# CS 213 SOFTWARE METHODOLOGY

Lily Chang

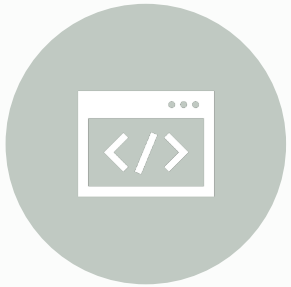
CS Department @ Rutgers New Brunswick

FALL 2023

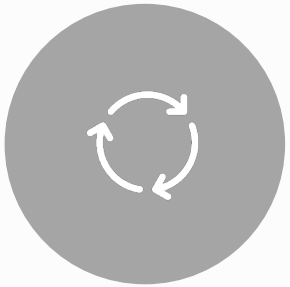
# Fragment

Lecture Note #20

# Fragment

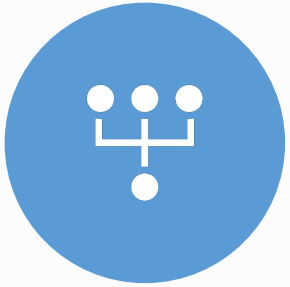


A Fragment represents a reusable portion of your app's UI.



A fragment defines and manages its own layout, has its own lifecycle, and can handle its own input events.

Fragments cannot live on their own--they must be hosted by an activity or another fragment.



The fragment's view hierarchy becomes part of, or attaches to, the host's view hierarchy.

# Create a Fragment

- Fragments require a dependency on the AndroidX Fragment library. You need to include it in the project's file in order to include this dependency. build.gradle

```
dependencies {
    def fragment_version = "1.4.1"

    // Java language implementation
    implementation "androidx.fragment:fragment:$fragment_version"
    // Kotlin
    implementation "androidx.fragment:fragment-ktx:$fragment_version"
}
```

# Fragments – Micro Activities

- Fragments introduce modularity and reusability into your activity's UI by allowing you to divide the UI into discrete chunks.
- Dividing your UI into fragments makes it easier to modify your activity's appearance at runtime.

