

안드로이드

앱 프로그래밍

with 코틀린 & 컴포즈



0

제트팩 컴포즈



22-1 컴포즈 이해하기

22-2 상태 다루기

22-3 컴포즈로 화면 구성하기

22-4 컴포즈로 뉴스 앱 만들기

22-1 컴포즈 이해하기

컴포즈란?

<< Modern Android UI Development >>

- 컴포즈(compose) UI란? 구글 제트팩에서 제공하는 최신 기술로 앱의 화면을 구성하는 방법
- 컴포즈의 사용 목적은 선언형 UI 프로그래밍과 상태 관리
- 레거시 뷰 시스템과 컴포즈는 상호 연동은 가능

- ❖ 처음 안드로이드 앱을 개발한다면, Kotlin + Compose로 개발해야.
- ❖ 이유는 구글이 Kotlin + Compose을 지속 투자하고 있기 때문.



Inspired By

- React (상태관리 및 함수형 UI framework)
- Vue.js
- Swift UI (iPhone의 Object-C를 대체한 IOS 공식 앱 개발 언어)
- Flutter (Dart언어, 안드로이드+IOS cross platform 앱개발)
- Litho (A declarative UI framework for Android by FaceBook)

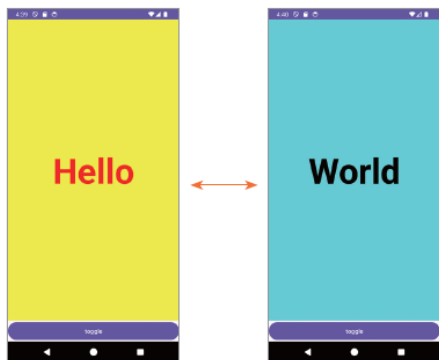
→ 코틀린 언어도 Java를 대체하는 언어이며 구글이 밀고 있음.

→ Compose UI도 기존 XML 기반 뷰 시스템을 대체함.

22-1 컴포즈 이해하기 - 장점

[장점 #1] 선언형 UI 프로그래밍

- 선언형 UI 프로그래밍(Declarative UI Programming)은 명령형 UI 프로그래밍(Imperative UI Programming)과 대치되는 개념
- 명령형 UI 프로그래밍은 개발자 코드에서 UI 구성과 관련된 모든 코드를 작성하는 방식
- 명령형 UI로 화면을 구성하게 되면 화면 출력을 위한 코드가 길어지게 되며, 개발자는 화면과 관련된 많은 API를 알고 있어야 한다.



• 레이아웃 XML 파일(명령형 UI 프로그래밍으로 작성한 예)

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <TextView
        android:id="@+id/textView"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:text="Hello"
        android:textSize="80sp"
        android:gravity="center"
        android:textColor="#FF0000"
        android:background="#FFFF00"
        android:textStyle="bold"/>
```

• 버튼 클릭 시 화면 변경 처리(명령형 UI 프로그래밍으로 작성한 예)

```
binding.button.setOnClickListener {
    binding.textView.run {
        if(text == "Hello"){
            text = "World"
            setTextColor(Color.BLACK)
            setBackgroundColor(Color.CYAN)
        }else {
            text = "Hello"
            setTextColor(Color.RED)
            setBackgroundColor(Color.YELLOW)
        }
    }
}
```

22-1 컴포즈 이해하기 - 장점

[장점 #1] 선언형 UI 프로그래밍

- 선언형 UI 프로그래밍은 화면에 어떻게 출력 혹은 구성되어야 한다는 정보만 선언하는 방식
- AI를 활용한 앱 개발에 유리하다. (Compose UI 핵심 문맥으로 AI에게 코드 작성 요청하면 된다.)

• 버튼 클릭 시 화면 변경 처리(선언형 UI 프로그래밍으로 작성한 예)

```
var textState by remember { mutableStateOf("Hello") }
var textColorState by remember { mutableStateOf(Color.Red) }
var textBackgroundColorState by remember { mutableStateOf(Color.Yellow) }
Column {
    Text(
        textState,
        fontSize = 40.sp,
        fontWeight = FontWeight.Bold,
        color = textColorState,
        textAlign = TextAlign.Center,
        modifier = Modifier
```

```
        .fillMaxWidth()
        .weight(1f)
        .background(textBackgroundColorState)
        .wrapContentHeight(align = Alignment.CenterVertically)
    )
    Button(modifier = Modifier.fillMaxWidth(), onClick = {
        if (textState == "Hello") {
            textState = "World"
            textColorState = Color.Black
            textBackgroundColorState = Color.Cyan
        } else {
            textState = "Hello"
            textColorState = Color.Red
            textBackgroundColorState = Color.Yellow
        }
    }) {
        Text("toggle")
    }
}
```

22-1 컴포즈 이해하기 - 장점

[장점 #1] 선언형 UI 프로그래밍

Imperative UI

VS

Declarative UI

Focuses on *How*

Inheritance is preferred

View and *logic* are separate

Tons of *Boilerplate Code*,
slow development

Focuses on *What*

Composition is preferred

Uses *one* language to build
entire application.

Less *Code, faster*
development

22-1 컴포즈 이해하기 - 장점

[장점 #2] 한 언어로 일관된 UI 개발

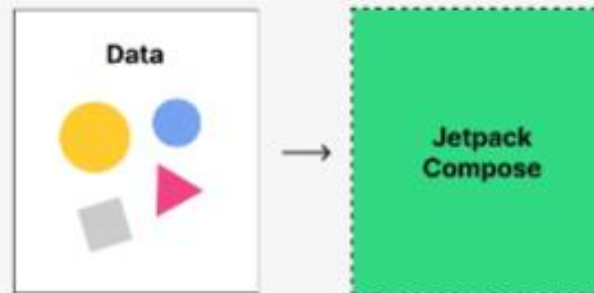
- XML view와 UI logic 사이를 오가는 번거로움이 사라졌다. 코틀린으로 UI 구성과 상태 data 처리를 모두 한다.
- AI를 활용한 앱 개발에 유리하다. (Compose UI 구성을 위한 코틀린 코드 작성을 AI에게 요청하면 된다.)

기존 XML 형태의 명령형 UI 동작 방식



- 상태 관리에 취약
- UI코드 작성시, 코드 분산으로 인해 유지보수가 힘들
- 디버깅에 취약

Jetpack Compose의 선언형 UI 동작 방식



XML을 버린 Jetpack Compose 동작 방식

- 상태 관리에 용이
- 선언형 방식을 통해 가독성이 높아져 유지보수에 용이
- 높은 재사용성을 가진 코드 작성 가능

22-1 컴포즈 이해하기 - 장점

[장점 #3] 코드가 간결하다.

- AI를 활용한 앱 개발에 유리하다. (소모하는 토큰 양이 작다)

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    android:padding="16dp">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, Jetpack Compose!"
        android:textColor="@android:color/black"
        android:textSize="24sp" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="This is a simple example of using Jetpack Compose to build a user interface."
        android:textColor="@android:color/darker_gray"
        android:textSize="16sp" />

</LinearLayout>
```

```
@Composable
fun SimpleUI() {
    MaterialTheme {
        Column(modifier = Modifier.padding(16.dp)) {
            Text(
                text = "Hello, Jetpack Compose!",
                style = MaterialTheme.typography.h1,
                color = Color.Black
            )
            Text(
                text = "This is a simple example of using Jetpack Compose to build a user interface.",
                style = MaterialTheme.typography.body1,
                color = Color.Gray
            )
        }
    }
}
```

22-1 컴포즈 이해하기 - 장점

[장점 #4] Reusability and Modularity

- 레고 조립하듯 Composables UI 요소들을 재활용 해가며 복잡한 UI로 만들어 갈 수 있다.

[장점 #5] Animations and Transitions

- Compose provides a modern and flexible animation system that is easier to implement than XML animations.
- 게임 개발에 적용

[장점 #6] Preview and Development Speed

- 미리보기 (preview)를 할 수 있어, 코드 수정 후 바로 효과를 확인할 수 있다. 개발이 빨라지고 쉬워진다.

22-1 컴포즈 이해하기 - 장점

[장점 #7] 용이한 상태 관리

- 상태는 데이터입니다.
- 상태는 화면에 출력되면서 다양한 이유(사용자 이벤트, 서버 네트워킹 등)로 변경되고, 상태 data가 변경되면, 컴포즈가 이를 인식하고 자동으로 화면을 갱신시켜 준다.
- 일관된 방식으로 상태 관리를 할 수 있다. 일정한 코드 패턴이 있어, AI 활용에도 유리하다.

22-1 컴포즈 이해하기 - 프로젝트

기존 XML 뷰 대비 컴포즈의, 프로젝트 구조 차이

❖ Compose의 철학:

"UI를 변수나 함수처럼 다루자" - 선언형 + 코틀린 네이티브 장점 극대화

1. 레이아웃 파일 제거

- XML → Kotlin 컴포저블 함수로 대체
- `setContentView(R.layout.activity_main)` → `setContent { MyApp() }`

2. 테마 시스템 혁신

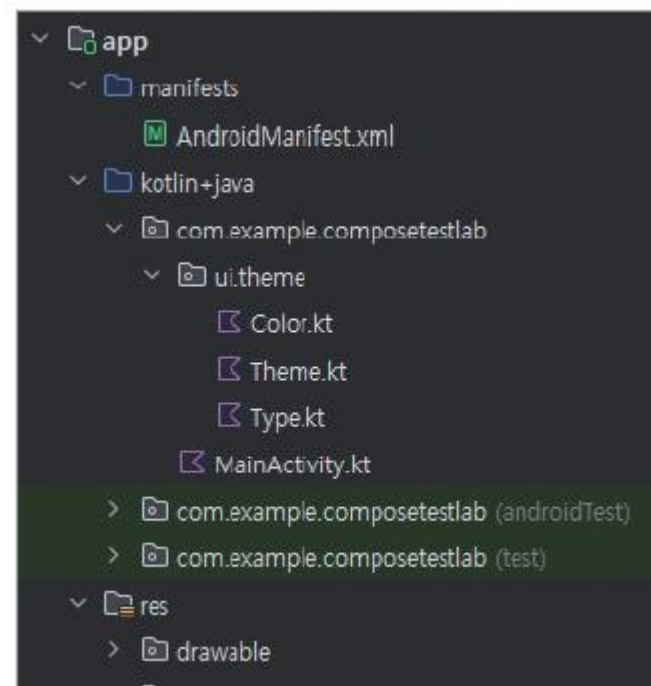
- `res/values/themes.xml` → `ui/theme/*.kt` (색상, 서체, 모서리, Material 테마)
- 정적 리소스 → 동적 Kotlin 객체 (런타임에 테마 속성 실시간 변경 가능)
- 완전한 코드 기반 테마: XML(`themes.xml`) 대신 Kotlin 객체로 관리

3. UI 컴포넌트 관리

- 뷰: XML `<include>` → 컴포즈: Kotlin 함수 호출 (`@Composable Component()`)
- 재사용성이 코드 수준에서 명시적

4. 리소스 공통점

- 아이콘(`res/drawable`), 문자열(`res/values/strings.xml`) 등은 동일하게 사용



22-1 컴포즈 이해하기 - 빌드 설정

build.gradle.kts(프로젝트 수준)

- 컴포즈는 코틀린으로 개발해야 하므로 자동으로 코틀린과 관련된 플러그인들이 추가된다.

