Android Mobile Application Development
Week 3: Android Design

With 2.9 million apps in the Google play store users have many choices when choosing an app.

- 25% of apps aren't used more than once
- 29% of users will switch to another app if an app doesn't meet their needs
    - 70% due to lagging load times
    - 67% too many steps
- 34% aren't opened more than 11 times

To attract and keep users, apps need to be well designed and provide a delightful app experience.

App Design

The fundamentals of app design are the same regardless of platform.

1. Design for the device
    - Mobile, mobile, mobile
    - All apps should be easy to figure out and use
    - Make onboarding quick and easy
        - Download, install, start instantly, no lengthy startup
        - Avoid requiring initial signup/login
        - Show content immediately
    - Avoid unnecessary interruptions
    - Interaction is through taps and gestures
        - The comfortable minimum size of tappable UI elements is 48 x 48 points
    - No "Home" in apps
    - Artwork and image should be useful and draw the user in
        - Adapt art to the screen size, high quality media is expected
    - Handle different device sizes and orientations

2. Content
    - Get users to the content they care about quickly
    - Provide only relevant, appropriate content that's useful to the immediate task
    - If in doubt, leave it out

3. Focus on the User
    - Apps should have a well-defined target user
        - target apps to a specific user level
    - Put the user in control
    - Think through the user flow
    - Get them to the relevant information quickly
    - Provide subtle but clear, immediate feedback
    - Create a compelling user experience
        - User interaction consistency

4. Focus on the task
    - Apps should have a well-defined goal
    - What problem is your app solving?

- How is it better or different from other apps?

5. Understand platform conventions
    - Utilize the back button provided on Android devices, don't add one to your apps
    - Use a Floating Action Button for the primary button/action on a screen, usually in the lower right corner
    - Understand the difference between using tabs (at the top) to segment related data and bottom navigation for navigating to different screens in an app
    - Use Up navigation to navigate to the logical parent screen in the app's hierarchy

Principles of Mobile App Design: Engage Users and Drive conversations
http://think.storage.googleapis.com/docs/principles-of-mobile-app-design-engage-users-and-drive-conversions.pdf (checklist at the end)

What design principle stood out to you?

Chapter 1: App Navigation and Exploration
- Show the value of your app upfront
- Organize and label menu categories to be user friendly
    - Be predictable
    - Less is more
- Allow users to go back easily in one step
    - Back button to navigate in reverse-chronological order through the history of screens the user has recently worked with.
    - Up navigation to navigate to the logical parent screen in the app's hierarchy.
- Make it easy to manually change location
- Create frictionless transitions between mobile apps and the mobile web.

Chapter 5: Form Entry
- Build user-friendly forms
    - ensure that form fields are not obstructed from view by interface elements such as the keyboard.
    - automatically advance each field up the screen.
    - include efficiencies like auto-populate, auto-capitalization, and credit card scanning.
    - Use the hint attribute instead of default text in an edittext so the user doesn't need to delete the default text
- Communicate form errors in real time
- Match keyboard with the required text inputs
    - Specify keyboard type using the
    - KeyboardOptions class provides keyboard configuration options that you can define and assign to the TextField keyboardOptions parameter
        - keyboardType
        - capitalization
        - autoCorrect
        - imeOptions lets you define the action button such as the search icon
- Provide helpful information in context in forms

Chapter 6: Usability and Comprehension

- Speak the same language as your users
    - No big words
    - Avoid jargon or unconventional terminology
    - Be descriptive
    - Be succinct
    - Avoid truncation
    - Make text legible
    - Clear communication and functionality should always take precedence over promoting the brand message.
    - Language will depend on your target user
        - Expert vs novice will expect different language
    - Mental model
        - A mental model is a user's beliefs and understanding about a system based on their previous experiences
        - A users mental model impacts how they use a system and helps them make predictions about it
        - Mismatched mental models are common, especially with designs that try something new
- Provide text labels and visual keys to clarify visual information
    - Labeled icons are more easily understood
    - Categories with visual indicators should include a key
- Be responsive with visual feedback after significant actions
    - Provide clear feedback
    - Communication
        - Toasts are used to provide the user with simple feedback in a small popup https://developer.android.com/guide/topics/ui/notifiers/toasts
            - no action required from the user, not clickable
            - fills a small amount of space
            - automatically disappears
        - SnackBars are very similar to toasts but more customizable. Snackbars animate up from the bottom of the screen and can be swiped away from the user. They are prominent enough that the user can see it, but not so prominent that it prevents the user from working with your app. They automatically disappear like Toasts.
          https://developer.android.com/training/snackbar
            - Snackbars allow you to add an action for a user response
                - A button will be added to the right of the message
            - Snackbars supercede Toasts and are the preferred method for displaying brief, transient messages to the user
        - A dialog is a small window that prompts the user to make a decision or enter additional information.
          https://developer.android.com/guide/topics/ui/dialogs
            - Dialogs take over the screen and requires the user to take an action before they can proceed.

