

- As discussed previously, Android provides a number of layout managers for the purpose of designing user interfaces.
- With Android 7, Google introduced a new layout that is intended to address many of the shortcomings of the older layout managers.
- This new layout, called ConstraintLayout, combines a simple, expressive and flexible layout system with powerful features built into the Android Studio Layout Editor tool to ease the creation of responsive user interface layouts that adapt automatically to different screen sizes and changes in device orientation.
- This chapter will outline the basic concepts of ConstraintLayout while the next chapter will provide a detailed overview of constraint-based layouts.
- It will also outline how they can be created using ConstraintLayout within the Android Studio Layout Editor tool.

How ConstraintLayout Works

- In common with all other layouts, ConstraintLayout is responsible for managing the positioning and sizing behavior of the visual components (also referred to as widgets) it contains.
- It does this based on the constraint connections that are set on each child widget.
- In order to fully understand and use ConstraintLayout, it is important to gain an appreciation of the following key concepts:
 - Constraints
 - Margins
 - Opposing Constraints
 - Constraint Bias
 - Chains
 - Chain Styles
 - Barriers

Constraints

- Constraints are essentially sets of rules that dictate the way in which a widget is aligned and distanced in relation to other widgets.
- The sides of the containing ConstraintLayout and special elements are called guidelines.

- Constraints also dictate how the user interface layout of an activity will respond to changes in device orientation, or when displayed on devices of differing screen sizes.
- In order to be adequately configured, a widget must have sufficient constraint connections such that it's position can be resolved by the ConstraintLayout layout engine in both the horizontal and vertical planes.

Margins

- A margin is a form of constraint that specifies a fixed distance.
- Consider a Button object that needs to be positioned near the top right-hand corner of the device screen.
- This might be achieved by implementing margin constraints from the top and right-hand edges of the Button connected to the corresponding sides of the parent ConstraintLayout as illustrated in Figure 9-1:

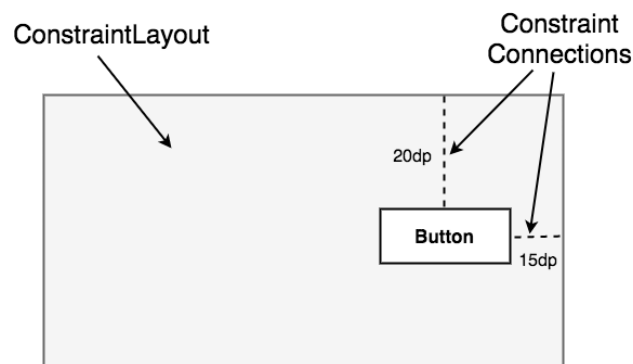


Figure 9-1

- As indicated in the above diagram, each of these constraint connections has associated with it a margin value dictating the fixed distances of the widget from two sides of the parent layout.
- Under this configuration, regardless of screen size or the device orientation, the Button object will always be positioned 20 and 15 device-independent pixels (dp) from the top and right-hand edges of the parent ConstraintLayout respectively as specified by the two constraint connections.
- While the above configuration will be acceptable for some situations, it does not provide any flexibility in terms of allowing the ConstraintLayout layout engine to adapt the position of the widget in order to respond to device rotation and to support screens of different sizes.
- To add this responsiveness to the layout it is necessary to implement opposing constraints.

Opposing Constraints

- Two constraints operating along the same axis on a single widget are referred to as opposing constraints. In other words, a widget with constraints on both its left and right-hand sides is considered to have horizontally opposing constraints.
- Figure 9-2, for example, illustrates the addition of both horizontally and vertically opposing constraints to the previous layout:

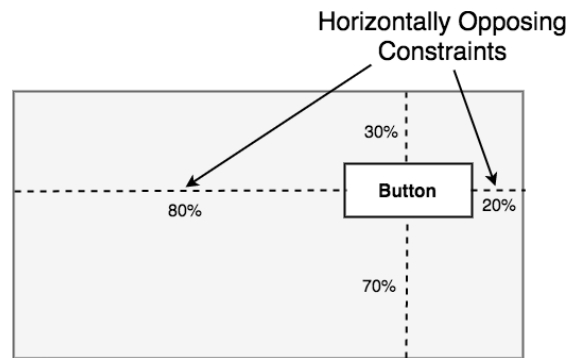
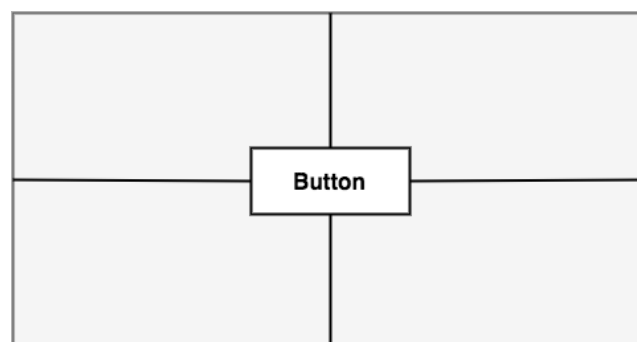


Figure 9-2

- The key point to understand here is that once opposing constraints are implemented on a particular axis, the positioning of the widget becomes percentage rather than coordinate based.
- Instead of being fixed at 20dp from the top of the layout, for example, the widget is now positioned at a point 30% from the top of the layout.
- In different orientations and when running on larger or smaller screens, the Button will always be in the same location relative to the dimensions of the parent layout.
- It is now important to understand that the layout outlined in Figure 9-2 has been implemented using not only opposing constraints, but also by applying constraint bias.

Constraint Bias

- It has now been established that a widget in a ConstraintLayout can potentially be subject to opposing constraint connections.
- By default, opposing constraints are equal, resulting in the corresponding widget being centered along the axis of opposition.
- Figure 9-3, for example, shows a widget centered within the containing ConstraintLayout using opposing horizontal and vertical constraints:

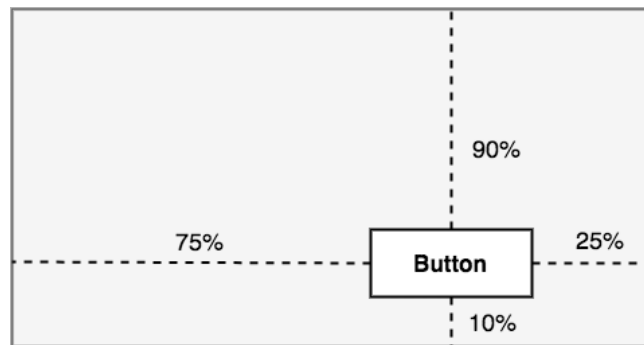


Widget Centered by Opposing Constraints

Figure 9-3

- To allow for the adjustment of widget position in the case of opposing constraints, the ConstraintLayout implements a feature known as constraint bias.

- Constraint bias allows the positioning of a widget along the axis of opposition to be biased by a specified percentage in favor of one constraint.
- Figure 9-4, for example, shows the previous constraint layout with a 75% horizontal bias and 10% vertical bias:



Widget Offset using Constraint Bias

Figure 9-4

- The next section, we will cover these concepts in greater detail and explain how these features have been integrated into the Android Studio Layout Editor tool.
- In the meantime, however, a few more areas of the ConstraintLayout class need to be covered.

Chains

- ConstraintLayout chains provide a way for the layout behavior of two or more widgets to be defined as a group.
- Chains can be declared in either the vertical or horizontal axis and configured to define how the widgets in the chain are spaced and sized.
- Widgets are chained when connected together by bi-directional constraints.
- Figure 9-5, for example, illustrates three widgets chained in this way:

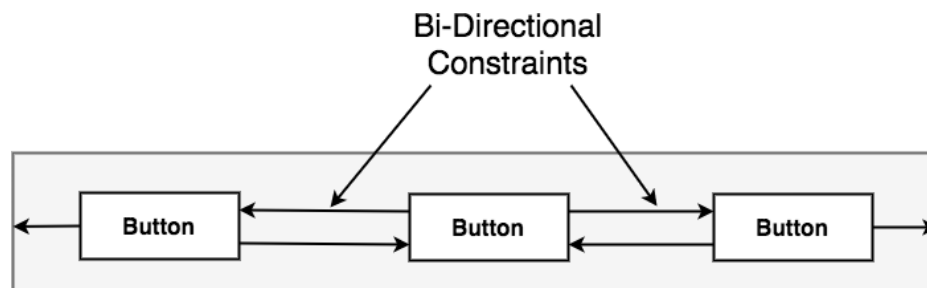


Figure 9-5

- The first element in the chain is the chain head which translates to the top widget in a vertical chain or, in the case of a horizontal chain, the left-most widget.

- The layout behavior of the entire chain is primarily configured by setting attributes on the chain head widget.

Chain Styles

- The layout behavior of a ConstraintLayout chain is dictated by the chain style setting applied to the chain head widget. The ConstraintLayout class currently supports the following chain layout styles:
- Spread Chain – The widgets contained within the chain are distributed evenly across the available space.
- This is the default behavior for chains.



Figure 9-6

- Spread Inside Chain – The widgets contained within the chain are spread evenly between the chain head and the last widget in the chain. The head and last widgets are not included in the distribution of spacing.

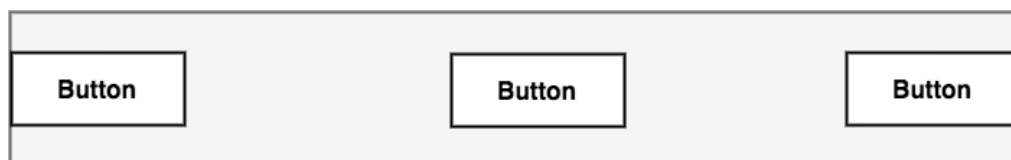


Figure 9-7

- Weighted Chain – Allows the space taken up by each widget in the chain to be defined via weighting properties.

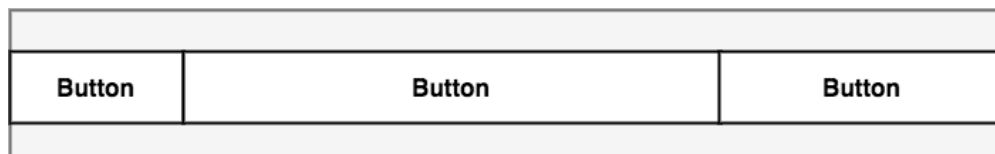


Figure 9-8

- Packed Chain – The widgets that make up the chain are packed together without any spacing. A bias may be applied to control the horizontal or vertical positioning of the chain in relation to the parent container.

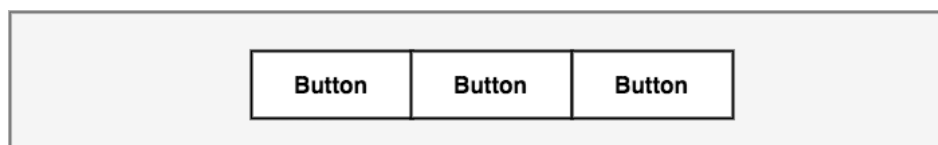
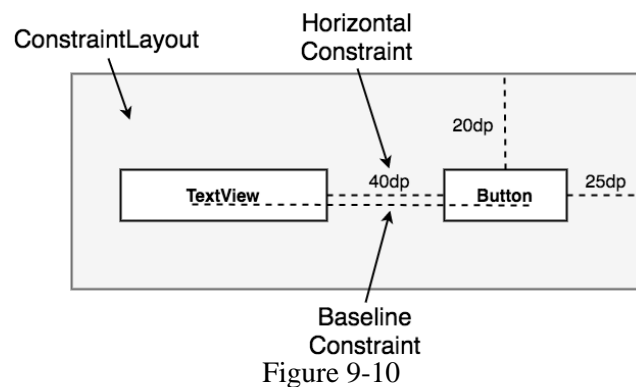


Figure 9-9

(continued)

Baseline Alignment

- So far, this chapter has only referred to constraints that dictate alignment relative to the sides of a widget (typically referred to as side constraints).
- A common requirement, however, is for a widget to be aligned relative to the content that it displays rather than the boundaries of the widget itself.
- To address this need, ConstraintLayout provides baseline alignment support.
- As an example, assume that the previous theoretical layout from Figure 9-1 requires a TextView widget to be positioned 40dp to the left of the Button.
- In this case, the TextView needs to be baseline aligned with the Button view.
- This means that the text within the Button needs to be vertically aligned with the text within the TextView.
- The additional constraints for this layout would need to be connected as illustrated in Figure 9-10:



- The TextView is now aligned vertically along the baseline of the Button and positioned 40dp horizontally from the Button object's left-hand edge.

