Lecture 6

COMP 3717- Mobile Dev with Android Tech

第6讲

COMP 3717 - 使用Android技术进行移动开发

What are Android activities?

- Activities are one of the essential building blocks of an Android app
- Each screen of an android app is represented by an Activity
- Typically, each app has a *main activity* which is usually the first screen of the app
  - The main entry point for user interaction

什么是 Android 活动?

- 活动是 Android 应用的基本构成组件之一
- Android 应用中的每个屏幕都由一个活动表示
- 通常，每个应用都有一个 主活动，它通常是应用的第一个屏幕
  - 用户交互的主要入口点

Why use Activities?

- Activities allow us to start other activities in the app
  - If we have multiple activities in our app, we can navigate to those
- Activities serve as entry points from other apps, for example
  - App A could launch app B using any of app B's activities as entry points
- In the past, it was common for apps to contain multiple Activities
  - Single-activity architecture is now the recommended approach

为什么要使用活动?

- 活动允许我们在应用中启动其他活动
  - 如果我们的应用中有多个活动，就可以在这些活动中进行导航
- 活动可作为来自其他应用的入口点，例如
  - 应用A可以使用应用B的任意一个活动作为入口点来启动应用B
- 在过去，应用程序包含多个活动是很常见的
  - 现在推荐采用单活动架构

What is the Activity Lifecycle?

- Each activity in your app goes through multiple states depending on what happens to that activity
- For instance, when you start, leave or re-enter an activity you have control over how certain functionality is handled

什么是 Activity 生命周期?

- 应用中的每个 Activity 都会根据该 Activity 发生的情况经历多个状态
- 例如，当您启动、离开或重新进入某个 Activity 时，您可以控制如何处理特定功能

What is the activity lifecycle?

- Let's say we are playing a zombie apocalypse game/app and are right in the middle of fighting a bunch of zombies… you are having trouble and switch to your internet browser app in search for a guide

- How the game properly handles pausing and/or saving progress here would be crucial for us

什么是活动生命周期?

- 假设我们正在玩一款僵尸末日游戏/应用程序，正处在与一群僵尸激烈战斗的中途……你遇到了困难，于是切换到浏览器应用程序中寻找攻略

- 游戏在此处如何正确处理暂停和/或保存进度对我们来说至关重要

What is activity lifecycle? (cont.)

- There are six main states an Activity goes through from creation to being destroyed
  - Created
  - Started
  - Resumed
  - Paused
  - Stopped
  - Destroyed

什么是活动生命周期? （续）

- 一个活动从创建到销毁会经历六个主要状态
  - 已创建
  - 已启动
  - 已恢复
  - 已暂停
  - 已停止
  - 已销毁

What is activity lifecycle? (cont.)

- These multiple states your Activity goes through are handled with specific callback functions

- Using these callbacks appropriately will go a long way in how efficient your app will be and will avoid…
  - App crashes
  - Wasted time and resources
  - Losing progress

什么是活动生命周期？（续）

- 您的 Activity 所经历的多个状态通过特定的回调函数进行处理

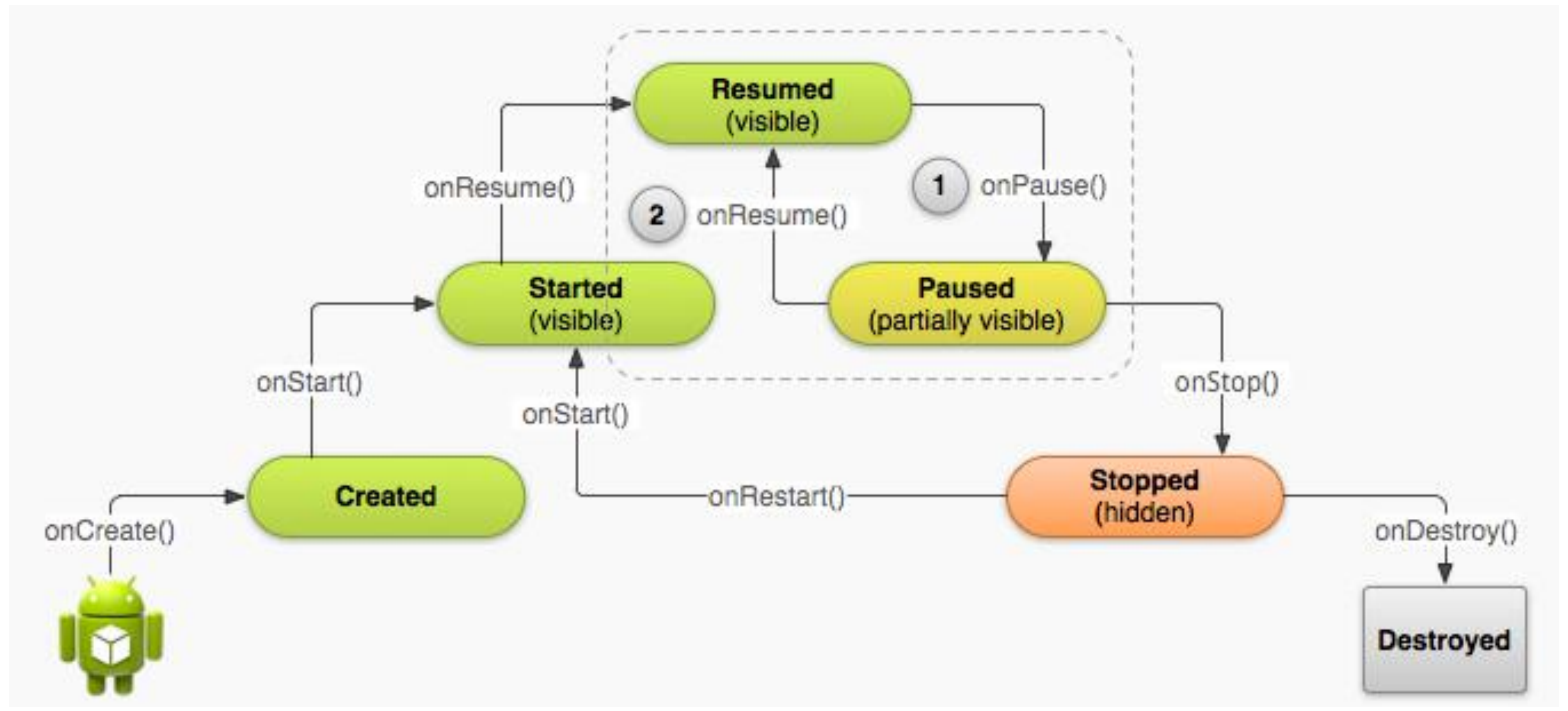
- 合理使用这些回调函数将极大提升您应用程序的效率，并避免……
  - 应用崩溃
  - 浪费时间和资源
  - 丢失进度

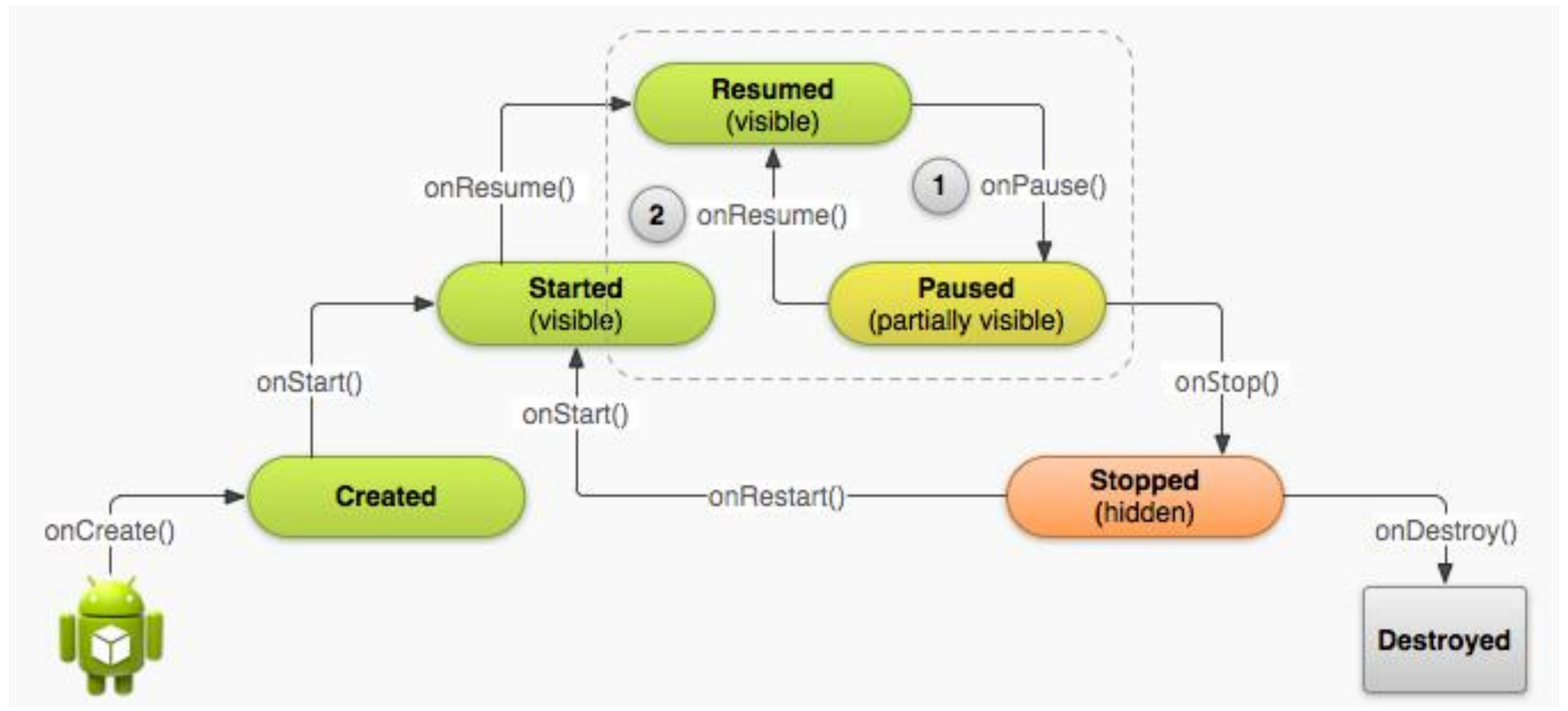
What is activity lifecycle? (cont.)

- Here are the six corresponding callbacks that are invoked depending on which state your activity transitions into
  - onCreate()
  - onStart()
  - onResume()
  - onPause()
  - onStop()
  - onDestroy()

什么是活动生命周期？（续）

- 以下是根据
  活动所转换到的状态而调用的六个相应回调
  - onCreate()
  - onStart()
  - onResume()
  - onPause()
  - onStop()
  - onDestroy()


Resumed (visible)
onResume()
2 onResume()
1 onPause()
Started (visible)
Paused (partially visible)
onStart()
onStart()
onStop()
Created
onRestart()
Stopped (hidden)
onCreate()
onDestroy()
Destroyed


Resumed (visible)
onResume()
2 onResume()
1 onPause()
Started (visible)
Paused (partially visible)
onStart()
onStart()
onStop()
Created
onRestart()
Stopped (hidden)
onCreate()
onDestroy()
Destroyed

Created - onCreate()

- Performs basic application start-up logic that should happen only once for the entire lifecycle of the activity

- Common logic that should go in *onCreate()*
  - Handling the activities previously saved state through a bundle

    super.onCreate(savedInstanceState);

  - Setting the layout for the Activity

    setContent{}

已创建 - onCreate()

- 执行基本的应用程序启动逻辑，该逻辑在整个 Activity 生命周期中仅应发生一次

- 应放在 *onCreate()* 中的常见逻辑
  - 通过 Bundle 处理 Activity 之前保存的状态

    super.onCreate(savedInstanceState);

  - 为 Activity 设置布局

    setContent{}

Started - onStart()

- The activity enters the *Started* state and invokes *onStart()* when entering the foreground
- The *Started* state's main purpose is to make the activity visible to the user and prepare the activity to enter the foreground
- Difference between *onStart()* and *onCreate()*
  - onCreate is only called once during an activities lifecycle whereas if the activity goes into the background, then comes back to the foreground onStart() will be invoked again

已启动 - onStart()

- 当活动进入前台时，会进入 已启动 状态并调用 *onStart()*
- 已启动 状态的主要目的是使活动对用户可见，并准备让活动进入前台
- *onStart()* 与 *onCreate()* 的区别
  - onCreate 在活动的生命周期中仅被调用一次，而如果活动进入后台后再回到前台，onStart() 将会被再次调用

Resumed - onResume()

- The *Resumed* state is when the activity completely enters the foreground in which it invokes *onResume()*
- This state stays active until something takes focus away from this activity, such as
  - Minimizing the app
  - Turning screen off on device
  - Navigating to another activity

已恢复 - onResume()

- 已恢复状态表示活动完全进入前台，此时会调用*onResume()*
- 此状态会持续保持活跃，直到有其他内容使该活动失去焦点，例如
  - 最小化应用
  - 关闭设备屏幕
  - 导航到另一个活动

Paused - onPause()

- The *Paused* state is a brief state that usually occurs when an event interrupts the *Resumed* state
- For example, the *Paused* state occurs, if the app begins to be minimized, or when another activity begins to take focus
- Releasing components in *onPause()* that were initialized in *onResume()* is good practice

已暂停 - onPause()

- 已暂停状态是一个短暂的状态，通常在某个事件中断了恢复状态时发生
- 例如，当应用开始被最小化时，就会进入已暂停状态或者当另一个活动开始获得焦点时
- 在*onPause()* 中释放于*onResume()* 中初始化的组件是一种良好的做法

Stopped - onStop()

- When the activity is completely out of foreground then onStop() is invoked and the *Stopped* state occurs
- Commonly happens when an app is completely minimized, or a new activity completely covers the entire screen
- During the *Stopped* state, the Activity still stays in memory and maintains all information

已停止 - onStop()

- 当 Activity 完全退出前台时，将调用 onStop()，此时进入 已停止 状态
- 通常发生在应用被完全最小化，或新的 Activity 完全覆盖整个屏幕时
- 在 已停止 状态期间，Activity 仍保留在内存中，并保持所有信息

Destroyed - onDestroy()

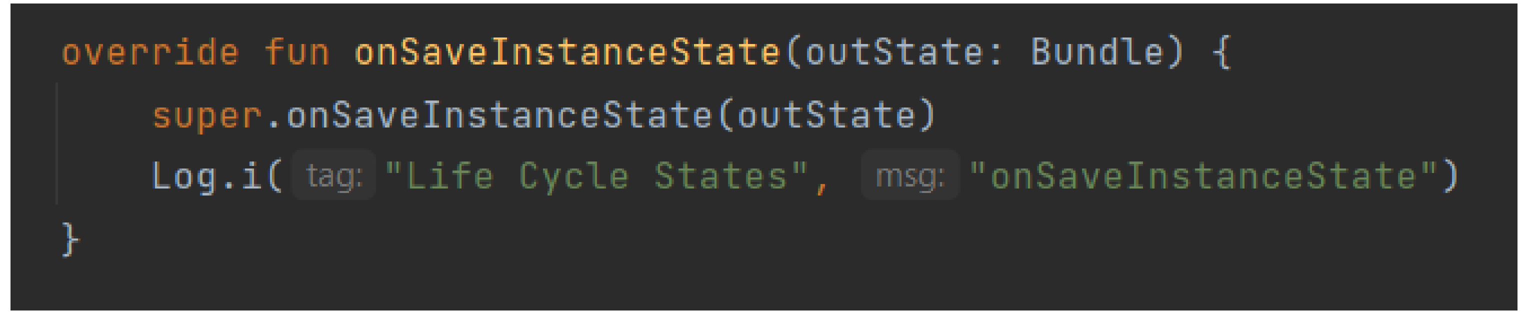
- The final lifecycle state which destroys the Activity instance from the activity stack
- Called when the Activity is completely dismissed, or a configuration change happens such as changing screen orientation
- The *UI state* is preserved through *onSaveInstanceState* if the *Activity* is destroyed and recreated by the system
    - Ex. Changing screen orientation

已销毁 - onDestroy()

- 最终的生命周期状态，从活动栈中销毁 Activity 实例
- 当 Activity 被完全关闭，或发生配置更改时调用，例如屏幕方向改变
- UI 状态 会通过 *onSaveInstanceState* 保存，如果 *Activity* 被系统销毁并重新创建
    - 例如：改变屏幕方向

onSaveInstanceState

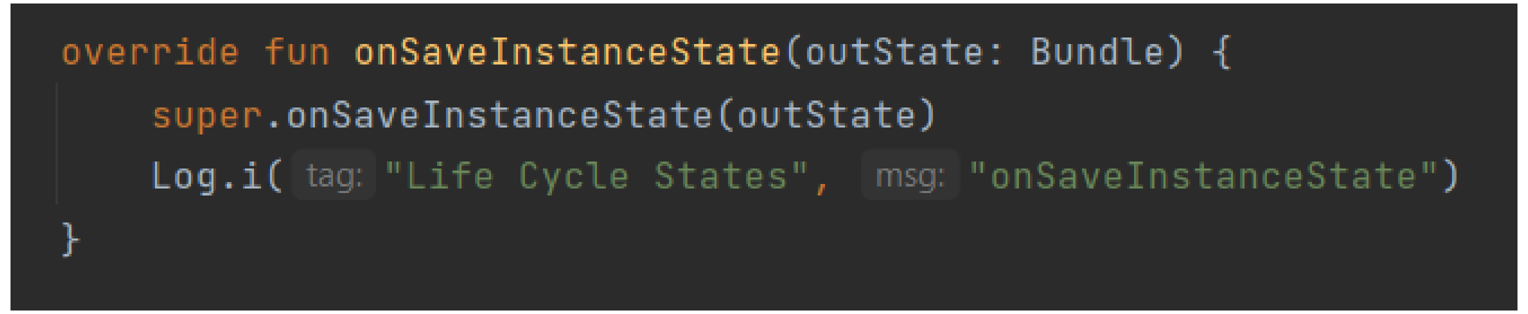
- Saves the state of the UI typically through a configuration change
- You can also override the default behaviour and restore small amounts of data when the activity state is re-created



```
override fun onSaveInstanceState(outState: Bundle) {
    super.onSaveInstanceState(outState)
    Log.i( tag: "Life Cycle States",  msg: "onSaveInstanceState")
}
```

onSaveInstanceState

- 通常通过配置更改来保存 UI 的状态
- 您还可以重写默认行为，在活动状态重新创建时恢复少量数据



```
override fun onSaveInstanceState(outState: Bundle) {
    super.onSaveInstanceState(outState)
    Log.i( tag: "Life Cycle States",  msg: "onSaveInstanceState")
}
```

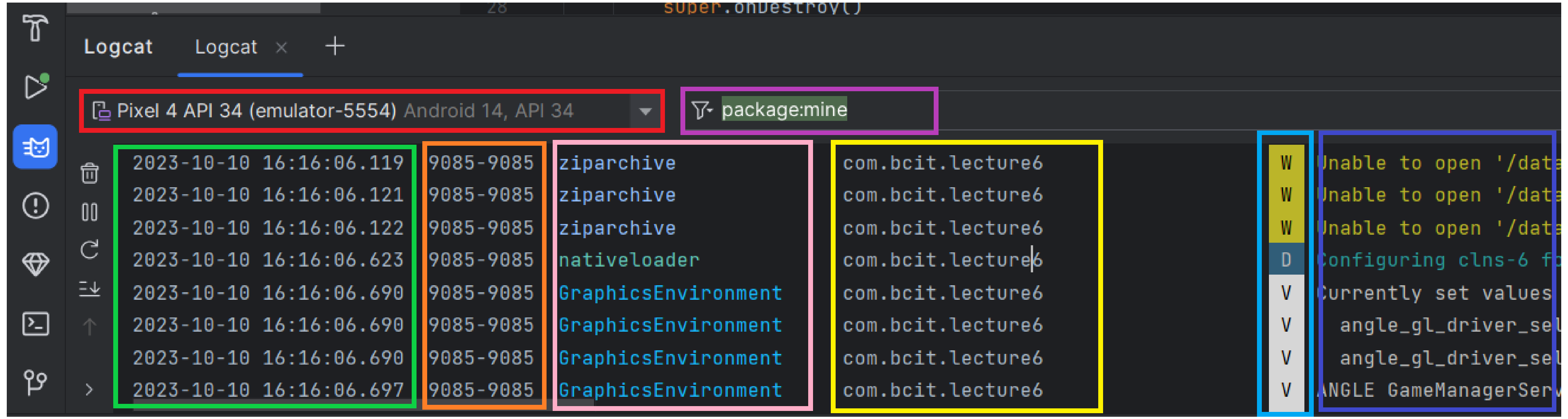
Logging in Android

- To see logs in android we use the logcat which displays lots of information
  - The device we are seeing logs for
  - The filter for our logs
  - The date and timestamp
  - The Process and Thread ID
  - The tag
  - The package
  - The priority level
  - The message

Android 中的日志

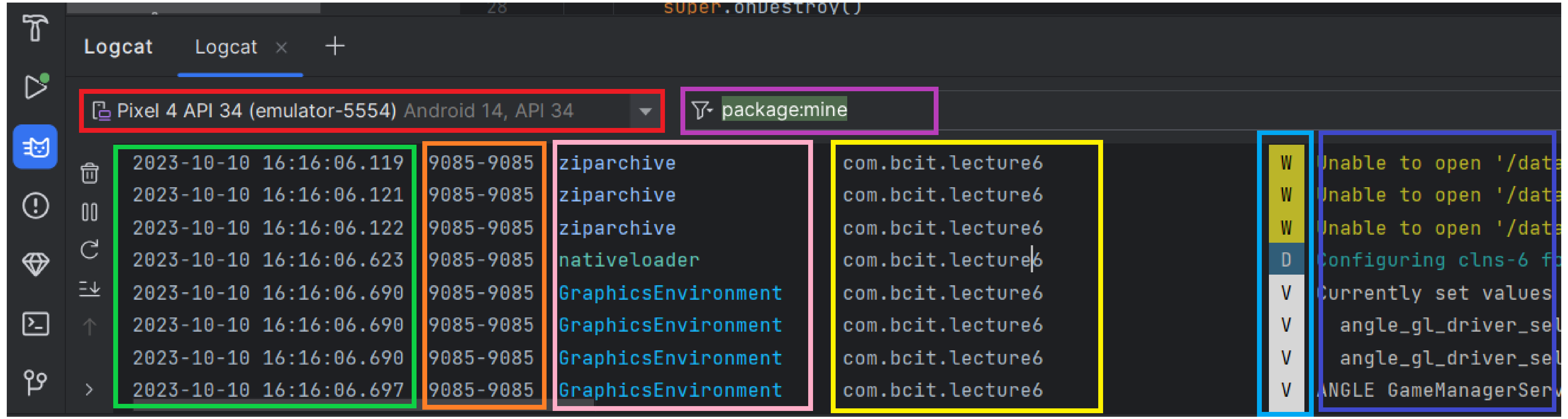
- 要在 Android 中查看日志，我们使用 logcat，它会显示大量信息
  - 正在查看日志的设备
  - 日志的筛选器条件
  - 日志的日期和时间戳
  - 进程和线程ID
  - 该标签
  - 该包
  - 该优先级
  - 该消息

Logging in Android (cont.)



```
2023-10-10 16:16:06.119 9085-9085 ziparchive          com.bcit.lecture6  W Unable to open '/dat
2023-10-10 16:16:06.121 9085-9085 ziparchive          com.bcit.lecture6  W Unable to open '/dat
2023-10-10 16:16:06.122 9085-9085 ziparchive          com.bcit.lecture6  W Unable to open '/dat
2023-10-10 16:16:06.623 9085-9085 nativeloader        com.bcit.lecture6  D Configuring clns-6 f
2023-10-10 16:16:06.690 9085-9085 GraphicsEnvironment com.bcit.lecture6  V Currently set values
2023-10-10 16:16:06.690 9085-9085 GraphicsEnvironment com.bcit.lecture6  V   angle_gl_driver_se
2023-10-10 16:16:06.690 9085-9085 GraphicsEnvironment com.bcit.lecture6  V   angle_gl_driver_se
2023-10-10 16:16:06.697 9085-9085 GraphicsEnvironment com.bcit.lecture6  V ANGLE GameManagerSer
```

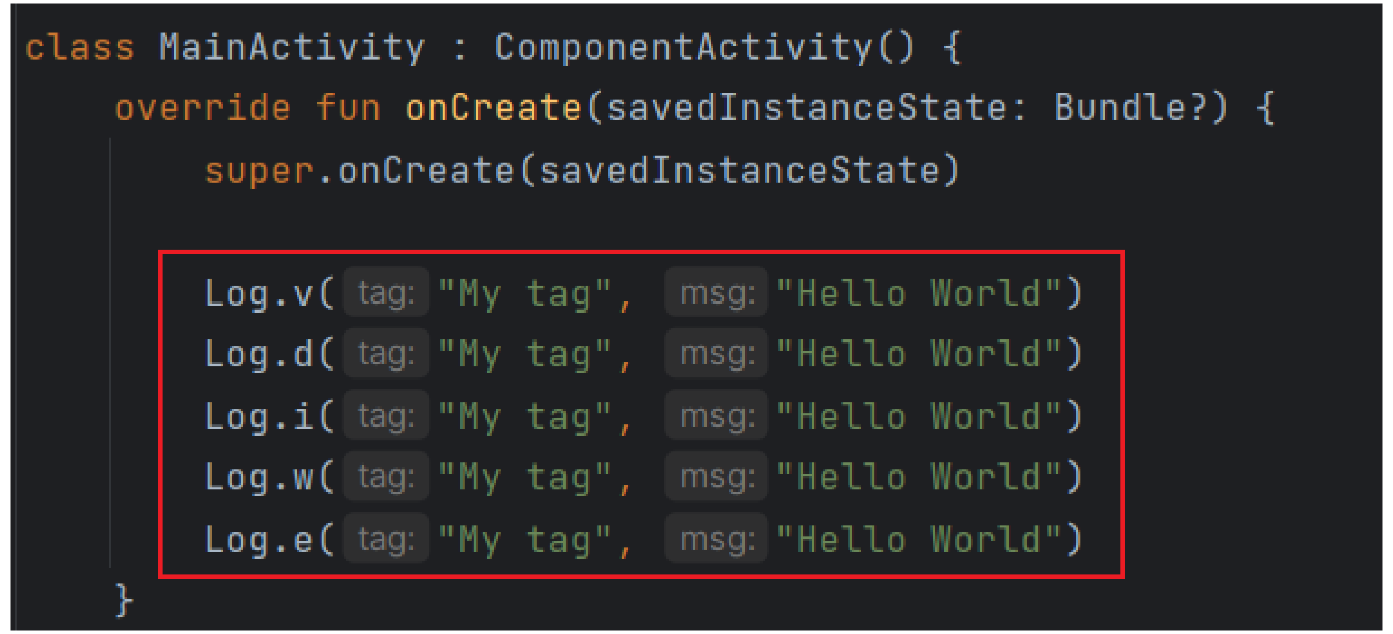
Android 中的日志记录（续）


super.onDestroy()
Logcat
Logcat
Pixel 4 API 34 (emulator-5554) Android 14, API 34
package:mine

2023-10-10 16:16:06.119 9085-9085 ziparchive   com.bcit.lecture6 W Unable to open '/dat
2023-10-10 16:16:06.121 9085-9085 ziparchive   com.bcit.lecture6 W Unable to open '/dat
2023-10-10 16:16:06.122 9085-9085 ziparchive   com.bcit.lecture6 W Unable to open '/dat
2023-10-10 16:16:06.623 9085-9085 nativeloader   com.bcit.lecture6 D Configuring clns-6 f
2023-10-10 16:16:06.690 9085-9085 GraphicsEnvironment com.bcit.lecture6 V Currently set values
2023-10-10 16:16:06.690 9085-9085 GraphicsEnvironment com.bcit.lecture6 V   angle_gl_driver_se
2023-10-10 16:16:06.690 9085-9085 GraphicsEnvironment com.bcit.lecture6 V   angle_gl_driver_se
2023-10-10 16:16:06.697 9085-9085 GraphicsEnvironment com.bcit.lecture6 V ANGLE GameManagerSer

Logging in Android (cont.)

