Advanced Mobile Application Development
Week 10: Navigation Principles

Mobile in Context: Design Principles of Flow and Navigation
https://www.youtube.com/watch?v=OZRczPw1BBw Material Design Navigation 12:25-18:30

Navigation and Flow
https://material.io/design/navigation/understanding-navigation.html
Understanding the different types of navigation available on Android is crucial to creating an app that's easy and intuitive to use. Keep in mind the goal of your app, its tasks, content, and target user.
There are three navigational directions:

- Forward navigation – moving between screens to complete a task
    - Downward in an app's hierarchy, from parent (higher level of hierarchy) to child (lower level of hierarchy)
        - We will look at how to implement this type of navigation using recycler views on Thursday
    - Sequentially through a flow, or sequence of screens (cards)
    - Directly from one screen to another (buttons or search)
    - Gestural navigation
        - Swipe or drag
- Reverse navigation – moving backwards through screens
    - reverse chronological navigation using the back button (within one app or across different apps)
    - hierarchically with up navigation from a child screen to their logical parent screen (within an app)
- Lateral navigation – moving between screens at the same level of hierarchy.
    - An app's primary navigation component should provide access to all destinations at the top level of its hierarchy.
        - Apps with two or more top-level destinations should provide lateral navigation

Lateral Navigation
Navigation drawer https://material.io/design/components/navigation-drawer.html

- Apps with 5+ top-level destinations or two+ levels of navigation hierarchy
- Quick navigation between unrelated destinations
- Types
    - Standard – allows interaction with both screen content and the drawer at the same time. They can be used on tablet and desktop, but they aren't suitable for mobile due to limited screen size.
    - Modal – elevated above most of the app's UI and don't affect the screen's layout grid. They block interaction with the rest of an app's content. Used on mobile where screen space is limited.
    - Bottom – modal drawers used with bottom app bars that are anchored to the bottom of the screen instead of the left edge.
- Creates history for the back button
- Reduces visibility for infrequently visited destinations
- Supports nested navigation for a deep navigational structure

- As phones have gotten larger the nav drawer hamburger menu is harder to access as 49% of the time users use their right thumb to interact with their device. Using bottom navigation increases reachability and the use of your app's core features.

Bottom navigation bar https://material.io/design/components/bottom-navigation.html

- Apps with 3-5 top level destinations (no "more" …)
- Frequent switching between views
- Mobile or tablet use
- Should usually persistent across screens to provide consistency
- Must include icons. Also provide text labels if needed to clarify the meaning
- When navigating to a destination's top-level screen any prior user interactions and screen states are reset (although you could implement saving state)
- Ergonomic, easy to switch between views
- Cross-fade animation is suggested with bottom navigation
- Not suggested along with a navigation drawer or tabs
- Doesn't create history for the back button

Tabs https://material.io/design/components/tabs.html#tabs

- Organize 2+ sets of data that are related, same level of hierarchy, siblings
- Easily switch between a few different categories of content
- Tap or swipe to navigate tabs
- Tabs exist inside the same parent screen so tabs don't create history for the system back button
- Also provides additional lateral navigation when paired with a top-level navigation component

Reverse Navigation
https://material.io/design/navigation/understanding-navigation.html#reverse-navigation
Back:
When your app is launched, a new task is created and becomes the base destination of the app's back stack.
The top of the stack is the current screen, and the previous destinations in the stack represent the navigation history. The back stack always has the start destination of the app at the bottom of the stack. Operations that change the back stack always operate on the top of the stack, either by pushing a new destination onto the top of the stack or popping the top-most destination off the stack. Navigating to a destination pushes that destination on top of the stack.
The Navigation component manages all of your back stack ordering for you, though you can also choose to manage the back stack yourself.

The Back button is in the system navigation bar at the bottom of the screen and is used to navigate in reverse-chronological order through the history of screens the user has recently worked with. When you tap the Back button, the current destination is popped off the top of the back stack, and you then navigate to the previous destination. It also handles the following:

- Dismisses floating windows such as dialogs or pop-ups
- Dismisses contextual action bars
- Removes the highlight from selected items
- Hides on screen keyboard

Upward:

Upward navigation is a way for the user to navigate to the logical parent screen in the app's hierarchy. This differs from back navigation because it is not based on the user's history or path, but a defined parent activity. Up navigation will therefore never exit the app.

