**CSCI 322/522** **10. Manual XML Layout Design, Constraint Sets and Apply Changes**

- While the design of layouts using the Android Studio Layout Editor tool greatly improves productivity, it is still possible to create XML layouts by manually editing the underlying XML.

- This chapter will introduce the basics of the Android XML layout file format.

- The structure of an XML layout file is actually quite straightforward and follows the hierarchical approach of the view tree.

- The first line of an XML resource file should ideally include the following standard declaration:

```
<?xml version="1.0" encoding="utf-8"?>
```

- This declaration should be followed by the root element of the layout, typically a container view such as a layout manager.

- This is represented by both opening and closing tags and any properties that need to be set on the view.

- The following XML, for example, declares a ConstraintLayout view as the root element, assigns the ID activity_main and sets match_parent attributes such that it fills all the available space of the device display:

```
<?xml version="1.0" encoding="utf-8"?>
  <androidx.constraintlayout.widget.ConstraintLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:paddingLeft="16dp"
  android:paddingRight="16dp"
  android:paddingTop="16dp"
  android:paddingBottom="16dp"
  tools:context=".MainActivity">
</androidx.constraintlayout.widget.ConstraintLayout>
```

- Note that in the above example the layout element is also configured with padding on each side of 16dp (density independent pixels).

- Any specification of spacing in an Android layout must be specified using one of the following units of measurement:

  in – Inches

  mm – Millimeters

  pt – Points (1/72 of an inch)

dp – Density-independent pixels. An abstract unit of measurement based on the physical density of the device display relative to a 160dpi display baseline.

sp – Scale-independent pixels. Similar to dp but scaled based on the user's font preference.

px – Actual screen pixels. Use is not recommended since different displays will have different pixels per inch. Use dp in preference to this unit.

- Any children that need to be added to the ConstraintLayout parent must be nested within the opening and closing tags. In the following example a Button widget has been added as a child of the ConstraintLayout:

```
<?xml version="1.0" encoding="utf-8"?>
  <androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/activity_main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="16dp"
    android:paddingRight="16dp"
    android:paddingTop="16dp"
    android:paddingBottom="16dp"
    tools:context=".MainActivity">

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

  </androidx.constraintlayout.widget.ConstraintLayout>
```

- As currently implemented, the button has no constraint connections.

- At runtime, therefore, the button will appear in the top left-hand corner of the screen (though indented 16dp by the padding assigned to the parent layout).

- If opposing constraints are added to the sides of the button, however, it will appear centered within the layout:

```
<Button
  android:text="@string/button_string"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:id="@+id/button"
  app:layout_constraintBottom_toBottomOf="parent"
  app:layout_constraintEnd_toEndOf="parent"
  app:layout_constraintStart_toStartOf="parent"
  app:layout_constraintTop_toTopOf="parent" />
```

- Note that each of the constraints is attached to the element named activity_main which is, in this case, the parent ConstraintLayout instance.

- To add a second widget to the layout, simply embed it within the body of the ConstraintLayout element.

- The following modification, for example, adds a TextView widget to the layout:

```
<?xml version="1.0" encoding="utf-8"?>
  <androidx.constraintlayout.widget.ConstraintLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  xmlns:tools="http://schemas.android.com/tools"
  android:id="@+id/activity_main"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:paddingLeft="16dp"
  android:paddingTop="16dp"
  android:paddingRight="16dp"
  android:paddingBottom="16dp"
  tools:context=".MainActivity">

  <Button
    android:text="@string/button_string"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/button"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

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

</androidx.constraintlayout.widget.ConstraintLayout>
```

- Once again, the absence of constraints on the newly added TextView will cause it to appear in the top left-hand corner of the layout at runtime.

- The following modifications add opposing constraints connected to the parent layout to center the widget horizontally, together with a constraint connecting the bottom of the TextView to the top of the button:

```
<TextView
  android:text="@string/text_string"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:id="@+id/textView"
  android:layout_marginBottom="77dp"
```

```
app:layout_constraintBottom_toTopOf="@+id/button"
app:layout_constraintEnd_toEndOf="parent"
app:layout_constraintStart_toStartOf="parent" />
```

- Also, note that the Button and TextView views have a number of attributes declared.
- Both views have been assigned IDs and configured to display text strings represented by string resources named button_string and text_string respectively.
- Additionally, the wrap_content height and width properties have been declared on both objects so that they are sized to accommodate the content (in this case the text referenced by the string resource value).
- Viewed from within the Preview panel of the Layout Editor in Code mode, the above layout will be rendered as shown in Figure 10-1:

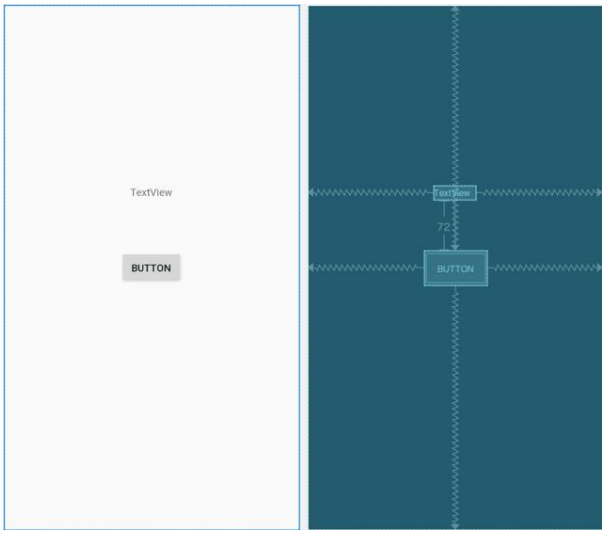


Figure 10-1

**Manual XML vs. Visual Layout Design**

- When to write XML manually as opposed to using the Layout Editor tool in design mode is a matter of personal preference.
- There are, however, advantages to using design mode.
- First, design mode will generally be quicker given that it avoids the necessity to type lines of XML.
- Additionally, design mode avoids the need to learn the intricacies of the various property values of the Android SDK view classes.
- Rather than continually refer to the Android documentation to find the correct keywords and values, most properties can be located by referring to the Attributes panel.
- All the advantages of design mode aside, it is important to keep in mind that the two approaches to user interface design are in no way mutually exclusive.

- As an application developer, it is quite likely that you will end up creating user interfaces within design mode while performing fine-tuning and layout tweaks of the design by directly editing the generated XML resources.

- Both views of the interface design are, after all, displayed side by side within the Android Studio environment making it easy to work seamlessly on both the XML and the visual layout.

- Up until this point in the book, all user interface design tasks have been performed using the Android Studio Layout Editor tool, either in text or design mode.

- An alternative to writing XML resource files or using the Android Studio Layout Editor is to write Java code to directly create, configure and manipulate the view objects that comprise the user interface of an Android activity.

- Within the context of this chapter, we will explore some of the advantages and disadvantages of writing Java code to create a user interface before describing some of the key concepts such as view properties and the creation and management of layout constraints.

- In the next chapter, an example project will be created and used to demonstrate some of the typical steps involved in this approach to Android user interface creation.

**Java Code vs. XML Layout Files**

- There are a number of key advantages to using XML resource files to design a user interface as opposed to writing Java code.

- In fact, Google goes to considerable lengths in the Android documentation to extol the virtues of XML resources over Java code.

- As discussed in the previous chapter, one key advantage to the XML approach includes the ability to use the Android Studio Layout Editor tool, which, itself, generates XML resources.

- A second advantage is that once an application has been created, changes to user interface screens can be made by simply modifying the XML file, thereby avoiding the necessity to recompile the application.
- Also, even when hand writing XML layouts, it is possible to get instant feedback on the appearance of the user interface using the preview feature of the Android Studio Layout Editor tool.

- In order to test the appearance of a Java created user interface the developer will, inevitably, repeatedly cycle through a loop of writing code, compiling and testing in order to complete the design work.

- In terms of the strengths of the Java coding approach to layout creation, perhaps the most significant advantage that Java has over XML resource files comes into play when dealing with dynamic user interfaces.

- XML resource files are inherently most useful when defining static layouts, in other words layouts that are unlikely to change significantly from one invocation of an activity to the next.

- Java code, on the other hand, is ideal for creating user interfaces dynamically at run-time.

- This is particularly useful in situations where the user interface may appear differently each time the activity executes subject to external factors.

- A knowledge of working with user interface components in Java code can also be useful when dynamic changes to a static XML resource based layout need to be performed in real-time as the activity is running.

- Finally, some developers simply prefer to write Java code than to use layout tools and XML, regardless of the advantages offered by the latter approaches.

**Creating Views**

- As previously established, the Android SDK includes a toolbox of view classes designed to meet most of the basic user interface design needs.

- The creation of a view in Java is simply a matter of creating instances of these classes, passing through as an argument a reference to the activity with which that view is to be associated.

- The first view (typically a container view to which additional child views can be added) is displayed to the user via a call to the setContentView() method of the activity.

- Additional views may be added to the root view via callsto the object's addView() method.

- When working with Java code to manipulate views contained in XML layout resource files, it is necessary to obtain the ID of the view.

- The same rule holds true for views created in Java.

- As such, it is necessary to assign an ID to any view for which certain types of access will be required in subsequent Java code.

- This is achieved via a call to the setId() method of the view object in question.

- In later code, the ID for a view may be obtained via the object's getId() method.

**View Attributes**

- Each view class has associated with it a range of attributes.

- These property settings are set directly on the view instances and generally define how the view object will appear or behave.

- Examples of attributes are the text that appears on a Button object, or the background color of a ConstraintLayout view.

- Each view class within the Android SDK has a pre-defined set of methods that allow the user to set and get these property values.

- The Button class, for example, has a setText() method which can be called from within Java code to set the text displayed on the button to a specific string value.

- The background color of a ConstraintLayout object, on the other hand, can be set with a call to the object's setBackgroundColor() method.

**Constraint Sets**

- While property settings are internal to view objects and dictate how a view appears and behaves, constraint sets are used to control how a view appears relative to its parent view and other sibling views.

- Every ConstraintLayout instance has associated with it a set of constraints that define how its child views are positioned and constrained.

- The key to working with constraint sets in Java code is the ConstraintSet class.