        - They are disruptive as the user must act on them in order to continue.
        - Only use a Dialog when the user's response is critical to your app flow as they are disruptive, otherwise use a SnackBar or Toast.
- Let the user control the level of zoom
- Ask for permissions in context
    - Communicate the value the access will provide
    - Users are more likely to grant permission if asked in the context of the relevant task
    - Users are only prompted once so you want to ask them at the most likely place for them to agree
        - Users can change permissions in settings but many won't
    - Every Android app runs in a limited-access sandbox. If an app needs to use resources or information outside of its own sandbox, the app has to request the appropriate permission.
    - You declare that your app needs a permission by listing the permission in the app manifest using a <uses-permission> element. In Android 6.0 Marshmallow(API 23) the permission system was redesigned so apps have to ask users for each permission needed at runtime. You must call the permission request dialog programmatically.
        - When the user responds to your app's permission request, the system invokes the onRequestPermissionsResult() method, passing it the user response. Your app has to override that method to find out whether the permission was granted and behave accordingly.
        - If an app tries to call some function that requires a permission which user has not yet granted, the function will suddenly throw an exception and the application will crash.
        - Your app must handle if the user denies permission or selects the "Don't ask again" option in the permission request dialog
        - Also, users are able to revoke the granted permission anytime through the device's settings so you always need to check to see if they've granted permission and request again if needed.

Small group discussion (lab 1):
What's an example of an app you've used that either does a good job, or not, of one of these design principles?

Material Design
Android Design https://developer.android.com/design
Material Design https://m2.material.io/
The Material design system is an adaptable system of guidelines, components, and tools that support the best practices of user interface design.

Google launched Material design in 2014 to provide guidance and advice to developers designing and developing apps. (not just Android)

Making Material Design
https://www.youtube.com/watch?v=rrT6v5sOwJg&list=PL8PWUWLnnIXPD3UjX931fFhn3_U5_2uZG

Design
https://m2.material.io/design
Material System Introduction https://material.io/design/introduction/

- Material Design is a visual language that synthesizes the classic principles of good design with the innovation of technology and science.

Material Foundation Overview https://m2.material.io/design/foundation-overview

- Layout
  - Understanding layout https://m2.material.io/design/layout/understanding-layout.html#principles
    - Layout principles – predictable, consistent, responsive
  - Pixel density https://m2.material.io/design/layout/pixel-density.html#pixel-density
- Color
  - Color system https://m2.material.io/design/color/the-color-system.html
    i. Color themes are designed to be harmonious, ensure accessible text, and distinguish UI elements and surfaces from one another.
  - Tools https://m2.material.io/design/color/the-color-system.html#tools-for-picking-colors
    i. Material Design color tool https://www.material.io/tools/color/
    ii. Material Design Palette https://www.materialpalette.com/ (click on two for primary and secondary colors)
    iii. 2014 Material Design color palettes
- Typography
  - Type system https://m2.material.io/design/typography/the-type-system.html#type-scale
  - Font size units
    - always use the “sp” (scale-independent pixel) unit for font sizes as it allows the font size to scale for the user based on their settings
- Iconography
  - System icons are bundled in Android Studio https://m2.material.io/design/iconography/system-icons.html#design-principles

And much more to explore.

