

AndroidManifest.xml

Giao tiếp giữa app và hđh

Android App

Res/menu: \*.xml

Chứa các file giao diện menu

java : \*.java

Thư mục chứa các file nguồn, file Activity

Res/layout :\*.xml

Chứa các file giao diện

Res/values

Chứa các tập mã nguồn

Res/mipmap

Chứa ảnh

Yêu cầu cài font trong cái thư mục font nhé, cho đẹp màu mè tí.

Content Providers

Quản lý data, database

Broadcast Receivers

Giao tiếp giữa app và android

Services

Dịch vụ chạy ngầm của app

# I. Ánh xạ cho giống với web tí nhé:

1. **AndroidManifest** ⬄ Web.config
2. **Java** :

* \*Activity.java ⬄ Controller
* \*.java ⬄ Các class, code khác vv

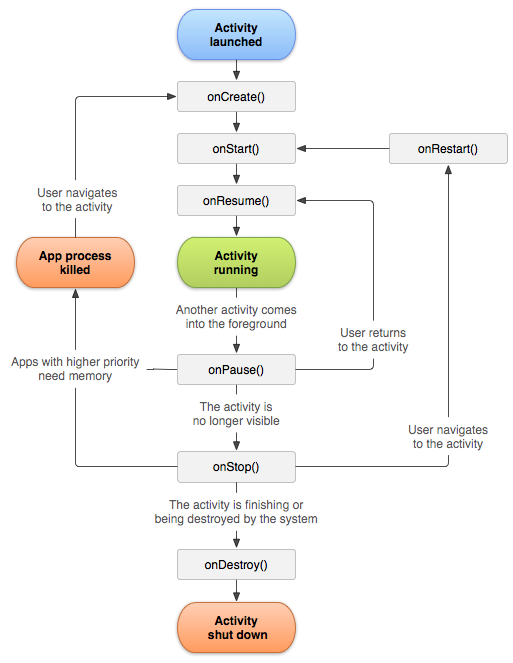
1. **Layout** ⬄ View
2. **Menu** ⬄ View
3. **Mimmap** ⬄ Content (chứa image)
4. **Values** ⬄ Style.css

# II. Vòng đời của Activity:

Một **Activity** khi tạo ra là một class và phải thừa kế những gì nó có nó để tạo ra giao diện sử dụng nó. Mỗi activity sẽ có 1 vòng đời từ khi bắt đầu đến khi kết thúc.

Mỗi quá trình của nó cần xem để code vào từng quá trình một cách hợp lý cho mỗi mục đích ứng dụng.

Tạm thời lấy hình internet để khỏi vẽ lại



Khi tạo ra theo mặc định có file **MainActivity**.**java** và file **activity**\_**main**.**xml** tương ứng là layout của nó.

Khi chạy tới **MainActivity**.**java**, activity này là một đối tượng, và đã có sự kiện **onCreate**() của chính nó. Để **MainActivity**.**java** khi khởi tạo có thể trả về một màn hình như đã thiết kế trong **activity**\_**main**.**xml** thì phải ghi đè lên hàm onCreate, sẽ dùng **@Override**

Đây là code:

@Override  
protected void onCreate(Bundle saveInstanceState){

super.onCreate(saveInstanceState);  
setContentView(R.layout.activity\_main);

}

Có thể thấy là code trong sự kiện **onCreate** đã chạy lệnh tạo một **ContentView** (giao diện đi cho dễ hiểu) là từ file **activity\_main.xml**. Như vậy có thể hiểu là, không nhất thiết **MainActivity.java** phải có file **activity\_main.xml** mới chạy được mà có thể có một file **abc.xml** chúng ta cũng có thể **setContentView(R.layout.abc)** hoặc cũng có thể viết code tạo giao diện trong chính **MainActivity.java**!

* Tạo giao diện trên file xml, code thao tác trên file java. Đặt tên 2 file đó giống nhau đi cho dễ quản lý. Đừng đặt A.java mà gọi tới B.xml chi cho khổ.
* Khuyến mại cuốn sách CleanCode, mọi người cùng đọc, code thì nên đọc, sắp code cũng nên đọc, quản lý cũng nên đọc mà không code cũng nên đọc, rất có ích cho việc đặt tên con cái sau này.  
  **Ấn vô, không phải cái hình đâu!**

\* Nói thêm về việc khi tạo ra một Activity:

Khi tạo ra nó thì cái view, cái layout .xml đấy nó đã được **đăng ký** vào trong file **AndroidManifest.xml**, chưa nói tới những cái khác trong AndroidManifest nó làm thì ví dụ tạo ra 1 Activity là **Sanpham.java** và **DanhSachSanPham.xml** thì trong **AndroidManifest** sẽ có <activity android:name=”.DanhSachSanPham” /> … thử xóa nó xem, chạy tới cái nào gọi DanhSachSanPham là lỗi liền.