App Navigation components

App bar

The App Bar is a consistent navigation element that is standard throughout modern Android applications. https://www.material.io/components/app-bars-top (similar to navigation bars on iOS https://material.io/design/platform-guidance/cross-platform-adaptation.html#cross-platform-guidelines)
The top app bar provides content and actions related to the current screen. It's used for branding, screen titles, navigation, and actions.
The App Bar can consist of:

- Consistent navigation (including navigation drawer)
- An application icon
- Up navigation to logical parent
- An application or activity-specific title
- Primary action icons for an activity
- Overflow menu

Terminology

The app bar is often called the action bar, but there is a difference. App bar is the general name, it can be implemented as an action bar or a toolbar.
Beginning with API level 11 all activities that use the default theme have an ActionBar as an app bar. However, app bar features have gradually been added to the native ActionBar over various Android releases. As a result, the native ActionBar behaves differently depending on what version of the Android system a device may be using.
The most recent features are added to the support library's version of Toolbar, and they are available on any device that can use the support library. Because of this you should use the support library's Toolbar class to implement your activities' app bars. Using the support library's toolbar helps ensure that your app will have consistent behavior across the widest range of devices.

Menus

https://developer.android.com/guide/topics/ui/menus
Android has three different types of menus:

1. Options menu: this is a menu of actions for an activity that is accessed in the app bar
2. Context menu: a floating menu that is often presented on a long click event. It provides actions that affect the selected content
3. Popup menu: a menu that pops up with a vertical list of items. Actions in a popup menu should **not** directly affect the corresponding content—that's what contextual actions are for. These are used for extended actions that relate to regions of content in your activity.

All menus should be created as a menu resource xml file that are inflated in your activity code.
Thought should be put into organizing and naming menus to reflect your user's mental model and be user friendly.

Floating Action Button (FAB)

https://material.io/components/buttons-floating-action-button

- Used to represent the most common or primary action in the screen
    - Add
    - Share

    - Play/pause
- The action associated with the FAB must be for the whole view, not just one part of it like one row
- There should usually only be one FAB on a screen

Fragments
https://developer.android.com/guide/fragments
A Fragment represents a modular and reusable portion of an app's UI. A fragment must be *hosted* by an activity or another fragment, they cannot live on their own.
Important fragment concepts:

- A fragment defines and manages its own layout
- A fragment has its own behavior with its own lifecycle callbacks.
- A fragment can handle its own events
- You can add or remove fragments in an activity while the activity is running.
- You can combine multiple fragments in a single activity to build a multi-pane UI.
- A fragment can be used in multiple activities.
- The fragment life cycle is closely related to the Activity lifecycle but not exactly the same
    - https://developer.android.com/guide/fragments/lifecycle
- When the activity is paused, all the fragments available in the activity will also be stopped.
- A fragment can implement a behavior that has no user interface component.

Navigation Architecture Component
https://developer.android.com/guide/navigation
The Navigation Architecture Component is a new component (part of Android Jetpack) that helps us implement navigation in our application. The Navigation component consists of three key parts:

- Navigation graph that includes all the destinations and actions in your app
    - destinations are the different areas users can navigate to in the app
    - actions represent the paths users can take between the destinations
- Navigation Host that is an empty container where destinations are swapped in and out as a user navigates through your app.
    - NavHost (NavHostFragment) is the default implementation that handles swapping fragment destinations
- NavController to manage app navigation and show the appropriate destination in the NavHost
    - Each NavHost has its own corresponding NavController

The Navigation component handles all fragment transactions, up and back navigation, and even provides basic animations and transitions for navigating.

Navigation Drawer
To see a simple example of a navigation drawer we'll create a new project using the Navigation Drawer template.
New phone/tablet app
Navigation Drawer Activity template
Name: NavDrawer
Package name: the fully qualified name for the project
Language: Kotlin
Minimum SDK: API 21 (21 is the minimum API for Material Design)
Leave legacy libraries unchecked (using these restricts usage of some newer capabilities)

Finish

Run the app to see what the template gave you.
Note that the drawer icon is displayed on all top-level (root level) destinations your app. They do not display an Up button in the app bar.

You can create everything to implement a nav drawer manually, but there are a lot of steps, so the template is really nice. (many tutorials walk you through this). Let's see what was created for us.

activity_main.xml is an instance of the DrawerLayout component which is a container for a drawer view.

