**Build a Responsive UI with ConstraintLayout   Part of** [**Android Jetpack**](https://developer.android.com/jetpack)**.**

[ConstraintLayout](https://developer.android.com/reference/android/support/constraint/ConstraintLayout.html) allows you to create large and complex layouts with a flat view hierarchy (no nested view groups). It's similar to [RelativeLayout](https://developer.android.com/reference/android/widget/RelativeLayout.html) in that all views are laid out according to relationships between sibling views and the parent layout, but it's more flexible than RelativeLayout and easier to use with Android Studio's Layout Editor.

All the power of ConstraintLayout is available directly from the Layout Editor's visual tools, because the layout API and the Layout Editor were specially built for each other. So you can build your layout with ConstraintLayout entirely by drag-and-dropping instead of editing the XML.

[ConstraintLayout](https://developer.android.com/reference/android/support/constraint/ConstraintLayout.html) is available in an API library that's compatible with Android 2.3 (API level 9) and higher. This page provides a guide to building a layout with ConstraintLayout in Android Studio 3.0 or higher. If you'd like more information about the Layout Editor itself, see the Android Studio guide to [Build a UI with Layout Editor](https://developer.android.com/studio/write/layout-editor.html).

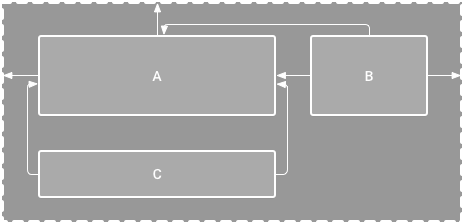
To see a variety of layouts you can create with ConstraintLayout, check out the [Constraint Layout Examples project on GitHub](https://github.com/googlesamples/android-ConstraintLayoutExamples).

**Constraints overview**

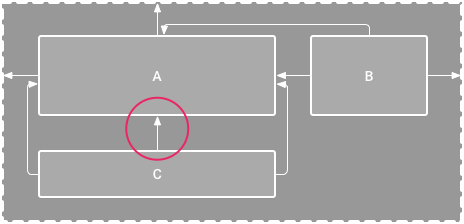
To define a view's position in ConstraintLayout, you must add at least one horizontal and one vertical constraint for the view. Each constraint represents a connection or alignment to another view, the parent layout, or an invisible guideline. Each constraint defines the view's position along either the vertical or horizontal axis; so each view must have a minimum of one constraint for each axis, but often more are necessary.

When you drop a view into the Layout Editor, it stays where you leave it even if it has no constraints. However, this is only to make editing easier; if a view has no constraints when you run your layout on a device, it is drawn at position [0,0] (the top-left corner).

In figure 1, the layout looks good in the editor, but there's no vertical constraint on view C. When this layout draws on a device, view C horizontally aligns with the left and right edges of view A, but appears at the top of the screen because it has no vertical constraint.



**Figure 1.** The editor shows view C below A, but it has no vertical constraint



**Figure 2.** View C is now vertically constrained below view A

Although a missing constraint won't cause a compilation error, the Layout Editor indicates missing constraints as an error in the toolbar. To view the errors and other warnings, click **Show Warnings and Errors** https://developer.android.com/studio/images/buttons/layout-editor-errors.png. To help you avoid missing constraints, the Layout Editor can automatically add constraints for you with the [Autoconnect and infer constraints](https://developer.android.com/training/constraint-layout#use-autoconnect-and-infer-constraints) features.

**Add ConstraintLayout to your project**

To use ConstraintLayout in your project, proceed as follows:

1. Ensure you have the maven.google.com repository declared in your module-level build.gradle file:

repositories {  
    google()  
}

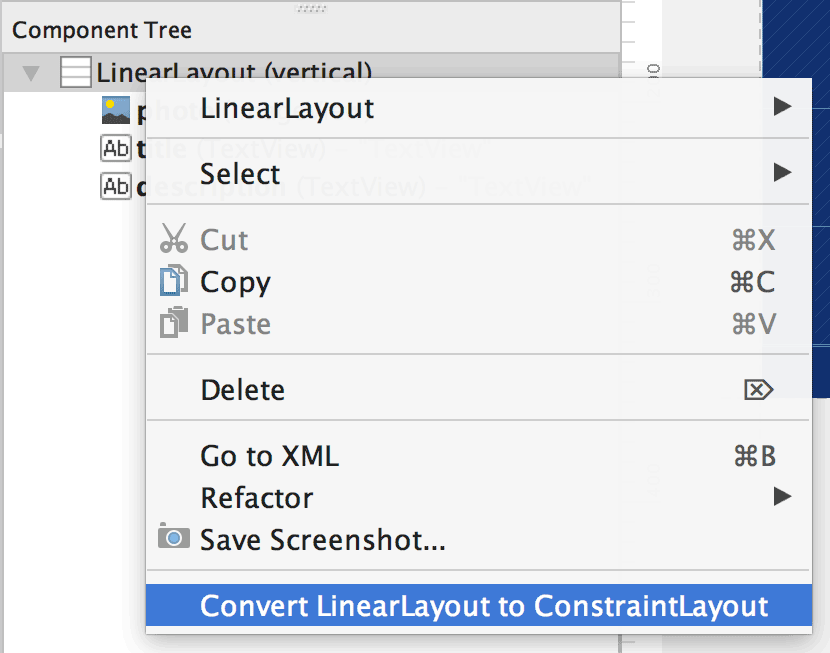
  Add the library as a dependency in the same build.gradle file, as shown in the example below. Note that the latest version might be different than what is shown in the example:

dependencies {  
    implementation 'com.android.support.constraint:constraint-layout:1.1.2'  
}

1. In the toolbar or sync notification, click **Sync Project with Gradle Files**.

Now you're ready to build your layout with ConstraintLayout.

**Convert a layout**



**Figure 3.** The menu to convert a layout to ConstraintLayout

To convert an existing layout to a constraint layout, follow these steps:

1. Open your layout in Android Studio and click the **Design** tab at the bottom of the editor window.
2. In the **Component Tree** window, right-click the layout and click **Convert *layout* to ConstraintLayout**.

**Create a new layout**

To start a new constraint layout file, follow these steps:

1. In the **Project** window, click the module folder and then select **File > New > XML > Layout XML**.
2. Enter a name for the layout file and enter "android.support.constraint.ConstraintLayout" for the **Root Tag**.
3. Click **Finish**.

**Add or remove a constraint**

To add a constraint, do the following:

**Video 1.** The left side of a view is constrained to the left side of the parent

1. Drag a view from the **Palette** window into the editor.

When you add a view in a ConstraintLayout, it displays a bounding box with square resizing handles on each corner and circular constraint handles on each side.

1. Click the view to select it.
2. Do one of the following:
   * Click a constraint handle and drag it to an available anchor point (the edge of another view, the edge of the layout, or a guideline).
   * Click **Create a connection** https://developer.android.com/studio/images/buttons/attributes-plus-icon_2x.pngin the view inspector at the top of the **Attributes** window.

When the constraint is created, the editor gives it a [default margin](https://developer.android.com/training/constraint-layout#adjust-the-view-margins) to separate the two views.

When creating constraints, remember the following rules:

* Every view must have at least two constraints: one horizontal and one vertical.
* You can create constraints only between a constraint handle and an anchor point that share the same plane. So a vertical plane (the left and right sides) of a view can be constrained only to another vertical plane; and baselines can constrain only to other baselines.
* Each constraint handle can be used for just one constraint, but you can create multiple constraints (from different views) to the same anchor point.

**Video 2.** Adding a constraint that opposes an existing one

To remove a constraint, select the view and then click the constraint handle. Or remove all the constraints by selecting the view and then clicking **Delete Constraints** https://developer.android.com/studio/images/buttons/layout-editor-action-clear.png.

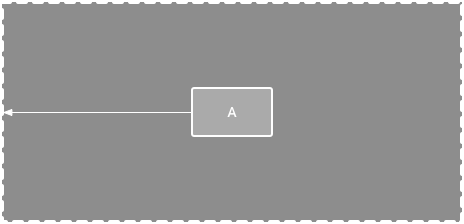
If you add opposing constraints on a view, the constraint lines become squiggly like a spring to indicate the opposing forces, as shown in video 2. The effect is most visible when the view size is set to "fixed" or "wrap content," in which case the view is centered between the constraints. If you instead want the view to stretch its size to meet the constraints, [switch the size to "match constraints"](https://developer.android.com/training/constraint-layout#adjust-the-view-size); or if you want to keep the current size but move the view so that it is not centered, [adjust the constraint bias](https://developer.android.com/training/constraint-layout#adjust-the-constraint-bias).

