Android UI Development with **Jetpack Compose**

Bring declarative and native UIs to life quickly and easily on Android using Jetpack Compose

Thomas Künneth

Table of Contents

8

Working with Animations

8
Working with Animations

In the previous chapters, I introduced you to many technical aspects of Jetpack Compose and showed you how to write well-behaving and good-looking apps. Now, adding animations and transitions will make your apps really shine! Compose simplifies the process of adding animation effects greatly over the old View-based approach.

In this chapter, you will learn important animation-related application programming interfaces, see animations of single and multiple properties, as well as transitions between composables in action, and master the relationship between state changes and visual interactions.

The main sections of this chapter are as follows:

- Using animation to visualize state changes
- Showing and hiding UI elements with animations
- Spicing up transitions through visual effects

We start by using animations to visualize state changes. Think of a simple use case: clicking a button might change the color of a UI object. But, just switching between colors feels somewhat abrupt, whereas a gradual change is much more visually pleasing. Also, if you want to change several values during the animation, Jetpack Compose can do that easily, too. I'll introduce you to the `updateTransition()` composable, which is used in such scenarios.

The *Showing and hiding UI elements with animations* section introduces you to the `AnimatedVisibility()` composable function. It allows you to apply enter and exit transitions, which will be played back while the content appears or disappears. We will also animate size changes and learn about the corresponding `animateContentSize()` modifier.

In the *Spicing up transitions through visual effects* section, we will be using the `Crossfade()` composable to switch between two layouts with a crossfade animation. Furthermore, you will learn about `AnimationSpec`. This interface represents the specification of an animation. A take on infinite animations concludes the section.

Technical requirements

This chapter is based on the `AnimationDemo` sample. Please refer to the *Technical requirements* section in *Chapter 1*, *Building Your First Compose App*, for information about how to install and set up Android Studio, and how to get the repository accompanying this book.

All the code files for this chapter can be found on GitHub at `https://github.com/PacktPublishing/Android-UI-Development-with-Jetpack-Compose/tree/main/chapter_08`.

Using animation to visualize state changes

State is app data that may change over time. In a Compose app, state (for example, a color) is represented through `State` or `MutableState` instances. State changes trigger recompositions. The following example shows a button and a box. Clicking the button toggles the color of the box between red and white by changing state:

```
@Composable
fun StateChangeDemo() {
  var toggled by remember {
    mutableStateOf(false)
  }
  val color = if (toggled)
    Color.White
  else
    Color.Red
  Column(
    modifier = Modifier
```

```
        .fillMaxSize()
        .padding(16.dp),
      horizontalAlignment = Alignment.CenterHorizontally
    ) {
      Button(onClick = {
        toggled = !toggled
      }) {
        Text(
          stringResource(R.string.toggle)
        )
      }
      Box(
        modifier = Modifier
          .padding(top = 32.dp)
          .background(color = color)
          .size(128.dp)
      )
    }
  }
```

In this example, `color` is a simple immutable variable. It is set each time `toggled` (a mutable `Boolean` state) changes (this happens inside `onClick`). As `color` is used with a modifier applied to `Box()` (`background(color = color)`), clicking the button changes the box color.

If you try the code, the switch feels very sudden and abrupt. This is because white and red are not very similar. Using an animation will make the change much more pleasant. Let's see how this works.

Animating single value changes

To animate a color, you can use the built-in `animateColorAsState()` composable. Replace the `val color = if (toggled) ...` assignment inside `StateDemo()` with the following code block. If you want to try it out, you can find a composable function called `SingleValueAnimationDemo()` in `AnimationDemoActivity.kt`, which belongs to the `AnimationDemo` sample:

```
  val color by animateColorAsState(
    targetValue = if (toggled)
      Color.White
```

```
    else
      Color.Red
  )
```

`animateColorAsState()` returns a `State<Color>` instance. Whenever `targetValue` changes, the animation will run automatically. If the change occurs while the animation is in progress, the ongoing animation will adjust to match the new target value.

> **Tip**
> Using the by keyword, you can access the color state like ordinary variables.