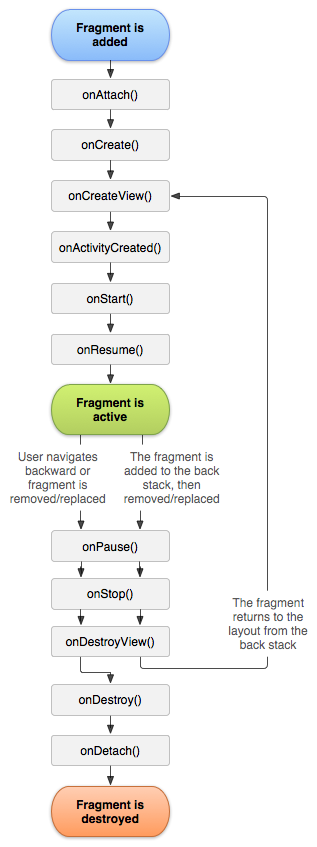
# III. Fragment:

**Fragment** nó cũng giống như **Usercontrol, PartialView** vv đại loại thế và nó có thể sử dụng trong một Activity, đọc tài liệu thì người ta viết là “có thể hiểu là sub-activity” => nó có thể sử dụng như một Activity!

Copy-Paste:

* Fragment cũng có layout của riêng của nó, cũng có các hành vi và vòng đời riêng.
* Chúng ta có thể thêm hoặc xóa Fragment trong một Activity trong khi Activity này đang chạy.
* Có thể kết hợp nhiều Fragment trong một Activity để xây dựng giao diện người dùng đa khung.
* Một Fragment có thể được sử dụng trong nhiều Activities.
* Vòng đời của Fragment có quan hệ chặt chẽ với vòng đời của Activity đang dùng nó điều này có nghĩa là khi Activity bị tạm dừng thì các Fragment sẽ dừng lại.
* Fragment có thể thực hiện một hành vi mà không có trong thành phần giao diện người dùng.
* Fragment được thêm vào API 11 trở lên.
* Bạn có thể tạo các Fragments bằng cách kế thừa lớp Fragment và Fragment được thêm vào layout bởi thẻ <fragment>

Thêm cái ảnh vòng đời của một Fragment vào, khá dài :



Về cách gọi Fragment trong code sẽ thực hiện như sau.

Có 2 file **FragmentImageContainer**.**java** và **fragment\_image\_container.xml**, trong đó file **FragmentImageContainer**.**java** sẽ trả về **fragment view fragment\_image\_container.xml**.

Từ một Activity để gọi fragment\_image\_container ta code như sau trong Activity:

//Khởi tạo biến frag là class FragmentImageContainer

final Fragment **frag** = **new** **FragmentImageContainer**();

//Có thể tạo một bundle để đưa dữ liệu sang fragment nếu cần thiết

Bundle **bundle** = new Bundle();

bundle.**putString**("**charName**", **charname**);

frag.**setArguments**(bundle);

FragmentManager **fragmentManager** = **getFragmentManager**();

FragmentTransaction fragmentTransaction = fragmentManager.**beginTransaction**();

fragmentTransaction.**replace**(R.id.**flImage**, **frag**);

fragmentTransaction.**addToBackStack**(**null**);

fragmentTransaction.**commit**();

Chú ý để có thể đổ fragment vào layout thì cần sử dụng **FragmentManager** , **FragmentTransaction**, vậy **FragmentTransaction** là gì, các phương thức để sử dụng nó như thế nào?

Trước tiên phải có một đối tượng quản lý Fragment là **FragmentManager**. Sau đó để thêm, thay đổi, xóa … fragment thì sẽ tạo đối tượng **FragmentTransaction** từ **FragmentManager**.

Những phương thức chủ yếu để sử dụng trong **FragmentTransaction** là **add**, **replace**, **remove** và **commit**. Như ví dụ trên đã thấy một đối tượng được đặt tên là fragmentTransaction, fragmentTransaction sẽ thực hiện việc thay thế dữ liệu có trong đối tượng FrameLayout đã được tạo sẵn trong Activity có Id là flImage bằng đối tượng frag đã tạo.