- This class contains a range of methods that allow tasks such as creating, configuring and applying constraints to a ConstraintLayout instance.

- In addition, the current constraints for a ConstraintLayout instance may be copied into a ConstraintSet object and used to apply the same constraints to other layouts (with or without modifications).

- A ConstraintSet instance is created just like any other Java object:

```
ConstraintSet set = new ConstraintSet();
```

- Once a constraint set has been created, methods can be called on the instance to perform a wide range of tasks.

**Establishing Connections**

- The connect() method of the ConstraintSet class is used to establish constraint connections between views.

- The following code configures a constraint set in which the left-hand side of a Button view is connected to the right- hand side of an EditText view with a margin of 70dp:

```
set.connect(button1.getId(), ConstraintSet.LEFT, editText1.getId(),
            ConstraintSet.RIGHT, 70);
```

**Applying Constraints to a Layout**

- Once the constraint set is configured, it must be applied to a ConstraintLayout instance before it will take effect.

- A constraint set is applied via a call to the applyTo() method, passing through a reference to the layout object to which the settings are to be applied:

```
set.applyTo(myLayout);
```

(continued)

**Parent Constraint Connections**

- Connections may also be established between a child view and its parent ConstraintLayout by referencing the ConstraintSet.PARENT_ID constant.

- In the following example, the constraint set is configured to connect the top edge of a Button view to the top of the parent layout with a margin of 100dp:

```
set.connect(button1.getId(), ConstraintSet.TOP,
            ConstraintSet.PARENT_ID, ConstraintSet.TOP, 100);
```

**Sizing Constraints**

- A number of methods are available for controlling the sizing behavior of views.

- The following code, for example, sets the horizontal size of a Button view to wrap_content and the vertical size of an ImageView instance to a maximum of 250dp:

```
set.constrainWidth(button1.getId(), ConstraintSet.WRAP_CONTENT);
set.constrainMaxHeight(imageView1.getId(), 250);
```

**Constraint Bias**

- As outlined in a previous chapter, when a view has opposing constraints it is centered along the axis of the constraints (i.e. horizontally or vertically).

- This centering can be adjusted by applying a bias along the particular axis of constraint.

- When using the Android Studio Layout Editor, this is achieved using the controls in the Attributes tool window.

- When working with a constraint set, however, bias can be added using the setHorizontalBias() and setVerticalBias() methods, referencing the view ID and the bias as a floating point value between 0 and 1.

- The following code, for example, constrains the left and right-hand sides of a Button to the corresponding sides of the parent layout before applying a 25% horizontal bias:

```
set.connect(button1.getId(), ConstraintSet.LEFT,
ConstraintSet.PARENT_ID, ConstraintSet.LEFT, 0);
set.connect(button1.getId(), ConstraintSet.RIGHT,
ConstraintSet.PARENT_ID, ConstraintSet.RIGHT, 0);
set.setHorizontalBias(button1.getId(), 0.25f);
```

**Alignment Constraints**

- Alignments may also be applied using a constraint set.

- The full set of alignment options available with the Android Studio Layout Editor may also be configured using a constraint set via the centerVertically() and centerHorizontally() methods, both of which take a variety of arguments depending on the alignment being configured.

- In addition, the center() method may be used to center a view between two other views.

- In the code below, button2 is positioned so that it is aligned horizontally with button1:

```
set.centerHorizontally(button2.getId(), button1.getId());
```

**Copying and Applying Constraint Sets**

- The current constraint set for a ConstraintLayout instance may be copied into a constraint set object using the clone() method.

- The following line of code, for example, copies the constraint settings from a ConstraintLayout instance named myLayout into a constraint set object:

```
set.clone(myLayout);
```

- Once copied, the constraint set may be applied directly to another layout or, as in the following example, modified before being applied to the second layout:

```
ConstraintSet set = new ConstraintSet();
set.clone(myLayout);
set.constrainWidth(button1.getId(), ConstraintSet.WRAP_CONTENT);
set.applyTo(mySecondLayout);
```

**ConstraintLayout Chains**

- Vertical and horizontal chains may also be created within a constraint set using the createHorizontalChain() and createVerticalChain() methods. The syntax for using these methods is as follows:

```
createHorizontalChain(int leftId, int leftSide, int rightId, int rightSide,
                      int[] chainIds, float[] weights, int style);
```

- Based on the above syntax, the following example creates a horizontal spread chain that starts with button1 and ends with button4.

- In between these views are button2 and button3 with weighting set to zero for both:

```
int[] chainViews = {button2.getId(), button3.getId()};
float[] chainWeights = {0, 0};

set.createHorizontalChain(button1.getId(), ConstraintSet.LEFT,
                          button4.getId(), ConstraintSet.RIGHT, chainViews,
                          chainWeights, ConstraintSet.CHAIN_SPREAD);
```

- A view can be removed from a chain by passing the ID of the view to be removed through to either the removeFromHorizontalChain() or removeFromVerticalChain() methods.

- A view may be added to an existing chain using either the addToHorizontalChain() or addToVerticalChain() methods.

- In both cases the methods take as arguments the IDs of the views between which the new view is to be inserted as follows:

```
set.addToHorizontalChain(newViewId, leftViewId, rightViewId);
```

**Guidelines**

- Guidelines are added to a constraint set using the create() method and then positioned using the setGuidelineBegin(), setGuidelineEnd() or setGuidelinePercent() methods.