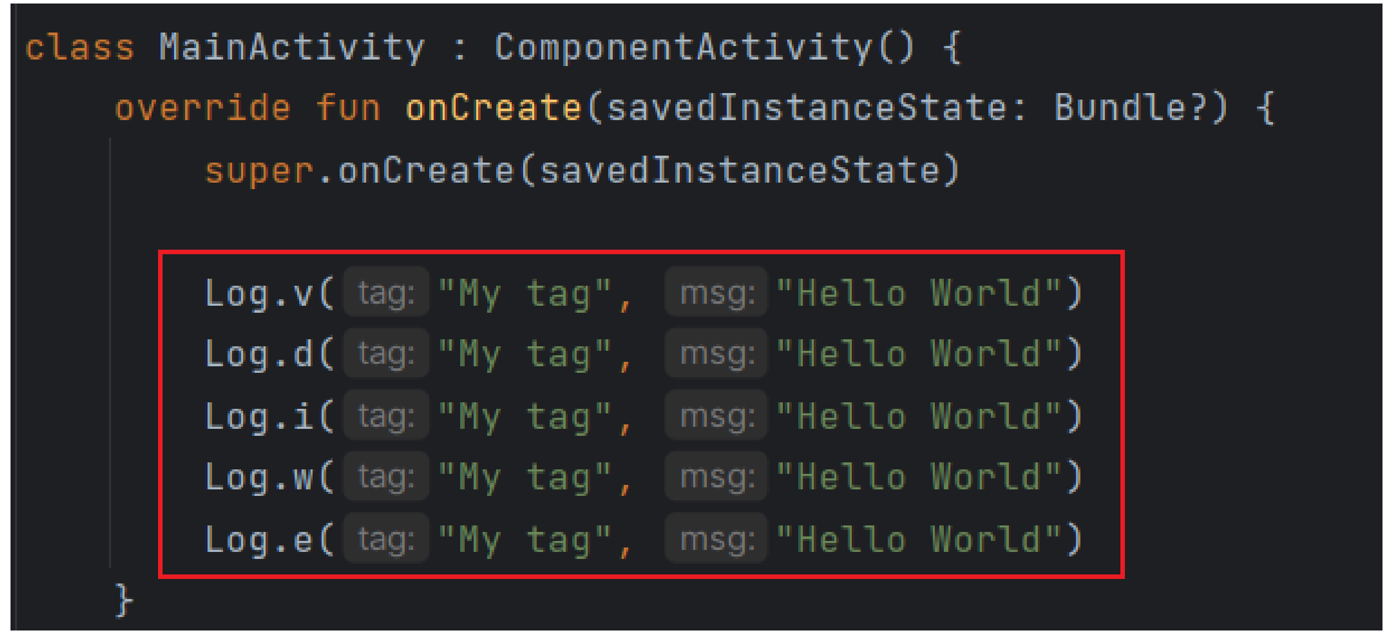
- Here is an example of log the same tag and message for each priority level
  - verbose, debug, info, warning, error



```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        Log.v( tag: "My tag",  msg: "Hello World")
        Log.d( tag: "My tag",  msg: "Hello World")
        Log.i( tag: "My tag",  msg: "Hello World")
        Log.w( tag: "My tag",  msg: "Hello World")
        Log.e( tag: "My tag",  msg: "Hello World")
    }
```

Android 中的日志记录（续）

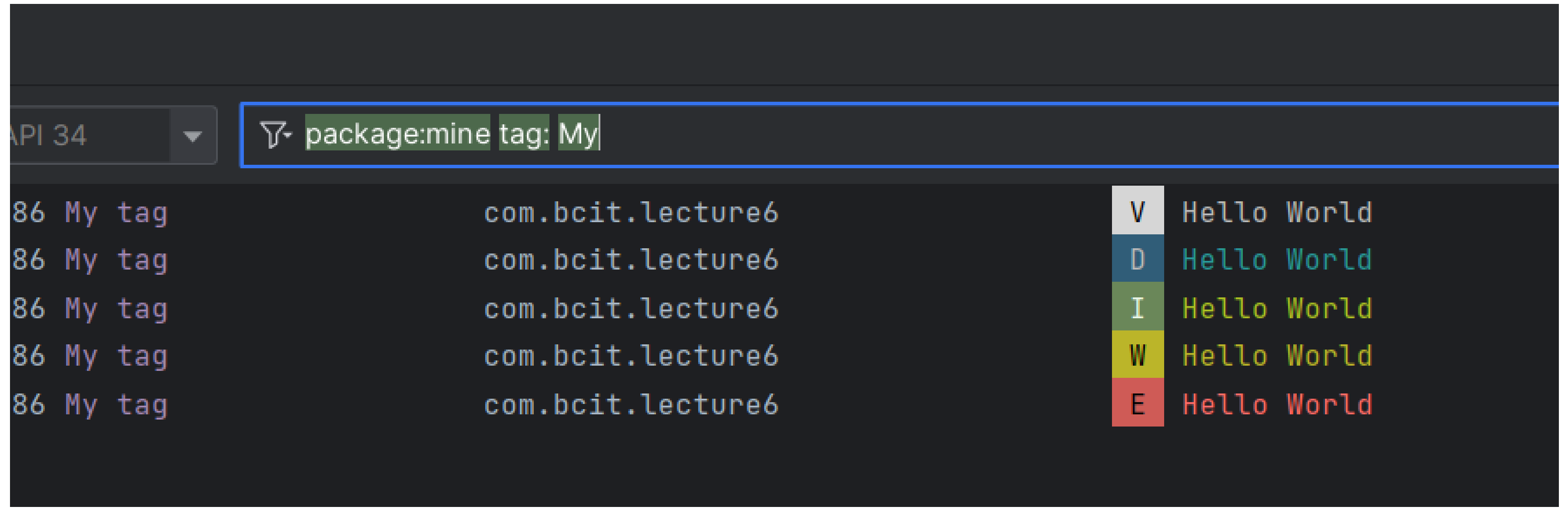
- 以下是一个针对每个优先级使用相同标签和消息进行日志记录的示例
  - 详细、调试、信息、警告、错误



```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        Log.v( tag: "My tag",  msg: "Hello World")
        Log.d( tag: "My tag",  msg: "Hello World")
        Log.i( tag: "My tag",  msg: "Hello World")
        Log.w( tag: "My tag",  msg: "Hello World")
        Log.e( tag: "My tag",  msg: "Hello World")
    }
```

Logging in Android (cont.)

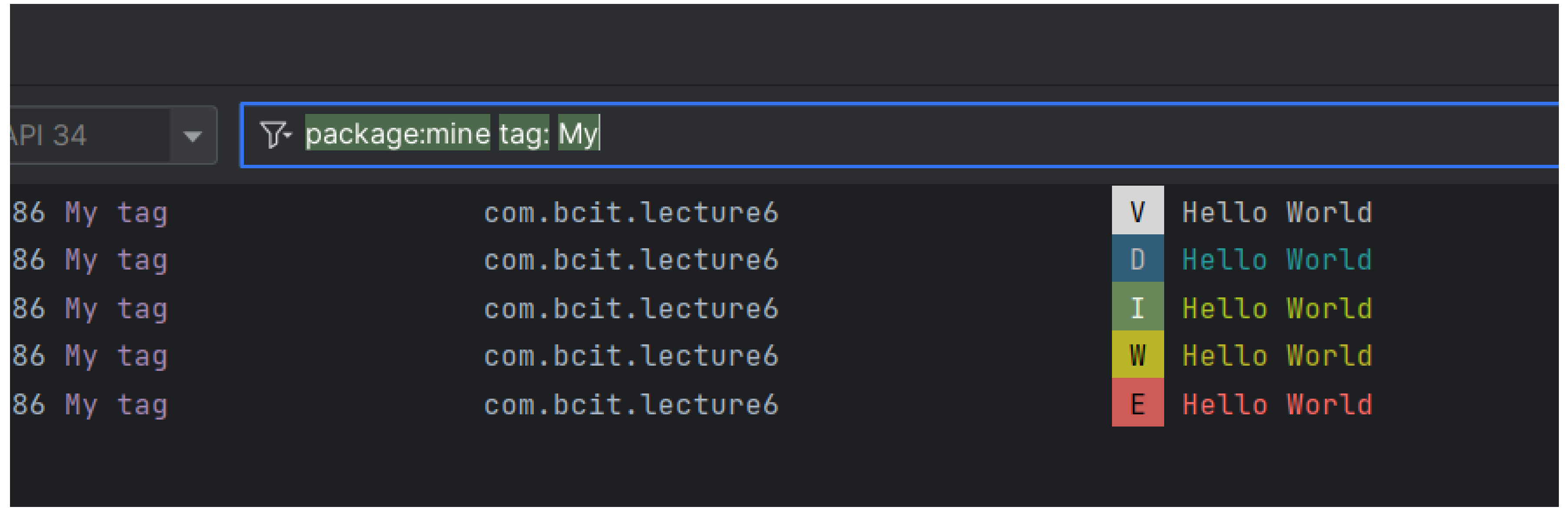
- When I run my code, I can filter out my logs
  - *package:mine* is the default filter
    - It will filter out all logs in your package (ex. com.bcit.lecture6)
  - To filter for your tag you can use *tag: [your tag here]*



```
PI 34   package:mine tag: My
86 My tag        com.bcit.lecture6        V  Hello World
86 My tag        com.bcit.lecture6        D  Hello World
86 My tag        com.bcit.lecture6        I  Hello World
86 My tag        com.bcit.lecture6        W  Hello World
86 My tag        com.bcit.lecture6        E  Hello World
```

Android 中的日志记录（续）

- 当我运行代码时，我可以过滤出我的日志
  - *package:mine* 是默认过滤器
    - 它将过滤掉你包中的所有日志（例如 com.bcit.lecture6）
  - 要按你的标签进行过滤，可以使用 *tag: [your tag here]*

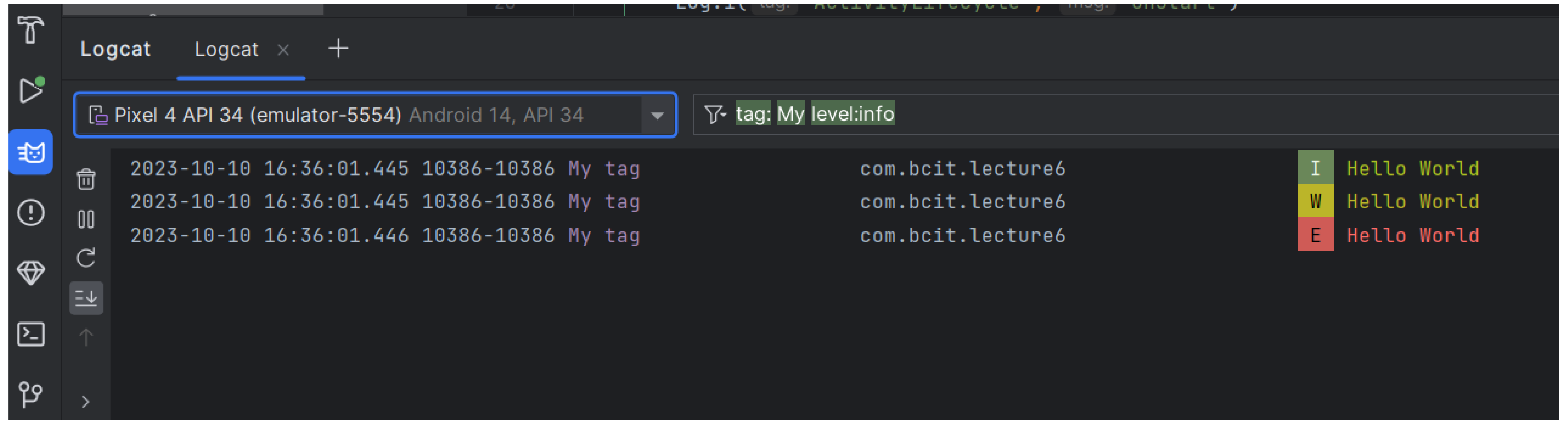


```
PI 34   package:mine tag: My
86 My tag        com.bcit.lecture6        V  Hello World
86 My tag        com.bcit.lecture6        D  Hello World
86 My tag        com.bcit.lecture6        I  Hello World
86 My tag        com.bcit.lecture6        W  Hello World
86 My tag        com.bcit.lecture6        E  Hello World
```

Logging in Android (cont.)

- Here we are filtering all tags in our whole project that start with *My*

- We are then only filtering those tags that are of priority *info* or higher



Logcat Logcat × +

Pixel 4 API 34 (emulator-5554) Android 14, API 34

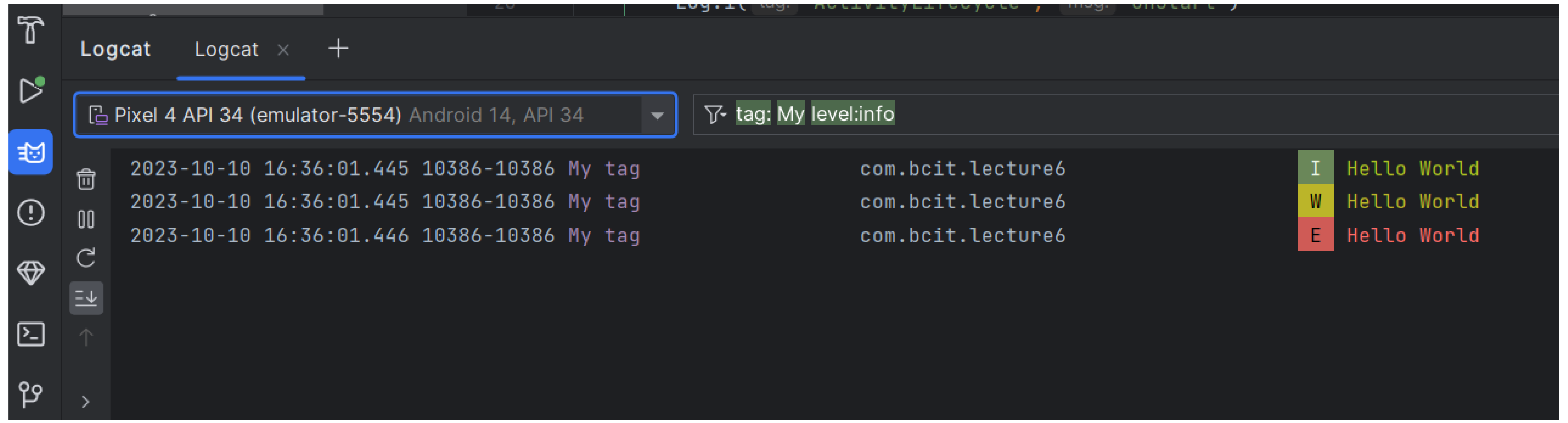
tag: My level:info

```
2023-10-10 16:36:01.445 10386-10386 My tag                 com.bcit.lecture6                   I  Hello World
2023-10-10 16:36:01.445 10386-10386 My tag                 com.bcit.lecture6                   W  Hello World
2023-10-10 16:36:01.446 10386-10386 My tag                 com.bcit.lecture6                   E  Hello World
```

Android 中的日志记录（续）

- 我们正在过滤整个项目中所有以 *My* 开头的标签

- 然后我们仅过滤那些优先级为 *info* 或更高的标签



Logcat Logcat × +

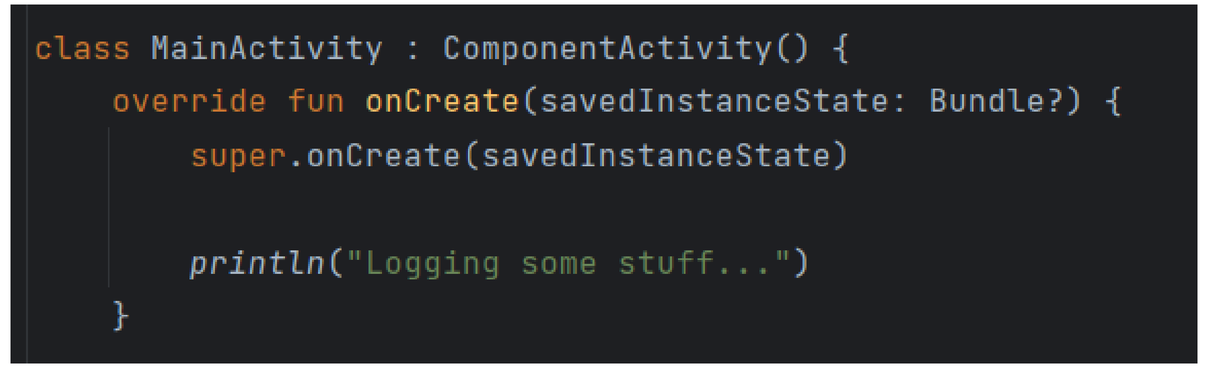
Pixel 4 API 34 (emulator-5554) Android 14, API 34

tag: My level:info

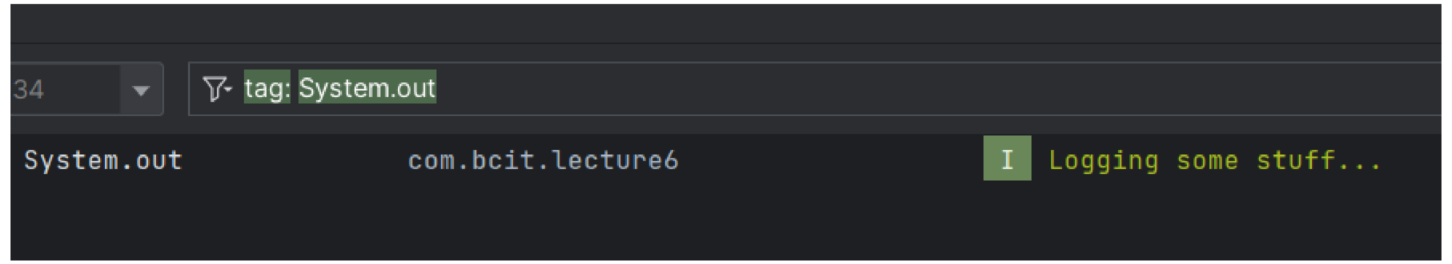
```
2023-10-10 16:36:01.445 10386-10386 My tag                 com.bcit.lecture6                   I  Hello World
2023-10-10 16:36:01.445 10386-10386 My tag                 com.bcit.lecture6                   W  Hello World
2023-10-10 16:36:01.446 10386-10386 My tag                 com.bcit.lecture6                   E  Hello World
```

Logging in Android (cont.)

- Logging with custom tags is recommended but you can still use *println* if you want
  - The tag is *System.out*



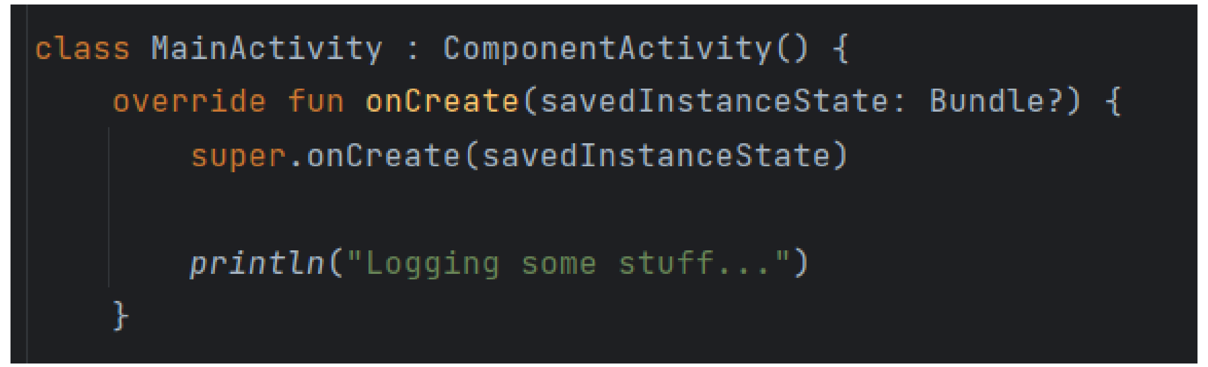
```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        println("Logging some stuff...")
    }
```



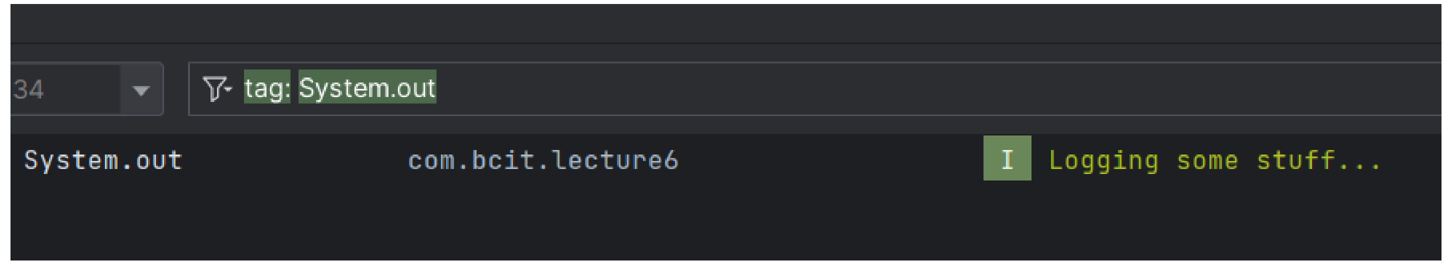
```
34  ▾   tag: System.out
System.out          com.bcit.lecture6          I  Logging some stuff...
```

Android 中的日志记录（续）

- 建议使用自定义标签进行日志记录，但如果你愿意，仍然可以使用 *println*
  - 该标签为*System.out*



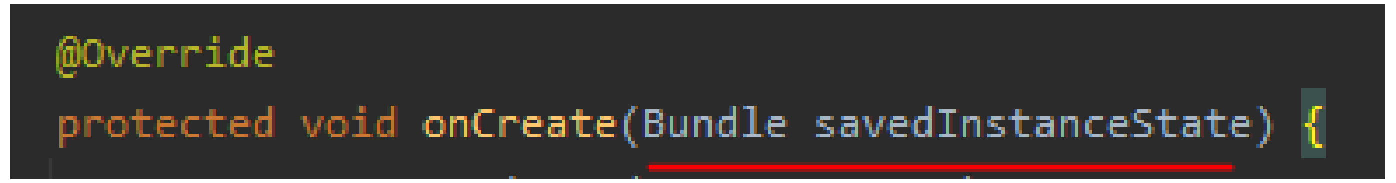
```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        println("Logging some stuff...")
    }
```



```
34  ▾   tag: System.out
System.out          com.bcit.lecture6          I  Logging some stuff...
```

Bundles

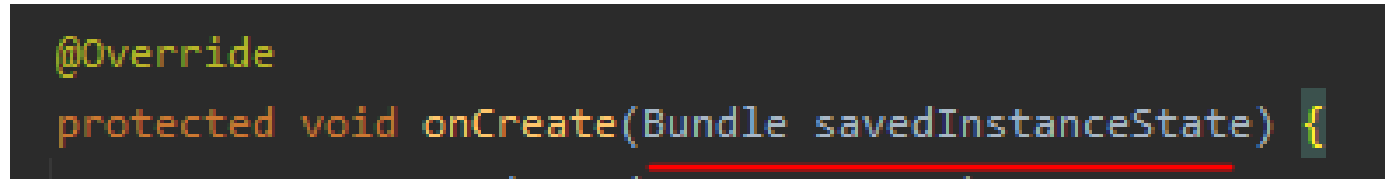
- Bundles in Android are very similar to Maps
  - Key-value mappings used with android specific components
- Used to pass data between activities through *intents*
  - When multiple activities were more common
- Used to restore state information



```
@Override
protected void onCreate(Bundle savedInstanceState) {
```

Bundles

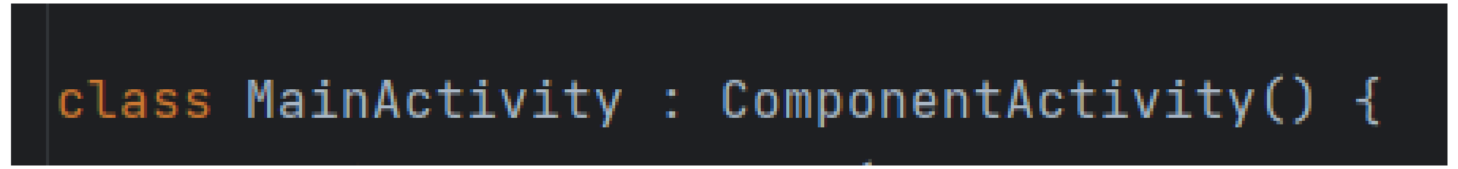
- Android 中的 Bundle 与 Map 非常相似
  - 与 Android 特定组件一起使用的键值映射
- 通过 *intents* 在活动之间传递数据时使用
  - 当多个活动更为常见时
- 用于恢复状态信息



```
@Override
protected void onCreate(Bundle savedInstanceState) {
```

Activities (cont.)

- You will notice your *MainActivity* is a subclass of *ComponentActivity*



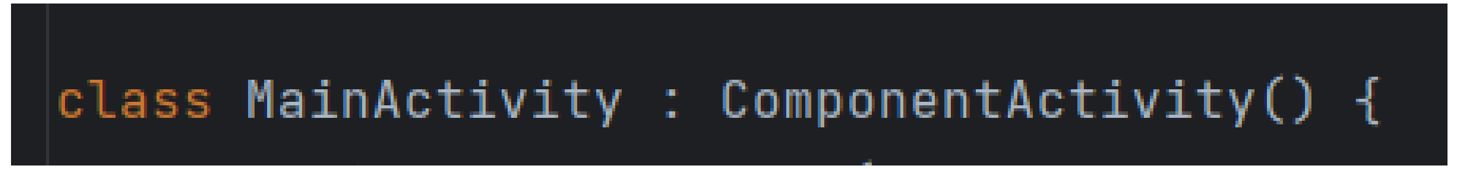
```
class MainActivity : ComponentActivity() {
```

- *ComponentActivity* is a subclass of *Activity*

- The ComponentActivity class allows us to use *composable functions*

活动（续）

- 你会注意到你的 *MainActivity* 是 *ComponentActivity* 的子类



```
class MainActivity : ComponentActivity() {
```

- ComponentActivity 是 Activity 的子类

- ComponentActivity 类使我们能够使用可组合函数

Jetpack Compose

- Modern toolkit for building native Android UI
  - Composable functions (Composables for short)
- Replaces the old way of using the Layout Editor with XML files
- Used by the Kotlin programming language

Jetpack Compose

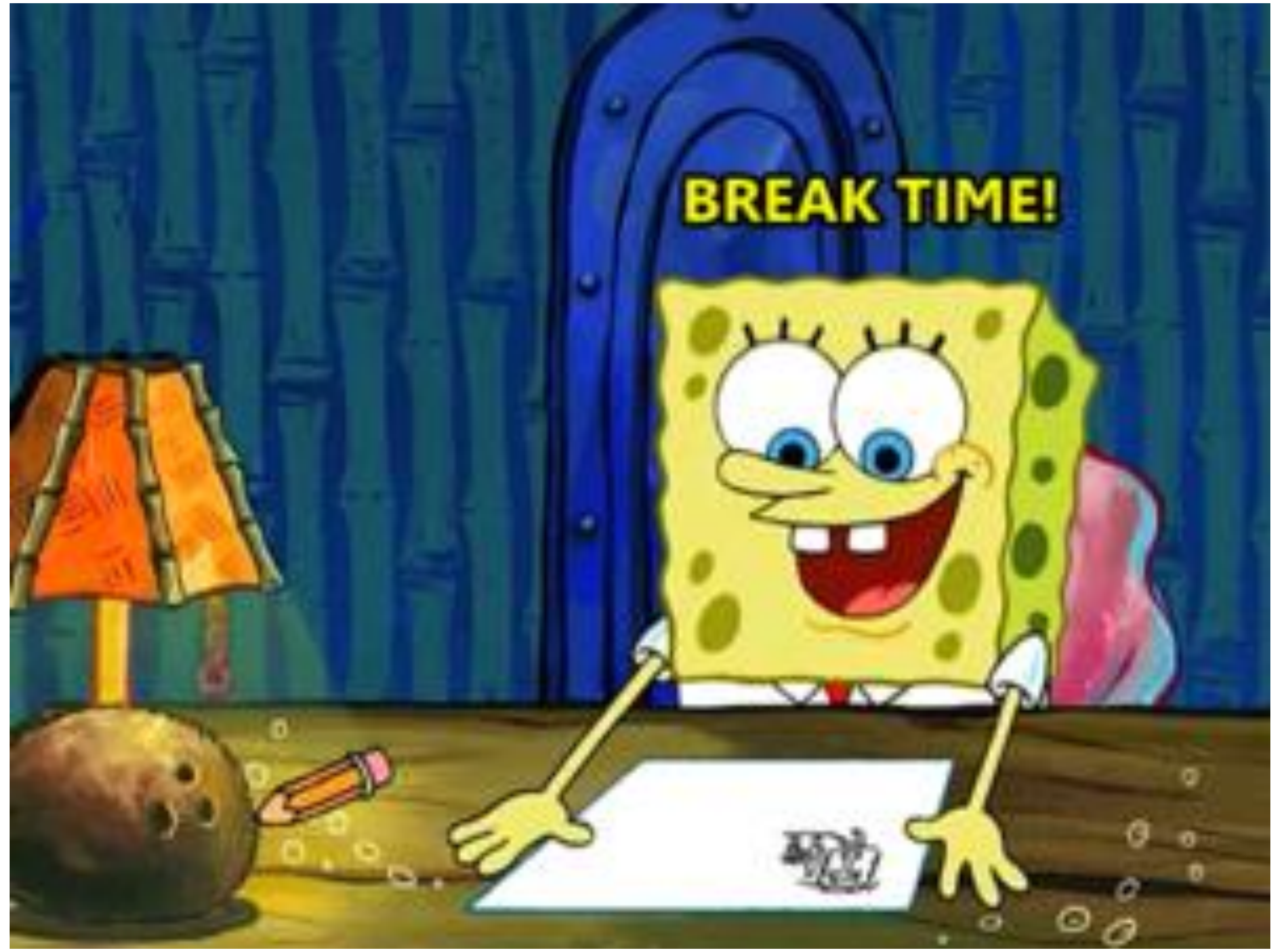
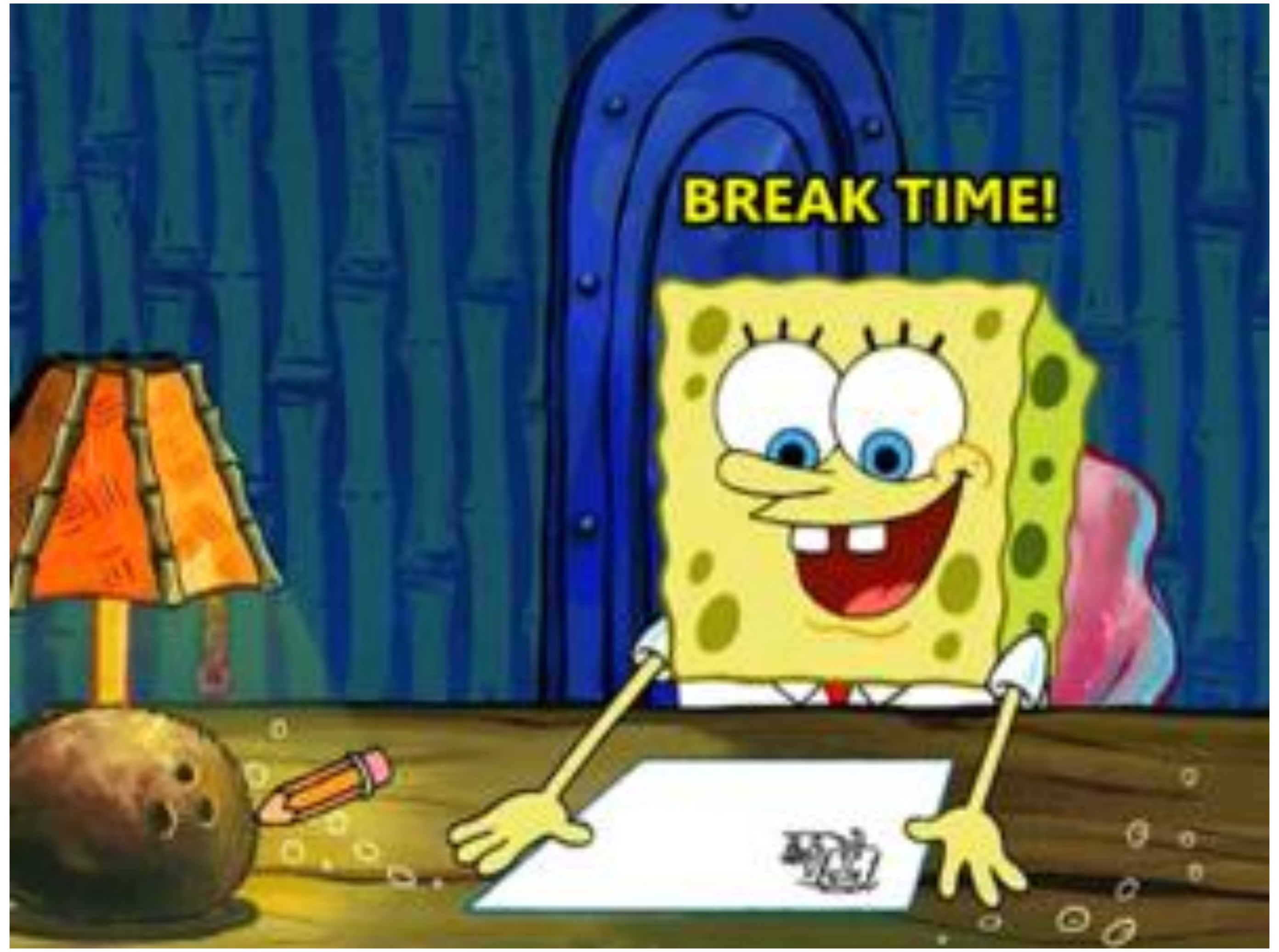
- 用于构建原生 Android UI 的现代工具包
  - 可组合函数（简称 Composables）
- 取代了使用 XML 文件和布局编辑器的旧方法
- 由 Kotlin 编程语言使用

Jetpack Compose (cont.)

