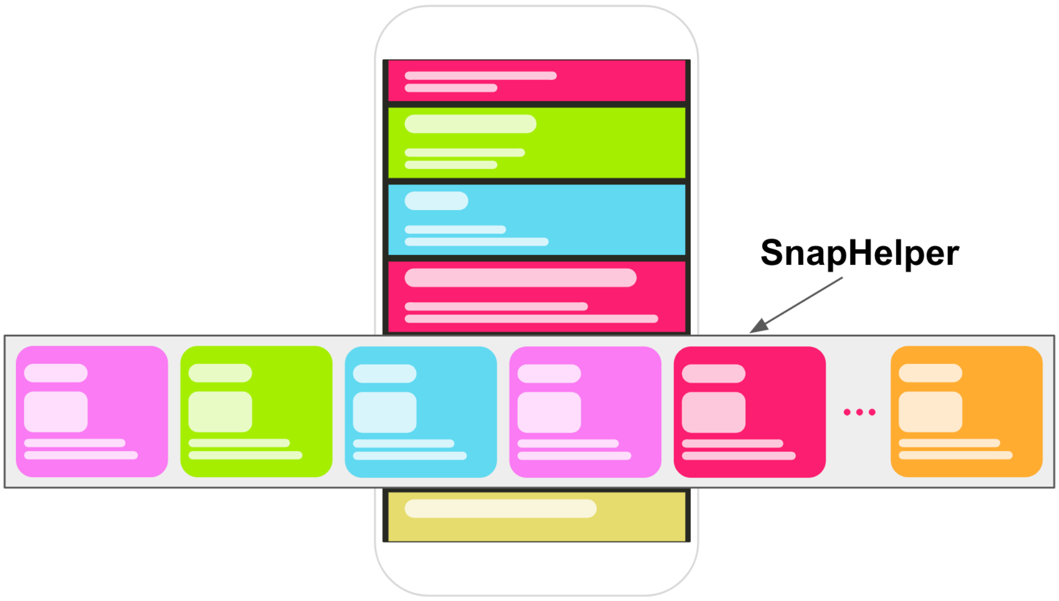
# Inside RecyclerView’s SnapHelper

Jag Saund
Sep 15, 2017 · 6 min read

Trying to make your RecyclerView snap to a particular item? Try using SnapHelper. See how you can customize it to make it work for you!

. . .



RecyclerView is the evolution to ListView. It brings a powerhouse of features. We can make it behave like a carousel or have it snap to a specific position. But manually doing this is difficult. That’s where Android’s support library has us covered. `SnapHelper` makes it a breeze to do this, but what about when we need to customize some of the details? This article takes a deeper look at the internals of the `SnapHelper` so you can tweak and make it work for you.

## What is SnapHelper

So you need to augment your `RecyclerView` to snap to the first visible item in the list. Or maybe you need it to snap to the item closest to the middle. Or maybe some other variation. One option might be to attach a `RecyclerView.OnFlingListener` and intercept the events. That's a lot of work and easy to overlook some things.
This is where `SnapHelper` comes in. The `SnapHelper` component is an abstract class. Android offers two variants for you: `LinearSnapHelper` and `PagerSnapHelper`. They offer pretty much all you need, but if it's not enough, you should consider extending one of them.
`LinearSnapHelper` snaps to the item that is closest to the middle of the `RecyclerView`.
`PagerSnapHelper` offers similar behavior to a `ViewPager` but requires that your item views have their layout parameters set to `MATCH_PARENT`.

## Using the SnapHelper in your project

Integrating one of the concrete `SnapHelper` implementations is very easy.

```kotlin
class MyActivity : AppCompatActivity {

  override fun onCreate(savedInstanceState: Bundle?) {
    val recyclerView = findViewById(R.id.list) as RecyclerView
    recyclerView.layoutManager = LinearLayoutManager(this)
    recyclerView.adapter = ExampleAdapter()
    LinearSnapHelper().attachToRecyclerView(recyclerView) // Configures the snap helper and attaches
  }
}
```

snaphelper.kt hosted with ♡ by GitHub view raw

How to use LinearSnapHelper with your RecyclerView

Let's take a deeper look at `SnapHelper`'s components.

## Components

- `SnapHelper` implements the `RecyclerView.OnFlingListener`.
- `LinearSnapHelper` or `PagerSnapHelper` are concrete implementations of `SnapHelper`. They offer most of the functionality needed to implement your custom `SnapHelper`. Consider extending one of these to suit your needs. For example, `LinearSnapHelper` snaps to the view closest to the middle of the parent.

