Advanced Mobile Application Development
Week 12: Navigation and Fragments

Navigation

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 from parent to child (lists)
  - Sequentially through a flow (cards)
  - Directly from one screen to another (buttons)
- Reverse navigation – moving backwards through screens
  - chronologically using the back button (within one app or across different apps)
  - hierarchically with up navigation (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#navigation-drawer

- 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

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

- Apps with 2 or 3-5 top level destinations (no "more" …)
- Mobile or tablet use
- Should usually persistent across screens to provide consistency
- When navigating to a destination's top-level screen any prior user interactions and screen states are reset(although you could implement saving state)
- 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
- 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

As phones have gotten larger the nav drawer hamburger menu is harder to access the 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.

Navigation Drawer
To see a simple example of a navigation drawer we'll use the Navigation Drawer template.
Create a new project called NavDrawer
Navigation Drawer Activity template
Package name: the fully qualified name for the project
Minimum SDK: API 21 (21 is the minimum API for Material Design)

Run the app to see what the template gave you. The Up button in the top left of the app bar is for the nav drawer. Click or drag from the left to open it.
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. 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 with the AppBarLayout and Toolbar. It also includes the content_main.xml file where you customize the appearance of the activity screen.
The last element is the NavigationView which is the nav drawer component that's actually displayed when you drag in from the left.
The NavigationView has the headerLayout attribute which points to a layout file for the top of the drawer, nav_header_main.xml where you control the appearance of the space at the top of the drawer.
The menu attribute points to activity_main_drawer where all the items in the nav drawer are defined within groups.
checkableBehavior attribute with the value of single keeps an item selected once it's been clicked. This is useful when you're using the items in the nav drawer to control navigation.
Let's take the Settings item that's in the options menu and add it to the navigation drawer instead.

In activity_main_drawer.xml add a new item
```
<item
   android:id="@+id/nav_settings"
   android:title="@string/action_settings" />
```
Note that the order of the items in the menu is controlled by the order of their declaration.

To 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.
```
android:icon="@drawable/ic_menu_settings"
```

In MainActivity at the bottom of the onCreate method we gets the reference to the drawer object, and then the drawer is set up. The toggle that controls when the drawer appears and disappears is defined and then added as a listener for when the user selects an item from the navigation drawer.

Those are all the steps you need to setup the drawer. The key thing to remember about setting up the drawer is that it's a wrapper around the rest of your activity layout. The drawer layout is the root element and then the navigation view component goes at the bottom. 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.

Navigation drawer menu events

MainActivity.java implements NavigationView.OnNavigationItemSelectedListener, and its required method onNavigationItemSelected(MenuItem item). This method is called when an item in the nav draw menu is clicked, and that menu is passed in.
The if statements are there to match the items in the nav drawer.
The call to closeDrawer() at the bottom of the method closes the drawer, otherwise it would be left open.

Let's implement one item in the nav drawer, Settings.
Create a new Activity and use the Settings Activity template.
Name: Settings
Title: Settings
Hierarchical Parent: MainActivity (click …)
Finish

In MainActivity.java in the method onNavigationItemSelected(MenuItem item) I'm going to change the if/else statements to a switch statement. Place the cursor after the if keyword and click on the light bulb to use an intention action by choosing replace if with switch. A switch statement is cleaner here.
I'll add a case statement for R.id.nav_settings and start the Settings Activity. (don't forget the break statement).

```
@Override
public boolean onNavigationItemSelected(MenuItem item) {
    int id = item.getItemId();
    switch (id) {
        case R.id.nav_camera:
            break;
        …
        case R.id.nav_settings:
            Intent intent = new Intent(this, SettingsActivity.class);
            startActivity(intent);
            break;
    }
    DrawerLayout drawer = (DrawerLayout) findViewById(R.id.drawer_layout);
    drawer.closeDrawer(GravityCompat.START);
    return true;
}
```

Now when you run the app and go into the nav drawer and click on Settings the app will navigate to the Settings Activity and close the nav drawer. You can drill down into the subpages of the Setting activity, and then when you click the up button on the action bar it goes back to its hierarchical parent, MainActivity.