// Top-level build file where you can add configuration options common to all sub-projects/modules.

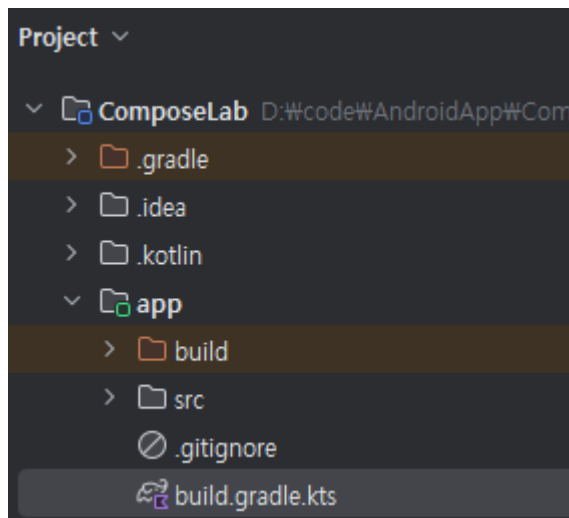
• 자동으로 추가된 코틀린 관련 플러그인 확인

```
plugins {  
    alias(libs.plugins.android.application) apply false  
    alias(libs.plugins.kotlin.android) apply false  
    alias(libs.plugins.kotlin.compose) apply false  
}
```

22-1 컴포즈 이해하기 - 빌드 설정

build.gradle.kts(모듈 수준)

- 컴포즈를 사용한다는 선언이 android 영역에 자동으로 추가된다.
- Dependencies 부분에 컴포즈를 사용하기 위한 라이브러리가 자동으로 추가되어 있다.



• 컴포즈 사용 선언과 컴포즈 관련 라이브러리 추가

```
android {  
  
    buildFeatures {  
        compose = true  
    }  
}  
  
dependencies {  
  
    implementation(libs.androidx.core.ktx)  
    implementation(libs.androidx.lifecycle.runtime.ktx)  
    implementation(libs.androidx.activity.compose)  
    implementation(platform(libs.androidx.compose.bom))  
    implementation(libs.androidx.ui)  
    implementation(libs.androidx.ui.graphics)  
    implementation(libs.androidx.ui.tooling.preview)  
    implementation(libs.androidx.material3)  
    (... 생략 ...)  
}
```

22-1 컴포즈 이해하기 - 기본 라이브러리

build.gradle.kts(모듈 수준)에 기본으로 포함되는 라이브러리 설명

Dependency	Description
libs.androidx.activity.compose	Compose를 Activity와 통합하기 위한 핵심 도구. Activity의 onCreate()에서 Compose UI 트리의 루트로 설정하는 setContent()를 지원합니다.
libs.androidx.lifecycle.runtime.ktx	Lifecycle Aware Component를 지원하기 위한 코어 라이브러리. ViewModel/LiveData 없이 순수 Lifecycle 관리 기능을 제공합니다.
libs.androidx.ui	화면 구성 기본 요소 제공 라이브러리. Row, Column, Box, ConstraintLayout 등의 레이아웃 컴포넌트와 사이즈/배치 기능을 포함합니다.
libs.androidx.ui.graphics	저수준 그래픽 기능 제공 라이브러리. Canvas API, 벡터 드로잉, 이미지 렌더링 등의 그래픽 연산을 지원합니다.
libs.androidx.ui.tooling.preview	Android Studio 내 Compose 미리보기 기능 제공 라이브러리. @Preview 어노테이션을 통한 실시간 UI 프리뷰를 지원합니다.
libs.androidx.material3	Google Material Design 3 구현 라이브러리. 머티리얼 테마 시스템과 버튼/카드/네비게이션 바 등 UI 컴포넌트를 제공합니다.

22-1 컴포즈 이해하기

build.gradle.kts(모듈 수준)

- `libs.androidx.activity.compose`: 컴포즈를 이용하는 액티비티를 지원하기 위한 라이브러리
- `libs.androidx.lifecycle.runtime.ktx`: 제트팩의 Lifecycle Aware Component 부분을 지원하기 위한 라이브러리입니다. lifecycle로 시작하는 많은 라이브러리들이 있는데 그중 lifecycle-runtime은 ViewModel, LiveData가 포함되지 않은 라이브러리이며 코어 Lifecycle Aware Component 부분을 지원하기 위한 라이브러리입니다.
- `libs.androidx.ui`: 사이즈, 배치 등 기본적으로 화면을 구성할 수 있는 요소들을 제공하는 라이브러 리입니다.
- `libs.androidx.ui.graphics`: 다양한 그래픽 기능을 제공하는 라이브러리입니다.
- `libs.androidx.ui.tooling.preview`: 컴포즈로 개발한 코드 미리보기 기능을 제공하는 라이브 러리입니다.
- `libs.androidx.material3`: 머티리얼 디자인이 적용된 화면 구성을 지원하는 라이브러리입니다.

22-1 컴포즈 이해하기

MainActivity.kt

- 컴포즈를 이용하는 액티비티는 `ComponentActivity`를 상속
- 화면 출력을 `setContent()` 함수를 이용
- `setContent()`의 매개변수에 지정된 `AndroidLabTheme()`, `Surface()`, `Greeting()` 등은 컴포저블 `composable` 함수라고 부르며 화면을 구성하는 역할
- 컴포즈로 프로그램을 작성한다는 것은 컴포저블 함수를 만드는 것을 의미
- 컴포저블 함수는 `@Composable` 어노테이션으로 선언되는 함수

• 자동으로 생성되는 메인 액티비티 확인

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            AndroidLabTheme {
                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
                    Greeting(
                        name = "Android",
                        modifier = Modifier.padding(innerPadding)
                    )
                }
            }
        }
    }
}
```

• Greeting 컴포저블 함수 선언

```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Text(
        text = "Hello $name!",
        modifier = modifier
    )
}
```

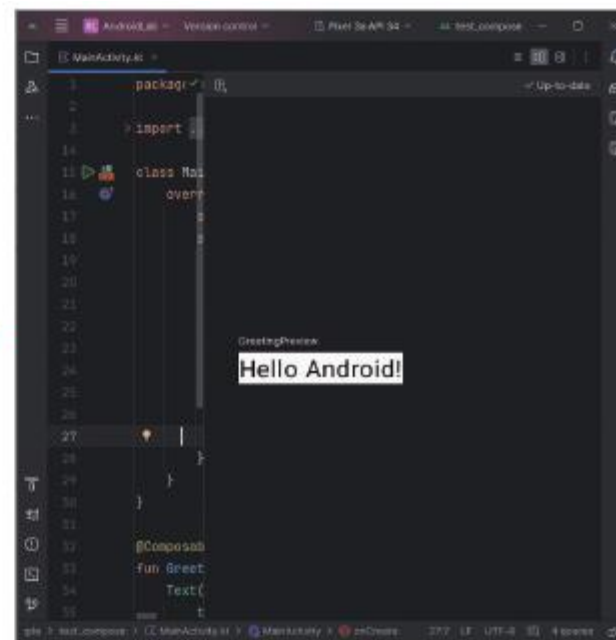
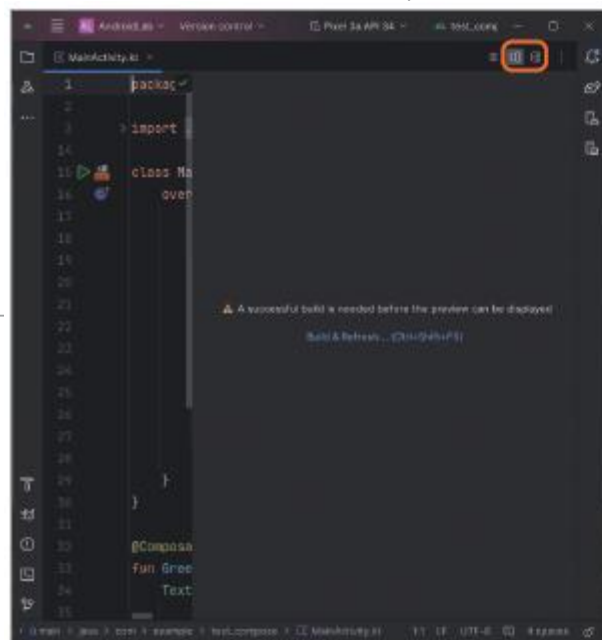
22-1 컴포즈 이해하기

MainActivity.kt

- GreetingPreview()는 @Preview 어노테이션으로 선언되며 실제 앱이 빌드되어 실행될 때의 화면 출력을 목적으로 하지 않고 안드로이드 스튜디오의 preview 화면에 출력될 내용을 표현하기 위한 컴포저블 함수

• GreetingPreview 컴포저블 함수 선언