- The shift from imperative UI programming to declarative has sped up development time significantly
  - Ex. React and Flutter have made the shift
- With imperative you focus on how the UI is created, positioned and updated
- With declarative we focus more on what the UI should look like, and the imperative part is taken care for us

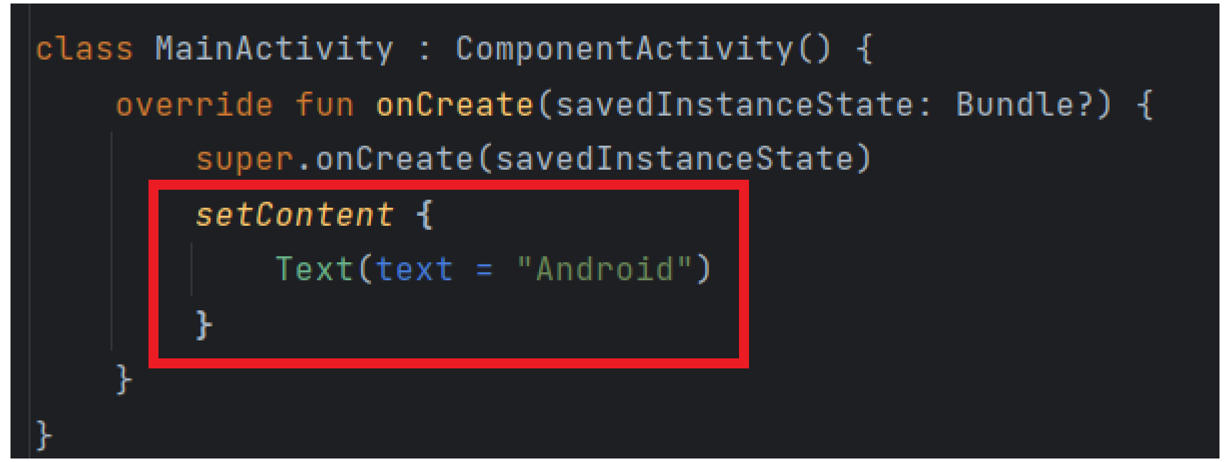
Jetpack Compose（续）

- 从命令式UI编程转向声明式的速度已加快显著缩短了开发时间
  - 例如，React 和 Flutter 已经完成了这一转变
- 在命令式编程中，你关注的是UI如何被创建、定位和更新
- 在声明式编程中，我们更关注UI应该呈现为什么样子，而命令式的部分则由系统自动处理


BREAK TIME!

BREAK TIME!

Composable Functions

- *setContent* defines the layout for the Activity
    - This is where we put our composables

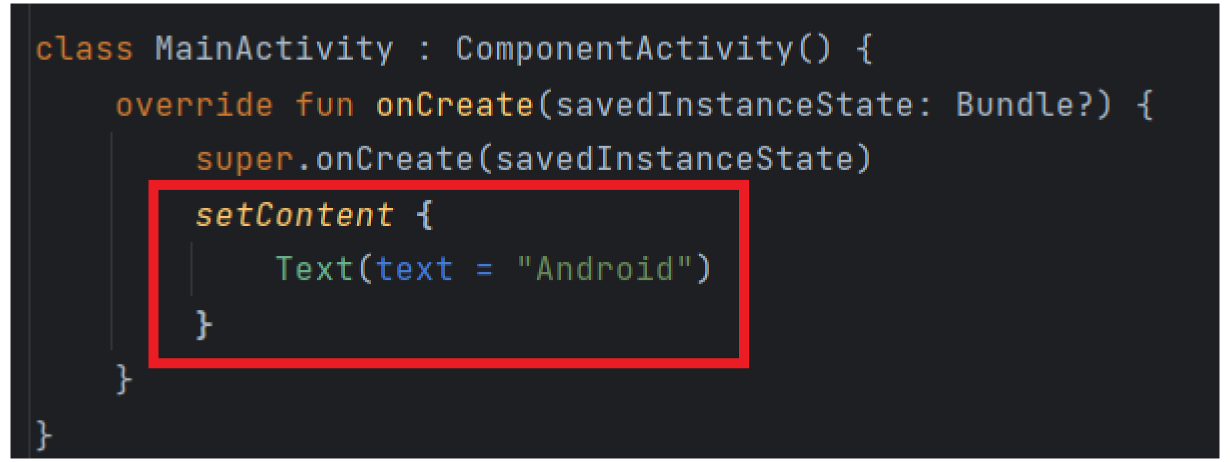


```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Text(text = "Android")
        }
    }
}
```

- *Text* is an example of a composable function

可组合函数

- *setContent* 用于定义 Activity 的布局
    - 这是我们放置可组合项的地方

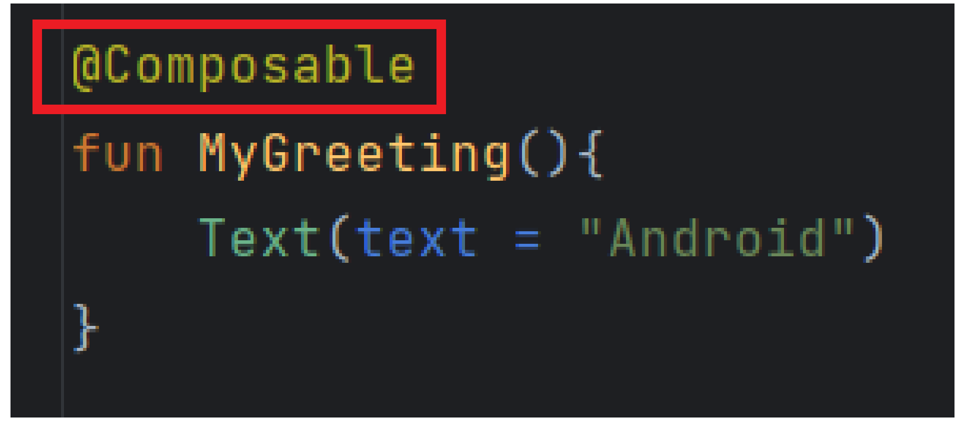


```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Text(text = "Android")
        }
    }
}
```

- *Text* 就是一个可组合函数的示例

Composable Functions (cont.)

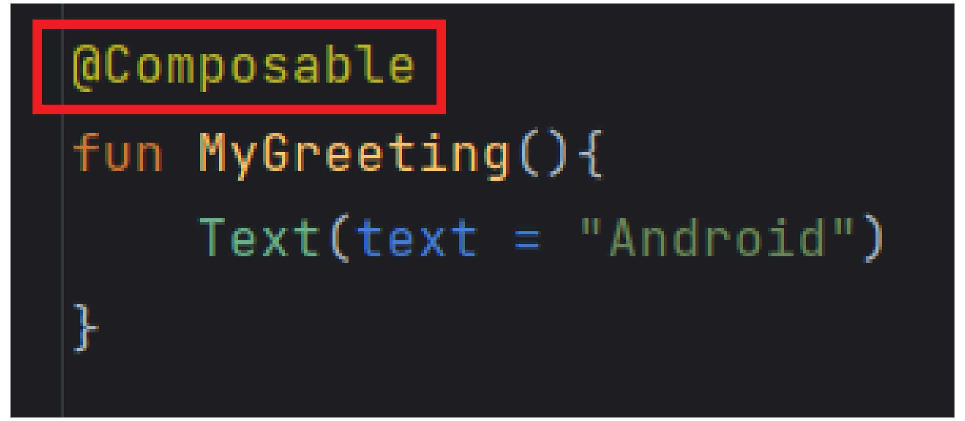
- Compose provides us with many composable functions to create our UI
  - Text, Button, Image, Card, and many more
- We can create our own composables using the annotation *@Composable*



```
@Composable
fun MyGreeting(){
    Text(text = "Android")
}
```

可组合函数（续）

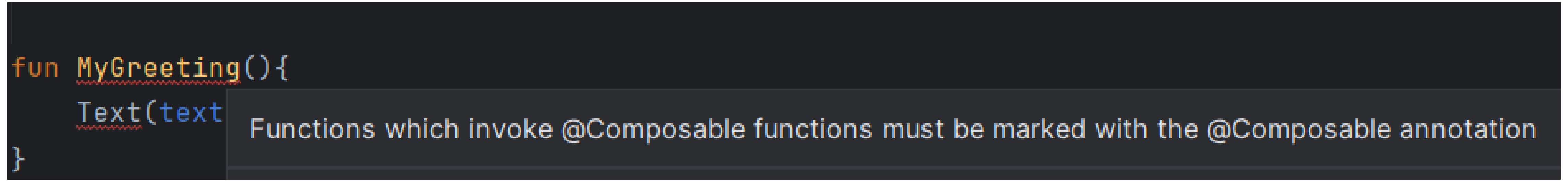
- Compose 为我们提供了许多可组合函数来创建我们的用户界面
  - 文本、按钮、图像、卡片以及更多
- 我们可以使用注解 *@Composable* 创建自己的可组合项



```
@Composable
fun MyGreeting(){
    Text(text = "Android")
}
```

Composable Functions (cont.)

- You cannot put a composable function inside a regular function

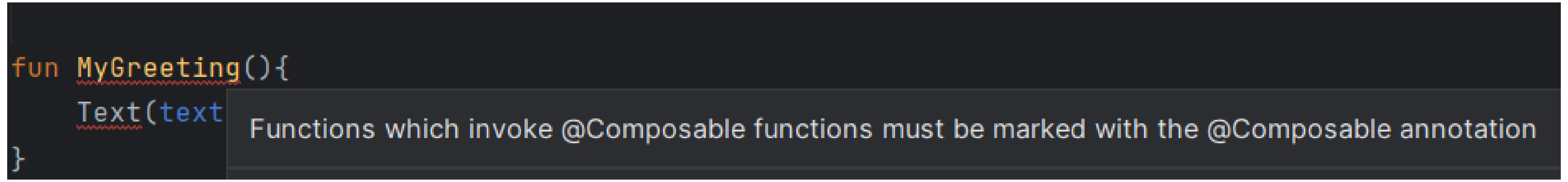


```
fun MyGreeting(){
    Text(text
}
```

Functions which invoke @Composable functions must be marked with the @Composable annotation

可组合函数（续）

- 你不能将可组合函数放在普通函数内部

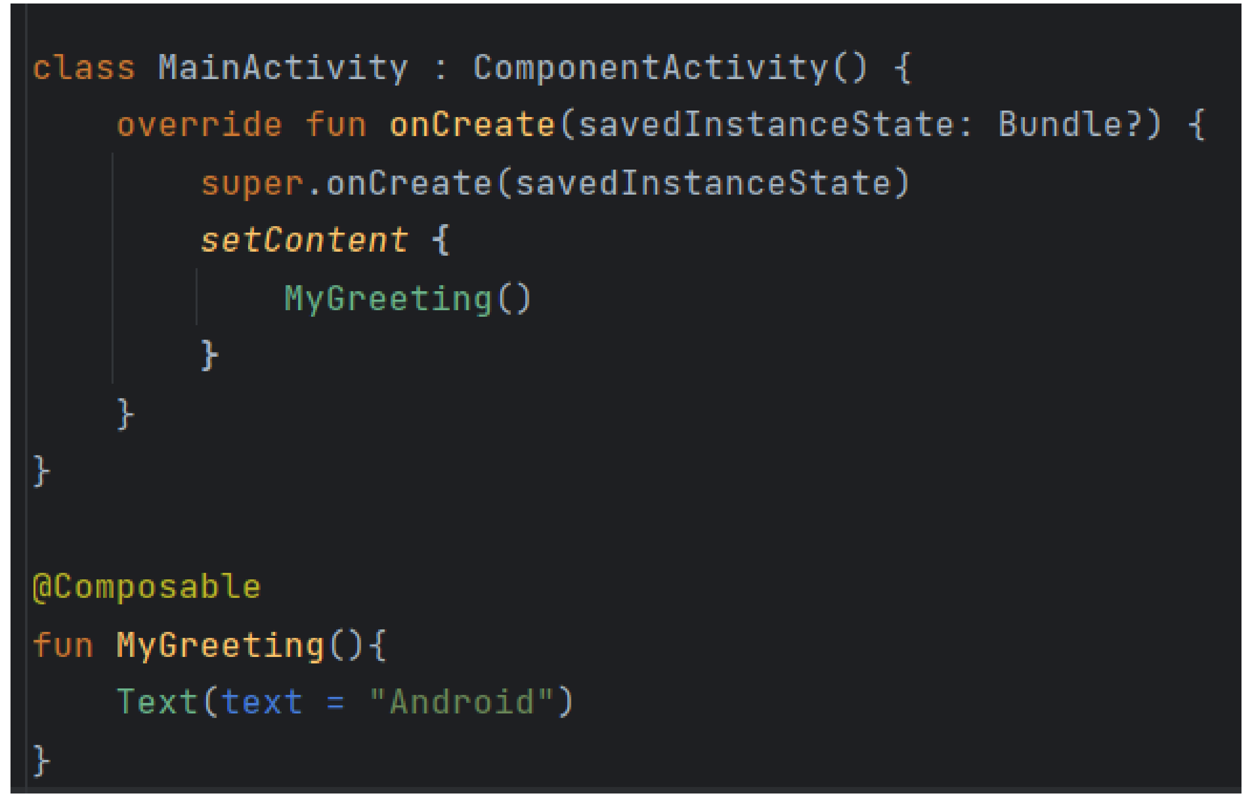


```
fun MyGreeting(){
    Text(text
}
```

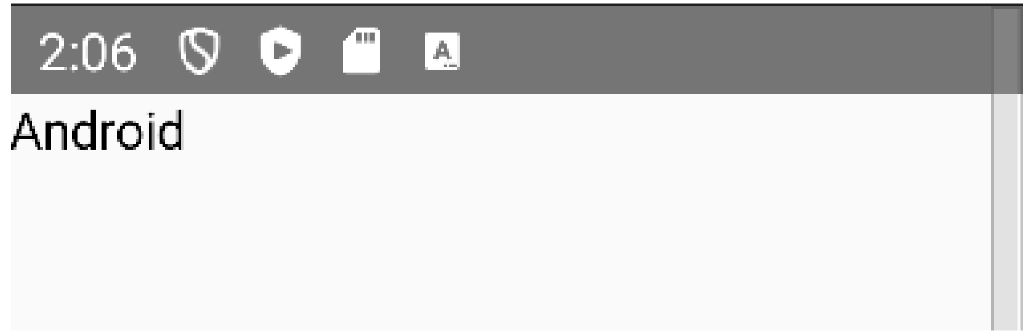
Functions which invoke @Composable functions must be marked with the @Composable annotation

Composable Functions (cont.)

- Lets try out our composable



```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            MyGreeting()
        }
    }
}

@Composable
fun MyGreeting(){
    Text(text = "Android")
}
```

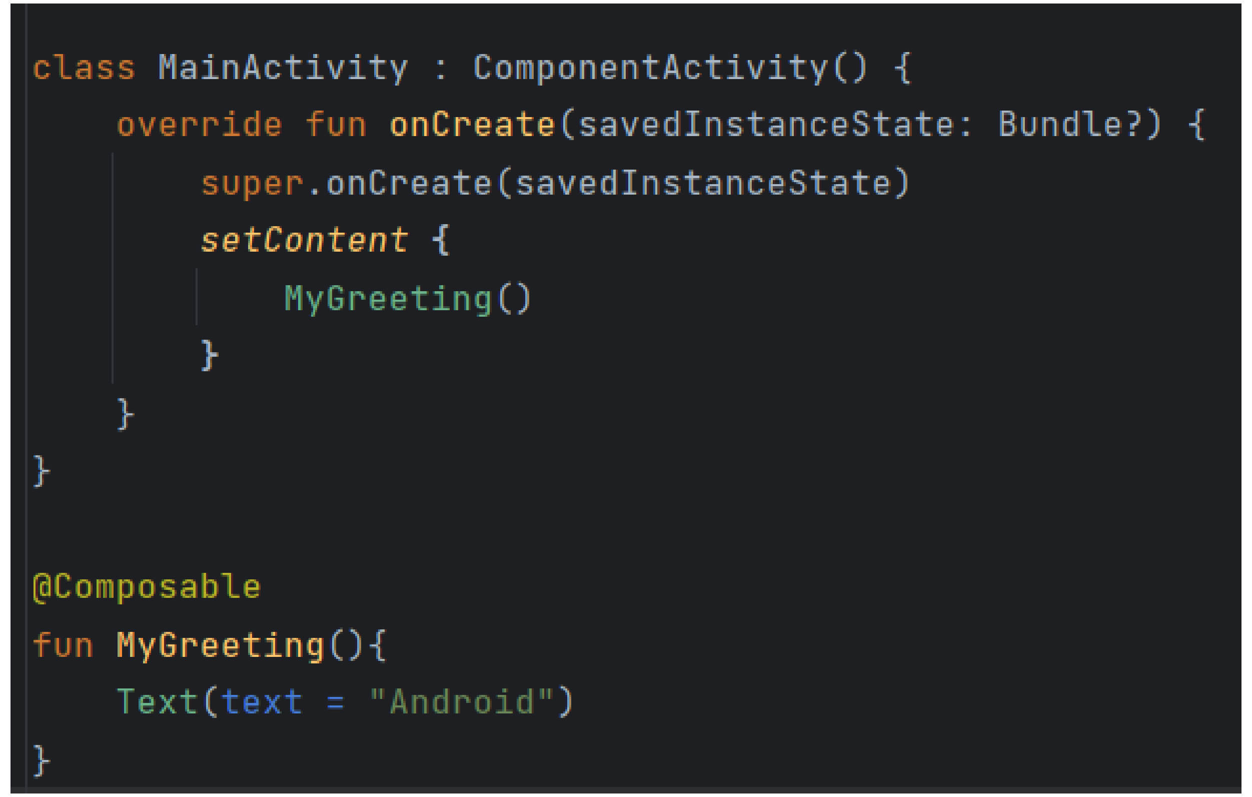


2:06

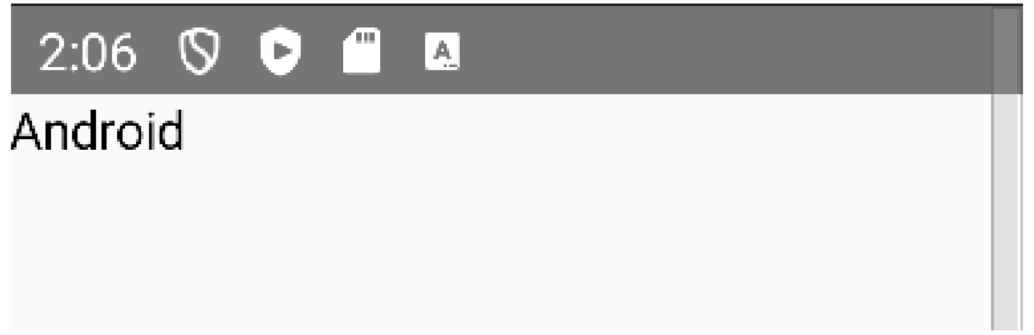
Android

可组合函数（续）

- 让我们试用一下我们的可组合函数



```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            MyGreeting()
        }
    }
}

@Composable
fun MyGreeting(){
    Text(text = "Android")
}
```

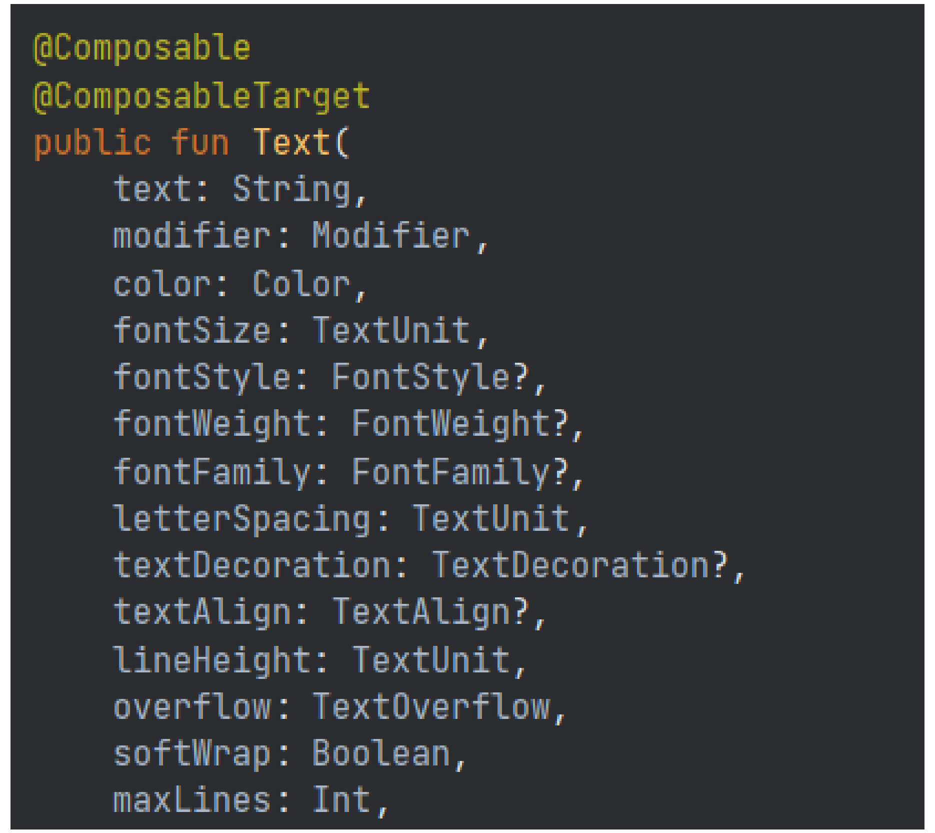


2:06

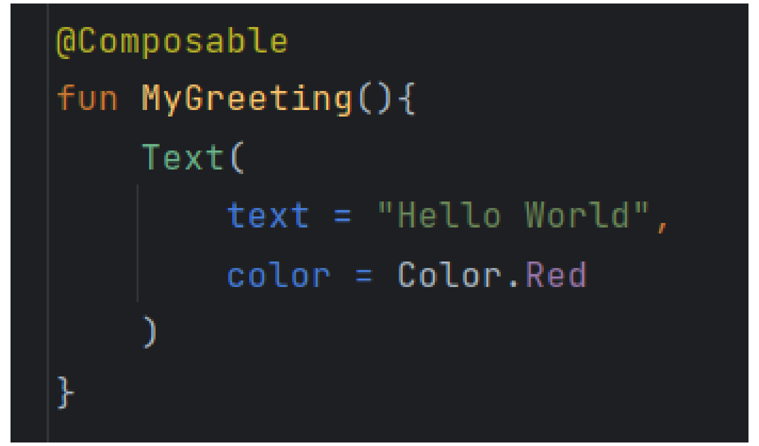
Android

Composable function parameters

- Like any other regular function in Kotlin, composables can have parameters



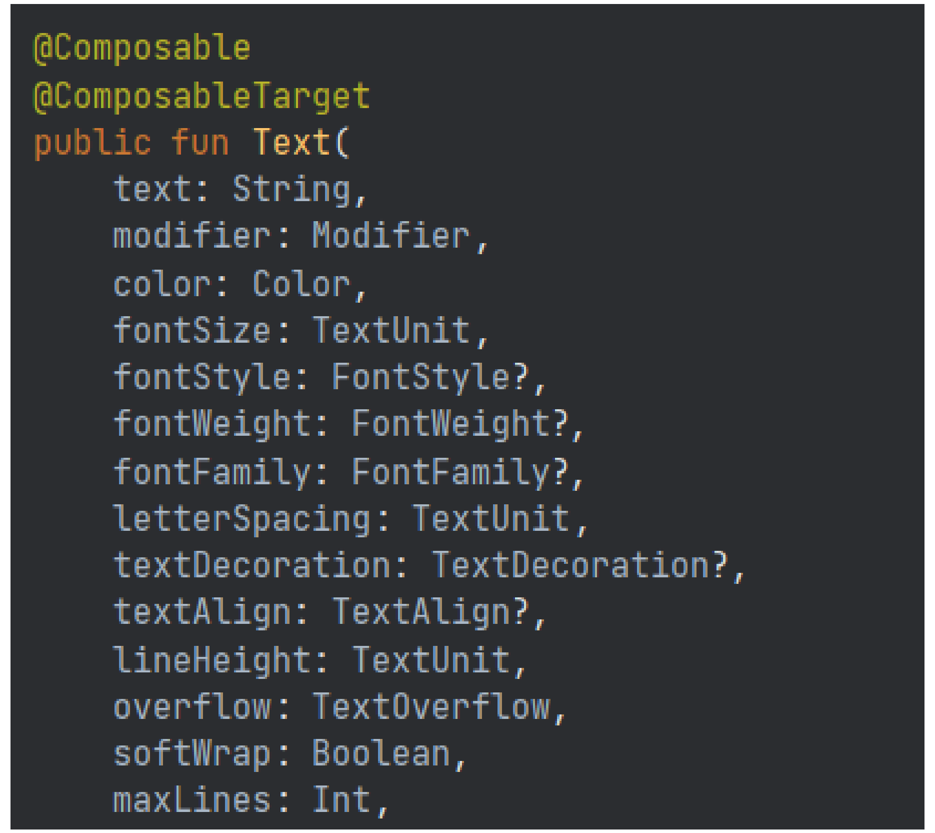
```
@Composable
@ComposableTarget
public fun Text(
    text: String,
    modifier: Modifier,
    color: Color,
    fontSize: TextUnit,
    fontStyle: FontStyle?,
    fontWeight: FontWeight?,
    fontFamily: FontFamily?,
    letterSpacing: TextUnit,
    textDecoration: TextDecoration?,
    textAlign: TextAlign?,
    lineHeight: TextUnit,
    overflow: TextOverflow,
    softWrap: Boolean,
    maxLines: Int,
```



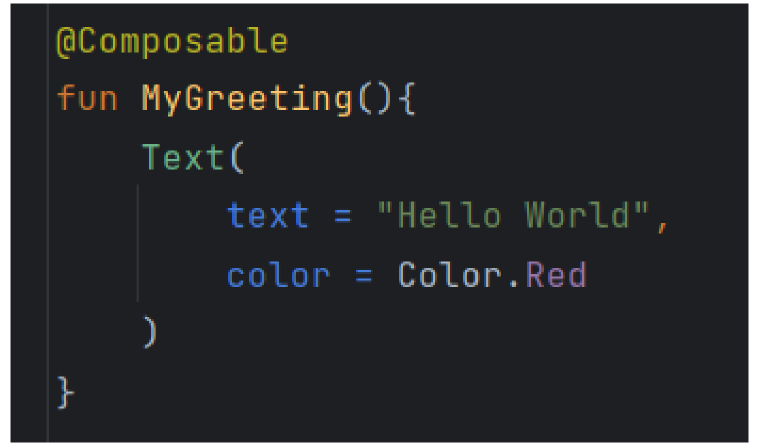
```
@Composable
fun MyGreeting(){
    Text(
        text = "Hello World",
        color = Color.Red
    )
}
```

可组合函数参数

- 与 Kotlin 中的任何其他普通函数一样，可组合函数也可以具有参数



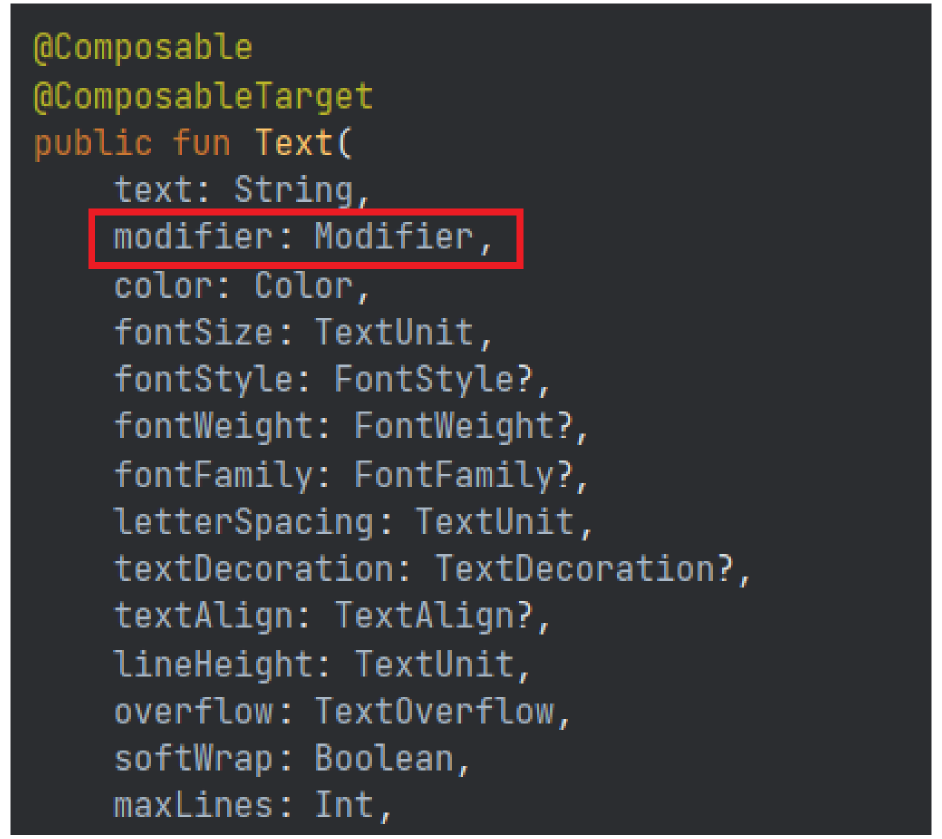
```
@Composable
@ComposableTarget
public fun Text(
    text: String,
    modifier: Modifier,
    color: Color,
    fontSize: TextUnit,
    fontStyle: FontStyle?,
    fontWeight: FontWeight?,
    fontFamily: FontFamily?,
    letterSpacing: TextUnit,
    textDecoration: TextDecoration?,
    textAlign: TextAlign?,
    lineHeight: TextUnit,
    overflow: TextOverflow,
    softWrap: Boolean,
    maxLines: Int,
```



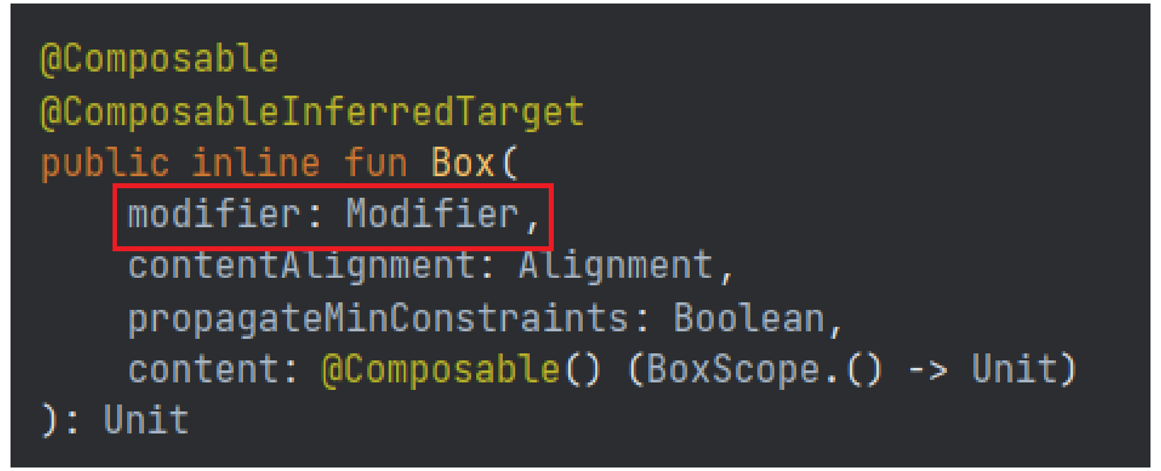
```
@Composable
fun MyGreeting(){
    Text(
        text = "Hello World",
        color = Color.Red
    )
}
```

Composable function parameters (cont.)

