ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Mobile Programing

Chapter 4.5. UI layer Libraries

View Binding

View Binding

- Tính năng của android cho phép dễ viết code thao tác với view hơn.
- Khi enable view binding trong module → android sinh các binding classes gắn với các layout file trong module đó.
- Sửa trong gradle module

```
android {
        ...
        buildFeatures {
                viewBinding true
        }
}
```

Cách sử dụng view binding

- Binding class được generated theo Pascal case và thêm Binding ở cuối
  - Ví dụ: activity_main → ActivityMainBinding
- Có thể ignore việc binding của một view bằng cách thiết lập thuộc tính: `tools:viewBindingIgnore="true"`
- Cách sử dụng
  - Trong onCreate gọi inflate để tạo thể hiện của lớp Binding

```
binding = ActivityMainBinding.inflate(getLayoutInflater());
View view = binding.getRoot();
setContentView(view);
```

  - Sau đó truy cập các thành phần view thông qua biến: binding

Sự khác biệt so với findViewbyId

- Null safety: view binding sẽ không bị lỗi null pointer exception khi một view bị invalid
- Type safety: view binding không bị lỗi cast exception

➔ Nếu có lỗi sẽ bị báo khi compile thay vì lỗi khi đang chạy (runtime)

Data Binding

Data binding

- Data binding giúp “liên kết” – bind các thành phần giao diện (UI components) với các data sources theo các “khai báo” (declaration) hơn là lập trình
- Thav vì:

Kotlin Java

```
TextView textView = findViewById(R.id.sample_text);
textView.setText(viewModel.getUserName());
```

- Có thể là

```
<TextView
    android:text="@{viewmodel.userName}" />
```

Data binding

```
android {
        ...
        dataBinding {
            enable true
        }
    }
```

Data binding

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android">

    <data>

        <variable
            name="account"
            type="com.example.trantrungphong.demodatabinding.Account"/>
    </data>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">

        <TextView
            android:layout_width="match_parent"
            android:layout_height="50dp"
            android:gravity="center"
            android:text="@{account.username}"/>

        <TextView
            android:layout_width="match_parent"
            android:layout_height="50dp"
            android:gravity="center"
            android:text="@{account.password}"/>
    </LinearLayout>
</layout>
```

Data binding

```java
public class User {
    public final String firstName;
    public final String lastName;

    public User(String _firstName, String _lastName)
    {
        firstName = _firstName;
        lastName = _lastName;
    }

    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }
}
```

Data binding

```
public class MainActivity extends AppCompatActivity {

    ActivityMainBinding dataBinding = null;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        //setContentView(R.layout.activity_main);

        dataBinding = DataBindingUtil.setContentView( activity: this, R.layout.activity_main);
        User user = new User( _firstName: "Hiep",  _lastName: "Hoang");
        dataBinding.setUser(user);
    }
}
```

Observable data object

- Có khả năng "báo" cho các thành phần khác về sự thay đổi dữ liệu
- Sử dụng plain-old object có thể dùng cho data binding, nhưng khi object thay đổi dữ liệu → không tự động cập nhật UI được.
- 3 loại observable classes: object, field, và collection

Observable data object

```java
public class User extends BaseObservable {
    public String firstName;
    public String lastName;

    public User(String _firstName, String _lastName)
    {
        firstName = _firstName;
        lastName = _lastName;
    }
    @Bindable
    public String getFirstName() { return firstName; }
    @Bindable
    public String getLastName() { return lastName; }

    public void setFirstName(String _firstName)
    {
        firstName = _firstName;
        notifyPropertyChanged(BR.firstName);
    }
    public void setLastName(String _lastName)
    {
        lastName = _lastName;
        notifyPropertyChanged(BR.lastName);
    }
}
```

Lifecycle-aware object

- Layout được bind với nguồn dữ liệu (data sources) mà có khả năng tự động thông báo cho UI về việc dữ liệu bị thay đổi → Lifecycle-aware binding.
- Data binding hỗ trợ: StateFlow và LiveData

Lifecycle-aware components

- Viết 1 class implements DefaultLifeCycleObserver
- Việc viết 1 lớp lifecycle-aware có thể áp dụng cho nhiều activity, hoặc fragment

Lifecycle-aware components