```
@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    AndroidLabTheme {
        Greeting("Android~")
    }
}
```



22-1 컴포즈 이해하기

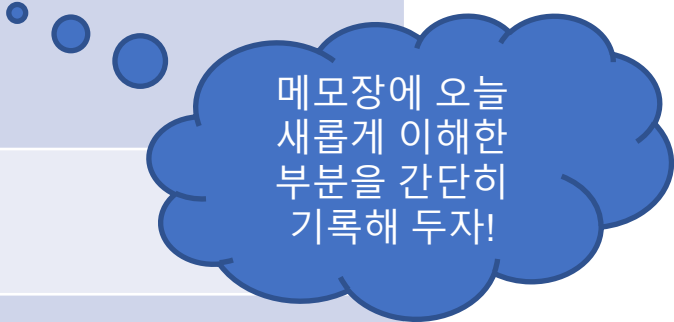
컴포저블 함수 좀 더 알기

구분	설명
정의	레고 블록과 같은 UI 빌딩 블록 Jetpack Compose의 핵심 구성 요소
기능	<code>Text()</code> , <code>Button()</code> 같은 기본 제공 컴포넌트를 조합해 커스텀 UI 화면 구성 (compile 되서 UI 트리로 만들어진다.)
작성 규칙	<code>@Composable</code> // 필수 어노테이션 <pre>fun MyComponent() { Text("Hello Compose!") // UI 요소 배치 }</pre>
호출 제한, 리턴값	컴포저블 함수는 다른 컴포저블 함수 안에서만 호출 가능 일반 함수에서는 호출 불가 (컴파일 오류 발생)
	리턴값을 가질 수 있지만 UI 구성에 사용되지 않음

22-1 컴포즈 이해하기

컴포저블 함수 좀 더 알기

구분	설명
정의	레고 블록과 같은 UI 빌딩 블록 Jetpack Compose의 핵심 구성 요소
기능	Text(), Button() 같은 기본 제공 컴포넌트를 조합해 커스텀 UI 화면 구성 (compile 되서 UI 트리로 만들어진다.)
작성 규칙	@Composable // 필수 어노테이션 fun MyComponent() { Text("Hello Compose!") // UI 요소 배치 }
호출 제한, 리턴값	컴포저블 함수는 다른 컴포저블 함수 안에서만 호출 가능 일반 함수에서는 호출 불가 (컴파일 오류 발생) 리턴값을 가질 수 있지만 UI 구성에 사용되지 않음



메모장에 오늘
새롭게 이해한
부분을 간단히
기록해 두자!

제트팩 컴포즈

22-1 컴포즈 이해하기

22-2 상태 다루기

22-3 컴포즈로 화면 구성하기

22-4 컴포즈로 뉴스 앱 만들기

22-2 상태 다루기

상태란?

- 컴포저블 함수는 함수 내에 상태가 선언되어 어떻게 사용되는지에 따라 **상태** 컴포저블(Stateful Composable)과 **비상태** 컴포저블(Stateless Composable)로 구분

➤ mutableStateOf 함수로 상태 선언

- 변수 자체를 상태라고 하지는 않습니다.

변수 선언의 예

```
var data1 = 0
```

- ❖ 사용자 이벤트 처리는 상태 관리와 결합하여 인터랙티브한 UI를 만든다.
- ❖ `onClick` 같은 이벤트에 람다 함수를 붙이고, 람다 함수에서 상태를 변경하면, UI가 변경을 반영한다.

```
var count by remember { mutableStateOf(0) }  
Button {  
    onClick = {  
        count++  
        Log.d("ClickEvent", "버튼이 클릭되었습니다.! 현재 값: $count")  
    }  
}{  
    Text("클릭 횟수: $count")  
}
```

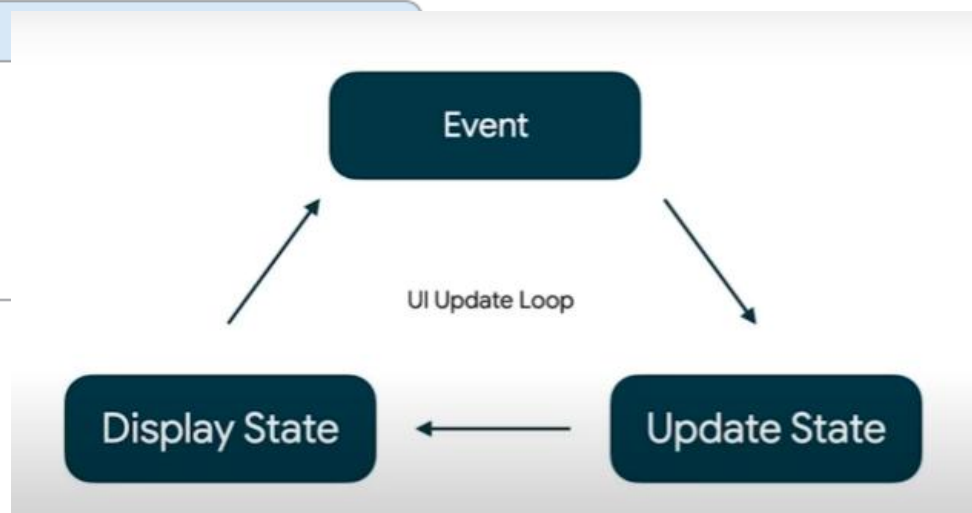
22-2 상태 다루기

mutableStateOf 함수로 상태 선언

- 컴포저블 함수 내에서 상태 선언은 다음과 같이 mutableStateOf 함수로 합니다.
- mutableStateOf()의 매개변숫값(여기서는 0)은 선언된 상태의 초기값
- 첫 번째는 상태를 이용하기 위한 변수이며 두 번째는 상태를 변경하기 위한 함수

• mutableStateOf 함수로 상태 선언

```
var (data2, setData2) = mutableStateOf(0)
fun changeData2() {
    setData2(Random.nextInt(0, 100))
}
```



22-2 상태 다루기

상태란?

- 컴포저블 함수는 함수 내에 상태가 선언되어 어떻게 사용되는지에 따라 **상태** 컴포저블(Stateful Composable)과 **비상태** 컴포저블(Stateless Composable)로 구분

➤ mutableStateOf 함수로 상태 선언

- 변수 자체를 상태라고 하지는 않습니다.

변수 선언의 예

```
var data1 = 0
```


22-2 상태 다루기

mutableStateOf 함수로 상태 선언

- 컴포저블 함수에서 다음과 같이 mutableStateOf 함수로 상태 선언
- mutableStateOf()의 매개변숫값(여기서는 0)은 선언된 상태의 초기값
- 상태를 이용하기 위한 변수 선언, 이어서 상태를 변경하기 위한 함수

• mutableStateOf 함수로 상태 선언

```
var (data2, setData2) = mutableStateOf(0)
fun changeData2() {
    setData2(Random.nextInt(0, 100))
}
```

- setter 함수를 호출하면 상태가 변경되고 화면이 재구성
- 상태값은 변경되고 컴포저블 함수가 다시 호출되기는 하지만 상태값은 항상 0
- 상태값을 변경해서 재구성이 이뤄지지만 변경된 상태값이 유지되지 않는 문제가 있다.

상태값을 유지하려면 remember 함수를 사용한다.

- 컴포저블이 재구성되어도 상태값을 유지되게 하려면 remember 함수를 같이 사용해야 합니다.
- by를 함께 사용하면 remember에 의해 상태가 유지 된다.

• remember 함수를 사용한 예(by를 함께 사용한 예)

```
var data3 by remember {
    mutableStateOf(0)
}
fun changeData3() {
    data3 = Random.nextInt(0, 100)
}
```

22-2 상태 다루기

remember 함수를 사용해 상태값 유지

- By를 사용하지 않고 remember 이용하면 MutableState 객체 반환
- MutableState는 상태의 이용 또는 변경을 감지하는 컴포즈의 클래스
- 상태값을 획득하거나 변경하기 위해서는 이 객체의 value 프로퍼티를 사용

• remember 함수를 사용한 예(by를 사용하지 않은 예)

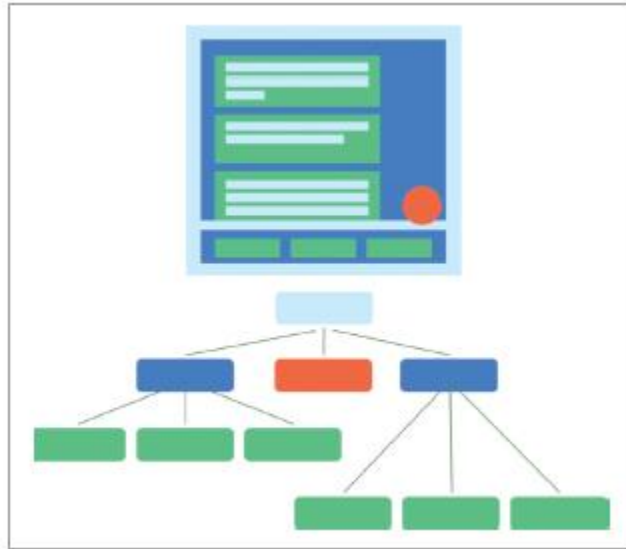
```
val data4: MutableState<Int> = remember {  
    mutableStateOf(0)  
}  
  
fun changeData4() {  
    data4.value = Random.nextInt(0, 100)  
}
```

- ✓ remember는 상태 객체의 생명주기 관리
- ✓ mutableStateOf()는 관찰 가능한 상태 컨테이너 생성
- ✓ .value는 MutableState 객체의 상태 읽기/쓰기의 유일한 통로
- ✓ 재구성 시 remember가 없으면 상태 초기화되는 문제 발생 (이전 질문에서 설명한 현상)

22-2 상태 다루기

단방향 데이터 흐름

- 상태 데이터의 단방향 흐름 Unidirectional Data Flow 은 컴포저블 함수의 계층 구조 측면에서 이야기하자면 상위 컴포저블의 상태를 하위 컴포저블이 사용 만 하는 것이지 직접 변경할 수 없다는 의미입니다.
- 데이터는 위에서 아래로 한 방 향으로만 흐른다는 것이 바로 단방향 데이터 흐름입니다.



22-2 상태 다루기

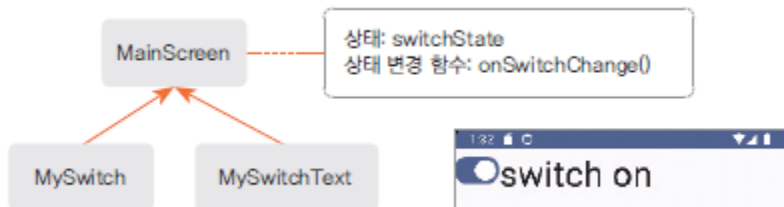
- 하위 컴포저블에서 상위 컴포저블에 선언된 상태를 변경하고 싶다면 상위 컴포저블 함수를 전달받아 그 함수를 호출해 상위 컴포저블에서 변경되도록 코드를 작성.

• 상위 컴포저블의 상태 변경

