State

상태란 무엇인가?

- 특정 변수나 Composable에서 값의 변화가 있었는지를 나타낸다. 변화가 발생한 경우에 상태가 업데이트 된다.
- 상태의 변경이 생기면 Recomposition이 발생한다.

Recomposition

- 상태가 변경되면 UI 요소가 새로운 데이터를 반영하여 화면을 다시 구성하는 과정이고, 일반적으로 상태가 변경되면 발생한다.
- Composable의 상태가 변경될 때 자동으로 UI를 갱신해준다.
- State<T> , MutableState<T>의 차이
  -State<T> : 변경할 수 없는 객체
  -MutableState<T> : 변경 가능한 객체

remember , mutableStateOf()

- mutableStateOf()
  -주어진 파라미터로 초기화 된 MutableState 객체를 반환한다.
  -상태가 변경되어 Recomposition이 발생하면 다시 초기화 된다.
- Remember
  -상태를 유지하기 위해서 mutableStateOf와 함께 사용되고, recomposition이 발생할 때, 해당 변수가 다시 초기화 되지 않도록 해준다.
- remember와 mutableStateOf 함수를 사용 방법들
  val exampleValue by remember { mutableStateOf(0) }
  -이후에 사용할 때 그냥 exampleValue로 사용 가능
  -State.getValue import해야 함
  -객체를 전달할 일이 없거나 간단하게 다루는 경우에 주로 사용

  val exampleValue = remember { mutbleStateOf(0) }
  -이후에 값을 이용할 때 exampleValue.value 로 사용
  -다른 컴포넌트와 상태를 공유하거나, ViewModel을 통해 값을 넘겨줘야 할 때 주로 사용

Remember의 한계

- remember는 기본적으로 composable이 재구성될 때에는 상태를 유지하지만, 구성 변경(configuragion change)가 발생하는 상황에서는 상태가 유지되지 않는다.
- 구성 변경이 발생하면 Activity/Fragment는 destroy되고, 새로 create된다. 이와 같은 경우에는 remember의 상태가 초기화 된다. Week03C 예시
- 구성 변경이 발생하는 경우는 주로 화면 회전, 다크모드 전환, 폰트 크기 변경 등 장치 설정이 변경될 때 발생한다
- 구성 변경이 발생해도 상태를 유지해야 한다면 remember 대신 rememberSavable 또는 ViewModel을 사용하면 좋다.

RememberSavable

- 내부적으로 Bundle을 사용해 화면 회전과 같은 구성 변경이 발생해도 상태를 유지한다
- Bundle에 데이터를 저장하기 때문에, Bundle이 지원하는 타입들에 대해서 상태를 저장할 수 있다
- 저장할 수 있는 기본 데이터 타입 : Int, Long, Double, Float, Boolean, String
- Bundle에 저장 가능한 타입
  -Parcelable로 구현한 객체
  -Serializable로 구현한 객체
  -Array (기본 데이터 타입의 배열)
  -ArrayList (기본 데이터 타입의 리스트)
  -Bundle 객체

- 기본 타입이 아닌 경우의 저장
  -Parcelable, Serializable로 구현한 객체들의 경우 기본 타입이 아니라면 listSaver, mapSaver 를 사용하거나 Custom Saver를 만들어서 사용해야한다.

```
interface Saver<Original : Any?, Saveable : Any>
```

복원할 타입 저장할 타입

- Parcelize를 사용한 예시

  -build.gradle의 plugin에 kotlin-parcelize를 추가하여 사용한다.

```
@Parcelize
data class City(val name: String, val country: String) : Parcelable

@Composable
fun CityScreen() {
    var selectedCity = rememberSaveable {
        mutableStateOf(City("Madrid", "Spain"))
    }
}
```

  -parcelize를 사용하면 이후에 객체의 한 속성만 변경하는 경우 recomposition이 일어나지 않기 때문에, selectedCity = selectedCity.copy(name = "Barcelona") 와 같이 객체 자체를 복사하여 값을 변경해야 recomposition이 일어난다.

- mapSaver 예시

  -mapSaver를 사용하여 값 집합으로 객체를 변환하여 객체를 저장하고, 이후에 불러올 때 집합에서 있는 값에서 객체로 변환하여 가져온다.

```
data class City(val name: String, val country: String)

val CitySaver = run {
    val nameKey = "Name"
    val countryKey = "Country"
    mapSaver(
        save = { mapOf(nameKey to it.name, countryKey to it.country) },
        restore = { City(it[nameKey] as String, it[countryKey] as String) }
    )
}

@Composable
fun CityScreen() {
    var selectedCity = rememberSaveable(stateSaver = CitySaver) {
        mutableStateOf(City("Madrid", "Spain"))
    }
}
```