Launcher Icons
More details on creating launcher icons.
https://developer.android.com/guide/practices/ui_guidelines/icon_design_adaptive

- Launcher(app) icons represent your app. They appear on the home screen, settings, sharing, and can also be used to represent shortcuts into your app
  - Legacy launcher icons Android 7.1(API 25 and before) 48x48dp

    - Adaptive icons were introduced in Android 8.0(API 26) which display as different shapes as needed. Each OEM provides a mask which is used to render the icons. You can control the look of your adaptive launcher icon by defining 2 layers, consisting of a background and a foreground.
        - Both layers must be sized at 108 x 108 dp.
        - The inner 72 x 72 dp of the icon appears within the masked viewport.
        - The system reserves the outer 18 dp on each of the 4 sides to create visual effects, such as parallax or pulsing.
    - We'll use the Image Asset Studio to create all our icons
- Android projects include default icons - ic_launcher.webp and ic_launcher_round.webp
- Launcher icons go into density specific res/mipmap folders (i.e. res/mipmap-mdpi)
- If you only include the higher resolution version of the icon Android will generate the lower resolutions
- There are also required icons for the action bar, dialog & tab, notifications, and small contextual
- Tablets and other large screen devices request a launcher icon that is one density size larger than the device's actual density, so you should provide your launcher icon at the highest density possible.

Asset Studio
https://developer.android.com/studio/write/image-asset-studio.html
Image Asset Studio that helps you generate your own app icons from material icons, custom images, and text strings. It generates a set of icons at the appropriate resolution for each pixel density that your app supports. Image Asset Studio places the newly generated icons in density-specific folders under the res/ directory in your project. At runtime, Android uses the appropriate resource based on the screen density of the device your app is running on.

- If you only include the higher resolution version of the icon Android will generate the lower resolutions
- There are also required icons for the action bar, dialog & tab, notifications, and small contextual
- Tablets and other large screen devices request a launcher icon that is one density size larger than the device's actual density, so you should provide your launcher icon at the highest density possible.

Launch/Splash Screen
https://developer.android.com/develop/ui/views/launch/splash-screen
Splash screens were not officially supported before Android 12 (API 31). Many apps implemented them but there were issues with the various methods.
Starting in Android 12 Android will automatically show an into-app motion at launch, a splash screen which showing your app icon (as defined in the AndroidManifest file), and a transition to your app.
The launch screen will only be shown when an app is launched from not running to running.
You can use the SplashScreen API to customize the splash screen.

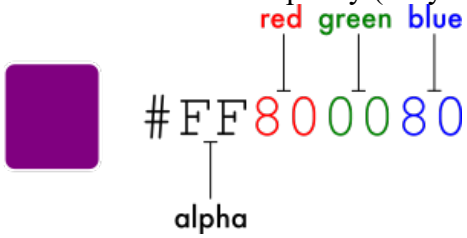
Material Design for Jetpack Compose
Jetpack Compose offers an implementation of Material Design by providing the components and theming needed to implement Material.
https://developer.android.com/jetpack/compose/designsystems/material

Material Components
https://m2.material.io/components?platform=android
Material Components are interactive building blocks for creating a user interface.

Material Theming
https://m2.material.io/design/material-theming/implementing-your-theme.html#color
A Material Theme is made up of color, typography, and shape attributes.

Color
https://developer.android.com/reference/kotlin/androidx/compose/ui/graphics/Color
Colors are created using the Color class. The Color class defines a color using 4 components packed in a single 64 bit long value. One of these components is always alpha while the other three components depend on the color space's color model.

- The most common color model is RGB which represents red, green, and blue values
- Color, in the Android system, is represented by a hexadecimal (hex) color value. A hex color code starts with a pound (#) character and is followed by six letters and/or numbers that represent the red, green, and blue (RGB) components of that color. The first two letters/numbers refer to red, the next two refer to green, and the last two refer to blue.
    - #FF is 100% opacity (fully opaque)




```
val Red = Color(0xff800080)
val Blue = Color(red = 0f, green = 0f, blue = 1f)
```

- Can also use defined properties such as Red, Blue, etc.
  https://developer.android.com/reference/kotlin/androidx/compose/ui/graphics/Color#public-companion-properties
  ```
  val Red = Color(Color.RED)
  ```

Material Design has predefined slots for color.

