Jetpack Compose

the big nerd ranch guide

Jetpack Compose: The Big Nerd Ranch Guide

Table of Contents

1

Introduction to Jetpack Compose

Since the beginning of Android, developers have built UIs using **`View`** classes and XML layout files. These APIs are provided by the Android OS and are part of the Android framework UI toolkit. Colloquially, we refer to these APIs as "framework views."

Building UIs with framework views has been the standard for making an Android app since the first release of the OS. But in recent times, the framework view system has left much to be desired. For starters, it is built into the OS itself. This means that getting the latest features requires users to update their entire OS, which is not always an option. It also requires developers to bump their apps' minimum SDK level, leaving behind users who are not able to upgrade.

Also, Android's framework UI toolkit is based around ideas like the view hierarchy, view classes that extend from one another, and updating the state of your view manually, line by line. Meanwhile, many front-end UI frameworks have moved on to more modern approaches that make building UIs easier and more streamlined.

To address both of these issues, Google has created a new UI toolkit called *Jetpack Compose*. Jetpack Compose replaces the built-in framework UI toolkit. It is part of the Jetpack suite of libraries, so a UI built with Compose is entirely separate from the Android OS. And because it is separate, you can get updates to Compose just as you would any external library.

Compose is designed in Kotlin (and, in fact, is exclusively available in Kotlin) and is a *declarative* UI framework. The benefits of a declarative UI toolkit will become apparent in the next chapter, when you learn about UI state in Compose. To give you a teaser, though: Compose automatically updates your UI when your application state changes. You *declare* your UI how you want it to appear at all times, and Compose will make it so.

This marks a radical departure from what you are used to. Jetpack Compose does not let you store a reference to any of your UI elements, which means no View Binding, no imperative UI updates – even the time-honored **`findViewById()`** function is not available in Compose. Initially, you might find Compose a bit tricky to reason about, since it requires you to think about your UI differently. However, as you will see in the next chapter, Compose plays well with the modern Android programming paradigms you have used throughout this book like making your UI state observable and updating UI elements reactively.

In our opinion, Jetpack Compose offers a more elegant and concise set of tools to build UIs than what is available in the framework UI toolkit. Google is also emphasizing Jetpack Compose, and we expect that many apps in the future will exclusively use Compose and leave framework views behind.

You may be wondering, "If Compose is the latest and greatest, why bother learning about the framework UI toolkit as well?" We are glad you asked.

Jetpack Compose hit version 1.0 and went stable in the summer of 2021. Since then, the Android development community has begun to transition to Compose – but a transition of this size takes time. If you are just starting your journey as an Android developer, you likely need to be familiar with *both* UI frameworks, as many existing apps, libraries, code snippets, and examples still rely on framework views and will for some time.

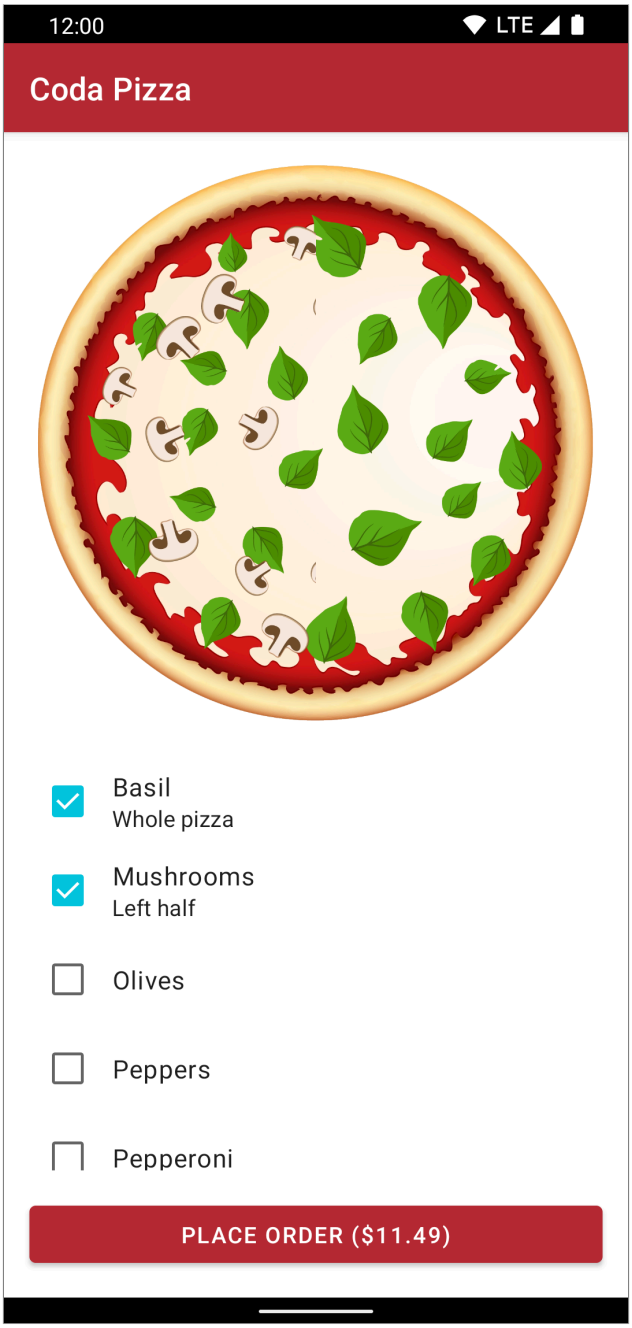
Over the next few chapters, we will walk you through the basics of building UIs in Compose. We will not be able to cover every feature Compose offers, but you will end with a solid foundation to build UIs in Compose. You can find more information about Jetpack Compose on its documentation page, `developer.android.com/jetpack/compose/documentation`.

Although the UI code in this project will be different than what you have been seeing, fear not. You are still working on Android – everything you have learned so far will be helpful as you venture into this new territory. Let's get started.

Creating a Compose Project

To get your feet wet with Compose, you will be creating an app for a pizza delivery service that allows users to customize their pizzas' toppings. This app will be called Coda Pizza. The finished product will look like Figure 1.1. In this chapter, you will focus on building out the scrollable list of toppings.

Figure 1.1 The finished product



Android Studio offers a template to create an empty Compose app, but you will not use it for Coda Pizza. The Compose templates in Android Studio include a fair amount of code that would just get in your way – plus they are likely to change as Compose evolves.

Instead, we will walk you through setting up a new project and then adding Jetpack Compose. This will allow you to explore Compose in more detail, and the steps involved in setting Compose up will be helpful if you find yourself migrating an existing app away from framework views. (Outside of this book, we encourage you to use the Compose templates for new apps once you have mastered the basics.)

Create a new Android Studio project with the name Coda Pizza and the package name `com.bignerdranch.android.codapizza`. Be sure to use the Empty Activity template, as it generates a fairly bare-bones project. Set the Minimum SDK to 24, and save the project wherever you would like.

With your new project open, your first task is to add Jetpack Compose. This is a multistep process. Compose is enabled in the **`buildFeatures`** block, like View Binding, but you also have to specify the Compose compiler version and add several dependencies. Delve into your `app/build.gradle` file (the one labeled (Module: Coda_Pizza.app)) and make these changes now.

Listing 1.1 Becoming a composer (`app/build.gradle`)

```
...
android {
    ...
    buildTypes {
        ...
    }

    buildFeatures {
        compose true
    }

    composeOptions {
        kotlinCompilerExtensionVersion '1.2.0'
    }

    compileOptions {
        ...
    }
    ...
}

dependencies {
    ...
    implementation 'androidx.constraintlayout:constraintlayout:2.1.4'

    implementation 'androidx.compose.foundation:foundation:1.2.1'
    implementation 'androidx.compose.runtime:runtime:1.2.1'
    implementation 'androidx.compose.ui:ui:1.2.1'
    implementation 'androidx.compose.ui:ui-tooling:1.2.1'
    implementation 'androidx.compose.material:material:1.2.1'
    implementation 'androidx.activity:activity-compose:1.5.1'

    testImplementation 'junit:junit:4.13.2'
    ...
}
```

Because Compose is built on the latest features in Kotlin, it has specific requirements about which version of Kotlin it supports. Compose performs many transformations on your code during compilation and uses cutting-edge features within Kotlin to make them work. The Compose compiler 1.2.0 requires Kotlin 1.7.0 – *exactly*. Double-check that your project's `build.gradle` file (the one labeled (Project: Coda_Pizza)) specifies this version, otherwise you will run into build errors.

Listing 1.2 Matching the Kotlin compiler version (`build.gradle`)

```
plugins {
    id 'com.android.application' version '7.2.2' apply false
    id 'com.android.library' version '7.2.2' apply false
    id 'org.jetbrains.kotlin.android' version '1.7.0' apply false
}
...
```

When you are done, sync your Gradle files to apply these changes.

Next, it is time to delete some code. Coda Pizza will be 100% Compose, so spend a moment to remove the current layout code. Start by removing the call to **`setContentView`** in **`MainActivity`**.

Listing 1.3 Removing the content view (`MainActivity.kt`)

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

Then, delete the `activity_main.xml` file from your `layout` resources folder.

Composing Your First UI

With your framework views out of the way, you are ready to write your first Compose UI. The default layout in the Empty Activity project template you have used throughout this book includes an empty **`Activity`** with the text “Hello World!” For your first Compose UI, you will remake this layout without any framework views. (Printing “Hello World!” to the screen is a time-honored coding tradition.)

To populate an activity with a Compose UI, you use a function called **`setContent`**. This function accepts a lambda expression, which is where you have access to Compose UI elements, called *composables*. Use the **`Text`** composable to show the text “Hello World!”

Listing 1.4 Writing a Compose UI (`MainActivity.kt`)

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Text(text = "Hello World!")
        }
    }
}
```

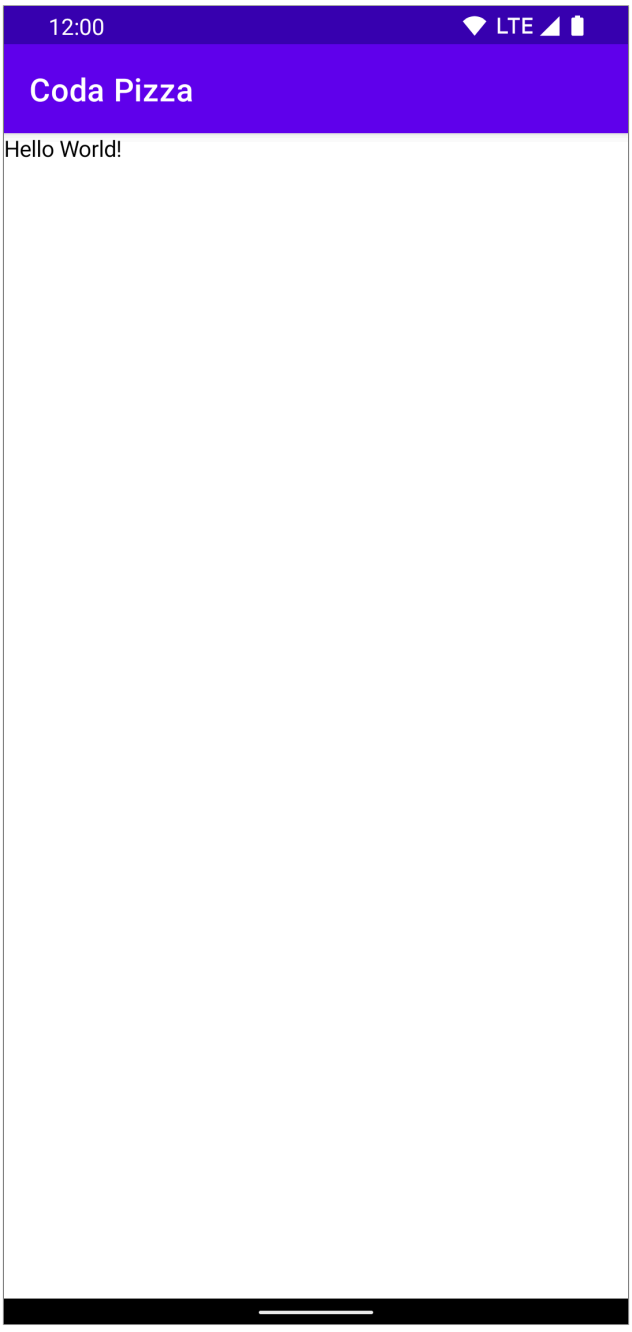
You will need to add two import statements for the code you just entered. For **`setContent`**, add an import for `androidx.activity.compose.setContent`, since you are setting an activity’s content. For the **`Text`** composable, add an import for `androidx.compose.material.Text`.

When using Jetpack Compose, you will find that there are many import statements to juggle, and it is not always obvious which you should choose from Android Studio’s list of suggestions. In general, your Compose imports will start with the `androidx.compose` package. Also, if you are looking to import a composable function, you can quickly identify them by looking closely at the icon in Android Studio’s list of suggested imports. Imports for composables will be marked with  .

Run Coda Pizza and confirm that your text appears as shown in Figure 1.2.

(Coda Pizza is not yet set up to handle dark mode, so be sure to disable the dark theme if you are using it on your test device. If you do not, Coda Pizza will show black text against a black background, making your text appear invisible. After Chapter 4, Coda Pizza will be legible in both modes.)

Figure 1.2 Hello, Compose!



Although this example is rudimentary, notice how concise it is compared to the old fashioned way of building UIs. Putting text onscreen took only a line of code, and all your view code is in Kotlin – no more jumping in and out of XML.

Layouts in Compose

Time to begin building out the views that Coda Pizza will present to its users. Let's focus on the scrollable list of toppings. First, as you did for your **`RecyclerView`** in CriminalIntent, you will construct a cell that will appear for each topping choice. You will come back to the actual scrolling behavior at the end of this chapter.

The cell will have three elements: the name of the topping, a checkbox indicating whether the topping is on the pizza, and a description of where the topping will appear on the pizza (the left half, the right half, or the whole pizza). Start with the two **`Text`** elements.

Experienced Android developers probably know that flat (non-nested) layouts are faster for the OS to measure and lay out. That is true for framework views, but Compose's efficiency makes it no longer a concern. Composables can be nested to create layouts as complex as you want. For example, to arrange elements vertically from top to bottom, you can place them inside a **`Column`** composable.

`Column` is analogous to a **`LinearLayout`** with the `vertical` orientation. It accepts a lambda, and the composables added to the lambda will be arranged from top to bottom. Try it out now.

Listing 1.5 The **`Column`** composable (`MainActivity.kt`)

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Text(text = "Hello World!")
            Column {
                Text(
                    text = "Pineapple"
                )

                Text(
                    text = "Whole pizza"
                )
            }
        }
    }
}
```

Run the app after making this change. You should see Pineapple at the top left of the screen, with the text Whole pizza underneath it.

Next, shift your attention to the checkbox. The checkbox will appear to the left of the two text elements. You can accomplish this using a **`Row`**, which behaves like a **`Column`** but lays its content out from left to right (or, if the user's device is set to a right-to-left language, from right to left).

Your **`Row`** will contain the **`Column`** of text plus a **`Checkbox`** composable. You will leave the behavior of the checkbox unimplemented for now, and we will revisit it in the next chapter.

Listing 1.6 The **`Row`** composable (`MainActivity.kt`)

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Row {
                Checkbox(
                    checked = true,
                    onCheckedChange = { /* TODO */ }
                )

                Column {
                    Text(
                        text = "Pineapple"
                    )

                    Text(
                        text = "Whole pizza"
                    )
                }
            }
        }
    }
}
```

Run the app again. You should now see that a checked checkbox appears in the top-left corner of the app with the text to its right (Figure 1.3). (If you press the checkbox, its state will not change. This is expected, and we will explain why in the next chapter when we talk about state in Jetpack Compose.)

Figure 1.3 Rows and columns

Composable Functions

Before you create the scrollable list of toppings, there is a bit of housekeeping to take care of. Right now, your entire UI is defined in your **`Activity`**. This can get unwieldy quickly, especially for large applications. You can break your UI into smaller chunks by refactoring your Compose code into functions.

Composables' names, like **`Row`** and **`Column`**, begin with capital letters – just like the names of framework views like **`Button`** and **`ImageView`**. But composables are not classes, like views: They are functions.

Remember when we said you cannot get a reference to a Compose UI element or call **`findViewById`** on one? No classes means there is nothing that can be referenced. At runtime, with some help from the Compose compiler, composable functions effectively turn into draw commands.

Compose comes with many prefab composables for basic components like buttons, switches, and text input fields (in addition to the ones you have already seen), but you can also write your own composable functions. Although the built-in composables are often simple, your own composables can combine other composables and be as simple or as complex as you want.

Try writing your own composable now by converting the content inside your **`setContent`** function into its own composable. You can make this change manually, or you can use Android Studio's built-in refactoring tools to make the change automatically. Start by highlighting the code in **`setContent`**'s lambda:

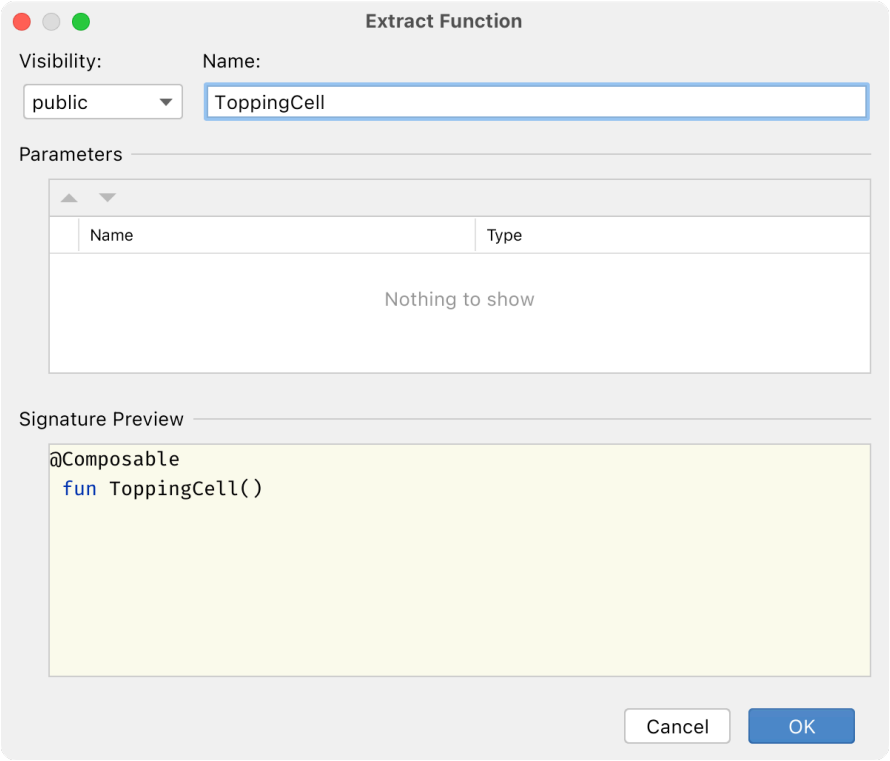
```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Row {
                Checkbox(
                    checked = true,
                    onCheckedChange = { /* TODO */ }
                )

                Column {
                    Text(
                        text = "Pineapple"
                    )

                    Text(
                        text = "Whole pizza"
                    )
                }
            }
        }
    }
}
```

Next, right-click the code and select Refactor → Function.... The Extract Function dialog will appear. Set the function's visibility to `public` and name the new function **`ToppingCell`** (Figure 1.4).

Figure 1.4 Extracting a composable function



Click OK to perform the refactor. Android Studio will extract the highlighted code into its own function. Your updated code should match the following:

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            ToppingCell()
        }
    }

    @Composable
    fun ToppingCell() {
        Row {
            Checkbox(
                checked = true,
                onCheckedChange = { /* TODO */ }
            )

            Column {
                Text(
                    text = "Pineapple"
                )

                Text(
                    text = "Whole pizza"
                )
            }
        }
    }
}
```

The new **ToppingCell** function looks almost identical to any Kotlin function. In fact, there is only one difference: the `@Composable` annotation. When a function is annotated with `@Composable`, it becomes a composable function. Composable functions can call other composable functions and can add elements onscreen when invoked. Composable functions can also call regular functions, but regular functions cannot call a composable function. (**setContent** is an exception to this rule. It can use composable functions because it is responsible for creating the composition itself.)

Notice that you named your composable **ToppingCell**, not **toppingCell**. As you have seen, it is conventional for the names of composable functions to start with a capital letter, and we recommend following this pattern.

Run the app and confirm that nothing has changed after your refactor. You should still see a checkbox and two lines of text in the top-left corner.

There is one more bit of cleanup to take care of before moving on. Although you have organized your UI into a smaller function, it is still a function on your **MainActivity** class. The composable does not access any information in your activity, so it can be declared in its own file to keep your activity small.

Create a new package called `ui` under the `com.bignerdranch.android.codapizza` package. Inside your new package, create a new file called `ToppingCell.kt`, then copy and paste your **ToppingCell** function into this file.

Listing 1.7 Putting **ToppingCell** in its own file (`ToppingCell.kt`)

```
@Composable
fun ToppingCell() {
    Row {
        Checkbox(
            checked = true,
            onCheckedChange = { /* TODO */ }
        )
        Column {
            Text(
                text = "Pineapple"
            )

            Text(
                text = "Whole pizza"
            )
        }
    }
}
```

Now you can delete the implementation of **ToppingCell** from **MainActivity**. You will need to add an import for your relocated **ToppingCell** function after making this change. (Remember, it is in the `com.bignerdranch.android.codapizza.ui` package.)

Listing 1.8 Using a composable defined in another file (`MainActivity.kt`)

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            ToppingCell()
        }
    }

    @Composable
    fun ToppingCell() {
        Row {
            Checkbox(
                checked = true,
                onCheckedChange = { /* TODO */ }
            )
            Column {
                Text(
                    text = "Pineapple"
                )

                Text(
                    text = "Whole pizza"
                )
            }
        }
    }
}
```

Previewing Composables

If you are a fan of Android Studio's design view for XML layouts, you may be wondering if you can preview your Compose layouts the same way. Preview functionality is available for Compose, but Android Studio needs a little help: You must opt in to previews for each composable. Do this now for your **`ToppingCell`** composable by annotating it with the `@Preview` annotation.

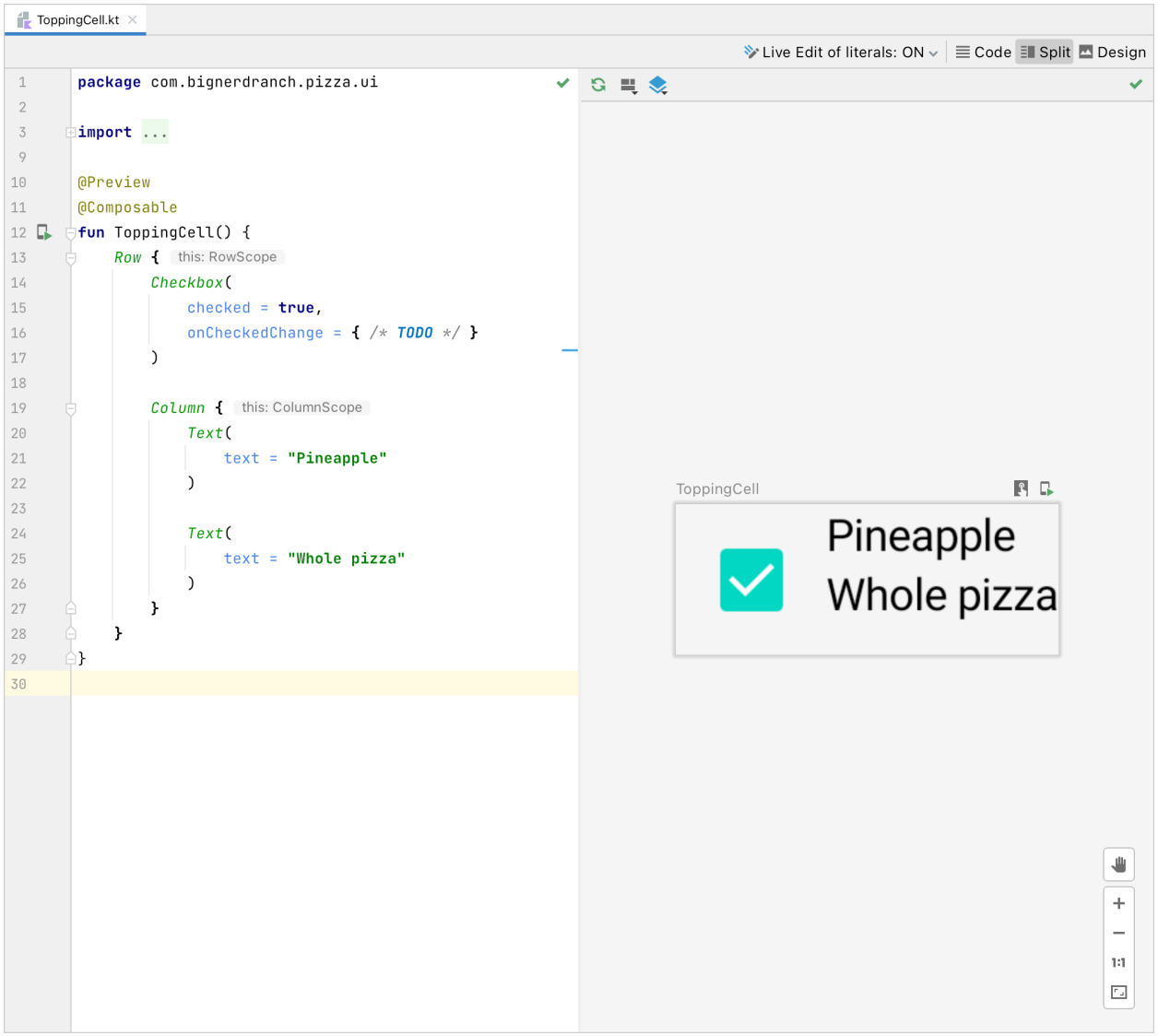
Listing 1.9 Enabling previews on a composable (`ToppingCell.kt`)

```
@Preview
@Composable
fun ToppingCell() {
    ...
}
```

Android Studio uses your project's compiled code to generate previews of composables. This means that your project must be built to show changes you have made in the preview. Build your project now by pressing the  Build icon in Android Studio's toolbar.

When the build finishes, click the Split tab with `ToppingCell.kt` open in the editor. You should see the preview in the right side of the editor (Figure 1.5). (If you do not, check that your project has built successfully, with no errors.)

Figure 1.5 Previewing a composable



As you saw with XML layouts, the preview matches what the user will see when your composable is placed onscreen. Remember that Android Studio needs to recompile your code before it can update the preview, so your changes will not appear instantly the way they did with XML layouts.

The `@Preview` annotation has one very noteworthy limitation: By default, it is not able to show previews for composables that have parameters (unless each parameter has a default value). When this happens, you need to specify the values to use for those arguments. There is a mechanism for doing this with the `@Preview` annotation, but we find it to be cumbersome to set up. Instead, many developers create a separate preview function. This dedicated preview function can pass the desired inputs to the composable being previewed while specifying no inputs of its own.

In just a moment, you are going to add parameters to your **`ToppingCell`** composable. To avoid breaking its preview, preemptively add a separate preview function in the same file:

Listing 1.10 A dedicated preview composable (`ToppingCell.kt`)

```
@Preview
@Composable
private fun ToppingCellPreview() {
    ToppingCell()
}

@Preview
@Composable
fun ToppingCell() {
    ...
}
```

This preview composable uses the `private` visibility modifier, allowing you to define the preview without exposing it for use in your production code. Refresh the preview by rebuilding the project or by clicking the Build Refresh button in the preview window. The preview should look identical except for being labeled ToppingCellPreview instead of ToppingCell.

Customizing Composables

Coda Pizza is off to a great start. You are ready to start making your composables look just the way you want them to. Previously, you accomplished this using XML attributes. In Compose, function parameters take the place of the attributes that you are accustomed to in XML.

You have already seen a few parameters on the built-in composables you have been using: `text` for the **`Text`** composable and `checked` and `onCheckedChange` for **`Checkbox`**. You are also free to add parameters to your own composables.

Declaring inputs on a composable function

Think about the **`ToppingCell`** composable. It will need to take in three pieces of information: the name of the topping, the placement of the topping, and what to do when the topping is clicked. Currently, these values are hardcoded – the topping is always pineapple, it is placed on the whole pizza, and nothing happens when you try to edit the topping. This will upset opponents of pineapple on pizza, so it is time to make your toppings more flexible.

The set of toppings and the options for the position of toppings will both have a fixed set of values. Instead of representing these using `String`s, enums are a better fit. Also, the hardcoded strings you have been using would not be easy to localize. Jetpack Compose supports loading from your string resources, and it is a good idea to use them.

So start by defining some string resources:

Listing 1.11 Adding string resources (`strings.xml`)

```
<resources>
    <string name="app_name">Coda Pizza</string>

    <string name="placement_none">None</string>
    <string name="placement_left">Left half</string>
    <string name="placement_right">Right half</string>
    <string name="placement_all">Whole pizza</string>

    <string name="topping_basil">Basil</string>
    <string name="topping_mushroom">Mushrooms</string>
    <string name="topping_olive">Olives</string>
    <string name="topping_peppers">Peppers</string>
    <string name="topping_pepperoni">Pepperoni</string>
    <string name="topping_pineapple">Pineapple</string>
</resources>
```

Next, create a new package called `com.bignerdranch.android.codapizza.model` to store the model classes you will use to define and represent a pizza. Create a new file in this package called `ToppingPlacement.kt` and define an enum to specify which part of a pizza a topping is present on.

Give the enum three cases: the whole pizza, the left half of the pizza, and the right half of the pizza. If a topping is not present on the pizza, you can represent that with a `null` value instead.

Listing 1.12 Specifying topping locations (`ToppingPlacement.kt`)

```
enum class ToppingPlacement(
    @StringRes val label: Int
) {
    Left(R.string.placement_left),
    Right(R.string.placement_right),
    All(R.string.placement_all)
}
```

(The `@StringRes` annotation is not required, but it helps Android Lint verify at compile time that constructor calls provide a valid string resource ID.)

Next, define another enum to specify all the toppings that a customer can add to their pizza. Place this enum in a new file called `Topping.kt` in the `model` package, and populate it as shown:

Listing 1.13 Specifying toppings (`Topping.kt`)

```
enum class Topping(
    @StringRes val toppingName: Int
) {
    Basil(
        toppingName = R.string.topping_basil
    ),

    Mushroom(
        toppingName = R.string.topping_mushroom
    ),

    Olive(
        toppingName = R.string.topping_olive
    ),

    Peppers(
        toppingName = R.string.topping_peppers
    ),

    Pepperoni(
        toppingName = R.string.topping_pepperoni
    ),

    Pineapple(
        toppingName = R.string.topping_pineapple
    )
}
```

With the models in place, you are ready to add parameters to **`ToppingCell`**. You will add three parameters: a `topping`, a nullable `placement`, and an `onClickTopping` callback. Be sure to provide values for these parameters in your preview composable, otherwise you will get a compiler error.

Listing 1.14 Adding parameters to a composable (`ToppingCell.kt`)

```
@Preview
@Composable
private fun ToppingCellPreview() {
    ToppingCell(
        topping = Topping.Pepperoni,
        placement = ToppingPlacement.Left,
        onClickTopping = {}
    )
}

@Composable
fun ToppingCell(
    topping: Topping,
    placement: ToppingPlacement?,
    onClickTopping: () -> Unit
) {
    ...
}
```

You will also need to update **`MainActivity`** to provide these arguments when it calls **`ToppingCell`**. Currently, **`MainActivity`** has a compiler error, which will prevent the preview from updating. Fix this now by specifying the required arguments for **`ToppingCell`**. You will revisit the `onClickTopping` callback later. For now, implement it with an empty lambda.

Listing 1.15 Fixing the compiler error (`MainActivity.kt`)

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            ToppingCell(
                topping = Topping.Pepperoni,
                placement = ToppingPlacement.Left,
                onClickTopping = {}
            )
        }
    }
}
```

Return to `ToppingCell.kt` and build the project to update the preview. Thanks to the changes you just made to **`ToppingCellPreview`**, you might expect the preview to show pepperoni on just the left side of the pizza. However, it still shows pineapple on the whole pizza. This is because you have not yet used the new inputs in your **`ToppingCell`**. Let's change that.

Resources in Compose

Start with the name of the topping. With the framework views you have seen before, you used the **`Context.getString(Int)`** function to turn a string resource into a `String` object you could show onscreen. In Compose, you can accomplish the same thing using the **`stringResource(Int)`** function. Take it for a spin.

Listing 1.16 Using string resources in Compose (`ToppingCell.kt`)

```
...
@Composable
fun ToppingCell(
    topping: Topping,
    placement: ToppingPlacement?,
    onClickTopping: () -> Unit
) {
    Row {
        Checkbox(
            checked = true,
            onCheckedChange = { /* TODO */ }
        )

        Column {
            Text(
                text = "Pineapple"
                text = stringResource(topping.toppingName)
            )

            Text(
                text = "Whole pizza"
            )
        }
    }
}
```

Build and refresh the preview. You should see that the topping name changes from the hardcoded Pineapple string to the Pepperoni string from your string resources. (If you wanted, you could also specify a specific string resource instead of accessing it in a variable. The same string lookup you just wrote could also be written as `stringResource(R.string.pepperoni)`, but you instead read it from the `topping` parameter to keep your composable dynamic.)

Control flow in a composable

Next, shift your attention to the placement text. This is a bit trickier because the `placement` input is nullable. A null value indicates that the topping is not on the pizza. In that case, the second text should not be visible and the **`Checkbox`** should not be checked.

To add this null check, you can wrap the second **`Text`** in an `if` statement. If the topping is present, this `if` statement will execute and add the label to the UI. Otherwise, the `if` statement will be skipped, and only one **`Text`** will end up onscreen.

