ĐẠI HỌC QUỐC GIA HÀ NỘI

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

Nguyễn Đức Quốc Đại

18020265

BÁO CÁO BÀI TẬP GIỮA KỲ

PHÁT TRIỂN ỨNG DỤNG DI ĐỘNG VỚI KOTLIN

Ngành: Công nghệ thông tin

HÀ NỘI - 2021

MỤC LỤC

TÓM TẮT

Hiện nay, với kỷ nguyên 4.0 đang phát triển mạnh mẽ, mọi hoạt động của con người đều gắn với ứng dụng trên Smartphone: 5G, AI, Mobile Wallet, E-learning, News… Hiểu về lập trình ứng dụng di động ngày càng cần thiết hơn trong cuộc sống. Nó giúp mở ra cơ hội nghề nghiệp đáng mơ ước cho nhiều bạn trẻ. Vì vậy bài báo cáo này sẽ trình bày những kiến thức cơ bản nhất về phát triển ứng dụng di động với ngôn ngữ Kotlin cho các bạn đọc xây dựng được nền tảng và định hướng được tương lai sau này.

Từ khóa: Android, Kotlin.

CHƯƠNG 1: Một số cú pháp cần chú ý

1.1. Null-Safety

Dấu "?": Cho phép chúng ta gán giá trị null cho 1 biến đã khai báo kiểu giá trị. Ví dụ: *ten: String = "DAI"* rồi thì không được gán *ten = null* mà muốn gán *ten = null* thì phải thêm *String?*

Dấu "!!": Không cho phép gán giá trị null. Ví dụ: Ten: String = "DAI"!! có nghĩa là *Ten* không được phép gán bằng *null*.

1.2. Lateinit

Thông thường, các thuộc tính được khai báo là có kiểu không phải *null* phải được khởi tạo trong *constructor*. Tuy nhiên, thường thì điều này không thuận tiện. Trong trường hợp này, bạn không thể cung cấp bộ khởi tạo không *null* trong phương thức khởi tạo, nhưng bạn vẫn muốn tránh kiểm tra *null* khi tham chiếu đến thuộc tính bên trong phần thân của một lớp, do vậy cần thêm *lateinit*.

1.3. Companion

Làm cho các biến hoặc phương thức trong lớp là của lớp chứ không phải của đối tượng.

CHƯƠNG 2: Xây dựng ứng dụng đầu tiên

2.1. Những phần chính của ứng dụng

- **Res** – *Resources* chứa những nội dung tĩnh được sử dụng trong ứng dụng.
- **GeneratedJava** – Khi *app* được *built* sẽ sinh ra những *file java* ở đây.
- **Manifests** – Chứa *file AndroidManifest.xml* mô tả chi tiết về *app* để hệ điều hành biết để chạy app.

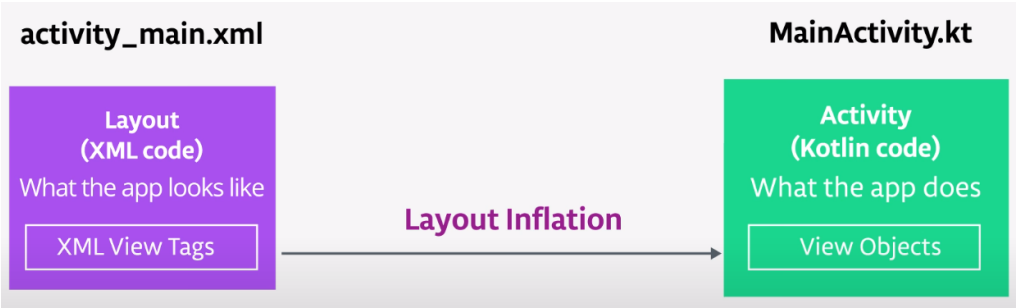
Như vậy, layouts màn hình ở trong *Res folder*, kotlin code ở trong *Java folder*, ảnh ở trong *Res folder*, các quyền của app ở trong *AndroidManifest*, code để *build* và *run* ở trong *Gradle scripts*.

VD: *MainActivity* được tham chiếu ở đây.

Hình 2.1: AndroidManifest

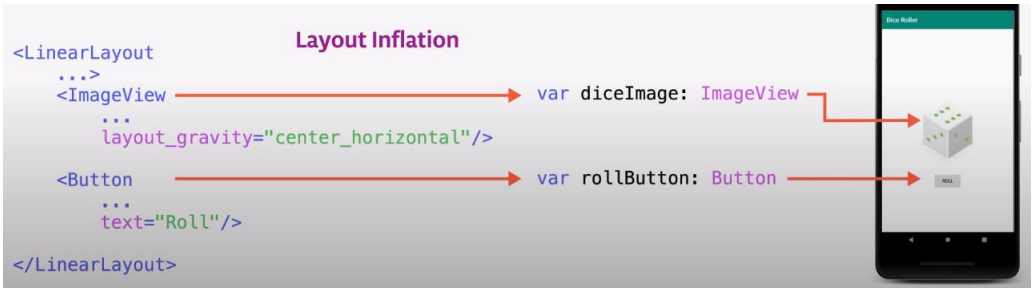
2.2. Activity và Layout

Activity_main.xml (XML code-Activity) sẽ mô tả *app* sẽ trông như thế nào ? Hầu hết tất cả các *activity* đều tương tác với người dùng, vì vậy lớp *Activity* sẽ đảm nhận việc tạo một cửa sổ cho bạn trong đó bạn có thể đặt giao diện người dùng của mình vào bằng *setContentView* (*View*). Còn *MainActivity.kt*: (Kotlin code - Layout) sẽ mô tả *app* sẽ làm những gì ?



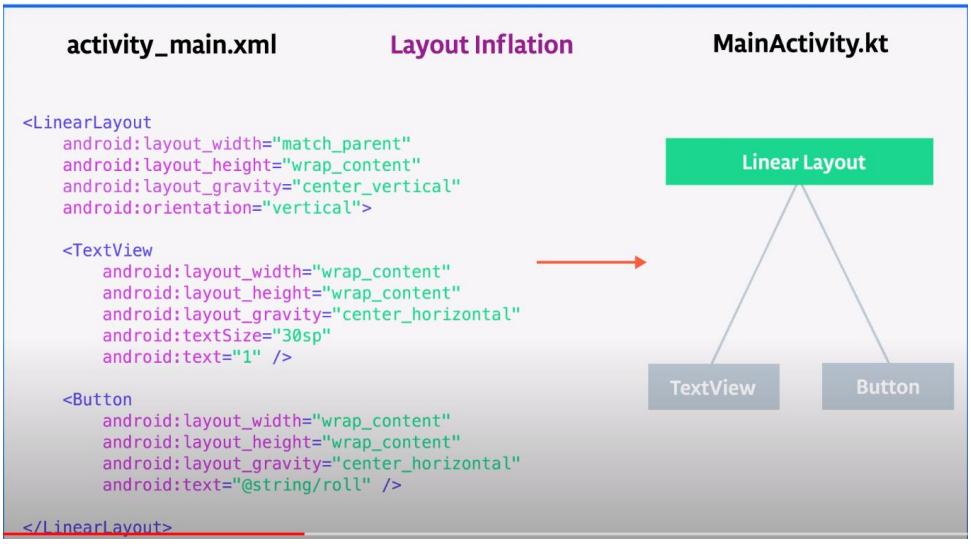
Hình 2.2: Quá trình inflate

Influate sẽ tham chiếu *layout* thành view *model* để hiển thị, cập nhật, … Tức là biến 1 thẻ *xml* thành 1 *instance* trong *activity*.



Hình 2.3: Ví dụ quá trình inflate

2.3. Kết nối với Button



Hình 2.4: Phân cấp layout

Inflate nó sẽ tham chiếu *xml* thành cây phân cấp như bên phải với các thẻ. *Activity* sẽ sử dụng *view* cây phân cấp đó. Khi định nghĩa ID cho mỗi thẻ xml, OS sẽ tự động tạo một số *integer* dạng *constant* với tên ID được đặt trong 1 *class* được tạo ra gọi là R (Resource). *findViewById* sẽ duyệt lần lượt từ ngọn đến lá trên cái cây phân cấp đó xem *node* nào trùng với cái ID nào. *Context object* cho phép giao tiếp và lấy thông tin về trạng thái hiện thời của *Android Operating System*.

Toast cần truy cập tới một *context* để có thể yêu cầu hệ điều hành hiển thị *toast*.

2.4. Namespaces

XML namespaces được sử dụng để cung cấp các phần tử và thuộc tính được đặt tên duy nhất trong một tài liệu XML. Một *instance* XML có thể chứa các tên phần tử hoặc thuộc tính từ nhiều hơn một từ vựng XML. Nếu mỗi từ vựng được cung cấp một *namespace*, sự không rõ ràng giữa các phần tử hoặc thuộc tính được đặt tên giống nhau có thể được giải quyết. Một ví dụ đơn giản là xem xét một *instance* XML có chứa các tham chiếu đến một khách hàng và một sản phẩm đã đặt hàng. Cả phần tử khách hàng và phần tử sản phẩm đều có thể có phần tử con có tên là id. Do đó, các

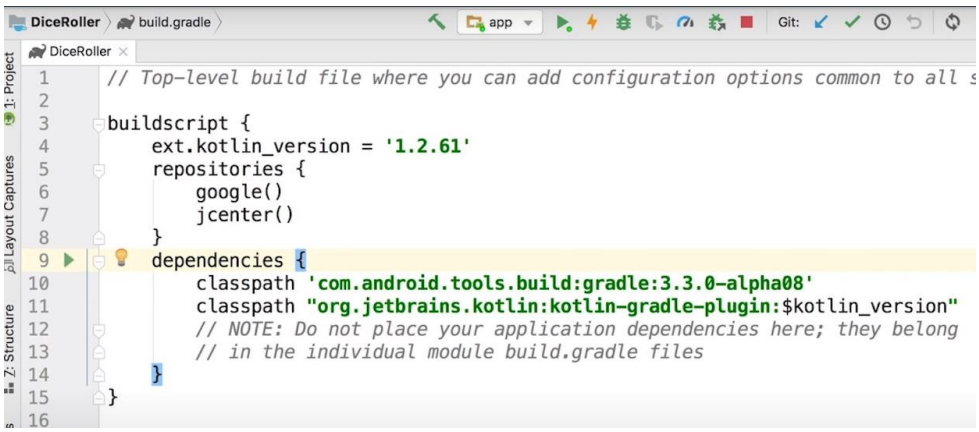
tham chiếu đến phần tử id sẽ không rõ ràng; đặt chúng trong các không gian tên khác nhau sẽ loại bỏ sự mơ hồ.

2.5. Giới thiệu về Gradle

Gradle mô tả những thiết bị nào mà *app* được *built* và *run*. Biên dịch *code* và *resource* thành mã thực thi. Khai báo và quản lý *code*, thư viện. Đăng ký *app* cho *Google Play* để người dùng có thể tải từ *Google Play* và chạy *test* tự động. Khi ấn *run* thì dãy các lệnh compile sẽ chạy trong *APK*.

