derivedStateOf가 필요한 시나리오는 무엇이고,
recomposition 최적화에 어떻게 도움이 되나요?

# derivedStateOf가 무엇인지

설명: 종속 상태 중 하나가 변경될 때만 파생된 값이 다시 계산되도록 보장하여 반응형 상태 관계를 관리하는 효과적인 도구

특징: 종속 상태가 자주 업데이트되더라도 계산된 값 자체가 변경될때만 recomposition을 트리거하여 recomposition을 최적화

주의점: 연산 작업이 요구되기 때문에 약간의 오버헤드를 동반

# derivedStateOf 사용 시기

1. 필터링된 목록이나 결합된 텍스트와 같이 기존 상태 값에서 연산이 필요할 때
   - 원본 State를 그대로 쓰면 UI마다 동일 로직 중복 → 비효율

   - `derivedStateOf`로 파생값을 정의해 연산 결과를 캐시 + 값 변화 시에만 UI 갱신

2. 파생된 값이 자주 변경되는 상태에 의존하지만 파생된 값이 변경될 때만 recomposition을 원할 때 (ex: 카드 정보 입력 검증 후 버튼 활성화)
   - 사용자는 매 키 입력마다 State 변화 발생
   - `derivedStateOf`는 캐시된 값과 비교해, **실제 유효성 결과가 바뀔 때만** 재구성

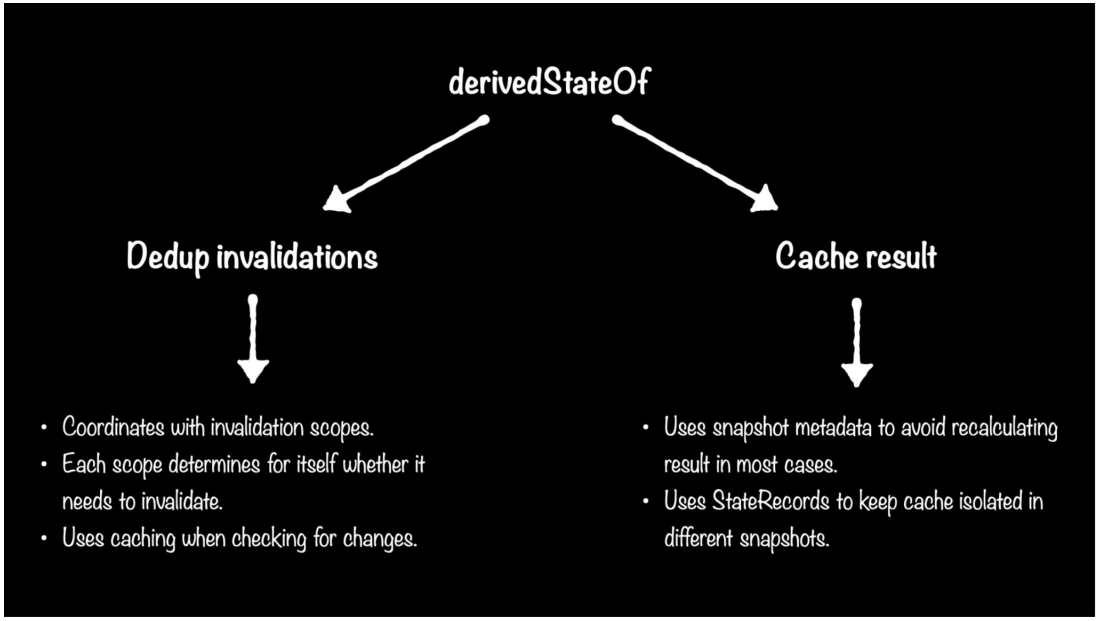
# derivedStateOf: 비교

```kotlin
@StateFactoryMarker
public fun <T> mutableStateOf(
    value: T,
    policy: SnapshotMutationPolicy<T> = structuralEqualityPolicy(),
): MutableState<T> = createSnapshotMutableState(value, policy)
```

```kotlin
@StateFactoryMarker
public fun <T> derivedStateOf(policy: SnapshotMutationPolicy<T>, calculation: () -> T): State<T> =
    DerivedSnapshotState(calculation, policy)
```

각각 반환 값이 다르고 이는 derivedStateOf는 setter가 없고 getter라는 점을 알 수 있음

# derivedStateOf 특징




중복 제거 무효화

- 무효화 범위와 일치합니다.
- 각 범위는 무효화 필요 여부를 스스로 결정합니다.
- 변경 사항 확인 시 캐싱을 사용합니다.

캐시 결과

- 대부분의 경우 결과를 다시 계산하지 않기 위해 스냅샷 메타데이터를 사용합니다

- StateRecords를 사용하여 캐시를 여러 스냅샷에 격리합니다

# Dedup invalidations

### 1. 중복 무효화 제거

- 여러 입력이 동시에 바뀌어도 즉시 여러 번 계산 X
- `invalid = true` 표시만 하고
- 실제 계산은 읽을 때 단 1번만 실행 -> val isValid by derivedStateOf { username.isNotBlank() && password.length > 7 }

### 2. Scope 단위 무효화

- 파생 상태를 사용하는 Composable scope만 다시 그림
- 전체 UI 재구성이 아닌 필요한 부분만 갱신

### 3. 캐싱 + 값 비교

- 마지막 결과를 캐시에 저장
- 새 결과와 같으면 재구성 스킵
- 결과가 달라질 때만 갱신 → 불필요한 연산 최소화

# derivedStateOf 캐시 결과 관리

### 1. 스냅샷 메타데이터 활용

- 대부분의 경우 재계산 불필요
- 마지막 계산 결과와 메타데이터 비교
- 값이 같으면 그대로 반환 → 불필요한 연산 회피

### 2. StateRecords로 스냅샷 격리

- Compose는 여러 스냅샷을 동시에 다룰 수 있음
- 각 스냅샷마다 독립된 캐시(StateRecord) 유지
- 병렬/과거 상태 간섭 없이 안정적으로 동작

# 명심해야 할 주요 사항

• 파생된 상태가 recomposition으로부터 값을 유지하도록 하려면 항상 remember와 함께

derivedStateOf를 사용

• 원활한 성능을 보장하기 위해 derivedStateOf에 아주 무거운 연산식을 사용하지 않는

것이 좋습니다. 오히려 해당 계산을 통한 오버헤드가 recomposition을 통해 UI를

업데이트하는 비용보다 더 크면 성능에 역효과가 날 수 있습니다.

참고 영상

참고 블로그