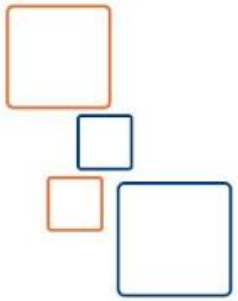
Android Applications Development Using Kotlin



Invest In Yourself,
Develop Your Career

Course Agenda

Day1

- Jetpack Compose
- Basic UI(Text, Button, Image..)
- Lists
- Modifiers

Day2

- Permissions
- Locations

Day3

- Broadcast Receivers
- Services
  - Background Services
  - Foreground Services
  - Started Services
  - Bound Services
  - IntentService
  - JobIntentService
- Notifications

Day4

- WorkManager

Day5

- Using coroutines in Android
  - Retrofit
  - Room
  - WorkManager

Day6

- Navigation
- Layouts

Course Duration and Evaluation

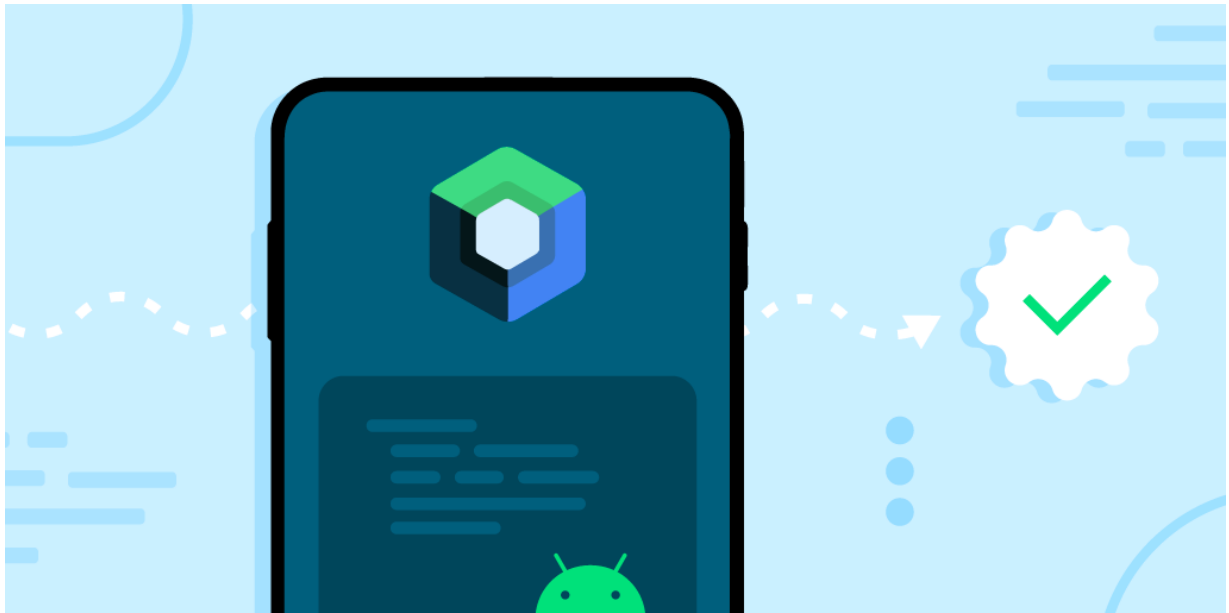
- **Duration (36 hours)**
  - **6 Lectures (18 hours)**
  - **6 Labs (18 hours)**

- **Evaluation Criteria**
  - **40% on Lab activity**
  - **60% on final project**

Let’s Start

What is Jetpack Compose?

- **Jetpack Compose** is a modern toolkit for building native Android UI.
- It was announced in 2019 and became stable in August 2020.



Why Jetpack Compose?

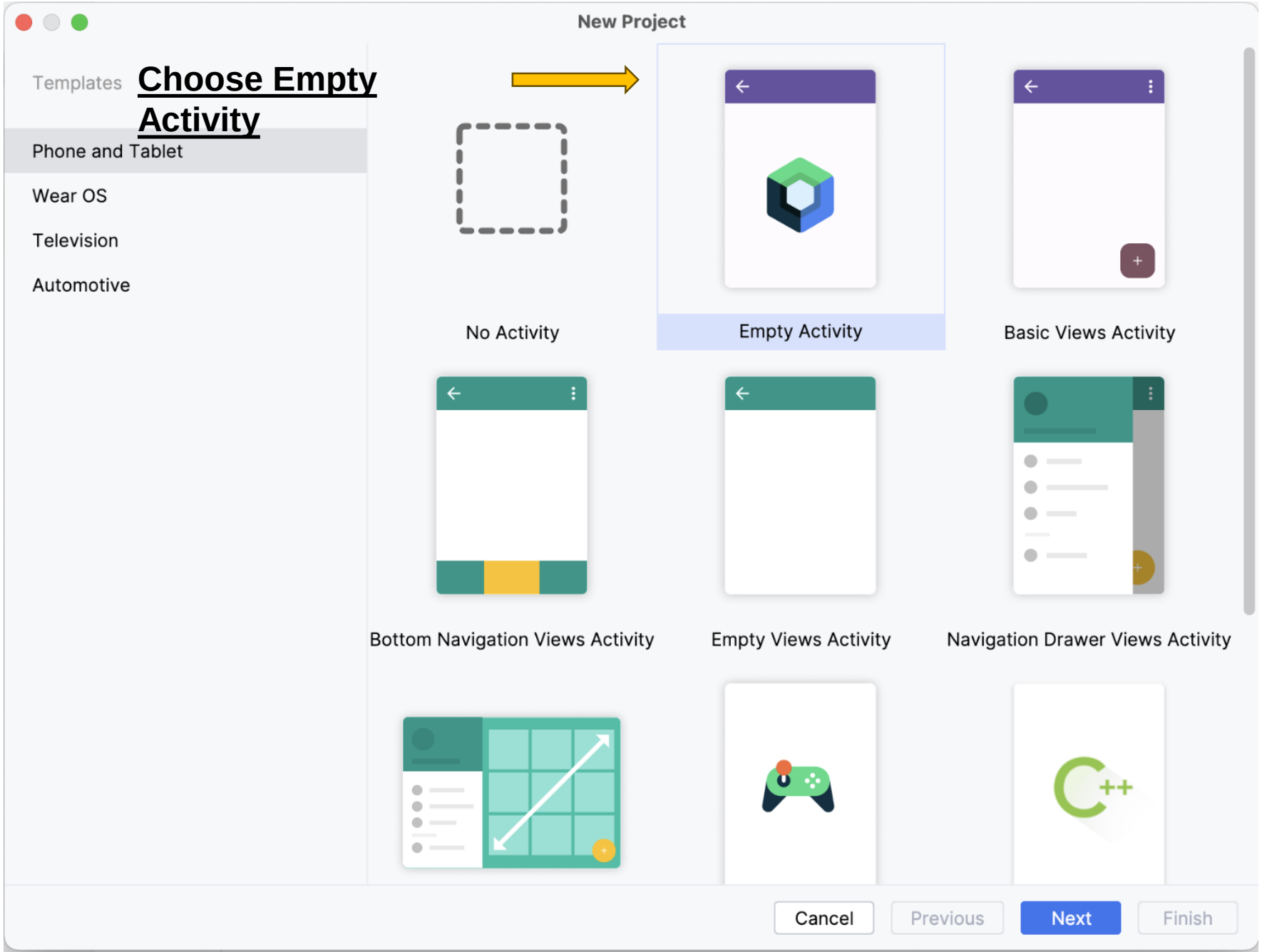
- Reduced Lines Of Code
- Intuitive
- Powerful
- Speeds up the development environment

Applications made by Jetpack Compose

Jetpack Compose

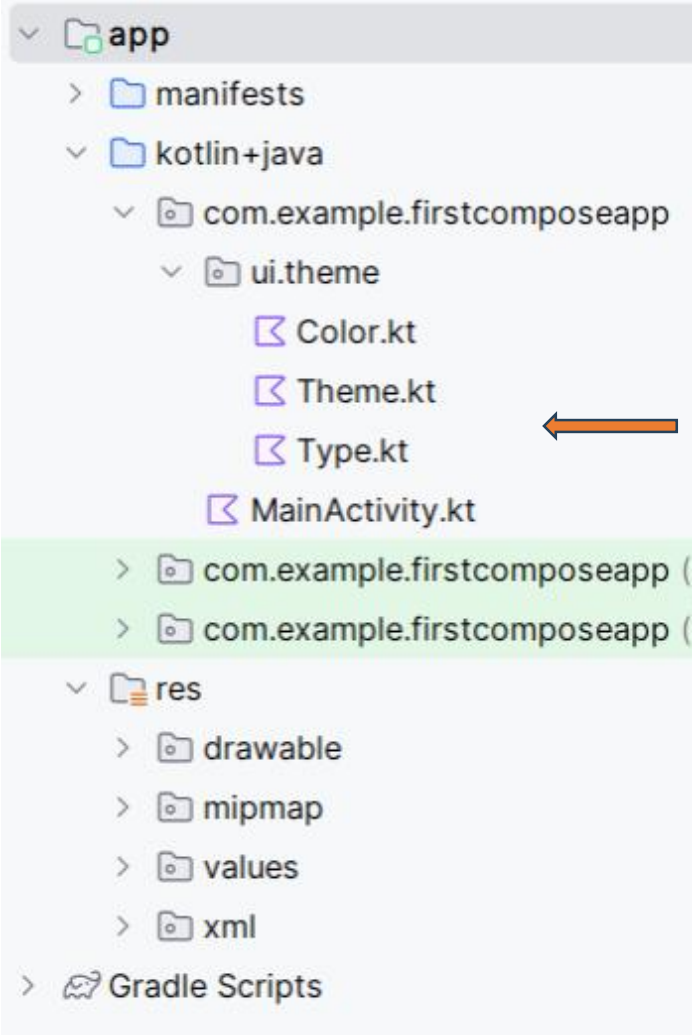
Java™ Education and Technology Services

Let’s Start a new Project using Compose


New Project
Templates
Choose Empty Activity
Phone and Tablet
Wear OS
Television
Automotive
No Activity
Empty Activity
Basic Views Activity
Bottom Navigation Views Activity
Empty Views Activity
Navigation Drawer Views Activity
C++
Cancel
Previous
Next
Finish

Compose projects have no layout files

- Many of the files and folders should look familiar to you, as they're the same ones that get generated for projects that don't use Compose. It includes :
- *Source Code: Activity subclass*
- *Resources :*
  - drawable
  - mipmap
  - values
  - xml
- The big difference is that **Android Studio doesn't generate any layout files for you**.
- This is because Compose projects use activity code to define the screen's appearance instead of layouts.



There's an extra package for the app's theme, along with files for the theme's colors, shapes, and typography.

What Compose activity code looks like

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

The activity needs to extend ComponentActivity

The onCreate() method needs to call setContent().

Compose activities extend ComponentActivity

- Activity doesn't extend **AppCompatActivity**
- It extends **ComponentActivity** instead.
- androidx.activity.ComponentActivity is a subclass of Activity, and it's used to define a basic activity that uses Compose for its UI instead of a layout file.
- Just like all the other activities you've seen, the activity overrides the onCreate() method.
- Instead of calling ***setContentView()*** to inflate the activity's layout, however, it uses **setContent()**.
- This is an extension function that's used to add Compose components— called **composables**—to an activity's UI so they run when the activity gets created.

Composition in Jetpack Compose

In Jetpack Compose, **Composition** refers to the process of building a UI hierarchy by executing composable functions.

- When you call composable functions (e.g., Text, Button, Column), Compose creates a **tree of nodes** that represent the UI structure - This tree is called the **composition**.
- Each composable function contributes to the tree by emitting one or more **nodes**.
- These nodes represent UI elements (e.g., text, buttons, layouts) and their properties (e.g., size, color, padding).
- The composition is **immutable** during a single frame.
- When state changes, Compose **recomposes** (rebuilds) the tree to reflect the new state.

UI Hierarchy

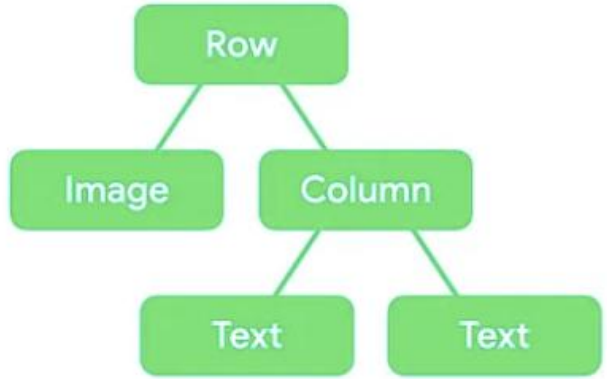
```
<LinearLayout >
        <ImageView/>
        <LinearLayout >

        <TextView/>

        <TextView/>
        </LinearLayout>
</LinearLayout>
```

Traditional XML based UI

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



Composable based UI

Composable Functions

- In Jetpack Compose in order to build the UI we use Composable functions
- Composable functions are the functions annotated with **`@Composable`**
- Let's start with the "Hello World!" example!

```
@Composable
 fun GreetingWorld() {
        Text(text = "Hello world!")
}
```

Composable Functions with Arguments

```
@Composable
fun Hello(name:String) {
    Text(text = "Hello $name! Welcome to Jetpack Compose")
}
```

- As you can see, the function is annotated with **@Composable**. This annotation is ***required for all composable functions***. If you omit the annotation, the code won't compile.

