# 19
# Animations and Interpolations

Here, we will see how we can use the `Animation` class to make our UI a little less static and a bit more interesting. As we have come to expect, the Android API will allow us to do some quite advanced things with relatively straightforward code, and the `Animation` class is no different.

This chapter can be approximately divided into these parts:

- An introduction to how animations in Android work and are implemented
- An introduction to a UI widget that we haven't explored yet, the `SeekBar` class
- A working animation app

First, let's explore how animations work in Android.

## Animations in Android

The normal way to create an animation in Android is through XML. We can write XML animations, and then load and play them through our Kotlin code on a specified UI widget. So, for example, we can write an animation that fades in and out five times over three seconds, then play that animation on an `ImageView` or any other widget. We can think of these XML animations as a script, as they define the type, order, and timing.

Let's explore some of the different properties we can assign to our animations, how to use them in our Kotlin code, and finally, we can make a neat animations app to try it all out.

## Designing cool animations in XML

We have learned that XML can be used to describe animations as well as UI layouts, but let's find out exactly how. We can state values for properties of an animation that describe the starting and ending appearance of a widget. The XML can then be loaded by our Kotlin code by referencing the name of the XML file that contains the animation and turning it into a usable Kotlin object, again, not unlike a UI layout.

Many animation properties come in pairs. Here is a quick look at some of the animation property pairs we can use to create an animation. Straight after we have looked at some XML, we will see how to use it.

### Fading in and out

Alpha is the measure of transparency. So, by stating the starting `fromAlpha` and ending `toAlpha` values, we can fade items in and out. A value of `0.0` is invisible, and `1.0` is an object's normal appearance. Steadily moving between the two makes a fading-in effect:

```
<alpha
   android:fromAlpha = "0.0"
   android:toAlpha = "1.0" />
```

### Move it, move it

We can move an object within our UI by using a similar technique; `fromXDelta` and `toXDelta` can have their values set as a percentage of the size of the object being animated.

The following code would move an object from left to right a distance equal to the width of the object itself:

```
<translate
android:fromXDelta = "-100%"
android:toXDelta = "0%"/>
```

In addition, there are the `fromYDelta` and `toYDelta` properties for animating upward and downward movement.

## Scaling or stretching

The `fromXScale` and `toXScale` properties will increase or decrease the scale of an object. As an example, the following code will change the object running the animation from normal size to invisible:

```
<scale
android:fromXScale = "1.0"
android:fromYScale = "0.0"/>
```

As another example, we could shrink the object to a tenth of its usual size using `android:fromYScale = "0.1"`, or make it 10 times as big using `android:fromYScale = "10.0"`.

## Controlling the duration

Of course, none of these animations would be especially interesting if they just instantly arrived at their conclusion. To make our animations more interesting, we can therefore set their duration in milliseconds. A millisecond is one thousandth of a second. We can also make timing easier, especially in relation to other animations, by setting the `startOffset` property, which is also in milliseconds.

The next code would begin an animation one third of a second after we started it (in code), and it would take two thirds of a second to complete:

```
android:duration = "666"
android:startOffset = "333"
```

## Rotate animations

If you want to spin something around, just use the `fromDegrees` and `toDegrees` properties. This next code, probably predictably, will spin a widget around in a complete circle because, of course, there are 360 degrees in a circle:

```
<rotate android:fromDegrees = "360"
        android:toDegrees = "0"
/>
```

## Repeating animations

Repetition might be important in some animations, perhaps a wobble or shake effect, so we can add a `repeatCount` property. In addition, we can specify how the animation is repeated by setting the `repeatMode` property.

The following code would repeat an animation 10 times, each time reversing the direction of the animation. The `repeatMode` property is relative to the current state of the animation. What this means is that if you rotated a button from 0 to 360 degrees, for example, the second part of the animation (the first repeat) would rotate the other way, from 360 back to 0. The third part of the animation (the second repeat) would, again, reverse and rotate from 0 to 360:

```
android:repeatMode = "reverse"
android:repeatCount = "10"
```

## Combining an animation's properties with sets

To combine groups of these effects, we need a `set` of properties. This code shows how we can combine all the previous code snippets we have just seen into an actual XML animation that will compile:

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
     ...All our animations go here
</set>
```

We still haven't seen any Kotlin with which to bring these animations to life. Let's fix that now.

## Instantiating animations and controlling them with Kotlin code

This next snippet of code shows how we would declare an object of the `Animation` type, initialize it with an animation contained in an XML file named `fade_in.xml`, and start the animation on an `ImageView` widget. We will soon do this in a project and also see where we can put the XML animations:

```
// Declare an Animation object
var animFadeOut: Animation? = null

// Initialize it
animFadeIn = AnimationUtils.loadAnimation(
                this, R.anim.fade_in)