- `Scroller` provides a way to calculate the distance `RecyclerView` needs to scroll to the target view.
- `RecyclerView.SmoothScroller` performs the act of smooth scrolling to a target view.
- `LinearSmoothScroller` extends `SmoothScroller` and will smooth scroll to the target view using a linear interpolator. The interpolator is switched to a decelerate interpolator once it approaches the target view.

## How it works

1. We construct our `SnapHelper` and attach it to a `RecyclerView`:
   `LinearSnapHelper().attachToRecyclerView(recyclerView)`
   The `SnapHelper` creates a new `OnScrollListener` and `OnFlingListener` and registers them with the `RecyclerView`. It also constructs a new `Scroller`. The `Scroller` calculates the distance `RecyclerView` needs to reach the target view.

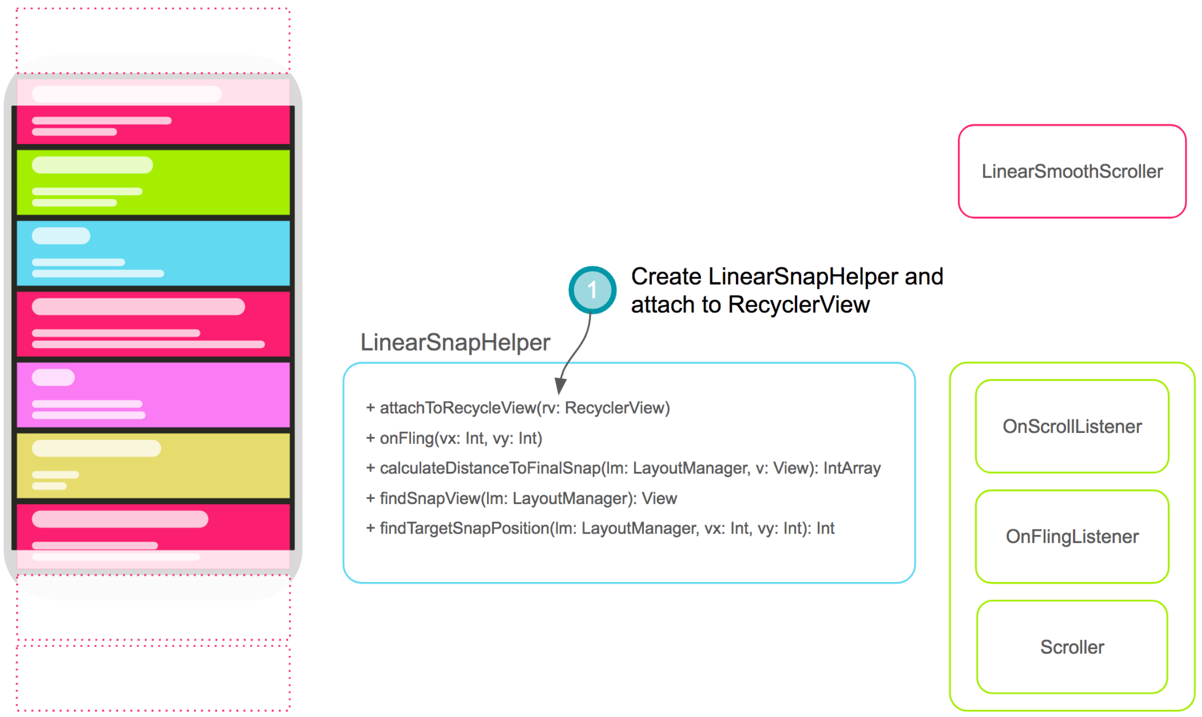
2. `RecyclerView.fling(int velocityX, int velocityY)` captures the fling velocity and forwards it to `SnapHelper`. Recall that `SnapHelper` implements `RecyclerView.OnFlingListener` and registered itself with the `RecyclerView`.

3. `SnapHelper`'s `onFling(int velocityX, int velocityY)` method is called. We check if the velocity in either x or y exceeds the minimum fling velocity. If it does then `snapFromFling` is invoked.

4. `snapFromFling` will do all the work to smooth scroll and snap to the target position. This requires calling a few methods to do the heavy lifting. First, a `LinearSmoothScroller` instance is created (by a call to `createSnapScroller`). Then the `SnapHelper` finds the target position in the adapter to snap to for the given `x` and `y` velocities (by a call to `findTargetSnapPosition`). This position is passed to the `LinearSmoothScroller`: `smoothScroller.setTargetPosition(targetPosition)`. Finally, the `SnapHelper` tells `LayoutManager` to smooth scroll to the target view:

```
layoutManager.startSmoothScroll(smoothScroller);
```

When everything is complete, the target view will be snapped in to position on screen.


