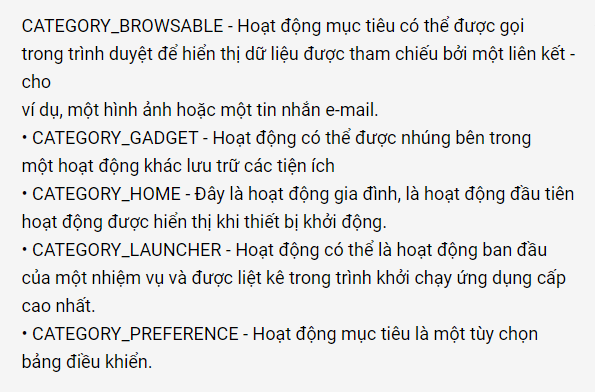
**Overview**

* Android là hệ điều hành được xây dựng trên nền tảng di động
* Kiến trúc android: 4 layer chính (từ dưới lên):
  + Tầng cuối cùng: Linux Kernel: Android được xây dựng dựa trên hdh Linux - đảm nhận xử lý các vđề IO, Thread, Mem, Driver, Power…
  + Libraries: là tập hợp các thư viện C/C++, hỗ trợ Database, xử lý ảnh, media, OpenGL + Android Runtime: Cung cấp thư viện core để chạy java, File access, Network access…, DVM (Máy ảo - Dalvik Virtual Machine (Tương tự Java Virtual Machine)) - tạo ra môi trường để chạy ứng dụng
  + Application Framework: Tập những khung hỗ trợ pt ứng dụng: Activity, Resources, View, Notification, Content Provider… (Bộ API). Ứng dụng xử lý chủ yếu ở libraries, tầng này chỉ đóng vai trò như 1 dạng kết nối
  + Application: ứng dụng ThirdParty, trình duyệt, danh bạ, lịch…
* .java -> (Java Compiler) -> .class -> (dx) -> .dex (dx : Dalvik Excutable)
* src: lưu trữ source code
* gen: tự động gen id (R.id/ R.java…)
* R class: truy cập resources… (tự sinh ra, không sửa)
* adb: Android Debug Bridge (hỗ trợ cài ứng dụng, log, debug…)

**Activity**

* Activity như là 1 điểm bắt đầu để ngdùng có thể giao tiếp với ứng dụng
* View, Fragment đều nằm trong activity
* Hạn chế tạo nhiều activity để giảm performance
* Khi startActivity, nó sẽ được lưu trữ vào 1 backstack (LIFO)
* Activity in Application Structure: 3 tầng:
  + UI layer: Activity, view, fragment (Giao diện)
  + Bussiness layer: Bussiness classes, service, receiver… (Xử lý logic)
  + Data storage layer: Databases, storage, shared preference (Data)
* Vòng đời activity: Create-Start-Resume-”Running”-Pause-Stop-Destroy
  + onCreate: là lúc init các thành phần
  + onStart: là lúc UI bắt đầu được visible (Chưa tương tác được)
    - onRestoreInstanceState()
  + onResume: là lúc UI đã visible và ngdùng có thể tương tác
    - onSaveInstanceState()
  + onPause: là khi có bất kì thứ j hiển thị đè lên activity (Trừ Toast)
  + onStop: là khi UI bị tắt đi hoặc chuyển qua UI khác (Không còn visible)
  + onDestroy: là khi activity bị hủy hoặc app bị kill
* Lifetime activity:
  + From create to destroy: Entire lifetime (Setup or release activity’s global state)
  + From start to stop: Visible lifetime (Maintain resources to show activity to user)
  + From resume to pause: Foreground lifetime (Maintain resources to interact with user)
* startActivityForResult: sử dụng với mục đích là gọi 1 activity mới, hoàn thành một số công việc trong activity đó và nhận kết quả trả về
* Task: đây là 1 nơi lưu trữ các callback của activity (LIFO) tương tự như 1 backstack
* LauchMode: Có thể khai báo trong manifest hoặc sử dụng addFlags(Intent.X)
  + Standard: LIFO
  + SingleTop: Nếu activity đó đang ở top, khởi tạo thêm 1 activity đó, nó sẽ không tạo mới, intent được sử dụng để khởi tạo sẽ được đưa tới Instance của activity đó qua onNewIntent()
  + SingleTask: Trong cùng 1 task, activity chỉ có thể xuất hiện 1 lần, khi gọi lần thứ 2, tất cả các activity giữa 2 lần gọi sẽ bị xóa đi, activity sẽ không được tạo mới, intent được đưa tới Instance như cách trên
  + SingleInstance: Activity khởi tạo sẽ được khởi tạo ở 1 task khác nếu task đó đã có activity rồi, hoặc nếu task đó có activity này, các activity mới khởi tạo sẽ được add vào 1 task mới (Tóm lại, activity này luôn ở riêng 1 task)

**Intent, Boardcast**

* 4 components chính:
  + Activities: Biểu diễn UI, thực hiện các hành động trên màn hình
  + Services: Chạy trong background để thực hiện các hoạt động, ví dụ chạy nhạc nền trong khi sử dụng ứng dụng khác
  + Boardcast Receivers: Về cơ bản là công cụ giao tiếp giữa ứng dụng vs ứng dụng hoặc ứng dụng vs hệ thống
  + Content Providers: Cung cấp dữ liệu từ ứng dụng này qua các ứng dụng khác tùy theo yêu cầu, những yêu cầu này được xử lý bởi bởi các phương thức của ContentResolver
* Intent: là phương tiện giao tiếp giữa các module cơ bản (nó là 1 dạng message bất động bộ), các module chạy trên các luồng khác nhau, để giao tiếp ta cần intent
  + Component name: Define component được start (optional)
  + Action: Có thể sử dụng action hệ thống hoặc customAction (ví dụ Intent.ACTION\_BATTERY\_LOW, Intent.ACTION\_CALL…)
    - Khi define action, intent sẽ được hệ thống cấu trúc các trường data và extras… (2 thành phần đi kèm action)
  + Data: Sử dụng Uri để tham chiếu đối tượng và thực hiện hành động, thường đi kèm và được quyết định bởi Action. (Ví dụ ACTION\_EDIT, Uri sẽ tham chiếu tới data đang cần edit) / (Ví dụ bạn cần người dùng đọc 1 bản hướng dẫn hoặc thỏa thuận người dùng, Uri sẽ là đường dẫn tuyệt đối, và sử dụng ACTION để quyết định hành động với file đó)
  + Category: Define cho đối tượng mà intent đang nhắm đến, nó như 1 chuỗi chứa thông tin bổ sung về loại thành phần xử lý intent. Để nhận các intent không tường minh thì bắt buộc phải khai báo trong <intent-filter>, mặc định là category.DEFAULT, một số category: 
  + Extras: Chứa cặp key-value để truyền dữ liệu, như đã nói ở trên, 1 số Action hệ thống đã có define sẵn data và extras, còn không thì có thể tự define
  + Flags: Nó có chức năng như siêu dữ liệu cho intent. Có thể ra lệnh cho hệ thống về cách hoạt động của đối tượng đang khởi tạo, cách xử lý đối tượng hiện tại sau khi đối tượng mới được khởi tạo…
* Có 2 loại intent:
  + Explicit Intent: Tường minh - là loại intent mà được define đích đến cụ thể và các đối tượng hành động
  + Implicit Intent: Không tường minh - không define rõ ràng đích đến, nó chỉ ra hành động cần làm (action) và dữ liệu cho hành động (data). Khi sử dụng, hệ thống sẽ sử dụng intent-filter để tìm kiếm tất cả các thành phần thích hợp để start nó. Nếu có nhiều intent-filter tương thích thì hệ thống sẽ hiển thị hộp thoại để người dùng lựa chọn manual
  + Intent-Filter: Là 1 thẻ trong manifest nhằm xác định loại intent mà đối tượng đang chứa intent-filter này muốn nhận. Nếu 1 app không có intent-filter để nhận, thì nó chỉ có thể được gọi = 1 explicit intent
  + Data-Transfer: Đối với implicit intent, cần define action, data và cả type nếu cần thiết, sau đó thì put và get extra như với explicit intent
* PendingIntent: thường sử dụng cho việc gửi dữ liệu cho ứng dụng khác (NotificationManager, AlarmManager, Thirdparty…), thậm chí cả việc cho phép ứng dụng khác sử dụng permission trong ứng dụng của bạn để thực hiện hành động
  + Ví dụ khi bạn muốn lấy 1 broadcast: sử dụng PendingIntent.getBroadcast() hoặc lấy 1 activity: sử dụng PendingIntent.getActivity()
  + PendingIntent # Intent thường
* Broadcast Receiver: Handle hàm hệ thống (Gửi/nhận tin nhắn, cuộc gọi), nó cũng handle từ các thirdparty…
  + Để sử dụng broadcast receiver, cần define các permission liên quan và intent-filter dùng để lọc intent
  + Khi sử dụng broadcast receiver, cần chú ý đến lifecycle của nó: Sau khi register và sử dụng, cần unregister nếu không sẽ dễ gặp lỗi memory leak

