WaffleStudio 2023
Rookies 21.5 Seminar

Android Seminar 1
2023.09.22 (금)

Instructor : 양주현 (@JuTaK97) / TA : 송동엽 (@eastshine2741)

오늘 배울 것

- View Binding
- 안드로이드 앱에서 Context란? (기초)
- ListView, RecyclerView
- ViewModel과 MVVM 패턴
- ViewModel과 LiveData

View Binding (1)

기존에 하던 findViewById는…

- 번거롭다! 일일히 타입 지정, id 입력…
- id가 틀리면 Exception 발생
- 레이아웃 계층이 커지면 성능 저하
- View를 다른 걸로 변경하면findViewById 한 곳을 일일이 다 바꿔줘야 한다.

```
// build.gradle.kts (:app)

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

View Binding (2)

- 이제 오른쪽과 같이 id를 이용해서 View를 바로 가져올 수 있다.

- xml에서 id는 snake_case로 선언, view binding에서는 camelCase로 가져온다.

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        val binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        binding.myText.text = "hi!"
    }
}
```

View Binding (3)

latent var 을 사용해서 클래스의 멤버 변수로 사용할 수도 있다.

이 경우 onCreate에서 **반드시** assign해 줘야 한다.

```
class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)
    }

    fun otherFunction() {
        Log.d("aaaa",  binding.myText.text.toString())
    }
}
```

Context (1)

안드로이드에서 가장 중요한 개념

공식 문서를 보면…

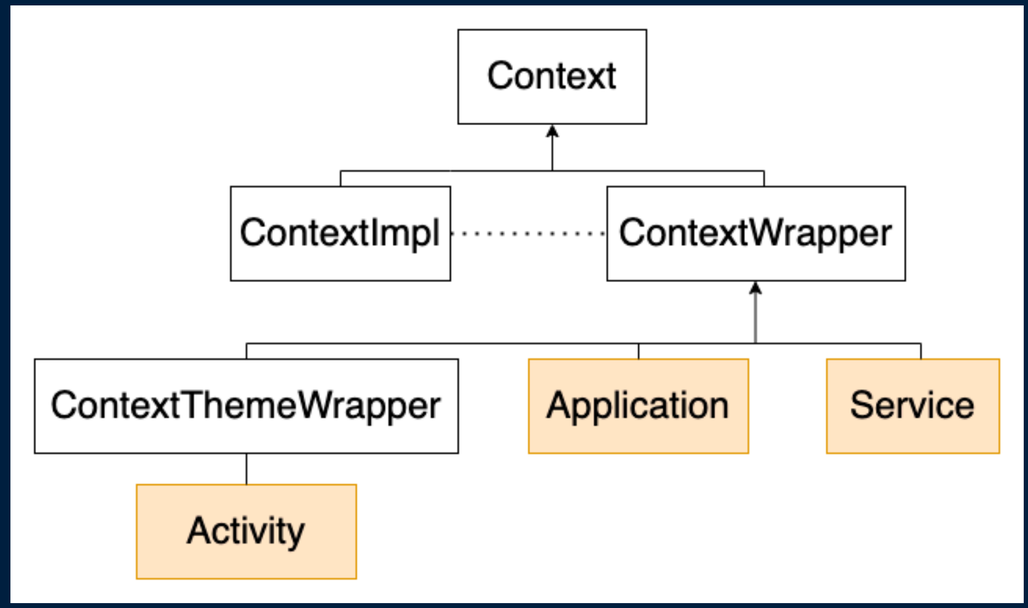
> Interface to global information about an application environment. This is an abstract class whose implementation is provided by the Android system. It allows access to application-specific resources and classes, as well as up-calls for application-level operations such as launching activities, broadcasting and receiving intents, etc.

요약하자면, 안드로이드 시스템에 접근할 수 있는 인터페이스

Context (2)

Context에는 복잡한 상속 관계가 존재한다.