Module:App : Đối với mỗi module của dự án cần một thư mục chứa các tệp nguồn và tài nguyên cho một phần chức năng rời rạc trong ứng dụng của bạn

2.6. Build.gradle



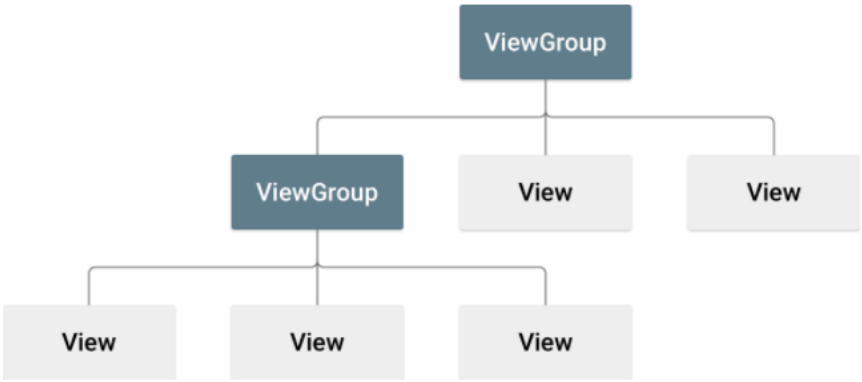
Hình 2.5: Ví dụ về gradle

- Repository: là các máy chủ từ xa, nơi mà chúng ta cần những code ngoài sẽ được tải từ đó. VD: Thư viện Android:
    - *Google repository* chứa những thư viện lõi của Androi được duy trì bởi Google.
    - *Jcenter* một kho lưu trữ lớn phổ biến trong số các nhà xuất bản nguồn mở.
- Dependency: Code ngoài, chẳng hạn như thư viện mà một dự án phụ thuộc.

CHƯƠNG 3: Layouts

3.1. Layouts

Một *layout* xác định cấu trúc cho giao diện người dùng trong ứng dụng của bạn, chẳng hạn như trong một *activity*. Tất cả các phần tử trong *layout* được xây dựng bằng cách sử dụng hệ thống phân cấp của các đối tượng *View* và *ViewGroup*. Một *view* thường vẽ thứ gì đó mà người dùng có thể nhìn thấy và tương tác. Trong khi *ViewGroup* là một vùng chứa vô hình xác định cấu trúc bố cục cho *View* và các đối tượng *ViewGroup* khác, như thể hiện trong hình 3.1:



Hình 3.1: Cây phân cấp view

Các đối tượng *View* thường được gọi là "widget" và có thể là một trong nhiều lớp con, chẳng hạn như *Button* hoặc *TextView*. Các đối tượng *ViewGroup* thường được gọi là “Layouts” có thể là một trong nhiều loại cung cấp cấu trúc *layout* khác nhau, chẳng hạn như *LinearLayout* hoặc *ConstraintLayout*

Có thể khai báo một *layout* theo hai cách:

- Khai báo các phần tử giao diện người dùng trong XML. *Android* cung cấp một ngôn ngữ XML đơn giản tương ứng với các lớp và lớp con *View*, chẳng hạn như các lớp dành cho *widget* và *layout*. Bạn cũng có thể sử dụng trình chỉnh sửa bố cục của *Android Studio* để xây dựng bố cục XML của mình bằng giao diện kéo và thả.
- Khởi tạo các phần tử bố cục trong thời gian chạy. Ứng dụng của bạn có thể tạo các đối tượng Vieiw và *ViewGroup* (và thao tác các thuộc tính của chúng) theo chương trình.

Khai báo giao diện người dùng của bạn trong XML cho phép bạn tách ứng dụng khỏi *code* kiểm soát hành vi của nó. Việc sử dụng tệp XML cũng giúp dễ dàng cung cấp các bố cục khác nhau cho các kích thước và hướng màn hình khác nhau. *Framework Android* cung cấp sự linh hoạt khi sử dụng một trong hai hoặc cả hai phương pháp này để xây dựng giao diện người dùng cho ứng dụng. Ví dụ: ta có thể khai báo bố cục mặc định của ứng dụng bằng XML, sau đó sửa đổi bố cục trong thời gian chạy.

3.2. ViewGroup

ViewGroup là một dạng *view* đặc biệt có thể chứa các *view* khác (được gọi là *con*.) Nhóm các *view* là lớp cơ sở cho *layouts* và các *views containers*. Lớp này cũng định nghĩa lớp *ViewGroup.LayoutParams* đóng vai trò là lớp cơ sở cho các tham số bố cục.

3.3. View

Lớp này đại diện cho khối xây dựng cơ bản cho các thành phần giao diện người dùng. Một *view* chiếm một vùng hình chữ nhật trên màn hình và chịu trách nhiệm vẽ và xử lý sự kiện. *View* là lớp cơ sở cho các *widget*, được sử dụng để tạo các thành phần giao diện người dùng tương tác (*buttons, text fields, …*). Lớp con *android.view.ViewGroup* là lớp cơ sở cho các *layouts*, là các vùng chứa vô hình chứa các *Views* khác (hoặc *ViewGroup*) và xác định các thuộc tính *layout* của chúng.

Sử dụng *View*: Tất cả các *view* trong 1 cửa sổ đều được sắp xếp giống một cái cây. Có nhiều lớp con chuyên biệt của view hoạt động như điều khiển hoặc có khả năng hiển thị văn bản, hình ảnh hoặc nội dung khác.

Khi đã tạo một cây view, thường có một số loại *operation* phổ biến:

- Thiết lập thuộc tính: ví dụ: thiết lập văn bản của *android.widget.TextView*, các thuộc tính có sẵn và các phương thức thiết lập chúng sẽ khác nhau giữa các lớp con khác nhau của *views*. Lưu ý rằng các thuộc tính đã biết tại thời điểm xây dựng có thể được đặt trong tệp bố cục XML.
- Thiết lập focus: *Framework* sẽ xử lý *focus* di chuyển theo phản hồi khi người dùng nhập vào. Để buộc *focus* vào một *view*, gọi *#requestFocus*.

- Thiết lập listeners: *View* cho phép người dùng thiết lập *listener* sẽ được thông báo khi có điều gì đó thú vị xảy ra với *view*. Ví dụ: tất cả các *view* sẽ cho phép bạn đặt *listener* được thông báo khi *view* tăng hoặc mất *focus*. Để có thể đăng ký một *listener* có thể dùng *setOnFocusChangeListener(android.view.View.OnFocusChangeListener)*. Các lớp con *view* khác cung cấp nhiều *listeners* chuyên biệt hơn. Ví dụ, một Button sẽ bắt một *listener* để thông báo cho khách hàng khi nút được nhấp.
- Thiết lập hiển thị: Ta có thể ẩn hoặc hiển thị *view* bằng cách sử dụng *setVisibility (int)*.

Frameworkd Android chịu trách nhiệm đo lường, bố trí và vẽ các *layout*. Không nên gọi các phương thức tự thực hiện các hành động này trên các view trừ khi đang triển khai một *android.view.ViewGroup*

3.4. LinearLayout

LinearLayout là một nhóm *view* sắp xếp tất cả các phần tử con theo một hướng duy nhất, theo chiều dọc hoặc chiều ngang. Bạn có thể chỉ định hướng bố cục bằng thuộc tính *android: orientation*.

Tất cả các phần tử con của *LinearLayout* lần lượt được xếp chồng lên nhau, do đó, danh sách dọc sẽ chỉ có một phần tử con trên mỗi hàng, bất kể chúng rộng bao nhiêu và danh sách ngang sẽ chỉ cao một hàng (chiều cao của con cao nhất + *padding*).

3.5. ScrollView

Một nhóm *view* cho phép cuộn *view* phân cấp được đặt trong nó. *ScrollView* có thể chỉ có một con trực tiếp được đặt trong đó. Để thêm nhiều view trong *ScrollView*, hãy đặt con trực tiếp mà bạn thêm một nhóm *view*, ví dụ *LinearLayout*, và đặt các view bổ sung trong *LinearLayout* đó. *Scroll view* chỉ hỗ trợ cuộn dọc. Để cuộn ngang, hãy sử dụng *HorizontalScrollView* để thay thế. Không bao giờ thêm *RecyclerView* hoặc *ListView* vào ScrollView. Làm như vậy dẫn đến hiệu suất giao diện người dùng kém và trải nghiệm người dùng kém.

3.6. Data Binding

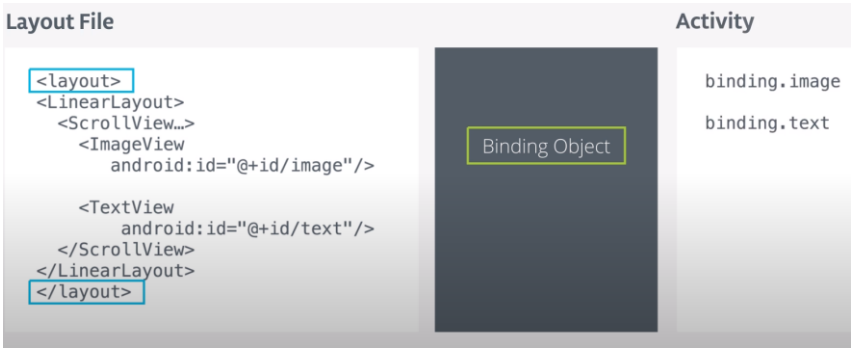
Kết nối *Layout* với *Activity* (*Fragment*) ở thời điểm biên dịch. Cho phép kết nối *layout* với một *activity* hoặc *fragment* tại thời điểm biên dịch. Trình biên dịch tạo ra một lớp trợ giúp được gọi là *binding class* khi *activity* được tạo ra, một *instance* của *binding class* cũng vậy. Sau đó, ta có thể truy cập view thông qua binding class được tạo này mà không cần thêm bất kỳ chi phí nào. Lớp *ActivityMainBindning* được tạo bởi trình biên dịch đặc biệt cho *activity* chính này và tên có tổ hợp từ *"Tên layout + Binding = ActivityMainBinding"*.

3.6.1. Data Binding - The Idea

Ý tưởng lớn về ràng buộc dữ liệu là tạo một đối tượng kết nối / ánh xạ / liên kết hai phần thông tin ở xa với nhau tại thời điểm biên dịch, để bạn không phải tìm kiếm nó trong thời gian chạy. Đối tượng hiển thị các ràng buộc này với bạn được gọi là đối tượng *Binding*. Nó được tạo ra bởi trình biên dịch và trong khi hiểu được cách thức hoạt động của nó là rất thú vị, nhưng không cần thiết phải biết các cách sử dụng cơ bản của ràng buộc dữ liệu.