**ViewGroup**

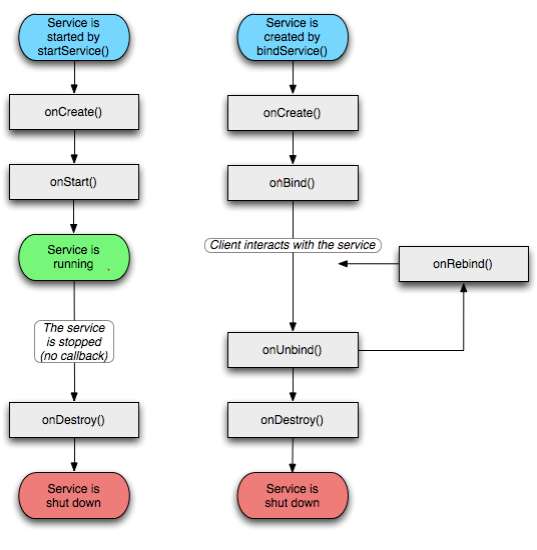
* Relative layout: là 1 viewgroup hiển thị các view ở vị trí tương đối, vị trí của view được hiển thị tương quan dựa trên các view anh em.
* Constrain layout: ra đời với mục đích làm đơn giản hóa cây layout, nó cung cấp công cụ để biểu diễn chính xác các thành phần trong layout mà không cần thêm bất cứ layout con nào hỗ trợ.
* Frame layout: là loại view đơn giản nhất, có thể chứa nhiều view nhỏ bên trong và mục đích chính của nó được sinh ra chính là để chứa view. Nó tạo thành cơ sở để thiết kế các view phức tạp hơn ở bên trong. Tuy nhiên khi có quá nhiều view con, không nên sử dụng frame layout vì cấu trúc thiết kế của nó quá đơn giản, khó xử lý và sắp xếp view con

**Fragment**

* Fragment là 1 thành phần android độc lập, được sử dụng bởi 1 activity, nó có view, có lifecycle riêng, fragment thường đi kèm file fragment\_xml, nếu không có xml, nó được gọi là headless fragment
* 1 Activity có thể sử dụng nhiều fragment cùng 1 lúc, fragment có thể khai báo trực tiếp bằng xml (static fragment) hoặc khai báo trong code (dynamic fragment)
* Vòng đời fragment: Attach-Create-CreateView-ActivityCreated-Start-Resume-Pause-Stop-DestroyView-Destroy-Detach
  + onAttach: tạo tham chiếu từ fragment đến activity cha và thực hiện 1 số bước khởi tạo
  + onCreate: khởi tạo fragment
  + onCreateView: tạo giao diện, trả về view là file xml tương ứng, chưa được khởi tạo đầy đủ
  + onActivityCreated: thực hiện nốt việc khởi tạo, có thể init
  + onStart: hiển thị fragment lên màn hình
  + onResume: chính thức hoạt động
  + onPause: bị tạm dừng bởi các thành phần hiển thị khác đè lên
  + onStop: bị ẩn đi (có thể vì gọi fragment khác)
  + onDestroyView: giao diện bị hủy, nếu được gọi lại, nó sẽ quay về onCreateView
  + onDestroy: bị hủy
  + onDetach: bị hủy hoàn toàn
* Các ứng dụng của fragment: tạo màn hình linh hoạt nhiều ngăn, kết hợp với các tiện ích khác: ActionBar, ViewPager, NavigationDrawer…
* FragmentManager: sử dụng để tương tác với các fragment đặt trong activity
  + FragmentTransaction: là 1 thành phần của FragmentManager được sử dụng để quản lý state của fragment, thực hiện các hành động tương tác trực tiếp với fragment như show(), add(), remove(), hide(), replace()...
  + Bản thân fragment cũng có thể chứa các fragment con, để sử dụng fragment con, sử dụng childFragmentManager và nó chỉ có tác dụng bên trong fragment chứa nó
* Fragment operations:
  + remove: loại bỏ fragment, view cũng bị xóa
  + add: thêm fragment
  + replace: thay thế fragment này bằng fragment khác
  + hide: ẩn fragment (view bị ẩn)
  + show: hiển thị fragment bị ẩn
* Fragment backstack:
  + LIFO
  + FragmentBackStack được tùy chỉnh bởi code, có thể để null hoặc khai báo tên rõ ràng cho việc return 1 fragment nhất định (fragmentTransaction.addToBackStack(String))
  + popBackStack (cơ chế LIFO)
  + popBackStack(name, flags) + flags: POP\_BACK\_STACK\_INCLUSIVE thì pop tất cả các fragment đứng trên fragment được gọi