You can use constraints to achieve different types of layout behavior, as described in the following sections.

**Parent position**

Constrain the side of a view to the corresponding edge of the layout.

In figure 4, the left side of the view is connected to the left edge of the parent layout. You can define the distance from the edge with margin.

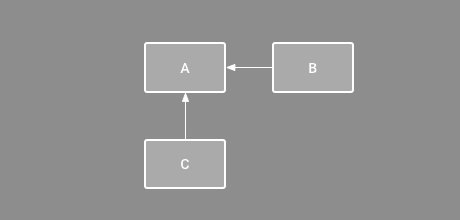


**Figure 4.** A horizontal constraint to the parent

**Order position**

Define the order of appearance for two views, either vertically or horizontally.

In figure 5, B is constrained to always be to the right of A, and C is constrained below A. However, these constraints do not imply alignment, so B can still move up and down.



**Figure 5.** A horizontal and vertical constraint

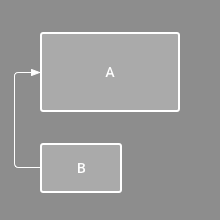
**Alignment**

Align the edge of a view to the same edge of another view.

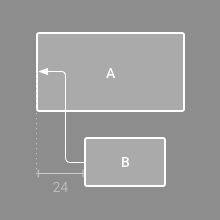
In figure 6, the left side of B is aligned to the left side of A. If you want to align the view centers, create a constraint on both sides.

You can offset the alignment by dragging the view inward from the constraint. For example, figure 7 shows B with a 24dp offset alignment. The offset is defined by the constrained view's margin.

You can also select all the views you want to align, and then click **Align** https://developer.android.com/studio/images/buttons/layout-editor-align.pngin the toolbar to select the alignment type.



**Figure 6.** A horizontal alignment constraint



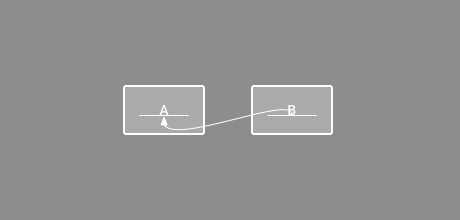
**Figure 7.** An offset horizontal alignment constraint

**Baseline alignment**

Align the text baseline of a view to the text baseline of another view.

In figure 8, the first line of B is aligned with the text in A.

To create a baseline constraint, select the text view you want to constrain and then click **Edit Baseline** https://developer.android.com/studio/images/buttons/layout-editor-action-baseline.png, which appears below the view. Then click the text baseline and drag the line to another baseline.



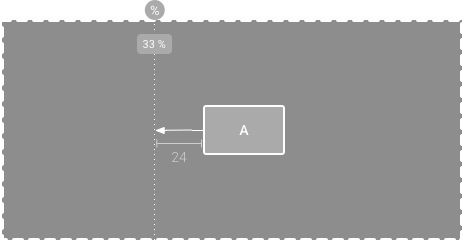
**Figure 8.** A baseline alignment constraint

**Constrain to a guideline**

You can add a vertical or horizontal guideline to which you can constrain views, and the guideline will be invisible to app users. You can position the guideline within the layout based on either dp units or percent, relative to the layout's edge.

To create a guideline, click **Guidelines** https://developer.android.com/studio/images/buttons/layout-editor-guidelines.pngin the toolbar, and then click either **Add Vertical Guideline** or **Add Horizontal Guideline**.

Drag the dotted line to reposition it and click the circle at the edge of the guideline to toggle the measurement mode.



**Figure 9.** A view constrained to a guideline

**Constrain to a barrier**

Similar to a guideline, a barrier is an invisible line that you can constrain views to. Except a barrier does not define its own position; instead, the barrier position moves based on the position of views contained within it. This is useful when you want to constrain a view to the a set of views rather than to one specific view.

For example, figure 10 shows view C is constrained to the right side of a barrier. The barrier is set to the "end" (or the right side in a left-to-right layout) of both view A and view B. So the barrier moves depending on whether the right side of view A or view B is is farthest right.