Go ahead and make this change now. While you are at it, update the `checked` input to **`Checkbox`** to check whether the topping is present on the pizza.

Listing 1.17 `if` statements in a composable (`ToppingCell.kt`)

```
...
@Composable
fun ToppingCell(
    topping: Topping,
    placement: ToppingPlacement?,
    onClickTopping: () -> Unit
) {
    Row {
        Checkbox(
            ~~checked = true,~~
            checked = (placement != null),
            onCheckedChange = { /* TODO */ }
        )

        Column {
            Text(
                text = stringResource(topping.toppingName)
            )

            if (placement != null) {
                Text(
                    ~~text = "Whole pizza"~~
                    text = stringResource(placement.label)
                )
            }
        }
    }
}
```

Refresh the preview once more and confirm that the placement text has updated to Left half, matching the value specified in **`ToppingCellPreview`**.

To confirm that your **`ToppingCell`** is appearing as expected when the topping is not present, you will need to update your preview function to specify a `null` input for the `placement`. You could adjust your existing preview to change the `placement` argument, but it can be helpful to preview several versions of a composable at the same time.

Create a second preview function to show what **`ToppingCell`** looks like when the topping is not added to the pizza. Give your two preview functions distinct names to clarify what they are previewing.

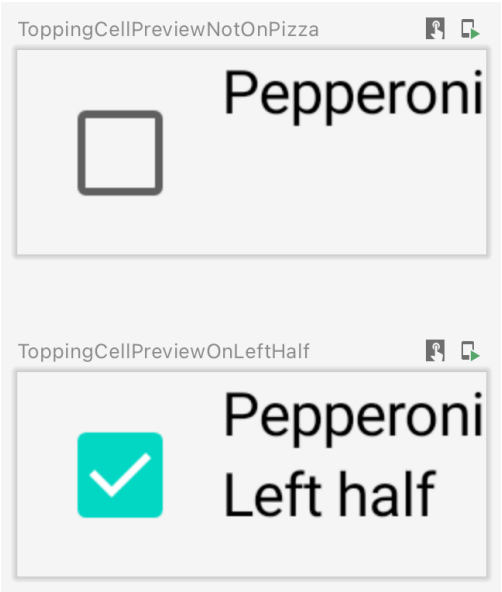
Listing 1.18 Adding another preview (`ToppingCell.kt`)

```
@Preview
@Composable
private fun ToppingCellPreviewNotOnPizza() {
    ToppingCell(
        topping = Topping.Pepperoni,
        placement = null,
        onClickTopping = {}
    )
}

@Preview
@Composable
private fun ToppingCellPreviewOnLeftHalf() {
    ToppingCell(
        topping = Topping.Pepperoni,
        placement = ToppingPlacement.Left,
        onClickTopping = {}
    )
}
...
```

Refresh the preview. You will now see two previews. In the one labeled ToppingCellPreviewNotOnPizza, only the “Pepperoni” label appears in the cell and the checkbox is unchecked (Figure 1.6).

Figure 1.6 No pepperoni, please



You have just observed the effects of control flow inside a composable. Because composables are functions, you can call them however you would like – including conditionally. Here, the **`Text`** composable was not invoked, so it is not drawn onscreen.

You can accomplish something similar with framework views by setting their visibility to `gone`. But with a framework view, the **`View`** itself would still be there, just not contributing to what is drawn onscreen. In Compose, the composable is simply not invoked. It does not exist at all.

`if` statements are not the only control flows you can use in a composable function. Composable functions are, at their core, merely Kotlin functions, so any syntax you can use in other functions can appear in a composable. `when` expressions, `for` loops, and `while` loops are all fair game in your composables, to name a few examples.

Aligning elements in a row

Take another look at the preview of your topping cell. You may have noticed that in the unselected state, it looks a bit awkward because the checkbox and the text are not vertically aligned. Worry not, there is another parameter you can specify to beautify this layout.

The **`Row`** and **`Column`** composables specify their own parameters that you can use to adjust the layout of their children. For a **`Row`**, you can use the `Alignment` parameter to adjust how its children are positioned vertically. (A **`Column`**'s `Alignment` will adjust the horizontal positioning of its children.)

By default, **`Row`**'s vertical alignment is set to `Alignment.Top`, meaning that the top of each composable will be at the top of the row. To center all items in the composable, set its alignment to `Alignment.CenterVertically`.

Listing 1.19 Specifying alignment (`ToppingCell.kt`)

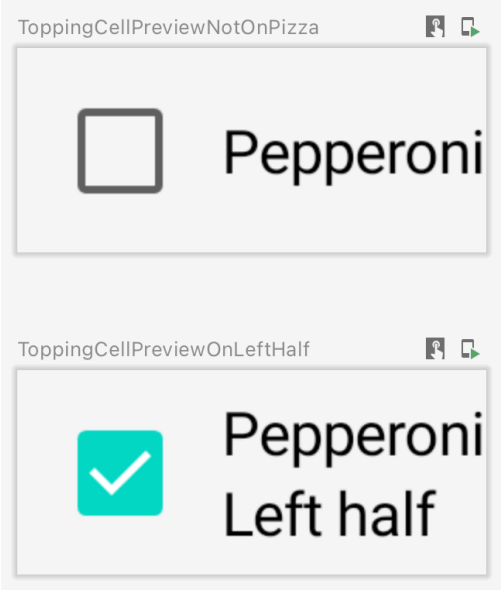
```
...
@Composable
fun ToppingCell(
    topping: Topping,
    placement: ToppingPlacement?,
    onClickTopping: () -> Unit
) {
    Row(
        verticalAlignment = Alignment.CenterVertically
    ) {
        ...
    }
}
```

Be sure to import `androidx.compose.ui.Alignment` from the options provided.

By the way: Do not confuse the `Alignment` parameter with the `Arrangement` parameter, which specifies how the extra horizontal space of a **`Row`** (or vertical space, for a **`Column`**) should be placed relative to its children.

Refresh the preview again and confirm that the topping name and checkbox are vertically aligned (Figure 1.7).

Figure 1.7 Aligning the contents of a row



Specifying text styles

Composable parameters are useful for arranging your content and setting values to display. They also serve an important role in styling your UI.

In the framework toolkit, you can set the `android:textAppearance` attribute to apply built-in styling to text elements in XML. In Compose, you can accomplish the same thing by setting the `style` parameter of the **`Text`** composable. Like the framework toolkit, Compose also has built-in text styles accessible through the **`MaterialTheme`** object. Take them for a spin now, applying the `body1` style to the name of the topping and `body2` to the placement of the topping.

When entering this code, be sure to choose the **`MaterialTheme`** object when prompted, not the **`MaterialTheme`** composable function. Their imports are the same, so no need to worry if you choose the wrong one initially – just note that Android Studio will autocomplete different code. You will see how the **`MaterialTheme`** function works in Chapter 4.

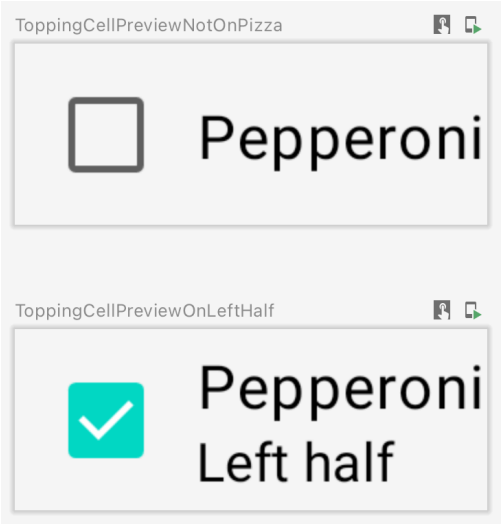
Listing 1.20 Setting text styles (`ToppingCell.kt`)

```
...
@Composable
fun ToppingCell(
    ...
) {
    Row(
        verticalAlignment = Alignment.CenterVertically
    ) {
        ...
        Column {
            Text(
                text = stringResource(topping.toppingName),
                style = MaterialTheme.typography.body1
            )

            if (placement != null) {
                Text(
                    text = stringResource(placement.label),
                    style = MaterialTheme.typography.body2
                )
            }
        }
    }
}
```

Update the previews by pressing the Build & Refresh label in the banner that says The preview is out of date or by building the project. You will see that the first line of text is larger than the second (Figure 1.8). The difference is subtle, but we promise – they are different sizes.

Figure 1.8 Text with style



The Compose Modifier

What about attributes like background color, margins, padding, and click listeners? In the framework view system, these attributes are inherited, making them accessible on every subclass of **`View`**. But composable functions do not have the luxury of inheritance.

Instead, Compose defines a separate type called **`Modifier`** where it defines general-purpose customizations that can be set on any composable. Modifiers can be chained and combined as desired. Between modifiers and a composable's function parameters, you can perform all the customizations you have used so far with framework views.

To modify a composable, pass a **`Modifier`** into the composable's `modifier` parameter. You can obtain an empty modifier to build on by referencing the **`Modifier`** object first. Then, you can chain a sequence of modifiers together to create a final **`Modifier`** object to set on the composable.

The padding modifier

Start by adding padding to the entire cell with the **`padding`** modifier. Set the vertical padding to 4dp and the horizontal padding to 16dp.

Listing 1.21 Adding padding (`ToppingCell.kt`)

```
...
@Composable
fun ToppingCell(
    topping: Topping,
    placement: ToppingPlacement?,
    onClickTopping: () -> Unit
) {
    Row(
        verticalAlignment = Alignment.CenterVertically,
        modifier = Modifier
            .padding(vertical = 4.dp, horizontal = 16.dp)
    ) {
        ...
    }
}
```

When prompted, be sure to choose the import for `androidx.compose.ui.Modifier`.

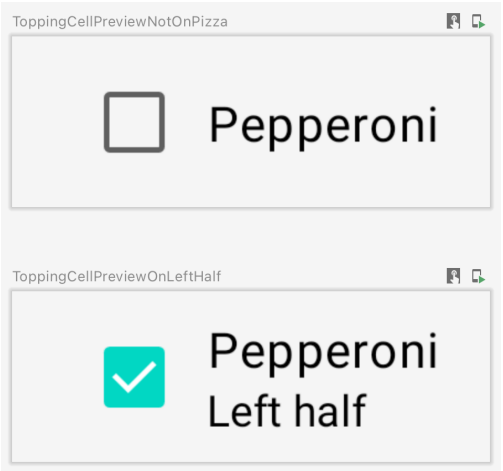
In case you are curious, there are a few other ways of specifying padding amounts. Some commonly used overloads allow you to specify the same padding amount on all sides (`Modifier.padding(all = 16.dp)`), or for all four sides independently (`Modifier.padding(top = 4.dp, bottom = 4.dp, start = 16.dp, end = 16.dp)`).

Recall from framework views that `dp` units are ideal for specifying margins and padding. Composables and modifiers specify which unit they expect a dimension to be specified in. The **`.dp`** extension property you are using returns a **`Dp`** object, which adds additional type safety to make sure you are using the correct units. There is also an **`.sp`** extension to specify an **`Sp`** value for text sizes.

Unlike framework views, these units are not interchangeable – if a Compose API needs a text size, it requests an **`Sp`** instance specifically. This also means that if you do not specify a unit, you will see a compiler error, because `Int` cannot be converted to **`Dp`** or **`Sp`** automatically.

Build and refresh your preview. You should see some additional spacing around your composable (Figure 1.9).

Figure 1.9 Room to breathe



By the way, you may also see borders around elements in your composable previews. These represent the bounds of objects in your composables, and can help you visualize how and why items are being positioned as they are. You can hover over a preview to reveal these bounds and click one of the items to navigate to the composable corresponding to that element.

Chaining modifiers and modifier ordering

If you want to further customize the appearance of your composables, you can chain modifiers. But beware: The order of modifiers matters. To see why, try adding a background to the **`ToppingCell`**. Place the **`background`** modifier after the **`padding`**.

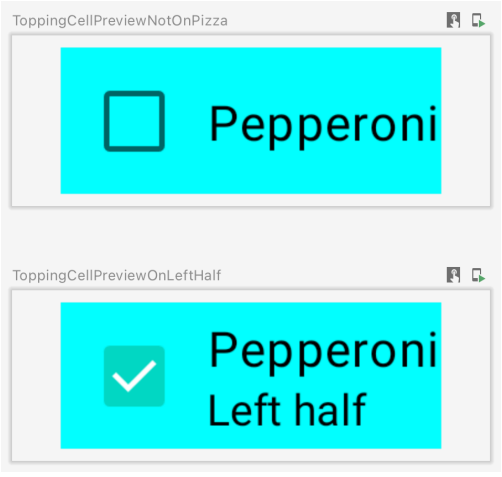
Listing 1.22 Adding a background (`ToppingCell.kt`)

```
...
@Composable
fun ToppingCell(
    topping: Topping,
    placement: ToppingPlacement?,
    onClickTopping: () -> Unit
) {
    Row(
        verticalAlignment = Alignment.CenterVertically,
        modifier = Modifier
            .padding(vertical = 4.dp, horizontal = 16.dp)
            .background(Color.Cyan)
    ) {
        ...
    }
}
```

When using the **`Color`** class, be sure to choose the import for `androidx.compose.ui.graphics`. Compose specifies its own class for colors (much like it does for **`Dp`** and **`Sp`**) and includes constants for a few colors as a convenience.

Where do you think the background will appear? Build and refresh your composable preview to see for yourself (Figure 1.10). Does this match your expectation?

Figure 1.10 Padding the background

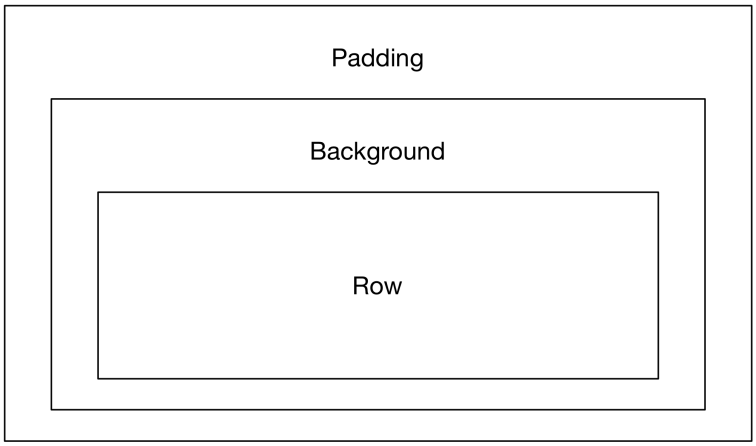


With this code, the background appeared *inside* the padding. Why?

As we said: In Compose, the order of modifiers matters. Modifiers are invoked from top to bottom. Each makes its contribution to the appearance of the composable, and then the next modifier is applied *inside* it. Once all modifiers have been applied, the content of the composable is placed inside the final modifier.

Looking at your current code, this means that the padding is added first, and then the background is added inside the padding. Finally, the **`Row`** and its contents are placed inside the background. Figure 1.11 illustrates how Compose is treating your **`Row`** and the two modifiers.

Figure 1.11 How Compose sees your modifiers



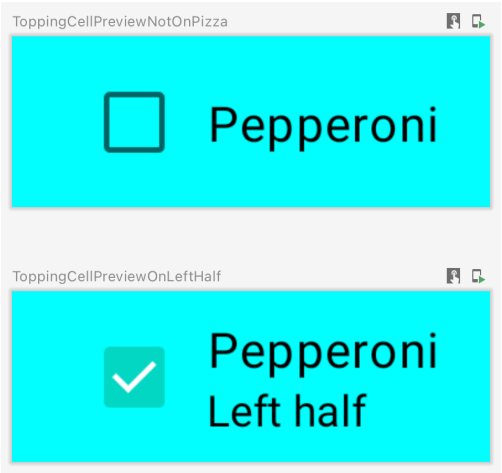
Now try moving the background modifier so that it comes before the padding modifier.

Listing 1.23 Reordering the background modifier (`ToppingCell.kt`)

```
...
@Composable
fun ToppingCell(
    topping: Topping,
    placement: ToppingPlacement?,
    onClickTopping: () -> Unit
) {
    Row(
        verticalAlignment = Alignment.CenterVertically,
        modifier = Modifier
            .background(Color.Cyan)
            .padding(vertical = 4.dp, horizontal = 16.dp)
            .background(Color.Cyan)
    ) {
        ...
    }
}
```

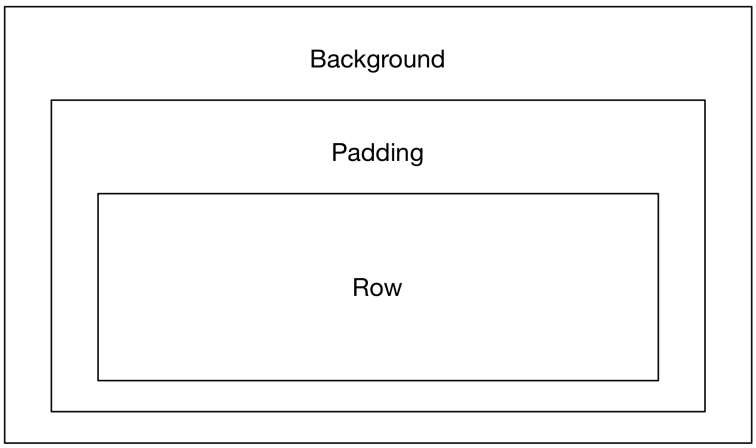
Build and refresh the preview. You will now see that the background fills the entire composable – padding included – and the rest of the content remains inside the padding (Figure 1.12).

Figure 1.12 Backgrounds with padding



The padding is placed inside the background now because the background is added first and the padding is added second (Figure 1.13).

Figure 1.13 How Compose sees your reordered modifiers



Sometimes, the order of your modifiers does not matter. For example, if you had several instances of the **`padding`** modifier to specify the padding amounts for the top, bottom, and sides of a composable, it would not matter what order you declared them in. But for many combinations of modifiers, you need to be careful about ordering.

If you ever find that one of your composables is not appearing as expected, think about the order of your modifiers and make sure they are not incorrectly affecting one another.

Remove the background, as it is a bit garish and will not be part of the final UI.

Listing 1.24 Removing the background (`ToppingCell.kt`)

```
...
@Composable
fun ToppingCell(
    topping: Topping,
    placement: ToppingPlacement?,
    onClickTopping: () -> Unit
) {
    Row(
        verticalAlignment = Alignment.CenterVertically,
        modifier = Modifier
            .background(Color.Cyan)
            .padding(vertical = 4.dp, horizontal = 16.dp)
    ) {
        ...
    }
}
```

The clickable modifier

Another crucial modifier is **`clickable`**, which is analogous to the venerable **`setOnClickListener`** method. The **`clickable`** modifier makes a composable clickable, and it accepts a lambda expression to define what to do when the view is pressed. Try it out now to make pressing the **`Row`** invoke the `onClickTopping` callback.

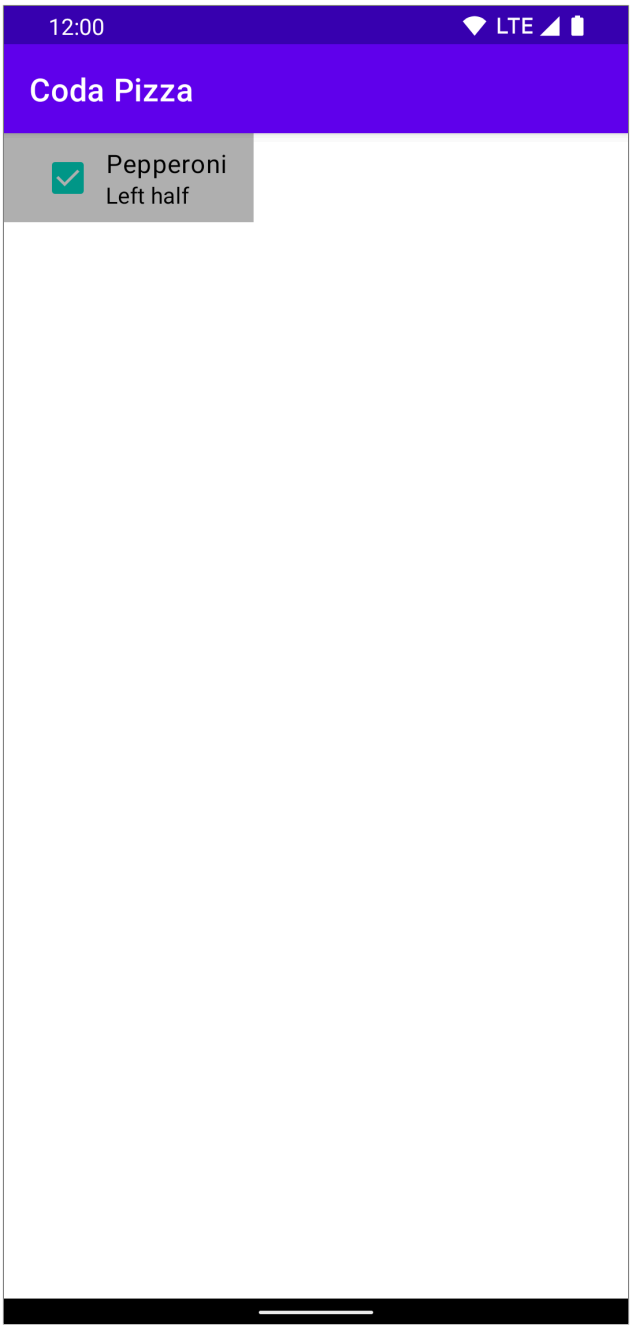
Be sure to add the **`clickable`** modifier as the *first* modifier. You want the entire composable (padding included) to be clickable. Also, besides defining click behavior, the **`clickable`** modifier darkens the background of the composable when it is pressed to indicate that it is being selected. You want this effect to extend through the padding, which is another reason to put the modifier first.

Listing 1.25 Making a composable clickable (`ToppingCell.kt`)

```
...
@Composable
fun ToppingCell(
    topping: Topping,
    placement: ToppingPlacement?,
    onClickTopping: () -> Unit
) {
    Row(
        verticalAlignment = Alignment.CenterVertically,
        modifier = Modifier
            .clickable { onClickTopping() }
            .padding(vertical = 4.dp, horizontal = 16.dp)
    ) {
        ...
    }
}
```

Run Coda Pizza in an emulator. You will see the pepperoni topping displayed in the top-left corner of your activity, matching the preview. Try clicking the topping. (Be sure to click near the text Pepperoni.) You have not specified any behavior to take place after the click, but you will see the background darken, indicating that the click was recognized (Figure 1.14).

Figure 1.14 Interacting with a clickable composable



Sizing composables

Currently, the only clickable area is near the label of the topping. If you try to click to the right of the topping cell, nothing will happen. (Try it for yourself.) This probably does not match your users' expectations – you should be able to click anywhere along the width of the screen to interact with the cell.

The reason that the clickable area does not fill the width of the screen is that your **`ToppingCell`** composable is only taking up as much space as it needs to to display its content. Effectively, its dimensions are implicitly set to wrap its content.

One way to make your composable consume all the available width is to make your **`Column`** take up all the remaining width in its container. You can do this with the **`weight`** modifier.

The **`weight`** modifier is a bit special in that it can only be used when your composable is placed inside another composable that supports weights – like **`Row`** and **`Column`**. The weight modifier behaves the same as the `layout_weight` attribute on a **`LinearLayout`**: Any extra space will be divvied up proportionally to views in the layout based on their weight. If only one of the composables specifies a weight, all the extra space will go to that composable.

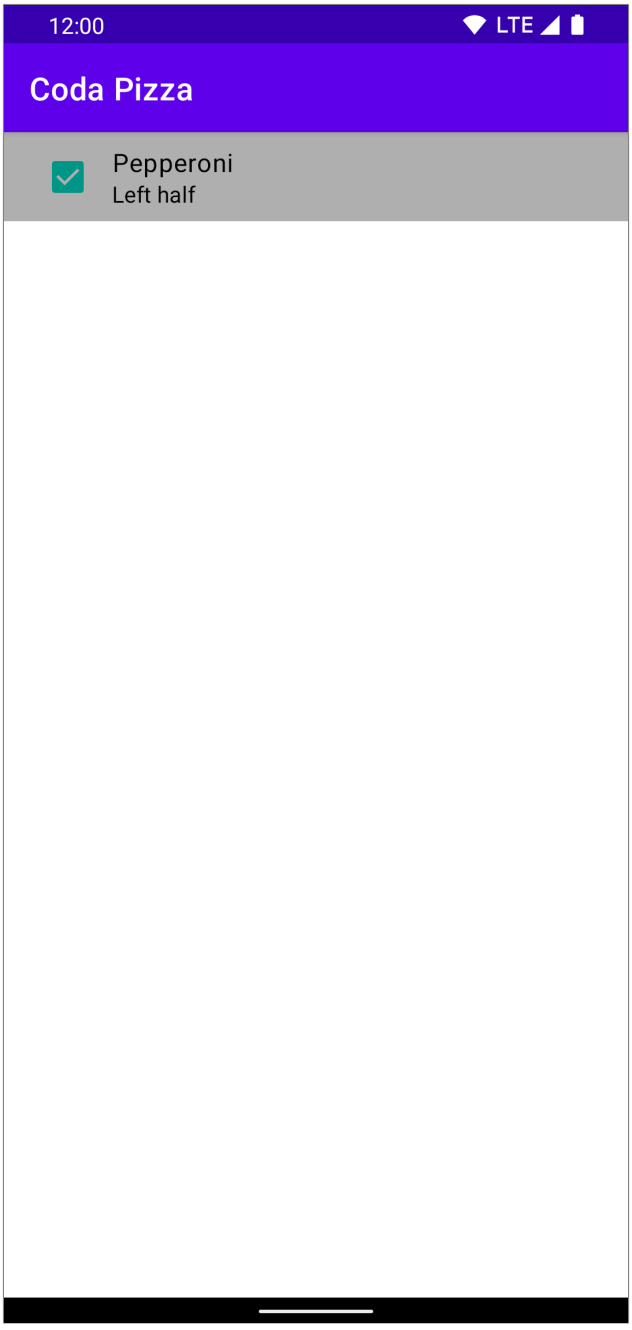
Try it out now. While you are adding a modifier to your **`Column`**, include some padding, which will give your checkbox and text some more breathing room.

Listing 1.26 Using the `weight` modifier (`ToppingCell.kt`)

```
...
@Composable
fun ToppingCell(
    topping: Topping,
    placement: ToppingPlacement?,
    onClickTopping: () -> Unit
) {
    Row(
        ...
    ) {
        ...
        Column(
            modifier = Modifier.weight(1f, fill = true)
                .padding(start = 4.dp)
        ) {
            ...
        }
    }
}
```

Rerun Coda Pizza in the emulator to see the changes. Press the right half of the screen next to the Pepperoni text. You should see that the touch indication appears for the topping cell, even though you are clicking the empty space next to it. You should also see that the touch indication fills the full width of the screen (Figure 1.15).

Figure 1.15 Interacting with a weighted clickable composable



We have just scratched the surface of what is available with the **`Modifier`** API. For example, you can more explicitly tell a composable how much space to take up with modifiers like **`wrapContentHeight`** and **`fillParentWidth`**. You can also specify dimension with modifiers like **`size`**, and you can get more creative about how to constrain a composable's size with modifiers like **`aspectRatio`** and **`sizeIn`**.

You can find a list of all the built-in modifiers at `developer.android.com/jetpack/compose/modifiers-list`. We encourage you to experiment and combine modifiers to build more complex UIs. You will likely find that modifiers are more predictable, flexible, and concise than what is available in the framework view classes.

Specifying a modifier parameter

Modifiers are a crucial part of customizing a composable, and they allow you to specify many common customizations for your UI elements. They are so important, in fact, that we recommend that *every* composable UI element that you define accept an optional **`Modifier`** input.

Even if you do not think you need to specify any modifiers on a composable, it is better to have the option readily available than to have to add it later if you change your mind. For **`ToppingCell`**, you may decide later that you want to add a background, change its padding, or set a size for the composable. In fact, later in this chapter, you will need to tell your **`ToppingCell`** how wide it should be.

Currently, the only way to change these attributes is to modify **`ToppingCell`** itself. But any changes made to the composable directly will appear everywhere you use **`ToppingCell`** in your app, which is not ideal. What if **`ToppingCell`** is used in multiple places and needs to be a different size in each place? To open the door to future customizations, you will add a **`Modifier`** parameter to **`ToppingCell`**.

To avoid requiring a **`Modifier`** instance for every single usage, give this parameter a default value of `Modifier` – the `Modifier` object that you have been building off of so far that represents an empty set of modifications. The official convention for the `modifier` parameter is to place it after your required parameters and before any other optional parameters.

To use the `modifier` parameter, pass it to your outer composable – your **`Row`**, in this case. Make these changes now (and be sure to change the capitalized `Modifier` to the lowercase `modifier`):

Listing 1.27 Allowing modifications (`ToppingCell.kt`)

```
...
@Composable
fun ToppingCell(
    topping: Topping,
    placement: ToppingPlacement?,
    modifier: Modifier = Modifier,
    onClickTopping: () -> Unit
) {
    Row(
        verticalAlignment = Alignment.CenterVertically,
        modifier = Modifier modifier
            .clickable { onClickTopping() }
            .padding(vertical = 4.dp, horizontal = 16.dp)
    ) {
        ...
    }
}
```

And that is it! We will revisit your lingering `TODO` in the next chapter when we discuss state, but for now you have finished implementing **`ToppingCell`**. It is ready to be used in your application – say, in a scrollable list.

Building Screens with Composables

You have seen a number of composables up to this point, with a range of complexity. On the simple end, you have seen atomic components like **`Row`** and **`Text`**. And you have built **`ToppingCell`**, a more complex composable that is built on top of other composables.

It is also possible to create a composable that renders the entire screen. In fact, the **`setContent`** function you called in **`MainActivity`** is a lambda that does just that. Because there are no limitations about what a single composable can do, you do not need another component like a **`Fragment`** to perform navigation. (You can, however, enlist the help of the AndroidX Navigation library, which has a Jetpack Compose flavor.)

Under Jetpack Compose, composables of all sizes are *the* building blocks of your UIs.

Right now, Coda Pizza only displays a single topping. Eventually, you will display several toppings, but you will not want to be constantly opening your activity's code to modify its UI. Instead, you can extract its content into a separate file. The result of this change will be that your **`MainActivity`** will call a single content composable, leaving the activity itself sparse with code.

To get started, define a new file called `PizzaBuilderScreen.kt` in the `com.bignerdranch.android.codapizza.ui` package. Define a new composable function called **`PizzaBuilderScreen`** in this file. **`PizzaBuilderScreen`** will be a composable that draws *all* of the main content inside the activity. If your code had navigation or other logic, you could place it in this composable.

Remember to give your new composable a **`Modifier`** parameter. Also, add the `@Preview` annotation to add a quick preview to the function. (You will not add any arguments to this function, so a separate preview function is not necessary.)

Listing 1.28 Defining a screen (`PizzaBuilderScreen.kt`)

```
@Preview
@Composable
fun PizzaBuilderScreen(
    modifier: Modifier = Modifier
) {

}
```

Here, we are using the convention of ending the composable's name with "Screen" to indicate that it fills the entire viewport of the window and represents a distinct portion of the app's UI. Apps can have many screens if they need to display several different UIs and navigate between them.

Your **`PizzaBuilderScreen`** will display several elements by the time Coda Pizza is complete: the app bar, the pizza customization preview, the list of toppings, and the PLACE ORDER button. For now, you will focus on two of these elements: the list of toppings and the PLACE ORDER button.

Before adding any new composables, take a brief detour to your `strings.xml` file to add the string resource you will use for the PLACE ORDER button.

Listing 1.29 The "place order" label (`strings.xml`)

```
<resources>
    <string name="app_name">Coda Pizza</string>

    <string name="place_order_button">Place Order</string>

    <string name="placement_none">None</string>
    ...
</resources>
```

Next, create two new private composables inside `PizzaBuilderScreen.kt`: one for the toppings list and one for the PLACE ORDER button. For the toppings list composable, call **`ToppingCell`** once for the time being. You will implement the scrolling list behavior in the next section. For the PLACE ORDER button, use the **`Button`** composable.

The **`Button`** composable takes in two required inputs: an `onClick` callback and a set of composable children to place inside the button. If you wanted to, you could add an icon or any other composables to spice up your button, but for now you will stick to just the **`Text`** composable.

It is conventional for button labels on Android to appear in all caps. This happens by default when using the framework **`Button`** view, but does not happen automatically in Compose. To respect this convention, manually capitalize your string using the **`toUpperCase`** function.

Listing 1.30 Defining content to put onscreen (`PizzaBuilderScreen.kt`)

```
@Preview
@Composable
fun PizzaBuilderScreen(
    modifier: Modifier = Modifier
) {

}

@Composable
private fun ToppingsList(
    modifier: Modifier = Modifier
) {
    ToppingCell(
        topping = Topping.Pepperoni,
        placement = ToppingPlacement.Left,
        onClickTopping = {},
        modifier = modifier
    )
}

@Composable
private fun OrderButton(
    modifier: Modifier = Modifier
) {
    Button(
        modifier = modifier,
        onClick = {
            // TODO
        }
    ) {
        Text(
            text = stringResource(R.string.place_order_button)
                .toUpperCase(Locale.current)
        )
    }
}
```

If prompted, be sure to choose the import for `androidx.compose.material.Button` as well as the imports for `androidx.compose.ui.text.toUpperCase` and `androidx.compose.ui.text.intl.Locale`.

Now you can place these composables in your **`PizzaBuilderScreen`** to set their position onscreen. Add a **`Column`** to **`PizzaBuilderScreen`** to place the **`ToppingsList`** at the top of the screen and the **`OrderButton`** at the bottom of the screen. The **`ToppingsList`** should fill all of the available height, so set its weight to 1 via its modifier. Also, make the **`OrderButton`** take up the full width of the screen, with 16dp of padding around it.

