

# Lecture 6

COMP 3717- Mobile Dev with Android Tech

# 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

# 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

# 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

# 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

# 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

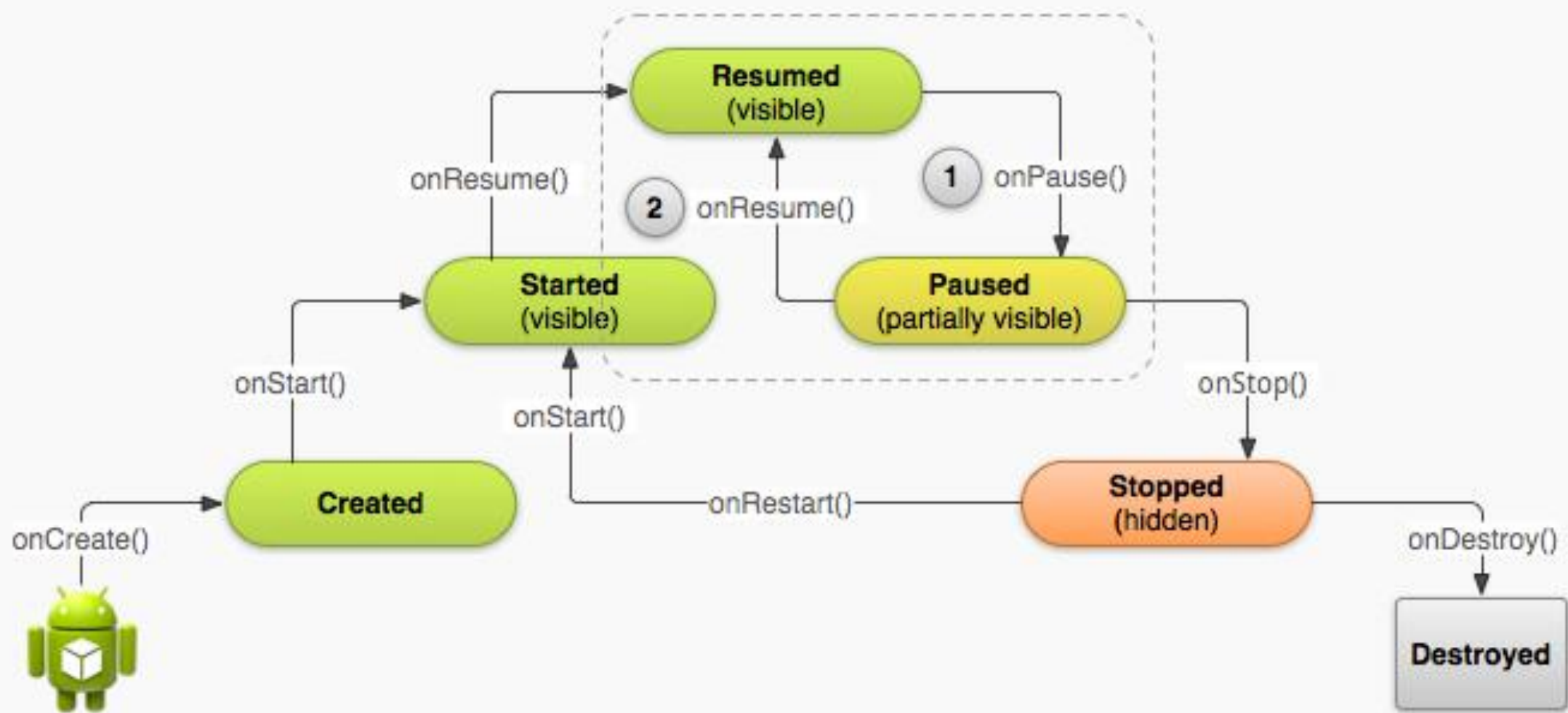
# What is activity lifecycle? (cont.)

- 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

# 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()





# 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{}*

# 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

# 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

# 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

# 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

# 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

# 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

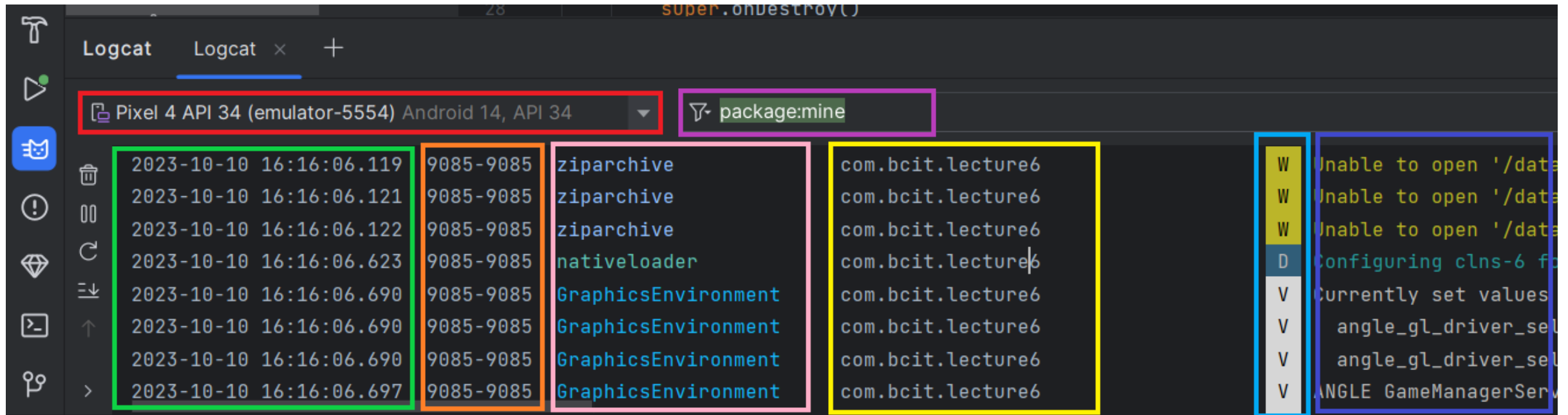
```
override fun onSaveInstanceState(outState: Bundle) {  
    super.onSaveInstanceState(outState)  
    Log.i( tag: "Life Cycle States", msg: "onSaveInstanceState")  
}
```