Working with Guidelines

- Guidelines are special elements available within the ConstraintLayout that provide an additional target to which constraints may be connected.
- Multiple guidelines may be added to a ConstraintLayout instance which may, in turn, be configured in horizontal or vertical orientations.
- Once added, constraint connections may be established from widgets in the layout to the guidelines.
- This is particularly useful when multiple widgets need to be aligned along an axis.
- In Figure 9-11, for example, three Button objects contained within a ConstraintLayout are constrained along a vertical guideline:

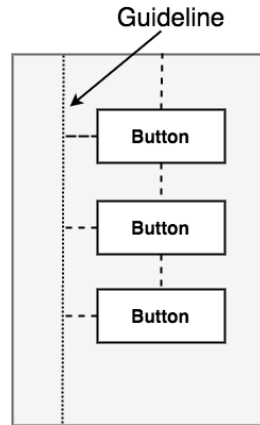


Figure 9-11

Configuring Widget Dimensions

- Controlling the dimensions of a widget is a key element of the user interface design process.
- The ConstraintLayout provides three options which can be set on individual widgets to manage sizing behavior.
- These settings are configured individually for height and width dimensions:
 - Fixed – The widget is fixed to specified dimensions.
 - Match Constraint – Allows the widget to be resized by the layout engine to satisfy the prevailing constraints.

Also referred to as the AnySize or MATCH_CONSTRAINT option.

- Wrap Content – The size of the widget is dictated by the content it contains (i.e. text or graphics).

Working with Barriers

- Rather like guidelines, barriers are virtual views that can be used to constrain views within a layout.
- As with guidelines, a barrier can be vertical or horizontal and one or more views may be constrained to it (to avoid confusion, these will be referred to as constrained views).
- Unlike guidelines where the guideline remains at a fixed position within the layout, however, the position of a barrier is defined by a set of so called reference views.
- Barriers were introduced to address an issue that occurs with some frequency involving overlapping views.
- Consider, for example, the layout illustrated in Figure 9-12 below:

(continued)

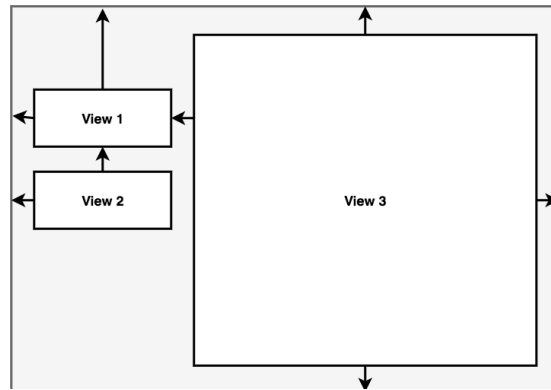


Figure 9-12

- The key points to note about the above layout is that the width of View 3 is set to match constraint mode, and the left-hand edge of the view is connected to the right hand edge of View 1.
- As currently implemented, an increase in width of View 1 will have the desired effect of reducing the width of View 3:

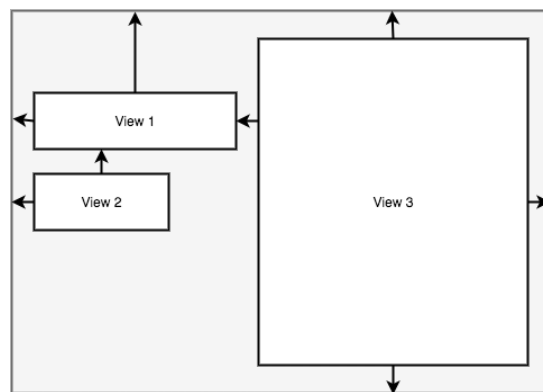


Figure 9-13

A problem arises, however, if View 2 increases in width instead of View 1:

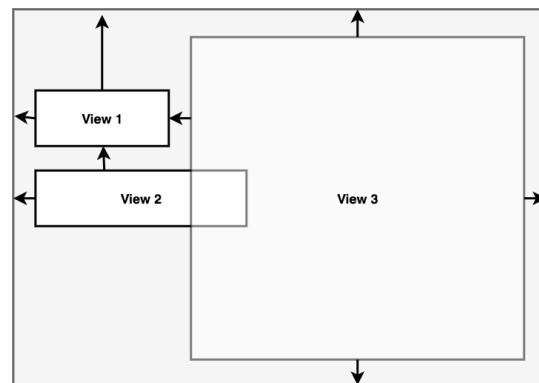


Figure 9-14

- Clearly because View 3 is only constrained by View 1, it does not resize to accommodate the increase in width of View 2 causing the views to overlap.

- A solution to this problem is to add a vertical barrier and assign Views 1 and 2 as the barrier's reference views so that they control the barrier position.
- The left-hand edge of View 3 will then be constrained in relation to the barrier, making it a constrained view.
- Now when either View 1 or View 2 increase in width, the barrier will move to accommodate the widest of the two views, causing the width of View 3 change in relation to the new barrier position:

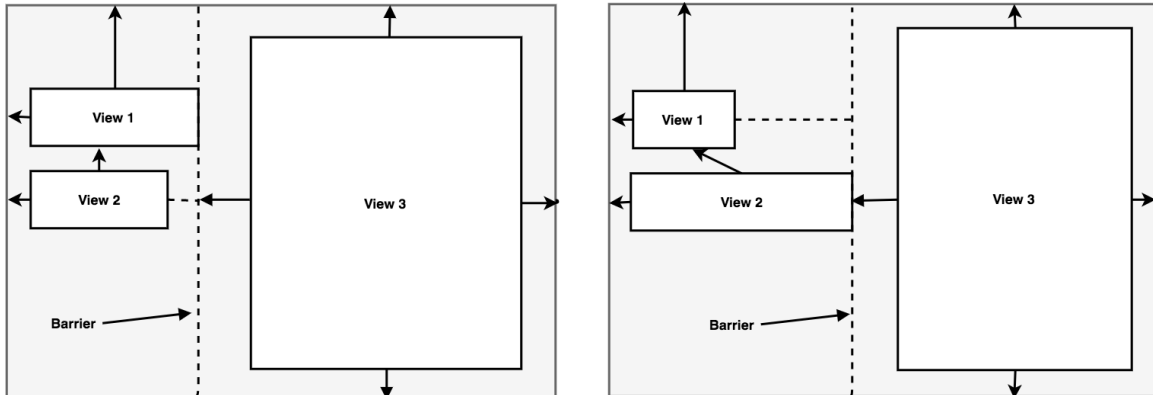


Figure 9-15

- When working with barriers there is no limit to the number of reference views and constrained views that can be associated with a single barrier.

Ratios

- The dimensions of a widget may be defined using ratio settings.
- A widget could, for example, be constrained using a ratio setting such that, regardless of any resizing behavior, the width is always twice the height dimension.

ConstraintLayout Advantages

- ConstraintLayout provides a level of flexibility that allows many of the features of older layouts to be achieved with a single layout instance where it would previously have been necessary to nest multiple layouts.
- This has the benefit of avoiding the problems inherent in layout nesting by allowing so called “flat” or “shallow” layout hierarchies to be designed leading both to less complex layouts and improved user interface rendering performance at runtime.
- ConstraintLayout was also implemented with a view to addressing the wide range of Android device screen sizes available on the market today.
- The flexibility of ConstraintLayout makes it easier for user interfaces to be designed that respond and adapt to the device on which the app is running.

- Finally, as will be demonstrated in a later section, the Android Studio Layout Editor tool has been enhanced specifically for ConstraintLayout-based user interface design.

ConstraintLayout Availability

- Although introduced with Android 7, ConstraintLayout is provided as a separate support library from the main Android SDK and is compatible with older Android versions as far back as API Level 9 (Gingerbread).
- This allows apps that make use of this new layout to run on devices running much older versions of Android.
- As mentioned more than once in previous chapters, Google has made significant changes to the Android Studio Layout Editor tool, many of which were made solely to support user interface layout design using ConstraintLayout.
- Now that the basic concepts of ConstraintLayout have been outlined in the previous chapter, this chapter will explore these concepts in more detail while also outlining the ways in which the Layout Editor tool allows ConstraintLayout-based user interfaces to be designed and implemented.

Design and Layout Views

- An earlier chapter explained that the Android Studio Layout Editor tool provides two ways to view the user interface layout of an activity in the form of Design and Layout (also known as blueprint) views.
- These views of the layout may be displayed individually or, as in Figure 9-16, side by side:

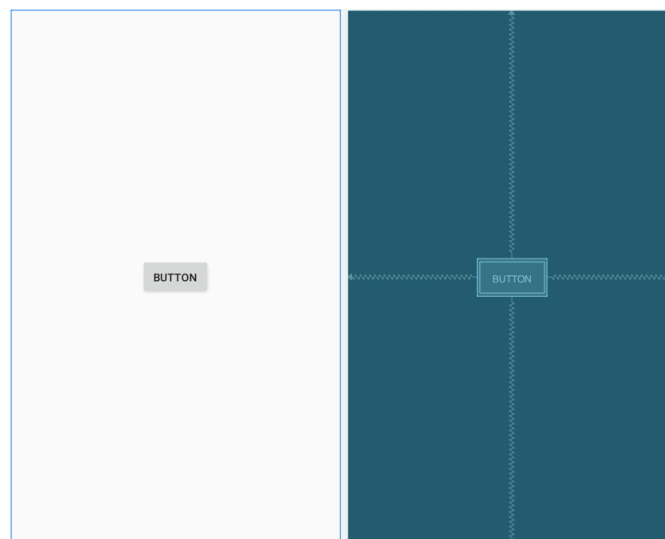


Figure 9-16

- The Design view (positioned on the left in the above figure) presents a “what you see is what you get” representation of the layout, wherein the layout appears as it will within the running app.
- The Layout view, on the other hand, displays a blueprint style of view where the widgets are represented by shaded outlines.

- As can be seen in Figure 9-16 above, Layout view also displays the constraint connections (in this case opposing constraints used to center a button within the layout).
- These constraints are also overlaid onto the Design view when a specific widget in the layout is selected or when the mouse pointer hovers over the design area as illustrated in Figure 9-17:

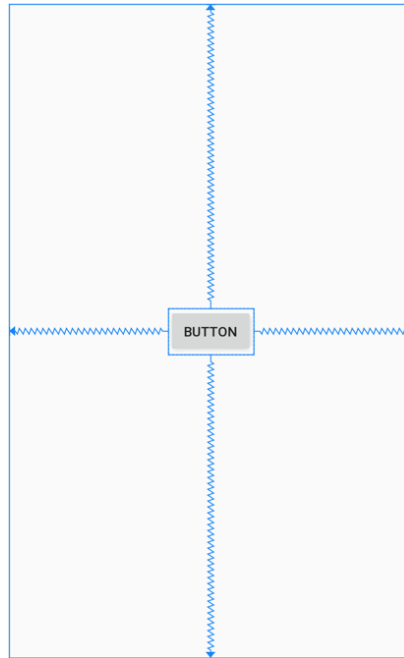


Figure 9-17

- The appearance of constraint connections in both views can be changed using the View Options menu shown in Figure 9-18:

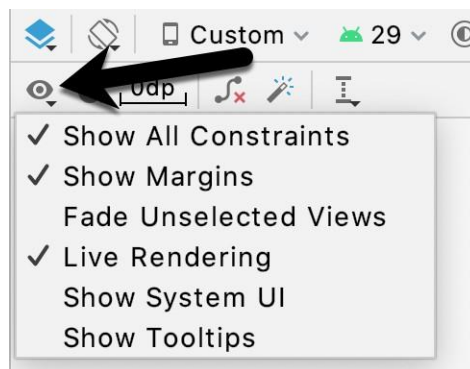


Figure 9-18

- In addition to the two modes of displaying the user interface layout, the Layout Editor tool also provides three different ways of establishing the constraints required for a specific layout design.