3.6.2. Data Binding and findViewById

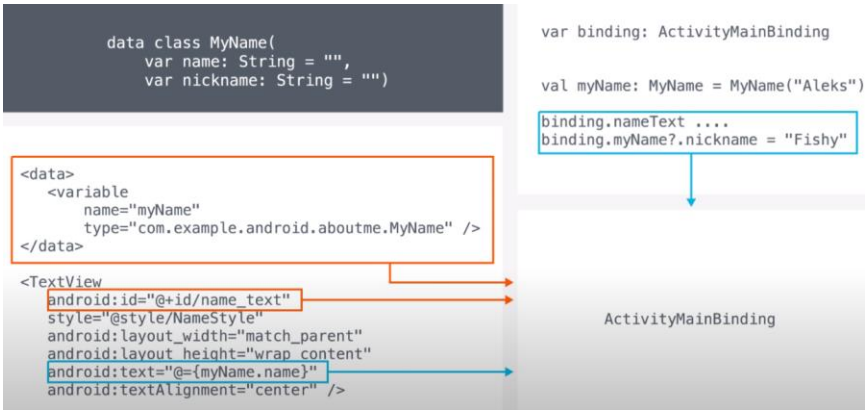
findViewById là một phương thức tốn kém tài nguyên vì nó sẽ phải duyệt cả cây view phân cấp mỗi khi nó được gọi. Khi bật liên kết dữ liệu, trình biên dịch tạo tham chiếu đến tất cả các view trong <layout> có *id* và gộp chúng trong một đối tượng *Binding*. Khi code, bạn tạo một instance của đối tượng *binding* sau đó tham chiếu các view thông qua *đối tượng binding* mà không cần thêm chi phí.



Hình 3.2: Tổng quan Binding

3.6.3. Data Binding Views and Data

Cập nhật dữ liệu và sau đó cập nhật dữ liệu được hiển thị trong view nhìn chung khá cồng kềnh và là nguồn gây ra lỗi. Giữ dữ liệu trong *view* cũng vi phạm việc phân tách dữ liệu với hiện thị. *Data binding* có thể giải quyết các vấn đề này. Đặt dữ liệu trong 1 “data class”. Thêm khối <data> trong <layout> để định danh dữ liệu như một biến sử dụng trong *views*. Các views sẽ tham chiếu đến các biến đó. Bộ biên dịch sẽ sinh ra các *binding object* để gắn *views* với dữ liệu. Trong *code* chính, ta có thể tham chiếu và cập nhật dữ liệu thông qua *binding object*, đối tượng này được cập nhật dữ liệu và do đó được hiển thị trong các view. Gắn *views* với dữ liệu sẽ thiết lập 1 nền tảng cho nhiều kỹ thuật nâng cao hơn sử dụng *data binding*.



Hình 3.3: Ví dụ binding

CHƯƠNG 4: App Navigation

4.1. Fragment

Fragment là một phần của giao diện người dùng của *activity* đó. Các *activity* có thể chứa 1 hoặc nhiều *fragments*. Bạn có thể hoán đổi *fragment* này sang *fragment* khác, đó là cách để tạo nhiều màn hình trong một *activity*. *Activity* kế thừa từ *context* nhưng các *fragment* thì không, bạn sẽ cần sử dụng thuộc tính *context* trong một *fragment* để có quyền truy cập vào *AppData* thường được liên kết với *context*. Trong một *Fragment*, không phải *inflate* view trong *onCreate* giống trong *activity* mà phải inflate trong *onCreateView*(). Sử dụng thuộc tính *context* trong Fragment để truy cập String và Ảnh. *UI Fragment* chứa 1 *layout* và chiếm 1 vùng trong "*Activity layout*".

4.2. Navigation Component

Các nguyên tắc trong navigation:

- Luôn có 1 nơi bắt đầu để người dùng truy cập vào
- Có thể quay lại màn hình trước

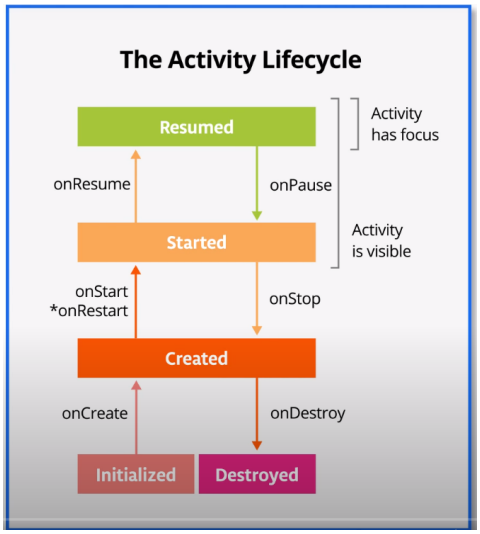
4.3. Explicit vs Implicit Intents

Explicit Intents: Khởi chạy một activity dựa trên tên lớp của nó.

Implicit Intents: Khởi chạy một activity dựa trên các tham số như action, data, data type.

CHƯƠNG 5: Activity & Fragment Lifecycle

5.1. Activity Lifecycle



Hình 5.1: Vòng đời activity

Vòng đời của 1 *activity* được thể hiện như hình vẽ, để có thể cụ thể hơn phần 5.2 sẽ trình bày cụ thể rõ từng hoạt động, do vòng đời của *activity* khá giống với vòng đời của *fragment*.

5.2. Fractment Lifecycle

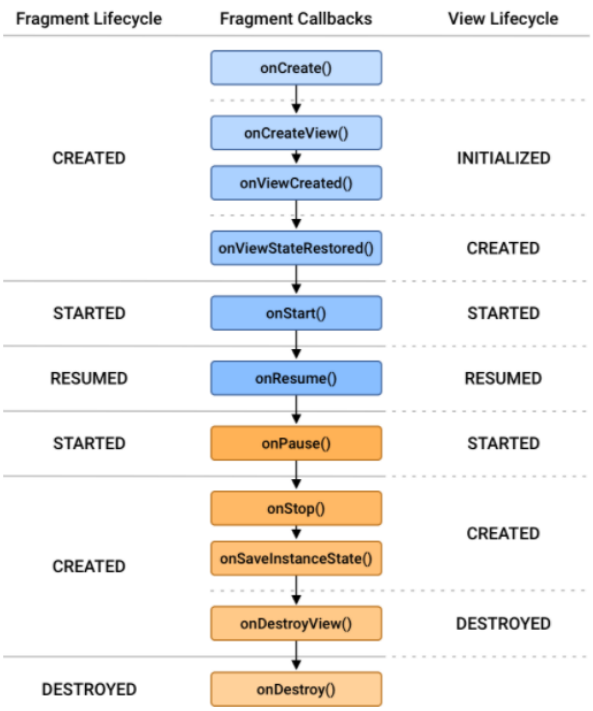


Figure 1. Fragment `Lifecycle` states and their relation both the fragment's lifecycle callbacks and the fragment's view `Lifecycle`.

Hình 5.2: Vòng đời fragment

Khi một fragment được khởi tạo nó bắt đầu ở trạng thái *INITIALIZED*. Để một *fragment* chuyển tiếp qua phần còn lại của vòng đời của nó, nó phải được thêm vào *FragmentManager*. *FragmentManager* chịu trách nhiệm xác định *fragment* nào nên ở trạng thái nào và sau đó chuyển chúng sang trạng thái đó. Ngoài vòng đời của *fragment*, *FragmentManager* cũng chịu trách nhiệm gắn các *fragment* vào *activity* và tách chúng ra khi *fragment* không còn được sử dụng. Lớp *Fragment* có hai phương thức *callback*: *onAttach*() và *onDetach*(), có thể ghi đè để thực hiện công việc khi một trong hai sự kiện này xảy ra. *Callback onAttach*() được gọi khi fragment đã được thêm vào *FragmentManager* và được gắn với activity chủ. Tại thời điểm này, *fragment* được kích hoạt và *FragmentManager* quản lý trạng thái vòng đời của nó. Các phương thức *FragmentManager* như *findFragmentById* () trả về *fragment* này. *onAttach*() luôn được gọi trước bất kì trạng thái Lifecycle nào. *Callback onDetach*() được gọi khi *fragment* bị dời khỏi *FragmentManager* và bị tách khỏi *activity* chủ. *Fragment* không

còn được kích hoạt và không còn có thể được truy xuất bằng cách sử dụng *findFragmentById(). onDetach*() luôn được gọi sau bất kì trạng thái *Lifecycle* nào.

5.2.1. Up state transitions

5.2.1.1. Fragment CREATED

Khi *fragment* của bạn đạt đến trạng thái CREATED, nó đã được thêm vào *FragmentManager* và phương thức *onAttach* () đã được gọi.

5.2.1.2. Fragment CREATED and View INITIALIZED

Vòng đời dạng xem của *fragment* chỉ được tạo khi *fragment* cung cấp một *valid View instance*. Trong hầu hết các trường hợp, bạn có thể sử dụng các hàm tạo phân đoạn lấy *@LayoutId*, tự động *inflate view* vào thời điểm thích hợp. Bạn cũng có thể ghi đè *onCreateView* () để tăng cường *inflate* hoặc tạo *view* của *fragment.*

5.2.1.3. Fragment and View CREATED

Sau khi *view* của *fragment* đã được tạo, trạng thái *view* trước đó, nếu có, sẽ được khôi phục và vòng đời của *view* sau đó được chuyển sang trạng thái CREATED. Chủ sở hữu vòng đời của view cũng phát ra sự kiện ON_CREATE cho những người quan sát của nó. Tại đây, bạn nên khôi phục bất kỳ trạng thái bổ sung nào được liên kết với view của *fragment*. Quá trình chuyển đổi này cũng gọi lại *callback onViewStateRestored*().

5.2.2. Downward state transitions

5.2.2.1. Fragment and View STARTED

Khi người dùng bắt đầu thoát khỏi *fragment* và trong khi *fragment* vẫn còn hiển thị, vòng đời cho *fragment* và *view* của nó được chuyển trở lại trạng thái STARTED và phát ra sự kiện ON_PAUSE cho *observers* của nó. Sau đó, *fragment* này sẽ gọi lại *callback onPause*().

5.2.2.2. Fragment and View CREATED

Một khi các *fragment* bị ẩn thì các *Lifecycle* của *fragment* và các *view* của nó được chuyển sang trạng thái *CREATED* và phát ra sự kiện ON_STOP cho những *observers* của chúng. Sự chuyển đổi trạng thái này không chỉ được kích hoạt bởi activity cha hoặc *fragment* cha bị dừng lại, mà còn bởi việc lưu trạng thái bởi *activity*

cha hoặc *fragment* cha. Hành vi này đảm bảo rằng sự kiện ON_STOP được gọi trước khi trạng thái của *fragment* được lưu. Điều này làm cho sự kiện ON_STOP trở thành điểm cuối cùng có thể an toàn để thực hiện một *FragmentTransaction* trên *FragmentManager* con.

5.2.2.3. Fragment CREATED and View DESTROYED

Sau khi thoát khỏi *animation transitions* và *view* của các *fragment* đã được tách ra window, thì *Lifecycle* của các *fragment* được chuyển sang trạng thái DESTROYED và phát ra sự kiện ON_DESTROY cho những *observers* của nó. Sau đó, *fragment* sẽ gọi lại onDestroyView (). Tại thời điểm này, *fragment's view* đã đi đến cuối vòng đời của nó và *getViewLifecycleOwnerLiveData* () trả về giá trị *null*. Tại thời điểm này, tất cả các tham chiếu đến *fragment's view* sẽ bị loại bỏ, cho phép *fragment's view* được thu thập.