**Service**

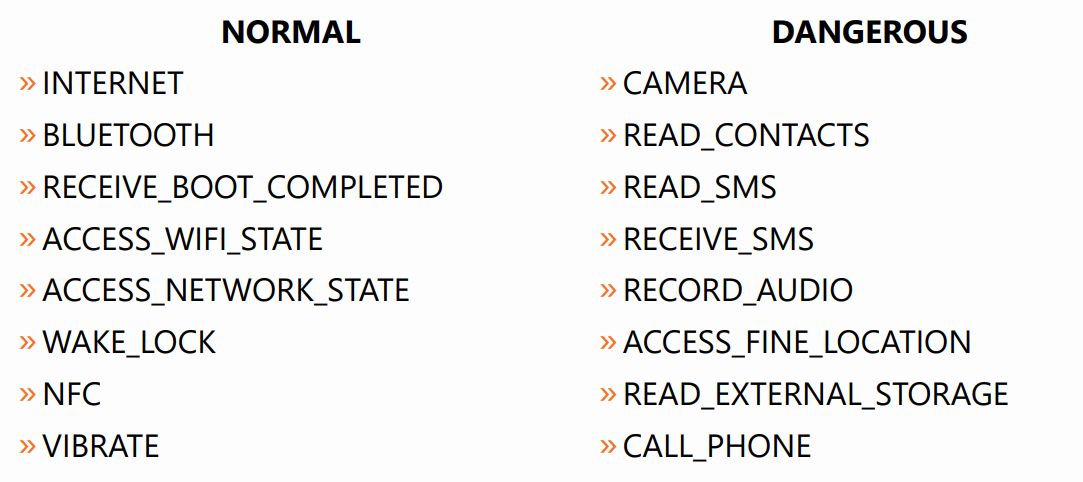
* Service thường được dùng để chạy tác vụ ngầm (không có UI), thường chạy trên cùng 1 process cùng với app
* Có 2 loại service chính tương ứng với 2 cách chạy service:
  + startService (UnBoundService):
    - Cần có application component khởi tạo (ví dụ như activity), thường dùng để chạy tác vụ dài, chạy cho tới khi bị hủy hoặc component bị hủy
  + bindService (BoundService):
    - Chạy cho tới khi những component gọi bindService đều unbind
    - Cung cấp giao diện client-server, cho phép các thành phần tương tác với nó (gửi yêu cầu, nhận kết quả), có thể có nhiều component ràng buộc với nó cùng lúc
  + Ngoài ra còn có foreground service (ví dụ như chơi nhạc, nó bắt buộc phải hiện thị 1 notification, nó chạy cả khi ứng dụng đóng)
* Lifecycle:



* Độ ưu tiên service:
  + Hệ thống bắt buộc phải dừng service khi k đủ bộ nhớ và phải khôi phục tài nguyên hệ thống cho activity.
  + Xét độ khó bị kill: Bound > Foreground > Background
* Một số giá trị trả về trong onStartCommand():
  + START\_NOT\_STICKY: Nếu hệ thống kill service, nó sẽ không được khởi động lại trừ khi có 1 intent đang chờ ở onStartCommand() \*required
  + START\_STICKY: Nó sẽ được khởi động lại, nếu không có intent chờ thì khởi động lại với inent null
  + START\_REDELEVER\_INTENT: Khởi động lại với intent là intent cuối cùng mà nó nhận được (thích hợp cho việc sử dụng khi download…)
  + START\_STICKY\_COMPATIBILITY: Tương tự sticky nhưng không chắc chắn khởi động lại
  + DEFAULT: là auto-options giữa sticky và compatibility
* Có 2 cách để khai báo 1 UnBoundService:
  + IntentService:
    - Thường được dùng khi thực hiện tác vụ ngắn, sau khi thực hiện thì dừng lại
    - Là 1 class con của Service, chạy ở background thread, chạy tuần tự
    - Không giao tiếp được với UI (Background)
    - onHandleIntent()
  + Service:
    - Thường sử dụng cho các tác vụ lâu dài, dừng thủ công
    - onStartCommand()
* Có 3 cách để khai báo 1 BoundService:
  + Binder: Extend class Binder và return 1 instance của nó trong hàm onBind(). Client nhận được Binder và truy cập trực tiếp tới các phương thức public trong Binder hoặc Service. Chỉ nên sử dụng với service private
  + Using a Messeger: Sử dụng cho việc tạo giao diện hoạt động trên nhiều process khác nhau bằng cách tạo giao diện cho service, sử dụng message. Service định nghĩ 1 handler để phản hồi các message khác nhau. (Gửi và nhận). IPC (Truyền thông liên tiến trình). Đây là cách đơn giản nhất để thực hiện truyên thông liên tiến trình (IPC), bởi vì Messenger xếp hàng tất cả các requests vào một luồng đơn
  + AIDL: (Ngôn ngữ định nghĩa giao diện Android - AIDL) phân tách các đối tượng thành các nguyên thủy mà hệ điều hành có thể hiểu và điều chỉnh chúng qua các quy trình để thực hiện IPC

**Permission**

* Là quyền/khả năng mà android cấp cho ứng dụng để truy cập vào các APIs giới giạn như Camera, Bluetooth, GPS…
* Mỗi ứng dụng đều có ID riêng, Android là 1 nền tả chạy đa nhân, cần truy cập tài nguyên hệ thống, để tránh xung đột -> Permission được sử dụng để config thủ công
* 3 Mục tiêu của permission:
  + Control: Người dùng có thể kiểm soát tất cả thông tin họ cần cung cấp
  + Transparency: Người dùng hiểu được ứng dụng cần dữ liệu gì và tại sao cần
  + Data minimization: Ứng dụng chỉ truy cập và sử dụng dữ liệu cần thiết cho nhiệm vụ hoặc hành động mà người dùng gọi
* Thực thi permission:
  + Gọi vào hệ thống
  + Bắt đầu 1 activity
  + Gửi và nhận broadcast
  + Truy cập và thao tác với content provider
  + Ràng buộc hoặc bắt đầu service
* 4 Levels:
  + Normal: Cấp quyền tự động, k cần hỏi người dùng
  + Dangerous: Cần sự chấp thuận của người dùng trước khi thực hiện
  + Signature: Được cấp khi và chỉ khi yêu cầu ứng dụng được ký bằng chữ ký đã khai báo
  + Special (SignatureOrSystem): Các quyền đặc biệt tương ứng với ứng dụng cụ thể. Chỉ nền tảng OEMs mới có thể xác định các quyền đặc biệt



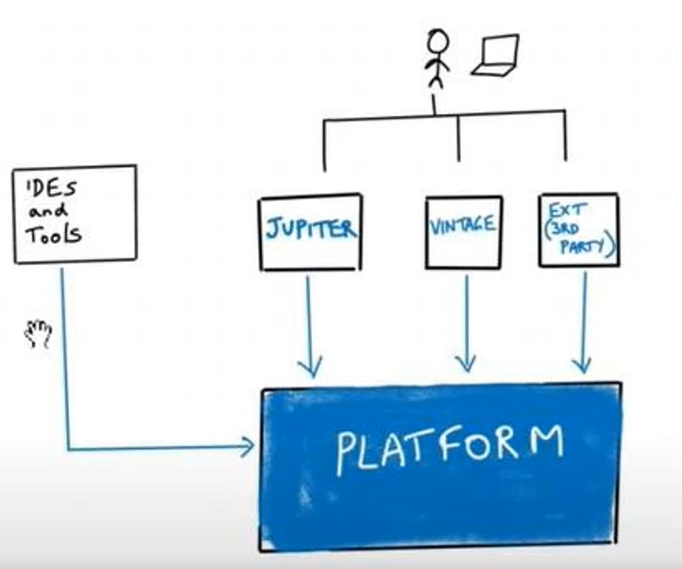
* 2 loại quyền cấp ứng dụng: (Quyền cần thiết để ứng dụng hoạt động)
  + System-defined: Permission do hệ thống định nghĩa
  + Application-defined: Permission do coder định nghĩa để bảo vệ ứng dụng
* Android sử dụng các quyền không chỉ để bảo vệ các ứng dụng mà còn để bảo vệ các thành phần riêng lẻ có bên trong một ứng dụng (Service, Broadcast, Content…)
* Nhóm permission: tập hợp các permission tương tự, được trình bày cùng nhau trong thời gian cài đặt để phê duyệt
* Cây permisson: sắp xếp 1 tập hợp các quyền để có thể yêu cầu tất cả các quyền trong cây

