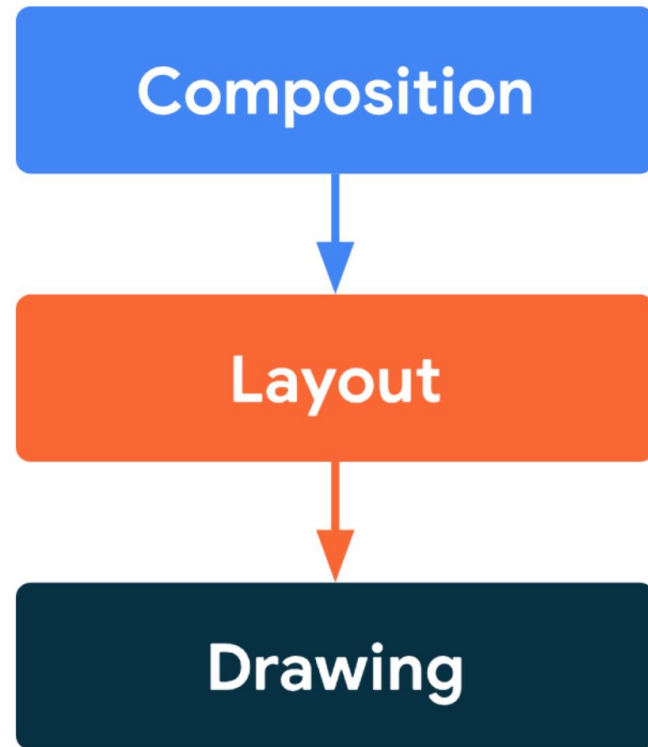
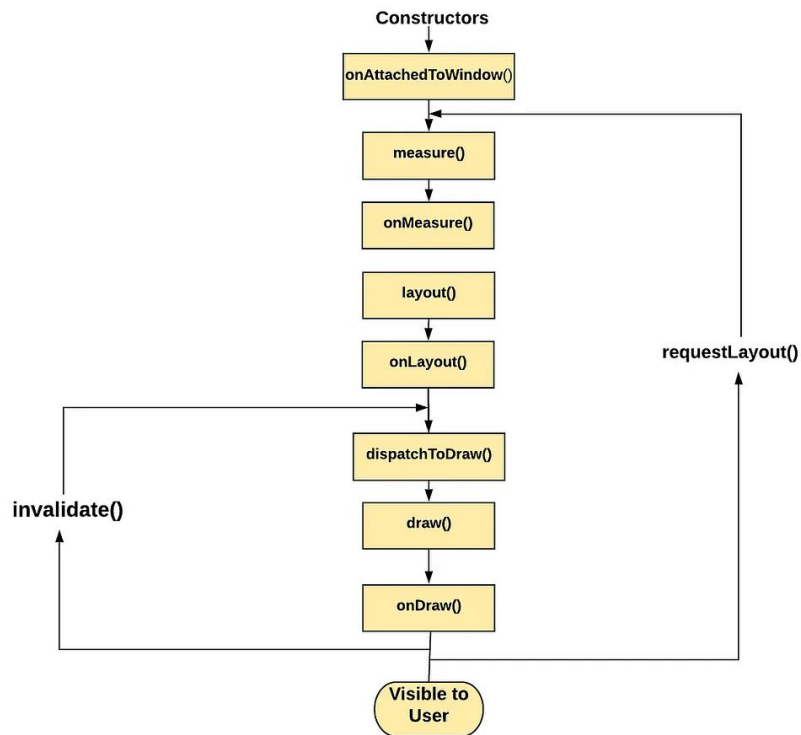