(continued)

Autoconnect Mode

- Autoconnect, as the name suggests, automatically establishes constraint connections as items are added to the layout. Autoconnect mode may be enabled and disabled using the toolbar button indicated in Figure 9-19:

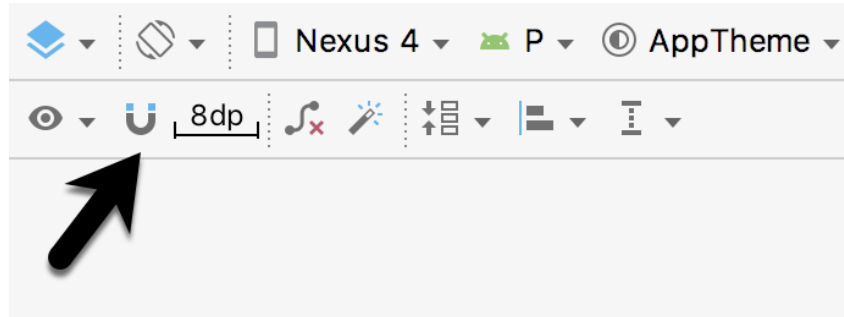


Figure 9-19

- Autoconnect mode uses algorithms to decide the best constraints to establish based on the position of the widget and the widget's proximity to both the sides of the parent layout and other elements in the layout.
- In the event that any of the automatic constraint connections fail to provide the desired behavior, these may be changed manually as outlined later in this chapter.

Inference Mode

- Inference mode uses a heuristic approach involving algorithms and probabilities to automatically implement constraint connections after widgets have already been added to the layout.
- This mode is usually used when the Autoconnect feature has been turned off and objects have been added to the layout without any constraint connections.
- This allows the layout to be designed simply by dragging and dropping objects from the palette onto the layout canvas and making size and positioning changes until the layout appears as required.
- In essence this involves “painting” the layout without worrying about constraints.
- Inference mode may also be used at any time during the design process to fill in missing constraints within a layout.
- Constraints are automatically added to a layout when the Infer constraints button (Figure 9-20) is clicked:

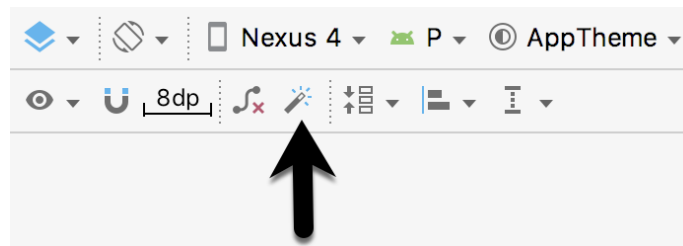


Figure 9-20

- As with Autoconnect mode, there is always the possibility that the Layout Editor tool will infer incorrect constraints, though these may be modified and corrected manually.

Manipulating Constraints Manually

- The third option for implementing constraint connections is to do so manually.
- When doing so, it will be helpful to understand the various handles that appear around a widget within the Layout Editor tool.
- Consider, for example, the widget shown in Figure 9-21:

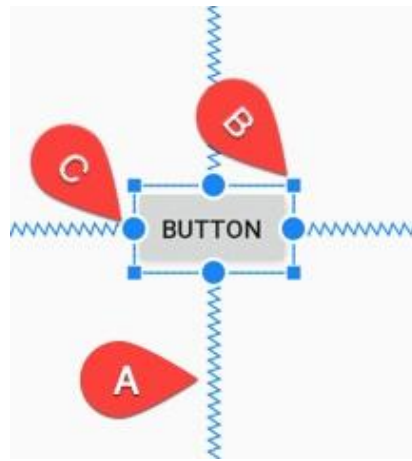


Figure 9-21

- Clearly the spring-like lines (A) represent established constraint connections leading from the sides of the widget to the targets.
- The small square markers (B) in each corner of the object are resize handles which, when clicked and dragged, serve to resize the widget.
- The small circle handles (C) located on each side of the widget are the side constraint anchors.
- To create a constraint connection, click on the handle and drag the resulting line to the element to which the constraint is to be connected (such as a guideline or the side of either the parent layout or another widget) as outlined in Figure 9-22.
- When connecting to the side of another widget, simply drag the line to the side constraint handle of that widget and release the line when the widget and handle highlight.



Figure 9-22

- If the constraint line is dragged to a widget and released, but not attached to a constraint handle, the layout editor will display a menu containing a list of the sides to which the constraint may be attached.

- In Figure 9-23, for example, the constraint can be attached to the top or bottom edge of the destination button widget:

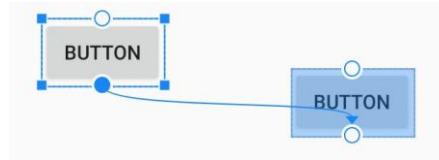


Figure 9-23

- An additional marker indicates the anchor point for baseline constraints whereby the content within the widget (as opposed to outside edges) is used as the alignment point.
- To display this marker, simply right-click on the widget and select the Show Baseline menu option.
- To establish a constraint connection from a baseline constraint handle, simply hover the mouse pointer over the handle until it begins to flash before clicking and dragging to the target (such as the baseline anchor of another widget as shown in Figure 9-24).
- When the destination anchor begins to flash green, release the mouse button to make the constraint connection:



Figure 9-24

- To hide the baseline anchors, right click on the widget a second time and select the Hide Baseline menu option.

Adding Constraints in the Inspector

- Constraints may also be added to a view within the Android Studio Layout Editor tool using the Inspector panel located in the Attributes tool window as shown in Figure 9-25.
- The square in the center represents the currently selected view and the areas around the square the constraints, if any, applied to the corresponding sides of the view:

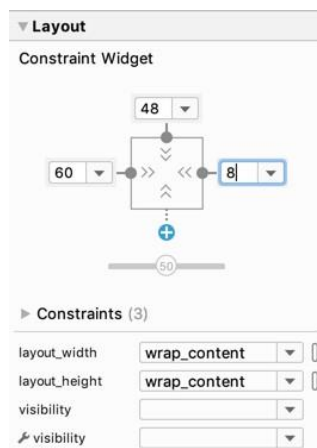


Figure 9-25

- The absence of a constraint on a side of the view is represented by a dotted line leading to a blue circle containing a plus sign (as is the case with the bottom edge of the view in the above figure).
- To add a constraint, simply click on this blue circle and the layout editor will add a constraint connected to what it considers to be the most appropriate target within the layout.

Viewing Constraints in the Attributes Window

- A list of constraints configured on the currently select widget can be viewing by displaying the Constraints section of the Attributes tool window as shown in Figure 9-26 below:

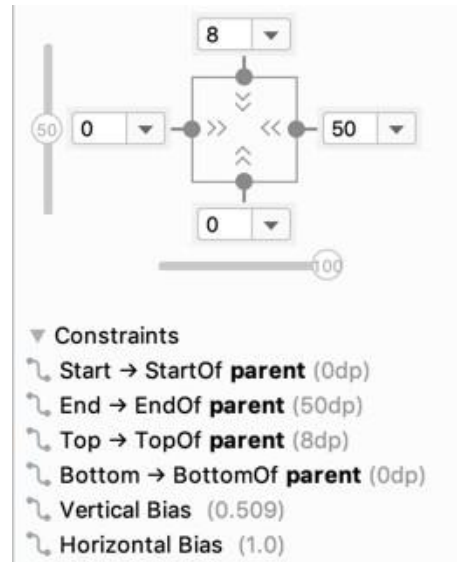


Figure 9-26

- Clicking on a constraint in the list will select that constraint within the design layout.

Deleting Constraints

- To delete an individual constraint, simply select the constraint either within the design layout or the Attributes tool window so that it highlights (in Figure 9-27, for example, the right-most constraint has been selected) and tap the keyboard delete key.
- The constraint will then be removed from the layout.

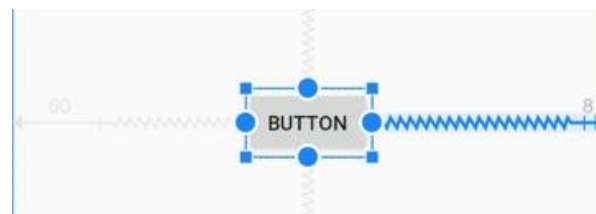


Figure 9-27

- Another option is to hover the mouse pointer over the constraint anchor while holding down the Ctrl (Cmd on macOS) key and clicking on the anchor after it turns red:

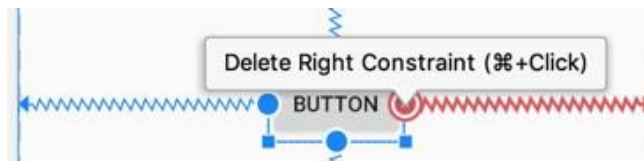


Figure 9-28

- Alternatively, remove all of the constraints on a widget by right-clicking on it selecting the Clear Constraints of Selection menu option.
- To remove all of the constraints from every widget in a layout, use the toolbar button highlighted in Figure 9-29:

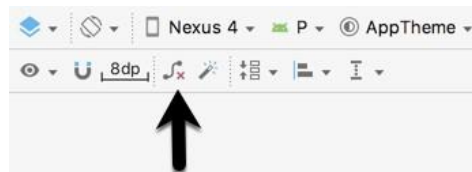


Figure 9-29

Adjusting Constraint Bias

- In the previous chapter, the concept of using bias settings to favor one opposing constraint over another was outlined.
- Bias within the Android Studio Layout Editor tool is adjusted using the Inspector located in the Attributes tool window and shown in Figure 9-30.
- The two sliders indicated by the arrows in the figure are used to control the bias of the vertical and horizontal opposing constraints of the currently selected widget.

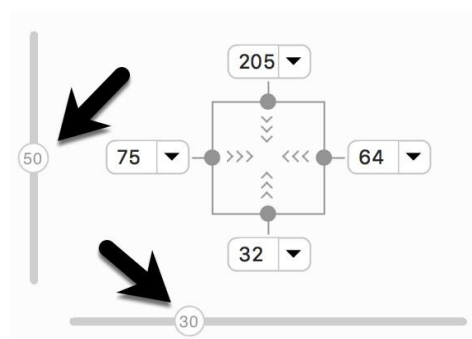


Figure 9-30

Understanding ConstraintLayout Margins

- Constraints can be used in conjunction with margins to implement fixed gaps between a widget and another element (such as another widget, a guideline or the side of the parent layout).
- Consider, for example, the horizontal constraints applied to the Button object in Figure 9-31:

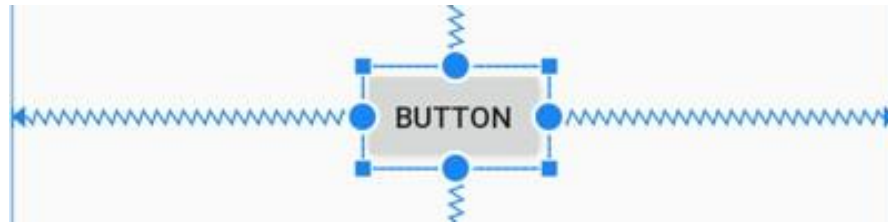


Figure 9-31

- As currently configured, horizontal constraints run to the left and right edges of the parent ConstraintLayout.
- As such, the widget has opposing horizontal constraints indicating that the ConstraintLayout layout engine has some discretion in terms of the actual positioning of the widget at runtime.
- This allows the layout some flexibility to accommodate different screen sizes and device orientation.
- The horizontal bias setting is also able to control the position of the widget right up to the right-hand side of the layout.
- Figure 9-32, for example, shows the same button with 100% horizontal bias applied:



Figure 9-32

- ConstraintLayout margins can appear at the end of constraint connections and represent a fixed gap into which the widget cannot be moved even when adjusting bias or in response to layout changes elsewhere in the activity.
- In Figure 9-33, the right-hand constraint now includes a 50dp margin into which the widget cannot be moved even though the bias is still set at 100%.



Figure 9-33

- Existing margin values on a widget can be modified from within the Inspector.

- As can be seen in Figure 9-34, a dropdown menu is being used to change the right-hand margin on the currently selected widget to 16dp.
- Alternatively, clicking on the current value also allows a number to be typed into the field.

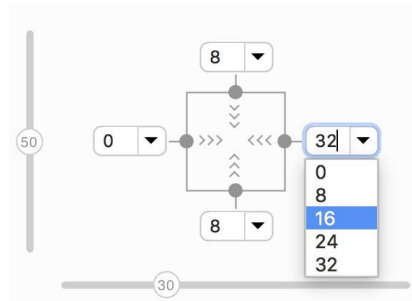


Figure 9-34

- The default margin for new constraints can be changed at any time using the option in the toolbar highlighted in Figure 9-35:

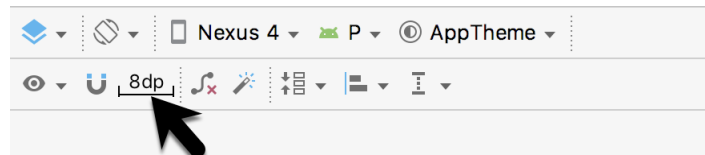


Figure 9-35

The Importance of Opposing Constraints and Bias

- As discussed in the previous chapter, opposing constraints, margins and bias form the cornerstone of responsive layout design in Android when using the ConstraintLayout.
- When a widget is constrained without opposing constraint connections, those constraints are essentially margin constraints.
- This is indicated visually within the Layout Editor tool by solid straight lines accompanied by margin measurements as shown in Figure 9-36.