LinearSmoothScroller
1
Create LinearSnapHelper and attach to RecyclerView
LinearSnapHelper
+ attachToRecycleView(rv: RecyclerView)
+ onFling(vx: Int, vy: Int)
+ calculateDistanceToFinalSnap(lm: LayoutManager, v: View): IntArray
+ findSnapView(lm: LayoutManager): View
+ findTargetSnapPosition(lm: LayoutManager, vx: Int, vy: Int): Int
OnScrollListener
OnFlingListener
Scroller

How SnapHelper works

## Tip

Need to limit the maximum fling distance? Here's how:

```kotlin
private var scroller: Scroller? = null
private val minX = -2000
private val maxX = 2000
private val minY = -2000
private val maxY = 2000

override fun attachToRecyclerView(recyclerView: RecyclerView?) {
    scroller = Scroller(recyclerView?.context, DecelerateInterpolator())
    super.attachToRecyclerView(recyclerView)
}
override fun calculateScrollDistance(velocityX: Int, velocityY: Int): IntArray {
    val out = IntArray(2)
    scroller?.fling(0, 0, velocityX, velocityY, minX, maxX, minY, maxY)
    out[0] = scroller?.finalX ?: 0
    out[1] = scroller?.finalY ?: 0
    return out
}
```

snaphelpertip1.kt hosted with ♡ by GitHub view raw

SnapHelper tip: Limit the maximum fling distance

## What the SmoothScroller does

The `RecyclerView` provides an abstract class: `RecyclerView.SmoothScroller`. The smooth scroller provides primitive capabilities to track the target view position and trigger scrolling programatically. Android provides a concrete implementation of this: `LinearSmoothScroller`. This class uses a `LinearInterpolator`. It switches to a `DecelerateInterpolator` once the target view becomes a child of the `RecyclerView`. This gives the impression that the `RecyclerView` slowly approaches the target view. There's 3 methods of particular importance:

1. `calculateSpeedPerPixel`: Easiest and simplest one where we convert display metrics to pixels per second

   ```
   return MILLISECONDS_PER_INCH / displayMetrics.densityDpi;
   ```

2. `calculateTimeForScrolling`: The default doesn't place an upper bound on time but does ensure that if the amount scrolled is very small that it always returns a positive time (avoiding rounding errors).

   ```
   (int) Math.ceil(Math.abs(dx) * MILLISECONDS_PER_PX);
   ```

3. `onTargetFound`: Called when the target view becomes a child of the `RecyclerView`. This is the last callback received by the `SmoothScroller`. We need to calculate the distance and time to the snap position. `calculateDistanceToFinalSnap` will provide the scroll distance (`SnapHelper` defines this as an abstract method). `calculateTimeForDeceleration` will provide the time given the distance computed above. `Action` object is updated with the time and distance details. `LinearSmoothScroller` uses this information to complete the scroll.

## Tip

Need to limit the time a smooth scroll can take? Here's how:

```
private const val MILLISECONDS_PER_INCH = 100
private const val MAX_SCROLL_ON_FLING_DURATION = 250 // ms

override fun createSnapScroller(layoutManager: LayoutManager?): LinearSmoothScroller? {
    if (layoutManager !is ScrollVectorProvider) return null
    return object : LinearSmoothScroller(context) {
        override fun onTargetFound(targetView: View?, state: State?, action: Action?) {
            if (targetView == null) return
            val snapDistances = calculateDistanceToFinalSnap(layoutManager, targetView)
            val dx = snapDistances[0]
            val dy = snapDistances[1]
            val time = calculateTimeForDeceleration(Math.max(Math.abs(dx), Math.abs(dy)))
            if (time > 0) {
```

```
                action?.update(dx, dy, time, mDecelerateInterpolator)
            }
        }

        override fun calculateSpeedPerPixel(displayMetrics: DisplayMetrics?): Float =
                MILLISECONDS_PER_INCH / displayMetrics.densityDpi

        override fun calculateTimeForScrolling(dx: Int): Int =
                Math.min(MAX_SCROLL_ON_FLING_DURATION, super.calculateTimeForScrolling(dx))
    }
}
```

snaphelpertip2.kt hosted with ♡ by GitHub view raw

SnapHelper tip: Limit the time a smooth scroll takes

## How to find the target adapter position

The `SnapHelper` needs a way of translating the fling velocity to an item position in the adapter. In order to do this, you need to implement `findTargetSnapPosition(LayoutManager layoutManager, int velocityX, int velocityY)`.

