### **POJO is meaning?**

* POJO - Plain Old Java Object - là 1 đối tượng java thuần (đơn giản, chỉ có java) không bị ràng buộc bởi bất kỳ mô hình, quy ước nào, chỉ có các thuộc tính và phương thức.

### Difference between Serializable and Parcelable

- Serializable là 1 khái niệm giúp chúng ta có thể chuyển đổi trạng thái của 1 đối tượng java thành 1 định dạng nào đó để object có thể được lưu trữ ở đâu đó và sau đó được sử dụng bởi 1 tiến trình khác.

|  |  |  |
| --- | --- | --- |
|  | Serializable | Parcelable |
|  | Là 1 interface chuẩn của java, không phải là 1 phần của Android SDK. | Là 1 interface thuộc Android SDK |
| Implement | Đơn giản | Phức tạp |
| Performance | Chậm  Không phụ thuộc nền tảng. | Nhanh  Phụ thuộc nền tảng. |
| Detail | is **marker** interface. Class chỉ đơn giản là implement nó để cho Java biết được class có thể serialize. Reflection được sử dụng trong quá trình xử lý, có thể tạo ra nhiều objects tạm thời -> performance chậm | Không cần reflection trong quá trình xử lý, bởi vì việc serialization được thực hiện 1 cách tường minh |
|  |  | Parcelable array can be pass via Intent in Android |

Reflection là 1 thuật ngữ mô tả việc kiểm tra các object, field, method tại thời điểm run-time. Ex: Object.class.getClass()

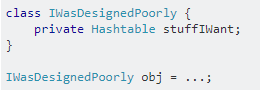
### StringBuffer and StringBuilder

|  |  |
| --- | --- |
| StringBuffer | StringBuilder |
| StringBuffer là synchronized - đồng bộ  - thread safe - an toàn luồng  - Tức là 2 hoặc nhiều thread **không thể** gọi 1 phương thức của StringBuffer **cùng 1 lúc**.  - Do đó mà tốc độ xử lý của StringBuffer **chậm hơn** StringBuilder. | StringBuilder là no-synchronized - bất đồng bộ  - not thread safe - không an toàn luồng  - Tức là 2 hoặc nhiều thread **có thể** gọi 1 phương thức của StringBuilder **cùng 1 lúc**  - Do đó mà tốc độ xử lý của StringBuilder **nhanh hơn** StringBuffer. |

**=>** Tóm lại, ở môi trường đơn luồng, StringBuilder xử lý nhanh hơn StringBuffer

1. Cách truy cập vào một thuộc tính private của một class

* Để truy cập vào một thuộc tính private của một class ta sử dụng đối tượng Field:





### Concept Rx

- Reactive programming là lập trình các luồng dữ liệu không đồng bộ.

- 4 point asynchronous:

+ Explicit execution - thực thi rõ ràng: khi 1 công việc được start ở 1 thread mới, hãy đảm bảo rằng có thể kiểm soát được nó.

+ Easy thread manager: trong lập trình không đồng bộ, thread manager được xem như là key.

+ Easily composable: dễ dàng kết hợp các công việc với nhau.

+ Minimum the side effect: trong lập trình đa luồng, nên làm giảm tối thiểu ảnh hưởng của 1 thread tới các thread khác.

- Rx là 1 thư viện được sử dụng trong lập trình không đồng bộ và dựa trên sự kiện bằng cách sử dụng các observable sequence.

- Nó giải quyết tốt các vấn đề về lập trình đa luồng: low-level thread, synchronization(đồng bộ hóa), thread-safety(an toàn luồng), concurrent data structure(cấu trúc dữ liệu đồng thời), non-block I/O.

- Là sự kết hợp của Observer pattern, Iterator pattern và lập trình hướng chức năng.

- Observer pattern: được sử dụng khi có mối quan hệ 1 nhiều giữa các object, nếu state của 1 object thay đổi, thì các object phụ thuộc nó cũng tự động thay đổi.

-Iterator pattern: cung cấp 1 cách truy cập các phần tử của collection object 1 cách tuần tự mà không cần biết cấu trúc của nó.

- Rx = Observable + Observers(subscriber) + Scheduler

- Observable: observable are nothing. Emit the data

- Observer: observers consumes the data stream emited.

- Scheduler: thread manager

Tham khảo thêm Rx: <https://docs.google.com/document/d/1wR1-K2srzgvK-LwGHUQEpUBvR4-uiE5T9HyHmmxIWHQ/edit?usp=sharing>

* subscribeOn(io): xác định luồng mà các operator tính toán để tạo ra Observable. Chỉ được gọi một lần duy nhất, nó chỉ có tác dụng khi gọi hàm subscribe() được gọi đến.
* observeOn(ui): xác định luồng mà operator hoặc subscribe gọi sau nó.
* Scheduler là đối tượng để quản lý các luồng thực hiện.

### Retrofit và OkHttp

* <https://docs.google.com/document/d/1FcNwmgkWbVkIcuEa517NqaIQNjrqSBF_mRfYy75edi4/edit#heading=h.u8km97ax5fae>

### Context

- Context là trạng thái của ứng dụng tại một thời điểm nhất định.

- Context là 1 lớp cơ bản chứa hầu hết các thông tin về môi trường ứng dụng của android, tức là mọi thao tác, tương tác với hệ điều hành đều phải thông qua lớp này.

- Context là 1 abstract class, nó cung cấp cho các lớp triển khai các phương thức truy cập vào tài nguyên của ứng dụng và hệ thống. Ví dụ như nó có thể khởi tạo và chạy các activities, broadcast, các intents....

- Sở dĩ hầu hết các lớp liên quan đến UI (layout, button, textview...) đều phải super context vì bản thân chúng đảm nhận việc truy cập resource(R.id, R.layout...). Nếu chúng ta không tham chiếu đến Context thì sẽ không dùng được các resource mà chúng ta đã tạo ra.

- Hệ thống cấp bậc Context

Context

| — ContextWrapper

| — — Application

| — — ContextThemeWrapper

| — — — Activity

| — — Service

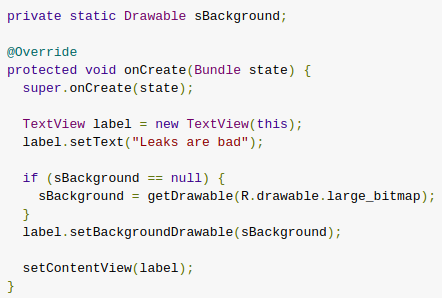
| — — — IntentService

| — MockContext

* Thông thường, bạn hay sử dụng 2 loại context là Application và Activity
* Application context: gắn liền với vòng đời của ứng dụng và luôn luôn giống nhau xuyên suốt vòng đời ứng dụng.
  + getApplication():
  + getApplicationContext():
* Activity context: gắn liền với vòng đời của Activity và nó sẽ bị hủy khi activity bị hủy.
  + getBaseContext():
  + Activity.this:
* Tips: Bất cứ khi nào bạn cần thao tác với Views hãy sử dụng Activity context, còn không sử dụng Application context là đủ.
* Ví dụ: khi bạn sử dụng Toast, bạn có thể sử dụng 1 trong 2 loại contex trên, nhưng vì nó có thể được show lên từ bất cứ nơi đâu trong ứng dụng, vì vậy bạn nên sử dụng Application context là hợp lý (Trong head-first android họ cũng sử dụng getApplicationContext());
* Tránh sử dụng getBaseContext() - lớp này được triển khai khi 1 class extends từ ContextWrapper. Mà lớp này lại có khoảng 40 lớp con trực tiếp và không trực tiếp. Vì vậy, nên gọi trực tiếp đến getContext, Activity, Fragment... để tránh gây ra memory leak.
* getApplicationContext() là 1 instance của class Application - được dùng để duy trì trạng thái global cho app.
* Khi nào sử dụng getApplicationContext():
* Khi nào không nên sử dụng getApplicationContext():

### Memory Leak context

* Rò rỉ bộ nhớ
* Memory leak xảy ra khi bộ nhớ được cấp phát nhưng không bao giờ được giải phóng.
* Trong khi làm việc với Android, đối tượng context rất hay được sử dụng, vì nó thường được sử dụng để load và truy cập vào resource.
* Leak: được hiểu là việc bạn giữ 1 tham chiếu tới context và nó ngăn cản việc GC(Garbage Collection - trình thu gom rác) thu gom nó.
* Khi xảy ra sự kiện xoay màn hình, activity sẽ bị destroy và recreate lại nhưng trạng thái của activity vẫn được giữ lại. Như vậy hệ thống sẽ load lại UI của ứng dụng từ resource. Trong trường hợp bạn cần load 1 cái gì đó nặng và không muốn phải load lại khi xoay màn hình thì bạn sẽ để nó là static.



* Đoạn code trên có vẻ khá nhanh, nhưng lại thực sự sai. Lúc này drawable sẽ giữ 1 tham chiếu đến TextView, mà chính nó lại có 1 tham chiếu đến context (activity). Đây là trường hợp đơn giản nhất dẫn đến memory leak, còn nhiều trường hợp nữa sẽ dẫn đến memory leak nhiều hơn và gây ra out of memory nhanh hơn(inner class).
* Có 2 cách đơn giản để tránh memory leak trong trường hợp này:
  + Tránh thoát khỏi context trong phạm vi
  + Sử dụng Application context: context này sẽ sống khi ứng dụng sống và nó không phụ thuộc vào vòng đời của activity. Nếu bạn cần giữ 1 đối tượng lâu dài mà cần đến context thì nên dùng application context.
* Tóm lại để tránh memory leak nên làm theo 1 số bước sau:
  + Không giữ reference đến context trong thời gian dài(1 reference nên đi cùng với vòng đời của activity)
  + Cố gắng sử dụng application context thay vì activity context
  + Nên để inner class là static và sử dụng WeakReference
  + Không khai báo biến static cho view hoặc activity
  + Nhớ rằng luôn unregister broadcast, timer trong activity. Cancel asyncTask, thread trong onDestroy
  + Sử dụng weakReference khi cần
  + 1 Garbage collector không chống lại được memory leak
* Bạn nào rảnh có thể dịch bài này giúp mọi người nha
* <https://android-developers.googleblog.com/2009/01/avoiding-memory-leaks.html>
* <https://android.jlelse.eu/9-ways-to-avoid-memory-leaks-in-android-b6d81648e35e>

### Một số cách tránh memory leak

* 1. Broadcast receiver

- Nếu đăng ký ở onStart() thì nên hủy đăng ký ở onStop().