Listing 1.31 Placing content in the **PizzaBuilderScreen** (PizzaBuilderScreen.kt)

```
@Preview
@Composable
fun PizzaBuilderScreen(
    modifier: Modifier = Modifier
) {
    Column(
        modifier = modifier
    ) {
        ToppingsList(
            modifier = Modifier
                .fillMaxWidth()
                .weight(1f, fill = true)
        )

        OrderButton(
            modifier = Modifier
                .fillMaxWidth()
                .padding(16.dp)
        )
    }
}
...
```

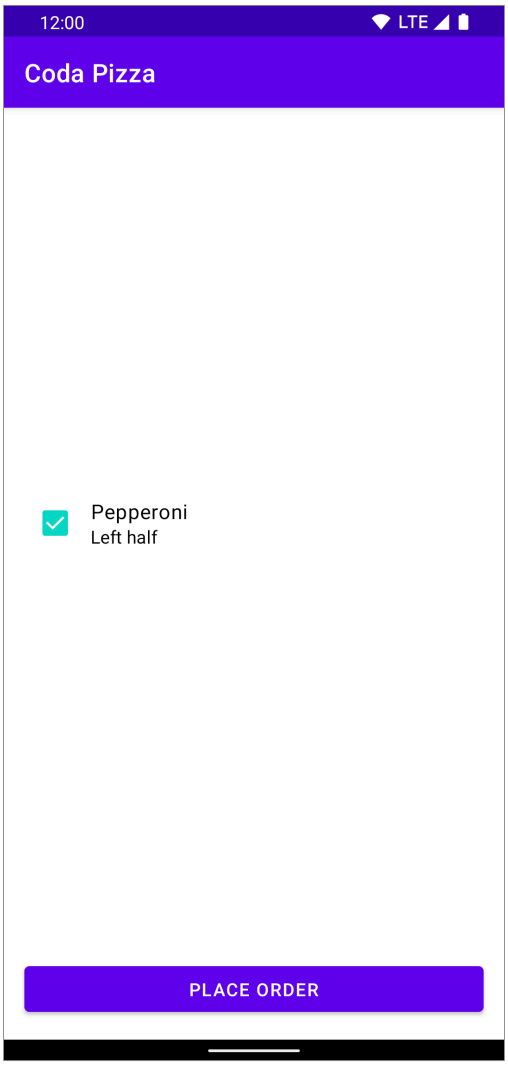
With **PizzaBuilderScreen** ready, you can now update your **MainActivity**'s **setContent** block to delegate to this function instead of creating the UI itself.

Listing 1.32 Using the **PizzaBuilderScreen** composable (MainActivity.kt)

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            ToppingCell(
                topping = Topping.Pepperoni,
                placement = ToppingPlacement.Left
            )
            PizzaBuilderScreen()
        }
    }
}
```

Run Coda Pizza. The app should look like Figure 1.16, with the PLACE ORDER at the bottom of the screen and the topping information in the center of the screen. The topping cell is centered on the screen because you have set it to fill the height of the screen and it is arranging its content to be centered within its bounds. Although the centering behavior is a bit awkward, your **ToppingsList** composable is being correctly placed to take up the leftover height of the screen for when you give it a final implementation.

Figure 1.16 **`PizzaBuilderScreen`** in action



Scrollable Lists with LazyColumn

The last step in the major scaffolding for Coda Pizza is to turn your **`ToppingsList`** composable into, well, a *list*. Right now it displays a single topping, but you want it to show all the available toppings in Coda Pizza's menu. Previously, you would use **`RecyclerView`** to achieve this goal, which generally involves a ritual of creating an adapter, view holder, and layout manager for even the most basic of lists.

In Compose, scrollable lists of items are created using **`LazyColumn`** (or **`LazyRow`**, if you want to scroll horizontally instead of vertically). At a very high level, **`LazyColumn`** behaves like **`RecyclerView`**: It only does work for the composables that will be drawn onscreen, and it can scroll through enormous lists with high efficiency. Unlike **`RecyclerView`**, a **`LazyColumn`** can be created with just a few lines of code.

Because you do not have the overhead of view objects or the ability to store references to composables, there is no **`ViewHolder`** class to create. And because you have the power of Compose's layout engine, an adapter is not necessary. The adapter's role is to turn indices into views and to recycle those views with new information, but Compose can spin up and tear down composables so efficiently on its own that **`LazyColumn`** only needs to know what to display at a given position. It will take care of the rest.

LazyColumn has one required parameter: a lambda expression to specify the contents of the list. Beware, though – unlike most of the other lambdas you have seen in this chapter, the lambda you pass to **LazyColumn** is *not* a `@Composable` function, meaning that you cannot directly add composable content to the list.

Instead, you add elements to the list by calling functions like **item**, **items**, or **itemsIndexed** inside **LazyColumn**'s lambda. Each of these builder functions accepts its own lambda to create a composable that defines what will be drawn for its position or positions in the list. You can add as many or as few items as you want, and you can easily combine datasets if desired.

For Coda Pizza, you will use a **LazyColumn** to display all the values of the **Topping** enum. Inside the **LazyColumn**, use the **items** function and pass a list of the **Topping** values to specify that they should appear in the list. In the lambda for the **items** builder, take the `topping` passed to the lambda and use it to create a row for that topping by calling your **ToppingCell**.

Make this change in `PizzaBuilderScreen.kt` now, deleting the **ToppingCell** you added as a placeholder.

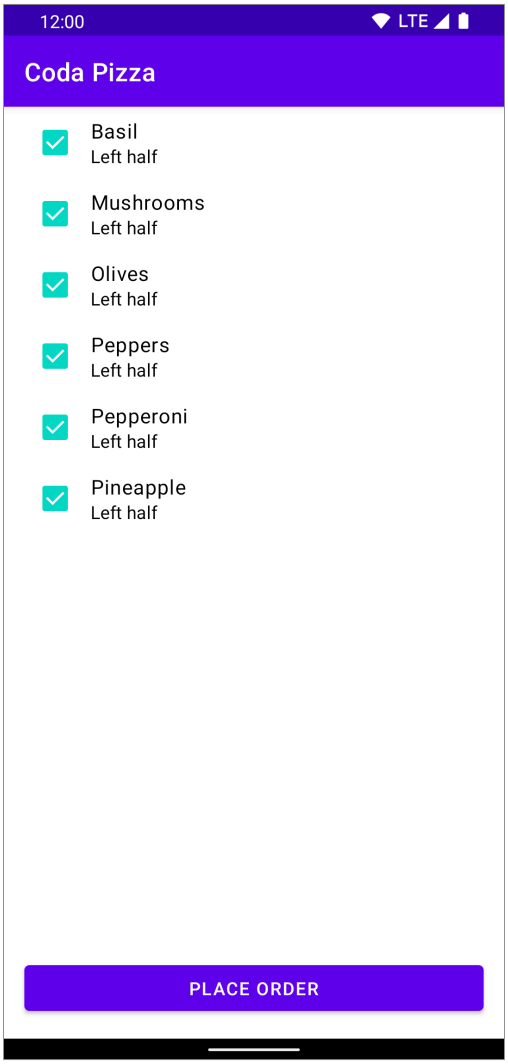
Listing 1.33 Using **LazyColumn** to show the list of toppings (`PizzaBuilderScreen.kt`)

```
...
@Composable
private fun ToppingsList(
    modifier: Modifier = Modifier
) {
    ToppingCell(
        topping = Topping.Pepperoni,
        placement = ToppingPlacement.Left,
        onClickTopping = {},
        modifier = modifier
    )
    LazyColumn(
        modifier = modifier
    ) {
        items(Topping.values()) { topping ->
            ToppingCell(
                topping = topping,
                placement = ToppingPlacement.Left,
                onClickTopping = {
                    // TODO
                }
            )
        }
    }
}
...
```

When entering this code, be sure to import the **items** function, if Android Studio does not do so automatically. The import for this function is `androidx.compose.foundation.lazy.items`. You may see an error about a type mismatch if you do not have this import statement.

Run Coda Pizza. You will see all the toppings listed in the order that they are declared in your **Topping** enum. They will all be set to appear on the left half of the pizza (Figure 1.17).

Figure 1.17 **`LazyColumn`** in action



The list still needs a bit of work – it is not possible to change the placement of a topping, for example – but spend a moment to marvel at your code. These few lines of code are all it takes to implement the **`ToppingsList`** composable, and your code can easily be adapted to add more content to the list or to change the appearance of its contents. Achieving this level of conciseness with **`RecyclerView`** is simply not possible. This is a testament to how Jetpack Compose makes it easier to build Android apps.

You have accomplished quite a bit in this chapter, from making your way through the fundamentals of Compose's layout system through giving Coda Pizza a solid foundation to build on as you continue to explore Jetpack Compose. In the next chapter, you will make your UI elements interactive, update your composables to react to user input, and explore how Jetpack Compose handles application state.

For the More Curious: Live Literals

In addition to previews, Jetpack Compose has several tricks up its sleeve to let you quickly iterate on UI designs. One of these is a feature called *live literals*. When live literals are enabled and you run your Compose app through Android Studio, any changes you make to hardcoded values ("literals") in your Compose UI will automatically be pushed to the app as it is running. Your UI will refresh with new values as they are typed, letting you preview changes instantly, without recompiling.

Live literals only work for simple hardcoded values like `Int`, `Boolean`, and `String`. They also must be enabled in the IDE. To check if you have live literals enabled, open Android Studio's preferences with Android Studio → Preferences.... The option for live literals is under the Editor section in the left half of the preferences window, on a page called Live Edit of literals. Navigate to this page and ensure Enable Live Edit of literals is checked to turn on live literals. Then re-launch your app to enable the feature.

Try out live literals by changing your padding values while Coda Pizza is running. Use larger and smaller values and watch as your content instantly shifts to match the new measurements you define.

Any other code you modify – like adding or removing modifiers entirely – will not update your UI until you rebuild and run Coda Pizza again; only your literals get updated automatically. Despite this limitation, the instantaneous nature of these updates makes live literals a great tool for putting the finishing touches on your UI's appearance.

When you are finished experimenting with live literals, make sure to revert any changes you made to Coda Pizza before proceeding to the next chapter.

2

UI State in Jetpack Compose

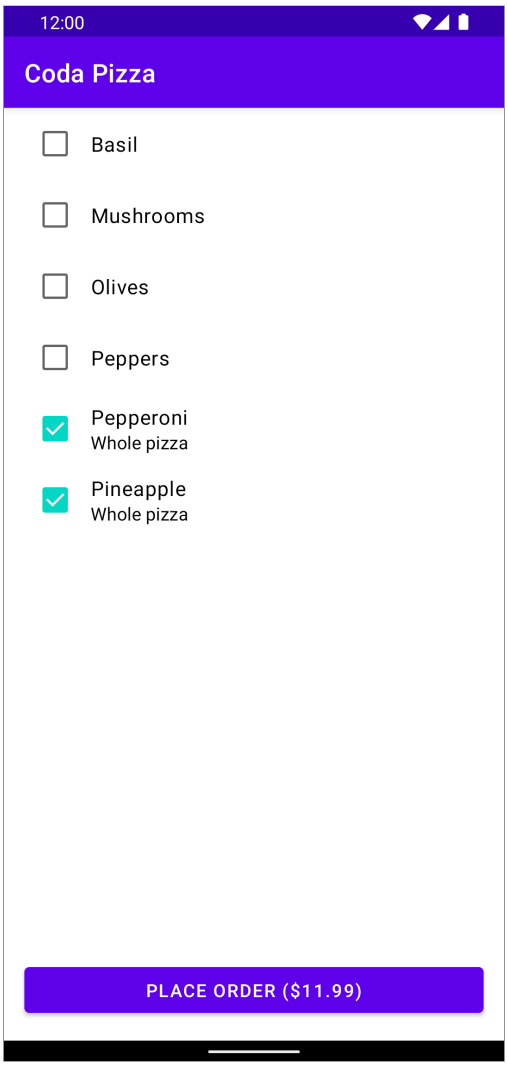
Coda Pizza is off to a good start. In the last chapter, you built out a scrollable list, created a composable to act as the cell for toppings, and set up the screen's layout, including an order button at the bottom of the page. But your app is missing something: Users cannot interact with it (other than scrolling the list of toppings). If users press a checkbox, it does not change from checked to unchecked or vice versa.

This is not what you are used to with framework views. With framework views, as soon as you added an interactive UI element to your UI, such as a checkbox, it would respond to click events by toggling its state, with no additional code needed.

In this chapter, you will learn why your current code does not respond to user inputs, and you will see how to incorporate UI state into your composables. By the end of this chapter, Coda Pizza will let users place and remove toppings on their pizzas. Pressing either the checkbox or the topping's cell will toggle whether the topping is on the pizza. You will also update the PLACE ORDER button to include the price for the pizza based on the number of toppings.

When you finish with this chapter, Coda Pizza will look like Figure 2.1.

Figure 2.1 Coda Pizza, now with state



In the next chapter, you will go a step further and display a dialog to ask the user where they want their topping to be placed.

Philosophies of State

Let's start by discussing how framework views approach state. The apps you have probably worked on in the past actually had two representations of UI state. The first was built by you, generally defined with data classes and managed by **`ViewModel`**s. This kind of state is often called *application state* or *app state*, because this state controls the behavior of your app as a whole and is how your app sees the world. Your Compose app will also have app state, and it will be defined similarly.

The second representation of UI state in framework views lived inside the **`View`**s themselves. Think about the framework **`CheckBox`** and **`EditText`** views. When the user presses a **`CheckBox`** or enters new text in an **`EditText`**, that input immediately updates the UI, whether you want it to or not.

The fragmentation of state with framework views means that one of the important responsibilities of your UI code so far has been keeping your app state and **`View`** state in sync. If you are using a unidirectional data flow architecture, for example, your app will take in events, update its model, generate new UI values, and then update

the UI. But when views have their own state, it is possible for events to short-circuit this flow and update the UI without your consent – possibly incorrectly.

So, when using framework views, you have to ensure that your UI updates as expected, which sometimes means immediately undoing state changes the view makes on itself. This back and forth shuffling of data is hard to manage and is the cause of many UI bugs in Android apps.

Compose is different. In an app created with Compose, UI state is stored in only one place. In fact, Compose does not store any UI state of its own. Instead, you are in complete control of your UI state. Take the **`Checkbox`** composable you are using in Coda Pizza:

```
Checkbox(
    checked = true,
    onCheckedChange = { /* TODO */ }
)
```

If this were a framework view, you would likely expect `checked` to set the initial checked state of the checkbox and `onCheckedChange` to be called whenever it changes between states. But in Compose, the semantics are a little different.

The `checked` parameter defines whether the **`Checkbox`** is *currently* checked. You hardcoded this value to `true`, so the checkbox is permanently in the checked state – unless you change your code. As you might guess, this is not really how developers design their checkbox composables. Later in this chapter, you will instead set this parameter to a variable, allowing your composable to update when the variable providing its value is reassigned.

Meanwhile, `onCheckedChange` is called whenever the **`Checkbox`** *requests* that its checked state change. In practice, this means that `onCheckedChange` is called each time the user interacts with the **`Checkbox`**, indicating that the user wants to toggle the box's checked state.

Generally, this lambda is defined so that it updates the input for `checked` with the new state – but it does not have to. Which is good, because in the finished Coda Pizza, you want the user to pick where the topping goes – you do not want to immediately toggle the state of the **`Checkbox`**.

All of this explains why your checkboxes are currently ignoring user input: Right now, you do nothing when the component requests that its state change, so the checked state of the boxes never changes.

By the way, this philosophy of state is why Compose is referred to as a *declarative* UI toolkit. You declare how you want your UI to appear; Compose does the heavy lifting of not only setting your UI up the first time but also keeping it up to date as its state changes. Your composables will have a live connection to any state they reference, and changes to their state will update any consumers of that state object with no extra effort on your part.

Defining Your UI State

The first step in adding state to a Compose application is to define the models you will use to store it. For Coda Pizza, you need a place to hold the state of which toppings are selected. Create a new file called `Pizza.kt` in the `com.bignerdranch.android.codapizza.model` package. In this file, create a new data class called **`Pizza`**.

Give your data class one property: the toppings on the pizza, a **`Map<Topping, ToppingPlacement>`**. If a topping is present on the pizza, it will be added to this map as a key. The value will be the topping's position on the pizza. If a topping is not on the pizza, it will not have an entry in this map.

Listing 2.1 The **`Pizza`** class (`Pizza.kt`)

```
data class Pizza(
    val toppings: Map<Topping, ToppingPlacement> = emptyMap()
)
```

Representing your pizza this way makes it easy to determine whether and where a topping is on the pizza. It also prevents you from making unsupported combinations, like adding two instances of pepperoni to the entire pizza. (Coda Pizza does not have an option to change the quantity of a topping, only its placement.)

Updating UIs with MutableState

With your **`Pizza`** model ready for use, you can begin incorporating UI state into your application. To get your bearings with how state behaves in Jetpack Compose, define a file-level property in `PizzaBuilderScreen.kt` to track selected toppings. Assign the `pizza` property's initial value to include some toppings by default: pepperoni and pineapple on the whole pizza. After defining your pizza state, also update your **`ToppingsList`** composable to determine the `placement` of a topping based on the `pizza` property.

Listing 2.2 Declaring state (`PizzaBuilderScreen.kt`)

```
private var pizza =
    Pizza(
        toppings = mapOf(
            Topping.Pepperoni to ToppingPlacement.All,
            Topping.Pineapple to ToppingPlacement.All
        )
    )
...
@Composable
private fun ToppingsList(
    modifier: Modifier = Modifier
) {
    LazyColumn(
        modifier = modifier
    ) {
        items(Topping.values()) { topping ->
            ToppingCell(
                topping = topping,
                placement = ToppingPlacement.Left,
                placement = pizza.toppings[topping],
                onClickTopping = {
                    // TODO
                }
            )
        }
    }
}
...
```

Run Coda Pizza. You will see the list of toppings, as before, but only the rows for pepperoni and pineapple will be checked, matching the values you specified in the `pizza` property. If you click a topping, it still will not change anything onscreen, because you have not reassigned the state.

In a moment, you will update the `onClickTopping` lambda in your **`ToppingsList`** composable so that topping selections can be changed. First, add a function to the **`Pizza`** class to make it easier to add and remove toppings.

Define a function called **`withTopping`** that returns a copy of the pizza with a given topping. The function should also accept a **`ToppingPlacement`** to indicate where the topping is being placed. If a `null` value is sent for the placement, the new pizza should have that topping removed. Use the **`copy`** function to make the updated **`Pizza`** instance:

Listing 2.3 Easy pizza changes (`Pizza.kt`)

```
data class Pizza(
    val toppings: Map<Topping, ToppingPlacement> = emptyMap()
) {
    fun withTopping(topping: Topping, placement: ToppingPlacement?): Pizza {
        return copy(
            toppings = if (placement == null) {
                toppings - topping
            } else {
                toppings + (topping to placement)
            }
        )
    }
}
```

Why create copies of your **`Pizza`** objects instead of making the `toppings` parameter a `var` or a **`MutableMap`**? As you will see shortly, Compose is aware of changes to your UI state – but only when your state object itself is reassigned. If a property of a UI state object changes, the UI will not update as expected. This can cause problems in your application. For this reason, we recommend making UI state classes only contain `val` properties.

With this helper function in place, you can implement your `onClickTopping` lambda. Do so now, setting the `pizza` property to an updated pizza. For now, keep the implementation simple: If the topping is on the whole pizza, your lambda should remove it; otherwise, it should be added to the whole pizza. To watch as your `pizza` is modified, also add a custom setter to the `pizza` property to print a log message each time your pizza's state is reassigned.

Listing 2.4 Updating UI state (`PizzaBuilderScreen.kt`)

```
private var pizza =
    Pizza(
        toppings = mapOf(
            Topping.Pepperoni to ToppingPlacement.All,
            Topping.Pineapple to ToppingPlacement.Left
        )
    )
    set(value) {
        Log.d("PizzaBuilderScreen", "Reassigned pizza to $value")
        field = value
    }
...
@Composable
private fun ToppingsList(
    modifier: Modifier = Modifier
) {
    LazyColumn(
        modifier = modifier
    ) {
        items(Topping.values()) { topping ->
            ToppingCell(
                topping = topping,
                placement = pizza.toppings[topping],
                onClickTopping = {
                    // TODO
                    val isOnPizza = pizza.toppings[topping] != null
                    pizza = pizza.withTopping(
                        topping = topping,
                        placement = if (isOnPizza) {
                            null
                        } else {
                            ToppingPlacement.All
                        }
                    )
                }
            )
        }
    }
}
...
```

Run Coda Pizza and press the cell labeled Pineapple to remove pineapple from your pizza. (You will need to press the cell itself – not the checkbox.) In Logcat, you should see the message reporting that the state has changed along with the new value of your `pizza` property:

```
    D/PizzaBuilderScreen: Reassigned pizza to Pizza(toppings={Pepperoni=All})
```

But despite the `pizza` state having changed, as the log shows, your UI did not update. It still shows pineapple as being on the whole pizza. This is in line with what you might expect based on your experience with framework UIs: Updating your application state without telling the UI it needs to update results in a stale UI. However, as you will see shortly, Jetpack Compose can be made aware of this reassignment so that it updates your UI automatically.

Before you wire up Compose to update your UIs automatically, do another quick test to confirm that your application state has correctly changed. With Coda Pizza still running, rotate your emulator or device to trigger a configuration change. This will cause your activity to be re-created, as you have seen before, which in turn means that your Compose UI will get rebuilt. Because your state is defined as a global variable, it will survive this configuration change and be used when your UI is redrawn. You will see that after rotating, the list of toppings updates to match your `pizza` state – no more pineapple.

Currently, your composable functions only know how to set up their initial state. The variables that define your UI state can change all they want, but your UI does not yet have a way to know about these changes, so your composables never update with fresh data. To fix this issue, you need a mechanism to tell your composable functions when and how to update.

In your time with the framework UI toolkit, you have been responsible for figuring out when changes to your application state should cause updates in your UI. But Compose updates your UI state by observing your application state. To allow this observation to happen, you need a **`MutableState`**.

`MutableState` (like its read-only sibling, **`State`**) is a wrapper object that keeps track of a single value. Whenever the value inside one of these state objects is reassigned, Compose is immediately notified of the change. Every composable that accesses the state object will then automatically update with the new value held in the state object.

Because your `pizza` state is not tracked with a state object, Compose cannot do anything in response to its value changing. Fix this by storing your pizza inside a **`MutableState`** instance. To create a **`MutableState`** object, use the **`mutableStateOf`** function, which requires an initial value. Pass an empty **`Pizza`** for this parameter to ensure that users start customizing their pizza from a clean slate. Also, remove the custom setter you added, as you no longer need the log message.

You can make this change without altering any of your usages of `pizza` by delegating your property to the mutable state value with the `by` keyword you have seen before. Delegation using `by` makes the property look like a normal property syntactically – but reads and writes will go through the **`MutableState`** so that Compose can keep track of state changes.

Type slowly when entering the delegation syntax. There are two functions you need to import to allow this syntax, which Android Studio can sometimes be finicky about. The full import statements for these two functions are shown in Listing 2.5.

Listing 2.5 Storing values in a **`MutableState`** (`PizzaBuilderScreen.kt`)

```
...
import androidx.compose.runtime.getValue
import androidx.compose.runtime.setValue

~~private var pizza =~~
~~    Pizza(~~
~~        toppings = mapOf(~~
~~            Topping.Pepperoni to ToppingPlacement.All,~~
~~            Topping.Pineapple to ToppingPlacement.Left~~
~~        )~~
~~    )~~
~~    set(value) {~~
~~        Log.d("PizzaBuilderScreen", "Reassigned pizza to $value")~~
~~        field = value~~
~~    }~~

private var pizza by mutableStateOf(Pizza())
...
```

Run Coda Pizza. Initially, all your toppings will be deselected, matching the empty **`Pizza`** state you use when initializing your application. Try clicking the cell for any of your toppings (again – click the cell itself, not the checkbox).

You will see that the topping you click is toggled: Its checkbox will become ticked and the label Whole pizza will appear under the topping name. Click it again, and the check mark and placement text will disappear.

Now click a checkbox. You will see that it registers the click with a visual touch indication (a dark circle), but it will not change from checked to unchecked or vice versa. Time to fix that.

The click behavior for the checkbox will be the same as the behavior for the cell itself: It will toggle the presence of the topping on the pizza. In Chapter 3, when you add a dialog to ask where on the pizza the topping should be placed, the two will still behave identically – both will show the dialog.

`ToppingCell` already has everything it needs to implement this behavior. Implement the `onCheckedChange` lambda for its **`Checkbox`** by calling the same `onClickTopping` lambda you used for your **`clickable`** modifier.

Listing 2.6 Implementing the **`Checkbox`** (`ToppingCell.kt`)

```
...
@Composable
fun ToppingCell(
    topping: Topping,
    placement: ToppingPlacement?,
    onClickTopping: () -> Unit,
    modifier: Modifier = Modifier
) {
    Row(
        verticalAlignment = Alignment.CenterVertically,
        modifier = modifier
            .clickable { onClickTopping() }
            .padding(vertical = 4.dp, horizontal = 16.dp)
    ) {
        Checkbox(
            checked = (placement != null),
            ~~onCheckedChange = { /* TODO */ }~~
            onCheckedChange = { onClickTopping() }
        )
        ...
    }
}
```

Run Coda Pizza again and click a checkbox. This time, you will notice that the checkbox toggles the presence of the topping, just like the cell does. And no matter where you click, the checkbox or the topping cell, both elements of the UI update at once. There is no intermediate step where you need to remember to update the placement text when the checkbox changes state, and the two will never be out of sync.

Recomposition

Compared to your work in the framework UI toolkit to keep your app state and UI state in sync, Compose's automatic UI updates could seem a bit magical. Allow us to dispel some of the magic.

To see how Compose updates your UI firsthand, add a log statement to your **`ToppingCell`** composable to print a message each time the function is invoked.

Listing 2.7 Pulling back the curtain (`ToppingCell.kt`)

```
...
@Composable
fun ToppingCell(
    topping: Topping,
    placement: ToppingPlacement?,
    onClickTopping: () -> Unit,
    modifier: Modifier = Modifier
) {
    Log.d("ToppingCell", "Called ToppingCell for $topping")
    Row(
        ...
    ) {
        ...
    }
}
```

Run Coda Pizza and open Logcat. Right after Coda Pizza starts, you will see that your new log message has printed several times – once for each topping that the **`LazyColumn`** placed onscreen. (You might see these logs print again as your UI settles. That is OK. The important thing is that **`ToppingCell`** gets called for each of your toppings when the list is rendered.)

```
D/ToppingCell: Called ToppingCell for Basil
D/ToppingCell: Called ToppingCell for Mushroom
D/ToppingCell: Called ToppingCell for Olive
D/ToppingCell: Called ToppingCell for Peppers
D/ToppingCell: Called ToppingCell for Pepperoni
D/ToppingCell: Called ToppingCell for Pineapple
```

Toggle the pepperoni topping by clicking its checkbox or name and take a close look at Logcat:

```
D/ToppingCell: Called ToppingCell for Basil
D/ToppingCell: Called ToppingCell for Mushroom
D/ToppingCell: Called ToppingCell for Olive
D/ToppingCell: Called ToppingCell for Peppers
D/ToppingCell: Called ToppingCell for Pepperoni
D/ToppingCell: Called ToppingCell for Pineapple
D/ToppingCell: Called ToppingCell for Pepperoni
```

You have just witnessed *recomposition* in action. Recomposition is the technique that Compose uses to update your UI when your state changes.

Compose keeps track of which composables are using which state. When a composable's state or any of its inputs change, Compose needs to update the composable for its new state. It does this by recomposing the composable.

When a composable is recomposed, the Compose runtime invokes the function again with its new inputs. The function is executed from the beginning, and whatever UI is created by this recomposition will replace whatever the composable had shown previously. Every expression in a function being recomposed gets called again – including your log expressions, in this case.

Here, Compose knows that the **`ToppingCell`** for pepperoni uses the `pizza` state to set both its checkbox and its topping placement label. Clicking the checkbox or cell modifies your `pizza` property with a new value that has an updated toppings map. Compose sees this reassignment, determines that this change affects the **`ToppingCell`** composable for pepperoni, and invokes the function again.

Compose has many tricks up its sleeve to avoid unnecessary work when your UI is recomposed. Here, only the inputs to the **`ToppingCell`** for pepperoni changed, so it will be the only function that gets recomposed. None of the other **`ToppingCell`**s were recomposed, nor were your **`PizzaBuilderScreen`** and **`OrderButton`** composables, so they are not called again.

Because your composables can be invoked at any time, you need to be careful about what you do inside a composable. A composable should be a *pure* function, meaning that it should have no *side effects* during its composition. A side effect is any operation that causes a change somewhere outside the function in question. For example, writing a value to a database or to a variable defined outside of the function itself would be side effects, because these operations impact the behavior of other parts of your application.

Side effects are dangerous in composables. Because you never know when your composable will be recomposed, you cannot control when or how many times an operation happens. If you are not careful, you could easily get into a situation where composition triggers recomposition – possibly in an endless cycle.

It is OK – and expected – to have side effects in a callback, like a click listener in response to user input. But a side effect should never appear inside the composition itself.

(Having said that, Compose does offer mechanisms to safely host side effects inside a composable. We will cover this in more detail in Chapter 5.)

Recomposition is a crucial part of how Compose works: Any time a Compose UI changes, there is a corresponding recomposition.

Recomposition can also happen without changing the UI. If any of the inputs to your composables change, they will always recompose – even if the UI they present is not affected. In this case, recomposition will be imperceptible to users.

Now that you have uncovered some of Compose's magic, you can remove the log that prints when **`ToppingCell`** is called.

Listing 2.8 Ending the magic show (`ToppingCell.kt`)

```
...
@Composable
fun ToppingCell(
    topping: Topping,
    placement: ToppingPlacement?,
    onClickTopping: () -> Unit,
    modifier: Modifier = Modifier
) {
    ~~Log.d("ToppingCell", "Called ToppingCell for $topping")~~
    Row(
        ...
    ) {
        ...
    }
}
```

remember

Currently, the `pizza` state is defined as a global variable. This is not ideal, since global variables in general can become tricky to manage and maintain. It would be better if the state were owned by the composable itself. Because composables are nothing more than functions, the only place to encapsulate state is inside the function itself.

This presents an issue: Because Compose will call your entire composable function from the ground up every time it recomposes your UI, any local variables you declare inside a composable function will be lost between compositions. If you tried to store state inside a composable, it would reset each time the composable is invoked – which is not an ideal mechanism for storing UI state.

To address this issue, you will use a function called **`remember`**. **`remember`** takes in a lambda expression as its argument. On the first composition, the lambda is invoked to generate the remembered value, which the function then returns.

On subsequent compositions, **`remember`** immediately returns the value from the previous composition. This allows you to persist information across compositions, which is imperative for any information that cannot be derived from the composable's inputs.

Composables can have any number of remembered values. Also, Compose keeps track of which instances of a composable have remembered which values. If you have several instances of the same composable, they will each remember their own values.

`remember` is frequently used with **`mutableStateOf`** to define state for a composable. In fact, Android Studio will flag your code with an error if you attempt to call **`mutableStateOf`** inside a composable without wrapping it in a **`remember`** block.

Armed with this knowledge, you are ready to move your `pizza` property. Right now, the only composable that needs your pizza state is the **`ToppingsList`**, making it an ideal candidate to store this state.

This will not be the final place that your pizza state is stored – in fact, you will find yourself needing to store this state in a different composable very soon. However, it is fairly common as your application evolves to move where

your state is defined – so much so that it is a rite of passage for new Compose developers. Luckily, this kind of refactoring is straightforward in Compose.

Relocate your `pizza` state into your **`ToppingsList`** composable (but do not let it get too comfortable).

Listing 2.9 Storing state inside a composable (`PizzaBuilderScreen.kt`)

```
~~private var pizza by mutableStateOf(Pizza())~~
...
@Composable
private fun ToppingsList(
    modifier: Modifier = Modifier
) {
    var pizza by remember { mutableStateOf(Pizza()) }

    LazyColumn(
        modifier = modifier
    ) {
        ...
    }
}
```

Although this code might look like you are delegating onto the **`remember`** function, keep in mind that **`remember`** will be returning a **`MutableState<Pizza>`**. *This* is the value that the `pizza` variable will delegate to, and it behaves exactly the same as the state delegation you set up before.

Run Coda Pizza again. You should see the same behavior as before, but now you have rid `PizzaBuilderScreen.kt` of a global variable that could add complications later.

State Hoisting

For your Coda Pizza app, you want the PLACE ORDER button to display the price of a pizza based on its toppings: A plain cheese pizza costs $9.99, and each topping adds $1.00 for the whole pizza or $0.50 for half the pizza. Define a computed property called `price` on your **`Pizza`** class to keep track of this price information.

Listing 2.10 Pricing pizzas (`Pizza.kt`)