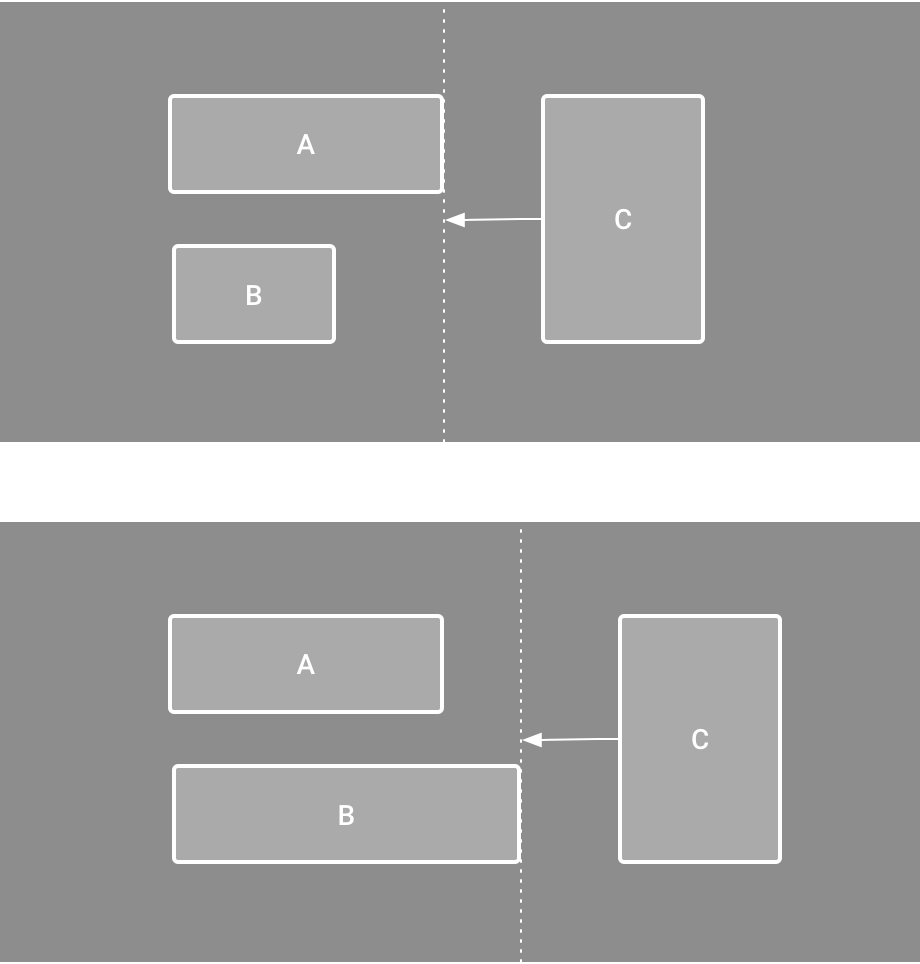
To create a barrier, follow these steps:

1. Click **Guidelines** https://developer.android.com/studio/images/buttons/layout-editor-guidelines.pngin the toolbar, and then click **Add Vertical Barrier** or **Add Horizontal Barrier**.
2. In the **Component Tree** window, select the views you want inside the barrier and drag them into the barrier component.
3. Select the barrier from the **Component Tree**, open the **Attributes** https://developer.android.com/studio/images/buttons/window-properties.pngwindow, and then set the **barrierDirection**.

Now you can create a constraint from another view to the barrier.

You can also constrain views that are *inside* the barrier to the barrier. This way, you can ensure that all views in the barrier always align to each other, even if you don't know which view will be the longest or tallest.

You can also include a guideline inside a barrier to ensure a "minimum" position for the barrier.