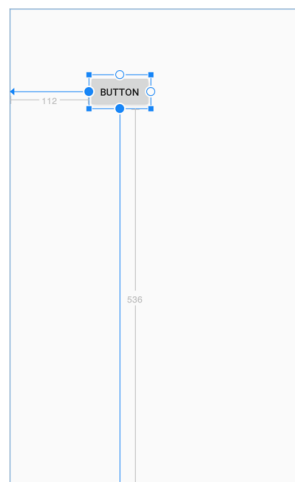


Figure 9-36

- The above constraints essentially fix the widget at that position.
- The result of this is that if the device is rotated to landscape orientation, the widget will no longer be visible since the vertical constraint pushes it beyond the top edge of the device screen (as is the case in Figure 9-37).
- A similar problem will arise if the app is run on a device with a smaller screen than that used during the design process.

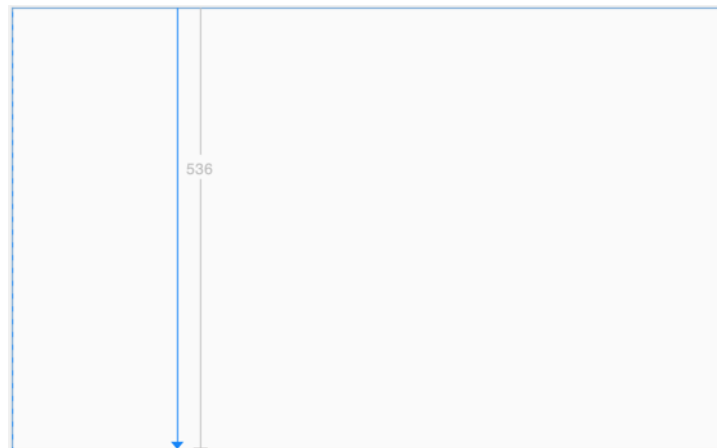


Figure 9-37

- When opposing constraints are implemented, the constraint connection is represented by the spring-like jagged line (the spring metaphor is intended to indicate that the position of the widget is not fixed to absolute X and Y coordinates):

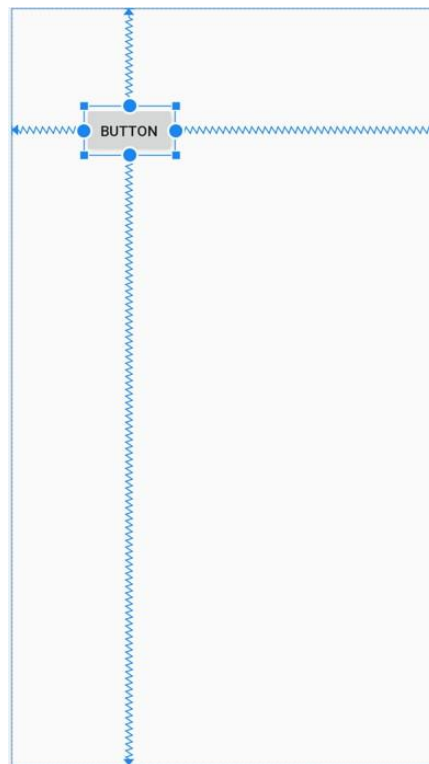


Figure 9-38

- In the above layout, vertical and horizontal bias settings have been configured such that the widget will always be positioned 90% of the distance from the bottom and 35% from the left-hand edge of the parent layout.
- When rotated, therefore, the widget is still visible and positioned in the same location relative to the dimensions of the screen:

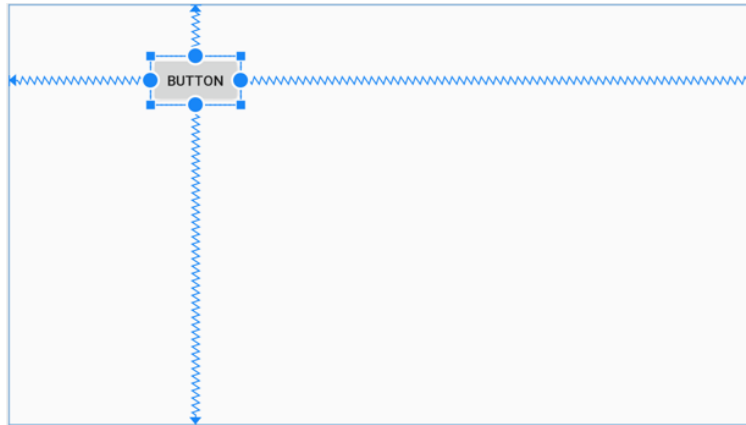


Figure 9-39

- When designing a responsive and adaptable user interface layout, it is important to take into consideration both bias and opposing constraints when manually designing a user interface layout and making corrections to automatically created constraints.

Configuring Widget Dimensions

- The inner dimensions of a widget within a ConstraintLayout can also be configured using the Inspector. As outlined in the previous chapter, widget dimensions can be set to wrap content, fixed or match constraint modes.
- The prevailing settings for each dimension on the currently selected widget are shown within the square representing the widget in the Inspector as illustrated in Figure 9-40:

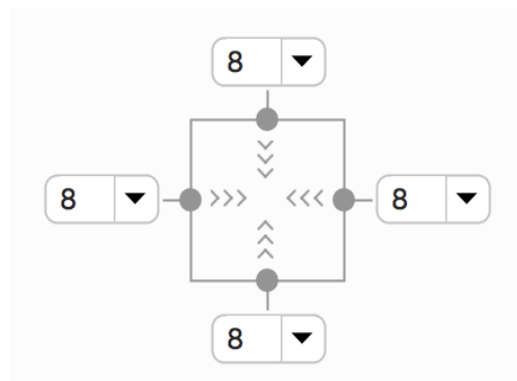


Figure 9-40

- In the above figure, both the horizontal and vertical dimensions are set to wrap content mode (indicated by the inward pointing chevrons).

- The inspector uses the following visual indicators to represent the three dimension modes:




• Fixed Size	• 
• Match Constraint	• 
• Wrap Content	• 

Table 9-41

- To change the current setting, simply click on the indicator to cycle through the three different settings.
- When the dimension of a view within the layout editor is set to match constraint mode, the corresponding sides of the view are drawn with the spring-like line instead of the usual straight lines.
- In Figure 9-42, for example, only the width of the view has been set to match constraint:

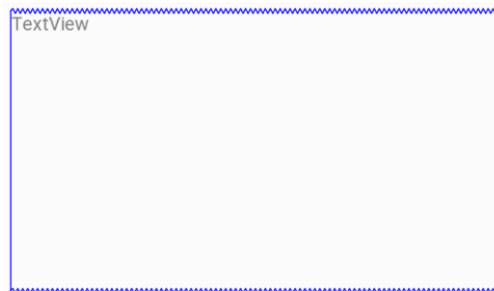


Figure 9-42

- In addition, the size of a widget can be expanded either horizontally or vertically to the maximum amount allowed by the constraints and other widgets in the layout using the Expand horizontally and Expand vertically options.
- These are accessible by right clicking on a widget within the layout and selecting the Organize option from the resulting menu (Figure 9-43).
- When used, the currently selected widget will increase in size horizontally or vertically to fill the available space around it.

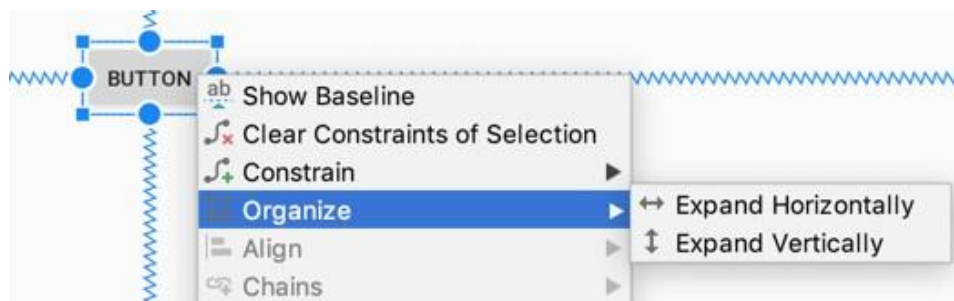


Figure 9-43

Adding Guidelines

- Guidelines provide additional elements to which constraints may be anchored.
- Guidelines are added by right-clicking on the layout and selecting either the Add Vertical Guideline or Add Horizontal Guideline menu option or using the toolbar menu options as shown in Figure 9-44:

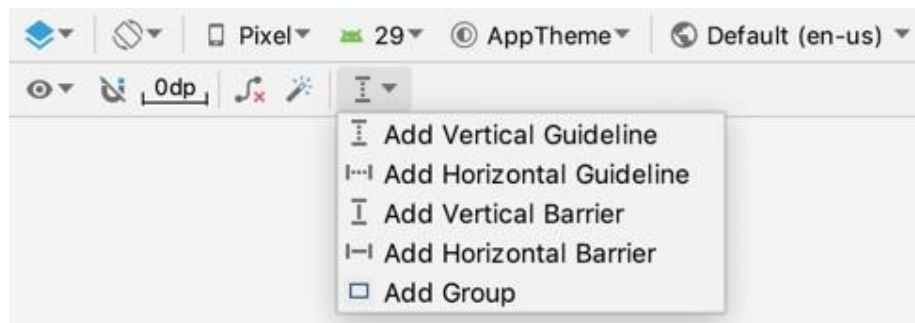


Figure 9-44

- Once added, a guideline will appear as a dashed line in the layout and may be moved simply by clicking and dragging the line.
- To establish a constraint connection to a guideline, click in the constraint handler of a widget and drag to the guideline before releasing.
- In Figure 9-45, the left sides of two Buttons are connected by constraints to a vertical guideline.
- The position of a vertical guideline can be specified as an absolute distance from either the left or the right of the parent layout (or the top or bottom for a horizontal guideline).
- The vertical guideline in the above figure, for example, is positioned 96dp from the left-hand edge of the parent.

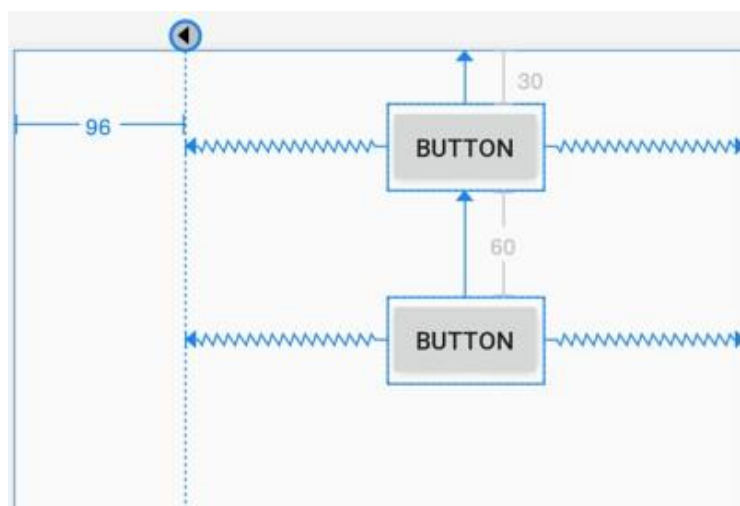


Figure 9-45

(continued)

- Alternatively, the guideline may be positioned as a percentage of the overall width or height of the parent layout.
- To switch between these three modes, select the guideline and click on the circle at the top or start of the guideline (depending on whether the guideline is vertical or horizontal).
- Figure 9-46, for example, shows a guideline positioned based on percentage:

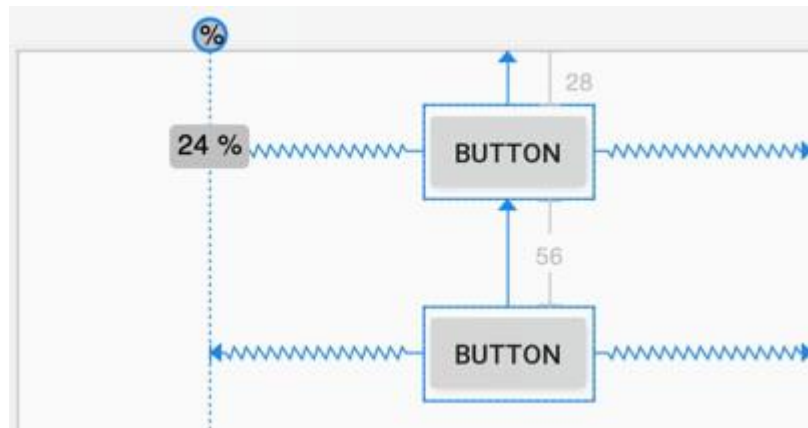


Figure 9-46

Adding Barriers

- Barriers are added by right-clicking on the layout and selecting either the Add Vertical Barrier or Add Horizontal Barrier option from the Helpers menu, or using the toolbar menu options as shown previously.
- Once a barrier has been added to the layout, it will appear as an entry in the Component Tree panel:

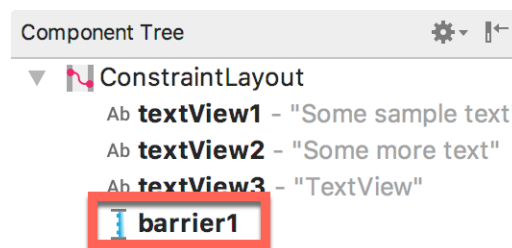


Figure 9-47

- To add views as reference views (in other words, the views that control the position of the barrier), simply drag the widgets from within the Component Tree onto the barrier entry.
- In Figure 9-48, for example, widgets named textView1 and textView2 have been assigned as the reference widgets for barrier1:

(continued)

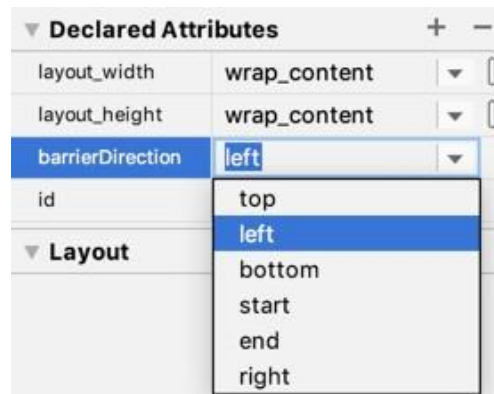


Figure 9-48

- After the reference views have been added, the barrier needs to be configured to specify the direction of the barrier in relation those views.
- This is the barrier direction setting and is defined within the Attributes tool window when the barrier is selected in the Component Tree panel:

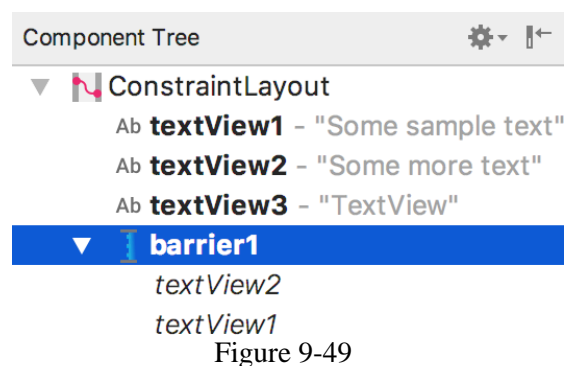


Figure 9-49

- The following figure shows a layout containing a barrier declared with textView1 and textView2 acting as the reference views and textView3 as the constrained view.
- Since the barrier is pushing from the end of the reference views towards the constrained view, the barrier direction has been set to end:

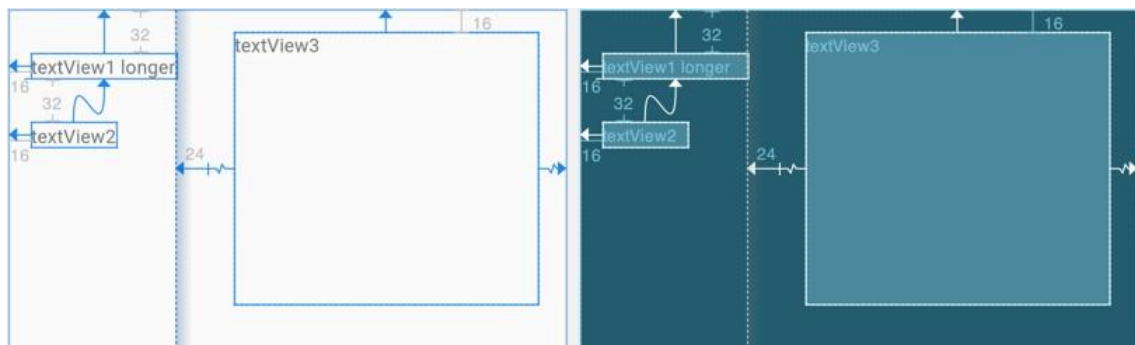


Figure 9-50

Widget Group Alignment and Distribution

- The Android Studio Layout Editor tool provides a range of alignment and distribution actions that can be performed when two or more widgets are selected in the layout.
- Simply shift-click on each of the widgets to be included in the action, right-click on the layout and make a selection from the many options displayed in the Align menu:

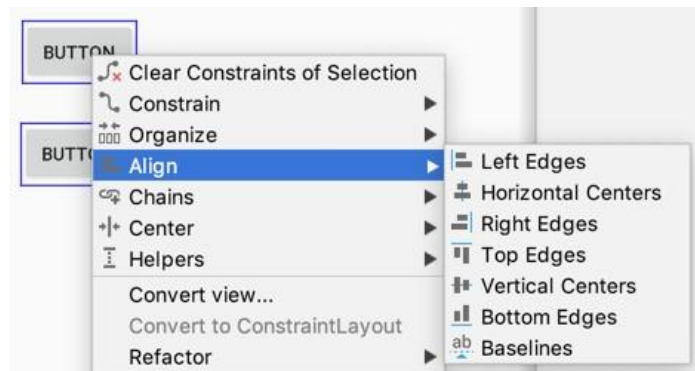


Figure 9-51

- As shown in Figure 9-52 below, these options are also available as buttons in the Layout Editor toolbar:

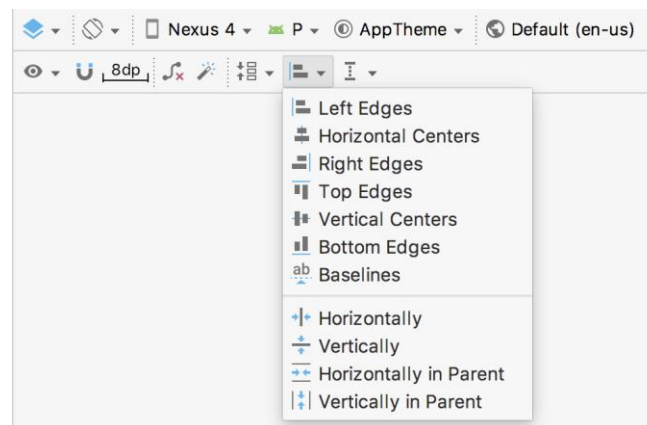


Figure 9-52

- Similarly, the Pack menu (Figure 9-53) can be used to collectively reposition the selected widgets so that they are packed tightly together either vertically or horizontally.
- It achieves this by changing the absolute x and y coordinates of the widgets but does not apply any constraints.
- The two distribution options in the Pack menu, on the other hand, move the selected widgets so that they are spaced evenly apart in either vertical or horizontal axis and applies constraints between the views to maintain this spacing.

(continued)

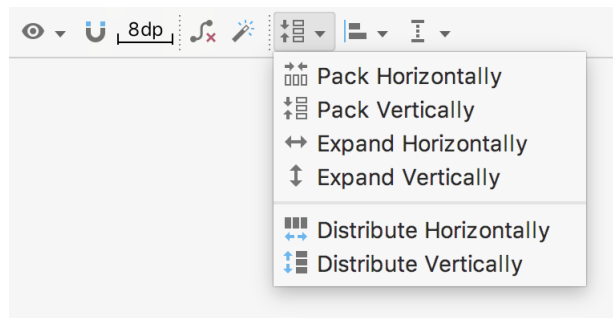


Figure 9-53

Converting Other Layouts to ConstraintLayout

- For existing user interface layouts that make use of one or more of the other Android layout classes (such as RelativeLayout or LinearLayout), the Layout Editor tool provides an option to convert the user interface to use the ConstraintLayout.
- When the Layout Editor tool is open and in Design mode, the Component Tree panel is displayed beneath the Palette.
- To convert a layout to ConstraintLayout, locate it within the Component Tree, right-click on it and select the Convert <current layout> to Constraint Layout menu option:

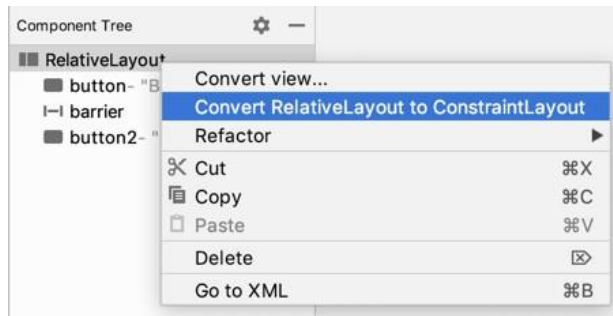


Figure 9-54

- When this menu option is selected, Android Studio will convert the selected layout to a ConstraintLayout and use inference to establish constraints designed to match the layout behavior of the original layout type.
- The previous chapters have introduced the key features of the ConstraintLayout class and outlined the best practices for ConstraintLayout-based user interface design within the Android Studio Layout Editor.
- Although the concepts of ConstraintLayout chains and ratios were outlined earlier, we have not yet addressed how to make use of these features within the Layout Editor.
- The focus of this chapter, therefore, is to provide practical steps on how to create and manage chains and ratios when using the ConstraintLayout class.

(continued)

Creating a Chain

- Chains may be implemented either by adding a few lines to the XML layout resource file of an activity or by using some chain specific features of the Layout Editor.
- Consider a layout consisting of three Button widgets constrained so as to be positioned in the top-left, top-center and top-right of the ConstraintLayout parent as illustrated in Figure 9-55:

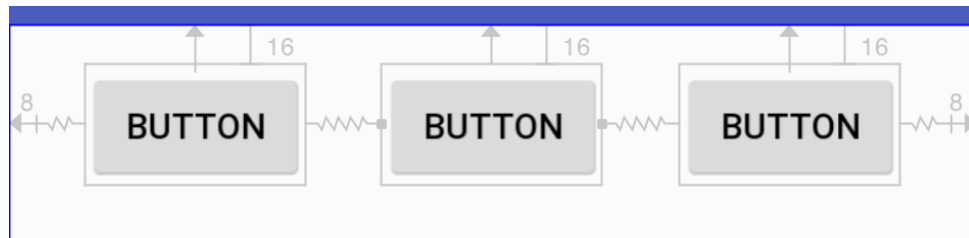


Figure 9-55

- To represent such a layout, the XML resource layout file might contain the following entries for the button widgets:

```
<Button
    android:id="@+id/button1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginTop="16dp"
    android:text="Button"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
/>

<Button
    android:id="@+id/button2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="16dp"
    android:text="Button"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintEnd_toStartOf="@+id/button3"
    app:layout_constraintStart_toEndOf="@+id/button1"
    app:layout_constraintTop_toTopOf="parent"
/>

<Button
    android:id="@+id/button3"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
```

```

        android:layout_marginEnd="8dp"
        android:layout_marginTop="16dp"
        android:text="Button"
        app:layout_constraintHorizontal_bias="0.5"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent"
    />

```

- As currently configured, there are no bi-directional constraints to group these widgets into a chain.
- To address this, additional constraints need to be added from the right-hand side of button1 to the left side of button2, and from the left side of button3 to the right side of button2 as follows:

```

<Button
    android:id="@+id/button1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginTop="16dp" android:text="Button"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintEnd_toStartOf="@+id/button2"
/>

```

```

<Button
    android:id="@+id/button2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="16dp"
    android:text="Button"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintEnd_toStartOf="@+id/button3"
    app:layout_constraintStart_toEndOf="@+id/button1"
    app:layout_constraintTop_toTopOf="parent"
/>

```

```

<Button
    android:id="@+id/button3"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginEnd="8dp"
    android:layout_marginTop="16dp"
    android:text="Button"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintStart_toEndOf="@+id/button2"
/>

```

- With these changes, the widgets now have bi-directional horizontal constraints configured.
- This essentially constitutes a ConstraintLayout chain which is represented visually within the Layout Editor by chain connections as shown in Figure 9-56 below.
- Note that in this configuration the chain has defaulted to the spread chain style.