- Primary is used in components and elements, like app bars and buttons.
    - onPrimary is the color of the text that appears on your primary color
    - primary variant can also be used to complement and provide accessible options for your primary color
- Secondary colors are used as accents on components, such as FABS and selection controls.
    - onSecondary is the color of the text that appears on your secondary color

    - secondary variant can also be used to complement and provide accessible options for your secondary colors

- Background color is found behind scrollable content
    - onBackground is the color of the text that appears on your background color

- Surface colors map to components such as cards, sheets, and menus
    - onSurface is the color of the text that appears on your surface color
- Error color indicates errors in components, such as text fields
    - onError is the color of the text that appears on your error color

Certain components are automatically mapped to color slots. If you don't define any of these colors, the default Material Design color will be used. You can always override a color for a specific element in your app.

The Color class also provides builder functions to create a set of light (`lightColors()`) and dark colors (`darkColors()`) to easily support dark mode.

You can then use your defined colors in your Material theme and access them using `MaterialTheme.colors.`

Dark mode
Your app should support dark mode (API 29 and higher) to handle the case of a device using a dark theme.
Dark mode

- reduces battery usage
- can improve visibility
- easier to use in a low-light environment

`isSystemInDarkTheme()` returns a Boolean which is `true` if the system is considered to be in 'dark theme'.

Typography
https://m2.material.io/design/typography/the-type-system.html#applying-the-type-scale
Material has 13 predefined text categories that make up the type scale.
https://developer.android.com/jetpack/compose/designsystems/material#typography
Compose defines a type system with typography, text style, and font related classes. These can be accessed via `MaterialTheme.typography`

- Typography()
  https://developer.android.com/reference/kotlin/androidx/compose/material/Typography
- TextStyle()
  https://developer.android.com/reference/kotlin/androidx/compose/ui/text/TextStyle
- Font
  https://developer.android.com/reference/kotlin/androidx/compose/ui/text/font/package-summary
    - FontFamily() creates font families to use in Compose

Shape
https://developer.android.com/jetpack/compose/designsystems/material#shape
Material Design defines a shape system with 3 sizes of components – small, medium, and large. Compose implements the shape system with the Shapes class. These are accessed via `MaterialTheme.shapes`.
https://developer.android.com/reference/kotlin/androidx/compose/material/Shapes

There are two types of shapes provided

- rounded shapes have curved corners
    - Components such as cards, menus, snackbars, tooltips, dialogs, and buttons all use 4dp rounded corners as the default.
    - Components with square shapes have 0dp rounded corners as the default, such as full-screen bottom sheets with square, or 90-degree angle, corners.
- cut shapes have angled corners
    - Component corners can be straight cut shapes which are always at a 45-degree angle. These corners can be different lengths, measured from the 0dp rounded corner, going along the outline of the shape.

Shape sizes can be set as absolute or percentage.

- Absolute size refers to having a specific value, such as 2dp. When a corner radius or cut length has an absolute size, it remains the same regardless of the component's height.
- Percentage is determined as the percentage of the height of the component so the corner shape will change as the component height changes.

Components are grouped into size categories. You can supply values to all components in a category. You can also override these values for a specific component.

There's a shape customization tool you can use to experiment with shapes and generate different shapes.
https://m2.material.io/design/shape/about-shape.html#shape-customization-tool

HelloAndroid (theme)
The Compose template that we used to create the project defines a theme and its components, which sets the color, typography, and shape of the components that make up the app.
In the java directory open your project directory (com.example.helloandroid) and go into ui.theme. There you will see Kotlin files for the theme, color, shape, and text.

Starting in the Color.kt file you will see the default colors defined using the Color class and assigned to variables that store these objects.

Type.kt defines a variable Typography which is a Typography object with a font family, font weight, and font size.

Shapes.kt defines a variable Shapes which is a Shapes object with values for small, medium, and large shapes.

In Theme.kt file you will see all of these used to define the theme of the app.
Two variables are defined for light and dark color palettes, each defining primary, primary variant, and secondary colors.
A composable function creates a theme using the name of your project as the name of the theme. It uses the Boolean returned by `isSystemInDarkTheme()` to set the palette for colors. Then it sets the MaterialTheme using colors and the variables defined in the other Kotlin files to set the shapes and typography. Content is set to a default composable assigned to the content parameter.

By default in MainActivity.kt `HelloAndroidTheme()` is the first line in `setContent` to provide the Material Theming to the whole app. It's also in `DefaultPreview()` as well. (we had removed these).