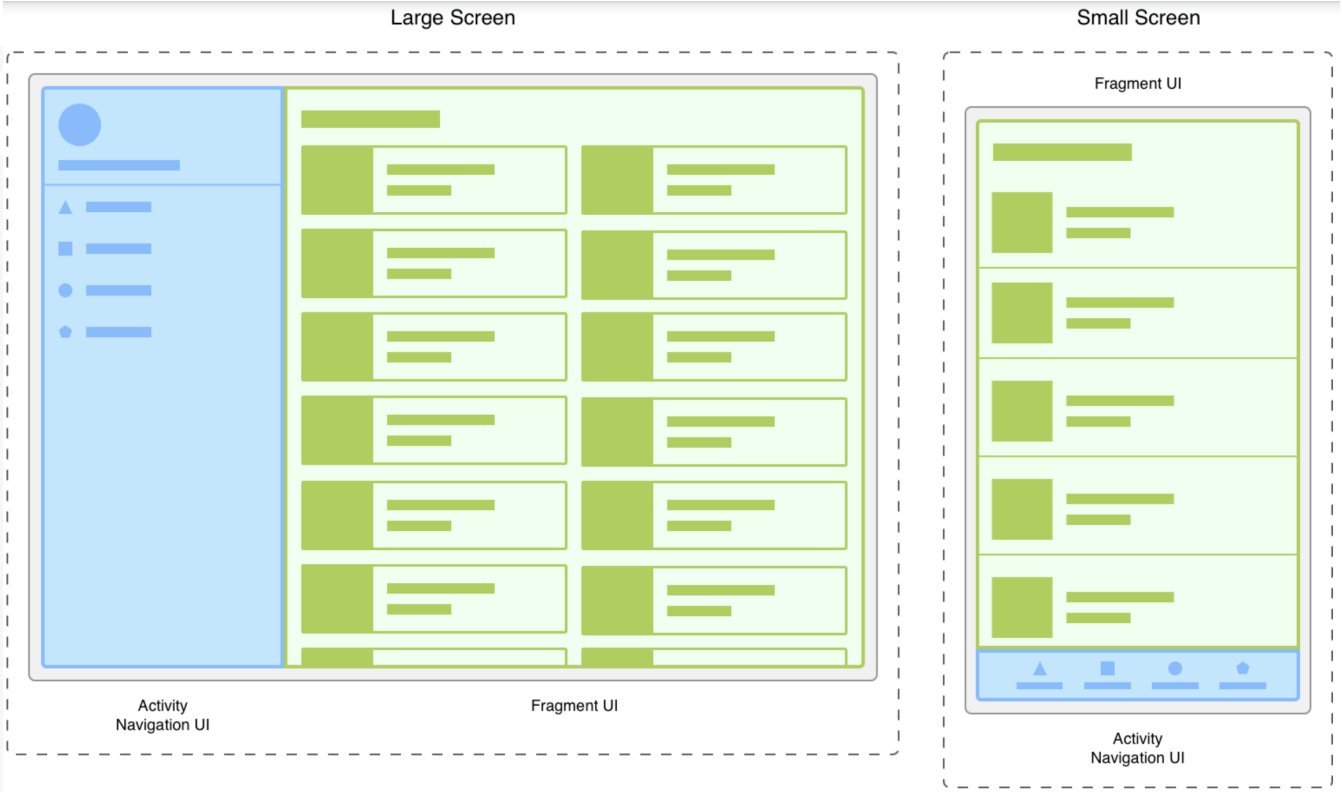
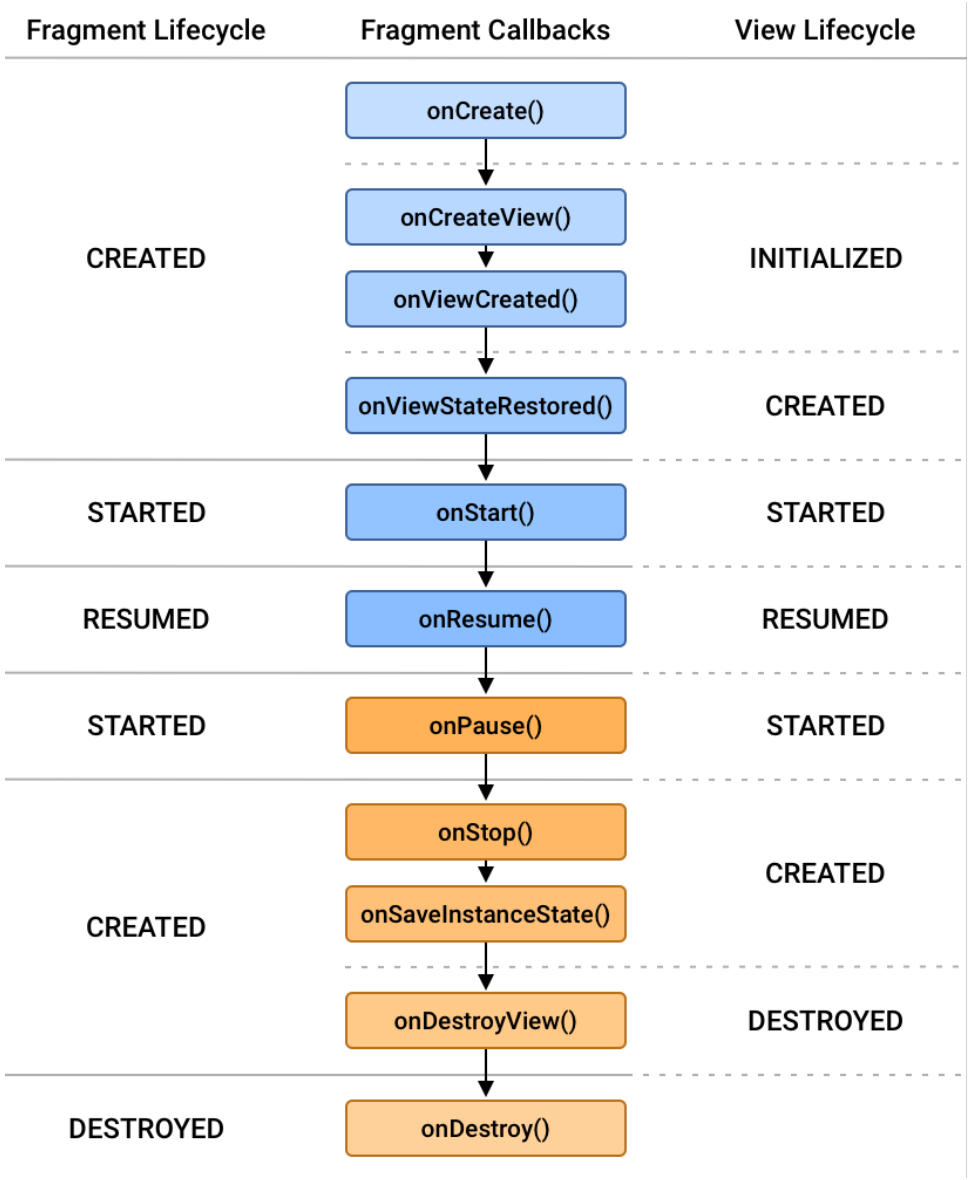