```
public class MyLifeCycle implements DefaultLifecycleObserver {
    Context mContext = null;
    Lifecycle mLifecycle = null;

    @Override
    public void onCreate(@NonNull LifecycleOwner owner) {
        DefaultLifecycleObserver.super.onCreate(owner);
        Log.d( tag: "hiephv",  msg: "aaa");
    }

    public MyLifeCycle(Context _context, Lifecycle _lifeCycle)
    {
        mLifecycle = _lifeCycle;
        mContext = _context;
    }

    public void Test()
    {
        if(mLifecycle.getCurrentState().isAtLeast(Lifecycle.State.CREATED))
        {
            Log.d( tag: "hiephv",  msg: "Da created");
        }
    }
}
```

Lifecycle-aware components

```java
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    //setContentView(R.layout.activity_main);

    dataBinding = DataBindingUtil.setContentView( activity: this, R.layout.activity_main);
    user = new User( _firstName: "Hiep",  _lastName: "Hoang");
    dataBinding.setUser(user);

    MyLifeCycle myLifeCycle = new MyLifeCycle( _context: MainActivity.this, getLifecycle());
    getLifecycle().addObserver(myLifeCycle);

    Button btnTest = dataBinding.btnTest;
    btnTest.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            Log.d( tag: "hiephv",  msg: "Ha ha ha");
            user.firstName = "Ha ha ha";
            user.setFirstName("Ha ha ha");