You can provide an optional listener to get notified when the animation is finished. The following line of code prints the color that matches the new state:

```
finishedListener = { color -> println(color)}
```

To customize your animation, you can pass an instance of `AnimationSpec<Color>` to `animateColorAsState()`. The default value is `colorDefaultSpring`, a private value in `SingleValueAnimation.kt`:

```
private val colorDefaultSpring = spring<Color>()
```

`spring()` is a top-level function in `AnimationSpec.kt`. It receives a damping ratio, a stiffness, and a visibility threshold. The following line of code makes the color animation very soft:

```
animationSpec = spring(stiffness = Spring.StiffnessVeryLow)
```

`spring()` returns `SpringSpec`. This class implements the `FiniteAnimationSpec` interface, which in turn extends `AnimationSpec`. This interface defines the specification of an animation, which includes the data type to be animated and the animation configuration, in this case, a spring metaphor. There are others. We will be returning to this interface in the *Spicing up transitions through visual effects* section. Next, we look at animating multiple value changes.

Animating multiple value changes

In this section, I will show you how to animate several values at once upon a state change. The setup is similar to `StateDemo()` and `SingleValueAnimationDemo()`: a `Column()` instance contains a `Button()` instance and a `Box()` instance. But this time, the content of the box is `Text()`. The button toggles a state, which starts the animation.

The following version of `MultipleValuesAnimationDemo()` does not yet contain an animation. It will be inserted below the comment reading **`FIXME: animation setup missing`**:

```
@Composable
fun MultipleValuesAnimationDemo() {
  var toggled by remember {
    mutableStateOf(false)
  }
  // FIXME: animation setup missing
  Column(
    modifier = Modifier
      .fillMaxSize()
      .padding(16.dp),
    horizontalAlignment = Alignment.CenterHorizontally
  ) {
    Button(onClick = {
      toggled = !toggled
    }) {
      Text(
        stringResource(R.string.toggle)
      )
    }
    Box(
      contentAlignment = Alignment.Center,
      modifier = Modifier
        .padding(top = 32.dp)
        .border(
          width = borderWidth,
          color = Color.Black
        )
        .size(128.dp)
    ) {
      Text(
        text = stringResource(id = R.string.app_name),
        modifier = Modifier.rotate(degrees = degrees)
      )
```

```
      }
    }
  }
```

The `Box()` shows a black border, whose width is controlled by `borderWidth`. To apply borders to your composable functions, just add the `border()` modifier. `Text()` is rotated. You can achieve this with the `rotate()` modifier. The `degrees` variable holds the angle. `degrees` and `borderWidth` will change during the animation. Here's how this is done:

```
val transition = updateTransition(targetState = toggled)
val borderWidth by transition.animateDp() { state ->
  if (state)
    10.dp
  else
    1.dp
}
val degrees by transition.animateFloat() { state ->
  if (state) -90F
  else
    0F
}
```

The `updateTransition()` composable function configures and returns a `Transition`. When `targetState` changes, the transition will run all of its child animations toward their target values. Child animations are added using `animate... ()` functions. They are not part of a `Transition` instance but are extension functions. `animateDp()` adds an animation based on density-independent pixels.

In my example, it controls the border width. `animateFloat()` creates a `Float` animation. This function is ideal for changing the rotation of `Text()`, which is a `Float` value. There are more `animate...()` functions, which operate on other data types. For example, `animateInt()` works with `Int` values. `animateOffset()` animates an `Offset` instance. You can find them in the `Transition.kt` file, which belongs to the `androidx.compose.animation.core` package.

`Transition` instances provide several properties reflecting the status of a transition. For example, `isRunning` indicates whether any animation in the transition is currently running. `segment` contains the initial state and the target state of the currently ongoing transition. The current state of the transition is available through `currentState`. This will be the initial state until the transition is finished. Then, `currentState` is set to the target state.

As you have seen, it is very easy to use state changes to trigger animations. So far, these animations have modified the visual appearance of one or more composable functions. In the next section, I will show you how to apply animations while showing or hiding UI elements.

