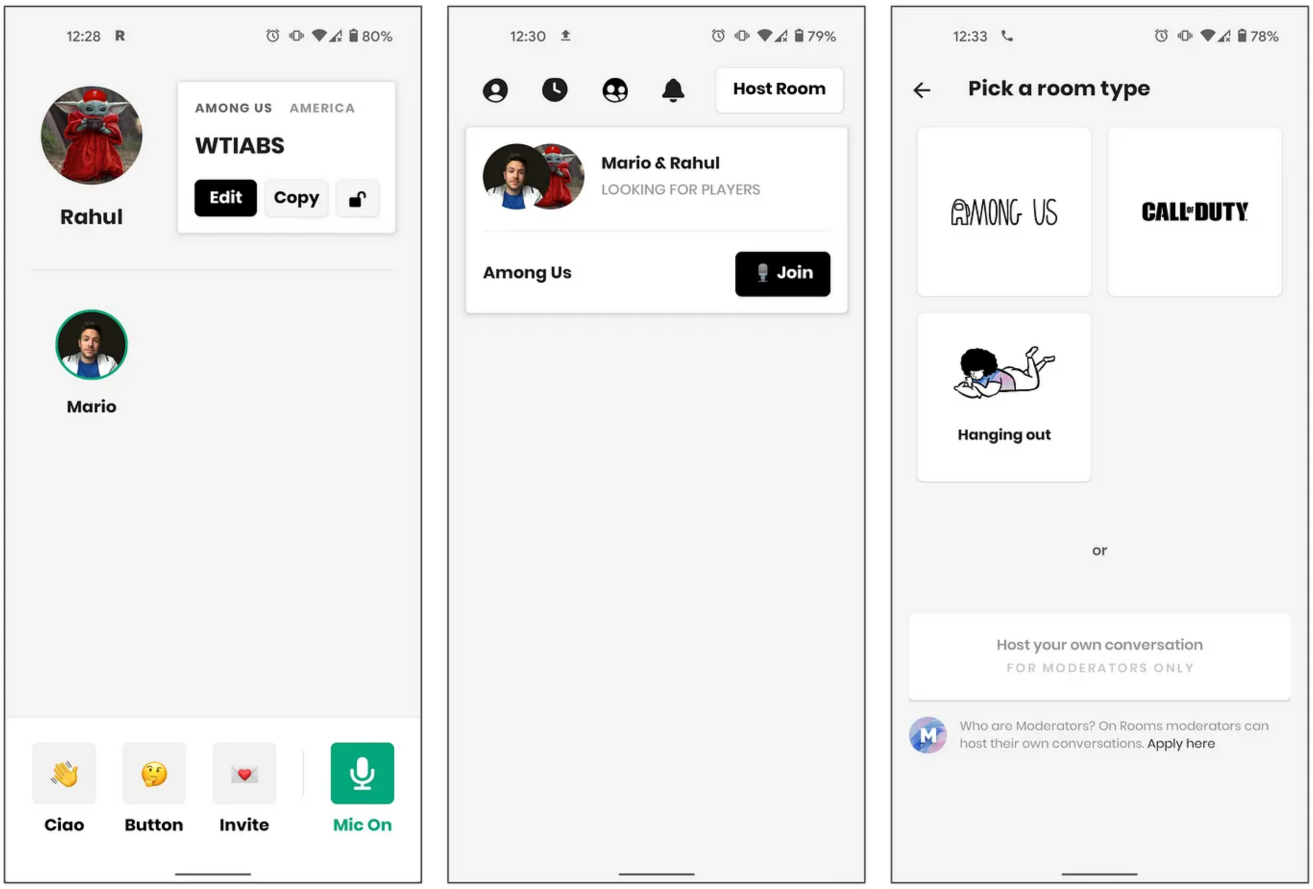
Supporting different screen sizes on Android with Jetpack Compose

In this post, we are going to discover the pitfalls of using hard-coded dimensions and a way to support different screen sizes.

