Week 4
(Module 4)

CS 5254

Android versions as a moving target

- Consider the API level history and release rate by year:

Years	API Levels	Release Rate
2008	1	1 per year
2009 – 2011	2 – 15	~4.7 per year
2012 – 2017	16 – 27	2 per year
2018 – 2021	28 – 31	1 per year
2022	32 – 33	2 per year

- In the early years, device manufacturers struggled to deliver the most current API level in products
  - Updates are also problematic, because carriers and manufacturers often customize the OS
  - This all leads to a very substantial portion of the market using older API levels
  - Supporting older systems is much easier now than it was in the past (but still not quite trivial)
- In 2018 Google Play began enforcing minimum target API levels for published apps
  - Currently (Sep. 2022) all **new apps** must target API level 31 or higher
  - Soon (Nov. 2022) all **app updates** must target API level 31 or higher
  - For the latest details:
    - `https://developer.android.com/google/play/requirements/target-sdk`
    - `https://support.google.com/googleplay/android-developer/answer/11926878`
  - This doesn't affect our assignments, and we're already targeting API level 32 anyway

Testing: JVM tests (also known as local tests)

- These are normal JUnit tests you may have encountered already in Java projects
  - These tests are very fast to run, because they don’t involve an emulator
    - However, they're also very limited, because they don't involve an emulator
- Each test function is identified by the `@Test` annotation
  - The `@Before` and `@After` annotations are functions to setup and cleanup before every `@Test`
  - Use `assertEquals(expectedValue, actualValue)` as necessary to confirm proper behavior
    - Other assert-related functions are available for convenience, but typically aren't necessary
- Within the test code we may access a view model (or any other model-related objects) only
  - We can't access any of our app's Activity or View objects

Testing: Instrumented tests

- These tests use the Espresso framework and run entirely on the device or emulator
  - Testing in this manner is much slower than local tests
    - Still it's much faster (and far more reliable) than manual testing!
- Each test function is identified by the `@Test` annotation
  - The `@Before` and `@After` annotations are functions to setup and cleanup before every @Test
  - In `@Before` we launch a starting activity scenario; in `@After` we close that activity
- **Espresso** works primarily with views via `onView()` with matchers (often just a view ID)
  - The test cases can essentially work with the app exactly as a user might
  - Once a view is matched, we generally either interact with it or assert something about it
    - Calling `perform(click())` will click a clickable view
    - Calling `check(matches(...))` will assert that the view has certain properties
  - We may also call `scenario.recreate()` to destroy and recreate the activity
    - Hopefully that sounds familiar by now, as this simulates a device rotation!
- Espresso waits until prior interactions are completed before starting a new interaction
  - However, animations such as button-press gradients, can still cause timing issues
    - Update the `build.gradle` file (module/app) to disable these animations:
      - `testOptions { animationsDisabled = true }`

Intents and results

- The easiest way to start a new activity from the current activity is to call `startActivity()`
  - This function requires only an `Intent` object to specify the activity we want to launch
  - Each intent can include **extra** data to be passed to the target activity
    - Extras are organized as mappings of unique `String` keys to simple (or `Serializable`) values
    - Conventionally, the target activity provides a means to construct an intent with proper extras
      - Still, when the target fetches an extra, a default value must be specified
  - However this mechanism doesn't provide any way to receive any results from the target
- To receive results from the target activity, an `ActivityResultLauncher` must be constructed
  - The `ActivityResultContracts` class provides convenience functions suitable for most cases
    - This can safely be done during initialization of the origin activity
    - A callback specifies what to do once a result has been returned from the target activity
  - Call the `launch()` function of the launcher to start the target activity
  - The target activity calls `setResult()` to return data to the origin activity
    - This requires a result code, plus optionally a blank intent with extras to hold the returned data
      - The code is usually `Activity.RESULT_OK` or `Activity.RESULT_CANCELED` but any `Int` is valid
- The device **Back** button/gesture dismisses the target activity and returns to the origin activity

Hints and tips for Project 1C

- New properties (and some functions) are needed in the `QuizViewModel` to support keeping counts
- When the `ResultActivity` is launched – or recreated – it still has access to its launching intent
  - This means it can read the extras during onCreate() and store them as properties, but..
    - ...this should only be done if Reset All hasn't been clicked yet, so...
      - ...a `ResultViewModel` is still required to maintain the Reset All status (clicked or not)
- Please keep in mind the MVC separation as our system grows in complexity
  - You may also need to fix some issues from P1A and P1B
- Beware that the provided tests definitely won't detect all possible issues
  - Please be sure to develop additional tests to ensure the system will pass the instructor tests