Ở đây, phương thức replace đã làm 2 việc là xóa (remove) fragment cũ trong layout chỉ định đó (nếu có) sau đó thêm (add) fragment mới vào.

Sau khi thực hiện xong các thao tác, phải sử dụng phương thức commit để các thay đổi có hiệu lực.

## **\* Sự kiện khi ấn nút BACK đối với fragment!**

Ví dụ từ Activity1 gọi Fragment1, sau đó gọi Fragment2 -> FragmentN

Activity1 => Fragment1 => Fragment2 => … => FragmentN to infinity and beyond!

Khi ấn nút back thì sẽ có 2 trường hợp như sau:

1. Từ FragmentN sẽ trở về Activity1
2. Từ FragmentN sẽ trở về Fragment được gọi trước đó

Với **trường hợp a**, sẽ không cần phải viết thêm code gì vào ví dụ trên.

Với **trường hợp b**, sẽ phải viết thêm 1 đoạn code liên quan tới việc ấn nút **BACK** và đối tượng **FragmentManager**.

Chú ý trong đoạn code trên có dòng

fragmentTransaction.**addToBackStack**(**null**);

Nếu muốn ấn nút BACK có thể quay trở về Fragment trước đó thì cần phải sử dụng phương thức addToBackStack. Đối số truyền vào là 1 chuỗi để định tên, thường là null. Nếu để null thì tất cả Fragment trong đoạn code đó sẽ được đưa vào BackStack. Sau đó sẽ Override lại hàm OnBackPress trong Activity:

@Override  
public void onBackPressed() {

if (getFragmentManager().**getBackStackEntryCount**() == 0) {

this.**finish**();

} else {

getFragmentManager().**popBackStack**();

}

}

Trước tiên sẽ xem trong **BackStack** có **Fragment** nào được lưu trong đó không, nếu như không có sẽ kết thúc câu lệnh này, trong đó **finish()** là hàm được gọi khi Activity này đã thực hiện xong và được đóng lại để quay về Activity nào đó trước đó. Ngược lại nếu như có Fragment trong **BackStack** thì sẽ quay về Fragment trước đó bằng **popBackStack**();

Đó là với các Fragment, còn nếu trong Fragment có webview để thể hiện trang web, muốn để nút BACK làm chức năng quay lại lịch sử duyệt web trước đó, thì có thể viết code như sau bằng cách Override hàm onBackPressed:

@Override  
public void onBackPressed() {

WebView mWebView = **new** **WebView**(this);

if (mWebView.**canGoBack**()) {

mWebView.**goBack**();

return;

}

super.**onBackPressed**();

}

## \* popBackStack, popBackStackImmediately, getBackStackEntryAt:

Để quay lại các Fragment trước đó thì trong code đã dùng thằng **popBackStack**(). Vậy nó làm gì? Sao addToBackStack lại đưa vào null? Đưa vào name addToBackStack(“F1”) thì sao?

Ví dụ 1 BackStack như sau:

**BackStack** [ Fragment1, Fragment2, Fragment1, Fragment3, Fragment2 …. FragmentN ]

Trong đó FragmentX có name lần lượt là name = “Fx”, ví dụ Fx1, Fx2 … FxN. Và index tăng dần từ 0 -> N-1.

Cơ bản thì cả 3 phương thức trên đều đưa về Fragment trước đó trong đó

popBackStack, popBackStackImmediately có 3 trường hợp sau:

1. **popBackStack**() : Mặc định đưa về Fragment trước đó
2. **popBackStack**(**index**, FragmentManager.POP\_BACK\_STACK\_INCLUSIVE ):  
   Đưa về Fragment ở vị trí index
3. **popBackStack**(**name**, FragmentManager.POP\_BACK\_STACK\_INCLUSIVE ):  
   Đưa về Fragment với tag name

**popBackStack** và **popBackStackImmediately** đều là phương thức trả về kiểu void, khác nhau là **popBackStackImmediately** khởi chạy ngay lập tức khi được gọi thôi.

Đối với getBackStackEntryAt thì chỉ có truyền vào đó một index, trả về BackStackEntry tại vị trí thứ index.

BackStackEntry fragEntry = getFragmentManager().**getBackStackEntryAt**(**index**);

Vậy tùy vào từng trường hợp muốn thao tác ra sao, mục đích thế nào mà sẽ chọn cho phù hợp thôi.