- One common parameter that many composables share is *modifier*



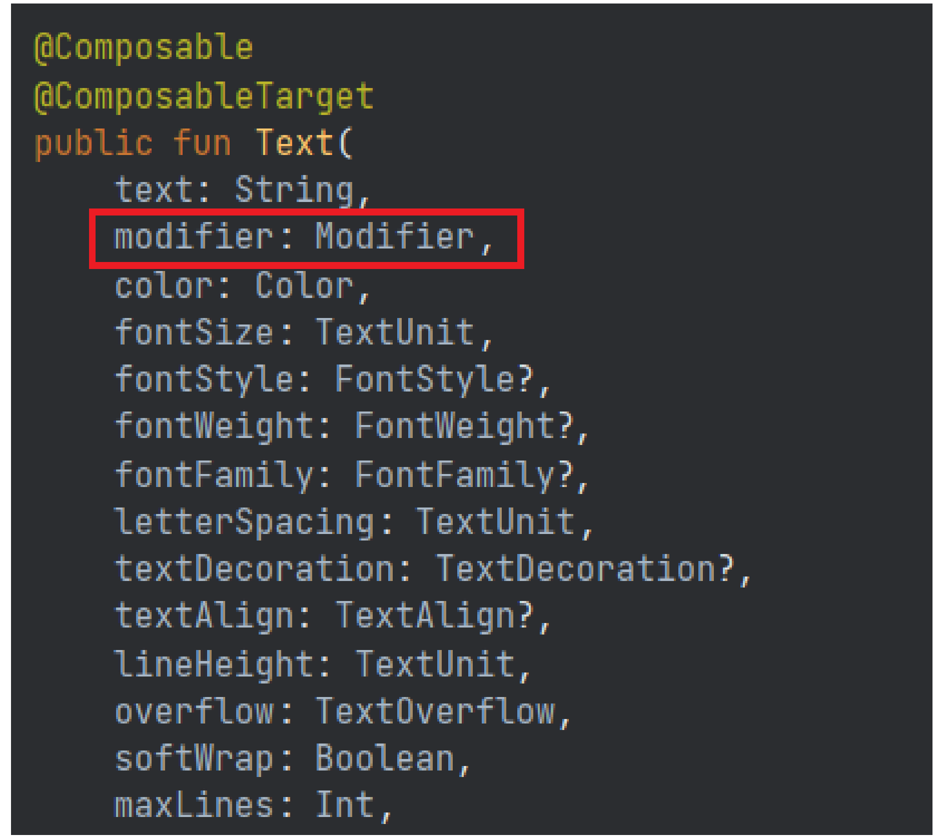
```
@Composable
@ComposableTarget
public fun Text(
    text: String,
    modifier: Modifier,
    color: Color,
    fontSize: TextUnit,
    fontStyle: FontStyle?,
    fontWeight: FontWeight?,
    fontFamily: FontFamily?,
    letterSpacing: TextUnit,
    textDecoration: TextDecoration?,
    textAlign: TextAlign?,
    lineHeight: TextUnit,
    overflow: TextOverflow,
    softWrap: Boolean,
    maxLines: Int,
```



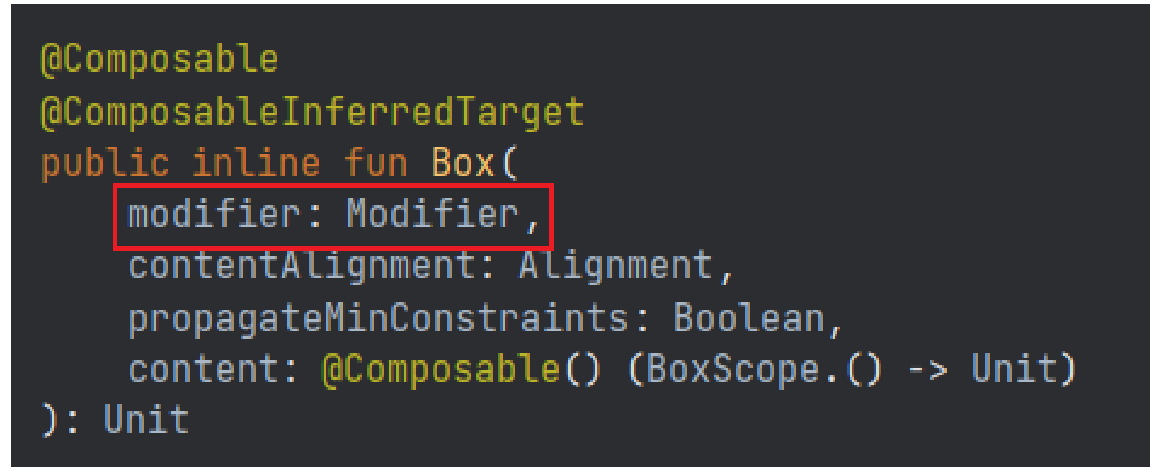
```
@Composable
@ComposableInferredTarget
public inline fun Box(
    modifier: Modifier,
    contentAlignment: Alignment,
    propagateMinConstraints: Boolean,
    content: @Composable() (BoxScope.() -> Unit)
): Unit
```

可组合函数参数（续）

- 许多可组合项共有的一个常见参数是 *modifier*



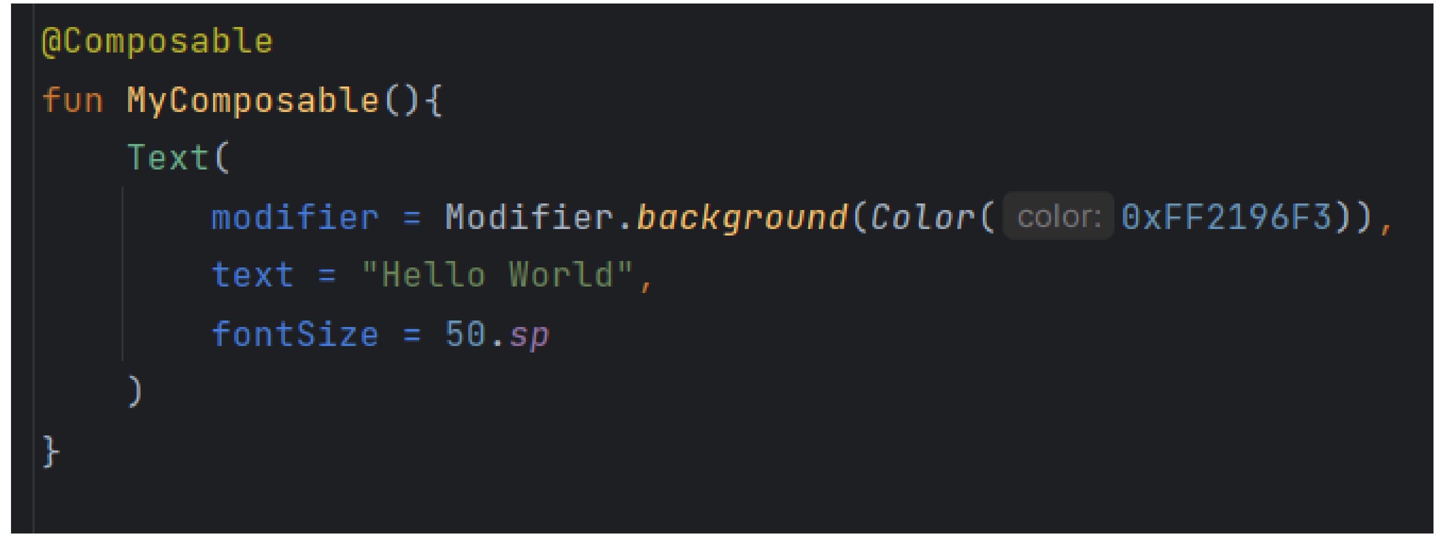
```
@Composable
@ComposableTarget
public fun Text(
    text: String,
    modifier: Modifier,
    color: Color,
    fontSize: TextUnit,
    fontStyle: FontStyle?,
    fontWeight: FontWeight?,
    fontFamily: FontFamily?,
    letterSpacing: TextUnit,
    textDecoration: TextDecoration?,
    textAlign: TextAlign?,
    lineHeight: TextUnit,
    overflow: TextOverflow,
    softWrap: Boolean,
    maxLines: Int,
```



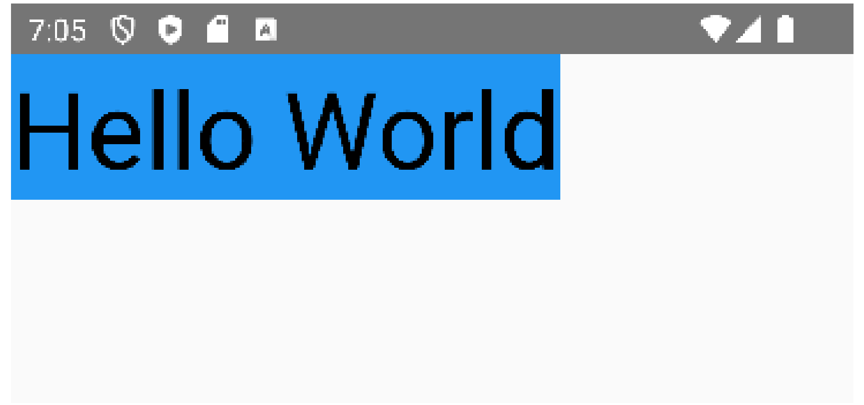
```
@Composable
@ComposableInferredTarget
public inline fun Box(
    modifier: Modifier,
    contentAlignment: Alignment,
    propagateMinConstraints: Boolean,
    content: @Composable() (BoxScope.() -> Unit)
): Unit
```

Composable modifiers

- *Modifiers* are used to decorate or configure a composable
  - They allow you change the size, layout, appearance and more to the composable



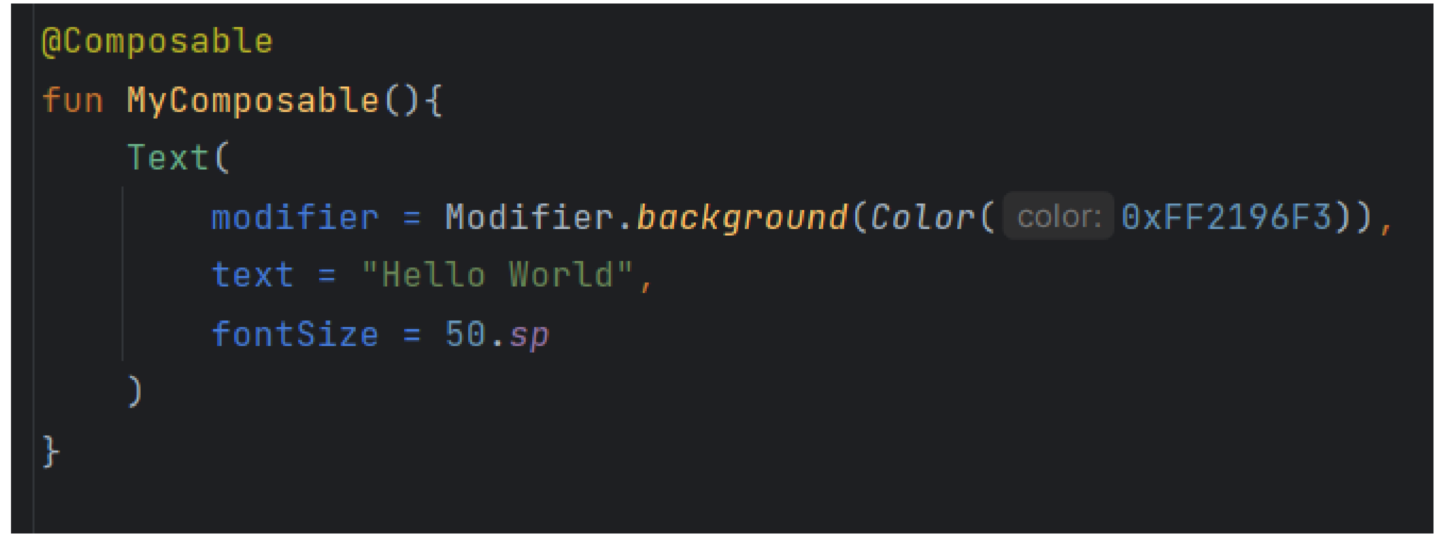
```
@Composable
fun MyComposable(){
    Text(
        modifier = Modifier.background(Color( color: 0xFF2196F3)),
        text = "Hello World",
        fontSize = 50.sp
    )
}
```



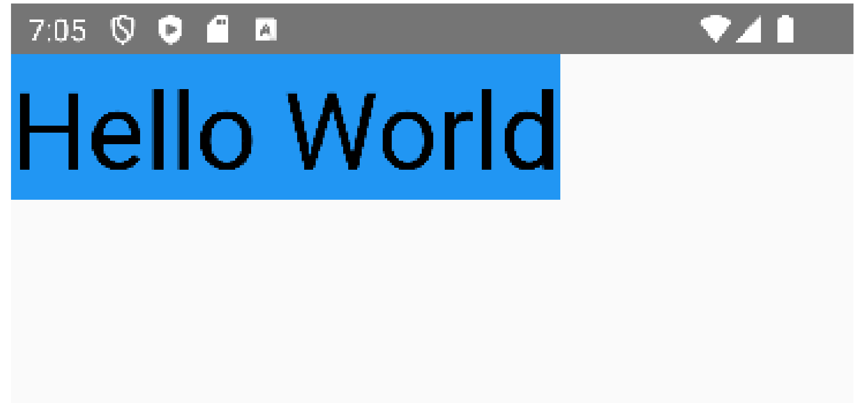
```
Hello World
```

可组合修饰符

- 修饰符 用于装饰或配置一个可组合项
  - 它们允许你更改可组合项的大小、布局、外观等内容



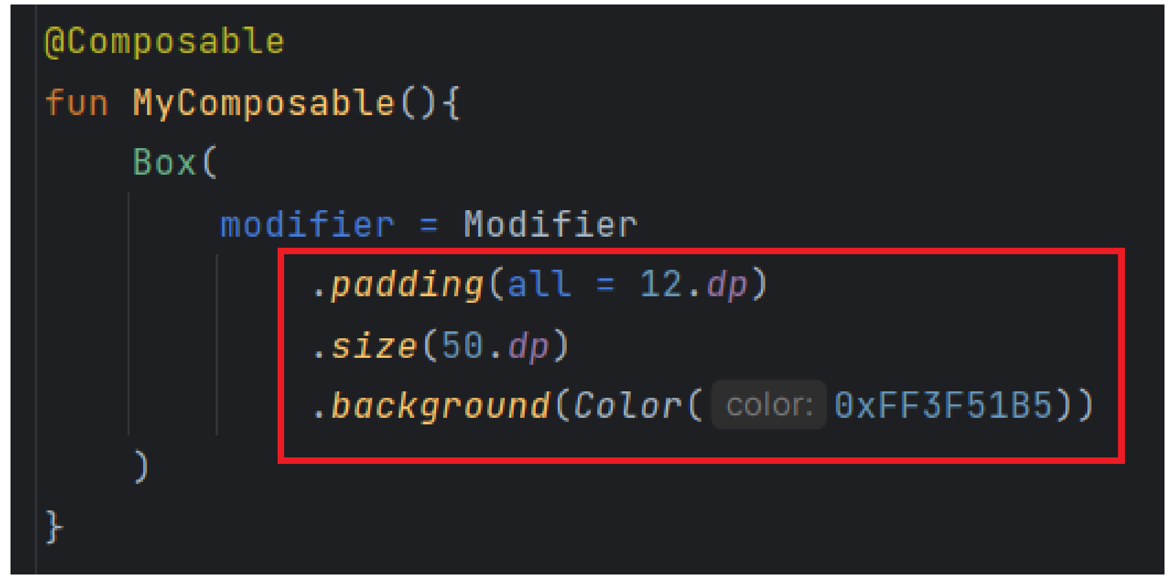
```
@Composable
fun MyComposable(){
    Text(
        modifier = Modifier.background(Color( color: 0xFF2196F3)),
        text = "Hello World",
        fontSize = 50.sp
    )
}
```



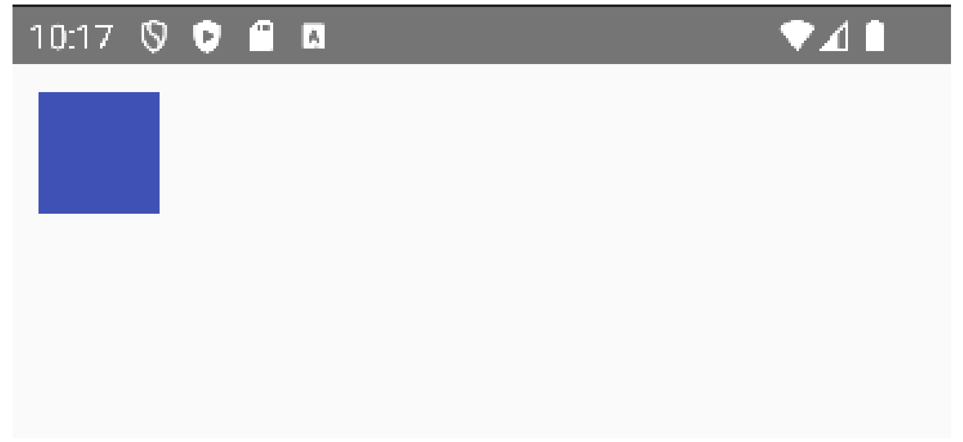
```
Hello World
```

Composable modifiers (cont.)

- A composable can have multiple *modifiers* by chaining them together



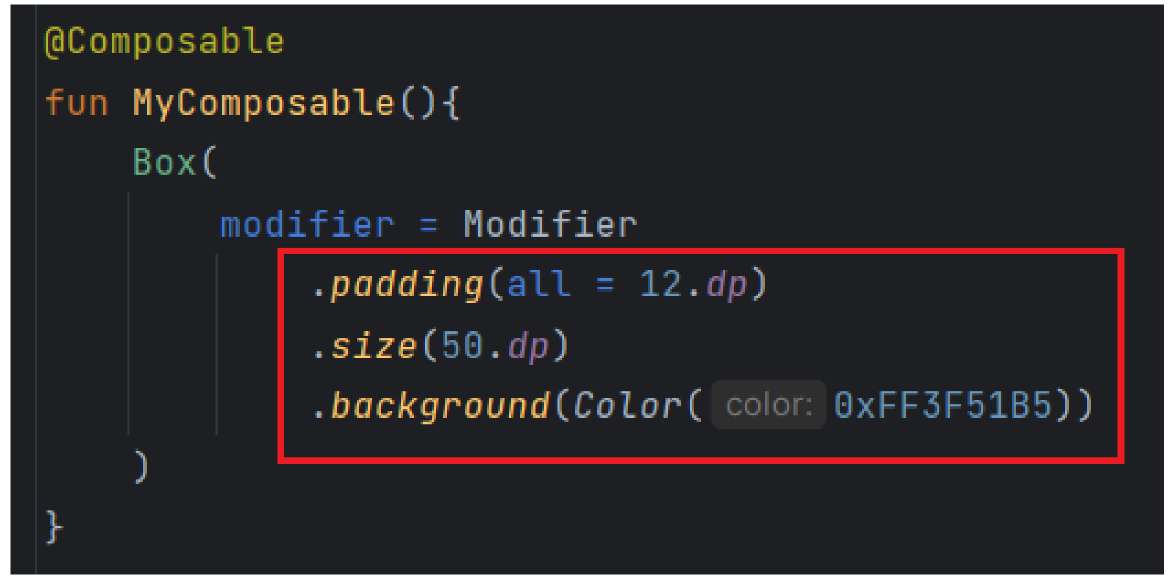
```
@Composable
fun MyComposable(){
    Box(
        modifier = Modifier
            .padding(all = 12.dp)
            .size(50.dp)
            .background(Color( color: 0xFF3F51B5))
    )
}
```



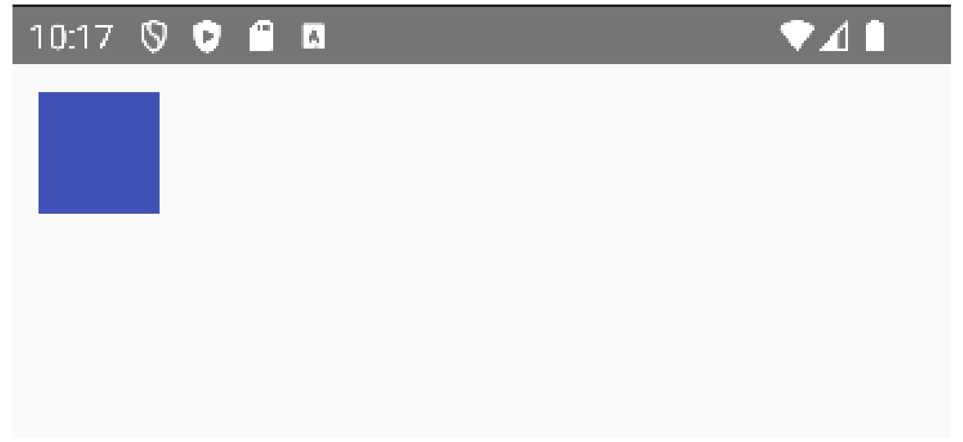
- A Box is a simple container

可组合的修饰符（续）

- 可组合项可以拥有多个 修饰符 ，通过 将它们链式连接起来



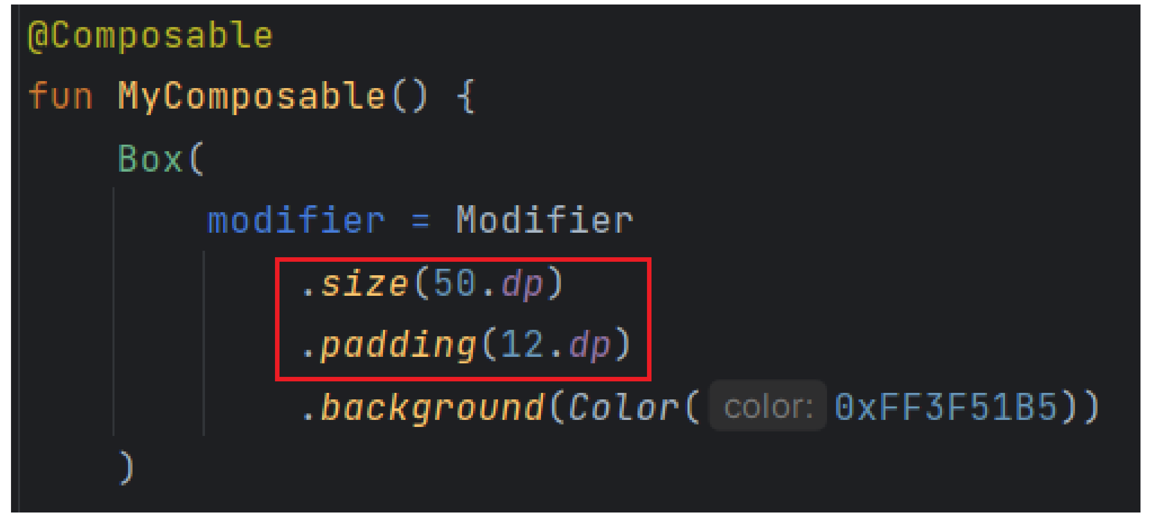
```
@Composable
fun MyComposable(){
    Box(
        modifier = Modifier
            .padding(all = 12.dp)
            .size(50.dp)
            .background(Color( color: 0xFF3F51B5))
    )
}
```



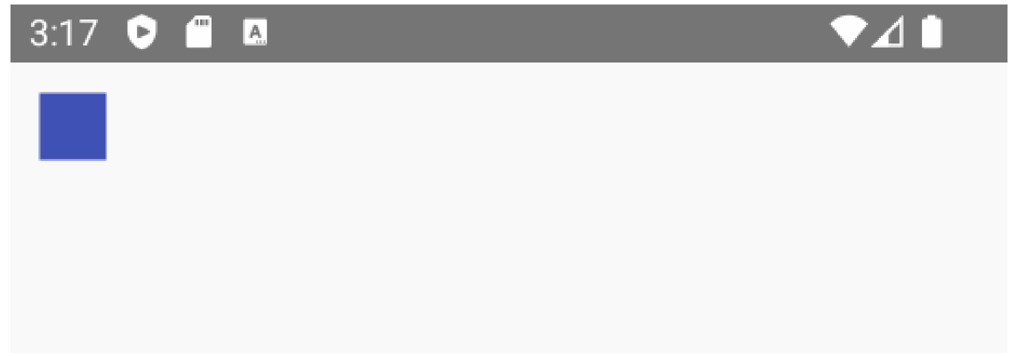
- Box 是一个简单的容器

Composable modifiers (cont.)

