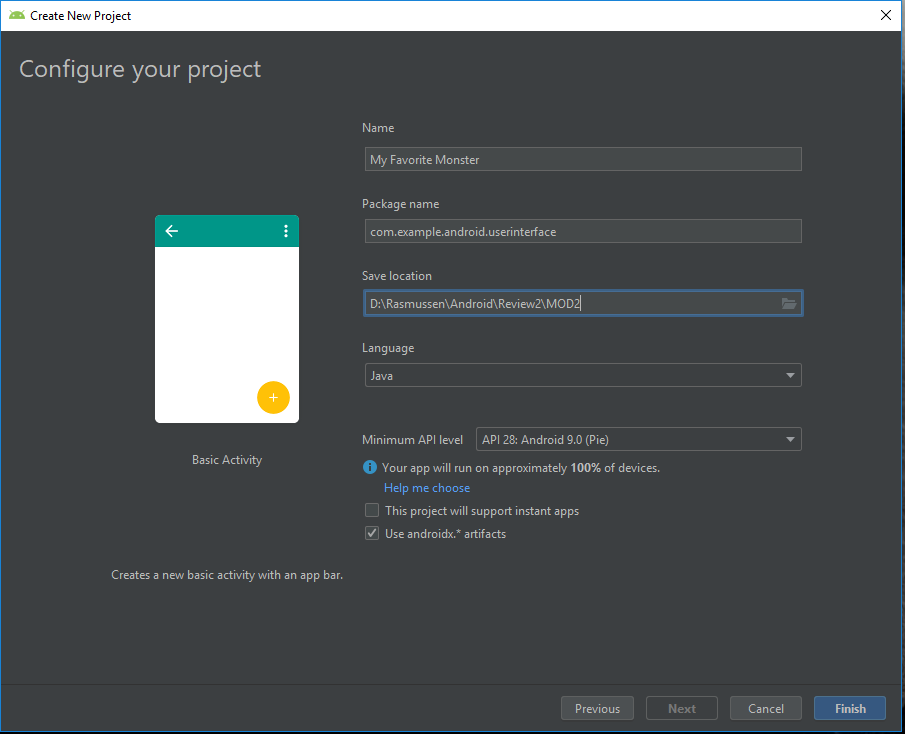
An application framework provides life cycle events which notifies an activity as its state changes.

When a user opens your Android app, their first experience is visual, a combination of screen layout, color palette, and other design elements is an absolutely critical part of how the user experiences your application. Android developers have a vast set of tools that let them build that interface. Since an app can run on a wide variety of devices with varying screen sizes, aspect ratios, and pixel densities, you need to design your app so it can adjust automatically to any screen it encounters.

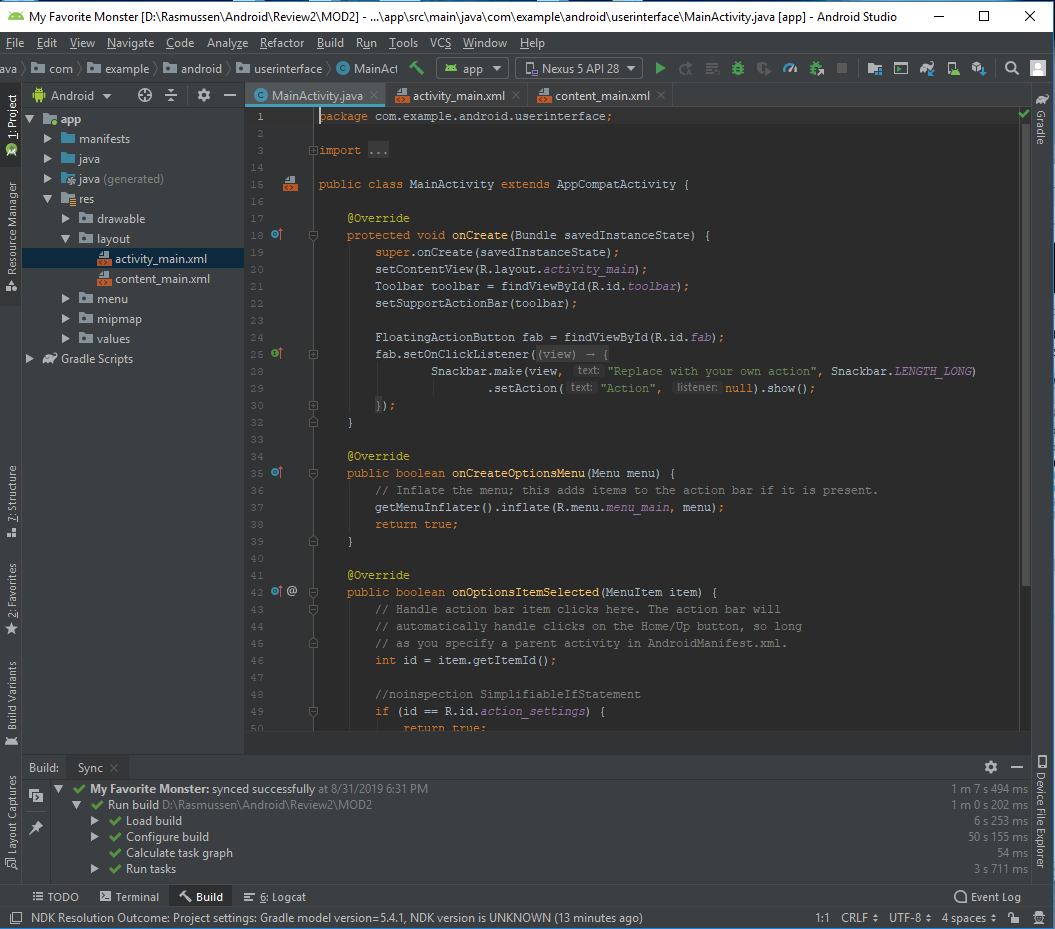
1. Load an XML layout file at runtime

This course is all about creating an Android app's visual appearance and behavior. The first place to start is with XML layout files which determine the initial appearance of your app's screens. Nearly every Android app uses an XML layout file to define the appearance of a screen. To get a sense of how this works, start a new Android project using the Empty Activity template. That template creates a single XML layout file with a small text view component that displays some read-only text. A step up in complexity would be to use the Basic Activity template. It has a two-level layout and it has a single material designed element called a floating action button. On this screen, give your app a name, say, “My Favorite Monster.”

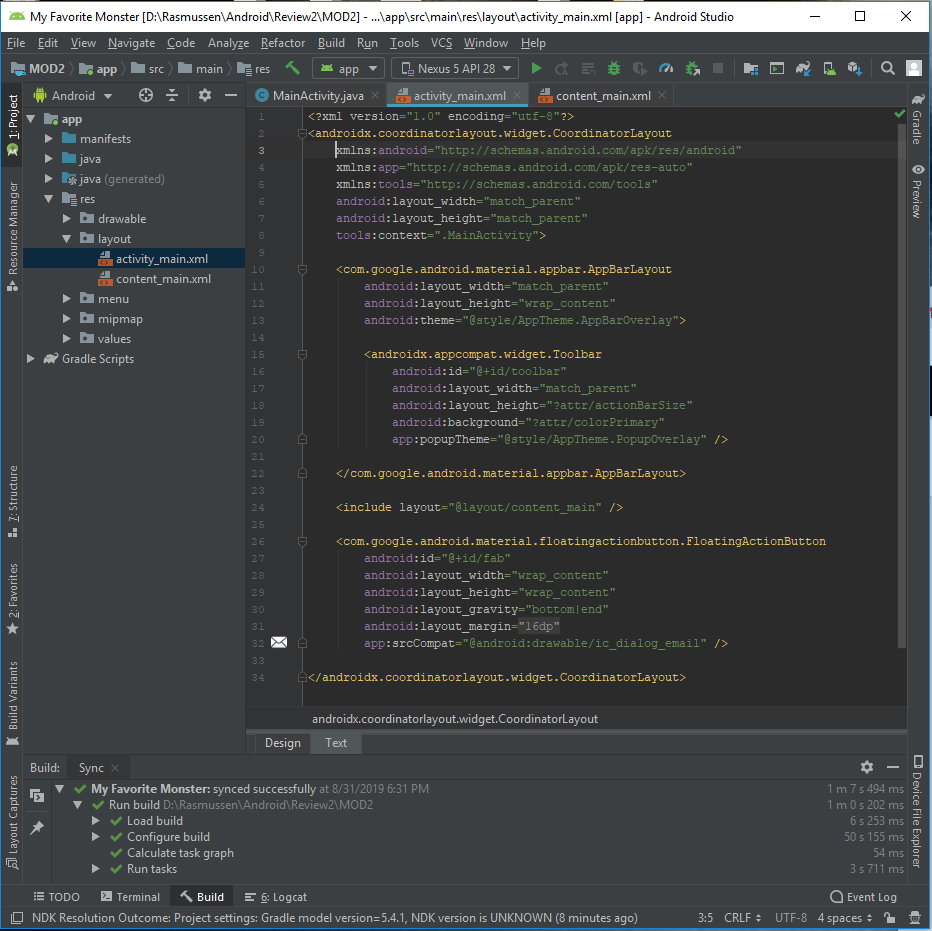




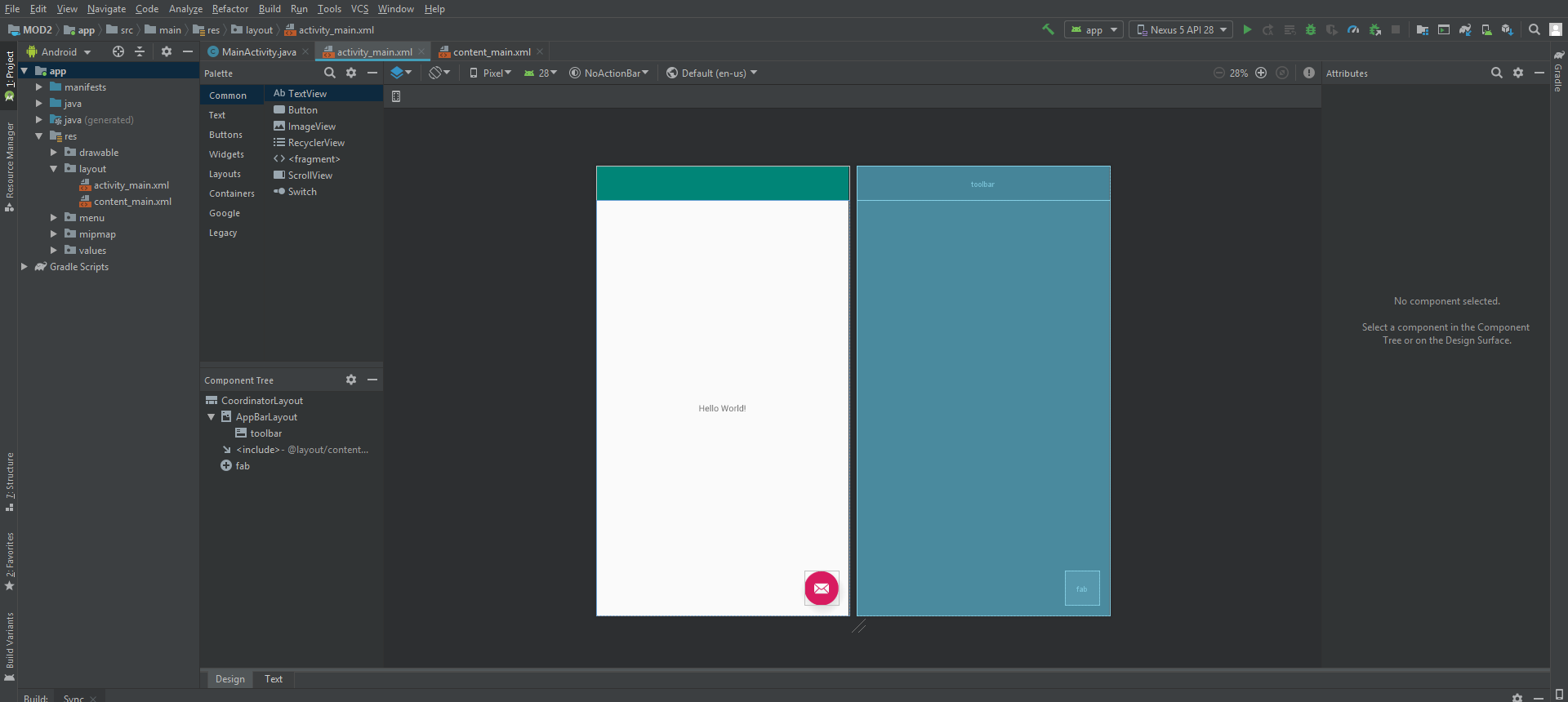
Set the package name to com.example.android.userinterface. When the project opens, it'll have a couple of different files open. The MainActivity class is the “start up” class.



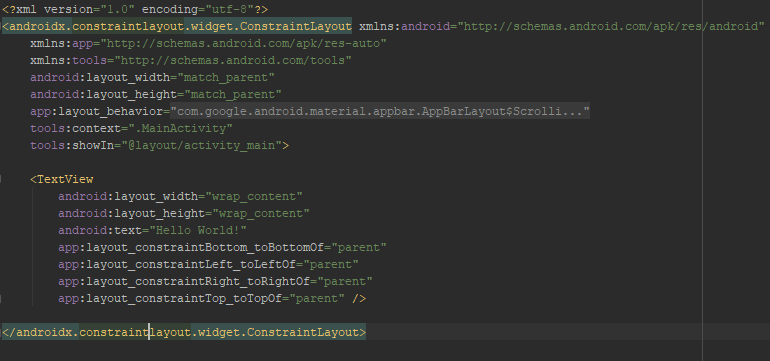
When the app starts, the onCreate function is called automatically by the application framework. The “setContentView” function references the activity main layout file. Now, notice in the layout directory there are two files, one of which you’ll modify which is “activity\_main.xml”



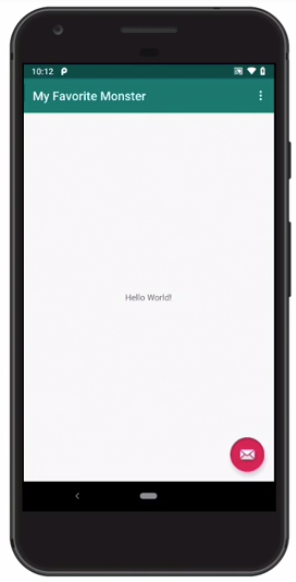
The root element is something called “CoordinatorLayout,” which is one kind of view group component that contains other views. Within the CoordinatorLayout, there's an “AppBarLayout” that contains a toolbar. At the bottom, there's a floating action button view component. If you look at this in design view, you see the button down in the lower right corner.



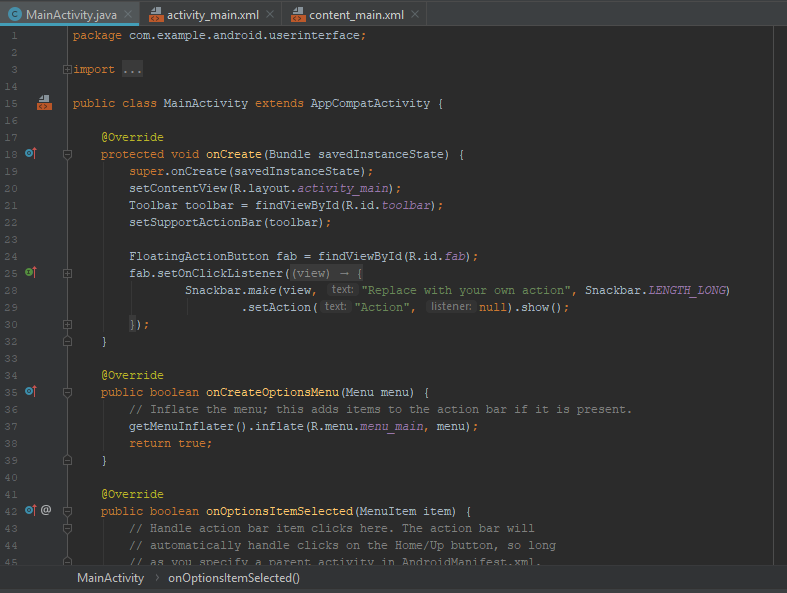
Notice that the “includes” declaration in activity\_main.xml references another layout file, called “content\_main.” It's displayed in the figure, but you can also jump to it by holding down control on Windows, or command on Mac, and clicking that link. Within the content\_main file, there's a root element called ConstraintLayout, and then within that, the text view.



When you run the app, you will see Hello World in the middle of the screen, and the floating action button at the bottom.



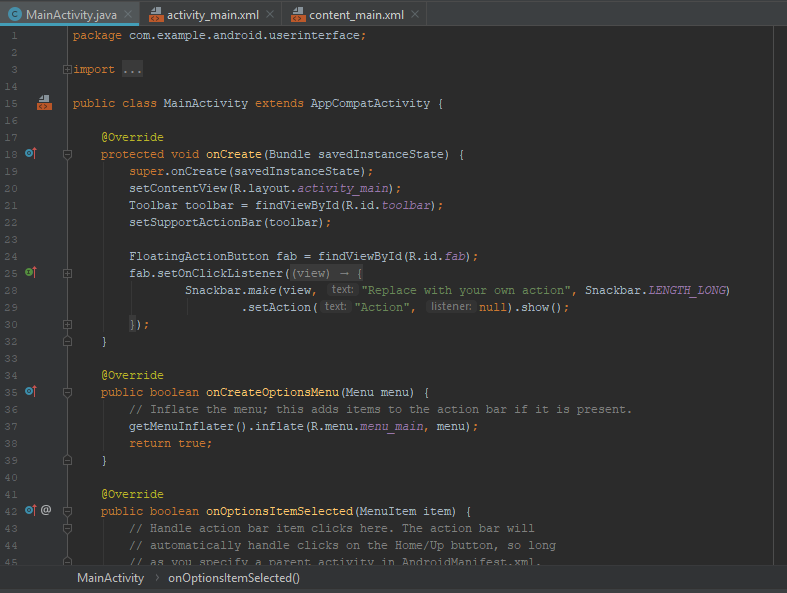
If you touch the floating action button, you will see the message “Replace with your own action.” So, that's the basics of how XML layout files are loaded at run time. The activity is brought to the screen, either because it's the launcher activity, or because it's called by some logic in the app. In the onCreate function, setContentView determines which layout file will be loaded.



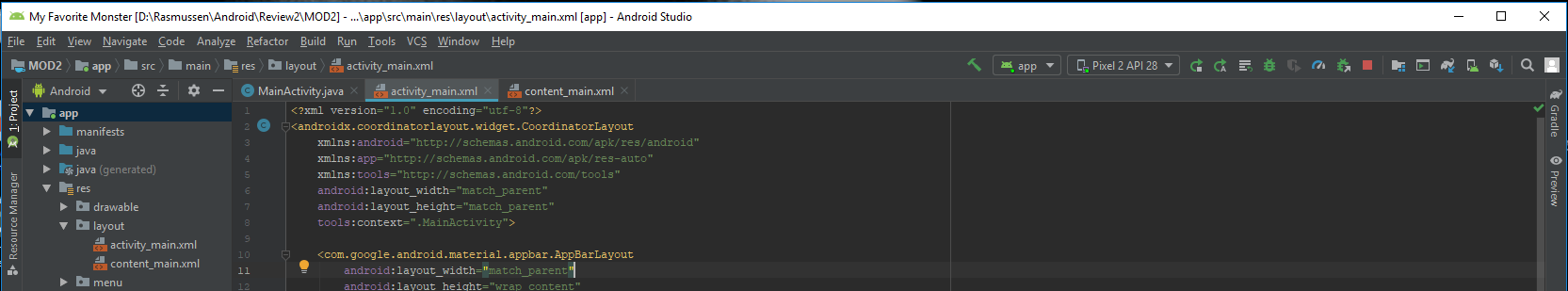
In this app, the primary layout file references a nested layout file, and as you customize your screen, this is where you'll make your changes.

2. View and ViewGroup Components

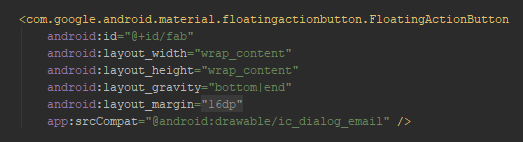
Every XML layout file contains a set of components that are interpreted as instructions for placing visual objects on the screen. These are XML files, but they're interpreted and executed in the background as Java Code. Most of the XML elements describe instances of classes. It's important to know which classes they are so you can look up their capabilities in the API documentation.