```
import com.bignerdranch.android.codapizza.model.ToppingPlacement.*

data class Pizza(
    val toppings: Map<Topping, ToppingPlacement> = emptyMap()
) {
    val price: Double
        get() = 9.99 + toppings.asSequence()
            .sumOf { (_, toppingPlacement) ->
                when (toppingPlacement) {
                    Left, Right -> 0.5
                    All -> 1.0
                }
            }
    ...
}
```

Make sure you import your `Left`, `Right`, and `All` **`ToppingPlacement`**s, not the Compose constants with the same names.

Now, you need to make the `pizza` state accessible in your **`OrderButton`** composable.

Currently, this state is held by **`ToppingsList`**, and there is not a great way to share information with **`OrderButton`**, because the two composables are siblings within **`PizzaBuilderScreen`**. From Compose's perspective, these two composables are entirely unrelated; they cannot communicate with one another directly.

You need to move the state where both **`ToppingsList`** and **`OrderButton`** can access it – like their shared parent, **`PizzaBuilderScreen`**. (We did warn you this was coming.)

The need to move state out of a composable and up to the composable's caller is so common in development with Compose that it has a name: *state hoisting*. The pattern for hoisting state involves removing state from a composable and defining it instead as a parameter of the composable. If the composable also needs to update the state, it should take in a lambda expression that will be called with information about the change being made.

Think about your **`Checkbox`**. It does not hold any state itself; instead, it accesses the same `pizza` state that is currently held in **`ToppingsList`**. This, remember, is how your UI elements stay so effortlessly in sync.

In fact, **`Checkbox`** follows the state hoisting pattern. The same is true of the other commonly used built-in composables that change appearance in response to user input (like **`TextField`**, **`Switch`**, and **`Slider`**). This lets composables that depend on these components maintain complete control of their children's behavior. By hoisting state out of a composable, you end up with a flexible component whose behavior can be customized.

By the way, although state hoisting is a great tool for making more generalized components, do not stress about ensuring that *all* your composables are stateless. Many composables can effectively hold their own state, and you are free to decide where your UI state is held. And if you change your mind, you can easily refactor your code to incorporate state hoisting – as you will see momentarily.

To hoist the `pizza` state out of **`ToppingsList`**, you will make three changes: First, you will move the declaration of the `pizza` state into **`PizzaBuilderScreen`**. Second, you will define two new parameters on **`ToppingsList`**: a **`Pizza`** object to display in the list and a lambda expression that will be called when the pizza is modified to supply a new value for the pizza. With these arguments in place, you will then update any writes to the `pizza` property with calls to your lambda.

Make these changes now to hoist your `pizza` state (Listing 2.11).

Listing 2.11 Pizza hoisting (`PizzaBuilderScreen.kt`)

```
@Preview
@Composable
fun PizzaBuilderScreen(
    modifier: Modifier = Modifier
) {
    var pizza by remember { mutableStateOf(Pizza()) }

    Column(
        modifier = modifier
    ) {
        ToppingsList(
            pizza = pizza,
            onEditPizza = { pizza = it },
            modifier = Modifier
                .fillMaxWidth()
                .weight(1f, fill = true)
        )
        ...
    }
}

@Composable
private fun ToppingsList(
    pizza: Pizza,
    onEditPizza: (Pizza) -> Unit,
    modifier: Modifier = Modifier
) {
    var pizza by remember { mutableStateOf(Pizza()) }

    LazyColumn(
        modifier = modifier
    ) {
        items(Topping.values()) { topping ->
            ToppingCell(
                topping = topping,
                placement = pizza.toppings[topping],
                onClickTopping = {
                    val isOnPizza = pizza.toppings[topping] != null
                    pizza = onEditPizza(pizza.withTopping(
                        topping = topping,
                        placement = if (isOnPizza) {
                            null
                        } else {
                            ToppingPlacement.All
                        }
                    ))
                }
            )
        }
    }
}
...
```

Run Coda Pizza and confirm that its behavior has not changed.

With the pizza state now owned by **`PizzaBuilderScreen`**, you are ready to show price information in your **`OrderButton`**. Start by updating your string resource file to include a spot for the price to appear, using a format string placeholder.

Listing 2.12 Adding a price tag (`strings.xml`)

```
<resources>
    ...
    <string name="place_order_button">Place Order (%1$s)</string>
    ...
</resources>
```

Next, pass a `pizza` instance from **`PizzaBuilderScreen`** to **`OrderButton`** and use an instance of **`NumberFormat`** to convert the `price` property to a formatted string that you can display.

Listing 2.13 Showing pizza prices (`PizzaBuilderScreen.kt`)

```
@Preview
@Composable
fun PizzaBuilderScreen(
    modifier: Modifier = Modifier
) {
    ...
    Column(
        modifier = modifier
    ) {
        ...
        OrderButton(
            pizza = pizza,
            modifier = Modifier
                .fillMaxWidth()
                .padding(16.dp)
        )
    }
}
...
@Composable
private fun OrderButton(
    pizza: Pizza,
    modifier: Modifier = Modifier
) {
    Button(
        modifier = modifier,
        onClick = {
            // TODO
        }
    ) {
        val currencyFormatter = NumberFormat.getCurrencyInstance()
        val price = currencyFormatter.format(pizza.price)
        Text(
            text = stringResource(R.string.place_order_button, price)
                .toUpperCase(Locale.current)
        )
    }
}
```

Run Coda Pizza and test your new feature: When no toppings are added, the order button's text should read PLACE ORDER ($9.99). For each topping you add to the pizza, the price will increase by $1, the price of a topping placed on the entire pizza.

Notice that your **`ToppingsList`** composable can automatically update the **`OrderButton`** composable. By simply editing the state that backs the UI, every consumer of the UI state instantly receives its latest value. There is no extra effort on your part to track down all the reasons a certain UI element might need to be updated.

By the way, **`NumberFormat`** objects are slightly expensive to allocate, so discarding them between compositions is a bit wasteful. This is another practical application of **`remember`**, which you can use to keep resources available between recompositions. Wrap your **`NumberFormat`** in a **`remember`** block to try this out for yourself.

Listing 2.14 **remember**ing a **NumberFormatter** (PizzaBuilderScreen.kt)

```
...
@Composable
private fun OrderButton(
    pizza: Pizza,
    modifier: Modifier = Modifier
) {
    Button(
        modifier = modifier,
        onClick = {
            // TODO
        }
    ) {
        val currencyFormatter = remember { NumberFormat.getCurrencyInstance() }
        val price = currencyFormatter.format(pizza.price)
        Text(
            text = stringResource(R.string.place_order_button, price)
                .toUpperCase(Locale.current)
        )
    }
}
```

Run Coda Pizza again. The behavior of your app will not change – and because this overhead is fairly small and modern phones are very fast, the performance difference should be imperceptible. But you can now rest easy knowing that you are not unnecessarily throwing out work that you will need to redo on the next composition.

State and Configuration Changes

Currently, Coda Pizza has a small problem, and it will instantly be familiar to you. Run Coda Pizza and add a topping or two to the pizza. Then rotate your device or emulator.

Yep. The topping selections are lost after the configuration change. **remember** persists state across recompositions, but it has its limits: When your **Activity** is destroyed and re-created, so is your composition. Because the composition is discarded, it will restart from a blank slate when your **Activity** is re-created and calls **setContent**.

This was not an issue when you declared the `pizza` state as a global variable. But now, because your state is associated with your composition hierarchy – and, by extension, your activity – it must obey the rules of the activity lifecycle. Every variable stored using **remember** will be lost after a configuration change (and process death), just like values stored in your **Activity**.

In your time with the framework UI toolkit, you saw two approaches to solve this problem: the `savedInstanceState` bundle and **ViewModel**. Although **ViewModel**s can be a great tool for managing UI state and can be used in Jetpack Compose, they require more setup than is warranted for your needs right now in Coda Pizza. `savedInstanceState` will be your solution, and you will access it using a variation of **remember** called **rememberSaveable**.

The “saveable” portion of this function’s name refers to the fact that any remembered value is also automatically written to your **Activity**’s `savedInstanceState` bundle when it is destroyed. Remembered values that were saved can be restored when your composition is re-created, so values remembered in this way also survive configuration changes.

rememberSaveable is called in the same way as **remember**, using a lambda to perform its initialization. Try it out now. (This change will introduce a problem in your code, which we will explain next.)

Listing 2.15 Remembering and saving (PizzaBuilderScreen.kt)

```
@Preview
@Composable
fun PizzaBuilderScreen(
    modifier: Modifier = Modifier
) {
    var pizza by remember Saveable { mutableStateOf(Pizza()) }
    ...
}
...
```

Run Coda Pizza. It will crash with the following exception:

```
    IllegalArgumentException: MutableState containing Pizza(toppings={}) cannot
    be saved using the current SaveableStateRegistry. The default implementation
    only supports types which can be stored inside the Bundle. Please consider
    implementing a custom Saver for this class and pass it as a stateSaver
    parameter to rememberSaveable().
```

Coda Pizza has crashed because it is attempting to write a **Pizza** to a **Bundle**. But **Bundle**s are restricted in the types they can store: Only instances of **Serializable**, **Parcelable**, and basic types like `String`, `Int`, `Long`, `Float`, and `Double` are allowed in a **Bundle**. To fix this crash, you need to convert **Pizza** into a type that can be added to a **Bundle**.

Parcelable and Parcelize

The most effective way to let a **Pizza** fit into a **Bundle** is to make it a **Parcelable** class. **Parcelable** is an interface provided by the Android OS that allows a class to be converted into and read out of a **Parcel** object. **Parcel**s allow for compact storage of objects and are ideal for use in a **Bundle**.

The process of manually implementing the **Parcelable** interface is a bit complex – plus there are limitations about which data types can appear in the **Parcel**. Luckily, there is a plugin to help. With a bit of setup, you can have a **Parcelable** implementation automatically generated for you.

To do this, you will need to add a new plugin to your project called Parcelize, which takes care of generating **Parcelable** implementations for you at build time. Because Parcelize is a compiler plugin, your **Parcelable** implementations will always stay up to date with your class definitions, preventing errors when converting a **Parcel** back into the original object.

To add the Parcelize plugin, first register it with your project by adding its plugin ID and version to your `build.gradle` file labeled (Project: Coda_Pizza).

Listing 2.16 Adding the Parcelize plugin (build.gradle)

```
plugins {
    id 'com.android.application' version '7.2.2' apply false
    id 'com.android.library' version '7.2.2' apply false
    id 'org.jetbrains.kotlin.android' version '1.7.0' apply false
    id 'org.jetbrains.kotlin.plugin.parcelize' version '1.7.0' apply false
}
...
```

Next, apply this plugin to your application by registering it in the `app/build.gradle` file.

Listing 2.17 Enabling the Parcelize plugin (app/build.gradle)

```
plugins {
    id 'com.android.application'
    id 'org.jetbrains.kotlin.android'
    id 'org.jetbrains.kotlin.plugin.parcelize'
}
...
```

After making these changes to your build configuration files, do not forget to click the Sync Now button to make Android Studio aware of your changes.

After the sync completes, you are ready to make your **`Pizza`** class implement the **`Parcelable`** interface. With the help of Parcelize, you can accomplish this with two small changes. First, annotate the **`Pizza`** class with `@Parcelize`. Second, make **`Pizza`** implement the **`Parcelable`** interface.

Listing 2.18 Parcelizing pizza (`Pizza.kt`)

```
@Parcelize
data class Pizza(
    val toppings: Map<Topping, ToppingPlacement> = emptyMap()
) : Parcelable {
    ...
}
```

(If prompted, be sure to choose the import for `kotlinx.parcelize.Parcelize` instead of `kotlinx.android.parcel.Parcelize`. The `kotlinx.android` package is a relic of the now deprecated Kotlin Android Extensions plugin.)

`Parcelable` contains a few functions that all implementers must define. As you type this code, you may notice that errors about missing function overrides disappear as soon as you annotate the class with `@Parcelize`. Parcelize will automatically provide the entire implementation for this interface with no extra effort on your part.

Run Coda Pizza. This time, it will not crash, and you will be presented with the familiar list of toppings. Add some toppings to the pizza and rotate the emulator or device. The state should survive the configuration change, and you should see the same selection of toppings – no matter how many times you rotate the phone or what other configuration changes Coda Pizza encounters (Figure 2.2). And because your state is stored in the `savedInstanceState` bundle, it will even survive process death.

Figure 2.2 Saving pizzas across configuration changes

State is a crucial part of any application, and in the world of Compose, it is entirely in your hands. But the discussion of state does not end there. In the next chapter, you will incorporate a dialog into Coda Pizza and see how Compose's philosophy of state lets you achieve a wide variety of app behaviors.

For the More Curious: Scrolling State

Jetpack Compose's ideas about state and recomposition are woven throughout the framework. In fact, Coda Pizza was leveraging these two concepts even before you added state of your own.

Think about your **`LazyColumn`**. It needs to keep track of its scroll position, which it does automatically. But it also uses the state hoisting pattern to allow its parent to manage the scrollable state. How does it do this? To see for yourself, take a look at its signature:

```
@Composable
fun LazyColumn(
    modifier: Modifier = Modifier,
    state: LazyListState = rememberLazyListState(),
    ...
)
```

`rememberLazyListState` does two things: It creates a **`LazyListState`** object with an initial position at the beginning of the list, and it remembers that state via the **`remember`** function. For many lists, you do not need to think about this behavior – the default, automatically managed scrolling state will simply do the right thing. But if you ever need to read or control the scroll position of the **`LazyColumn`**, the option is always available.

To take the scroll state into your own hands, you can create your own **`LazyListState`** and pass it in as the `state` parameter. The code to do so might look like this:

```
val listState = rememberLazyListState()
LazyColumn(
    state = listState
) {
    // Add items to the LazyColumn
}
```

This code effectively does the same thing as the default, automatically managed `state` parameter, but you now have a reference to the state being used. This lets you both read the scroll state and modify it, if desired, which you can do like this:

```
// Determine whether the user is currently scrolled to the top of the list
val isAtTopOfList = (listState.firstVisibleItemIndex == 0) &&
    (listState.firstVisibleItemScrollOffset == 0)

// Scroll to the top of the list from the current scroll position
coroutineScope.launch {
    // Suspends until the scroll animation finishes
    listState.scrollToItem(index = 0, scrollOffset = 0)
}
```

The **`LazyListState`** backs its scroll position properties with **`State`** objects, meaning that the scroll position can be observed and trigger recompositions just as you have seen for other state. Most of the built-in composables will explicitly require state, but other composables that have an implicit or self-managed state will still offer some mechanism to read and control the state, as you saw with **`LazyColumn`**.

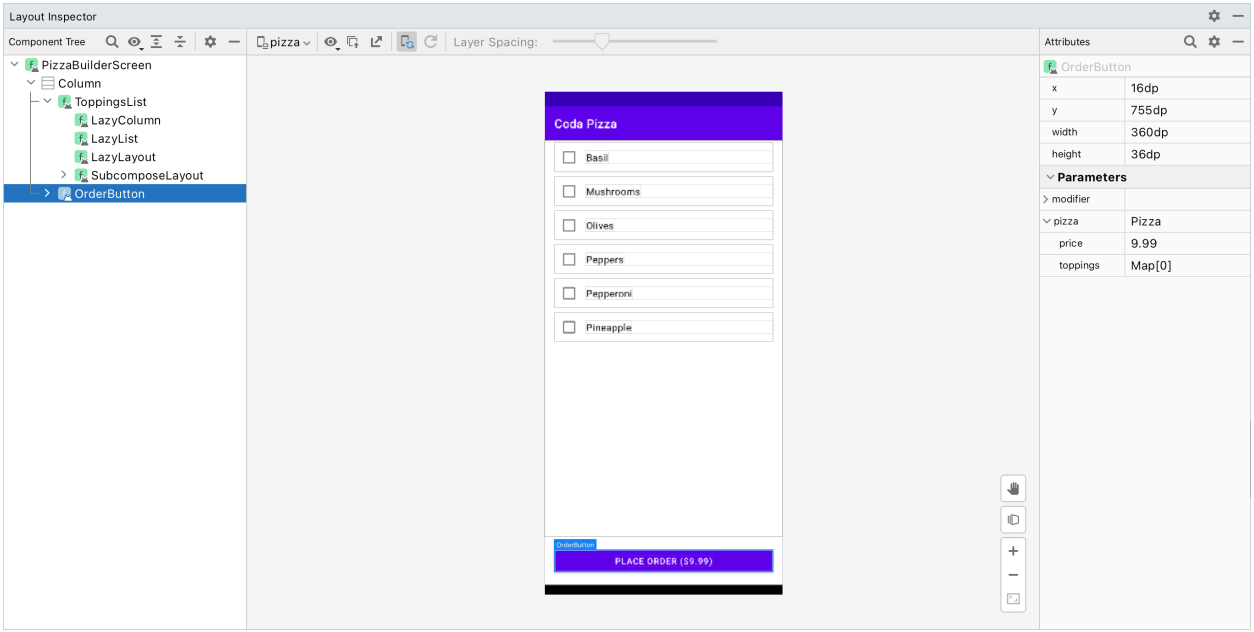
These design paradigms make the built-in composables highly flexible, and you can apply the same ideas to your own composables if you need to make flexible, reusable components like the ones included with Compose.

For the More Curious: Inspecting Compose Layouts

Sometimes you will need to debug your Compose UIs just as you have debugged traditional UIs using the framework view system. With framework views, you might have explored the layout inspector. The layout inspector allows you to see the configuration of your views, including their nesting, attributes, and position. You can also break the view into layers to see exactly what is being rendered by a given view.

The layout inspector (as well as many other Android UI debugging tools) fully supports Jetpack Compose. Try it out for yourself by opening the layout inspector with its menu option, Tools → Layout Inspector, then running Coda Pizza. The layout inspector will open in the bottom of the IDE (Figure 2.3).

Figure 2.3 Inspecting your Compose UI



Explore the component tree on the left side of the layout inspector. You will see all of the composables in your layout, represented as a hierarchy of nodes. Double-clicking a node will take you to the composable call that contributed the element to your UI.

Find and select your **`OrderButton`** composable in the tree. The screen preview in the center of the layout inspector will be marked with the bounding box of the button, and the attributes window on the right will update to show the composable's attributes. In the Parameters section, you will see all the inputs that were passed to the composable.

You cannot use the layout inspector to edit the attributes of a composable (nor the attributes of a **`View`**), but getting a better understanding of your layout can be invaluable for debugging.

Try experimenting with the other techniques you have learned to debug your apps. You will find that many or all of them still work in the declarative world of Jetpack Compose.

3

Showing Dialogs with Jetpack Compose

Understanding how Jetpack Compose treats state and recomposition are crucial to effectively using Compose. The ramifications of state changes in your code being the trigger for UI updates extend throughout the framework. A great example of this is how Compose handles dialogs.

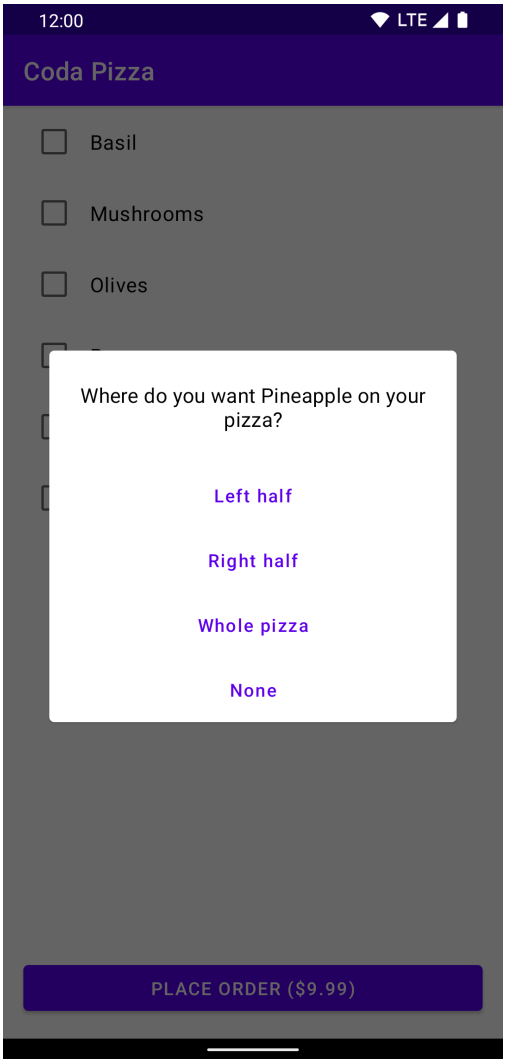
When you want to display a dialog in the framework UI toolkit, you use a class like **`AlertDialog`** or **`DatePickerDialog`** – ideally, wrapped in a **`DialogFragment`**. To display one of these dialogs, you call the appropriate **`show`** function. The dialog then does whatever it wants to do and dismisses itself. If there is a result it needs to send back to whatever component summoned it, it needs to safely transfer the data back to that location – often with a fair amount of orchestration.

In Compose, dialogs follow the same rule of declarative UIs that you have seen before: If you say that a dialog should be shown, it will be shown. You are in control of when the dialog disappears – the dialog can request that it be dismissed, but the call site gets the final say.

Because the state of whether the dialog is visible is known by the composable that hosts the dialog, there is no need to wrap the dialog in another container to manage its lifecycle. And because the dialog is directly managed by another composable, transferring data between the two is as simple as declaring a lambda expression to serve as a callback – no need to set up a finicky line of communication between two components.

In Coda Pizza, you will use a dialog to ask your users where they would like a topping to be placed on their pizza. The finished dialog will look like Figure 3.1, and it will appear any time the user selects a topping from the list.

Figure 3.1 The final dialog



Your First Dialog in Compose

There are several types of dialogs available in Compose, including an **`AlertDialog`** composable that mimics the appearance of its framework counterpart. You will be using **`Dialog`**, which is more agnostic about its content and will let you build a more custom UI. Regardless of which dialog flavor you choose, the semantics are largely the same – especially when it comes to how the dialog's state is managed.

On its own, the **`Dialog`** function renders an empty window, so you will need to create a new composable that builds off of **`Dialog`** and provides the dialog's view. Building this UI will require a fair amount of code, so your composable for this dialog should be in its own file to keep your code organized.

Create a new file in the `ui` package called `ToppingPlacementDialog.kt`. In your new file, define a composable function called **`ToppingPlacementDialog`** that calls the **`Dialog`** function.

`Dialog` requires two inputs: an `onDismissRequest` lambda and a content lambda. We will revisit the role of `onDismissRequest` later – for now, leave it empty. To begin your work on the content, create a placeholder UI to show in the dialog: a boring red box. You will use this to make sure you have everything set up before adding more functionality. Use a **`Box`** composable and make it visible by setting its background color to `Color.Red` and its width and height to 64dp:

Listing 3.1 Painting the dialog red (`ToppingPlacementDialog.kt`)

```
@Composable
fun ToppingPlacementDialog() {
    Dialog(onDismissRequest = { /* TODO */ }) {
        Box(
            modifier = Modifier
                .background(Color.Red)
                .size(64.dp)
        )
    }
}
```

(Be sure to add the import for `androidx.compose.ui.window.Dialog` instead of `android.app.Dialog`.)

To see your dialog in action, you need to call **`ToppingPlacementDialog`**. Because **`Dialog`** appears in its own window, it does not matter where in your composition this function call appears. The result will always be the same: a fullscreen dialog with the specified content. In this case, we recommend showing the dialog from **`ToppingsList`**.

`ToppingsList` is a good candidate for managing the dialog because it is a convenient place to manage the dialog's state. When any of the **`ToppingCell`**s in its **`LazyColumn`** are clicked, the dialog should be shown. **`ToppingsList`** already has visibility into how **`ToppingCell`**s are created, making this a small change.

You do not want to store this state too far up your composition hierarchy, since each level above **`ToppingsList`** means that you need another pair of parameters to access and change the state. No other component in Coda Pizza will need to be aware of this state, so managing it directly in **`ToppingsList`** will prevent unnecessary clutter in your code and make it easier to read.

Add a call to **`ToppingPlacementDialog`** in **`ToppingsList`**.

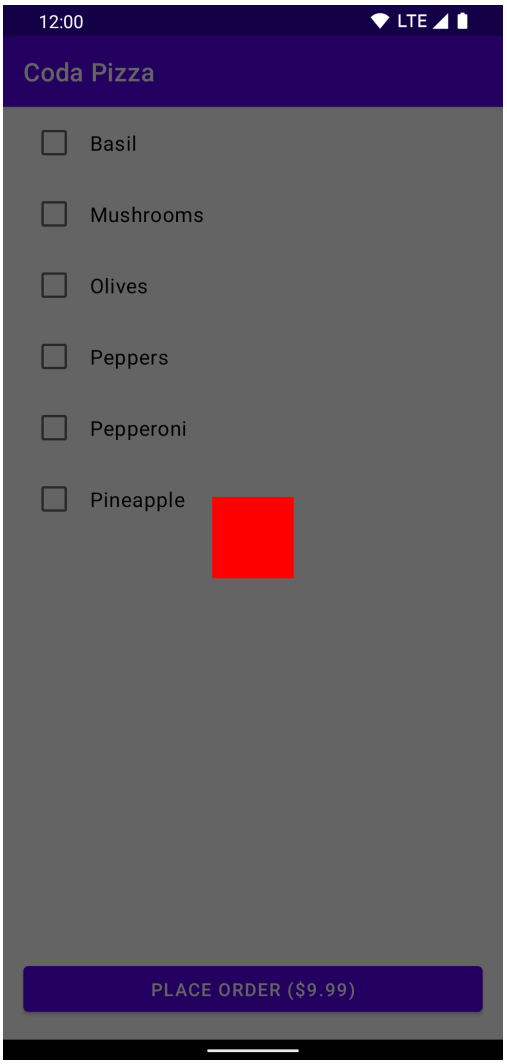
Listing 3.2 Showing a dialog (`PizzaBuilderScreen.kt`)

```
...
@Composable
private fun ToppingsList(
    pizza: Pizza,
    onEditPizza: (Pizza) -> Unit,
    modifier: Modifier = Modifier
) {
      ToppingPlacementDialog()

      LazyColumn(
          modifier = modifier
      ) {
          ...
      }
  }
  ...
```

What do you think will happen when this code executes? To find out, run Coda Pizza. When the app launches, you will see your familiar list of toppings behind a dark overlay with a red square in the center of the window (Figure 3.2).

Figure 3.2 A test dialog



The red square is your dialog, and the dark overlay is being added by the system (the same overlay appears behind dialogs in the framework UI toolkit, as you may have noticed). With the dialog open, try dismissing it – either with the Back button or by clicking outside the dialog (on the dark backdrop). Despite your efforts, you will be unable to dismiss it.

Dismissing the Dialog

Whenever a dialog is part of your composition, it will be shown onscreen. You have not told your **`ToppingPlacementDialog`** when it should stop being displayed, so attempts to dismiss it will do nothing. And none of Compose's dialog functions has a parameter to set the dialog's visibility, so you will need another way to dismiss your dialog.

In Chapter 1, you made another UI element that was visible only some of the time: the placement label in your **`ToppingCell`**. You use an `if` statement so that the **`Text`** composable is only invoked when you want it to be visible. You will use the same approach to show and hide the dialog.

With the dialog's visibility controlled by an `if` statement, the job of the `onDismissRequest` lambda will be to update the condition set by the `if` statement so that the **`Dialog`** function will not be invoked again when your UI is

recomposed. This means that, for your dialog's visibility to be managed correctly, **`ToppingPlacementDialog`** will need to forward its requests to be dismissed.

Declare a new parameter called `onDismissRequest`, mirroring the parameter from the base **`Dialog`** composable, and pass it to your **`Dialog`**.

Listing 3.3 Forwarding dismiss requests (`ToppingPlacementDialog.kt`)

```
@Composable
fun ToppingPlacementDialog(
    onDismissRequest: () -> Unit
) {
    Dialog(onDismissRequest = { /* TODO */ } onDismissRequest) {
        Box(
            modifier = Modifier
                .background(Color.Red)
                .size(64.dp)
        )
    }
}
```

With this parameter in place, **`ToppingPlacementDialog`** now has everything it needs to have its visibility managed. To track whether the dialog should be visible, define a new **`MutableState`** property and surround your call to **`ToppingPlacementDialog`** in an `if` statement that checks this state. You will want the dialog state to persist across configurations, so use **`rememberSaveable`** instead of **`remember`**.

Your new state will be driven by two events: When a topping is clicked, the dialog should be shown. When the dialog requests to be dismissed, it should be hidden. Drive this state by setting your `onClickTopping` and `onDismissRequest` callback implementations with an update to your state.

Listing 3.4 Managing your dialog's state (`PizzaBuilderScreen.kt`)

```
...
@Composable
private fun ToppingsList(
    pizza: Pizza,
    onEditPizza: (Pizza) -> Unit,
    modifier: Modifier = Modifier
) {
      var showToppingPlacementDialog by rememberSaveable { mutableStateOf(false) }

      if (showToppingPlacementDialog) {
          ToppingPlacementDialog(
              onDismissRequest = {
                  showToppingPlacementDialog = false
              }
          )
      }

      LazyColumn(
          modifier = modifier
      ) {
          items(Topping.values()) { topping ->
              ToppingCell(
                  topping = topping,
                  placement = pizza.toppings[topping],
                  onClickTopping = {
                      val isOnPizza = pizza.toppings[topping] != null
                      onEditPizza(pizza.withTopping(
                          topping = topping,
                          placement = if (isOnPizza) {
                              null
                          } else {
                              ToppingPlacement.All
                          }
                      ))
                      showToppingPlacementDialog = true
                  }
              )
          }
      }
  }
  ...
```

By the way, although you could technically use Kotlin's trailing lambda syntax to omit the parameter name and parentheses after **`ToppingPlacementDialog`**, we do not recommend it. Callbacks for composables are most effectively identified by their name, and it can be difficult to determine what a lambda does in Compose when used with the trailing lambda syntax.

We recommend only using the trailing lambda syntax with a composable when you are passing in its main content. If the parameter name is anything besides `content`, the conventional name for the "primary" content of a composable, the trailing lambda syntax can remove a label that is important in understanding how your UI will appear.

Run Coda Pizza. Now, the app displays the toppings list, as before. Clicking a topping shows the placeholder dialog, which can now be dismissed by clicking outside the dialog or by pressing the Back button. The dialog is not dismissed if the user presses the red square itself, which lets the user interact with the dialog without dismissing it.

Setting the Dialog's Content

Now that the dialog can show and hide itself, you can focus on its content. Start by adding a string resource to show in the dialog.

Listing 3.5 Asking important questions (`strings.xml`)

```
<resources>
  ...
  <string name="place_order_button">Place order (%1$s)</string>

  <string name="placement_prompt">Where do you want %1$s on your pizza?</string>
  <string name="placement_none">None</string>
  <string name="placement_left">Left half</string>
  <string name="placement_right">Right half</string>
  <string name="placement_all">Whole pizza</string>
  ...
</resources>
```

Now you are ready to start building the real UI to appear in the dialog. Remove the temporary **`Box`** and replace it with a **`Card`**. **`Card`** includes a background, drop shadow, and rounded corners – exactly what you need for your dialog. Its children are stacked on top of one another (like a **`FrameLayout`**'s), so you will only include one direct child in the **`Card`**.

In your **`Card`**, add a **`Text`** with the prompt you just declared. Place the **`Text`** in a **`Column`**, because you will need to add buttons underneath the prompt shortly. You will also need to add a new parameter to accept the name of the topping being added to the pizza.

Listing 3.6 Asking the right question (`ToppingPlacementDialog.kt`)

```
@Composable
fun ToppingPlacementDialog(
    topping: Topping,
    onDismissRequest: () -> Unit
) {
    Dialog(onDismissRequest = onDismissRequest) {
        Box(
            modifier = Modifier
                .background(Color.Red)
                .size(64.dp)
        )
        Card {
            Column {
                val toppingName = stringResource(topping.toppingName)
                Text(
                    text = stringResource(R.string.placement_prompt, toppingName),
                    style = MaterialTheme.typography.subtitle1,
                    textAlign = TextAlign.Center,
                    modifier = Modifier.padding(24.dp)
                )
            }
        }
    }
}
```

Because you have added a new parameter to **`ToppingPlacementDialog`**, **`ToppingsList`** will now have a compiler error. Your dialog needs to know not only whether it should be visible but also what content it should show. More specifically, your dialog needs to know which topping was selected, not just that *a* topping was selected.

To keep track of this information, you will need to be a bit more clever about the dialog state you store. Instead of keeping track of whether the dialog should appear, your state can instead track which topping you are in the process of putting on the pizza.

If the user has not selected a topping, this state should be null to indicate that no topping was selected. Otherwise, the most recently selected topping can be remembered and shown in the dialog. Make this change now, replacing your current `showToppingPlacementDialog` state.

Listing 3.7 Smarter state (`PizzaBuilderScreen.kt`)

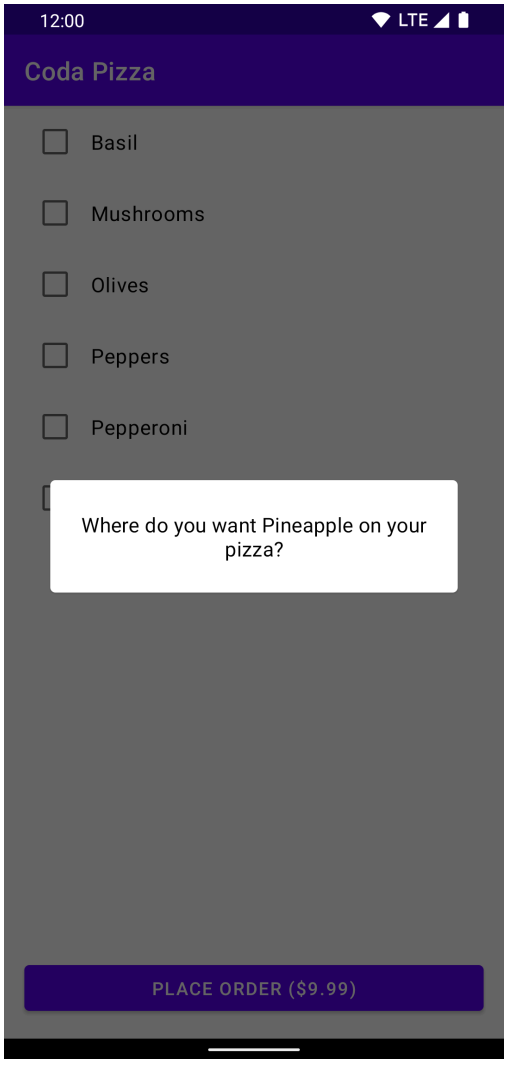
```
...
@Composable
private fun ToppingsList(
    pizza: Pizza,
    onEditPizza: (Pizza) -> Unit,
    modifier: Modifier = Modifier
) {
    var showToppingPlacementDialog by rememberSaveable { mutableStateOf(false) }
    var toppingBeingAdded by rememberSaveable { mutableStateOf<Topping?>(null) }

    if (showToppingPlacementDialog) {
    toppingBeingAdded?.let { topping ->
        ToppingPlacementDialog(
            topping = topping,
            onDismissRequest = {
                showToppingPlacementDialog = false
                toppingBeingAdded = null
            }
        )
    }