- Order is very important when working with composable modifiers



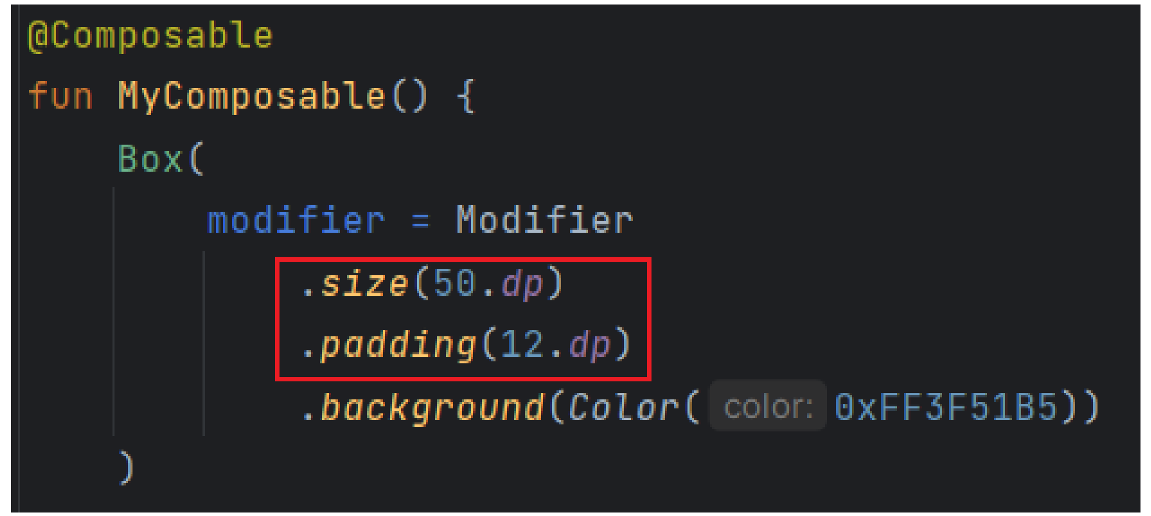
```
@Composable
fun MyComposable() {
    Box(
        modifier = Modifier
            .size(50.dp)
            .padding(12.dp)
            .background(Color( color: 0xFF3F51B5))
    )
```



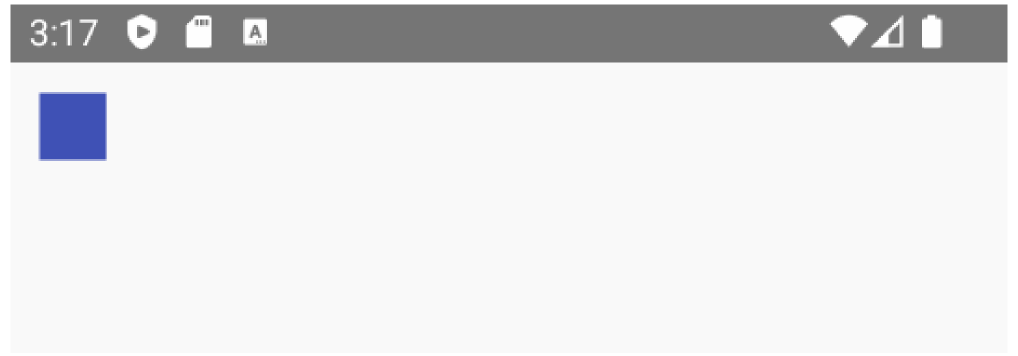
- We swapped size and padding, and you can see the result is different

可组合修饰符（续）

- 在使用可组合修饰符时，顺序非常重要



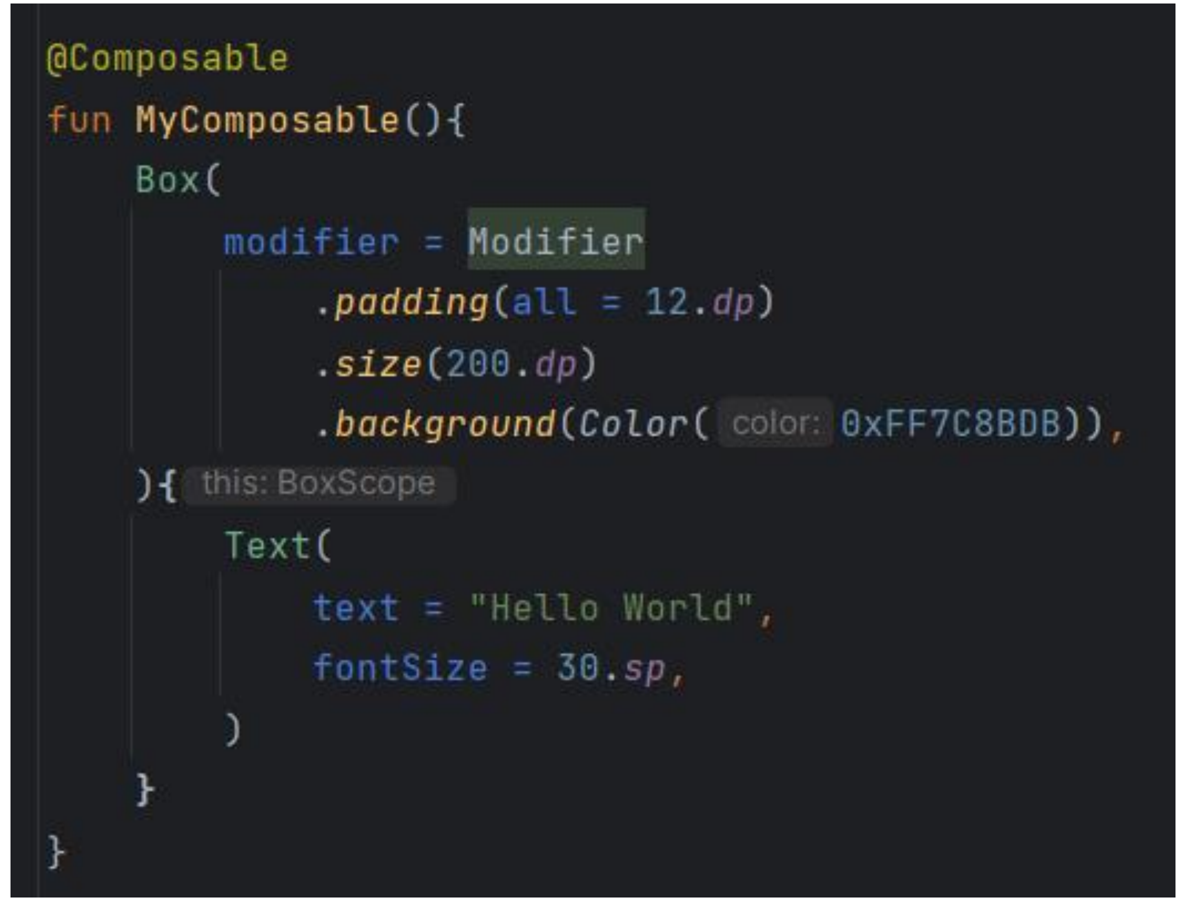
```
@Composable
fun MyComposable() {
    Box(
        modifier = Modifier
            .size(50.dp)
            .padding(12.dp)
            .background(Color( color: 0xFF3F51B5))
    )
```



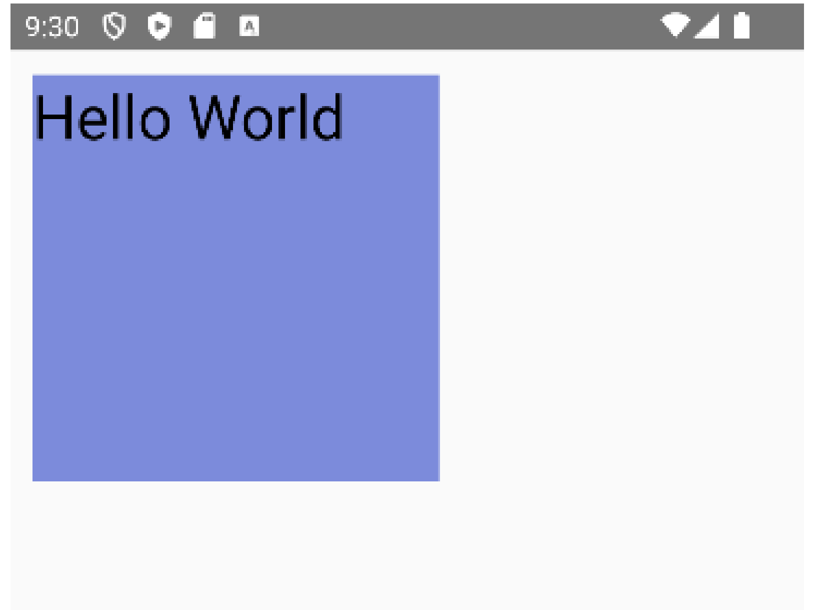
- 我们 交换了大小和内边距，你可以看到结果是不同的

Wrapping composables

- When building out and customizing your UI, composables are typically wrapped in other composables



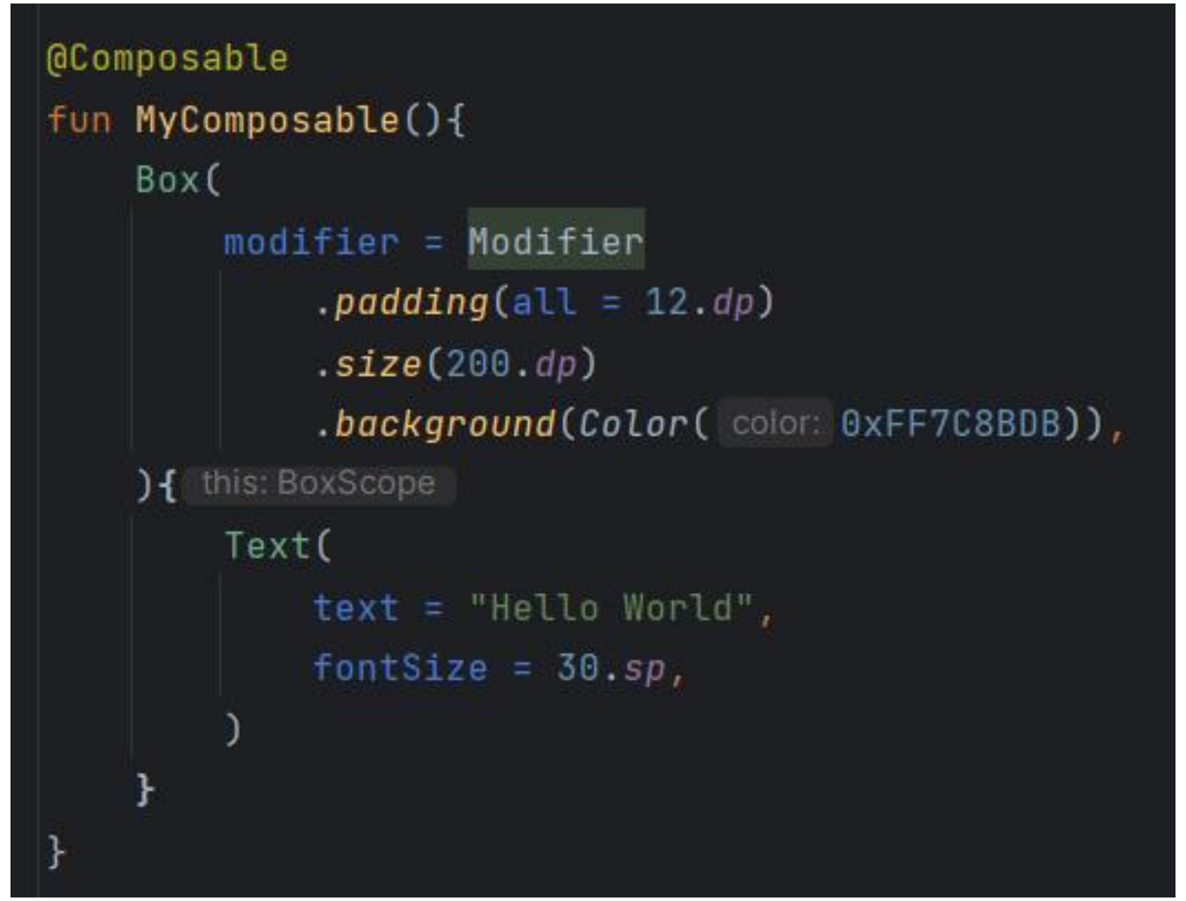
```
@Composable
fun MyComposable(){
    Box(
        modifier = Modifier
            .padding(all = 12.dp)
            .size(200.dp)
            .background(Color( color: 0xFF7C8BDB)),
    ){ this: BoxScope
        Text(
            text = "Hello World",
            fontSize = 30.sp,
        )
    }
}
```



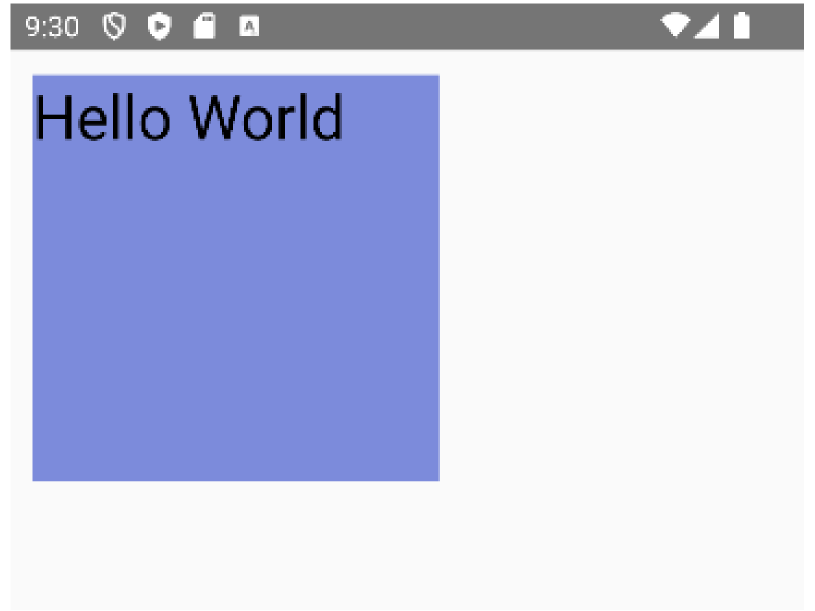
Hello World

包装可组合项

- 在构建和自定义用户界面时，可组合项通常是包装在其他可组合项中



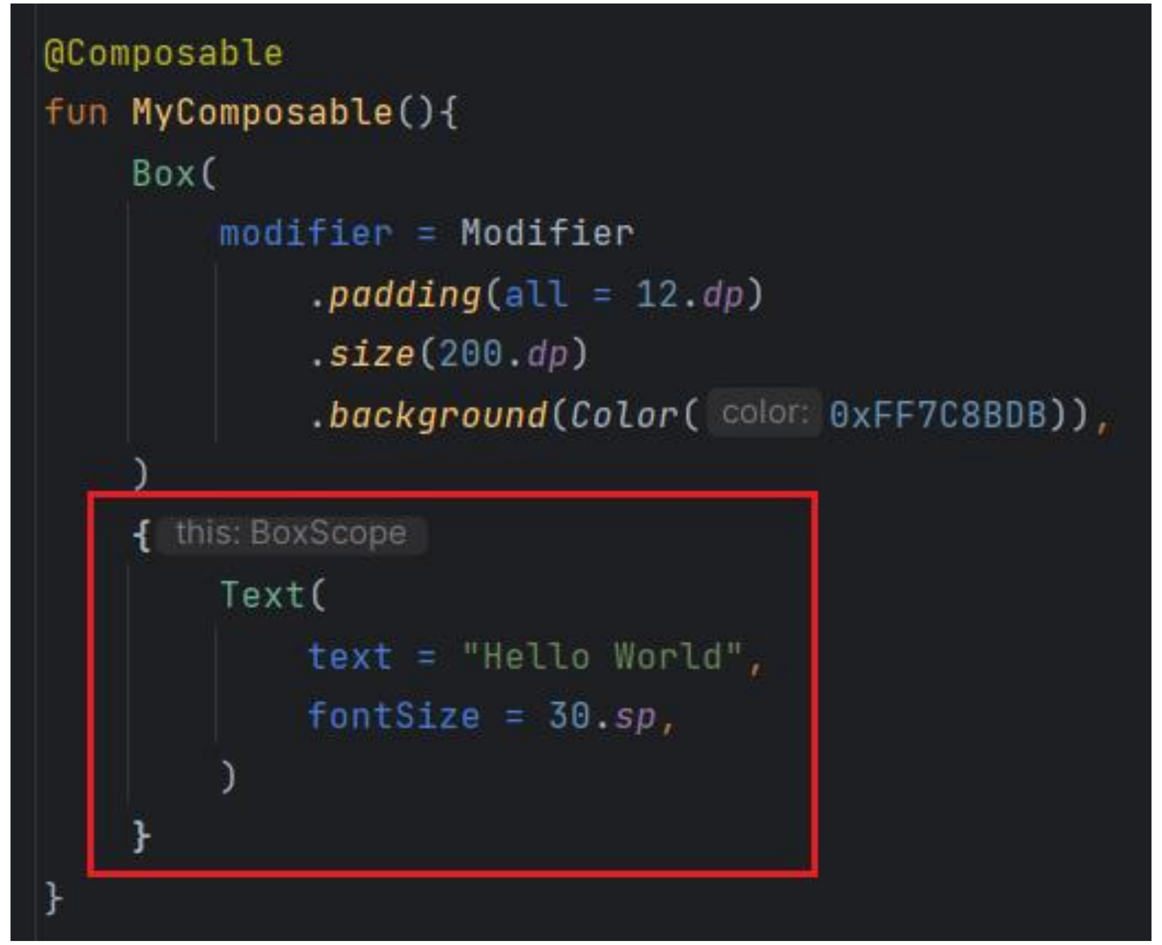
```
@Composable
fun MyComposable(){
    Box(
        modifier = Modifier
            .padding(all = 12.dp)
            .size(200.dp)
            .background(Color( color: 0xFF7C8BDB)),
    ){ this: BoxScope
        Text(
            text = "Hello World",
            fontSize = 30.sp,
        )
    }
}
```



Hello World

Wrapping composables (cont.)

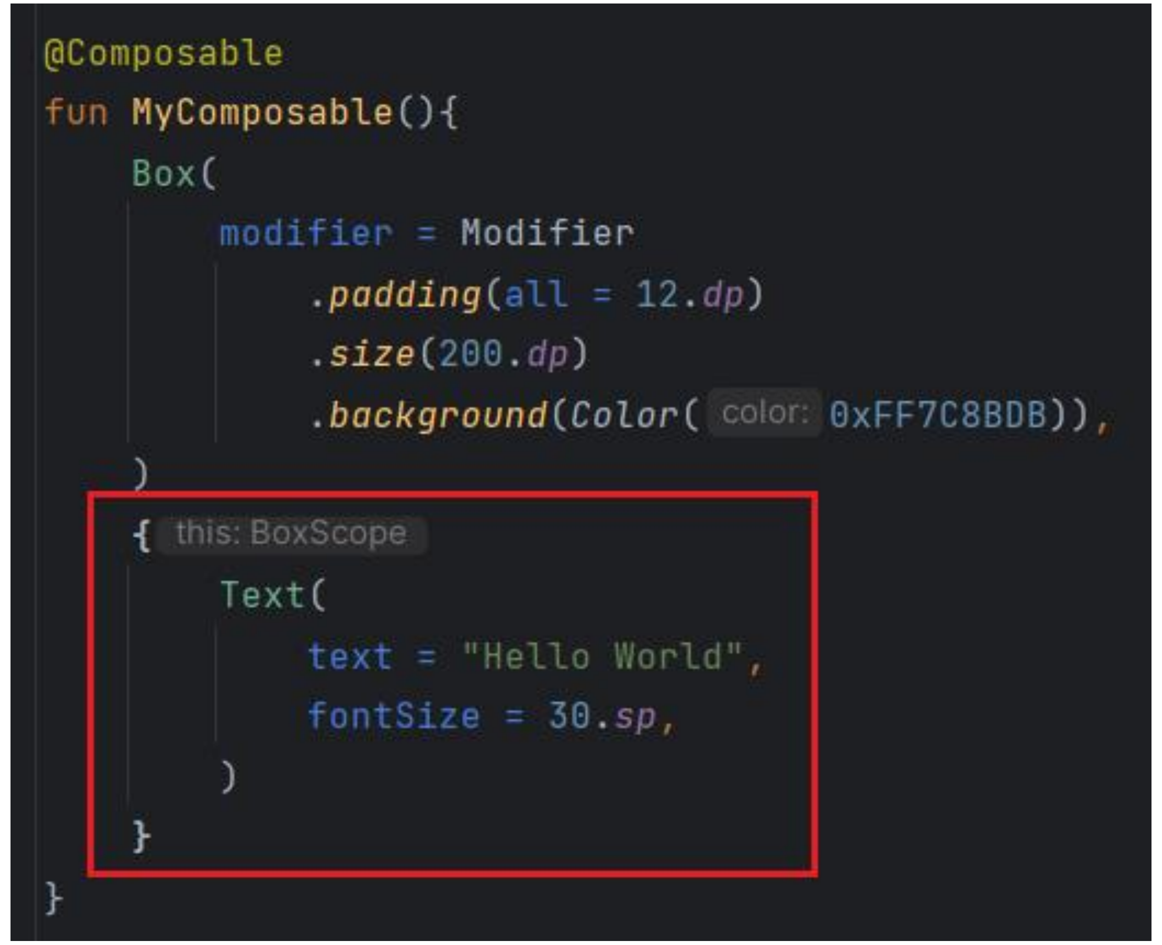
- When wrapping composables, the child composable goes inside the parent's scope



```
@Composable
fun MyComposable(){
    Box(
        modifier = Modifier
            .padding(all = 12.dp)
            .size(200.dp)
            .background(Color( color: 0xFF7C8BDB)),
    )
    { this: BoxScope
        Text(
            text = "Hello World",
            fontSize = 30.sp,
        )
    }
}
```

包装可组合项（续）

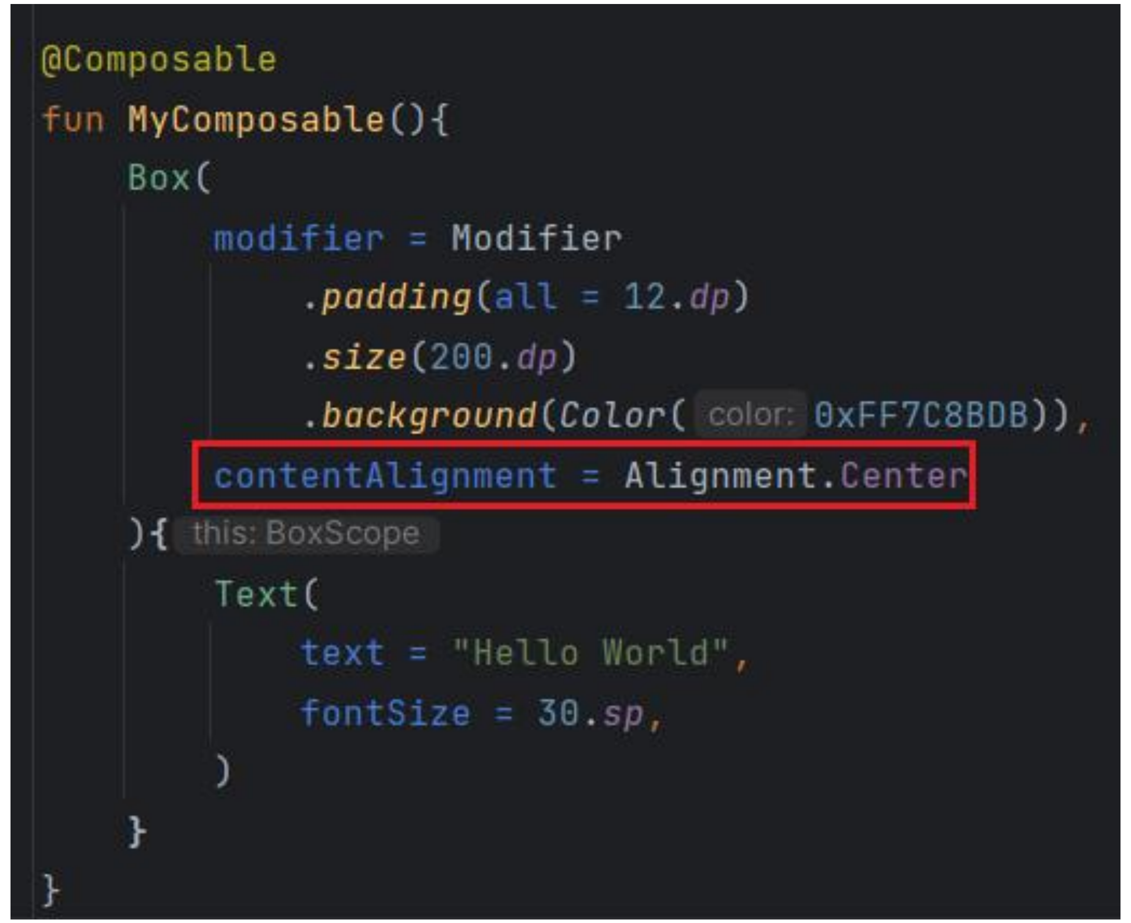
- 在包装可组合项时，子可组合项应放在父级的作用域



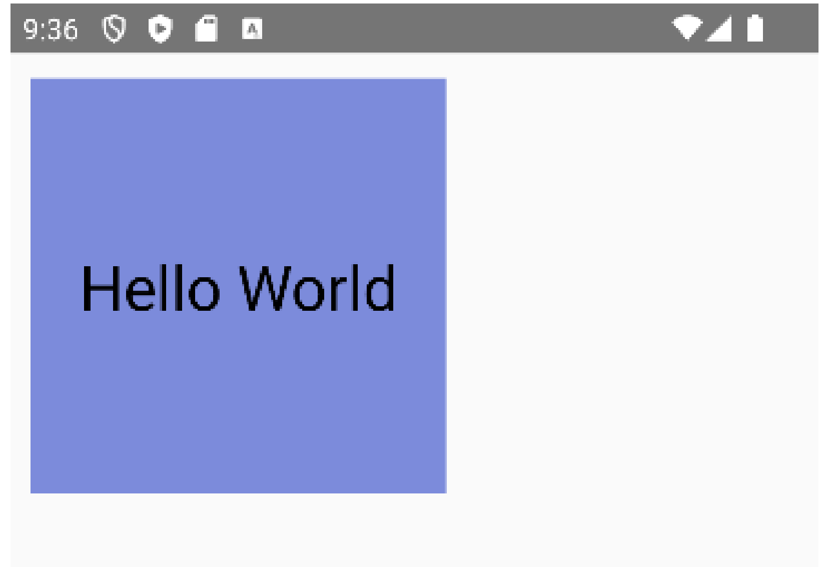
```
@Composable
fun MyComposable(){
    Box(
        modifier = Modifier
            .padding(all = 12.dp)
            .size(200.dp)
            .background(Color( color: 0xFF7C8BDB)),
    )
    { this: BoxScope
        Text(
            text = "Hello World",
            fontSize = 30.sp,
        )
    }
}
```

Wrapping composables (cont.)

- The parent composable enforces constraints on its children

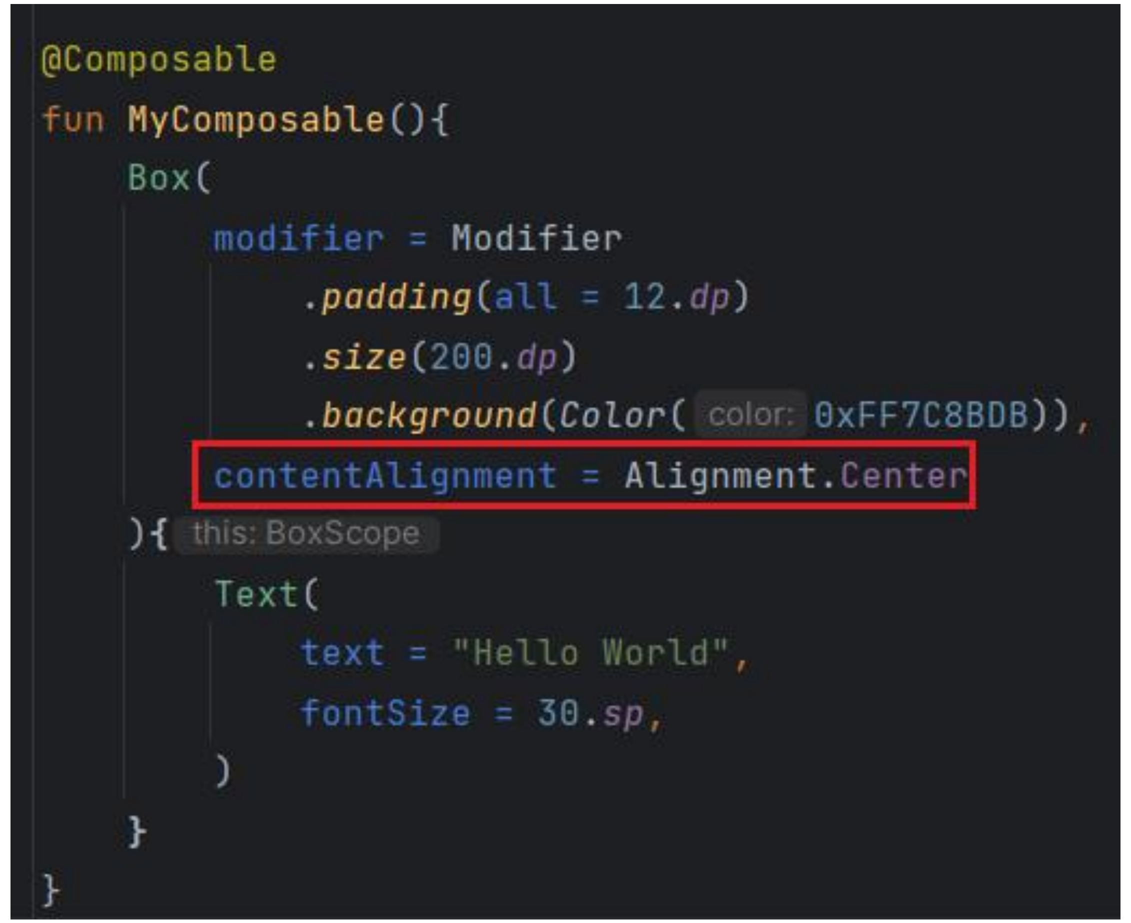


```
@Composable
fun MyComposable(){
    Box(
        modifier = Modifier
            .padding(all = 12.dp)
            .size(200.dp)
            .background(Color( color: 0xFF7C8BDB)),
        contentAlignment = Alignment.Center
    ){ this: BoxScope
        Text(
            text = "Hello World",
            fontSize = 30.sp,
        )
    }
}
```

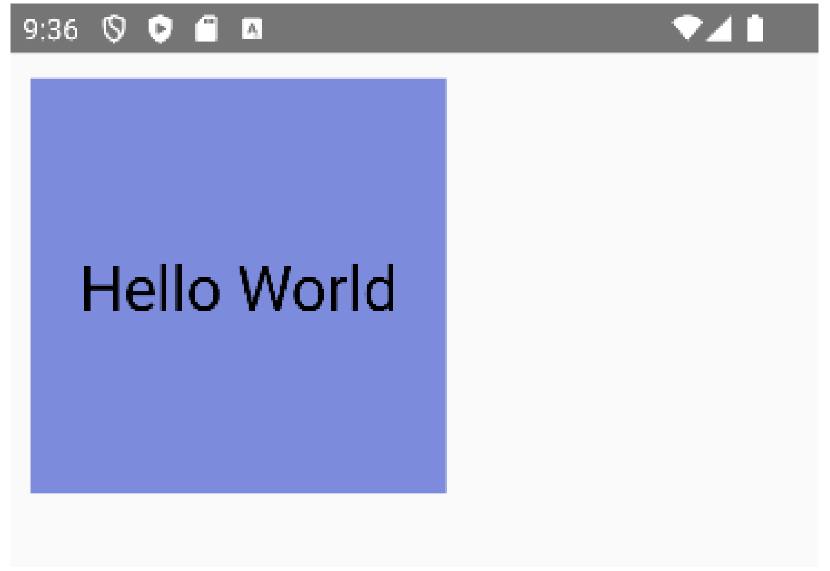


可组合项的包装（续）

- 父级可组合项对其子项施加约束

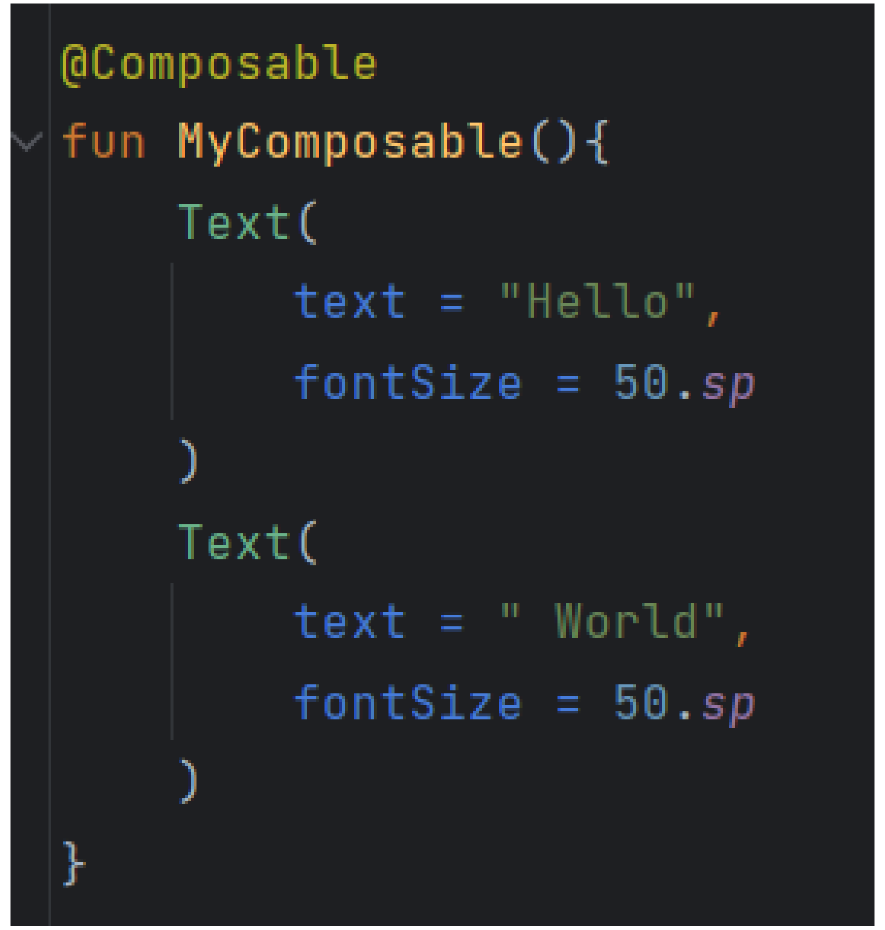


```
@Composable
fun MyComposable(){
    Box(
        modifier = Modifier
            .padding(all = 12.dp)
            .size(200.dp)
            .background(Color( color: 0xFF7C8BDB)),
        contentAlignment = Alignment.Center
    ){ this: BoxScope
        Text(
            text = "Hello World",
            fontSize = 30.sp,
        )
    }
}
```

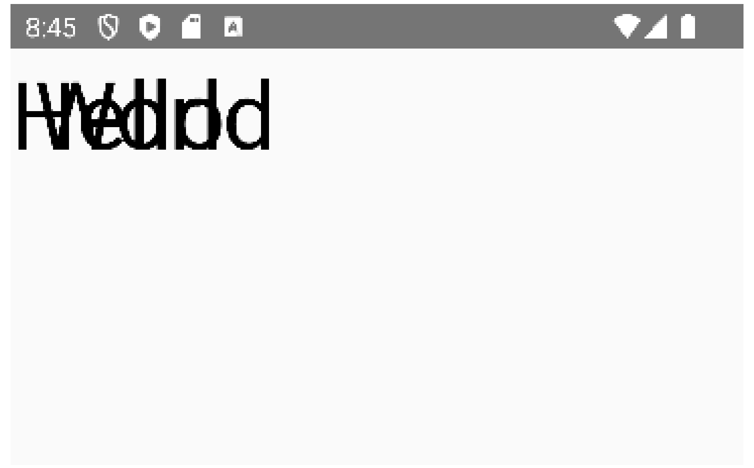


Composables (cont.)