**Notification**

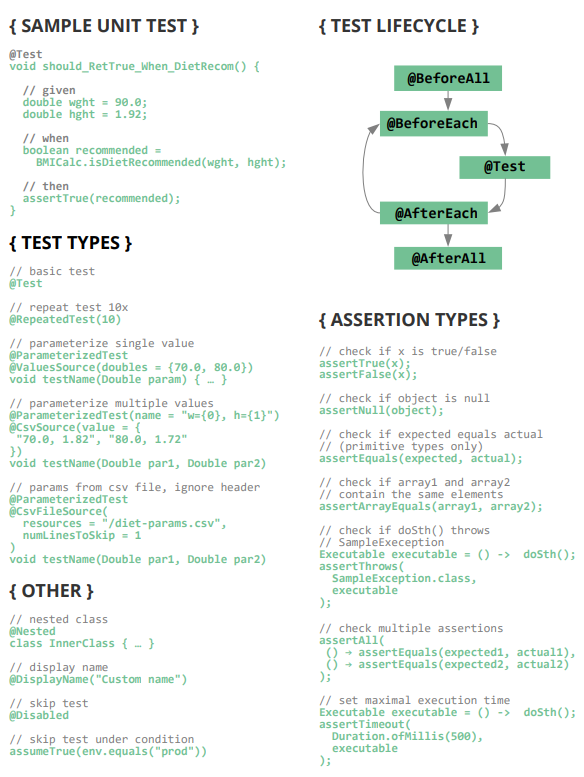
* Thông báo là 1 message mà android hiển thị bên ngoài giao diện ứng dụng, nó có thể đem theo lời nhắc, nút hành động, có thể thông qua thông báo để mở ứng dụng hoặc thao tác trực tiếp với ứng dụng
* Từ API 26 trở đi (Android 8.0) thì tất cả notifications cần phải được gán cho channel
* Sử dụng NotificationCompat.Buider để config, có các thuộc tính như setSmallIcon, ContentTittle, ContentText, Style.BigStyle, AddAction…
* Sử dụng NotificationManager để config channel, 1 channel có thể chứa nhiều thông báo cùng kiểu, có thể thông qua thuộc tính importance để cài đặt mức độ quan trọng của thông báo

**Unit Test**

* Unit test (Kiểm thử đơn vị) là quá trình kiểm thử được thực thi trong quá trình phát triển ứng dụng. Mục tiêu của unit test là cô lập các đơn vị và xác minh tính chính xác của phần đơn vị đó.
* Ưu điểm:
  + Phát triển nhanh hơn (Hạn chế bug và thay đổi tư duy code)
  + Cấu trúc code tốt hơn (Bắt buộc phân bổ để thuận tiện cho test)
* Nhược điểm:
  + Tốn time (Đã code lại còn phải test :)))
  + Không có gì đảm bảo có thể test hết 100% các trường hợp, và đôi khi không phải ai có thể code cũng có thể viết test
* JUnit:
  + JUnit là 1 open source framework đơn giản dùng cho việc tạo các unit testing tự động
  + JUnit là 1 khung test mã nguồn mở, được sử dụng để viết và chạy test
  + Cung cấp annotation để định dạng các test method
  + Cung cấp assertion để kiểm thử các kết quả mong đợi
  + Cung cấp các trình chạy để chạy test…
* JUnit5:
  + JUnit 5 có mục tiêu là áp dụng phong cách lập trình của java 8
  + JUnit 5 mạnh mẽ và linh hoạt hơn JUnit 4
  + JUnit5 # JUnit4+1
  + JUnit 4 đã có hơn 10 năm, không update mới, không sử dụng các testing patterns mới, không update với các features mới của ngôn ngữ…



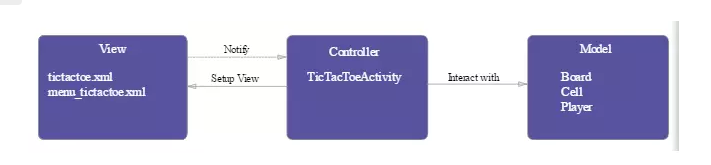
* Platform là 1 khái niệm mới dùng để viết và chạy junit code.
* Jupiter là 1 tên gọi mới của Junit (Hành tinh jupiter - thứ 5)
* 2 kiểu test:
  + UI Testing:
    - Espresso
    - UIAutomator
    - Appium
  + Local unit test:
    - Junit
    - Mockito
    - MockK
    - Robolectric
* C0/C1/C2 coverage:
  + Test coverage được hiểu same same như độ phủ của test (lượng kiểm tra đã thực hiện / tổng trường hợp kiểm tra cần thiết)
    - C0: Tỉ lệ bao phủ câu lệnh (Ta sẽ quan tâm đến các xử lý, và tạo ra test case bằng các điều kiện sao cho tất cả các xử lý đều được thực hiện trên 1 lần)
    - C1: Tỉ lệ bao phủ nhánh (Ta chú ý đến điều kiện, và tạo ra test case để bao phủ các kết quả đánh giá như True/False liên quan đến tất cả các điều kiện)
    - C2: Tỉ lệ bao phủ điều kiện (Việc tập trung vào điều kiện trong nhánh điều kiện, bao phủ test toàn bộ các kết quả điều kiện)

****

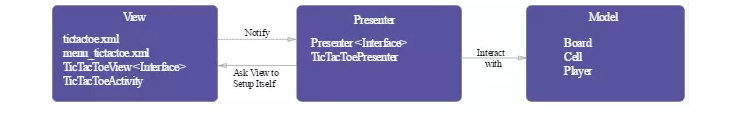
* Mock:
  + Là kỹ thuật mà coder dùng để tạo ra các đối tượng giả
* Stub:
  + Là 1 chương trình hoặc thành phần giả lập thay thế cho phần chưa code xong, stub được dùng cho việc kiểm thử (ví dụ cần test 4 module mà 1 module chưa xong, chương trình giả lập module này để test gọi là stub)
* Mockito:
  + Mockito được xây dựng cho java
  + Là 1 framework mock cho phép bạn viết các test đẹp với API sạch sẽ, đơn giản.
  + Mockito giả lập dữ liệu, tạo ra các instance độc lập mà mình không cần phải tạo class
  + Kiểm tra được cả các phương thức không có giá trị trả về
* MockK:
  + MockK được xây dựng cho kotlin
  + Giống Mockito, cho phép tạo và stub các đối tượng bên trong test code
  + Mock có thể mock được hầu hết các thành phần trong code (Object, Extensions, Constructor, Private function, module, suspen function, capturing, chainned…)

**Design Architecture**

* MVC:
  + (Model - View - Controller)
* Model: Data + State + Business Logic
* View: Giao tiếp với Controller và render cho UI
* Controller: Điều khiển tổng thể, View giao tiếp với Model qua Controller
* MVC khó Unit test vì controller gắn rất chặt với các API và View. Nếu View thay đổi, Controller cũng phải thay đổi theo. Ngoài ra C càng ngày càng lớn theo thời gian



* MVP:
  + (Model - View - Presenter) Phá vỡ Controller thành View + Activity
* Model: Tương tự MVC
* View: Khác với Controller ở MVC thì View ở đây có cả định hướng các event tới Presenter để Presenter xử lý
* Presenter: Hành động theo model và view, lấy dữ liệu từ model, xử lý theo view yêu cầu và hiển thị

