Build a Scrollable List.

▼ Add a Scrollable List.

The `Card` Composable.

The `Card` composable in Jetpack Compose is a convenient way to create card-like containers that can hold content such as text, images, or other composables. It provides a structured and styled container, typically with rounded corners and a shadow, which can be used to emphasize content or group related information.

Key Features of `Card` Composable.

1. **Elevated Container:**
    - The `Card` composable provides elevation, which means it casts a shadow to give a sense of depth.
2. **Rounded Corners:**
    - By default, `Card` composables have rounded corners, making them visually distinct.
3. **Customization:**
    - The `Card` composable is highly customizable with parameters like `modifier`, `shape`, `backgroundColor`, `contentColor`, and `elevation`.

Basic Usage.

Here's a simple example of using the `Card` composable:

```
import androidx.compose.foundation.layout.*
import androidx.compose.material3.*
import androidx.compose.runtime.Composable
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.tooling.preview.Preview
import androidx.compose.ui.unit.dp

@Composable
fun SimpleCard() {
    Card(
        modifier = Modifier
            .padding(16.dp)
            .fillMaxWidth(),
        shape = MaterialTheme.shapes.medium,
        backgroundColor = MaterialTheme.colorScheme.surfac
e,
        contentColor = MaterialTheme.colorScheme.onSurface,
        elevation = CardDefaults.cardElevation(8.dp)
    ) {
        Column(
            modifier = Modifier.padding(16.dp),
            verticalArrangement = Arrangement.Center,
            horizontalAlignment = Alignment.CenterHorizonta
lly
        ) {
            Text(text = "This is a Card", style = MaterialT
heme.typography.titleMedium)
            Text(text = "It has rounded corners and elevati
on.", style = MaterialTheme.typography.bodyMedium)
        }
    }
}

@Preview(showBackground = true)
@Composable
fun SimpleCardPreview() {
    SimpleCard()
}
```

Explanation:

1. **Card Composable:**
   - The `Card` composable is used as a container with a `modifier` to set padding and width, and other parameters to customize its appearance.
2. **Modifier:**
   - The `modifier` parameter is used to set padding around the card and make it fill the maximum available width.
3. **Shape:**
   - The `shape` parameter sets the shape of the card, using a medium rounded shape from the theme.
4. **Background and Content Colors:**
   - `backgroundColor` and `contentColor` set the card's background and content colors, respectively.
5. **Elevation:**
   - The `elevation` parameter sets the shadow elevation of the card, giving it a sense of depth.
6. **Column Layout:**
   - Inside the card, a `Column` composable arranges its children vertically with padding.

Advanced Usage with Clickable Card.

You can also make the `Card` composable clickable by using the `clickable` modifier:

```
@Composable
fun ClickableCard(onClick: () -> Unit) {
    Card(
        modifier = Modifier
            .padding(16.dp)
            .fillMaxWidth()
            .clickable(onClick = onClick),
        shape = MaterialTheme.shapes.medium,
        backgroundColor = MaterialTheme.colorScheme.surfac
e,
        contentColor = MaterialTheme.colorScheme.onSurface,
        elevation = CardDefaults.cardElevation(8.dp)
    ) {
        Column(
            modifier = Modifier.padding(16.dp),
            verticalArrangement = Arrangement.Center,
            horizontalAlignment = Alignment.CenterHorizonta
lly
        ) {
            Text(text = "Clickable Card", style = MaterialT
heme.typography.titleMedium)
            Text(text = "Tap to perform an action.", style
= MaterialTheme.typography.bodyMedium)
        }
    }
}

@Preview(showBackground = true)
@Composable
fun ClickableCardPreview() {
    ClickableCard(onClick = { /* Handle click */ })
}
```

Explanation:

1. **Clickable Modifier:**
    - The `clickable` modifier is used to make the card respond to click events. The `onClick` lambda handles the click action.
2. **Composable Function:**
    - `ClickableCard` is a composable function that takes a lambda function as a parameter for the click action.