**Figure 10.** View C is constrained to a barrier, which moves based on the position/size of both view A and view B

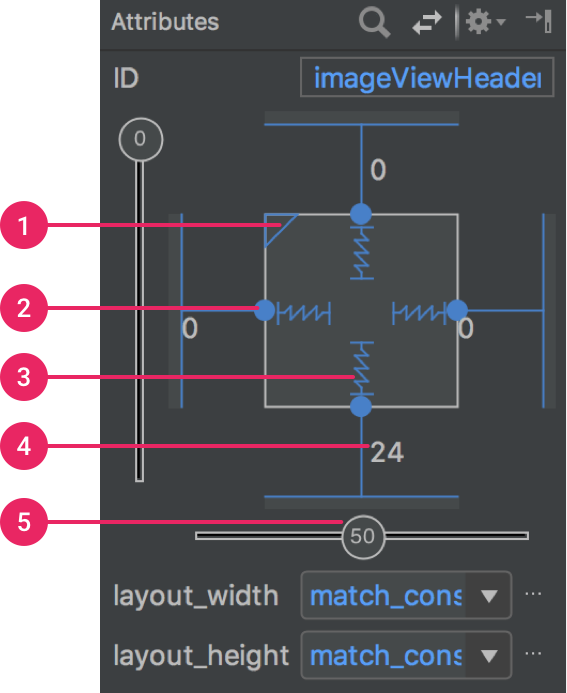
**Adjust the constraint bias**

When you add a constraint to both sides of a view (and the view size for the same dimension is either "fixed" or "wrap content"), the view becomes centered between the two constraints with a bias of 50% by default. You can adjust the bias by dragging the bias slider in the **Attributes** window or by dragging the view, as shown in video 5.

If you instead want the view to stretch its size to meet the constraints, [switch the size to "match constraints"](https://developer.android.com/training/constraint-layout#adjust-the-view-size).

**Video 5.** Adjusting the constraint bias

**Adjust the view size**



**Figure 11.** The **Attributes** window includes controls for 1 size ratio, 2 delete constraint, 3 height/width mode, 4 margins, and 5 constraint bias.

You can use the corner handles to resize a view, but this hard codes the size so the view will not resize for different content or screen sizes. To select a different sizing mode, click a view and open the **Attributes** https://developer.android.com/studio/images/buttons/window-properties.pngwindow on the right side of the editor.

Near the top of the **Attributes** window is the view inspector, which includes controls for several layout attributes, as shown in figure 11 (this is available only for views in a constraint layout).

You can change the way the height and width are calculated by clicking the symbols indicated with callout 3 in figure 11. These symbols represent the size mode as follows (click the symbol to toggle between these settings):

* https://developer.android.com/studio/images/buttons/layout-width-fixed.png**Fixed**: You specify a specific dimension in the text box below or by resizing the view in the editor.
* https://developer.android.com/studio/images/buttons/layout-width-wrap.png**Wrap Content**: The view expands only as much as needed to fit its contents.
* https://developer.android.com/studio/images/buttons/layout-width-match.png**Match Constraints**: The view expands as much as possible to meet the constraints on each side (after accounting for the view's margins). However, you can modify that behavior with the following attributes and values (these attributes take effect only when you set the view width to match constraints):
  + **layout\_constraintWidth\_default**
    - **spread**: Expands the view as much as possible to meet the constraints on each side. This is the default behavior.
    - **wrap**: Expands the view only as much as needed to fit its contents, but still allows the view to be smaller than that if the constraints require it. So the difference between this and using **Wrap Content** (above), is that setting the width to **Wrap Content** forces the width to always exactly match the content width; whereas using **Match Constraints** with **layout\_constraintWidth\_default** set to **wrap** also allows the view to be smaller than the content width.
  + **layout\_constraintWidth\_min**

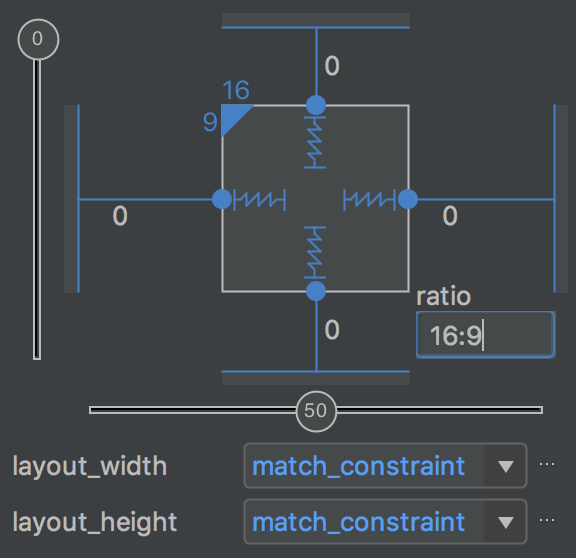
This takes a dp dimension for the view's minimum width.

* + **layout\_constraintWidth\_max**

This takes a dp dimension for the view's maximum width.

* However, if the given dimension has only one constraint, then the view expands to fit its contents. Using this mode on either the height or width also allows you to [set a size ratio](https://developer.android.com/training/constraint-layout#ratio).