하지만 결국 Activity도 Context이고,
Application도 Context이다.



저번 과제에서 toast를 사용할 때…

```
Toast.makeText(this, "hi!", Toast.LENGTH_SHORT).show()
```

Context 자리에 this로 Activity를 넣어 줬다. 토스트라는 시스템의 동작에 접근하기 위해 context를 요구한 것.

Context (2)

- 액티비티를 시작할 때 context에 this로 액티비티를 넣어 줬다.
- 앱 리소스에 접근할 때 필요하다.

```
val drawable = context.getDrawable(R.drawable.my_image)
val string = context.getString(R.string.app_name)
```

- 파일에 접근할 때 필요하다.

```
val files = context.dataDir.listFiles()
val cacheFiles = context.cacheDir.listFiles()
```

Context (3)

- 기기 정보를 얻는 데 필요하다.

```
val height = context.resources.configuration.screenHeightDp
val density = context.resources.displayMetrics.density
```

- UI 컴포넌트 (뷰 등)을 동적으로 생성할 때 필요하다.

```
binding.myText.setOnClickListener {
    val textViewRuntime = TextView(this).apply {
        text = System.currentTimeMillis().toString()
    }
    binding.root.addView(textViewRuntime)
}
```

ListView (1)

여러 아이템의 리스트를 그려보자!

우리에게 있는 것 : 데이터의 리스트 - listOf(“apple”, “banana”, “cherry”)

우리가 정해야 하는 것 : 데이터를 어떻게 UI로 그릴 것인가?

-> Adapter가 이 일을 담당한다.

리스트의 아이템을 어떻게 그릴지 xml 레이아웃으로 전달해 준다.

ListView (2)

1. 먼저 activity_main.xml에 ListView를 추가한다.

2. 리스트의 아이템을 그릴 설계도를 만든다. (list_item.xml)

3. 데이터와 UI를 매핑시켜 줄 Adapter를 생성한다.

```
val items = listOf(“apple”, “banana”, “cherry”)
val adapter = ArrayAdapter(this, R.layout.list_item, items)
```

4. ListView에 방금 만든 adapter를 등록해 준다.

```
binding.list.adapter = adapter
```

```
<ListView
    android:id="@+id/list"
    android:layout_width="wrap_content"
    android:layout_height=“wrap_content"/>
```

```
<!— list_item.xml —>
<TextView xmlns:android="http://
schemas.android.com/apk/res/android"
    android:id="@+id/description"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"/>
```

ListView (3)

String 데이터 하나를 TextView 하나에 매핑해서 UI를 그렸는데…

-> 조금 더 복잡한 아이템을 그리고 싶으면? Adapter를 상속해서 나만의 Adapter를 만들면 된다.

```
class MyListAdapter<T>(private val list: ArrayList<T>): BaseAdapter() {

    override fun getCount(): Int = list.size
    override fun getItemId(position: Int): Long = 0
    override fun getItem(position: Int): T = list[position]

    override fun getView(position: Int, convertView: View?, parent: ViewGroup?): View {
        // position 위치의 데이터를 어떻게 View로 그릴래?
    }
}
```

ListView (4)

getView 함수에서 리스트의 position 번째 item을 어떻게 그릴 지 말해주면 된다.

view binding으로 xml을 inflate해서 view를 가져오고, 내용을 채운 다음 반환해 준다.

```
override fun getView(position: Int, convertView: View?, parent: ViewGroup?): View {
    val user = getItem(position)

    val binding = ListItemBinding.inflate(LayoutInflater.from(context))
    binding.title.text = user.name
    binding.subtitle.text = user.githubId
    binding.description.text = user.email

    return binding.root
}
```

ViewHoler

하지만 이렇게 하면 스크롤을 해서 View가 사라지고 새로 생길 때마다 View를 일일이 만들고 내용을 다시 채운다.

