Jetpack Compose

상태 호이스팅(state hoisting) 이점

상태 호이스팅의 정의

상태 관리를 호출하는 쪽(call site)으로 이동시켜 상태 저장(stateful) 컴포저블을 상태 비저장(stateless) 컴포저블로 변환하는 데 사용되는 디자인 패턴

일반적 패턴: 상태를 상위에 두고 아래와 같이 Getter와 Setter를 매개변수로 사용

- `getter : value: T`: 표시할 현재 값
- `setter : onValueChange: (T) -> Unit`: `T`가 제안된 새 값인 경우 값을 변경하도록 요청하는 이벤트

상태 호이스팅에 중요한 속성

단일 정보 소스: 상태를 복제하는 대신 옮겼기 때문에 정보 소스가 하나만 있습니다.

캡슐화됨: 스테이트풀(Stateful) 컴포저블만 상태를 수정할 수 있습니다. 철저히 내부적 속성입니다.

공유 가능함: 호이스팅한 상태를 여러 컴포저블과 공유할 수 있습니다.
다른 컴포저블에서 `name`을 읽으려는 경우 호이스팅을 통해 그렇게 할 수 있습니다.

가로채기 가능함: 스테이트리스(Stateless) 컴포저블의 호출자는 상태를 변경하기 전에 이벤트를 무시할지 수정할지 결정할 수 있습니다.

분리됨: 스테이트리스(Stateless) 컴포저블의 상태는 어디에나 저장할 수 있습니다.

상태 호이스팅 예제 코드

```
@Composable
fun GreetingScreen(modifier: Modifier) {
    var name by rememberSaveable { mutableStateOf( value = "") }
    Column(modifier) {
        HelloContent(name = name, onNameChange = { name = it })
        GoodBuyContent(name = name, onNameChange = { name = it })
    }
}

1 Usage
@Composable
fun HelloContent(name: String, onNameChange: (String) -> Unit) {
    Column(modifier = Modifier.padding( all = 16.dp)) {
        Text(
            text = "Hello, $name",
            modifier = Modifier.padding(bottom = 8.dp),
            style = MaterialTheme.typography.bodyMedium
        )
        OutlinedTextField(value = name, onValueChange = onNameChange, label = { Text( text = "Name") })
    }
}

1 Usage
@Composable
fun GoodBuyContent(name: String, onNameChange: (String) -> Unit) {
    Column(modifier = Modifier.padding( all = 16.dp)) {
        Text(text = "GoodBuy, ${name.take( n = 2)}",
            modifier = Modifier.padding(bottom = 8.dp),
            style = MaterialTheme.typography.bodyMedium)
        OutlinedTextField(value = name, onValueChange = {if (name.length < 4) onNameChange }, label = { Text( text = "Name") })
    }
}
```

단일 정보 소스:

GreetingScreen에서 모두 같은 이름

캡슐화됨: name setter 이벤트는 모두 GreetingScreen에서 관리

공유 가능함 ->

name의 변경점을 다른 **composable**과 공유 가능

가로채기 가능함 ->

하위 컴포저블이 상태 표시와 변경에 조건을 걸 수 있음

분리됨 ->

Composable은 **Ui**를 랜더링하는 책임만 가질 수 있게 상태를 분리해서 비즈니스 로직 또는 **Ui** 로직의 책임은 각 **StateHolder**로 책임 분리 가능

상태 호이스팅의 이점

재사용성 :상태를 외부로 올려서 컴포저블이 Stateless로 동작,
동일 컴포저블을 다양한 화면·컨텍스트에서 활용 가능.

테스트 용이성: 상태를 내부에 두지 않고 매개변수에만 의존 -> 입력값만 바꿔도 예측 가능한 테스트 작성 가능.

관심사 분리: 상태 관리는 상위, UI는 화면 렌더링에만 집중. 비즈니스 로직과 UI 코드가 명확히 분리됨.

단방향 데이터 흐름: 상태가 단일 출처에서 관리·전달됨. 다중 소스 충돌로 인한 예측 불가능한 동작을 줄임.

상태 관리 강화: ViewModel·상위 컨테이너에서 상태를 일원화.복잡한 흐름·복원·저장을 체계적으로 처리 가능.

상태 호이스팅 이점 : 재사용성

```kotlin
@Composable
fun MultiCardsSection(
    cards: List<PaymentCardUiModel>,
    modifier: Modifier = Modifier,
) {
    Column(
        verticalArrangement = Arrangement.spacedBy( space = 36.dp),
        modifier = modifier,
    ) {
        Spacer(Modifier.height( height = 12.dp))
        cards.forEach { card ->
            PaymentCard( paymentCardUiModel = card)
        }
    }
}
```

```kotlin
@Composable
fun SingleCardsSection(
    onAddClick: () -> Unit,
    card: PaymentCardUiModel,
    modifier: Modifier = Modifier,
) {
    Column(
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.spacedBy( space = 36.dp),
        modifier = modifier,
    ) {
        Spacer(Modifier.height( height = 12.dp))
        PaymentCard( paymentCardUiModel = card)
        PaymentCreateCard( onAddCardCategoryClick = onAddClick)
    }
}
```

각 화면 부분에서 PaymentCard를 재사용 가능

상태 호이스팅 이점 : 테스트

```
@Test
fun 잘못된_만료일이면_에러메시지가_보인다() {
    val initial =
        CreateCardUiState(
            cardNumber = "",
            expiryDate = "13/22",
            ownerName = "",
            password = "",
            expiryDateErrorTextRes = "월은 01~12여야 합니다",
        )
    setContentWithState(initial)

    val errorText = "월은 01~12여야 합니다"
    rule.onNodeWithText(errorText).assertIsDisplayed()
}
```