We could just change the default values and theme or create a new one. We'll create a new theme called CMUTheme and use it in our app so we can go through that process.

Color
Using the Material Color tool https://www.material.io/resources/color pick out some colors for your app.
In Color.kt define some new variables with the colors you picked out.

```
val CMUred = Color(0xffc62828)
val CMUredvariant = Color(0xfff05545)
val CMUreddark = Color(0xff8e0000)
```

In Theme.kt define a new palette.

```
private val CMULightColorPalette = lightColors(
    primary = CMUred,
    primaryVariant = CMUredvariant,
    onPrimary = Color.White,
    secondary = CMUreddark
)
```

Now let's create a new composable function for a new theme.

```
@Composable
fun CMUTheme(darkTheme: Boolean = isSystemInDarkTheme(),
content: @Composable () -> Unit){
    MaterialTheme(
        colors = CMULightColorPalette,
        typography = Typography,
        shapes = Shapes,
        content = content
    )
}
```

In MainActivity.kt update the setContent{} block to use the theme. It should be the first thing in setContent{} and other composables are called from the theme.

```
setContent {
    CMUTheme() {
        Greeting()
    }
}
```

We also need to add it to our preview composable.

```
@Preview(showBackground = true)
@Composable
fun DefaultPreview() {
    CMUTheme(darkTheme = false) {
        Greeting()
    }
}
```

You should see that the button is now using our new red color and if you run it in the emulator the text field label is also red. That's because these components use the primary color by default. We can also use these colors for components that don't use them by default.
In Greeting() let's change the background color of the Box from black to one of our colors.

```
Box(
    contentAlignment = Alignment.Center,
    modifier = Modifier
        .fillMaxWidth()
        .height(60.dp)
        .background(CMUreddark)
)
```

When you run it what do you notice? The default color of the text is black and that does not have enough contrast on the dark red for it to be accessible. So let's change that to be white. We don't need to define the color white since it's a predefined color. In MessageText() can change the color used for the Text composable.

```
Text(
    stringResource(R.string.greeting) + " " + newName,
    color = Color.White,
    fontSize = 24.sp,
    textAlign = TextAlign.Center
)
```

Shape
Many components use the defined shapes by default. For example, Button and TextField are in the small shape group.
See the shape scheme reference for the complete mapping.
https://m2.material.io/design/shape/applying-shape-to-ui.html#shape-scheme

In Shape.kt define a new variable for a new Shapes object. We'll change the rounded corner for the small shape to something much larger so we can easily see the change.

```
val CMUShapes = Shapes(
    small = RoundedCornerShape(20.dp),
    medium = RoundedCornerShape(4.dp),
    large = RoundedCornerShape(0.dp)
)
```

In Theme.kt update the CMUTheme to use this new object for shapes.

```
MaterialTheme(
    colors = themeColors,
    typography = Typography,
    shapes = CMUShapes,
    content = content
)
```

In preview or the emulator you'll see the 4 sides of the button and the top edges of the text field have a larger curve now. A value of 12 probably looks better.
If you don't want all the corners to use the same value you can specify different values.

```
small = RoundedCornerShape(
    topStart = 12.dp,
    topEnd = 12.dp,
    bottomEnd = 12.dp,
    bottomStart = 12.dp
)
```

You can also use a percentage value by not specifying .dp.

```
    small = RoundedCornerShape(20)
```

You could also use CutCornerShape() but that would look weird for our button and text field.

You probably noticed that the text field only rounds the top edges. That's how it's defined in Material Design. You can instead use the clip modifier to clip the entire text field and give it a rounded corner shape.

```
.clip(RoundedCornerShape(12.dp))
```

If you want to change the buttons corner radius but not the text field's radius then you shouldn't update the shape used for all components. Instead apply it to only your button (and leave the small shape at the default of 4.dp)

```
Button(onClick= clicked, shape = RoundedCornerShape(20.dp) ) {}
```

Type
The type system has some pre-defined styles that you can use and customize.
Hierarchy is communicated through differences in font weight (Light, Medium, Regular), size, letter spacing, and case.