이는 메모리 낭비&성능 저하로 이어지고, ViewHolder 패턴으로 해결한다.

```
private class ViewHolder(
    var title: TextView,
    var subtitle: TextView,
    var description: TextView,
)
```

```
val view: View
val holder: ViewHolder

if (convertView == null) {
    view = LayoutInflater.from(context).inflate(R.layout.list_item, null)
    holder = ViewHolder(
        title = view.findViewById(R.id.title),
        subtitle = view.findViewById(R.id.subtitle),
        description = view.findViewById(R.id.description),
    )
    view.tag = holder
} else {
    holder = convertView.tag as ViewHolder
    view = convertView
}

holder.apply {
    title.text = user.name
    subtitle.text = user.githubId
    description.text = user.email
}
```

RecyclerView (1)

모든 단점을 해결한 궁극체가 바로 RecyclerView이다.

RecyclerView를 사용하기 위해서는 Library를 추가해 줘야 한다. build.gradle.kte (:app)의 dependencies 안에 다음 라인을 추가한다.

```
implementation("androidx.recyclerview:recyclerview:1.3.1")
```

그리고 activity_main.xml에 RecyclerView를 추가해 준다.

```
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/recycler_view"
    android:layout_width="wrap_content"
    android:layout_height="match_parent"/>
```

RecyclerView (2)

RecyclerView 역시 Adapter가 필요하다. (복습: Adapter는 데이터와 UI의 매핑을 해 주는 역할을 한다.)

라이브러리 내에 RecyclerView.Adapter가 abstract class로 정의되어 있고, ViewHolder 또한 라이브러리 내에 정의되어 있다. 우리는 이를 상속받아서 입맛에 맞게 구현해 주면 된다.

```
class MyMultiAdapter: RecyclerView.Adapter<RecyclerView.ViewHolder>() {

}
```

기본 형태는 위와 같이 하면 되고,

```
override fun getItemCount(): Int

override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): RecyclerView.ViewHolder

override fun onBindViewHolder(holder: RecyclerView.ViewHolder, position: Int)
```

이렇게 3개의 메소드를 필수로 override해 주면 된다.

RecyclerView (3)

먼저, 데이터는 어댑터의 생성자를 통해 전달해 준다. 그러면 getItemCount 함수는 쉽게 구현할 수 있다.

```
class MyMultiAdapter(private val data: List<MyData>) : RecyclerView.Adapter<RecyclerView.ViewHolder>() {
    override fun getItemCount(): Int {
        return data.size
    }
```

그러면 MainActivity.kt에서 이렇게 어댑터를 만들어서 넣어준다. (ListView와 비슷하다)

```
val items = mutableListOf<Data>(
    // 데이터들
)
val adapter = MyMultiAdapter(items)
binding.recyclerView.adapter = adapter
binding.recyclerView.layoutManager = LinearLayoutManager(this)
```

RecyclerView (4)

RecyclerView의 ViewHolder는 abstract class이다. 즉 우리가 필요한 만큼 "여러 종류로" 상속해서 사용할 수 있다.

커뮤니티 사이트의 게시글 리스트를 만든다고 생각해 보자. 제목만 보이는 항목도 있고, 제목과 본문 1줄 미리보기가 보이는 항목도 있고, 이미지 카드와 제목이 상하로 배치된 항목도 있다. 이런 경우 이 ListView의 item의 설계도는 하나의 xml 파일로 표현할 수 없다. 이럴 때 여러 종류의 ViewHolder가 필요하다.