## **\* Sự kiện khi có sự thay đổi trong BackStack:**

Như trên, để thao tác trong việc ấn nút **BACK** giữa các **Fragments** trong **BackStack** thì trước tiên phải add **Fragment** vào trong **BackStack**. Vậy để mỗi lần có sự thay đổi trong BackStack như thêm, xóa Fragment trong BackStack hay muốn rollback (quay lại trạng thái trước đó) thì sẽ dùng phương thức **OnBackStackChangedListener** – tạm dịch : “Lắng nghe sự đổi thay“ - 2016 :v

fragmentManager.**addOnBackStackChangedListener**(**new** FragmentManager.**OnBackStackChangedListener**() {

@Override  
public void **onBackStackChanged**() {

//Please code something here

//like this getFragmentManager().**popBackStack**();

//or this int ten = 10;

}

});

## **\* Tương tác giữa Fragment và Activity**

Để đưa dữ liệu từ **Activity** sang **Fragment** trong đoạn code trên sử dụng **Bundle** và fragment.**setArguments**(bundle) đó để sang Fragment chỉ cần gọi **getArguments()** là có thể sử dụng được.

Vậy ngược lại khi thao tác trên **Fragment**, làm sao để dữ liệu có thể tương tác lại với những thành phần có trên **Activity**?

Tiện cho sử dụng, khai báo một biến activity trên code của **Fragment** đó:

Activity **view** = **getActivity**();

Sau đó có thể dùng biến này trong **Fragment** để thao tác với các thành phần trên **Activity** đó như bình thường.

**view**.findViewById(R.id.textView1).setText(“avada kedavra”);

# IV. Intent, Shared Preferences

Sau khi tạo ra MainActivity, SecondActivity, ThirdActivity vv… Giờ muốn tương tác với nhau, trao đổi dữ liệu qua lại với nhau thì sao?

Chưa tìm hiểu nhiều là có cách nào để có thể truyền dữ liệu từ Activity này sang Activity khác để tương tác với nhau một cách hiệu quả và nhanh chóng, không mất công phải xuống database ví dụ như khi ấn vào Nút A ở Trang1 sẽ chạy Trang2 và có dữ liệu từ Trang1. Thì đã xem qua về 2 thằng là Intent và Shared Preferences, 2 thằng này khác nhau và sử dụng khác nhau luôn.

* Intent: Là một tin nhắn được gửi boardcast trong app của chúng ta hay với những thành phần trong hệ điều hành, background app, service vv.
* SharedPreferences: Là một kiểu lưu trữ dữ liệu theo dạng key, value. Nếu xem thì cũng na ná như cách dùng Session, Cookies vậy.

## Intent:

Intent có **2 loại Intent**. Một cái là **tường minh (explicit intent)** là nó sẽ được chỉ định rõ ràng các mục tiêu mà nó gửi dữ liệu còn cái còn lại là **không tường minh (implicit intent)** nó sẽ không chỉ định rõ mục tiêu mà nó gửi dữ liệu tới.

Để cho dễ hiểu, trong câu hỏi đặt ra ở đầu khi muốn gửi một tin nhắn chứa dữ liệu từ MainActivity sang SecondActivity thì đã xác định mục tiêu gửi dữ liệu, sẽ dùng Intent tường minh còn không tường minh ví dụ khi ấn vào nút Gửi mail chẳng hạn, nó sẽ đưa dữ liệu là nội dung email nhưng mục tiêu của nó không xác định, có thể là dùng Gmail, Outlook vv và sẽ hiện lên cho mình chọn.

Chưa nghiên cứu nhiều về cái không tường minh, còn cái tường minh thì để sử dụng thì làm như sau:

Trên **MainActivity.java**

//Intent tenbien = new Intent(MainActivity, SecondActivity);

Intent sendIntent = **new** Intent(**MainActivity**.**this**, **SecondActivity**.**class**);

Intent này chưa có bất cứ dữ liệu nào, để truyền dữ liệu vào nó thì trước tiên chúng ta sẽ biết là Intent có thể đưa dữ liệu vào theo kiểu **key, value** hoặc một **Bundle** (cứ hiểu như ngày xưa là Session chứa được dữ liệu kiểu int, boolean hoặc một Datatable :v )

String ten = “Park Chym Sung”

int tuoi = 18;

String gioiTinh = “FyMeu”;

String ngaySinh = “01/02/2012”;

sendIntent.**putExtra**(“HoTen”, ten);

sendIntent.**putExtra**(“Tuoi”, tuoi);

