Jonathan Kleve
Student ID: 001313505
C196 Mobile Application Development

1. Developing my application for a tablet rather than a phone would necessitate a significant shift in design philosophy to leverage the larger screen real estate effectively. The user interface (UI) on a phone is typically optimized for a linear, vertical flow, with activities taking up the entire screen. However, a tablet-optimized app would embrace a more spatially open design and leverage multi-pane layouts. Fragments would become essential for creating such a UI. For instance, instead of navigating between activities, a tablet app could use fragments to display a list of items (e.g., terms, courses) and their details side-by-side. A ListFragment could occupy the left pane, while a DetailFragment would display the selected item's information in the right pane. This pattern enhances the user experience by reducing the need for constant back-and-forth navigation.
   Layouts would also differ considerably. While my current layouts are ConstraintLayouts used to properly arrange views vertically within the confined space, tablet layouts could utilize more sophisticated structures like GridLayout or FlexboxLayout to better distribute elements across the wider screen. For example, a form for editing a term or course could be presented in a two-column layout on a tablet. This format would be able to show the user a before and after preview side by side before confirmation. These layouts would be built on different resource qualifiers (e.g., layout-sw600dp for tablets) as they are tailored to larger screens.

2. My application was developed using Android Studio. The minimum SDK version is API 26, which means the application is designed to be compatible with devices running Android 8.0 and later. The target SDK version is API 34 (Android 14), which indicates that the application is optimized to take advantage of the latest features and APIs available in Android 14.

3. Developing this mobile application presented several challenges, primarily centered around managing data persistence, handling asynchronous operations, and ensuring proper user interface updates. One significant hurdle was implementing the database to store and retrieve application data. I was entirely unfamiliar with SQLite, so learning its capabilities, syntax, and how it was built into Android took some time for me to learn to implement it properly.
   Another challenge involved handling asynchronous operations as I implemented LiveData objects for the more robust ability to check for asynchronous updates. This

decision was made under the assumption that an instructor could update an assessment, etc., while a student had the details open on the app. The student would not be aware of potentially significant changes until the point at which they did something to cause the information to be reloaded. However, using LiveData caused some issues that required me to add code to manage the background database operations more cautiously and explicitly.

4. In order to address those issues, I first pieced out the aspects of the overall function into helper classes to abstract the database operations and used transactions to ensure data integrity. I also implemented custom data structures and serialization techniques to manage the relationships between different data entities. Returning to the main problem, I had to ensure the data was properly loaded before it was accessed. I had to carefully manage LiveData observers and use Executors to handle thread management, ensuring UI updates occurred on the main thread while database operations were performed on a background thread. This implementation was utilized to prevent the UI from being blocked; however, it introduced potential issues with data consistency and race conditions.
   Addressing the asynchronous operation challenges required a more iterative approach. I spent considerable time understanding Android's threading model and how to use background tasks effectively. I used Log statements extensively to trace the execution flow of my code and verify that long-running operations were indeed occurring on background threads. To manage thread execution, I experimented with AsyncTask and Handler, carefully considering their limitations and potential issues. I also implemented callback mechanisms to update the UI after the background tasks were completed. I used Android's runtime permission request system to handle permissions, including checking for existing permissions, requesting them if necessary, and providing user feedback if permissions were denied.

5. If I were to undertake this project again, I would prioritize a more modular architecture from the outset. Initially, I laid out the program with a more monolithic approach due to lack of experience, which led to tightly coupled components and made it slightly more challenging to test and maintain the code. In a future iteration, I would implement a more precise separation of concerns using the Model-View-ViewModel (MVVM) pattern. I have attempted to implement it; however, I am certain it is far from completely tidy and clean-cut. I would also investigate implementing fragments, as I did not incorporate any, and their modularity makes them a great asset to Android UI development.

In addition, I would put much more consideration into writing unit and integration tests. For the current scope, my testing was primarily focused on manual testing on my personal device. However, a comprehensive suite of automated tests would have significantly improved the robustness and reliability of the application. I would also consider leveraging a testing framework to test the individual components of the application. Such frameworks would catch potential bugs early in the development process and make future refactoring less risky.

6. Emulators are software programs that simulate Android devices on a computer. They provide a virtual environment that mimics the hardware and software configurations of various Android devices, allowing developers to test their applications without needing physical devices. Being integrated into Android Studio and at no additional cost makes emulators very convenient and accessible. They also provide detailed logging and debugging tools to make diagnosing issues significantly easier. In addition, emulators can be configured to imitate a wide range of devices and Android versions. This feature allows developers to ensure their applications work on a diverse selection of devices. In short, emulators are readily available, versatile, and easy to use. However, emulators can also be very taxing on computer performance, especially if attempting to run tests on multiple emulators on the same computer. Emulators are not physical, so testing involving the use of certain components like a gyroscope, camera, or GPS might be limited, if not impossible. In addition, physical hardware devices often have specific quirks in their behavior, which emulation cannot replicate.
   In contrast, a development device is a physical Android device used for testing. These tools allow for the most accurate representation of how the application will perform for end users. The developer can properly utilize all of the physical hardware components of the device for testing. In addition, apps generally run better on physical devices as the software was made to work on them and the devices have their own dedicated, physical hardware powering them. However, development devices are often expensive, especially when considering multiple devices of various types. Managing the physical devices and setting up debugging tools on them can also be a hassle.