- Let's say we wanted to display two different Text composables



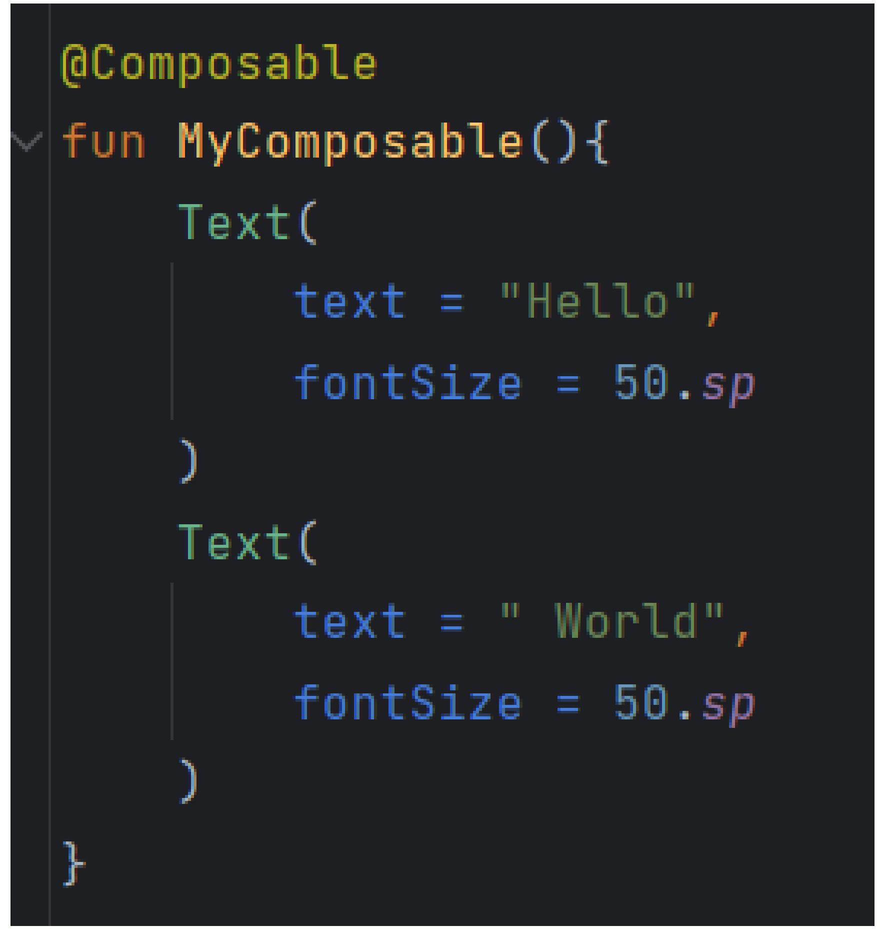
```
@Composable
fun MyComposable(){
    Text(
        text = "Hello",
        fontSize = 50.sp
    )
    Text(
        text = " World",
        fontSize = 50.sp
    )
}
```



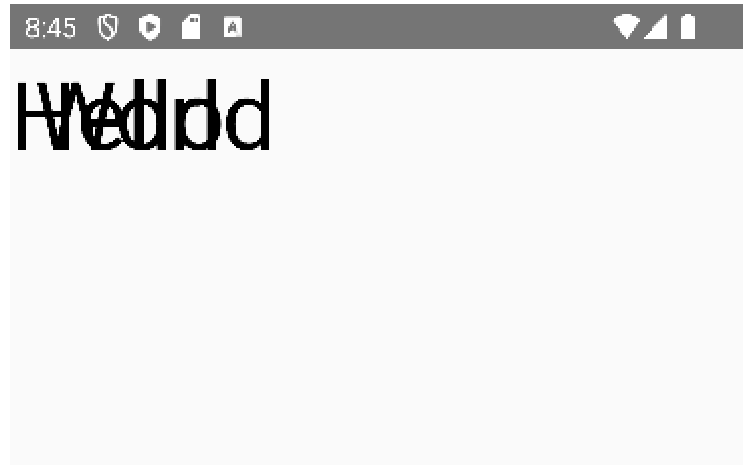
- Composables will just stack on top of each other if we don't provide a way to arrange them

可组合项（续）

- 假设我们想要显示两个不同的 Text 可组合项



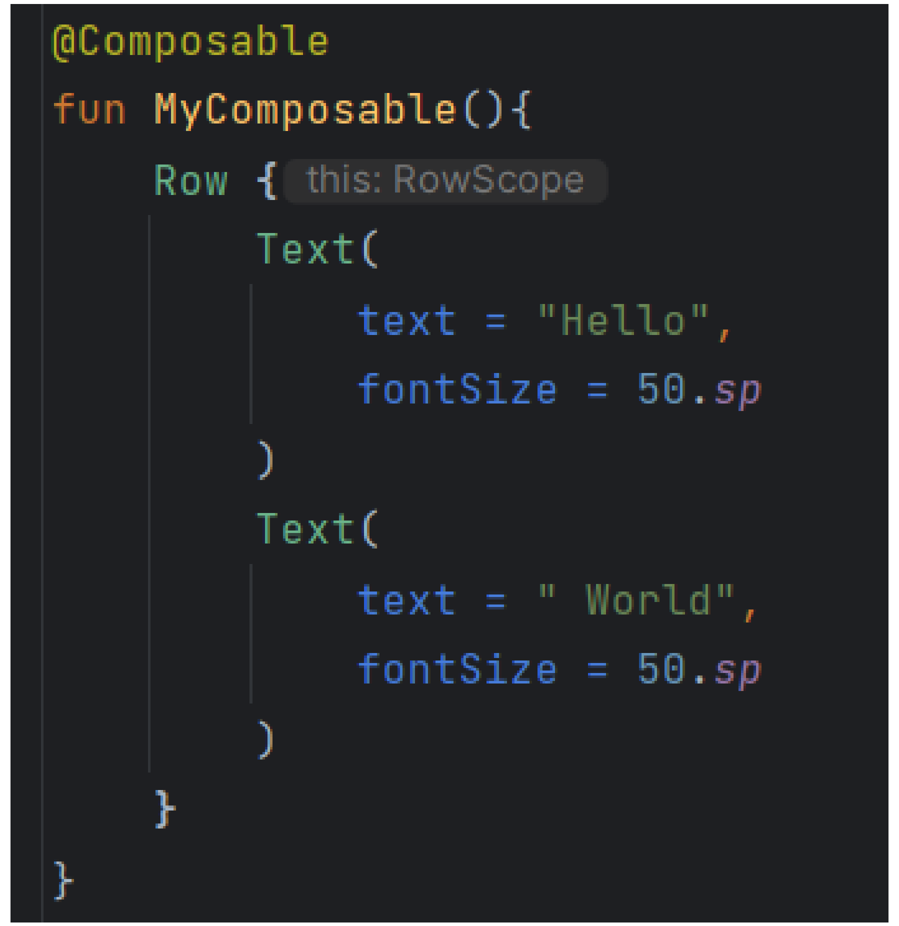
```
@Composable
fun MyComposable(){
    Text(
        text = "Hello",
        fontSize = 50.sp
    )
    Text(
        text = " World",
        fontSize = 50.sp
    )
}
```



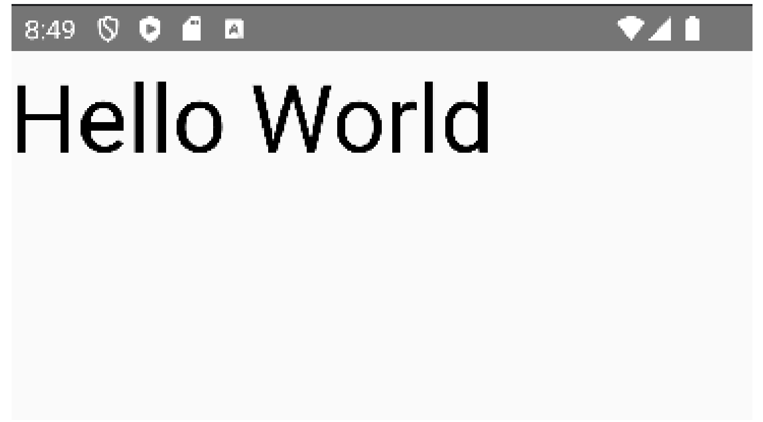
- 如果我们不提供布局方式，可组合项将简单地彼此堆叠

Row

- A *Row* allows us to arrange elements horizontally



```
@Composable
fun MyComposable(){
    Row { this: RowScope
        Text(
            text = "Hello",
            fontSize = 50.sp
        )
        Text(
            text = " World",
            fontSize = 50.sp
        )
    }
}
```

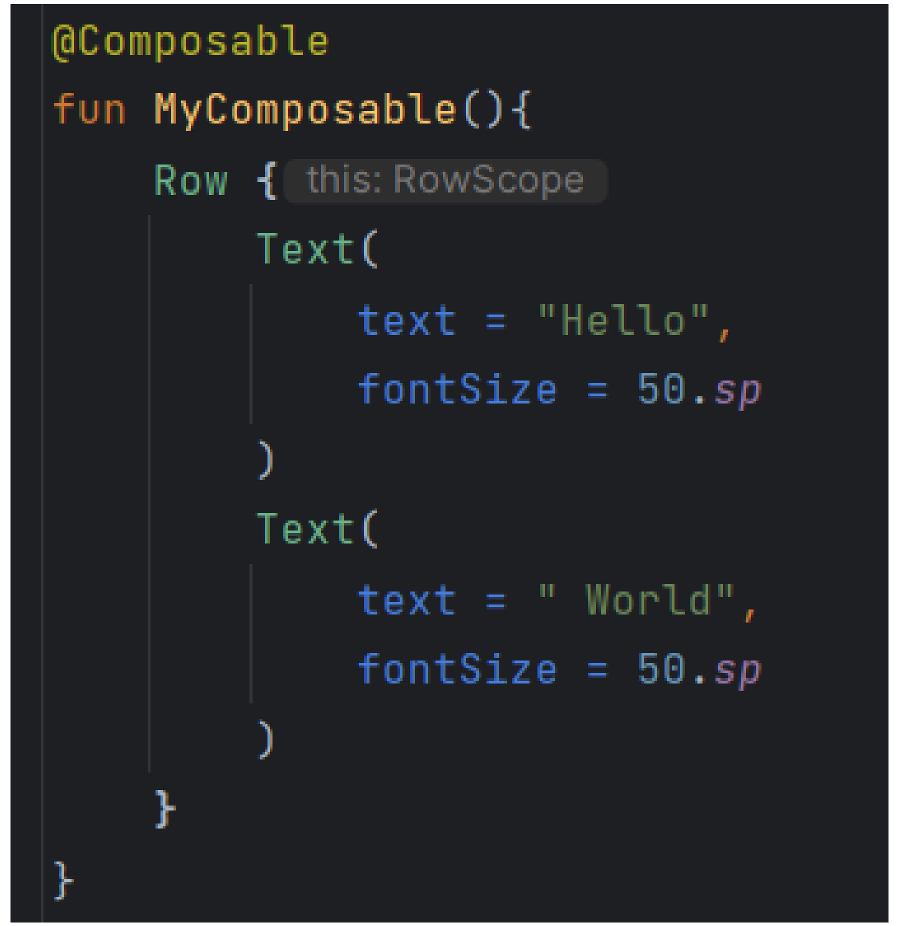


8:49

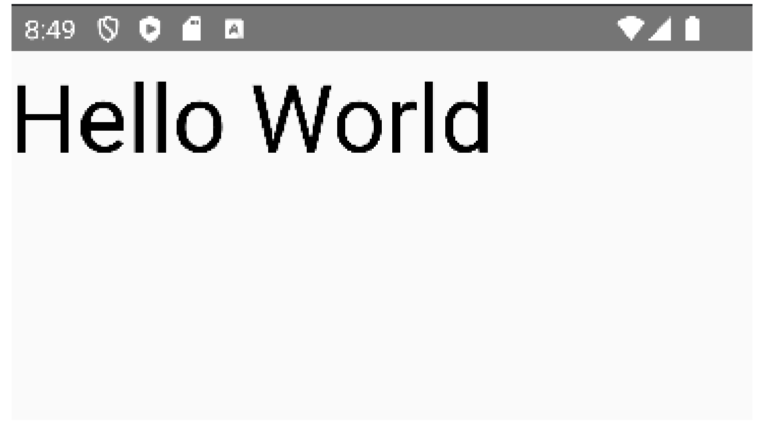
Hello World

Row

- 行 可以帮助我们水平排列元素



```
@Composable
fun MyComposable(){
    Row { this: RowScope
        Text(
            text = "Hello",
            fontSize = 50.sp
        )
        Text(
            text = " World",
            fontSize = 50.sp
        )
    }
}
```

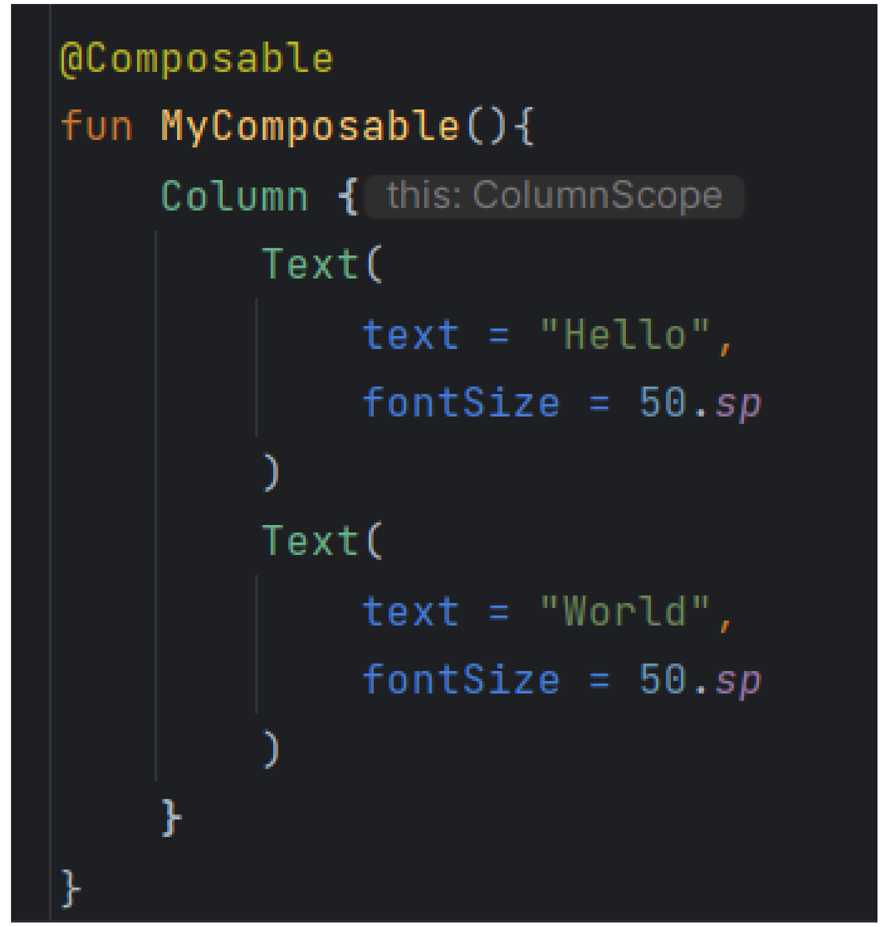


8:49

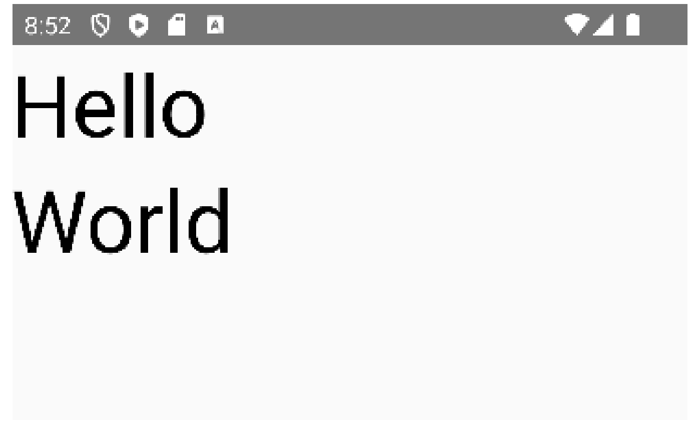
Hello World

Column

- A *Column* allows us to arrange elements vertically



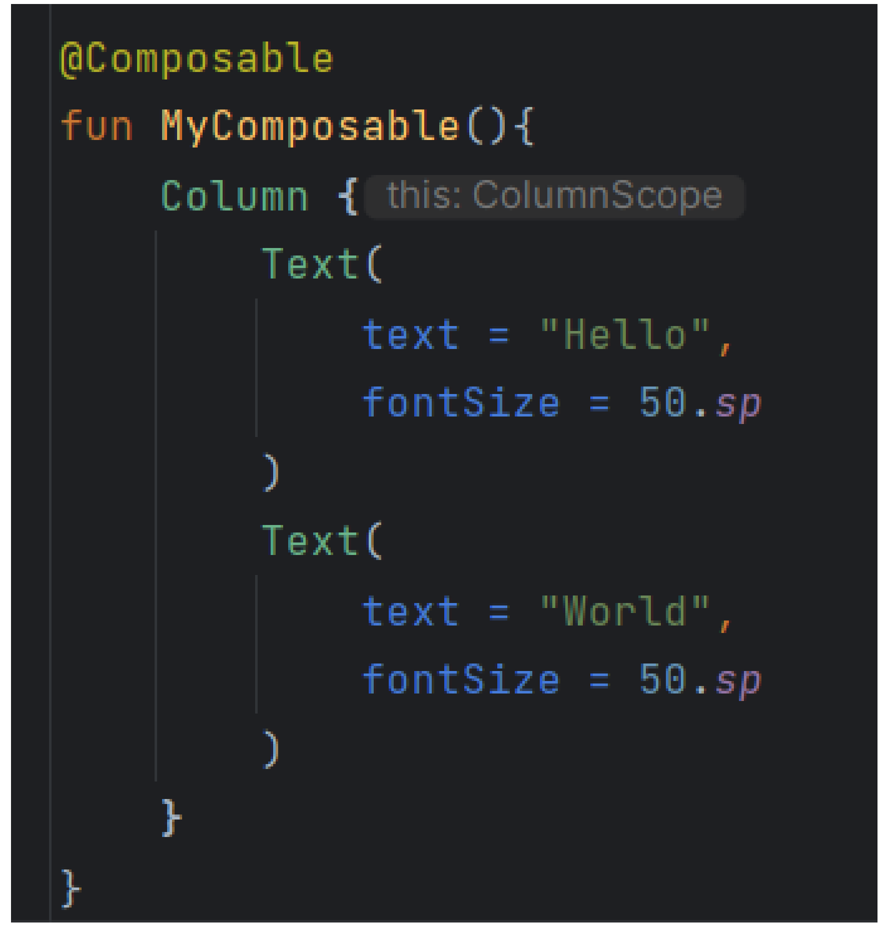
```
@Composable
fun MyComposable(){
    Column { this: ColumnScope
        Text(
            text = "Hello",
            fontSize = 50.sp
        )
        Text(
            text = "World",
            fontSize = 50.sp
        )
    }
}
```



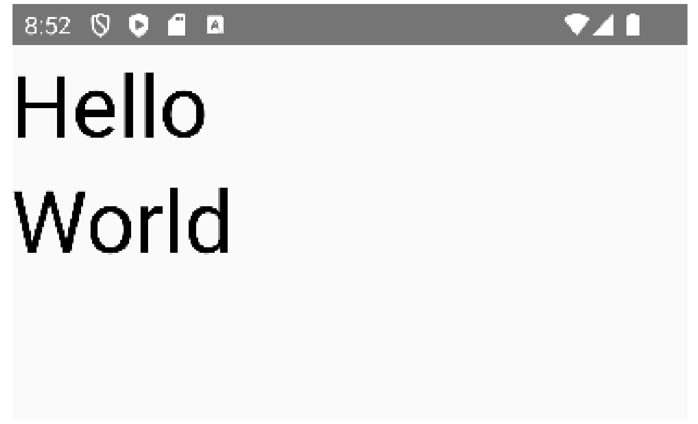
Hello
World

列

- 一个列使我们能够垂直排列元素



```
@Composable
fun MyComposable(){
    Column { this: ColumnScope
        Text(
            text = "Hello",
            fontSize = 50.sp
        )
        Text(
            text = "World",
            fontSize = 50.sp
        )
    }
}
```



Hello
World

Rows and Columns

- Two common properties of a Row and Column is *alignment* and *arrangment*

- *Arrangment* is done on the main axis
  - Horizontally for a Row
  - Vertically for a Column

- *Alignment* is done on the cross axis
  - Horizontally for a Column
  - Vertically for a Row

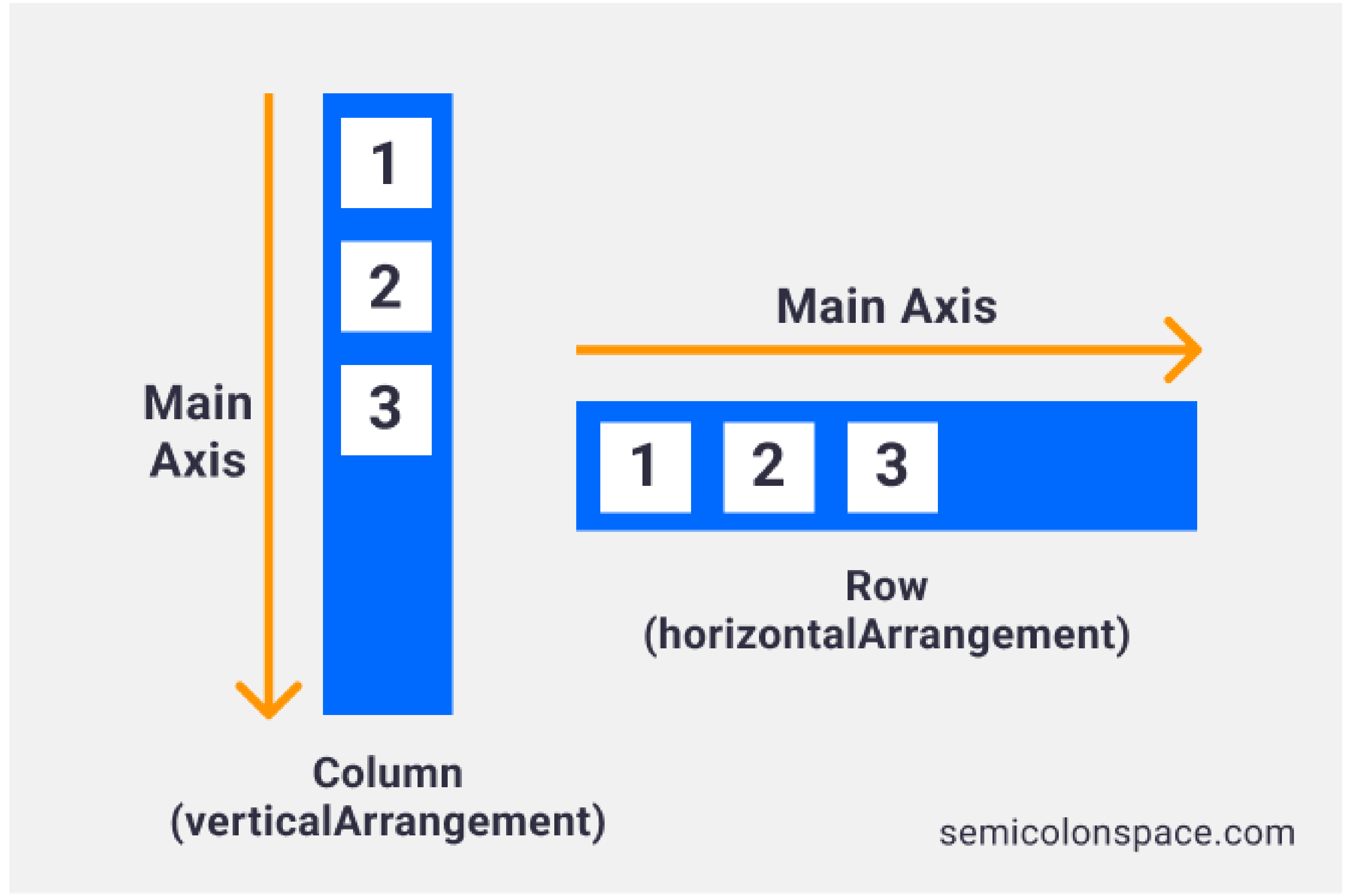
行与列

- 行和列的两个常见属性是对齐和排列

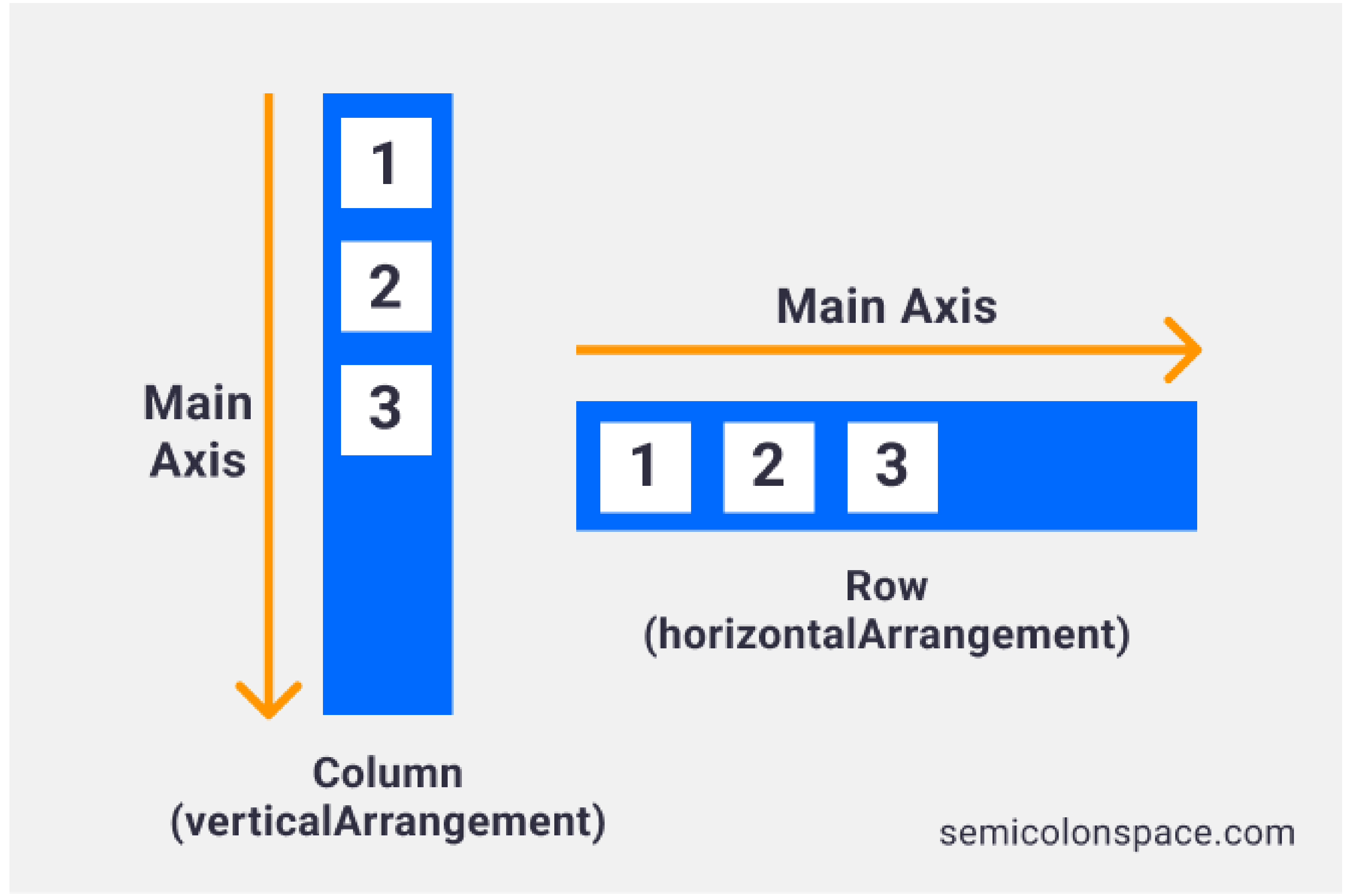
- 排列沿主轴进行
  - 对于行，为水平方向
  - 对于列，为垂直方向

- 对齐沿交叉轴进行
  - 列的水平方向
  - 行的垂直方向

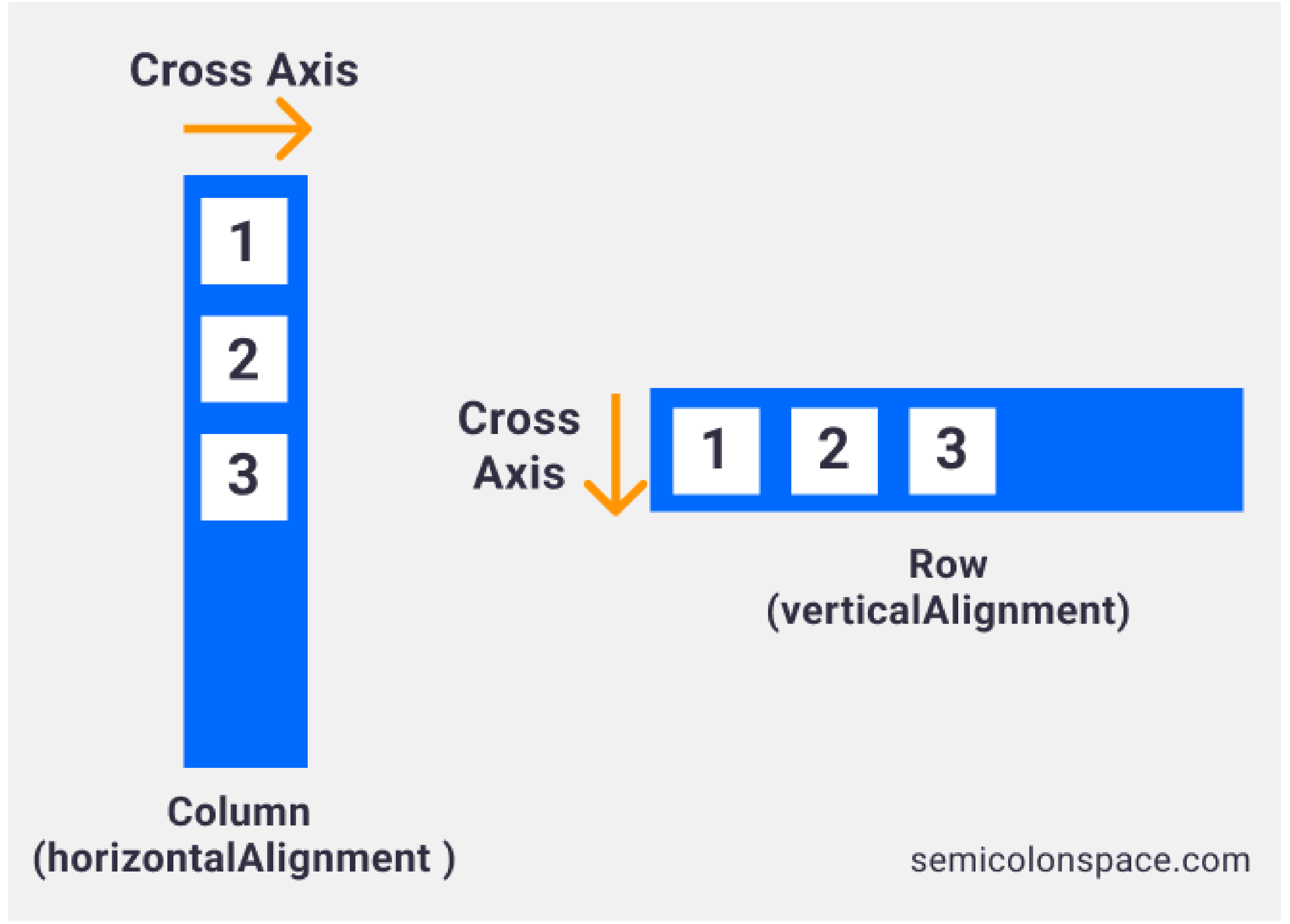
Rows and Columns (cont.)