// Start the animation on the ImageView
// with an id property set to imageView
imageView.startAnimation(animFadeIn)
```

We already have quite a powerful arsenal of animations and control features for things such as timing. But the Android API gives us a little bit more than this as well.

## More animation features

We can listen for the status of animations much like we can listen for clicks on a button. We can also use **interpolators** to make our animations more life-like and pleasing. Let's look at listeners first.

### Listeners

If we implement the `AnimationListener` interface, we can indeed listen to the status of animations by overriding the three functions that tell us when something has occurred. We could then act based on these events.

`OnAnimationEnd` announces the end of an animation, `onAnimationRepeat` is called each time an animation begins a repeat, and – perhaps predictably – `onAnimationStart` is called when an animation has started animating. This might not be the same time as when `startAnimation` is called if a `startOffset` is set in the animations XML:

```
override fun onAnimationEnd(animation: Animation) {
   // Take some action here

}

override fun onAnimationStart(animation: Animation) {

   // Take some action here

}

override fun onAnimationRepeat(animation: Animation){

   // Take some action here

}
```

We will see how `AnimationListener` works in the Animations demo app, and we'll also put another widget, `SeekBar`, into action.

### Animation interpolators

If you can think back to high school, you might remember exciting lessons about calculating acceleration. If we animated something at a constant speed, at first glance, things might seem OK. If we then compared the animation to another that uses gradual acceleration, then the latter would almost certainly be more pleasing to watch.

It is possible that if we were not told the only difference between the two animations was that one used acceleration and the other didn't, we wouldn't be able to say *why* we preferred it. Our brains are more receptive to things that conform to the norms of the world around us. Therefore, adding a bit of real-world physics, such as acceleration and deceleration, improves our animations.

The last thing we want to do, however, is start doing a bunch of mathematical calculations just to slide a button onto the screen or spin some text in a circle.

This is where **interpolators** come in. They are animation modifiers that we can set in a single line of code within our XML.

Some examples of interpolators are `accelerate_interpolator` and `cycle_interpolator`:

```
android:interpolator="@android:anim/accelerate_interpolator"
android:interpolator="@android:anim/cycle_interpolator"/>
```

We will put some interpolators, along with some XML animations and the related Kotlin code, into action next.

You can learn more about interpolators and the Android `Animation` class on the Android developer website here: `http://developer.android.com/guide/topics/resources/animation-resource.html`.

## Animations demo app – introducing SeekBar

That's enough theory, especially with something that should be so visible. Let's build an animation demo app that explores everything we have just discussed, and a bit more as well.

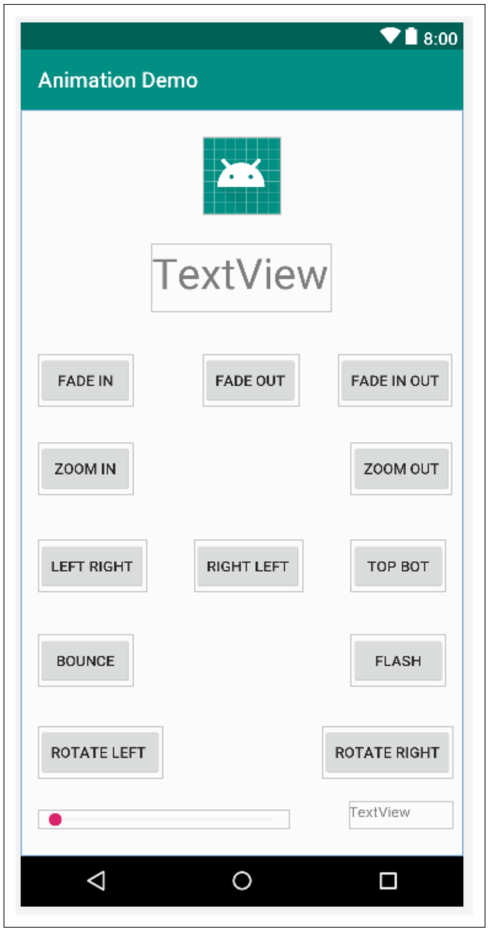
This app involves small amounts of code in lots of different files. Therefore, I have tried to make it plain what code is in what file, so you can keep track of what is going on. This will make the Kotlin we write for this app more understandable as well.

The app will demonstrate rotations, fades, translations, animation events, interpolations, and controlling duration with a `SeekBar` widget. The best way to explain what `SeekBar` does is to build it and then watch it in action.

## Laying out the animation demo

Create a new project called `Animation Demo` using the **Empty Activity** template, leaving all the other settings at their usual settings. As usual, should you wish to speed things up by copying and pasting the layout, the code, or the animation XML, it can all be found in the `Chapter19` folder.

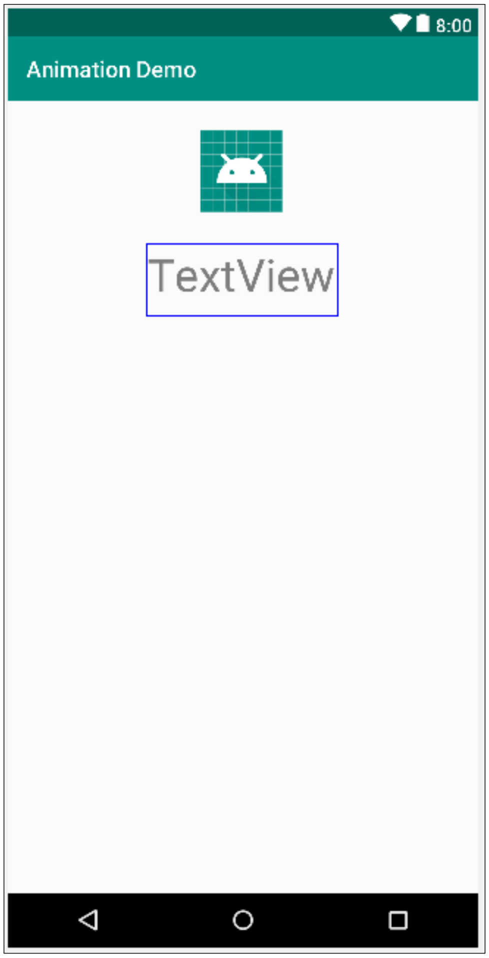
Use the following reference screenshot of the finished layout to help guide you through the next steps:



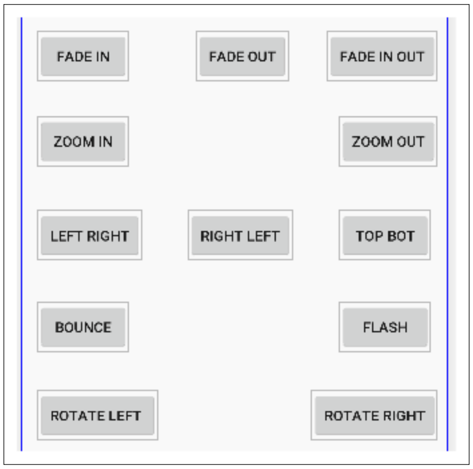
Here is how you can lay out the UI for this app:

1. Open `activity_main.xml` in the design view of the editor window.
2. Delete the default **Hello world!** `TextView`.

3. Add an **ImageView** to the top-center portion of the layout. Use the previous reference screenshot to guide you. Use the `@mipmap/ic_launcher` to show the Android robot in `ImageView` when prompted to do so by selecting **Project | ic_launcher** in the pop-up **Resources** window.
4. Set the `id` property of the `ImageView` to `imageView`.
5. Directly below the `ImageView`, add a `TextView`. Set the `id` to `textStatus`. I made my `TextView` a little bigger by dragging its edges (not the constraint handles) and changed its `textSize` attribute to `40sp`. The layout so far should look something like this next screenshot:



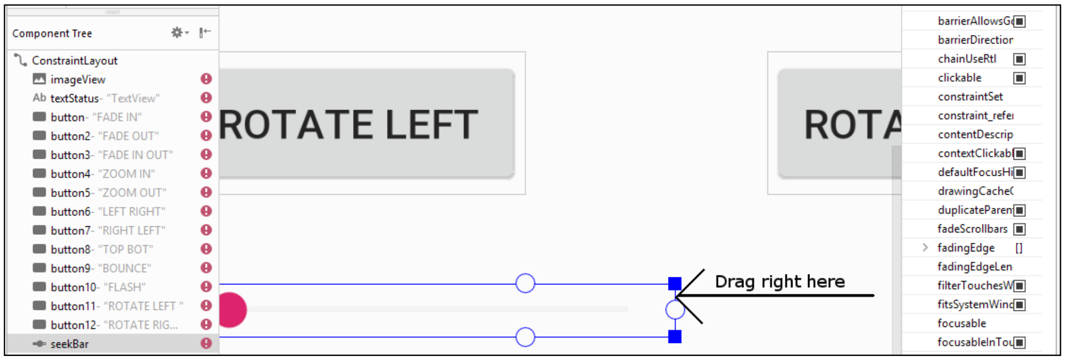
6. Now we will add a large selection of **Button** widgets to the layout. Exact positioning is not vital, but the exact `id` property values we add to them later in the tutorial will be. Follow this next screenshot to lay out 12 buttons in the layout. Alter the `text` attribute so that your buttons have the same text as those in the next screenshot. The `text` attributes are detailed specifically in the next step in case the screenshot isn't clear enough:



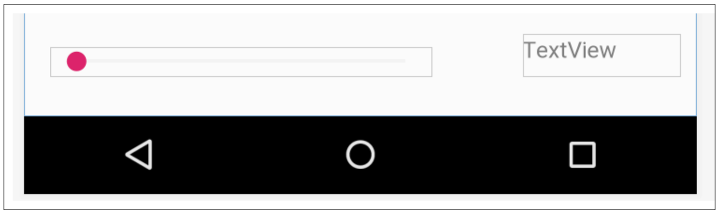
[ To make the process of laying out the buttons quicker, lay them out just approximately at first, then add the `text` attributes from the next step, and then fine-tune the button positions to get a neat layout. ]