상황에 따른 에러 내용을 직접 넣어서 어떤 에러가 나올지 테스트 할 수 있다

상태 호이스팅 이점 : 관심사 분리

```
class CardsStateHolder(
    cardsUiState: CardsUiState,
) {
    5 Usages
    var cardsUiState by mutableStateOf( value = cardsUiState)
        private set

    1 Usage
    fun addCard(cardUiModel: PaymentCardUiModel) {
        val currentCardsUiState = cardsUiState
        cardsUiState = when (currentCardsUiState) {
            is CardsUiState.Multiple -> CardsUiState.Multiple( cards = currentCardsUiState.cards + cardUiModel)
            CardsUiState.None -> CardsUiState.Single( card = cardUiModel)
            is CardsUiState.Single -> CardsUiState.Multiple( cards = listOf(currentCardsUiState.card, cardUiModel))
        }
    }
}
```

로직에 따라 **Ui** 상태의 변경에 관한 관심사를 분리

상태 호이스팅 이점: 상태 관리 강화

```
class CardsStateHolderSaver : Saver<CardsStateHolder, CardsUiState> {
    override fun SaverScope.save(value: CardsStateHolder): CardsUiState? = value.cardsUiState

    override fun restore(value: CardsUiState): CardsStateHolder? = CardsStateHolder( cardsUiState = value)
}
```

```
AndroidpaymentsTheme {
    val cardsStateHolder = rememberSaveable(saver = CardsStateHolderSaver()) {
            CardsStateHolder(CardsUiState.of( paymentCards = paymentCardUiModelSamples))
        }
    val cardAddLauncher = cardAddLauncher(cardsStateHolder)
    Scaffold(
        modifier = Modifier.fillMaxSize(),
        topBar = {
            CardsTopBar(
                cardsUiState = cardsStateHolder.cardsUiState,
                onAddClick = {
                    val intent = CreateCardActivity.instance( context = this)
                    cardAddLauncher.launch( input = intent)
                },
            )
        },
    ) { innerPadding ->
        CardsScreen(
            cardsUiState = cardsStateHolder.cardsUiState,
            onAddClick = {
                val intent = CreateCardActivity.instance( context = this)
                cardAddLauncher.launch( input = intent)
            },
            Modifier
                .padding( paddingValues = innerPadding)
                .fillMaxWidth(),
        )
    }
```

상태를 상위로 올리면서
PaymentsTheme에서 복원 저장

번외: 상태 호이스팅 핵심 규칙

1. 상태는 *적어도* 그 상태를 사용하는 모든 컴포저블의 **가장 낮은 공통 상위 요소**로 끌어올려야 합니다(읽기).
2. 상태는 *최소한* **변경될 수 있는 가장 높은 수준**으로 끌어올려야 합니다(쓰기).
3. **동일한 이벤트에 대한 응답으로 두 상태가 변경**되는 경우 두 상태를 **함께 끌어올려야** 합니다.

UI 상태 종류

- **화면 UI 상태**: 화면에 표시해야 하는 *항목*입니다. 예를 들어 `NewsUiState` 클래스에는 **UI**를 렌더링하는 데 필요한 뉴스 기사와 기타 정보가 포함될 수 있습니다. 이 상태는 앱 데이터를 포함하므로 대개 계층 구조의 다른 레이어에 연결됩니다.

  -> VIewModel 까지의 상태 호이스팅 필요

- **UI 요소 상태**: 렌더링 방식에 영향을 주는 **UI** 요소에 고유한 속성을 나타냅니다. **UI** 요소는 표시하거나 숨길 수 있으며 특정 글꼴이나 글꼴 크기, 글꼴 색상을 적용할 수 있습니다

  -> 공통된 상태를 사용하는 장소까지 호이스팅 ()

애플리케이션 로직 종류

- **비즈니스 로직:** 은 앱 데이터에 대한 제품 요구사항의 구현.

  파일이나 데이터베이스에 저장하는 이 로직은 일반적으로 도메인 또는 데이터 레이어에 배치
  일반적으로 노출되는 메서드를 호출하여 이 로직을 이러한 레이어에 위임합니다.

- **UI 로직**은 화면에 UI 상태를 표시하는 *방법*과 관련.
  올바른 검색창 힌트를 가져오는 것, 특정 항목으로 스크롤,버튼을 클릭할 때 특정 화면으로의 탐색 로직

상태 홀더 및 책임

상태 홀더의 책임은 앱이 읽을 수 있도록 상태를 저장.
로직이 필요한 경우 상태 홀더는 중개자 역할을 하며 필요한 로직을 호스팅하는 데이터 소스에 대한 액세스 권한을 제공합니다

- **간단한 UI**: UI가 상태를 바인딩합니다.
- **유지관리**: 상태 홀더에 정의된 로직을 UI 자체를 변경하지 않고도 반복할 수 있습니다.
- **테스트 가능성**: UI 및 상태 생성 로직을 독립적으로 테스트할 수 있습니다.
- **가독성**: 코드 리더가 UI 표시 코드와 UI 상태 생성 코드 간의 차이점을 명확하게 알아볼 수 있습니다.

상태 홀더 유형

비즈니스 로직 상태 홀더

UI 로직 상태 홀더

구분 기준