5.2.2.4. Fragment DESTROYED

Nếu *fragment* bị loại bỏ hoặc nếu *FragmentManager* bị phá hủy, vòng đời của fragment sẽ được chuyển sang trạng thái DESTROYED và gửi sự kiện ON_DESTROY cho những *observers* của nó. Sau đó, *fragment* này sẽ gọi lại *onDestroy* (). Tại thời điểm này, *fragment* đã đi đến cuối vòng đời của nó.

CHƯƠNG 6: Kiến trúc ứng dụng (UI Layer)

6.1. Application Architecture

Kiến trúc cách thiết kế các lớp của ứng dụng và mối quan hệ giữa chúng. Chia kiến trúc thành các lớp, mỗi lớp có các trách nhiệm riêng biệt, được xác định rõ ràng:

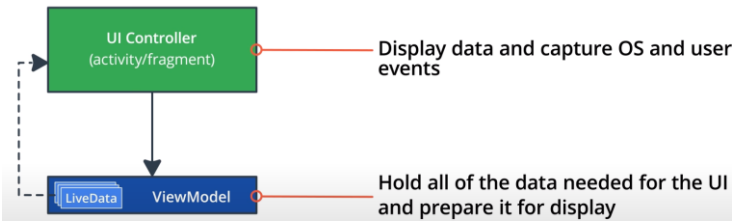
- UI controller: mô tả những activity và fragment nào. Ví dụ: Hiển thị dữ liệu và bắt sự kiện người dùng và hệ điều hành.
- ViewModel: một *helper class* giúp chứa dữ liệu cho một *Activity* hoặc *Fragment* để phục vụ tách dữ liệu khỏi *logic* bộ điều khiển giao diện người dùng. Một *ViewModel* được giữ lại miễn là phạm vi *Activity*/*Fragment* của nó còn tồn tại, kể cả khi *Activity*/Fragment bị phá hủy và được tạo lại do thay đổi cấu hình; Điều này cho phép *ViewModel* cung cấp dữ liệu cho các *instance* của *activity* hoặc *fragments* được tạo lại. Gói dữ liệu giao diện người dùng trong

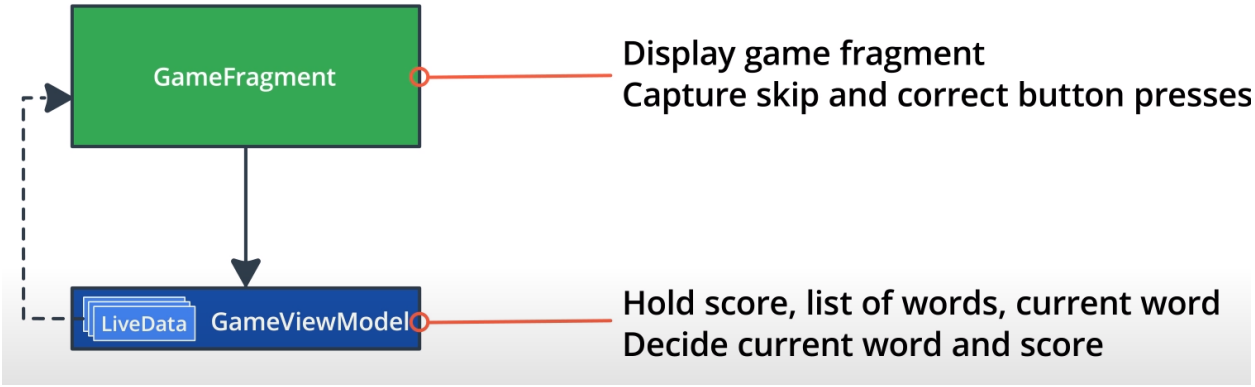
ViewModel với *LiveData* cung cấp dữ liệu một ngôi nhà nhận biết vòng đời có thể quan sát được. *LiveData* xử lý mặt thông báo của mọi thứ trong khi *ViewModel* đảm bảo rằng dữ liệu được giữ lại một cách thích hợp.

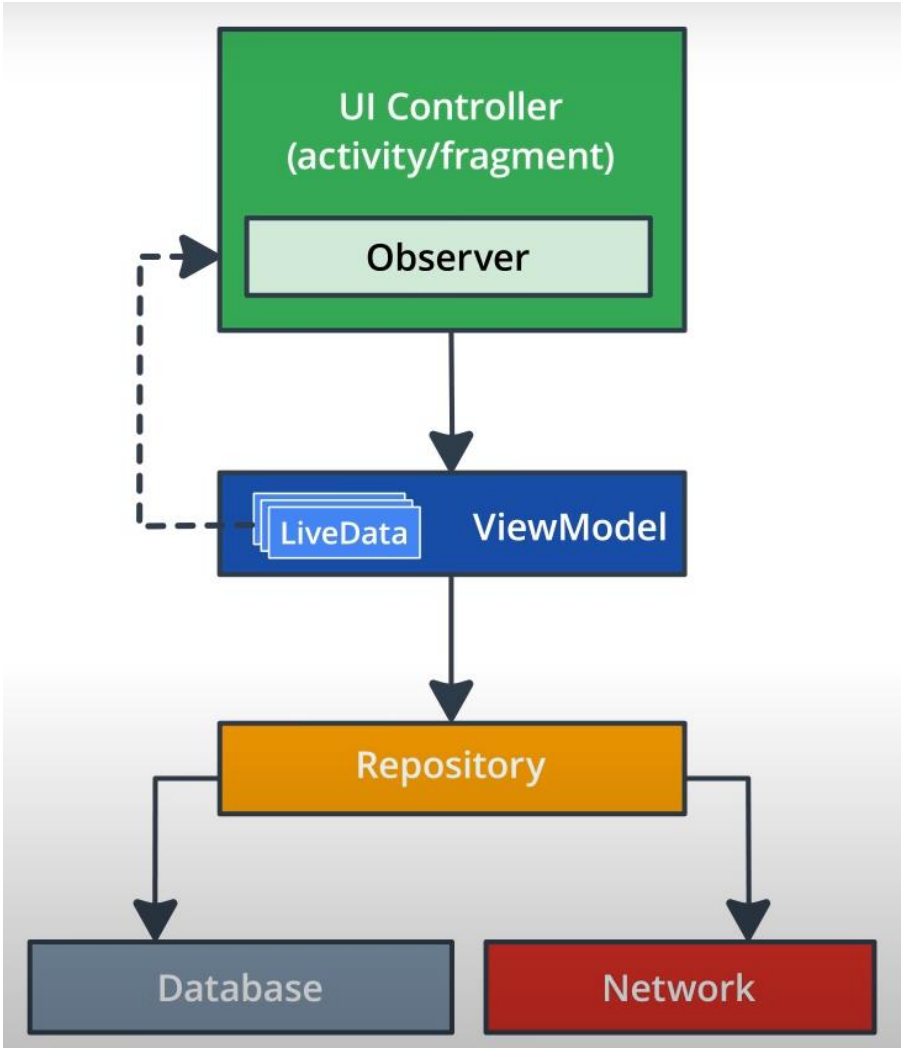
- LiveData: là một lớp lưu trữ dữ liệu nhận biết vòng đời có thể quan sát được.

UI code sẽ đăng ký những thay đổi trong dữ liệu cơ bản, được gắn với *LifecycleOwner* và *LiveData* đảm bảo *observer*: Nhận cập nhật dữ liệu trong khi Vòng đời ở trạng thái hoạt động (STARTED or RESUMED), bị xóa khi *LifecycleOwner* bị phá hủy, nhận dữ liệu cập nhật khi *LifecycleOwner* khởi động lại do thay đổi cấu hình hoặc được khởi động lại từ ngăn xếp phía sau. Điều này giúp loại bỏ nhiều con đường dẫn đến rò rỉ bộ nhớ và giảm sự cố bằng cách tránh cập nhật các hoạt động đã dừng. *LiveData* có thể được quan sát bởi nhiều listener, mỗi *listener* được gắn với một chủ sở hữu vòng đời như *Fragment* hoặc *Activity*. Truyền thông tin từ *ViewModel* tới bộ điều khiển UI thì nó sẽ cập nhật và vẽ lại màn hình.

VD:

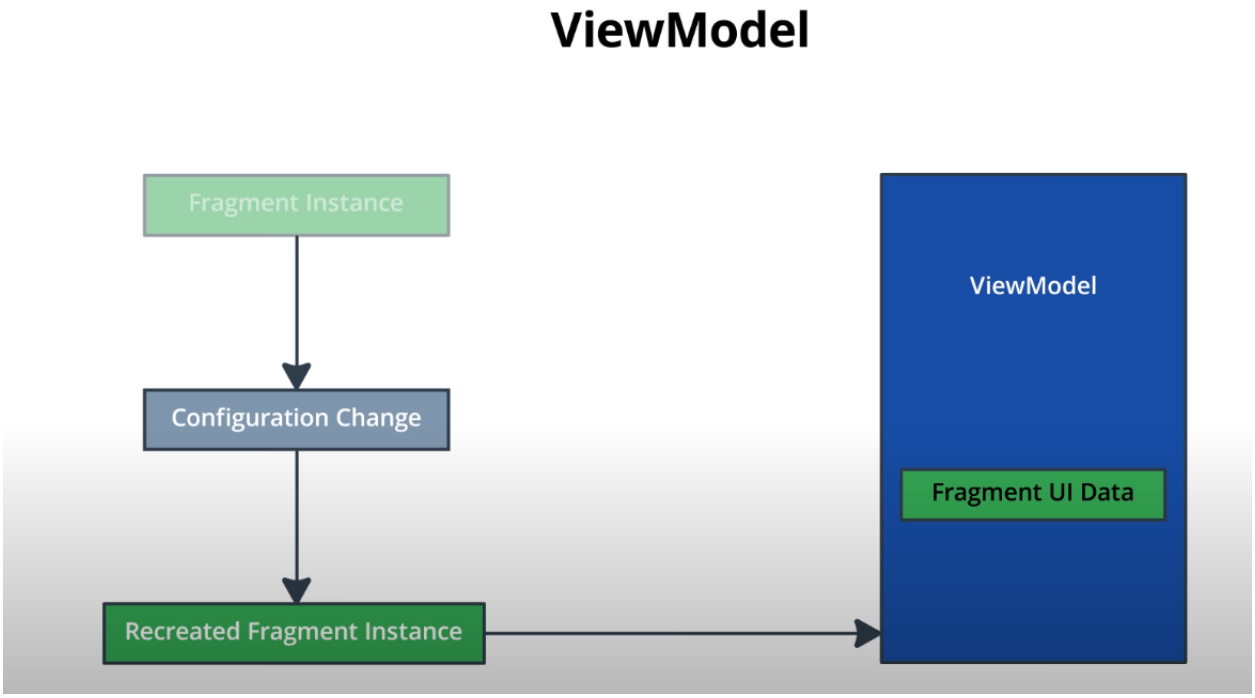






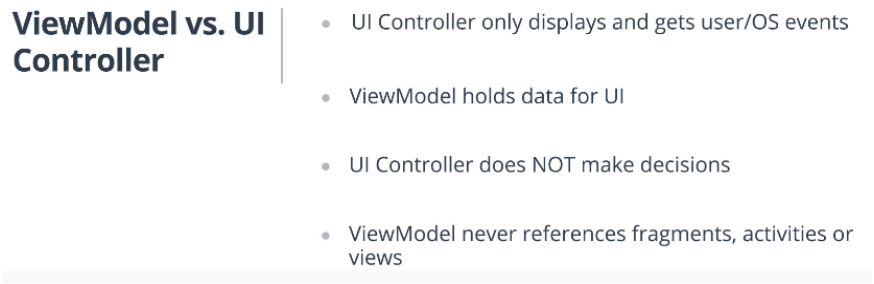
Hình 6.1: Kiến trúc app

6.2. ViewModel



Hình 6.2: ViewModel

Dữ liệu UI sẽ được lưu trong *viewModel*. Khi 1 *instance fragment* mà bị thay đổi gì đó (chẳng hạn như xoay màn hình) thì OS sẽ tạo ra 1 *instance fragment* mới, và chỉ cần tham chiếu đến *ViewModel* là ta có thể lấy lại dữ liệu. Còn nếu không có *viewModel* thì khi xoay màn hình tất cả mọi dữ liệu sẽ bị mất.