Rahul Sainani · Follow
Published in ProAndroidDev
4 min read · May 6, 2021



Rooms, an app built with Jetpack Compose — dimensions case study.

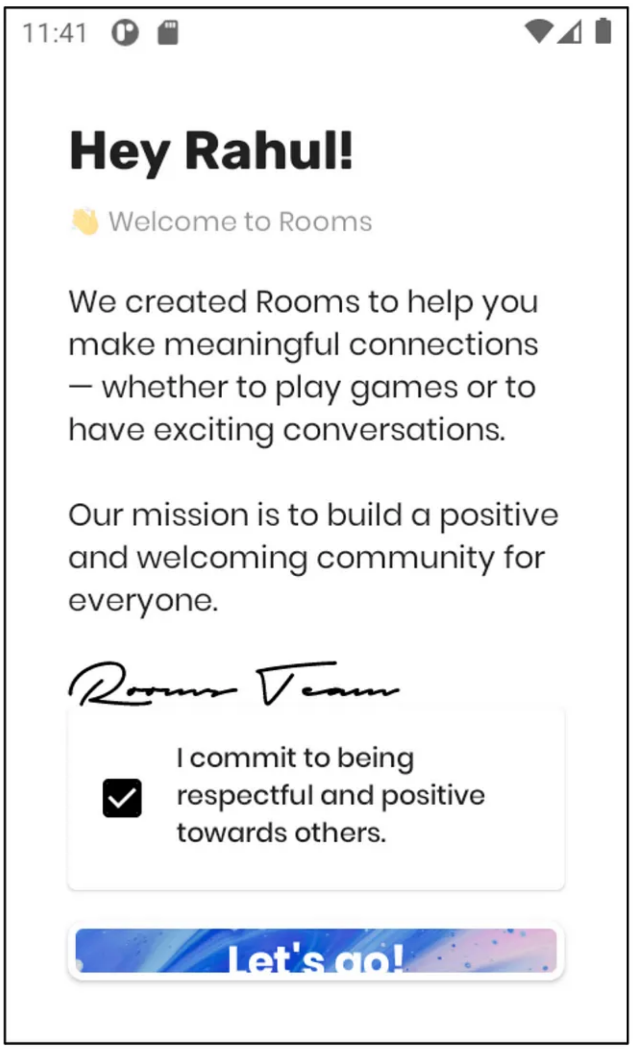
Since the early days of Android, it has been recommended to use Density-independent pixel (abbreviated dp or dip) instead of Pixels for creating layouts and UI elements. Dp helps in creating layouts on multiple screen sizes with different densities easier and more manageable.

. . .

> *"Dp is a virtual pixel unit that takes the same amount of space as 1 pixel on 160 dpi (mdpi) screen." — Android Design Bytes, gift that keeps on giving* ❤

That means 50dp will take roughly the same amount of physical real world space on a Pixel 5 and on a Nexus One, roughly the size of a finger tip. But if you think about it, we have much less space on smaller devices (regardless of the density) as they are, as you guessed, smaller. Yet we use the same set of dimensions for those devices as our latest and biggest devices.