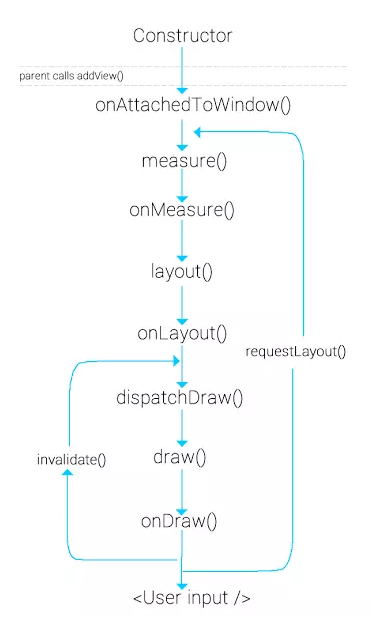


* MVVM:
  + (Model - View - ViewModel)
* Model: Tương tự MVC
* View: View bind tới các biến observable và hành động tiếp xúc bởi ViewModel
* ViewModel: Chịu trách nhiệm cho model, chuẩn bị dữ liệu observable cần thiết cho View, cung cấp móc cho view để pass các event tới model. Tuy nhiên ViewModel lại k gắn với View



**Custom View**

* View Life Cycle:

****

* Có thể khai báo thêm thuộc tính = xml (attrs) và = code (binding adapter)
* Có thể custom giữa view có sẵn và tự tạo hoặc tự tạo full
* Bắt buộc phải implments 2 phương thức: onMeansure, onDraw

**Animation**

* Property Animations: được sử dụng để thay đổi các thuộc tính của đối tượng ví dụ như translateX, textScaleX
* View Animations: được sử dụng để làm các animation như xoay, thay đổi kích thước, trong suốt…
* Drawable Animations: tạo các animation từ drawable resource

**Drawable**

* Bitmap Drawable
* Shape Drawable
* State List Drawable/ Color State List
* Vector Drawable
* Animation-list Drawable (gif)
* Level-list Drawable (sạc pin)
* Layer-list Drawable (chồng drawable)
* Nine-patch Drawable (kéo giãn bitmap)
* Ripple Drawable (hiệu ứng click chuột)
* Transistion Drawable

**Advanced Android Component - Dialog - Setting Screen - ViewPager2**

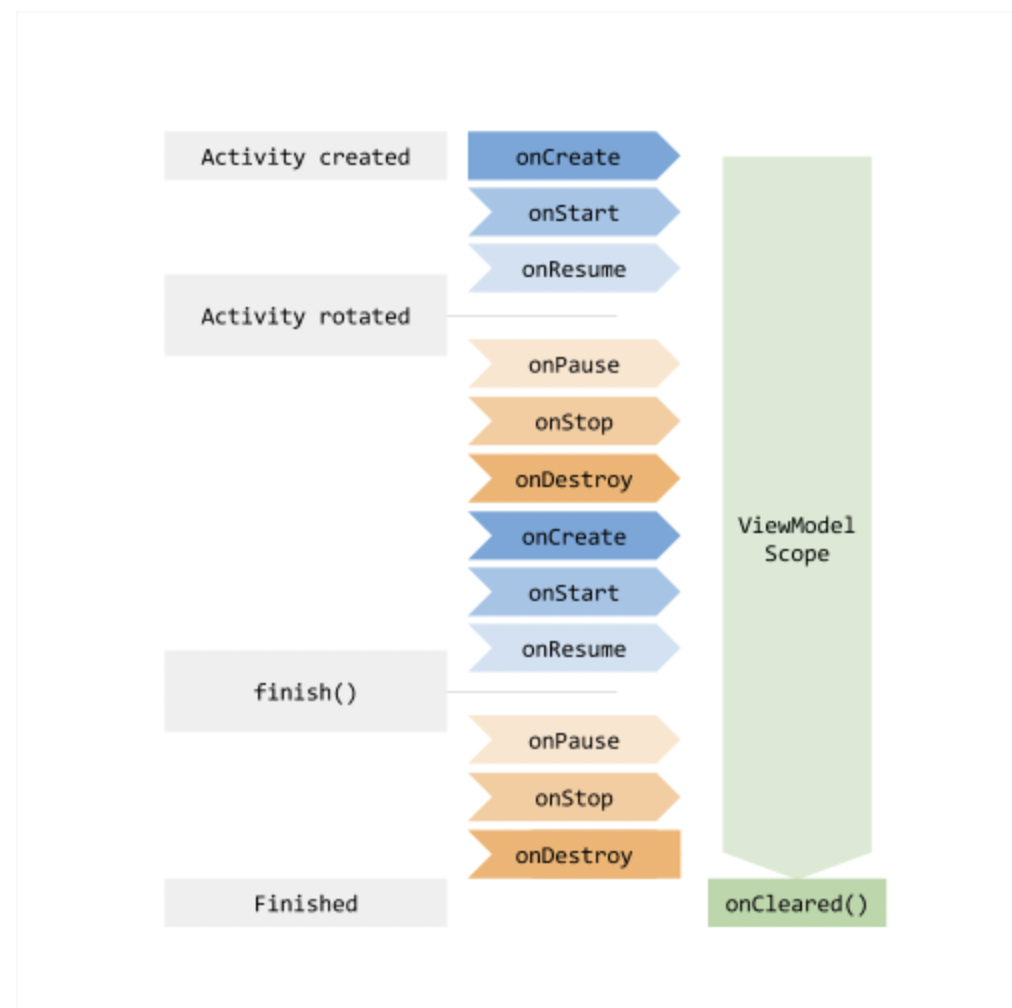
* Dialog:
  + Dialog là 1 dạng cửa sổ nhỏ hiển thị đè lên trên view
  + Khi xuất hiện dialog, view sẽ đi vào trạng thái onPause()
  + Một số loại Dialog: AlertDialog, DatePickerDialog, Dialog Custom Layout và Dialog Fragment
* Setting Screen:
  + Sử dụng setting preference và shared preference để lưu trữ các trạng thái setting
* ViewPager2:
  + ViewPager là 1 widget của android giúp chuyển giao diện người dùng mà không cần thay đổi activity hay fragment. (Nó thực hiện bằng cách di chuyển qua lại giữa chúng mà không làm chúng bị hủy hay dừng lại)
  + ViewPager2 có tính năng nổi trội nhất so với ViewPager đó là RecyclerView.
  + ViewPager2 có thể sử dụng RecyclerView để hiển thị layout content.
  + Có thể vertical

**Advanced Android Component - RecyclerView - Navigation - Toolbar**

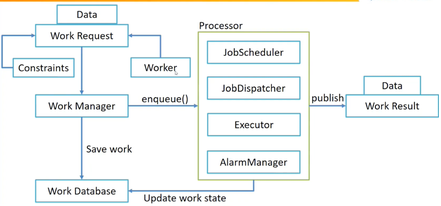
* RecyclerView:
  + Nó là 1 ViewGroup dùng để hiển thị các View tương đồng.
  + Nó được coi là sự kế thừa của ListView và GridView
  + Lí do mà RecyclerView được ưa chuộc là do nó có thể lắng nghe và thay đổi dữ liệu của từng view theo yêu cầu của người dùng
  + Một số thành phần:
* LayoutManager: đây là phần cần có để có thể khởi tạo RecyclerView, nó sẽ bố cục cách hiển thị các View thành phần
* Adapter: Xác định số item qua getCount, bố cục xml của từng item qua ViewHolder và tùy chỉnh dữ liệu qua BindViewHolder
* ItemAnimator: Đây là 1 công cụ giúp tạo hoạt ảnh cho các item của RecyclerView khi các item đó thay đổi
* Navigation:
  + Navigation là 1 menu kéo hiển thị như 1 ngăn xếp ở cạnh biên bên trái màn hình.
  + Muốn sử dụng thì phải khai báo Drawer và import thư viện, sau đó tạo menu item và add vào NavigationView
* Toolbar:
  + Toolbar chính là thanh công cụ nằm ở phía trên cùng của màn hình hiển thị, nó có thể được custom với các phím chức năng, navigation drawer, menu options….
  + Ngược với toolbar là bottom-navigation, là dạng thanh công cụ nằm ở dưới cùng màn hình, nó thường được dùng với viewpager (ngoài tablayout)