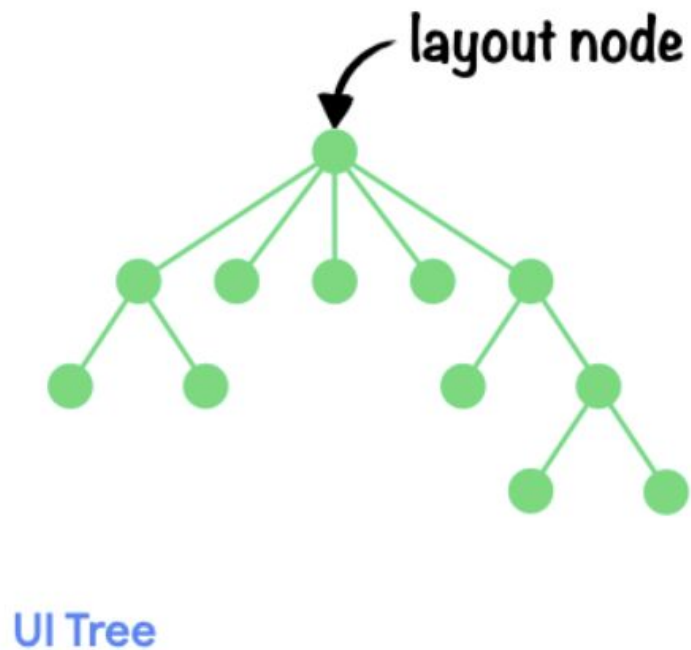
Compose Phases

View vs Compose

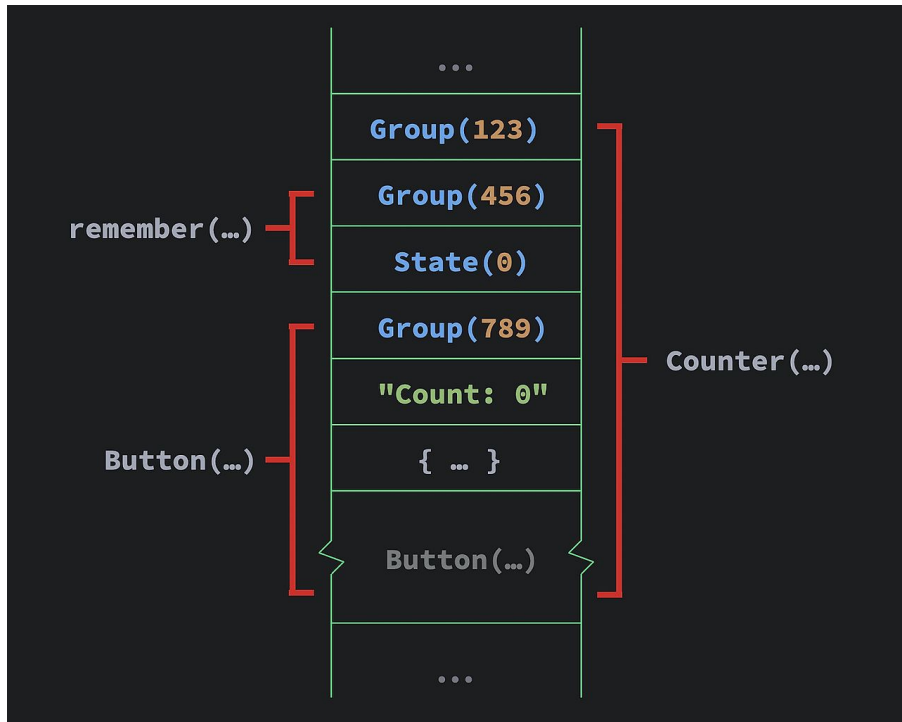


Composition

```
Column {  
  Row {  
    Icon(..)  
    Title(..)  
  }  
  HeaderImage(..)  
  Title(..)  
  Subtitle(..)  
  Row {  
    Image(..)  
    Column {  
      Text(..)  
      Text(..)  
    }  
  }  
}
```



Composition



SlotTable에 저장 (Gap Buffer)

Gap 이동 $\rightarrow O(n)$

그 외의 연산 $\rightarrow O(1)$

만약 이전과 입력값이 동일하다면,
이전의 결과를 그대로 반환한다.
(Positional Memoization)

입력이 안정적이고 변경되지 않은 경우,
컴포지션을 건너뛰다.

Composition

안정적인 유형으로 간주되려면 다음 계약을 준수해야 합니다.