3. **Usage in Preview:**

    - The `ClickableCardPreview` function demonstrates how to use the `ClickableCard` composable with a sample click handler.

Conclusion

The `Card` composable in Jetpack Compose is a powerful and flexible way to create card-like containers for your UI. It supports various customizations, including shape, background color, content color, and elevation.

You can also make cards interactive by using the `clickable` modifier. By understanding and utilizing these features, you can create visually appealing and functional card components in your Compose applications.

The `LazyColumn` Composable.

The `LazyColumn` composable in Jetpack Compose is used to efficiently display a vertically scrolling list of items.

💡 Unlike `Column`, which lays out all its children at once, `LazyColumn` only lays out the currently visible items and dynamically creates and disposes of items as they scroll into and out of view.

This lazy loading mechanism is essential for performance, especially when dealing with large lists.

Key Features of `LazyColumn`

1. **Efficient Rendering:**

    - Only renders items that are currently visible on the screen, improving performance by conserving memory and processing power.

2. **Scrollable:**

    - Automatically provides scrolling capabilities for the items within it.

3. **Item-based API:**

    - Provides a DSL (Domain Specific Language) for defining items within the list using the `items` or `item` functions.

4. **Customizability:**
    - Allows customization of the item layout and supports adding headers, footers, and separators.

Basic Usage.

Here's a simple example of using `LazyColumn` to display a list of strings:

```
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Surface
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.compose.ui.tooling.preview.Preview

@Composable
fun SimpleLazyColumn() {
    val itemsList = listOf("Item 1", "Item 2", "Item 3", "I
tem 4", "Item 5")

    LazyColumn(
        modifier = Modifier.fillMaxSize()
    ) {
        items(itemsList) { item ->
            Text(
                text = item,
                style = MaterialTheme.typography.bodyLarge,
                modifier = Modifier.padding(16.dp)
            )
        }
    }
}

@Preview(showBackground = true)
@Composable
fun SimpleLazyColumnPreview() {
    SimpleLazyColumn()
}
```

Explanation:

1. **LazyColumn Composable:**
    - `LazyColumn` is used to create a vertically scrolling list.
2. **Modifier:**
    - The `modifier` parameter with `fillMaxSize()` makes the `LazyColumn` fill the entire available space.
3. **items Function:**
    - The `items` function takes a list of data (`itemsList`) and a lambda that describes how to display each item. In this case, it displays each string in a `Text` composable with padding.

Advanced Usage.

You can customize the `LazyColumn` to include headers, footers, and different item layouts:

```
import androidx.compose.foundation.layout.*
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items
import androidx.compose.foundation.lazy.itemsIndexed
import androidx.compose.material3.*
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.compose.ui.tooling.preview.Preview
import androidx.compose.ui.unit.dp
```

```
@Composable
fun AdvancedLazyColumn() {
    val itemsList = List(20) { "Item $it" }

    LazyColumn(
        modifier = Modifier.fillMaxSize()
    ) {
        item {
            Text(
                text = "Header",
                style = MaterialTheme.typography.titleLarg
e,
                modifier = Modifier
                    .fillMaxWidth()
                    .padding(16.dp)
            )
        }

        itemsIndexed(itemsList) { index, item ->
            Row(
                modifier = Modifier
                    .fillMaxWidth()
                    .padding(16.dp)
            ) {
                Text(
                    text = "Index: $index",
                    style = MaterialTheme.typography.bodyMe
dium
                )
                Spacer(modifier = Modifier.width(8.dp))
                Text(
                    text = item,
                    style = MaterialTheme.typography.bodyLa
rge
                )
            }
        }

        item {
            Text(
                text = "Footer",
                style = MaterialTheme.typography.titleLarg
e,
                modifier = Modifier
                    .fillMaxWidth()
                    .padding(16.dp)
            )
        }
    }
}

@Preview(showBackground = true)
@Composable
fun AdvancedLazyColumnPreview() {
    AdvancedLazyColumn()
}
```