    LazyColumn(
        modifier = modifier
    ) {
        items(Topping.values()) { topping ->
            ToppingCell(
                topping = topping,
                placement = pizza.toppings[topping],
                onClickTopping = {
                    showToppingPlacementDialog = true
                    toppingBeingAdded = topping
                }
            )
        }
    }
}
...
```

Run Coda Pizza once again and select any of the toppings. You will now see a dialog – one that actually looks more like a dialog this time – asking the user about the topping they just selected (Figure 3.3). Dismissal will work as it has before, but because there are no placement options yet, your users are still confined to the dullness of cheese pizzas.

Figure 3.3 The beginning of a dialog



Time to add those topping placement options. You will add four options to this dialog: Whole pizza, Left half, Right half, and None. There are several composables you could use to create these options, but **`TextButton`** is a great fit for your needs.

Each of the buttons you will add to this dialog will require a similar set of customizations. They will all need to fill the width of the dialog, they will all have 8dp of padding, and they will all pull their labels from the topping's string resource. To make these buttons a bit easier to add, start by declaring a **`ToppingPlacementOption`** composable in `ToppingPlacementDialog.kt`. You will use this composable to add the choices to your dialog.

Listing 3.8 Defining a reusable button (`ToppingPlacementDialog.kt`)

```
@Composable
fun ToppingPlacementDialog(
    topping: Topping,
    onDismissRequest: () -> Unit
) {
    ...
}

@Composable
private fun ToppingPlacementOption(
    @StringRes placementName: Int,
    onClick: () -> Unit,
    modifier: Modifier = Modifier
) {
    TextButton(
        onClick = onClick,
        modifier = modifier.fillMaxWidth()
    ) {
        Text(
            text = stringResource(placementName),
            modifier = Modifier.padding(8.dp)
        )
    }
}
```

Much like **`Button`**, **`TextButton`** accepts a lambda to define the label of the button. This means that, despite its name, it is possible to place something like an icon inside a **`TextButton`**. **`TextButton`** simply has a few optimizations that make it ideal for hosting **`Text`**, like automatically setting the text color with an appropriate button color. But even with these handy default customizations, you do not want to duplicate this hierarchy for each button you want to add to the dialog. With your **`ToppingPlacementOption`** composable, adding a button to the dialog is a single function call away.

Next, you can declare the four buttons in the dialog. You could declare them one by one – but remember that you have control flow at your disposal: You can use a loop to add several items onscreen at once. Try it out now by iterating over all values of **`ToppingPlacement`**. (Leave each button's `onClick` callback blank for now.) Remember that you did not add the "none" option as a case to **`ToppingPlacement`**, so you will need to manually add the fourth option to this dialog.

Listing 3.9 Adding options (`ToppingPlacementDialog.kt`)

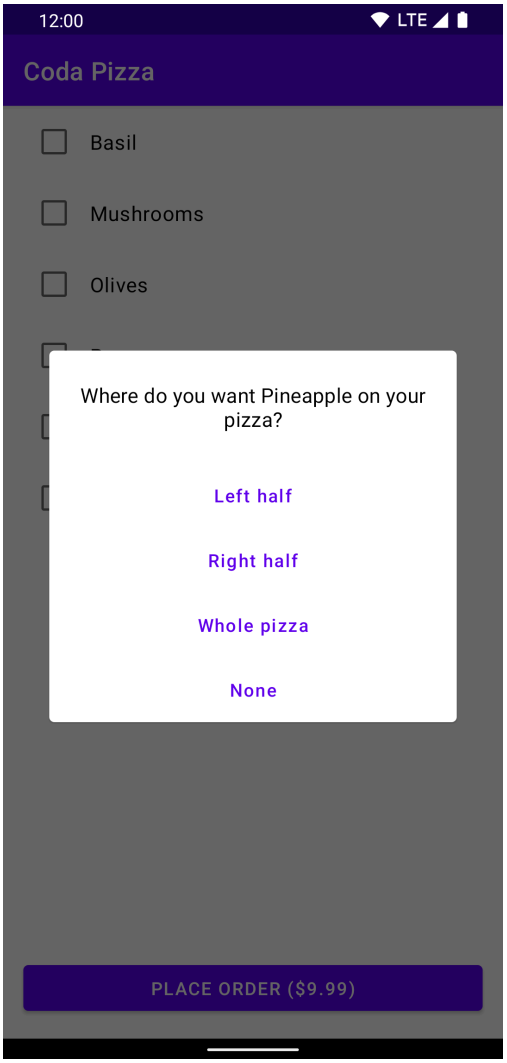
```
@Composable
fun ToppingPlacementDialog(
    topping: Topping,
    onDismissRequest: () -> Unit
) {
    Dialog(onDismissRequest = onDismissRequest) {
        Card {
            Column {
                val toppingName = stringResource(topping.toppingName)
                Text(
                    ...
                )

                ToppingPlacement.values().forEach { placement ->
                    ToppingPlacementOption(
                        placementName = placement.label,
                        onClick = { /* TODO */ }
                    )
                }

                ToppingPlacementOption(
                    placementName = R.string.placement_none,
                    onClick = { /* TODO */ }
                )
            }
        }
    }
}
...
```

Run Coda Pizza. When you click a topping, you will now be presented with the full list of options for placing the topping (Figure 3.4).

Figure 3.4 Do you want to build a pizza?



Sending Results from a Dialog

Your final task is to wire up all the buttons in your dialog so they can correctly update the user's pizza. Currently, the options in this dialog do nothing more than offer confirmation that they were, in fact, pressed.

Each of the options in this dialog should do two things when pressed: It should notify the creator of the dialog which choice was selected, and it should dismiss the dialog. You already have everything you need to make the options dismiss the dialog. The `onDismissRequest` callback can be reused after selecting an option to indicate that the dialog is requesting to be dismissed.

Update your two empty `onClick` callbacks with this behavior.

Listing 3.10 Dismissing the dialog (`ToppingPlacementDialog.kt`)

```
@Composable
fun ToppingPlacementDialog(
    topping: Topping,
    onDismissRequest: () -> Unit
) {
    Dialog(onDismissRequest = onDismissRequest) {
        Card {
            Column {
                ...
                ToppingPlacement.values().forEach { placement ->
                    ToppingPlacementOption(
                        placementName = placement.label,
                        onClick = { /* TODO */ }
                            onDismissRequest()
                        }
                    )
                }

                ToppingPlacementOption(
                    placementName = R.string.placement_none,
                    onClick = { /* TODO */ }
                        onDismissRequest()
                    }
                )
            }
        }
    }
}
...
```

Run Coda Pizza. Click any of the toppings and select any of the four placements. Although the pizza itself will not change, notice that the dialog is dismissed.

To update the pizza, you will again use the state hoisting pattern you used in the **`ToppingsList`**. **`ToppingPlacementDialog`** is not in control of the pizza state, but its parent is. To modify the state, **`ToppingPlacementDialog`** will need to take in another lambda to request a change to the pizza.

Add a function parameter called `onSetToppingPlacement`. This parameter will be a lambda that passes the selected **`ToppingPlacement`** value (or a null value, if None was selected). After you have this parameter in place, invoke it in each button's `onClick` callback before dismissing the dialog.

Listing 3.11 Sending results back (ToppingPlacementDialog.kt)

```
@Composable
fun ToppingPlacementDialog(
    topping: Topping,
    onSetToppingPlacement: (placement: ToppingPlacement?) -> Unit,
    onDismissRequest: () -> Unit
) {
    Dialog(onDismissRequest = onDismissRequest) {
        Card {
            Column {
                ...
                ToppingPlacement.values().forEach { placement ->
                    ToppingPlacementOption(
                        placementName = placement.label,
                        onClick = {
                            onSetToppingPlacement(placement)
                            onDismissRequest()
                        }
                    )
                }

                ToppingPlacementOption(
                    placementName = R.string.placement_none,
                    onClick = {
                        onSetToppingPlacement(null)
                        onDismissRequest()
                    }
                )
            }
        }
    }
}
...
```

To use this returned value, you will also need to update **ToppingsList** to handle the topping placement selection. **ToppingsList** will then delegate to its onEditPizza callback so that **PizzaBuilderScreen** can commit the change.

Listing 3.12 Handling the result (PizzaBuilderScreen.kt)

```
...
@Composable
private fun ToppingsList(
    pizza: Pizza,
    onEditPizza: (Pizza) -> Unit,
    modifier: Modifier = Modifier
) {
    var toppingBeingAdded by rememberSaveable { mutableStateOf<Topping?>(null) }

    toppingBeingAdded?.let { topping ->
        ToppingPlacementDialog(
            topping = topping,
            onSetToppingPlacement = { placement ->
                onEditPizza(pizza.withTopping(topping, placement))
            },
            onDismissRequest = {
                toppingBeingAdded = null
            }
        )
    }
    ...
}
...
```

Your dialog is now complete. Run Coda Pizza and customize a pizza to your heart's content. Spend a moment admiring the power of choice, and notice that your UI – including the PLACE ORDER button – automatically updates, all at once, for each topping you change.

This is the power of declarative programming in Jetpack Compose: You did nothing to tell any component onscreen to update, nor did you specify where a change would be coming from. But because you wrapped your values in **`State`** objects, Compose takes care of your UI updates, regardless of how or why the UI needs to change. And because dialogs are simply composables, you have the full flexibility to communicate directly to them without jumping through any hoops.

In the next chapter, you will finish your work on Coda Pizza by adding a pizza preview image and customizing some of the app's visual elements.

Challenge: Pizza Sizes and Drop-Down Menus

For this challenge, you will expand the customization options available in Coda Pizza. Currently, it is only possible to order a pizza in a single size. You will change that by adding another UI element to prompt the user to choose a pizza size.

For some UI interactions, a dialog can be a bit intrusive. It forces your users to interact with a specific message and hides the rest of your application's UI. As an alternative, you can use a drop-down menu to show a set of options that blocks a much smaller portion of your UI.

Creating a drop-down menu in Compose is similar to how you created a **`Dialog`**. A dropdown can be shown using the **`DropdownMenu`** composable. We have copied its signature below, and you can find its full documentation with the rest of the Material composables at `developer.android.com/reference/kotlin/androidx/compose/material/package-summary`.

```
@Composable
fun DropdownMenu(
    expanded: Boolean,
    onDismissRequest: () -> Unit,
    modifier: Modifier = Modifier,
    offset: DpOffset = DpOffset(0.dp, 0.dp),
    properties: PopupProperties = PopupProperties(focusable = true),
    content: @Composable ColumnScope.() -> Unit
)
```

There are two notable differences between how you use a **`DropdownMenu`** and how you use a **`Dialog`**. First, **`DropdownMenu`** specifies an `expanded` parameter, which controls whether the menu is expanded (visible) or collapsed (hidden). This means that you do not need to wrap your usage of **`DropdownMenu`** in an `if` statement, like you did with **`Dialog`**.

Second, where the **`DropdownMenu`** is drawn onscreen is directly affected by where it is placed in your composition hierarchy. A **`Dialog`** always fills the full size of your app and draws over all other composables. But **`DropdownMenu`** is *anchored* to its parent composable, meaning that it will appear in the same area of your screen as the composable that hosts the menu. It is conventional for menus on Android to expand outward from and on top of the UI element that caused the menu to appear. Keep this in mind when you are deciding where to nest your **`DropdownMenu`**.

Take **`DropdownMenu`** for a spin by adding a dropdown near the top of the screen that lets the user change their pizza's size. You will also find the **`DropdownMenuItem`** composable handy, for adding choices into your drop-down menu. Give your customers four size options: small, medium, large, and extra large. Smaller pizza sizes should be less expensive than larger pizzas, and your pizzas should be large by default. (You are free to decide Coda Pizza's exact prices for this challenge.)

You will need to define new UI state to track the selected size as part of this challenge. We recommend defining a new enum called **`Size`** to declare the size options and adding a size property to your **`Pizza`** data class to track the user's size selection.

4

Theming Compose UIs

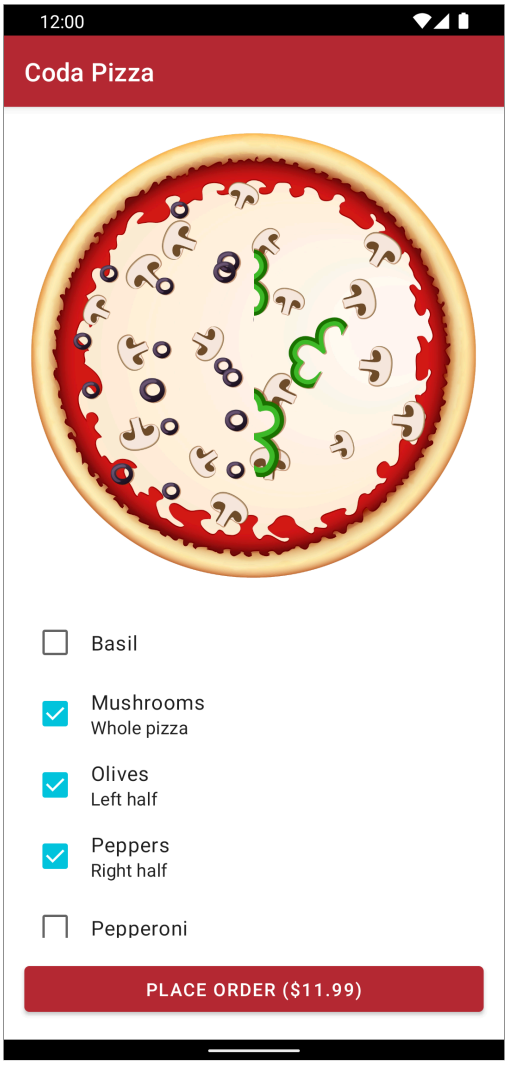
Coda Pizza's users can now customize their pizzas, but the app itself is still the equivalent of plain cheese: It gets the job done, but it has no particular personality. In this chapter, you will spend some time polishing Coda Pizza and adding some visual flair.

You will start by adding a preview of the user's customized pizza. It will be placed above your list of toppings and will automatically update every time the user changes the toppings on their pizza. You will also learn how to specify a theme in Compose, which works differently than the application themes you have seen in a `themes.xml` file.

Finally, you will reflect on what Coda Pizza does *not* have compared to the other apps you have built in the past – and what Compose means for the future of Android development.

When you finish, Coda Pizza will look like Figure 4.1.

Figure 4.1 Coda Pizza's final form



Images

Coda Pizza offers 4,096 combinations of toppings for your users to select. Creating a preview image for each combination is not feasible, so you will need to generate the preview for the user's pizza on the fly. You will do this by combining several images, layering them on top of each other to get the final result.

The preview will start with an image of a plain cheese pizza. For each topping the user adds, you will overlay an image of that topping on the base image. If a topping is only on half of the pizza, you will crop the image so that it only appears on the correct half.

Start by importing the images you will use to build your pizza previews. If you do not already have them, download the solutions to the exercises in this book from `www.bignerdranch.com/android-5e-solutions`. Unzip this archive and locate the solution for Coda Pizza in the `28. Theming Compose UIs/Solution/CodaPizza` folder. Navigate to `app/src/main/res/drawable` and copy the `.webp` files in that folder into the `app/src/main/res/drawable` folder of your project.

The images you just copied to your project are `.webp`s instead of `.xml` vectors, like you might have used before. Vectors are a great choice for UI elements and simple designs, but not all images can be effectively expressed as a vector. Complex images and photographs, in particular, are either impossible to represent as vectors or can lead to performance issues if they are used as vectors.

Your pizza preview images fall into this category, so you need another format, like `.webp`. By the way, Android also supports other image formats in your resources, including `.png` and `.jpeg` files. We have chosen `.webp` here because it offers smaller file sizes.

With your assets in place, you can begin to create the composable that will show pizza previews. Initially, you will only show the base image. Later, you will programmatically draw other toppings on top of this image.

To display images in Compose, you use the **`Image`** composable. When calling **`Image`**, you provide a **`Painter`** to specify the image you want to display. **`Painter`** is analogous to the **`Drawable`** class you have used before. It declares something that can be "painted" to the screen, like a vector image, bitmap image, solid color, or gradient.

You can obtain a **`Painter`** for one of your drawable resources by calling **`painterResource`**. Much like the **`stringResource`** function you saw earlier, **`painterResource`** will trigger Compose to take care of querying your resources, loading the right image, and converting it into a **`Painter`** that can be used with your **`Image`**.

Create a new file in the `ui` package called `PizzaHeroImage.kt`. (A *hero image* is a large image placed prominently at the top of a page.) Define a new composable called **`PizzaHeroImage`** that will show pizza previews. The **`PizzaHeroImage`** should have two arguments: a **`Pizza`** and the compulsory **`Modifier`** argument. Use **`Image`** to show the image of a plain pizza, and give your new composable a preview function to see your changes in Android Studio.

Listing 4.1 Let me imagine it... (`PizzaHeroImage.kt`)

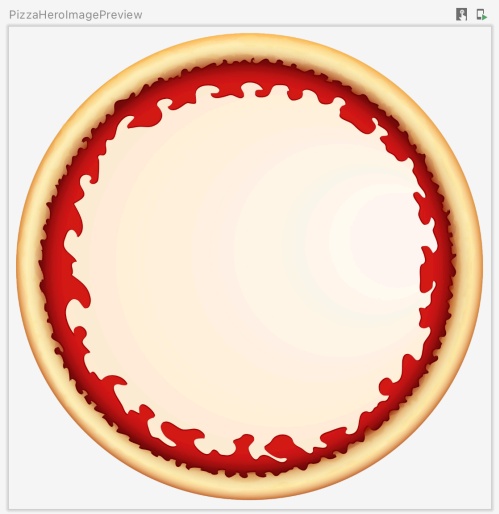
```
@Preview
@Composable
private fun PizzaHeroImagePreview() {
    PizzaHeroImage(
        pizza = Pizza(
            toppings = mapOf(
                Topping.Pineapple to ToppingPlacement.All,
                Topping.Pepperoni to ToppingPlacement.Left,
                Topping.Basil to ToppingPlacement.Right
            )
        )
    )
}

@Composable
fun PizzaHeroImage(
    pizza: Pizza,
    modifier: Modifier = Modifier
) {
    Image(
        painter = painterResource(R.drawable.pizza_crust),
        contentDescription = null,
        modifier = modifier
    )
}
```

Be sure to import **`Image`** from the `androidx.compose.foundation` package.

Change to the split view in your editor and build your project to update the preview. When the build completes, you should see the image of a plain pizza in the preview (Figure 4.2).

Figure 4.2 Plain pizza



Image's contentDescription

When you added your **`Image`**, you also had to specify a `contentDescription`. Like the `android:contentDescription` XML attribute you have used with framework views, this argument is used for accessibility: It gives screen readers text to read out. But unlike `android:contentDescription`, this parameter is mandatory and must always be provided.

You set the `contentDescription` to `null` initially. But you should respect this parameter's purpose and provide a content description for your image. Start by adding a string resource to describe the image.

Listing 4.2 Defining a content description (`strings.xml`)

```
<resources>
    ...
    <string name="pizza_preview">Pizza preview</string>
</resources>
```

With the string resource in place, you can specify the content description of the **`Image`**.

Listing 4.3 Describing the content (`PizzaHeroImage.kt`)

```
@Composable
fun PizzaHeroImage(
    pizza: Pizza,
    modifier: Modifier = Modifier
) {
    Image(
        painter = painterResource(R.drawable.pizza_crust),
        ~~contentDescription = null,~~
        contentDescription = stringResource(R.string.pizza_preview),
        modifier = modifier
    )
}
```

Having a content description in place will not change your app's appearance, but it does make your app more accessible to users who rely on screen readers.

Adding more images

Next, you will work on stacking toppings on the base pizza to form the final preview. The first step is to track which image to draw when a topping is placed on a pizza.

Add a new property to your **`Topping`** enum called `pizzaOverlayImage`. This will keep track of which image to use for each topping. After you add the property, give each case of the enum a value to associate the toppings with their images.

Listing 4.4 Associating the preview images (`Topping.kt`)

```
enum class Topping(
    @StringRes val toppingName: Int,
    @DrawableRes val pizzaOverlayImage: Int
) {
    Basil(
        toppingName = R.string.topping_basil,
        pizzaOverlayImage = R.drawable.topping_basil
    ),

    Mushroom(
        toppingName = R.string.topping_mushroom,
        pizzaOverlayImage = R.drawable.topping_mushroom
    ),

    Olive(
        toppingName = R.string.topping_olive,
        pizzaOverlayImage = R.drawable.topping_olive
    ),

    Peppers(
        toppingName = R.string.topping_peppers,
        pizzaOverlayImage = R.drawable.topping_peppers
    ),

    Pepperoni(
        toppingName = R.string.topping_pepperoni,
        pizzaOverlayImage = R.drawable.topping_pepperoni
    ),

    Pineapple(
        toppingName = R.string.topping_pineapple,
        pizzaOverlayImage = R.drawable.topping_pineapple
    )
}
```

You are now ready to add toppings to your pizza previews. Ultimately, your layout preview will show toppings matching what you specified in **`PizzaHeroImagePreview`**: pepperoni on the left half, pineapple on the whole pizza, and basil on the right half (Figure 4.3).

Figure 4.3 Pizza preview goal

To do this, you will layer more **`Image`** composables – one for each topping on the pizza – on top of the base pizza **`Image`**.

Start by wrapping your base **`Image`** in a **`Box`**. Unlike the **`Column`** and **`Row`** composables, which place their content one after the other, the **`Box`** composable stacks its content. Then, use a `for` loop to add a new **`Image`** for each topping. For now, make each topping appear on the whole pizza.

Listing 4.5 An image for each topping (`PizzaHeroImage.kt`)

```
...
@Composable
fun PizzaHeroImage(
    pizza: Pizza,
    modifier: Modifier = Modifier
) {
    Box(
        modifier = modifier
    ) {
        Image(
            painter = painterResource(R.drawable.pizza_crust),
            contentDescription = stringResource(R.string.pizza_preview),
            modifier = modifier
        )

        pizza.toppings.forEach { (topping, placement) ->
            Image(
                painter = painterResource(topping.pizzaOverlayImage),
                contentDescription = null,
                modifier = Modifier.focusable(false)
            )
        }
    }
}
```

For your new **`Image`** composables, you use the **`focusable`** modifier to disable focus. This tells screen readers to ignore your topping images so that they do not see the pizza preview as multiple components. Because you disable focus, there is no need to specify a `contentDescription`.

Refresh your previews. You should see pepperoni, pineapple, and basil, all on the whole pizza, centered on top of the cheese. So far, so good.

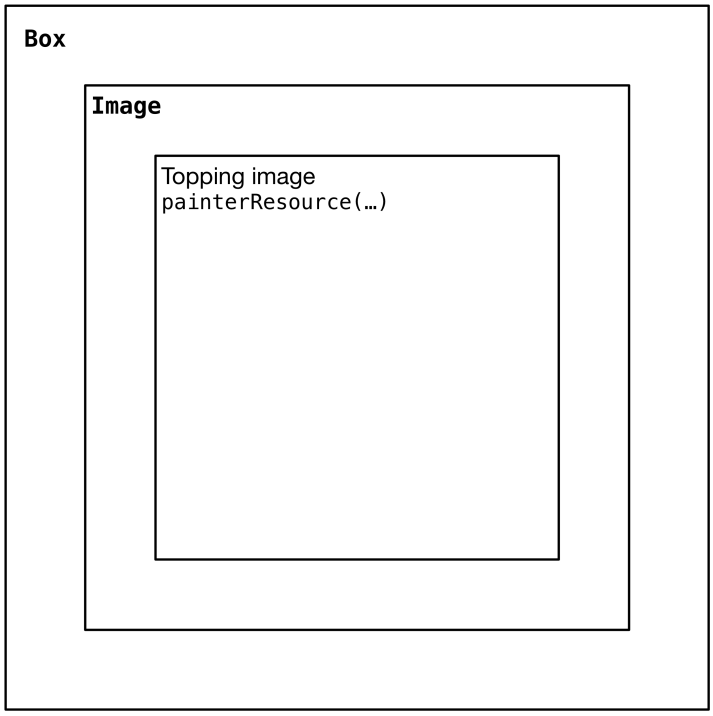
Customizing the Image composable

Your topping **`Image`**s are all contained in the same **`Box`**. Because they are the same size, they are stacked on top of one another, resulting in what appears to be a single pizza image with the desired toppings. This setup works perfectly if your toppings are all placed on the entire pizza – but Coda Pizza does not limit users to full-pizza toppings. Your next task is to show only half of a topping layer when the topping is on half of the pizza.

Before you tackle this task, think about the structure of your topping images: Each image that is presented onscreen is drawn by a **`Painter`** and is hosted in an **`Image`** composable. The **`Image`** has its own bounds, as does the **`Painter`**. And, remember, the **`Painter`** *is* the topping image you display in your UI.

Figure 4.4 shows the relationship between your topping images and your **`Image`** composables. Keep this in mind as you tackle Coda Pizza's next feature, which will require customizing how your **`Image`** composables display their images.

Figure 4.4 Layers of pizza



Although the topping image's bounds are currently the same as its **`Image`** container's, this will need to change soon.

With the theory out of the way, it is time to make your previews accurately show toppings that are on only half of the pizza. At a high level, this will require four steps:

- Set the **`Image`** composable's size to the full height of the pizza and half of its width.
- Crop the image of your topping inside the **`Image`**'s bounds so that only half of the topping layer is visible.
- Align the topping image in the bounds of the crop to ensure that the correct half of the topping is visible.
- Arrange the **`Image`** composable so that it appears on the correct half of the pizza.

aspectRatio

Start by setting the size of the topping **`Image`**. By default, an **`Image`**'s size is determined by the image being displayed. If you want to display an **`Image`** at a different size, you can use a modifier to alter its size. There are several modifiers that will do this, including the **`size`** modifier to set an exact size for the image.

But the size of the pizza preview will be dynamic, as it will fit the width of the user's device. So you do not want to hardcode the size of the image. Instead, you need to set the size of each topping layer relative to the size of its container. One way to accomplish this is with the help of aspect ratios.

Aspect ratio compares a rectangle's width to its height. The pizza preview has an aspect ratio of 1:1 – it is a perfect square, as wide as it is tall. Toppings that appear on half of the pizza will have the same height as the preview, but half of the width. This means that a topping image's aspect ratio should be 1:2 when it is on half of the pizza – it will be twice as tall as it is wide.

To set a composable's aspect ratio, use the **`aspectRatio`** modifier. If a topping is on half of the pizza, set its aspect ratio to 1:2. Aspect ratios are passed as floating point values instead of ratios, so the 1:2 aspect ratio is specified as `0.5`. For toppings placed on the entire pizza, set the aspect ratio to 1:1 by passing in `1.0`. To ensure your base pizza is always a perfect square, set its aspect ratio to 1:1 as well. Finally, make sure that the pizza crust always fills the full bounds of the **`PizzaHeroImage`** by adding the **`fillMaxSize`** modifier.

Listing 4.6 Setting aspect ratios (`PizzaHeroImage.kt`)

```
...
@Composable
fun PizzaHeroImage(
    pizza: Pizza,
    modifier: Modifier = Modifier
) {
    Box(
        modifier = modifier
            .aspectRatio(1f)
    ) {
        Image(
            painter = painterResource(R.drawable.pizza_crust),
            contentDescription = stringResource(R.string.pizza_preview),
            modifier = Modifier.fillMaxSize()
        )

        pizza.toppings.forEach { (topping, placement) ->
            Image(
                painter = painterResource(topping.pizzaOverlayImage),
                contentDescription = null,
                modifier = Modifier.focusable(false)
                    .aspectRatio(when (placement) {
                        Left, Right -> 0.5f
                        All -> 1.0f
                    })
            )
        }
    }
}
```

(To use your **`ToppingAlignment`** enum values without the `ToppingAlignment.` prefix, add an import for `com.bignerdranch.android.codapizza.model.ToppingPlacement.*` at the top of the file.)

Build your project to get an updated preview for PizzaHeroImagePreview. You will see that the pepperoni and basil toppings now appear at half of their original size, vertically centered and on the left half of the pizza (Figure 4.5).

Figure 4.5 Small toppings

Although the pepperoni and basil **`Image`** composables are, correctly, half their original size, they are scaling down their contents so that the entire image can be displayed. You will fix that next.

contentScale

When the image you are displaying and the container that holds it do not have matching aspect ratios, Compose needs some strategy to handle the discrepancy. Here, the pepperoni and basil **`Image`**s have a ratio of 1:2, but the **`Box`** that contains them has a ratio of 1:1.

To tell Compose how to handle this difference, you set the **`Image`**'s `contentScale`. The default content scale is `Fit`, which scales the entire content image to fit the **`Image`**'s bounds, while preserving its original aspect ratio. This is not what you want, as you need to show the left or right half of the topping's image.

To change this behavior, you can add a `contentScale` argument when calling **`Image`**. Specifying the `Crop` behavior will scale the image to fit the bounds of the **`Image`**, cropping any excess that extends beyond the composable's bounds.

Listing 4.7 Specifying a content scale (`PizzaHeroImage.kt`)

```
...
@Composable
fun PizzaHeroImage(
    pizza: Pizza,
    modifier: Modifier = Modifier
) {
    Box(
        ...
    ) {
        ...
        pizza.toppings.forEach { (topping, placement) ->
            Image(
                painter = painterResource(topping.pizzaOverlayImage),
                contentDescription = null,
                contentScale = ContentScale.Crop,
                modifier = Modifier.focusable(false)
                    ...
            )
        }
    }
}
```

Refresh the preview of PizzaHeroImagePreview. Now, the pepperoni and basil fill the height of the pizza preview – but they also extend off the edge of the pizza. Although your topping images are being cropped, you are not seeing their left halves – you are seeing their centers (Figure 4.6). This is because the images themselves (the **`Painter`**s) are centered within the bounds of their **`Image`** composables.

Figure 4.6 Cropped toppings

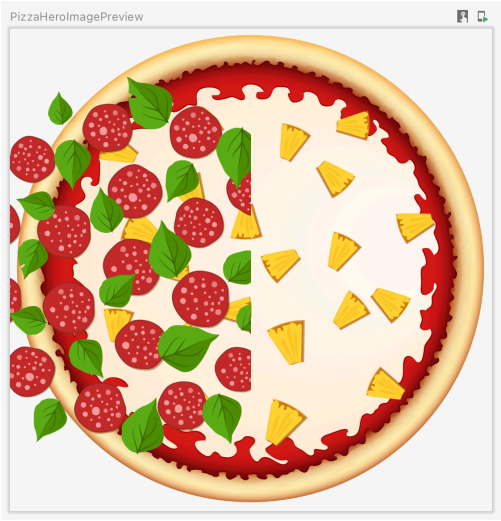


Image alignment

To change which portion of the image is shown and which portion gets cropped, you can set the `alignment` property on your **`Image`**. When a topping is placed on the left half of the pizza, you want to align the topping image's left edge with the left edge of the **`Image`**. You can do this by setting the **`Image`**'s alignment to `TopStart`, which aligns the top-left corner of its content with the top-left corner of the composable itself. Similarly, if a topping is on the right half of the pizza, you can use the `TopEnd` alignment to show the right half of the topping. If the topping is on both sides of the pizza, you can use the default `Center` alignment.

Listing 4.8 Aligning the image (`PizzaHeroImage.kt`)

```
...
@Composable
fun PizzaHeroImage(
    pizza: Pizza,
    modifier: Modifier = Modifier
) {
    Box(
        ...
    ) {
        ...
        pizza.toppings.forEach { (topping, placement) ->
            Image(
                painter = painterResource(topping.pizzaOverlayImage),
                contentDescription = null,
                contentScale = ContentScale.Crop,
                alignment = when (placement) {
                    Left -> Alignment.TopStart
                    Right -> Alignment.TopEnd
                    All -> Alignment.Center
                },
                modifier = Modifier.focusable(false)
                    ...
            )
        }
    }
}
```

Refresh the preview once again. The pepperoni is now properly placed on the pizza: It covers the left half of the pizza, without spilling over. But the basil is still misplaced. Although it is the correct size and shape to fill the right half, it is positioned over the left half of the pizza, spilling off the left edge of the crust (Figure 4.7).

Figure 4.7 Partially aligned toppings

The align modifier

To place toppings on the right half of the pizza preview, you need to make one last change to your **`PizzaHeroImage`**. By setting the `alignment` parameter, you specified where you wanted the topping image to be painted inside the bounds of the **`Image`** composable. But the **`Image`** composable itself is always aligned to the top-left corner of its container, the **`Box`**.

To position the **`Image`** composable, you will need another tool: the **`align`** modifier. This modifier can be used to align the composable children of a **`Box`**. Much like the **`weight`** modifier you have used before, the **`align`** modifier is contextually available only when your content appears in a **`Box`**.

Set the alignment of your topping **`Image`** so that toppings on the right half of the pizza are aligned to the right edge of the **`Box`**, using the `CenterEnd` alignment. This will causes the image to appear at the end (right) of the **`Box`** and vertically centered. Although the toppings are already aligned correctly when placed on the left half or entire pizza, specify alignments for those cases as well – `CenterStart` and `Center`, respectively. This will allow you to build your modifier with a single fluent chain of function calls.

Listing 4.9 Aligning the toppings (`PizzaHeroImage.kt`)