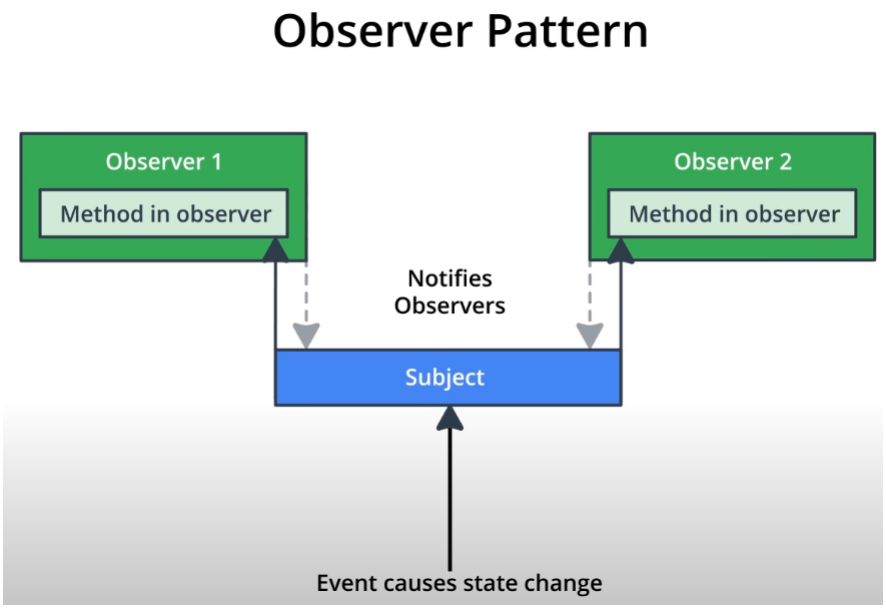


Hình 6.3: ViewModel với UI Controller

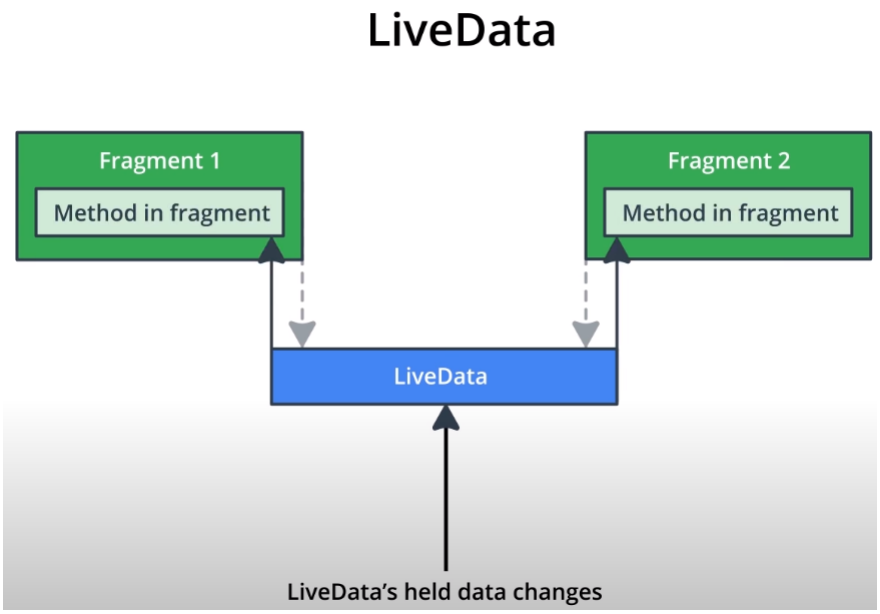
6.3. Live Data

Là một lớp giữ dữ liệu có thể quan sát được và nhận biết được vòng đời. Được sử dụng để kết nối từ *ViewModel* tới UI *Controller*. *Observer pattern* là nơi bạn có

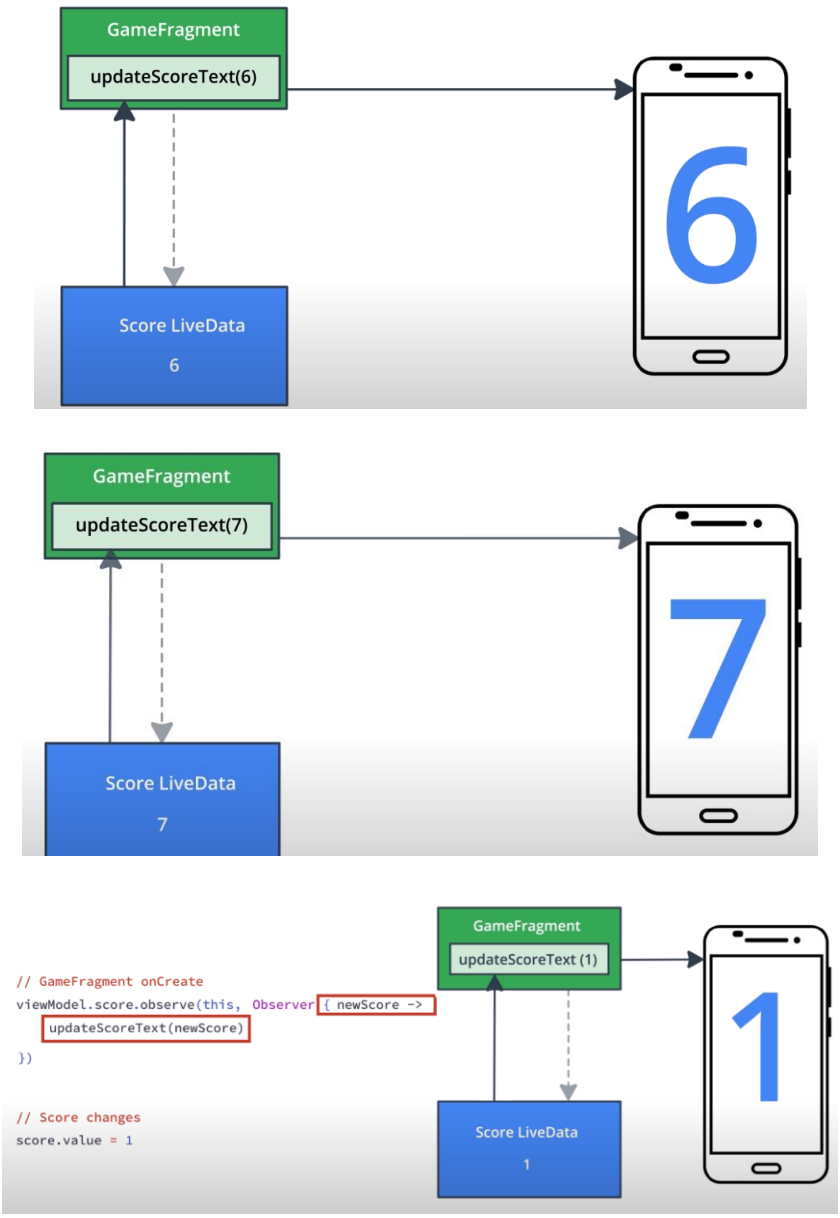
một đối tượng được gọi là SUBJECT (theo dõi danh sách các đối tượng khác được gọi là observers).



Hình 6.4: Observer Pattern



Hình 6.5: LiveData



Hình 6.6: Ví dụ LiveData

GameFragment đang *observer Score LiveData*, bất cứ khi nào *score.value* thay đổi thì nó sẽ thực hiện hàm *updateScoreText*(*newScore*)

6.4. Lifecycle Awareness

LiveData là một lớp lưu trữ dữ liệu có thể quan sát được. Không giống như một *LiveData* có thể quan sát thông thường, *LiveData* nhận biết được vòng đời, có nghĩa là nó biết vòng đời của các thành phần ứng dụng khác, chẳng hạn như *activities*, *fragments*, hoặc *services*. Nhận thức này đảm bảo *LiveData* chỉ cập nhật các trình quan sát thành phần ứng dụng đang ở trạng thái vòng đời hoạt động. *LiveData* coi như

một *observer*, được đại diện bởi lớp *Observer*, đang ở trạng thái hoạt động nếu vòng đời của nó ở trạng thái STARTED hoặc RESUMED. *LiveData* chỉ thông báo cho những *observer* đang hoạt động về các cập nhật. Những *observer* không hoạt động đã đăng ký để xem các đối tượng *LiveData* không được thông báo về các thay đổi. Ta có thể đăng ký một trình observer được ghép nối với một đối tượng triển khai giao diện *LifecycleOwner*. Mối quan hệ này cho phép người quan sát bị loại bỏ khi trạng thái của đối tượng *Lifecycle* tương ứng chuyển thành DESTROYED. Điều này đặc biệt hữu ích cho các hoạt động và mảnh vỡ vì chúng có thể quan sát các đối tượng *LiveData* một cách an toàn và không lo bị rò rỉ — *activities* và *fragments* ngay lập tức được hủy đăng ký khi vòng đời của chúng bị phá hủy.