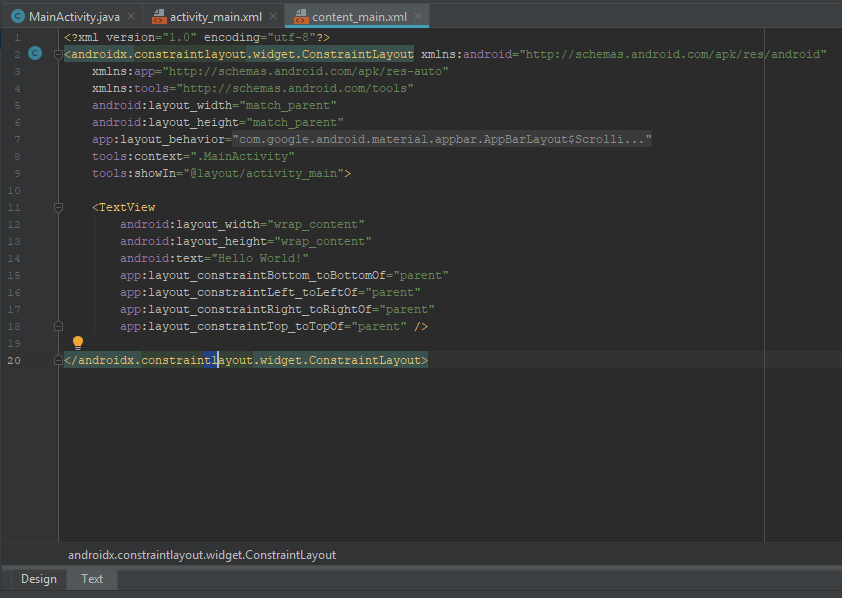
In this XML layout file, notice that some of these declarations have complete qualified class and package names. That's because these components aren't apart of the core android SDK; instead, they're delivered in a separate library called the Material Design Library. When you create a new project, this package might start with "andriod.support" or it might start with "androidx", depending on how you create the project. They both refer to the same components. The "androidx" packages are newer. In your primary XML layout file, the coordinator layout is the root element. It manages all the visual components in this screen. The app bar layout contains the toolbar, and the toolbar is where you see the icons and menus at the top of the screen.



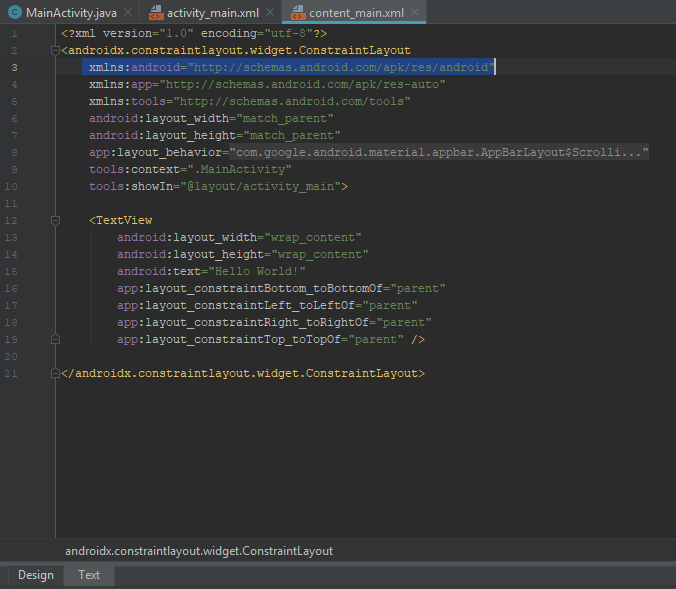
The floating action button is placed at the bottom of the screen and that's because of the layout gravity setting of the bottom and the end.



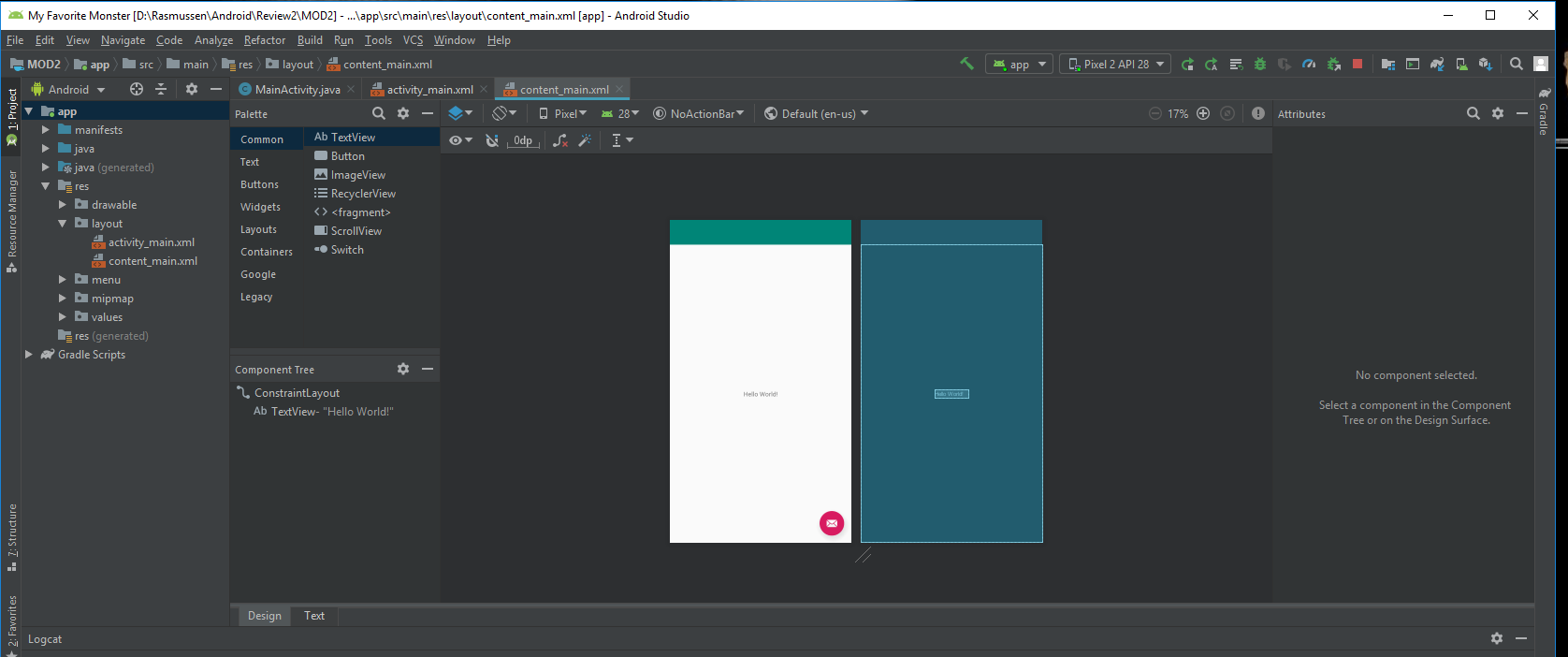
Within a coordinator layout, you use layout gravity. Within a constraint layout, you use constraints. This text view component has constraints to the bottom, top, left and right of the parent.



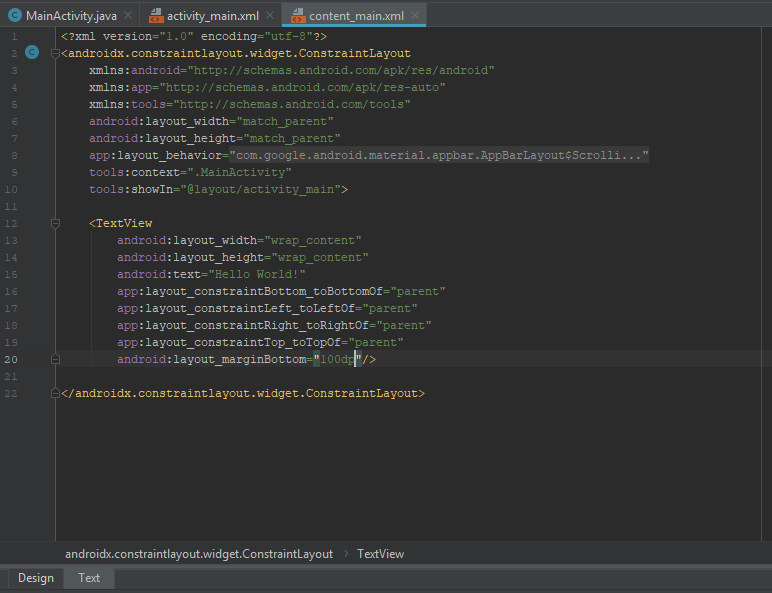
That's why it's placed directly in the center of the screen. Notice that the text view component is declared without a fully qualified package and that's because the text view is a part of the core Android SDK, rather than an external library. Also notice that in each of these layout files, there are some name space declarations. The first one is required for all XML layout files.



The Android prefix is then used for attributes that are a part of the core android SDK. The app name space is only required if you're using components from external libraries. The app prefix is used for attributes that aren't part of the core android SDK. The tools name space is used for attributes that only effect what you see in Android Studio, but don't effect what happens on the actual device when you run the app. For example, the tools “showIn” attribute, says that this layout file should be shown in design view in the context of activity main. When you go into design view, you see the floating action button which is actually declared in the outer layout file.



In addition to key words like "parent," you can also use certain dimensions. For example, let's say you wanted the text view component to be a little bit closer to the top of the screen. You could add a margin bottom setting. Start typing "margin" and then use intellisense to choose "layout bottom margin." Set that value at 100 DP, which is 100 device independent pixels.

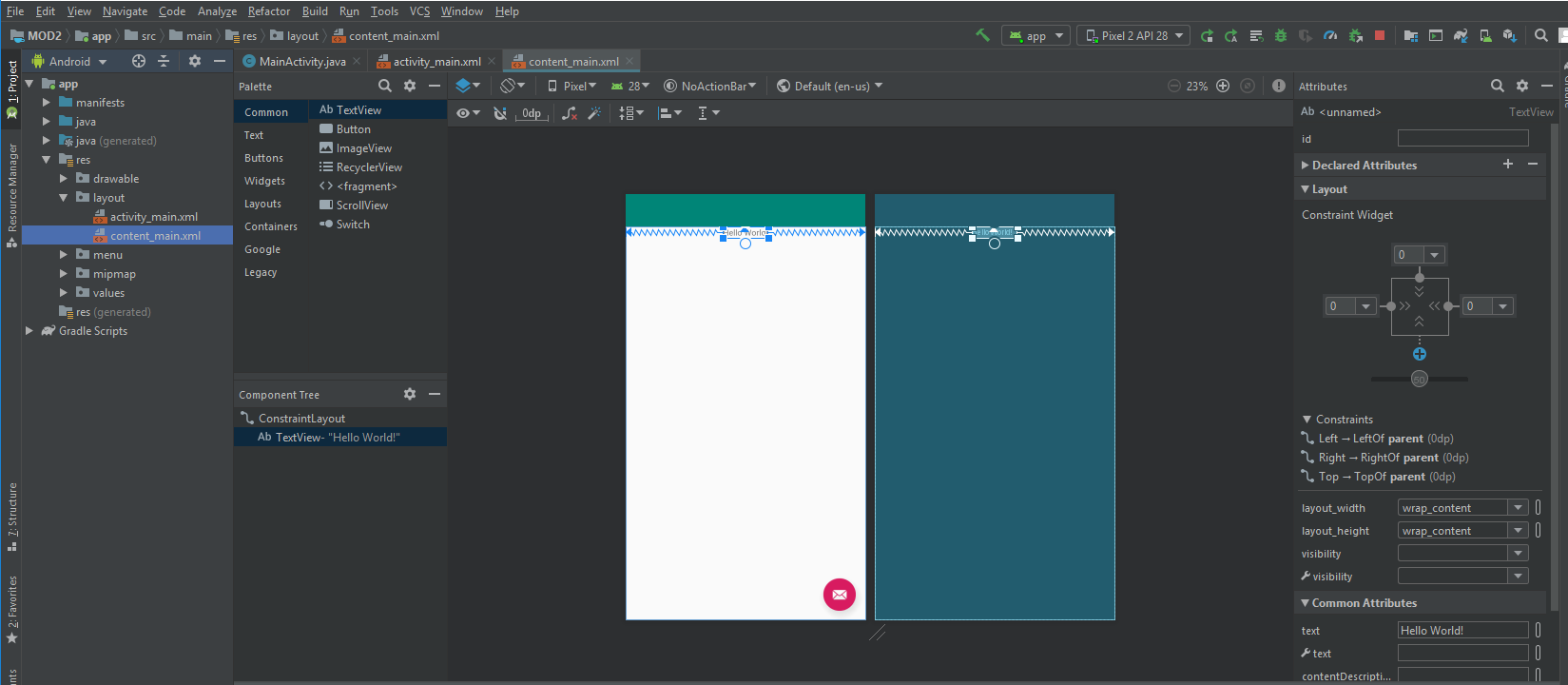


When you look at the text view, you’ll see it's not quite vertically centered as it's a little bit higher.

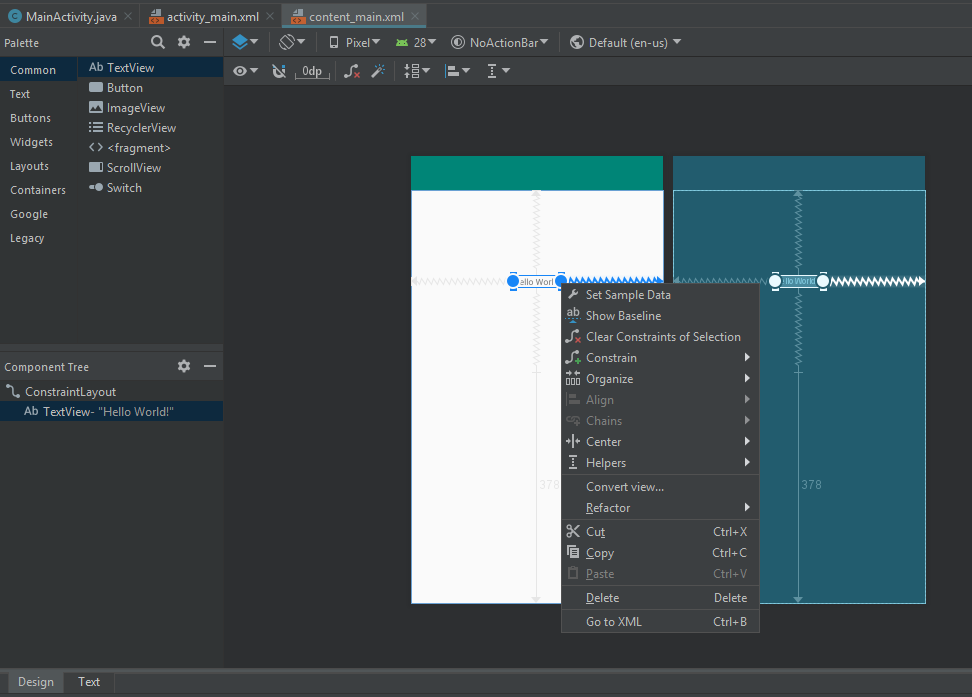
Alternatively, you can completely remove certain constraints. In design view, click on the text view and then you’ll click the icon at the bottom and that deletes the bottom constraint. When you do this, the text view floats to the top of the screen. The use of the constraint layout container is a critical part of how you design the visual appearance of screens in modern Android apps. As you build your own Android apps, you can design your own activities and screens to make them look exactly the way you need them.

3. Display View Components in a ViewGroup

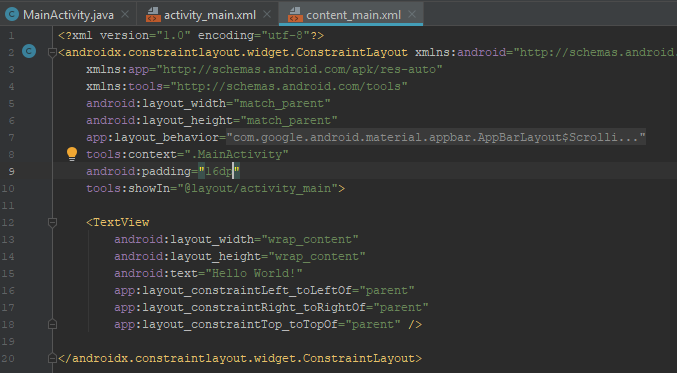
Each visual widget on the screen is a view, and it's implemented either in the core Android SDK or in an external library. You can tell the difference between these by the presence or absence of a package declaration. The presence of a package means that the component is defined in an external library, and the absence means it's a part of the core SDK. To implement a simple screen layout start with the content.main.xml file in design view.



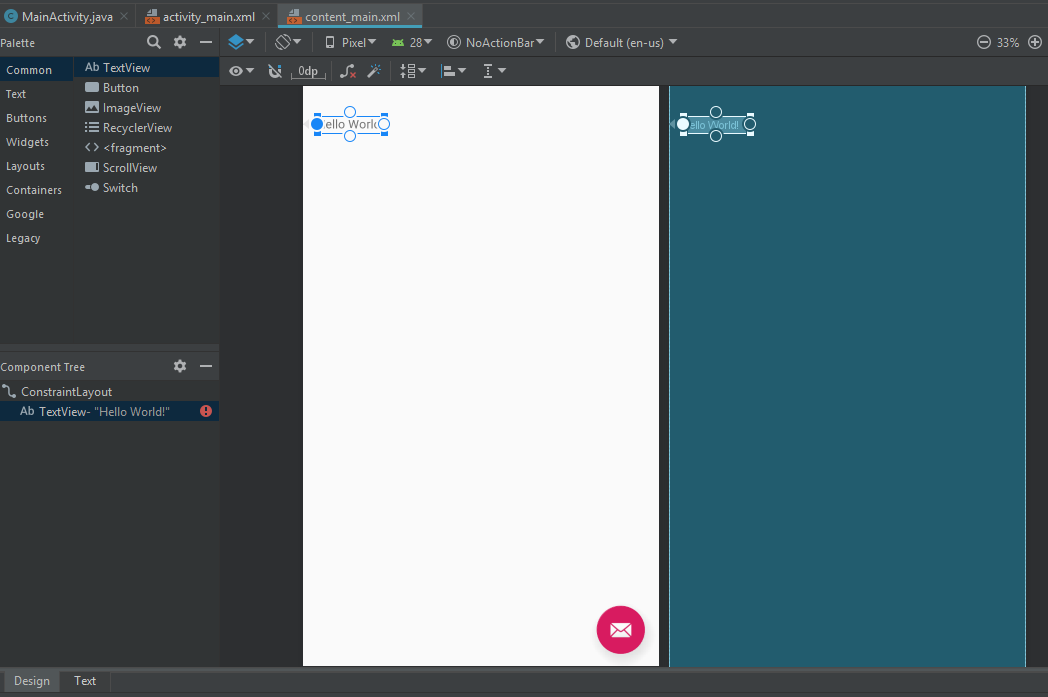
Select the label and zoom in to make this a little bit easier to work with. Remove the constraint to the right edge of the screen.



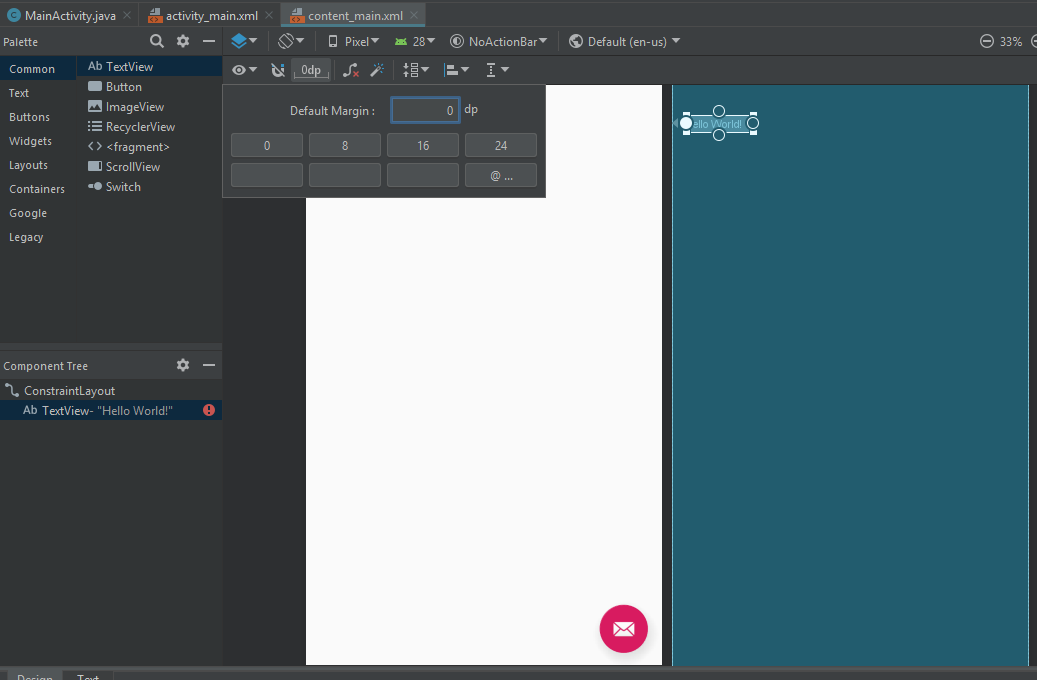
Now you’ll want to create a little bit of space between the top and the left edges of the screen and that label. Go into text view, and in the constraint layout, add a padding setting.