7. Add the `text` values as they are in the screenshot; here are all the values from left to right and top to bottom: `FADE IN`, `FADE OUT`, `FADE IN OUT`, `ZOOM IN`, `ZOOM OUT`, `LEFT RIGHT`, `RIGHT LEFT`, `TOP BOT`, `BOUNCE`, `FLASH`, `ROTATE LEFT`, and `ROTATE RIGHT`.
8. Add a `SeekBar` widget from the **Widgets** category of the palette, on the left, below the buttons. Set the `id` property to `seekBarSpeed` and the `max` property to `5000`. This means that `SeekBar` widget will hold a value between 0 and 5,000 as it is dragged by the user from left to right. We will see how we can read and use this data soon.

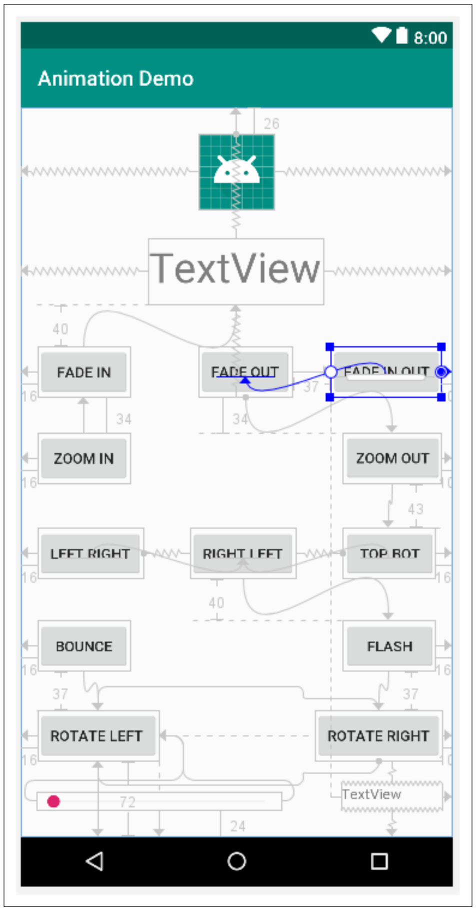
9. We want to make the `SeekBar` widget much wider. To achieve this, you use the exact same technique as with any widget; just drag the edges of the widget. However, as the `SeekBar` widget is quite small, it is hard to increase its size without accidentally selecting the constraint handles. To overcome this problem, zoom into the design view by holding the *Ctrl* key and rolling the middle mouse wheel forward. You can then grab the edges of the `SeekBar` widget without touching the constraint handles. I have shown this in action in the next screenshot:



10. Now, add a `TextView` widget just to the right of the `SeekBar` widget and set its `id` property to `textSeekerSpeed`. This step, combined with the previous two, should look like this screenshot:



11. Tweak the positions to look like the reference screenshot at the start of these steps, and then click the **Infer Constraints** button to lock the positions. Of course, you can do this manually if you want the practice. Here is a screenshot with all the constraints in place:



12. Next, add the following `id` properties to the buttons, as identified by the text property that you have already set. If you are asked whether you want to **Update usages...** as you enter these values, select **Yes**:

| **Existing text property** | **Value of id property to set** |
|---|---|
| Fade In | `btnFadeIn` |
| Fade Out | `btnFadeOut` |
| Fade In Out | `btnFadeInOut` |
| Zoom In | `btnZoomIn` |
| Zoom Out | `btnZoomOut` |
| Left Right | `btnLeftRight` |
| Right Left | `btnRightLeft` |
| Top Bot | `btnTopBottom` |

| Existing text property | Value of id property to set |
|---|---|
| Bounce | btnBounce |
| Flash | btnFlash |
| Rotate Left | btnRotateLeft |
| Rotate Right | btnRotateRight |

We will see how to use this newcomer to our UI (`SeekBar`) when we get to coding the `MainActivity` class in a few sections time.

## Coding the XML animations

Right-click on the **res** folder and select **New | Android resource directory**. Enter `anim` in the `Directory name:` field and left-click **OK**.

Now right-click on the new **anim** directory and select **New | Animation resource file**. In the **File name:** field, type `fade_in` and then left-click **OK**. Delete the contents and add this code to create the animation:

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
   android:fillAfter="true" >

   <alpha
   android:fromAlpha = "0.0"
   android:interpolator =
             "@android:anim/accelerate_interpolator"

   android:toAlpha="1.0" />
</set>
```