```
@Composable
fun MainScreen() {
    var switchState by remember {
        mutableStateOf(true)
    }
    val onSwitchChange = { value: Boolean -> switchState = value}
    Row {
        MySwitch(switchState = switchState, onSwitchChange = onSwitchChange)
        MySwitchText(switchState = switchState)
    }
}
```

```
@Composable
fun MySwitch(switchState: Boolean, onSwitchChange: (Boolean) -> Unit){
    Switch(checked = switchState, onCheckedChange = onSwitchChange)
}

@Composable
fun MySwitchText(switchState: Boolean) {
    Text(when(switchState) {
        true -> "switch on"
        else -> "switch off"
    }, fontSize = 40.sp)
}
```



22-2 상태 다루기

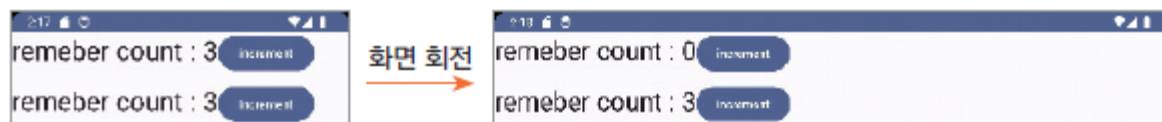
rememberSaveable 함수

- 화면 회전과 같은 액티비티의 상태 변경에도 상태가 그대로 유지되게 하고 싶은 경우 rememberSaveable 함수를 이용해 상태를 선언

• rememberSaveable 함수로 상태 선언

```
var aData by remember {
    mutableStateOf(0)
}
var bData by rememberSaveable {
    mutableStateOf(0)
}
Column {
    Row {
        Text("remeber count : ${aData.toString()}", fontSize = 30.sp)
        Button(onClick = { aData++ }) {
            Text(text = "increment")
        }
    }
    Spacer(modifier = Modifier.height(10.dp))
    Row {
        Text("remeber count : ${bData.toString()}", fontSize = 30.sp)
        Button(onClick = { bData++ }) {
```

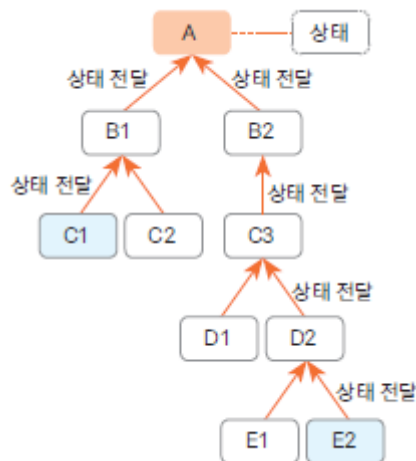
```
            Text(text = "increment")
        }
    }
}
```



22-2 상태 다루기

ProvidableCompositionLocal 객체 이용

- 여러 컴포저블 함수에서 필요한 상태를 공통의 조상 함수(상위 함수)에 선언하고 하위 함수인 컴포저블 함수를 호출하면서 상태를 매개변수로 전달하는 방식은 복잡한 경우에는 계속 매개변수로 상태 데이터를 전달한다는 것이 프로그램의 효율성 측면에서 문제가 있습니다.



22-2 상태 다루기

- `ProvidableCompositionLocal`을 이용하면 상위 함수의 상태 데이터를 하위 함수에 매개변수로 일일이 전달하지 않아도 됩니다.
- `compositionLocalOf()` 혹은 `staticCompositionLocalOf()` 함수를 이용해 `ProvidableCompositionLocal`을 선언
- `compositionLocalOf()`와 `staticCompositionLocalOf()`의 차이는 상태가 변경될 때 재구성 되는 컴포저블 함수에 있습니다.
- `staticCompositionLocalOf()`는 상태가 변경되면 그 상태를 이용한 하위 함수부터 모든 하위 함수가 재구성됩니다.
- `compositionLocalOf()`는 상태를 이용한 컴포저블 함수만 재구성합니다

• `ProvidableCompositionLocal` 선언

```
val LocalProvider: ProvidableCompositionLocal<Int> = compositionLocalOf {0}  
val StaticLocalProvider: ProvidableCompositionLocal<Int> = staticCompositionLocalOf {0}
```

22-2 상태 다루기

- ProvidableCompositionLocal을 이용해 상태를 하위 컴포저블에 공개

• ProvidableCompositionLocal로 상태 공개 예

```
CompositionLocalProvider(LocalProvider provides count) {  
    LocalChild()  
}
```

- 하위 컴포저블에서 상태를 이용할 때는 ProvidableCompositionLocal 객체의 current 프로퍼티를 사용하면 됩니다.

• ProvidableCompositionLocal 객체의 프로퍼티 사용 예

```
@Composable  
fun LocalChildChild() {  
    Text("local child child : ${LocalProvider.current}")  
}
```


22-3 컴포즈로 화면 구성하기

모디파이어 사용하기

- 모디파이어 Modifier란 Modifier 객체 타입으로 표현되는 컴포저블의 설정 정보
- 컴포저블을 위한 공통 설정은 모디파이어를 사용
- 컴포즈의 내장 컴포저블은 모두 매개변수로 modifier가 선언 되어 있습니다.
- 컴포저블을 설정하기 위한 모디파이어에는 100개 이상의 함수 가 존재
- 패딩을 설정하기 위한 padding(), 컴포저블의 테두리를 설정하기 위한 border()라는 함수 등



22-3 컴포즈로 화면 구성하기

- then 함수를 이용해 다른 모디파이어를 조합할 수도 있습니다.
- 기본 설정을 위한 모디파이어 객체를 하나 준비하고 다른 모디파이어 객체에 새로운 설정을 추가하거나 기본 설정값을 변경

• then 함수로 모디파이어 추가

```
val modifier = Modifier
    .border(width = 10.dp, color = Color.Red)
    .padding(all = 30.dp)
    .background(Color.Yellow)

val secondModifier = Modifier.background(Color.Blue)
Text(
    text = "Hello World!",
    fontSize = 30.sp,
    modifier = modifier.then(secondModifier)
)
```

22-3 컴포즈로 화면 구성하기

Row와 Column 함수

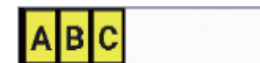
- 컴포저블 여러 개를 등록하고 배치하기 위한 레이아웃
- Row, Column, Box, ConstraintLayout
- Row는 가로 방향 배치에 관여하고 Column이 세로 방향 배치에 관여한다

• Row 컴포저블 함수를 활용해 가로 방향으로 문자열을 배치

```
@Composable
fun MainScreen() {
    Column {
        Row(modifier = Modifier.background(Color.Yellow)) {
            MyText(data = "A")
            MyText(data = "B")
            MyText(data = "C")
        }
    }
}
```

```
@Composable
fun MyText(data: String, modifier: Modifier = Modifier) {
    Text(
        data,
        fontSize = 50.sp,
        fontWeight = FontWeight.Bold,
        textAlign = TextAlign.Center,
        modifier = Modifier
            .border(width = 4.dp, color = Color.Black)
            .padding(10.dp)
            .then(modifier)
    )
}
```

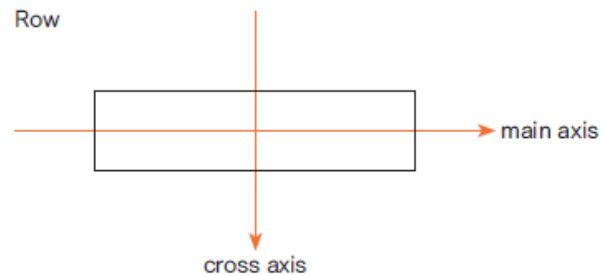
▶ 실행 결과



22-3 컴포즈로 화면 구성하기

정렬

- 배치되는 방향을 주축(main axis), 반대 방향을 반대축(cross axis)이라 부르며 Row의 주축은 가로 방향이고, 반대축은 세로 방향입니다.
- Column의 경우 주축은 세로 방향이고, 반대축은 가로 방향



22-3 컴포즈로 화면 구성하기

- 정렬은 Alignment 혹은 Arrangement 객체를 이용하는데 주축에 대한 배치는 Arrangement를, 반대축의 배치는 Alignment를 이용
- Row, Column이 차지하는 화면 영역에 추가되는 컴포저블은 기본으로 좌측 상단에 정렬

• Alignment나 Arrangement를 사용하지 않은 경우

```
Row(modifier = Modifier
    .background(Color.Yellow)
    .fillMaxWidth()
    .height(300.dp)
) {
    MyText(data = "A")
    MyText(data = "B")
    MyText(data = "C")
}
```

▶ 실행 결과



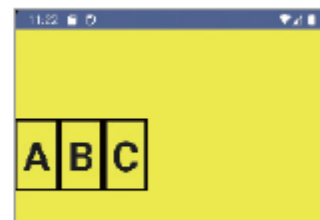
22-3 컴포즈로 화면 구성하기

- Row에는 `verticalAlignment` 속성을 이용해 수직 방향의 정렬 위치를 설정합니다.
 - `Alignment.Top`: 상단 정렬
 - `Alignment.CenterVertically`: 세로 방향 중앙 정렬
 - `Alignment.Bottom`: 하단 정렬

• Row에 `verticalAlignment` 속성을 활용