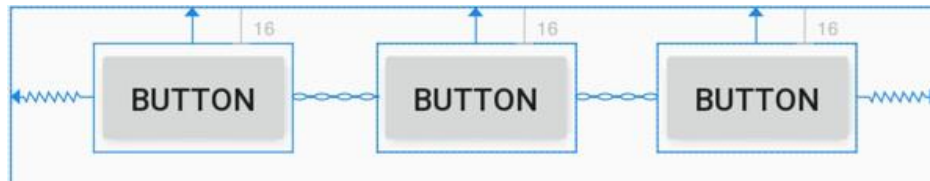


Figure 9-56

- A chain may also be created by right-clicking on one of the views and selecting the Chains > Create Horizontal Chain or Chains > Create Vertical Chain menu options.

Changing the Chain Style

- If no chain style is configured, the ConstraintLayout will default to the spread chain style.
- The chain style can be altered by right-clicking any of the widgets in the chain and selecting the Cycle Chain Mode menu option.
- Each time the menu option is clicked the style will switch to another setting in the order of spread, spread inside and packed.
- Alternatively, the style may be specified in the Attributes tool window unfolding the layout_constraints property and changing either the horizontal_chainStyle or vertical_chainStyle property depending on the orientation of the chain:

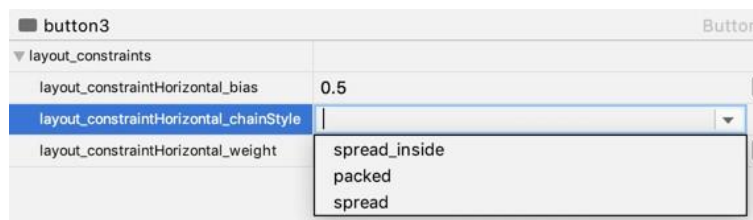


Figure 9-57

Spread Inside Chain Style

- Figure 9-58 illustrates the effect of changing the chain style to the spread inside chain style using the above techniques:

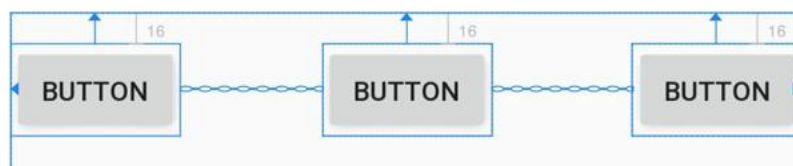


Figure 9-58

Packed Chain Style

- Using the same technique, changing the chain style property to packed causes the layout to change as shown in Figure 9-59:

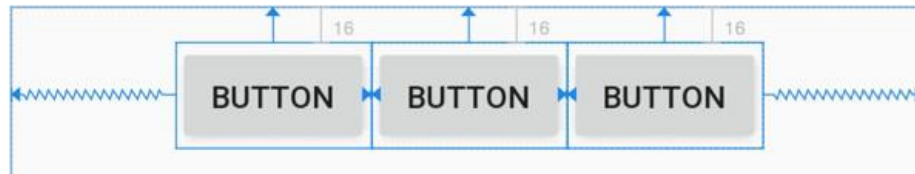


Figure 9-59

Packed Chain Style with Bias

- The positioning of the packed chain may be influenced by applying a bias value.
- The bias can be any value between 0.0 and 1.0, with 0.5 representing the center of the parent.
- Bias is controlled by selecting the chain head widget and assigning a value to the `horizontal_bias` or `vertical_bias` attribute in the Attributes panel.
- Figure 9-60 shows a packed chain with a horizontal bias setting of 0.2:

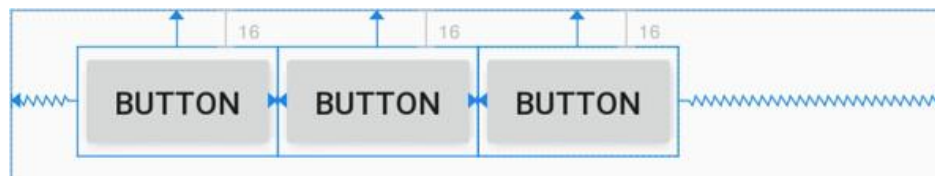


Figure 9-60

Weighted Chains

- The final area of chains to explore involves weighting of the individual widgets to control how much space each widget in the chain occupies within the available space.
- A weighted chain may only be implemented using the spread chain style and any widget within the chain that is to respond to the weight property must have the corresponding dimension property (height for a vertical chain and width for a horizontal chain) configured for match constraint mode.
- Match constraint mode for a widget dimension may be configured by selecting the widget, displaying the Attributes panel and changing the dimension to `match_constraint` (equivalent to 0dp).
- In Figure 9-61, for example, the `layout_width` constraint for a button has been set to `match_constraint` (0dp) to indicate that the width of the widget is to be determined based on the prevailing constraint settings:

(continued)

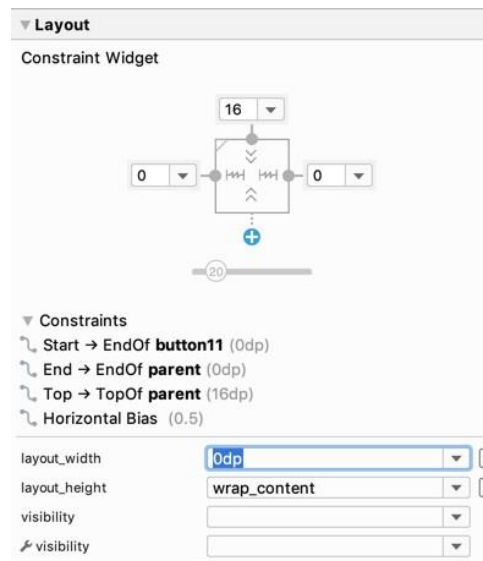


Figure 9-61

- Assuming that the spread chain style has been selected, and all three buttons have been configured such that the width dimension is set to match the constraints, the widgets in the chain will expand equally to fill the available space:



Figure 9-62

- The amount of space occupied by each widget relative to the other widgets in the chain can be controlled by adding weight properties to the widgets.
- Figure 9-63 shows the effect of setting the `layout_constraintHorizontal_weight` property to 4 on button1, and to 2 on both button2 and button3:

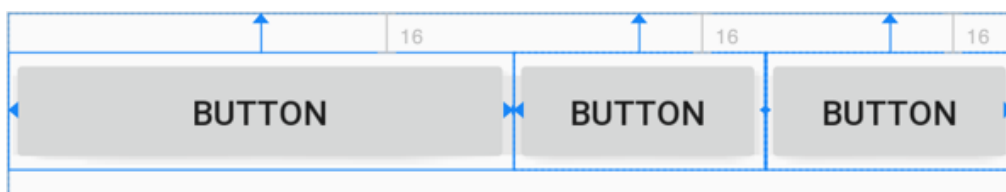


Figure 9-63

- As a result of these weighting values, button1 occupies half of the space ($4/8$), while button2 and button3 each occupy one quarter ($2/8$) of the space.

(continued)

Working with Ratios

- ConstraintLayout ratios allow one dimension of a widget to be sized relative to the widget's other dimension (otherwise known as aspect ratio).
- An aspect ratio setting could, for example, be applied to an ImageView to ensure that its width is always twice its height.
- A dimension ratio constraint is configured by setting the constrained dimension to match constraint mode and configuring the layout_constraintDimensionRatio attribute on that widget to the required ratio.
- This ratio value may be specified either as a float value or a width:height ratio setting. The following XML excerpt, for example, configures a ratio of 2:1 on an ImageView widget:

```
<ImageView
    android:layout_width="0dp"
    android:layout_height="100dp"
    android:id="@+id/imageView"
    app:layout_constraintDimensionRatio="2:1"
/>
```

- The above example demonstrates how to configure a ratio when only one dimension is set to match constraint.
- A ratio may also be applied when both dimensions are set to match constraint mode.
- This involves specifying the ratio preceded with either an H or a W to indicate which of the dimensions is constrained relative to the other.
- Consider, for example, the following XML excerpt for an ImageView object:

```
<ImageView
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:id="@+id/imageView"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintDimensionRatio="W,1:3"
/>
```

- In the above example the height will be defined subject to the constraints applied to it.
- In this case constraints have been configured such that it is attached to the top and bottom of the parent view, essentially stretching the widget to fill the entire height of the parent.
- The width dimension, on the other hand, has been constrained to be one third of the ImageView's height dimension.

- Consequently, whatever size screen or orientation the layout appears on, the ImageView will always be the same height as the parent and the width one third of that height.
- The same results may also be achieved without the need to manually edit the XML resource file.
- Whenever a widget dimension is set to match constraint mode, a ratio control toggle appears in the Inspector area of the property panel.
- Figure 9-64, for example, shows the layout width and height attributes of a button widget set to match constraint mode and 100dp respectively, and highlights the ratio control toggle in the widget sizing preview:

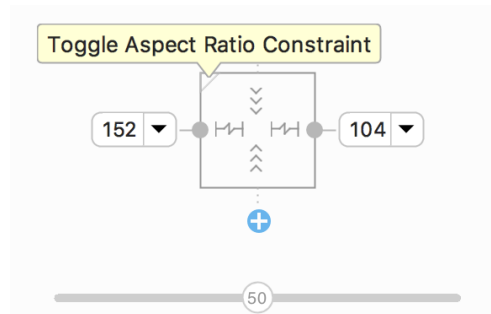


Figure 9-64

- By default the ratio sizing control is toggled off. Clicking on the control enables the ratio constraint and displays
- Working with ConstraintLayout Chains and Ratios in Android Studio an additional field where the ratio may be changed:

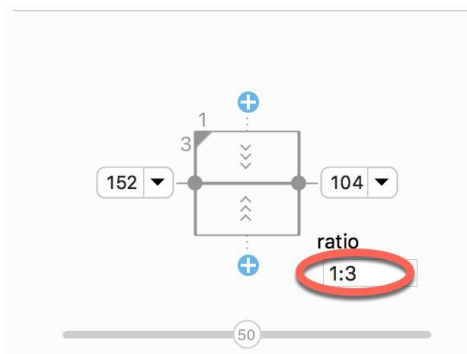
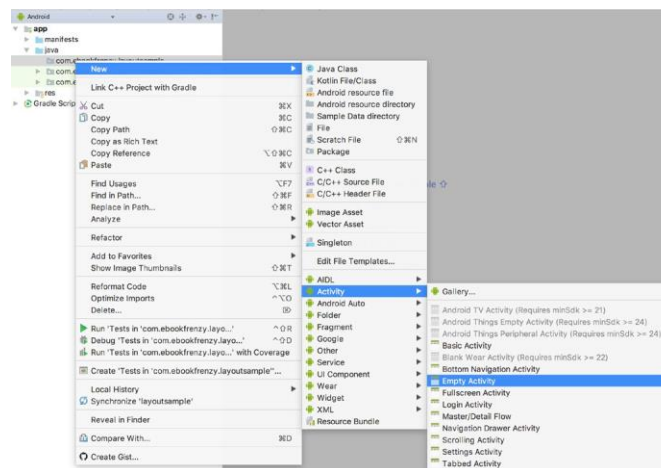


Figure 9-65

- By far the easiest and most productive way to design a user interface for an Android application is to make use of the Android Studio Layout Editor tool.
- The goal of this chapter is to provide an overview of how to create a ConstraintLayout-based user interface using this approach.
- The exercise included in this chapter will also be used as an opportunity to outline the creation of an activity starting with a “bare-bones” Android Studio project.

- Having covered the use of the Android Studio Layout Editor, the chapter will also introduce the Layout Inspector tool.
- **An Android Studio Layout Editor Tool Example**
- The first step in this phase of the example is to create a new Android Studio project. Begin, therefore, by launching Android Studio and closing any previously opened projects by selecting the File > Close Project menu option.
- Select the Start a new Android Studio project quick start option from the welcome screen. In previous examples, we have requested that Android Studio create a template activity for the project. We will, however, be using this tutorial to learn how to create an entirely new activity and corresponding layout resource file manually, so make sure that the No Activity option is selected before clicking on the Next button
- Enter LayoutSample into the Name field and specify edu.niu.your_Z-ID.layoutsample 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.
- Creating a New Activity
- Once the project creation process is complete, the Android Studio main window should appear. The first step in the project is to create a new activity. This will be a valuable learning exercise since there are many instances in the course of developing Android applications where new activities need to be created from the ground up.
- Begin by displaying the Project tool window if it is not already visible using the Alt-1/Cmd-1 keyboard shortcut. Once displayed, unfold the hierarchy by clicking on the right facing arrows next to the entries in the Project window. The objective here is to gain access to the app > java > edu.niu.your_Z-ID.layoutsample folder in the project hierarchy. Once the package name is visible, right-click on it and select the New > Activity > Empty Activity menu option as illustrated in Figure 9-66. Alternatively, select the New > Activity > Gallery... option to browse the available templates and make a selection using the New Android Activity dialog.