Explanation:

1. **Header and Footer:**
    - The `item` function is used to add a single item, such as a header or footer, to the `LazyColumn`.
2. **itemsIndexed Function:**
    - The `itemsIndexed` function provides both the index and the item, allowing for more customized layouts based on the position.
3. **Row Layout:**
    - Each item in the list is displayed in a `Row` with an index and the item text, showcasing how to combine multiple composables within each item.

Conclusion.

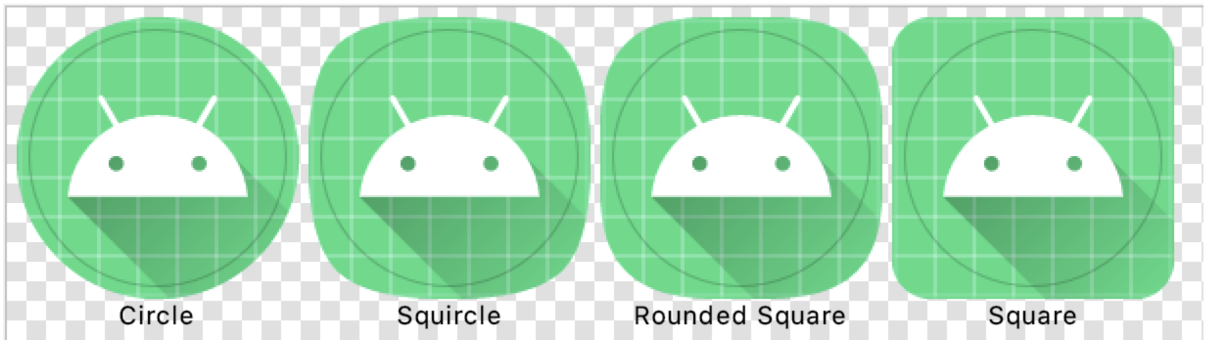
The `LazyColumn` composable is a powerful tool in Jetpack Compose for efficiently displaying large lists of items. Its lazy loading mechanism ensures that only visible items are rendered, improving performance.

By using the `items` and `item` functions, you can easily create and customize lists with headers, footers, and different item layouts. Understanding and utilizing `LazyColumn` is essential for building scalable and performant Compose applications.

▼ Change the App Icon.

You may hear an app icon referred to as a **launcher** icon. Launcher refers to the experience when you hit the Home button on an Android device to view and organize your apps, add widgets and shortcuts, and more.

If you've used different Android devices, you may have noticed that the launcher experience may look different, depending on the device manufacturer. Sometimes they create a custom launcher experience that's signature to their brand.



Regardless of the shape the device manufacturer chooses, the goal is for all the app icons on a single device to have a uniform shape for a consistent user experience. That's why the Android platform introduced support for **adaptive icons** (as of API level 26).

By implementing an adaptive icon for your app, your app is able to accommodate a large range of devices by tailoring the launcher icon based on a device's display.

This code-lab provides you with image source files for the **Affirmations** app launcher icon. You will use a tool in Android Studio, called **Image Asset Studio**, to generate different versions of the launcher icons. Afterwards, you can take what you learned and apply it to app icons for other apps!

Launcher Icons.

The goal is for your launcher icon to look crisp and clear, regardless of the device model or screen density. Screen density refers to how many pixels per inch or dots per inch (dpi) are on the screen.

The drawable folders contain the vectors for the launcher icon in XML files.

💡 A vector, in the case of a drawable icon, is a series of instructions that draw an image when it is compiled.

`mdpi`, `hdpi`, `xhdpi`, etc., are density qualifiers that you can append onto the name of a resource directory, like `mipmap,` to indicate that they are resources for devices of a certain screen density.