- Nếu đăng ký ở onResume() thì nên hủy đăng ký ở onPause()

- Và nên hủy đăng ký ở onDestroy

* 1. Static Activity and View Reference

- Nếu bạn khai báo view hoặc activity là static thì sẽ ngăn GC không thu hồi được bộ nhớ khi activity bị destroy. Vì vậy đừng bao giờ khai báo chúng là static.

* 1. Singleton class reference

- Không nên truyền activity context vào class singleton, thay vào đó là truyền application context.

- Hủy singleton trong onDestroy. Nếu bạn truyền activity context vào trong class singleton thì hãy đảm bảo nó được set về null.

* 1. Inner class reference

- Như đã đề cập ở trên, đừng bao giờ khai báo static cho biến inner class

- Nên khai báo inner class là static class

- Hoặc là sử dụng weakReference cho view/activity, vì GC có thể thu gom chúng.

* 1. Anonymous class reference

- Tương tự ý trên

* 1. AsyncTask reference

- Luôn cancel asyncTask trong onDestroy

- Nếu class AsyncTask được khai báo trong activity thì nó nên để static class (như ý trên).

- Có thể sử dụng weakReference

* 1. Handler reference
  2. Thread reference

- …

- gọi thread.interrupt trong onDestroy

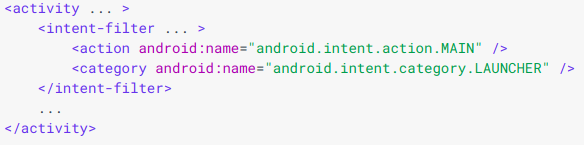
* 1. TimerTask reference

- cancel timer trong onDestroy

* 1. x

### Task and back stack

* 1 **task** là 1 tập các activities tương tác với người dùng. Những activities này được sắp xếp trong 1 **stack** - gọi là **back stack.**
* Khi 1 activity A đang chạy, 1 activity B được khởi chạy từ A, thì A sẽ được add vào back stack. Khi người dùng nhấn vào button back, A sẽ được pop khỏi back stack mà không phải tạo mới 1 activity.
* Khi ứng dụng ở chế độ **multi-screen(Android 7.0 - API 24)**, thì ở mỗi màn hình hệ thống sẽ quản lý task riêng biệt, mỗi màn hình có thể có nhiều task.
* Summary
  + Khi activity A start activity B, activity A sẽ stopped, nhưng sẽ được hệ thống giữ lại trạng thái(vị trí scroll, text input). Và khi đang ở activity B mà nhấn **Back**, activity A sẽ resumed và trạng thái trước đó sẽ được restore.
  + Khi người dùng nhấn vào nút Home, activity hiện tại sẽ bị stop, và task sẽ đi vào trạng thái background đồng thời trạng thái của các activity cũng được giữ lại. Sau đó, nếu người dùng select launch icon, task sẽ trở lại trạng thái foreground, đồng thời state của các activity cũng được restore.
  + Nếu người dùng nhấn Back, activity hiện sẽ bị destroy và popped khỏi stack.
  + Tại 1 thời điểm, activity có thể có nhiều instance và ở nhiều task khác nhau.
* **Launch mode**: cung cấp cách để 1 instance của activity được truy cập ở task hiện tại. Có 2 cách định nghĩa:
  + **Sử dụng Manifest**: Khi khai báo trong manifest, thì bạn sẽ xác định cách mà activity được truy cập trong task khi nó được start
    - **standard**: default mode, tạo mới 1 instance của activity khi mà nó được gọi.
    - **singleTop**: nếu 1 instance của activity đang tồn tại ở top của current task, hệ thống sẽ điều hướng đến instance đó thông qua onNewIntent(). Không thì sẽ tạo mới 1 instance
    - **singleTask:** Nếu activity chưa tồn tại instance, thì sẽ tạo 1 task mới và khởi tạo 1 instance của activity ở task này. Nếu activity đã tồn tại 1 instance ở 1 task rồi, thì hệ thống sẽ điều hướng đến instance đó thông qua onNewIntent(), và clear hết các activity ở trên nó.
    - **singleInstance**: Tương tự như singleTask, ngoại trừ việc task chỉ chứa duy nhất instance của activity đó mà không có bất cứ 1 instance của activity nào khác.
  + **Sử dụng Intent flags**: Khi 1 activity được start bởi startActivity(), bạn có thể set **flag** cho intent để xác định cách mà activity được truy cập trong task hiện tại.
    - FLAG\_ACTIVITY\_NEW\_TASK: = singleTask
    - FLAG\_ACTIVITY\_SINGLE\_TOP: = singleTop
    - FLAG\_ACTIVITY\_CLEAR\_TOP: nếu activity đã tồn tại 1 instance ở task hiện tại, thì hệ thống sẽ điều hướng đến instance thông qua onNewIntent(), đồng thời clear hết các instance khác ở phía trên nó, và lúc này activity sẽ ở top của task. Nếu activity chưa tồn tại instance nào, nó sẽ tạo mới instance và ở top của task.
* **Affinities:** sẽ quyết định 1 activity sẽ được khởi chạy ở task nào, mặc định các activity của app sẽ chạy trên 1 task. Nhưng bạn có thể tùy chỉnh được, với thuộc tính taskAffinity trong manifest. Giá trị của taskAffinity là String và duy nhất. Hệ thống sẽ sử dụng tên để định danh default task affinity cho app. taskAffinity được sử dụng trong 2 trường hợp
  + Khi activity được khởi chạy bởi intent có cờ FLAG\_ACTIVITY\_NEW\_TASK: Bình thường, activity sẽ khởi chạy ở 1 task mới. Nhưng nếu tồn tại 1 task nào đó, có cùng affinity với activity này, thì activity này sẽ được chạy ở task đó. Nếu không có thì nó sẽ chạy ở new task như bình thường.
  + Khi activity set thuộc tính allowTaskReparenting = true: 1 activity có thể chuyển từ task hiện tại của nó đang start tới 1 task có cùng affinity với nó, khi task có cùng affinity đó trở lại trạng thái foreground.
* **Clear the back stack**: sau 1 khoảng thời gian người dùng rời khỏi task, hệ thống sẽ tự động clear hết các activity trong task đó và chỉ giữ lại root activity. Root activity là activity nằm ở bottom của stack. Thông thường đó là activity được khởi chạy đầu tiên.



Tuy nhiên, có 1 số trường hợp activity được khởi chạy đầu tiên không phải là root activity, ví dụ:

Task có 3 activity: A(root) -> B -> C, C sẽ start activity D với intent có flag FLAG\_ACTIVITY\_CLEAR\_TASK|FLAG\_ACTIVITY\_NEW\_TASK, thì tất cả các task, và activity sẽ được start ở 1 new task và nó trở thành root activity.

* Một số thuộc tính sau có thể tùy chỉnh việc này:
  + alwaysRetainTaskState = true: task sẽ retain tất cả activity trong stack của nó trong thời gian dài.
  + clearTaskOnLaunch = true: ở root activity, thì khi người dùng rời khỏi task và trở lại thì hệ thống sẽ clear hết các activity có trong task, ngoại trừ root activity.
  + finishOnTaskLaunch = true: với activity nào set thuộc tính này (kể cả root activity) thì khi người dùng rời khỏi task và trở lại thì hệ thống sẽ clear những activity này.
* **Starting task**
* x
* x
* x
* x

### OOP

* OOP là cách nhìn nhận 1 bài toán trong thực tế dưới dạng các đối tượng, xem xét các tính chất đối tượng và các hành động tương tác giữa các đối tượng đó. Sau đó, biểu diễn chúng dưới dạng các class trong ngôn ngữ lập trình với các tính chất là các trường - field, các hành động động là các phương thức - method.
* (OOP là cách nhìn nhận và quan sát các đối tượng trong tự nhiên, rồi nắm bắt những hành vi và đặc tính của chúng và biểu diễn chúng dưới dạng các class với các đặc điểm là các thuộc tính, các hành vi là các phương thức)
* Object là tất cả những thực thể trong tự nhiên có thể sờ và nhìn thây đc
* Class là 1 khuôn mẫu hay bản thiết kế để tạo ra Object
* Object là 1 thể hiện của Class: trạng thái và hành vi
* **Tính đóng gói:** là việc ẩn đi các biểu diễn bên trong của 1 lớp: bao gồm data và 1 số phương thức xử lý khỏi các view bên ngoài.
* Mục đích: che giấu dữ liệu
* Tính đóng gói được thể hiện thông qua access modifier

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Class | package | Subclass | Global |
| public | Y | Y | Y | Y |
| protected | Y | Y | Y | **N** |
| default | Y | Y | **N** | **N** |
| private | Y | **N** | **N** | **N** |

* **Tính trừu tượng:** là việc loại bỏ đi những thông tin không cần thiết của 1 lớp và chỉ giữ lại những thông tin cốt lõi.
* Hay nói cách khác tính trừu tượng là một tiến trình ẩn đi các chi tiết của trình triển khai và chỉ hiển thị các tính năng tới người dùng
* Mục đích: làm đơn giản hóa 1 lớp, để tập trung vào những cốt lõi cần thiết thay vì phải implement chi tiết. Nó thể hiện 1 mô hình, khung nhìn tổng quát của 1 thực thể.
* Tinh truu tuong duoc the hien boi abstract class va interface
  + Abstract class: dược thể hiện bởi từ khóa abstract
  + 1 class có thể trừu tượng từ 0 - 100 %
  + 1 abstract class có thể không có abstract method nào. Nhưng 1 abstract method chắc chắn phải nằm trong 1 abstract class.
* sự khác nhau giữa abstract class và interface

|  |  |  |
| --- | --- | --- |
|  | Abstract | Interface |
| Abstract method | có hoặc không | từ java 8 trở đi chỉ chứa các abstract method |
| Đa kế thừa | không hỗ trợ | có hỗ trợ |
| Implement và extend | - abstract class có thể implement một interface  - abstract class có thể extend một lớp và imp nhiều interface | - một interface không thể extend một abstract class  - một interface chỉ có thể extend các interface khác |
| từ khóa | sử dụng từ khóa abstract để khai báo một abstract class | sử dụng từ khóa interface để khai báo một interface |
| ... | ... | ... |

* **Tính kế thừa:** là việc tạo ra 1 lớp từ những lớp đã có trước đó.
* Mục đích: tái sử dụng lại code đã có trước đó và tránh dư thừa dữ liệu.
  + Để tạo class kế thừa từ class trước, sử dụng từ khóa extends
  + Cách này làm gia tăng số lượng class nhưng lợi ích nó đem lại rất lớn
  + Để class con sử dụng được các thuộc tính, phương thức của lớp cha, sử dụng từ khóa **super**
  + Các loại kế thừa:
    - A extends B: đơn kế thừa
    - A extends B extends C: kế thừa nhiều tầng
    - A, B extends C: kế thừa thứ bậc
  + Java không hỗ trợ đa kế thừa(để đơn giản hóa và giảm sự phức tạp)
  + Nhưng với interface thì 1 class có thể implement nhiều class.