- 두 인스턴스의 `equals` 결과가 동일한 두 인스턴스의 경우 *항상* 동일합니다.
- 유형의 공개 속성이 변경되면 컴포지션에 알림이 전송됩니다.
- 모든 공개 속성 유형도 안정적입니다.

`@Stable` 주석을 사용하여 안정적이라고 명시되지 않더라도 Compose 컴파일러가 안정적인 것으로 간주하며 이 계약에 포함되는 중요한 일반 유형이 있습니다.

- 모든 원시 값 유형: `Boolean`, `Int`, `Long`, `Float`, `Char` 등
- 문자열
- 모든 함수 유형 (람다)

Composition

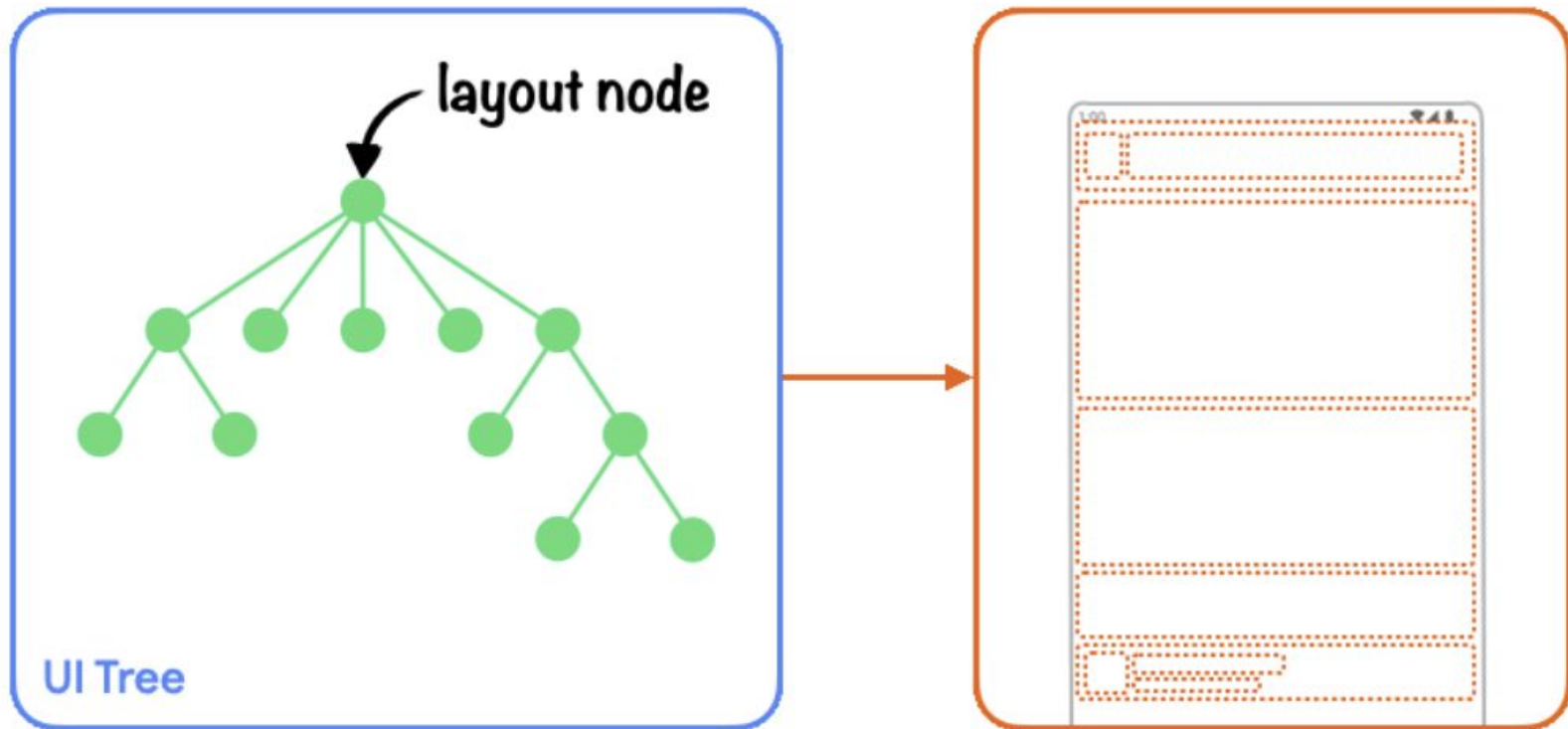
안정적이지만 변경할 수 있는 한 가지 중요한 유형은 Compose의 `MutableState` 유형입니다. 값이 `MutableState` 로 유지되는 경우 `State` 의 `.value` 속성이 변경되면 Compose에 알림이 전송되므로 상태 객체는 전체적으로 안정적인 것으로 간주됩니다.