***run 함수 : 블록 실행하고, 마지막 표현식 반환**

- listSaver 예시

  -index로 map의 key를 대신하여 사용

```
data class City(val name: String, val country: String)

val CitySaver = listSaver<City, Any>(
    save = { listOf(it.name, it.country) },
    restore = { City(it[0] as String, it[1] as String) }
)

@Composable
fun CityScreen() {
    var selectedCity = rememberSaveable(stateSaver = CitySaver) {
        mutableStateOf(City("Madrid", "Spain"))
    }
}
```

- CustomSaver 예시

  -아래 예시와 같이 직접 저장 방식을 작성하여 사용할 수도 있다.

```
import androidx.compose.runtime.saveable.Saver

data class Holder(var value: Int)

// this Saver implementation converts Holder ob
// to Int which we can save
val HolderSaver = Saver<Holder, Int>(
    save = { it.value },
    restore = { Holder(it) }
)
```

ViewModel

- RememberSavable보다 더 복잡한 데이터나 로직을 저장하여 사용할 수 있다.
- RememberSavable과는 달리 Bundle의 제약을 받지 않기 때문에 대부분의 데이터를 자유롭게 관리할 수 있다.
- 저장 가능한 타입

  -RememberSavable로 저장 가능한 타입들

  -Parcelable, Serializable로 객체를 구현하지 않아도 객체를 저장할 수 있다.

  -사용자가 정의한 클래스나 데이터 구조
- 비동기 작업 관리

  -LiveData, Flow, StateFlow 등의 비동기 데이터 스트림을 이용하여 데이터를 관리할 수 있다.

  -viewModelScope를 통해 비동기 작업을 관리할 수 있다.

  -Compose와 함께 사용할 때에는 주로 UI에서 collectAsState로 viewModel의 데이터들을 가져와서 사용한다.
- 구성 변경에서 상태를 유지하지만, 앱을 종료했다가 다시 실행하는 것과 같이 프로세스 자체가 종료되고 다시 시작할 때에는 상태가 사라진다. 이때 SavedStateHandle을 사용하면 일부 상태를 Bundle로 저장해둘 수 있다.
- 참고사항)

  -Jetpack Compose를 사용한 안드로이드 개발에서는 ViewModel에서 상태를 관리하고 로직을 처리하기에 좋고 이 상태들을 UI와 연결하는게 간단하기 때문에 압도적으로 MVVM(Model – View – ViewModel) 아키텍처를 많이 사용한다.

  -MVVM에서 View는 UI(Composable), Model(Room, Retrofit등 데이터 처리), ViewModel(상태, 로직 등을 관리)하는 방식으로 구조를 짜서 사용

  -MVVM 구조가 사실상의 표준이라고 한다.

상태 호이스팅(State Hoisting)

- 상태에 따라 Composable을 분류할 수 있다.

  -Stateful Composable : 상태를 직접 관리하는 composable, 간단하지만 재사용성과 테스트의 용의성이 떨어짐

  -Stateless Composable : 상태를 직접 가지고 있지 않아서 구현하기에 좀 더 복잡할 수 있으나 재사용성과 상태 공유에 용이하다.
- 상태 호이스팅은 composable이 가지고 있는 상태를 해당 composable을 호출한 composable에서 관리하는 것을 의미한다.

- 위의 왼쪽 코드에서 오른쪽 코드로 변경하면 MyTextField에 상태 호이스팅을 적용한 것

Navigation

- 앱에서 다양한 화면 사이의 이동을 할 수 있게 해주는 기능이다. NavController, NavGraph, NavHost를 사용하여 구현한다.
- Build.gradle에 android.navigation.compose를 추가하고, androidx.navigation.compose.NavHost를 import하여 사용
- NavController
  -앱의 화면간 이동을 담당하고, 현재 경로를 관리한다. navigate, popBackStack 등의 메소드를 사용한다.
- NavGraph
  -navigation에서의 구조를 정의한다. Composable을 통해 주어진 경로와 UI(화면)을 연결한다.
  -프로젝트의 규모가 커져서 화면들을 분류할 필요가 있는 경우 분류된 화면들에 해당하는 NavGraph도 구분하여 작성하는게 유지보수에 편리할 수 있다. 이렇게 규모가 커서 기능별로 나누는 경우를 Subgraph라고도 한다.
- NavHost
  -NavGraph와 NavController를 연결하여 UI를 보여준다. 실제 보이는 화면을 구성하는 역할
  -주로 NavGraph의 코드에 함께 작성하여 관리
- Navigation Back Stack
  -화면을 이동할 때마다 화면들이 navigation back stack에 쌓인다.
  -popupTo(), launchSingleTop 등의 옵션들을 사용하여 추가적인 설정을 할 수 있다.