Right-click on the **anim** directory and select **New | Animation resource file**. In the `File name:` field, type `fade_out` and then left-click **OK**. Delete the contents and add this code to create the animation:

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
   android:fillAfter = "true" >

   <alpha
         android:fromAlpha = "1.0"
         android:interpolator =
              "@android:anim/accelerate_interpolator"

   android:toAlpha = "0.0" />
</set>
```

Right-click on the **anim** directory and select **New | Animation resource file**. In the `File name:` field, type `fade_in_out` and then left-click **OK**. Delete the contents and add this code to create the animation:

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
    android:fillAfter = "true" >

    <alpha
          android:fromAlpha="0.0"
          android:interpolator =
          "@android:anim/accelerate_interpolator"

          android:toAlpha = "1.0" />

    <alpha
       android:fromAlpha = "1.0"
          android:interpolator =
         "@android:anim/accelerate_interpolator"

         android:toAlpha = "0.0" />
</set>
```

Right-click on the **anim** directory and select **New | Animation resource file**. In the `File name:` field, type `zoom_in` and then left-click **OK**. Delete the contents and add this code to create the animation:

```
<set xmlns:android="http://schemas.android.com/apk/res/android"
    android:fillAfter = "true" >

    <scale
        android:fromXScale = "1"
        android:fromYScale = "1"
        android:pivotX = "50%"
        android:pivotY = "50%"
        android:toXScale = "6"
        android:toYScale = "6" >
    </scale>
</set>
```

Right-click on the **anim** directory and select **New | Animation resource file**. In the `File name:` field, type `zoom_out` and then left-click **OK**. Delete the contents and add this code to create the animation:

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android">
    <scale
        android:fromXScale = "6"
        android:fromYScale = "6"
        android:pivotX = "50%"
        android:pivotY = "50%"
        android:toXScale = "1"
        android:toYScale = "1" >
    </scale>
</set>
```

Right-click on the **anim** directory and select **New | Animation resource file**. In the `File name:` field, type `left_right` and then left-click **OK**. Delete the contents and add this code to create the animation:

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android">
    <translate

        android:fromXDelta = "-500%"
        android:toXDelta = "0%"/>
</set>
```

Again, right-click on the **anim** directory and select **New | Animation resource file**. In the **File name:** field, type `right_left` and then left-click **OK**. Delete the entire contents and add this code to create the animation:

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android">
    <translate
        android:fillAfter = "false"
        android:fromXDelta = "500%"
        android:toXDelta = "0%"/>
</set>
```

As before, right-click on the **anim** directory and select **New | Animation resource file**. In the **File name:** field, type `top_bot` and then left-click **OK**. Delete the entire contents and add this code to create the animation:

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android">
```

```
    <translate
        android:fillAfter = "false"
        android:fromYDelta = "-100%"
        android:toYDelta = "0%"/>
</set>
```

You guessed it; right-click on the **anim** directory and select **New | Animation resource file**. In the **File name:** field, type `flash` and then left-click **OK**. Delete the contents and add this code to create the animation:

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android">
    <alpha android:fromAlpha = "0.0"
        android:toAlpha = "1.0"
        android:interpolator =
           "@android:anim/accelerate_interpolator"

        android:repeatMode = "reverse"
        android:repeatCount = "10"/>
</set>
```

Just a few more to go – right-click on the **anim** directory and select **New | Animation resource file**. In the **File name:** field, type `bounce` and then left-click **OK**. Delete the contents and add this code to create the animation:

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
    android:fillAfter = "true"
    android:interpolator =
         "@android:anim/bounce_interpolator">

    <scale
        android:fromXScale = "1.0"
        android:fromYScale = "0.0"
        android:toXScale = "1.0"
        android:toYScale = "1.0" />

</set>
```

Right-click on the **anim** directory and select **New | Animation resource file**. In the **File name:** field, type `rotate_left` and then left-click **OK**. Delete the contents and add this code to create the animation. Here we see something new, `pivotX="50%"` and `pivotY="50%"`. This makes the rotate animation central on the widget that will be animated. We can think of this as setting the *pivot* point of the animation:

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android">
    <rotate android:fromDegrees = "360"
        android:toDegrees = "0"
        android:pivotX = "50%"
        android:pivotY = "50%"
        android:interpolator =
           "@android:anim/cycle_interpolator"/>
</set>
```

Right-click on the **anim** directory and select **New | Animation resource file**. In the **File name:** field, type `rotate_right` and then left-click **OK**. Delete the contents and add this code to create the animation:

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android">
    <rotate android:fromDegrees = "0"
        android:toDegrees = "360"
        android:pivotX = "50%"
        android:pivotY = "50%"
        android:interpolator =
             "@android:anim/cycle_interpolator"/>

</set>
```

Phew! Now we can write the Kotlin code to add our animations to our UI.