6.5. Work with Live Data objects

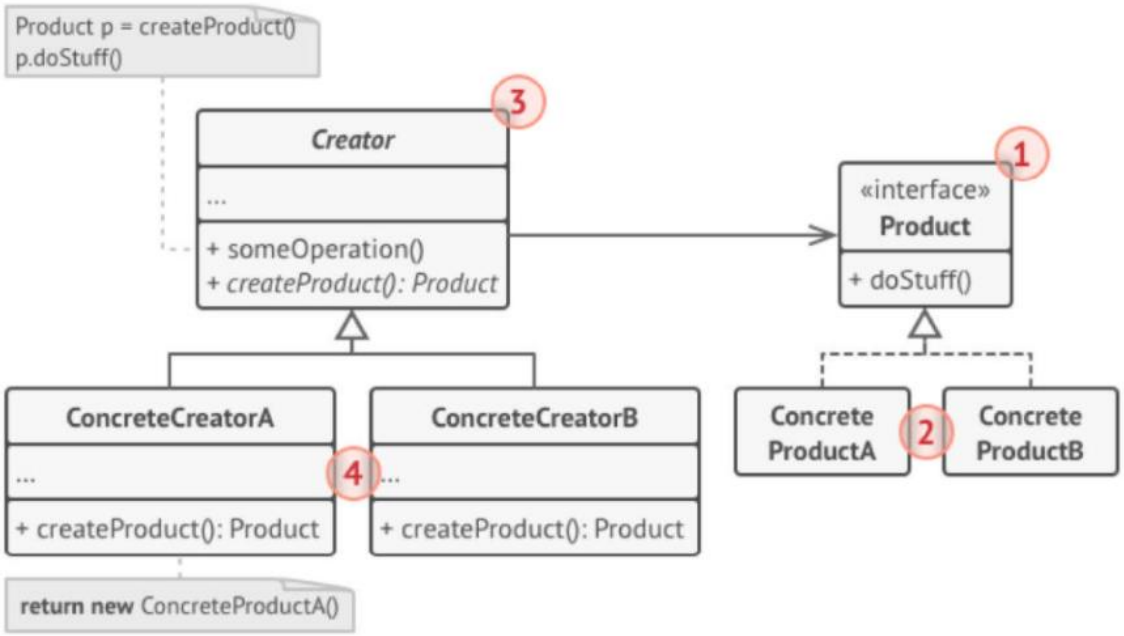
Tạo một *instance* của *LiveData* để chứa một loại dữ liệu nhất định. Điều này thường được thực hiện trong lớp *ViewModel*. Tạo một đối tượng *Observer* định nghĩa phương thức *onChanged* (), phương thức này kiểm soát những gì xảy ra khi dữ liệu của đối tượng *LiveData* thay đổi. Bạn thường tạo một đối tượng *Observer* trong UI *Controller*, chẳng hạn như *activity* hoặc *fragments*. Đính kèm đối tượng *Observer* vào đối tượng *LiveData* bằng phương thức *Observe*(). Phương thức *Obser* () nhận một đối tượng *LifecycleOwner*. Điều này đăng ký đối tượng *Observer* với đối tượng *LiveData* để nó được thông báo về các thay đổi. Bạn thường đính kèm đối tượng *Observer* trong bộ UI *controller*, chẳng hạn như *activity* hoặc *fragments*.

6.6. Add a ViewModelFactory

Để chuyển dữ liệu vào *ViewModel*, phải tạo fatory cho phép bạn xác định một hàm tạo tùy chỉnh cho *ViewModel* được gọi khi bạn sử dụng *ViewModelProviders*. Một *factory* là một đối tượng để tạo các đối tượng khác - chính thức thì *factory* là một hàm hoặc phương thức trả về các đối tượng của một nguyên mẫu hoặc lớp khác nhau từ một số lệnh gọi phương thức, được giả định là "new". Nói một cách rộng rãi hơn, một chương trình con trả về một đối tượng "new" có thể được gọi là "fatory".

Factory Method Pattern: *Factory method* là một *pattern* thuộc nhóm *creational patterns*. Định nghĩa *interface* giúp *client* tạo *object* nhưng ủy quyền cho các *concreate factory* để xác định class nào được trả về cho client. Được sử dụng với mục

đích: Đưa toàn bộ logic của việc tạo mới object vào trong factory, che giấu logic của việc khởi tạo nhằm giảm sự phụ thuộc nhằm tăng tính mở rộng. Vì những đặc điểm trên nên *factory pattern* thường được sử dụng trong các thư viện (người sử dụng đạt được mục đích tạo mới object và không cần quan tâm đến cách nó được tạo ra)



Hình 6.7: Factory pattern

Product là một *interface* chung mà tất cả các product con đều phải implements.

Mỗi *concrete product* là một *product* con cụ thể *implements* từ interface Product.

Cliass *Creator* là nơi khai báo *factory method* có kiểu dữ liệu trả về là *Product* cho phép trả về các *product* mới.

Mỗi *Concrete Creator* là một class kế thừa từ *Creator* có nhiệm vụ *override factory method* và trả về loại *product* tương ứng.

6.7. Add LiveData Data Binding

Bạn sẽ sử dụng *LiveData* để tự động cập nhật layout thông qua data binding.. *Binding.setLifecleOwner* (this): cho phép *LiveData* tự động cập nhật dữ liệu ràng buộc bởi *LAYOUTS.*

CHƯƠNG 7: Kiến trúc ứng dụng (Persistence)

7.1. SQLite Primer

SQLite triển khai một công cụ cơ sở dữ liệu SQL có các đặc điểm sau:

- Khép kín (không yêu cầu các thành phần khác)
- Serverless (không yêu cầu server backend)
- Không cấu hình (không cần phải định cấu hình cho ứng dụng của bạn)
- Giao dịch (các thay đổi trong một giao dịch duy nhất trong SQLite xảy ra hoàn toàn hoặc không xảy ra)

SQLite là công cụ cơ sở dữ liệu được triển khai rộng rãi nhất trên thế giới.

Mã nguồn cho SQLite nằm trong miền công cộng.

Ví dụ về bảng:

- A database named DATABASE_NAME
- A table named WORD_LIST_TABLE
- Columns for _id, word, and definition

After inserting the words alpha and beta, where alpha has two definitions, the table might look like this:

DATABASE_NAME > WORD_LIST_TABLE

_id	Word	Definition
1	"alpha"	"first letter"
2	"beta"	"second letter"
3	"alpha"	particle"

Ta có thể thông tin trong 1 hàng sử dụng *_id*.

Ta có thể sử dụng ngôn ngữ truy vấn SQL để tương tác với *database*, các truy vấn có thể phức tạp nhưng chỉ có 4 phương thức cơ bản:

- Inserting rows
- Deleting rows
- Updating values in rows
- Retrieving rows that meet given criteria

Trong Android, đối tượng truy cập dữ liệu (DAO) cung cấp các phương pháp tiện lợi để chèn, xóa và cập nhật cơ sở dữ liệu.

Truy vấn thường có dạng:

SELECT word, definition

FROM WORD_LIST_TABLE

WHERE word="alpha"

Một ví dụ cơ bản:

```
SELECT columns

FROM table

WHERE column="value"
```

SELECT <<columns>>: Chọn cột để trả về. Sử dụng "*" để trả về tất cả các cột.

FROM <<table>>: Chỉ định bảng mà lấy kết quả.

WHERE: Optional keyword that precedes conditions that have to be met, for example column="value". Common operators are =, *, LIKE, <, and >. To connect multiple conditions, use AND or OR.

ORDER BY: Cụm từ khóa tùy chọn để sắp xếp kết quả theo cột được chỉ định. Chỉ định ASC cho tăng dần và DESC cho giảm dần. Nếu bạn không chỉ định đơn hàng, bạn sẽ nhận được đơn hàng mặc định, có thể không có thứ tự.

LIMIT: Từ khóa để chỉ định một số kết quả hạn chế.

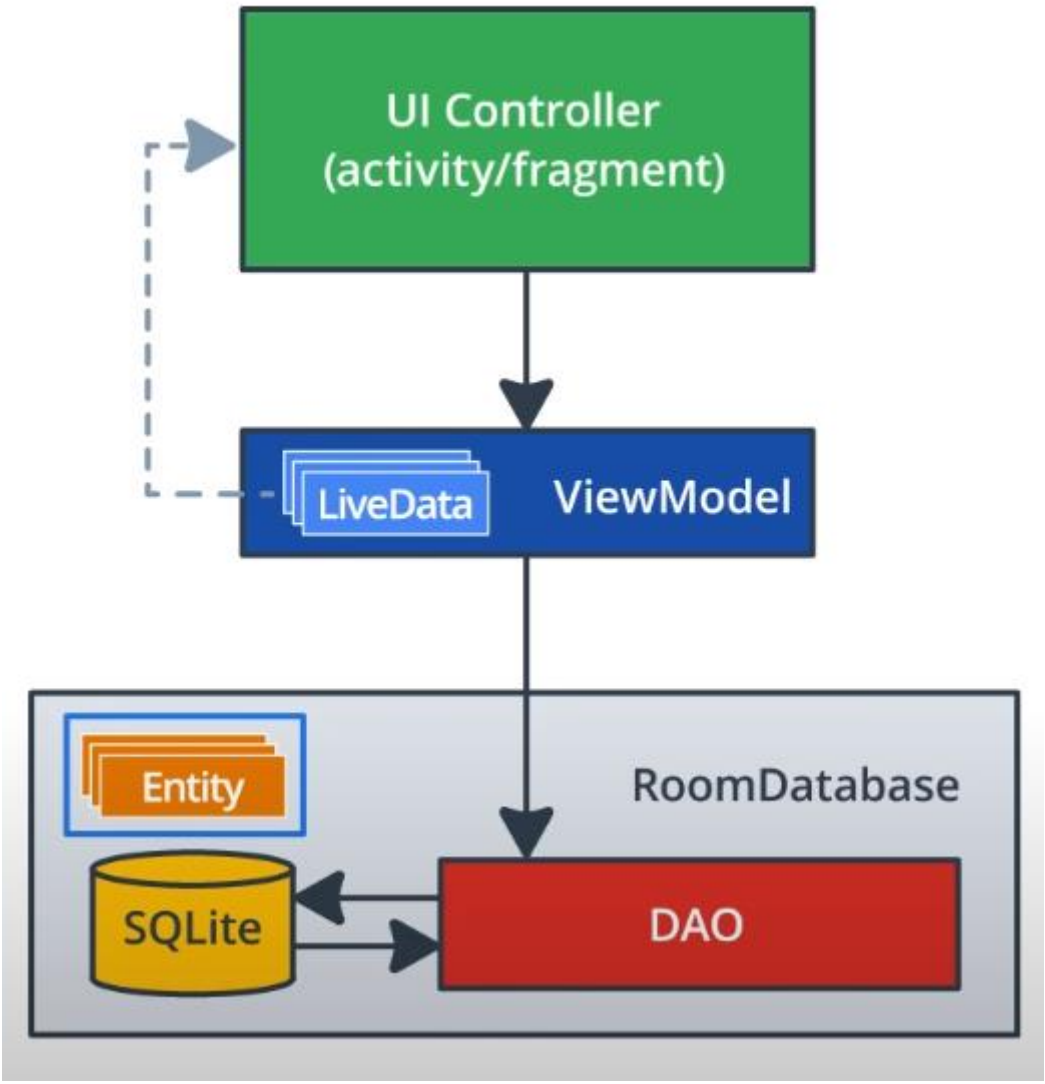
Các truy vấn sau sử dụng bảng đã xác định trước đó WORD_LIST_TABLE.