Fragments
https://developer.android.com/guide/components/fragments
We've been using Activities in our apps but sometimes you want more flexibility and reusability in your apps. (slide)
Android introduced fragments in Android 3.0 (API level 11), primarily to support more dynamic and flexible UI designs on large screens, such as tablets.
A fragment is a modular section of an activity, which has its own lifecycle, receives its own input events, and which you can add or remove while the activity is running (sort of like a "sub activity" that you can reuse in different activities).
A Fragment represents a behavior or a portion of user interface that enable your app to be more modular, reusable, and adaptable. A fragment must always be hosted in an activity but you can manipulate each fragment independently, such as add or remove them.
Modularity

- Fragments allow you to break your activities up into smaller modular components
- Divide complex activity code across fragments for better organization and maintenance

Reusability

- Separating behavior or UI parts into fragments allow you to share them across multiple activities
- Fragments can easily be reused and adapted for different device sizes, orientation, or other criteria

Adaptability

- Representing sections of a UI as different fragments lets you utilize different layouts depending on screen orientation and size (slide)
- You can one or more fragments embedded in an activity (slide)

Creating Fragments
To create a fragment you subclass the **Fragment** class. The Fragment class has a lot of the same methods as the **Activity** class.

- *Fragment* subclasses require an empty default constructor. Android Studio creates one for you when you use the Fragment template.

A fragment is usually used as part of an activity's user interface and contributes its own layout to the activity.
Create the layout for a fragment

- Use **<fragment>** in your activity's layout file to specify a fragment
- Use **<FrameLayout>** in your activity's layout file to specify an area in your layout where you can add a fragment later programmatically
  - FrameLayout is a type of view group that's used to block out an area on the screen.
- Implement onCreateView() to inflate the fragment layout

Adding a fragment to an Activity
You can then ad a fragment to an activity by either

- Adding the fragment to the activity's layout file
  - When the system creates the activity's layout it instantiates each fragment by calling its onCreateView() method to retrieve the fragment's layout
- While an activity is running you can programmatically add the fragment to an existing ViewGroup.
  - Use the **FragmentTransaction** class to programmatically interact with fragments
    - Get an instance of **FragmentTransaction** from your **FragmentActivity**
    - Perform changes using methods such as **add(), remove(), and replace().**
    - **commit()** commits the changes

Bottom Navigation
To see a simple example of bottom navigation we'll use the Bottom Navigation template.
Create a new project called BottomNav
Bottom Navigation Activity
Package name: the fully qualified name for the project
Minimum SDK: API 21 (21 is the minimum API for Material Design)

Look at the layout file and you'll see the bottom navigation widget. But where are the buttons defined?
app:menu="@menu/navigation"
In the menu resources folder look at navigation.xml and you'll see the three items. They each have an icon that's a drawable resource, a title that's a string resource, and an id. To add more items you just add it to this xml file.

In MainActivity.java the listener is set up for us and the onNavigationItemSelected(@NonNull MenuItem item) method is implemented using a switch off the id. This is where you add the logic for each item, and just add another case for any items you add. If you run it you'll see the different messages written to the TextView when you tap any of the navigation items.

You can also use the Empty template and add the navigation.xml file and the code in MainActivity.java, this just makes it a bit easier.

activity_main.xml
Remove textview
Add FrameLayout (in the palette layouts section) which will act as a container for our fragments that we'll control programatically. Make sure it has an id. Add constraints.

Change the strings.xml and navigation.xml files to reflect your apps content.
You can also change the 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 group work)
Name: ic_group_work
Change navigation.xml to use the new icon.

Now let's add one fragment for each item in the bottom navigation.
File | New | Fragment | Fragment (Blank)
HomeFragment
Check Create Layout
Uncheck include fragment factory methods
Unheck Include interface callbacks
Let the project sync.