- The fitsSystemWindows attribute is set to true so the drawer expands to fill the available space. The <include> refers to the app_bar_main layout which has all of the other content that appears in the main activity.
- app_bar_main.xml includes the CoordinatorLayout that holds a AppBarLayout, Toolbar and FAB.
    - It also includes the content_main.xml file which has a fragment of type NavHostFragment which uses the navigation graph navigation/mobile_navigation
- NavigationView is the navigation component that's displayed when you swipe in from the left.
    - the headerLayout attribute which points to the nav_header_main.xml layout file for the drawer's header view
    - the menu attribute points to activity_main_drawer where all the nav items are defined

In activity_main_drawer.xml the checkableBehavior attribute is defined for the entire group and it takes either of the three values:

- single: Only one item from the group can be checked (used to control navigation)
- all: All items can be checked (checkboxes)
- none: No items are checkable

Open navigation/mobile_navigation and in design mode you'll see the navigation editor. Under destinations you can see the host and graphs. The host is listed in content_main.xml and handles swapping the fragments during navigation. The Graphs are the views that we are able to navigate into. Each Graph will be swapped in and out to the host when the navigation happens.
If you click on a graph you'll see it's a fragment with a label from strings.xml, an id which is the identifier we'll use to reference it (such as when switching between graphs), and a class which has also been created for us.

Let's update it with a custom theme using the Material UI color picker https://material.io/resources/color
I went into custom and used CU Gold #CFB87C as the primary and CU Dark Gray #565A5C as the secondary. (source: https://www.colorado.edu/brand/how-use/color)
This produced the following colors that I added into colors.xml

```
<color name="primaryColor">#cfb87c</color>
<color name="primaryLightColor">#ffeaac</color>
<color name="primaryDarkColor">#9d884f</color>
<color name="secondaryColor">#565a5c</color>
<color name="secondaryLightColor">#838789</color>
<color name="secondaryDarkColor">#2d3133</color>
<color name="primaryTextColor">#000000</color>
<color name="secondaryTextColor">#ffffff</color>
```

In themes.xml define a new style with a MaterialComponents parent theme.

```
<style name="CUTheme" parent="Theme.MaterialComponents.DayNight">
    <item name="colorPrimary">@color/primaryColor</item>
    <item name="colorPrimaryDark">@color/primaryDarkColor</item>
    <item name="colorOnPrimary">@color/primaryTextColor</item>
    <item name="colorSecondary">@color/secondaryColor</item>
    <item name="colorSecondaryVariant">@color/secondaryDarkColor</item>
    <item name="colorOnSecondary">@color/secondaryTextColor</item>
</style>

<style name="CUTheme.NoActionBar">
    <item name="windowActionBar">false</item>
    <item name="windowNoTitle">true</item>
</style>

<style name="CUTheme.AppBarOverlay"
parent="ThemeOverlay.MaterialComponents.Dark.ActionBar" />

<style name="CUTheme.PopupOverlay"
parent="ThemeOverlay.MaterialComponents.Light" />
```

Remember to update the AndroidManifest.xml file with the new theme name (two places in this project).
Run it, how does it look?
Everything is using the new theme except the header of the nav drawer.
If you look in nav_header_main.xml you'll see there's a linear layout that determines the layout of the header.
The layout has an ImageView and two TextViews. Change or remove those to get the layout you want.
To change the image you can add images into the drawable folder. Mipmap is only for launcher icons.
wrap_content will use the size of the image. If the image is large enough there might not be room to show the text views so either change the image dimensions or the layout height and width of the image.
To change the text in the TextView you'll need to change the strings.xml file.
Note that the various dimensions are stored in values/dimens.xml so you'll need to change them there if needed.

According to Material Design if side navigation such as a navigation drawer exists, include "Settings" below all other items (except Help & Feedback). Otherwise place it in the toolbar menu.
https://www.material.io/design/platform-guidance/android-settings.html#placement

So let's take the Settings item that's in the options menu and add it to the navigation drawer instead.

First add an icon in res/drawable right click and pick new Vector Asset. (Use Image Asset to create various png files for older Android versions).
For asset type pick clip art and click on the clip art image to select an icon provided.
(Under Action I picked settings)
Name: ic_menu_settings (to match the naming convention)
In activity_main_drawer.xml add the icon to the item.

In menu/activity_main_drawer.xml add a new item

```
<item
       android:id="@+id/nav_settings"
```

```
        android:icon="@drawable/ic_menu_settings"
        android:title="@string/action_settings" />
```
Note that the order of the items in the menu is controlled by the order in which they're declared.