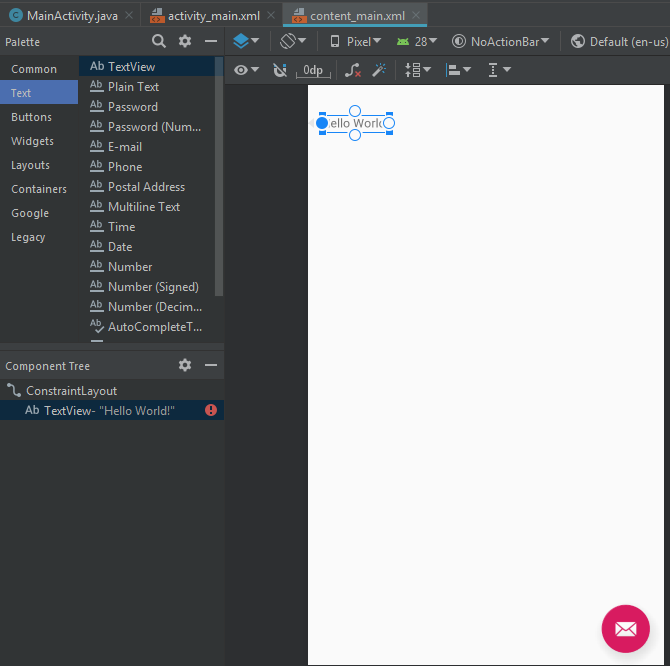
Set that to 16dp, or device independent pixels. Typically, you'll want to work in multiples of eight. There's something called the material design specification or guidelines that can help you standardize how your application works. In reality though, you can design it however you like, but that padding setting will control the padding around all edges of the screen. When you go into design view, the label floats down and to the right.



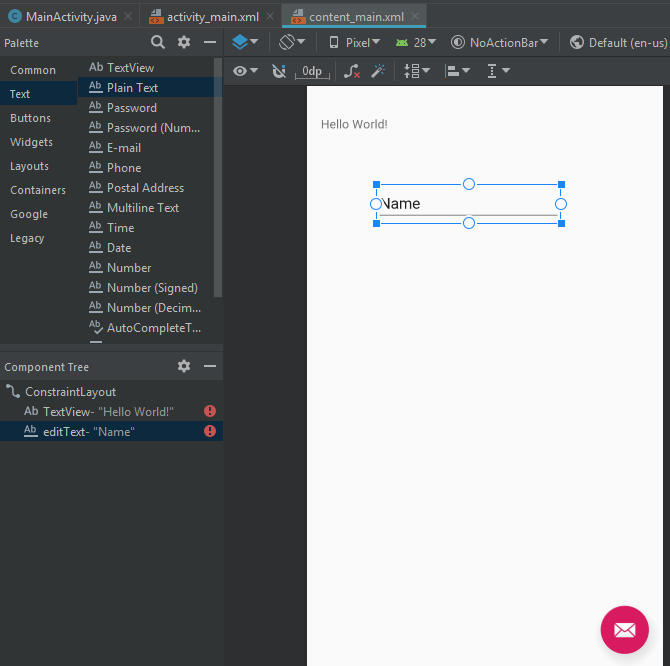
Next you’re going to add a view component that can accept text from the user. Before you add it, make a change to the automatic margins that are set whenever you add a new component. Change it to zero.



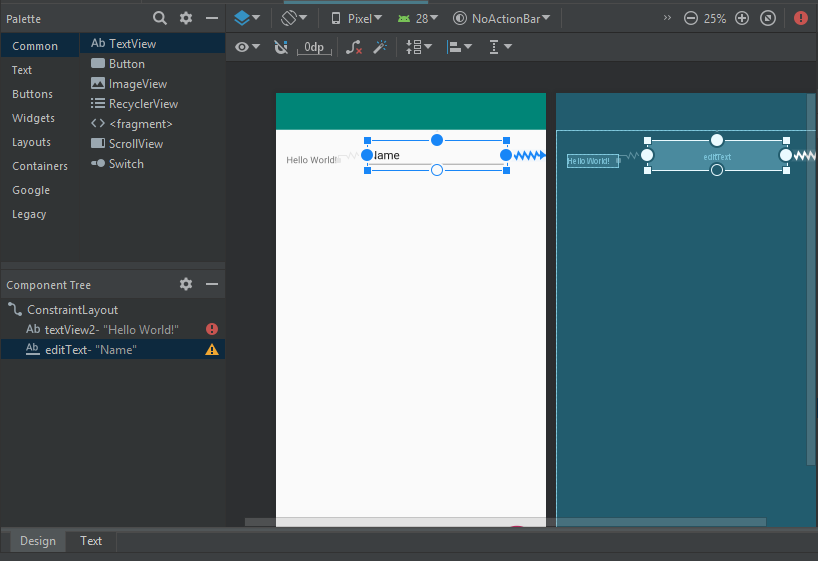
Next, go to the Palette to the Text category.



Choose the element that accepts plain text from the user. A TextView is read-only text. A Plain Text component, which is really called an edit text component, will let the user type something in. Drag that onto the screen.



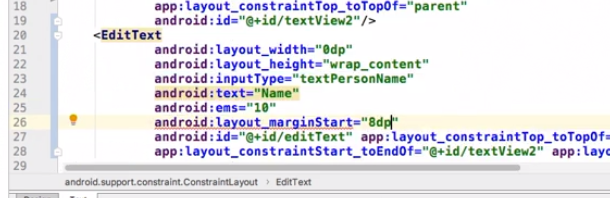
Constrain the top of that component to the top of the screen, and constrain the left edge to the right edge of the label. In addition, constrain this to the right side of the screen. That component will now float between the label and the right edge.



If, instead, you want to stretch it, here's a little trick. Go to Text mode and change the width of the component from wrap-content to 0dp.



This means that this component should stretch to fill the available space. Next, go back to Text mode where you’ll add in something called marginStart. Set that to 8dp.

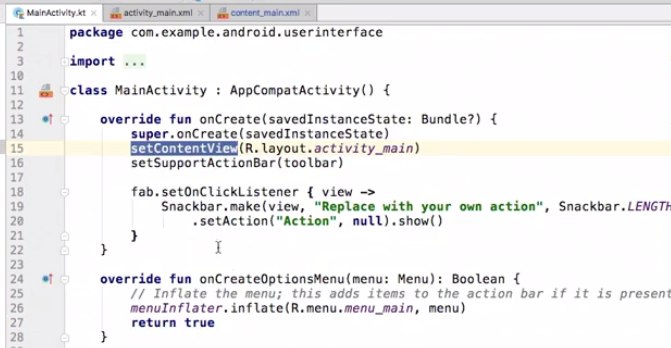


If you get a squiggly line then you can use an intention action. Place the cursor in the error indicator, and the press option and return on Mac or alt and enter on Windows and add the marginLeft setting. That got added at the bottom of the component.

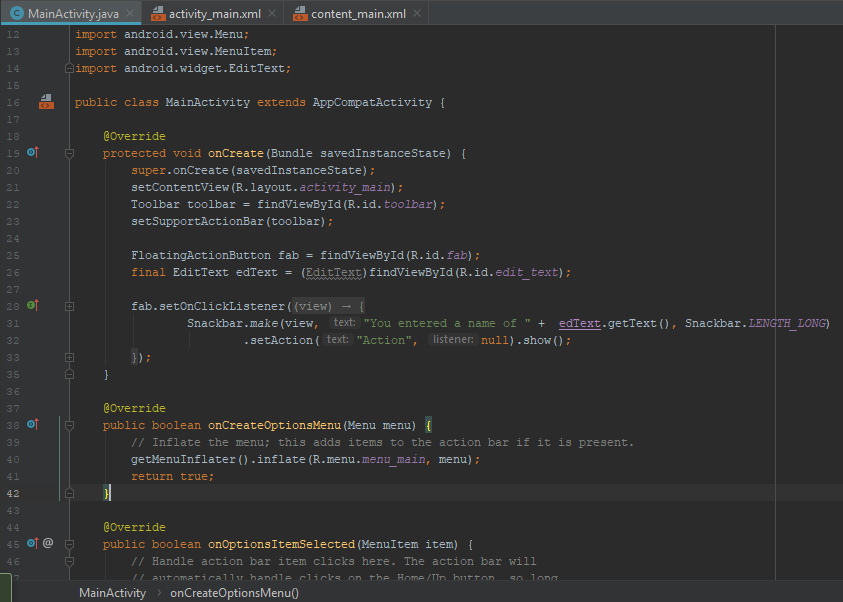


Take another look, and now you have bit of space between the label and the text component.

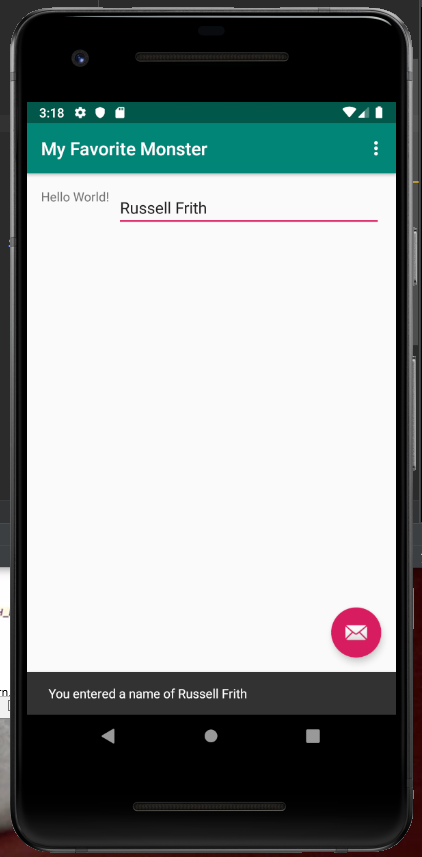
Finally, you need to be able to reference these objects at runtime. To do that, you need to add an ID. Notice, when you added the component, an ID was added automatically. Each view component that has an ID starts with the expression “@+id/” and that means add a new ID. The ID itself has to be all text, you can't use any special characters or spaces and you can either use camel case, or some developers prefer using underscores. To reference that component at runtime, go to MainActivity.java.



Notice there's already some code to display a message. Change that message to say "You entered a name of". You’re going to reference that object, which you named “edit\_text.” You will access the control using the Java resource class “R” as shown in the code below. More information will be provided later on this important class. Notice that a variable named “edText” was declared in order to reference the “edit\_text.” Notice also that it had to be declared “final” as it was used in an inner class.



This will retrieve the text that the user entered and you can now use it in your code. Run the app and see how it behaves. Touch that edit text component, type your name, and then press the floating action button. You should see the message, “You entered a name of \_\_\_.” If you want to change this to make it a little bit more attractive, go back to the layout file and change this text view's text from “Hello World” to “First Name.” So both the text view and the edit text components are view components that are a part of the core Android SDK. You can learn about other view components in the core SDK by looking at the documentation for the SDK's android.widget package.



4. Units of Measurement

The Android framework supports a specific set of units of measurement to control the placement and size of visual objects. These are known as dimensions and dimension values. The first and most important unit of measurement to discuss is called device independent pixels, abbreviated as “dp.” You typically use dps to set the positioning and sizing of containers and widgets. Device independent pixels are calculated based on a theoretical screen that has 160 pixels per square inch (ppi.) If you say an object is 100 dps wide and then you run that app on a screen with 160 pixels per square inch, it'll be exactly 100 pixels. But if you run that same app on a device whose screen has a greater pixel density, let's say one that has 320 pixels per square inch, then the device independent pixel measurement will be adjusted automatically for you. The ratio of device independent pixels to physical pixels changes with the screen density. There's another similar unit of measurement called scale independent pixels, and the abbreviation for that is “sp.” The calculation is very similar, but you use sp for font sizes and dp for object size and positioning. Android also supports fixed units of measurement such as pixels, points, millimeters, and inches, but Android developers almost never use them. This is because of the wide variation in devices in the market. By using the independent pixels, dps or sps, you'll allow your application to adjust automatically. Fixed units of measurement don't adjust to different devices. Device independent measurements do. You can hard-code your dimensions in your XML layout files.

For example, consider this TextView component.

|  |  |
| --- | --- |
|  |  |

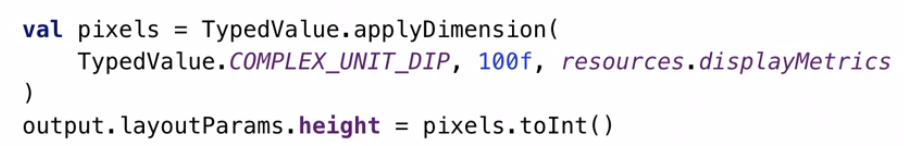
Its width and height are set to wrap\_content, which means it'll size itself automatically depending on the font size and the amount of text. The placement is determined by the marginLeft and marginTop settings. Now, those particular attributes, marginLeft and marginTop, are used in particular kinds of containers or view groups such as the linear layout and a few others. If you're using a constraint layout container, you would instead use constraint attributes. You can declare those values using dp-based expressions. The result will be that the object is 100 pixels from the top and 100 pixels from the left, but exactly what that means will differ depending on the device and the pixel density.

You can declare dimensions as resources, and you can do this in XML files that are placed in the resources directory under the values subdirectory. This is an example of a dimensions resource file.

|  |  |
| --- | --- |
|  |  |

Each dimension has a name and a value. The name is translated as a resource ID. And then to reference that value in an XML layout file, you would start with @dimen/ and then the ID of the resource.

You can also set dimensions using some code. Each view component has a “layoutParams” parameter. The height and width and other values are properties of that object. For example, if you wanted to explicitly set the height of a button, you might say **myButton.layoutParams.height =** and then assign an integer value. The problem with this, though, is that you're setting dimensions as absolute or physical pixels. That's not a very good practice because what it means will differ depending on the device. So instead, you have to convert your pixels to device independent pixels programmatically. First you would use a function called **TypedValue.applyDimension**. You would pass in the unit you want to measure, then you pass in a floating value representing the physical size, and then you pass in resources.displayMetrics.

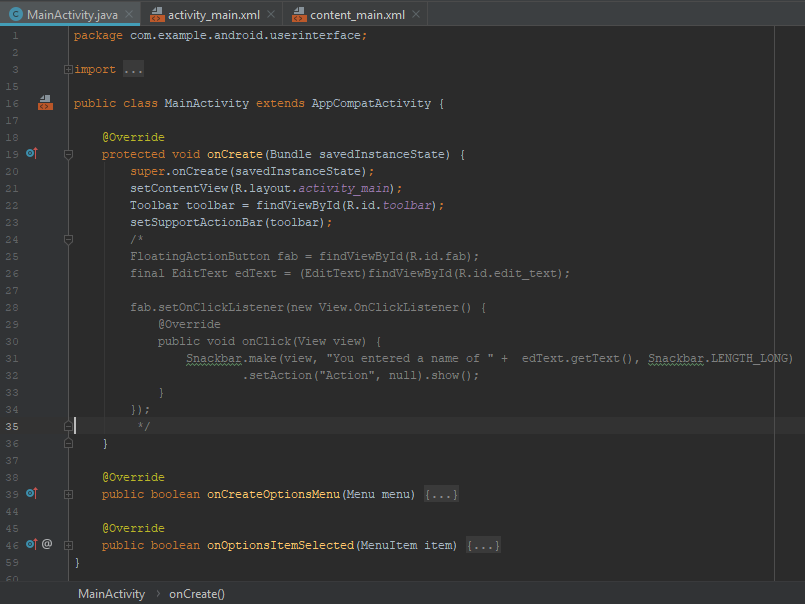


That's an expression that gets information about the current device. The value that's returned is now dynamic based on that device. You can then convert that value to an integer, it starts off as a long value, and then assign that to the height or the width or whatever other dimension you want to apply it to. Most of the time you might only work with XML layout files, and you'll only do this sort of runtime resizing or placement when you need to do something more complex. But it's important to know that when you declare it in XML, it's really being executed at runtime. When you want to change things dynamically at runtime, this is the code that you would use. Either way, you're using device independent measurements so that your application will adapt automatically to whatever device it's running on.

II. Using ViewGroup Components

1. Position Views with LinearLayout

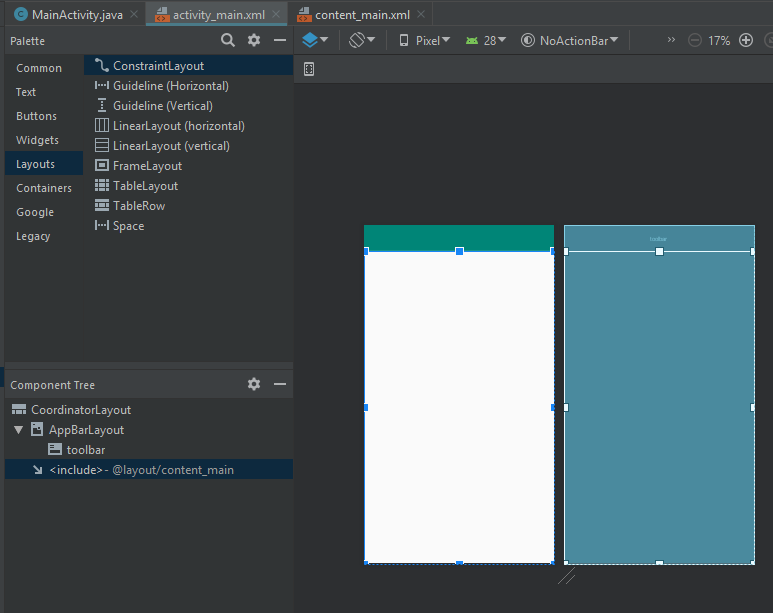
An Android view group component is a container that manages and lays out child objects. Each view group is extended from a view group superclass that implements critical shared functionality and each specific view group has its own rules and logic. For example, consider the previous MainActivity class that has everything stripped out except a very simple onCreate function.



The outer layout file activity main has been changed so that it no longer has a floating action button. The nested layout content main also is empty.

|  |  |
| --- | --- |
|  |  |

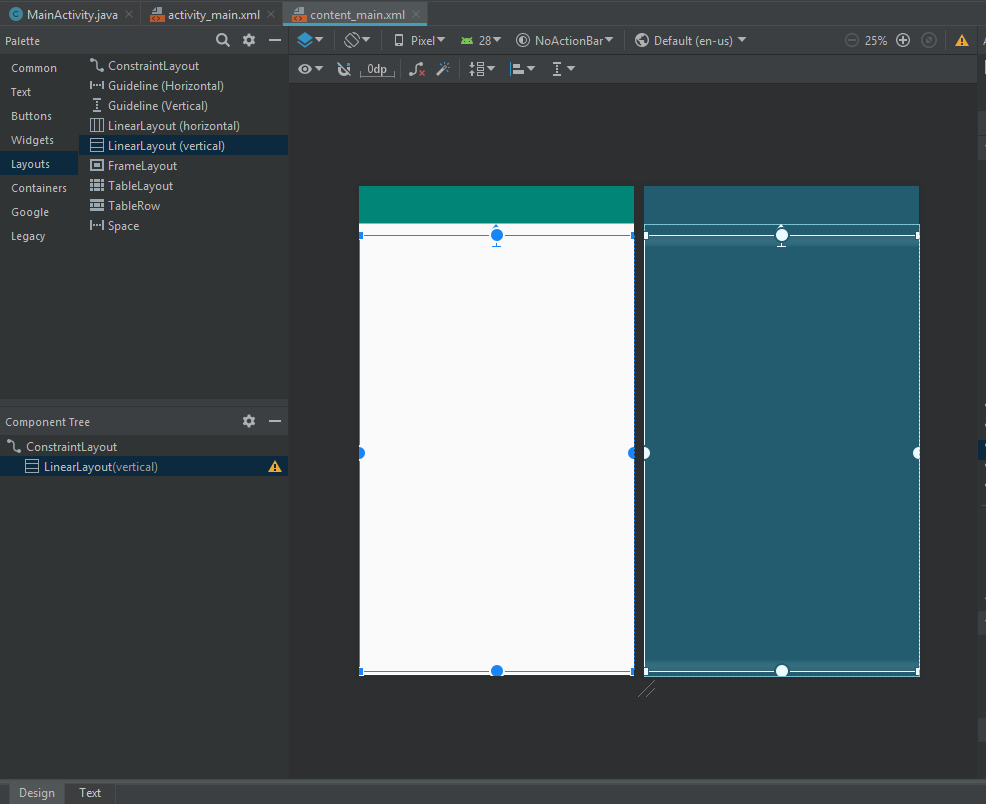
The LinearLayout is one of Android's original container classes. It automatically lays out its child view objects from top to bottom. If its orientation is vertical or from left to right, if it's horizontal. Go to design mode in content\_main.xml and go to the Layouts category and notice that there are two entries for LinearLayout, one labeled horizontal and the other labeled vertical.