# 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**

# Logging in Android (cont.)



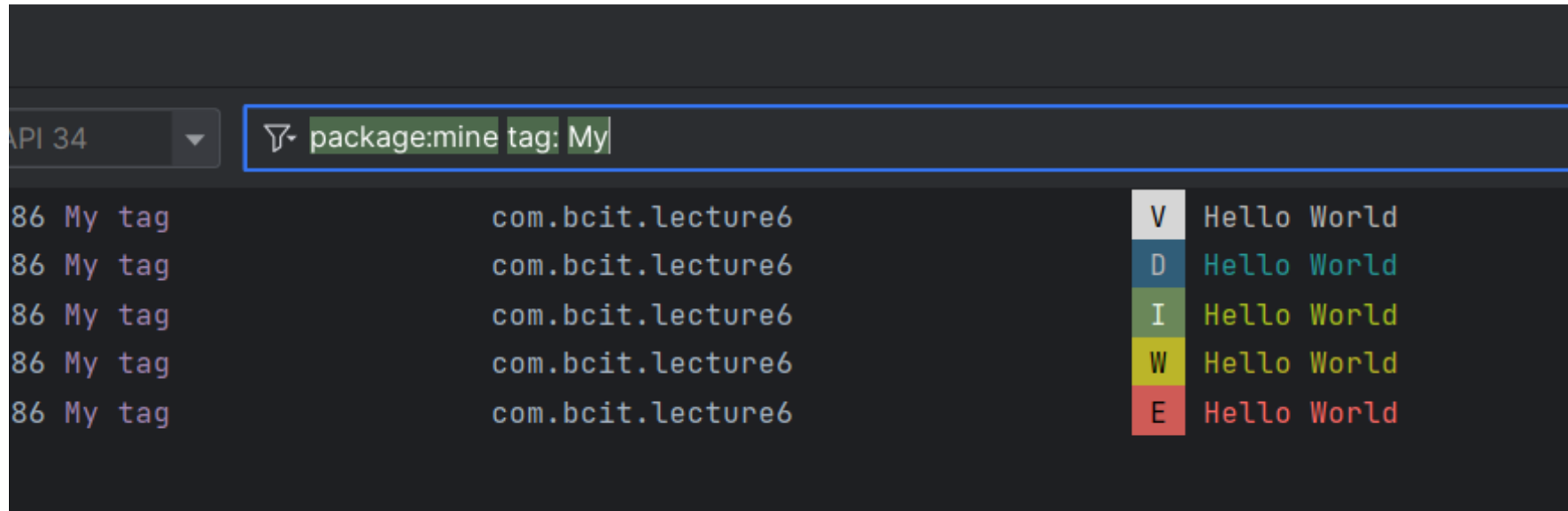
# Logging in Android (cont.)

