Ian Valiante

Part 3

**Android App Implementation**

Converting my Java Pokémon console game to an Android application using Android Studio presents some significant challenges, primarily due to the fundamental differences between console-based and event-driven architectures. In a console application, the flow of execution is linear and synchronous. The program asks the user for input, waits for the response, processes it, and then moves on to the next step. With this model, you can capture user input directly using blocking calls such as Scanner.nextLine() and display output using System.out.println(). However, Android apps operate in a completely different paradigm. They are inherently event-driven, responding asynchronously to user interactions such as button clicks, text input, and other UI events. That is, the application does not execute code in a strict order but instead reacts to events as they occur according to the Android system lifecycle.

One of the main challenges when porting my game is handling user input and output. In a console application, the program stops executing until input is received, so reading input immediately after displaying a message to the user works without a problem. In contrast, in an Android app, if you set a query in a TextView and then try to read from an EditText immediately after, it doesn’t work as expected. The application doesn’t wait for the user to provide input. Instead, it often continues executing subsequent lines of code, trying to handle empty or stale input. This mismatch occurs because the main thread in Android is responsible for rendering the UI and handling user interactions without blocking, keeping the application responsive. Therefore, rather than relying on sequential input prompts to enable input and output, an event-based approach is required where actions are triggered by user events, such as clicking a submit button.

Another important issue is managing the game thread without blocking the main UI thread. In console applications, you can use loops or sequential prompts because the program runs on a single thread, and blocking calls does not affect responsiveness. However, in Android apps, performing long-running tasks or blocking the main thread can cause the application to become unresponsive, which can result in a crash and an ANR (Application Not Responding) error. To solve this problem, you need to restructure the game logic into discrete steps triggered by user interaction. Instead of a continuous loop, the game progresses through states driven by event listeners and callbacks. This ensures that each part of the game is executed only in response to specific user actions, keeping the application responsive and stable.

It is also essential to separate the game logic from the user interface to create maintainable and testable Android applications. In the current setup, the CardGame class couples the game logic with Android-specific UI components such as TextView and EditText. This tight coupling makes it difficult to maintain and test my code independently of the Android framework. To improve this, I would need to encapsulate my game logic in a separate class that is independent of the Android UI elements. Activities or fragments should primarily act as controllers, handling user interactions, updating the UI, and interacting with the game logic. Adopting architectural patterns such as Model-View-ViewModel (MVVM) can further strengthen this separation, resulting in a more organized and scalable codebase.

State management is another area that requires careful consideration when porting to Android. While console applications typically support simple states through variables in a linear flow, Android applications must handle more complex states due to their lifecycles, which include states such as pause, stop, or terminate based on user interaction and system conditions.

Android Studio has features such as the ability to convert Java code to Kotlin also make the transition easier. Kotlin offers modern programming features, including conciseness and compatibility with Java, making it easier to implement event-driven logic and asynchronous operations. Integrating Kotlin’s higher-order functions and coroutines allows you to handle asynchronous operations more easily and efficiently, better fitting the Android architecture and improving the overall responsiveness of the application.

In short, successfully porting my Pokémon game to an Android app requires more than just replacing console input and output with Android UI components. It requires a fundamental redesign of my game architecture to accommodate Android’s event-driven, multithreaded environment. By adopting asynchronous programming, separating game logic from UI components, efficiently managing application state, and using modern Android development techniques and tools, i could transform my console Pokémon game into a responsive, user-friendly Android application. This process involves significant changes and a steep learning curve, but adopting these strategies will ultimately result in a more stable and supported application that delivers engaging gaming experiences on mobile devices.`