**State 객체 내부의 value가 변경될 경우, 해당 객체를 읽었던 모든 컴포저블(스코프) 탐색
→ Compose Runtime이 알아서 가장 작은 범위만 갱신**

```
// Marking the type as stable to favor skipping and smart recompositions.
@Stable
interface UiState<T : Result<T>> {
    val value: T?
    val exception: Throwable?

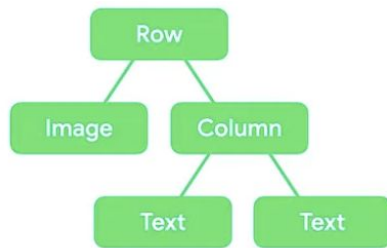
    val hasError: Boolean
        get() = exception != null
}
```

Layout



Layout

```
Row {  
  Image(..)  
  Column {  
    Text(..)  
    Text(..)  
  }  
}
```




레이아웃 단계에서 트리는 다음 3단계 알고리즘을 사용하여 순회합니다.

1. **하위 요소 측정:** 노드가 하위 요소를 측정합니다(있는 경우).
2. **자체 크기 결정:** 이러한 측정값을 기반으로 노드가 자체 크기를 결정합니다.
3. **하위 요소 배치:** 각 하위 노드는 노드의 자체 위치를 기준으로 배치됩니다.

Layout

레이아웃 단계는 측정과 배치라는 두 단계로 구성됩니다. 측정 단계에서는 `Layout` 컴포저블에 전달된 측정 람다와 `LayoutModifier` 인터페이스의 `MeasureScope.measure` 메서드 등을 실행합니다. 배치 단계에서는 `layout` 함수의 배치 블록과 `Modifier.offset { ... }`의 람다 블록, 유사한 함수를 실행합니다.

이러한 각 단계의 상태 읽기는 레이아웃에 영향을 미치고 그리기 단계에도 영향을 줄 수 있습니다. 상태의 `value`가 변경되면 Compose UI는 레이아웃 단계를 예약합니다. 크기나 위치가 변경된 경우 그리기 단계도 실행합니다.

 **요점:** 측정 단계와 배치 단계의 다시 시작 범위는 별개이므로 배치 단계의 상태 읽기가 그 전 측정 단계를 다시 호출하지 않습니다. 그러나 이 두 단계는 서로 관련된 경우가 많으므로 배치 단계의 상태 읽기가 측정 단계에 속하는 다른 다시 시작 범위에 영향을 미칠 수 있습니다.

Layout

```
Box {  
    val listState = rememberLazyListState()  
  
    Image(  
        // ...  
        // Non-optimal implementation!  
        Modifier.offset(  
            with(LocalDensity.current) {  
                // State read of firstVisibleItemScrollOffset in composition  
                (listState.firstVisibleItemScrollOffset / 2).toDp()  
            }  
        )  
    )  
  
    LazyColumn(state = listState) {  
        // ...  
    }  
}
```