- 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")  
    }  
}
```

# 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]*

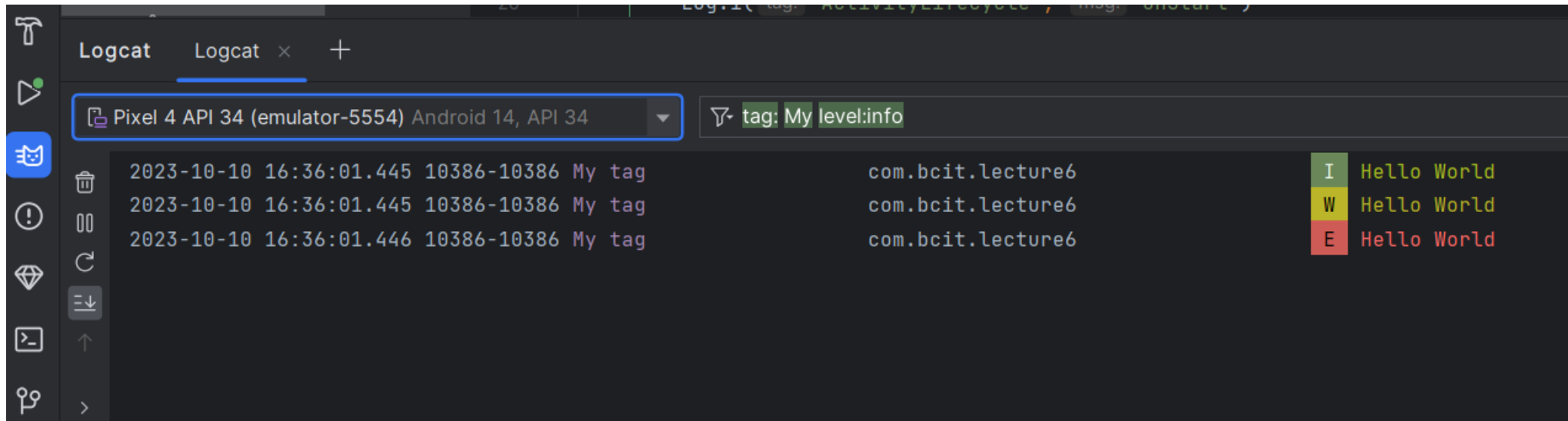


The screenshot shows the Logcat window in Android Studio. The filter bar at the top contains the text "package:mine tag: My". Below the filter bar, there are five log entries, all from the package "com.bcit.lecture6" and with the tag "My tag". The log levels are Verbose (V), Debug (D), Info (I), Warning (W), and Error (E), each with a corresponding colored icon. The log messages are "Hello World".

Time	Tag	Package	Level	Message
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.)

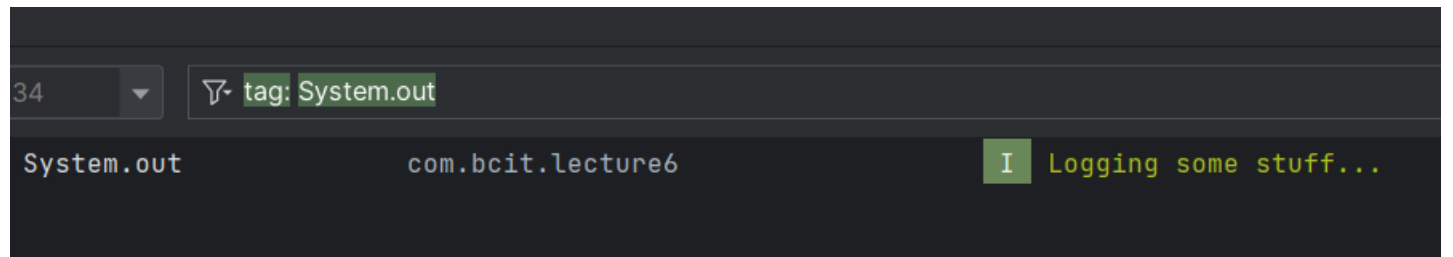
- Here we are filtering all tags in our whole project that start with *My*
- We are then only filtering those tags that are of priority *info* or higher



# 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...")  
    }  
}
```



# 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) {
```

## 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*

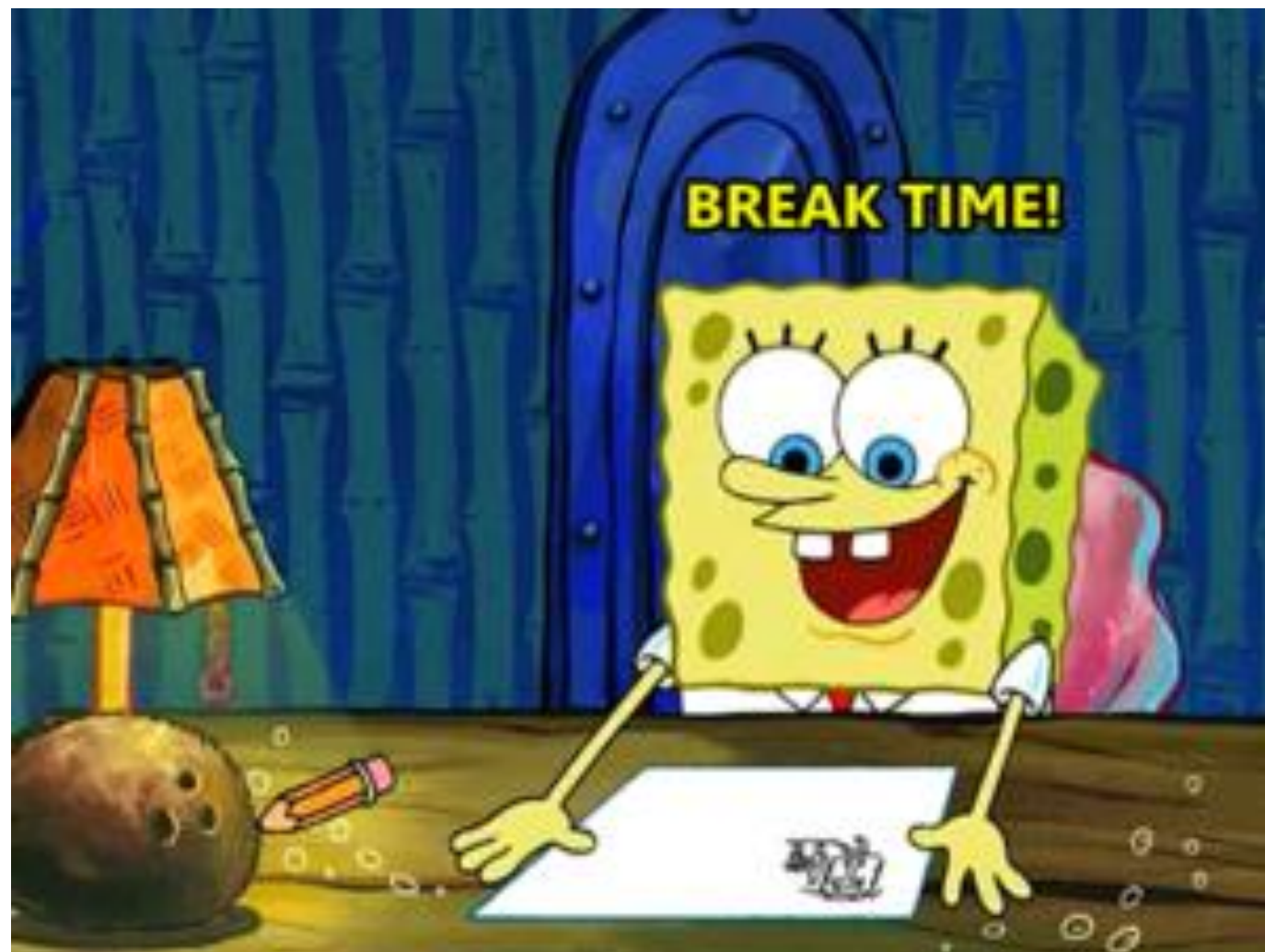


# Jetpack Compose

- 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 (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



# 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

# 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")  
}
```