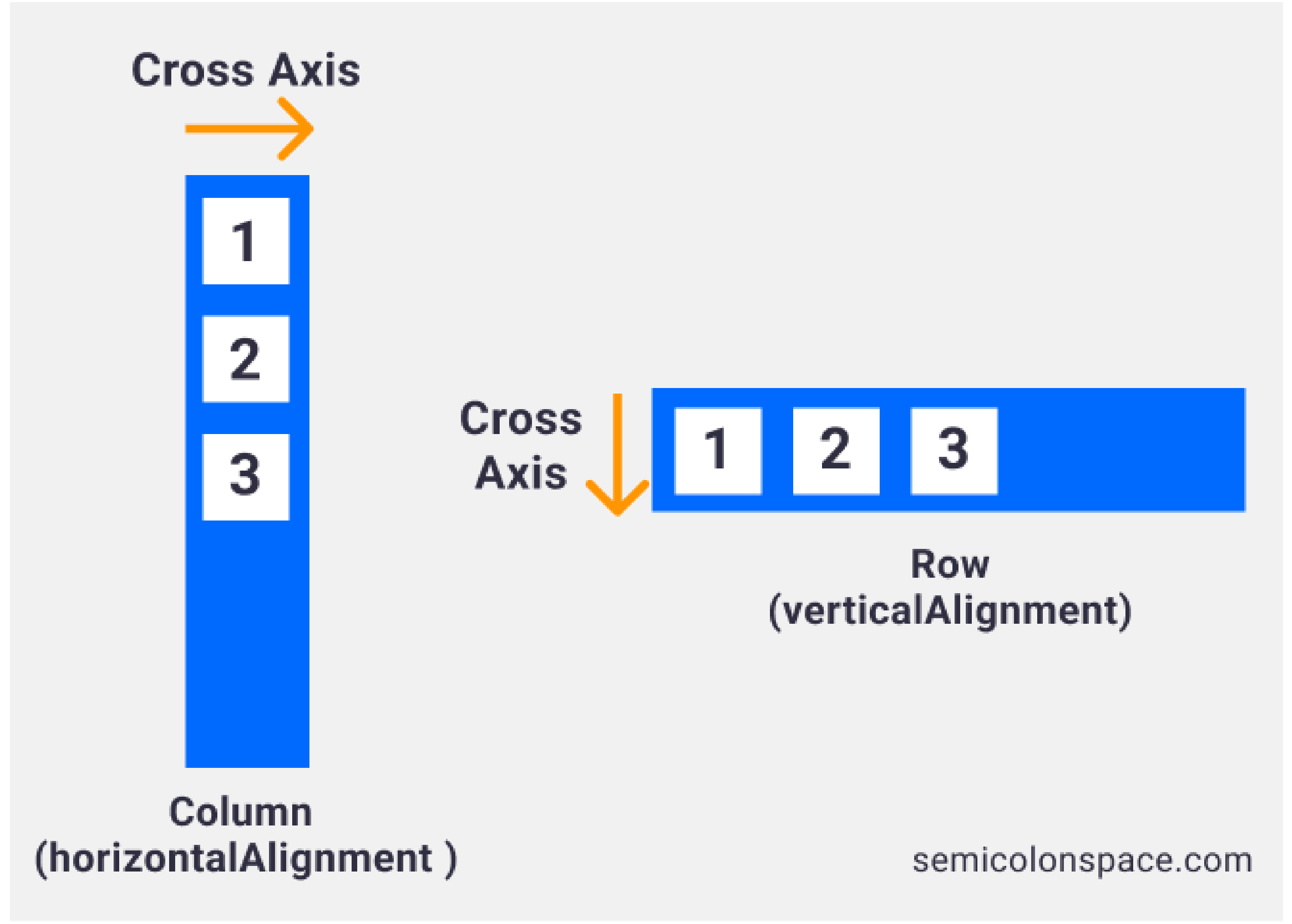
行和列（续）



Rows and Columns (cont.)

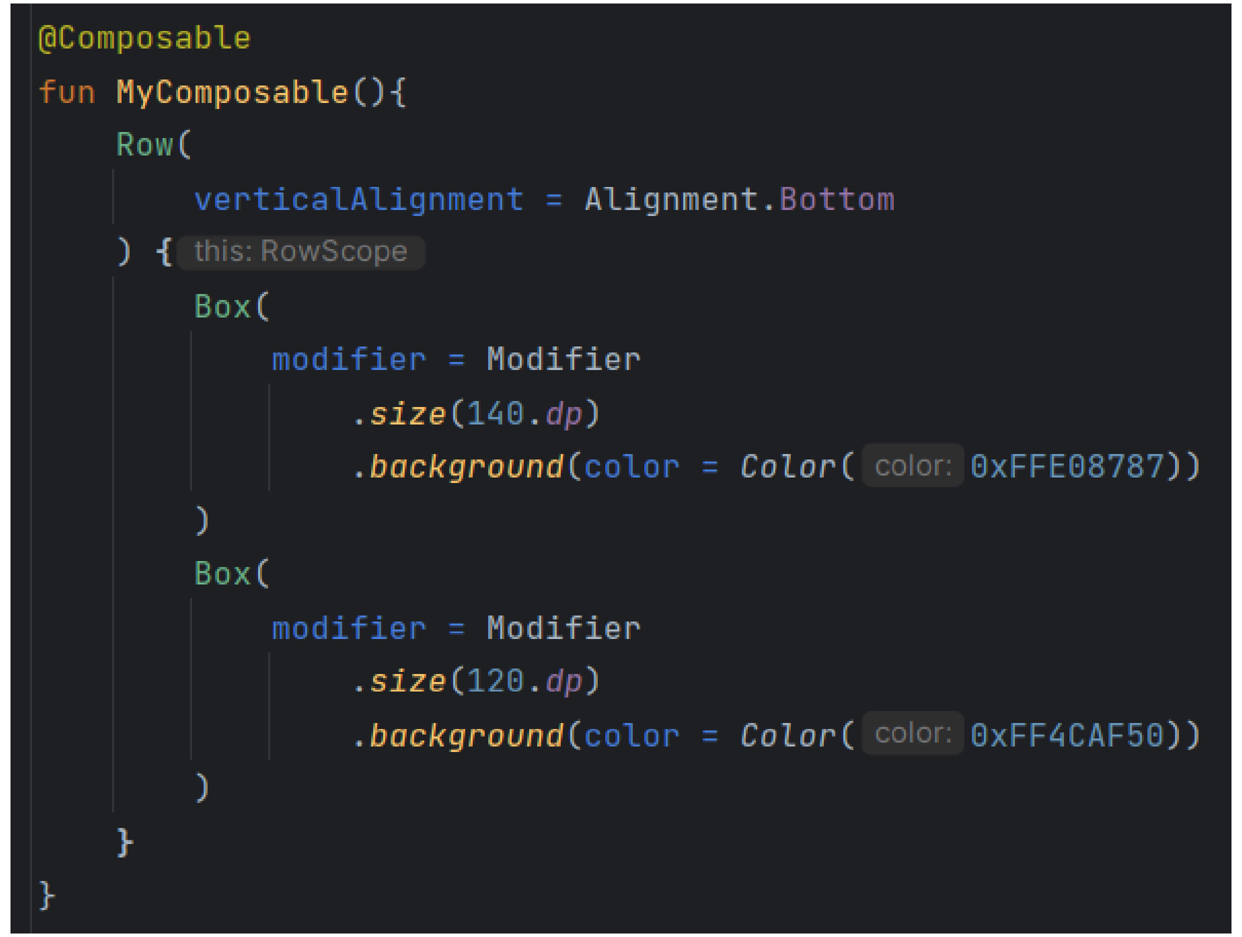


行和列（续）

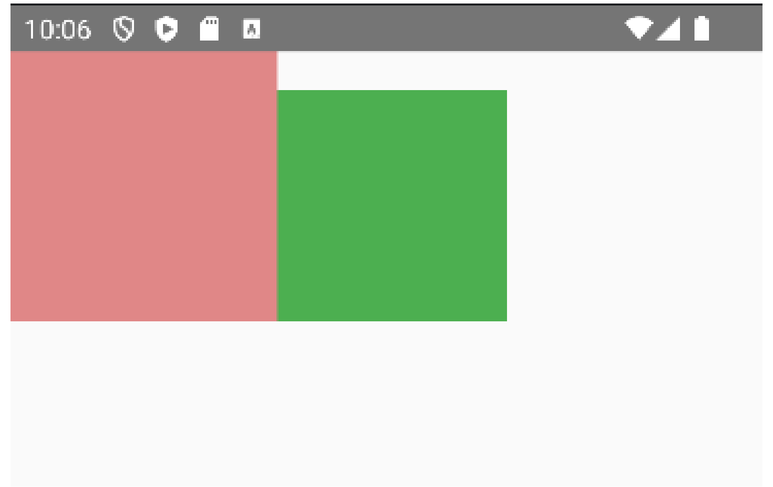


Rows and Columns (cont.)

- Here we align two boxes in a row to the bottom

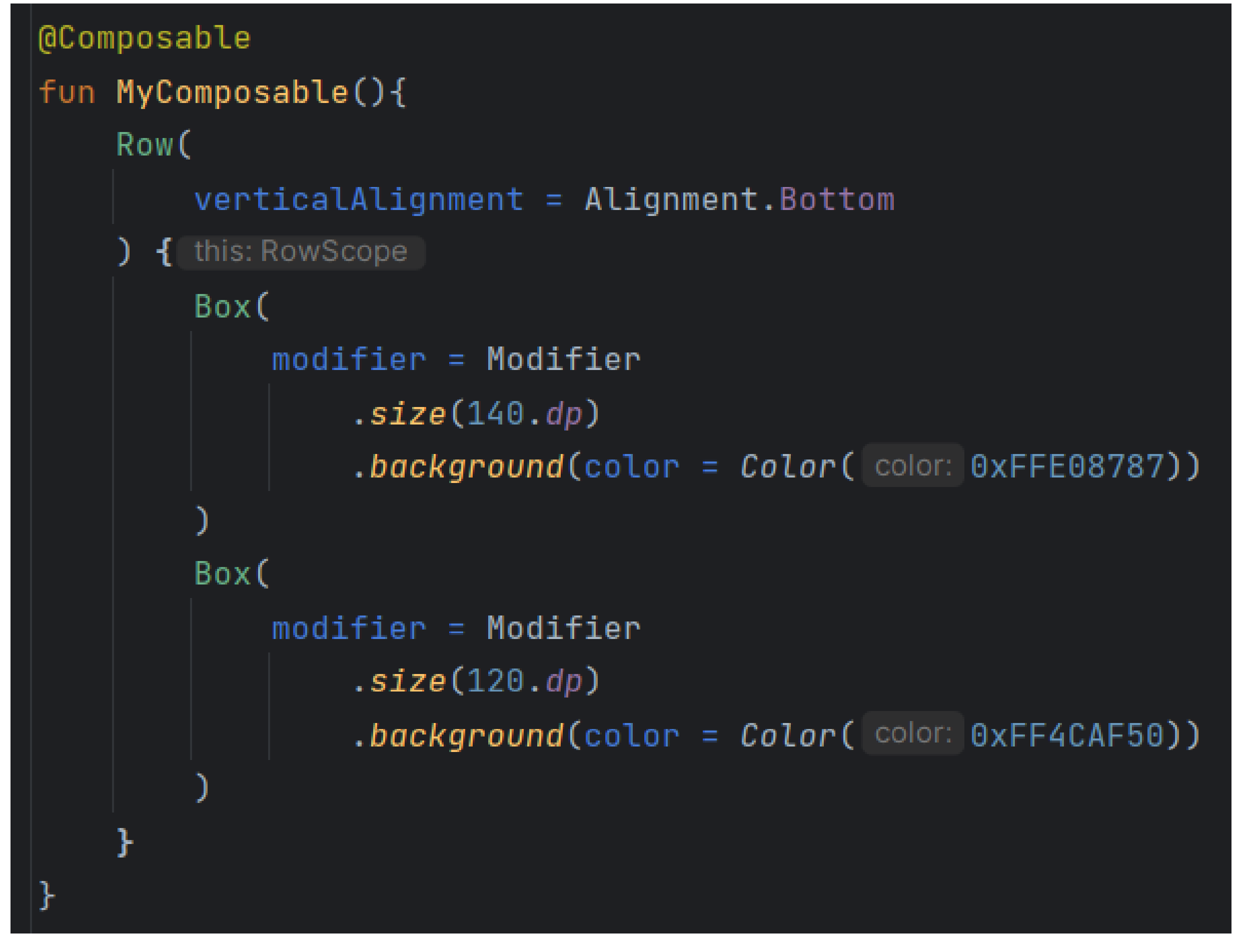


```
@Composable
fun MyComposable(){
    Row(
        verticalAlignment = Alignment.Bottom
    ) { this: RowScope
        Box(
            modifier = Modifier
                .size(140.dp)
                .background(color = Color( color: 0xFFE08787))
        )
        Box(
            modifier = Modifier
                .size(120.dp)
                .background(color = Color( color: 0xFF4CAF50))
        )
    }
}
```

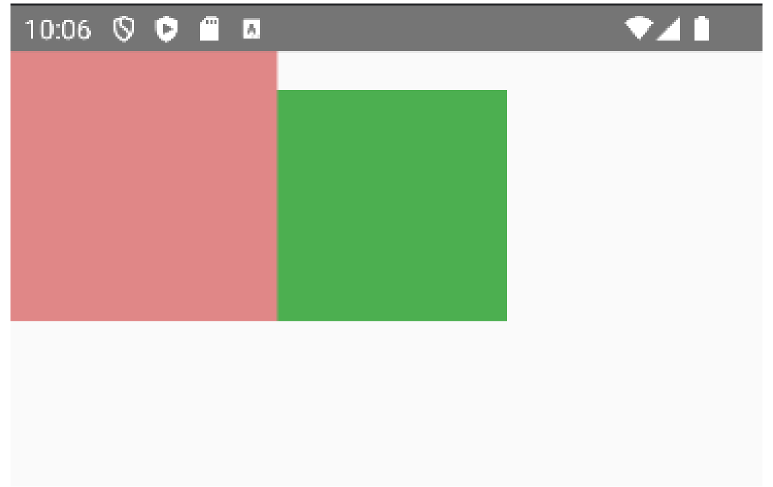


行和列（续）

- 这里我们将两个框在一行中与底部对齐

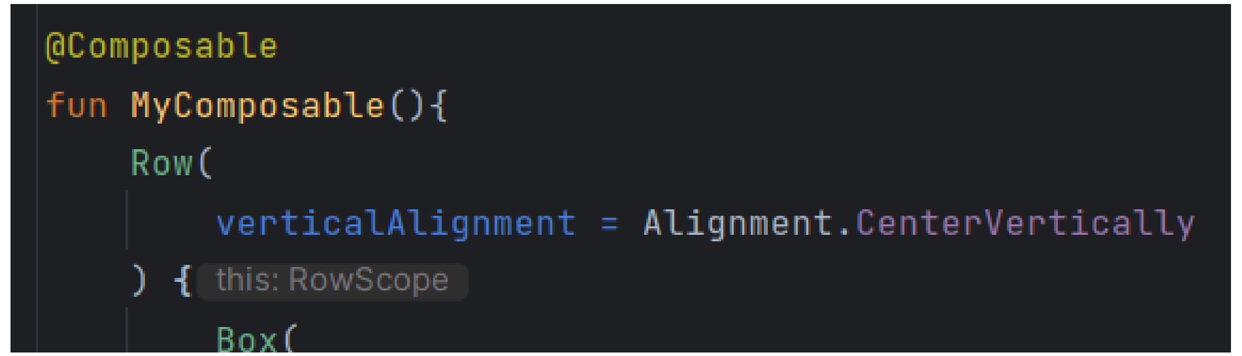


```
@Composable
fun MyComposable(){
    Row(
        verticalAlignment = Alignment.Bottom
    ) { this: RowScope
        Box(
            modifier = Modifier
                .size(140.dp)
                .background(color = Color( color: 0xFFE08787))
        )
        Box(
            modifier = Modifier
                .size(120.dp)
                .background(color = Color( color: 0xFF4CAF50))
        )
    }
}
```

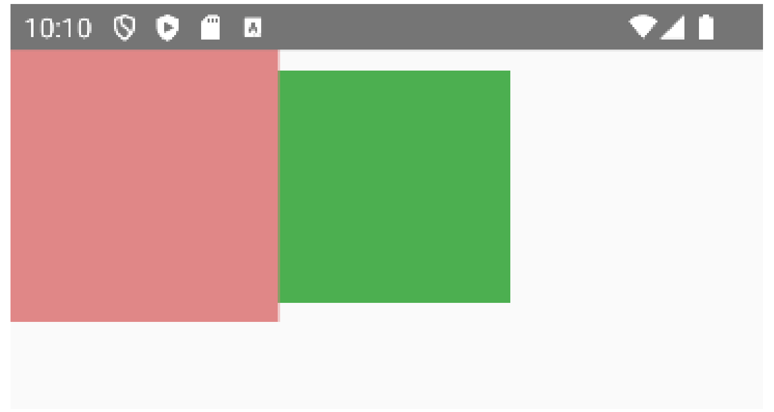


Rows and Columns (cont.)

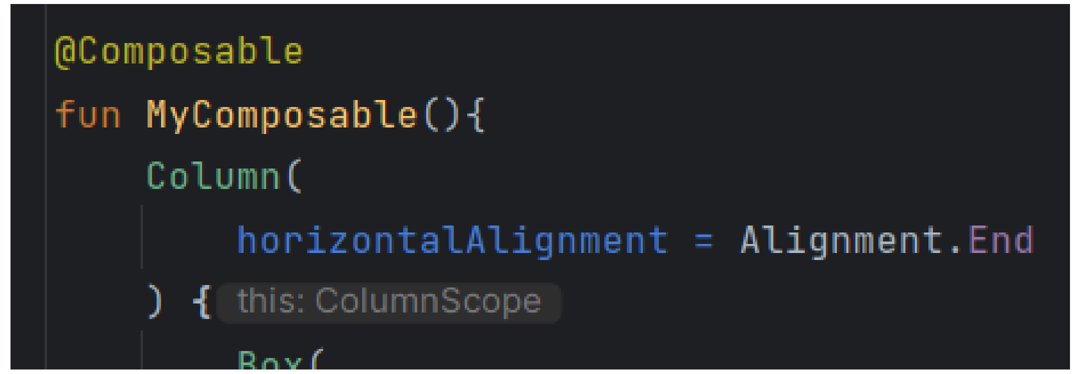
- Here we align the boxes in a row to the center



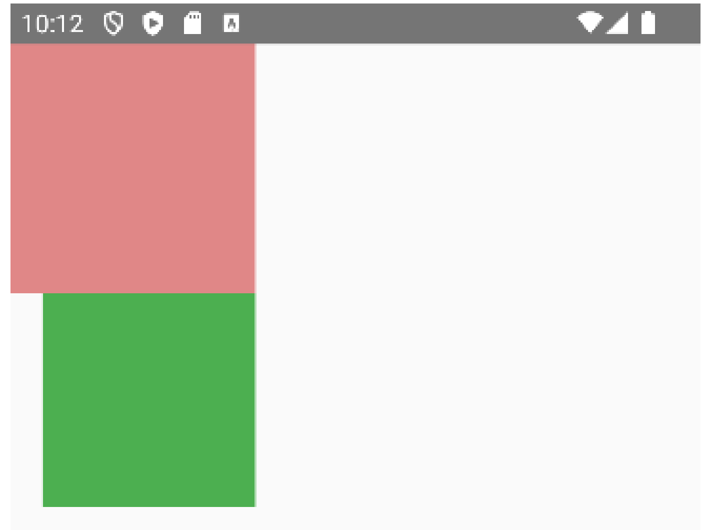
```
@Composable
fun MyComposable(){
    Row(
        verticalAlignment = Alignment.CenterVertically
    ) { this: RowScope
        Box(
```



- Here we align the boxes in a column to the end

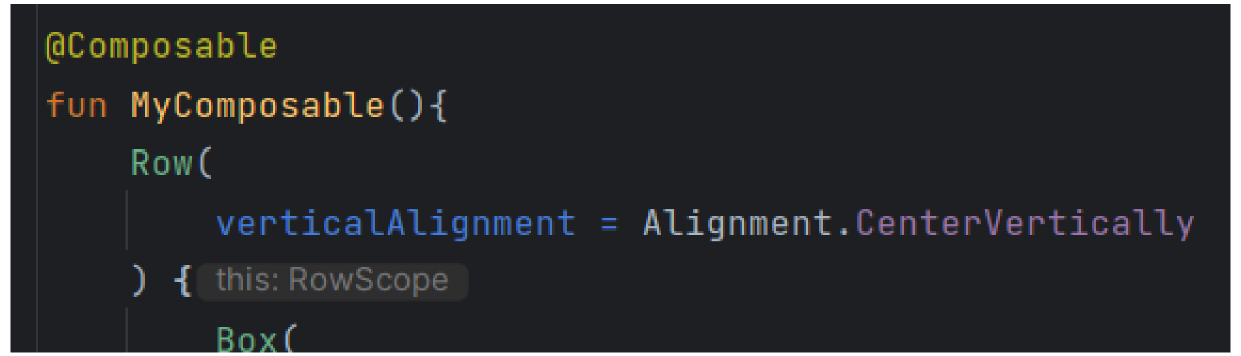


```
@Composable
fun MyComposable(){
    Column(
        horizontalAlignment = Alignment.End
    ) { this: ColumnScope
        Box(
```

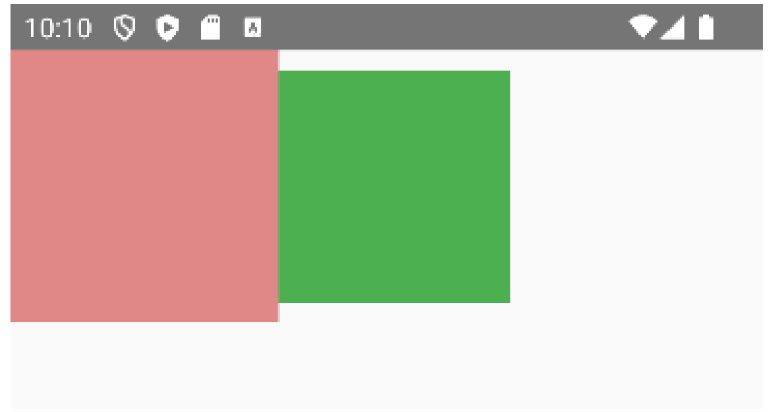


行和列（续）

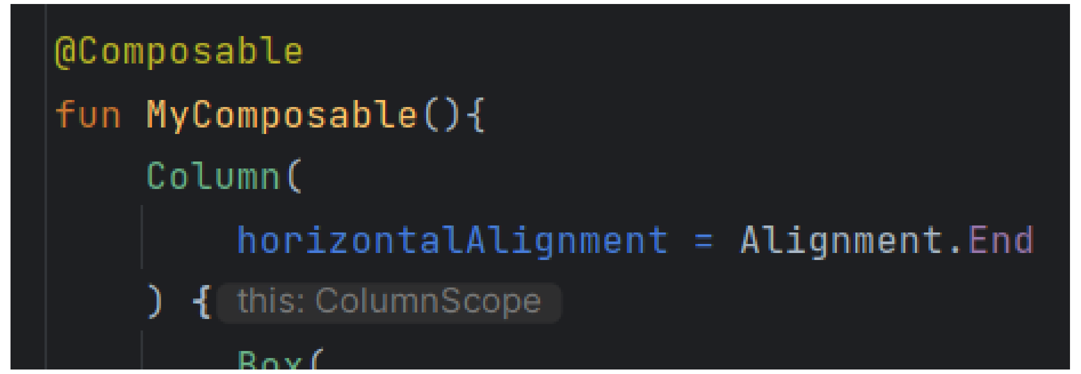
- 这里我们将行中的盒子居中对齐



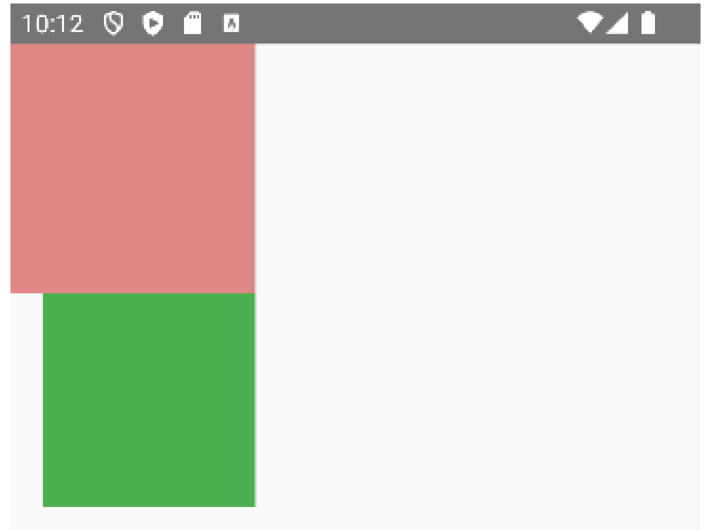
```
@Composable
fun MyComposable(){
    Row(
        verticalAlignment = Alignment.CenterVertically
    ) { this: RowScope
        Box(
```



- 这里我们将列中的盒子向末端对齐

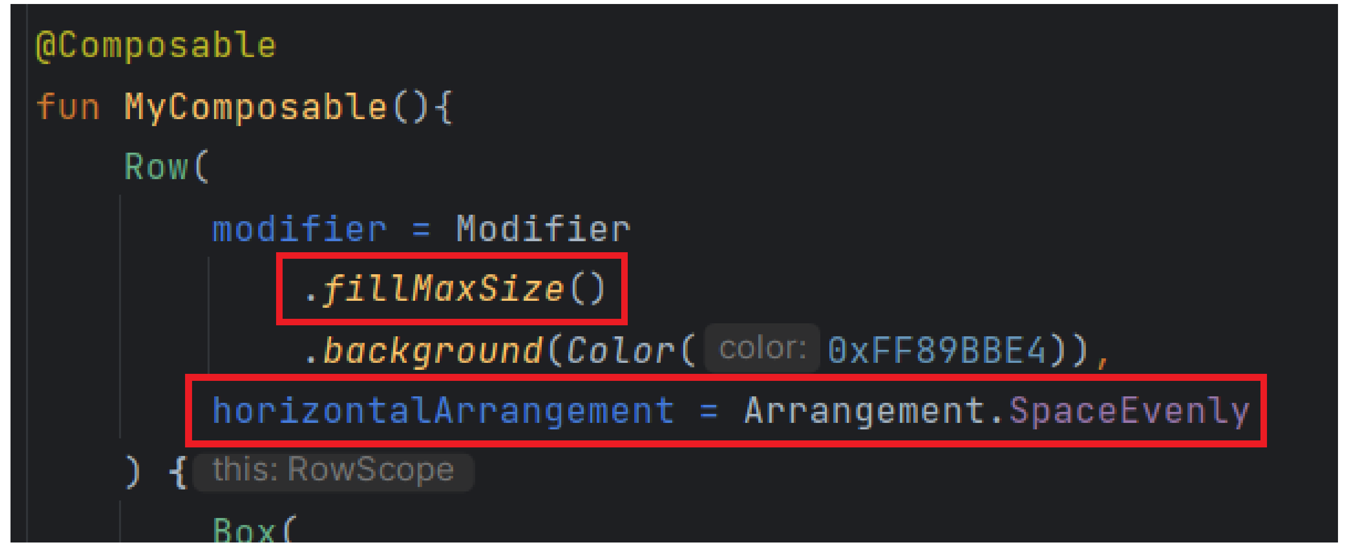


```
@Composable
fun MyComposable(){
    Column(
        horizontalAlignment = Alignment.End
    ) { this: ColumnScope
        Box(
```

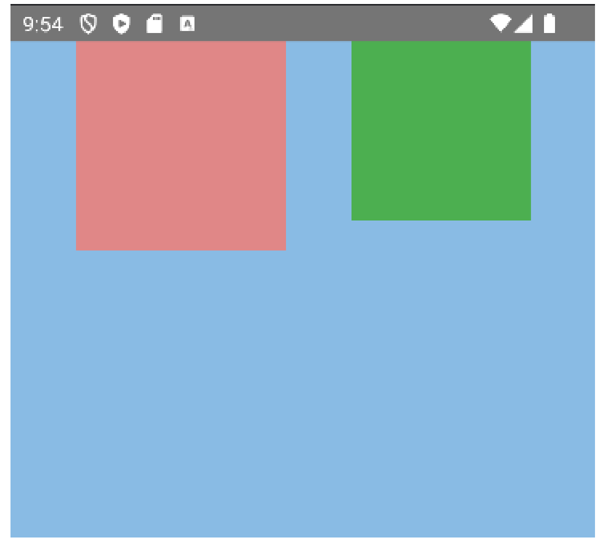


Rows and Columns (cont.)