**Figure 1.** Two versions of the same screen on different screen sizes. On the left, a large screen contains a navigation drawer that is controlled by the activity and a grid list that is controlled by the fragment. On the right, a small screen contains a bottom navigation bar that is controlled by the activity and a linear list that is controlled by the fragment.

# Fragment Lifecycle

- For a fragment to transition through the rest of its lifecycle, it must be added to a FragmentManager.
- The FragmentManager is responsible for determining what state its fragment should be in and then moving them into that state.
- Beyond the fragment lifecycle, FragmentManager is also responsible for attaching fragments to their host activity and detaching them when the fragment is no longer in use.



```
class ExampleFragment extends Fragment {
    public ExampleFragment() {
        super(R.layout.example_fragment);
    }
}
```

# Create a Fragment class

- To create a fragment, extend the AndroidX Fragment class, and override its methods to insert your app logic, similar to the way you would create an Activity class.
- To create a minimal fragment that defines its own layout, provide your fragment's layout resource to the base constructor,

# Adding a Fragment to an Activity

- Generally, your fragment must be embedded within an AndroidX FragmentActivity to contribute a portion of UI to that activity's layout.
- You need to add a FragmentContainerView (or FrameLayout) that defines the **location where the fragment should be placed** within the activity's view hierarchy
- The android:name attribute specifies the class name of the Fragment to instantiate. When the activity's layout is inflated, the specified fragment is instantiated, onInflate() is called on the newly instantiated fragment, and a FragmentTransaction is created to add the fragment to the FragmentManager.
- Add a fragment programmatically, follow the link below https://developer.android.com/guide/fragments/create#add-programmatic
- See the sample code posted on Canvas

```
<!-- res/layout/example_activity.xml -->
<androidx.fragment.app.FragmentContainerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/fragment_container_view"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:name="com.example.ExampleFragment" />
```

# FragmentManager / FragmentTransaction

a FragmentTransaction is used to instantiate a fragment and add it to the activity's layout.

While your activity is running, you can make fragment transactions such as adding, removing, or replacing a fragment.

FragmentManager, is the class responsible for performing actions on your app's fragments; you can access the FragmentManager through the getSupportFragmentManager() method.

The FragmentManager manages the fragment back stack. At runtime, the FragmentManager can perform back stack operations like adding or removing fragments in response to user interactions. Each set of changes are committed together as a single unit called a FragmentTransaction.