- In the following code, a vertical guideline is created and positioned 50% across the width of the parent layout. The left side of a button view is then connected to the guideline with no margin:

```
set.create(R.id.myGuideline, ConstraintSet.VERTICAL_GUIDELINE);
set.setGuidelinePercent(R.id.myGuideline, 0.5f);
set.connect(button.getId(), ConstraintSet.LEFT, R.id.myGuideline,
            ConstraintSet.RIGHT, 0);
set.applyTo(layout);
```

**Removing Constraints**

- A constraint may be removed from a view in a constraint set using the clear() method, passing through as arguments the view ID and the anchor point for which the constraint is to be removed:

```
set.clear(button.getId(), ConstraintSet.LEFT);
```

- Similarly, all of the constraints on a view may be removed in a single step by referencing only the view in the clear() method call:

```
set.clear(button.getId());
```

**Scaling**

- The scale of a view within a layout may be adjusted using the ConstraintSet setScaleX() and setScaleY() methods which take as arguments the view on which the operation is to be performed together with a float value indicating the scale. In the following code, a button object is scaled to twice its original width and half the height:

```
set.setScaleX(myButton.getId(), 2f); set.setScaleY(myButton.getId(), 0.5f);
```

(continued)

**Rotation**

- A view may be rotated on either the X or Y axis using the setRotationX() and setRotationY() methods respectively both of which must be passed the ID of the view to be rotated and a float value representing the degree of rotation to be performed.

- The pivot point on which the rotation is to take place may be defined via a call to the setTransformPivot(), setTransformPivotX() and setTransformPivotY() methods.

- The following code rotates a button view 30 degrees on the Y axis using a pivot point located at point 500, 500:

```
set.setTransformPivot(button.getId(), 500, 500);
set.setRotationY(button.getId(), 30);
set.applyTo(layout);
```

- Having covered the theory of constraint sets and user interface creation from within Java code, the next chapter will work through the creation of an example application with the objective of putting this theory into practice.

- For more details on the ConstraintSet class, refer to the reference guide at the following URL:

```
https://developer.android.com/reference/android/support/constraint/ConstraintSet.html
```

- This section will take the concepts introduced earlier in this chapter and put them into practice through the creation of an example layout created entirely in Java code and without using the Android Studio Layout Editor tool.

- Launch Android Studio and select the Start a new Android Studio project option from the quick start menu in the welcome screen and, within the resulting new project dialog, choose the Empty Activity template before clicking on the Next button.

- Enter JavaLayout into the Name field and specify `edu.niu.your_Z-ID.javalayout` as the package name.

- Before clicking on the Finish button, change the Minimum API level setting to API 26: Android 8.0 (Oreo) and the Language menu to Java.

- Once the project has been created, the MainActivity.java file should automatically load into the editing panel.

- As we have come to expect, Android Studio has created a template activity and overridden the onCreate() method, providing an ideal location for Java code to be added to create a user interface.

**Adding Views to an Activity**

- The onCreate() method is currently designed to use a resource layout file for the user interface. Begin, therefore, by deleting this line from the method:

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

- The next modification is to add a ConstraintLayout object with a single Button view child to the activity.

- This involves the creation of new instances of the ConstraintLayout and Button classes.

- The Button view then needs to be added as a child to the ConstraintLayout view which, in turn, is displayed via a call to the setContentView() method of the activity instance:

```
package edu.niu.your_Z-ID.javalayout;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import androidx.constraintlayout.widget.ConstraintLayout;
import android.widget.Button;
import android.widget.EditText;

public class MainActivity extends AppCompatActivity
{

  @Override
  protected void onCreate(Bundle savedInstanceState)
  {
    super.onCreate(savedInstanceState);
    configureLayout();
  }

  private void configureLayout()
  {
    Button myButton = new Button(this);
    ConstraintLayout myLayout = new ConstraintLayout(this);
    myLayout.addView(myButton);
    setContentView(myLayout);
  }
}
```

- When new instances of user interface objects are created in this way, the constructor methods must be passed the context within which the object is being created which, in this case, is the current activity.

- Since the above code resides within the activity class, the context is simply referenced by the standard this keyword:

```
Button myButton = new Button(this);
```

- Once the above additions have been made, compile and run the application (either on a physical device or an emulator). Once launched, the visible result will be a button containing no text appearing in the top left-hand corner of the ConstraintLayout view as shown in Figure 10-2:

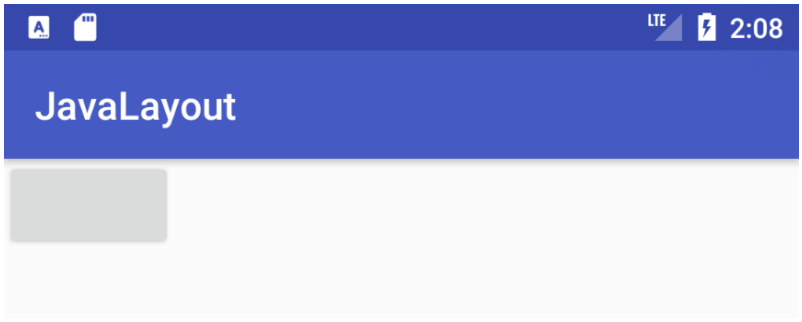


Figure 10-2

**Setting View Attributes**

- For the purposes of this exercise, we need the background of the ConstraintLayout view to be blue and the Button view to display text that reads “Press Me” on a yellow background.