Preview Composable functions

- Another feature of using composable functions is that you can **preview** them within Android Studio without having to load the app onto a device.
- You can preview any composable function so long as it doesn't have any arguments, and you can even use this technique to preview entire **compositions**—UIs made up of composables.
- You say you want to preview a composable function by annotating it with **@Preview**. The following code, for example, specifies a composable function named **PreviewMainActivity** that lets you preview two Hello composables arranged in a column:

```
@Preview
@Composable
fun PreviewGreetingWorld(){
    Text(text = "Hello world!")
}
```

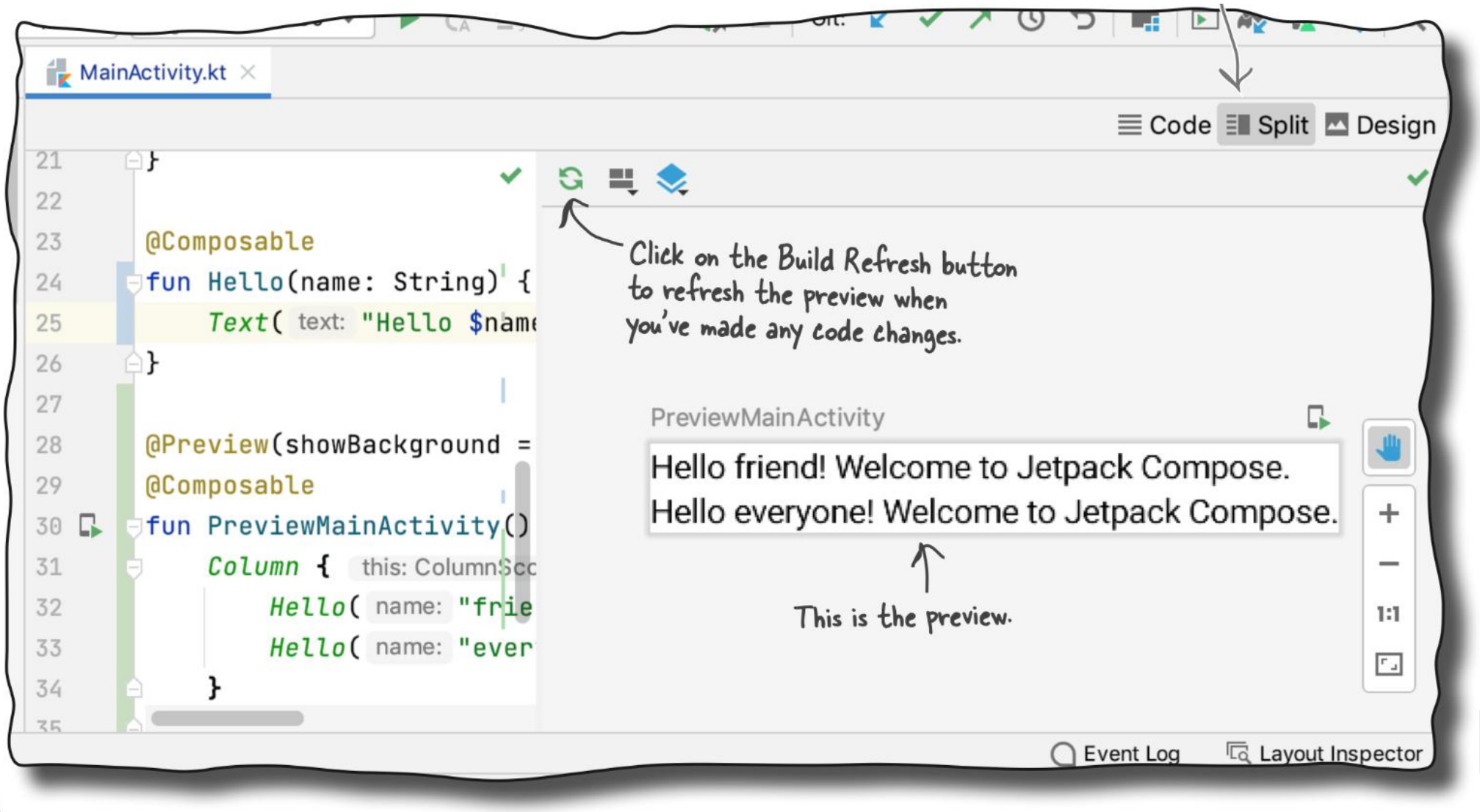
You can add different paramters to @Preview

Add this function to the end of the file so it's outside the MainActivity class definition.

More about @Preview

Preview Composable functions

The split option let U see the file's code and preview side by side



Composable Functions

PascalCase VS camelCase

`PascalCase`

Composable functions are always responsible for drawing the UI,
so they return Unit. Since they return Unit,
the naming convention follows **PascalCase**.
(e.g., Text(), Row())

`camelCase`

Composable functions that return a state rather than directly drawing the UI,
the naming convention follows **camelCase**.
(e.g., animateFloatAsState())

Basic UI Blocks

Component	Description
Text	High level element that displays text and provides semantics / accessibility information.
TextField	Text fields allow users to enter text into a UI.
Image	A component used to display images in the UI.
Button	A clickable UI element that performs an action when pressed.
CheckBox	A UI element that allows users to toggle between checked and unchecked states.
RadioButton	A UI element that allows users to select one option from a group of choices.

Don’t Forget

- In the context of building components, where previously the ***View*** class was employed, the ***@Composable*** annotation is now utilized to define components within Jetpack Compose framework.

- You’ll notice that you don’t have to extend any class or override constructors or other functions. All you need to care about is that you write a function and use this new fancy annotation

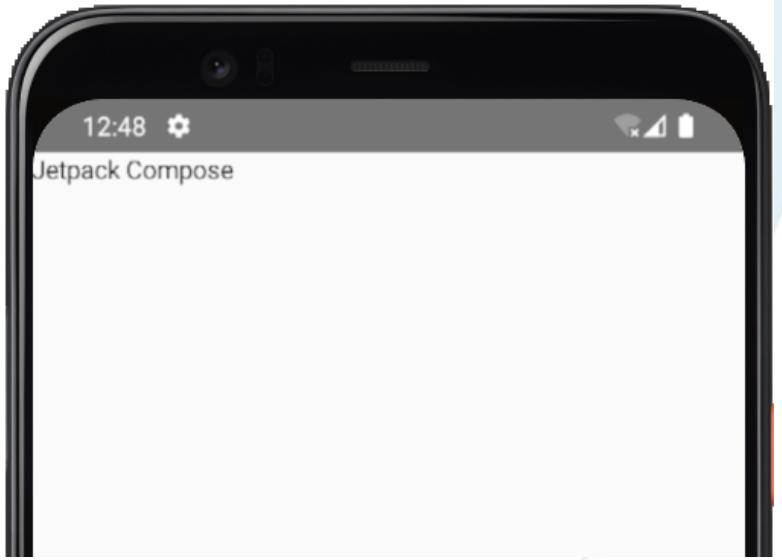
Demo

Hello World!

Use a Text composable to display text

- We’re going to make MainActivity display some text by adding a **Text** composable to the call to **setContent()**. You can think of Text as being the Compose equivalent of a text view.

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



Composable Functions

You can use any property of Text and any other composables using **named-arguments** and set their values.

```
@Composable
fun Text(
    text: String,
    modifier: Modifier = Modifier,
    color: Color = Color.Unspecified,
    fontSize: TextUnit = TextUnit.Unspecified,
    fontStyle: FontStyle? = null,
    fontWeight: FontWeight? = null,
    fontFamily: FontFamily? = null,
    letterSpacing: TextUnit = TextUnit.Unspecified,
    textDecoration: TextDecoration? = null,
    textAlign: TextAlign? = null,
    lineHeight: TextUnit = TextUnit.Unspecified,
    overflow: TextOverflow = TextOverflow.Clip,
    softWrap: Boolean = true,
    maxLines: Int = Int.MAX_VALUE,
    minLines: Int = 1,
    onTextLayout: (TextLayoutResult) -> Unit = {},
    style: TextStyle = LocalTextStyle.current
) {
```

Most UIs have multiple composables

- When you have a UI that includes multiple composables, you need to specify how they should be arranged. If you don't, Compose will stack the composables on each others like this:

```
Text(text = "MAD")
Text(text = "JETS")
```

Java™ Education and Technology Services

Layouts

What are layouts in Jetpack Compose?

- Standard Layouts:
  - **Column**
  - **Row**
  - **Box**

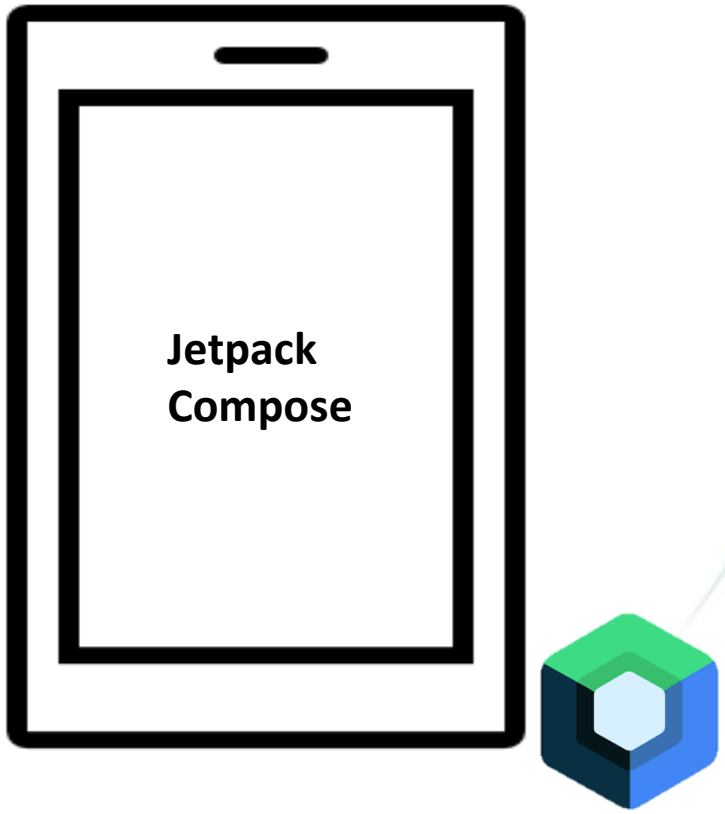
- There are other many layouts but they are the standard layouts.

Layouts - Column

1) Column is A layout composable that places its children in a Vertical sequence

it's like a "Vertical Linear Layout".

```
@Composable
fun MyColumn() {
    Column {
        Text(text = "Jetpack")
        Text(text = "Compose")
    }
}
```

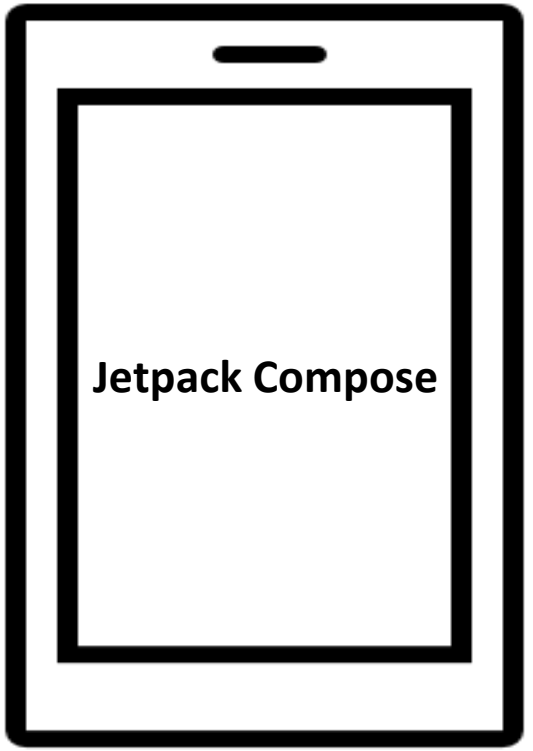


Layouts - Row

2) Row is A layout composable that places its children in a Horizontal sequence

it's like a "Horizontal Linear Layout".

```
@Composable
fun MyRow() {
   Row {
      Text(text = "Jetpack")
      Text(text ="Compose")
   }
}
```



Layouts - Box

3) Box is A layout composable that places its children stacked over each other

It’s like a “Frame Layout”.

```
@Composable
fun MyBox() {
    Box {
        Text(text = “Jetpack”)
        Text(text =“Compose”)
    }
}
```

Accessing Resources

In Jetpack Compose, you can use various @Composable functions to access resources efficiently

- **stringResource(id:Int)**: Loads a string from res/values/strings.xml.

```
Text(text = stringResource(R.string.app_name))
```

- **painterResource(id: Int)**: Loads a raster or vector images from res/values/drawable.xml.

```
Image( painter = painterResource(R.drawable.weather), …)
Icon(painter = painterResource(R.drawable.next), …)
```

- **colorResource(id: Int):** Loads a color from image.

```
Divider(color = colorResource(R.color.purple_200))
```

Also read **developer.android** for more resources

The full code for MainActivity.kt

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Column {
                Hello(name = "MAD")
                Hello(name = "JETS")
            }
        }
    }
}
@Composable
fun Hello(name: String) {
    Text(text = "Hello $name! Welcome to Jetpack Compose", fontSize = 20.sp)
}
```

Display Composable Functions

Image

```
@Composable
fun MyImage() {
    Image(
        painter = painterResource(id = R.drawable.dog),
        contentDescription = stringResource(id = R.string.dog_content_description),
        contentScale = ContentScale.Crop
    )
}
```

the ***ContentScale*** property in Jetpack Compose functions similarly to the ***ScaleType*** attribute in traditional Android **ImageView**, dictating how an image should be scaled or resized within its bounding box.

Display Composable Functions

Button

Simple Button:

```
@Composable
fun MyButton() {
    Button(onClick = {}) {
        Text(text = stringResource(R.string.login))
    }
}
```

- As you see here button takes lambda as click Listener
- Has a text as a caption
- It may have any another composable(Like icon, image ..etc.).

Java™ Education and Technology Services

Modifier

Modifier

- Modifier allow you to **decorate** or augment a composable.
- Modifier tell a UI element how to layout, display, or behave within its parent layout.

- Modifier let you do these sorts of things:
  - Change the composable's size, layout, behavior, and appearance
  - Add information, like accessibility labels
  - Process user input
  - Add high-level interactions, like making an element clickable, scrollable, draggable, or zoomable

Some Modifiers for the Layout

- **Modifier.width()**

  You can use this to set the width of a Composable.

- **Modifier.height()**

  You can use this to set the height of a Composable.

- **Modifier.size()**

  You can use this to set the width and height of a Composable.

- **Modifier.padding()**

  You can use it to set padding to Composables that take a modifier as an argument.

Some Modifiers for the Layout

- **Modifier.fillMaxWidth()**

  This will set the width of the Composable to the maximum available width.

  This is similar to **MATCH_PARENT** from the classic View system.

- **Modifier.fillMaxHeight()**

  This will set the height of the Composable to the maximum available height.

  This is similar to **MATCH_PARENT** from the classic View system.

- **Modifier.fillMaxSize()**

  This will set the height/width of the Composable to the maximum available height/width.

Some Modifiers for the Drawing

- **Modifier.background()**

  With this modifier you can set a background color/shape for the Composable.

- **Modifier.clip()**

  This modifier can clip the Composable to rectangle, rounded, or circle.

- **Modifier.border()**

  This modifier can add a border to the Composable with its size.

Some Modifiers for the Gesture

- **Modifier.clickable**

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

- **Modifier.scrollable**

  You can use this to make a Composable scrollable

- **Modifier.draggable**

  You can use this to make a Composable draggable

- **Modifier.swipeable**

  You drag elements which, when released, animate towards typically two or more anchor points defined in an orientation

Java™ Education and Technology Services

List and Grids

Lazy List

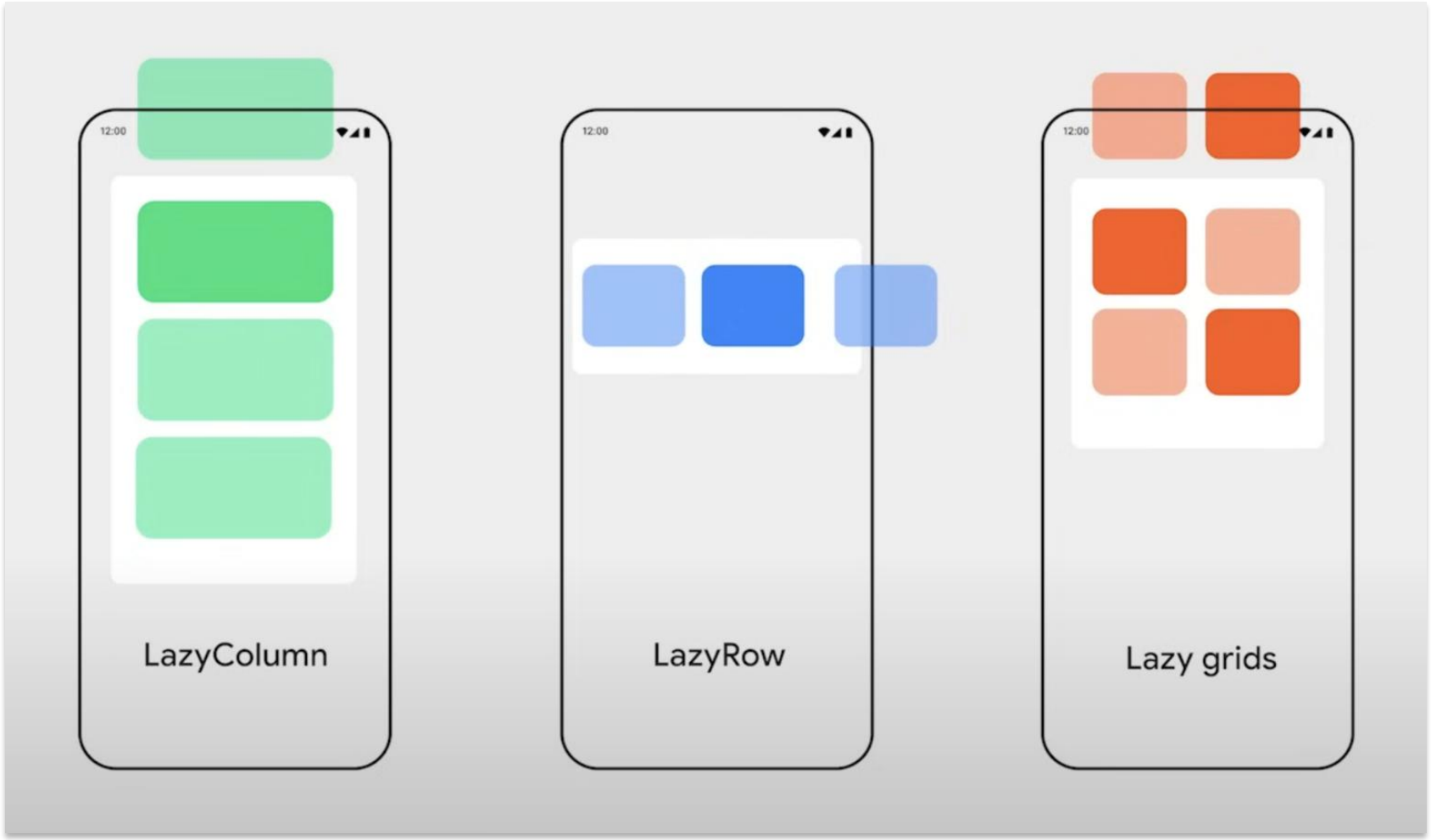
Lazy List

- Many apps need to display a collection of items, and when we think about that, we can use **Column** layout with **verticalScroll()** and make a scrollable list.

- If you need to display a large number of items (or a list of an unknown length), using a layout such as **Column** can cause performance issues, since all the items will be composed and laid out whether or not they are visible.

- Compose provides a set of components which only compose and lay out items which are visible in the component's viewport. These components include **LazyColumn** and **LazyRow.**

Lazy List



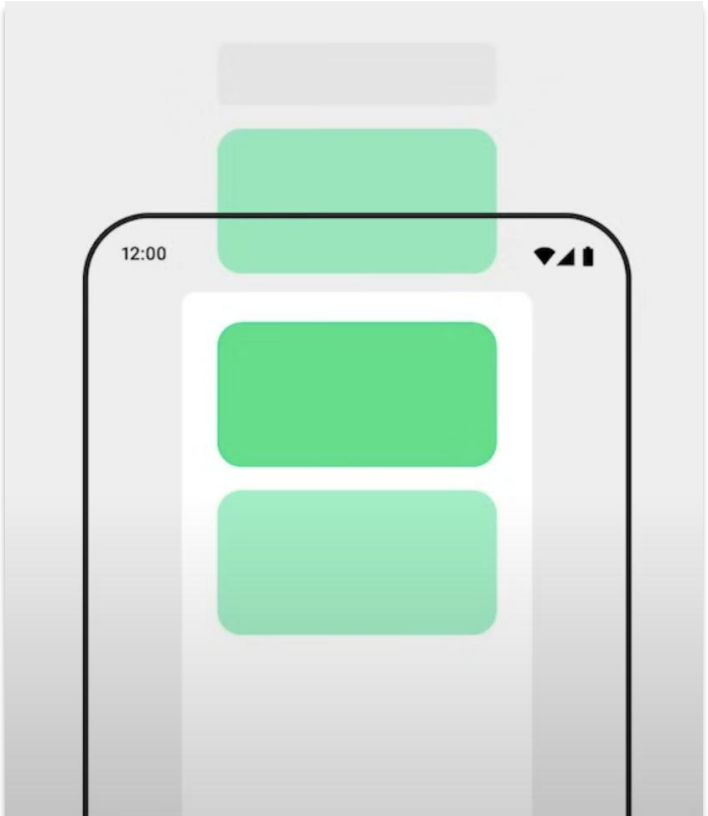
Jetpack Compose vs Traditional XML

Jetpack Compose	Traditional XML	Description
LazyColumn OR LazyRow	RecyclerView + LayoutManager	**container** that handle the display of items in a list & **handle the layout logic** for vertical and horizontal scrolling
items **OR** itemsIndexed	Adapter + ViewHolder	**responsible for defining how list items are created and bound to data**
Row [single item]	Row [single item]	**The row that will hold the data for each item.**

Lazy List

- **LazyListScope** provides a number of functions for describing items in the layout. At the most basic, **item()** adds a single item, and **items(Int) / itemsIndexed(List)** adds multiple items.
- **Lazy Column**:

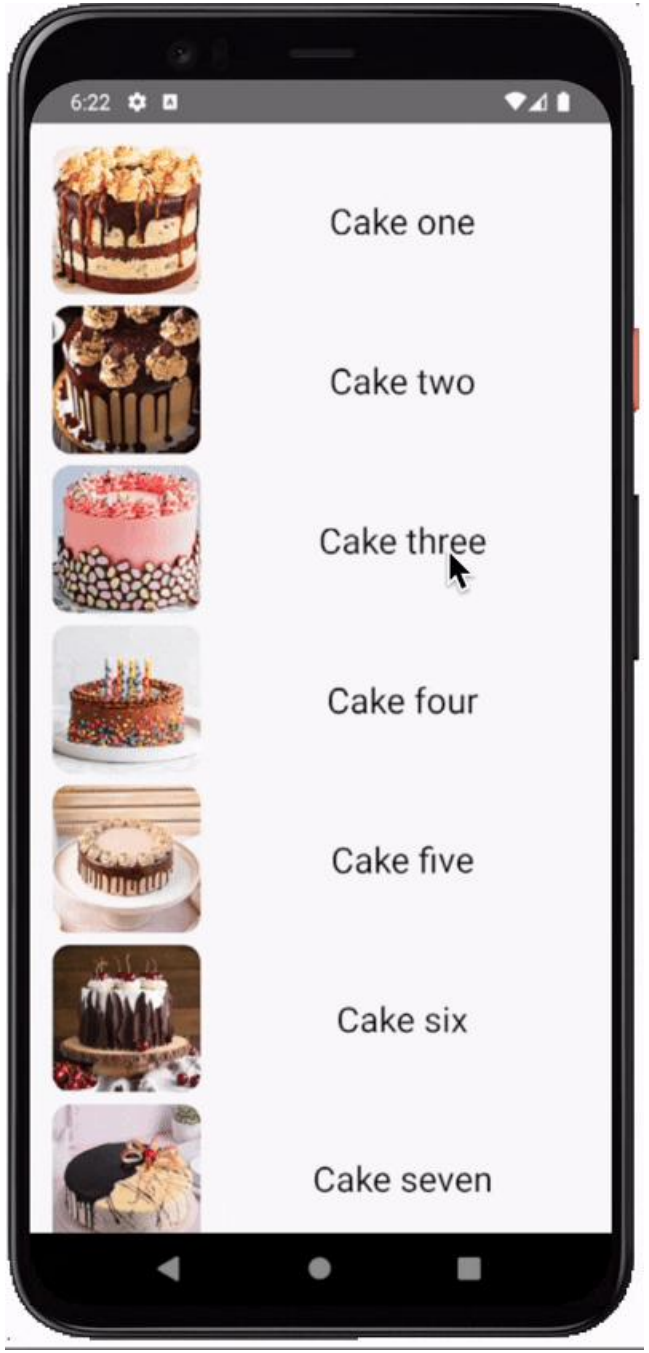
```
LazyColumn {
    item {
        Text(header)
    }
    items(data) { item->
        Item(item)
    }
}
```



Lazy List

- **Lazy Column**

```
val cakes = listOf(
Cake(R.drawable.one, “Cake one”…) …)
LazyColumn(
    modifier = Modifier.fillMaxSize()
) {
    items(cakes.size) {
        CakeRow(cakes[it])
    }
}
```



Lazy List

- **Lazy Row**

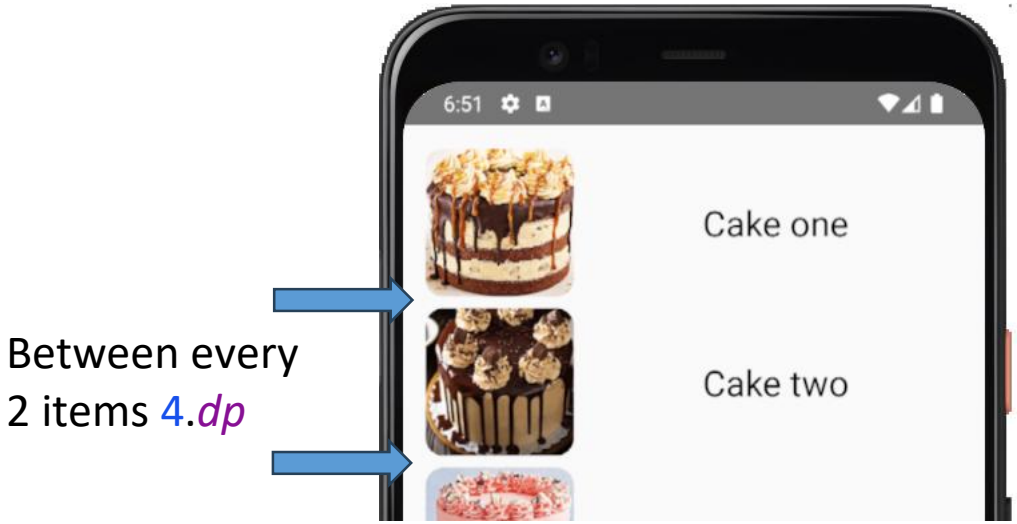
```
val cakes = listOf(
Cake(R.drawable.one, “Cake one”…) …)
LazyRow(
    modifier = Modifier.fillMaxSize()
) {
    items(cakes.size) {
        CakeRow(cakes[it])
    }
}
```

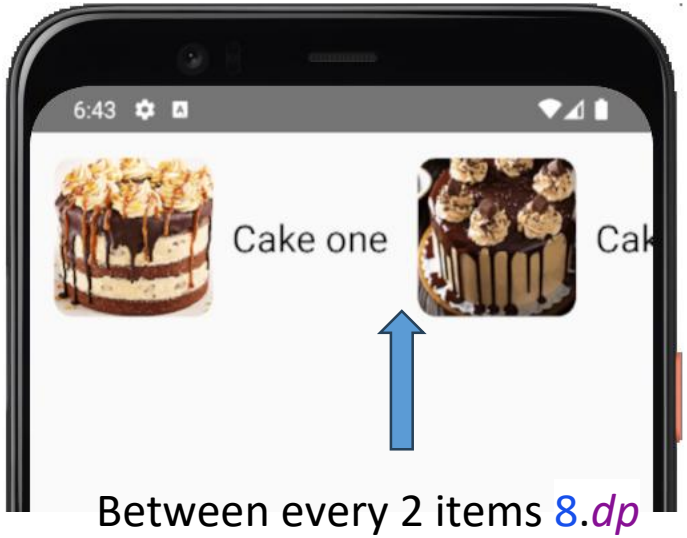
Lazy List

Content spacing

When using **LazyColumn** or **LazyRow**, you may need to add spacing between list items. This can be achieved using the following parameters:

- with **LazyColumn**: verticalArrangement = Arrangement.spacedBy(4.*dp*)
- with **LazyRow**: horizontalArrangement = Arrangement.spacedBy(8.*dp*)





Lazy List

Content padding

Sometimes you'll need to add padding around the edges of the content. The lazy components allow you to pass some **PaddingValues** to the contentPadding parameter to support this:

```
LazyColumn( contentPadding =
                PaddingValues(horizontal = 16.dp, vertical = 8.dp), ) { // ...
}
```

Java™ Education and Technology Services

Labs

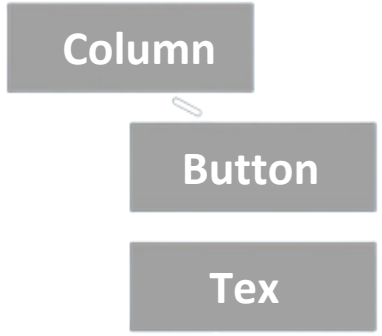
Lab 1

- Create an application that displays a Lazy List whose each row consists of:
- Image
- 2 Texts

Recomposition

Recomposititon

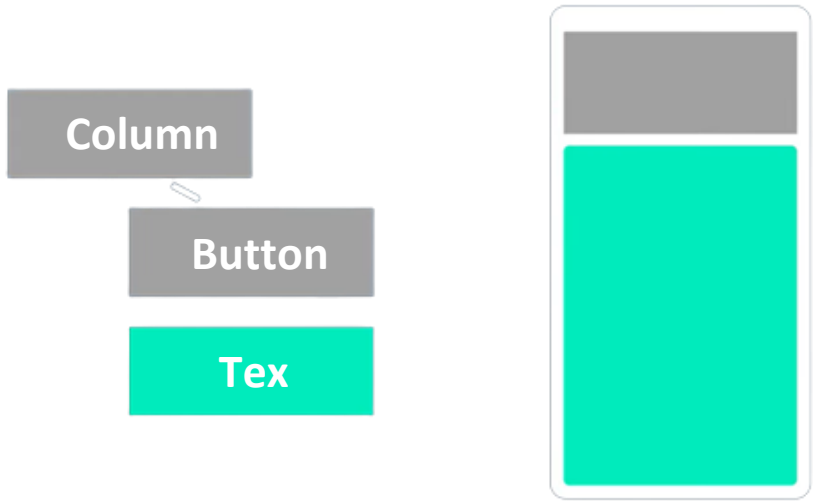
- Jetpack Compose UIs are **dynamic** & **reactive**.
- When state changes, the UI elements that depend on this state are "recomposed," meaning they are redrawn to reflect the current state.
- **Recomposition** allows any composable function to be **re-invoked** at any time to rerender the component based on new data.



Recomposititon

As we discussed, recomposing the entire UI tree can be computationally expensive, which uses computing power and battery life.

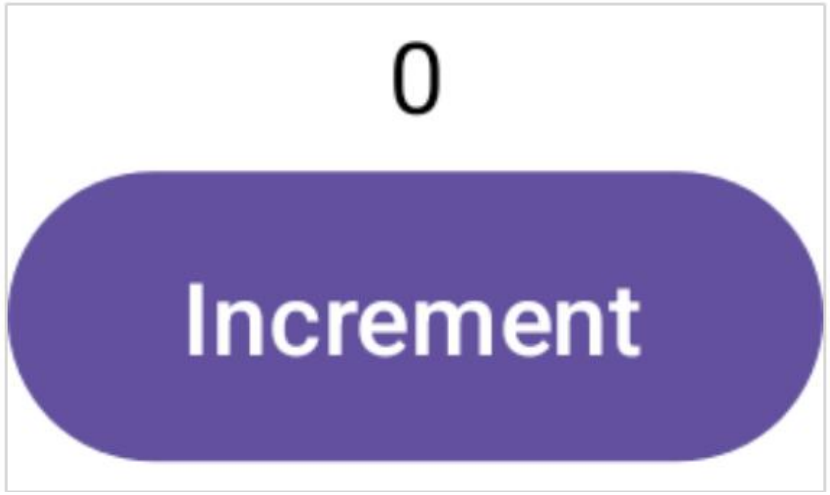
Compose solves this problem with this *intelligent recomposition*.



State Observation

Jetpack Compose - Demo

Let’s make this design:



Jetpack Compose - Demo

```
@Composable
private fun Counter() {
    var count = remember {
        mutableStateOf(0)
    }
    Column(
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        Text(text = "${count.value}")
        Button(onClick = { count.value++ }) {
            Text(text = "Increment")
        }
    }
}
```

Why You Can't Use Non-Observable Variables

- If you use a non-observable variable in Jetpack Compose, changes to that variable won't trigger a recomposition.
- This is because Compose has no way of knowing that the variable has changed.
- Compose cannot track the state and react to changes of non-observable variables.

State Observation in Compose

- **mutableStateOf**: creates instance of MutableState which is a state holder that Compose can observe.
- When the value of a MutableState changes, Compose is notified and can trigger a recomposition of the UI.
- **remember**: is used to store a value across recompositions.
- Without remember, the state would be re-initialized every time the composable function is recomposed, losing its previous value.

TextField state observation

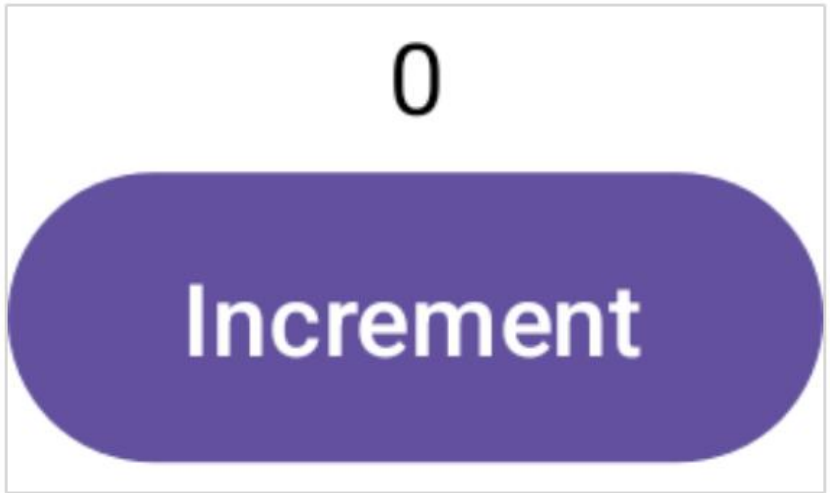
TextField

```
@Composable
fun MyTextField() {
    val textValue = remember { mutableStateOf ( "" ) }

    TextField (
        value = textValue.value,
        onValueChange = {
            textValue.value = it
        },
        label = { Text("Label") }
    )
}
```

Counter Demo - Orientations

What happens on configuration change?



Counter Demo - Orientations

```
@Composable
private fun Counter() {
    var count = rememberSaveable {
        mutableStateOf(0)
    }
    Column(
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        Text(text = "${count.value}")
        Button(onClick = { count.value++ }) {
            Text(text = "Increment")
        }
    }
}
```

rememberSaveable automatically saves any value that can be saved across configuration changes

lateinit

Using `lateinit`, the initial value does not need to be assigned.

Furthermore, the type doesn't have to be nullable, so `?.` and `!!` are not used.

It acts as a promise to initialize this variable later on.

`lateinit` is only applied with `var`

Syntax:

```
lateinit var identifierName : Type
```

To check whether a lateinit var was really initialized use :: operator

lateinit Example

Of course you can access the lateinit var and use it without any check but then the normal result will be :

UninitializedPropertyAccessException

```
lateinit var person: Person

fun main() {

    println(person.name)
}
```

```
NestedAndInnerKt
"C:\Program Files\Android\Android Studio1\jre\bin\java.exe" ...
Exception in thread "main" kotlin.UninitializedPropertyAccessException: lateinit property person has not been initialized
```

lateinit Example

```
lateinit var person: Person

fun main() {
    //To check whether this property is initialized use :: with isInitialized
    if(::person.isInitialized){
        println(person.name)
    }else{
        person = Person( name: "Layla",  age: 35,  id: 123,  height: 155)
        println(person.name)
    }
}
```

```
NestedAndInnerKt
"C:\Program Files\Android\Android Studio1\jre\bin\java.exe" ...
Layla

Process finished with exit code 0
```

lazy Example

```
val person2: Person by lazy {
    println("This block of code will be computed only one time")
    Person( name: "Layla",  age: 35,  id: 123 ,  height: 155)  ^lazy
}

fun main() {
    if (person2.name.length< 10){
        println("Name is less than 10 ${person2.name}")
    }
    println(person2.name)
}
```

```
NestedAndInnerKt
"C:\Program Files\Android\Android Studio1\jre\bin\java.exe" ...
This block of code will be computed only one time
Name is less than 10 Layla
Layla

Process finished with exit code 0
```

lateinit vs. lazy

keyword	lateinit	lazy
Usage	Unlike what Kotlin asks for, properties can be initialized later on and you cannot supply a non-null initializer in the constructor, but you still want to avoid null checks when referencing the property inside the body of a class.	There are certain classes whose object initialization is very heavy and so much time taking that it results in the delay of the whole class creation process.
Used with	`var` only	`val` only

Syntax	`lateinit var variableName: DataType`	`val variableName: DataType by lazy{`
		`  //your initialization`
		`}`

Locations

What Location-Based Services Do

LBS approaches

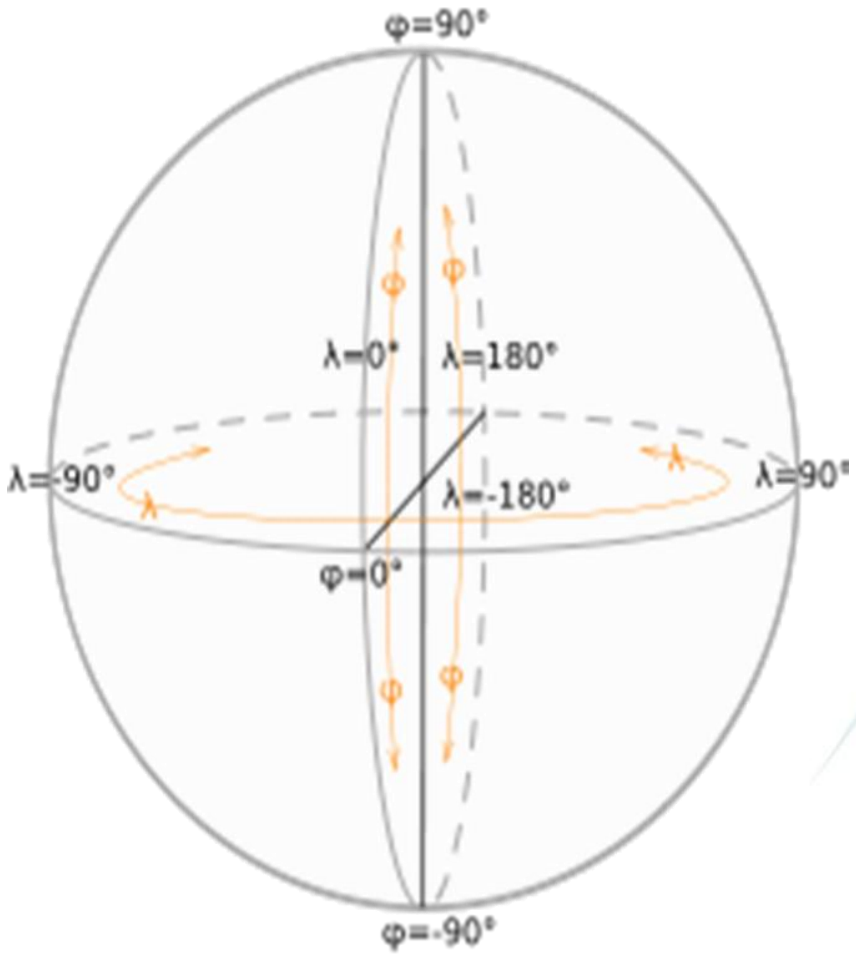
- There are two basic approaches to implementing location-based services:
  - Process location data in a server and deliver results to the device.
  - Obtain location data for a device-based application that uses it directly.

Location Based Services Expressions

- LBS must use real-time positioning methods.
- Accuracy depends on the method used.
- Locations can be expressed in spatial terms or as text descriptions.
- A spatial location can be expressed in the widely used *latitude-longitude-altitude* coordinate system.
- A text description is usually expressed as a street address, including city, postal code,..

Spatial terms

- **Latitude:**
  - 0-90 degrees north or south of the equator
- **Longitude:**
  - 0-180 degrees east or west of the prime meridian, which passes through Greenwich
- **Altitude:**
  - Meters above sea level



Determining the Device's Location

- Mobile phone network
- Short-range *positioning beacons*
- Satellites

Mobile phone network

- The current cell ID can be used to identify the Base Transceiver Station (BTS) that the device is communicating with and the location of that BTS.
- The accuracy of this method depends on the size of the cell.
- A GSM cell may be anywhere from 2 to 20 kilometers in diameter.

Short-range positioning beacons

- In relatively small areas, such as a single building, a local area network can provide locations along with other services.

- For example, appropriately equipped devices can use Bluetooth for short-range positioning.

Satellites

- The Global Positioning System (GPS)
- GPS determines the device's position by calculating differences in the times signals from different satellites take to reach the receiver.
- GPS is the most accurate method (between 4 and 40 meters)

Satellites contd.

- The extra hardware can be costly, consumes battery while in use, and requires some warm-up after a cold start to get an initial fix on visible satellites.
- The GPS receiver must has a clear view of the sky

Locations in Android

- Android gives your applications access to the location services supported by the device through the classes in the **`android.location`** package.

- The central component of the location framework is the **`LocationManager`** system service, which provides APIs to determine location and bearing of the underlying device (if available).

LocationManager

- This class provides access to the system location services.

- These services allow applications to obtain:
  - periodic updates of the device's geographical location, to fire an application-specified Intent when the device enters the proximity of a given geographical location.

LocationManager contd.

- To instantiate this class:

  `Context.getSystemService(Context.LOCATION_SERVICE)`

- Then, your application is able to:

  1. Query for the list of all **`LocationProviders`** for the last known user location.
  2. Register/unregister for periodic updates of the user's current location from a location provider (specified either by criteria or name).
  3. Register/unregister for a given Intent to be fired if the device comes within a given proximity (specified by radius in meters) of a given latitude and longitude

Location Providers

- The **`LocationManager`** class includes static string constants that return the provider name for the two most common Location Providers:
  - **`LocationManager.GPS_PROVIDER`**
  - **`LocationManager.NETWORK_PROVIDER`**