• Figure 9-66

- In the resulting New Android Activity dialog, name the new activity MainActivity and the layout activity_main.

- The activity will, of course, need a layout resource file so make sure that the Generate Layout File option is enabled.
- In order for an application to be able to run on a device it needs to have an activity designated as the launcher activity.
- Without a launcher activity, the operating system will not know which activity to start up when the application first launches and the application will fail to start.
- Since this example only has one activity, it needs to be designated as the launcher activity for the application so make sure that the Launcher Activity option is enabled before clicking on the Finish button.
- At this point Android Studio should have added two files to the project.
- The Java source code file for the activity should be located in the app > java > edu.niu.your_Z-ID.layoutsample folder.
- In addition, the XML layout file for the user interface should have been created in the app > res > layout folder.
- Note that the Empty Activity template was chosen for this activity so the layout is contained entirely within the activity_main.xml file and there is no separate content layout file.
- Finally, the new activity should have been added to the AndroidManifest.xml file and designated as the launcher activity.
- The manifest file can be found in the project window under the app > manifests folder and should contain the following XML:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    package="edu.niu.your_Z-ID.layoutsample">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true" android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Preparing the Layout Editor Environment

- Locate and double-click on the activity_main.xml layout file located in the app > res > layout folder to load it into the Layout Editor tool.
- Since the purpose of this tutorial is to gain experience with the use of constraints, turn off the Autoconnect feature using the button located in the Layout Editor toolbar.
- Once disabled, the button will appear with a line through it as is the case in Figure 9-67:

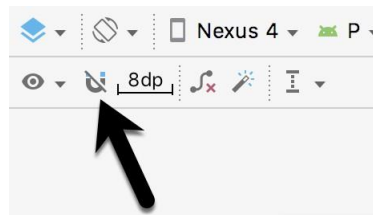


Figure 9-67

- If the default margin value to the right of the Autoconnect button is not set to 8dp, click on it and select 8dp from the resulting panel.
- The user interface design will also make use of the ImageView object to display an image.
- Before proceeding, this image should be added to the project ready for use later in the chapter.
- This file is named galaxy6s.png and can be found in the project_icons folder of the sample code download available from the following URL:

<https://www.ebookfrenzy.com/retail/androidstudio40/index.php>

- Within Android Studio, display the Resource Manager tool window (View > Tool Windows > Resource Manager).
- Locate the galaxy6s.png image in the file system navigator for your operating system and drag and drop the image onto the Resource Manager tool window.
- In the resulting dialog, click Next followed by the Import button to add the image to project and the image should now appear in the Resource Manager as shown in 68 below:

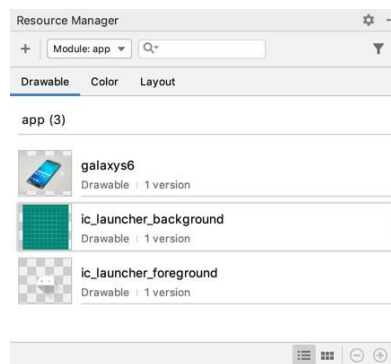


Figure 9-68

- The image will also appear in the res > drawables section of the Project tool window:

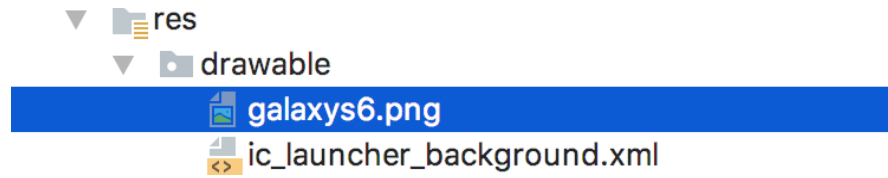


Figure 9-69

Adding the Widgets to the User Interface

- From within the Common palette category, drag an ImageView object into the center of the display view.
- Note that horizontal and vertical dashed lines appear indicating the center axes of the display.
- When centered, release the mouse button to drop the view into position.
- Once placed within the layout, the Resources dialog will appear seeking the image to be displayed within the view.
- In the search bar located at the top of the dialog, enter “galaxy” to locate the galaxys6.png resource as illustrated in Figure 9-70.

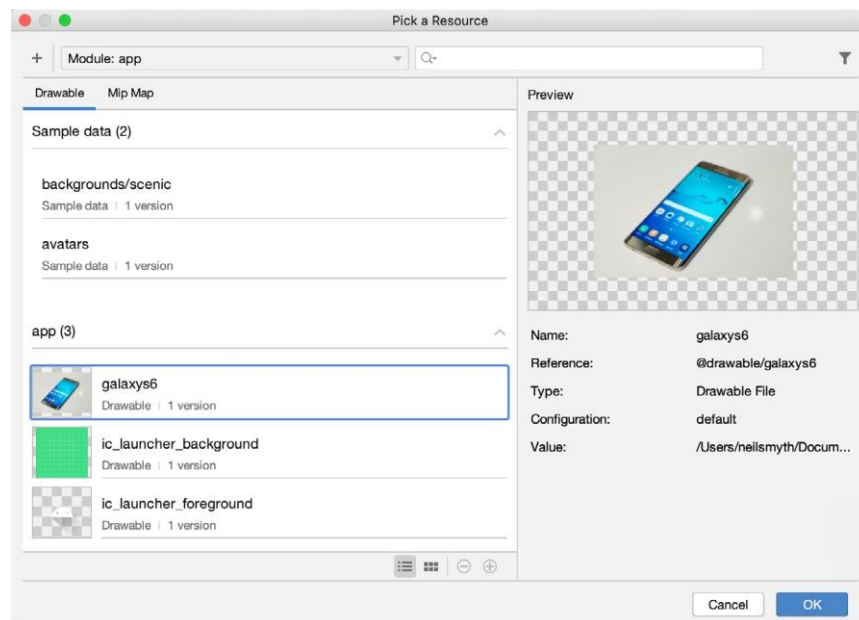


Figure 9-70

- Select the image and click on OK to assign it to the ImageView object.
- If necessary, adjust the size of the ImageView using the resize handles and reposition it in the center of the layout.
- At this point the layout should match Figure 9-71:

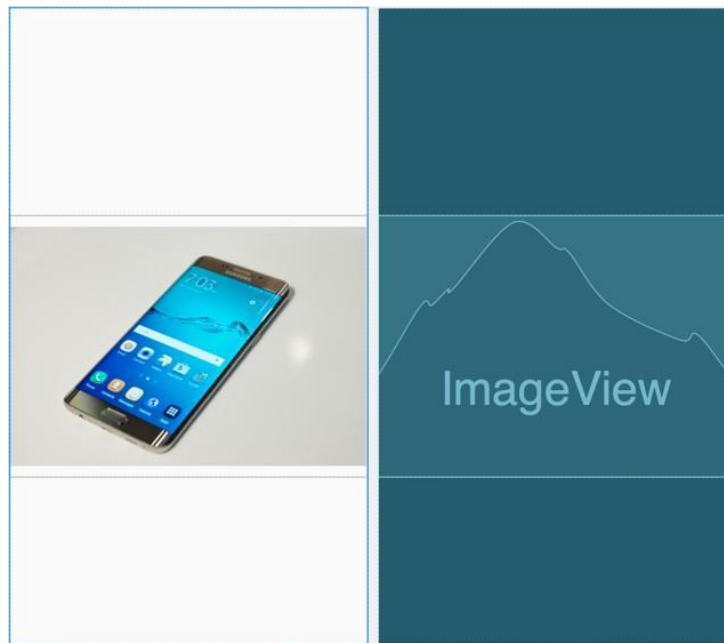


Figure 9-71

- Click and drag a TextView object from the Common section of the palette and position it so that it appears above the ImageView as illustrated in Figure 9-72.
- Using the Attributes panel, unfold the textAppearance attribute entry in the Common Attributes section, change the textSize property to 24sp, the textAlignment setting to center and the text to “Samsung Galaxy S6”.

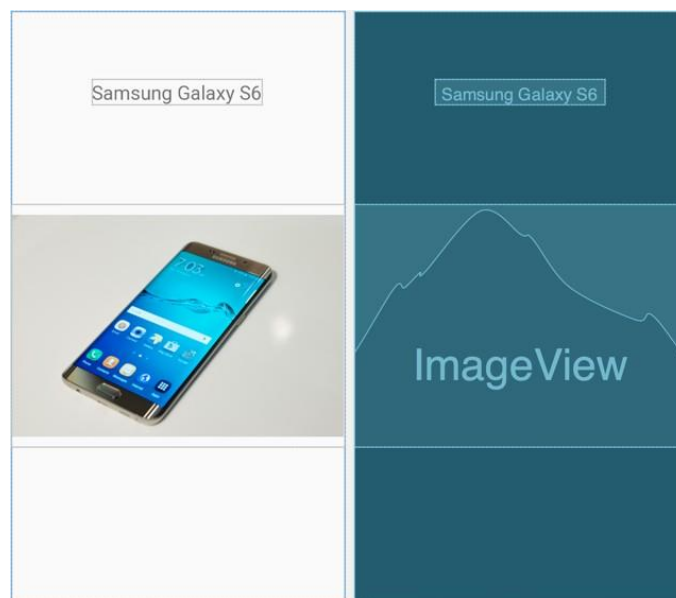


Figure 9-72

- Next, add three Button widgets along the bottom of the layout and set the text attributes of these views to “Buy Now”, “Pricing” and “Details”. The completed layout should now match Figure 9-73:

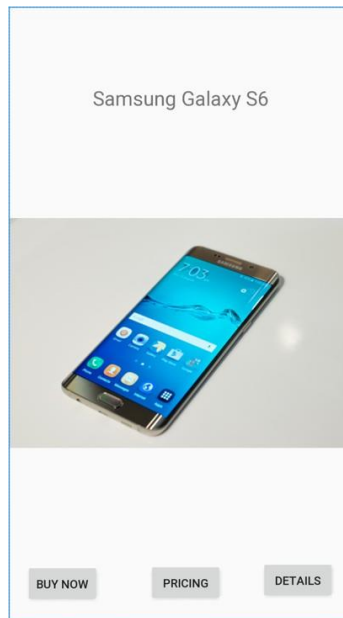


Figure 9-73

- At this point, the widgets are not sufficiently constrained for the layout engine to be able to position and size the widgets at runtime.
- Were the app to run now, all of the widgets would be positioned in the top left-hand corner of the display.
- With the widgets added to the layout, use the device rotation button located in the Layout Editor toolbar (indicated by the arrow in Figure 9-74) to view the user interface in landscape orientation:

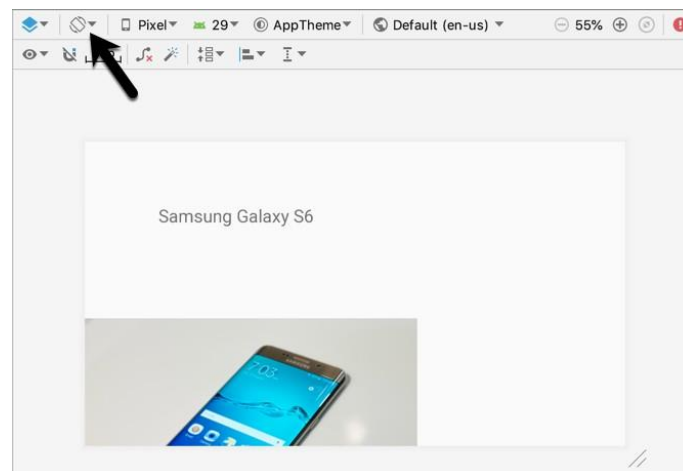


Figure 9-74

- The absence of constraints results in a layout that fails to adapt to the change in device orientation, leaving the content off center and with part of the image and all three buttons positioned beyond the viewable area of the screen.
- Clearly some work still needs to be done to make this into a responsive user interface.