// chỉ cần gọi super, thì cũng có thể hiểu đó là phương thức Override

public void say() {

super.say();

}

// sử dụng annotation

@Override

public void say() {

}

* **Tính đa hình:** là việc 1 hành động của lớp có nhiều hình thái khác nhau tại mỗi thời điểm khác nhau, duoc the hien o 2 kieu: override va overload
* biểu hiện là phương thức cùng tên, nhưng khác nhau về cách biểu diễn(khác nhau về số lượng tham số hoặc khác nhau về kiểu của tham số)(overload); một lớp con có thể ghi đè 1 phương thức của lớp cha(override).
  + Có 2 kiểu đa hình:
    - Override: đa hình run-time, tức là phương thức nào được gọi đến sẽ được quyết định lúc thực thi chương trình.
    - Overload: đa hình complie-time, tức là phương thức nào được gọi sẽ được quyết định ngay ở trình biên dịch.

### Up casting and down casting

Xảy ra trên cùng 1 cây phân cấp.

// up casting: là khả năng nhìn nhận 1 lớp dẫn xuất(con) như 1 lớp cơ sở(lớp cha)

* ngầm định ()

Employee e = new Employee();

Person p = e;

// down casting: là chuyển đổi kiểu của một đối tượng từ một thể hiện của lớp cha xuống thành một đối tượng là thể hiện của lớp con.

* Ép kiểu

Person p2 = new Person();

Employee e2 = (Employee) p;

### Abstract and Interface

* Xét về mục đích sử dụng, abstract class được dùng để đơn giản hóa (tổng quát hóa) một đối tượng, class bao gồm các field và phương thức abstract và không abstract. quan hệ is a
* Interface được sử dụng để định nghĩa các hành động của một đối tượng có thể có mà không triển khai nó. Nó được xem như 1 contract, khuôn mẫu, chức năng mở rộng. quan hệ can do

|  |  |
| --- | --- |
| Abstract | Interface |
| is a | can do |
| Không hỗ trợ đa kế thừa | Hỗ trợ đa kế thừa |
| Trừu tượng từ 0 - 100% | Trừu tượng 100%, tức là chỉ chứa các phương thức abstract |
| public, protected, private | Các biến mặc định là public static final |
| Có constructor | Không có constructor |
| có thể định nghĩa thân hàm | không định nghĩa thân hàm |

### Class and Object

- Class là 1 khuôn mẫu, khuôn hình để tạo nên các đối tượng - object

- Object là 1 thể hiện của class

### List and ArrayList

* List là 1 interface, arraylist là 1 class implement từ List.
* Khi khai báo 1 biến kiểu list như sau:

List list = new ArrayList();

Thì biến list chỉ sử dụng được các phương thức mà interface list định nghĩa, và không thể sử dụng được các phương thức khác của lớp ArrayList. Tuy nhiên, các phương thức trong List chỉ là abstract, nên khi run thì nó sẽ gọi đến phương thức đó mà ArrayList đã implement. -> đa hình.

* Ví dụ khác tương tự:

public interface Navigator {

public static final int x = 10;

public abstract void open();

}

public class ToolbarNavigator implements Navigator {

@Override

public void open() {

System.out.println("Toolbar navigator");

}

public void close() {

}

}

**Navigator navigator = new ToolbarNavigator();**

**navigator.open();**

Ở đây, biến navigator chỉ có thể truy cập đến method open.

Mặc dù khi, ctrl + click vào method open trình biên dịch vẫn chỉ đến method open của interface Navigator, nhưng khi chạy thì nó sẽ chạy vào method open của class ToolbarNavigator

### Activity

* Activity là 1 trong 4 component chính của Android. Là nơi tập trung các thao tác của người dùng có thể thực hiện. Hầu hết các activity đều tương tác với người dùng, nên nó cung cấp 1 cửa sổ (màn hình) để hiển thị UI cho người dùng thao tác thông qua hàm setContentView(). Cửa sổ này có thể full màn hình hoặc chỉ như 1 floating windows
* Activity là một trong 4 component chính của android, biểu diễn giao diện của một màn hình nào đó trong ứng dụng. và một ứng dụng có thể có nhiều màn hình

### Các trạng thái chính của activity

- Active - foreground

- pause

- stop

- destroy

**Note**: Khi goi activity gọi finish, thì những câu lệnh phía sau lệnh này vẫn được thực hiện bình thường, sau đó activity sẽ vào onPause, onStop, onDestroy.

Start Activity từ notification: <https://developer.android.com/training/notify-user/navigation#java>

### Fragment

- Fragment là 1 phần cung cấp giao diện để người dùng tương tác với ứng dụng tương tự như activity. Nó ra đời để giải quyết vấn đề đa màn hình.

- 1 Fragment khi được sử dụng nó phải được gán vào 1 activity, vì thế vòng đời của activity ảnh hưởng trực tiếp đến vòng đời của fragment.

- Khi activity ở trạng thái pause thì fragment cũng ở trạng thái paused, tương tự với stop. Nhưng khi activity đang ở trạng thái resumed(running) thì fragment có thể thực hiện các hành động add, replace, remove.. bởi transaction.

### Giao tiếp giữa Activity và Fragment

- Có 2 cách:

- Cách 1: sử dụng callback từ Fragment, Activity implement interface

- Cách 2: từ activity gọi findFragment

- <https://developer.android.com/training/basics/fragments/communicating>

### FragmentTransaction

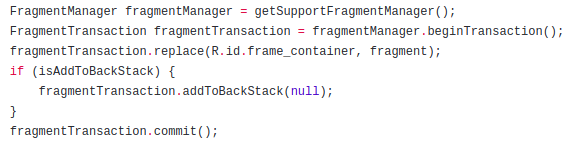
- **add**: thực hiện thêm 1 fragment vào view container của activity

- **remove**: thực hiện xóa 1 fragment khỏi view container của activityessentially

- **replace**: bản chất = remove + add: thực hiện remove hết tất cả các fragment đã được add vào view container id, và add 1 fragment mới vào view container này.

- Khi 1 fragment được add bởi phương thức add(), thì fragment sẽ không được tự động add vào back stack.

- Trước khi thực hiện các transaction: add, remove, replace đối với 1 fragment, nếu transaction này được add vào back stack thì transaction đó vẫn tồn tại trong back stack).



- Nếu replace fragment A như trên, và sau đó tiếp tục gọi replace B, thì khi ở B mà back lại thì fragment A sẽ được restore từ back stack. (vì transaction đã được add to back stack)

### Activity lifecycle

* <https://docs.google.com/document/d/1htn6AXrLDUSW0DuCifMYdYbdrV86LFvmUdO71h1RbAQ/edit>
* Bấm nút Home: onPause, onStop.
  + Nếu ứng dụng không bị clear bởi task đa nhiệm, thì khi mở lại từ task đa nhiệm, hoặc icon app, nó sẽ vào onRestart, onStart, onResume.
  + Nếu ứng dụng bị clear khỏi task đa nhiệm, thì khi mở lại bởi icon app nó sẽ vào onCreate, onStart, onResume.
* Bấm nút Back: onPause, onStop, onDestroy
* Có ứng dụng khác kích hoạt: onPause, onStop
* Xoay màn hình: onPause, onStop, onDestroy, onCreate, onStart, onResume
* Tắt nguồn: onPause, onStop
* Đa màn hình: onPause
* Gọi 1 intent không tường minh, show kiểu dialog: onPause
* **onDestroy** khi người dùng nhấn nút back.
* **onPause**: khi vẫn nhìn thấy giao diện nhưng không tương tác được.

### Fragment lifecycle

- <https://docs.google.com/document/d/1oxmc49fAP2kpemIzxQyb49fN9rDjL9LDHOIviD_HtiU/edit>

- onPause: khi người dùng nhấn nút back, hoặc fragment bị replace hoặc remove. Hoặc fragment được add to back stack sau đó replace hoặc remove

- onDestroyView: Nếu fragment được add to back stack thì khi nó trở lại từ back stack nó sẽ vào onCreateView.

- Điểm khác biệt lớn nhất giữa lifecycle của Fragment và Activity là các lưu trữ trong back stack.

- Activity được đưa vào back stack và được quản lý bởi hệ thống

- Fragment được đưa vào back stack và được quản lý bởi Activity chứa nó. Fragment được lưu lại bởi phương thức addToBackStack().