- When working with *arrangment*, the Row or Column needs to fills it's parents space



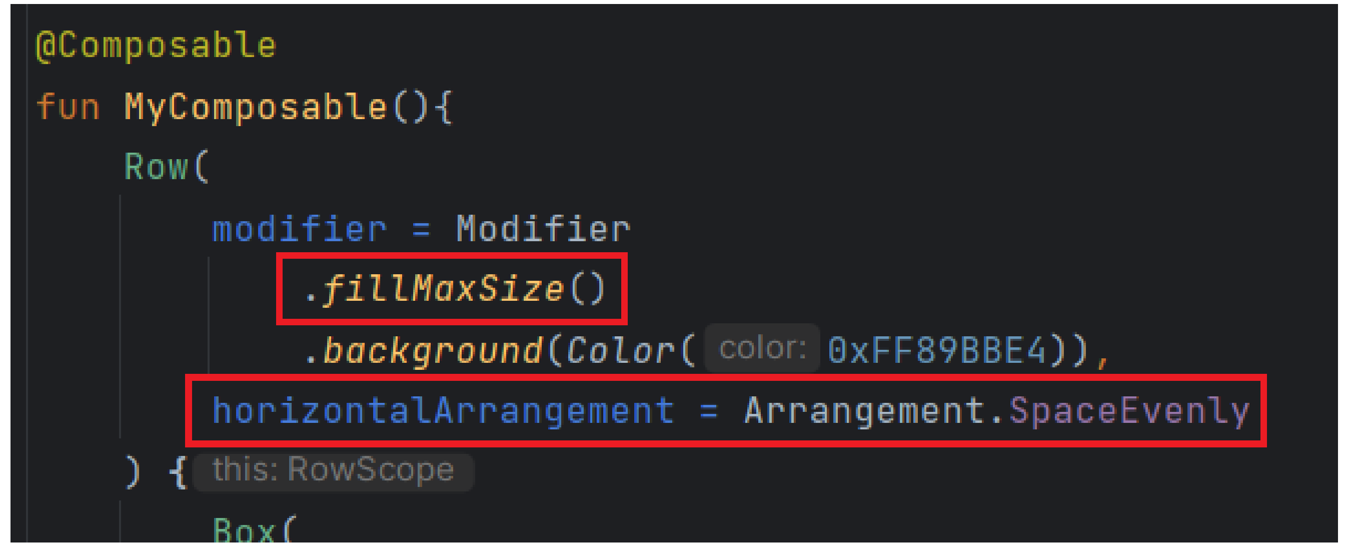
```
@Composable
fun MyComposable(){
    Row(
        modifier = Modifier
            .fillMaxSize()
            .background(Color( color: 0xFF89BBE4)),
        horizontalArrangement = Arrangement.SpaceEvenly
    ) { this: RowScope
        Box(
```



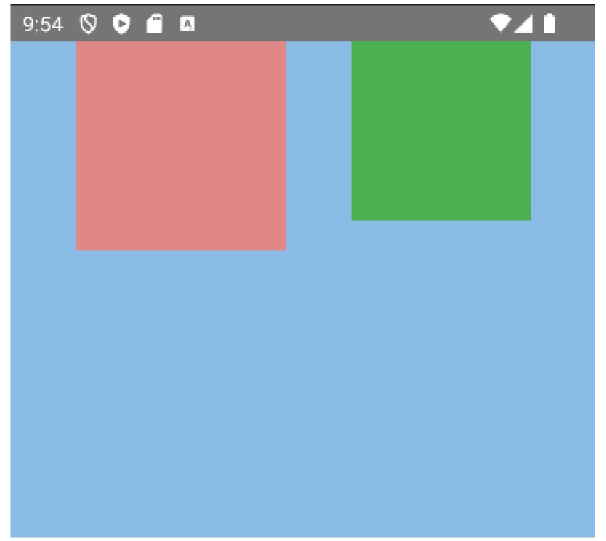
- We can fillMaxSize(), fillMaxWidth(), or fillMaxHeight()

行与列（续）

- 在使用 *arrangment* 时，行或列需要填充其父级空间



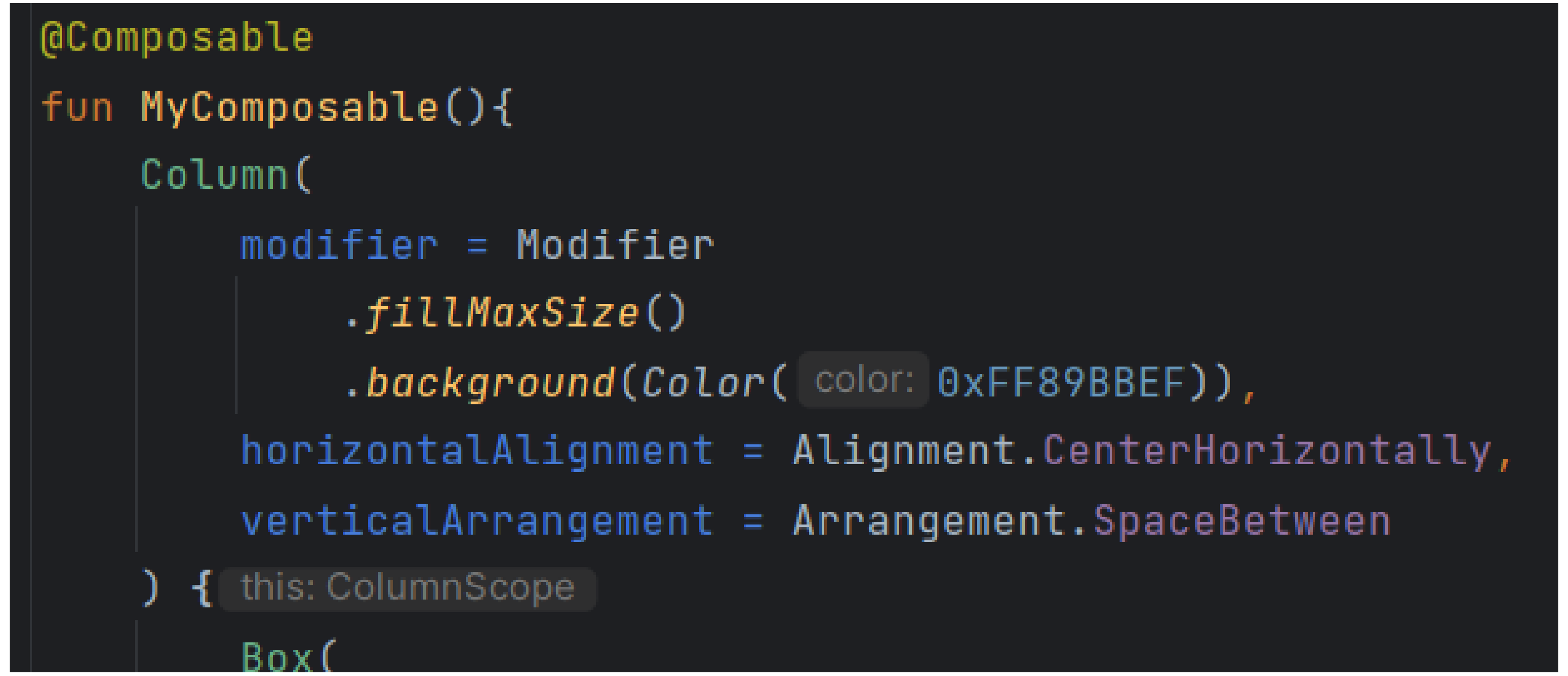
```
@Composable
fun MyComposable(){
    Row(
        modifier = Modifier
            .fillMaxSize()
            .background(Color( color: 0xFF89BBE4)),
        horizontalArrangement = Arrangement.SpaceEvenly
    ) { this: RowScope
        Box(
```



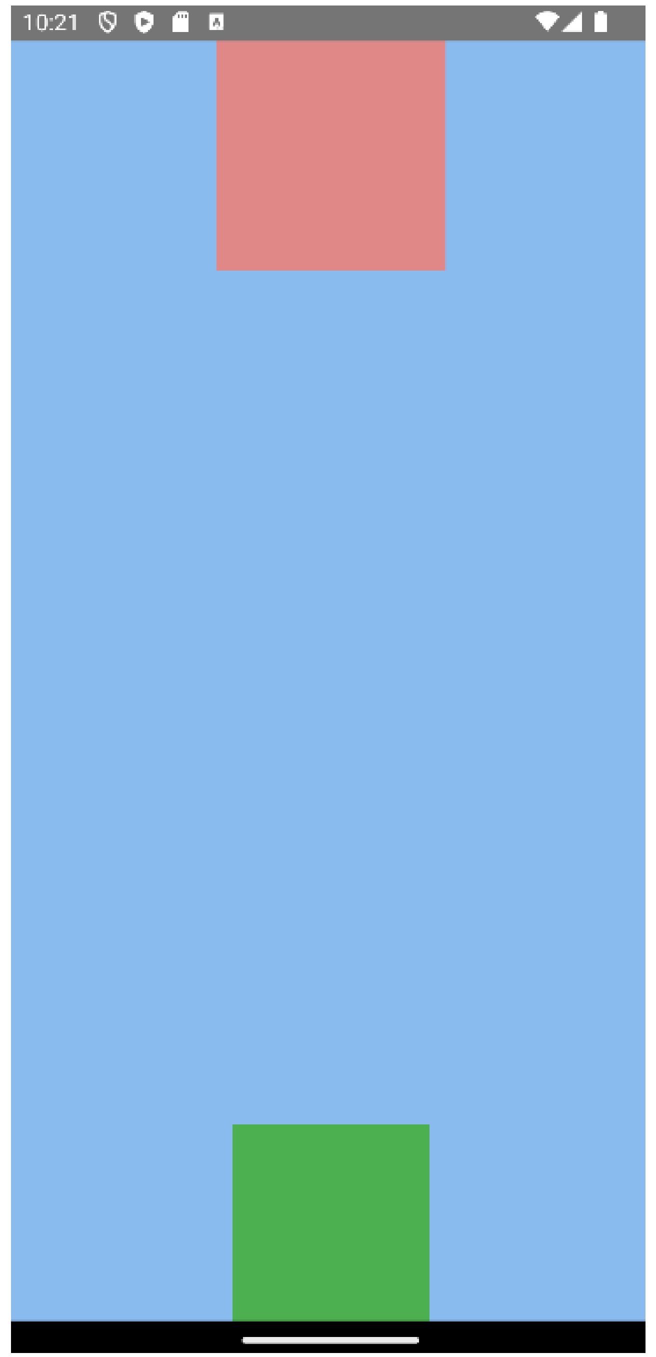
- 我们可以使用 fillMaxSize()、fillMaxWidth() 或 fillMaxHeight()

Rows and Columns (cont.)

- Here we align our boxes in a column to the center, and arrange them with space between

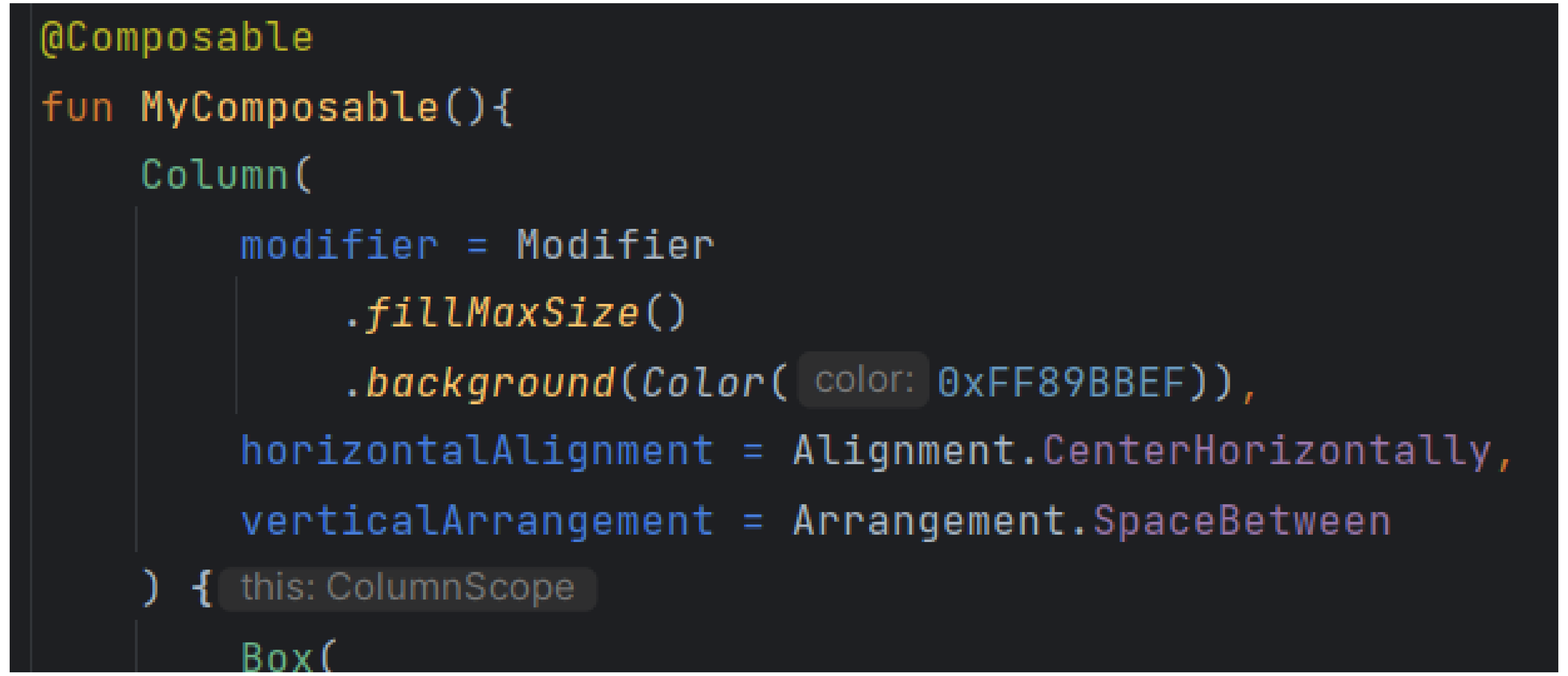


```
@Composable
fun MyComposable(){
    Column(
        modifier = Modifier
            .fillMaxSize()
            .background(Color( color: 0xFF89BBEF)),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.SpaceBetween
    ) { this: ColumnScope
        Box(
```

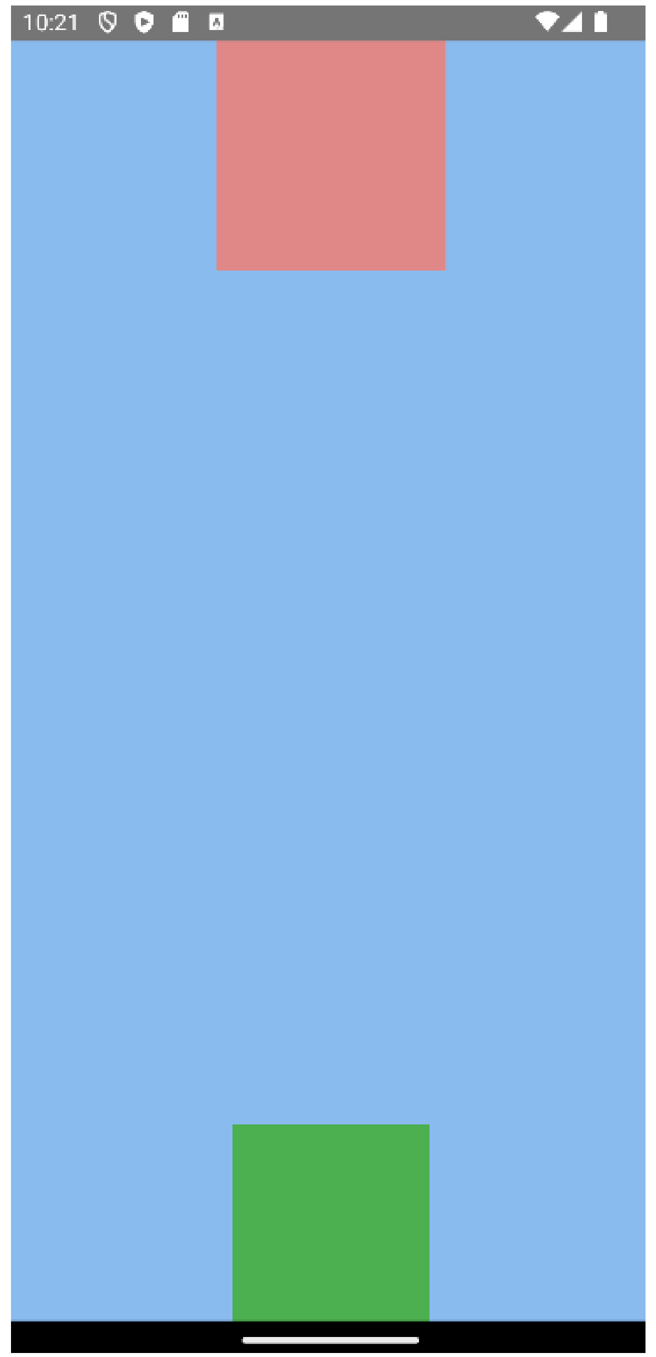


行和列（续）

- 我们将盒子在列中居中对齐，并用间距排列它们

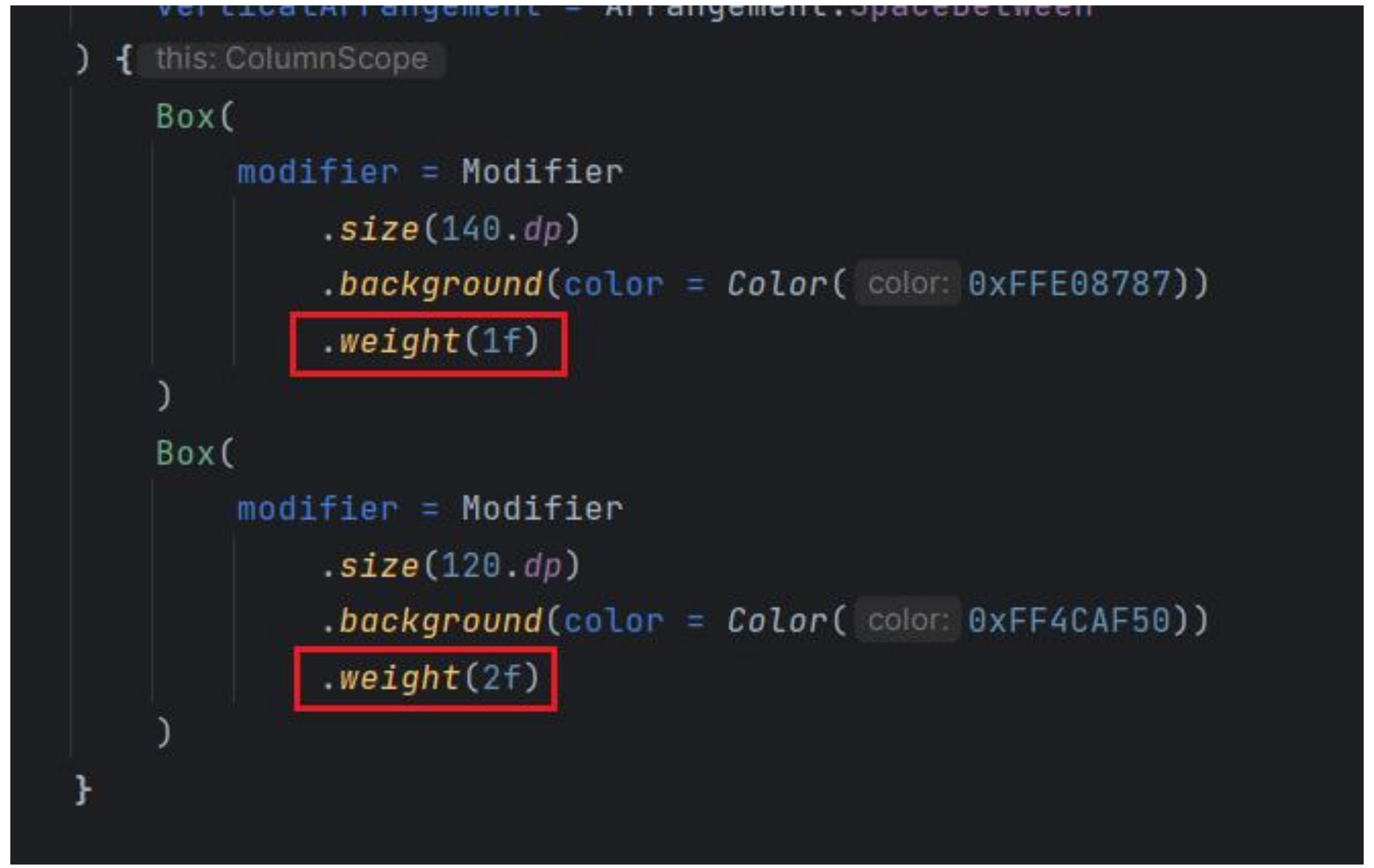


```
@Composable
fun MyComposable(){
    Column(
        modifier = Modifier
            .fillMaxSize()
            .background(Color( color: 0xFF89BBEF)),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.SpaceBetween
    ) { this: ColumnScope
        Box(
```

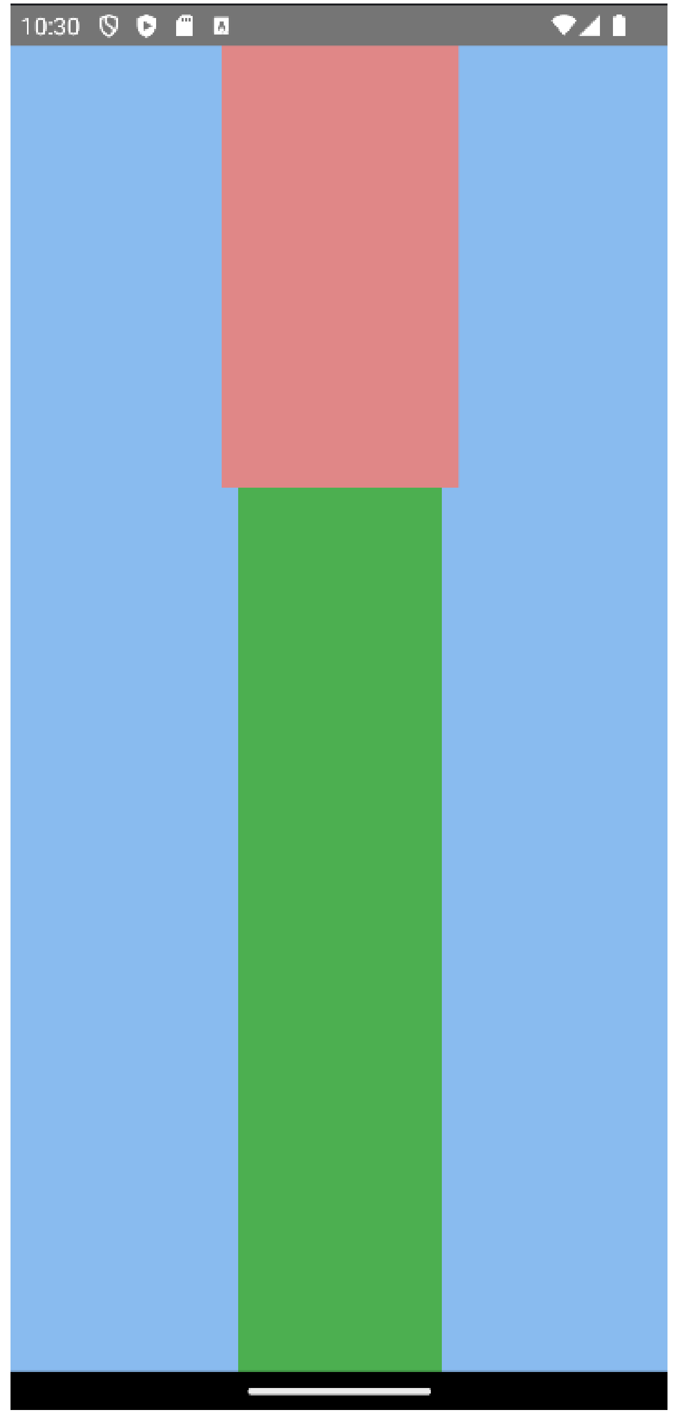


Rows and Columns (cont.)

- To have your children fill the available space, we use can use the *weight* modifier

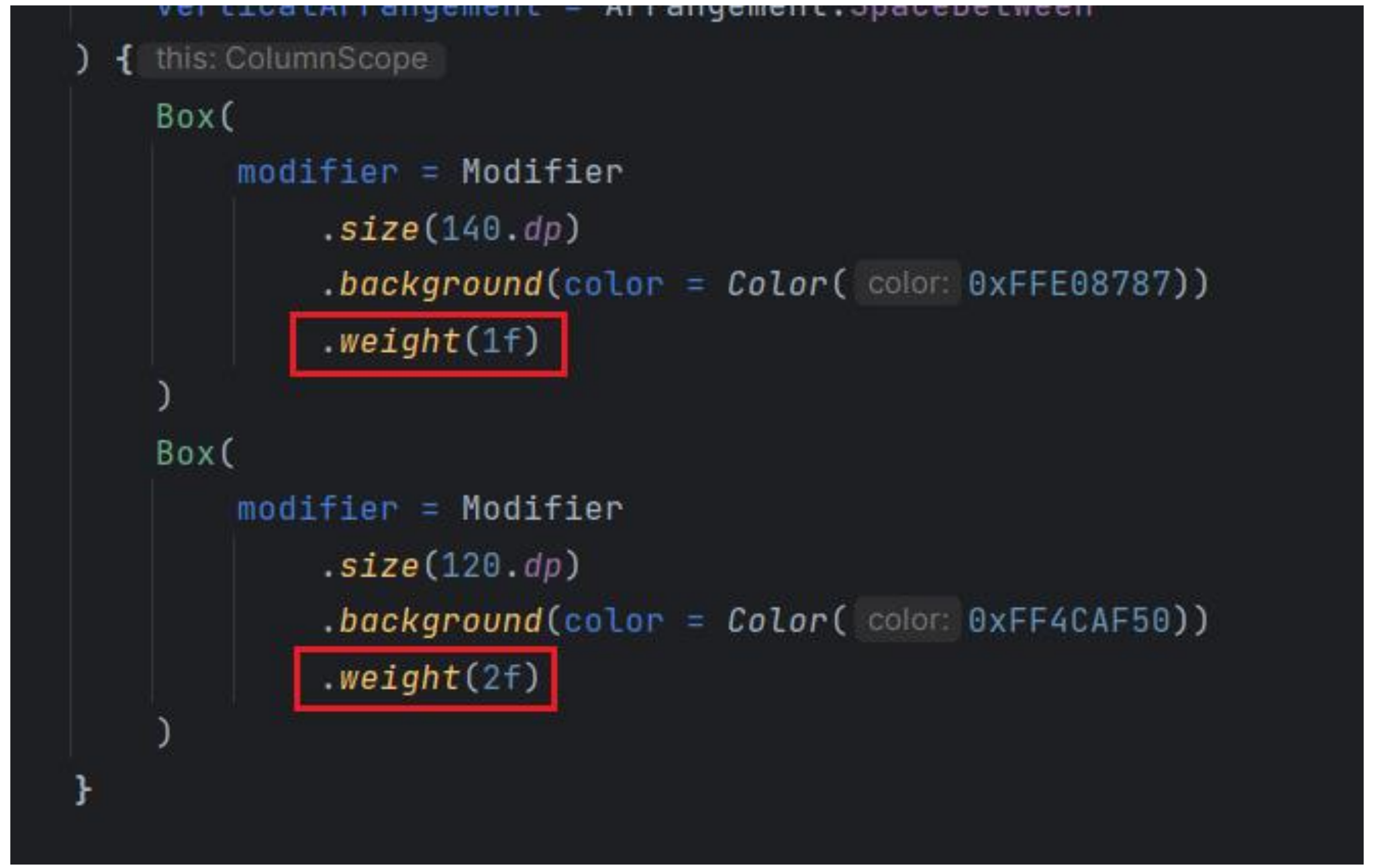


```
    verticalArrangement = Arrangement.SpaceBetween
) { this: ColumnScope
    Box(
        modifier = Modifier
            .size(140.dp)
            .background(color = Color( color: 0xFFE08787))
            .weight(1f)
    )
    Box(
        modifier = Modifier
            .size(120.dp)
            .background(color = Color( color: 0xFF4CAF50))
            .weight(2f)
    )
}
```



行和列（续）

- 要让子元素填满可用空间，我们可以使用 *weight* 修饰符



```
    verticalArrangement = Arrangement.SpaceBetween
) { this: ColumnScope
    Box(
        modifier = Modifier
            .size(140.dp)
            .background(color = Color( color: 0xFFE08787))
            .weight(1f)
    )
    Box(
        modifier = Modifier
            .size(120.dp)
            .background(color = Color( color: 0xFF4CAF50))
            .weight(2f)
    )
}
```