이런 특성의 데이터를 나타내는 데는 sealed class만한 것이 없다.

```
sealed class MyMultiData(val viewType: ViewType) {
    data class TypeA(val texts: List<String>) : MyMultiData(ViewType.A)

    data class TypeB(@DrawableRes val imageRes: Int) : MyMultiData(ViewType.B)

    data class TypeC(val num: Int) : MyMultiData(ViewType.C)

    enum class ViewType { A, B, C } // viewType은 잠시 무시하자
}
```

RecyclerView (5)

그러면 각 데이터 종류마다 xml 파일과 ViewHolder를 하나씩 만들어 준다. (마치 음식 종류별로 다른 용기에 담는 것처럼)

```
inner class TypeAViewHolder(private val binding: TypeAItemViewBinding) :
    RecyclerView.ViewHolder(binding.root) {
}

inner class TypeBViewHolder(private val binding: TypeBItemViewBinding) :
    RecyclerView.ViewHolder(binding.root) {
}

inner class TypeCViewHolder(private val binding: TypeCItemViewBinding) :
    RecyclerView.ViewHolder(binding.root) {
}
```

(복습) view binding에 의해, type_a_item_view.xml을 만들면 TypeAItemViewBinding 객체가 자동으로 만들어진다.

RecyclerView (6)

이제 아까 남겨뒀던 함수들을 마저 볼 때가 됐다.

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): RecyclerView.ViewHolder
```

ViewHolder의 부모 viewGroup과 viewType이라는 것을 받는다.

(복습) 우리는 프레임워크 위에 있다. Adapter의 함수 onCreateViewHolder는 프레임워크가 호출하고, 우리는 프레임워크가 언제 이 함수를 호출하는지 알아서 적절히 구현해 주는 것을 공부하고 있다.

Int 타입인 viewType은 뭘까? 라이브러리 주석의 설명은 다음과 같다.

> integer value identifying the type of the view needed to represent the item

즉, 우리가 3가지 종류의 아이템을 표시하려고 하면 3종류의 int 값으로 이를 구분한다는 것이다.
그런데 이 int값은 누가 어떻게 알고 넣어 줄까?

RecyclerView (7)

필수 override 함수가 아닌 함수중에 이런 게 있다.

```
override fun getItemViewType(position: Int): Int {
    return 0
}
```

가장 기본적인 RecyclerView.Adapter는 리스트가 한 종류의 ViewHolder만 갖는다고 가정한다. 그래서 디폴트로 상시 0을 반환하는 함수로 정의되어 있다.

우리는 입맛에 맞게 RecyclerView.Adapter를 상속해서 커스텀하는 중이므로 이 함수도 상속해서 바꿔 줘야 하고, 우리가 사용 중인 데이터 종류마다 어떤 Int값으로 구분할 지 이 함수에서 정해 주면 된다.

아까 sealed class 정의할 때 ViewType이라는 enum class가 있었는데, 바로 지금을 위해 만든 것이다.
그냥 Int를 쓰면 코드를 읽을 때 뭐가 뭔지 구분이 잘 가지 않으므로 enum을 사용해 코드의 가독성을 높인다.

```
override fun getItemViewType(position: Int): Int {
    return data[position].viewType.ordinal
}
```

RecyclerView (8)

onCreateViewHolde로 다시 돌아오면, 이제 viewType이 뭔지 안다. (어떤 Int 값이 어떤 ViewHolder에 대응하는지 우리가 방금 정했다)

함수의 body를 이렇게 채우는 것은 이제 자연스럽다. (가독성을 높이기 위해 enum을 사용한 효과가 보인다!)

```
return when (viewType) {
    MyMultiData.ViewType.A.ordinal -> {
        TypeAViewHolder(TypeAItemViewBinding.inflate(LayoutInflater.from(parent.context)))
    }

    MyMultiData.ViewType.B.ordinal -> {
        TypeBViewHolder(TypeBItemViewBinding.inflate(LayoutInflater.from(parent.context)))
    }

    MyMultiData.ViewType.C.ordinal -> {
        TypeCViewHolder(TypeCItemViewBinding.inflate(LayoutInflater.from(parent.context)))
    }

    else -> throw IllegalStateException("Invalid ViewType") // 발생할 일은 없다.
}
```

RecyclerView (9)

이제 데이터 종류별로 어떻게 그릴지 정의할 일이 남았고, 마지막 남은 override fun onBindViewHolder 를 보자.