### Thread, Handler, AsyncTask

* <https://docs.google.com/document/d/10depWXRM-IEJh16vMB8NmyRCdiNN9h_7M9kVD3rMQD4/edit#>

### Broadcast Receiver

* Broadcast hoạt động như 1 hệ thống gửi và nhận tin nhắn trên các ứng dụng và bên ngoài luồng người dùng thông thường.
* Có 2 cách đăng ký sử dụng Broadcast
  + Manifest
  + Context register services
* Có 3 cách để gửi Broadcast
  + sendOrderedBroadcast
  + sendBroadcast
  + LocalBroadcastManager.sendBroadcast

### Content Provider

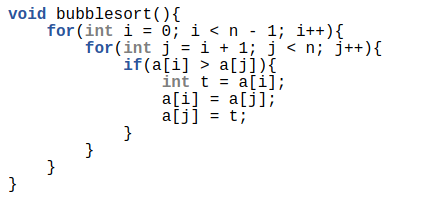
- Content provider giúp quản lý quyền truy cập vào dữ liệu được lưu trữ bởi nó hoặc ứng dụng khác.Và nó cung cấp 1 cách mà các ứng dụng có thể trao đổi dữ liệu với nhau. Mỗi ứng dụng sẽ đóng gói dữ liệu, và quy định những cơ chế bảo mật khác nhau để các ứng dụng khác có thể truy cập vào.

- Content provider được xem như 1 interface để các ứng dụng giao tiếp với nhau trong việc trao đổi dữ liệu.

- có thể giúp ứng dụng quản lý truy cập dữ liệu được lưu trữ bởi chính bản thân nó, được lưu trữ bởi ứng dụng khác và cung cấp một cách để chia sẻ dữ liệu với những ứng dụng khác. Các content provider sẽ đóng gói dữ liệu, cung cấp những cơ chế để định nghĩa an toàn dữ liệu. Content provider là những giao diện chuẩn để kết nối dữ liệu trong một tiến trình với code chạy trong những tiến trình khác. Cài đặt một content provider có nhiều ưu điểm. Quan trọng nhất là ta có thể thiết đặt một content provider để cho phép những ứng dụng khác có thể truy cập một cách an toàn và chỉnh sửa dữ liệu của ứng dụng như được mô tả bởi hình dưới đây:

### Một số thuật toán sắp xếp

* 1. Sắp xếp nổi bọt - bubble sort
* Ý tưởng: sử dụng 2 lần duyệt lồng nhau
* Lần duyệt 1: duyệt qua n phần tử
* Lần duyệt 2: tại mỗi phần tử của lần duyệt 1, so sánh a[i] và a[j] để đưa phần tử lớn hơn(nhỏ hơn) về đầu dãy

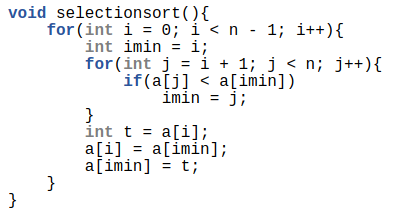


* 1. Sắp xếp chọn - selection sort

- Ý tưởng: sử dụng 2 lần duyệt lồng nhau

- Lần duyệt 1: duyệt qua n phần tử

- Lần duyệt 2: tại mỗi phần tử của lần duyệt 1, tìm phần tử nhỏ nhất(lớn nhất) để đưa nó về vị trí đầu tiên.

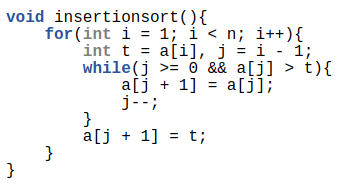


* 1. Sắp xếp chèn - insertion sort

- Ý tưởng: sử dụng 2 lần duyệt

- Lần duyệt 1: duyệt qua n phần tử

- Lần duyệt 2:



* 1. Sắp xếp trộn - merge sort

- Ý tưởng: chia để trị

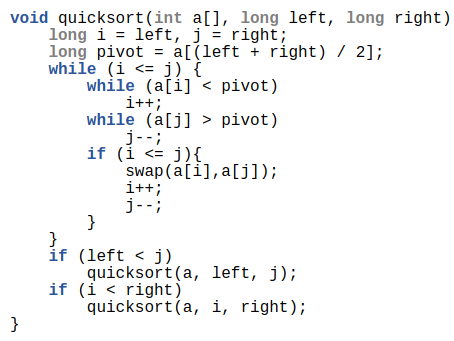
- Cách chia: chia 2 ....

=> bài toán quay về gộp 2 mảng đã được sắp xếp.

* 1. Sắp xếp nhanh - quick sort

- Ý tưởng: chia để trị

- Cách chia: chọn pivot....



* 1. x
  2. x

|  |  |  |
| --- | --- | --- |
|  | Độ phức tạp |  |
| Bubble sort | O(n\*n) |  |
| Selection sort | O(n\*n) |  |
| Insertion sort | O(n\*n) |  |
| merge sort | O(n \* log(n)) | Có tính ổn định |
| quick sort | O(n) | Phụ thuộc vào dữ liệu |

### Java Memory Management

|  |  |
| --- | --- |
| **Stack** | **Heap** |
| Cấp phát bộ nhớ tĩnh cho để thực thi các luông nào đó (thread). Nó lưu các giá trị nguyên thủy, method, và reference tới object trong bộ nhớ heap.  Hoạt động theo cơ chế Last-in-First-out.  Stack lưu trữ giá trị nguyên thủy và reference đến đối tượng được sử dụng trong method, khi method kết thúc, vùng nhớ đó sẽ được giải phóng.  - Bộ nhớ stack thì rất nhỏ so với bộ nhớ heap | Cấp phát bộ nhớ động cho các object lúc run-time, bất cứ khi nào đối tượng được tạo, nó sẽ được lưu trong bộ nhớ heap.  Khi bộ nhớ heap đầy, trình thu gom rác (Garbage Collection) sẽ thực hiện quét và xóa những đối tượng không còn được sử dụng đi.  - Đôi khi heap được chia nhỏ thành 3 vùng nhớ như sau:   |  |  |  | | --- | --- | --- | | Young Generation | Old Generation | Permanent Generation | | Dành để cấp phát cho các đối tượng mới | Khi 1 đối tượng tồn tại đủ lâu trong bộ nhớ, nó sẽ được chuyển vào đây | Chứa metadata lúc runtime và các method của ứng dụng | |
| So sánh | |
| Được sử dụng khi 1 luồng thực thi | Được dùng trong tất cả trường hợp sử dụng bộ nhớ |
| Chỉ chứa các biến dữ liệu nguyên thủy và các biến tham chiếu tới các đối tượng trong bộ nhớ heap | Chỉ chứa các object |
| Chỉ truy cập được khi ở cùng 1 thread | Có thể truy cập 1 cách toàn cục |
| Cơ chế LIFO | Phức tạp |
| Thời gian tồn tại ngắn | Tồn tại từ lúc chương trình bắt đầu thực thi đến khi kết thúc |
| Khi stack đầy -> exception  java.lang.StackOverFlowError | Khi heap đầy -> exception  java.lang.OutOfMemoryError |
| Kích thước nhỏ, tốc độ truy cập nhanh | Kích thước lớn, truy cập chậm hơn stack |

### Thread

* <https://docs.google.com/document/d/10depWXRM-IEJh16vMB8NmyRCdiNN9h_7M9kVD3rMQD4/edit>

### Java là ngôn ngữ thông dịch hay biên dịch

* Cả 2 đều đúng, tùy thuộc vào góc độ mà bạn nhìn nhận (insider or outsider)
* Biên dịch: java được xem như compiler để biên dịch file .java thành file .class dạng byte-code (cách nhìn từ phía trong).
* Thông dịch: mỗi lần chạy chương trình, máy ảo JVM sẽ thông dịch file. class để máy tính có thể hiểu vào chạy chương trình (cách nhìn từ phía ngoài)
* Thông dịch là gì?
  + Đến câu lệnh nào thì sẽ nó sẽ được dịch ra byte code luôn để máy tính thực thi - dịch song song
* Biên dịch là gì?
  + Dịch cả toàn bộ chương trình thành byte code sau đó máy mới thực thi từ byte code này - dịch không song song.

1. Class and Object and Instance.

* Object là những sự vật được mô tả thông qua những thuộc tính và phương thức
* Class là các đối tượng có những đặc điểm, phương thức chung sẽ được định nghĩa thành 1 class, Class như là 1 bản thiết kế mà từ đó các đối tượng cụ thể được sinh ra.
* Instance là khởi tạo 1 đối tượng( new Object thì ms là instance, còn chỉ khai báo bình thường thì nó vẫn chỉ là biến)

### 4 tính chất của OOP

* Tính đóng gói, kế thừa, đa hình, trừu tượng.

1. Tính đóng gói

* Là việc đưa các thuộc tính

1. Access modifier

* private, default, protected, public

1. Tính kế thừa

* Là cách tạo ra 1 lớp từ 1 lớp đã có trước đó.
* Mục đích: tái sử dụng code: thuộc tính và các phương thức của lớp cha.
* Trong java không có khả năng đa kế thừa, để có được khả năng này, sử dụng interface
* Các loại kết thừa:
  + Đơn kế thừa
  + Kế thừa nhiều tầng
  + Kế thừa thứ bậc (kế thừa có cấu trúc)
* extends: is a: là
* implement: can do: có thể thực hiện

1. Tính trừu tượng

* Là việc lược bỏ những tính chất không cần thiết của đối tượng, chỉ giữ lại những thông tin cần thiết trong bài toán.

1. Tính đa hình

một tác vụ được thực thi theo nhiều cách khác nhau. có hai loại đa hình:

* run time : override
* compile time: overload

1. Luồng thực hiện của try catch final

* Các câu lệnh cần thực thi được viết trong try{} và được thực thi khi chạy ctrinh.
* catch{}: khi câu lệnh trong try có ngoại lệ xảy ra thì nó sẽ nhảy vào khối này để bắt ngoại lệ.
* finally{}: khối này luôn luôn được chạy cho dù là dòng lệnh trong try có ngoại lệ.

1. Phân biệt value type và reference type

* **Câu trả lời chưa được xác nhận**: cả 2 được lưu trong heap(cái này đúng ko vậy, đối tượng nguyên thủy tưởng sẽ được lưu trong stack, chỗ nào new ms đc lưu trong heap chứ nhỉ, reference là sao), heap đều cấp phát bộ nhớ cho chúng

(tớ nhớ k lầm thì reference thì k dc cấp phát)

Với đối tượng nguyên thủy, thì

Với object,

Trong java, đối tượng nguyên thủy và object được truyền by value

* Kiểu reference type thì chỉ lưu địa chỉ của biến chứ không lưu trực tiếp giá trị.
* Kiểu value type nó lưu trực tiếp giá trị lên bộ nhớ.

