

17. A Guide to the Android ConstraintLayout

As discussed in the chapter entitled [“A Guide to the Android ConstraintLayout”](#), Android provides a number of layout managers for the purpose of designing user interfaces. With Android 7, Google has 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 how constraint-based layouts can be created using ConstraintLayout within the Android Studio Layout Editor tool.

17.1 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

17.1.1 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 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 its position can be resolved by the `ConstraintLayout` layout engine in both the horizontal and vertical planes.

17.1.2 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 17-1](#):

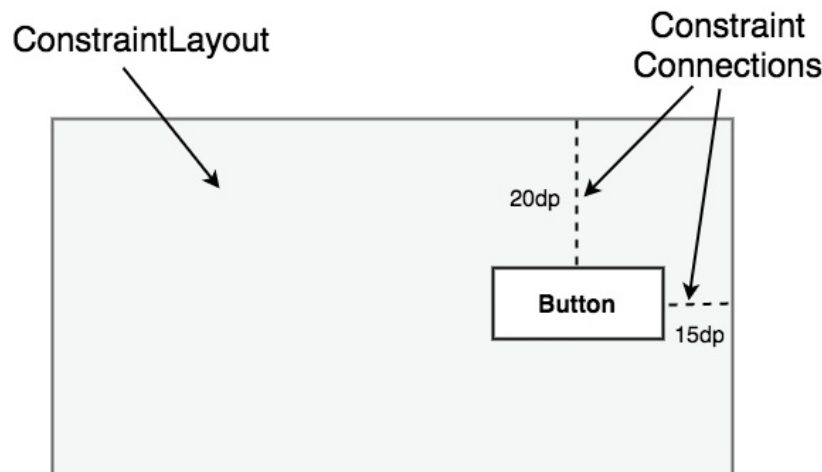


Figure 17-1

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.

17.1.3 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 17-2](#), for example, illustrates the addition of both horizontally and vertically opposing constraints to the previous layout:

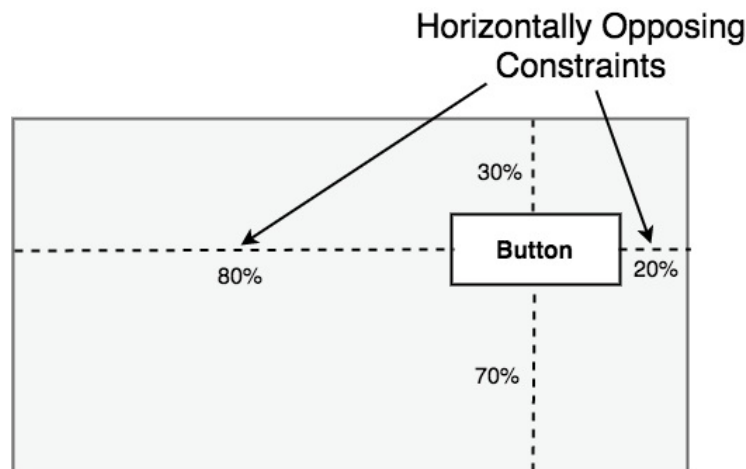


Figure 17-2

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 17-2](#) has been implemented using not only opposing constraints, but also by applying *constraint bias*.

17.1.4 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 17-3](#), for example, shows a widget centered within the containing `ConstraintLayout` using opposing horizontal and vertical constraints:

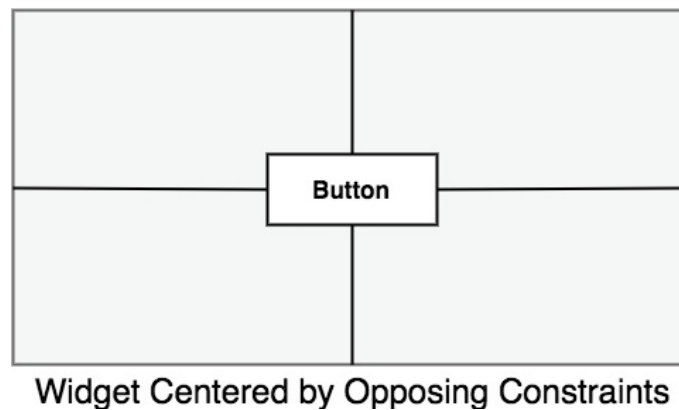


Figure 17-3

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 17-4](#), for example, shows the previous constraint layout with a 75% horizontal bias and 10% vertical bias:

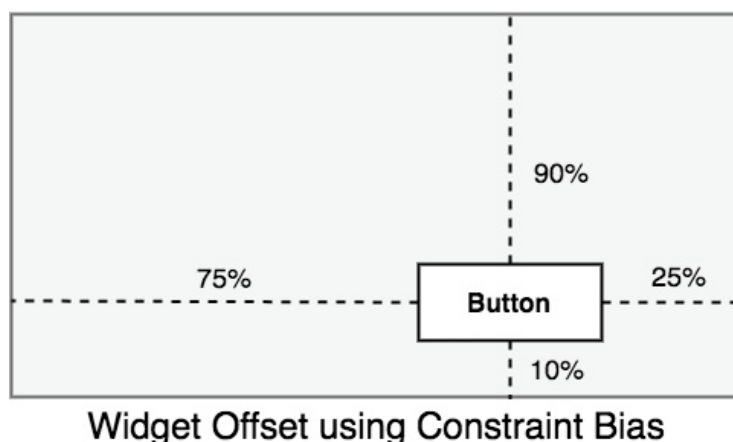


Figure 17-4

The next chapter, entitled [“A Guide to using ConstraintLayout in Android Studio”](#), 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.

17.1.5 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 17-5](#), for example, illustrates three widgets chained in this way:

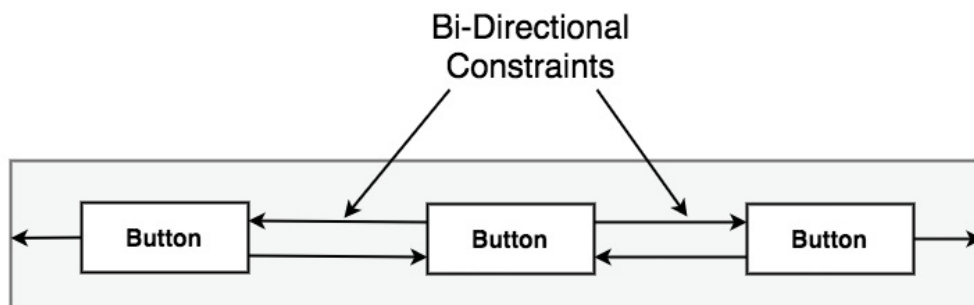


Figure 17-5

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.

17.1.6 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.

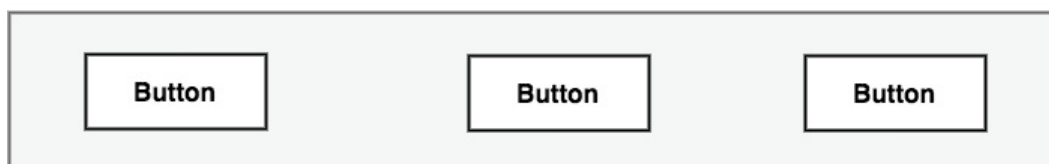


Figure 17-6

- **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 17-7

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

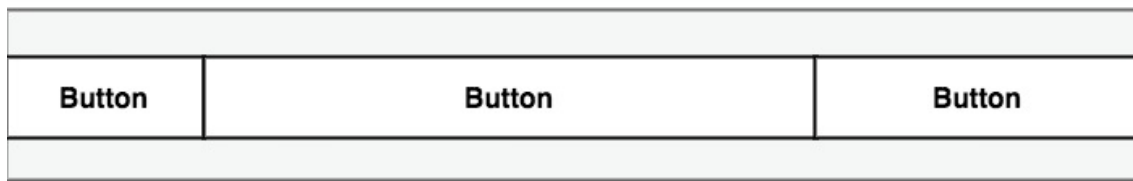


Figure 17-8

- **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.