```
val provider: LocationProvider? =
locationManager.getProvider(LocationManager.GPS_PROVIDER)
```

Example

Get the GPS if it is enabled, or turn it on yourself

```
override fun onStart() {
    super.onStart()
    val locationManager: LocationManager = getSystemService(Context.LOCATION_SERVICE) as LocationManager
    val provider: ~~LocationProvider~~? = locationManager.~~getProvider~~(LocationManager.GPS_PROVIDER)
    val isGPSEnabled : Boolean = locationManager.isProviderEnabled(LocationManager.GPS_PROVIDER)
    if(!isGPSEnabled){
        //Build an alert dialog here that requests the user enable
        //the location service, then when the user clicks the "OK" button
        //call enable locationSettings
        enableLocationSettings()
    }
}

private fun enableLocationSettings() {
    val settingsIntent : Intent = Intent(Settings.ACTION_LOCATION_SOURCE_SETTINGS)
    startActivity(settingsIntent)
}
```

Location Providers cont'd

- To get a list of names for all the providers available on the device:

```
val enabledOnly: Boolean = true
val providers: List<String> =
locationManager.getProviders(enabledOnly)
```

Criteria

- A class indicating the application criteria for selecting a location provider.

- Providers maybe ordered according to
  - Accuracy (fine or coarse)
  - Power usage (low, medium, high),
  - Ability to report altitude
  - Speed
  - Monetary cost

Criteria contd.

- Constructors
  - **Criteria** ()
  - **Criteria** (criteria: Criteria)
- Methods
  - fun **setAccuracy** (accuracy: Int)
  - fun **setAltitudeRequired** (aRequired: Boolean)
  - fun **setCostAllowed** (costAllowed: Boolean)
  - fun **setHorizontalAccuracy** (accuracy: Int)

Finding Provider by Criteria

- To get the best provider matches your criteria:

```
val criteria: Criteria = Criteria()
criteria.accuracy = Criteria.ACCURACY_COARSE
criteria.powerRequirement = Criteria.POWER_LOW
criteria.isAltitudeRequired = false
criteria.isSpeedRequired = false
criteria.isCostAllowed = true
val bestProvider =
locationManager.getBestProvider(criteria, true)
```

Finding Provider by Criteria contd.

- If more than one Location Provider matches your criteria, the one with the greatest accuracy is returned.
- If no Location Providers meet your requirements the criteria are loosened, in the following order, until a provider is found:
  - Power use
  - Accuracy
  - Ability to return speed, and altitude

Location

- A class representing a geographic location sensed at a particular time (a "fix").
- A location consists of:
  - latitude
  - Longitude
  - optionally information on altitude, speed, and bearing.
- Get a location using:

```
locationManager.getLastKnownLocation(LocationManager.GPS_PROVIDER)
```

LocationListener Interface

- Used for receiving notifications from the **`LocationManager`** when the location has changed.

- These methods are called if the **`LocationListener`** has been registered with the location manager service using the

```
requestLocationUpdates (provider:String, minTime:Long,
minDistance: Float, listener: LocationListener)
```

LocationListener Interface contd.

- `fun onLocationChanged (location: Location)`
- `fun onProviderDisabled (provider: String)`
- `fun onProviderEnabled (provider: String)`
- `fun onStatusChanged (provider: String, status: Int, extras: Bundle)`

Permissions

- Uses-Permission is needed
  - `ACCESS_COARSE_LOCATION`
    - permission if your application uses a network-based location provider only.
  - `ACCESS_FINE_LOCATION`
    - the permission for more accurate GPS.
    - Note that declaring the `ACCESS_FINE_LOCATION` permission implies `ACCESS_COARSE_LOCATION` already

Google Play Services

- Using the Google Play services location APIs, your app can request the last known location of the user's device.
- In most cases, you are interested in the user's current location, which is usually equivalent to the last known location of the device.
- Specifically, use the fused location provider Client to retrieve the device's last known location.

Setup Google Play Services

- Ensure downloading it via the SDK Manager
- Add the library in build.gardle (App Module)

```
com.google.android.gms:play-services-location:21.1.0
```

Fused Location Provider Client

- The fused location provider client is one of the location APIs in Google Play services.
- It is the main entry point for interacting with the fused location provider.
- It manages the underlying location technology and provides a simple API so that you can specify requirements at a high level, like: **high accuracy** or **low power**.
- It also optimizes the device's use of battery power.

Create FusedLocationProviderClient

```
private lateinit var fusedClient : FusedLocationProviderClient
// ..
override fun onCreate(savedInstanceState: Bundle?) {
      // ...
    fusedClient =
LocationServices.getFusedLocationProviderClient(this);

}
```

Get Last Location

```
fun getLastLocation ( ) : Task<Location>
```

- Returns the best most recent location currently available.
- If a location is not available, null will be returned.
- The best accuracy available while respecting the location permissions will be returned.
- It is particularly well suited for applications that **do not require** an accurate location and that **do not want to maintain** extra logic for location updates.

Example

```
fusedClient.lastLocation.addOnSuccessListener(this@MainActivity, object:
OnSuccessListener<Location>{
    override fun onSuccess(location: Location?) {
        Toast.makeText(this@MainActivity, location.toString(),
Toast.LENGTH_LONG).show()
    }
})
```

Or using another form

```
fusedClient.lastLocation.addOnSuccessListener { location : Location? ->
    // Got last known location. In some rare situations this can be null.
    Toast.makeText(this@MainActivity, location.toString(),Toast.LENGTH_LONG).show()
 }
```

Request Location Updates

Requests location updates with a callback on the specified Looper thread.

```
fun  requestLocationUpdates (request : LocationRequest,
                             callback: LocationCallback,
                             looper:Looper ) : Task<Void>
```

1 2 3

```
mFusedLocationClient =
LocationServices.getFusedLocationProviderClient(this)
mFusedLocationClient.requestLocationUpdates(
        locationRequest, locationCallback,
        Looper.myLooper()
)
```

1 2 3

Set up a location request

- To determine the level of accuracy for location requests set:
  - the update interval,
  - fastest update interval, and
  - priority

LocationRequest.Builder

Request	Method
Update interval	**`setIntervalMillis(milliSeconds: Long )`** Sets the desired interval of location updates. Location updates may arrive faster than this interval (but no faster than specified by setMinUpdateIntervalMillis(long)) or slower than this interval (if the request is being throttled for example).
Fastest update interval	**`setMinUpdateIntervalMillis( )`** - Sets the fastest allowed interval of location updates. Location updates may arrive faster than the desired interval (setIntervalMillis(long)), but will never arrive faster than specified here.
Priority	**`setPriority( Priority_Constant)`** - Sets the Priority of the location request. The default value is Priority.PRIORITY_BALANCED_POWER_ACCURACY.

Priority Constants

`PRIORITY_BALANCED_POWER_ACCURACY`

- Use this setting to request location precision to within a city block, which is an accuracy of approximately 100 meters.
- This is considered a coarse level of accuracy, and is likely to consume less power.
- With this setting, the location services are likely to use **WiFi** and **cell tower positioning.**

Note, however, that the choice of location provider depends on many other factors, such as which sources are available.

Priority Constants

`PRIORITY_HIGH_ACCURACY:`

- Use this setting to request the most precise location possible.
- With this setting, the location services are more likely to use GPS to determine the location.

Priority Constants

PRIORITY_LOW_POWER

- Use this setting to request city-level precision, which is an accuracy of approximately 10 kilometers.
- This is considered a coarse level of accuracy, and is likely to consume less power.

Priority Constants

`PRIORITY_NO_POWER`

- Use this setting if you need negligible impact on power consumption, but want to receive location updates when available.
- With this setting, your app **does not trigger** any location updates, but receives locations triggered by other apps.

Example

```
1  val locationRequest = LocationRequest.Builder(0).apply {
       setPriority(Priority.LocationRequest.PRIORITY_HIGH_ACCURACY)
   }.build()
```

LocationCallBack

- Used for receiving notifications from the `FusedLocationProviderClient` when the device location has changed or can no longer be determined.

- The methods are called if the LocationCallback has been registered with the location client using the `requestLocationUpdates(LocationRequest, LocationCallback, Looper)` method.

Example

```
Private lateinit var  locationCallback : LocationCallback

2 locationCallback = object : LocationCallback() {
        override fun onLocationResult(locationResult: LocationResult)
        {
            Log.i(TAG, locationResult.lastLocation.toString())
        }
}
```

Request Location Updates

```
mFusedLocationClient =
LocationServices.getFusedLocationProviderClient(this)
mFusedLocationClient.requestLocationUpdates(
        1 locationRequest, 2 locationCallback,
        3 Looper.myLooper()
)
```

Runtime Permissions

Check If Permissions are granted:

```
fun checkPermissions(): Boolean{
    var result = false
    if ((ContextCompat.checkSelfPermission(this,
     ACCESS_COARSE_LOCATION) == PackageManager.PERMISSION_GRANTED)
    ||
       (ContextCompat.checkSelfPermission(this,
     ACCESS_FINE_LOCATION) ==  PackageManager.PERMISSION_GRANTED))
     {
        result  = true
     }
    return result
}
```

Runtime Permissions

Request Permissions:

```
private const val My_LOCATION_PERMISSION_ID = 5005
ActivityCompat.requestPermissions(this,
arrayOf(Manifest.permission.ACCESS_COARSE_LOCATION,
        Manifest.permission.ACCESS_FINE_LOCATION),
        My_LOCATION_PERMISSION_ID)
```

Runtime Permissions

Result:

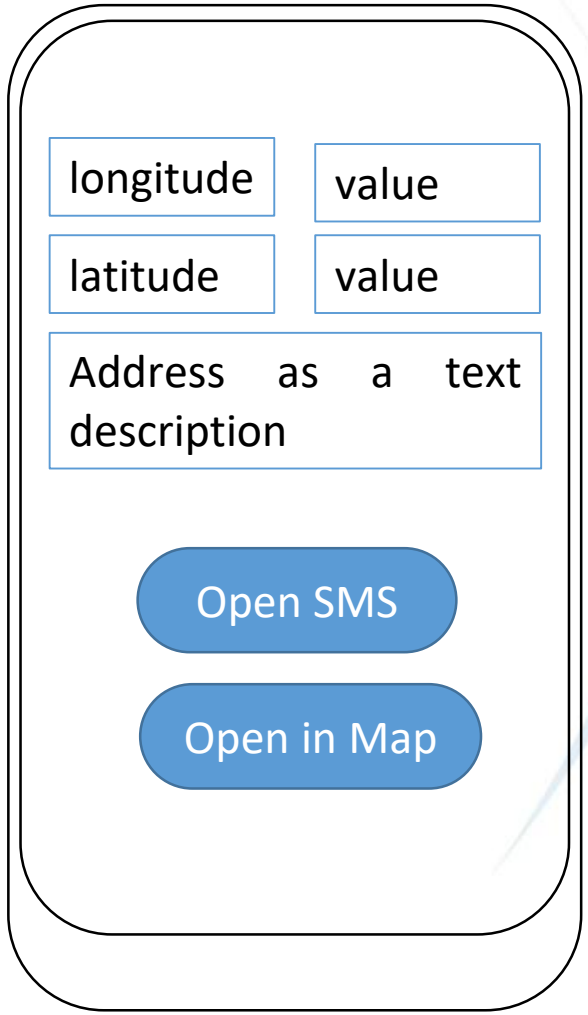
```
override fun onRequestPermissionsResult(requestCode: Int,
                                        permissions:Array<String>,
                                        grantResults: Array<Int>)
{
    super.onRequestPermissionsResult(requestCode,
                                     permissions,
                                     grantResults)