Bundle duLieuCuaDoiTuong = **new** Bundle();

duLieuCuaDoiTuong.putString(“NgaySinh”, ngaySinh);

duLieuCuaDoiTuong.putString(“GioiTinh”, gioiTinh);

sendIntent.**putExtra**(duLieuCuaDoiTuong);

Để khởi chạy cái intent này thì gọi như sau:

MainActivity.this.**startActivity**(sendIntent);

Trên SecondActivity.java, là activity mà ta gửi dữ liệu tới

Intent getIntent = **getIntent**();

String ten = getIntent.**getStringExtra**(“HoTen”);

int tuoi = getIntent.**getIntExtra**(“Tuoi”);

Bundle duLieuTrongCaiBundle = **getIntent**().**getExtras**();

String ngaySinh = duLieuTrongCaiBundle.**getString**("NgaySinh ");

String gioiTinh = duLieuTrongCaiBundle.**getString**("GioiTinh ");

Còn về Intent không tường minh thì chỉ biết cơ bản như sau vì chưa viết nhiều liên quan tới nó:

Intent intent = **new** Intent(**Intent**.**ACTION**\_VIEW, Uri.parse(“http://thi\*\*\*\*\*.com”));

MainActivity.this.**startActivity**(intent);

Thì trong cái sự kiện gọi hàm chứa code trên nó sẽ chạy đến cái app nào sử dụng cái url được, ở đây là một trình duyệt web mặc định có sẵn, nếu xài 2 cái app duyệt web thì nó sẽ hỏi.

## SharedPreferences:

Trước khi sử dụng cần hiểu thêm 1 chút về cái **SharedPreferences** này, ngay từ cái tên của nó là nghe cũng hiểu được đôi phần là gì rồi. Vậy sao cần dùng nó?

Nếu như một dữ liệu gì đó như thông tin người dùng, sản phẩm, nhân vật cần lưu xuống database thì một số như tổng 2 số, đơn giá vv thì chỉ cần dùng biến tạm tính toán ngay lúc cần là được rồi. Nhưng có một số biến cần phải lưu lâu dài, kể cả khi tắt app đi rồi mở lên vẫn còn lưu trữ và cũng không nhất thiết phải lưu xuống database như nhớ đăng nhập (remember me :v ), username, vài cái cấu hình game như độ sáng, âm lượng vv thì chỉ cần lưu vào **SharedPreferences** là được rồi và khác **Intent** khi có thể đưa một số kiểu dữ liệu phức tạp thì **SharedPreferences** chỉ lưu những dữ liệu đơn giản thôi.

Cũng đơn giản, ví dụ một cái cấu hình trò chơi đi cho nhiều biến lưu, cách sử dụng như sau:

//SharedPreferences tenbienshare = getSharedPreferences(TenPreferences, Mode);

SharedPreferences shareData = getSharedPreferences(“CAUHINH”, Context.MODE\_PRIVATE);

Hoặc:

SharedPreferences shareData = getSharedPreferences(“CAUHINH”, MODE\_PRIVATE);

Ở đây **Mode** là các mode lưu trữ của SharedPreferences, thường là để MODE\_PRIVATE cho bảo mật, ngoài ra còn nhiều mode khác mà muốn thì phải tìm hiểu từng mode 1 thôi.

Tên của cái cấu hình này sẽ đặt tạm là **CAUHINH** đi.

Để **đưa dữ liệu vào** nó, phải qua một cái **Editor**:

SharedPreferences.**Editor** editor = shareData.**edit**();

editor.**putString**(“TenNguoiChoi”, “HuongHana”);

editor.**putInt**(“Tuoi”, 20);

editor.**putInt**(“DoSang”, 90);

editor.**putBoolean**(“TrongTrang”, false);

editor.**commit**(); /\*hoặc dùng \*/ editor.**apply**();

Có thể thấy là ban đầu phải tạo cái editor, sau đó đưa dữ liệu vào rồi xác nhập lưu những dữ liệu theo các khóa (key : TenNguoiChoi, Tuoi, DoSang …) đó bằng commit hoặc apply. Vậy commit và apply khác nhau điều gì?

**apply**() được thêm vào từ bản 2.3, nó cũng là commit nhưng không có dữ liệu trả về.

**commit**() là thực hiện lưu dữ liệu nhưng có kết quả trả về là true hoặc false;

Một cái là void còn cái kia là return value, có thể xài commit như sau:

boolean luuThanhCong1 = editor.commit(); //trả về true hoặc false

boolean luuThanhCong2 = editor.apply(); //Báo lỗi

Theo tìm hiểu thì chả ai xài cái biến trả về cả nên nhà phát triển cho thêm cái apply vô xài cho lẹ.