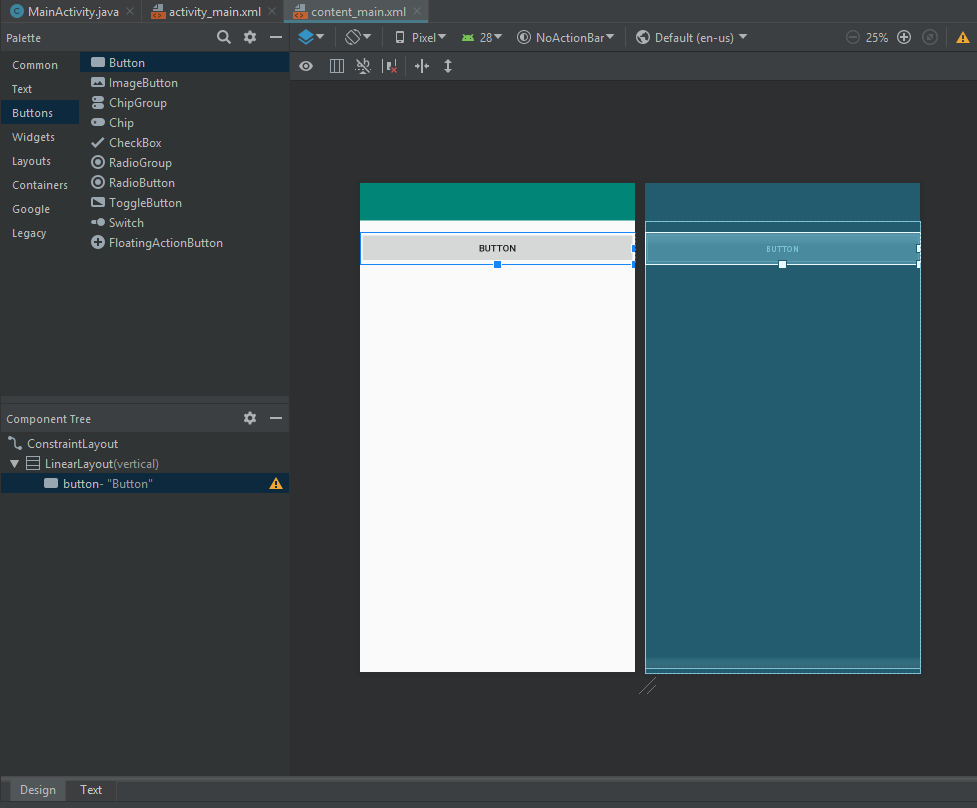
These create the same code but with one attribute that's different which is the orientation attribute. Drag in a vertical LinearLayout and then anchor that to the top and the left side of the container, which is the ConstraintLayout. Now, add two more constraints. Add bottom to bottom of and set that to parent and end to end of and set that to parent as well.



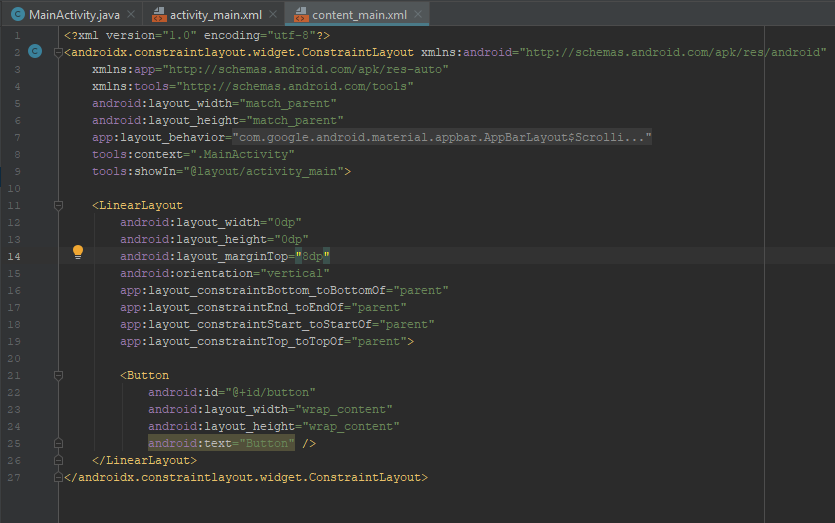
Next go to design view and notice that the LinearLayout now expands to fill the screen.



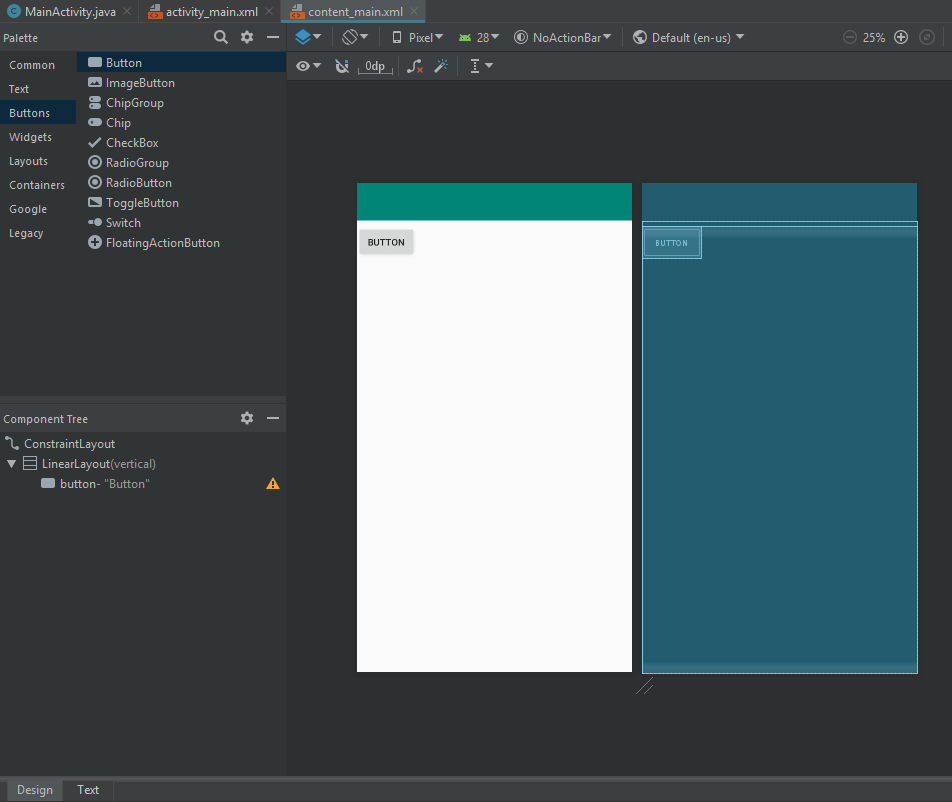
Go to the buttons category and drag in a button. Notice that the button jumps to the top of the screen.



Go back to text mode and notice that the layout width was automatically set as match parent. Change that to wrap content and that will cause the button to shrink down so it only accommodates the text in the button. In addition change both the width and the height to 0dp, which is an instruction to say calculate those sizes automatically.



Go to design mode and observe the button shrink down.

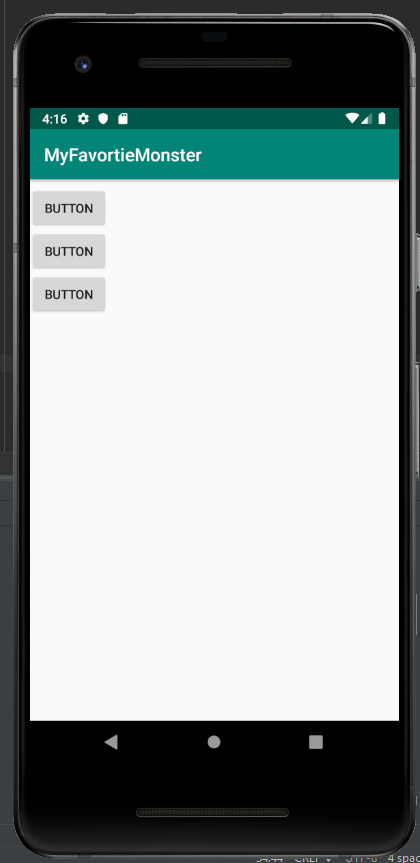


Return to text mode and copy and paste that button component. Remove the ID because that has to be unique.

Select the button code and press Command + D on Mac or Control + D on Windows and that duplicates the selected code. Go back to design view and now the buttons are laid out from top to bottom.

|  |  |
| --- | --- |
|  |  |

Run the app on your AVD to verify that what you see in design mode matches what happens on the AVD and there are the buttons laid out from top to bottom.



Back in Android Studio go to text mode and change the LinearLayout's orientation attribute from vertical to horizontal, and once again, go to design mode and now the buttons are laid out from left to right.

|  |  |
| --- | --- |
|  |  |

Back in code, add another attribute called gravity. The gravity setting can either be set to center, center\_horizontal or center\_vertical. If you set it to center, and then look at design mode again. The buttons float in the center of the screen.

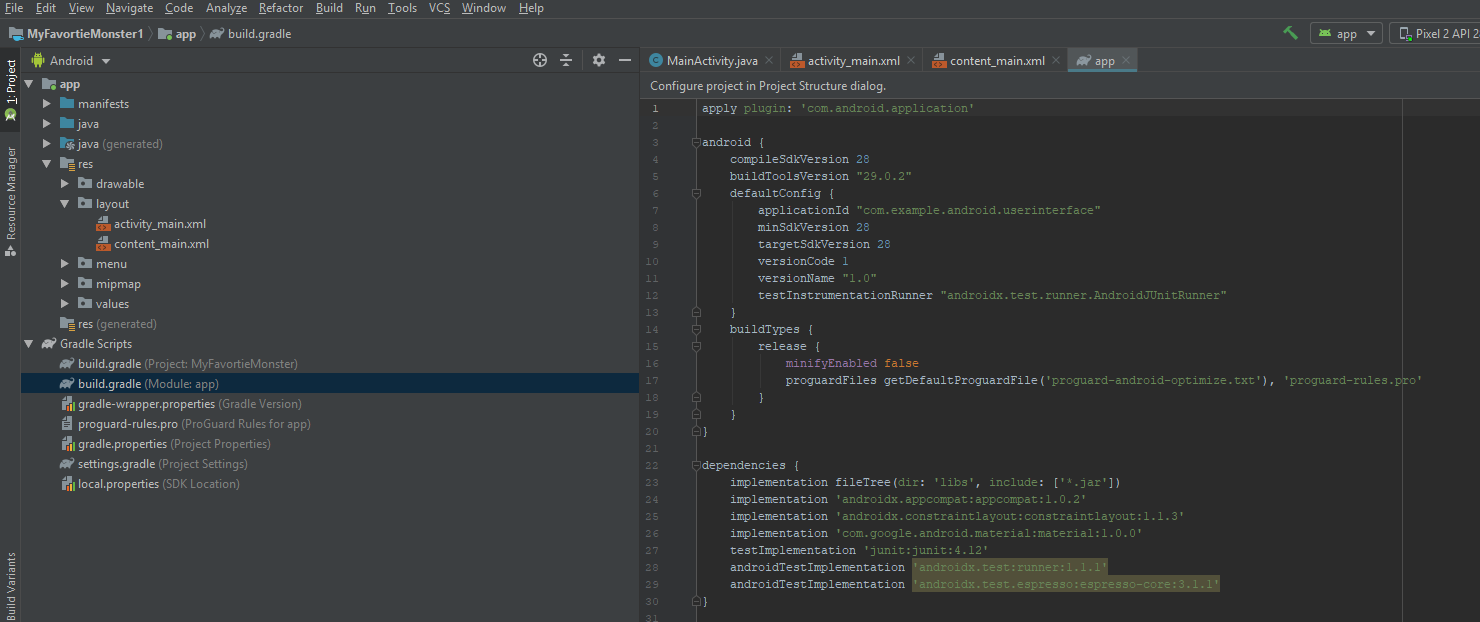
|  |  |
| --- | --- |
|  |  |

If you change that to central horizontal they go to the top of the screen but they're still centered horizontally. You can combine these three attributes in various ways. For example center\_bottom would mean center horizontally, but push the components to the bottom of the layout. As you're experimenting with layouts, you should periodically run the application and verify that what you're seeing in design mode matches what happens when it's really running.

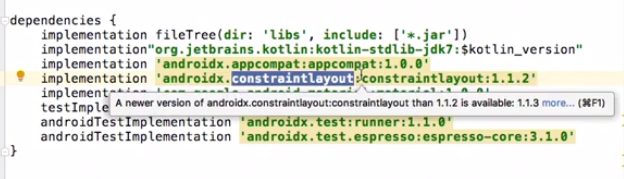
2. Manage View Positions with Constraints

The ConstraintLayout ViewGroup container was introduced in early 2017 after a lengthy public Beta. Since then it's replaced many of the older ViewGroup components such as RelativeLayout. It has improved performance, better memory usage, and it's highly reliable. Unlike the legacy ViewGroup components, ConstraintLayout is delivered in its own library, and it's imported into a project with a dependency declaration in the module's Gradle file.

Under Gradle scripts look at build.gradle, and you’ll see constraintlayout.



This course uses AndroidX packages, so the package is androidx.constraintlayout. There may be warning indicators tell you that there's a more recent version of each of these components available. So for each of those lines you place the cursor, use an intention action with option + return on MAC or alt + enter on Windows and change to whatever is the most recent version. In this example there are four places where there is a warning.



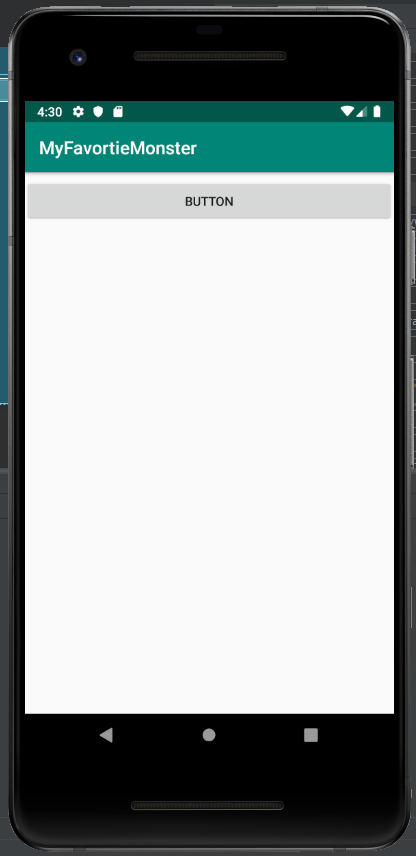
At the top of the screen, if you still see the link to resync your project, click on it, or you can select Build, Rebuild Project. Now you know you're using the most recent versions of any of those components.

Back in the Layout file, go to design mode and once again and drag a button onto the screen.

When you drag and drop a button into a ConstraintLayout, it might look like you've placed it in a specific location, but if you run the application on an actual device, you'll see that the button's location isn't fixed. It will in fact float to the top left corner of the ConstraintLayout container. To fix that, you actually have to add your own constraints. Do that by dragging and dropping, and if the display is too small, expand it.

The button is now placed in the top left corner explicitly, and when you run the application, you'll see the same thing on the actual device. If you want to center the button horizontally, just constrain to the end or to the right side.

If you see jagged lines then those are a visual indicator that says that the button component isn't going to change size and instead the amount of space between the button and the edges will change. If you want to change that so the button expands to fill the available space, go to text mode and change the width from the constant value of wrap content to 0dp. The button will expand and its width fills the available space. You should always test on an AVD to make sure that what you're seeing in design mode matches what happens on the AVD.



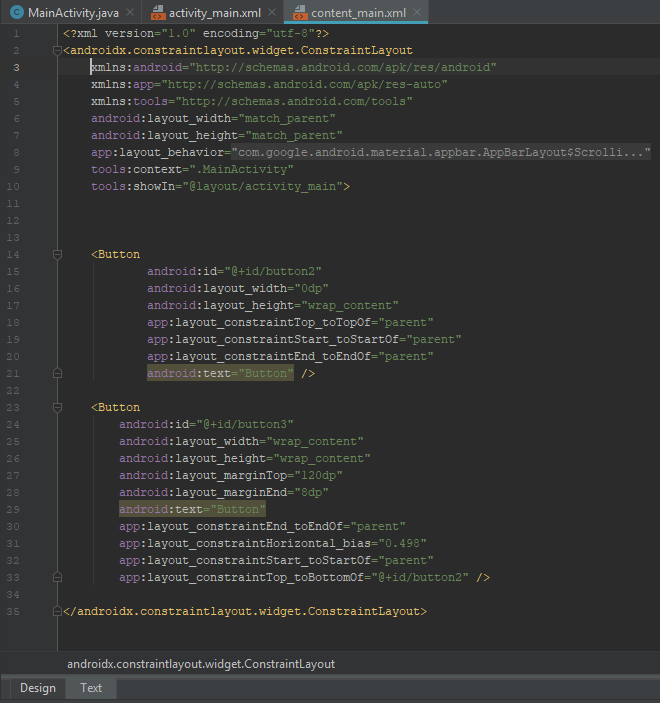
You can also constrain objects to each other. If the linear layout constraint is present, then remove it. Drag in another button component, and once again its placement isn't fixed yet.



Constrain it to the left and right borders, but now constrain the top of this button to the bottom of the first one.

If you want to change how much distance you have, you can click and drag and move it down a bit, and notice that as you drag there's a visual indicator of how many device independent pixels there are between the first and second button. In code, that translates as an attribute of margin top for the second button.

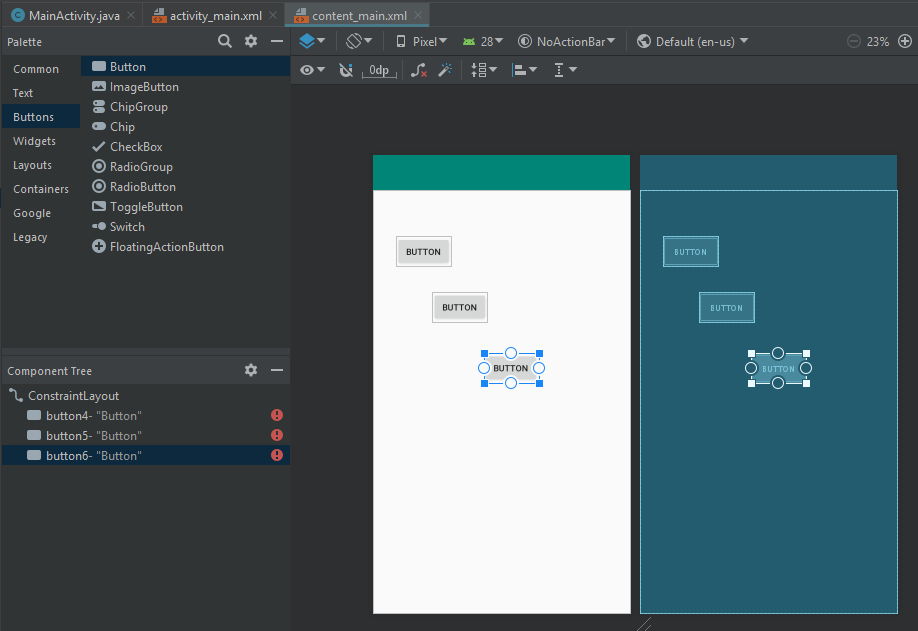
If you want to change that in code, you can do it as shown below.



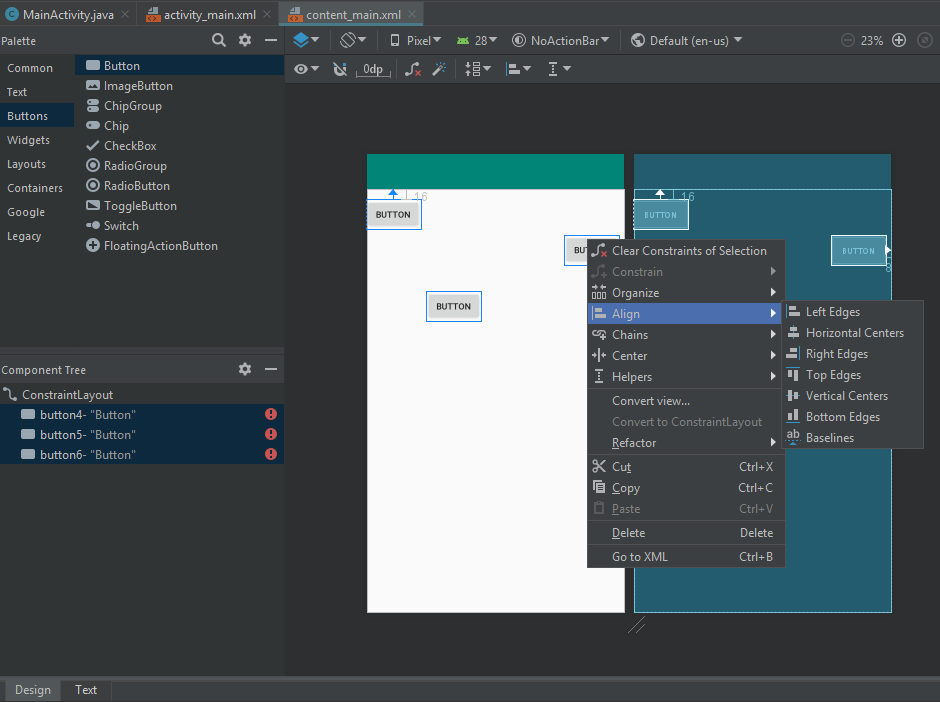
You can go back and look at it in design view, and you’ll see the change reflected there. If you only use the ConstraintLayout component in these very simple ways, you'll end up with clean code and great performance.