- Both of these tasks can be achieved by setting attributes on the views in the Java code as outlined in the following code fragment.

- In order to allow the text on the button to be easily translated to other languages it will be added as a String resource.

- Within the Project tool window, locate the **app** > **res** > **values** > **strings.xml** file and modify it to add a resource value for the “Press Me” string:

```
<resources>
  <string name="app_name">JavaLayout</string>
  <string name="press_me">Press Me</string>
</resources>
```

- Although this is the recommended way to handle strings that are directly referenced in code, to avoid repetition of this step throughout the remainder of the book, many subsequent code samples will directly enter strings into the code.

- Once the string is stored as a resource it can be accessed from within code as follows:

```
getString(R.string.press_me);
```

- With the string resource created, add code to the configureLayout() method to set the button text and color attributes:

```
.
.
import android.graphics.Color;

public class MainActivity extends AppCompatActivity
```

```
{
  private void configureLayout()
  {
    Button myButton = new Button(this);
    myButton.setText(getString(R.string.press_me));
    myButton.setBackgroundColor(Color.YELLOW);

    ConstraintLayout myLayout = new ConstraintLayout(this);
    myLayout.setBackgroundColor(Color.BLUE);

    myLayout.addView(myButton); setContentView(myLayout);
  }
.
.
```

- When the application is now compiled and run, the layout will reflect the property settings such that the layout will appear with a blue background and the button will display the assigned text on a yellow background.

**Creating View IDs**

- When the layout is complete it will consist of a Button and an EditText view.

- Before these views can be referenced within the methods of the ConstraintSet class, they must be assigned unique view IDs.

- The first step in this process is to create a new resource file containing these ID values.

- Right click on the **app** > **res** > **values** folder, select the **New** > **Values** resource file menu option and name the new resource file id.xml. With the resource file created, edit it so that it reads as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <item name="myButton" type="id" />
  <item name="myEditText" type="id" />
</resources>
```

- At this point in the tutorial, only the Button has been created, so edit the configureLayout() method to assign the corresponding ID to the object:

```
private void configureLayout()
{
  Button myButton = new Button(this);
  myButton.setText(getString(R.string.press_me));
  myButton.setBackgroundColor(Color.YELLOW);
  myButton.setId(R.id.myButton);
  .
  .
```

(continued)

**Configuring the Constraint Set**

- In the absence of any constraints, the ConstraintLayout view has placed the Button view in the top left corner of the display.

- In order to instruct the layout view to place the button in a different location, in this case centered both horizontally and vertically, it will be necessary to create a ConstraintSet instance, initialize it with the appropriate settings and apply it to the parent layout.

- For this example, the button needs to be configured so that the width and height are constrained to the size of the text it is displaying and the view centered within the parent layout.

- Edit the configureLayout() method once more to make these changes:

```
.
.
import androidx.constraintlayout.widget.ConstraintSet;
.
.
private void configureLayout()
{
  Button myButton = new Button(this);
  myButton.setText(getString(R.string.press_me));
  myButton.setBackgroundColor(Color.YELLOW); myButton.setId(R.id.myButton);

  ConstraintLayout myLayout = new ConstraintLayout(this);
  myLayout.setBackgroundColor(Color.BLUE);

  myLayout.addView(myButton); setContentView(myLayout);

  ConstraintSet set = new ConstraintSet();

  set.constrainHeight(myButton.getId(), ConstraintSet.WRAP_CONTENT);
  set.constrainWidth(myButton.getId(), ConstraintSet.WRAP_CONTENT);

  set.connect(myButton.getId(), ConstraintSet.START,
              ConstraintSet.PARENT_ID, ConstraintSet.START, 0);
  set.connect(myButton.getId(), ConstraintSet.END, ConstraintSet.PARENT_ID,
              ConstraintSet.END, 0);
  set.connect(myButton.getId(), ConstraintSet.TOP, ConstraintSet.PARENT_ID,
              ConstraintSet.TOP, 0);
  set.connect(myButton.getId(), ConstraintSet.BOTTOM,
              ConstraintSet.PARENT_ID,ConstraintSet.BOTTOM, 0);
  set.applyTo(myLayout);
}
```

- With the initial constraints configured, compile and run the application and verify that the Button view now appears in the center of the layout:

(continued)


Figure 10-3

**Adding the EditText View**

- The next item to be added to the layout is the EditText view.
- The first step is to create the EditText object, assign it the ID as declared in the id.xml resource file and add it to the layout.
- The code changes to achieve these steps now need to be made to the configureLayout() method as follows:

```
private void configureLayout()
{
  Button myButton = new Button(this);
  myButton.setText(getString(R.string.press_me));
  myButton.setBackgroundColor(Color.YELLOW);
  myButton.setId(R.id.myButton);

  EditText myEditText = new EditText(this);
  myEditText.setId(R.id.myEditText);