# Composable Functions (cont.)

- You cannot put a composable function inside a regular function

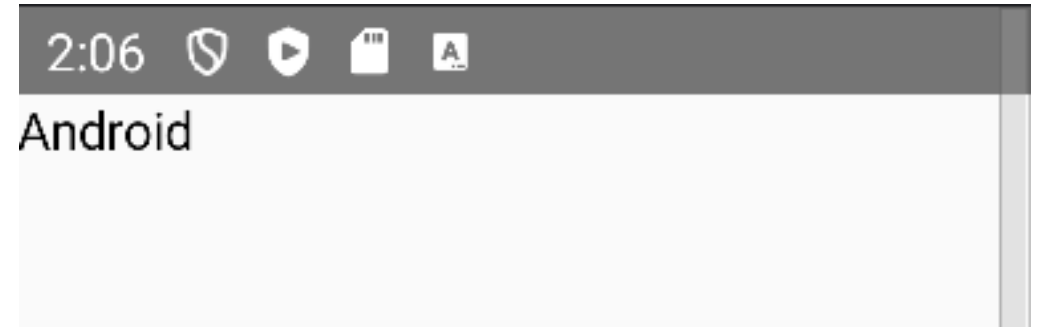
```
fun MyGreeting() {  
    Text(text  
}
```

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

# 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")  
}
```



# 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
    )
}
```



# Composable function parameters (cont.)

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

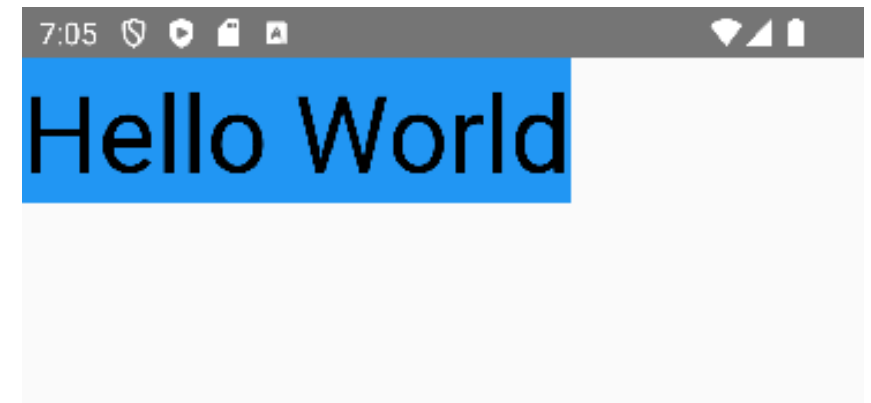
```
@Composable
@ComposableTarget
public fun Text(
    text: String,
    modifier: Modifier,
    color: Color,
    fontSize: TextUnit,
    fontStyle: FontStyle?,
    fontWeight: FontWeight?,
    fontFamily: FontFamily?,
    letterSpacing: TextUnit,
    textDecoration: TextDecoration?,
    textAlign: 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
```

# Composable modifiers

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

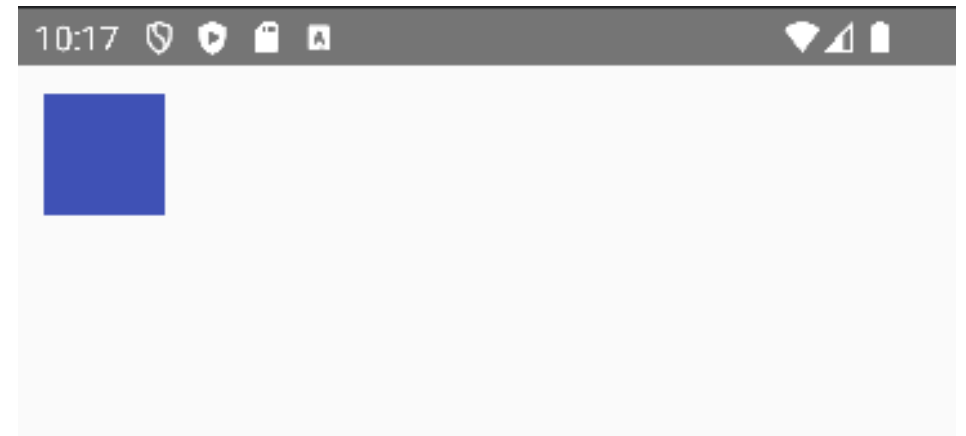
```
@Composable
fun MyComposable(){
    Text(
        modifier = Modifier.background(Color( color: 0xFF2196F3)),
        text = "Hello World",
        fontSize = 50.sp
    )
}
```