Showing and hiding UI elements with animations

Often, your UI will contain information that need not be visible all the time. For example, in an address book you may want to show only key attributes of a contact, and present detailed information upon request, typically after a button click. However, just showing and hiding the additional data feels sudden and abrupt. Using animations leads to a more pleasant experience, so let's look into this more.

Understanding AnimatedVisibility()

In this section, we will look at my sample composable `AnimatedVisibilityDemo()`. It belongs to the `AnimationDemo` project. Like `StateDemo()`, `SingleValueAnimationDemo()`, and `MultipleValuesAnimationDemo()`, it uses a `Column()` instance, which contains a `Button()` instance and a `Box()` instance. This part of the code is simple and straightforward, so there is no need to repeat it in print. The button toggles a state, which starts the animation. Let's see how this works:

```
AnimatedVisibility(
  visible = visible,
  enter = slideInHorizontally(),
  exit = slideOutVertically()
) {
  Box(
    modifier = Modifier
      .padding(top = 32.dp)
      .background(color = Color.Red)
      .size(128.dp)
  )
}
```

The box is wrapped in `AnimatedVisibility()`. This built-in composable function animates the appearance and disappearance of its content, when the `visible` parameter changes. You can specify different `EnterTransition` and `ExitTransition` instances. In my example, the box enters by sliding in horizontally and exits by sliding out vertically.

Currently, there are three transition types:

- Fade
- Expand and shrink
- Slide

They can be combined using `+`:

```
enter = slideInHorizontally() + fadeIn(),
```

The combination order doesn't matter as the animations start simultaneously.

If you do not pass a value for `enter`, the content will default to fading in while expanding vertically. Omitting `exit` will cause the content to fade out while shrinking vertically.

> **Please Note**
>
> At the time of writing, `AnimatedVisibility()` is experimental. To use it in your app, you must add the `@ExperimentalAnimationApi` annotation. This will change with Jetpack Compose 1.1.

In this section, I showed you how to animate the appearance and disappearance of content. A variation of this subject is to visualize size changes (if either `width`, `height`, or both are `0`, the UI element is no longer visible). Let's find out how to do this in the following section.

Animating size changes

Sometimes you may want to change the amount of space a UI element requires onscreen. Think of text fields. In compact mode, your app could show only three lines, whereas in detail mode it might display 10 lines or more. My `SizeChangeAnimationDemo()` sample composable (*Figure 8.1*) uses a slider to control the `maxLines` value of `Text()`:

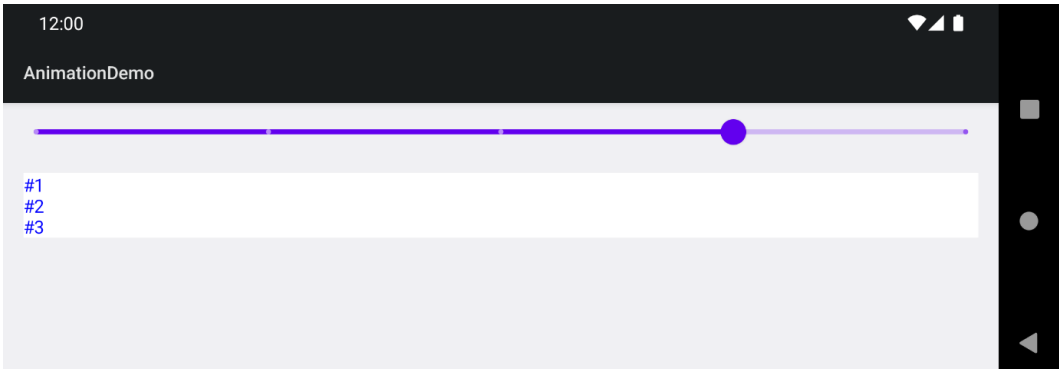

Figure 8.1 – The AnimationDemo sample showing SizeChangeAnimationDemo()