    if (requestCode == My_LOCATION_PERMISSION_ID ) {
        if ( grantResults[0]==PackageManager.PERMISSION_GRANTED){
            getLocation()
        }
    }
}
```

Demo

Lab

- Get the device GPS location
- Return the street address of that location (Reverse Geocoding)
- Open the SMS view and load the address in the message body, and add a mobile number in the receiver.
- **Bonus**: Add a button that will open a Map application and add a pin to your current location



Java™ Education and Technology Services

Services

Services

- A service is a component that runs in the background to perform long-running operations without needing to interact with the user.

- For example,
  - a service might play music in the background while the user is in a different application.
  - It might fetch data over the network without blocking user interaction with an activity.

Service States

- **Started**
  - A service is **started** when an application component, such as an activity, starts it by calling *`startService()`*
  - Once started, a service can run in the background indefinitely, even if the component that started it is destroyed.

Service States cont'd

- **Bound**
  - A service is **bound** when an application component binds to it by calling *`bindService()`*
  - A bound service offers a client-server interface that allows components to interact with the service, send requests, get results, and even do so across processes with interprocess communication (IPC).

Service Creation

- Create a Kotlin class that extends the Service base class or one of its existing subclasses.

- The **Service** base class defines various callback methods – some of them need to be overridden in your subclass.

Service Methods cont'd

- **`onCreate()`**
  - The system calls this method when the service is first created using *`startService()`* or *`bindService()`*
  - This call is required to perform one-time setup.

- **`onDestroy()`**
  - The system calls this method when the service is no longer used and is being destroyed.
  - Your service should implement this to clean up any resources such as threads, registered listeners, receivers, etc

Service Methods

- **`onStartCommand()`**
  - The system calls this method when another component, such as an activity, requests that the service be started, by calling *`startService()`*
  - If you implement this method, it is your responsibility to stop the service when its work is done, by calling *`stopSelf()`* or *`stopService()`* methods.

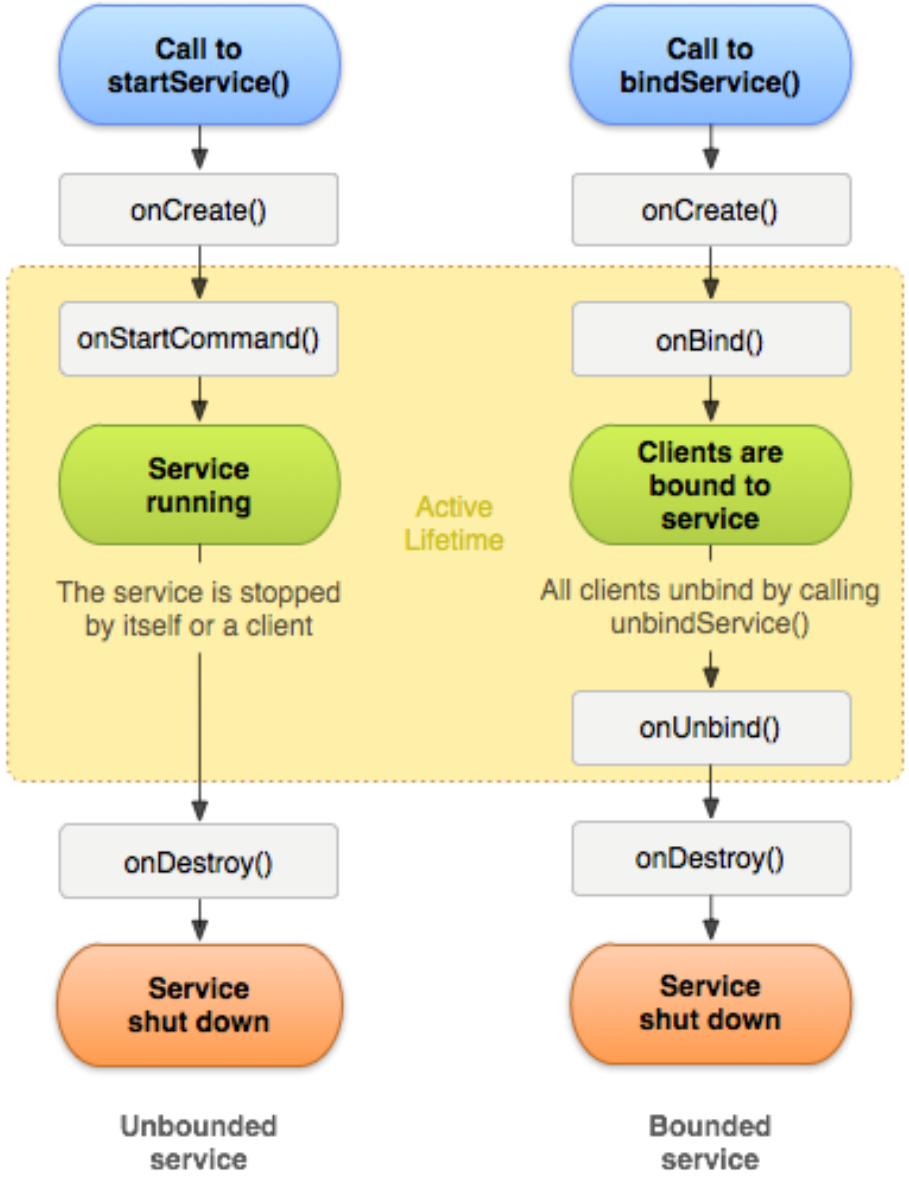
Service Methods cont'd

- **`onBind()`**
  - The system calls this method when another component wants to bind with the service by calling *`bindService()`*.
  - If you implement this method, you must provide an interface that clients use to communicate with the service, by returning an *`IBinder`* object.
  - You must always implement this method, but if you don't want to allow binding, then you should return *null*.

Service Methods cont'd

- **`onUnbind()`**
  - The system calls this method when all clients have disconnected from a particular interface published by the service. This is called by calling *`unbindService( )`*
- **`onRebind()`**
  - The system calls this method when new clients have connected to the service, after it had previously been notified that all had disconnected in its *`onUnbind(Intent)`*.

Service Lifecycle



Demo

Intent Service

- The `IntentService` class is a convenience class (subclassed from the `Service` class) that sets up a worker thread for handling background tasks and handles each request in an asynchronous manner.
- Once the service has handled all queued requests, it simply exits.
- All that is required when using the `IntentService` class is that the `onHandleIntent()` method be implemented containing the code to be executed for each request.

Example

Creating a Service Class

```
class MyIntentService (): IntentService("MyIntentService") {

      private val TAG = “ServiceExample”

      override fun onHandleIntent(intent: Intent?) {
            Log.i(TAG, “Intent Service started”)
      }

}
```

Example Cont'd

Adding Service to the Manifest

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.ServiceAndBroadCastDemo"
        tools:targetApi="31">
        <service
            android:name=".MyIntentService"
            android:exported="false"/>
        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Example Contd.

Starting the Service

```
class MainActivity : ComponentActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
          val intent = Intent(this, MyIntentService::class.java)
          startService(intent)
        }
    }
}
```

Foreground Services

- Foreground services perform operations that are noticeable to the user.
- Foreground services show a status bar notification, to make users aware that your app is performing a task in the foreground and is consuming system resources.

- To make a foreground service you should add this permission to your manifest:

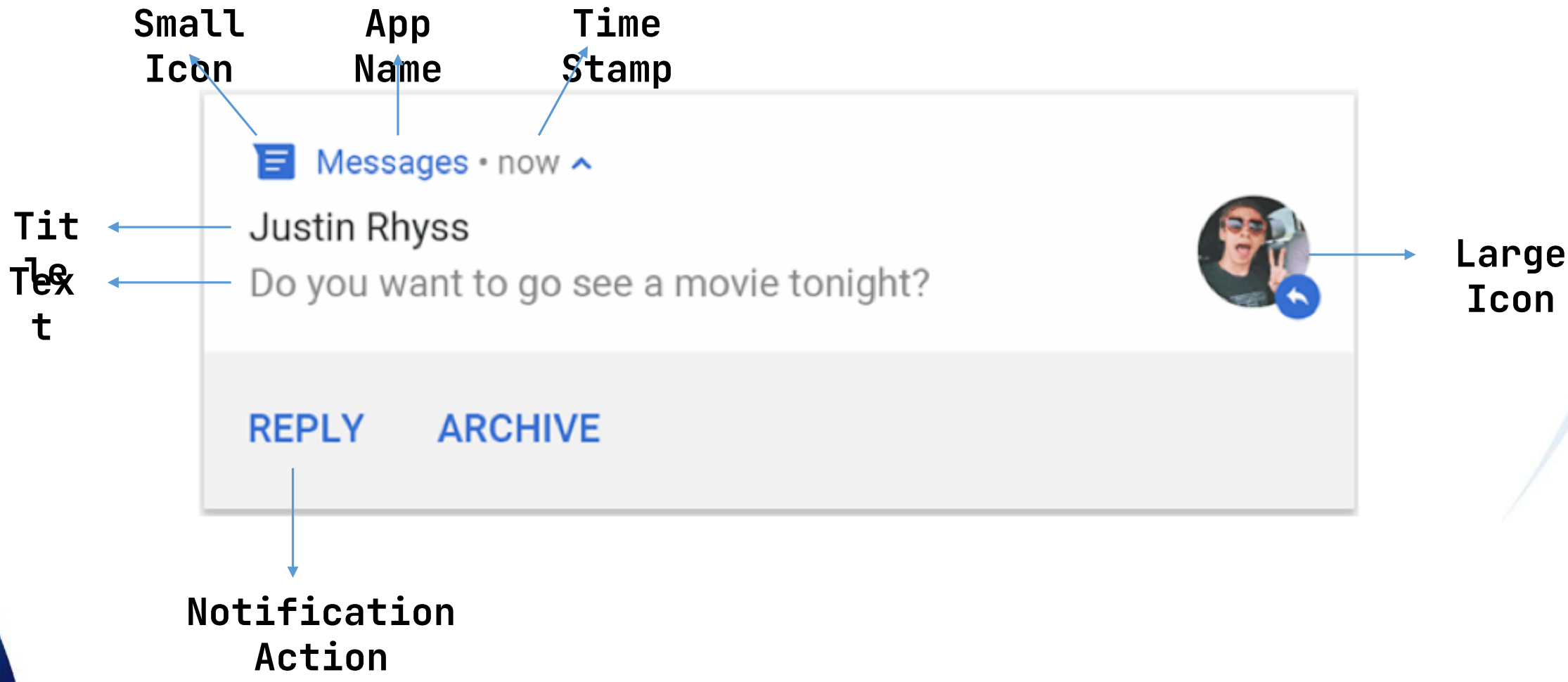
```
<uses-permission android:name="android.permission.FOREGROUND_SERVICE"/>
```

Foreground Services

- If your app targets Android 14 (API level 34) or higher, it must specify at least one foreground service type for each foreground service within your app. You should choose a foreground service type that represents your app's use case.

```
<service
            android:name=".MyForegroundService"
            android:foregroundServiceType=“shortService"
            android:exported="false">
</service>
```

Notification Anatomy



Notifications

- Set the notification's content and channel using a **`NotificationCompat.Builder`** object.
- **`setSmallIcon( )`** sets a small icon
- **`setContentTitle( )`** sets a title, .
- **`setContentText( )`** sets the body text
- **`setPriority( )`** sets the notification priority that determines how intrusive the notification should be on Android 7.1 and lower. (For Android 8.0 and higher, you must instead set the channel importance

Example – API Level 25-

```
const val CHANNEL_ID = “CHANNEL_ID”

// some code goes here
val builder: NotificationCompat.Builder =
      NotificationCompat.Builder(this, CHANNEL_ID)
         .setSmallIcon(R.drawable.notification_icon)
         .setContentTitle(textTitle)
         .setContentText(textContent)
         .setPriority(NotificationCompat.PRIORITY_DEFAULT)
```

Example – API Level 26+


```
private fun createNotificationChannel() {
    // Create the NotificationChannel, but only on API 26+ because
    // the NotificationChannel class is new and not in the support library
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        val name = getString(R.string.channel_name)
        val description: String = getString(R.string.channel_description)
        int importance = NotificationManager.IMPORTANCE_DEFAULT
        val channel = NotificationChannel(CHANNEL_ID, name, importance)
        channel.description = description
        // Register the channel with the system; you can't change the importance
        // or other notification behaviors after this
        val notificationManager = getSystemService(NotificationManager::class.java)
        notificationManager.createNotificationChannel(channel)
    }}
```

Notification's Tap Action

```
// Create an explicit intent for an Activity in your app
val intent = Intent(this, AlertDetails::class.java)
intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK or Intent.FLAG_ACTIVITY_CLEAR_TASK)
val pendingIntent = PendingIntent.getActivity(this, 0, intent,
                                              PendingIntent.FLAG_IMMUTABLE)

val builder = NotificationCompat.Builder(this, CHANNEL_ID)
        .setSmallIcon(R.drawable.notification_icon)
        .setContentTitle("My notification")
        .setContentText("Hello World!")
        .setPriority(NotificationCompat.PRIORITY_DEFAULT)
        // Set the intent that will fire when the user taps the notification
        .setContentIntent(pendingIntent)
        .setAutoCancel(true)
```

Show the notification

```
// notificationId is a unique int for each notification that you must define
notificationManager.notify(notificationId, builder.build())
```

Don’t forget:
Starting Android 13 You need to add the notification permission in your manifest and inside your code.

```
<uses-permission android:name="android.permission.POST_NOTIFICATIONS"/>
```

JobIntentService

- Helper for processing work that has been enqueued for a job/service.
- When running on Android O or later, the work will be dispatched as a job via **`JobScheduler.enqueue.`**
- When running on older versions of the platform, it will use **`Context.startService`**.
- You must publish your subclass as a JobService in your manifest for the system to interact with.

JobIntentService

10 min

- **`abstract fun onHandleWork (intent: Intent)`**
- Called serially for each work dispatched to and processed by the service.
- This method is called on a background thread, so you can do long blocking operations here.
- Upon returning, that work will be considered complete and either the next pending work dispatched here or the overall service destroyed now that it has nothing else to do.

JobIntentService

- Enqueues work for the subclass of **`JobIntentService`**.
- This will either:
  - directly start the service (when running on pre-O platforms) **or**
  - enqueue work for it as a job (when running on O and later).

JobIntentService

- In either case, a wake lock will be held for you to ensure you continue running.
- The work you enqueue will ultimately appear at **`onHandleWork(Intent)`**.

- Add this permission in the manifest file

```
<uses-permission android:name="android.permission.WAKE_LOCK"/>
```

- Inside the service tag add the following permission

```
android:permission="android.permission.BIND_JOB_SERVICE"
```

JobIntentService

Inside the JobInterntService subclass

```
companion object{
    val JOB_ID =1

    fun myEnqueueWork(context: Context, work: Intent) {
        enqueueWork(context, MyJobIntentService::class.java, JOB_ID, work)
    }
}
```

To start your JobIntentService call your created method myEnqueueWork()

```
val intent = Intent(this@MainActivity, MyJobIntentService::class.java)
MyJobIntentService.myEnqueueWork(this, intent)
```

Demo

Bound Service

- Bound services, unlike started services, provide a mechanism for implementing communication between an Android service and one or more client components.

- Bound services are created as sub-classes of the Android Service class and must, at a minimum, implement the **`onBind()`** method.

Bound Service Contd.

- Client components bind to a service via a call to the **`bindService()`** method.

- The first bind request to a bound service will result in a call to that service's **`onBind()`** method (subsequent bind request do not trigger an **`onBind()`**call).

Client – Server Interaction

- Clients wishing to bind to a service must also implement a **`ServiceConnection`** in a subclass.
- The **`ServiceConnection`** subclass should override:
  - **`onServiceConnected()`** and
  - **`onServiceDisconnected()`**
- These methods which will be called once the client-server connection has been established or disconnected respectively.

Client – Server Interaction Contd.

- In the case of the **`onServiceConnected()`** method, this will be passed an **`IBinder`** object containing the information needed by the client to interact with the service.

Client-Server Implementation

1. Subclass Service – (The Bound service)
2. Subclass Binder – within the Bound Service (As inner class)
   a) Implement a method that returns a reference to the Bound Service (reference to the outer class)
3. Inside service subclass add one or more methods to be called by the clients.
4. Create an object from your Binder subclass.
5. Let the **`onBind()`** returns the binder object.
6. Register your service in the Manifest file.

Client-Server Implementation Contd.

7. Create a class implements **`ServiceConnection`**
   - Inside `onServiceConnected`:
     a) Cast the incoming `IBinder` to your Binder subclass
     b) Get a reference to your BoundService from the binder
8. Bind the client to the service.
9. Let the client interact with the Bound Service.

Example

- Application that have a button that shows the time when it is clicked.
- The time will be gotten from a bound service and will be shown on a Text on an Activity.

BoundService – MyLocalBinder

```
class BoundService : Service() {

    private val myBinder: IBinder = MyLocalBinder()

    override fun onBind(intent: Intent): IBinder {
        return myBinder
    }
    fun getCurrentTime () : String{
        val dateFormat = SimpleDateFormat( pattern: "HH:mm:ss mm/dd/yyyy", Locale.US)
        return dateFormat.format(Date())
    }

    inner class MyLocalBinder : Binder() {
        fun getService (): BoundService{
            return this@BoundService
        }
    }
}
```

Manifest

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.KotlinBoundServiceDemo"
        tools:targetApi="31">
        <service
            android:name=".BoundService"
            android:enabled="true"
            android:exported="true"></service>

        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>

            <meta-data
                android:name="android.app.lib_name"
                android:value="" />
        </activity>
    </application>

</manifest>
```

LocalBoundActivity - ServiceConnection

```
class MainActivity : ComponentActivity() {
    private var myService =  mutableStateOf<BoundService?>( value: null)
    private var isBound by mutableStateOf( value: false)

    private val serviceConnection = object : ServiceConnection {
        override fun onServiceConnected(name: ComponentName?, service: IBinder?) {
            val binder = service as BoundService.MyLocalBinder
            myService.value = binder.getService()
            isBound = true
        }

        override fun onServiceDisconnected(name: ComponentName?) {
            isBound = false
        }
    }
```

Activity Contd.