# 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))
    )
}
```

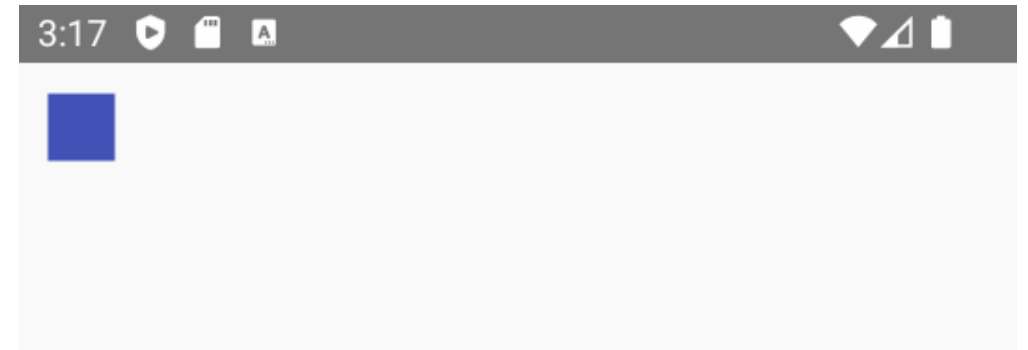


- A Box is a simple container

# 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))
    )
}
```

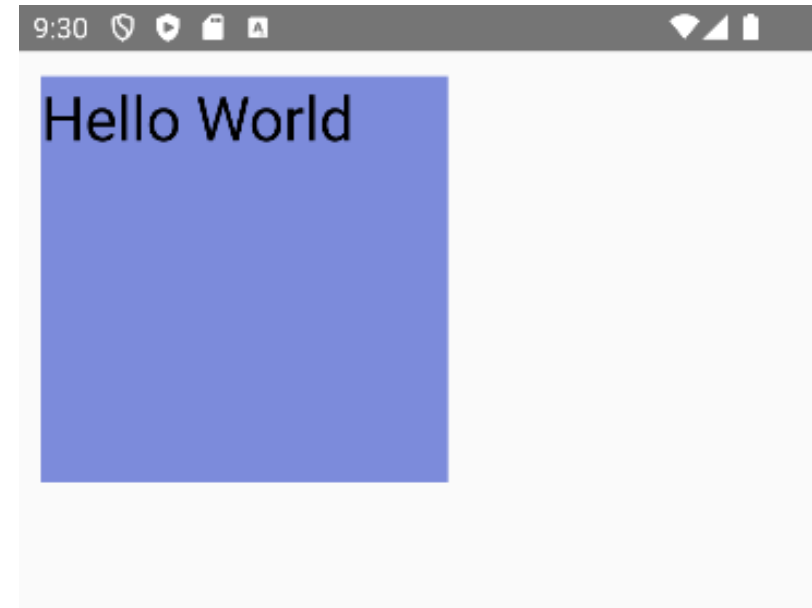


- We **swapped size and padding**, and you can see the result is different

# 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,
        )
    }
}
```



# Wrapping composables (cont.)

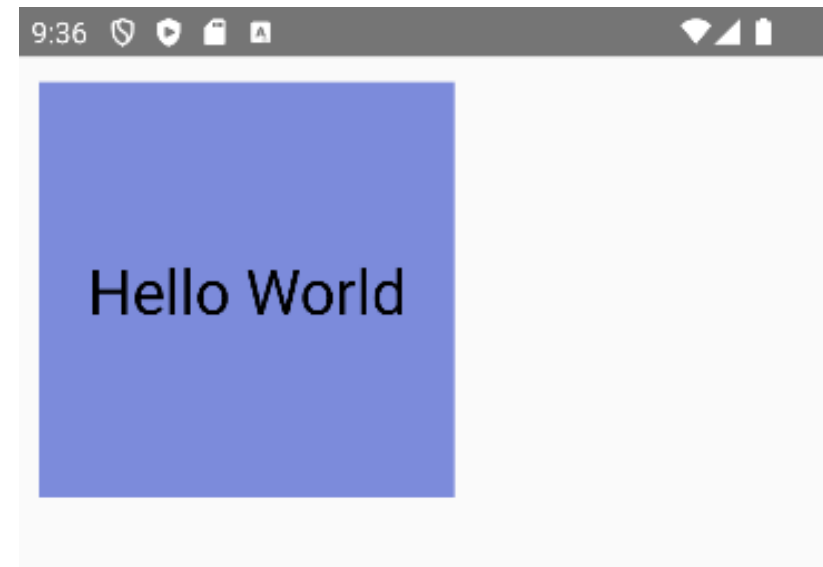
- When wrapping composables, the child composable goes inside the parent's scope

```
@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,
        )
    }
}
```

# 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,
        )
    }
}
```