The Problem — Same dimensions on Nexus One and Pixel 5



11:41

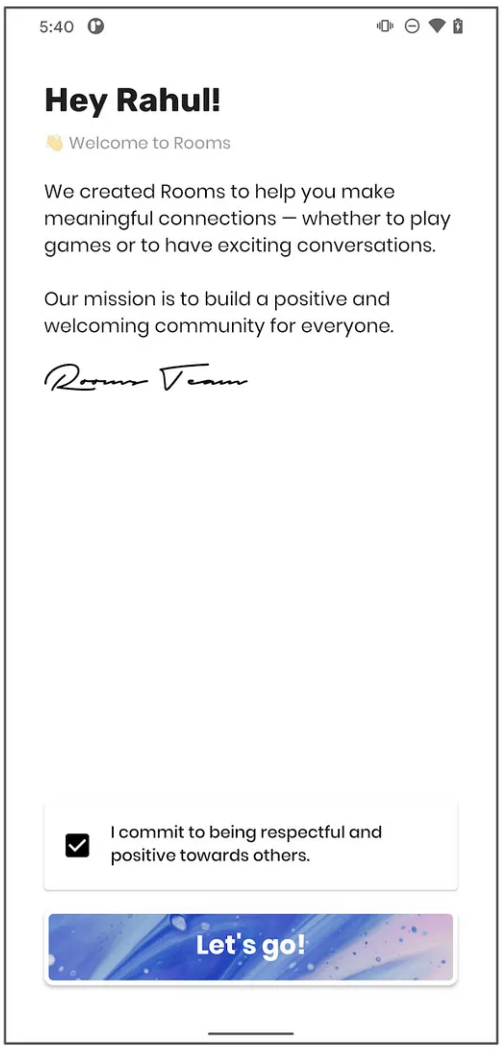
Hey Rahul!

👋 Welcome to Rooms

We created Rooms to help you make meaningful connections — whether to play games or to have exciting conversations.

Our mission is to build a positive and welcoming community for everyone.

☑ I commit to being respectful and positive towards others.



5:40

Hey Rahul!

👋 Welcome to Rooms

We created Rooms to help you make meaningful connections — whether to play games or to have exciting conversations.

Our mission is to build a positive and welcoming community for everyone.

☑ I commit to being respectful and positive towards others.

Here's the same screen using the same dimensions on Nexus One (left) and Pixel 5 (right).

Here we can see that while the screen looks pretty neat on a Pixel 5, it looks quite bad on a smaller device. The padding takes a considerable proportion of the space, the text size is way too big, and there's not enough space left for the button so it gets cropped. Even though the screenshot on the post doesn't look too shabby, it leaves a lot to be desired when experienced on an actual device. Obviously we can make the screen scroll-able and that would be fine for most apps but we want to do better, don't we? 🤓

In the screenshot above, we had been using Dimensions by creating a *Dimens* object with all the app specific dimensions in one place.

```
/**
 * An 8dp grid system. Smaller components can align to a 2dp 'sub' grid.
 */
object Dimens {
    val grid_0_25 = 2.dp
    val grid_0_5 = 4.dp
```

```kotlin
    val grid_1 = 8.dp
    val grid_1_5 = 12.dp
    val grid_2 = 16.dp
    val grid_2_5 = 20.dp
    val grid_3 = 24.dp
    val grid_3_5 = 28.dp
    val grid_4 = 32.dp
    val grid_4_5 = 36.dp
    val grid_5 = 40.dp
    val grid_5_5 = 44.dp
    val grid_6 = 48.dp
}
```

Dimens.kt hosted with ❤ by GitHub view raw

Initial dimensions approach

Although this allowed us to define dimensions in Kotlin, we did not take advantage of resource configurations as individual dimensions have the same values for all the screen sizes and configurations.

> *For instance, Dimens.grid_1 will always be 8.dp regardless of the screen size.*

We can take inspiration from the way colors are set in the Theme that allows apps to support multiple themes with relative ease. The idea is to provide values based on a certain configuration, let's see how colors are setup and how do they update when the theme is updated.

We define two *Colors* objects *LightThemeColors* and *DarkThemeColors* and based on the *darkTheme* flag one of them is provided to the CompositionLocal.

```kotlin
// To simplfy the example, we will use the defaults provided in the Material Theme
private val LightThemeColors = lightColors()

private val DarkThemeColors = darkColors()

@Composable
fun ProvideAppColors(
    colors: Colors,
    content: @Composable () -> Unit
) {
    val colorPalette = remember { colors }
    CompositionLocalProvider(LocalAppColors provides colorPalette, content = content)
}

private val LocalAppColors = staticCompositionLocalOf {
        LightThemeColors
}

@Composable
fun AppTheme(
    darkTheme: Boolean = isSystemInDarkTheme(),
    content: @Composable () -> Unit
) {
    val colors = if (darkTheme) DarkThemeColors else LightThemeColors

    ProvideAppColors(colors = colors) {
        MaterialTheme(
            colors = colors,
            shapes = Shapes,
            typography = typography,
            content = content
        )
    }
}

object Theme {
    val colors: Colors
        @Composable
        get() = LocalAppColors.current
}