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

    val intent = Intent( packageContext: this, BoundService::class.java)
    bindService(intent, serviceConnection, Context.BIND_AUTO_CREATE)

    setContent {
        LocalBoundScreen(myService = myService.value)
    }
}
```

Composable Screen.

```
@Composable
fun LocalBoundScreen(myService: BoundService?) {
    val currentTime = remember { mutableStateOf( value: "Press the button to get time") }
    Column(
        modifier = Modifier.fillMaxSize().padding(16.dp),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center)
    {
        Text(text = currentTime.value, fontSize = 20.sp, modifier = Modifier.padding(8.dp))
        Button(onClick = { myService?.let { currentTime.value = it.getCurrentTime() } })
        { Text( text: "Get Current Time") }
    }
}
```

Java™ Education and Technology Services

BroadcastReceiver

BroadcastReceiver

- An application listens for specific broadcast intents by registering a broadcast receiver.
- To implement receiver, extend **`BroadcastReceiver`** class and override the **`onReceive()`**
- The broadcast receiver may then be registered, either:
  - within code (for example within an activity), or
  - within a manifest file. Before Oreo

BroadcastReceiver Contd.

- Part of the registration implementation involves the creation of intent filters to indicate the specific broadcast intents the receiver is required to listen for.

- When a matching broadcast is detected, the **`onReceive()`** method of the broadcast receiver is called, at which point the method has 5 seconds within which to perform any necessary tasks before returning

BroadcastReceiver Contd.

- It is important to note that a broadcast receiver does not need to be running all the time.

- In the event that a matching intent is detected, the Android runtime system will automatically start up the broadcast receiver before calling the **`onReceive()`** method.

Broadcast Example in Kotlin

Receiver

```
class MyReceiver : BroadcastReceiver() {
    override fun onReceive(context: Context, intent: Intent) {
        // This method is called when the BroadcastReceiver
        // is receiving an Intent broadcast.
        TODO( reason: "MyReceiver.onReceive() is not implemented") }
}
```

Sender

```
fun broadcastIntent(context: Context) {
    val intent = Intent().apply { this: Intent
        action = "com.example.SendBroadCast"
        addFlags(Intent.FLAG_INCLUDE_STOPPED_PACKAGES)
    }
    context.sendBroadcast(intent)
}
```

Receiver Example

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.KotlinBroadCastRecieverDemo"
        tools:targetApi="31">
        <receiver
            android:name=".MyReceiver"
            android:enabled="true"
            android:exported="true"></receiver>

        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>

            <meta-data
                android:name="android.app.lib_name"
                android:value="" />
        </activity>
    </application>

</manifest>
```

Register / Unregister

Register

```
val intentFilter: IntentFilter = IntentFilter( action: "com.example.SendBroadCast")
receiver = MyReceiver()
registerReceiver(receiver, intentFilter)
```

Unregister

```
unregisterReceiver(receiver)
```

Note: For apps targeting **`Build.VERSION_CODES.UPSIDE_DOWN_CAKE`** (Android 14) either **`RECEIVER_EXPORTED`** or **`RECEIVER_NOT_EXPORTED`** must be specified if the receiver isn't being registered for system broadcasts or a **`SecurityException`** will be thrown.

Context in Compose

In traditional Android development, the Context is usually available and can be accessed directly within classes like Activity, Service, and sometimes in Fragment or View. However, when using Jetpack Compose, we don’t have direct access to Context within Composable functions.

Fortunately, the Jetpack Compose library provides a way to access the Context from within these Composable functions using Ambient or Providers.

Using LocalContext

To get access to the context within a Composable, you can use the **LocalContext** object. LocalContext is an ambient, which can be used to access the Context that is local to a part of the Composition.

Example…

Context in Compose

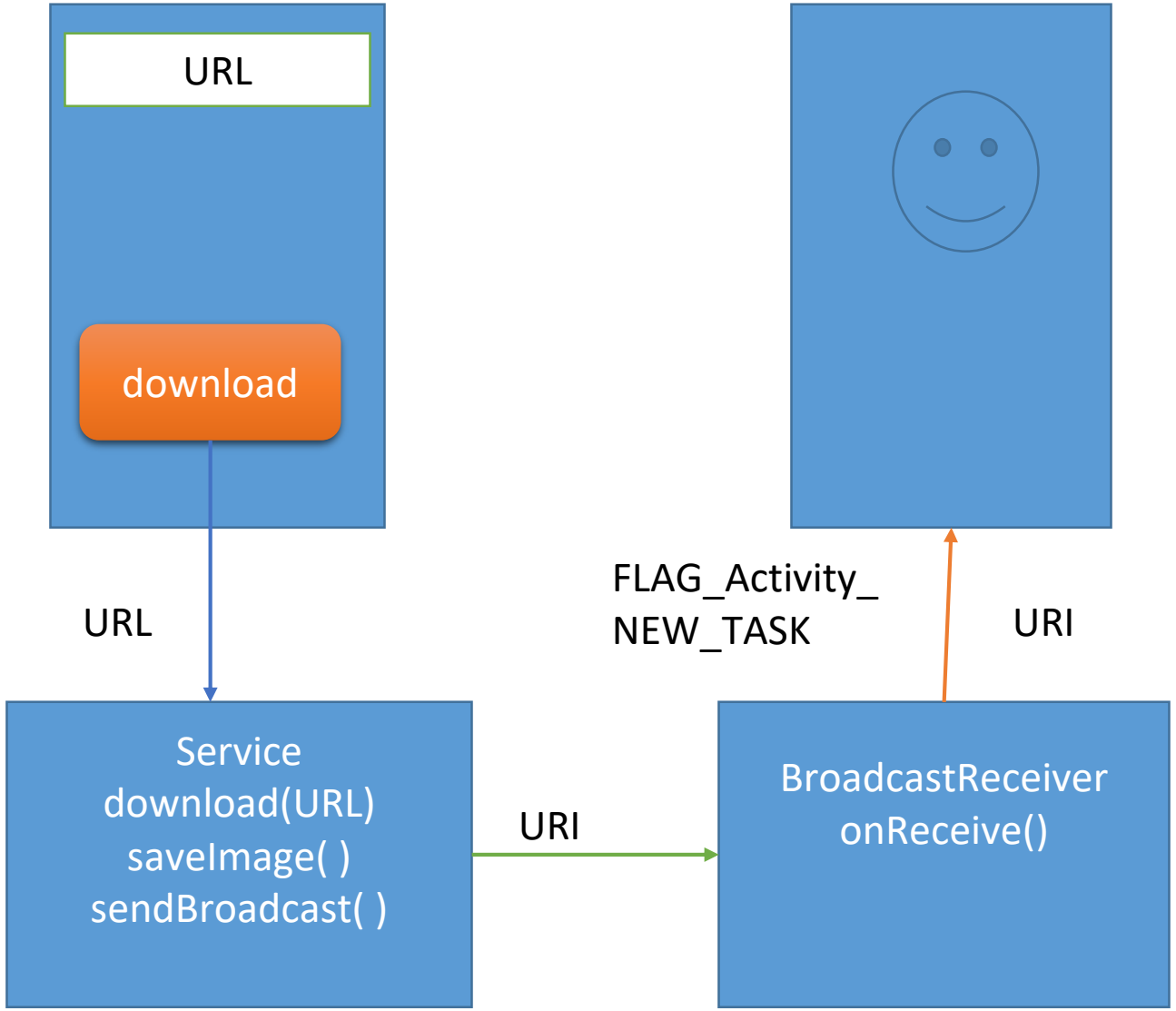
- **LocalContext.current** gives you the Context in which the Composable function is being invoked.
- Once you have the Context, you can use it as you would in traditional Android development. For example, you can use it to fetch resources, show a Toast, or start an Activity:

```
@Composable
fun MyIntent() {
    val context = LocalContext.current
    Button(onClick = {
        val intent = Intent(context, SecondActivity::class.java)
        context.startActivity(intent)
    }) {
        Text(text = "Go to Second Activity")
    }
}
```

Lab

Create a bound service that the client can interact with, in order to get date and time, then write the date and time returned on a Text

Lab

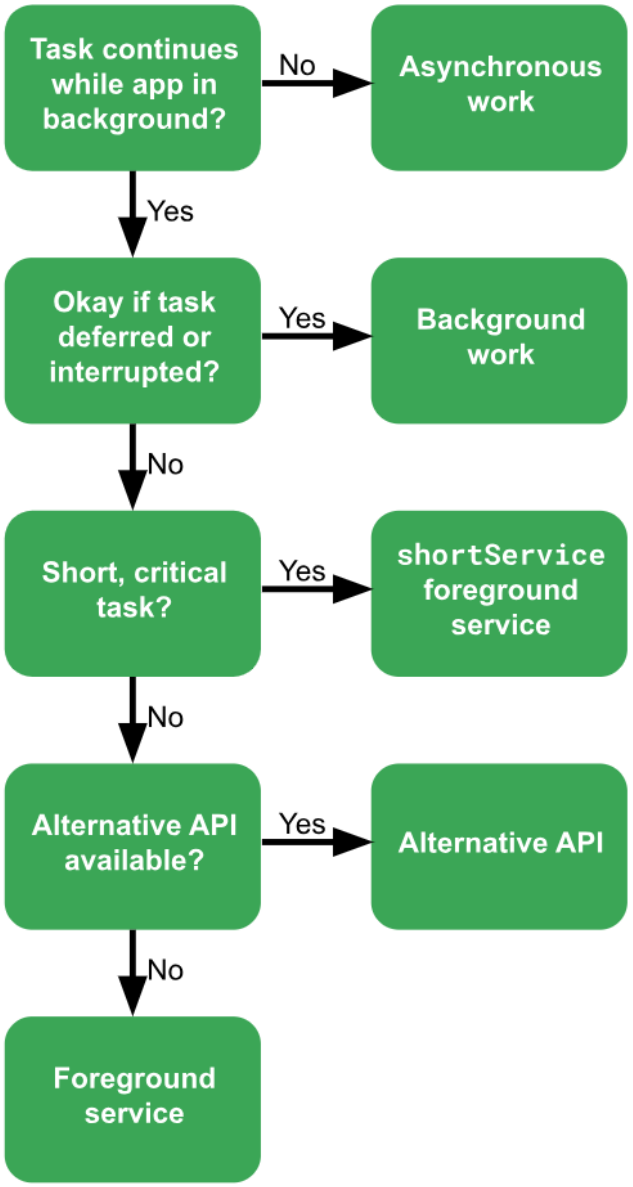

URL
download
URL
Service
download(URL)
saveImage( )
sendBroadcast( )
URI
BroadcastReceiver
onReceive()
FLAG_Activity_
NEW_TASK
URI

Java™ Education and Technology Services

WorkManager

Types of Background Work

- How to decide running your work?