```
Row(modifier = Modifier
    .background(Color.Yellow)
    .fillMaxWidth()
    .height(300.dp),
    verticalAlignment = Alignment.CenterVertically
) {
    MyText(data = "A")
    MyText(data = "B")
    MyText(data = "C")
}
```

▶ 실행 결과



22-3 컴포즈로 화면 구성하기

- Column에서는 `horizontalAlignment` 속성을 이용해 수평 방향의 정렬 위치를 설정
 - `Alignment.Start`: 왼쪽 정렬
 - `Alignment.CenterHorizontally`: 가로 방향 중앙 정렬
 - `Alignment.End`: 오른쪽 정렬
- `Arrangement`를 이용해 Row의 가로 방향을 정렬할 때 `horizontalArrangement` 속성을 이용합니다.
- Column에서 세로 방향 정렬을 하고자 한다면 `verticalArrangement`를 이용하여 `Arrangement.Top`, `Arrangement.Bottom` 등을 지정
 - `Arrangement.Start`: 가로 방향 왼쪽 정렬
 - `Arrangement.Center`: 중앙 정렬
 - `Arrangement.End`: 가로 방향 오른쪽 정렬
 - `Arrangement.Top`: 세로 방향 윗쪽 정렬
 - `Arrangement.Bottom`: 세로 방향 아래 정렬
 - `Arrangement.SpaceEvenly`: 컴포저블 사이에 균등한 여백을 주고 정렬
 - `Arrangement.SpaceBetween`: 컴포저블 사이를 균등한 여백으로 지정. 양쪽 끝 컴포저블과 레이아웃의 간격은 없음.
 - `Arrangement.SpaceAround`: 각 컴포저블 왼쪽, 오른쪽의 여백을 동일하게 지정해 정렬

22-3 컴포즈로 화면 구성하기

- Row에 Alignment와 Arrangement 속성을 활용

```
Row(modifier = Modifier
    .background(Color.Yellow)
    .fillMaxWidth()
    .height(300.dp),
    verticalAlignment = Alignment.CenterVertically,
    horizontalArrangement = Arrangement.Center
) {
    (... 생략 ...)
}
```

▶ 실행 결과

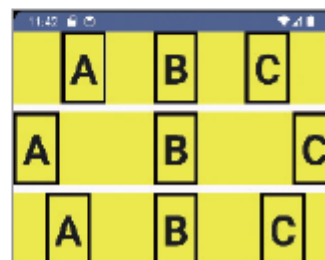


22-3 컴포즈로 화면 구성하기

• Arrangement.SpaceEvenly, Arrangement.SpaceBetween, Arrangement.SpaceAround 사용 예

```
Row(modifier = Modifier
    horizontalArrangement = Arrangement.SpaceEvenly
) {
    (... 생략 ...)
}
Row(modifier = Modifier
    horizontalArrangement = Arrangement.SpaceBetween
) {
    (... 생략 ...)
}
Row(modifier = Modifier
    horizontalArrangement = Arrangement.SpaceAround
) {
    (... 생략 ...)
}
```

▶ 실행 결과



22-3 컴포즈로 화면 구성하기

스코프 모디파이어

- Row, Column에 추가되는 여러 컴포저블 함수 가 동일 스코프
- 스코프 모디파이어는 Row, Column에 설정되는 것이 아니라 Row, Column에 추가되는 개별 컴포저블에 설정
- 자신이 포함된 스코프(Row 혹은 Column 영역) 내에 어떻게 배치되고, 어떤 크기를 가질지 를 지정하기 위해 사용되는 모디파이어 속성

```
Row { (this: RowScope)
    MyText(data = "A")
    MyText(data = "B")
    MyText(data = "C")
}
```

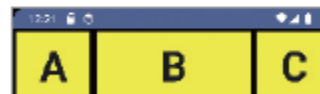
22-3 컴포즈로 화면 구성하기

- 대표적인 스코프 모디파이어로 weight 함수
- weight()는 스코프 내에서 개별 컴포저블의 사이즈를 지정하기 위해서 사용
- size()는 동일 스코프에 지정되는 다른 컴포저블을 고려하지 않고 자신의 사이즈를 결정
- weight()는 다른 컴포저블의 weight()값을 같이 이용해 사이즈를 결정

• weight 값에 따른 사이즈 지정

```
Row(modifier = Modifier.background(Color.Yellow)) {  
    MyText(data = "A", modifier = Modifier.weight(1f))  
    MyText(data = "B", modifier = Modifier.weight(2f))  
    MyText(data = "C", modifier = Modifier.weight(1f))  
}
```

▶ 실행 결과



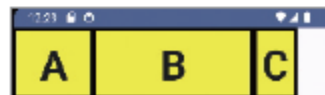
22-3 컴포즈로 화면 구성하기

- fill 매개변수값을 true/ false로 지정할 수 있는데 fill이 true면 가중치에 의해 계산된 사이즈만큼 화면에 차지하게 되고 false면 가중치에 의해 계산된 사이즈가 무시됩니다.

• fill에 의한 사이즈 적용 여부

```
Row(modifier = Modifier.background(Color.Yellow)) {  
    MyText(data = "A", modifier = Modifier.weight(weight = 1f, fill = true))  
    MyText(data = "B", modifier = Modifier.weight(weight = 2f, fill = true))  
    MyText(data = "C", modifier = Modifier.weight(weight = 1f, fill = false))  
}
```

▶ 실행 결과

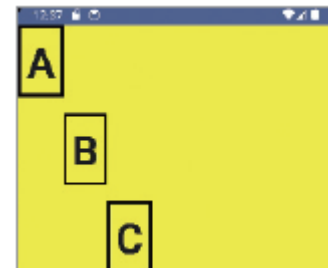


22-3 컴포즈로 화면 구성하기

- align 함수로 지정되는 모디파이어는 개별 컴포저블에 설정해 동일 스코프 내에 컴포저블이 어느 위치에 배치 될 것인지를 결정
- Alignment.CenterHorizontally, Alignment.Start, Alignment.End, Alignment.CenterVertically, Alignment.Top, Alignment.Bottom을 적용해 개별 컴포저블이 동일 스코프에서 어느 위치에 포함되어야 하는 지를 지정

• align 함수를 이용해 모디파이어 지정

```
Row(modifier = Modifier.background(Color.Yellow).height(300.dp).fillMaxWidth()) {  
    MyText(data = "A", modifier = Modifier.align(Alignment.Top))  
    MyText(data = "B", modifier = Modifier.align(Alignment.CenterVertically))  
    MyText(data = "C", modifier = Modifier.align(Alignment.Bottom))  
}
```



22-3 컴포즈로 화면 구성하기

Box

- Box에 추가되는 컴포저블을 동일한 위치에 겹쳐서 출력

• Box 함수의 레이아웃 사용

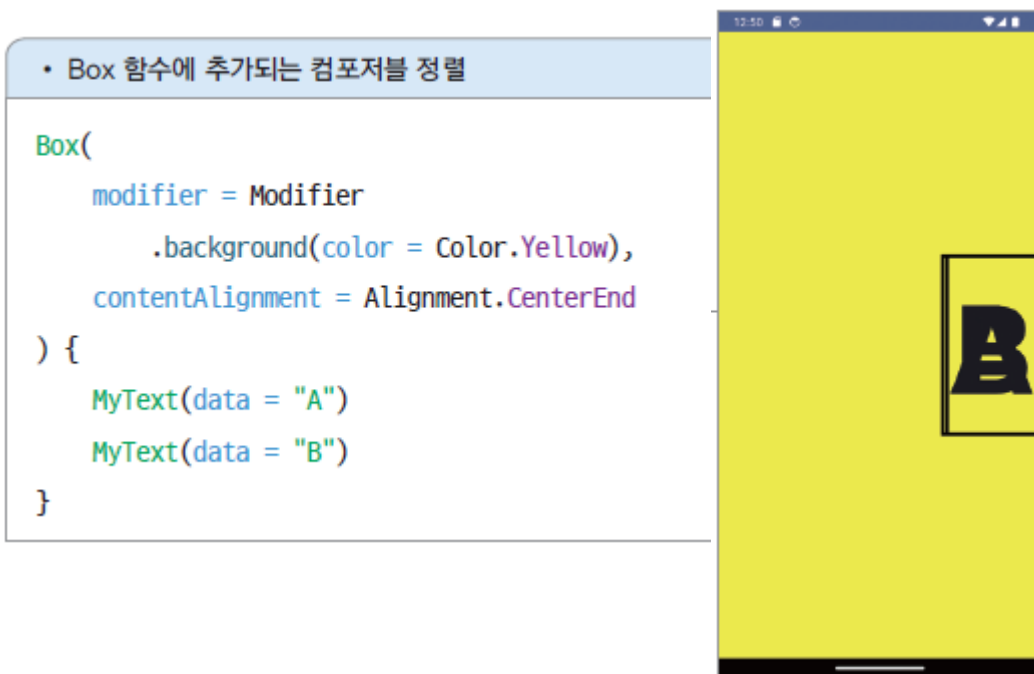
```
Box {  
    MyText(data = "A")  
    MyText(data = "B")  
}
```

▶ 실행 결과



22-3 컴포즈로 화면 구성하기

- Box 함수에 추가되는 컴포저블의 정렬은 `contentAlignment`를 이용하여 `Alignment.TopStart`, `Alignment.TopCenter`, `Alignment.TopEnd`, `Alignment.CenterStart`, `Alignment.Center`, `Alignment.CenterEnd`, `Alignment.BottomStart`, `Alignment.BottomCenter`, `Alignment.BottomEnd` 등을 사용해 정렬

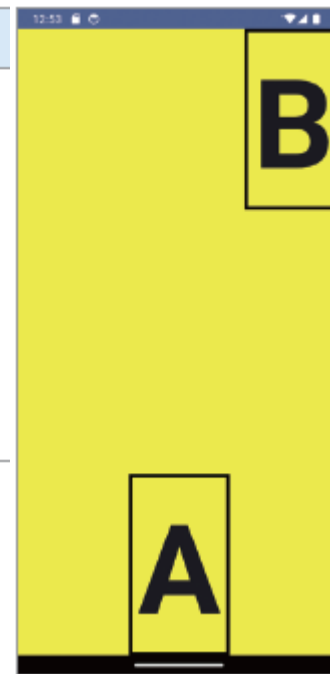


22-3 컴포즈로 화면 구성하기

- BoxScope 내에 추가되는 컴포저블에 align 모디파이어를 사용하면 개별 컴포저블의 위치를 지정할 수도 있습니다.

• align 모디파이어 사용