The general setup follows the examples from the previous sections: a `Column()` instance acts as a container for some composable functions, in this case a `Slider()` instance and a `Text()` instance. Then a state change triggers the animation. Here is the code:

```
@Composable
fun SizeChangeAnimationDemo() {
  var size by remember { mutableStateOf(1F) }
  Column(
    modifier = Modifier
      .fillMaxSize()
      .padding(16.dp)
  ) {
    Slider(
      value = size,
      valueRange = (1F..4F),
      steps = 3,
      onValueChange = {
        size = it
      },
      modifier = Modifier.padding(bottom = 8.dp)
    )
    Text(
      text = stringResource(id = R.string.lines),
      modifier = Modifier
        .fillMaxWidth()
        .background(Color.White)
```

```
            .animateContentSize(),
        maxLines = size.toInt(),
        color = Color.Blue
      )
    }
  }
```

`size` is a mutable `Float` state. It is passed to `Slider()` as its default value. When the slider is moved, `onValueChange {}` is invoked. The lambda expression receives the new value, which is assigned to `size`. The `Text()` composable uses the state as a value for `maxLines`.

The animation is handled by the `animateContentSize()` modifier. It belongs to the `androidx.compose.animation` package. The modifier expects two parameters, `animationSpec` and `finishedListener`. I introduced both briefly in the *Animating single value changes* section. `animationSpec` defaults to `spring()`. If you want the lines to appear all at once after some delay, you can add the following:

```
animationSpec = snap(1000)
```

A snap animation immediately switches the animating value to the end value. You pass the number of milliseconds to wait before the animation runs. It defaults to `0`. Now, `snap()` returns an instance of `SnapSpec`, an implementation of `AnimationSpec`. We will turn to this interface in the *Spicing up transitions through visual effects* section.

The default value of `finishedListener` is `null`. You can provide an implementation, if your app wants to get notified when the size change animation is finished. Both the initial value and the final size are passed to the listener. If the animation is interrupted, the initial value will be the size at the point of interruption. This helps determine the direction of the size change.

This concludes our look at showing and hiding UI elements with animations. In the next section, we focus on exchanging parts of the UI. For example, we will be using `Crossfade()` to switch between two composable functions with a crossfade animation.

Spicing up transitions through visual effects

So far, I have shown you animations that modify certain aspects of a UI element, like its color, size, or visibility. But sometimes you may want to *exchange* parts of your UI. Then, `Crossfade()` comes in handy. It allows you to switch between two composable functions with a crossfade animation. Let's look at my `CrossfadeAnimationDemo()` sample (*Figure 8.2*), part of the `AnimationDemo` project, to see how this works:

Figure 8.2 – The AnimationDemo sample showing CrossfadeAnimationDemo()

A switch toggles between two screens. As we are focusing on animation, I kept the `Screen()` composable very simple, just a box with customizable background color, and a big text centered inside. You can find its source code in `AnimationDemoActivity.kt`.

Crossfading composable functions

Like most examples in this chapter, `CrossfadeAnimationDemo()` uses a `Column()` as the root element. The column contains a switch, and the screen to display. Which one is shown depends on a mutable `Boolean` state:

```
@Composable
fun CrossfadeAnimationDemo() {
  var isFirstScreen by remember { mutableStateOf(true) }
  Column(
    modifier = Modifier
      .fillMaxSize(),
    horizontalAlignment = Alignment.CenterHorizontally
  ) {
    Switch(
      checked = isFirstScreen,
      onCheckedChange = {
```

```
          isFirstScreen = !isFirstScreen
        },
        modifier = Modifier.padding(top = 16.dp,
                                    bottom = 16.dp)
      )
      Crossfade(targetState = isFirstScreen) { it ->
        if (it) {
          Screen(
            text = stringResource(id = R.string.letter_w),
            backgroundColor = Color.Gray
          )
        } else {
          Screen(
            text = stringResource(id = R.string.letter_i),
            backgroundColor = Color.LightGray
          )
        }
      }
    }
  }
```

The `onCheckedChange` lambda expression of `Switch()` toggles `isFirstScreen`. This state is passed to `Crossfade()` as the `targetState` parameter. Like in the other animations I showed you so far, it triggers the animation every time the value changes. Specifically, the content called with the old value will be faded out, and the content called with the new one will be faded in.