Tham khảo: <http://www.tutorialsteacher.com/csharp/csharp-value-type-and-reference-type>

1. Generic

* Generic type là 1 class hoặc interface được tham số hóa kiểu dữ liệu. Nói nôm na là cách thức lập trình tổng quát cho phép 1 object hoạt động với nhiều kiểu dữ liệu khác nhau.
* được sinh ra để tránh un cast exception

### Có bao nhiêu kiểu dữ liệu nguyên thủy: 8

|  |  |
| --- | --- |
|  | default value |
| byte | 0 |
| short | 0 |
| int | 0 |
| long | 0L |
| float | 0.0f |
| double | 0.0d |
| boolean | false |
| char | ‘\u0000’ |

* String là kiểu dữ liệu tham chiếu(Object)
* Giá trị default của String hay bất kỳ Object nào đều là null.

### Biến lớp, local và instance

* **local**: là biến được khai báo trong constructor, phương thức, và bất kỳ khối lệnh. nó sẽ bị hủy khi kết thúc phương thức hay khối lệnh.(các tham số truyền vào hoặc các biến trong các phương thức)
* **class/static**: là biến được khai báo với 1 lớp, bên ngoài bất cứ phương thức nào, với từ khóa static.
* **instance**: là biến được khai báo trong 1 lớp, nhưng bên ngoài constructor, phương thức, và bất kỳ khối lệnh nào.(các biến thuộc tính private String name,....)

### Android là gì?

* Là 1 hệ điều hành dựa trên nền tảng Linux được thiết kế dành cho các thiết bị di động có màn hình cảm ứng.
* Ban đầu được phát triển bởi Tổng công ty Android với sự giúp đỡ từ Google, và sau đó được Google mua lại vào năm 2005 và ra mắt vào năm 2007.

1. Threadpool

* Tạo ra 1 số lượng thread nhất định, nếu có nhiều hơn số lượng thread đó, thì nó sẽ đợi cho đến khi các thread khác thực hiện xong.
* Là 1 pool chứa các thread có nhiệm vụ giới hạn số lượng thread được chạy bên trong ứng dụng tại 1 thời điểm, tái sử dụng nhiều lần.
* Khi cần thread để xử lý 1 tarsk nào đó. chương trình sẽ tìm trong pool. Nếu có thread sẵn sàng thì dùng thread đó mà k cần tạo mới. nếu k thì sẽ tạo mới và thêm vào pool.
* >chương trình chạy nhanh hơn vì k tốn time tạo thread làm giảm số lượng thread phải tạo.

1. Các thư viện load ảnh
   1. Các thư viện load ảnh có thể kể đến như là: Glide, Picasso, ImageLoader, Fresco.
   2. Glide

Cách mà Glide load ảnh lên View như sau:

* Đầu tiên nó vào memory cache để xem image với url đó đã được lưu hay chưa, nếu được lưu rồi thì nó chỉ việc load lên View.
* Nếu chưa được lưu thì nó đọc tiếp trong disk cache, nếu trong này được lưu image với url rồi thì nó sẽ load lên View đồng thời lưu vào memory cache.
* Nếu chưa có thì nó sẽ tải ảnh từ url và hiển thị lên View đồng thời lưu vào disk cache và memory cache.
  1. So sánh Glide và Picasso

|  |  |
| --- | --- |
| **Glide** | **Picasso** |
| Phương thức .with() của Glide có thêm các tham số Activity/Fragment.  khi đó thì việc load ảnh của Glide sẽ gắn với vòng đời của Activity hay Fragment. khi nó pause thì Glide cũng pause, và nó onResume thì nó cũng load ảnh trở lại. | Chỉ có thể truyền context vào phương thức .with(). |
| Bitmap mặc định là ở dạng RGB\_565. ảnh sẽ không được nét. | Mặc định ảnh ở dạng ARGB\_8888: ảnh load về sẽ nét hơn. |
| Bộ nhớ: Glide sẽ cache kích thước của ImageView, do vậy muốn hiển thị một ảnh với nhiều kích thước khác nhau thì phải tải nhiều lần và bộ nhớ sẽ dành 1 vùng để lưu kích thước image. do vậy Glide sẽ chiếm dụng nhiều bộ nhớ hơn. | Picasso thì sẽ cache kích thước đầy đủ của ảnh, do vậy nó tốn ít bộ nhớ hơn so với Glide. |
| Tốc độ: Glide load ảnh nhanh hơn vì nó chỉ cần lấy ảnh trong cache và hiển thị lên ImageView. (cache ảnh có độ tương tự, trước tiên thay đổi kích thước sau đó lưu) | Picasso chậm hơn vì nó phải load ảnh với kích thước đầy đủ vào bộ nhớ và để cho CPU thay đổi kích thước của ảnh trong quá trình đang chạy. |
| Có khả năng tải một ảnh GIF | không thể tải ảnh GIF |
| Có thể thiết lập cách hình ảnh xuất hiện với một Animator. | Chỉ có một hiệu ứng là Fade-in. |

* 1. Nếu không sử dụng thư viện load ảnh, bạn sẽ thực hiện load ảnh như thế nào?

### Cơ chế Cache ảnh của Glide

* <https://bumptech.github.io/glide/doc/caching.html>
* Mặc định Glide kiểm tra nhiều lớp cache trước khi đi đến 1 new request cho 1 image
  + 1: Active resource: Kiểm tra ngay tại thời điểm này, image này có đang được hiển thị ở 1 view khác hay không?
  + 2: Memory cache: Kiểm tra xem image này gần đây có được tải và vẫn còn lưu trong bộ nhớ không?
  + 3: Resource: Kiểm tra xem trước đây, image này có được giải mã, chuyển đổi và ghi vào đĩa đệm không?
  + 4: Data: Dữ liệu này có được thu thập và lưu vào bộ nhớ trước đó không?
* 2 bước đầu là kiểm tra xem image có trong bộ nhớ không, nếu có thì hiển thị ngay lập tức. 2 bước sau là kiểm tra xem image có trong disk không, nếu có thì cũng hiển thị ngay lập tức. Nếu qua 4 bước trên không có image, thì glide sẽ request trực tiếp đến image gốc.
* continue

1. List view, recycler view, load image library và so sánh các thư viện đó, fragment khác gì view.

### Cơ chế hoạt động của ListView

* Với mỗi item, listview yêu cầu adapter cung cấp 1 View thông qua hàm getView(), lúc này 1 view sẽ được tạo và hiển thị.
* Tuy nhiên ListView sẽ không tạo mới tất cả các item, mà nó sẽ sử dụng 1 cơ chế để sử dụng lại các view trước đó, cơ chế này gọi là Recycler.
* Tức là nếu có 1000 item, thì listview chỉ tạo ra những item được visible trong bộ nhớ + view được recycler.
* Số lượng item mà listview tạo = số lượng item visible + 1 item
* **public** View getView(**int** position, View convertView, ViewGroup parent)
* Lần đầu tiên, đối với những item visible, convertView sẽ là null và bạn sẽ tạo 1 view mới và return nó.
* Khi listview bắt đầu scroll up(tương tự với scroll down), thì 1 item ở top sẽ invisible và nó sẽ được chuyển xuống bottom thông qua convertView. Tại hàm getView, bạn check convertView, nếu không null, thì bạn chỉ cần set lại data cho nó và return nó.
* <http://android.amberfog.com/?p=296>

### Cơ chế hoạt động của RecyclerView, tái sử dụng ViewHolder.

* Bắt đầu có từ Android 5
* Về cơ bản nó giống với listview(cũng sử dụng cơ chế recycler)
* **onCreateViewHolder**: được gọi khi RecyclerView cần 1 tạo mới 1 ViewHolder đại diện cho 1 item. Tại hàm này bạn sẽ tạo mới 1 view theo cách thủ công hoặc sử dụng inflate.
* **onBindViewHolder**: được gọi khi RecyclerView cần hiển thị dữ liệu tại 1 vị trí xác định. Tại đây, nên update nội dung của item view thông qua position. Không giống như ListView, RecyclerView sẽ không gọi lại phương thức này nếu như vị trí của item bị thay đổi trong dataset trừ khi bản thân item invalidated hoặc vị trí mới không thể xác định.

### Điểm khác nhau giữa ListView và RecyclerView

* Với Listview, nó không bắt buộc việc sử dụng lại các view trước đó, tức là trong hàm getview, bạn có thể hoàn toàn tạo mới 1 view tại bất cứ item nào. Còn với RecyclerView việc tạo mới và update view chia thành 2 phương thức onCreateViewHolder() và onBindViewHolder().
* ViewHolder: mục đích của ViewHolder là để giữ lại tham chiếu của View, nhờ đó phương thức findViewById chỉ được gọi 1 vài lần, thay vì phải gọi cho toàn bộ tập dữ liệu, trong ListView việc sử dụng ViewHolder là không bắt buộc (vì khi sử dụng ListView thì không phải ai cũng nắm được ViewHolder pattern này.), do đó nó không hạn chế được việc gọi phương thức findViewById. Còn đối với RecyclerView việc triển khai 1 adapter thì đã bắt buộc phải triển khai ViewHolder.
* LayoutManager: với ListView bạn chỉ có thể hiển thị các item theo chiều dọc. Với recycler view, nó cung cấp class LayoutManager để chịu trách nhiệm việc bố trí layout cho các item, vì thế recyclerview có thể hiển thị danh sách các item theo chiều dọc, ngang, sole.
* ItemDecoration: recyclerview cung cấp lớp này để bạn có thể dễ dàng add thêm 1 số tùy chỉnh cho các item. VD, mặc định recyclerview không có divider(đường chia giữa các item), để làm được điều này bạn có thể custom dễ dàng. Tuy Listview cũng có divider nhưng nó không được linh hoạt như recyclerview.
* ItemAnimator: RecyclerView.ItemAnimatior xử lý animation, mặc định ListView là fade thì phải.
* NotifyAdapter: ListView chỉ cung cấp notifyDatasetChange còn RecyclerView cung cấp nhiều hơn: notifyInsertItem, notifyRemoveItem, notifyItemChange