**Note**: You cannot use match\_parent for any view in a ConstraintLayout. Instead use "match constraints" (0dp).



**Figure 12.** The view is set to a 16:9 aspect with the width based on a ratio of the height.

**Set size as a ratio**

You can set the view size to a ratio such as 16:9 if at least one of the view dimensions is set to "match constraints" (0dp). To enable the ratio, click **Toggle Aspect Ratio Constraint** (callout 1 in figure 11), and then enter the *width*:*height* ratio in the input that appears.

If both the width and height are set to match constraints, you can click **Toggle Aspect Ratio Constraint** to select which dimension is based on a ratio of the other. The view inspector indicates which is set as a ratio by connecting the corresponding edges with a solid line.

For example, if you set both sides to "match constraints", click **Toggle Aspect Ratio Constraint** twice to set the width be a ratio of the height. Now the entire size is dictated by the height of the view (which can be defined in any way) as shown in figure 12.

**Adjust the view margins**

To ensure that all your views are evenly spaced, click **Margin** https://developer.android.com/studio/images/buttons/layout-editor-margin.pngin the toolbar to select the default margin for each view that you add to the layout. Any change you make to the default margin applies only to the views you add from then on.

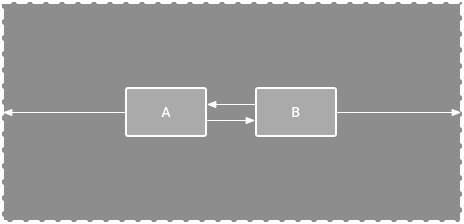
You can control the margin for each view in the **Attributes** window by clicking the number on the line that represents each constraint (in figure 11, callout 4 shows the bottom margin is set to 24dp).



**Figure 13.** The toolbar's **Margin** button.

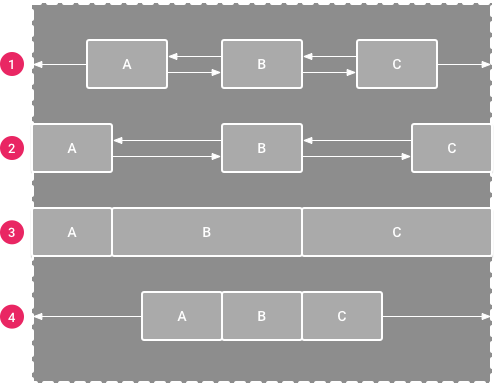
All margins offered by the tool are factors of 8dp to help your views align to Material Design's [8dp square grid recommendations](https://material.google.com/layout/metrics-keylines.html).

**Control linear groups with a chain**



**Figure 14.** A horizontal chain with two views

A chain is a group of views that are linked to each other with bi-directional position constraints. The views within a chain can be distributed either vertically or horizontally.



**Figure 15.** Examples of each chain style

Chains can be styled in one of the following ways:

1. **Spread:** The views are evenly distributed (after margins are accounted for). This is the default.
2. **Spread inside:** The first and last view are affixed to the constraints on each end of the chain and the rest are evenly distributed.
3. **Weighted:** When the chain is set to either **spread** or **spread inside**, you can fill the remaining space by setting one or more views to "match constraints" (0dp). By default, the space is evenly distributed between each view that's set to "match constraints," but you can assign a weight of importance to each view using the layout\_constraintHorizontal\_weight and layout\_constraintVertical\_weight attributes. If you're familiar with layout\_weight in a [linear layout](https://developer.android.com/guide/topics/ui/layout/linear.html), this works the same way. So the view with the highest weight value gets the most amount of space; views that have the same weight get the same amount of space.
4. **Packed:** The views are packed together (after margins are accounted for). You can then adjust the whole chain's bias (left/right or up/down) by changing the chain's head view bias.

The chain's "head" view (the left-most view in a horizontal chain and the top-most view in a vertical chain) defines the chain's style in XML. However, you can toggle between **spread**, **spread inside**, and **packed** by selecting any view in the chain and then clicking the chain button https://developer.android.com/studio/images/buttons/layout-editor-action-chain.pngthat appears below the view.