```
override fun onBindViewHolder(holder: RecyclerView.ViewHolder, position: Int)
```

viewholder를 받고, position을 받는다. viewholder는 아까 onCreateViewHolder에서 우리가 정한 ViewHolder 일 것이고, position은 data 내의 위치이다.

즉, 이 함수는 데이터의 position 번째의 holder에 담긴 아이템을 그리는 곳이다.
position 번째 아이템의 데이터는 data[position] 으로 쉽게 얻을 수 있으니, 데이터에 매핑되는 UI를 그리면 된다.

data의 값 대로 UI에 정보를 그려야 할 텐데, ViewHolder를 정의할 때 생성자에 binding을 받았었다. 이걸 이용해야 하므로(binding.textView.text = data.text 같이 할 것이다), ViewHolder마다 이렇게 함수를 만들어 주자.

```
inner class TypeCViewHolder(private val binding: TypeCItemViewBinding): RecyclerView.ViewHolder(binding.root) {
    fun bind(num: Int) {
        binding.num.text = num.toString() // type_c_item_view에 id가 "num"인 TextView가 있다
    }
}
```

RecyclerView (10)

다른 ViewHolder에도 비슷하게 onBindViewHolder 함수에서 data[position] 으로 가져온 데이터에서 필요한 정보들을 뽑아서 bind 함수(이름은 뭐든 좋다)에 파라미터로 전달한 후, ViewHolder가 갖고 있는 binding의 각 구성 요소마다 정보를 채워 주면 된다. (text라던가 image source라던가…)

예를 들어 ImageView 하나가 있는 TypeBViewHolder는 아래와 같다.