## Wiring up the Animation demo app in Kotlin

Open the `MainActivity.kt` file. Now, following the class declaration, we can declare the following properties for animations:

```
var seekSpeedProgress: Int = 0

private lateinit var animFadeIn: Animation
private lateinit var animFadeOut: Animation
private lateinit var animFadeInOut: Animation

private lateinit var animZoomIn: Animation
private lateinit var animZoomOut: Animation

private lateinit var animLeftRight: Animation
private lateinit var animRightLeft: Animation
private lateinit var animTopBottom: Animation

private lateinit var animBounce: Animation
private lateinit var animFlash: Animation


private lateinit var animRotateLeft: Animation
private lateinit var animRotateRight: Animation
```

You will need to add the following `import` statement at this point:

```
import android.view.animation.Animation;
```

In the preceding code, we used the `lateinit` keyword when declaring the `Animation` instances. This will mean that Kotlin will check that each instance is initialized before it is used. This avoids us using `!!` (null checks) each time we use a function on one of these instances. For a refresher on the `!!` operator, refer to *Chapter 12, Connecting Our Kotlin to the UI and Nullability*.

We also added an `Int` property, `seekSpeedProgress`, which will be used to track the current value/position of `SeekBar`.

Now, let's call a new function from `onCreate` after the call to `setContentView`:

```
override fun onCreate(savedInstanceState: Bundle?) {
   super.onCreate(savedInstanceState)
   setContentView(R.layout.activity_main)

   loadAnimations()
}
```

At this point, the new line of code will have an error until we implement the new function.

Now we will implement the `loadAnimations` function. Although the code in this function is quite extensive, it is also very straightforward. All we are doing is using the `loadAnimation` function of the `AnimationUtils` class to initialize each of our `Animation` references with one of our XML animations. You will also notice that, for the `animFadeIn` `Animation`, we also call `setAnimationListener` on it. We will write the functions to listen for events shortly.

Add the `loadAnimations` function:

```
private fun loadAnimations() {

   animFadeIn = AnimationUtils.loadAnimation(
                this, R.anim.fade_in)
   animFadeIn.setAnimationListener(this)
   animFadeOut = AnimationUtils.loadAnimation(
                this, R.anim.fade_out)
   animFadeInOut = AnimationUtils.loadAnimation(
                this, R.anim.fade_in_out)

   animZoomIn = AnimationUtils.loadAnimation(
                this, R.anim.zoom_in)
   animZoomOut = AnimationUtils.loadAnimation(
                this, R.anim.zoom_out)

   animLeftRight = AnimationUtils.loadAnimation(
                 this, R.anim.left_right)
   animRightLeft = AnimationUtils.loadAnimation(
                 this, R.anim.right_left)
   animTopBottom = AnimationUtils.loadAnimation(
                 this, R.anim.top_bot)

   animBounce = AnimationUtils.loadAnimation(
                 this, R.anim.bounce)
   animFlash = AnimationUtils.loadAnimation(
                 this, R.anim.flash)

   animRotateLeft = AnimationUtils.loadAnimation(
                 this, R.anim.rotate_left)
   animRotateRight = AnimationUtils.loadAnimation(
                 this, R.anim.rotate_right)
}
```

You will need to import one new class at this point:

```
import android.view.animation.AnimationUtils
```

Now, we will add a click-listener for each button. Add this code immediately before the closing curly brace of the `onCreate` function:

```
btnFadeIn.setOnClickListener(this)
btnFadeOut.setOnClickListener(this)
btnFadeInOut.setOnClickListener(this)
btnZoomIn.setOnClickListener(this)
btnZoomOut.setOnClickListener(this)
btnLeftRight.setOnClickListener(this)
btnRightLeft.setOnClickListener(this)
btnTopBottom.setOnClickListener(this)
btnBounce.setOnClickListener(this)
btnFlash.setOnClickListener(this)
btnRotateLeft.setOnClickListener(this)
btnRotateRight.setOnClickListener(this)
```

The code we just added creates errors in all the lines of code. We can ignore them for now, as we will fix them shortly and discuss what happened.

Now, we can use a lambda to handle the `SeekBar` interactions. We will override three functions, as it is required by the interface when implementing `OnSeekBarChangeListener`:

- A function that detects a change in the position of the `SeekBar` widget, called `onProgressChanged`
- A function that detects the user starting to change the position, called `onStartTrackingTouch`
- A function that detects when the user has finished using the `SeekBar` widget, called `onStopTrackingTouch`

To achieve our goals, we only need to add code to the `onProgressChanged` function, but we must still override them all.