            myLifeCycle.Test();
        }
    });
```

Cách cài đặt tốt nhất cho lifecycle-aware components

- Giữ cho activity đơn giản nhất có thể: không nên viết các đoạn code acquire dữ liệu trong các callback của activity
- Sử dụng ViewModel để acquire dữ liệu
- Observe a LiveData object để cập nhật dữ liệu ngược về UI
- Để các logic liên quan đến dữ liệu trong ViewModel (ví dụ không nên gọi AsyncTask trong activity để fetch dữ liệu, mà gọi trong ViewModel, sau đó cập nhật dữ liệu lên UI)
- Sử dụng data binding để kết nối giữa views và UI controller.

Tại sao nên có ViewModel

- Activity thường xuyên bị hủy và tạo lại khi có "configuration changes"
- Activity có nhiệm vụ: hiển thị UI, bắt các sự kiện người dùng, handle các giao tiếp với OS (ví dụ permission requests)
- "Bắt" Activity load dữ liệu từ database hoặc internet là không hợp lý → có thể tạo ra nhiều "ghost task" dẫn đến leak bộ nhớ.

→ ViewModel

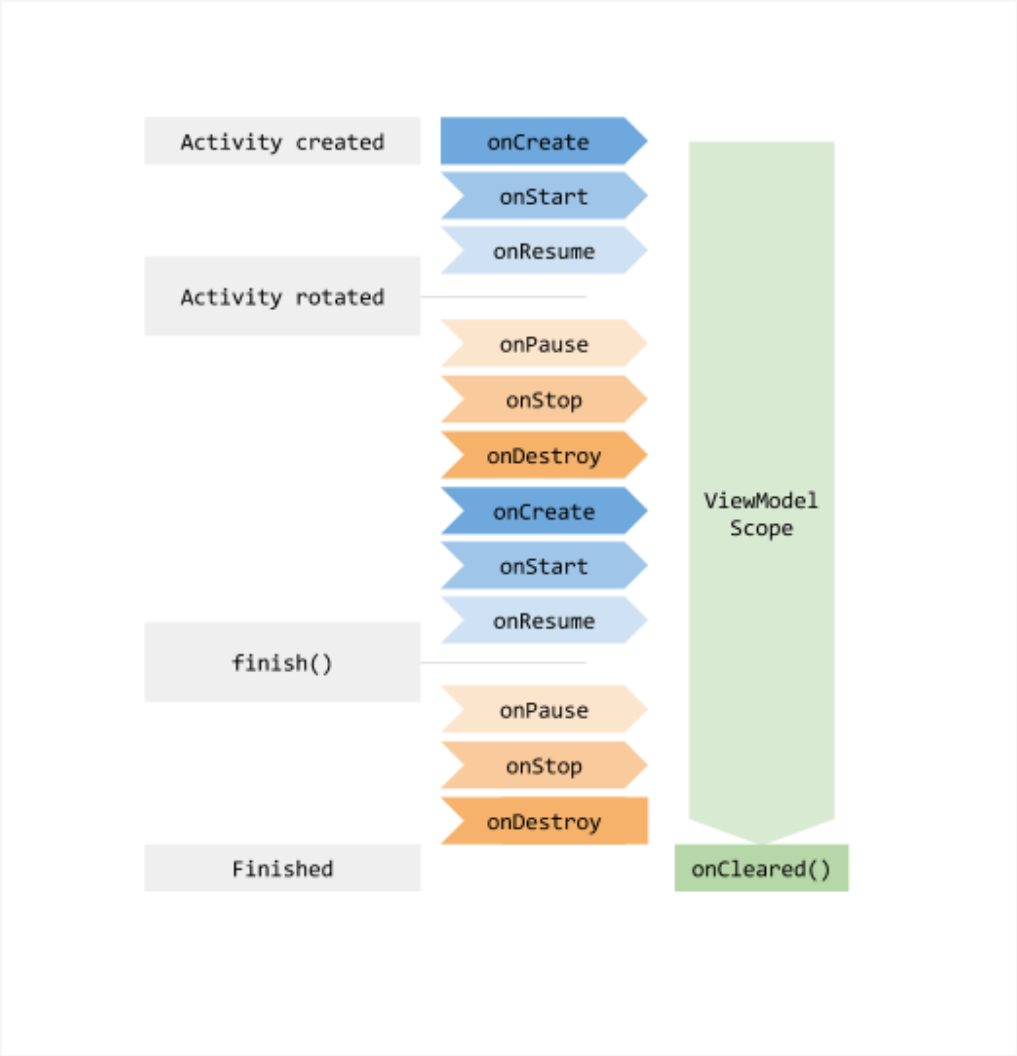
View Model

- Làm thế nào để tách biệt UI và data
- Nếu tạo class riêng chỉ chứa data của Activity → class đó không biết được trạng thái vòng đời của Activity

→ View Model

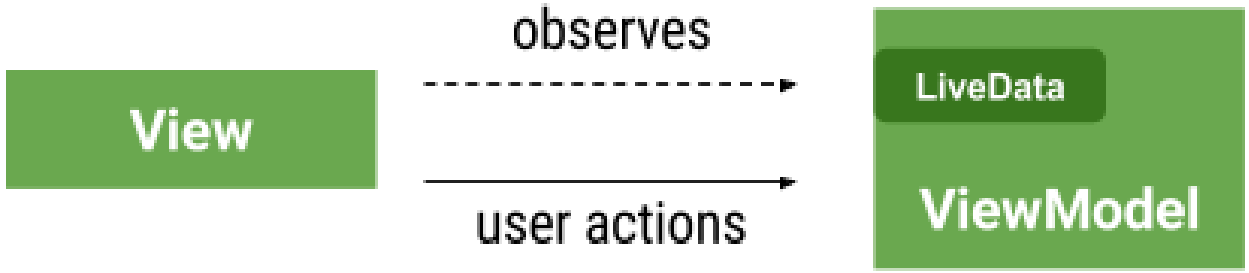
- Chuẩn bị và quản lý dữ liệu của UI Component (Activity, Fragment)
- View model được tạo cùng phạm vi với Activity (Fragment) và được giữ lại ngay cả khi Activity và fragment bị recreated.
- Được dùng để chia sẻ dữ liệu giữa các fragment

View model life cycle



LiveData

- LiveData là observable object nhưng là loại lifecycle-aware object



Lợi ích khi dùng View model

- ViewModel tồn tại khi quay màn hình hoặc các configuration change khác
- ViewModel vẫn running trong khi activity đang trong back stack.
- ViewModel là lifecycle-aware.
- ViewModel với sự hỗ trợ của LiveData có thể phản ứng lại sự thay đổi của UI. Mỗi khi data thay đổi, UI sẽ được cập nhật dựa trên sự quan sát LiveData với data hiện tại trong ViewModel