Adding the Constraints

- Constraints are the key to creating layouts that can adapt to device orientation changes and different screen sizes.
- Begin by rotating the layout back to portrait orientation and selecting the TextView widget located above the ImageView.
- With the widget selected, establish constraints from the left, right and top sides of the TextView to the corresponding sides of the parent ConstraintLayout as shown in Figure 9-75:



Figure 9-75

- With the TextView widget constrained, select the ImageView instance and establish opposing constraints on the left and right-hand sides with each connected to the corresponding sides of the parent layout.
- Next, establish a constraint connection from the top of the ImageView to the bottom of the TextView and from the bottom of the ImageView to the top of the center Button widget.
- If necessary, click and drag the ImageView so that it is still positioned in the vertical center of the layout.
- Next, establish constraints on the left and right hand sides of the ImageView to the corresponding sides of the ConstraintLayout container.
- With the ImageView still selected, use the Inspector in the attributes panel to change the top and bottom margins on the ImageView to 24 and 8 respectively and to change both the widget height and width dimension properties to match_constraint so that the widget will resize to match the constraints.
- These settings will allow the layout engine to enlarge and reduce the size of the ImageView when necessary to accommodate layout changes:

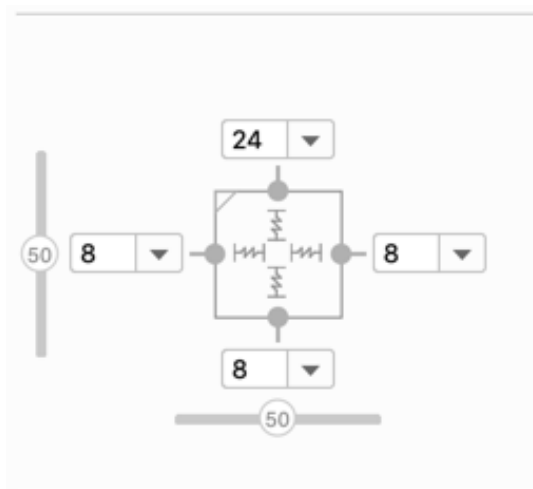


Figure 9-76

- Figure 9-77, shows the currently implemented constraints for the ImageView in relation to the other elements in the layout:

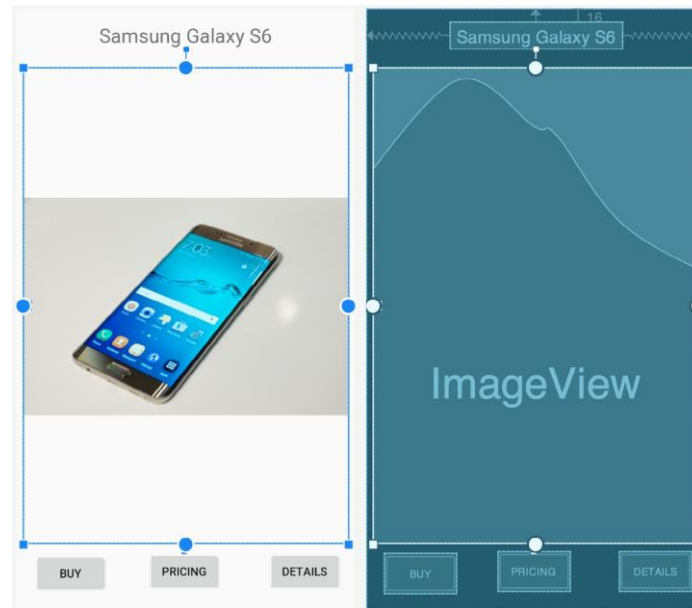


Figure 9-77

- The final task is to add constraints to the three Button widgets.
- For this example, the buttons will be placed in a chain.
- Begin by turning on Autoconnect within the Layout Editor by clicking the toolbar button highlighted earlier.
- Next, click on the Buy Now button and then shift-click on the other two buttons so that all three are selected.
- Right-click on the Buy Now button and select the Chains > Create Horizontal Chain menu option from the resulting menu.
- By default, the chain will be displayed using the spread style which is the correct behavior for this example.
- Finally, establish a constraint between the bottom of the Buy Now button and the bottom of the layout.
- Repeat this step for the remaining buttons.
- On completion of these steps the buttons should be constrained as outlined in Figure 9-78:



Figure 9-78

Testing the Layout

- With the constraints added to the layout, rotate the screen into landscape orientation and verify that the layout adapts to accommodate the new screen dimensions.
- While the Layout Editor tool provides a useful visual environment in which to design user interface layouts, when it comes to testing there is no substitute for testing the running app.
- Launch the app on a physical Android device or emulator session and verify that the user interface reflects the layout created in the Layout Editor.
- An Android Studio Layout Editor ConstraintLayout Tutorial Figure 9-79, for example, shows the running app in landscape orientation:

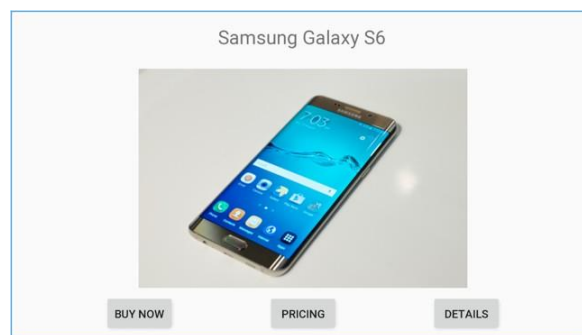


Figure 9-79

- The user interface design is now complete.
- Designing a more complex user interface layout is a continuation of the steps outlined above.
- Simply drag and drop views onto the display, position, constrain and set properties as needed.

Using the Layout Inspector

- The hierarchy of components that make up a user interface layout may be viewed at any time using the Layout Inspector tool.
- In order to access this information the app must be running on a device or emulator.
- Once the app is running, select the Tools > Layout Inspector menu option followed by the process to be inspected using the menu marked A in Figure 9-80 below).
- Once the inspector loads, the left most panel (B) shows the hierarchy of components that make up the user interface layout.
- The center panel (C) shows a visual representation of the layout design.
- Clicking on a widget in the visual layout will cause that item to highlight in the hierarchy list making it easy to find where a visual component is situated relative to the overall layout hierarchy.

- Finally, the rightmost panel (marked D in Figure 9-80) contains all of the property settings for the currently selected component, allowing for in-depth analysis of the component's internal configuration.
- Where appropriate, the value cell will contain a link to the location of the property setting within the project source code.

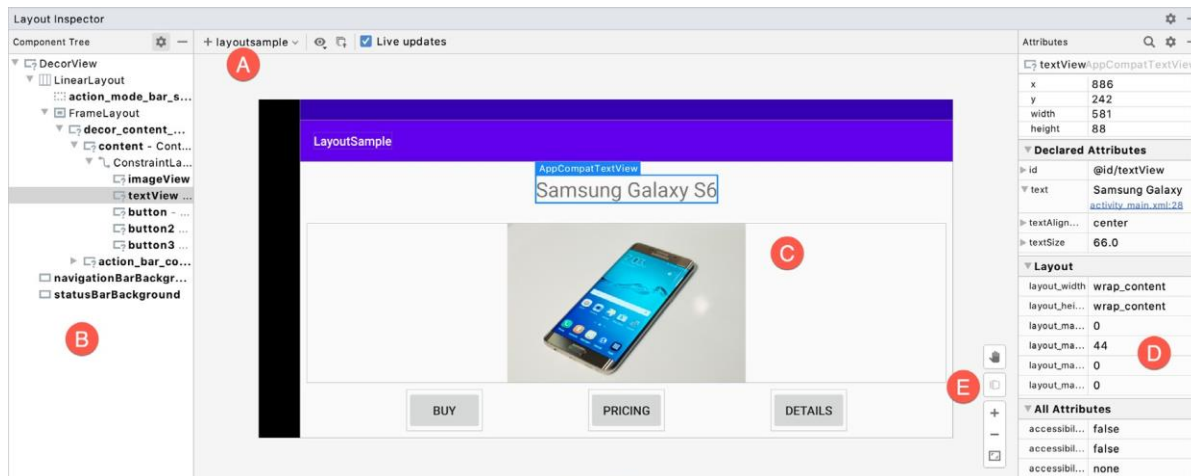


Figure 9-80

- The button marked E switches the view to 3D mode, “exploding” the hierarchy so that it can be rotated and inspected.
- This can be useful for tasks such as identifying obscured views:

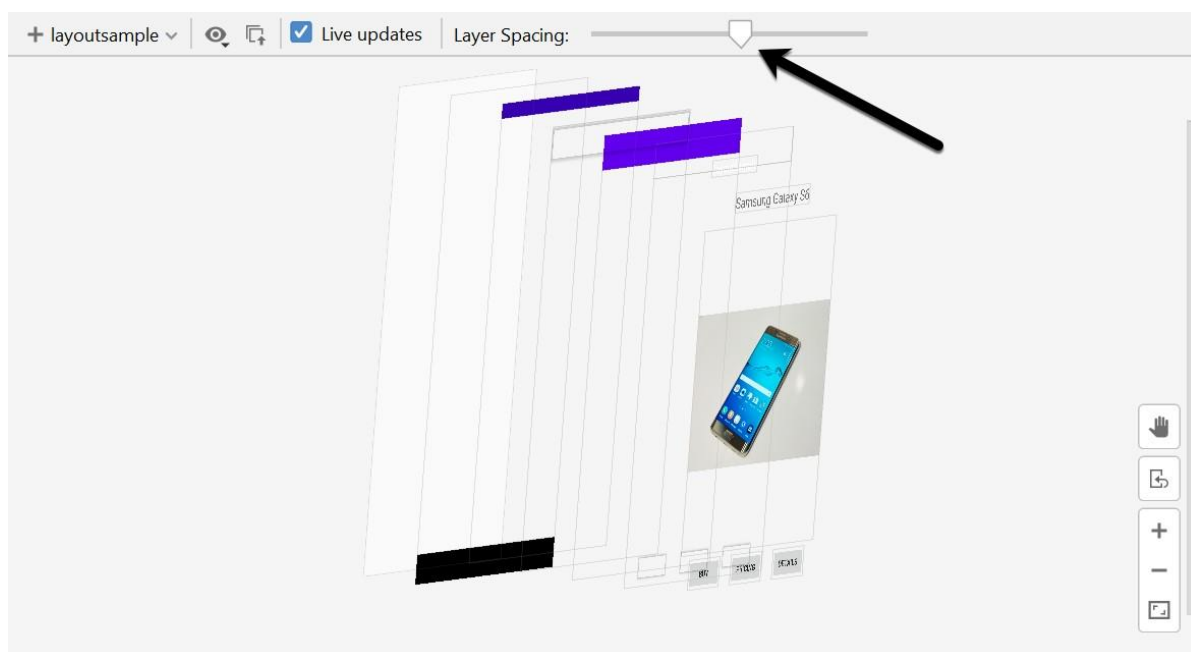


Figure 9-81

- Click and drag the rendering to rotate it in three dimensions, using the slider indicated by the arrow in the above figure to increase the spacing between the layers.

- Click the button marked E in Figure 9-81 above to switch back to 2D mode.

Summary

- ConstraintLayout is a layout manager introduced with Android 7.
- It is designed to ease the creation of flexible layouts that adapt to the size and orientation of the many Android devices now on the market.
- ConstraintLayout uses constraints to control the alignment and positioning of widgets in relation to the parent ConstraintLayout instance, guidelines, barriers and the other widgets in the layout.
- ConstraintLayout is the default layout for newly created Android Studio projects and is the recommended choice when designing user interface layouts.
- With this simple yet flexible approach to layout management, complex and responsive user interfaces can be implemented with surprising ease.
- A redesigned Layout Editor tool combined with ConstraintLayout makes designing complex user interface layouts with Android Studio a relatively fast and intuitive process.
- This chapter has covered the concepts of constraints, margins and bias in more detail while also exploring the ways in which ConstraintLayout-based design has been integrated into the Layout Editor tool.
- Both chains and ratios are powerful features of the ConstraintLayout class intended to provide additional options for designing flexible and responsive user interface layouts within Android applications.
- As outlined in this chapter, the Android Studio Layout Editor has been enhanced to make it easier to use these features during the user interface design process.
- The Layout Editor tool in Android Studio has been tightly integrated with the ConstraintLayout class.
- This chapter has worked through the creation of an example user interface intended to outline the ways in which a ConstraintLayout-based user interface can be implemented using the Layout Editor tool in terms of adding widgets and setting constraints.
- This chapter also introduced the Live Layout Inspector tool which is useful for analyzing the structural composition of a user interface layout.