# Perform a Transaction

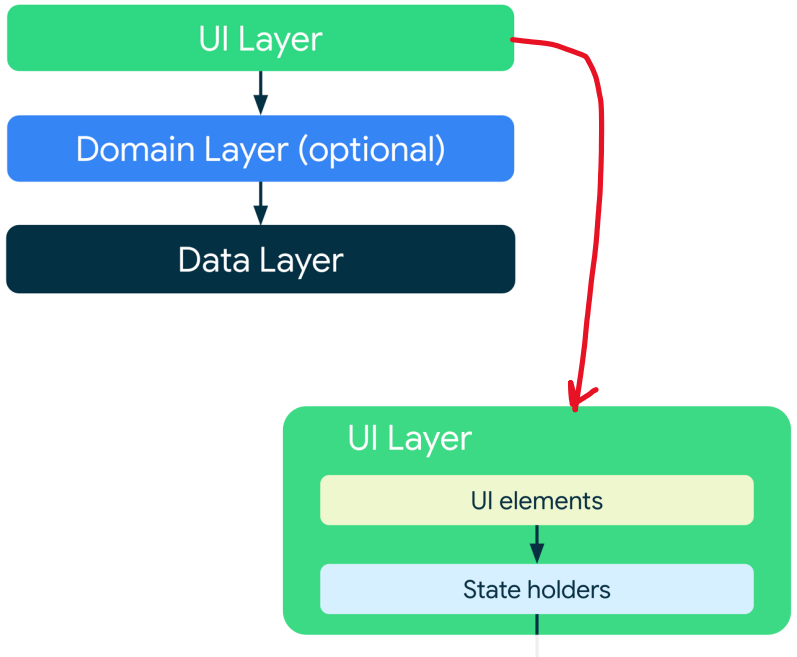
- To display a fragment within a layout container, use the FragmentManager to create a FragmentTransaction. Within the transaction, you can then perform an add() or replace() operation on the container
- For example,

```
FragmentManager fragmentManager = getSupportFragmentManager();
fragmentManager.beginTransaction()
    .replace(R.id.fragment_container, ExampleFragment.class, null)
    .setReorderingAllowed(true) //optimize the transaction
    .addToBackStack("name") // back button to the previous fragment
    .commit();
```

# Android Jetpack

Jetpack is a suite of libraries to help developers follow best practices, reduce boilerplate code, and write code that works consistently across Android versions and devices so that developers can focus on the code they care about.

https://developer.android.com/jetpack/getting-started



# MATERIAL DESIGN

- User Experience (UX) Design
- Resources: https://m3.material.io/

# Testing your app

Local unit tests located at module-name/src/test/java/

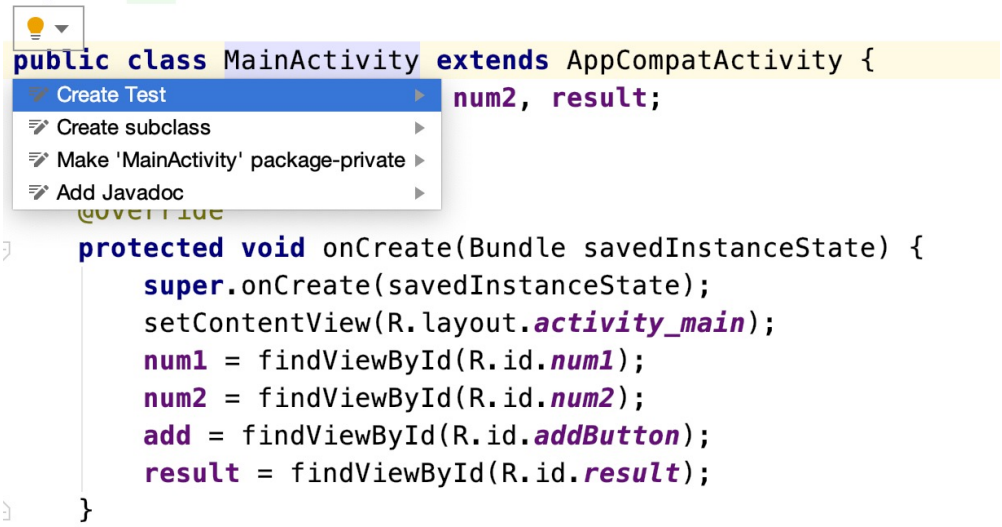
- These are tests that run on your machine's local Java Virtual Machine (JVM)

Instrumented tests located at module-name/src/androidTest/java/

- These are tests that run on a hardware device or emulator
- Use these tests when writing integration and functional UI tests to automate user interaction, or when your tests have Android dependencies that mock objects cannot satisfy.