**Jetpack Architecture Component - ViewModel - DataBinding - WorkManager**

* ViewModel:
  + ViewModel là 1 class chịu trách nhiệm quản lý và chuẩn bị dữ liệu cho UI (Activity/Fragment)
  + ViewModel còn cung cấp cách dễ dàng để giao tiếp giữa các activity với nhau, các activity với fragment hoặc các fragment với nhau thông qua “phạm vi"
  + ViewModel sẽ không bị destroyed khi activity hoặc fragment của nó bị destroyed bởi 1 configuration changed (ví dụ như xoay màn hình). Instance mới sẽ được kết nối lại với ViewModel cũ
  + Lợi ích:
* ViewModel có vòng đời tồn tại ngay cả khi UI đã bị configuration changed
* Running ngay cả khi activity ở trong backstack
* Kết hợp cùng livedata có thể lắng nghe phản ứng của UI
* Dễ hiểu và dễ testing



* DataBinding:
  + DataBinding là 1 thư viện giúp hỗ trợ cho việc ánh xạ view và xử lý event một cách nhanh chóng và ngắn gọn hơn
  + DataBinding có thể kết hợp cùng ViewModel và LiveData trong mô hình MVVM giúp triển khai code rành mạch, logic và dễ dàng hơn
  + Ngoài ra DataBinding còn cung cấp 1 số phương thức và anotation hỗ trợ custom attrs của view
* WorkManager:
  + WorkManager là 1 thư viện được thiết kế cho việc lập lịch và quản lý các tác vụ ngầm
  + Chúng ta đã có Job Scheduler API, Firebase Job Dispatcher… cũng sử dụng để lập lịch
  + Trong WorkManager còn định nghĩa 1 thành phần là Worker, nó là 1 abstract class tương đương với tác vụ hoặc công việc cần được thực hiện ở background
  + 4 lí do chính để thư viện này đc lựa chọn:
* Dễ lập lịch: ChXo phép ứng dụng lập lịch cho task chạy với điều kiện nhất định, tác vụ này còn có thể liên kết tác vụ khác để chạy song song
* Dễ hủy bỏ
* Dễ truy vấn: Cho phép hiển thị việc quản lí tiến trình lên UI
* Hỗ trợ mạnh mẽ tất cả các version của android



**Jetpack Architecture Component - Navigation Component - Repository Pattern**

* Navigation Component:
  + Nó là 1 phần của Android Jetpack và gói AndroidX nhằm mục đích đơn giản hóa điều hướng UI
  + Cài đặt Navigation Component bằng cách:
* Import thư viện hỗ trợ
* Tạo layout, code và tạo Navigation Graph
* Add actions, arguments
* Triển khai
* Một số lợi ích:
* Xử lý thay cho các fragment transactions
* Xử lý chính xác các hành động trên nút back của màn hình và hệ thống
* Hỗ trợ và cung cấp các tài nguyên tiêu chuẩn hóa cho animation
* Xử lý liên kết sâu
* Hỗ trợ Navigation Drawer, Option Menu, Bottom Navigation…
* Hỗ trợ ViewModel hoàn chỉnh
* Repository Pattern:
  + Design Pattern là 1 kỹ thuật trong lập trình hướng đối tượng, nó như 1 bản mẫu thiết kế, các giải pháp cho các vấn đề chung trong lập trình, đảm bảo cung cấp giải pháp tối ưu (Nghe hơi giống giải thuật)
  + Design Pattern chia làm 3 nhóm chính:
    - Creational: Sử dụng trong việc khởi tạo đối tượng, giúp cung cấp các thủ thuật liên quan tới khởi tạo đối tượng
    - Structural: Thường sử dụng để giải quyết các mối quan hệ liên quan tới entities (thực thể)
    - Behavioral: Sử dụng trong việc thể hiện các hành vi của các đối tượng, tương tác giữa đối tượng và đối tượng
  + Repository Pattern đóng vai trò trung gian giữa tầng Business Logic và Data Access giúp truy cập dữ liệu an toàn và chặt chẽ hơn
  + Repository Pattern:
* Là nơi duy nhất thay đổi quyền dữ liệu
* Là nơi duy nhất chịu trách nhiệm cho việc mapping dữ liệu vào pj
* Tăng tính bảo mật và dễ dàng code
* Dễ dàng thay thế bằng 1 implementation cho việc testing

**Saving Data - Persistent**

* Internal Storage: (Bộ nhớ trong)
  + Lưu các file riêng tư của ứng dụng (app-private files) vào trong hệ thống file
* External Storage: (SD Card/…)
  + Lưu trữ file trên hệ thống file chia sẻ được ra bên ngoài (hình ảnh/video)
* Shared Preferences:
  + Lưu trữ dữ liệu nguyên thủy bởi các cặp key-value
* Database:
  + Lưu trữ dữ liệu các dữ liệu có cấu trúc trong cơ sở dữ liệu private
* Room Database: Là 1 phần trong Android Architecture Components, nó giúp giảm thiểu số lượng câu truy vấn bằng các anotation có sẵn và xác định truy vấn tại compile time. Thành phần của nó bao gồm: Sqlite database, DAO, Entity, Room database (Đây là abstract layer nằm ngoài cùng có vai trò quản lý và truy cập tới tầng dưới cùng của Sqlite, nó sử dụng DAO để thực hiện các query tới db)
* SQLiteHelper: Sử dụng các câu lệnh query trực tiếp tới db, chính xác và tin cậy nhưng khó triển khi vì đôi khi sẽ xảy ra lỗi với các câu truy vấn
* Network:
  + Firebase