```
fun bind(@DrawableRes imageRes: Int) {
    binding.image.setImageResource(imageRes)
}
```

만약 Adapter에 제공하는 데이터가 바뀌었다면? 새 아이템이 추가되었다면?
MainActivity에서 val items = mutableListOf() 처럼 mutable로 만들었던 건 지금을 위한 것이었다…
버튼을 하나 만들고, 클릭 이벤트에서 새 item 객체를 만들고 items.add(newItem)를 해 보자.

items.add(newItem) 을 했으면 adapter에게 “너한테 준 데이터가 바뀌었어” 를 알려 줘야 하고,
RecyclerView.Adapter에 notifyDataSetChanged 라는 함수가 있다.

이 함수는 데이터셋 전체가 바뀌었을 때 쓰는 거라 오버헤드가 크므로, notifyItemChanged(int position) 등의 특정 위치의 아이템이 바뀌었다고 Adapter에게 통지하는 더 가벼운 함수들을 이용하면 된다.

ViewModel (1)

방금까지 만든 앱에서 다크모드를 바꿔 보면 items.add() 로 추가했던 아이템들이 사라진다. 왜?

다크모드를 전환하면, Activity 수명 주기(복습하기!)에서 onDestroy()가 불리고 onCreate(), onStart(), onResume()이 차례로 다시 불리기 때문이다. 우리가 런타임에 추가했던 item들은 메모리에서 사라지고, 코드상으로 넣어 준 것들만 다시 코드가 실행되어 RecyclerView를 채우는 것이다.

이런 상황은 다크모드 뿐만 아니라 여러 상황에서 나타나는데, 대표적으로 아래와 같은 것들이 있다.

1. 화면 회전
2. 메모리 부족으로 인한 시스템 단의 동작

이를 방지하는 방법은 앱의 캐시나 데이터 DB에 저장하는 것도 있겠지만, 좀 더 효율적이고 간편한 방법이 필요하다.

문제의 근원은 Activity Lifecycle이니까 Activity Lifecycle에 관계 없는 (또는 액티비티보다 더 오래 살아가는) 무언가가 정보를 들고 있으면 된다.

그리고 이 역할을 하는 것이 바로 ViewModel이다.

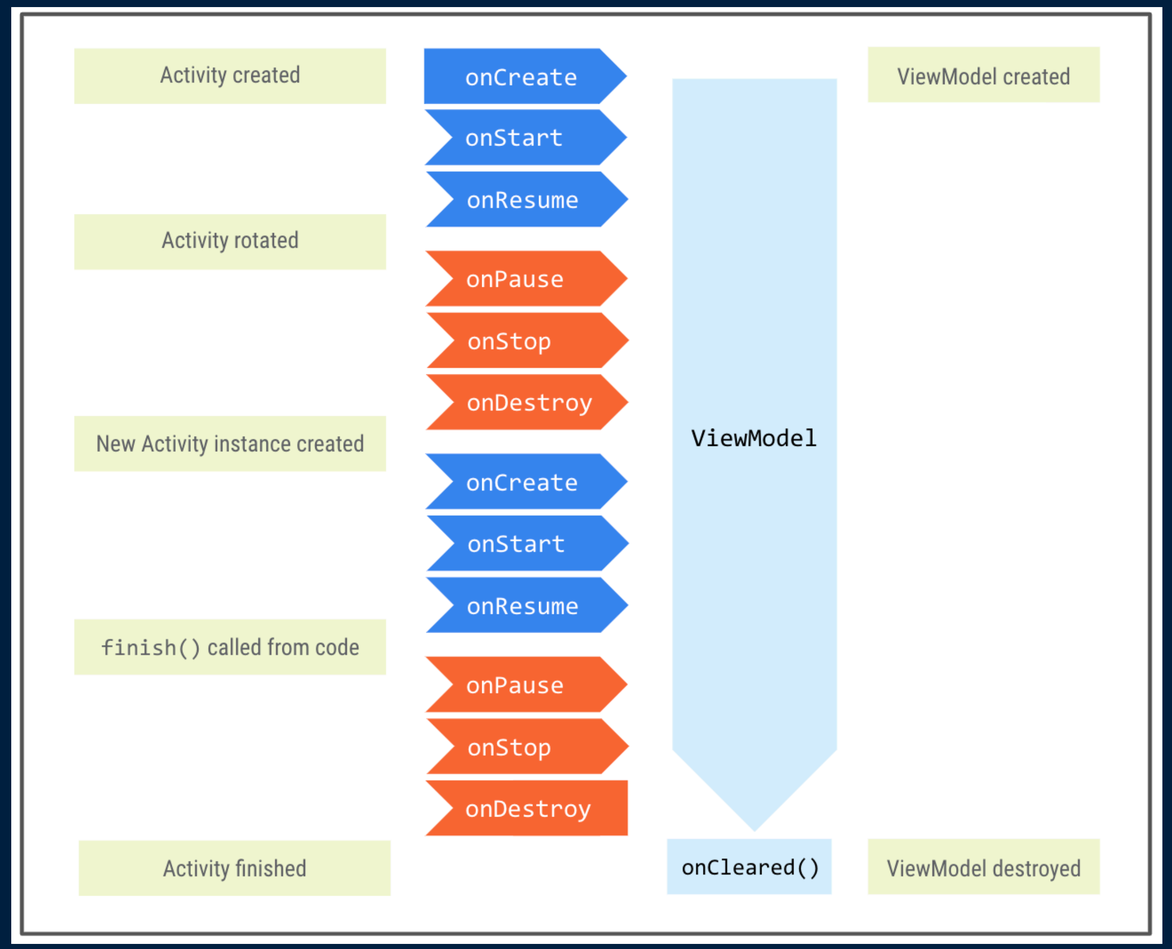
ViewModel (2)

ViewModel 클래스는 lifecycle scope를 가진다.

ViewModel의 scope를 액티비티로 넣어 주면, 뷰모델은 액티비티의 Lifecycle이 어떻게 변화하든 상관 없이 독립적으로 존재한다.

ViewModel이 사라지는 건 scope가 완전히 끝날 때 (Activity의 경우 finished될 경우) 이다.

이러한 장점을 이용해서여러 UI가 공유하는 데이터를 저장할 때 뷰모델을 사용할 수 있다.



ViewModel (3)

ViewModel을 사용하기 위해서 아래 dependency를 추가해 준다.