```

Theme.kt hosted with ❤ by GitHub view raw

Colors provided to CompositionLocal based on darkTheme flag

To use colors in the composable screens reference them through the theme object.

```kotlin
@Composable
fun AppProgressBar() {
    CircularProgressIndicator(color = Theme.colors.onBackground)
}
```

ReferencingColors.kt hosted with ❤ by GitHub view raw

Color of the progress bar would be the one set as onBackground in the selected theme

Similarly, we can define dimensions for different configurations and take advantage of smallest width configurations. We can define a default set of dimensions and one for devices with shortest width of at least 360dp.

```
class Dimensions(
    val grid_0_25: Dp,
    val grid_0_5: Dp,
    val grid_1: Dp,
    val grid_1_5: Dp,
    val grid_2: Dp,
    val grid_2_5: Dp,
    val grid_3: Dp,
    val grid_3_5: Dp,
    val grid_4: Dp,
    val grid_4_5: Dp,
    val grid_5: Dp,
    val grid_5_5: Dp,
    val grid_6: Dp,
    val plane_0: Dp,
    val plane_1: Dp,
    val plane_2: Dp,
    val plane_3: Dp,
    val plane_4: Dp,
    val plane_5: Dp,
    val minimum_touch_target: Dp = 48.dp,
)

val smallDimensions = Dimensions(
    grid_0_25 = 1.5f.dp,
    grid_0_5 = 3.dp,
    grid_1 = 6.dp,
    grid_1_5 = 9.dp,
    grid_2 = 12.dp,
    grid_2_5 = 15.dp,
    grid_3 = 18.dp,
    grid_3_5 = 21.dp,
    grid_4 = 24.dp,
    grid_4_5 = 27.dp,
    grid_5 = 30.dp,
    grid_5_5 = 33.dp,
    grid_6 = 36.dp,
    plane_0 = 0.dp,
    plane_1 = 1.dp,
    plane_2 = 2.dp,
    plane_3 = 3.dp,
    plane_4 = 6.dp,
    plane_5 = 12.dp,
)

val sw360Dimensions = Dimensions(
    grid_0_25 = 2.dp,
    grid_0_5 = 4.dp,
    grid_1 = 8.dp,
    grid_1_5 = 12.dp,
    grid_2 = 16.dp,
    grid_2_5 = 20.dp,
    grid_3 = 24.dp,
    grid_3_5 = 28.dp,
    grid_4 = 32.dp,
    grid_4_5 = 36.dp,
    grid_5 = 40.dp,
    grid_5_5 = 44.dp,
    grid_6 = 48.dp,
    plane_0 = 0.dp,
    plane_1 = 1.dp,
    plane_2 = 2.dp,
    plane_3 = 4.dp,
    plane_4 = 8.dp,
    plane_5 = 16.dp,
)
```

This is similar to defining two different **dimens.xml** files in **res/values** and **res-values-sw360dp**. This forces us to define all the non default dimensions for different configurations at compile time. With the xml approach, we do not get these checks, easily anyway.