All we do in the `onProgressChanged` function is assign the current value of the `SeekBar` object to the `seekSpeedProgress` member variable, so it can be accessed from elsewhere. Then, we use this value along with the maximum possible value of the `SeekBar` object, obtained by using `seekBarSpeed.max`, and output a message to the `textSeekerSpeed` `TextView`.

Add the code we have just discussed before the closing curly brace of the `onCreate` function:

```
seekBarSpeed.setOnSeekBarChangeListener(
         object : SeekBar.OnSeekBarChangeListener {

   override fun onProgressChanged(
                seekBar: SeekBar, value: Int,
                fromUser: Boolean) {

         seekSpeedProgress = value
         textSeekerSpeed.text =
               "$seekSpeedProgress of $seekBarSpeed.max"
  }

  override fun onStartTrackingTouch(seekBar: SeekBar) {}

  override fun onStopTrackingTouch(seekBar: SeekBar) {}
})
```

Now, we need to alter the `MainActivity` class declaration to implement two interfaces. In this app, we will be listening for clicks and for animation events, so the two interfaces we will be using are `View.OnClickListener` and `Animation.AnimationListener`. You will notice that to implement more than one interface, we simply separate the interfaces with a comma.

Alter the `MainActivity` class declaration by adding the highlighted code we have just discussed:

```
class MainActivity : AppCompatActivity(),
        View.OnClickListener,
        Animation.AnimationListener {
```

At this stage, we can add and implement the required functions for those interfaces. First, the `AnimationListener` functions, `onAnimationEnd`, `onAnimationRepeat`, and `onAnimationStart`. We only need to add a little code to two of these functions. In `onAnimationEnd`, we set the `text` property of `textStatus` to `STOPPED`, and in `onAnimationStart`, we set it to `RUNNING`. This will demonstrate our animation listeners are indeed listening and working:

```
override fun onAnimationEnd(animation: Animation) {
   textStatus.text = "STOPPED"
}

override fun onAnimationRepeat(animation: Animation) {
}

override fun onAnimationStart(animation: Animation) {
   textStatus.text = "RUNNING"
}
```

The `onClick` function is quite long, but nothing complicated. Each option of the `when` block handles each button from the UI, sets the duration of an animation based on the current position of the `SeekBar` widget, sets up the animation so it can be listened to for events, and then starts the animation.

You will need to use your preferred technique to import the `View` class:

```
import android.view.View;
```

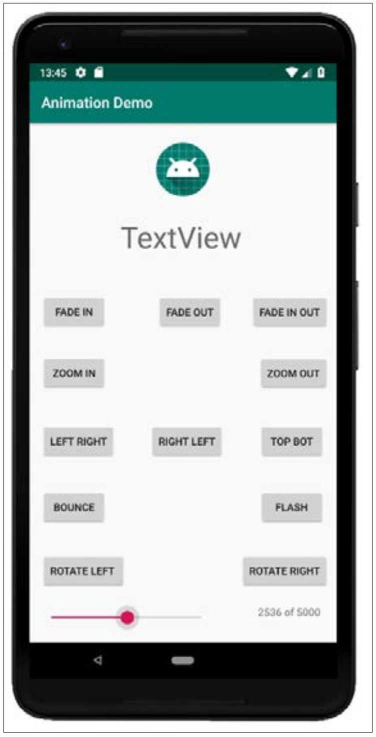
Add the `onClick` function we have just discussed, and we have then completed this mini app:

```
override fun onClick(v: View) {
when (v.id) {
  R.id.btnFadeIn -> {
        animFadeIn.duration = seekSpeedProgress.toLong()
        animFadeIn.setAnimationListener(this)
        imageView.startAnimation(animFadeIn)
  }

  R.id.btnFadeOut -> {
        animFadeOut.duration = seekSpeedProgress.toLong()
        animFadeOut.setAnimationListener(this)
        imageView.startAnimation(animFadeOut)
  }

  R.id.btnFadeInOut -> {

        animFadeInOut.duration = seekSpeedProgress.toLong()
        animFadeInOut.setAnimationListener(this)
        imageView.startAnimation(animFadeInOut)
  }

  R.id.btnZoomIn -> {
        animZoomIn.duration = seekSpeedProgress.toLong()
        animZoomIn.setAnimationListener(this)
```