3. Distributes Views with Constraint Chains

The ConstraintLayout ViewGroup has a feature named “Chains” that lets you distribute objects either horizontally or vertically on the screen. Start with an empty project and drag in three button components and place them anywhere on the screen.

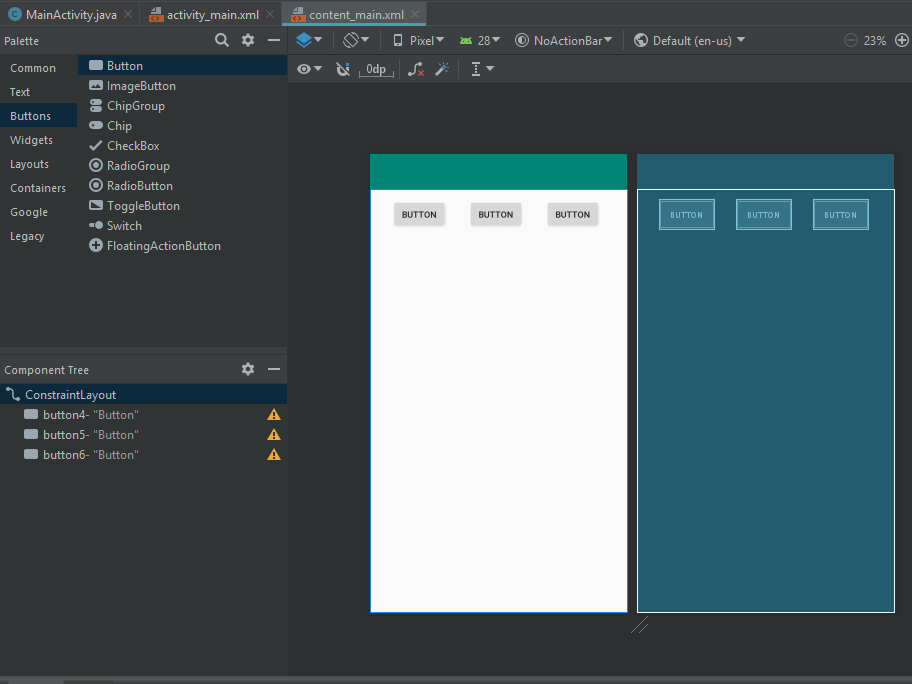


For the first button, line it to the top and to the left, and for the third button align it to the right.

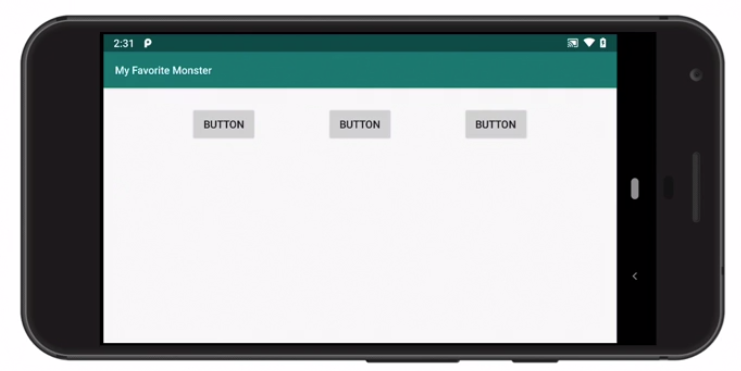


Next, click on the first button, hold down the Shift key and click on the other two buttons, so that they're all selected. To align them to the top, right-click and choose Align, Top Edges. This chains the buttons together. The second button's top edge is aligned to the first button's top, the third one to the second one.

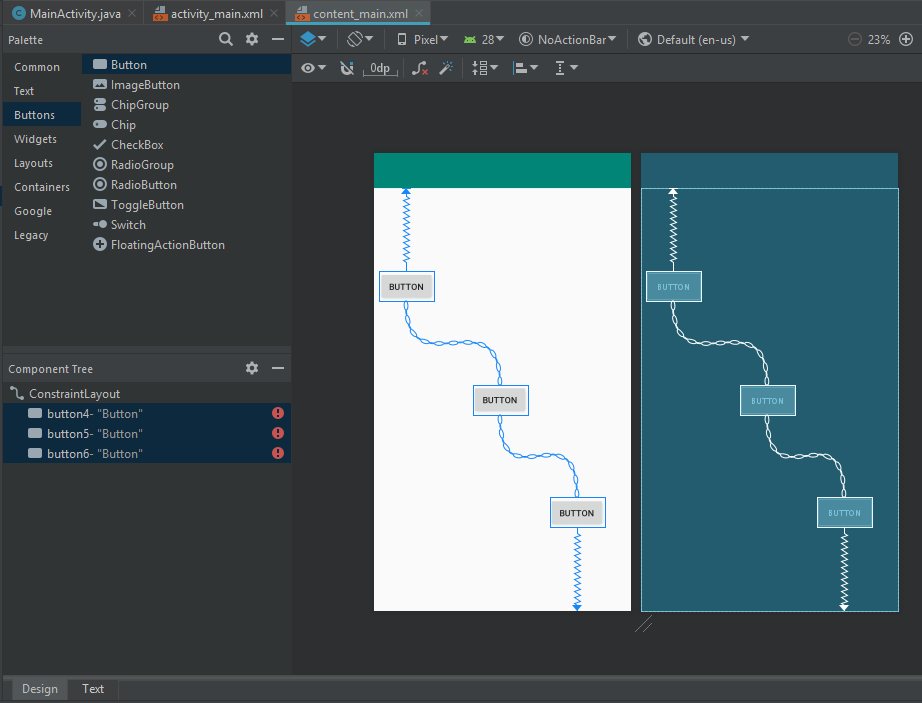
To distribute the buttons evenly across the screen, once again, make sure all three are selected. Right-click and choose “Chains,” and then “Create Horizontal Chain.” Now the buttons have an equal amount of space on the right, on the left, and between them. There's an extra margin to the right and left of the outer buttons, but otherwise they're all evenly distributed. Check to see what the appearance is on an AVD. The buttons are evenly distributed, and again there's a little bit of extra space on the left and the right.



Turn the AVD to landscape orientation, notice the buttons move, but once again they're evenly distributed horizontally.



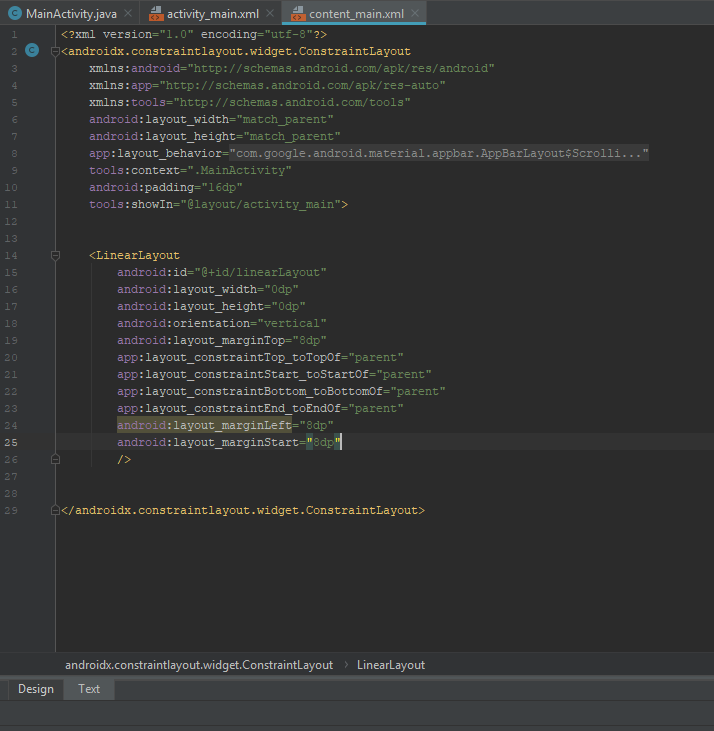
Turn the AVD again, go back to portrait, and once again they move around as needed. You can also distribute the buttons vertically. Right click and choose “Chains,” “Create Vertical Chain,” and now the buttons are distributed horizontally and vertically at the same time.



The combination of constraints and chains makes it very easy to create dynamic layouts with outstanding performance that adapt to any screen that the app encounters.

4. Display Views Programmatically by Using View References with “findViewById()”

Each view group is a container that can have as many child view objects as are needed. Most of the time you'll probably declare your views in XML layouts, but there will also be times when you might want to add views at runtime using Java code. The version of the app below has a linear layout that's nested within the constraint layout. It has an ID of linearLayout that you’ll be able to address dynamically in Java code.



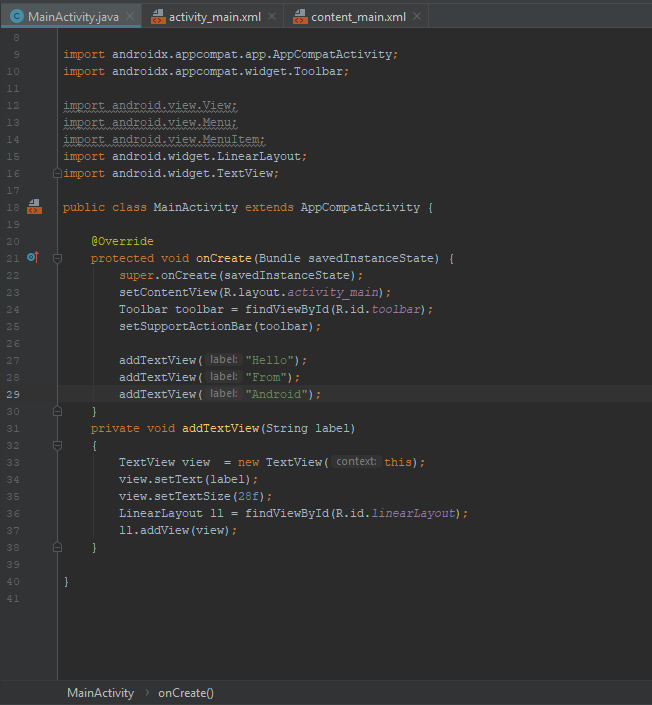
To access your MainActivity class, press the shift key twice for “search everywhere” and start typing “MainActivity” and then select it to open it. Create a private function called “addTextView.” It will receive one argument named “label” and set the type as “string.” Within the function, create a local variable named “view” which will be instantiated using the TextView's constructor. Type the name of the class and then press Enter or Return. This ensures that you have the required import at the top of the file. Use the constructor that receives the context and pass in “this.”

This TextView is bound to the current context but it's not actually displayed anywhere yet. That'll come later. Next you need to set a couple of properties. First, set the text property. In Java you would use a method named “setText.” Assign the text value of the view object to the label that was passed in as an argument.

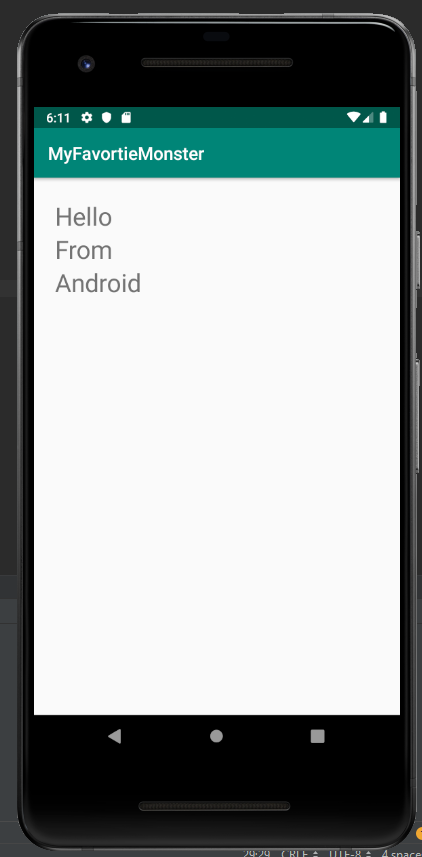
Next you'll set the size of the text with view.textSize. Use a floating value of 28f, which is 28 DPs or device independent pixels.

Now you’re going to use that linear layout. It has an id in the xml layout file of linearLayout and you can reference it by using the “findViewById()” method and the “R” java class. Call the layout's addView() function and pass in the view object.

Now in the onCreate function, that function three times. Call it the first time and say “Hello.” Duplicate that line a couple of times and pass “From” and then pass “Android” to the corresponding duplicate calls.

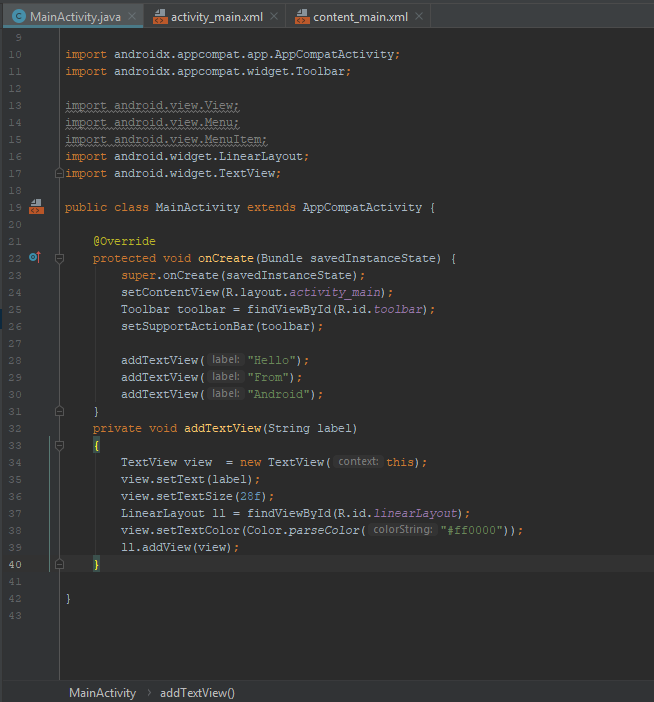


Run the application. As the application opens, the three textView objects are placed on the screen. Because they're within the vertical linear layout, they're stacked from top to bottom.

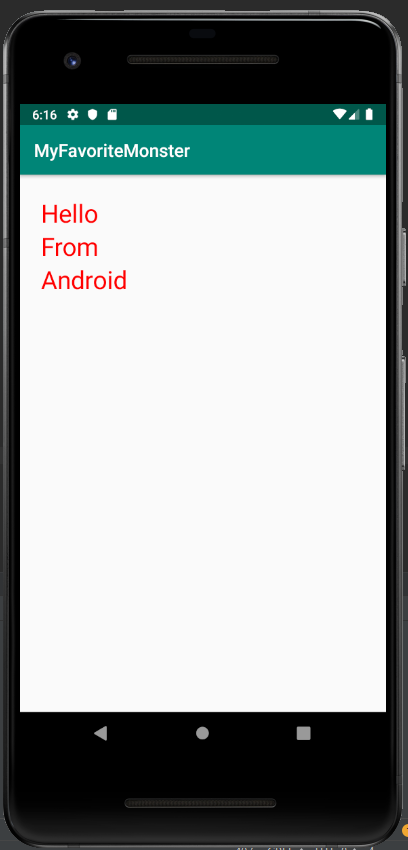


You can add other dynamic code. For example, let's say you wanted to change the color of the TextViews. Place the cursor before the call to “addView” and call a function named “setTextColor.”

This function takes an integer value. In order to get the right integer value, start with color and choose that class. Next, call the function “parseColor” and you'll pass in a string. This is a hexadecimal value similar to what you might use in a webpage. “#ff0000” means make it red.



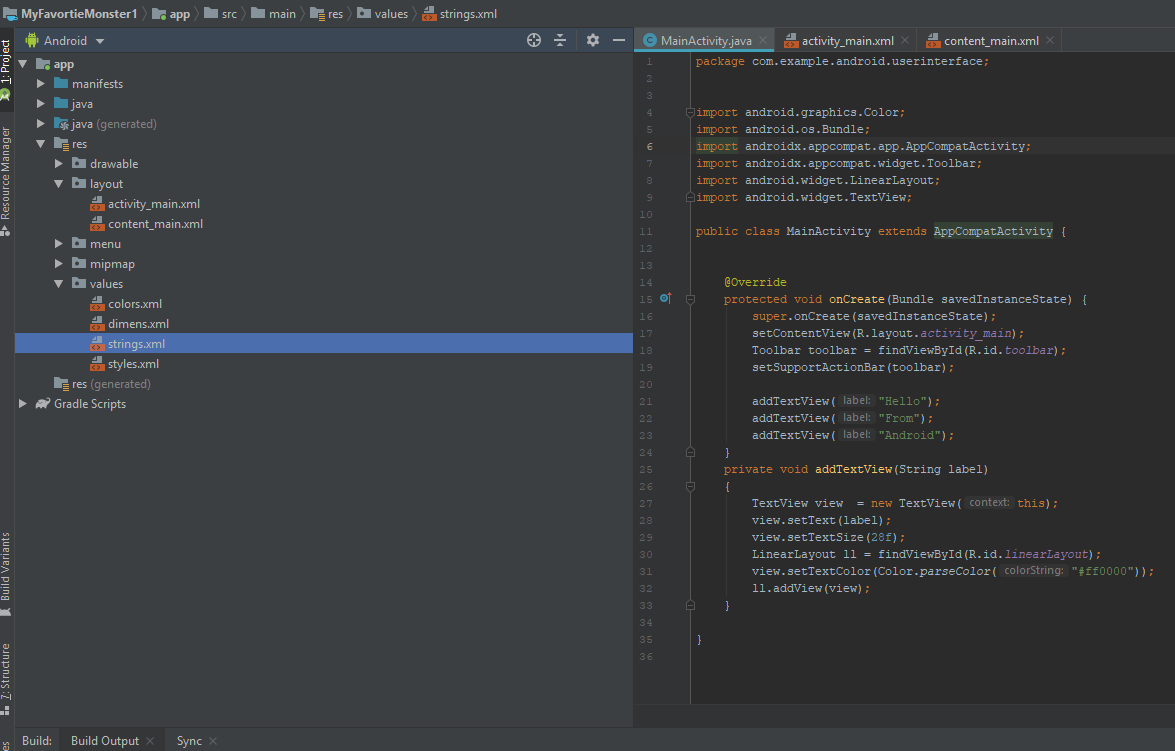
Run the code again and this time when the textViews appear, the text is red. The ability to programmatically add views to your layout at runtime lets you build dynamic interfaces that can react to user gestures or different conditions.



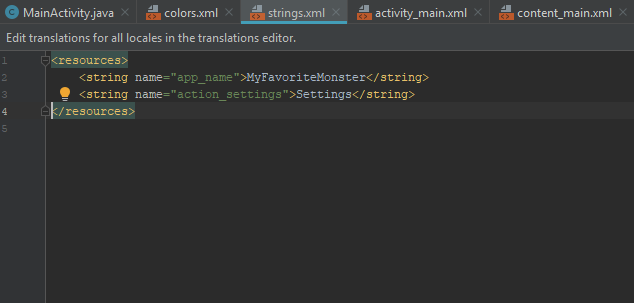
III. Display Text and Images

1. Manage String Values as Resources

Every Android app depends on the use of resources. There are many different kinds of resources, and they're stored under the **res** directory within your app module. There are image files in the drawable resource directory, XML layout files in Layout, XML menu files in Menu, more images in mipmap, and then a variety of resources in the values directory. Within files in the values directory, the type of resource is determined by the element name. So you have color resources, dimension resources, style resources, and string resources.

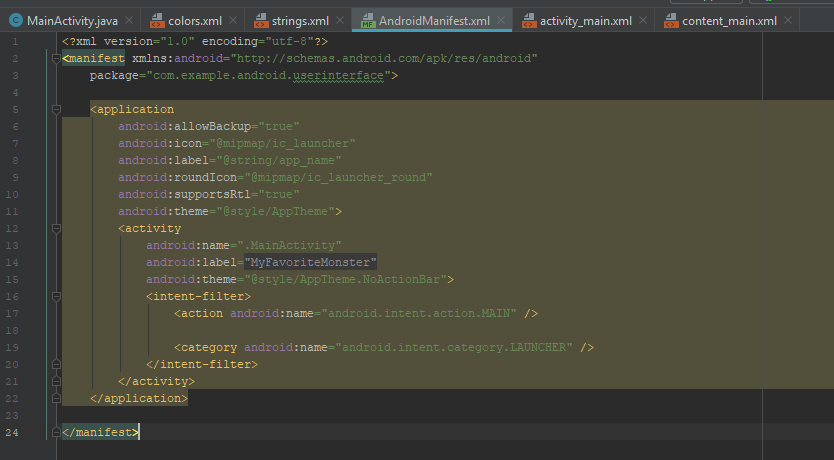


A string resource is defined in this file and then can be referenced in your XML layout files and in your code.