# 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
    )
}
```



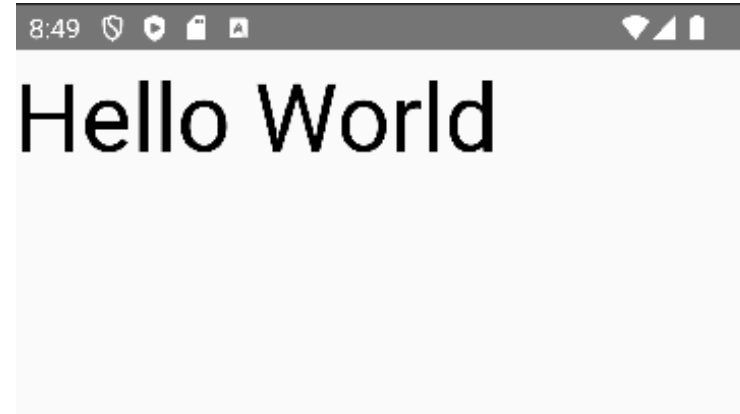
- Composables will just stack on top of each other if we don't provide a way to arrange them



# Row

- A *Row* allows us to arrange elements horizontally

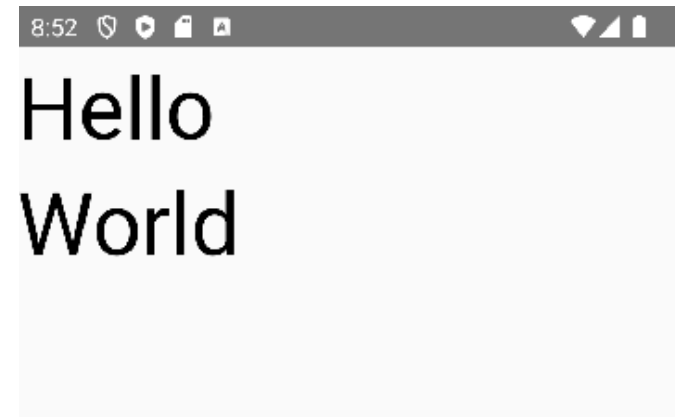
```
@Composable
fun MyComposable(){
    Row { this: RowScope
        Text(
            text = "Hello",
            fontSize = 50.sp
        )
        Text(
            text = " World",
            fontSize = 50.sp
        )
    }
}
```