In Type.kt define a variable for a new Typography object.
First we'll make the font size for body1 a little larger.

```
val CMUTypography = Typography(
    body1 = TextStyle(
        fontFamily = FontFamily.Default,
        fontWeight = FontWeight.Normal,
        fontSize = 24.sp
    )
)
```

In Theme.kt update MaterialTheme to use this new typography object.

```
MaterialTheme(
    colors = themeColors,
    typography = CMUTypography,
    shapes = CMUShapes,
    content = content
)
```

In MessageText() we had hard coded the fontSize for the text, let's remove that so our text styles are used.

Use the interactive preview or run it in the emulator to see which composables use body1.
By comparison the button text now looks small so let's change that.
We could use one of the heading styles, say h2, and make the text a bit larger and bold. Add that to CMUTypography.

```
    h2 = TextStyle(
        fontFamily = FontFamily.Default,
        fontWeight = FontWeight.Bold,
        fontSize = 18.sp
    )
```

Then we could use that style for the text of the button.

```
fun SayHi(clicked: () -> Unit){
    Button(onClick= clicked) {
        Text(
            stringResource(id = R.string.buttonHello),
            style = MaterialTheme.typography.h2
        )
    }
}
```

This is ok but if we want this for all of our buttons we can just assign this style to all buttons. In our CMUTypography object we can define a style for button.

```
    button = TextStyle(
        fontFamily = FontFamily.Default,
        fontWeight = FontWeight.Bold,
        fontSize = 18.sp
    )
```

Remove the style you added to the button text in SayHi() and you'll see that the button text is now using the style you defined for button in Type.kt.

You can also download fonts from https://fonts.google.com and add them to a res/font package (you need to create if it's not present) and use them for the fontFamily parameter when creating a TextStyle.

Dark theme
Your apps should also support dark theme. Here are the steps needed.
Select and define colors for dark theme and add them to Color.kt.
In Theme.kt add a dark color palette. Update your theme to set your theme colors to a palette based on the darkTheme Boolean passed into your theme.

```
@Composable
fun CMUTheme(darkTheme: Boolean = isSystemInDarkTheme(),
content: @Composable () -> Unit){
    val themeColors = if (darkTheme) {
        CMULightColorPalette  //use dark color palette here
    } else {
        CMULightColorPalette
    }
    MaterialTheme(
        colors = themeColors,
        typography = CMUTypography,
        shapes = CMUShapes,
        content = content
    )
}
```

To test define a second preview that sets darkTheme to true.

```
@Preview(showBackground = true)
@Composable
fun DarkDefaultPreview() {
    CMUTheme(darkTheme = true) {
        Greeting()
    }
}
```

In the emulator or device you can go into Settings | Display and toggle dark theme on.
In our example you won't see a difference in dark theme because we haven't set up a dark palette.

Obviously we could do a lot more but this gives you an idea of how you can use Material theming and its components in your app.

Launcher icons
https://developer.android.com/studio/write/image-asset-studio.html
Add your own launcher icon to your app.

Select the res folder right-click and select New > Image Asset
Or File > New > Image Asset.
Name: ic_scotty

Icon type: Launcher Icons (Adaptive and Legacy)
Use Adaptive and Legacy if you're supporting Android 8, otherwise you can just do Legacy.
Name: leave as ic_launcher so you don't need to change it in the Android_manifest.xml file

Foreground Layer:
Layer Name: ic_launcher_foreground

- Select Image to specify the path for an image file.
- Select Clip Art to specify an image from the material design icon set.
- Select Text to specify a text string and select a font.

Scaling – trim and resize as needed.

Background Layer:
Layer Name: ic_launcher_background

- Asset Type, and then specify the asset in the field underneath. You can either select a color or specify an image to use as the background layer.

Scaling – trim and resize as needed.

Next
Res Directory: main
Output Directories: main/res

Finish
Image Asset Studio adds the images to the mipmap folders for the different densities.

You can see the various launcher files created in the mipmap folder.

Now run your app and look at your launcher icon by clicking the right button, or go to the home screen.

To use these new launcher icons you need to go into the Android_manifest.xml file and change the icon and roundIcon attributes to use the new icons.

```
android:icon="@mipmap/ic_scotty"
android:roundIcon="@mipmap/ic_scotty_round"
```

Now run your app and look at your launcher icon by clicking the right button or go to the home screen.