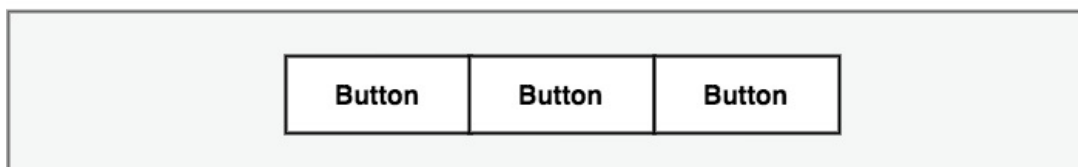


Figure 17-9

17.2 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 17-1](#) 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 17-10](#):

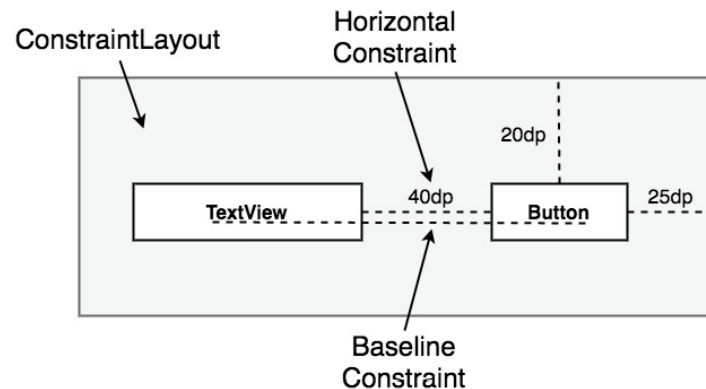


Figure 17-
10

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

17.3 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 17-11](#), for example, three `Button` objects contained within a `ConstraintLayout` are constrained along a vertical guideline:

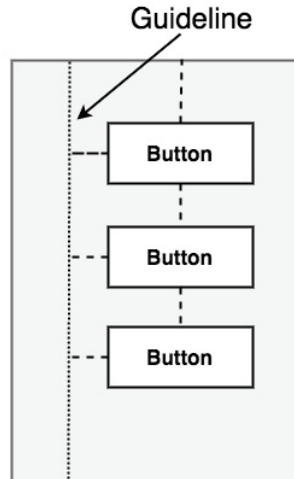


Figure 17-
11

17.4 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).

17.5 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 17-12](#) below:

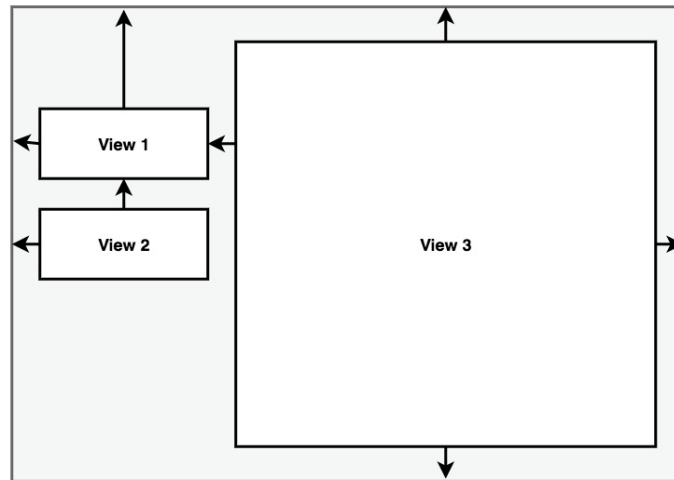


Figure 17-
12

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:

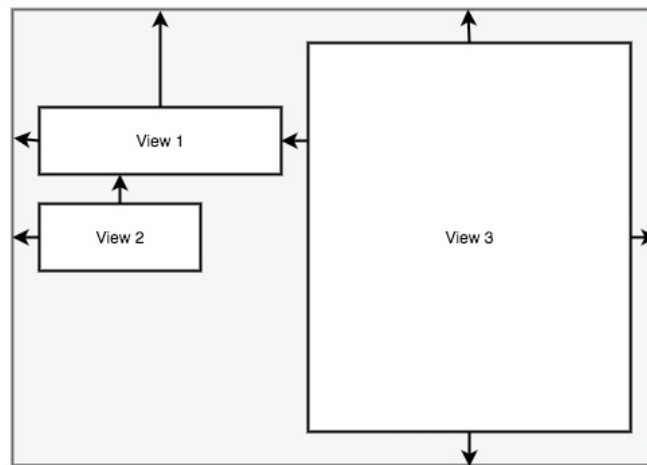


Figure 17-
13

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

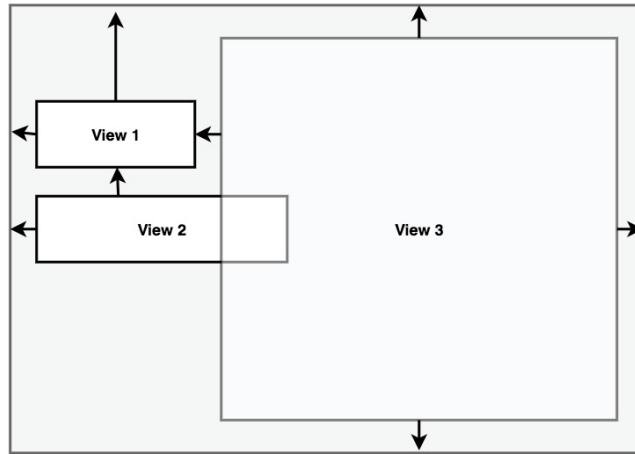


Figure 17-
14

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:

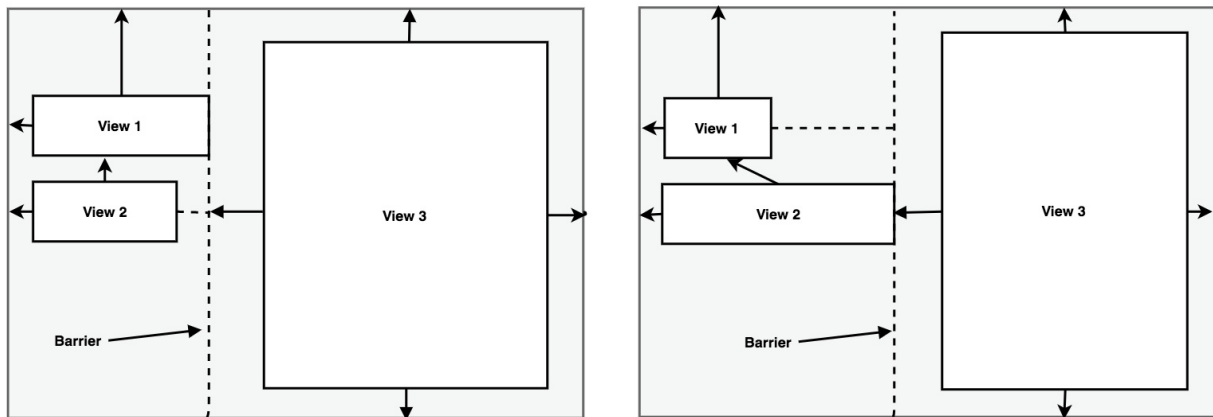


Figure 17-
15

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.

17.6 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.

17.7 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 the chapter entitled [*“A Guide to using ConstraintLayout in Android Studio”*](#), the Android Studio Layout Editor tool has been enhanced specifically for ConstraintLayout-based user interface design.

17.8 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.

17.9 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.