# Column

- A *Column* allows us to arrange elements vertically

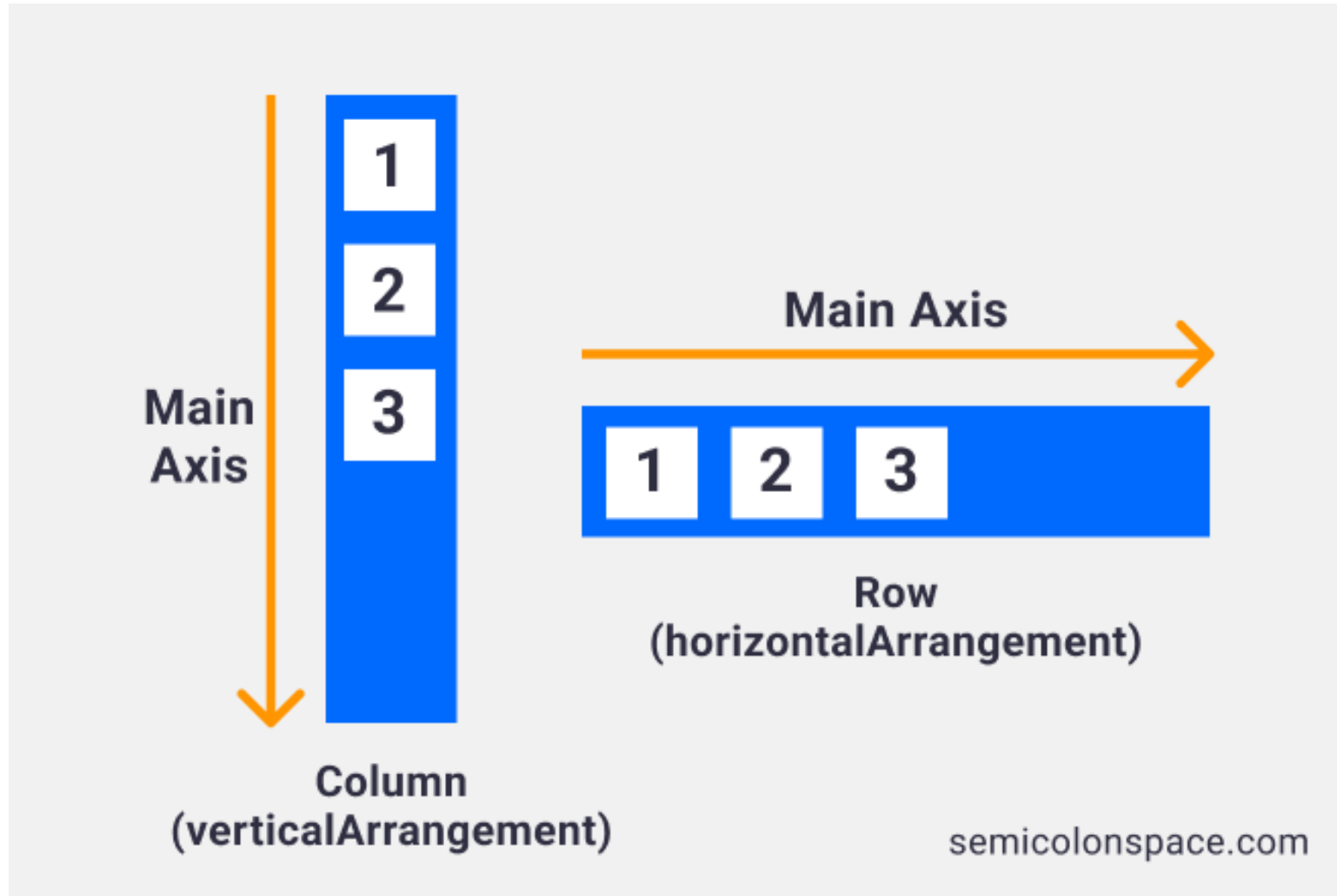
```
@Composable
fun MyComposable(){
    Column { this: ColumnScope
        Text(
            text = "Hello",
            fontSize = 50.sp
        )
        Text(
            text = "World",
            fontSize = 50.sp
        )
    }
}
```



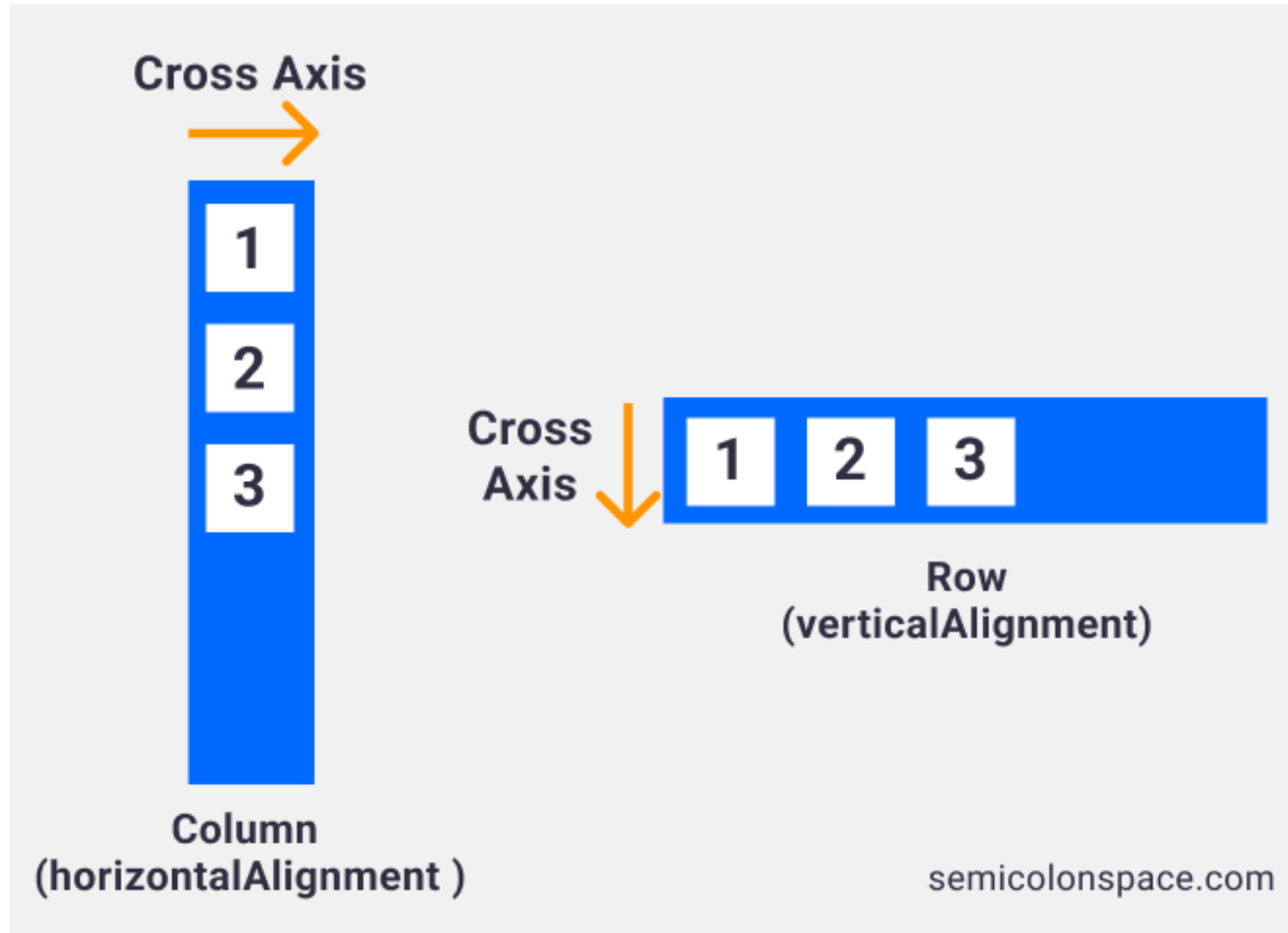
# Rows and Columns

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

# Rows and Columns (cont.)



# Rows and Columns (cont.)



# 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))
        )
    }
}
```