Để **sử dụng dữ liệu** từ SharedPreferences làm như sau:

SharedPreferences getCauHinh = **getSharedPreferences**(“CAUHINH” , Context.MODE\_PRIVATE);

String ten = getCauHinh.**getString**(“TenNguoiChoi”, “”);

String tuoi = getCauHinh.**getInt**(“Tuoi”, 0);

String doSang = getCauHinh. **getInt** (“DoSang”, 0);

String trongTrang = getCauHinh.**getBoolean**(“TrongTrang”, false);

Trong các hàm **getXXX** đó thì cái đối số thứ nhất là key còn đối số thứ hai là dữ liệu mặc định nếu như key đó không có dữ liệu.

Để có thể **clear** dữ liệu trong một SharedPreferences nào đó thì làm như sau

SharedPreferences getCauHinh = getSharedPreferences(“CAUHINH” , Context.MODE\_PRIVATE);

SharedPreferences.**Editor** editor = shareData.**edit**();

editor.**clear**();

editor.**commit**();

## \* Mẹo:

Vì tính chất của SharedPreferences nên dùng lưu mấy cái cấu hình trong setting của app, cấu hình profile sẽ tiện hơn nhiều so với lưu xuống database, đỡ mất một bảng, và xài chủ yếu dữ liệu thô thôi mà.

Trong một vòng đời của Activity có **onPause** và **onResume** vậy 2 cái sự kiện này nó chạy thế nào?

a) Mở Activity mới lên làm che khuất toàn bộ Activity cũ (không nhìn thấy Activity cũ): sảy ra sự kiện **onPause** rồi **onStop** đối với Activity cũ

b) Mở Activity mới lên làm che khuất một phần Activity cũ (vẫn nhìn thấy Activity cũ): Sảy ra sự kiện **onPause** với Activity cũ.

– Khi quay trở về Activity cũ thì sau khi thực hiện xong các hàm cần thiết, chắc chắn nó phải gọi hàm **onResume** để phục hồi lại trạng thái ứng dụng

– Như vậy ta thường lưu lại trạng thái của ứng dụng trong sự kiện **onPause** và đọc lại trạng thái ứng dụng trong sự kiện **onResume**

# V. Tạo thông báo Toast

Toast là cái thông báo hiện trên màn hình của app ấy. Gọi cũng dễ, cứ như là msgbox.show ấy.

Toast.makeText(getApplicationContext(), “Lời nhắn ghi vô đây” ,Toast.LENGTH\_SHORT).show();

Trong đó đối số thứ 3 là thời gian hiển thị, thì có 2 loại là **Toast.LENGTH\_SHORT** và **Toast.LENGTH\_LONG**.

# VI. Database sử dụng SQLite

Trước khi đi vào chi tiết thì để có thể tương tác giữa app và database thì có nhiều cách và nhiều kỹ thuật code khác nhau để xử lý. Trước khi một project, app được viết thì phải có sự phân tích và chọn lựa 1 phương pháp code với database thích hợp rồi mới bắt đầu làm được.

Trong kỹ thuật code hiện có **3** mô hình làm việc như sau là:

* Code First
* Database First
* Model First

Cả 3 cách đều là những mô hình để có thể làm việc với database, đều có ưu nhược điểm riêng và chọn vào từng trường hợp làm việc khác nhau.

Chi tiết hơn về từng mô hình thì **Code First** nghĩa là **code trước**, **database để sau**; **Database First** là làm **database trước**, **code viết sau**; **Model First** là làm **mô hình database trước** sau đó **database sẽ tự tạo ra sau** và sau đó là công việc code.

Nếu chưa có database thì có thể dùng code first để viết ra cấu trúc database, định dạng dữ liệu vv rồi code sẽ sinh ra database hoặc dùng database first, tức là tạo ra database sau đó sẽ map những trường dữ liệu của database, chuỗi kết nối vào trong code để làm việc. Ở đây sẽ nói về 2 phương pháp này là chủ yếu!

Hiện tại code đã viết là tìm hiểu, làm theo **hướng dẫn là về code first**, trong code sẽ khai báo tên database tên bảng, cột, trường dữ liệu sau đó code sẽ tự sinh database để làm việc sau này.