- `mdpi` - resources for medium-density screens (~160 dpi)
- `hdpi` - resources for high-density screens (~240 dpi)
- `xhdpi` - resources for extra-high-density screens (~320 dpi)
- `xxhdpi` - resources for extra-extra-high-density screens (~480 dpi)
- `xxxhdpi` - resources for extra-extra-extra-high-density screens (~640 dpi)
- `nodpi` - resources that are not meant to be scaled, regardless of the screen's pixel density
- `anydpi` - resources that scale to any density.

💡 You may wonder why launcher icon assets are located in `mipmap` directories, separate from other app assets located in `drawable` directories. This is because some launchers may display your app icon at a larger size than what's provided by the device's default density bucket.

For example, on an `hdpi` device, a certain device launcher may use the `xhdpi` version of the app icon instead. These directories hold the icons that account for devices that require icons with a density that is higher or lower than the default density.

💡 To avoid a blurry app icon, be sure to provide different bitmap images of the icon for each density bucket (`mdpi`, `hdpi`, `xhdpi`, etc.). Note that device screen densities won't be precisely 160 dpi, 240 dpi, 320 dpi, etc. Based on the device's screen density, Android selects the resource at the closest larger density bucket and then scales it down.

Adaptive Icons.

Foreground and Background Layers.

As of the Android 8.0 release (API level 26), there’s support for adaptive icons, which allows for more flexibility and interesting visual effects. For developers, that means your app icon is made up of 2 layers: a foreground layer and a background layer.

💡 Adaptive icons were added in API level 26 of the platform, so they should be declared in the `mipmap` resource directory that has the `-v26` resource qualifier on it.
That means the resources in this directory will only be applied on devices that are running API 26 (Android 8.0) or higher. The resource files in this directory are ignored on devices running version 25 or older in favor of the density bucketed mipmap directories.

While a vector drawable and a bitmap image both describe a graphic, there are important differences.

A bitmap image doesn't understand much about the image that it holds, except for the color information at each pixel. On the other hand, a vector graphic knows how to draw the shapes that define an image.

These instructions are composed of a set of points, lines, and curves along with color information. The advantage is that a vector graphic can be scaled for any canvas size, for any screen density, without losing quality.

A vector drawable is Android's implementation of vector graphics, intended to be flexible on mobile devices. You can define them in XML with these possible elements. Instead of providing versions of a bitmap asset for all density buckets, you only need to define the image once. Thus, reducing the size of your app and making it easier to maintain.

Download New Assets.

Because the edges of your icon could get clipped, depending on the shape of the mask from the device manufacturer, it's important to put the key information about your icon in the " safe zone." The safe zone is a circle of diameter 66 dpi in the center of the foreground layer. The content outside of the safe zone should not be essential, such as the background color, and okay if it gets clipped.

Change the App Icon.

Check out this link for comprehensive instructions:

Change the app icon | Android Developers

Change the app icon for the Affirmations app.

https://developer.android.com/codelabs/basic-android-kotlin-compose-training-change-app-icon?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-3-pathway-2%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-compose-training-change-app-icon#4

▼ Quiz.

Which Composable should you use to create a vertically scrollable grid with an undetermined number of items?

💡 To create a vertically scrollable grid with an undetermined number of items in Jetpack Compose, you should use the `LazyVerticalGrid` composable.

The `LazyVerticalGrid` composable is part of the Jetpack Compose Foundation library and allows you to create a grid layout that only renders the visible items and dynamically loads more as you scroll.

Key Features of `LazyVerticalGrid`.

1. **Efficient Rendering:**
    - Only renders the items currently visible on the screen and dynamically loads items as you scroll.
2. **Grid Layout:**
    - Allows you to specify the number of columns and configure how items are arranged within the grid.
3. **Customizability:**
    - You can customize the appearance and behavior of the grid items using standard Compose modifiers and layout composables.

Basic Usage.