### Sự khác nhau của adapter position và layout position

* Về cơ bản 2 giá trị này không khác nhau, nó chỉ khác nhau trong trường hợp sau:
* Khi có nhiều dữ liệu và thời gian tính toán để load dữ liệu lên giao diện, thì khi đó Adapter đang thay đổi dữ liệu nhưng giao diện thì chưa. Trong khoảng thời gian đó thì kết quả của của phương thức getAdapterPosition() và getLayoutPosition() sẽ có kết quả khác nhau.
* getLayoutPosition(): trả về vị trí của ViewHolder sau khi layout mới nhất được cập nhật. Nó được dùng chủ yếu bởi các thành phần trong RecyclerView. Nếu trong trường hợp LayoutManager cần gọi 1 phương thức bên ngoài để yêu cầu position thì nên sử dụng getAdapterPosition.
* getAdapterPosition: trả về vị trí của item được được represented (đại diện) bởi ViewHolder này. RecyclerView không xử lý bất kỳ cập nhật nào của adapter cho đến lần layout được thay đổi tiếp theo. Điều này có thể gây ra mâu thuẫn với getLayoutPosition(). Tuy nhiên, vấn đề này cũng không quan trọng lắm vì thời gian cập nhật giao diện rất là nhanh 16ms. Nó chỉ xảy ra mâu thuẫn khi bạn sử dụng vị trí của ViewHolder để truy cập vào Adapter.

1. Cơ chế cache ảnh của Glide, load ảnh kích thước lớn vào ảnh nhỏ.

**+Cơ chế cache ảnh của glide:**

Sử dụng cache có thể giúp ta hạn chế download, decode ảnh lần nữa, vừa tăng performance cho ứng dụng.

Glide check nhiều lớp cache trước khi gửi request 1 image:

-Activity resource: image được hiển thị trong view khác hay k?

-Memory cache: image đã được load và tồn tại trong memory?

-Resource: image đã được decoded và ghi vào disk cache trước đó?

-Data: dữ liệu image đã được thu thập để ghi vào disk cache trước đó?

Hai bước đầu tiên kiểm tra url đó đã có trong memory cache chưa?

-Nếu đã có thì chỉ load ảnh từ memory cache

-Nếu k có thì check ở disk cache

-Nếu có ở trong disk cache thì nó sẽ load ảnh vào view và lưu vào memory

-Nếu k có trong cả 2 thì load ảnh, hiển thị lên view rồi lưu vào memory và disk cache.

**+Bitmap Pool:**

Android update UI mỗi 16ms, nếu quá thời gian này, android sẽ hủy bỏ frame đó. Nếu android bỏ qua nhiều frame, FPS sẽ ít đi khiến cho ứng dụng giật lag làm cho Garbage Collector phải liên tục hoạt động khiến UI bị đơ, lag. -> Bitmap Pool:

Bitmap Pool là nơi chứa các bitmap không còn sử dụng nhưng có thể sử dụng lại để decode, load bitmap mới vào cùng vùng nhớ. Pool này hoạt động dựa trên khái niệm inBitmap

***Ví dụ:***

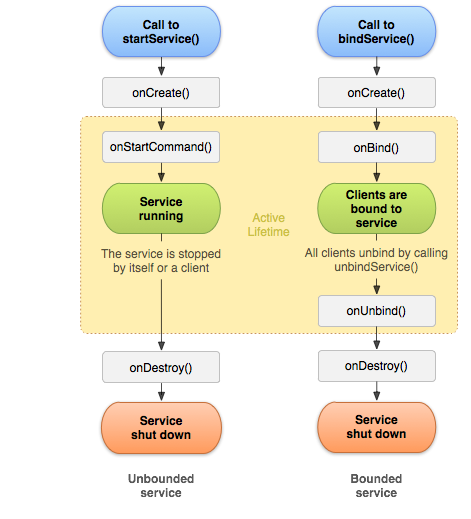
Giả sử app cần load 2 bitmap ( 1 và 2). Khi load bitmap 1 hệ thống sẽ cấp phát bộ nhớ cho bitmap này, khi không còn cần dùng đến nó, thay vì hủy bỏ thì sẽ dùng bitmap 1 như là inBitmap cho bitmap 2. Do đó, 1 phần bộ nhớ được sử dụng lại.

Source tham khảo:

<https://viblo.asia/p/android-thu-vien-load-anh-glide-hoat-dong-nhu-the-nao-Az45bgnwKxY>

1. Truy vấn dữ liệu SQL cơ bản.
2. **Service**
   1. Giới thiệu

* Service là một context dùng để thực thi các tác vụ chạy ngầm mà không liên quan đến giao diện.
* Chỗ nào có Context đều khởi tạo được một Service.
  1. phân loại
* Có 3 loại service:
  + Forceground Service: Đây là một service mà nó thực hiện một số thao tác mà người dùng chú ý, có thể thấy rõ ràng. Ví dụ một ứng dụng nghe nhạc, ta có thể control nhạc bằng cách dùng loại service này.
  + Background Service: loại này sẽ thực hiện các hành động mà không được người dùng chú ý trực tiếp. ví dụ một ứng dụng sử dụng service này để thu gom bộ nhớ, công việc này không gây được sự chú ý từ người dùng.
  + Bound Service: Một service được gọi là Bound khi một thành phần của ứng dụng ràng buộc với nó bởi lời gọi bindService(). Một Bound Service cung cấp một giao diện Client - Server cho phép các thành phần tương tác với nó.
  1. Vòng đời của Service:



* 1. Hàm onStartCommand()

hệ thống gọi phương thức này bằng cách gọi hàm startServce() khi 1 thành phần khác yêu cầu service khởi động. Khi phương thức này thực hiện, dịch vụ được khởi động và chạy nền vô thời hạn. Dừng bằng cách thành phần khác gọi tới stopService() hoặc chính nó gọi stopSelf().

Các giá trị trả về trong onStartCommand:

* START\_STICKY: nếu service bị hệ thống kill, nếu onStartCommand không có 1 intent nào thì nó sẽ khởi động lại với 1 intent null
* START\_NOT\_STICKY: nếu service bị hệ thống kill, thì service sẽ k được khởi động lại trừ khi có 1 intent đang chờ trong onStartCommand.
* START\_REDELIVER\_INTENT: nếu service bị hệ thống kill, nó sẽ khởi động lại với 1 intent là intent cuối cùng mà service nhận.
  1. Hàm onBind()

hệ thống gọi phương thức này bằng cách gọi hàm bindService() khi 1 thành phần muốn liên kết vs service. Khi triển khai phương thức này, phải cung cấp 1 giao diện để nó trả về 1 IBinder.

* 1. Hàm onCreate():

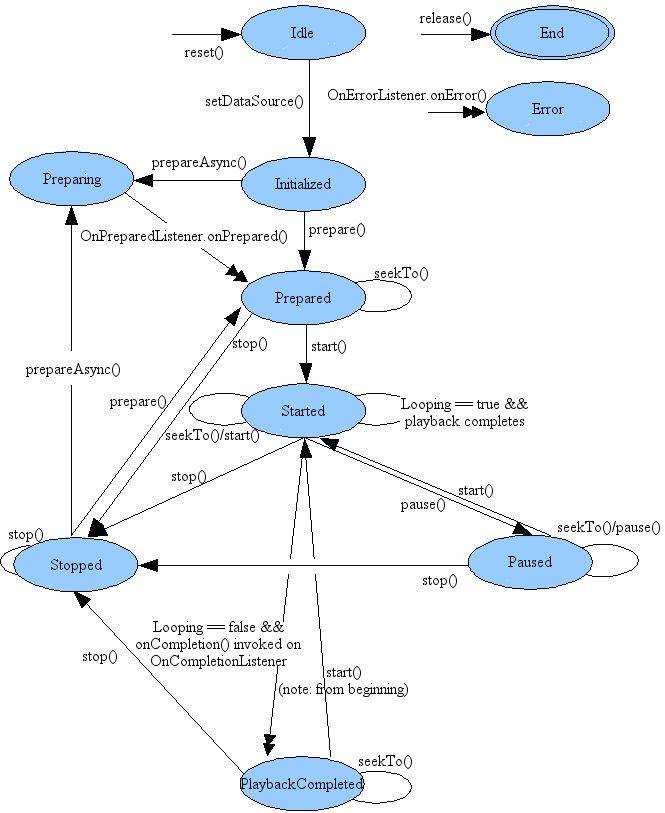
thiết lập khi service được chạy và nó chỉ chạy 1 lần trước khi gọi onStartCommand hoặc onBind. Nếu service đang chạy thì phương thức này k được gọi nữa.