```
...
@Composable
fun PizzaHeroImage(
    pizza: Pizza,
    modifier: Modifier = Modifier
) {
    Box(
        ...
    ) {
        ...
        pizza.toppings.forEach { (topping, placement) ->
            Image(
                painter = painterResource(topping.pizzaOverlayImage),
                contentDescription = null,
                contentScale = ContentScale.Crop,
                alignment = when (placement) {
                    Left -> Alignment.TopStart
                    Right -> Alignment.TopEnd
                    All -> Alignment.Center
                },
                modifier = Modifier.focusable(false)
                    .aspectRatio(when (placement) {
                        Left, Right -> 0.5f
                        All -> 1.0f
                    })
                    .align(when (placement) {
                        Left -> Alignment.CenterStart
                        Right -> Alignment.CenterEnd
                        All -> Alignment.Center
                    })
            )
        }
    }
}
```

Refresh your preview again. At last, the toppings on your pizza are accurately placed in the preview. You should see pineapple on the entire pizza, pepperoni on the left half, and basil on the right half. All the toppings should be correctly sized and vertically centered, and nothing should be outside of the crust (Figure 4.8).

Figure 4.8 The final preview

Adding a header to LazyColumn

Your **`PizzaHeroImage`** is now complete, but you will not yet see the fruits of your labor if you run Coda Pizza. Allow your efforts to pay off by adding **`PizzaHeroImage`** as an item in your **`ToppingsList`**'s **`LazyColumn`**.

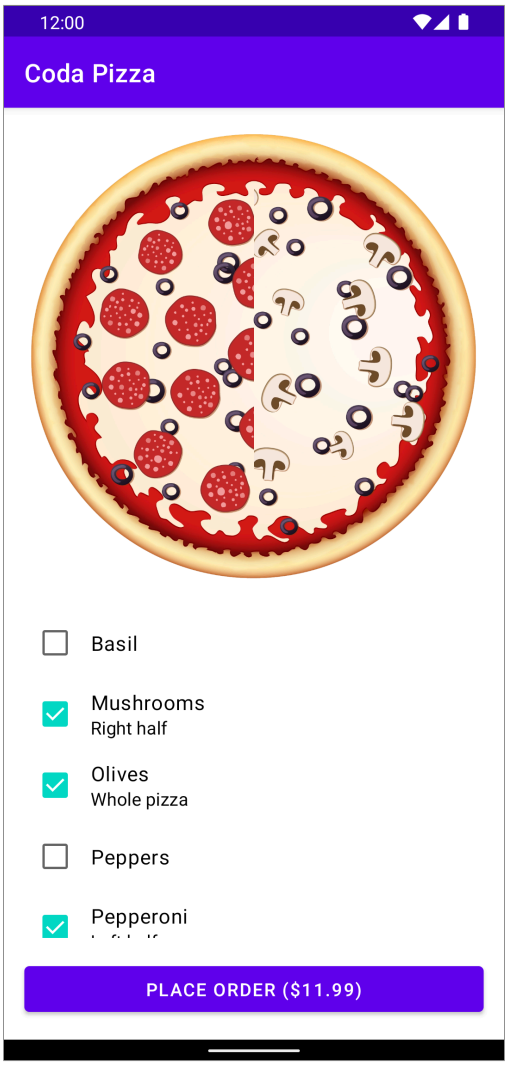
Listing 4.10 Adding more items to a LazyColumn (`PizzaBuilderScreen.kt`)

```
...
@Composable
private fun ToppingsList(
    pizza: Pizza,
    onEditPizza: (Pizza) -> Unit,
    modifier: Modifier = Modifier
) {
    ...
    LazyColumn(
        modifier = modifier
    ) {
        item {
            PizzaHeroImage(
                pizza = pizza,
                modifier = Modifier.padding(16.dp)
            )
        }

        items(Topping.values()) { topping ->
            ToppingCell(
                topping = topping,
                placement = pizza.toppings[topping],
                onClickTopping = {
                    toppingBeingAdded = topping
                }
            )
        }
    }
}
...
```

Run Coda Pizza. You will now see the pizza's preview proudly presented above your list of toppings. Add any toppings you want to the pizza. Because you have already tracked your **`Pizza`** object using the **`State`** class, your preview will automatically update, with no additional effort on your part (Figure 4.9). As you scroll through the list of toppings, the pizza preview will scroll with the other content.

Figure 4.9 Pizza preview in action



MaterialTheme

With your pizza preview in place, it is time to add a fresh coat of paint to Coda Pizza. Currently, Coda Pizza is using the default theme, but you can add your own customizations to specify your application's colors, typographic styles, and shapes for various components like **`Button`**s and **`Card`**s. For Coda Pizza, you will stick to changing your app's colors, although the steps are similar for other customizations.

With the framework views, the themes that you are familiar with were defined in XML. But Compose has its own theming system that does not leverage XML styles or the theming system used by framework views.

Themes are stored in an object called `MaterialTheme`, which you used in Chapter 1 to set **`Text`** styles. Currently, you are using a default theme, which is where your application colors are coming from. Let's change that.

To change the values in your `MaterialTheme` object, you will use the **`MaterialTheme`** composable function. This function accepts parameters to change your theme's colors, typographic styles, and component shapes. The **`MaterialTheme`** composable also accepts a lambda expression, which is where your content will be placed. Any theme configuration you specify only affects content placed inside this lambda. This means that your theme should be specified very early in your composition.

Create a new file in your `ui` package called `AppTheme.kt`. This is where all of your theme information will be stored. In this file, create a new composable called **`AppTheme`**. This function will call **`MaterialTheme`**, passing all the theme attributes you want to use to customize Coda Pizza's appearance.

Listing 4.11 Declaring a theme (`AppTheme.kt`)

```
@Composable
fun AppTheme(
    content: @Composable () -> Unit
) = MaterialTheme(
    colors = lightColors(
        primary = Color(0xFFB72A33),
        primaryVariant = Color(0xFFA6262E),
        secondary = Color(0xFF03C4DD),
        secondaryVariant = Color(0xFF03B2C9),
    )
) {
    content()
}
```

You set the `colors` property of your theme to be a palette with a light background and a few specific colors for Coda Pizza. There are several other colors you can specify, but you will rely on the defaults provided by **`lightColors`**. You also did not provide other styling information for your app's typography, so the defaults will be used.

For your theme to be used, you must wrap your application content in an **`AppTheme`** composable. Do so in **`MainActivity`**, right inside the **`setContent`** call, to ensure that the theme is applied to your entire application.

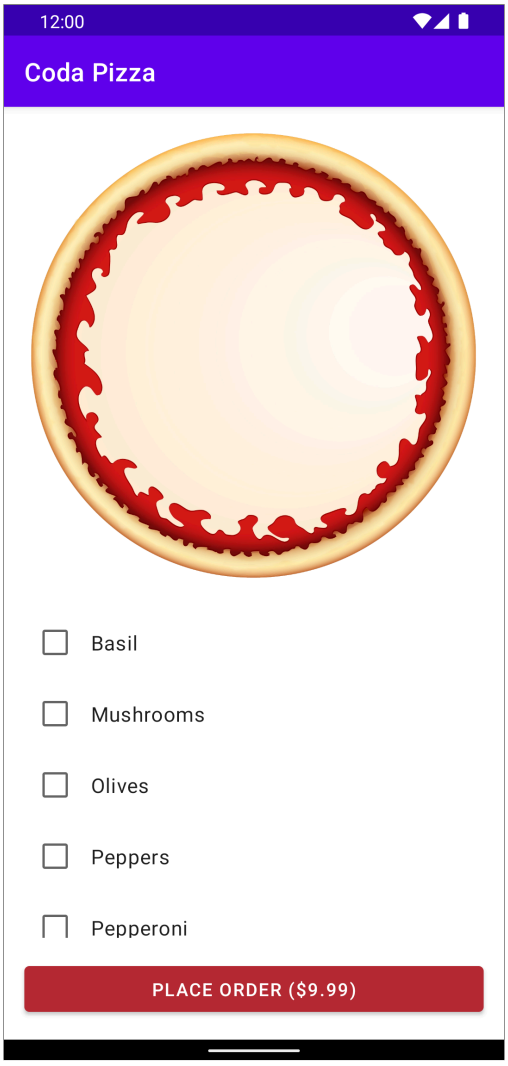
Listing 4.12 Applying a theme (`MainActivity.kt`)

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            AppTheme {
                PizzaBuilderScreen()
            }
        }
    }
}
```

With that change, many UI elements in Compose are made aware of your theme. But you can also manually access these values in your composition through the `MaterialTheme` object. For example, to access your theme's primary color, you can call `MaterialTheme.colors.primary`.

Run Coda Pizza. Your PLACE ORDER button is now red, matching the value specified in your theme, but the app bar and status bar are still stubbornly purple, as shown in Figure 4.10. (We promise, the app bar and the button are different colors!)

Figure 4.10 A partially themed Coda Pizza



What gives?

Your activities are automatically given an app bar when they extend from **`AppCompatActivity`** and use a style with a built-in app bar. And your **`MainActivity`** is doing just that. And because this app bar is provided as a framework view, it is unaware of any themes you created in your Compose code.

Although you could manually keep your Compose themes and framework themes in sync with one another, it would be better if your Compose theme were the source of truth for your application. Luckily, you can take the app bar into your own hands and render it using a composable. But before you can make this change, you need to remove the built-in app bar.

Speaking of removing things from your theme, there are a number of things in your project that you no longer need now that Compose is managing your styles. First, because your app's theme is defined in Compose, you do not need to specify theme attributes to define the same customizations. There are a few circumstances where you will still need to edit themes, even in 100% Compose apps – such as if you need to customize the color of system bars or set a custom splash screen for your app. But these issues will not come up for Coda Pizza.

Also, because your colors are defined entirely in your Compose code, Coda Pizza will not need its `colors.xml` file. Last, you do not need the Material Components library, since it only provides styling for framework views.

Start by tidying up your application theme. The project template you used to create Coda Pizza includes themes for both light mode and night mode. Jetpack Compose is in complete control of your application theme, so you do not need to provide a separate night mode theme for your application. Remove this theme variation by deleting the `res/values/themes/themes.xml` file labeled (night) in Android Studio.

Next, you need to remove the default app bar from your activity. To do this, you can change your theme to use a theme with the `NoActionBar` suffix. Use the AppCompat-provided `Theme.AppCompat` theme to remove your dependence on the Material Components library. At the same time, remove all the style declarations from your theme, which are unnecessary because these values are now set in your Compose theme.

Listing 4.13 Removing framework styles (`themes.xml`)

```
<resources xmlns:tools="http://schemas.android.com/tools">
    <!-- Base application theme. -->
    <style name="Theme.CodaPizza"
        parent="Theme.MaterialComponents.DayNight.DarkActionBar">
    <style name="Theme.CodaPizza" parent="Theme.AppCompat.Light.NoActionBar">
        <!-- Primary brand color. -->
        <item name="colorPrimary">@color/purple_500</item>
        <item name="colorPrimaryVariant">@color/purple_700</item>
        <item name="colorOnPrimary">@color/white</item>
        <!-- Secondary brand color. -->
        <item name="colorSecondary">@color/teal_200</item>
        <item name="colorSecondaryVariant">@color/teal_700</item>
        <item name="colorOnSecondary">@color/black</item>
        <!-- Status bar color. -->
        <item name="android:statusBarColor" tools:targetApi="l">
            ?attr/colorPrimaryVariant</item>
        <!-- Customize your theme here. -->
    </style>
</resources>
```

Next, delete your `colors.xml` resource file. You will not need to access these colors, and you removed the only reference to them when you deleted your theme attributes.

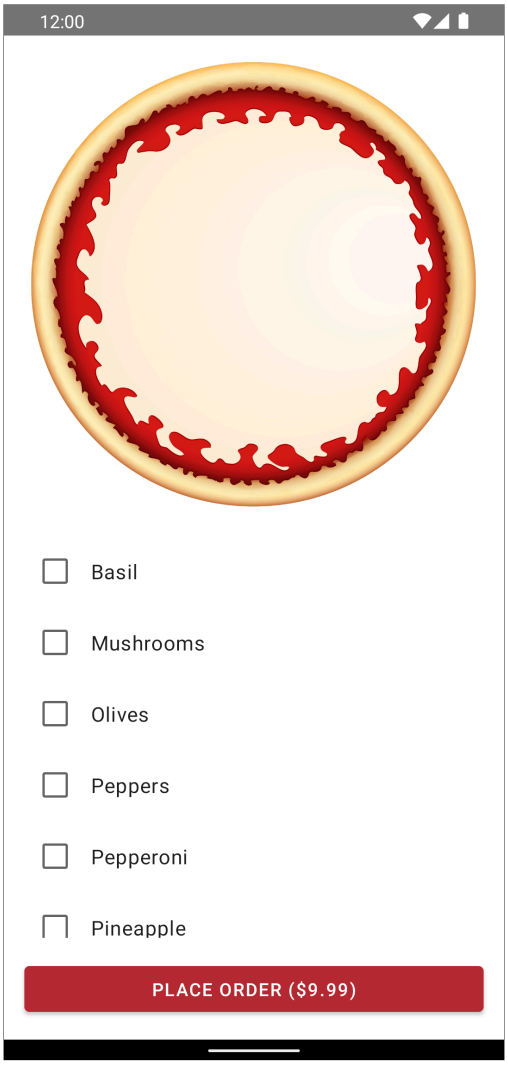
You no longer reference the Material Components library, so now you can remove it from your project. Removing unused dependencies reduces your application size, gets rid of unnecessary classes that clutter your IDE's autocompletion suggestions, and can improve compilation performance. Delete this dependency by taking a trip to your `app/build.gradle` file.

Listing 4.14 Removing Material Components (`app/build.gradle`)

```
...
dependencies {
    implementation 'androidx.core:core-ktx:1.8.0'
    implementation 'androidx.appcompat:appcompat:1.5.0'
    implementation 'com.google.android.material:material:1.6.1'
    implementation 'androidx.constraintlayout:constraintlayout:2.1.4'
    ...
}
```

Remember to perform a Gradle sync after making these changes. When the sync completes, run Coda Pizza. The app bar has disappeared, and your composable content is now the only content that appears onscreen (Figure 4.11).

Figure 4.11 A full-screen composable



Scaffold and TopAppBar

App bars are an important part of your app, both visually and for navigation and menus. It is a good idea for Coda Pizza to include an app bar, even if it is just to show a title.

To reinstate your app bar, you will use the **`TopAppBar`** composable. **`TopAppBar`** accepts several composables as inputs, each of which can add content in a specific region of the app bar. You will only set the `title` parameter to show the application name, matching the default behavior of the automatically provided app bar – but with the benefit of using your Compose theme.

If you wanted, you could also provide a parameter for a `navigationIcon` to add an element to be the Up button. There is also a parameter called `actions`, which takes on the role of adding items to the right side of the app bar – like what you accomplished with a menu in the framework-provided app bar.

This pattern of accepting several composable lambdas as inputs is called *slotting*, as each of the components slots into a specific area of the app bar. Slotting makes composables much more flexible about what they can display than most framework views. You can pass any composable into any slot, which gives you complete control over what appears in each slot.

In the framework UI toolkit, you were limited to displaying a string as the title. And when you add a **`TopAppBar`**, you will typically pass a **`Text`** in for the `title` slot. But because the `title` argument takes in a composable, you have the

power to use elements like an image, loading spinner, drop-down menu, or checkbox within that space – to name a few examples.

Take the **`TopAppBar`** composable for a spin, placing it at the top of your **`PizzaBuilderScreen`** composable.

Listing 4.15 Adding an app bar (`PizzaBuilderScreen.kt`)

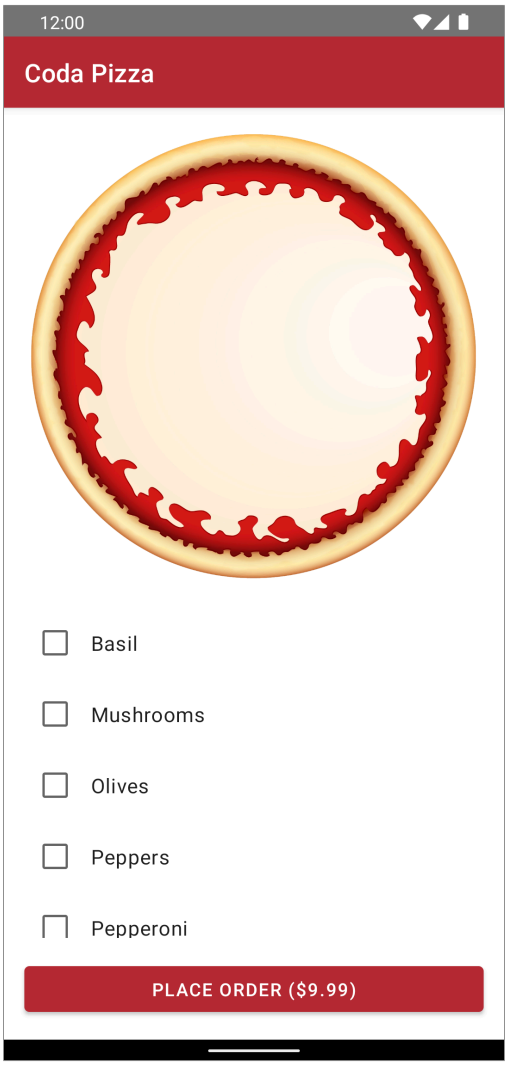
```
@Preview
@Composable
fun PizzaBuilderScreen(
    modifier: Modifier = Modifier
) {
    var pizza by rememberSaveable { mutableStateOf(Pizza()) }

    Column(
        modifier = modifier
    ) {
        TopAppBar(
            title = { Text(stringResource(R.string.app_name)) }
        )

        ToppingsList(
            ...
        )
        ...
    }
}
...
```

Run Coda Pizza. An app bar is back at the top of the screen – much like the one you previously had in Coda Pizza and the ones you have seen in the other apps you have built. But this time, the app bar is red, matching the color you set in your **`AppTheme`** composable (Figure 4.12).

Figure 4.12 TopAppBar and AppTheme in action



Your app bar is now functional, which is good. What is not so good is that **`PizzaBuilderScreen`**'s **`Column`** now has several elements in it. As your application grows in complexity, it can become harder to work with these large building blocks for your application's UI. Take the **`TopAppBar`**, for example. It *must* come first in this column if it is to be drawn at the top of the screen. If you accidentally add another composable before it, your app bar will appear toward the middle of the screen.

You can make the components in your UI easier to manage using another composable called **`Scaffold`**. **`Scaffold`** is designed to help lay out your application – it effectively acts as a skeleton for your app's layout. It uses the slotting pattern to define regions of your application where content can be labeled and consistently placed. There are two main slots you are interested in: the `topBar` slot and the `content` slot.

The `topBar` slot is designed for components like your **`TopAppBar`**. Composables placed in this slot always appear at the top of the screen, above your main content. The `content` slot, meanwhile, is for your app's primary content. There are other slots for elements like bottom bars and snackbars, each of which will always appear appropriately around your `content`.

Update **`PizzaBuilderScreen`** to use a **`Scaffold`**. You will still use a **`Column`** to lay out your toppings list and order button, but the **`Scaffold`** will separate the app bar from the content.

Listing 4.16 Using a scaffold (`PizzaBuilderScreen.kt`)

```
@Preview
@Composable
fun PizzaBuilderScreen(
    modifier: Modifier = Modifier
) {
    var pizza by rememberSaveable { mutableStateOf(Pizza()) }

    Column(
        modifier = modifier
    ) {
    Scaffold(
        modifier = modifier,
        topBar = {
            TopAppBar(
                title = { Text(stringResource(R.string.app_name)) }
            )
        },
        content = {
            Column {
                TopAppBar(
                    title = { Text(stringResource(R.string.app_name)) }
                )

                ToppingsList(
                    ...
                )

                OrderButton(
                    ...
                )
            }
        }
    )
}
```

Run Coda Pizza again. The app will look and behave the same, but your **`TopAppBar`** now has a guaranteed, designated space. **`Scaffold`** follows its own blueprints to position the content of its slots, so it does not matter how you specify the content of a slot or what order you put them in. The argument you provide for `topBar` appears at the top of the screen – always.

CompositionLocal

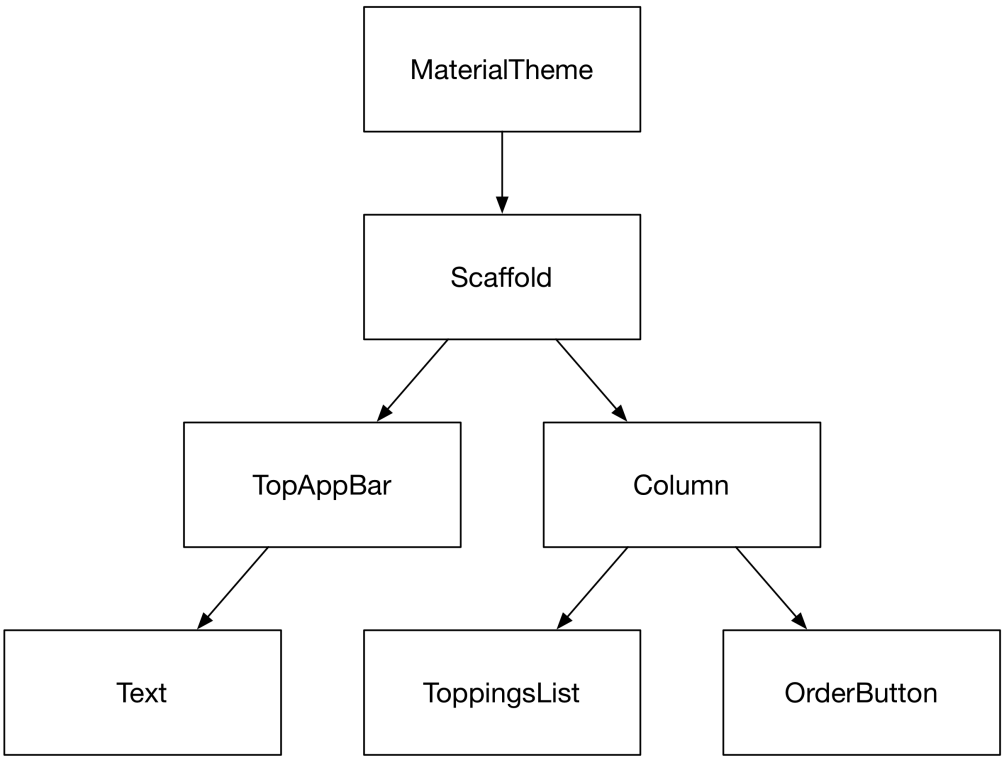
In this chapter, you have seen a few examples where nesting one composable inside another causes the inner composable's appearance to change. Using the **`MaterialTheme`** composable, for example, causes all components nested within it to be aware of your theme and use its colors.

When you added the **`Text`** to your **`TopAppBar`**, you might have noticed something interesting. Despite the fact that you did not specify any parameters besides the `text` itself, the **`Text`** was appropriately styled with the correct font size and color to appear in your app bar. How did this happen?

When you build a UI using Jetpack Compose, a composition hierarchy is created at runtime for all your composables – much like the view hierarchy for framework UIs. Whenever you call a composable function to add a UI element to your app, a corresponding node is added to this composition hierarchy. You cannot get a reference to this hierarchy or its nodes, but they are still there to organize your composables. (Whereas **`View`** objects are themselves the nodes in a framework UI's hierarchy, and you can get direct access to them whenever you like.)

For Coda Pizza, the top of the composition hierarchy looks something like Figure 4.13:

Figure 4.13 Coda Pizza's UI hierarchy



And **`ToppingsList`** and **`OrderButton`** have their own children, down to the **`Text`**s and **`Checkbox`** seen onscreen.

Experienced Android developers probably know that nesting view hierarchies too deeply could degrade an app's performance. But although Coda Pizza's UI hierarchy is many layers deep, fear not. Jetpack Compose is substantially more efficient than Android's framework UI toolkit when it comes to managing, laying out, and drawing UI elements. Deeply nested layouts are not a performance concern with Compose in the way they were for the other apps you built – which is also why you did not reach for a tool like **`ConstraintLayout`** in Coda Pizza.

The top of this hierarchy is your **`MaterialTheme`** composable. As you have seen in this chapter, using the **`MaterialTheme`** composable sets the values that will be returned by the `MaterialTheme` object. And these values have a scope.

When you call the **`MaterialTheme`** composable, the theme values are stored and made accessible for all its children. Whenever a component needs to access a theme attribute, you use the `MaterialTheme` object and access the colors, shapes, or typography with code like `MaterialTheme.colors.primary`. Values referenced in this way are tracked by instances of a class called **`CompositionLocal`** so that every composable that is a child of the **`MaterialTheme`** node can access your theme information.

You placed the **`MaterialTheme`** composable at the root of your composition because you want the theme to affect everything you display in your composition. If you added a composable as a sibling to the **`MaterialTheme`**, it would be unaware of the themes you set elsewhere in the UI hierarchy and would use the default material theme.

If for some reason you want to have different themes for different parts of your composition, you can also nest one **`MaterialTheme`** composable inside another. The inner theme will override the theme values from the outer **`MaterialTheme`** composable, but only for the children of the inner **`MaterialTheme`**.

Back to the question of where the style of the **`Text`** in your **`TopAppBar`** is coming from. In addition to the theme attributes set in the `themes.xml` resource file and **`AppTheme`** composable (if you have one), some composables also specify preferred theme attributes that Compose will take into account. (These are officially termed *current* theme attributes, but we find that term unnecessarily confusing.)

For example, while your application has text color specifications, the **`TextButton`** composable you are using for the topping placement options in your dialog overrides this color specification, setting a style that works with the overall theme but is specific to its own environment. When the **`Text`** child of your **`TextButton`**s asks for a text color, it gets this overridden value, not the value from your theme.

`TopAppBar` does the same thing to its text, setting a color that works with the background color set by the app theme. In this way, "current" theme attributes like the ones set by **`TextButton`** and **`TopAppBar`** provide automatic localized theming that coordinates with the app's overall theme. The **`Text`** composable determines its styles by first looking at the text size, color, and so on set by its parent (or another direct ancestor) and then falling back to the theme for any styles that are not set.

Behind the scenes, these behaviors are all driven by the same **`CompositionLocal`** class we mentioned earlier.

`CompositionLocal`s are variables that are defined for a part of your composition hierarchy. When a `CompositionLocal` is defined, it is accessible to all of its composable children – and can be overridden deeper in the hierarchy if the same `CompositionLocal` is set again. Theme information is often defined this way – many children need to share and access theme information, and **`CompositionLocal`**s make that sharing easy.

Instances of **`CompositionLocal`** propagate theme information automatically. But you can also access `CompositionLocal` variables yourself to get many more values and resources associated with your composition. To see this in action, it is time to implement one more feature in Coda Pizza: the PLACE ORDER button.

Unfortunately, Coda Pizza will not result in a pizza being delivered to your address. But it can present you with a **`Toast`** (which, arguably, has many similarities to pizza). To set one up, you will replace your final TODO, which is lingering in **`OrderButton`**'s `onClick` callback.

First, prepare Coda Pizza to display a toast by adding a string resource for the message that will appear when the order is placed.

Listing 4.17 A consolation toast's message (`strings.xml`)

```
<resources>
  <string name="app_name">Coda Pizza</string>

  <string name="place_order_button">Place Order (%1$s)</string>
  <string name="order_placed_toast">Order submitted!</string>
  ...
</resources>
```

To show the toast, you need to obtain a **`Context`**. You could accomplish this by adding a `context` parameter to **`OrderButton`** and passing your activity context all the way down your composition hierarchy, but that would be messy and would not scale well if you needed to access many properties.

Instead, Compose includes a `CompositionLocal` out of the box that stores the context that hosts your composable UI. You can use this `CompositionLocal` to access the context regardless of where you are in the composition.

To read the value of a `CompositionLocal` variable, you first obtain a reference to the corresponding **`CompositionLocal`** class itself. Then you can get the value of the variable for the current position in the composition hierarchy via its `current` property. The convention for naming a `CompositionLocal` is to use the prefix "Local" followed by the name or type of the variable being provided. So the composition's local context is stored in `LocalContext`.

Using the `LocalContext` property, obtain a **`Context`**. Then, implement your **`OrderButton`**'s `onClick` lambda to show a toast. Because `CompositionLocal`s give you the *current* value of the variable, they can only be read inside the

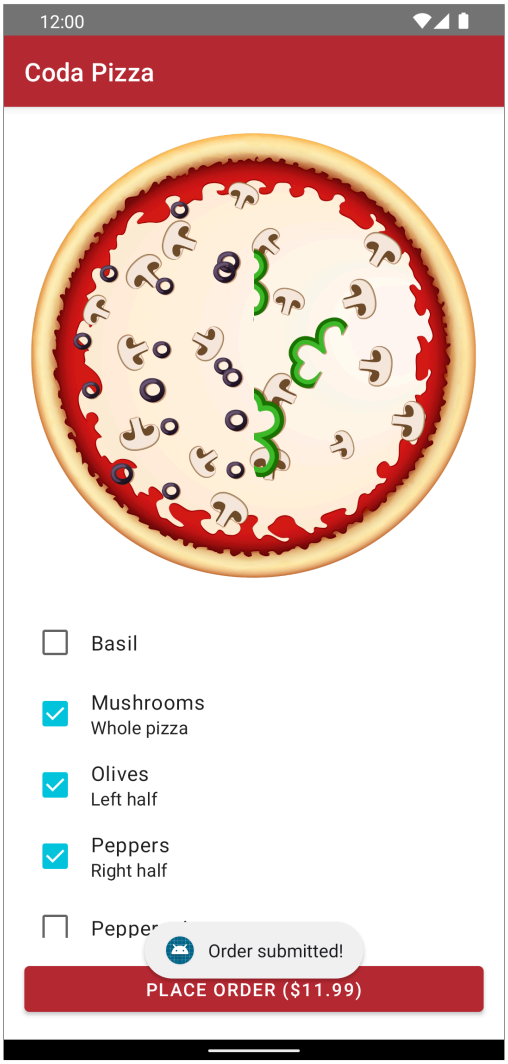
composition itself. This means that you must obtain the context outside the click listener, because your click listener cannot access the composition hierarchy.

Listing 4.18 Using a **`Context`** inside a composable (`PizzaBuilderScreen.kt`)

```
...
@Composable
private fun OrderButton(
    pizza: Pizza,
    modifier: Modifier = Modifier
) {
    val context = LocalContext.current
    Button(
        modifier = modifier,
        onClick = {
            // TODO
            Toast.makeText(context, R.string.order_placed_toast, Toast.LENGTH_LONG)
                .show()
        }
    ) {
        val currencyFormatter = remember { NumberFormat.getCurrencyInstance() }
        val price = currencyFormatter.format(pizza.price)
        Text(text = stringResource(R.string.place_order_button, price))
    }
}
```

Run Coda Pizza and press the PLACE ORDER button. You should see a toast with the message Order submitted! near the bottom of the screen (Figure 4.14).

Figure 4.14 A toast to pizza



`CompositionLocals` are a convenient way to get more information about your composition. Many `CompositionLocals` are predefined and readily available should you need them. Some of these values, like your theme, allow you to easily customize portions of your UI hierarchy. Other `CompositionLocals`, meanwhile, have values that do not change in the composition hierarchy and can give you information about the composition itself.

Using the built-in `CompositionLocals`, you can access values including the **`Lifecycle`** of the component hosting your composition, the clipboard, and the size of the display. And because `CompositionLocals` are tracked by Compose itself, you can access all these values without declaring new parameters. This flexibility makes `CompositionLocals` a great choice for storing information you need to access sporadically throughout your UI.

You can also define your own `CompositionLocals`, if you need to. This is not something you will do for Coda Pizza, but if you want to know more about this process, take a look at the section called “For the More Curious: Creating Your Own CompositionLocals” near the end of this chapter.

Removing AppCompat

Coda Pizza is now fully operational, but there is one more change you can make to look toward the future. Almost every single Android app ever created has relied on several Jetpack libraries, the most fundamental being the AppCompat library.

AppCompat back-ports many important UI behaviors to ensure consistency across versions of Android. It acts as the building block for many other Jetpack libraries, including ConstraintLayout and the Material Design Components. It also brings along many other tools you have used, including **`Fragment`**s.

Despite the importance of these components in the other apps you have built, these dependencies are designed for the world of the framework UI toolkit. Compose does not depend on AppCompat; it reinvents so many APIs that AppCompat does not provide the same value as it does for apps with framework views. In fact, AppCompat arguably does not provide *any* value if your UI exclusively uses Compose.

But AppCompat is still present in your application, increasing its size and adding more dependencies to download when your project builds. You can reclaim these resources and part ways with the framework views entirely by removing AppCompat from your project.

But do not jump straight to your `build.gradle` file to delete the dependency. There are still a few references to AppCompat you must remove first.

The first reference to AppCompat that you need to remove is in your **`MainActivity`**. Your **`MainActivity`** extends from **`AppCompatActivity`**, following the recommendation for all apps using the framework UI toolkit. In addition to back-porting behaviors to older versions of Android, **`AppCompatActivity`** also provides hooks that other Jetpack libraries, like ViewModel, require.

If you replace **`AppCompatActivity`** with the platform-provided **`Activity`** class, you will lose the ability to use several other Jetpack libraries, which is not ideal. Instead, you can use **`ComponentActivity`**, which exists in the middle ground between the base **`Activity`** class and the full-fledged **`AppCompatActivity`** class.

`ComponentActivity` exists outside AppCompat and provides hooks so that other libraries that need deeper access to your activity, such as the AndroidX Lifecycle library and ViewModel, can do what they need to do. Using **`ComponentActivity`** allows these integrations to continue working, while removing your dependence on the AppCompat library.

To migrate to **`ComponentActivity`**, update your **`MainActivity`** class to change which variation of **`Activity`** it extends from. Also, delete the import statement for **`AppCompatActivity`**.

Listing 4.19 Removing AppCompatActivity (`MainActivity.kt`)

```
~~import androidx.appcompat.app.AppCompatActivity~~
...
class MainActivity : ~~AppCompatActivity()~~ ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            AppTheme {
                PizzaBuilderScreen()
            }
        }
    }
}
```

There is one final usage of AppCompat lingering in your project. It is in the application theme you specify for Coda Pizza. To remove this reference, you will need to change the theme that your application theme, `Theme.CodaPizza`, is based on.