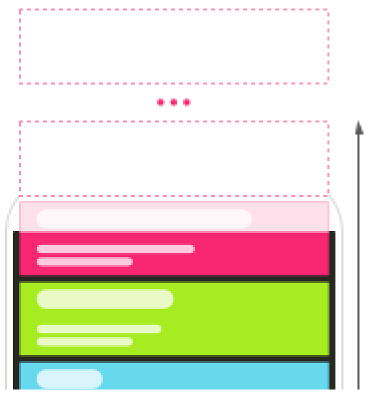
*Note: This doesn't have to be accurate — treat it as more of an estimate.*

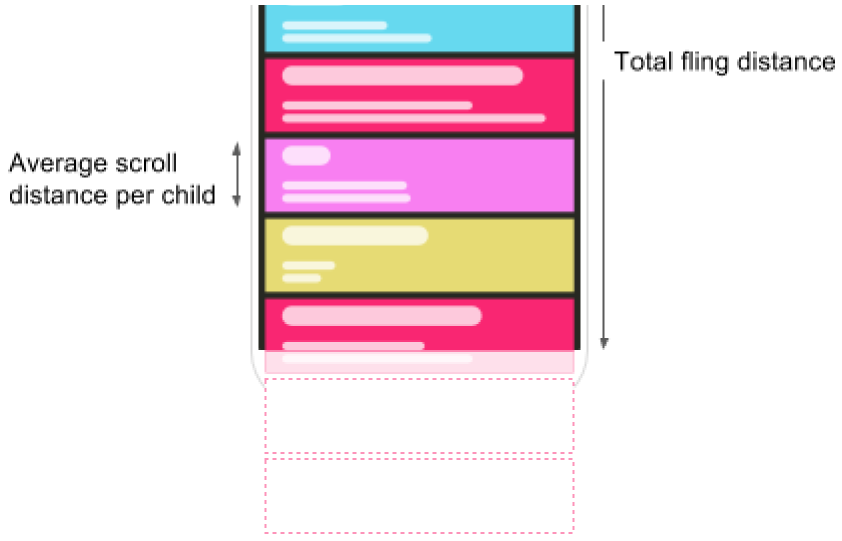
To get a better understanding of how to do this, let's take a deeper look at how `LinearSnapHelper` determines the target position. Recall that `LinearSnapHelper`'s goal is to snap to the item that is closest to the center.

First, it finds the current center view and its position in the adapter. Next, with the help of the `Scroller`, the `LinearSnapHelper` calculates the distance the `RecyclerView` should scroll – using `calculateScrollDistance(int velocityX, int velocityY)`. We need to map the total scroll distance to a number of adapter items. `LinearSnapHelper` accomplishes this by calculating the average `RecyclerView` child height currently visible. This represents the average amount to scroll per child. Now we can estimate the total number of children to scroll:

`val delta = totalScrollDistance / avgScrollDistancePerChild`.

Finally, the new target snap position is estimated as `currentCenterPos + delta`.





Estimating number of children to scroll

One interesting thing to note here is that the direction to scroll in comes from the velocity. However, this may not match the order of children in the `LayoutManager`. To overcome this, a `ScrollVectorProvider` is used to get the direction (*Note: All `LayoutManager`s provided by Android implement the `ScrollVectorProvider`*):

```
val vectorProvider = layoutManager as
RecyclerView.SmoothScroller.ScrollVectorProvider
val vectorForEnd =
vectorProvider.computeScrollVectorForPosition(itemCount - 1)
```

If we assume that we are scrolling vertically then the value in `vectorForEnd` allows us to define the direction as:

```
if (vectorForEnd.y < 0) {
  delta = -delta;
}
val targetPos = Math.max(0, Math.min(itemCount - 1, currentPos +
delta))
```

## How to find the target snap view

Once the `RecyclerView` begins to settle, we need to find a reference view to snap to from the current set of child views. This is the purpose of `findSnapView(LayoutManager layoutManager)`. Let's take a deeper look at how `LinearSnapHelper` determines the target snap view. `LinearSnapHelper` walks over the children currently attached to the

`RecyclerView` . It keeps track of the view that is closest to the center. At the end, it returns the view which it found to be closest to the center.

. . .

Creating a widget that snaps to a particular position was once a very difficult. Accomplishing this with a `ListView` was extremely difficult. `RecyclerView` made it easier but there was still a tremendous amount of heavy lifting. With `SnapHelper` , and it's two concrete implementations — `LinearSnapHelper` and `PagerHelper` , this becomes painless and simple. Extending their functionality is also simple. Typically this requires modifying `findTargetSnapPosition` , `findSnapView` , or `calculateDistanceToFinalSnap` . We looked at the details of `SnapHelper` 's components and how they interact with each other. This knowledge should aid in extending `SnapHelper` to tailor it to your needs. Share your tips and how you use `SnapHelper` !

Android App Development Recyclerview AndroidDev Android Development
Android Developers