```
Box(  
    modifier = Modifier  
        .background(color = Color.Yellow)  
) {  
    MyText(data = "A", modifier = Modifier.align(Alignment.BottomCenter))  
    MyText(data = "B", modifier = Modifier.align(Alignment.TopEnd))  
}
```



22-3 컴포즈로 화면 구성하기

- 화면에 나타나거나 사라지게 하는 AnimatedVisibility 컴포저블이 제공

• AnimatedVisibility 함수를 활용해 한 박스만 노출

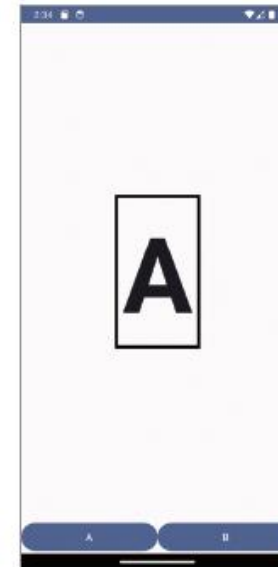
```
var aVisible by remember {
    mutableStateOf(mutableListOf(true, false))
}

Column {
    Box(modifier = Modifier.weight(1f).fillMaxWidth()) {
        this@Column.AnimatedVisibility(
            visible = aVisible.get(0), enter = fadeIn(), exit = fadeOut(),
            modifier = Modifier.align(Alignment.Center)
        ) {
            MyText(data = "A")
        }

        this@Column.AnimatedVisibility(
            visible = aVisible.get(1), enter = fadeIn(), exit = fadeOut(),
            modifier = Modifier.align(Alignment.Center)
        ) {
```

```
            MyText(data = "B")
        }
    }
    Row {
        Button(modifier = Modifier.weight(1f), onClick = {
            aVisible = mutableListOf(true, false)
        }) {
            Text("A")
        }
        Button(modifier = Modifier.weight(1f), onClick = {
            aVisible = mutableListOf(false, true)
        }) {
            Text("B")
        }
    }
}
```

▶ 실행 결과



22-3 컴포즈로 화면 구성하기

ConstraintLayout

- ConstraintLayout은 다양한 조건으로 컴포저블을 배치하기 위한 레이아웃

• ConstraintLayout 사용 선언

```
implementation("androidx.constraintlayout:constraintlayout-compose:1.1.0")
```

- ConstraintLayout에는 부모 혹은 다른 컴포저블을 기준으로 컴포저블의 배치 조건을 명시
- 다른 컴포저블을 지정하기 위해서는 컴포저블을 식별할 수 있는 식별자가 있어야 하는데 이것을 참조라고 합니다.
- 참조 선언은 이와 같이 `createRef()`를 이용
- 여러 개의 참조를 선언하고 싶으면 `createRefs()`를 이용

• 참조 선언

```
val text1 = createRef()
val (button1, button2) = createRefs()
```

22-3 컴포즈로 화면 구성하기

- 선언된 참조를 특정 컴포저블에 지정하려면 다음과 같이 `constrainAs` 모디파이어를 이용합 니다.
- 위치에 관한 조 건은 나의 위치. `linkTo`(대상의 위치) 형태로 작성합니다.

• `constrainAs`로 선언된 참조를 특정 컴포저블에 지정

```
ConstraintLayout(  
    Modifier  
        .size(150.dp, 150.dp)  
        .background(Color.Yellow)) {  
    //-----  
    MyText(data="Hello", modifier = Modifier.constrainAs(text1){  
        top.linkTo(parent.top, margin = 10.dp)  
        bottom.linkTo(parent.bottom, margin = 10.dp)  
    })  
}
```

▶ 실행 결과



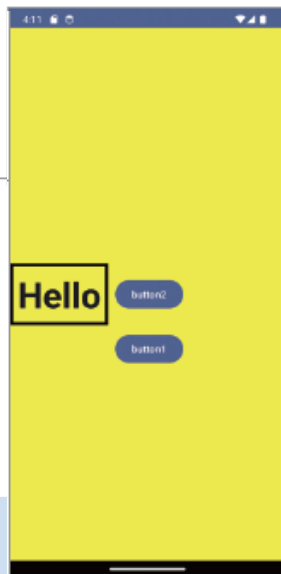
22-3 컴포즈로 화면 구성하기

- 다른 컴포저블을 기준으로 컴포저블의 위치를 지정하고 싶다면 기준이 되는 컴포저블에 미리 선언한 참조를 이용

• 선언된 참조를 사용해 다른 컴포저블 기준으로 위치 지정

```
Button(onClick = { /*TODO*/ }, modifier = Modifier.constrainAs(button1){
    top.linkTo(text1.bottom, margin=10.dp)
    start.linkTo(text1.end, margin = 10.dp)
}) {
    Text(text = "button1")
}
Button(onClick = { /*TODO*/ }, modifier = Modifier.constrainAs(button2){
    centerVerticallyTo(parent)
    centerHorizontallyTo(button1)
}) {
    Text(text = "button2")
}
```

▶ 실행 결과



22-3 컴포즈로 화면 구성하기

리스트

- 목록 화면을 구성하기 위해 LazyColumn, LazyRow, LazyVerticalGrid와 같은 함수를 제공합니다.
- 'Lazy'가 붙는 이유는 '늦은 로딩'을 지원하는 컴포저블이기 때문입니다.
- LazyColumn을 이용하면 다음과 같이 항목을 세로 방향으로 나열
- items() 를 이용해 각 항목이 어떻게 구성되는지를 명시

• LazyColumn으로 세로 방향으로 항목 나열

```
LazyColumn {  
    val datas = listOf<String>("one", "two", "three", "four")  
    items(datas) { item ->  
        Text("$item ")  
    }  
}
```

▶ 실행 결과



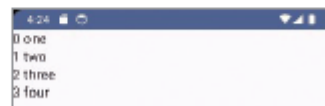
22-3 컴포즈로 화면 구성하기

- items() 대신 itemsIndexed()를 사용할 수 있는데 이렇게 되면 각 항목별로 index까지 출력

• itemsIndexed 함수로 항목별 인덱스 출력

```
LazyColumn {  
    val datas = listOf<String>("one", "two", "three", "four")  
    itemsIndexed(datas){index, item ->  
        Text("$index $item")  
    }  
}
```

▶ 실행 결과



22-3 컴포즈로 화면 구성하기

내비게이션

- 컴포저블을 교체하면서 화면 전환
- 화면 전환을 하기 위해서는 navigation 라이브러리를 이용

• navigation 라이브러리 추가

```
implementation("androidx.navigation:navigation-compose:2.8.4")
```

22-3 컴포즈로 화면 구성하기

- navigation 라이브러리의 핵심 클래스는 NavHost와 NavHostController입니다.
- NavHost는 화면 단위의 컴포저블을 등록하고 화면을 스택 구조로 관리하는 역할
- NavHostController 는 실제 필요한 경우 화면 전환을 하기 위한 함수를 제공

• navigation 라이브러리 추가

```
@Composable
fun MainScreen() {
    //NavController를 먼저 선언하고 NavController로 화면을 등록하는 NavHost를 만든다.
    val navController = rememberNavController()
    NavHost(navController = navController, startDestination = "home" ){
        composable("home"){ HomeScreen(navController) }
        composable("one"){ OneScreen(navController) }
        composable("two"){ TwoScreen(navController) }
    }
}
```


22-3 컴포즈로 화면 구성하기

- NavHost에 등록된 컴포저블에서 화면 전환 명령을 내릴 때는 NavHost에 등록된 NavController의 navigate 함수를 호출하면서 매개변수로 이동하고자 하는 화면의 이름을 명시

• 화면 전환을 위한 navigate 함수 호출

```
Button(onClick = { navController.navigate("one") }) {  
    Text("Go One")  
}
```

22-3 컴포즈로 화면 구성하기

이전 화면으로 이동

- 안드로이드의 '뒤로 가기' 버튼에 의한 이동은 별도로 처리하지 않아도 이전 화면 컴포저블로 잘 이동됩니다.
- 프로그램적으로 이전 화면으로 전 환하고 싶다면 NavController의 popBackStack 함수를 호출합니다.

• 이전 화면으로의 전환을 위한 popBackStack 함수 호출

```
Button(onClick = { navController.popBackStack() }) {  
    Text("Go Back")  
}
```

22-3 컴포즈로 화면 구성하기

- 이전 컴포저블 화면이 아닌 특정 컴포저블 화면으로 이동하고 싶다면 `navigate()` 함수를 호출하고 `popUpTo()` 함수로 옵션을 지정하면 됩니다.

• 특정 컴포저블 화면으로 이동하기 위한 `popUpTo` 함수 호출

```
Button(onClick = {  
    navController.navigate("onesub"){  
        popUpTo("home")  
    }  
}) {  
    Text("Go OneSub")  
}
```

22-3 컴포즈로 화면 구성하기

데이터 전달

- 이름에 전달해야 하는 데이터를 같이 명시해서 등록할 수 있습니다.

• composable 함수에 데이터 추가

```
composable("one/{userId}/{no}", arguments = listOf(navArgument("userId"){  
    type = NavType.StringType  
}, navArgument("no"){  
    type = NavType.IntType  
})){ navBackStackEntry ->  
    OneScreen(  
        navController,  
        navBackStackEntry.arguments?.getString("userId"),  
        navBackStackEntry.arguments?.getInt("no")  
    )  
}
```

• navigate 함수를 호출해 전달할 데이터 명시

```
Button(onClick = { navController.navigate("one/kkang/20") }) {  
    Text("Go One")  
}
```

22-3 컴포즈로 화면 구성하기

- 이전 화면으로 되돌아갈 때 현재 화면에서의 결과 데이터를 전달하겠다면 popBack Stack 함수 실행 전에 다음과 같이 set 함수를 이용해 결과 데이터를 먼저 담으면 됩니다..

• set 함수로 결과 데이터를 담아 전달