> *Note that we can also define dimensions with a default value like* **minimum_touch_target** = 48.*dp*

Now, we have defined both sets of dimensions but how does the system know which one to use? We use CompositionLocal and provide dimensions based on the current configuration.

```
@Composable
fun ProvideDimens(
    dimensions: Dimensions,
    content: @Composable () -> Unit
) {
    val dimensionSet = remember { dimensions }
    CompositionLocalProvider(LocalAppDimens provides dimensionSet, content = content)
}

private val LocalAppDimens = staticCompositionLocalOf {
    smallDimensions
}

@Composable
fun AppTheme(
    darkTheme: Boolean = isSystemInDarkTheme(),
    content: @Composable () -> Unit
) {
    val colors = if (darkTheme) DarkThemeColors else LightThemeColors
    val configuration = LocalConfiguration.current
    val dimensions = if (configuration.screenWidthDp <= 360) smallDimensions else sw360
    val typography = if (configuration.screenWidthDp <= 360) smallTypography else sw360

    ProvideDimens(dimensions = dimensions) {
        ProvideColors(colors = colors) {
            MaterialTheme(
                colors = colors,
                shapes = Shapes,
                typography = typography,
                content = content,
            )
        }
    }
}

object AppTheme {
    val colors: Colors
        @Composable
        get() = LocalAppColors.current

    val dimens: Dimensions
        @Composable
        get() = LocalAppDimens.current
}

val Dimens: Dimensions
    @Composable
    get() = AppTheme.dimens
```

```kotlin
        get() = AppTheme.dimens

```

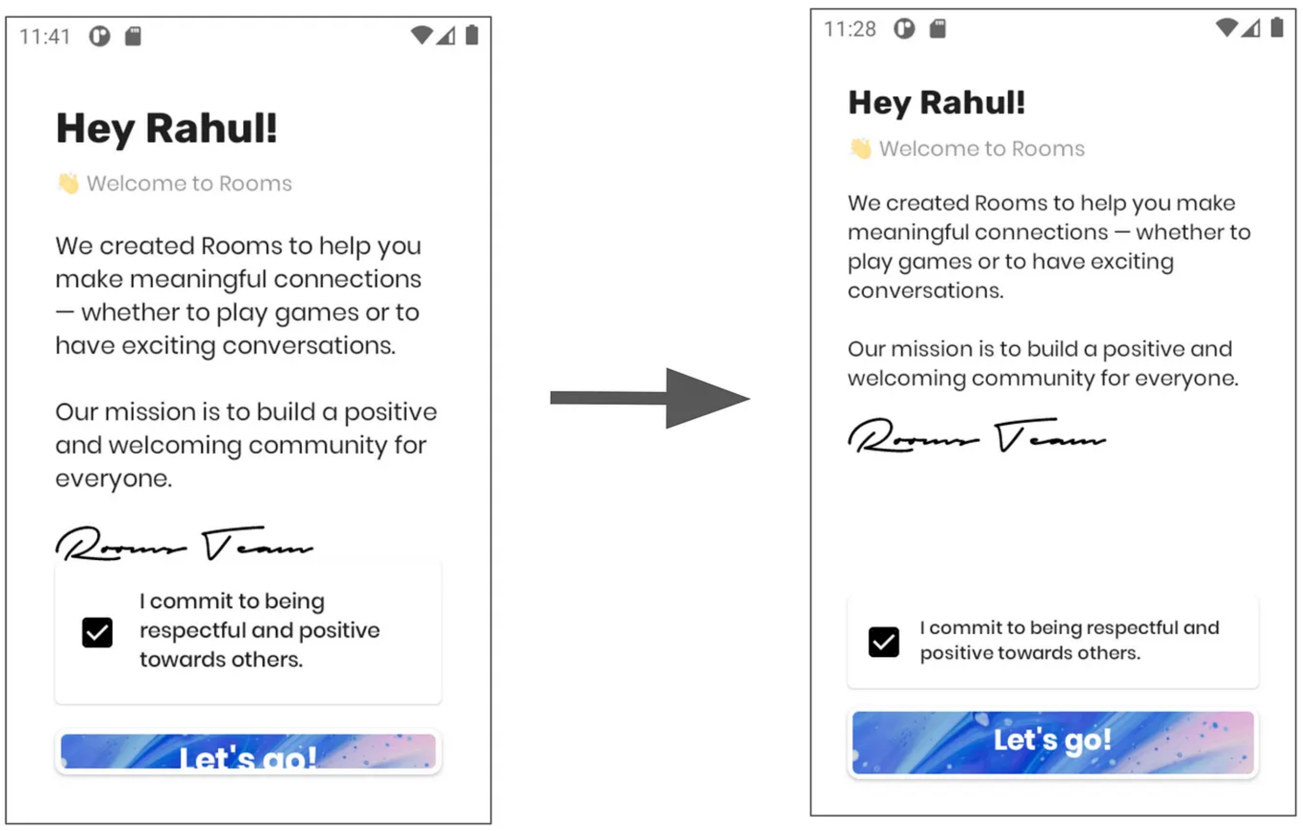
Theme-Dimens.kt hosted with ❤ by **GitHub** view raw