* 1. onDestroy():

### MediaPlayer

Làm sao để chuyển từ byte sang nhạc

* 1. MediaPlayer là một framework hỗ trợ để play những định dạng như video, nhạc trong app của mình. Nó là một thành phần không có giao diện nhưng muốn play video thì phải kết hợp với suffaceView.
  2. VideoView là một thành phần tùy biến sẵn có của android, nó là sự kết hợp của MediaPlayer và sufaceView, giúp thực hiện video dễ dàng hơn.
  3. Các trạng thái của MediaPlayer:



* Khi một đối tượng MediaPlayer vừa được tạo bằng cách sử dụng new hoặc sau khi reset() được gọi, nó đang ở trạng thái Idle, và sau khi release() được gọi, nó đang ở trạng thái end. Giữa hai trạng thái này là vòng đời của đối tượng MediaPlayer.
* Có một sự khác biệt quan trọng giữa đối tượng MediaPlayer mới khởi tạo và đối tượng MediaPlayer sau khi reset() được gọi. Đó là lỗi khi mà bạn gọi các phương thức như là: getCurrentPosition(), getDuration(), getVideoHeight(), getVideoWidth()... ở trạng thái Idle:
  + Nếu bất kỳ phương thức nào trong số này được gọi ngay sau khi đối tượng MediaPlayer được khởi tạo, call back OnErrorListener.onError() sẽ không được gọi bởi hệ thống và trạng thái của đối tượng vẫn không thay đổi.
  + Nếu các phương thức này được gọi ngay sau khi reset(), call back OnErrorListener.onError() sẽ được gọi và đối tượng sẽ được chuyển sang trạng thái Error.
* Khi đối tượng MediaPlayer không được dùng nữa thì phải release() ngay lập tức để giải phóng tài nguyên. Trạng thái End là trạng thái Media không được dùng nữa.
* Khi đối tượng Media được tạo bằng cách sử dụng new ở trạng thái Idle, các đối tượng được tạo bởi phương thức **create** KHÔNG ở trạng thái Idle mà nó ở trạng thái **prepared**.
* Nhìn chung, 1 lỗi có thể phát sinh do bất cứ lý do nào: định dạng video không hỗ trợ, độ phân giải quá cao, timeout...và khi có lỗi nó sẽ trả lại callback OnErrorListener.onError() nếu bạn đăng ký nó setOnErrorListener(listener).
  + 1 điều quan trọng là khi media player bị lỗi thì dù bạn có đăng ký hay không đăng ký listener này, thì nó cũng chuyển sang trạng thái Error.
  + Khi ở trạng thái Error, bạn có thể gọi reset() để media player trở lại trạng thái idle.
  + Tốt nhất bạn nên đăng ký listener error để xử lý 1 cách tốt nhất.
  + Có thể nảy sinh IllegalStateException khi bạn gọi các phương thức prepare(), prepareAsync() hoặc setDataSource().
* Khi goi setDataSource(...), media player sẽ chuyển từ trạng thái idle sang initialized
  + 1 IllegalStateException sẽ nảy sinh nếu bạn gọi setDataSource ở bất kỳ 1 trạng thái nào khác ngoài idle.
  + Khi gọi phương thức này, bạn nên cẩn thận với IOException hoặc IllegalArgumentException có thể xảy ra.
* Media player phải ở trạng thái prepared trước khi start.
  + Có 2 cách(sync và async) để media player đạt được trạng thái này. Bằng cách gọi 1 trong 2 phương thức prepare() hoặc prepareAsync(). Khi phương thức prepare trả về hoặc prepare hoàn thành, nó sẽ gọi đến callback onPrepared() bằng cách đăng ký listener với setOnPreparedListener(listener).
    - Trạng thái preparing chỉ là trạng thái tạm thời, và trong lúc media preparing thì việc gọi bất kỳ phương thức nào đều là không xác định
    - 1 IllegalArgumentException sẽ nảy sinh nếu bạn gọi prepare() hoặc prepareAsync khi media ở 1 trạng thái không khả thi.
    - Trong khi media preparing thì có thể gọi đến 1 số phương thức để điều chỉnh: volume, loop, screenOnWhitePlaying.
  + Để start, bắt buộc phải gọi phương thức start(). Sau khi phương thức này trả về, media ở trạng thái started. để kiểm tra trạng thái này có thể gọi phương thức isPlaying().
    - xx
    - Gọi phương thức start() không ảnh hưởng đến đối tượng media khi nó thực sự đã ở trạng thái started.
  + Media có thể pause hoặc stop, hoặc có thể điều chỉnh vị trí play của media. Để pause, gọi phương thức pause(), sau khi phương thức này return media ở trạng thái paused. Việc chuyển từ trạng thái started sang paused và ngược lại là async.
    - Khi goi start(), media sẽ play tiếp tại vị trí trước đó nó bị paused.
    - Gọi phương thức pause() không ảnh hưởng đến đối tượng media khi nó thực sự đã ở trạng thái paused.
  + Khi ở các trạng thái: started, paused, prepared, PlaybackCompleted mà gọi stop() thì media sẽ chuyển sang trạng thái stopped.
    - Khi ở trạng thái stopped, media không thể started trừ khi gọi 1 trong 2 phương thức prepare() hoặc prepareAsync().
    - Gọi phương thức stop() không ảnh hưởng đến đối tượng media khi nó thực sự đã ở trạng thái stopped.
  + Vị trí play có thể được điều chỉnh bởi việc gọi seekTo()
    - Đây là 1 lời gọi async, mặc dù bình thường bạn thấy nó return ngay lập tức, nhưng trong môi trường network bạn nên đăng ký listener setOnSeekCompleteListener().
    - phương thức này có thể được gọi khi media ở các trạng thái khác: prepared, paused, PlaybackCompleted.
    - Có thể get vị trí hiện tại của media với getCurrentPosition()
  + Khi media kết thúc stream, nó sẽ complete.
    - Nếu media được set loop = true nó sẽ quay trở lại trạng thái started.
    - Nếu set loop = false, nó sẽ trả về ở callback onCompleteListener nếu bạn đăng ký listener. Lúc này media ở trạng thái PlaybackCompleted.
    - Trong khi media ở trạng thái PlaybackCompleted, gọi start() có thể restart lại resource.
* Sự khác nhau của prepare() và prepareAsync():
  + prepare(): sau khi gọi phương thức này thì nó sẽ ở trạng thái prepared. chạy trên thread mà bạn gọi nó, và nếu như nó mất thời gian dài để lấy dữ liệu thì nó sẽ block UI thread và sẽ gây ra ANR.
  + prepareAsync(): sau khi gọi phương thức này thì nó ở trạng thái preparing. chạy trên background thread, do vậy UI thread sẽ không bị block. Nhưng player sẽ không được prepared, do vậy bạn phải gọi onPrepareListener.onPrepare() để chắc chắn rằng player đã sẵn sàng.

1. Service có cần giao diện không. Windowmanager cần giao diện

tham khảo : https://developer.android.com/reference/android/view/WindowManager.

1. Một số so sánh giữa thread, asynctask, handler, service

* Tham khảo: <https://github.com/trantronghien/component_in_android/wiki/%C4%90a-Ti%E1%BA%BFn-Tr%C3%ACnh-Trong-Android>

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **tiêu chí** | **Thread** | **Handler** | **AsyncTask** | **Service** | **IntentService** |
| **khái niệm** | là luồng thực thi trong chương trình | liên kết, trao đổi giữa các thread với nhau |  | thành phần ứng dụng thực hiện các hoạt động lâu dài trong background mà k cung cấp UI | lớp con của Service thực thi các yêu cầu tuần tự |
| **mục đích** | -xử lý các task lâu  -xử lý song song | gửi và xử lý đối tượng Message và Runnable liên kết vs MessageQueue của Thread tạo ra nó | xử lý công việc trong background thread và giao tiếp với UI thread mà k cần dùng thread hay handler. | thực hiện các task nhỏ trong time lâu dài | xử lý long task k cần UI |
| **chạy trên** | thread riêng | main thread | worker thread | main thread | thread riêng |
|  |  |  |  |  |  |

### Service

<https://docs.google.com/document/d/1GNgsmGbZ04auVrjqm2Rri4xaCaaix8Ptg8pQ7VNF85Y/edit>

### Collections

* <https://docs.google.com/document/d/1IpedV9qdBcjJqiE0nKwYymgqeXRcOPQig_6rP9abcso/edit>

### Array và ArrayList

* Dưới 1000: Tốc độ như nhau
* Trên: Array nhanh hơn
* ArrayList chỉ truyền đc kiểu đối tượng.

### saveInstanceState: chỉ lưu biến nguyên thủy, lưu đối tượng lớn thì sử dụng gì.

* Để lưu giữ trạng thái của UI có thể sử dụng các cách sau đây:
  + ViewModel
  + SaveInstanceState
  + Persistent storage

|  |  |  |  |
| --- | --- | --- | --- |
|  | ViewModel | SaveInstanceState | Persistent storage |
| Vị trí lưu | trong bộ nhớ | serialized to disk | disk hoặc network |
| Tồn tại khi cấu hình thay đổi | Yes | Yes | Yes |
| Tồn tại khi hệ thống khởi động lại sau khi bị kill | No | Yes | Yes |
| Tồn tại khi activity complete và finish | No | No | Yes |
| Giới hạn dữ liệu lưu trữ | Lưu trữ tốt các đối tượng phức tạp, nhưng bị giới hạn bộ nhớ | Chỉ lưu trữ các giá trị nguyên thủy, đơn giản đối tượng nhỏ như String | Chỉ bị giới hạn bởi không gian của disk và giá cả cho việc lưu trữ qua network |
| Thời gian đọc và ghi | Nhanh (chỉ truy cập bộ nhớ) | chậm (yêu cầu serialization/deserialization và truy cập disk) | chậm (yêu cầu truy cập disk và các transaction trên network) |

* x

### Thread pool

* dùng để giới hạn số lượng thread được thực thi trong ứng dụng tại cùng 1 thời điểm. Nếu không có sự giới hạn này thì cứ mỗi lần thread được tạo mới thì nó sẽ chạy -> hiệu năng của hệ thống.
* …
* VD: download 100 file: tạo 10 thread mỗi thread download 10 file.

### Observer pattern

* Observer pattern định nghĩa 1 hoặc nhiều sự phụ thuộc giữa các object. Nếu 1 object thay đổi trạng thái thì các object khác cũng tự động thay đổi theo.

### Iterator pattern

- Iterator pattern cung cấp 1 cách truy cập các phần tử của collection object 1 cách tuần tự mà không cần biết cấu trúc của object.