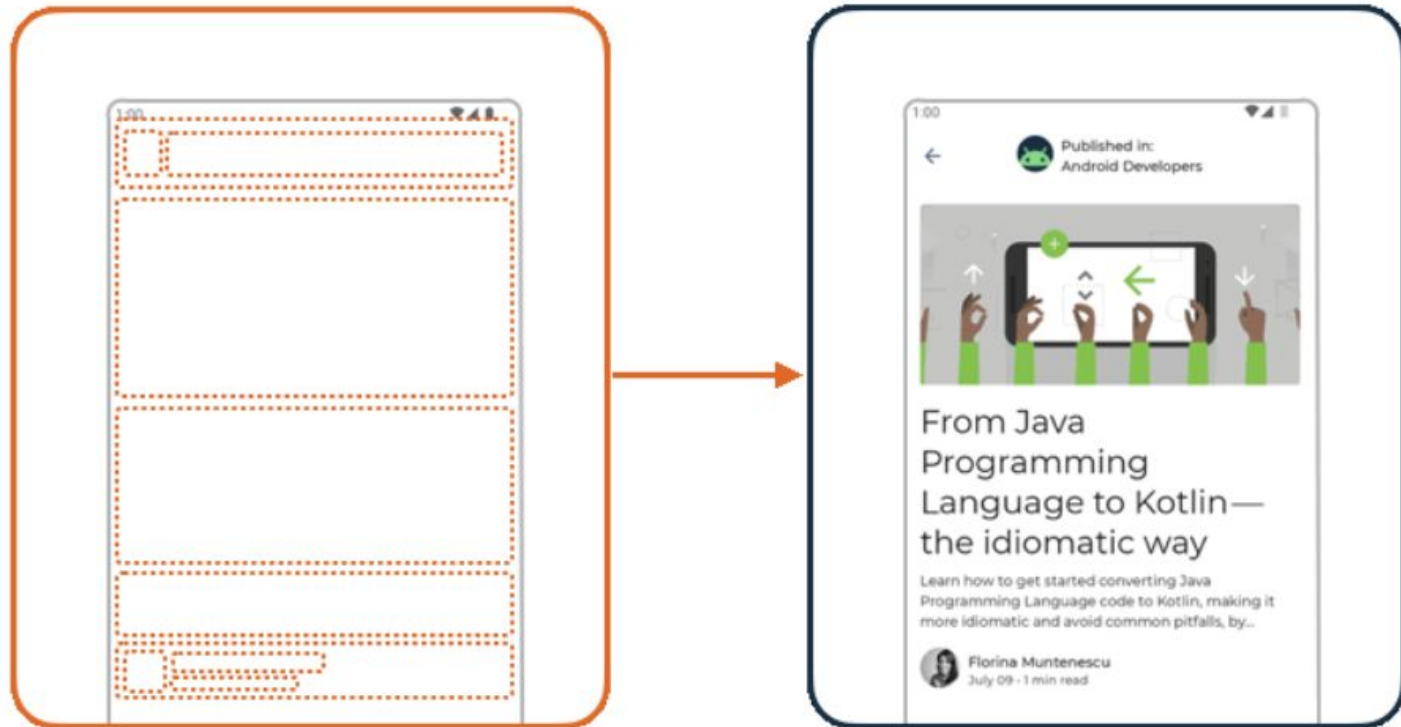
Layout

```
Box {  
    val listState = rememberLazyListState()  
  
    Image(  
        // ...  
        Modifier.offset {  
            // State read of firstVisibleItemScrollOffset in Layout  
            IntOffset(x = 0, y = listState.firstVisibleItemScrollOffset / 2)  
        }  
    )  
  
    LazyColumn(state = listState) {  
        // ...  
    }  
}
```

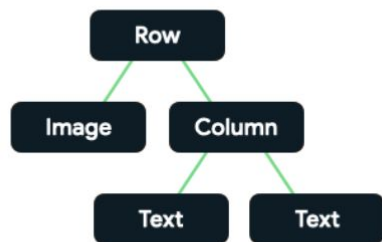
[PhasesSnippets.kt](https://github.com/PhasesSnippets) 

Modifier에 제공하는 랩다는 Layout 단계에서 호출하기 때문

Drawing



Drawing



Florina Muntenescu

Jul 2 · 1 min read

1. `Row` 는 배경색과 같은 콘텐츠를 그립니다.
2. `Image` 가 자체적으로 그려집니다.
3. `Column` 가 자체적으로 그려집니다.
4. 첫 번째와 두 번째 `Text` 는 각각 자체적으로 그려집니다.

Phases

State reads