AppCompat themes provide many customizations to the built-in themes provided by the platform to ensure consistency and to bring new features to the views you can use. But these benefits only apply to framework views, which are nowhere to be found in Coda Pizza.

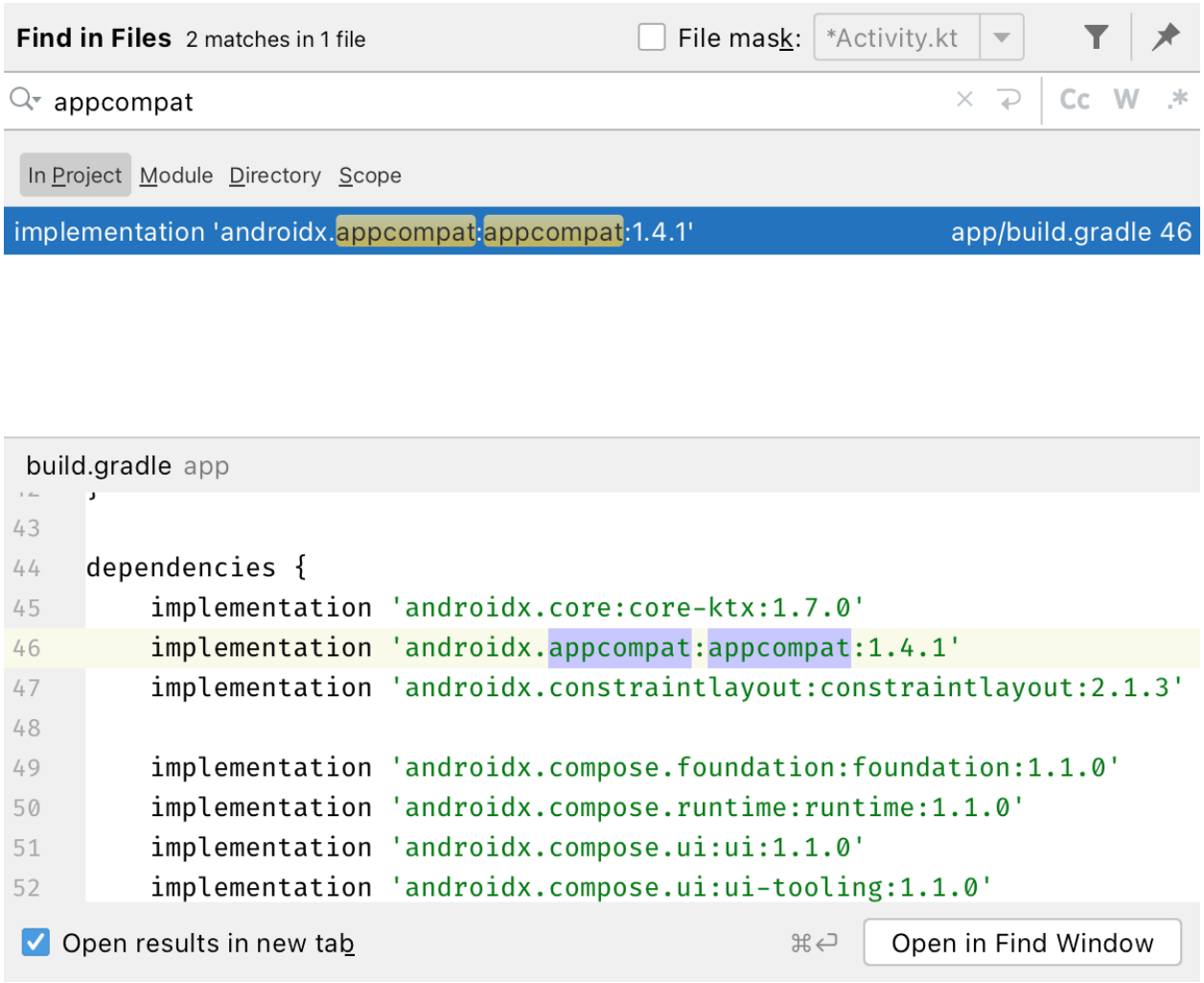
Because these customizations are unnecessary, you can safely remove the reference to `Theme.AppCompat` and replace it with a reference to the `Theme.Material` theme that ships with the platform. Although there may be discrepancies in this theme's appearance across versions, nothing in the theme will affect your Compose UI, making the differences negligible.

Listing 4.20 Using a platform theme (`themes.xml`)

```
<resources xmlns:tools="http://schemas.android.com/tools">
    <!-- Base application theme. -->
    <style name="Theme.CodaPizza" parent="Theme.AppCompat.Light.NoActionBar">
    <style name="Theme.CodaPizza" parent="android:Theme.Material.NoActionBar">
    </style>
</resources>
```

At this point, Coda Pizza no longer references anything from the AppCompat library. To confirm this, open the Find in Files dialog by pressing Command-Shift-F (Ctrl-Shift-F). In the dialog, enter the query appcompat to search every file in your project for this term (Figure 4.15).

Figure 4.15 Looking for AppCompat



You will see one hit in your Gradle build file, but none of your Kotlin files should contain references. If you see any Kotlin files in these results, double-check that the code in these files matches the code in this book exactly, with no leftover references to AppCompat. You may also need to delete a few lingering import statements in your project if Android Studio did not automatically remove them.

After cleaning up any rogue references to AppCompat, your last step is to remove the venerable dependency from your project. While you are removing AppCompat's dependency, also remove the dependency for ConstraintLayout, which was automatically added with the blank project template.

Listing 4.21 Saying goodbye to AppCompat (`app/build.gradle`)

```
...
dependencies {
    implementation 'androidx.core:core-ktx:1.8.0'
    implementation 'androidx.appcompat:appcompat:1.5.0'
    implementation 'androidx.constraintlayout:constraintlayout:2.1.4'
    ...
}
```

Sync your changes, then run Coda Pizza one last time. It should behave exactly as it did before – but now, Android Studio can build your project ever so slightly faster, and your application size will be smaller.

Although the benefits of removing AppCompat are somewhat small, this change signifies the start of a major paradigm shift powered by Jetpack Compose. Under Jetpack Compose, many of the tools you have become familiar with are no longer necessary. Components like **`RecyclerView`** and **`Fragment`** can be big sources of complexity in an Android app, but you do not need them in Compose.

We love Jetpack Compose for its ease of use and effectiveness at building UIs in Android apps. You have only scratched the surface of what Jetpack Compose can do, and we encourage you to experiment with it in your own apps. You will almost certainly encounter and create many framework views in your time as an Android developer, because they have been the only way to build UIs for more than a decade. But we expect that you will see fewer and fewer framework UIs over time as Compose brings about a renaissance in the world of Android development.

For the More Curious: Accompanist

Jetpack Compose is stable, but still in its early days. For many apps, the Compose dependencies you used with Coda Pizza offer every component and API you need to build the UI of your dreams. For other apps, though, you might find that the Compose libraries do not *quite* have all the features you need.

Jetpack Compose version 1.1, for example, does not offer a built-in way to change the status bar or navigation bar colors, request permissions from a composable, or make your UI respond to display cutouts and notches, to name a few examples. But fear not: In addition to the mainstream Compose APIs, Google also offers a set of libraries to provide this functionality and more.

Accompanist is a constantly evolving set of libraries that offer functionality that is not yet built into the mainstream Compose dependencies. They offer a quick way for developers to access these features in their Compose apps. The goal for most Accompanist libraries is that they will eventually graduate out of Accompanist and into the official library they are part of.

For example, Accompanist previously included support for loading images with Coil in Compose, but that functionality has since been moved into Coil itself. Developing features this way allows the Compose team to more effectively design and experiment with these APIs.

Because of these evolutions, Accompanist leans toward being an experimental library. Regardless, we encourage you to take a look at Accompanist and see which of its features are useful for your app. These features are ready to go, and – despite the “experimental” designation – ready for use in production apps.

If you do choose to incorporate Accompanist, keep in mind that its APIs are likely to change over time. Features in Accompanist that make their way into the official Compose dependencies will eventually be removed, which will require you to make updates in your application.

It is too early to tell what the future of Compose looks like, but it looks like Accompanist will be a useful breeding ground for supporting a larger set of features within Compose. For more information on Accompanist, including its latest version and which features it can offer, see its official documentation at `google.github.io/accompanist`.

For the More Curious: Creating Your Own CompositionLocals

In this chapter, you learned about several built-in `CompositionLocal`s and used the `LocalContext` to obtain a **`Context`** in your composable. If you want, you can also define your own `CompositionLocal`s.

Declaring a `CompositionLocal` is particularly useful when you want to give your composables access to new values without introducing an additional parameter. This works best when the information being provided applies to many composables and can be shared across an entire section of your composition hierarchy. Much like global variables, `CompositionLocal`s can be dangerous if used haphazardly. If a value should only be available to one composable, we recommend sticking to parameters.

Suppose your application needed to track analytics to see what features your users rely on most. You could create a class called **`AnalyticsManager`** to implement the analytics logging yourself. But many composables would likely need to report analytics, and you do not want to concern yourself with passing instances of **`AnalyticsManager`** through layer after layer of composables. That is where `CompositionLocal`s come in.

Making a `CompositionLocal` is a two-step process. First, you need to define the `CompositionLocal`. Second, you need to set the value of the `CompositionLocal` in your UI hierarchy.

`CompositionLocal`s are defined by creating a public, file-level property of type **`CompositionLocal`** – like, say, a `LocalAnalyticsManager`. This value basically acts as a key to get hold of the corresponding value. You can assign a value for this property using the **`compositionLocalOf`** function.

This function takes in a lambda to provide a default value for the `CompositionLocal`. For many `CompositionLocal`s, including your hypothetical `LocalAnalyticsManager`, there is no default value – a value must always be explicitly set in the composition itself. In these cases, you can simply throw an exception indicating that the `CompositionLocal` was read before it was set.

```
val LocalAnalyticsManager = compositionLocalOf<AnalyticsManager> {
    error("AnalyticsManager not set")
}
```

With the `CompositionLocal` defined, you can then specify its value at runtime. You do this using the **`CompositionLocalProvider`** composable. **`CompositionLocalProvider`** takes in a set of all the `CompositionLocal`s you want to specify, along with a value for each one. When a component requests one of the `CompositionLocal`s in the provider, the value you specify will be returned.

```
@Composable
fun PizzaBuilderScreen(
    analyticsManager: AnalyticsManager,
    modifier: Modifier = Modifier
) {
    CompositionLocalProvider(
        LocalAnalyticsManager provides analyticsManager
    ) {
        Scaffold(
            modifier = modifier,
            ...
        )
    }
}
```

You may be asking, “OK, but how does **`PizzaBuilderScreen`** obtain its `analyticsManager`?” This question has several answers, and in your own code, you will have to decide for yourself how to answer this question. If **`AnalyticsManager`** is easy to create, you may be able to instantiate it directly inside **`PizzaBuilderScreen`**. (If you do this, make sure to **`remember`** it!)

Alternatively, you could create this value elsewhere in your application – possibly as a singleton – and pass it through your composition hierarchy as an argument. Either approach is valid, and it is up to you and your team to decide how dependencies like `analyticsManager` should make their way through your code.

With the `CompositionLocal` and its provider in place, `LocalAnalyticsManager` is ready to be used. To obtain an **`AnalyticsManager`**, you call `LocalAnalyticsManager.current` inside a composable function. The Compose runtime will look at your composition hierarchy to find an appropriate provider for this value. Once a provider is found, the value that was set will be returned by the `CompositionLocal`.

If several providers are found, the closest parent in the hierarchy will be chosen and its value will be used. If no provider is found, the default value of the `CompositionLocal` (specified with the **`compositionLocalOf`**) will be provided.

Storing values this way allows for easy access throughout your composition hierarchy, but we recommend using `CompositionLocal`s sparingly. They are great for accessing more general or widely used dependencies. But you can find yourself getting into trouble if you hold your application state in a `CompositionLocal`, as this makes it difficult to track down exactly where a value is coming from.

Challenge: Animations

Jetpack Compose has many animation APIs to add pizzazz to UIs. You can find the full list at `developer.android.com/jetpack/compose/animation`. The same page also has tips to help you decide which function you should use to achieve a certain type of animation.

For this challenge, add some grandeur to Coda Pizza by incorporating animations into your UI. Currently, adding a topping to Coda Pizza causes your pizza's preview to change abruptly. Make this change more graceful by fading the topping onto the pizza. (Hint: Try using the **`Crossfade`** composable.)

For more of a challenge, add some excitement when placing an order. When the user presses the PLACE ORDER button, make their pizza preview spin in a complete circle.

This will require several changes to your code, including refactoring your **`OrderButton`** with a new lambda parameter to be called when an order is placed. You will also need to update your **`ToppingsList`** composable to accept information about the pizza preview's rotation. The pizza can be rotated using the **`Modifier.rotate(Float)`** modifier. There are several animation APIs that can drive this animation, but we recommend using either **`animateFloatAsState`** or **`Animatable`**.

5
Effects and Coroutines

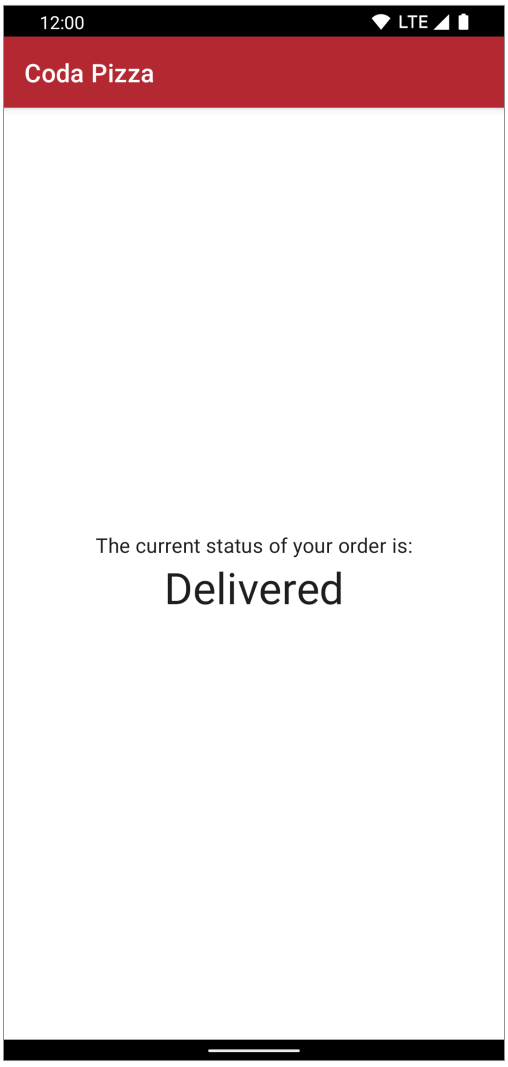
Up until now, CodaPizza has been a fairly straight forward app: it holds some state and you use Compose to render that state into UI. But modern apps are rarely that simple. Currently, all the changes to the state of your app happens within your composables. **`PizzaBuilderScreen`** maintains the state of an individual pizza, and renders UI based on that state. But how do you interact with the world outside of your composable environment? This is where side effects come into play.

A *side effect* is a change to the state of the app that happens outside the scope of a composable function. Many modern apps have some sort of backend component. A side effect can come in the form of your app posting some data to the backend servers. Or it could be a network request where you update your app state by pulling down the latest data from the servers. With Jetpack Compose being so focused on providing a framework to declaratively turn state into UI, you need controlled ways to manage side effects. Recomposition happens often in Compose and you do not want to needlessly perform work or bog down backend servers with redundant network calls.

In this chapter, you are going to connect your existing Compose code to an asynchronous data source that lives outside the Compose context. You will add a new class, **`OrderingRepository`**, to enable users to more realistically create, price, and order pizzas. You will also build a brand new screen to track the progress of a pizza you just ordered. You will execute this asynchronous work using the Kotlin coroutines library. This library allows the runtime to efficiently use resources while it is waiting for responses by pausing and resuming execution of long-running work. This allows you to write code that looks like straightforward, imperative logic without the messiness or incomprehensibility of callbacks or threads.

We will guide you step-by-step through this process. With the foundations of Jetpack Compose now established, the next few chapters are going to focus more on how to write production ready code. You will refactor some of your existing composables and add new ones in. In the end, you will have simple, reusable components and the knowledge to create your own apps using Jetpack Compose.

Figure 5.1 The chapter goal



The Effects APIs

In this chapter, you are going to be adding in three user-facing features. These features will display some text, so before we even get started with new ideas, add in some strings into the project.

Listing 5.1 Adding in some strings (`strings.xml`)

```
<resources>
    ...
    <string name="pizza_preview">Pizza preview</string>

    <string name="welcome_to_coda_pizza">Welcome to Coda Pizza!</string>
    <string name="recommended_placement">Recommended placement: %1$s</string>
    <string name="unknown_price">$??.??</string>
    <string name="ordering_pizza">Ordering pizza…</string>
    <string name="cancel">Cancel</string>
    <string name="status_not_started">Not started</string>
    <string name="status_accepted">Accepted</string>
    <string name="status_being_prepared">In the oven</string>
    <string name="status_being_delivered">Out for delivery</string>
    <string name="status_delivered">Delivered</string>
    <string name="current_order_status">The current status of your order is:</string>
</resources>
```

With that out of the way, we can start to dig into the Effects API. Back in Chapter 2, you performed some logging within a composable and you saw the recomposition process in action. Any time that the parameters passed into a composable changed, Compose would invoke the composable function and display the updated result. You also saw how to store and mutate state within a composable and how that would also cause your app to recompose.

Well-written composable functions are idempotent, meaning that when the function is invoked multiple times with the same arguments, the output remains the same. Another way of saying that is that composable functions should be "side-effect free". If your composable function does something that should only happen one time, such as navigate to a different screen or log out to the analytics system that the user has visited a particular screen, then the function is not idempotent.

In the real-world, apps are interactive and respond to user input. State changes and as a result, the UI updates to match. And you cannot always represent that with your current understanding of the Compose toolset. Back in Chapter 2, you managed and mutated state using the **`mutableStateOf()`** function. But that was contained within the composables in which you used that function. Up until now, all the state that you have mutated has been contained within the composable functions in which it is defined.

The first function in the Effects API that you will use is **`SideEffect`**. SideEffect is useful for publishing state to a component exists outside of the Compose context. Compose relies heavily on the fact that it manages transforming state into UI, and that the two always stay in sync. **`SideEffect`** gives you the ability expand the scope in which Compose pushes out changes to state.

Open up `PizzaBuilderScreen.kt`. Within the **`ToppingsList`**, add a **`SideEffect`** to log out some state held within the composable. Also add some additional logging outside of the **`SideEffect`** lambda expression. It will highlight what makes **`SideEffect`** unique and how it fits into the process in which the Compose runtime turns your composables into UI rendered on screen.

Listing 5.2 Trying out side effects (`PizzaBuilderScreen.kt`)

```
@Composable
private fun ToppingsList(
    pizza: Pizza,
    onEditPizza: (Pizza) -> Unit,
    modifier: Modifier = Modifier
) {
    var toppingBeingAdded by rememberSaveable { mutableStateOf<Topping?>(null) }

    val nullMessage = if (toppingBeingAdded != null) "is" else "is not"
    SideEffect {
        Log.d("ToppingsList", "Inside the SideEffect: there $nullMessage a topping")
    }
    Log.d("ToppingsList", "Outside the SideEffect: there $nullMessage a topping")

    toppingBeingAdded?.let { topping ->
        ...
    }
    ...
}
```

Run the app and try adding and removing toppings. In the logs, you will notice that a handful of statements get printed out. But their order might surprise you a little.

```
D/ToppingsList: Outside the SideEffect: there is not a topping
D/ToppingsList: Inside the SideEffect: there is not a topping
D/ToppingsList: Outside the SideEffect: there is a topping
D/ToppingsList: Inside the SideEffect: there is a topping
D/ToppingsList: Outside the SideEffect: there is not a topping
D/ToppingsList: Inside the SideEffect: there is not a topping
...
```

Even though the logging statement inside **`SideEffect`** is above the logging statement outside the **`SideEffect`**, the one outside is executed first. For people familiar with callbacks and lambda expressions, this might not be too surprising. With callbacks and lambda expressions, developers know that the order of execution for a chunk of code might not simply go from the top to the bottom in sequence. But under the hood, Jetpack Compose takes things a step further and this highlights an important aspect of how the Compose runtime renders the UI seen on screen.

When your Compose code is compiled into an app that you can run on a device, the compiler transforms your code into code that the Compose runtime can use. Individual functions get parameters added to them, chunks of code inserted, and much more, all in the name of providing a framework that keeps UI and state in sync. This results in code that looks much more complicated than the code that you originally wrote, but it enables the runtime to be fast and efficient in performing all the necessary work to eventually draw pixels on a screen.

In order to draw a single frame of UI on screen, the Compose runtime goes through three distinct steps. The phases are: composition, layout, and drawing. Folks with experience with most UI toolkits (even Android's legacy toolkit) might be familiar with the last two steps. But the first one might be new.

During Composition, the runtime executes your composable code and determines which components should be shown. Back in Chapter 3, you were able to toggle the visibility of a dialog using some state and an `if` statement. If that `if` statement evaluated to false, that code does not get executed during the composition phase of rendering. And if that code is not executed, then none of the composables or the UI elements that they emit will be shown to the user.

Once the runtime knows what elements to display, it can follow up with the remaining two phases. During the layout phase, the runtime is responsible for measuring out and positioning all the UI that will be shown. The drawing phase paints the pixels for each UI element to the screen. These steps work like many traditional UI toolkits.

The code within a **`SideEffect`** is executed after the composition phase but before the layout phase of rendering. Composition, and recomposition, happen often for composables, so that code within a **`SideEffect`** can be invoked many, many times as users interact with your app.

Composition, and the following two phases, happens often. Devices repeat these steps over and over, looping multiple times a second to show the latest, most updated UI to the user. Most modern devices render 60 frames a second, giving each frame only 16 milliseconds to go through the entire rendering process.

Because the rendering loop happens so often, the Compose runtime has many optimizations to make that loop happen quickly every time. If the input provided to a composable and the state within it has not changed since the last time it was rendered, then the runtime will skip the composition phase for that composable and reuse the result from the previous frame. Likewise, if the result of the composition phase has not changed from the last frame and other conditions remain the same, then the runtime will reuse the result for the layout phase from the previous frame. The compose runtime tries to do the minimum amount of work in order to keep the state and UI in sync, and only updates the UI as much as it needs to.

The optimizations to make that rendering loop fast and efficient go much, much further. The runtime keeps track of many details about the results of each phase of the rendering process. In the future, the runtime could execute composable functions in parallel or in a different order in order to speed up the rendering loop. Performing side effects without the **`SideEffect`** function in that environment could lead to unexpected results. That is why when you need to execute some code on every successful composition, you should use the **`SideEffect`** function.

The **`SideEffect`** API is useful for synchronizing state between the Compose context and components outside of it. Fun fact: under the hood, the **`Dialog`** composable that you used back in Chapter 3 uses a **`SideEffect`** in order to synchronize state between itself and the core platform functionality that it relies on. But often times, you do not need code to run as often as the code within a **`SideEffect`** does. So remove this logging code before moving on.

Listing 5.3 Cleaning up your code (`PizzaBuilderScreen.kt`)

```
@Composable
private fun ToppingsList(
    pizza: Pizza,
    onEditPizza: (Pizza) -> Unit,
    modifier: Modifier = Modifier
) {
    var toppingBeingAdded by rememberSaveable { mutableStateOf<Topping?>(null) }

    val nullMessage = if (toppingBeingAdded != null) "is" else "is not"
    SideEffect {
        Log.d("ToppingsList", "Inside the SideEffect: there $nullMessage a topping")
    }
    Log.d("ToppingsList", "Outside the SideEffect: there $nullMessage a topping")

    toppingBeingAdded?.let { topping ->
        ...
    }
    ...
}
```

Coroutines in Compose-land

In modern computing, it is common to need multiple actions to take place at the same time. For Android specifically, a ubiquitous occurence is the situation where your app is drawing and updating UI while performing a network request. Many programming languages rely on the concept of a *thread* for work that runs in the background – or, as it is often called, *asynchronously*. Threads are responsible for managing execution of your program. A thread has a sequence of instructions that it executes, performing them in the order they are declared in.

An individual thread can only do so much work in a set period of time, so to keep the system responsive to the user while also performing complicated tasks, developers distribute work across many threads. On an individual device, the system can have multiple threads, and each of those threads can execute their instructions simultaneously.

Unfortunately, threads are a fairly low-level API, making them difficult to work with. There is an implementation of threads on the Java platform, and you can create threads directly on Android, but it is very easy to make mistakes when doing so – mistakes that can lead to the application wasting resources or crashing unexpectedly.

This is where *coroutines* come in. Coroutines are Kotlin's first-party solution for defining work that will run asynchronously and are fully supported on Android. They are based on the idea of functions being able to *suspend*, meaning that a function can be paused until a long-running operation completes. When the code running in a coroutine is suspended, the thread that the coroutine was executing on is free to work on other things, like drawing your UI, responding to touch events, or making more expensive calculations.

Coroutines provide a high-level and safer set of tools to help you build asynchronous code. Under the hood, Kotlin's coroutines use threads to perform work in parallel, but you often do not have to worry about this detail. Coroutines make it easy to start work on the main thread, hop over to a background thread to perform asynchronous work, and then return the result back to the main thread.

To keep this book at a reasonable length, we cannot explain Kotlin coroutines in full. If coroutines are entirely new to you, JetBrains has excellent documentation on how to use them (`kotlinlang.org/docs/coroutines-overview.html`). Also, there is a little book written by some very cool folks titled "Kotlin Programming: The Big Nerd Ranch Guide." That book does an excellent job explaining the basics of coroutines and how to use them, as well as other Kotlin topics. We highly recommend that book. In this chapter and throughout this book, we will primarily focus on how to use coroutines in the context of an Android app.

In order to see how coroutines interact with Jetpack Compose, you are going to add some functionality into CodaPizza to perform querying the list of toppings, pricing of your pizza, and finally placing the order. But before you can mess with coroutines, you need to perform some basic refactoring on **`ToppingsList`**. Instead of directly referencing the values of the **`Topping`** enum, pass in as an argument a list of toppings.

Listing 5.4 Passing in toppings (`PizzaBuilderScreen.kt`)

```
@Composable
private fun ToppingsList(
    pizza: Pizza,
    toppings: List<Topping>,
    onEditPizza: (Pizza) -> Unit,
    modifier: Modifier = Modifier
) {
    ...
    LazyColumn(
        modifier = modifier
    ) {
        item { ... }

        items(~~Topping.values()~~ toppings) { topping ->
            ToppingCell(
                topping = topping,
                placement = pizza.toppings[topping],
                onClickTopping = {
                    toppingBeingAdded = topping
                }
            )
        }
    }
}
```

With that refactoring done, you can dive right into the coroutines. Within the solutions for this chapter, find `OrderingRepository.kt` and add it to your project.

Take a quick peek at `OrderingRepository.kt`. Within `OrderingRepository.kt`, you will find a class named **`OrderingRepository`**. This class is structured in the "repository" software design pattern and its role is to contain the business logic for pricing and ordering pizzas. From the outside, **`OrderingRepository`** looks like a normal class that would probably call some backend servers to provide functionality to your app. There are functions to calculate the price of a pizza, place an order, and keep track of the status of an order. But if you look at the implementation, you will notice that no real ordering happens, the functionality is simply faked out.

But the way that you interact with it is just the same as if you were making network requests on a real backend server. Making network requests is an asynchronous process, so these APIs use Kotlin coroutines to reflect that. Functions are either suspending functions or functions that return **`Flow`**s.

The first function from **`OrderingRepository`** that you will call will be **`getToppings()`**. **`getToppings()`** is a suspending function and you cannot directly call suspending functions in the body of a composable function. If you remember from your previous coroutine knowledge, you need a coroutine scope in order to call suspending functions. And this is where the **`LaunchedEffect`** function comes into play.

`LaunchedEffect` allows you to perform long-running side-effects in a controlled manner using coroutines. Often times, you only want to perform some work a single time. Right now, the only place you can write code that executes once is in click listeners. If the user clicks the order button, the code within that **`onClick`** lambda expression executes a single time and a toast is displayed to the user. But all code that you want executed a single time is not the result of user interaction.

A common situation in which you want to perform one-time work is when a screen in your app first displays to the user. In that scenario, you might want to fetch some data from the internet or set up some initial state for your screen. For that, you will want to reach to **`LaunchedEffect`**.

`LaunchedEffect` has two parameters. The second parameter is a lambda expression that will be executed in a similar fashion as the lambda expression you used with **`SideEffect`**. The special detail about this lambda expression is that it provides you with a coroutine scope, meaning within that lambda expression, you can call **`getToppings()`** to get the price of your pizza. The first parameter is the key and it is used to determine when to execute the lambda expression. For now, you will pass in **`Unit`** as the key, which will execute the lambda expression a single time right when **`PizzaBuilderScreen`** enters the composition tree.

The list of toppings will now live within **`PizzaBuilderScreen`** as mutable state. You will get access to an instance of **`OrderingRepository`** by passing it as an argument into **`PizzaBuilderScreen`** with a default value. `OrderingRepository.kt` has already defined a **`CompositionLocal`**, which we talked about back in Chapter 4, and it will allow you to access a single instance of the class that is stored in the composition hierarchy for you.

Listing 5.5 Using coroutines to load the toppings (`PizzaBuilderScreen.kt`)

```
@Composable
fun PizzaBuilderScreen(
    orderingRepository: OrderingRepository = LocalOrderingRepository.current,
    modifier: Modifier = Modifier
) {
    var pizza by rememberSaveable { mutableStateOf(Pizza()) }

    var toppings by remember { mutableStateOf(listOf<Topping>()) }
    LaunchedEffect(Unit) {
        toppings = orderingRepository.getToppings()
    }

    Scaffold(
        modifier = modifier,
        topBar = { ... },
        content = {
            Column {
                ToppingsList(
                    pizza = pizza,
                    toppings = toppings,
                    onEditPizza = { pizza = it },
                    modifier = Modifier
                        .fillMaxWidth()
                        .weight(1f, fill = true)
                )
                ...
            }
        }
    )
}
```

Run the app. The screen will now pop up blank, with no ingredients, but after two seconds, the list of toppings will display onscreen. This series of events happens once. Thankfully, this asynchronous work does not happen every

time the composable recomposes, as it would if you tried to do something similar with a **SideEffect**. The lambda expression for this **LaunchedEffect** only executes once as the **PizzaBuilderScreen** composable appears on screen.

Similar to what you know from the **Activity**, **Fragment**, or many other components in the Android ecosystem, composables do have a lifecycle. Thankfully, compared to those components mentioned, the lifecycle for composables is much simpler. For an individual composable, it is either alive in the composition tree or it is non-existent. For the **LaunchedEffect** you just used, the lambda expression within it have been executed a single time as the composable moved from the non-existent state to the alive state.

To see this lifecycle more clearly and how it relates to **LaunchedEffect**, let's use another API in a different spot. This time you are going to provide a suggestion on where the user should place a topping. Open up `ToppingPlacementDialog.kt`. Within **ToppingPlacementDialog**, use a **LaunchedEffect** to perform some asynchronous work. This time you will call **OrderingRepository**'s' **getToppingPlacementSuggestion()** function and display the result as a toast.

Listing 5.6 Logging in the dialog (`ToppingPlacementDialog.kt`)

```
@Composable
fun ToppingPlacementDialog(
    topping: Topping,
    onSetToppingPlacement: (placement: ToppingPlacement?) -> Unit,
    onDismissRequest: () -> Unit
) {
    val context = LocalContext.current
    val orderingRepository = LocalOrderingRepository.current
    LaunchedEffect(Unit) {
        val placementSuggestion = orderingRepository.getToppingPlacementSuggestion()
        val placementText = context.getString(placementSuggestion.label)
        val toastText = context.getString(R.string.recommended_placement, placementText)
        Toast.makeText(context, toastText, Toast.LENGTH_SHORT).show()
    }

    Dialog(onDismissRequest = onDismissRequest) {
        ...
    }
}
```

Run the app and add a few toppings to your pizza. You will see a toast display every time a few seconds after a new dialog appears on screen. This happens because as you toggle the visibility of your **ToppingPlacementDialog** composable, you add and remove it from the composition hierarchy. A new composable is created as you add it to the composition hierarchy and the **LaunchedEffect** is executed at the beginning of its lifecycle.

You are not quite done with this composable just yet. Open up Logcat. Run the app again and try adding a topping again. However, this time, quickly dismiss the dialog as soon as it pops up. You will see a logging statement that looks like this in Logcat:

```
    ...
    D/OrderingRepository: Request to get suggestion cancelled.
    ...
```

Jetpack Compose and Kotlin coroutines deeply integrate with each other to enable you to easily and safely write asynchronous code. This example above shows you how coroutine cancelation works with composables and **LaunchedEffect**. The implementation within **OrderingRepository**'s **getToppingPlacementSuggestion()** prints out a log whenever a **CancellationException** is thrown. For **ToppingPlacementDialog**, the coroutine scope within **LaunchedEffect** is canceled when the composable is removed from the composition hierarchy. That means that the work within the scope is stopped and cleaned up whenever the composable moves from the alive to the non-existent state.

Before moving on, remove the code you just added. The placement suggestion feature is not quite ready for production. The suggestions feel pretty random.

Listing 5.7 Cleaning up (`ToppingPlacementDialog.kt`)

```
@Composable
fun ToppingPlacementDialog(
    topping: Topping,
    onSetToppingPlacement: (placement: ToppingPlacement?) -> Unit,
    onDismissRequest: () -> Unit
) {
    val context = LocalContext.current
    val orderingRepository = LocalOrderingRepository.current
    LaunchedEffect(Unit) {
        val placementSuggestion = orderingRepository.getToppingPlacementSuggestion()
        val placementText = context.getString(placementSuggestion.label)
        val toastText = context.getString(R.string.recommended_placement, placementText)
        Toast.makeText(context, toastText, Toast.LENGTH_SHORT).show()
    }

    Dialog(onDismissRequest = onDismissRequest) {
        ...
    }
}
```

Asynchronously producing state

Before you can start on the next feature, you need to do some refactoring. Currently, all the logic for calculating the price of the pizza is embedded in the **`OrderButton`**. As things get more complicated, it is not a good idea to keep important business logic deeply embedded in otherwise simple composables.