**Content Provider**

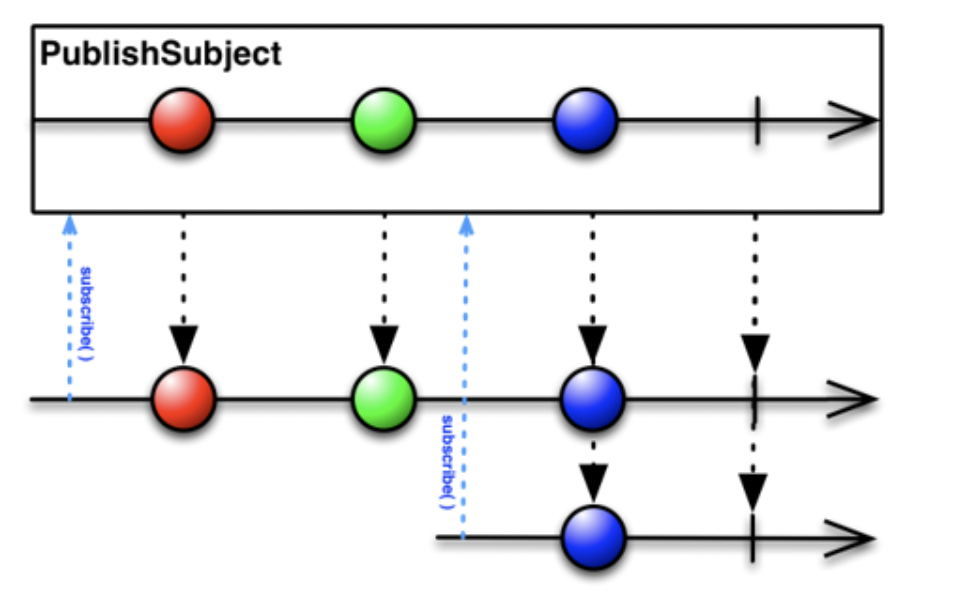
* Là thành phần để quản lý truy cập dữ liệu, nó cung cấp các phương thức hỗ trợ để ứng dụng có thể truy cập dữ liệu từ ứng dụng khác thông qua ContenResolver
* Content Provider có thể giúp cho 1 ứng dụng quản lý quyền truy cập đến dữ liệu được lưu bởi chính nó hoặc ứng dụng khác
* Content Provider:
* Cho phép chia sẻ dữ liệu từ ứng dụng này tới các ứng dụng khác
* Gửi dữ liệu sang widget
* Trả về 1 kết quả gợi ý khi search thông qua Framework sử dụng SearchRecentSuggestionsProvider
* Đồng bộ dữ liệu với server thông qua AbstractThreadedSyncAdapter
* Tải dữ liệu lên UI sử dụng CursorLoader (Con trỏ như trong SQLiteHelper)
* Content Provider khá giống 1 cơ sở dữ liệu, nó hỗ trợ các phương thức insert, update, delete… và query
* Content URI là 1 URI định danh dữ liệu trong 1 provider, nó bao gồm kí hiệu tên của provider (Authority) và tên chỉ tới bảng: content://authority/path/id

**IPC**

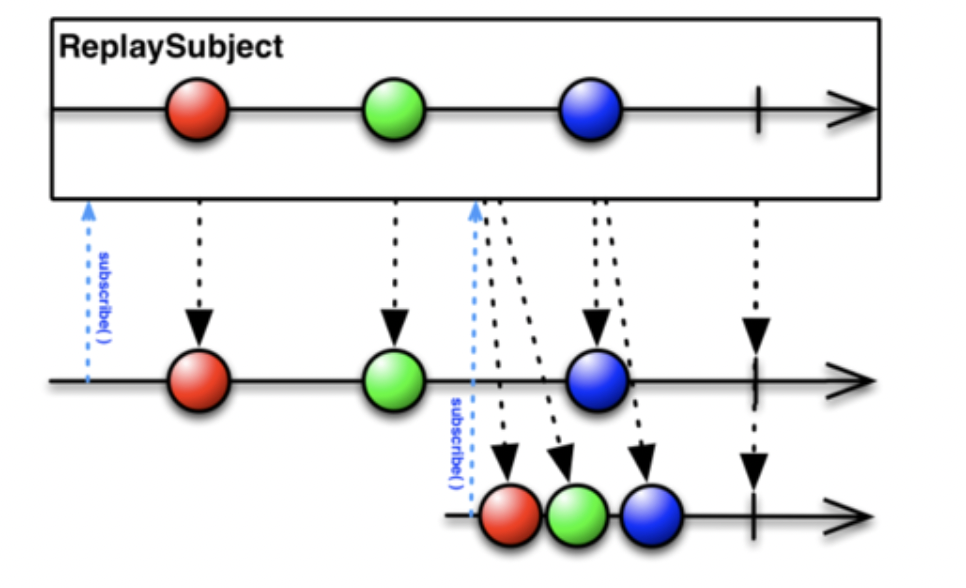
* **IPC** (Inter - Process Communication) là 1 cơ chế cho phép các các quá trình giao tiếp với nhau và đồng bộ hóa các hành động của chúng
* Trong IPC thường phân chia thành client và server, trong đó client yêu cầu dữ liệu và máy chủ đáp ứng yêu cầu của máy khách
* 2 mô hình IPC:
* Share Memory: Giao tiếp giữa các tiền trình sử dụng bộ nhớ dùng chung
* Messing Passing: Giao tiếp với nhau mà không dùng chung bộ nhớ, sử dụng send và receive nguyên thủy
* 2 cách triển khai: Messenger và AIDL
* Messenger
  + Thực hiện giao tiếp bằng Messenger Object và cho phép dịch vụ xử lý cuộc gọi tại 1 thời điểm (pipe)
  + Các buớc triển khai:
* Service implements 1 handler nhận 1 cuộc gọi lại cho mỗi 1 cuộc gọi từ client
* Service sử dụng handler để tạo 1 đối tượng Messenger (tham chiếu đến handler)
* Messegner tạo IBinder mà Service trả về cho khách hàng qua onBind()
* Client sử dụng IBinder để khởi tạo Messeger (tham chiếu tới handler của service) mà client sử dụng để gửi các message tới service
* Service từng message trong handler của nó (trong hanleMessage())
* AIDL (Android Interface Definition Language)
  + Tạo tệp .aidl xác định IPC để cho phép client truy cập service và sử lý đa luồng tới service (nhiều client đồng thời truy cập)
  + Content Provider # AIDL: Content Provider chỉ chia sẻ dữ liệu ứng dụng mà không chia sẻ phương thức xử lý logic
  + Chỉ nên sử dụng AIDL khi muốn các ứng dụng client có thể kết nối tới service của sever thông qua truyền thông liên tiến trình (IPC), còn nếu đơn giản chỉ muốn sử dụng service thì nên sử dụng Binder
  + Các bước triển khai:
* Đầu tiên bạn cần tạo file AIDL và định nghĩa các interfaces
* Rebuild và implements các interfaces trong Service
* Expose interface trong clients
* Sử dụng method của server