As you can see, we provide *smallDimensions* for small devices and *sw360Dimensions* for devices with *screenWidthDp* of at least 360. We can also define multiple typography objects with different font sizes in the same way. Dimensions can now be referenced through the theme object as shown below.

```kotlin
Row(
  modifier = Modifier.padding(
    horizontal = AppTheme.dimens.grid_2,
    vertical = AppTheme.dimens.grid_3
  ),
) {
  ...
}
```

ReferencingDimens.kt hosted with ❤ by **GitHub** view raw

And now our screen looks much better on a Nexus One!


11:41

Hey Rahul!

👋 Welcome to Rooms

We created Rooms to help you make meaningful connections — whether to play games or to have exciting conversations.

Our mission is to build a positive and welcoming community for everyone.

☑ I commit to being respectful and positive towards others.

Let's go!

11:28

Hey Rahul!

👋 Welcome to Rooms

We created Rooms to help you make meaningful connections — whether to play games or to have exciting conversations.

Our mission is to build a positive and welcoming community for everyone.

☑ I commit to being respectful and positive towards others.

Let's go!

A classic before and after screenshot for effect! 😎

If you are migrating the app from the Android View system to Jetpack Compose and don't want to redefine dimensions again, another approach would be to directly reference the dimensions from xml (using dimensionResource(id=R.dimen.grid_1)) similar to how strings are used in Compose. You would have to define multiple sets of dimensions in **res/values** and **res/values-sw360dp** to have the desired result.

. . .

In most cases, we can rely on various modifiers like *fillMaxWidth*, *fillMaxHeight*, *wrapContent* or think of our layouts in percentages/fractions or aspect ratios but there are definitely cases where it's important to use dimensions. Following the same pattern, we can define dimensions or integers for sw600dp (7 inch Tablets) and sw720dp (10 inch Tablets) whenever they make a comeback on Android. 🙃 In many cases the layouts for tablets are different and "traditionally" have resided in **res/layout-sw600dp** or **res/layout-sw720dp** or other configuration for even larger devices, but we can still rely on common dimensions being defined for those configurations so we do not duplicate them in all the layout files/composable screens for larger screen sizes.

. . .

Migrating existing Dimens

In case you already have Dimens defined in the object like in the beginning of the post, you can use *replaceWith* param from the Deprecated annotation for IntelliJ to assist with the migration.

```kotlin
@Deprecated(
    message = "Use AppTheme.dimens instead to support smaller screen sizes as well",
    replaceWith = ReplaceWith(expression = "AppTheme.dimens", imports = ["_.ui.theme.Ap
)
object Dimens {
  ...
}

class Dimensions(
  ...
)

val defaultDimensions = Dimensions(
  ...
)

val sw360Dimensions = Dimensions(
  ...
)
```

ReplaceDimens.kt hosted with ❤ by GitHub view raw

. . .

References

- DesignBytes: Density-independent Pixels
- Supporting different screen sizes
- CompositionLocal Documentation
- dimensionResource()
- Deprecated Annotation Documentation

. . .

That's all folks! Feel free to comment or message me if you have any questions.

GitHub | LinkedIn | Twitter

Android Jetpack Compose Android App Development Kotlin AndroidDev