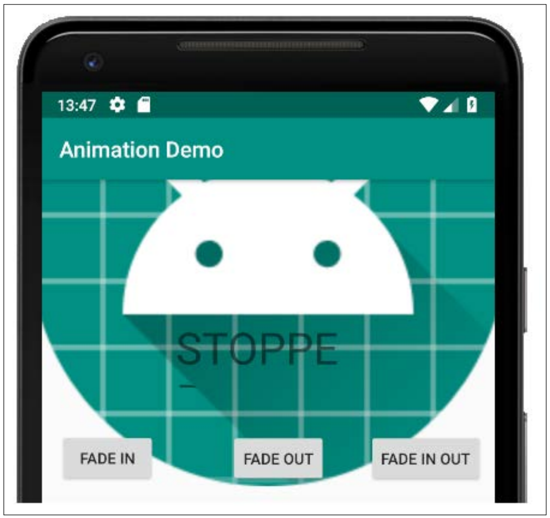
```
            imageView.startAnimation(animZoomIn)
      }

      R.id.btnZoomOut -> {
            animZoomOut.duration = seekSpeedProgress.toLong()
            animZoomOut.setAnimationListener(this)
            imageView.startAnimation(animZoomOut)
      }


      R.id.btnLeftRight -> {
            animLeftRight.duration = seekSpeedProgress.toLong()
            animLeftRight.setAnimationListener(this)
            imageView.startAnimation(animLeftRight)
      }

      R.id.btnRightLeft -> {
            animRightLeft.duration = seekSpeedProgress.toLong()
            animRightLeft.setAnimationListener(this)
            imageView.startAnimation(animRightLeft)
      }

      R.id.btnTopBottom -> {
            animTopBottom.duration = seekSpeedProgress.toLong()
            animTopBottom.setAnimationListener(this)
            imageView.startAnimation(animTopBottom)
      }

      R.id.btnBounce -> {
            /*
            Divide seekSpeedProgress by 10 because with
            the seekbar having a max value of 5000 it
            will make the animations range between
            almost instant and half a second
            5000 / 10 = 500 milliseconds
            */
            animBounce.duration =
                  (seekSpeedProgress / 10).toLong()
            animBounce.setAnimationListener(this)
            imageView.startAnimation(animBounce)
      }

      R.id.btnFlash -> {
            animFlash.duration = (seekSpeedProgress / 10).toLong()
```

```
                animFlash.setAnimationListener(this)
                imageView.startAnimation(animFlash)
        }

        R.id.btnRotateLeft -> {
                animRotateLeft.duration = seekSpeedProgress.toLong()
                animRotateLeft.setAnimationListener(this)
                imageView.startAnimation(animRotateLeft)
        }

        R.id.btnRotateRight -> {
                animRotateRight.duration = seekSpeedProgress.toLong()
                animRotateRight.setAnimationListener(this)
                imageView.startAnimation(animRotateRight)
        }
    }

    }
```

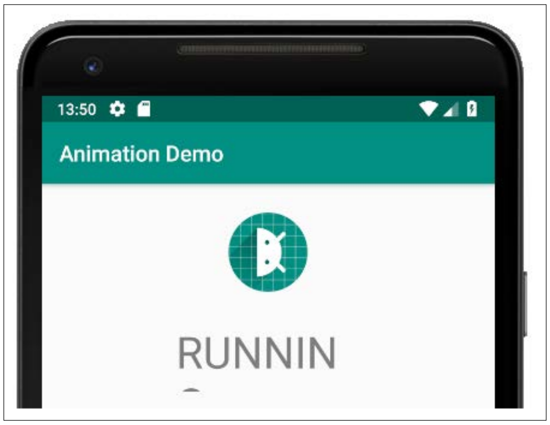
Now, run the app and move the `SeekBar` widget to roughly the center so that the animations run for a reasonable amount of time:



Click the **ZOOM IN** button:



Notice how the text on the Android robot changes from **RUNNING** to **STOPPED** at the appropriate time. Now, click one of the **ROTATE** buttons:



Most of the other animations don't do themselves justice in a screenshot, so be sure to try them all out for yourself.

## Frequently asked questions

Q.1) We know how to animate widgets now, but what about shapes or images that I create myself?

A) An `ImageView` widget can hold any image you like. Just add the image to the `drawable` folder and then set the appropriate `src` attribute on the `ImageView` widget. You can then animate whatever image is being shown in the `ImageView` widget.

Q.2) But what if I want more flexibility than this, more like a drawing app or even a game?

A) To implement this kind of functionality, we will need to learn about another general computing concept known as **threads**, as well as some more Android classes (such as `Paint`, `Canvas`, and `SurfaceView`). We will learn how to draw anything from a single pixel to shapes, and then move them around the screen, starting in the next chapter, *Chapter 20*, *Drawing Graphics*.

## Summary

Now we have another app-enhancing trick up our sleeves. In this chapter, we saw that animations in Android are quite straightforward. We designed an animation in XML and added the file to the `anim` folder. Next, we got a reference to the animation in XML with an `Animation` object in our Kotlin code.

We then used a reference to a widget in our UI, set an animation to it using `setAnimation`, and passed in the `Animation` object. We commenced the animation by calling `startAnimation` on the reference to the widget.

We also saw that we can control the timing of animations and listen for animation events.

In the next chapter, we will learn about drawing graphics in Android. This will be the start of several chapters on graphics, where we will build a kid's-style drawing app.