```
implementation("androidx.activity:activity-ktx:1.7.2")
```

그리고 액티비티에 다음과 같이 선언해서 사용할 수 있다.

```
class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding
    private val viewModel: MainViewModel by viewModels()
    // ...
```

by viewModels()는 activity-ktx에서 제공하는 ComponentActivity의 확장 함수로, Activity의 뷰모델을 lazy하게 간편하게 만들어 준다.

MVVM (1)

잘 짠 코드란 무엇일까?
여러 요소가 있겠지만, “가독성이 좋고” “유지보수가 쉬운” 코드일 수록 좋다.

앱의 규모가 커지고 View의 개수가 많아지고, View마다 수십 줄 짜리 로직이 붙기 시작하면 가독성과 유지보수를 위해 코드를 좋은 형태로 분리시키고 정리할 필요가 있다. 이를 우리는 디자인 패턴 이라고 부른다.

UI 개발에서 가장 대표적으로 사용되는 아키텍처 디자인 패턴은 MVVM (Model - View - ViewModel)이다.
학생들의 인적사항 목록을 예쁜 카드 형태로 표시한다고 생각해 보면,

Model은 데이터와 비즈니스 로직에 해당한다.
학생의 인적사항을 나타내는 data class나, 이를 가공하는 다양한 로직 (예: 학과별로 groupBy하기, 성적 순으로 sort하기 등등)이 Model이다.

View는 UI의 외관 (예시에서는 카드 형태 UI)이다. UI의 xml 파일, view binding으로 얻어온 각종 View를 뜻한다.

ViewModel은 이 둘을 연결해 주는 역할을 한다. View는 데이터가 어떻게 생겼고 어떻게 가공되는지 알지 못하고, Model은 UI가 어떻게 생겼는지 알지 못한다. 이들은 오직 ViewModel을 통해서 소통한다.

MVVM (2)

Model은 ViewModel에서 비즈니스 로직을 거쳐 가공된다.

```
class ViewModel {
    val rawData: Model = db.queryBy(someLogic)
    val name = rawData.someLogic()
}
```

그리고 View는 ViewModel에서 가공이 완료된 데이터만 참조해서 UI를 채운다.

```
binding.name.text = viewModel.name
```

이와 같이 UI와 비즈니스 로직을 분리하면 모듈화가 쉽고 재사용성이 증가합니다. (대충 좋다는 뜻)
아직은 와닿지 않더라도.. 이런 게 있구나 정도로 받아들이고 넘어가면 됩니다.

LiveData

일반적으로 reactive programming (반응형 프로그래밍)을 ViewModel과 함께 사용한다.

ViewModel은 데이터 스트림(stream)을 들고 있고 View는 이를 관찰(observe)하는 형태로, 데이터의 변경에 따라 UI를 자동으로 업데이트하도록 구현할 수 있다.
안드로이드에서 LiveData를 이용해 쉽게 reactive programming을 구현할 수 있다.

```
// ViewModel
private val _data = MutableLiveData<Int>(0)
val data: LiveData<Int> = _data
fun handleClickEvent() {
    _data.value = _data.value?.plus(1)
}

// Activity
binding.myText.setOnClickListener { viewModel.handleClickEvent() }
viewModel.data.observe(this) {
    binding.myText.text = it.toString()
}
```

과제 공지

과제 2: Tic-Tac-Toe 게임 만들기

- 과제 테마 : ViewModel, LiveData / RecyclerView, ViewHolder

- 과제 2 기한 : 10월 5일 목 자정 (목~금 넘어가는 밤)

- 과제 링크 : https://github.com/wafflestudio/seminar-2023-android-assignment/tree/main/assignment-2