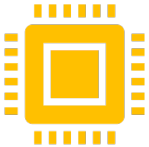
# Add a new test

- Open the Java file containing the code you want to test.
- Click the class or method you want to test, then press Ctrl+Shift+T (⇧⌘T).
- In the menu that appears, click Create New Test.
- In the Create Test dialog, edit any fields and select any methods to generate, and then click OK.
- In the Choose Destination Directory dialog, click the source set corresponding to the type of test you want to create: androidTest for an instrumented test or test for a local unit test. Then click OK.



# Build.Gradle

Also be sure you specify the test library dependencies in your app module's build.gradle file:



Gradle is an open-source build automation tool

Creation of a software build, including compiling computer source code into binary code, packaging binary code, and running automated tests.

Gradle runs on the JVM and you must have a Java Development Kit (JDK) installed to use it.

Gradle Scripts

build.gradle (project)

build.gradle(app)

# BUILD.GRADLE (APP)

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])

    implementation 'androidx.appcompat:appcompat:1.2.0'
    implementation 'androidx.constraintlayout:constraintlayout:2.0.4'
    // Required for local unit tests (JUnit 4 framework)
    testImplementation 'junit:junit:4.12'
    // Required for instrumented tests
    androidTestImplementation 'androidx.test.ext:junit:1.1.2'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.3.0'
    implementation 'com.google.android.material:material:1.2.1'
    androidTestImplementation 'org.junit.jupiter:junit-jupiter'
    androidTestImplementation 'org.junit.jupiter:junit-jupiter'
    androidTestImplementation 'org.junit.jupiter:junit-jupiter'
    androidTestImplementation 'org.junit.jupiter:junit-jupiter'
}
```

# Automating your test

JUnit test framework

- Test the methods in a class in isolation without the UI

Espresso testing framework, provided by AndroidX Test, provides APIs for writing UI tests to simulate user interactions within a single target app

- One approach to UI testing is to simply have a human tester perform a set of user operations on the target app and verify that it is behaving correctly. However, this manual approach can be time-consuming, tedious, and error-prone.
- A more efficient approach is to write your UI tests such that user actions are performed in an automated way. The automated approach allows you to run your tests quickly and reliably in a repeatable manner.

# Creating an Espresso test class

## Find

- Find the UI component you want to test in an Activity (for example, a sign-in button in the app) by calling the onView() method, or the onData() method for AdapterView controls

## Simulate

- Simulate a specific user interaction to perform on that UI component, by calling the ViewInteraction.perform() or DataInteraction.perform() method and passing in the user action (for example, click on the sign-in button).

## Use

- Use the ViewAssertions methods to check that the UI reflects the expected state or behavior, after these user interactions are performed.

```
@Test
public void changeText_sameActivity() {
    // Type text and then press the button.
    onView(withId(R.id.editTextUserInput))
            .perform(typeText(stringToBetyped), closeSoftKeyboard());
    onView(withId(R.id.changeTextBt)).perform(click());

    // Check that the text was changed.
    onView(withId(R.id.textToBeChanged))
            .check(matches(withText(stringToBetyped)));
}
```

# Espresso test example

- Before building your UI test with Espresso, make sure to set a dependency reference to the Espresso library in the build.gradle (app)

# Another Expresso Test Example

```
@Test
public void greeterSaysHello() {
    onView(withId(R.id.name_field)).perform(typeText("Steve"));
    onView(withId(R.id.greet_button)).perform(click());
    onView(withText("Hello Steve!")).check(matches(isDisplayed()));
}
```

# Expresso packages

- espresso-core - Contains core and basic View matchers, actions, and assertions. See Basics and Recipes.
- espresso-web - Contains resources for WebView support.
- espresso-idling-resource - Espresso's mechanism for synchronization with background jobs.
- espresso-contrib - External contributions that contain DatePicker, RecyclerView and Drawer actions, accessibility checks, and CountingIdlingResource.
- espresso-intents - Extension to validate and stub intents for hermetic testing.
- espresso-remote - Location of Espresso's multi-process functionality.

# Espresso test recorder

- The Espresso Test Recorder tool lets you create UI tests for your app without writing any test code.
- To start recording a test,
  - Click Run > Record Espresso Test.
  - In the Select Deployment Target window, choose the device on which you want to record the test. If necessary, create a new Android Virtual Device. Click OK.