Query	Result
SELECT * FROM WORD_LIST_TABLE	Lấy các hàng trong bảng WORD_LIST_TABLE.
SELECT word, definition FROM WORD_LIST_TABLE WHERE _id > 2	Selects the word and definition columns of all items with an id greater than 2. Returns[["alpha", "particle"]]
SELECT _id FROM WORD_LIST_TABLE WHERE word="alpha" AND definition LIKE "%art%"	Returns the id of the word alpha with the substring art in the definition. [["3"]]
SELECT definition FROM WORD_LIST_TABLE ORDER BY word DESC LIMIT 1	Selects all definitions. Sorts in reverse and gets the first row after the list is sorted. Sorting is by the column specified which is word. Note that we can sort by a column that we don't return! [["second letter"]]
SELECT * FROM WORD_LIST_TABLE LIMIT 2,1	Returns 1 item starting at position 2. Position counting starts at 1 (not zero!). Returns [["2", "beta", "second letter"]]

7.2. Designing Entities

Entity: Đối tượng hoặc khái niệm được lưu trữ trong cơ sở dữ liệu. Lớp thực thể định nghĩa một bảng, mỗi instance được lưu trữ dưới dạng một hàng bảng.

Request: Yêu cầu dữ liệu hoặc thông tin từ bảng cơ sở dữ liệu hoặc các bảng hoặc yêu cầu thực hiện một số hành động trên dữ liệu.



Hình 7.1: Kiến trúc app mịn hơn

7.3. Data Access Object (DAO)

For common database operations, the Room library provides convenience annotation such as @Insert, @Delete, @Update, @Query (Cái này giúp viết bất kỳ query nào được hỗ trợ bởi SQLite chứ không phải viết ở giao diện hỗ trợ query database)

```
@Dao
interface SleepDatabaseDao {

    @Insert
    fun insert(night: SleepNight)

    @Query("SELECT * FROM daily_sleep_quality_table " +
           "ORDER BY nightId DESC")
    fun getAllNights(): LiveData<List<SleepNight>>

}
```

Hình 7.2: DAO

2 dòng đầu định nghĩa 1 *interface* giải thích rằng DAO sẽ bảo bộ biên dịch về vai trò của *inferface* để định nghĩa cách truy cập vào *Room database*. 2 dòng tiếp định nghĩa 1 hàm *insert* 1 tham số kiểu *SleepNight*. 2 dòng cuối là truy vấn, *Room* sẽ cập nhật *LiveData* bất cứ khi nào có gì đó thay đổi nhờ vào *LiveData*.

Để tạo 1 *Room* Database:

- Tạo 1 lớp trừu tượng public kế thừa Room database
- Chú thích lớp với cơ sở dữ liệu và trong các đối số khai báo thực thể cho cơ sở dữ liệu và thiết lập phiên bản.
- Liên kết với DAO.
- Lấy tham chiếu tới Database.

7.5. MultiThreading

Làm nhiều nhiệm vụ cùng được gọi là *Multitasking*. Theo cách tương tự, nhiều luồng chạy cùng lúc trong một máy được gọi là *Multi-Threading*. Về mặt kỹ thuật, một luồng là một đơn vị của một quy trình. Nhiều luồng như vậy kết hợp với nhau để tạo thành một quy trình. Điều này có nghĩa là khi một tiến trình bị phá vỡ, số luồng tương đương sẽ có sẵn. Có nhiều *processor*, mỗi *thread* là 1 đơn vị thực thi song song (concurrent unit of execution). Mỗi *thread* có *call stack* riêng cho các phương thức được gọi, các tham số và biến địa phương của chúng. Mỗi thực thể máy ảo (mỗi máy ảo dành cho 1 tiến trình – một ứng dụng đang chạy), khi được chạy, sẽ có ít nhất một *thread* chính chạy, thông thường có vài thread khác dành cho các nhiệm vụ phục vụ *thread* chính. Ứng dụng có thể bật các thread bổ sung để phục vụ các mục đích cụ thể. Các *thread* trong cùng một máy ảo tương tác và đồng bộ hóa với nhau qua việc sử dụng các đối tượng dùng chung (*shared objects*) và các *monitor* (*module* kiểm soát việc dùng chung) gắn với các đối tượng này.

Có hai cách chính để chạy một thread từ trong mã ứng dụng.

- Tạo một lớp mới extend lớp *Thread* và *override* phương thức *run*().
- Tạo một *instant* mới của lớp *Thread* với một đối tượng *Runnable* . Trong cả hai cách, cần gọi phương thức *start*() để thực sự chạy *Thread* mới.

Cách tiếp cận của Android đối với các việc tốn thời gian: Một ứng dụng có thể có một hoạt động tốn thời gian, tuy nhiên, ta muốn UI vẫn đáp ứng tốt đối với các tương tác của người dùng. Android cung cấp hai cách để xử lý tình huống này:

- Thực hiện thao tác đó trong một service ở *background* và dùng *notification* để thông báo cho người dùng về bước tiếp theo
- Thực hiện thao tác đó trong một *background thread*. Các *thread* của *Android* tương tác với nhau bằng cách sử dụng (a) các đối tượng *Handler* và (b) post các đối tượng *Runnable* tới *view* chính.

7.6. Handler Class

Khi một tiến trình được tạo cho một ứng dụng, *main thread* của nó được dành riêng để chạy một *message queue*, *queue* này quản lý các đối tượng bậc cao của ứng dụng

(activity, intent receiver, v.v..) và các cửa sổ mà chúng tạo ra. Ta có thể tạo các *thead* phụ, chúng tương tác với *thread* chính của ứng dụng qua một *Handler*. Khi ta tạo một *Handler* mới, nó được gắn với *message queue* của *thread* tạo ra nó – từ đó trở đi, nó sẽ gửi các *message* và các *runnable* tới *message queue* đó và thực thi chúng khi chúng ra khỏi *message queue*.

Hai ứng dụng chính của *Handler*:

- Xếp lịch cho các *message* và *runnable* cần được thực thi vào thời điểm nào đó trong tương tai,
- Xếp hàng một *action* cần thực hiện tại một *thread* khác

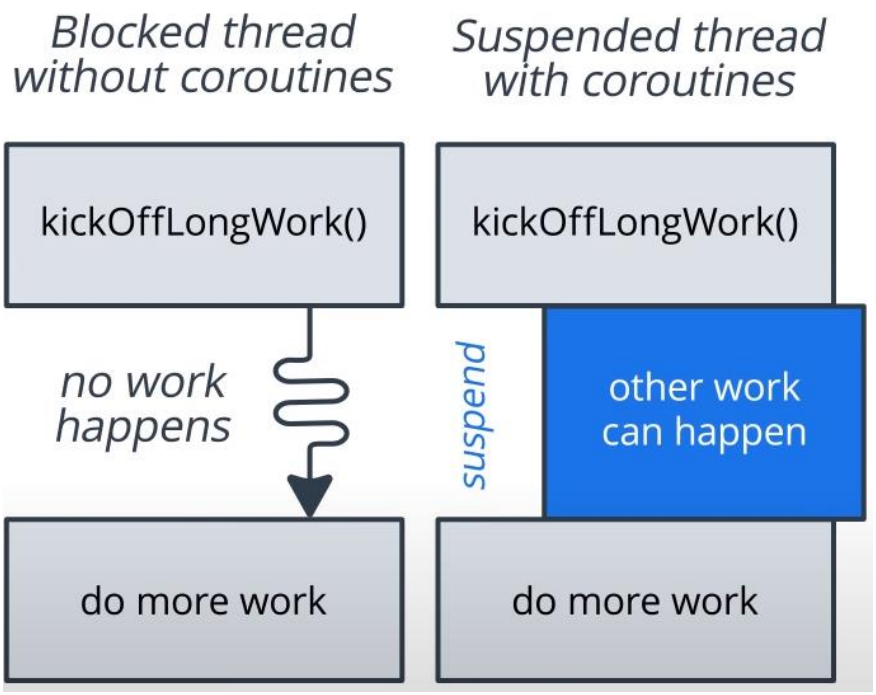
7.7. AsyncTask class

AsyncTask cho phép sử dụng UI *thread* một cách dễ dàng và đúng cách. AsyncTask cho phép thực hiện các hoạt động background và gửi kết quả cho UI thread mà không phải thao tác với *thread* và/hoặc *handler*. Một tác vụ không đồng bộ (asynchronous task) là một nhiệm vụ tính toán chạy tại một *background thread* và kết quả sẽ được gửi cho *UI thread*. Một *asynchronous task* được định nghĩa bởi: Kiểu tổng quát, trạng thái chính, phương thức hỗ trợ.

7.8. Coroutines

Định nghĩa gốc: “Handle long-running tasks elegantly and efficiently:

- Asybchronous: The corountine runs independently from the main execution steps of your program: paralell
- Non-blocking: The system is not blocking the main or UI thread. User có trải nghiệm mượt vì UI interaction will always have priority.
  Khác nhau giữa block và suspend”



Hình 7.3: Khác nhau giữa block và suspend

- Sequential code

To use corountine, need:

- Job: is anything that can be canceled. A background job. Conceptually, a job is a cancellable thing with a life-cycle that culminate in its completion.
- Dispatcher: send off coroutines to run on various threads.
- Scope: combine information, including a job and dispatcher to define the context in which the corountine runs.

CHƯƠNG 8: RecyclerView

8.1. Giới thiệu

RecyclerView giúp dễ dàng hiển thị hiệu quả tập dữ liệu lớn. Bạn cung cấp dữ liệu và xác định giao diện của từng mục và thư viện *RecyclerView* sẽ tự động tạo các phần tử khi chúng cần. Như tên của nó, *RecyclerView* tái chế các phần tử riêng lẻ đó. Khi một mục cuộn ra khỏi màn hình, *RecyclerView* không phá hủy *view* của nó. Thay vào đó, *RecyclerView* sử dụng lại view cho các mục mới đã cuộn trên màn hình. Việc

tái sử dụng này giúp cải thiện đáng kể hiệu suất, cải thiện khả năng phản hồi của ứng dụng và giảm mức tiêu thụ điện năng.

Ngoài là tên của lớp, *RecyclerView* còn là tên của thư viện.

- Hiệu quả.
- Display complex items Hiển thị các items phức tạp.
- Có thể tùy chỉnh

RecyclerView Layout:

- Layout list or grid
- Scroll horizontal or vertical
- Supports custom layouts

Other list:

- ListView
- GridView
- LinearLayout: Hiển thị ít phần tử tầm 3 đến 5 cái.

8.2. Key classes

Một số lớp khác nhau làm việc cùng nhau để tạo danh sách động của bạn:

RecyclerView là ViewGroup chứa các *view* tương ứng với dữ liệu của bạn. Bản thân nó là một *view*, vì vậy bạn thêm *RecyclerView* vào bố cục của mình theo cách bạn thêm bất kỳ phần tử giao diện người dùng nào khác.