`Crossfade()` receives an `animationSpec` of type `FiniteAnimationSpec<Float>`. It defaults to `tween()`. This function returns a `TweenSpec` instance configured with the given duration, delay, and easing curve. The parameters default to `DefaultDurationMillis` (300 ms), `0`, and `FastOutSlowInEasing`. The easing curve is represented by instances of `CubicBezierEasing`. This class models third-order Bézier curves. Its constructor receives four parameters:

- The *x* and *y* coordinates of the first control point
- The *x* and *y* coordinates of the second control point

The documentation explains that the line through the point (0, 0) and the first control point is tangent to the easing at the point (0, 0), and that the line through the point (1, 1) and the second control point is tangent to the easing at the point (1, 1). `CubicBezierEasing` is an implementation of the `Easing` interface (the `androidx.compose.animation.core` package). Besides `FastOutSlowInEasing`, you can choose from three other predefined curves: `LinearOutSlowInEasing`, `FastOutLinearInEasing`, and `LinearEasing` to customize your animation.

As `Crossfade()` receives an `animationSpec` of type `FiniteAnimationSpec<Float>`, you can, for example, pass the following code to use a spring animation with very low stiffness:

```
animationSpec = spring(stiffness = Spring.StiffnessVeryLow)
```

In the next section, we look at how the different specifications of an animation are related.

Understanding animation specifications

`AnimationSpec` is the base interface for defining animation specifications. It stores the data type to be animated and the animation configuration. Its only function, `vectorize()`, creates a `VectorizedAnimationSpec` instance with the given `TwoWayConverter` (which converts a given type to and from `AnimationVector`).

The animation system operates on `AnimationVector` instances. `VectorizedAnimationSpec` describes how these vectors should be animated, for example, simply interpolating between the start and end values (as you have seen with `TweenSpec`), showing no animation at all (`SnapSpec`), or applying spring physics to produce the motion (`SpringSpec`).

The `FiniteAnimationSpec` interface extends `AnimationSpec`. It is directly implemented by the `RepeatableSpec` and `SpringSpec` classes. It overrides `vectorize()` to return `VectorizedFiniteAnimationSpec`. Now, `FiniteAnimationSpec` is the parent of the interface `DurationBasedAnimationSpec`, which overrides `vectorize()` to return `VectorizedDurationBasedAnimationSpec`. Then, `DurationBasedAnimationSpec` is implemented by the `TweenSpec`, `SnapSpec`, and `KeyframesSpec` classes.

To create a `KeyframesSpec` instance, you can invoke the `keyframes()` function and pass an initialization function for the animation. After the duration of the animation, you pass mappings of the animating value at a given amount of time in milliseconds:

```
animationSpec = keyframes {
      durationMillis = 8000
```

```
        0f at 0
        1f at 2000
        0f at 4000
        1f at 6000
    }
```

In this example, the animation takes 8 seconds, which is longer than you'd ever practically use, but allows you to observe the changes. If you apply the code snippet to `CrossfadeAnimationDemo()`, you will notice that each letter is visible twice during the course of the animation.

So far, we have looked at finite animations. What if you want an animation to continue forever? Jetpack Compose does this in the `CircularProgressIndicator()` and `LinearProgressIndicator()` composables. `InfiniteRepeatableSpec` repeats the provided animation until it is canceled manually.

When used with transitions or other animation composables, the animation will stop when the composable is removed from the compose tree. `InfiniteRepeatableSpec` implements `AnimationSpec`. The constructor expects two arguments, `animation` and `repeatMode`. The `RepeatMode` enum class defines two values, `Restart` and `Reverse`. The default value for `repeatMode` is `RepeatMode.Restart`, meaning each repeat restarts from the beginning.