**Explore Reactive Programming**

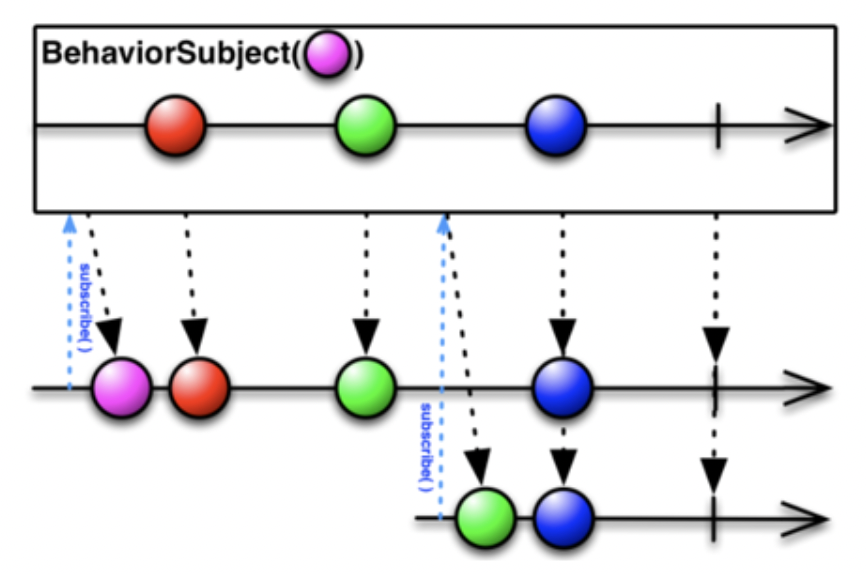
* **Reactive Programming** về cơ bản là dựa trên sự kiện lập trình bất đồng bộ. Mọi thứ được thể hiện ra là 1 luồng dữ liệu bất đồng bộ (Asynchronous Data Stream) và 1 hành động sẽ được thực hiện khi nó phát ra các giá trị
* Lợi ích của RP là việc xử lý multi thread giúp đẩy nhanh công việc
* Rx
  + Reactive Extension (ReactiveX hay Rx) là 1 thư viện follow theo quy tắc của RP, nó soạn ra chương trình bất đồng bộ và dựa trên sự kiện bằng cách sử dụng các chuỗi quan sát được. Nó cung cấp các interfaces và methods giúp code đơn giản và sáng sủa
  + Nó cũng là sự kết hợp giữa Observer pattern, Iterator pattern và Funtional programming
* Rx Java
  + Là Rx cho Java, nó có thể tạo ra bất kì luồng dữ liệu không đồng bộ trên bất kì thread nào, chuyển đổi dữ liệu và sử dụng dữ liệu trên bất kì thread nào thông qua Observer.
* RxAndroid
  + Là Rx dành cho Android, nó được hình thành từ Rx java + thêm các lớp mới. Cụ thể hơn là với Schedulers (AndroidSchedulers.mainThread()), nó đóng vai trò quan trọng trong việc hỗ trợ đa luồng trong ứng dụng
  + Một số Schedulers cơ bản (tương tự như dispatcher trong coroutines):
* .mainThread() cung cấp quyền truy cập tới UI, liên quan tới việc xử lý sự kiện và tương tác người dùng (có xảy ra ARN)
* .IO() liên quan tới đọc ghi file, đĩa, db…
* .computation() đòi hỏi nhiều CPU xử lý dữ liệu lớn như bitmap… (Số lượng CPU xử lý phụ thuộc vào số lõi CPU có sẵn của máy)
* .single() xử lý tuần tự nhiệm vụ
* …
* 2 components chính của Rx java:
* Observable: là 1 luồng dữ liệu (data stream) làm công việc nào đó và trả về dữ liệu (data)
* Observer: là những đối tượng lắng nghe observable, nhận dữ liệu từ observable phát ra
* Emitter là 1 instance của Observable: cũng là đối tượng và có thể quan sát được, nó dùng để thay đổi dữ liệu trả về tùy chỉnh
* Subject 4 loại:
* Publish Subject: Lắng nghe từ thời điểm được gọi



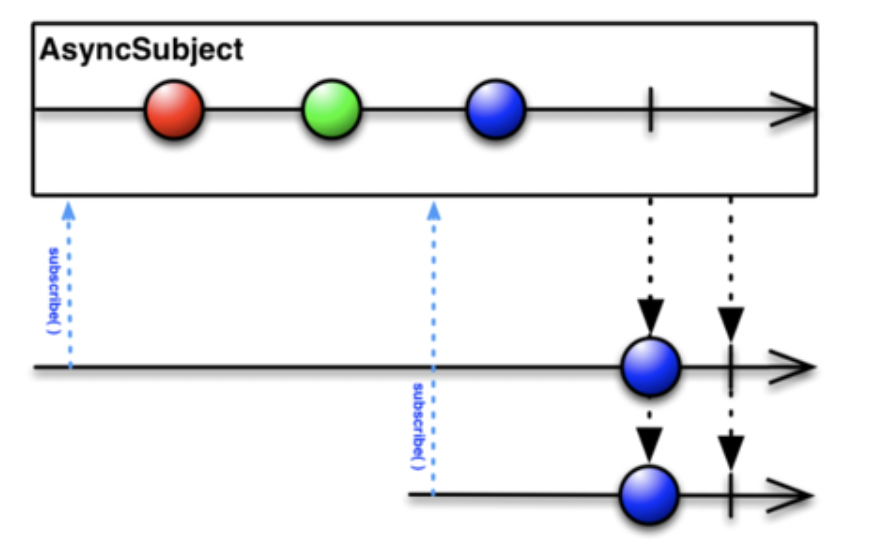
* Replay Subject: Lắng nghe lại từ đầu



* Behavior Subject: Nghe cái vừa được trình bày và tiếp theo



* Async Subject: Nghe cái cuối cùng



* Một số operators:
* Just: Phát ra các giá trị tương tự được cung cấp trong các đối số
* Filtering:
  + Filter: Chỉnh sửa sự kiện cụ thể chỉ được phát ra từ observable
  + Skip: Loại bỏ n phần tử đầu tiên
  + Take: Lấy n phần tử đầu tiên
* Combining:
  + Merge: Hợp nhất nhiều observable thành 1 bằng cách kết hợp các phần tử được phát ra, có thể xảy ra hiện tương trộn
  + Zip: Kết hợp phát xạ của nhiều operator với nhau thông qua 1 chức năng được chỉ định và phát ra các mục đơn lẻ cho mỗi tổ hợp dựa trên kết quả của chức năng
* Debounce: Chỉ phát ra tín hiệu sau 1 quãng time nhất định, trong time đó k phát ra tín hiệu nào
* Distinct: Loại bỏ các giá trị trùng lặp đc bắn ra từ Observable
* Transforming (Chuyển đổi kiểu dữ liệu)
* Map: Map chuyển đổi 1 item thành 1 item khác thông qua hàm
* FlatMap: Chuyển đổi các item phát ra bởi 1 observable thành 1 observable khác
* Count: Đếm số lượng item đc trả về bởi observable và trả ra giá trị
* Reduce: Áp dụng func cho từng item
* Error Handling: Observable k ném ra exception mà nó thông báo cho các observables có lỗi xảy ra bằng cách bắn callback onError() sau đó thoát khỏi luồng (onErrorComplete, onErrorResumeNext…)
* Coroutine Flow:
  + Tương tự như Rx Java, nó được phát triển nhờ vào coroutines của Kotlin, có các operator tương tự như observable

**Debugging**

* **Breakpoint**
* **LogTag (d: debug, i: info, w: warning, e: error…)**

**Advanced Build System**

* Gradle App
* Gradle Project
* Settings.gradle

**Dependency Injection**

* **Dependency Injection** là 1 kỹ thuật nhằm giảm sự phụ thuộc giữa các class, thay vì để class tự sinh ra thì kỹ thuật này sẽ định nghĩa chúng
* Hilt:
  + là 1 thư viện với định nghĩa 1 cách tiêu chuẩn để sử dụng DI trong ứng dụng
  + cung cấp container cho mỗi android class trong pj và quản lý vòng đời của nó 1 cách tự động
  + Hilt sử dụng thư viện phổ DI phổ biến nhất đó là Dagger, nhược điểm của dagger là build tốn time do sinh ra code ở compile time.
  + Dagger cung cấp 1 số tiện ích như Compile Time Correctness, Runtime Performance…
* Dagger:
  + Gồm 4 phần chính:
* Module: 1 nhà cung cấp sự phụ thuộc và class như 1 người dùng
* Component: Cầu nối để cung cấp sự phụ thuộc từ nhà cung cấp tới ng dùng
* Provides: Là 1 annotation đc sử dụng trong class Module để cung cấp sự phụ thuộc
* Inject: Là 1 annotation đc sử dụng để xác định sự phụ thuộc bên trong người dùng
* Hilt-Dagger:
* Hilt và Dagger có thể cùng tồn tại trong 1 codebase nhưng tốt nhất là nên sử dụng hilt để quản lý tất cả việc sử dụng dagger của ứng dụng