Create two more fragments, DashboardFragment and NotificationsFragment, for the other items.
Change the name and text in strings.xml for the textView in each fragment layout as a way to test the loading of fragment that we're about to add. (Change the name because when you add the next fragment the template will try to add the same string name and you'll get an error.)
Update the layout file with the new string name.

```
<string name="home_fragment">Home fragment</string>
<string name="dashboard_fragment">Dashboard fragment</string>
<string name="notifications_fragment">Notifications fragment</string>
```

MainActivity.java
Remove all references to **mTextMessage** since we removed the textView (these should be red at this point anyway).
We'll create a method that will handle the loading of the fragments.

```
private void loadFragment(Fragment fragment){
    if (fragment != null){
        FragmentTransaction transaction = getSupportFragmentManager().beginTransaction();
        transaction.replace(R.id.frameLayout, fragment);
        transaction.commit();
    }
}
```

In onCreate() let's load the first fragment

```
loadFragment(new HomeFragment());
```

Update onNavigationItemSelected(MenuItem item) to load the appropriate fragment.

```
public boolean onNavigationItemSelected(@NonNull MenuItem item) {
    Fragment fragment;
    switch (item.getItemId()) {
        case R.id.navigation_home:
            fragment = new HomeFragment();
            loadFragment(fragment);
            return true;
        case R.id.navigation_dashboard:
            fragment = new DashboardFragment();
            loadFragment(fragment);
            return true;
        case R.id.navigation_notifications:
            fragment = new NotificationsFragment();
            loadFragment(fragment);
            return true;
    }
    return false;
}
```

If you run it now the three items in the bottom navigation should load the appropriate fragment.
It would also be nice if we could change the title in the app bar.

Make sure you have the v7 appcompat library in your Gradle file(Module: app) as a dependency.
implementation **'com.android.support:appcompat-v7:28.0.0'**

Open styles.xml (app/res/values)
You will probably have the following:
<**style name="AppTheme" parent="android:Theme.AppCompat.Light.DarkActionBar**">
Create a new style with a parent that has no action bar.

<style name=" NoActionBarTheme" parent="Theme.AppCompat.Light.NoActionBar">
You can use the colors already defined or go into colors.xml and update or add new colors.
In the AndroidManifest.xml file change the theme to use your new theme.
android:theme="@style/ NoActionBarTheme">
Now when you run your app you should not see an action bar.

Here you'll see the advantage of fragments. Since all fragments are being loaded into the FrameLayout in the activity_main layout file all we need to do is add a toolbar in the activity_main layout.
In the design view drag out a toolbar and add it to the top of the layout. Adjust the other views and constraints accordingly. This might be a bit tricky so first make the frame layout shorter so you can add the toolbar at the top. The toolbar should not be in the frame layout.
For the toolbar I set top, start, and end constraints equal to 0 and made the layout_width match_constraints and layout_height wrap_content.
I changed the top constraint of the frame layout to be to the bottom of the toolbar (instead of parent).
Make sure your toolbar has an ID.

Go into MainActivity.java and add an instance variable for the toolbar at the Activity level.
Toolbar **toolbar**;

In onCreate() get access to the toolbar and then set the action bar to be the toolbar. We'll also set the title for the first fragment.

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    //get toolbar and set it as the app bar
    toolbar = findViewById(R.id.toolbar);
    setSupportActionBar(toolbar);

    //load the first fragment
    getSupportActionBar().setTitle("Home");
    loadFragment(new HomeFragment());
…
}
```

Make sure the import for the toolbar class is the v7 one or you'll get an error that setSupportActionBar() can't be found.

Update each case statement onNavigationItemSelected(MenuItem item) to set the title for each fragment.

```
case R.id.navigation_heroes:
    toolbar.setTitle("Home");
    fragment = new HomeFragment();
    loadFragment(fragment);
    return true;
```

Now when you run it the toolbar's title should change for each navigation item.