Trong project đã có 1 class tên **MyDatabase.java** chứa những câu lệnh liên quan tới database như :

* Khai báo những thông tin cần thiết của database (tên, version, bảng, cột, kiểu)
* Tạo database (nếu chưa có database)
* Đóng/Mở kết nối tới database
* Các hàm thêm sửa xóa lấy

Trong code đã có hết rồi có thể chạy, debug xem thử kết quả ra sao và vì code cũng dài nên sẽ tập trung vào những ý chính:

Trước tiên là phải khai báo những thông tin của database (nếu thích). Khai báo thì giúp để quản lý một cách dễ dàng hơn thôi ví dụ 2 kiểu như sau:

**Cách 1:**

public OpenHelper(Context context) {

super(context, “DB\_MASSAGE”);

}

**Cách 2:**

String tenDatabase = “DB\_MASSAGE”;

public OpenHelper(Context context) {

super(context, tenDatabase);

}

Với cách 1 thì khi nào chỉ có duy nhất 1 hàm này sử dụng hoặc 2, 3 chỗ code cần và ta biết chắc chắn lúc nào nó cũng là “DB\_MASSAGE” thì có thể “tạm” sử dụng nếu như dùng ở 100 chỗ thì khi nào cái tên DB\_MASSAGE phải thay đổi thì rất mệt trong việc sửa chữa (CTRL + H). Còn với cách 2 thì khi nào muốn thay đổi tên database thì đơn giản chỉ là sửa giá trị của biến tenDatabase thành giá trị khác là xong.

**Bước 1: Khai báo những giá trị cần thiết của database mới**

//Tên database

private static final String **DATABASE\_NAME** = "**DB\_HOCANDROID**";

//Version của database

private static final int **DATABASE\_VERSION** = **1**;

//Tên table

private static final String **TABLE\_NOTE** = "**TB\_NOTE**";

//Tên các cột và kiểu dữ liệu của từng cột

public static final String **COLUMN\_ID** = "**\_id**";

public static final String **COLUMN\_NOTE** = "**note**";

**Bước 2: Khởi tạo database**

Để tạo database bằng SQLite thì phải sử dụng các thư viện của SQLite rồi. Khi tạo ra class MyDatabase thì code của nó sẽ như sau:

public class MyDatabase {}

Hoàn toàn có thể extend class này để viết những code liên quan tới việc khởi tạo database bằng cách

public class **MyDatabase** **extends** **SQLiteOpenHelper** {}

Nhưng trong code demo thì thay vì extend class MyDatabase thì có thể tạo ra 1 class con bên trong MyDatabase để làm việc đó

//Trong class MyDatabase có class riêng chỉ liên quan tới việc khai báo, khởi tạo database  
//còn lại thì là những hàm để sử dụng database được viết riêng

private static class **OpenHelper** **extends** **SQLiteOpenHelper** {

/\*Hàm dựng khởi tạo 1 OpenHelper\*/

public **OpenHelper**(Context context) {

**super**(context, DATABASE\_NAME, null, DATABASE\_VERSION);

}

//Tạo database trong sự kiện onCreate của class OpenHelper, cách viết thì cứ cộng string  
 //lại sao cho thành câu lệnh tạo bảng đúng theo cú pháp của SQL thôi.

@Override

public void onCreate(SQLiteDatabase arg0) {

arg0.**execSQL**("**CREATE TABLE** " + TABLE\_NOTE +

" (" + COLUMN\_ID + " INTEGER PRIMARY KEY AUTOINCREMENT, " +

COLUMN\_NOTE + " TEXT NOT NULL);"

);

}

//Trong sự kiện onUpgrade của database, nó sẽ thực hiện khi nào version của database  
 //có sự thay đổi. Sẽ viết cú pháp drop table đi và khởi tạo lại.

@Override

public void **onUpgrade**(SQLiteDatabase arg0, int arg1, int arg2) {

arg0.**execSQL**("**DROP TABLE IF EXISTS** " + TABLE\_NOTE);

onCreate(arg0);

}

}

**Bước 3: Những hàm để sử dụng database (mở kết nối, thêm sửa xóa lấy, đóng kết nối)**

Hiện trong ví dụ chưa có nhiều vì chưa tìm hiểu thêm, để code với database có nhiều kiểu lắm, xài hàm có sẵn như db.query(“tb\_users”,arg0,arg1…) hay viết thành cú pháp “select \* from tb\_users” rồi chuyển cho database xử lý vv. Cá nhân thì vẫn thích viết cú pháp select \* from … rồi gửi tới database xử lý hơn.