### 7 Core value

* 1. Appreciate Teamwork
  2. Think outside the box
  3. Have a guts to challenge
  4. Think positive
  5. Speed up
  6. Be professional
  7. Focus on the point

### Jetpack

* Jetpack là 1 tập hợp các component, thư viện của Android giúp cho developer phát triển ứng dụng một cách đơn giản hơn.
* Jetpack bao gồm gói thư viện androidx.\* riêng biệt không được gộp vào các APIs cơ bản của android.
* Nó được chia làm 4 nhóm chính:
  + Foundation: cung cấp những phần core của hệ thống, Kotlin extensions, multidex và auto testing
    - AppCompat
    - Android KTX
    - MultiDEX
    - Testing
  + Architecture: chứa các lớp giúp quản lý lifecycle của các thành phần UI, xử lý dữ liệu liên tục…
    - Room: Là một Persistence Library, nó là một abstract layer cung cấp cách thức truy cập dữ liệu trong cơ sở dữ liệu SQLite.
    - LiveData: Thông báo với View khi mà cơ sở dữ liệu thay đổi.
    - LifeCycle: Quản lý vòng đời của Activity và Fragment
    - ViewModel: là một class được thiết kế để lưu trữ và quản lý dữ liệu trong một lifecycle riêng, nó cho phép dữ liệu được bảo toàn ngay cả khi màn hình bị xoay.
    - Data Binding: Khai báo liên kết dữ liệu quan sát được với các thành phần UI.
    - WorkManager: Là một thư viện được thiết kế cho việc lập lịch và quản lý các tác vụ ngầm.
  + Behavior: hỗ trợ việc thiết kế, kiểm thử, bảo trì app
    - Download Manager
    - Media & Playback
    - Notifications
    - Permission
    - Sharing
    - Slices
  + UI: dễ dàng tạo 1 ứng dụng đơn giản nhưng kém phần thú vị cho người dùng.
    - Animation
    - Layout
    - Fragment
* giúp bạn tạo được 1 ứng dụng Android tuyệt vời 1 cách dễ dàng và nhanh chóng.

### Retrofit

* Retrofit sử dụng network layer của nó để cache từ OKHttp-có khả năng cache header response
* Cache thường được lưu vào internal storage của ứng dụng
* Các bước làm việc với retrofit:
  + Map object với json data
  + Định nghĩa API interface
  + Xây dựng builder retrofit gồm các url + endpoint và các hoạt động liên quan.

### SQL Transaction

* Là 1 nhóm các câu lệnh SQL. Nếu 1 transaction được thực hiện thành công, tất cả các thay đổi dữ liệu được thực hiện trong transaction sẽ được lưu vào cơ sở dữ liệu. Nếu 1 transaction bị lỗi và được rollback, thì tất cả dữ liệu sẽ được khôi phục lại thời điểm trước khi thực hiện transaction.
* Đặc điểm:
  + Bảo toàn
  + Nhất quán
  + Độc lập
  + Bền bỉ
* Câu lệnh:
  + COMMIT: lưu các thay đổi
  + ROLLBACK: khôi phục lại các thay đổi
  + SAVEPOINT: tạo ra các điểm trong transaction để rollback
  + SET TRANSACTION: thiết lập các thuộc tính cho transaction

### Value Type and Reference type

* kiểu dữ liệu đc gọi là value type nếu nó chứa một giá trị trong không gian ô nhớ của chính nó, hay nói cách khác các biến của kiểu dữ liệu này trực tiếp chứa giá trị của chúng.
* reference value : lưu địa chỉ nơi giá trị đang được lưu trữ, hay nói cách khác reference chứa một con trỏ nó sẽ trỏ tới vị trí bộ nhớ khác chứa dữ liệu.

### Câu hỏi phỏng vấn anh Kiên(div2)



* Bản chất của databinding là cho phép bạn đặt các biểu thức expression vào trong XML.

**Ngày 20/11**

- 4 tính chất của OOP

- Tính đóng gói thể hiện như thế nào?

- Phạm vi của protected

- Tính đa hình thể hiện như thế nào?

- Overload và override?

- Bộ nhớ heap và stack

- Abstract và interface

- 4 component của Android

- Vòng đời activity

- Sử dụng gì để chuyển activity

- Service em biết gì

- Load dữ liệu ở Project 1 em dùng gì

- AsyncTask là gì

- Thread là gì

- Handler là gì

- Intent có mấy loại

- Intent tường minh là như thế nào

- Intent không tường mình là như thế nào

- Nếu gọi 1 Intent không tường minh mà không có thành phần nào xử lý thì sẽ xảy ra tình huống gì => exception

- Activity với Context khác gì nhau.

- Đã bao giờ gặp lỗi OutofMemory chưa

- Tại sao load ảnh với bitmap lại bị OutofMemory

- Tại sao load ảnh với thư viện lại tránh được OutofMemory

- Biến static và biến thông thường khác nhau như thế nào

- Biến static được khởi tạo và giải phóng như thế nào

- Tại sao lại sử dụng MVP

- Đã bao giờ làm việc với unittest với android chưa

- MediaPlayer có play được video không

- Muốn playvideo thì sử dụng gì

**Ngày 21/11**

- 4 tính chất của OOP

- Interface vs Abstract class

- static vs non static

- Bộ nhớ trong java, hủy bỏ đối tượng, thu gom rác?

- Lưu trạng thái Activity?

- SQL?: ORDER BY?, join bảng?

- Activity phải Context không?

- MVP, so sánh MVP vs MVVM?, Data Binding trong MVVM?

- Glide: load ảnh từ 1 url ntn?, Glide vs Picasso?

**Ngày 22/11**

Em bắt đầu với Android ntn? Gặp những khó khăn gì?

Tài liệu chủ yếu của em là gì?

GridLayout, TableLayout, so sánh?

GridLayout có dynamic đc ko?

Constraint với Relative?

scale ảnh gồm những loại nào?

So sánh centerCrop, centerInside

Get ảnh Url dùng gì?

So sánh Glide vs Picasso

Glide cache ảnh ntn? Resize ảnh ntn? Muốn load kích thước chuẩn của ảnh bằng glide ntn?

Diskcache là gì?

Internal storage là gì? Xoá được không?

Các cách lưu trữ dữ liệu local

Shared preferences

Cấu hình sqlite ntn? Xử lý khoá chính ntn? Nối bảng ntn? Khác biệt leftjoin vs inner join?

Heap vs stack, Giải phóng đối tượng ntn?

4 component

Vòng đời activity

Service

foreground, background là gì?

### Câu hỏi phỏng vấn anh Hoài - anh Định (div3)

- Khi xoay màn hình, làm thế nào để Activity không chạy lại vào onCreate()? set thuộc tính configChange cho activity trong Manifest.xml

- Khi gọi notifyDataSetChanged() trong Adapter của RecyclerView thì sẽ hoạt động như thế nào? tương tự với các method: notifyItemInserted(), notifyItemRemoved(), ..... Nó sẽ duyệt lại tất cả data rồi set lại cho các item trong list, nó sẽ vào lại hàm onBindViewHolder() cho tất cả item.

- Trong mô hình MVP, Presenter có chứa context không? Không, presenter không được chứa context, nếu muốn sử dụng context, hãy reference nó từ View

- Trong mô hình MVP, V ở đây là những thành phần nào? Activity, Fragment.

- Trong mô hình MVVM, không code logic trong layout có sao không?

- Rx sinh ra để làm gì?

- Base network của Retrofit là gì? OkHttp

- Architecture gồm những thành phần nào?

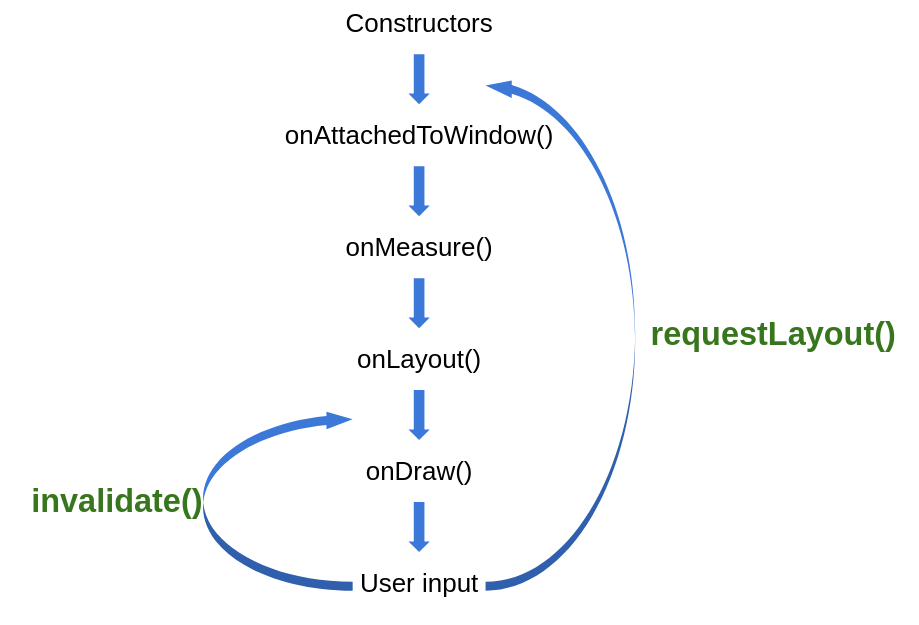
- ViewModel dùng để làm gì?

- Coding convention em thấy như vậy là đã đầy đủ chưa?

- Em có ý kiến gì về convention nào không?

- Em đã dùng convention của IDE bao giờ chưa?

1. Vòng đời của View - View Lifecycle



* Constructors: tạo form cho view và inflate view từ file layout.
* onAttachedToWindow(): được gọi khi view được gắn vào một cửa sổ.
* onMeasure(): được gọi khi xác định các yêu cầu về kích thước cho View này và tất cả các view con của nó.
* onLayout(): được gọi khi view chỉ định kích thước và vị trí cho tất cả các con của nó.
* onDraw(): được gọi khi view hiển thị nội dung của nó.

### Sự khác nhau giữa @+id và @id

- @+id: sử dụng khi muốn thêm mới 1 resource integer vào file R.java nếu nó chưa tồn tại.

- @id: sử dụng khi muốn tham chiếu (refer) đến 1 resource đã tồn tại trong R.java

Bình thường, tất cả đều sử dụng @+id cũng không có gì sai cả.

1. RecyclerView

<https://docs.google.com/document/d/1UwL-PSbu4b3QwD_ZtHvizPipvBDGJDf6AEP8AAPBXD4/edit#>

1. Sự khác nhau giữa Vector và ArrayList

<https://stackoverflow.com/questions/2986296/what-are-the-differences-between-arraylist-and-vector>

<https://www.erpgreat.com/java/difference-between-arraylist-and-vector.htm>

1. x
2. x
3. x
4. x
5. x
6. x