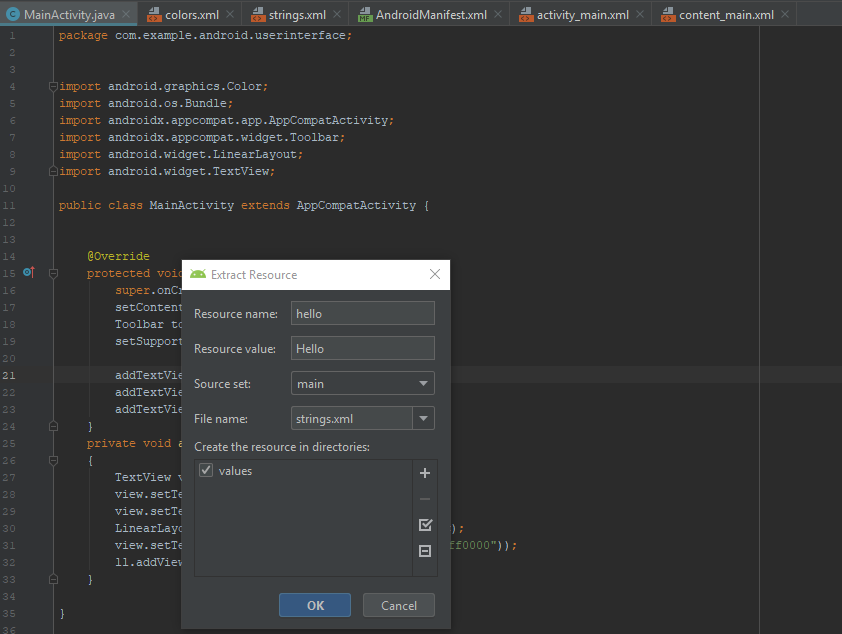


By creating your strings as resources, it makes it easy to create translations so you can localize the application for different human languages. It's a good practice to make all of your strings resources so you can easily localize later on if you need to.

Here’s an explanation of how these resources are used. First of all, in your XML files you can reference these using the following syntax. Go to my manifest file and show that the “app\_name” resource is referenced with “@string/” and the resource id.

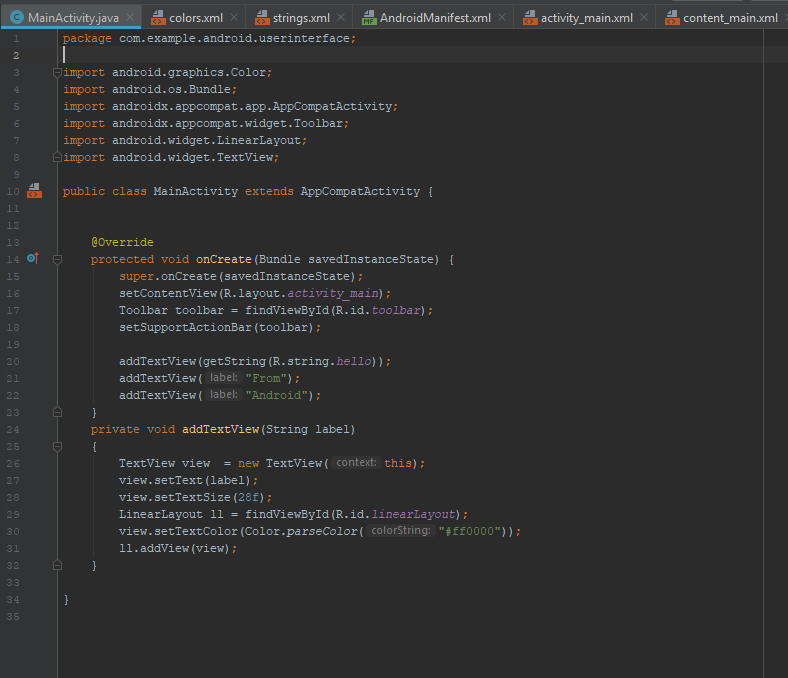


Resource id's are always lower case with underscores where you need to separate words but otherwise no spaces or special characters. To reference resources in code, you can use a function named “get\_string()”. In the main activity class, place the cursor inside one of the literal strings and use an intention action, and that's option/return on Mac or alt/enter on Windows, and then choose extract string resource.



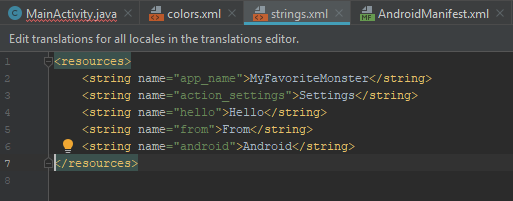
As show in the illustration below, set the resource name as “hello,” all lower case, and you can place this in any of the resource files, but it makes sense to put it in strings.xml. Click OK.

Now you’re referencing the string resources with the getString() function.



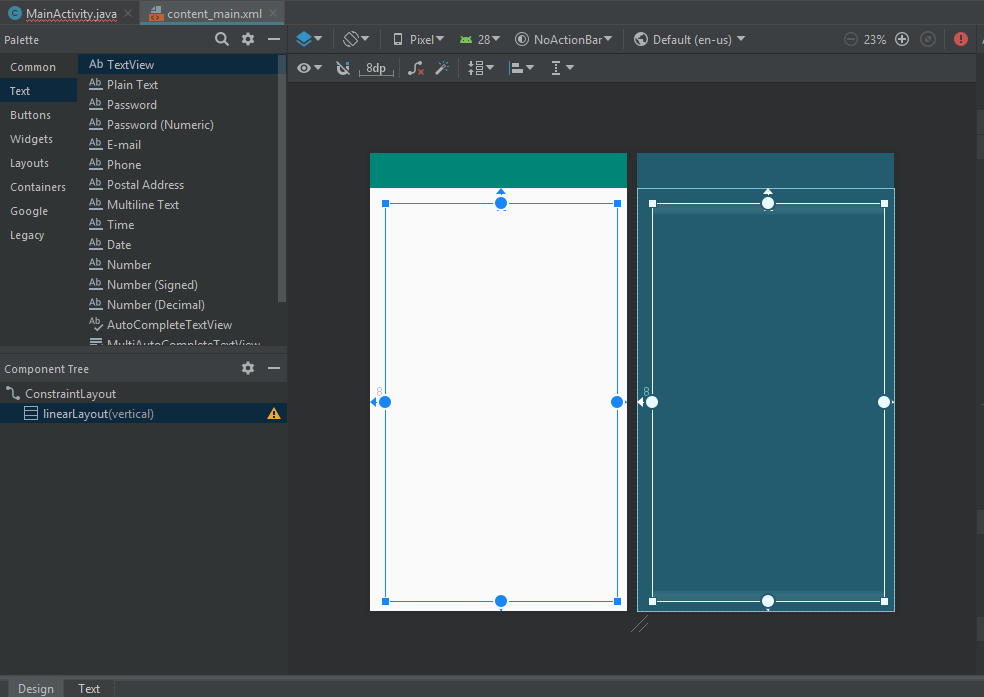
The syntax “R.string.hello” refers to a class named “R” that's being dynamically generated in the background. You can find this file by pressing “shift” twice for search everywhere, then type R.java and look for a file in the directory matching your application space package. You'll find a whole variety of resource id's here including your own resource id's and resource id's that are generated from the Android SDK.

Go back to your MainActivity class and do the same thing for the other two strings. Extract a string resource and set the second on as “from” and then do the same thing for “Android.” Control or Command click on one of those resources and show that they've all been defined in the strings.xml file. When you run the application it does the same thing it did before, but now the application is more flexible and maintainable because you’re defining all of your strings in one location. As an application gets larger, you might want to create multiple string resource files. You can name them anything you want because what matters is the string element and not the file name.



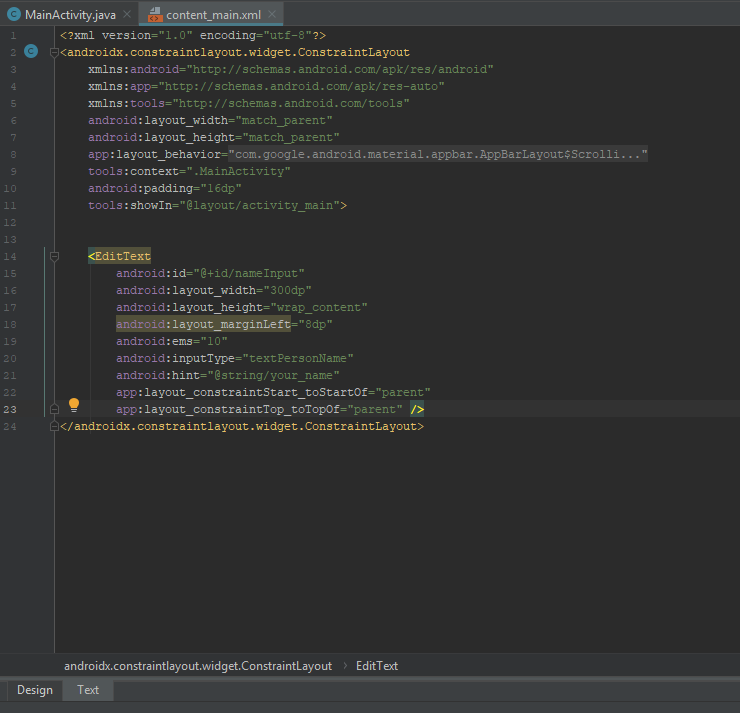
2. Manage Data Entry with EditText

The Text View component presents read-only text, but when you want to create a data entry form to collect information from the user, you need an **EditText** component. These are available on the palette when you're in Design View in an XML layout file and there are a number of versions that you can use.



The “Plain Text” one looks for simple text, but there's also one that receives a password and masks the data entry with asterisks and others that validate the data entry for various values, so instead of seeing a standard qwerty-style keyboard, you see a numeric keypad. These are all actually the same component, the **EditText** component, but with different attributes. To demonstrate this, drag in a plain text object and anchor it to the top and to the left and this out a bit, so it's a specific width.

Go into Text Mode and see what the result was and change this explicit width to 300dp and change the height back to wrap\_content. Notice that this has an input type of textPersonName, which, as the user types a value in, the first character will be uppercase. There's an initial text value and so you can remove it. Add a hint which will not show up on the text box until the user starts typing something. Set the hint as “Your Name,” then use an intention action and extract that as a string resource of “your\_name.” Next, change the ID. Use an ID of “nameInput” and then get rid of the margin settings that were generated. All you need are the constraint to the top and the constraint to the start and left.

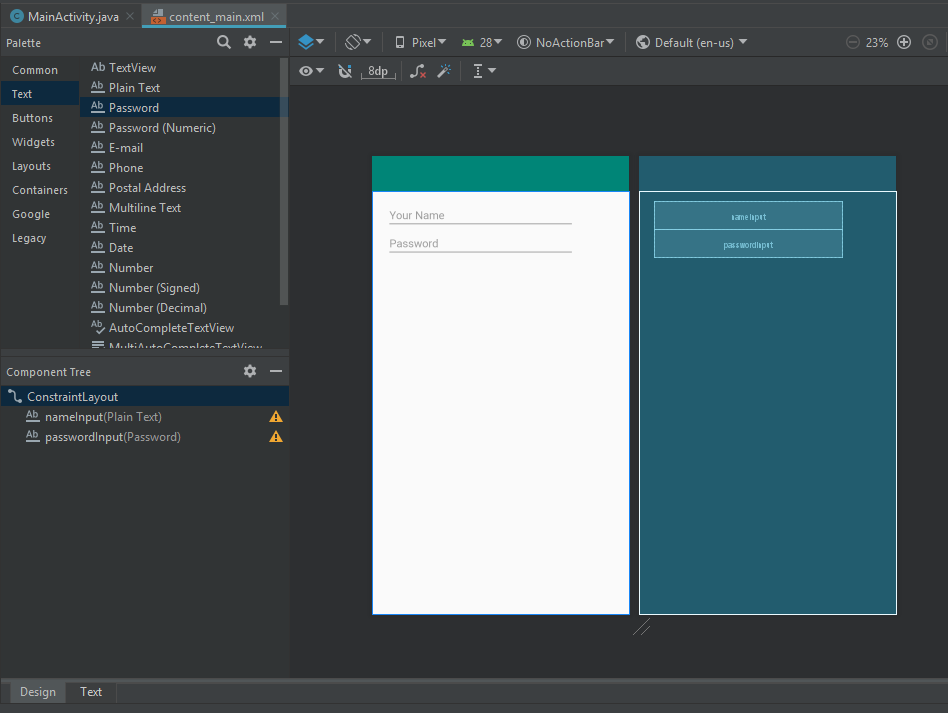


Go back to Design Mode and drag another EditText in, this one for a password. Constrain the top of this to the bottom of the first EditText and constrain this one to the left. Go into Text Mode and make the following changes.

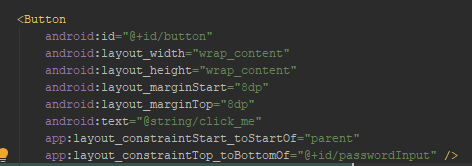
Set the width explicitly as 300dp, accept the input type of “textPassword,” set the ID as “passwordInput,” and get rid of the margin settings. Go back to Text and add a hint for the password, which will be “Password” and then once again, extract that as a string resource.

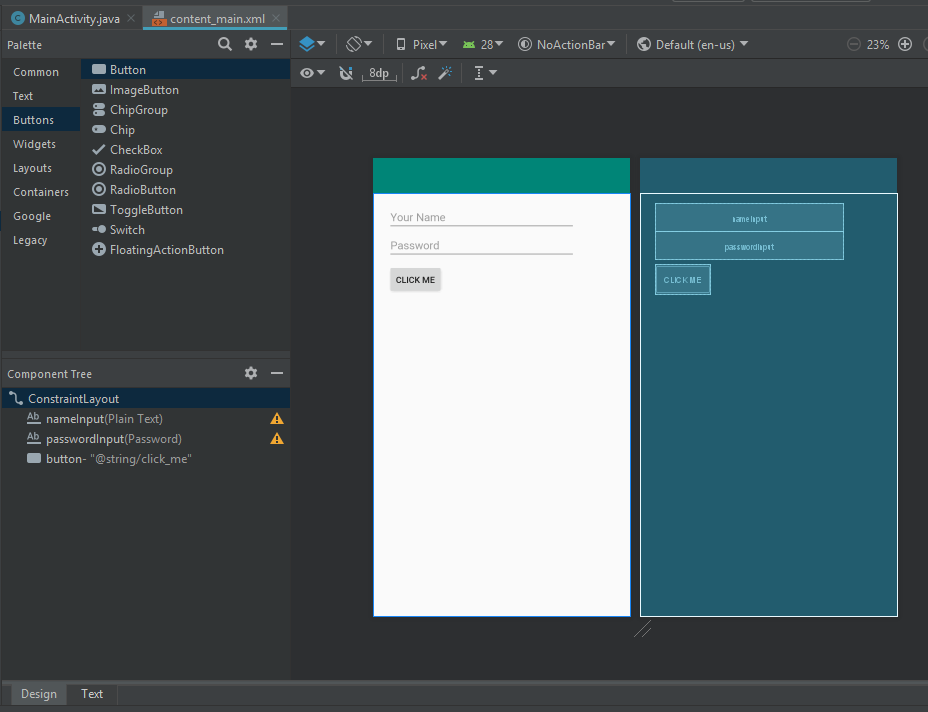


This is what it now looks like in Design View.

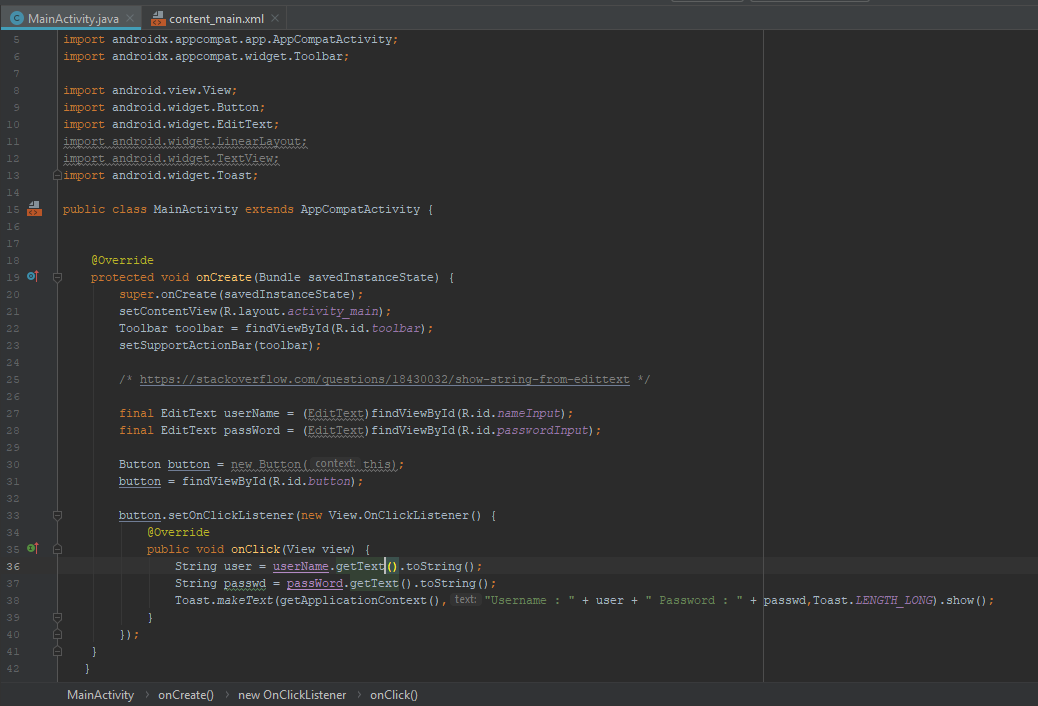


Finally, add a button. Go back to the Common category, drag in a button, constrain that to the bottom of the password and to the left and set the text of the button to “Click Me.” Once again, extract the string resource. The ID is button and that's fine. Get rid of any remaining margin settings and here's what the data entry tool looks like.

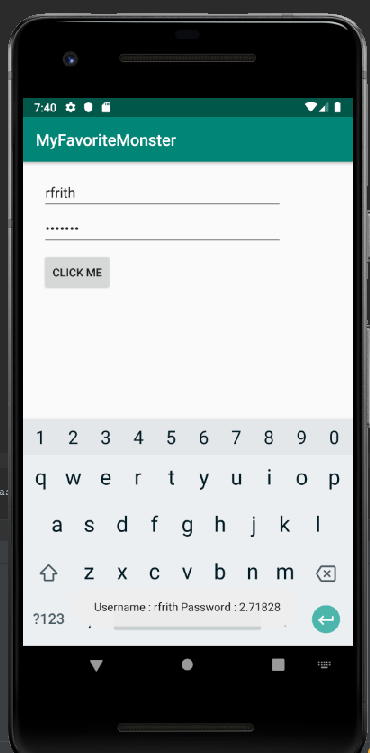




Event handlers will be covered more in-depth in another module, for now, you’ll add an event handler for the button and you'll do that in the main activity. In your “onCreate” function, reference the button object and as has happened previously. Implement the following code to reference the user name and password text fields. You’ll retrieve those values in the button’s event handler using the following code. You may want to display this information on the screen. If this were a real login form, you wouldn't do this, but this will just prove that everything is working correctly. You’re going to use something called a “Toast Message,” which is a message you can display on the screen whenever you need to. You will use a class name “Toast” and from there, you'll call a function called “makeText” which requires three arguments. The context will be the current activity and then strings as shown below. After the second string, pass on a duration and use a constant of the Toast class “LENGTH\_LONG” and then from there, call a function called “show.” So “makeText” creates a Toast object and show displays it.