Khi tạo ra class **MyDatabase** và để khởi tạo đối tượng thì code như sau:

//Tạo biến context; Vậy Context là gì? Sao phải tạo nó?  
//Thì Context là một Interface mở tới những thông tin truy xuất trong //Application Enviroment.  
//Đây là một lớp trừu tượng được hđh cung cấp cho phép truy cập vào tài //nguyên ứng dụng và các class liên quan. Khởi tạo nó để tí thằng //OpenHelper nó xài.

private static **Context** **context**;

//Khai báo biến SQLiteDatabase nè \*Đây là ghi chú vô dụng

static **SQLiteDatabase** **db**;

//Khai báo thằng OpenHelper nè \*Đây là ghi chú vô dụng

private **OpenHelper** **openHelper**;

//Khởi tạo dữ liệu, đưa dữ liệu vào biến context \*Đây là ghi chú vô dụng

public **MyDatabase**(Context c) {

MyDatabase.**context** = c;

}

//Hàm mở kết nối tới database.  
//Gọi tới class OpenHelper rồi chạy getWritableDatabase() là được

public **MyDatabase** **open**() throws SQLException {

openHelper = **new** OpenHelper(context);

db = openHelper.**getWritableDatabase**();

return **this**;

}

//Hàm đóng kết nối với database

public void **close**() {

openHelper.**close**();

}

//Lấy dữ liệu từ database

public String **getData**() {

//Đã thấy sao phải khai báo biến từ đầu cho dễ chưa

String[] columns = new **String[]**{ **COLUMN\_ID**, **COLUMN\_NOTE** };

//Cái thèng query này nó query cái table và trả về một con trỏ TT\_TT  
 //Tại sao mấy cái sau null thì là sẽ viết ra các đối số của thằng query  
 // này cho hình dung   
 //Cursor query (String table, String[] columns, String selection,  
 // String[] selectionArgs,   
 //String groupBy, String having, String orderBy)

**Cursor** c = db.**query**(**TABLE\_NOTE**, **columns**, null, null, null, null, null);

String **result** = "";

//getColumnIndex(Cột); là lấy chỉ số, vị trí của cột ...

int iRow = c.**getColumnIndex**(COLUMN\_ID);

int iN = c.**getColumnIndex**(COLUMN\_NOTE);

//Vòng lặp lấy dữ liệu của con trỏ, nhìn khá là rắc rối, mai mốt tìm cách khác  
 //nó chuyển về vị trí bắt đầu của con trỏ, điều kiện là khác vị trí trước của vị trí kết thúc của  
 //con trỏ, tăng dần là c.moveToNext()

for (c.**moveToFirst**(); **!**c.**isAfterLast**(); c.**moveToNext**()) {

result = result + " " + c.**getString**(iRow)

+ " - id:" + c.**getString**(iRow)

+ " - note:" + c.**getString**(iN) + "\n";

}

c.**close**();

return **result**;

}

//Hàm tạo dữ liệu.

public long **createData**(String **noteValue**) {

ContentValues cv = new ContentValues();

cv.**put**(**COLUMN\_NOTE**, **noteValue**);

return db.**insert**(**TABLE\_NOTE**, null, **cv**);

}

**Bước 4: Sử dụng MyDatabase và test**

Trong **Activity** được ví dụ trong project. Trên giao diện cũng chả có gì, chỉ có cái TextView thôi. Dữ liệu sẽ được thêm khi trang đó mở lên và hiển thị ra màn hình do code viết trong sự kiện **onCreate**().

//Khai báo database, trong đó this này truyền sang bên class MyDatabase khởi tạo  
//chính là cái Context c đó

private MyDatabase database = **new** MyDatabase(this);

@Override  
protected void **onCreate**(Bundle savedInstanceState) {

database.**open**();

/\*========================1========================\*/

database.**createData**(“Note1: Gặp anh sịp đỏ thật vui! <3”);

database.**createData**(“Note2: Cuộc sống có thể không cần uống nước!”);

/\*========================2========================\*/

String ds = database.**getData**();

/\*========================3========================\*/

TextView noteList = (TextView) findViewById(R.id.txtNoteList);

noteList.setText(ds);

/\*========================4========================\*/

database.**close**();

}

Code còn có trong cả file **YourDatabase.java** nữa nhé

▄██████████████▄▐█▄▄▄▄█▌  
██████▌▄▌▄▐▐▌███▌▀▀██▀▀  
████▄█▌▄▌▄▐▐▌▀███▄▄█▌Có gì viết thêm vào nhé  
▄▄▄▄▄██████████████▀