To create a chain of views quickly, select them all, right-click one of the views, and then select either **Center Horizontally** or **Center Vertically**, to create either a horizontal or vertical chain, respectively. For more information, see the following video:

**Video 4.** Creating a chain from the action menu

A few other things to consider when using chains:

* A view can be a part of both a horizontal and a vertical chain, making it easy to build flexible grid layouts.
* A chain works properly only if each end of the chain is constrained to another object on the same axis, as shown in figure 14.
* Although the orientation of a chain is either vertical or horizontal, using one does not align the views in that direction. So be sure you include other constraints to achieve the proper position for each view in the chain, such as [alignment constraints](https://developer.android.com/training/constraint-layout#alignment).

**Automatically create constraints**

Instead of adding constraints to every view as you place them in the layout, you can move each view into the positions you desire, and then click **Infer Constraints** https://developer.android.com/studio/images/buttons/layout-editor-infer.pngto automatically create constraints.

Infer Constraints scans the layout to determine the most effective set of constraints for all views. It makes a best effort to constrain the views to their current positions while allowing flexibility. You might need to make some adjustments to be sure the layout responds as you intend for different screen sizes and orientations.

Autoconnect is a separate feature that is either on or off. When turned on, it automatically creates two or more constraints for each view as you add them to the layout, but only when appropriate to constrain the view to the parent layout. Autoconnect does not create constraints to other views in the layout.

Autoconnect is disabled by default. You can enable it by clicking **Turn on Autoconnect** https://developer.android.com/studio/images/buttons/layout-editor-autoconnect-on.pngin the Layout Editor toolbar.

**Keyframe animations**

Within a ConstraintLayout, you can animate changes to the size and position of elements by using [ConstraintSet](https://developer.android.com/reference/android/support/constraint/ConstraintSet) and [TransitionManager](https://developer.android.com/reference/android/transition/TransitionManager).

**Note:** TransitionManager is available in the Support Library for Android 4.0 (API level 14) or higher.

A ConstraintSet is a lightweight object that represents the constraints, margins, and padding of all child elements within a ConstraintLayout. When you apply a ConstraintSet to a displayed ConstraintLayout, the layout updates the constraints of all of its children.

To build an animation using ConstraintSets, specify two layout files which act as a start and end keyframe for the animation. You can then load a ConstraintSet from the second keyframe file and apply it to the displayed ConstraintLayout.

**Important:** ConstraintSet animations animate only the size and position of child elements. They do not animate other attributes (such as color).

The code example below shows how to animate moving a single button to the bottom of the screen.

// MainActivity.kt  
  
fun onCreate(savedInstanceState: Bundle?) {   
    super.onCreate(savedInstanceState)  
    setContentView(R.layout.keyframe\_one)  
    constraintLayout = findViewById(R.id.constraint\_layout) // member variable  
}  
  
fun animateToKeyframeTwo() {  
    val constraintSet = ConstraintSet()  
    constraintSet.load(this, R.layout.keyframe\_two)  
    TransitionManager.beginDelayedTransition();]  
    constraintLayout.apply(constraintSet)  
}

// layout/keyframe1.xml  
// Keyframe 1 contains the starting position for all elements in the animation as well as final colors and text sizes  
  
<?xml version="1.0" encoding="utf-8"?>  
<android.support.constraint.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    xmlns:app="http://schemas.android.com/apk/res-auto"  
    android:layout\_width="match\_parent"  
    android:layout\_height="match\_parent">  
  
    <Button  
        android:id="@+id/button2"  
        android:layout\_width="0dp"  
        android:layout\_height="wrap\_content"  
        android:text="Button"  
        app:layout\_constraintEnd\_toEndOf="parent"  
        app:layout\_constraintStart\_toStartOf="parent"  
        **app:layout\_constraintTop\_toTopOf="parent"** />  
</android.support.constraint.ConstraintLayout>

// layout/keyframe2.xml  
// Keyframe 2 contains another ConstraintLayout with the final positions  
  
<?xml version="1.0" encoding="utf-8"?>  
<android.support.constraint.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    xmlns:app="http://schemas.android.com/apk/res-auto"  
    android:layout\_width="match\_parent"  
    android:layout\_height="match\_parent">  
  
    <Button  
        android:id="@+id/button2"  
        android:layout\_width="0dp"  
        android:layout\_height="wrap\_content"  
        android:text="Button"  
        app:layout\_constraintEnd\_toEndOf="parent"  
        app:layout\_constraintStart\_toStartOf="parent"  
        **app:layout\_constraintBottom\_toBottomOf="parent"** />  
</android.support.constraint.ConstraintLayout>