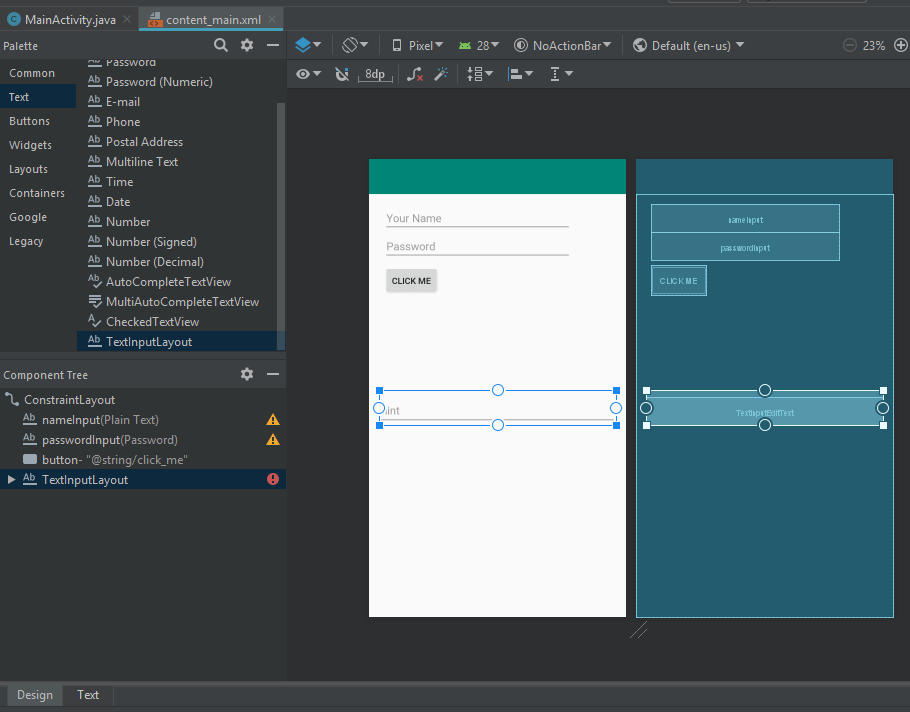


Test your application. Notice the hint “Your name” is displayed in a dim font and when you click into it and start typing, the hint goes away. Type in any value for a password and touch that down arrow at the bottom of the screen to get rid of the onscreen keyboard. Touch “CLICK ME” to get the result. You’ll see the concatenated string displayed in the Toast message and after a certain period of time, it goes away

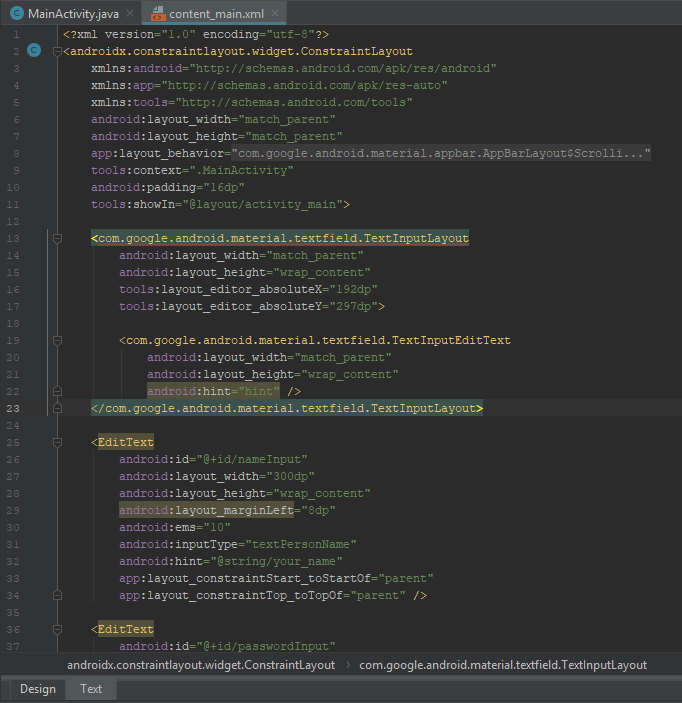


3. Manage Data Entry with TextInputLayout

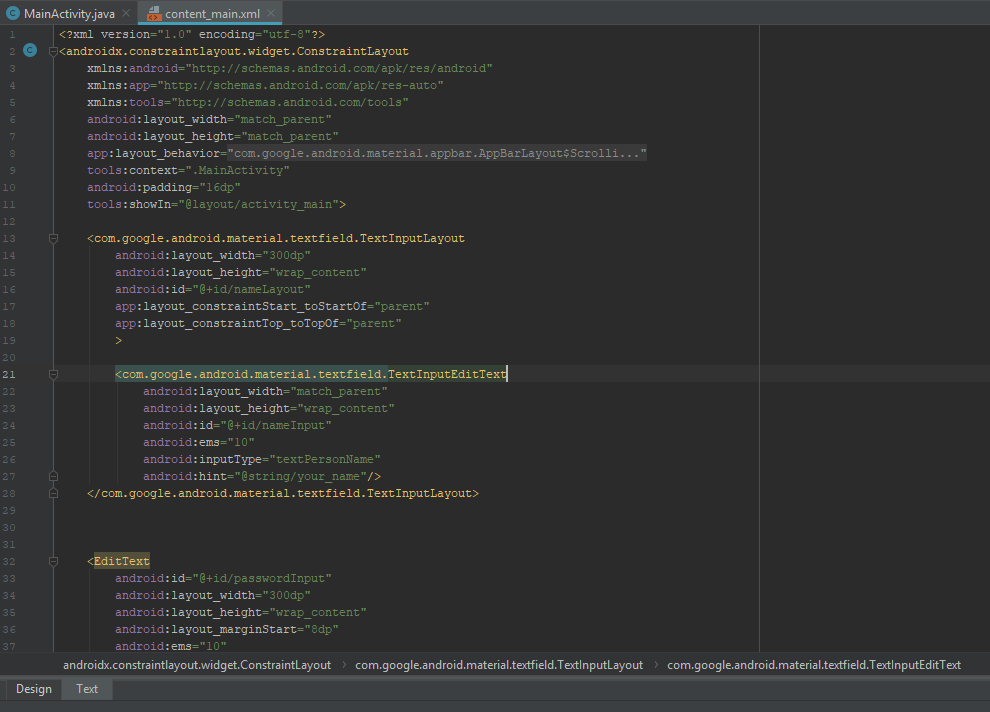
The EditText component does a great job of collecting data from the user, but it has one user interface flaw. When you set a hint in the EditText component, it's displayed as long as the text is empty. As soon as the user types something in, the hint goes away and the user experience sometimes involves having to back out the text so they can see the hint again. You can solve this with a component called “TextInputLayout.” It's a Material Design component and is not a part of the core Android SDK. It's backward compatible to older versions of Android and it's provided by an external library. To use it in this form, go to **Design Mode**, and then go to the **Text** category and down at the bottom is where you’ll find the TextInputLayout. Drag and drop it into your form.



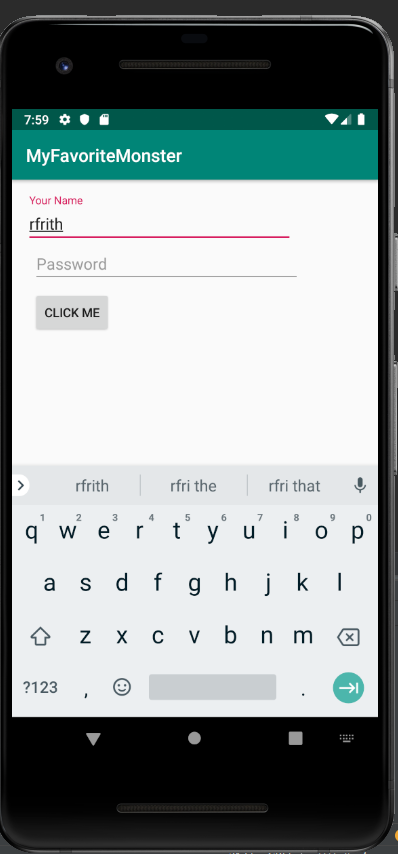
Go to **Text Mode** and look at the generated code. You can always tell when a component is a part of an external library when you drag it in in Design Mode because the generated code will include the complete package, not just the component name. Now take that generated code and cut and paste it and place it up toward the top of the form, above the first EditText component. This will make it easier to start moving some attributes around.



First, match the width of the EditText component to the width of the TextInputLayout, that's 300 DP. Next, take the four attributes, starting with “inputType” and ending with “id,” and copy those. Replace the hint by pasting them into place. You now have a collision between the id as shown with red underline. Next take the constraint settings below the second id, and paste those in the input layout. Get rid of the tools attributes for absoluteY and absoluteX. Finally, get rid of the old EditText component because it has been completely replaced. You have one error that you need to deal with. You can no longer constrain the second EditText to the nameInput component because they aren't peers; that is, they don't have the same parent container. So add an id to this layout as shown below and give this an id of “nameLayout.”

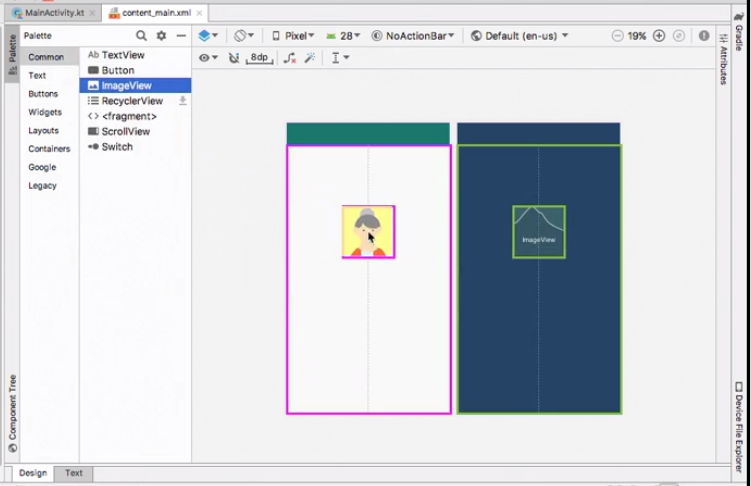


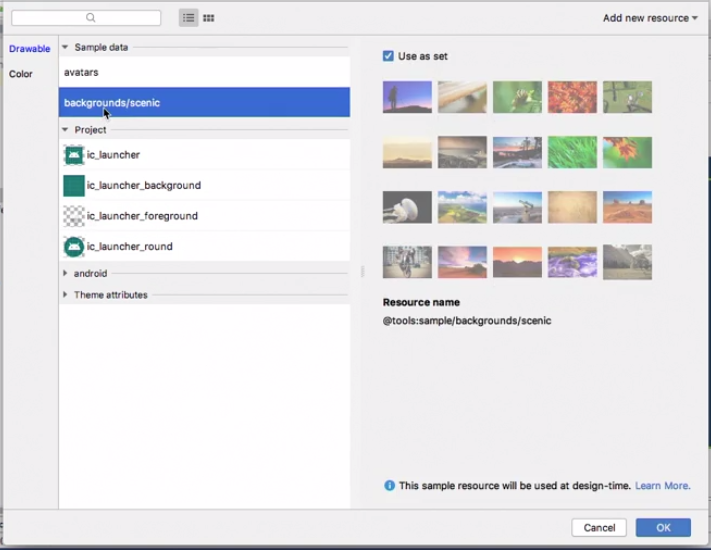
Look at this in Design View, and it looks the same in Design View as it did before. Run the application on AVD and see the difference. When the form first appears, you still see the hint, but when you give it focus the hint floats up to the top. You can start typing in values while still seeing the reminder of what the EditText component is for. The TextInputLayout component is a vital tool for making your data entry forms in Android apps easier to use for the user.



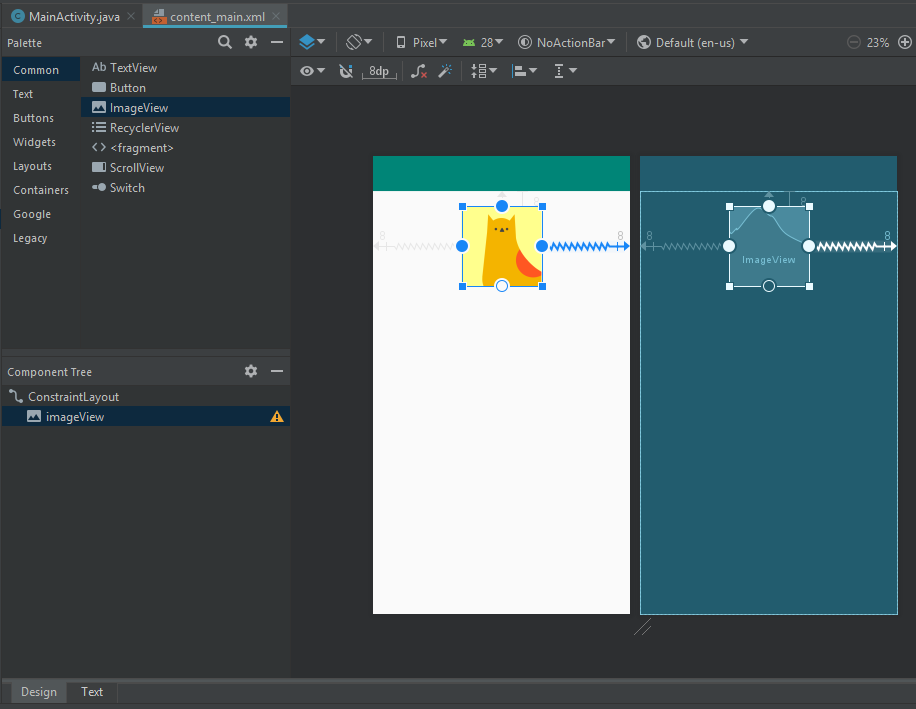
4. Display Image Resources

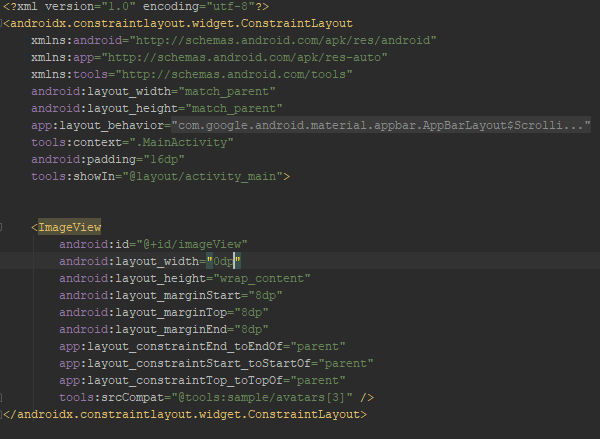
In order to display an image in an Android app, you can use either the image component which is passive and only displays the image, or the image button which displays the image and can be treated as a button. The user can touch or click it to make something happen. This section will present the simpler component, the image view. Drag and drop an image view into your screen from the palette. When you drop it, a resources dialogue appears, and if you have any image files in your project already, you can choose one of them, or you can choose one of the avatars, or scenic backgrounds to work with while you're designing the screen. These images are provided in Android studio, but they only show up in Android studio. If you deploy your application to a device you won't see them. Choose the avatars collection, uncheck the use as set option, and choose the cat image.

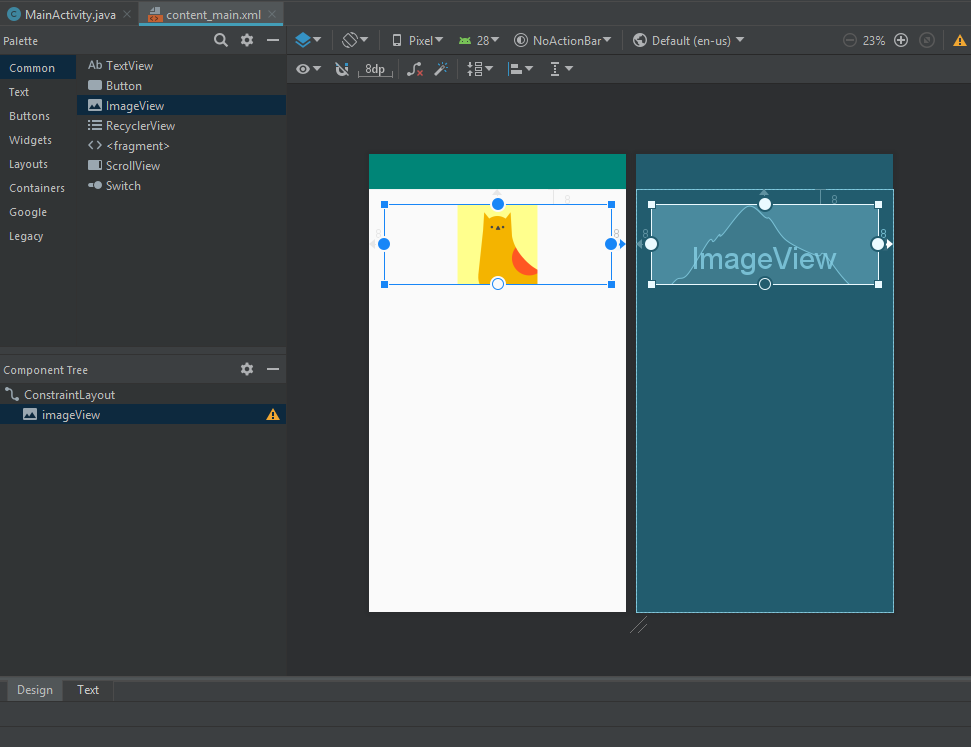




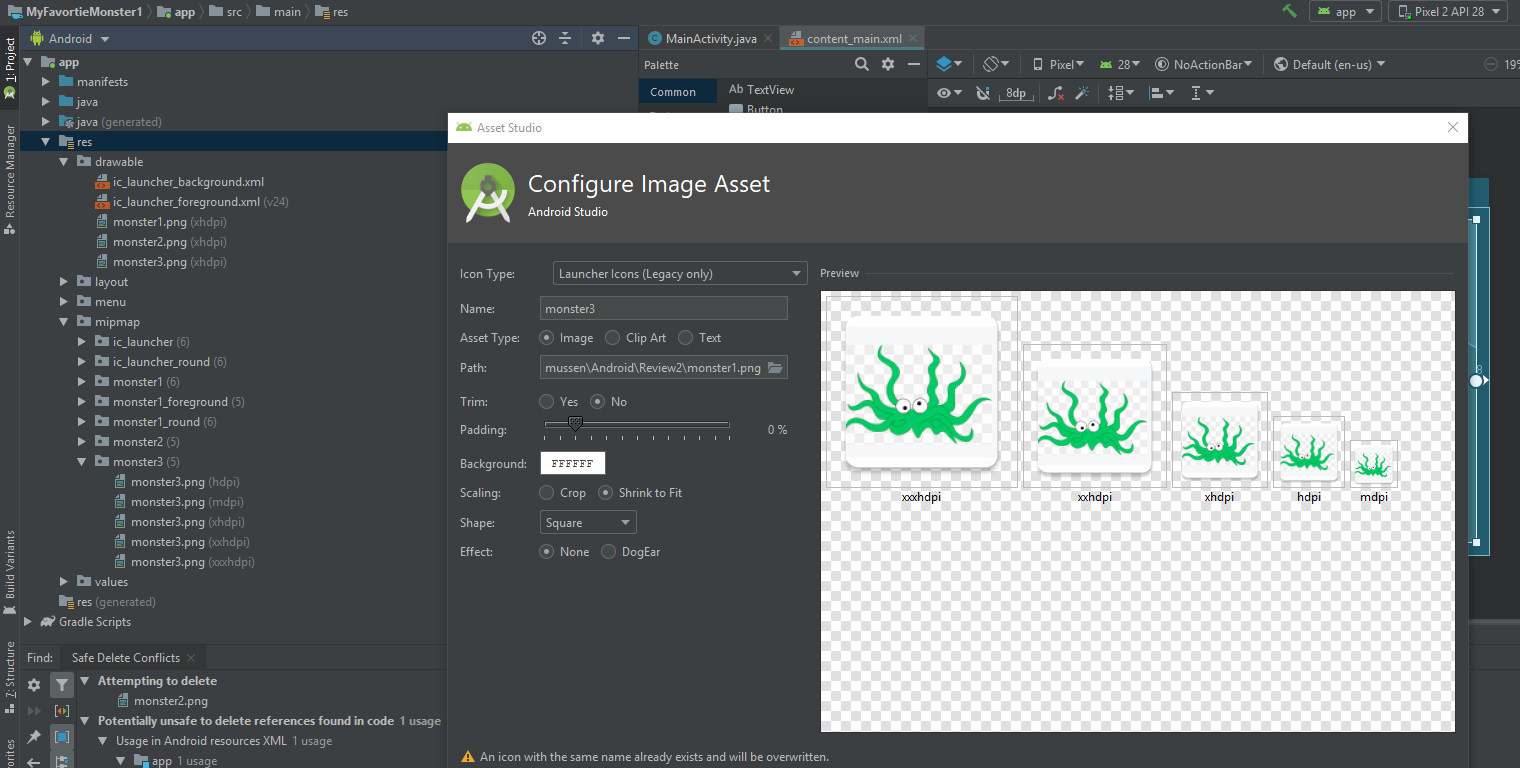
Now you’ll want to place the image view on the screen, by constraining it to the top, to the left, and to the right. Initially this means place the image view in the center of the screen horizontally but don't change it's size. You’ll modify that behavior in text mode, by setting the width to zero DP, and that means stretch the image view to fill the screen horizontally, but notice it doesn't change the size of the image itself.





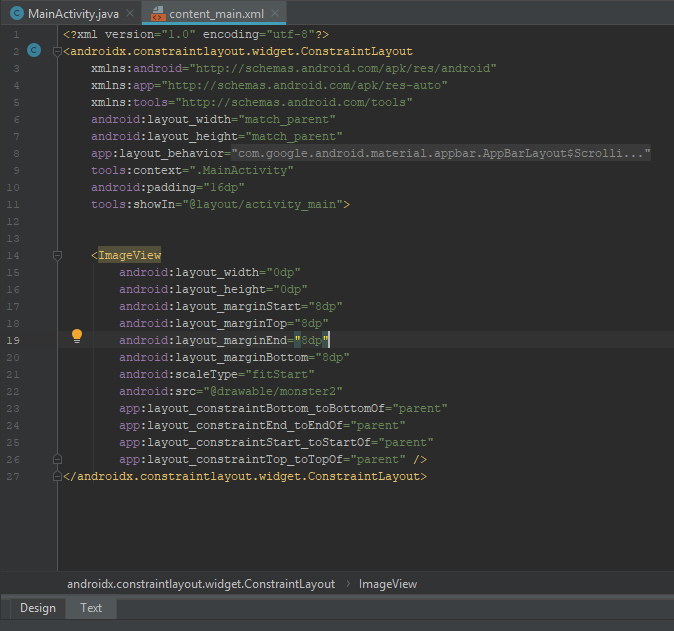


Next, you will work with some actual image files. To add images to your project, right click on the “res” folder and select “Configure Image Asset.” The following Asset Studio window will popup. Add images to the mipmap subfolder as follows.