  ConstraintLayout myLayout = new ConstraintLayout(this);
  myLayout.setBackgroundColor(Color.BLUE);

  myLayout.addView(myButton);
  myLayout.addView(myEditText);

  setContentView(myLayout);
  .
  .
}
```

- The EditText widget is intended to be sized subject to the content it is displaying, centered horizontally within the layout and positioned 70dp above the existing Button view.
- Add code to the configureLayout() method so that it reads as follows:

```
.
.
set.connect(myButton.getId(), ConstraintSet.START,
            ConstraintSet.PARENT_ID,ConstraintSet.START, 0);
set.connect(myButton.getId(), ConstraintSet.END, ConstraintSet.PARENT_ID,
            ConstraintSet.END, 0);
set.connect(myButton.getId(), ConstraintSet.TOP, ConstraintSet.PARENT_ID,
            ConstraintSet.TOP, 0);
set.connect(myButton.getId(), ConstraintSet.BOTTOM,
            ConstraintSet.PARENT_ID, ConstraintSet.BOTTOM, 0);

set.constrainHeight(myEditText.getId(), ConstraintSet.WRAP_CONTENT);
set.constrainWidth(myEditText.getId(), ConstraintSet.WRAP_CONTENT);

set.connect(myEditText.getId(), ConstraintSet.START,
            ConstraintSet.PARENT_ID, ConstraintSet.START, 0);
set.connect(myEditText.getId(), ConstraintSet.END, ConstraintSet.PARENT_ID,
            ConstraintSet.END, 0);
set.connect(myEditText.getId(), ConstraintSet.BOTTOM, myButton.getId(),
            ConstraintSet.TOP, 70);

set.applyTo(myLayout);
```

- A test run of the application should show the EditText field centered above the button with a margin of 70dp.

**Converting Density Independent Pixels (dp) to Pixels (px)**

- The next task in this exercise is to set the width of the EditText view to 200dp.

- As outlined in a previous chapter, it is better to use density independent pixels (dp) rather than pixels (px).

- In order to set a position using dp it is necessary to convert a dp value to a px value at runtime, taking into consideration the density of the device display.

- In order, therefore, to set the width of the EditText view to 200dp, the following code needs to be added to the class:

```
package edu.niu.your_Z-ID.javalayout;
.
.
import android.content.res.Resources;
import android.util.TypedValue;

public class MainActivity extends AppCompatActivity

private int convertToPx(int value)
{
  Resources r = getResources();
```

```
    int px = (int) TypedValue.applyDimension(
      TypedValue.COMPLEX_UNIT_DIP, value, r.getDisplayMetrics());
    return px;
  }

  private void configureLayout()
  {
    Button myButton = new Button(this);
    myButton.setText(getString(R.string.press_me));
    myButton.setBackgroundColor(Color.YELLOW);
    myButton.setId(R.id.myButton);

    EditText myEditText = new EditText(this);
    myEditText.setId(R.id.myEditText);