You can use `infiniteRepeatable()` to create an `InfiniteRepeatableSpec` instance. My `InfiniteRepeatableDemo()` sample composable (*Figure 8.3*) shows you how to do this:

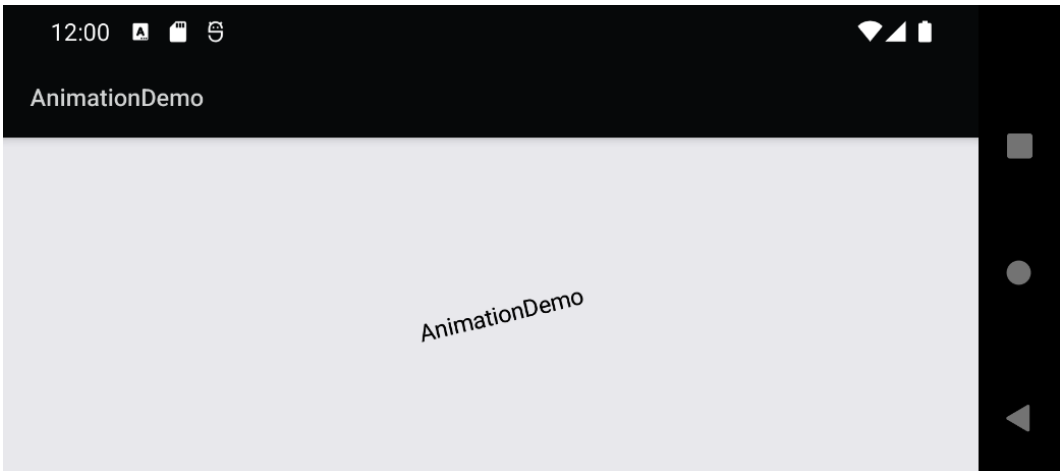

Figure 8.3 – The AnimationDemo sample showing InfiniteRepeatableDemo()

The composable rotates a text clockwise from 0 to 359 degrees. Then, the animation restarts. `Text()` is centered inside `Box()`:

```
@Composable
fun InfiniteRepeatableDemo() {
  val infiniteTransition = rememberInfiniteTransition()
  val degrees by infiniteTransition.animateFloat(
    initialValue = 0F,
    targetValue = 359F,
    animationSpec = infiniteRepeatable(animation =
                        keyframes {
      durationMillis = 1500
      0F at 0
      359F at 1500
    })
  )
  Box(
    modifier = Modifier.fillMaxSize(),
    contentAlignment = Alignment.Center
  ) {
    Text(text = stringResource(id = R.string.app_name),
    modifier = Modifier.rotate(degrees = degrees))
  }
}
```

To create a potentially infinite animation, you first need to remember an infinite transition using `rememberInfiniteTransition()`. You can then invoke `animateFloat()` on the transition instance. This returns `State<Float>`, which is used with the `rotate()` modifier. `infiniteRepeatable()` is passed to `animateFloat()` as its `animationSpec` parameter. The animation itself is based on keyframes. We need to define only two frames, the first representing the start, and the second representing the end angle.

If you want the text to return to its initial angle rather than rotating continuously, you can change the `repeatMode` parameter to the following:

```
repeatMode = RepeatMode.Reverse
```

Then you should add short delays to the beginning and the end. `keyframes {}` should look like this:

```
keyframes {
      durationMillis = 2000
      0F at 500
      359F at 1500
}
```

This concludes our look at animation specifications. To finish this chapter, let me briefly summarize what you have learned, and what you can expect in the next chapter.

Summary

This chapter showed you how easy it is to use Jetpack Compose to enrich your apps with animations and transitions. We started by using simple animations to visualize state changes. For example, I introduced you to `animateColorAsState()`. We then used `updateTransition()` to obtain `Transition` instances and invoked extension functions such as `animateDp()` and `animateFloat()` to animate several values based on state changes simultaneously.

The *Showing and hiding UI elements with animations* section introduced you to the `AnimatedVisibility()` composable function, which allows you to apply enter and exit transitions. They are played back while the content appears or disappears. You also learned how to animate size changes using the `animateContentSize()` modifier.

In the final main section, *Spicing up transitions through visual effects*, we used the `Crossfade()` composable function to switch between two layouts with a crossfade animation. Furthermore, you learned about `AnimationSpec` and related classes and interfaces. I concluded the section with a take on infinite animations.

In *Chapter 9*, *Exploring Interoperability APIs*, you will learn how to mix old-fashioned views and composable functions. We will once again return to ViewModels as a means for sharing data between both worlds. And I will show you how to integrate third-party libraries in your Compose app.