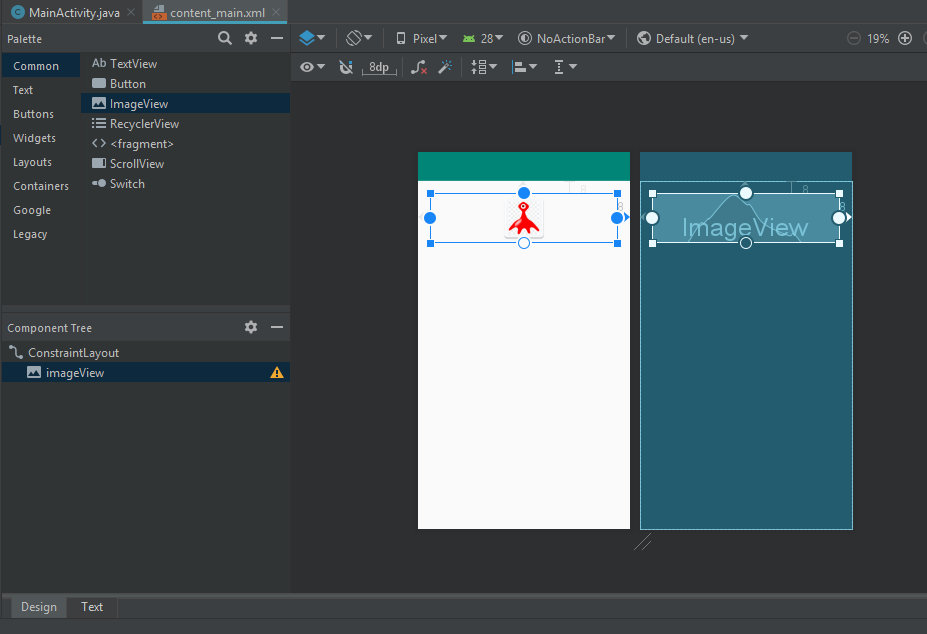


The listed images were in the author’s asset directory, but not necessarily in yours. Copy your images to the clipboard and then go to your app directory, then to the src folder, followed by the res folder, and then to the drawable directory. Paste you images into that folder. You may have to use those images in the “xxxhdpi” format.

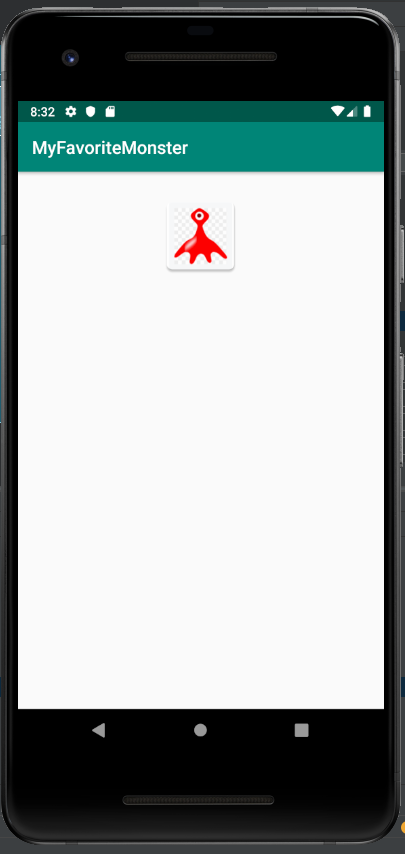
You now have three different PNG files to work with. The image view component can display PNG files, JPG files, and the newer WEBP format. Now you’re going to display a PNG file. Go to text mode. Remove the “tools” based attribute. Instead use the source attribute and start with “@drawable” and choose monster two.



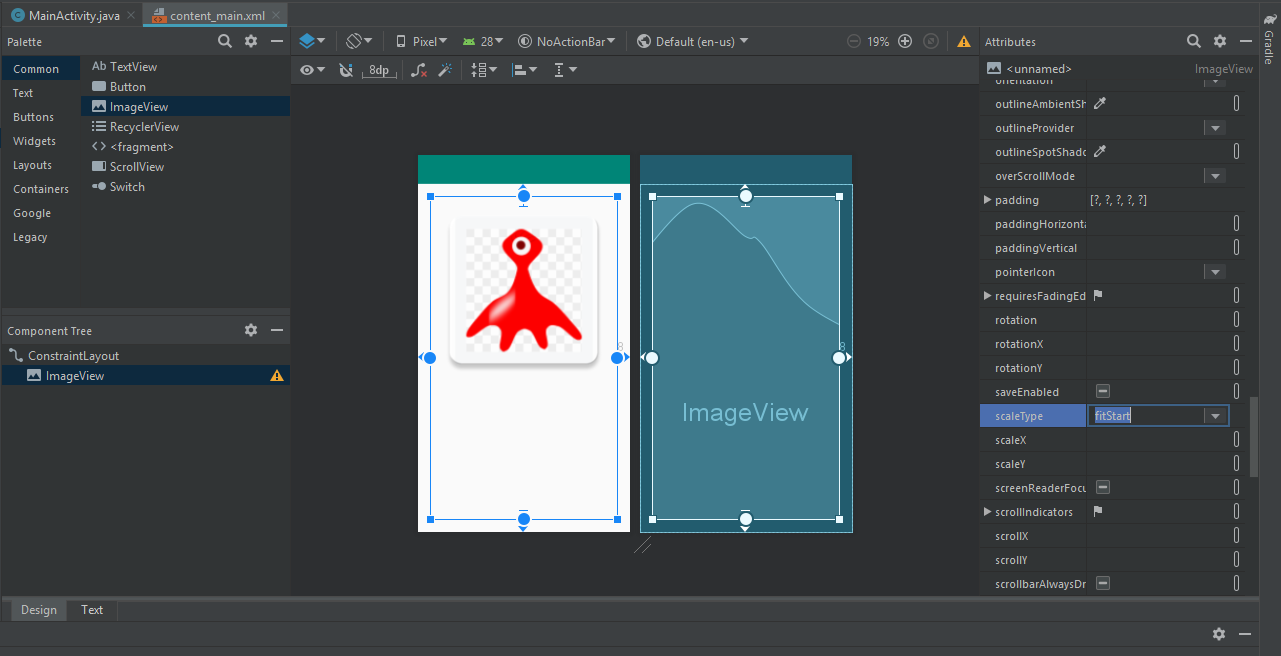
So this is an image resource, just like with string resources, dimension resources, and all the others, there's a prefix, “@drawable” matches the name of the directory “drawable.” The rest of the expression matches the first part of the file name. You don't need to include the file extension as Android figures that out for you. Look at this in design view, and now your displaying your image.



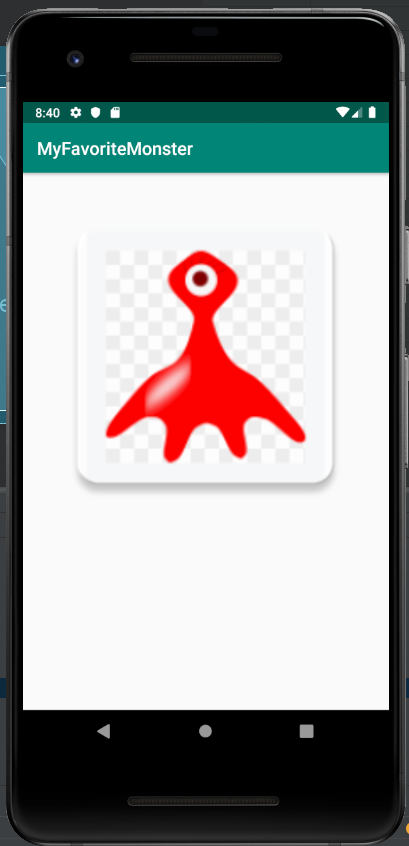
Run this in your AVD to verify that the image shows up.



There are a number of things you can do to control the size of the image. You’re going to constrain this image view to the bottom of the screen and that initially means place the image view in the vertical center. In text mode change the height to zero DP and that means stretch the image view vertically, but again it doesn't change the image itself. You can modify how the image is displayed within the image view with a scale type attribute. Search for that attribute in the attributes window.

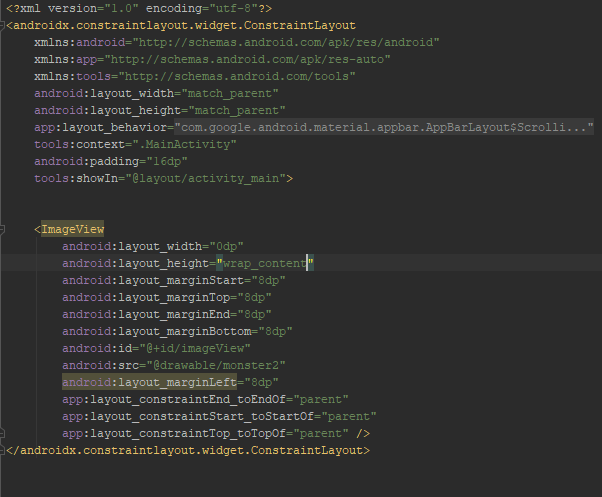


Next you’re going to try a couple of different settings. In an image view that is taller than it is wide, “fitStart” means push the image to the top. If it were wider than it is tall, it would mean push it to the start, which in a left to right language means to the left, and right to left means the other way. So “fitStart” in this context means push it to the top, “fitEnd” means push it to the bottom. “fitXY” means stretch the image and Modify its aspect ratio to fill the image view’s available space. “fitCenter” squeezes the image down again, and “centerCrop” means make the image as large as it needs to be but crop it to fit. This example uses “fitCenter.” Run the application again, once again to verify that what you’re seeing in design mode matches what you see on the AVD.

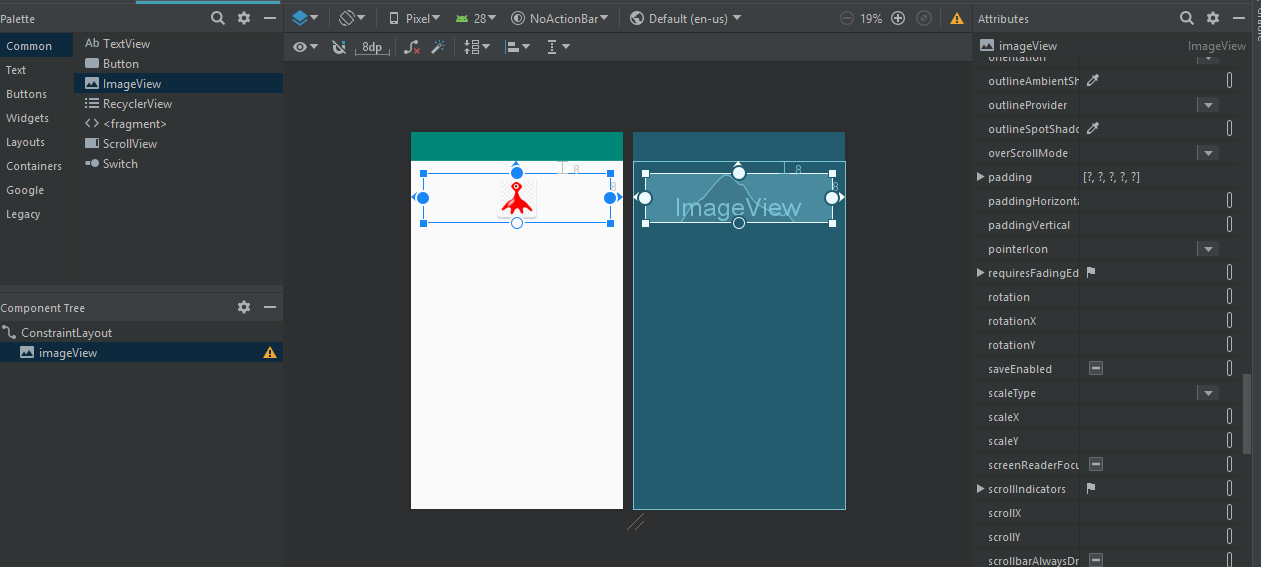


6. Load Image Resources at Runtime

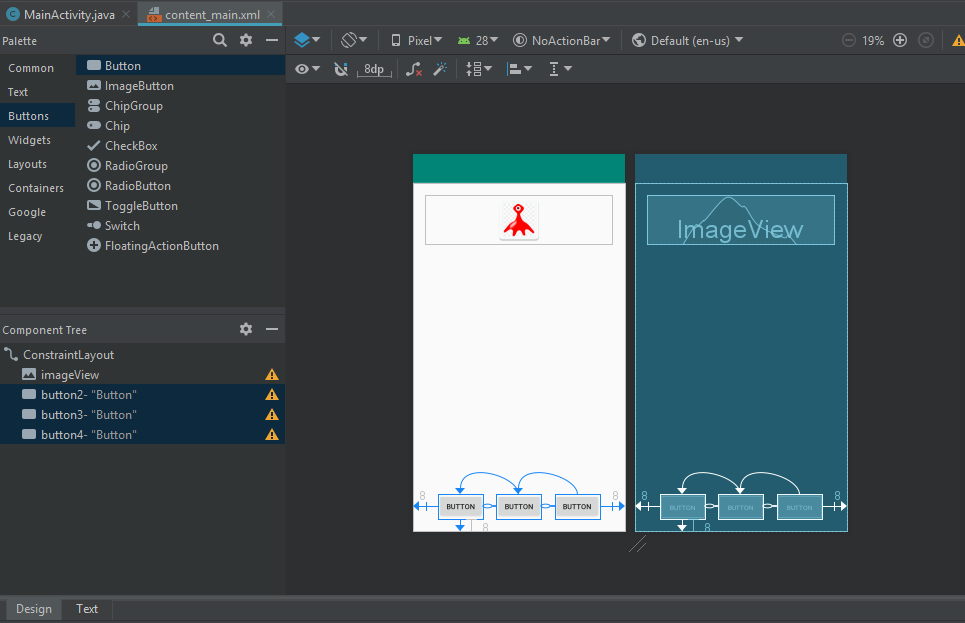
You can control which image is displayed at run time in a few different ways. When you store your images as resources, it takes just one line of code to change from one image to another. Go to text view and remove a couple of attributes here. Remove the margin bottom constraint, bottom to bottom of, and scale type attribute, and then go to the layout height and change it back to wrap content. Make your content\_main.xml like the following:



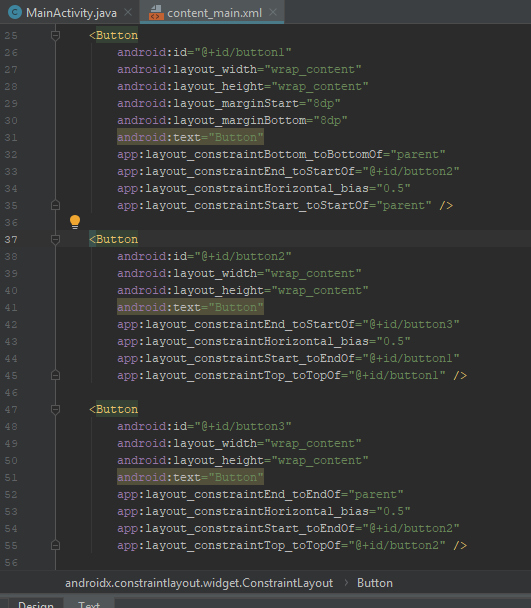
Look at this in design view, and now your image view size is adjusting automatically to whichever image you choose to display. That will become important because you may be working with images of different aspect ratios, one that's tall, one that's square, and another one that's tall, and you want the image view to adjust automatically.



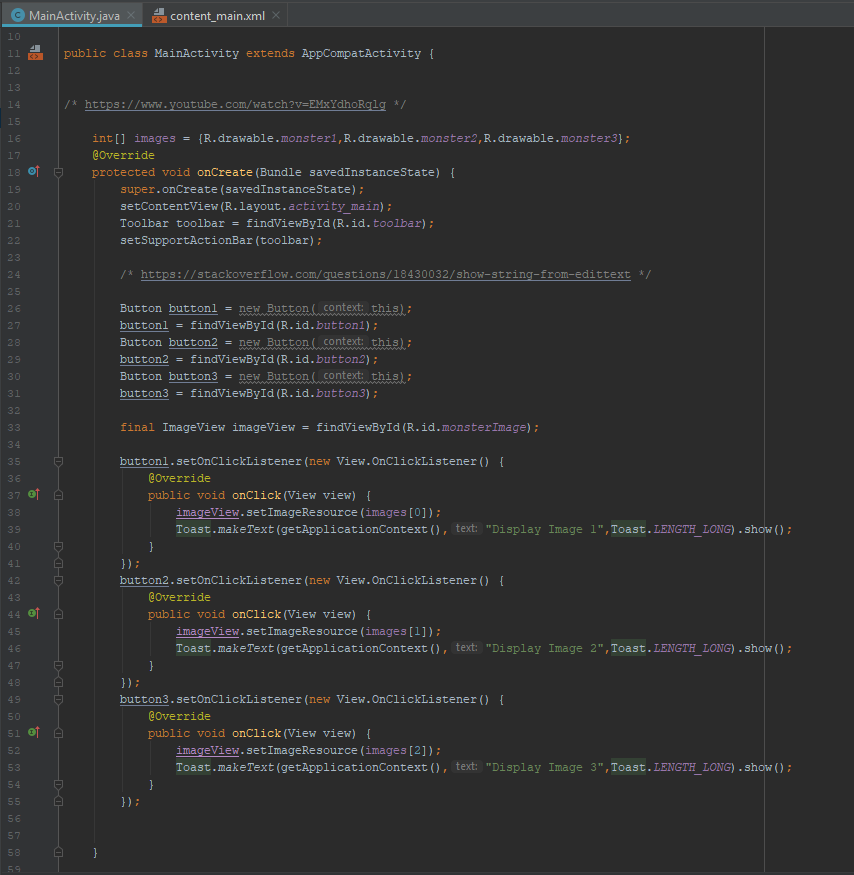
Add a few buttons into this layout. It doesn't matter initially where you drop them, then you’ll choose the first button and constrain it to the left side and to the bottom. Constrain the second button to the right, and choose all three buttons and align their top edges, and then distribute them horizontally using a chain.



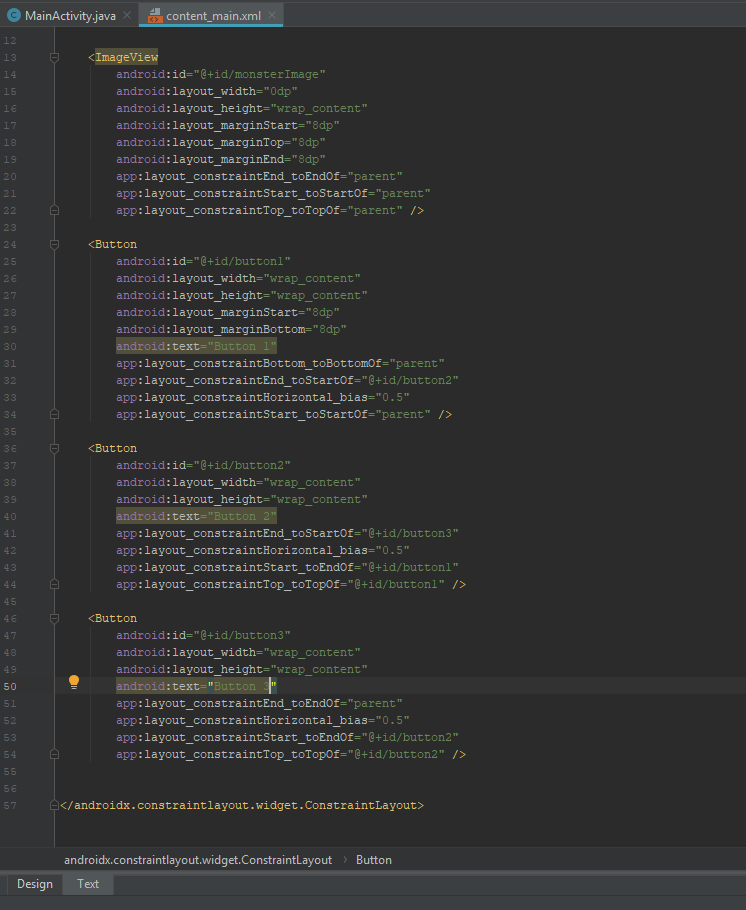
Now in text mode, set the IDs of the three buttons. Reset any constraints that are pointing to the first button to match the IDs.



Now you’re ready to add event listeners for the three buttons. Do that in your MainActivity class. In the “onCreate” function, enter the following code:



Go back to your content\_ main.xml file to get the ID of the image view. For this particular example, the final edit of the file is the following:



So now run the application on your AVD and when you touch the first button, you’ll change the image that's displayed.