# Rows and Columns (cont.)

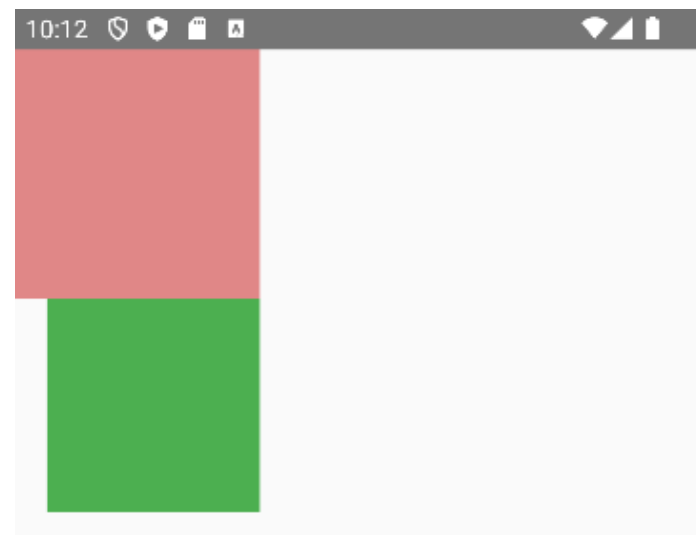
- Here we align the boxes in a row to the center

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



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

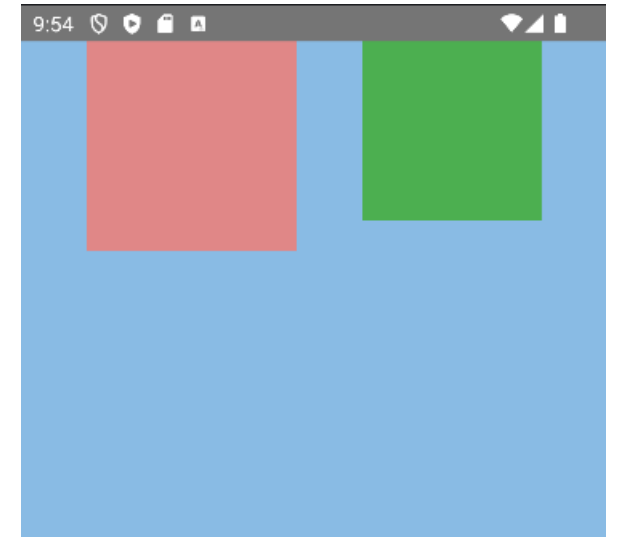
```
@Composable
fun MyComposable(){
    Column(
        horizontalAlignment = Alignment.End
    ) { this: ColumnScope
        Box(
```



# Rows and Columns (cont.)

- When working with *arrangement*, the Row or Column needs to fill its parent's space

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



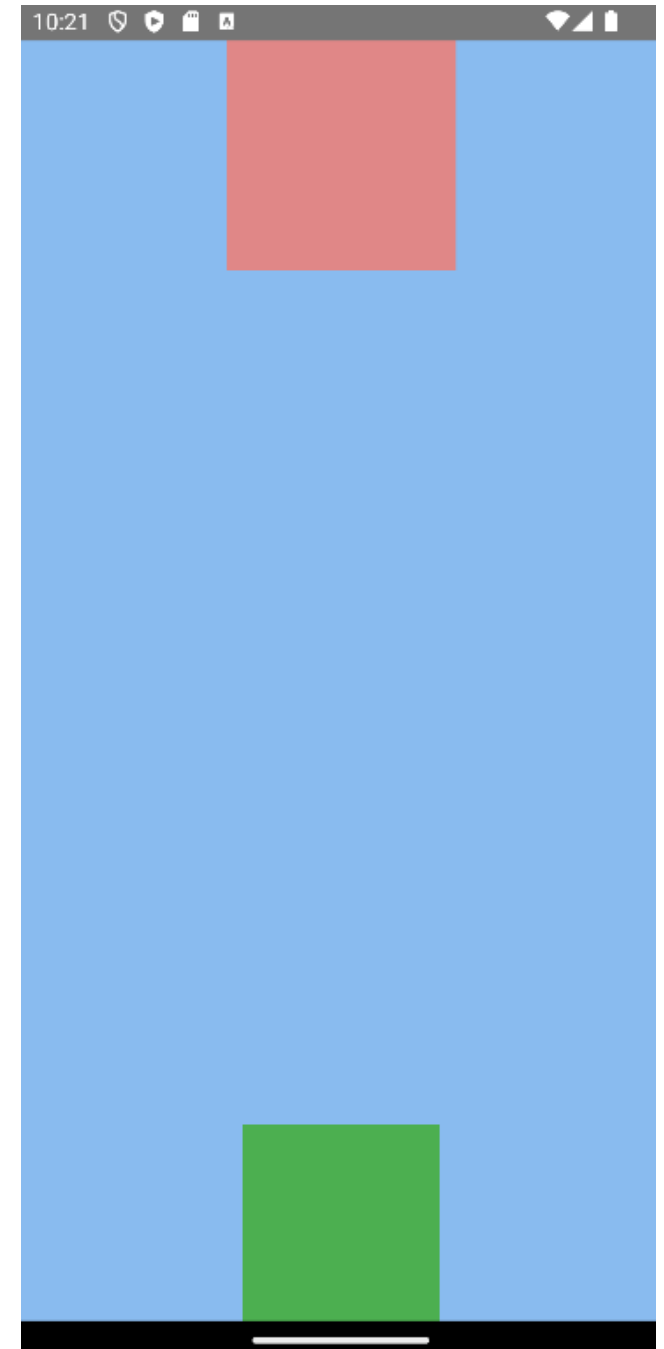
- We can fillMaxSize(), fillMaxWidth(), or fillMaxHeight()



# 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(
```



# 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)  
  )  
}
```