    int px = convertToPx(200);
    myEditText.setWidth(px);
    .
    .
  }
```

- Compile and run the application one more time and note that the width of the EditText view has changed as illustrated in Figure 10-4:

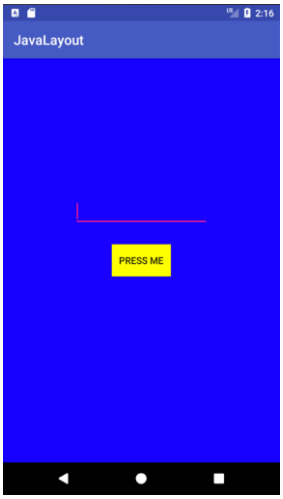


Figure 10-4

- Now that some of the basic concepts of Android development using Android Studio have been covered, now is a good time to introduce the ***Android Studio Apply Changes*** feature.

- As all experienced developers know, every second spent waiting for an app to compile and run is time better spent writing and refining code.

**Introducing Apply Changes**

- In early versions of Android Studio, each time a change to a project needed to be tested Android Studio would recompile the code, convert it to Dex format, generate the APK package file and install it on the device or emulator.

- Having performed these steps the app would finally be launched ready for testing.
- Even on a fast development system this is a process that takes a considerable amount of time to complete.

- It is not uncommon for it to take a minute or more for this process to complete for a large application.

- Apply Changes, in contrast, allows many code and resource changes within a project to be reflected nearly instantaneously within the app while it is already running on a device or emulator session.

- Consider, for the purposes of an example, an app being developed in Android Studio which has already been launched on a device or emulator.

- If changes are made to resource settings or the code within a method, Apply Changes will push the updated code and resources to the running app and dynamically "swap" the changes.

- The changes are then reflected in the running app without the need to build, deploy and relaunch the entire app.

- In many cases, this allows changes to be tested in a fraction of the time it would take without Apply Changes.

**Understanding Apply Changes Options**

- Android Studio provides three options in terms of applying changes to a running app in the form of Run App, Apply Changes and Restart Activity and Apply Code Changes.

- These options can be summarized as follows:

  - Run App - Stops the currently running app and restarts it. If no changes have been made to the project since it was last launched, this option will simply restart the app.

    If, on the other hand, changes have been made to the project, Android Studio will rebuild and reinstall the app onto the device or emulator before launching it.

  - Apply Code Changes - This option can be used when the only changes made to a project involve modifications to the body of existing methods or when a new class has been added.

    When selected, the changes will be applied to the running app without the need to restart either the app or the currently running activity.

    This mode cannot, however, be used when changes have been made to any project resources such as a layout file.

    Other restrictions include the addition or removal of methods, changes to a method signature, renaming of classes and other structural code changes.

    It is also not possible to use this option when changes have been made to the project manifest.

- Apply Changes and Restart Activity - When selected, this mode will dynamically apply any code or resource changes made within the project and restart the activity without reinstalling or restarting the app.

  Unlike the Apply Code changes option, this can be used when changes have been made to the code and resources of the project, though the same restrictions involving some structural code changes and manifest modifications apply.

**Using Apply Changes**

- When a project has been loaded into Android Studio, but is not yet running on a device or emulator, it can be launched as usual using either the run (marked A in Figure 10-5) or debug (B) button located in the toolbar:


Figure 10-5

- After the app has launched and is running, the icon on the run button will change to indicate that the app is running and the Apply Changes and Restart Activity and Apply Code Changes buttons will be enabled as indicated in Figure 10-6 below:

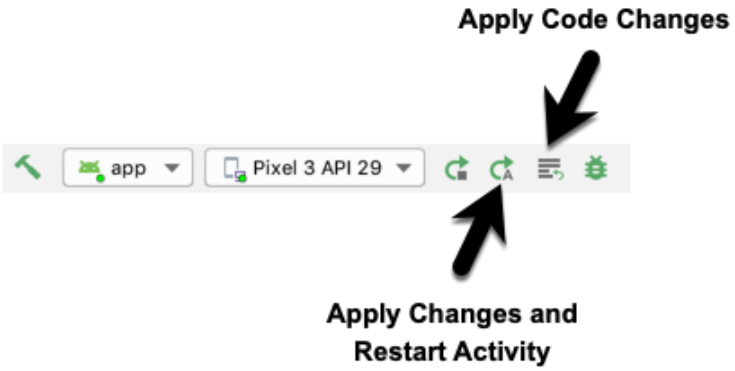


Figure 10-6

- If the changes are unable to be applied when one of the Apply Changes buttons is selected, Android Studio will display a message indicating the failure together with an explanation.

- Figure 10-7, for example, shows the message displayed by Android Studio when the Apply Code Changes option is selected after a change has been made to a resource file:

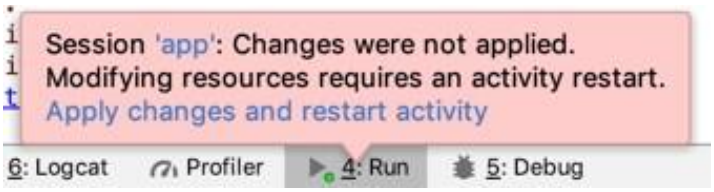


Figure 10-7

- In this situation, the solution is to use the Apply Changes and Restart Activity option (for which a link is provided).

- Similarly, the following message will appear when an attempt to apply changes that involve the addition or removal of a method is made:

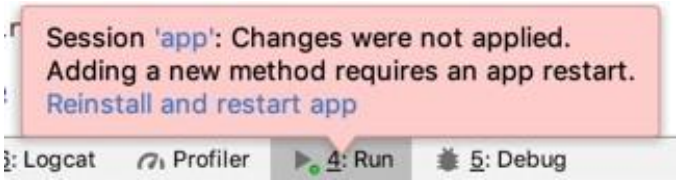


Figure 10-8

- In this case, the only option is to click on the Run App button to reinstall and restart the app.

- As an alternative to manually selecting the correct option in these situations, Android Studio may be configured to automatically fall back to performing a Run App operation.

**Configuring Apply Changes Fallback Settings**

- The Apply Changes fallback settings are located in the Android Studio Preferences window which is displayed by selecting the **File > Settings** menu option (**Android Studio > Preferences** on macOS).

- Within the Preferences dialog, select the Build, Execution, Deployment entry in the left-hand panel followed by Deployment as shown in Figure 10-9:

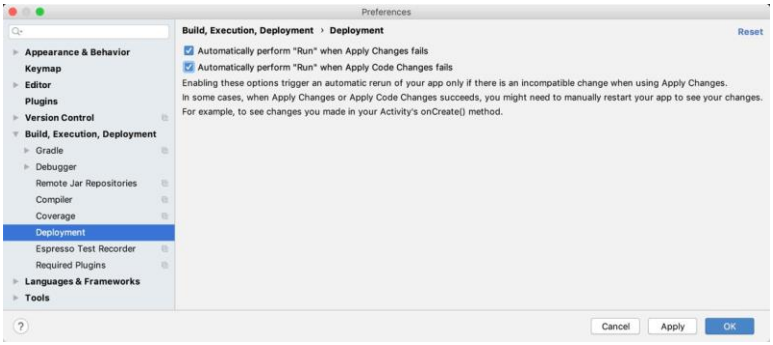


Figure 10-9

- Once the required options have been enabled, click on Apply followed by the OK button to commit the changes and dismiss the dialog.

- After these defaults have been enabled, Android Studio will automatically reinstall and restart the app when necessary.

**An Apply Changes Tutorial**

- Launch Android Studio, select the Start a new Android Studio project quick start option from the welcome screen and, within the resulting new project dialog, choose the Basic Activity template before clicking on the Next button.

- Enter ApplyChanges into the Name field and specify `edu.niu.your_Z-ID.applychanges` as the package name.

- Before clicking on the Finish button, change the Minimum API level setting to API 26: Android 8.0 (Oreo) and the Language menu to Java.

**Using Apply Code Changes**

- Begin by clicking on the run button and selecting a suitable emulator or physical device as the run target.

- After clicking the run button, track the amount of time before the example app appears on the device or emulator.

- Once running, click on the action button (the button displaying an envelope icon located in the lower right- hand corner of the screen).

- Note that a Snackbar instance appears displaying text which reads “Replace with your own action” as shown in Figure 10-10:

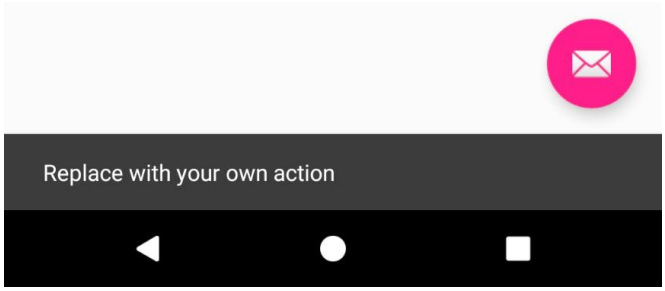


Figure 10-10

- Once the app is running, the Apply Changes buttons should have been enabled indicating that certain project changes can be applied without having to reinstall and restart the app.

- To see this in action, edit the MainActivity. java file, locate the onCreate method and modify the action code so that a different message is displayed when the action button is selected:

```
FloatingActionButton fab = findViewById(R.id.fab);
fab.setOnClickListener(new View.OnClickListener()
{
  @Override
  public void onClick(View view)
  {
    Snackbar.make(view, "Apply Changes is Amazing!", Snackbar.LENGTH_LONG);
    setAction("Action", null).show();
  }
});
```

- With the code change implemented, click on the Apply Code Changes button and note that a message appears within a few seconds indicating the app has been updated.

- Tap the action button and note that the new message is now displayed in the Snackbar.

(continued)

**Using Apply Changes and Restart Activity**

- Any resource change will require use of the Apply Changes and Restart Activity option.
- Within Android Studio select the **app > res > layout > fragment_first.xml** layout file.
- With the Layout Editor tool in Design mode, select the default TextView component and change the text property in the attributes tool window to "Hello Android".
- Make sure that the fallback options outlined in "Configuring Apply Changes Fallback Settings" above are turned off before clicking on the Apply Code Changes button.
- Note that the request fails because this change involves project resources.
- Click on the Apply Changes and Restart Activity button and verify that the activity restarts and displays the new text on the TextView widget.

**Using Run App**

- As previously described, the addition or removal of a method requires the complete re-installation and restart of the running app.
- To experience this, edit the MainActivity.java file and add a new method after the onCreate method as follows:

```
public void demoMethod()
{
}
```

- Clicking on either of the two Apply Changes buttons will result in the request failing.
- The only way to run the app after such a change is to click on the Run App button.

**Summary**

- The Android Studio Layout Editor tool provides a visually intuitive method for designing user interfaces.
- Using a drag and drop paradigm combined with a set of property editors, the tool provides considerable productivity benefits to the application developer.
- User interface designs may also be implemented by manually writing the XML layout resource files, the format of which is well structured and easily understood.
- The fact that the Layout Editor tool generates XML resource files means that these two approaches to interface design can be combined to provide a "best of both worlds" approach to user interface development.

- As an alternative to writing XML layout resource files or using the Android Studio Layout Editor tool, Android user interfaces may also be dynamically created in Java code.

- Creating layouts in Java code consists of creating instances of view classes and setting attributes on those objects to define required appearance and behavior.

- How a view is positioned and sized relative to its ConstraintLayout parent view and any sibling views is defined through the use of constraint sets.

- A constraint set is represented by an instance of the ConstraintSet class which, once created, can be configured using a wide range of method calls to perform tasks such as establishing constraint connections, controlling view sizing behavior and creating chains.

- The example activity created in this chapter has, of course, created a similar user interface (the change in background color and view type notwithstanding) as that created in the earlier "Manual XML Layout Design in Android Studio" chapter.

- If nothing else, this chapter should have provided an appreciation of the level to which the Android Studio Layout Editor tool and XML resources shield the developer from many of the complexities of creating Android user interface layouts.

- There are, however, instances where it makes sense to create a user interface in Java.

- This approach is most useful, for example, when creating dynamic user interface layouts.

- Apply Changes is a feature of Android Studio designed to significantly accelerate the code, build and run cycle performed when developing an app.

- The Apply Changes feature is able to push updates to the running application, in many cases without the need to re-install or even restart the app.

- Apply Changes provides a number of different levels of support depending on the nature of the modification being applied to the project.