```
Button(onClick = {  
    //이전 화면으로 되돌아가면서 데이터 전달  
    navController.previousBackStackEntry?.savedStateHandle?.set("msg", "kim")  
    navController.popBackStack()  
}) {  
    Text("Go Back")  
}
```

22-3 컴포즈로 화면 구성하기

- 받는 곳에서도 동일한 이름으로 획득하면 됩니다.

• 결과 데이터 획득

```
val msg =  
    navController.currentBackStackEntry?.savedStateHandle?.get<String>("msg")
```

컴포즈로 뉴스 앱 만들기

1단계. 모듈 생성하기

- Ch22_Compose라는 이름의 새로운 모듈을 만듭니다.

2단계. build.gradle.kts 작성하기

- Retrofit 라이브러리와 이미지 출력을 위한 coil 라이브러리를 추가

3단계. 실습 파일 복사하기

- Model, retrofit 폴더와 MyApplication.kt 파일 복사



컴포즈로 뉴스 앱 만들기

4단계. AndroidManifest.xml 작성하기

- 퍼미션과 Application 객체 등록

5단계. Item.kt 작성하기

- 목록의 항목을 구성하기 위한 컴포저블 작성

6단계. MainScreen 작성하기

- 서버와 네트워크 통신을 하고 서버 데이터로 목록을 구성하는 MainScreenn.kt 작성

Do it! 실습

컴포즈로 뉴스 앱 만들기

7단계. MainActivity.kt 작성하기

- mainScreen 을 등록

8단계. 앱 실행하기





감사합니다

단단히 마음먹고 떠난 사람은
산꼭대기에 도착할 수 있다.
산은 올라가는 사람에게만 정복된다.

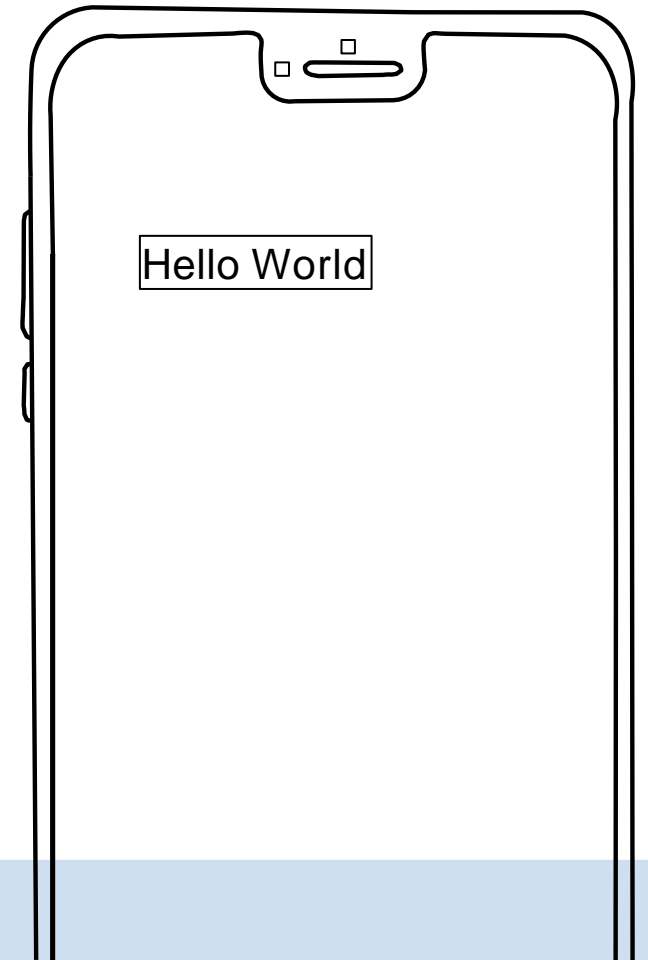


윌리엄 셰익스피어
William Shakespeare

Modifiers

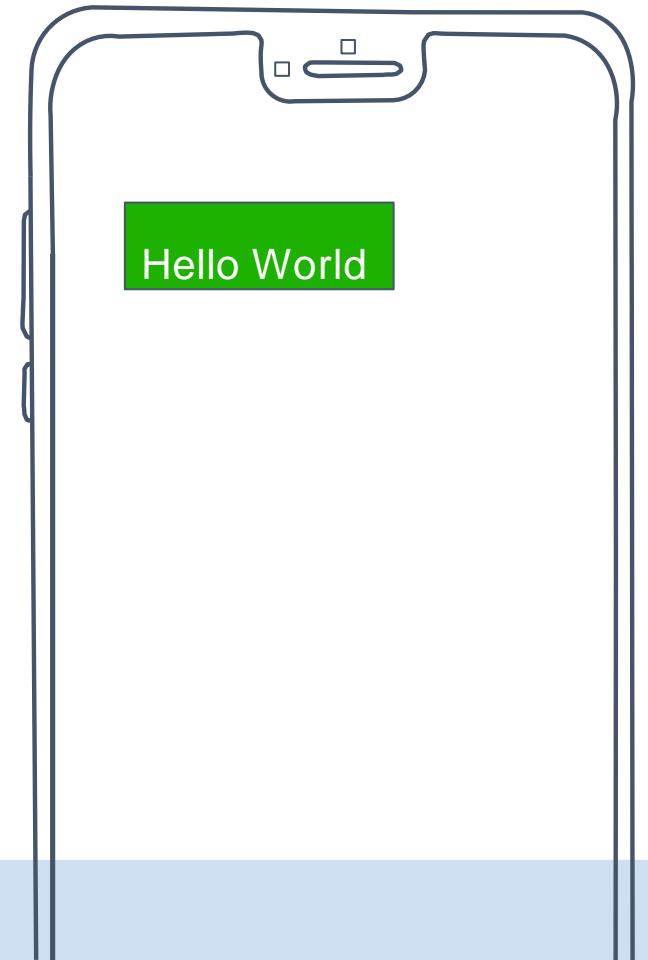
Modifiers

```
Text(  
    modifier = Modifier  
        .padding(10.dp),  
    text = "Hello World"  
)
```



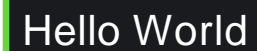
Modifiers

```
Text(  
    modifier = Modifier  
        .background(Color.Green),  
    text = "Hello World"  
)
```



Modifiers

```
Text(  
    modifier = Modifier  
        .border(width = 2.dp, Color.Green),  
    text = "Hello World"  
)
```



Hello World

Modifiers

Modifiers

```
Text(  
    modifier = Modifier  
        .border(width = 2.dp, Color.Red)  
        .background(Color.Green)  
        .padding(12.dp) ,  
    text = "Hello World"  
)
```



Hello World

Modifiers

```
Text(  
    modifier = Modifier  
        .padding(12.dp)  
        .border(width = 2.dp, Color.Red)  
        .background(Color.Green) ,  
    text = "Hello World"  
)
```

Hello World

List of Compose modifiers

Actions

Scope: Any

```
Modifier.clickable(  
    enabled: Boolean,  
    onClickLabel: String?,  
    role: Role?,  
    onClick: () -> Unit  
)
```

Configure component to receive clicks via input or accessibility "click" event.

Scope: Any

```
Modifier.clickable(  
    interactionSource: MutableInteractionSource,  
    indication: Indication?,  
    enabled: Boolean,  
    onClickLabel: String?,  
    role: Role?
```

On this page

[Actions](#)

[Alignment](#)

[Animation](#)

[Border](#)

[Drawing](#)

[Focus](#)

[Graphics](#)

[Keyboard](#)

[Layout](#)

[Padding](#)

[Pointer](#)

[Position](#)

[Semantics](#)

[Scroll](#)

[Size](#)

[Testing](#)

✦ Recommendations

[Get started with Jetpack Compose](#)

Updated Mar 9, 2022



Column

➡ *Column* {

}

Column

```
Column {  
    Text("Hello World")  
    ➡ Text("Hello World")  
    Text("Hello World")  
}
```

Column



```
Column (  
    verticalArrangement =  
) {  
    Text("Hello World")  
    Text("Hello World")  
    Text("Hello World")  
}
```

Column

```
Column (  
    → verticalArrangement = Arrangement.Center  
) {  
    Text("Hello World")  
    Text("Hello World")  
    Text("Hello World")  
}
```


Column

```
Column (
```



```
Arrangement.SpaceBetween
```

```
) {
```

```
Text("Hello World")
```

```
Text("Hello World")
```

```
Text("Hello World")
```

```
}
```

Column



```
Column(  
    horizontalAlignment =  
) {  
    Text("Hello World")  
    Text("Hello World")  
    Text("Hello World")  
}
```

Column

```
Column (
```

➡ `horizontalAlignment = Alignment.End`

```
) {
```

```
    Text("Hello World")
```

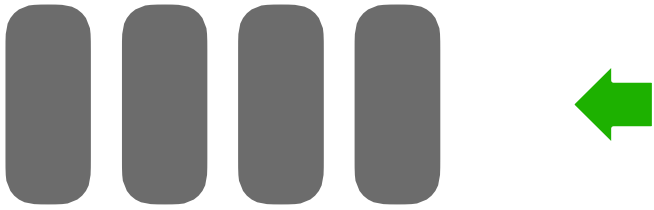
```
    Text("Hello World")
```

```
    Text("Hello World")
```

```
}
```

Column

```
Column (  
    Alignment.CenterHorizontally  
) {  
    Text("Hello World")  
    Text("Hello World")  
    Text("Hello World")  
}
```



Row

```
Row {  
    Text("Hello")  
    Text("World")  
    Text("!")  
}
```

Row w



```
Row(  
    horizontalArrangement =  
) {  
    Text("Hello World")  
    Text("Hello World")  
    Text("Hello World")  
}
```

Row

```
Row(  
    Arrangement.SpaceBetween  
) {  
    Text("Hello")  
    Text("World")  
    Text("!")  
}
```


Row

W



```
Row(  
    verticalAlignment =  
) {  
    Text("Hello World")  
    Text("Hello World")  
    Text("Hello World")  
}
```

Row w