Thankfully, it is relatively easy to extract business logic from composables. Open back up `PizzaBuilderScreen.kt`. Instead of calculating the price within **`OrderButton`**, pass the formatted price in as a parameter. Also, pass in a lambda expression to be invoked when the user clicks the button. That way you can perform all the complicated logic of ordering a pizza elsewhere while keeping **`OrderButton`** simple.

Listing 5.8 Extracting business logic (`PizzaBuilderScreen.kt`)

```
@Composable
private fun OrderButton(
    pizza: Pizza,
    formattedPrice: String,
    onClick: () -> Unit,
    modifier: Modifier = Modifier
) {
    val context = LocalContext.current

    Button(
        modifier = modifier,
        onClick = {
            Toast.makeText(context, R.string.order_placed_toast, Toast.LENGTH_LONG)
                .show()
        }
        onClick = onClick
    ) {
        val currencyFormatter = remember { NumberFormat.getCurrencyInstance() }
        val price = currencyFormatter.format(pizza.price)
        Text(
            text = stringResource(R.string.place_order_button, price formattedPrice)
                .toUpperCase(Locale.current)
        )
    }
}
```

Include the business logic to format the price in **`PizzaBuilderScreen`**. You will add logic to perform work when the button is clicked later.

Listing 5.9 Setting things up (`PizzaBuilderScreen.kt`)

```
@Preview
@Composable
fun PizzaBuilderScreen(
    orderingRepository: OrderingRepository = LocalOrderingRepository.current,
    modifier: Modifier = Modifier
) {
    ...
    LaunchedEffect(Unit) {
        toppings = orderingRepository.getToppings()
    }

    val currencyFormatter = remember { NumberFormat.getCurrencyInstance() }
    val formattedPrice = currencyFormatter.format(pizza.price)

    Scaffold(
        modifier = modifier,
        topBar = { ... },
        content = {
            Column {
                ToppingsList(...)

                OrderButton(
                    pizza = pizza,
                    formattedPrice = formattedPrice,
                    onClick = {},
                    modifier = Modifier
                        .fillMaxWidth()
                        .padding(16.dp)
                )
            }
        }
    )
}
```

Run the app to confirm that the UI still displays the correct information as you build a pizza.

In order to invoke the **`calculateFormattedPrice()`** on the **`OrderingRepository`**, you could lean on the same tools as you have before. Using a piece of state and a **`LaunchedEffect`**, you could create something that calculates the price of the pizza. But in this common case, Compose actually provides a specialized API: **`produceState()`**.

The **`produceState()`** function is implemented under the hood with a piece of state and a **`LaunchedEffect`**, so you just need to provide the same three parameters: an initial value, a key, and a lambda expression. Use **`produceState()`** to calculate and store the price of the pizza and display that state in the **`OrderButton`**.

Listing 5.10 Calling a suspending function (PizzaBuilderScreen.kt)

```
@Preview
@Composable
fun PizzaBuilderScreen(
    orderingRepository: OrderingRepository = LocalOrderingRepository.current,
    modifier: Modifier = Modifier,
) {
    ...
    LaunchedEffect(Unit) {
        toppings = orderingRepository.getToppings()
    }

    ~~val currencyFormatter = remember { NumberFormat.getCurrencyInstance() }~~
    ~~val formattedPrice = currencyFormatter.format(pizza.price)~~
    val initialPrice = stringResource(R.string.unknown_price)
    val formattedPrice by produceState(initialPrice, Unit) {
        value = orderingRepository.calculateFormattedPrice(pizza)
    }

    Scaffold(...)
}
```

Run the app. After a few seconds, you will see the text within the **OrderButton** display the initial price of the pizza. But if you add or remove toppings from the pizza, you will notice that the price never updates. The lambda expression for your **produceState()** is only executed once, right when **PizzaBuilderScreen** is composed for the first time. On recompositions of **PizzaBuilderScreen**, that lambda expression is not executed.

On the one hand, it is probably good that **calculateFormattedPrice()** is not called every time that **PizzaBuilderScreen** recomposes. The **calculateFormattedPrice()** function is supposed to resemble a network request, and performing unnecessary network requests is very wasteful. What you need here is a way to perform the asynchronous work of calculating the price of the pizza, but only performing that work when you need to. And that is where the key parameter on your **produceState()** comes into play.

This key is used to signal to the **produceState()** that the lambda expression should be re-executed. Whenever this key changes to a different value, the lambda expression will be executed again. Right now, you passed in Unit, and that never changes, so the lambda expression is only executed a single time right when **PizzaBuilderScreen** is composed for the first time.

Now if you passed in the pizza as the key instead, the lambda expression would run right when **PizzaBuilderScreen** is composed for the first time and every time that the pizza was modified after that. And that would, in turn, update the price in the **OrderButton**. Do that and run the app.

Listing 5.11 Calling a suspending function (PizzaBuilderScreen.kt)

```
@Preview
@Composable
fun PizzaBuilderScreen(
    orderingRepository: OrderingRepository = LocalOrderingRepository.current,
    modifier: Modifier = Modifier,
) {
    ...
    val formattedPrice by produceState(initialPrice, ~~Unit~~ pizza) {
        value = orderingRepository.calculateFormattedPrice(pizza)
    }

    Scaffold(...)
}
```

Run the app and confirm that the price updates as you add and remove toppings from the pizza.

Diving deeper into coroutines

With pricing working, you are now going to move onto placing an order for your pizza. This is another asynchronous task and it requires you to call a different function on **OrderingRepository**: **placeOrder()**. With

LaunchedEffect, you are provided a coroutine scope to do some asynchronous work. However, in this situation, you want to perform some asynchronous work when the user presses the the **OrderButton**. Within the click listener, you need to launch a coroutine scope in order to invoke **placeOrder()**.

Thankfully, unlike other spots in your Compose code, the code within the click listener is only executed once per user press. But unfortunately, the code within the click listener executes synchronously on the UI thread. By launching a coroutine scope within the click listener, you can asynchronously execute that work and also get all the benefits and functionality of coroutines.

In order to execute coroutine code that spans over many recompositions, you need to use the **rememberCoroutineScope()** function. It is built on top of the ubiquitous **remember** function and it allows you to perform long running operations within the composable world. You cannot use **LaunchedEffect** here because within the click listener, you are not inside a composable context. You cannot call the **Text** composable, for example.

You can use the **rememberCoroutineScope()** function here, however. Even better, the **rememberCoroutineScope()** function produces a coroutine scope that is scoped to the composable context in which it is invoked. So if you create a coroutine scope using this function, that scope will be alive for the same period of time in which the composable in which it is called is attached to the composition tree. If the composable in which it is invoked is removed from the composition hierarchy, that coroutine scope is cancelled. This gives you a nice environment to execute asynchronous tasks that live in the same lifecycle that their parent composables do. Use this function to launch a coroutine that will order your pizza when you click the **OrderButton**.

Listing 5.12 Launching a coroutine (PizzaBuilderScreen.kt)

```
@Preview
@Composable
fun PizzaBuilderScreen(
    orderingRepository: OrderingRepository = LocalOrderingRepository.current,
    modifier: Modifier = Modifier,
) {
    ...
    val formattedPrice by produceState(initialPrice, pizza) {
        value = orderingRepository.calculateFormattedPrice(pizza)
    }

    val scope = rememberCoroutineScope()

    Scaffold(
        modifier = modifier,
        topBar = { ... },
        content = {
            Column {
                ToppingsList(...)

                OrderButton(
                    formattedPrice = formattedPrice,
                    onClick = {
                        scope.launch {
                            orderingRepository.placeOrder(pizza)
                        }
                    },
                    modifier = Modifier
                        .fillMaxWidth()
                        .padding(16.dp)
                )
            }
        }
    )
}
```

It is a good idea to give the user some indication that work is happening in the background. For this situation, your app will display an opaque overlay on the screen while the order is taking place. Performing asynchronous work

without any indication that it is happening can lead the users to think something went wrong, so use a small piece of state, represented by a `boolean`, in order to track whether or not the user is placing their order.

Listing 5.13 Tracking loading state (`PizzaBuilderScreen.kt`)

```
@Preview
@Composable
fun PizzaBuilderScreen(
    orderingRepository: OrderingRepository = LocalOrderingRepository.current,
    modifier: Modifier = Modifier,
) {
    ...
    val scope = rememberCoroutineScope()
    var isOrdering by remember { mutableStateOf(false) }

    Scaffold(
        modifier = modifier,
        topBar = { ... },
        content = {
            Column {
                ToppingsList(...)

                OrderButton(
                    formattedPrice = formattedPrice,
                    onClick = {
                        scope.launch {
                            isOrdering = true
                            orderingRepository.placeOrder(pizza)
                            isOrdering = false
                        }
                    },
                    modifier = Modifier
                        .fillMaxWidth()
                        .padding(16.dp)
                )
            }
        }
    )
}
```

With the basic state set up, include some UI to overlay on the screen while the work is happening. Enclose all your UI in a **`Box`** and use a conditional check to determine whether or not you should display this UI, just like you did with the dialog.

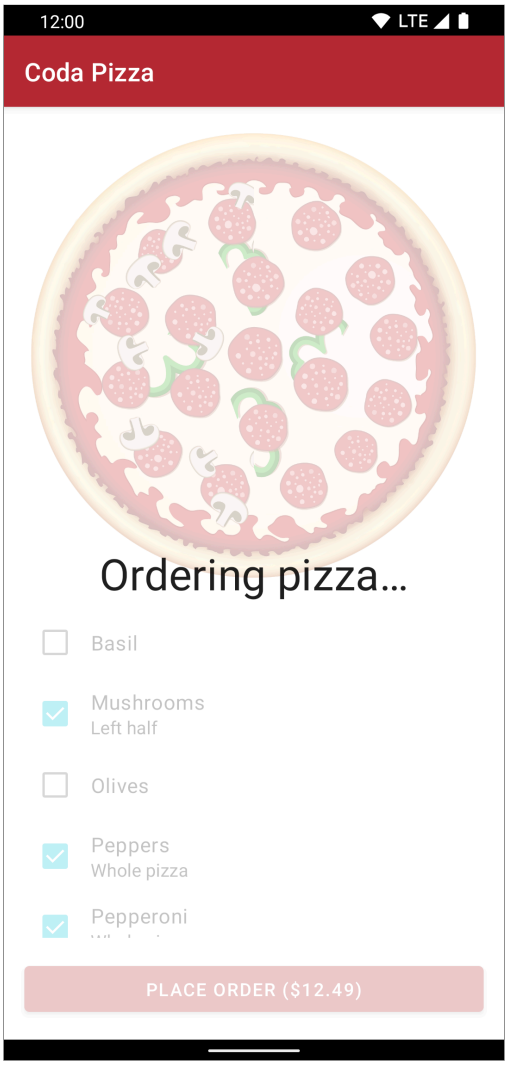
Listing 5.14 Showing loading UI (`PizzaBuilderScreen.kt`)

```
@Preview
@Composable
fun PizzaBuilderScreen(
    orderingRepository: OrderingRepository = LocalOrderingRepository.current,
    modifier: Modifier = Modifier,
) {
    ...
    Scaffold(
        modifier = modifier,
        topBar = { ... },
        content = {
            Box {
                Column { ... }

                if (isOrdering) {
                    Column(
                        verticalArrangement = Arrangement.Center,
                        horizontalAlignment = Alignment.CenterHorizontally,
                        modifier = Modifier
                            .background(Color.White.copy(alpha = 0.75f))
                            .fillMaxSize(),
                    ) {
                        Text(
                            stringResource(R.string.ordering_pizza),
                            style = MaterialTheme.typography.h4
                        )
                    }
                }
            }
        }
    )
}
```

Run the app and order a pizza. You will now see an overlay while the order is being submitted.

Figure 5.2 Your order being placed



One of the many smart things that the developers of coroutines did is that they built in the ability to manage and even cancel asynchronous work. You have already seen that in action with **`LaunchedEffect`**. When a composable contains a running **`LaunchedEffect`** and it is removed from the composition hierarchy, the coroutine scope for that **`LaunchedEffect`** is canceled. Also if there is currently running work within the lambda of a **`LaunchedEffect`** and the key changes, that work is cancelled and restarted.

The same thing happens with the coroutine scope that you created using **`rememberCoroutineScope()`**. If the state of your app changes and **`PizzaBuilderScreen`** gets removed from the composition hierarchy, the work within that coroutine scope gets cancelled.

Sometimes people make mistakes during ordering. It would be nice to give folks a chance to cancel their order after they pressed the **`OrderButton`** but before all the asynchronous work has completed. You have already learned a bit about how Compose integrates coroutines and their lifecycles into the composition process. Whenever a composable leaves the composition tree, all coroutine scopes, such as the ones used with **`LaunchedEffect`** and **`rememberCoroutineScope()`**, are cancelled and the work within them is stopped and cleaned up.

The coroutine library also gives developers direct APIs to cancel coroutines. You can also manually cancel work within a coroutine scope using the **`Job`** that is returned from the expression that launches the coroutine. By keeping a reference to that **`Job`**, you can cancel that work whenever you want by invoking its **`cancel()`** function.

Create a piece of state, holding a reference to the **`Job`** returned from launching the coroutine to calculate the price of your pizza. You will not hold a reference to the value until the click listener is invoked, so make the value nullable and initialize it to null.

Listing 5.15 Leaning into coroutines (`PizzaBuilderScreen.kt`)

```
@Preview
@Composable
fun PizzaBuilderScreen(
    orderingRepository: OrderingRepository = LocalOrderingRepository.current,
    modifier: Modifier = Modifier,
) {
    ...
    val scope = rememberCoroutineScope()
    var isOrdering by remember { mutableStateOf(false) }
    var currentJob by remember { mutableStateOf<Job?>(null) }

    Scaffold(
        modifier = modifier,
        topBar = { ... },
        content = {
            Box {
                Column {
                    ToppingsList(...)

                    OrderButton(
                        formattedPrice = formattedPrice,
                        onClick = {
                            currentJob = scope.launch {
                                isOrdering = true
                                orderingRepository.placeOrder(pizza)
                                isOrdering = false
                                currentJob = null
                            }
                        },
                        modifier = Modifier
                            .fillMaxWidth()
                            .padding(16.dp)
                    )
                }
                ...
            }
        }
    )
}
```

The reason why you treat the **`Job`** as a piece of mutable state is that it actually ties in nicely with displaying the loading UI. By assigning a value when you start the asynchronous work for ordering a pizza and nulling out that value whenever the work stops, you can easily display that loading UI at the appropriate time simply by using a null check. With that, you can add a cancel button to your loading UI to finish the work on this feature. Also, you can remove the `isOrdering` property since you are not using it anymore.

Listing 5.16 Adding a cancel button (`PizzaBuilderScreen.kt`)

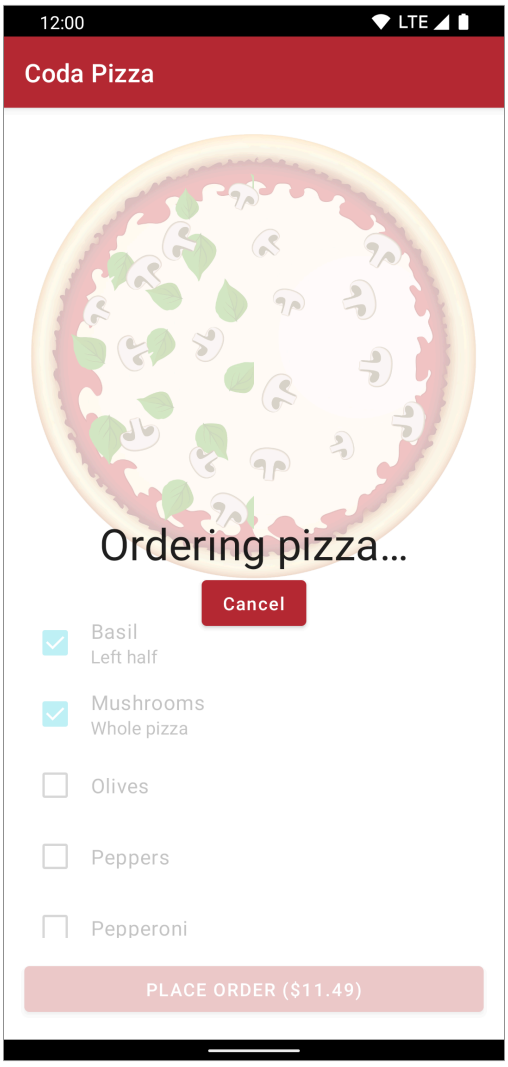
```
@Preview
@Composable
fun PizzaBuilderScreen(
    orderingRepository: OrderingRepository = LocalOrderingRepository.current,
    modifier: Modifier = Modifier,
) {
    ...
    val scope = rememberCoroutineScope()
    var isOrdering by remember { mutableStateOf(false) }
    var currentJob by remember { mutableStateOf<Job?>(null) }

    Scaffold(
        modifier = modifier,
        topBar = { ... },
        content = {
            Box {
                Column {...}

                if (isOrdering currentJob != null) {
                    Column(...) {
                        Text(
                            stringResource(R.string.ordering_pizza),
                            style = MaterialTheme.typography.h4
                        )
                        Button(
                            onClick = {
                                currentJob?.cancel()
                                currentJob = null
                            }
                        ) {
                            Text(stringResource(R.string.cancel))
                        }
                    }
                }
            }
        }
    )
}
```

Run the app. You can now order a pizza, but if you forgot your favorite topping, you can cancel your order and try again with it added in.

Figure 5.3 Your order being placed, but now cancelable



Collecting from a `Flow`

Now that you have the basics of using coroutines and suspending functions in Jetpack Compose, you will turn your focus towards the other piece of the asynchronous puzzle: **`Flow`**s. In coroutines, a **`Flow`** represents a stream of data. They come in many shapes and forms, but at the end of the day, the way you consume the data from within the stream is that you "collect" from the stream. As new values get put onto the stream, they get pushed to where you collect the values.

In order to try out collecting a **`Flow`**, you will create a new screen within CodaPizza. This new screen will be responsible for tracking an order you place as your pizza goes through preparation, cooking, and delivery. For this chapter, you will use mocked out data, but in Chapter 6, you will track the order you place.

Start off by creating a new screen. Create a new file in the `ui` package called `PizzaTrackerScreen.kt`. In your new file, define a composable function called **`PizzaTrackerScreen`** that will define your new screen. This **`PizzaTrackerScreen`** will take in a **`UUID`** that represents the order ID of your pizza along with the standard **`Modifier`**. Most of this code will look familiar to other code you have already written. The only thing new here is the **`OrderStatus`** enum, which is defined in `OrderingRepository.kt`.

Listing 5.17 Creating a new screen (PizzaTrackerScreen.kt)

```
@Composable
fun PizzaTrackerScreen(
    orderId: UUID,
    modifier: Modifier = Modifier
) {
    Scaffold(
        modifier = modifier,
        topBar = {
            TopAppBar(
                title = { Text(stringResource(R.string.app_name)) }
            )
        },
        content = {
            Column(
                verticalArrangement = Arrangement.Center,
                horizontalAlignment = Alignment.CenterHorizontally,
                modifier = Modifier.fillMaxSize()
            ) {
                Text(
                    stringResource(R.string.current_order_status),
                    style = MaterialTheme.typography.subtitle1,
                    textAlign = TextAlign.Center
                )
                Text(
                    stringResource(OrderStatus.NotStarted.stringResource),
                    style = MaterialTheme.typography.h4
                )
            }
        }
    )
}
```

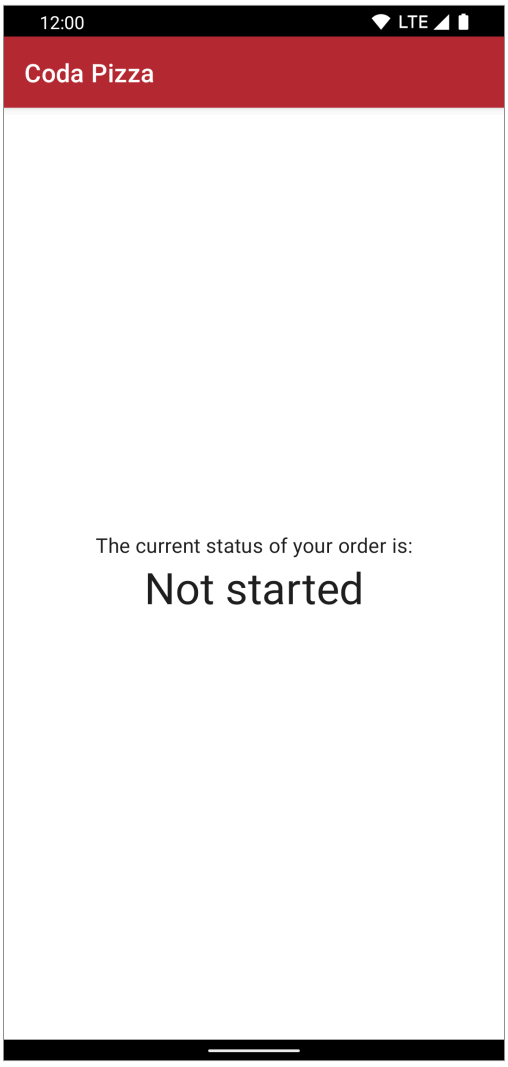
With that screen set up, update **MainActivity** to display it. Until you set up navigation in Chapter 6, use the DemoPizzaId property within **OrderingRepository** for the ID of the pizza you want to track.

Listing 5.18 Displaying the new screen (MainActivity.kt)

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            AppTheme {
                PizzaBuilderScreen()
                // PizzaBuilderScreen()
                PizzaTrackerScreen(OrderingRepository.DemoPizzaId)
            }
        }
    }
}
```

Run the app. You should see a static screen telling you that your order has not been started.

Figure 5.4 Your order has not been started



For this app, you are going to collect a **`Flow`** from your **`OrderingRepository`**, using the **`getOrderStatus()`** function. It will return a **`Flow<OrderStatus>`** and it will push updates to your order status in the form of one of the **`OrderStatus`** values.

Pass in an instance of the **`OrderingRepository`** class by using the **`LocalOrderingRepository CompositionLocal`**. In order to collect the values from this **`Flow<OrderStatus>`**, you could use the **`LaunchedEffect`** or **`produceState()`** functions. They both provide coroutine scopes in which you can invoke the **`collect()`** function on your **`Flow`**.

But similar to how **`produceState()`** was a convenience function wrapping **`LaunchedEffect`** that helped in a specific use case, there is a specific API for collecting from a **`Flow`** within a composable. Cleverly, this function is called **`collectAsState()`**. Once you provide an initial value for this state as a function parameter, you can easily transform a **`Flow`** into composable state that will automatically recompose as new values are emitted onto the stream of data.

Use this **`collectAsState()`** to transform the **`Flow<OrderStatus>`** into state that you can render in the **`PizzaTrackerScreen`** composable.

Listing 5.19 Collecting from a **Flow** (PizzaBuilderScreen.kt)

```
@Composable
fun PizzaTrackerScreen(
    orderId: UUID,
    orderingRepository: OrderingRepository = LocalOrderingRepository.current,
    modifier: Modifier = Modifier
) {
    val orderStatus by orderingRepository.getOrderStatus(orderId)
        .collectAsState(OrderStatus.NotStarted)

    Scaffold(
        modifier = modifier,
        topBar = {...},
        content = {
            Column(...) {
                Text(...)
                Text(
                    stringResource(OrderStatus.NotStarted.stringResource),
                    stringResource(orderStatus.stringResource),
                    style = MaterialTheme.typography.h4
                )
            }
        }
    )
}
```

Run the app once more. You will see your order status update as it moves through the various stages of the ordering process.

In this chapter, you used the Effects APIs and coroutines to interact with components outside of a single composable scope in an asynchronous manner. In the next chapter, you will connect your two screens within the app and perform navigation.

6

Navigation in Jetpack Compose

With the screens built and working independently, let's connect them and have them work together. In this chapter, you will use the Navigation component from Google for navigating between the two screens. Instead of showing the order status of a demo order, at the end of this chapter, you will show the order status for your order. After submitting you order on the **`PizzaBuilderScreen`**, you will navigate to the **`PizzaTrackerScreen`** and see the updates for your order as it gets processed.

The Compose Navigation component is built using many of the same pieces of the **`Fragment`** Navigation component, so many ideas and concepts from that library will translate cleanly over to CodaPizza and Jetpack Compose. Like the **`Fragment`** Navigation component, the Compose Navigation component is built around the idea of defining your application's screens as destinations that the user can navigate to. Once the destinations for your application are defined, you use a controller provided by the library to move the user throughout the app.

Like other libraries you have used in Android projects, you begin integrating the Compose Navigation component by declaring it as a dependency in your `app/build.gradle` file (the one labeled (Module: Coda_Pizza.app)).

Listing 6.1 Adding the dependency (`app/build.gradle`)

```
...
dependencies {
    ...
    implementation 'androidx.activity:activity-compose:1.5.1'

    implementation "androidx.navigation:navigation-compose:2.5.1"

    testImplementation 'junit:junit:4.13.2'
    ...
}
```

Do not forget to click the Sync Project with Gradle Files button or the Sync Now button after you have made these changes.

The Compose Navigation component operates solely in the composition hierarchy and most of the configuration happens at the highest levels of the hierarchy. Open up `MainActivity.kt`. In this file, you will define a few components which encompass the basic setup for the library.

The first component is the **`composable()`** function, which you will use to define screens as destinations within your app. Each destination will have a route, in the form of a uniquely defined string, and the content which that screen will display, in the form of a composable. The route can be thought of as a relative URL for a website.

On the imaginary website for CodaPizza, `codapizza.bignerdranch.com`, a realistic path for the pizza building screen would look something like `codapizza.bignerdranch.com/builder`. So here, you will use `"builder"` as the route for **`PizzaBuilderScreen`**.

The second component of the compose navigation library is the **`NavHost`**. The **`NavHost`** is an empty container which will display the contents for each destination. All of the destinations within your app will be declared within the **`NavHost`**. Also, every app needs an initial screen to display, so you will define the starting destination in your app as a parameter on the **`NavHost`**.

The final component of the Compose Navigation component is the **NavHostController**. It is the component that you will interact with in order to perform the navigation. It holds the state of your navigation backstack, so use **rememberNavController()** so that the compose hierarchy can keep track of it.

Listing 6.2 Getting the basic setup (MainActivity.kt)

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            AppTheme {
                // PizzaBuilderScreen()
                PizzaTrackerScreen(OrderingRepository.DemoPizzaId)
                val navController = rememberNavController()

                NavHost(navController = navController, startDestination = "builder") {
                    composable("builder") { PizzaBuilderScreen() }
                    composable("tracker") { PizzaTrackerScreen(OrderingRepository.DemoPizzaId) }
                }
            }
        }
    }
}
```

Run the app to confirm that your app launches and shows the **PizzaBuilderScreen**.

Now that you have defined both of the app's screens as possible destinations, you can navigate between them. In order to navigate from one screen to another, you call the **navigate()** function on the **NavHostController**. It takes a route in order to determine which destination to navigate to, so in order to navigate to the **PizzaTrackerScreen**, pass in "tracker" as a parameter to the **navigate()** function call.

In order to execute this work, you could pass your **NavHostController** into **PizzaBuilderScreen** and perform the navigation logic within **PizzaBuilderScreen**. But that is not the best solution available. Doing so would needlessly expose implementation details to **PizzaBuilderScreen** when all that **PizzaBuilderScreen** needs is a lambda expression to invoke when its **OrderButton** is pressed. So instead, define your navigation logic outside of the body of **PizzaBuilderScreen** and pass it in as a parameter.

Listing 6.3 Passing in a lambda expression to navigate (MainActivity.kt)

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            AppTheme {
                val navController = rememberNavController()

                NavHost(navController = navController, startDestination = "builder") {
                    composable("builder") { PizzaBuilderScreen() }
                    composable("builder") {
                        PizzaBuilderScreen(
                            onOrder = { orderId: UUID ->
                                navController.navigate("tracker")
                            }
                        )
                    }
                    composable("tracker") { PizzaTrackerScreen(OrderingRepository.DemoPizzaId) }
                }
            }
        }
    }
}
```

With the new parameter being passed into **PizzaBuilderScreen**, you now need to hook it up with your **OrderButton**. Open up PizzaBuilderScreen.kt. After declaring the lambda expression as a parameter, invoking

it in the click listener for **`OrderButton`** is pretty standard. The **`placeOrder()`** function on the **`OrderingRepository`** instance returns a **`UUID`**, which represents the ID of your newly submitted order, and you can pass that along to perform the navigation.

However, this navigation only happens after the simulated network call of the **`placeOrder()`** function. In situations like this, where you invoke lambda expressions in long-running operations that are passed in as parameters, you should reach towards the **`rememberUpdatedState()`** function.

Back in Chapter 5, you used the key parameter on the **`LaunchedEffect`** in order to restart the work of calculating the price of your pizza whenever you modified the pizza you were building. It is important that the price was accurately calculated based on the latest state of the pizza. But when you are placing the order for the pizza, you throw up an overlay that prevented any user input from modifying the pizza. Unlike the pricing feature, you did not want to restart the work of ordering the pizza unnecessarily and that was a straightforward way to control the ways in which the pizza could be altered. It would be a bad user experience if you got a pizza with different toppings than the one you placed the order for. It would also be bad if you inadvertantly placed multiple orders.

This highlights a tricky aspect of using lambda expression parameters in long-running operations. When a **`LaunchedEffect`** or coroutine scope starts executing, it "captures" everything used within that scope, including variables and lambda expressions. Normally, Compose helps prevent you from using outdated state by recomposing. But in this scenario, the coroutine scope within the click listener does not restart on recompositions. In order to ensure the **`LaunchedEffect`** always uses the correct lambda expression, treat that lamdba expression as a piece of state. That is what **`rememberUpdatedState()`** does for you. That way, when the **`LaunchedEffect`** actually does invoke the lambda expression after placing the order, Compose is always able to provide the latest version of the lambda expression.

Listing 6.4 Hooking up the navigation (`PizzaBuilderScreen.kt`)

```
@Composable
fun PizzaBuilderScreen(
    modifier: Modifier = Modifier,
    orderingRepository: OrderingRepository = LocalOrderingRepository.current,
    onOrder: (UUID) -> Unit = {},
) {
    ...
    var currentJob by remember { mutableStateOf<Job?>(null) }

    val performNavigation by rememberUpdatedState(onOrder)

    Scaffold(
        modifier = modifier,
        topBar = { ... },
        content = {
            Box {
                Column {
                    ToppingsList(...)

                    OrderButton(
                        formattedPrice = formattedPrice,
                        onClick = {
                            currentJob = scope.launch {
                                val orderId = orderingRepository.placeOrder(pizza)
                                currentJob = null
                                performNavigation(orderId)
                            }
                        },
                        ...
                    )
                }
                ...
            }
        }
    )
}
```

Run the app. Now after placing the order, the user will be navigated to the **PizzaTrackerScreen**. But on that screen, the user will still be tracking the order for the demo order, not the one they just placed. The last task you need to do is enable the **PizzaTrackerScreen** to take in an argument when navigating to it.

In order for **PizzaTrackerScreen** to accept arguments, you first need to update the route. Open up `MainActivity.kt`. Within the definition for the destination's route, you allow for arguments by including placeholders, which are in the form of variables surrounded by the curly bracket characters. For other types, you need to specify the type of the argument being passed, but the Compose Navigation component defaults to passing arguments as strings. In this situation, you will pass the **UUID** identifying the order as a string, so you do not have to do anything extra.

When calling the **navigate()** function on the **NavHostController**, you simply rely on kotlin's ability to interpolate strings. It will call the **toString()** method on the **UUID**, including it in the route string.

Listing 6.5 Passing an argument (`MainActivity.kt`)

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            AppTheme {
                val navController = rememberNavController()

                NavHost(navController = navController, startDestination = "builder") {
                    composable("builder") {
                        PizzaBuilderScreen(
                            onOrder = { orderId: UUID ->
                                navController.navigate("tracker")
                                navController.navigate("tracker/$orderId")
                            }
                        )
                    }
                    composable("tracker") { PizzaTrackerScreen(OrderingRepository.DemoPizzaId) }
                    composable("tracker/{orderId}") {
                        PizzaTrackerScreen(OrderingRepository.DemoPizzaId)
                    }
                }
            }
        }
    }
}
```

Once you are passing the order identifier through the route as an argument, the last piece of the puzzle is using that identifer when composing **PizzaTrackerScreen**. Through the lambda expression for the destination, the Navigation component provides an instance of the **NavBackStackEntry** class. This **NavBackStackEntry** represents all of the data and information for **PizzaTrackerScreen** in the navigation backstack. You can access the arguments passed into the destination through the **NavBackStackEntry**. Pull off the identifier for your order from the arguments and use that to show the order status of your order in **PizzaTrackerScreen**.

Listing 6.6 Using the argument (`MainActivity.kt`)

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            AppTheme {
                val navController = rememberNavController()

                NavHost(navController = navController, startDestination = "builder") {
                    ...
                    composable("tracker/{orderId}") { navBackStackEntry ->
                        PizzaTrackerScreen(OrderingRepository.DemoPizzaId)
                        val orderId = navBackStackEntry.arguments?.getString("orderId")
                            ?: throw IllegalStateException("Missing Order ID")

                        PizzaTrackerScreen(UUID.fromString(orderId))
                    }
                }
            }
        }
    }
}
```

Run the app. Now, after placing the order for your pizza, the app will navigate to the tracking screen for it.