What you can build at the end of this semester?

A lot as long as you don't build everything yourself.

- e.g. Compose Cookbook

What skills do you need to build upon others work and modify it to fit your own ideas?

Ability to trace code and understand why it is implemented this way.

Challenge of what you have learnt in Unit.3

Outline

1. Activity
2. Fragment
3. ViewModel
4. LiveData

1. Activity

1.1. Why should we care about *activity lifecycle*?

We might need to be careful about `onCreate` (as for state initialization) and `onDestroy` (as the states will be lost), but what about others like `onStart`, `onResume`, etc?

Good implementation of the lifecycle callbacks can help ensure that your app avoids:

- **Crashing** if the user receives a phone call or switches to another app while using your app.
- **Consuming valuable system resources when the user is not actively using it**.
- Losing the user's progress if they leave your app and return to it at a later time.
- Crashing or losing the user's progress when the screen rotates between landscape and portrait orientation.

For more info: Official Doc

2. Fragment

2.1. Why do we need `Fragment`?

Instead of using `Fragment1`, `Fragment2` for different pages, can't we just use `Activity1`, `Activity2`? What's the difference?

- **Modularity**: Fragments introduce modularity and reusability into your activity’s UI by allowing you to divide the UI into discrete chunks. Activities are an ideal place to put global elements around your app's user interface, such as a navigation drawer. Conversely, **fragments are better suited to define and manage the UI of a single screen or portion of a screen**.

2.2. Fragment Lifecycle vs. ViewBiding

Q1. Why do we initiize the `binding` in `onCreateView`, but set its attribute in `onViewCreated`? Can't we just have them all in 1 method?

```
class StartFragment : Fragment() {
  ...
  override fun onCreateView(...): View? {
    val fragmentBinding = FragmentStartBinding.inflate(inflater, container, false)
    binding = fragmentBinding
    return fragmentBinding.root
  }

  override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)
    binding?.startFragment = this
  }
}
```

- Actually there is no obvious difference between `onCreateView` and `onViewCreated`, but the second is called immediately after the first. However, it is recommended to initialize the sub component inside a view in `onViewCreated`
- Here's an alternative way to acheive the same effect

```
class StartFragment : Fragment(R.layout.fragment_start) {
  ...

  override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    binding = FragmentStartBinding.bind(view)
    binding?.startFragment = this
  }
}
```

Q2. Why do we need to have `binding = null` in `onDestroyView`? Shouldn't it be automatically removed once the Fragment dies?

```
class StartFragment : Fragment() {
  private var binding: FragmentStartBinding? = null

  override fun onDestroyView() {
    super.onDestroyView()
    binding = null
  }
}
```

`onDestroyView` is called when the view hierarchy associated with the fragment is being removed, but not the fragment itself. Therefore, if we don't have `binding = null`, then it will keep a reference to a viewbinding object that will no longer be used before `onDestroy` is called

3. ViewModel

3.1. Why do we need ViewModel?

Instead of inheriting from `ViewModel`,

```
class OrderViewModel : ViewModel() {
  ...
}
```

can't we just have a custom class to store the view state?

```
class OrderViewState() {
  ...
}
```

The ViewModel class is designed to store and manage UI-related data in a lifecycle conscious way. **The ViewModel class allows data to survive configuration changes such as screen rotations**.

3.2. What does property delegation(`by`) do? Why do we need to use it when declaring a `ViewModel` in a `Fragment`?

```
private val viewModel: OrderViewModel by OrderViewModel()
```

Can't we just use `=` instead?

```
private val viewModel: OrderViewModel = OrderViewModel()
```

Kotlin `getter` / `setter` & *backing field*

```
val viewModel: OrderViewModel
  get() {
    return field
  }
  set(value) {
    field = value
  }
```

Property Delegation

```
val viewModel: OrderViewModel
  get() {
    return OrderViewModel.getValue()
  }
  set(value) {
    OrderViewModel.setValue(value)
  }
```

By using property delegation, `ViewModel` will handle the lifecycle for us.

If we use `=` instead, everytime when configuration changes, the `Fragment` will get a new instance of `OrderViewModel`, which will loss the state.

4. LiveData

4.1. Why do we need LiveData?

With LiveData, we have to attach an observer in different places to notify the change:

```
// OrderViewModel
val quantity = LiveData<Int>(0)

// StartFragment
viewModel.quantity.observe(lifeCycleOwner) { value ->
  observer1 = value
  observer2 = value
}
```

Can't we just use a custom setter?

```
// OrderViewModel
public var quantity: Int = 0
  set(value) {
    observer1 = value
    observer2 = value
    field = value
  }
```

LiveData is an observable data holder class. Unlike a regular observable, **LiveData is lifecycle-aware**, meaning it respects the lifecycle of other app components, such as activities, fragments, or services. This awareness ensures **LiveData only updates app component observers that are in an active lifecycle state.**

4.2. Why do we write LiveData in ViewModel like this

```
private val _quantity = MutableLiveData<Int>()
val quantity: LiveData<Int> = _quantity
```

Protection: Never directly expose mutable data from a ViewModel.