Those are all the steps you need to setup the drawer. The key thing to remember about the navigation drawer is that it's a wrapper around the rest of your activity layout. The drawer layout is the root element which contains the app bar layout and then the navigation view component which includes the header layout and the drawer menu. You define your header in a layout file, and you define your menu in a menu resource file, just like you do for options and pop-up menus.

In MainActivity in the onCreate method we get the reference to the drawer layout and the navigation view.
The NavController instance gets access to the nav_host_fragment defined in the nav graph which is used to manage the app navigation.
AppBarConfiguration(setOf()) connects the fragments in the nav graph with the drawer layout.
The NavigationUI class contains static methods that manage navigation with the top app bar and the navigation drawer.

- setupActionBarWithNavController() adds navigation support to the default action bar by binding it to the nav controller
- setupWithNavController() binds the navigation view and nav controller to handle the navigation from the drawer.

Then onSupportNavigateUp() is implemented to handle Up navigation when not at a top level fragment showing the drawer icon.

Let's implement the item Settings that we just added to the nav drawer.
First we'll create a fragment for Settings. Following the project's organization create a new package in java/com.example.navdrawer/ui by right clicking on that folder, select new | package and name it settings.
Right click on the package, select new | Fragment | Fragment (Blank)
Fragment name: Settings
Fragment layout name: fragment_settings
Source language: Kotlin
(we're not going to worry about the ViewModel the others have today)

You can go into the strings file and change the string created for this new fragment.
```
<string name="hello_settings_fragment">This is a settings fragment</string>
```

In the layout file fragment_home.xml change the string resource for the TextView to match your strings file.
In design mode you can right-click on FrameLayout and convert it to a ConstraintLayout like the others. Add constraints for the textview so it's centered like the other textviews in the other fragments.

Now let's add the navigation. Open the navigation graph navigation/mobile_navigation.xml.
In design mode find the icon to add a new destination (it's above all the destinations with a green +)
Click to add a new destination and pick fragment_settings.
Select the new destination and look at the attributes.

- Type is Fragment
- You can update Label to @string/action_settings

- The id for this destination MUST match the id you gave the menu item in menu/activity_main_drawer.xml as that's how they get connected. There we used nav_settings so make that the id here as well.
- Name should be the class you just created Settings (com.example.navigationdrawer.ui.settings)

Run the app and try out the Settings menu.
Why is that the only fragment that shows the up navigation instead of the drawer icon?
In MainActivity when we instantiate mAppBarConfiguration to set up the drawer layout we haven't added the new settings menu. Add that in the list with the others.

```
appBarConfiguration = AppBarConfiguration(setOf(
        R.id.nav_home, R.id.nav_gallery, R.id.nav_slideshow,
        R.id.nav_settings), drawerLayout)
```

Options menu
Lastly, let's look at the options menu. We already saw that the menu items are defined in menu/main.xml.
Typically we wouldn't want Settings in both the navigation drawer and the menu, but since it's there let's just use it.
In MainActivity onCreateOptionsMenu() inflates the menu which adds it to the action bar for this activity.
When the user selects an item from the options menu the onOptionsItemSelected() method is automatically called and the MenuItem selected is passed in.
Let's implement onOptionsItemSelected() and use NavigationUI to handle the menu items as well.

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    val navController =
findNavController(R.id.nav_host_fragment_content_main)
    return item.onNavDestinationSelected(navController) ||
super.onOptionsItemSelected(item)
}
```

The key is the same – the menu item id MUST match the id of the destination in the navigation graph. We gave settings the id nav_settings so update menu/main.xml so the item has the same id.
Now the options menu should be operational as well.

Notice that we're creating an instance of the NavController twice – in OnCreate() and in onOptionsItemSelected(). To be more efficient we can define this at the class level and initialize it in onCreate(). Then we won't need to do that again in onOptionsItemSelected().

```
private lateinit var navController: NavController
```

Why lateinit? If you define a variable that will be assigned a view you either need to make it a nullable variable, assign the value of null, and treat it as a nullable variable throughout the class or use lateinit. Lateinit lets you defer property initialization. Basically it promises the compiler that the variable won't be left as null but will be initialized before calling any operations on it. onCreate() is run early in the lifecycle so this is a good place to initialize it, and all view instances. This way we don't need to treat it as a nullable variable throughout the class.

Bottom Navigation

The Bottom Navigation template results in a very similar setup.

- activity_main.xml has a bottom navigation view instead of a drawer
- menu xml file defines the bottom nav items
- navigation graph manages the navigation using fragments
- MainActivity class doesn't need to implement onSupportNavigateUp() unless you implement up navigation