Here's an example of using `LazyVerticalGrid` to display a grid of items:

First, you need to include the necessary dependency in your `build.gradle` file if it's not already included:

```
dependencies {
    implementation "androidx.compose.foundation:foundation:
<latest_version>"
}
```

Then, you can use the `LazyVerticalGrid` in your Compose code:

```
import androidx.compose.foundation.ExperimentalFoundationAp
i
import androidx.compose.foundation.layout.*
import androidx.compose.foundation.lazy.grid.GridCells
import androidx.compose.foundation.lazy.grid.LazyVerticalGr
id
import androidx.compose.foundation.lazy.grid.items
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Surface
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.compose.ui.tooling.preview.Preview
import androidx.compose.ui.unit.dp

@OptIn(ExperimentalFoundationApi::class)
@Composable
fun SimpleLazyVerticalGrid() {
    val itemsList = List(100) { "Item $it" }

    LazyVerticalGrid(
        columns = GridCells.Fixed(3), // Define the number
of columns
        modifier = Modifier
            .fillMaxSize()
            .padding(16.dp)
    ) {
        items(itemsList) { item ->
            Box(
                modifier = Modifier
                    .padding(4.dp)
                    .aspectRatio(1f) // Maintain a square a
spect ratio
            ) {
                Text(
                    text = item,
                    style = MaterialTheme.typography.bodyLa
rge,
                    modifier = Modifier
                        .fillMaxSize()
                        .padding(8.dp)
                )
            }
        }
    }
}

@Preview(showBackground = true)
@Composable
fun SimpleLazyVerticalGridPreview() {
    SimpleLazyVerticalGrid()
}
```

Explanation:

1. **LazyVerticalGrid Composable:**
    - `LazyVerticalGrid` is used to create a grid layout that scrolls vertically.
2. **GridCells.Fixed:**
    - `GridCells.Fixed(3)` specifies that the grid should have a fixed number of columns (3 in this case).

3. **items Function:**
    - The `items` function is used to iterate over the list of items and display each item in the grid.
4. **Box Layout:**
    - Each item is displayed inside a `Box` composable, which is padded and maintains a square aspect ratio using the `aspectRatio` modifier.
5. **Text Composable:**
    - The `Text` composable displays the item text within each grid cell.

Customizing the Grid.

You can further customize the `LazyVerticalGrid` by using different types of `GridCells` or by adjusting the item layout:

Using Adaptive Grid Cells.

To create a grid with adaptive cell sizes, use `GridCells.Adaptive`:

```
@OptIn(ExperimentalFoundationApi::class)
@Composable
fun AdaptiveLazyVerticalGrid() {
    val itemsList = List(100) { "Item $it" }

    LazyVerticalGrid(
        columns = GridCells.Adaptive(minSize = 128.dp), //
Define the minimum cell size
        modifier = Modifier
            .fillMaxSize()
            .padding(16.dp)
    ) {
        items(itemsList) { item ->
            Box(
                modifier = Modifier
                    .padding(4.dp)
                    .aspectRatio(1f) // Maintain a square a
spect ratio
            ) {
                Text(
                    text = item,
                    style = MaterialTheme.typography.bodyLa
rge,
                    modifier = Modifier
                        .fillMaxSize()
                        .padding(8.dp)
                )
            }
        }
    }
}

@Preview(showBackground = true)
@Composable
fun AdaptiveLazyVerticalGridPreview() {
    AdaptiveLazyVerticalGrid()
}
```

Explanation:

1. **GridCells.Adaptive:**

    - `GridCells.Adaptive(minSize = 128.dp)` specifies that each grid cell should have a minimum size of 128.dp, and the grid will adapt to fit as many columns as possible based on the available width.

By using `LazyVerticalGrid`, you can create efficient, scrollable grids with flexible item layouts, making it an ideal choice for displaying large lists of items in a grid format in Jetpack Compose.