```
Row(  
    Alignment.CenterVertically  
) {  
    Text("Hello World")  
    Text("Hello World")  
    Text("Hello World")  
}
```

Column & Row

Column

Row

Vertical Arrangement

Horizontal Arrangement

Horizontal Alignment

Vertical Alignment



Column & Row

Row {

Image (...)



}



Column & Row

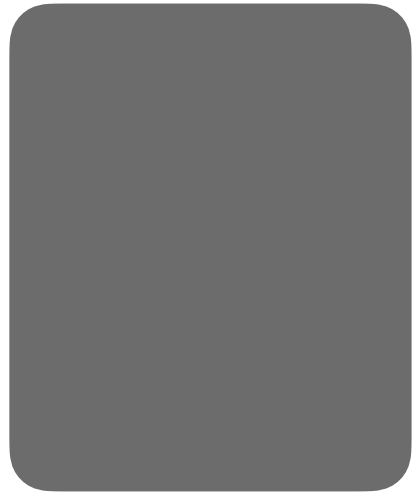
```
Row {  
  Image (...)  
  Column {  
      
  }  
}
```



Column & Row

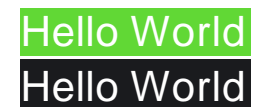
```
Row {  
  Image (...)  
  Column {  
    Text ("...")  
    Text ("...")  
  }  
}
```






Layout Modifiers

```
Column(  
    Modifier.background(Color.Green)  
) {  
    Text(text = "Hello World")  
    Text(text = "Hello World")  
}
```




Hello World
Hello World

Layout Modifiers



```
Column(  
    Modifier  
        .background(Color.Green)  
        .fillMaxWidth()  
) {  
    Text(text = "Hello World")  
    Text(text = "Hello World")  
}
```



Hello World
Hello World

Layout Modifiers

```
Column(  
    Modifier  
        .background(Color.Green)  
        .fillMaxSize()  
) {  
    Text(text = "Hello World")  
    Text(text = "Hello World")  
}
```



Hello World
Hello World

Layout Modifiers

```
Column(  
    Modifier  
        .background(Color.Green)  
        .width(200.dp)  
) {  
    Text(text = "Hello World")  
    Text(text = "Hello World")  
}
```



Hello World
Hello World



Box {

Image (...)

}



Box

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



Box

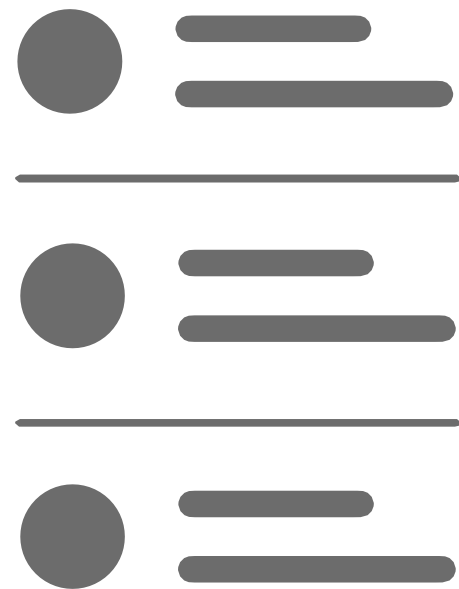
```
Box {  
    Image (...)  
  
    Column (  
        Modifier.align( Alignment.BottomEnd  
        )  
    ) {  
        Text (...)  
        Text (...)  
    }  
}
```





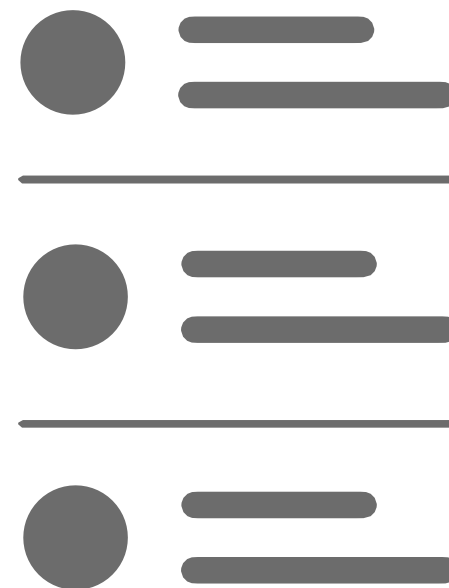
List

```
@Composable
fun EmployeeListView(items: List<Item>) {
    LazyColumn {
        
    }
}
```



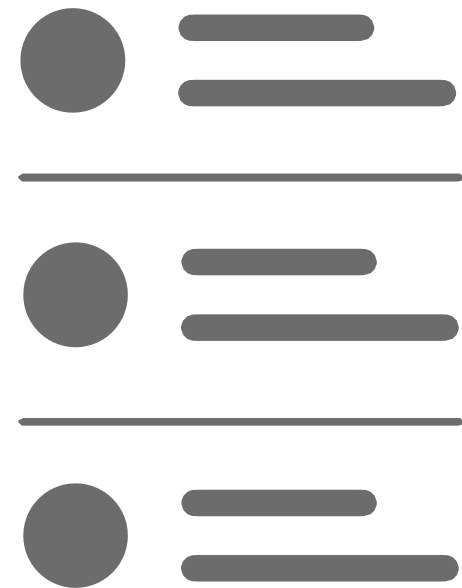
List

```
@Composable
fun EmployeeListView(items: List<Item>) {
    LazyColumn {
        items(items) { item →
            [REDACTED]
        }
    }
}
```



List

```
@Composable
fun EmployeeListView(items: List<Item>) {
    LazyColumn {
        items(items) { item →
            ItemRow(item)
        }
    }
}
```





Scaffold

```
Scaffold(  
  topBar = { ←  
  
  }  
) {  
  
}
```



Scaffold

```
Scaffold(  
  topBar = {  
    Text(text = "Home")  
  }  
) {  
  
}
```



Scaffold

```
Scaffold(  
  topBar = {  
    Text(text = "Home")  
  }  
) {  
  Text(text = "Hello World")  
}
```



Layouts

Managing State

Managing State



Managing State

```
var counter by remember {  
    mutableStateOf(0)  
}
```



Managing State

```
Column {  
    Text(  
        text = counter.toString()  
    )  
    ...  
}
```

Managing State

```
Button(  
    onClick = { counter += 1 }  
) {  
    Text(text = "+")  
}
```



Managing State

```
Button(  
    onClick = { counter -- }  
) {  
    Text(text = "-")  
}
```



Recomposition

Managing State

```
var counter by remember {  
    mutableStateOf(0)  
}
```



Managing State

Managing State

Search

×

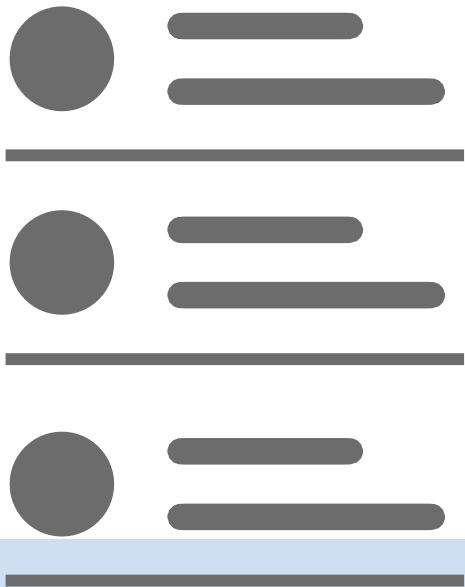
Managing State

View Model

Search View

Search

x



Managing State

```
@Composable  
fun SearchView() {
```

Search



Managing State

```
@Composable
fun SearchView() {

    var query by remember {
        mutableStateOf("")
    }

}
```



Managing State

```
@Composable
fun SearchView() {

    OutlinedTextField(
        value = query
    )

}
```



Managing State

```
@Composable
fun SearchView() {

    OutlinedTextField(
        nValueChange = {
            query = it
        }
    )

}
```



Managing State

```
@Composable
fun SearchView() {

    OutlinedTextField(
        keyboardOptions = KeyboardOptions(
            imeAction = ImeAction.Search
        )
    )

}
```



Managing State

```
@Composable
fun SearchView() {

    OutlinedTextField(
        keyboardOptions = KeyboardActions(
            onSearch = {

            }
        )
    )
}
```



Managing State

```
@Composable
fun SearchView(onSearch: (String) ->Unit) {
```



Managing State

```
@Composable
fun SearchView(onSearch: (String) ->Unit) {
```

```
    OutlinedTextField(
        keyboardOptions = KeyboardActions(
            onSearch = {
                onSearch(query)
            }
        )
    )
}
```



Managing State

View Model

Search



Search View

Managing State



```
SearchView { query →  
    viewModel.onSearch(query)  
}
```



Managing State

Flows

View

View Model

Flows

Flows

```
class MyViewModel: ViewModel() {  
  
    val stateFlow = _stateFlow.asStateFlow()  
  
}
```

Flows

```
@Composable  
  
fun HomeView() {  
    viewModel.stateFlow.collectAsState()  
}
```

Managing State

Managing State

What is a side effect?


Side Effects

Launched Effect

Launched Effect

```
@Composable
fun HomeView() {


    var counter by remember { mutableStateOf(0) }

}
```

Launched Effect

- `@Composable`
 - `fun HomeView() {`
 - `var counter by remember { mutableStateOf(0) }`
 - `LaunchedEffect` {
 - `while (true) {`


 - `}`
- `}`

Launched Effect

- `@Composable`
- `fun HomeView() {`
 - `var counter by remember { mutableStateOf(0) }`
 - `LaunchedEffect`
 - `{`
 - `while (true)`



Launched Effect

```
@Composable
fun HomeView() {

    var counter by remember { mutableStateOf(0) }

    LaunchedEffect(key1 = Unit) {
        while (true) {
            delay(2000)
            counter ++
        }
    }
}
```

Disposable Effect

- Triggers on first composition or key change
- Calls `onDispose` on terminate

Disposable Effect

```
@Composable
fun HomeView() {

    DisposableEffect(    ...) {
        onDispose {
            callback.remove()
        }
    }

}
```


Side Effects

- Launched Effect
- Disposable Effect