Mỗi phần tử riêng lẻ trong danh sách được xác định bởi đối tượng *View holder*. Khi *View holder* được tạo, nó không có bất kỳ dữ liệu nào được liên kết với nó. Sau khi *View holder* được tạo, *RecyclerView* sẽ liên kết nó với dữ liệu của nó. Bạn xác định *view holder* bằng cách extending *RecyclerView.ViewHolder*. *RecyclerView* yêu cầu các view đó và gắn các *views* với dữ liệu của chúng, bằng cách gọi các phương thức *adapter*. Bạn xác định *adapter* bằng cách extending *RecyclerView.Adapter*. *Layout manager* sắp xếp các phần tử riêng lẻ trong danh sách của bạn. Bạn có thể sử dụng một trong những *Layout manager* do thư viện *RecyclerView* cung cấp hoặc bạn

có thể xác định trình quản lý bố cục của riêng mình. Các *Layout manager* đều dựa trên lớp trừu tượng *LayoutManager* của thư viện.

8.3. Các bước triển khai RecyclerView

Trước hết, hãy quyết định *list* hoặc *grid* sẽ trông như thế nào. Thông thường, bạn sẽ có thể sử dụng một trong các *layout managers* tiêu chuẩn của thư viện RecyclerView. Thiết kế mỗi phần tử trong danh sách sẽ trông như thế nào và hoạt động như thế nào. Dựa trên thiết kế này, hãy *extend* lớp ViewHolder. Phiên bản *ViewHolder* của bạn cung cấp tất cả các chức năng cho các mục trong danh sách của bạn. *View holder* là một trình bao bọc xung quanh *View* và *View* đó được quản lý bởi *RecyclerView*. Xác định *Adapter* liên kết dữ liệu của bạn với các dạng xem *ViewHolder*.

8.4. Lập Layout

Các mục trong RecyclerView của bạn được sắp xếp bởi một lớp *LayoutManager*. Thư viện *RecyclerView* cung cấp ba *layout managers*, xử lý các tình huống bố cục phổ biến nhất:

- *LinearLayoutManager* sắp xếp các mục trong danh sách một chiều.
- *GridLayoutManager* sắp xếp tất cả các mục trong một grid hai chiều:
    - Nếu lưới được sắp xếp theo chiều dọc, *GridLayoutManager* cố gắng làm cho tất cả các phần tử trong mỗi hàng có cùng chiều rộng và chiều cao, nhưng các hàng khác nhau có thể có chiều cao khác nhau.
    - Nếu lưới được sắp xếp theo chiều ngang, *GridLayoutManager* sẽ cố gắng làm cho tất cả các phần tử trong mỗi cột có cùng chiều rộng và chiều cao, nhưng các cột khác nhau có thể có chiều rộng khác nhau.
- *StaggeredGridLayoutManager* tương tự như *GridLayoutManager*, nhưng nó không yêu cầu các mục trong một hàng có cùng chiều cao (đối với lưới dọc) hoặc các mục trong cùng cột có cùng chiều rộng (đối với lưới ngang). Kết quả là các mục trong một hàng hoặc cột có thể bù trừ cho nhau.

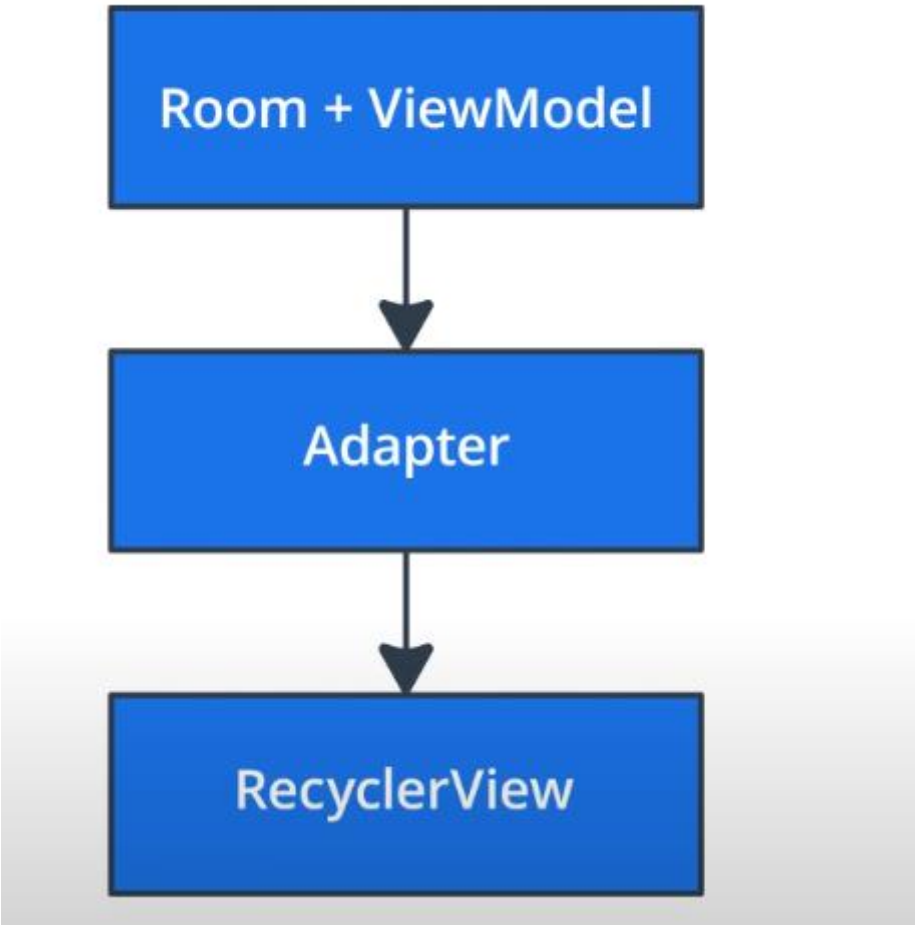
Triển khai adapter và view holder: Khi bạn đã xác định bố cục của mình, bạn cần triển khai *Adapter* và *ViewHolder* của mình. Hai lớp này làm việc cùng nhau để xác định

cách dữ liệu của bạn được hiển thị. *ViewHolder* là một trình bao bọc xung quanh một *View* có chứa bố cục cho một mục riêng lẻ trong danh sách. *Adapter* tạo các đối tượng *ViewHolder* khi cần thiết và cũng thiết lập dữ liệu cho các *view* đó. Quá trình liên kết các *view* với dữ liệu của chúng được gọi là *binding*.

Khi bạn xác định *adapter*, bạn cần ghi đè ba phương thức chính:

- onCreateViewHolder (): *RecyclerView* gọi phương thức này bất cứ khi nào nó cần tạo một *ViewHolder* mới. Phương thức này tạo và khởi tạo *ViewHolder* và *View* liên quan của nó, nhưng không điền nội dung vào *view* — ViewHolder vẫn chưa bị ràng buộc với dữ liệu cụ thể.
- onBindViewHolder (): *RecyclerView* gọi phương thức này để liên kết *ViewHolder* với dữ liệu. Phương thức này tìm nạp dữ liệu thích hợp và sử dụng dữ liệu để điền vào bố cục của *view holder*. Ví dụ: nếu RecyclerView hiển thị danh sách tên, thì phương thức này có thể tìm tên thích hợp trong danh sách và điền vào tiện ích *TextView* của *view holder*.
- getItemCount (): *RecyclerView* gọi phương thức này để lấy kích thước của tập dữ liệu. Ví dụ: trong một ứng dụng sổ địa chỉ, kích thước có thể là tổng số địa chỉ. RecyclerView sử dụng điều này để xác định thời điểm không còn mục nào có thể được hiển thị.

8.5. You first RecyclerView



Hình 8.1: Recycle View

Adapter: giao diện convert thành 1 thứ gì đó.

Adapter interface cần có:

- Bao nhiêu items.
- Vẽ 1 item như nào.
- Tạo 1 view mới.

RecyclerView tương tác với ViewHolders (Chứa các views, chứa thông tin cho RecyclerView, giao diện chính của RecyclerView). Classs chứa những đối tượng để view lên màn hình (Các items). File xml vẽ giao diện mỗi dòng. Adapter là bản vẽ, khi có dữ liệu và giao diện mỗi dòng thì vẽ lên bản vẽ thôi.

RecycleView bên trong có các layout item. Mình có danh sách các entity data, cần chuyển cái data để render lên cái item đấy. ViewHolder là cách để vẽ cái item.

DiffUtils hỗ trợ các thuật toán để hỗ trợ tk nào thay đổi cho vào adapter + viewholder để render các thành phần của recyleView không ảnh hưởng đến các item khác.

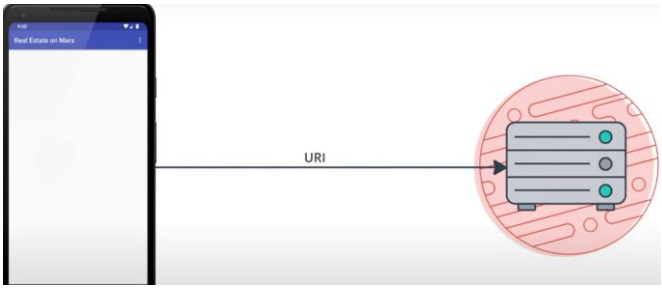
CHƯƠNG 9: CONNECT TO THE INTERNET

9.1. RESTful Services

URI: Chỉ định dữ liệu bạn muốn.

Một URL là một URI.

Request được thực hiện đối với các dịch vụ Web RESTful theo cách chuẩn hóa thông qua URI.



Hình 9.1: URI

Mỗi yêu cầu dịch vụ Web chứa một URI và được chuyển tới máy chủ của bằng cùng một giao thức HTTP được các trình duyệt Web sử dụng.

Hình 9.2: HTTP URI

Các hoạt động mà yêu cầu HTTP có thể cho máy chủ biết phải làm gì:

- GET
- Post/ PUT
- Delete

Hầu như bất kỳ dịch vụ RESTful nào cũng sẽ chấp nhận các tham số truy vấn như một phần của URI. *Query parameters* giống như *parameters* của *funtion* trong *Kotlin*. File *MarsApiService* chứa các tầng *network* (The API that the ViewModel uses to communicate with our web sevice).

Định nghĩa thêm (gốc): “Retrofit is a library that creates a network API for our app based on the content from our web service. Retrofit basics: What Retrofit needs to build our Network API: Base URL of our web service, Converter factory that allows Retrofit to return the server response in useful format”.

KẾT LUẬN

Những ngành nghề liên quan tới công nghệ thông tin ngày càng phát triển mạnh mẽ để bắt kịp với xu hướng chung. Trong đó, lập trình ứng dụng di động iOS – Android (hay còn gọi là thiết kế App Mobile) trở nên phổ biến. Những kiến thức trên bài báo cáo chỉ là những phần cơ bản về Kotlin để hành trang tiếp bước nâng cao sau này. Với kết quả thu được như vậy, bài báo cáo sẽ cố gắng phát triển thu thập thêm những kiến thức nâng cao hơn chuyên sâu hơn về Kotlin để bắt kịp công nghệ hiện tại.

TÀI LIỆU THAM KHẢO:

[1] **https://classroom.udacity.com**

[2] **https://developer.android.com**

[3] **https://kotlinlang.org/**