WorkManager

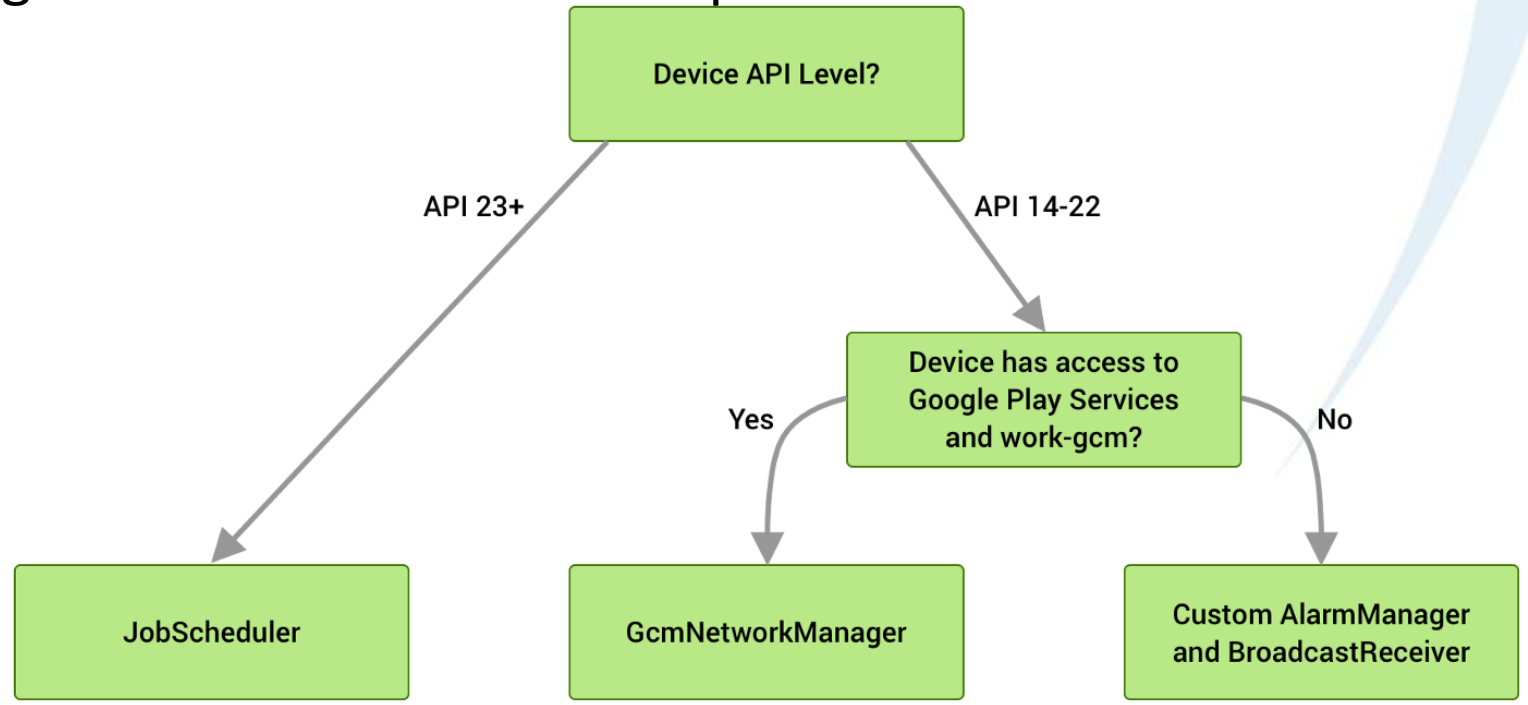
- The WorkManager, Jetpack architecture component, API makes it easy to schedule deferrable, asynchronous tasks that must be run reliably .

- These APIs let you create a task and hand it off to WorkManager to run when the work constraints are met.

- WorkManager allows observation of work status and the ability to create complex chains of work.

Work Manager

- WorkManager uses an underlying job dispatching service when available based on the following criteria:
  - Uses JobScheduler for API 23+
  - Uses a custom AlarmManager + BroadcastReceiver implementation for API 14-22



WorkManager Features

- Work constraints
- Robust scheduling
- Expedited work
- Flexible retry policy
- Work Chaining
- Built-In threading interoperability

Important classes to know

- `Worker` - does the work on a background thread, override `doWork()` method
- `WorkRequest` - request to do some work
- `WorkManager` - schedules the `WorkRequest` to be run
- `Constraint` - conditions on when the work can run

Dependencies

Inside your build.gradle module app in the dependencies section add:

```
//for Kotlin + workManager
implementation "androidx.work:work-runtime-ktx:2.7.1"
```

Define the work

- Defines the work that shall be done.
- To create some work for `WorkManager` to run, extend the Worker class and override the `doWork()` method.
- The `doWork()` method runs asynchronously on a background thread provided by `WorkManager`.
- The Result returned from `doWork()` informs the `WorkManager` service whether the work succeeded and, in the case of failure, whether or not the work should be retried.

Define the work

```
class MyWorker(appContext: Context, workerParams: WorkerParameters) :
        Worker(appContext, workerParams) {

    override fun doWork(): Result {

        // Do the work here.

        // Indicate whether work finished successfully with the Result
        return Result.success()
    }
}
```

Important classes to know

- `Worker` - does the work on a background thread, override `doWork()` method
- `WorkRequest` - request to do some work
- `WorkManager` - schedules the `WorkRequest` to be run
- `Constraint` - conditions on when the work can run

WorkRequests

- Defines how and when should the work be run.
- Can be scheduled to run once or repeatedly
  - `OneTimeWorkRequest`
  - `PeriodicWorkRequest`
- Persisted across device reboots
- Can be chained to run sequentially or in parallel
- Can have constraints under which they will run

Schedule a OneTimeWorkRequest

Create `WorkRequest`:

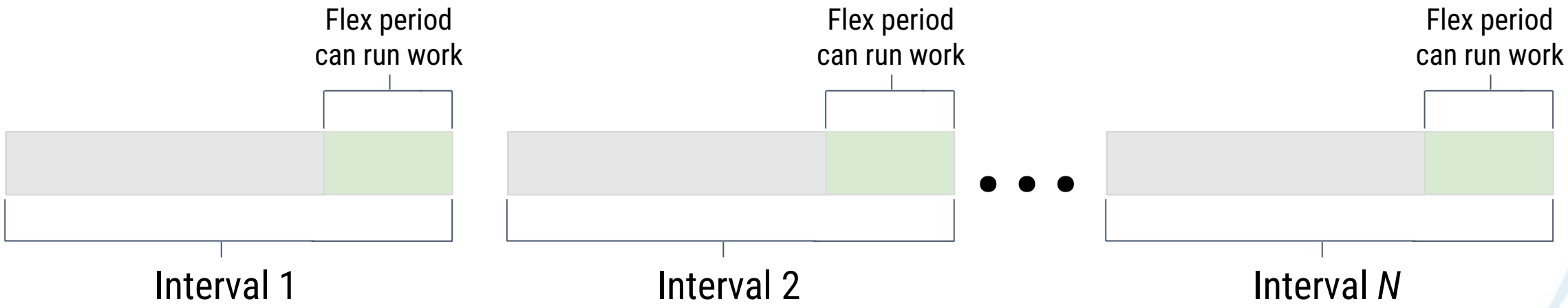
```
val myWorkRequest: WorkRequest =
    OneTimeWorkRequestBuilder<MyWorker>()
        .build()
```

Add the work to the `WorkManager` queue:

```
WorkManager.getInstance(myContext)
    .enqueue(myWorkRequest)
```

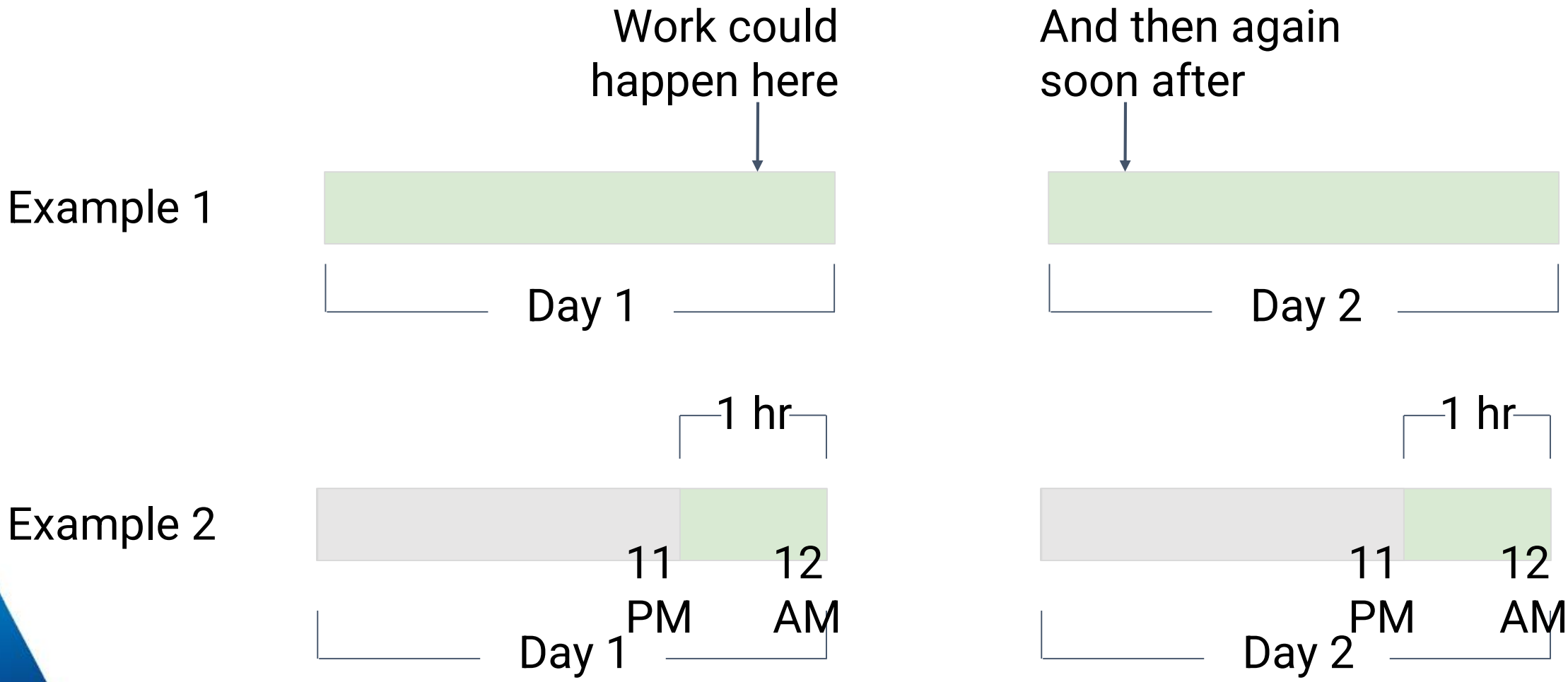
Schedule a PeriodicWorkRequest

- Set a repeat interval
- Set a flex interval (optional)



Specify an interval using `TimeUnit` (e.g., `TimeUnit.HOURS`, `TimeUnit.DAYS`)

Flex interval



PeriodicWorkRequest example

```
val repeatingRequest =
PeriodicWorkRequestBuilder<RefreshDataWorker>(
        1, TimeUnit.HOURS,    // repeatInterval
        15, TimeUnit.MINUTES  // flexInterval
    ).build()
```

Important classes to know

- `Worker` - does the work on a background thread, override `doWork()` method
- `WorkRequest` - request to do some work
- `WorkManager` - schedules the `WorkRequest` to be run
- `Constraint` - conditions on when the work can run

Java™ Education and Technology Services

Work input and output

Send input data to Worker

```
val complexMathWork = OneTimeWorkRequestBuilder<MathWorker>()
    .setInputData(
        workDataOf(
            "X_ARG" to 42,
            "Y_ARG" to 421,
        )
    ).build()

WorkManager.getInstance(myContext).enqueue(complexMathWork)
```

Send input data to Worker

```
val inputData = Data.Builder()
inputData.putLong("MyKey", 100L)
inputData.putString("STR_KEY", "STR_VALUE")
inputData.build()

val complexMathWork = OneTimeWorkRequestBuilder<MathWorker>()
    .setInputData(inputData)
    .build()

WorkManager.getInstance(myContext).enqueue(complexMathWork)
```

Define Worker with input and output

```
class MathWorker(context: Context, params: WorkerParameters):
      Worker(context, params)  {

    override fun doWork(): Result {
        val x = inputData.getInt(KEY_X_ARG, 0)
        val y = inputData.getInt(KEY_Y_ARG, 0)
        val result = computeMathFunction(x, y)
        val output: Data = workDataOf(KEY_RESULT to result)
        return Result.success(output)
    }
}
```

Result output from doWork()

Result status	Result status with output
`Result.success()`	`Result.success(output)`
`Result.failure()`	`Result.failure(output)`
`Result.retry()`	

Receiving Data from Worker

- To get extra information about work request we can use **`WorkInfo`** class
- **`WorkInfo:`** is a class that represents information about a particular WorkRequest containing the id of the WorkRequest, its current State, output, tags, and run attempt count.
- Note that output is only available for the terminal states (State.SUCCEEDED and State.FAILED).

Receiving Data from Worker

```
var request = OneTimeWorkRequestBuilder<MySimpleWorker>()
    .setInputData(data.build())
    .build()

WorkManager.getInstance(this).enqueue(request)


val infos: WorkInfo =
     WorkManager.getInstance(this)
     .getWorkInfoById(request.id).get()

val result = infos.outputData.getString(Constants.OUTPUT)
Log.i(TAG, "onCreate: $result")
```

WorkRequest constraints

Constraints

Constraints ensure that work is deferred until optimal conditions are met.

- `setRequiredNetworkType`
- `setRequiresBatteryNotLow`
- `setRequiresCharging`
- `setRequiresStorageNotLow`
- `requiresDeviceIdle`

Constraints example

```
val constraints = Constraints.Builder()
    .setRequiredNetworkType(NetworkType.CONNECTED)
    .setRequiresCharging(true)
    .setRequiresBatteryNotLow(true)
    .setRequiresDeviceIdle(true)
    .build()

val myWorkRequest: WorkRequest = OneTimeWorkRequestBuilder<MyWork>()
    .setConstraints(constraints)
    .build()
```

Important classes to know

- `Worker` - does the work on a background thread, override `doWork()` method
- `WorkRequest` - request to do some work
- `WorkManager` - schedules the `WorkRequest` to be run
- `Constraint` - conditions on when the work can run

Chaining Work

- WorkManager allows you to create and enqueue a chain of work that specifies multiple dependent tasks and defines what order they should run in.
- To create a chain of work, you can use

1. `WorkManager.beginWith(OneTimeWorkRequest)`
2. `WorkManager.beginWith(List<OneTimeWorkRequest>)`

- After that you can call

1. `then(OneTimeWorkRequest)`
2. `then(List<OneTimeWorkRequest>)`

For more info follow the link here

Unique Work

- Unique work is a powerful concept that guarantees that you only have one instance of work with a particular *name* at a time.
- Unlike tags, unique names are only associated with a single instance of work.
- Unique work can be applied to both one-time and periodic work by calling:

```
WorkManager.enqueueUniqueWork() //for one time work

WorkManager.enqueueUniquePeriodicWork() //for periodic work
```

Unique Work Cont'd

- Both `enqueueUniqueWork()` and `enqueueUniquePeriodicWork()` have 3 parameters:

1. **uniqueWorkName** - A String used to uniquely identify the work request.
2. **existingWorkPolicy** - An enum which tells WorkManager what to do if there's already an unfinished chain of work with that unique name.
3. **work** - the WorkRequest to schedule.

```
val request =OneTimeWorkRequestBuilder<MyWorker>().build()
WorkManager.getInstance(this).enqueueUniqueWork(
            "sendLogs", ExistingWorkPolicy.KEEP, request)
```

Retry Policy

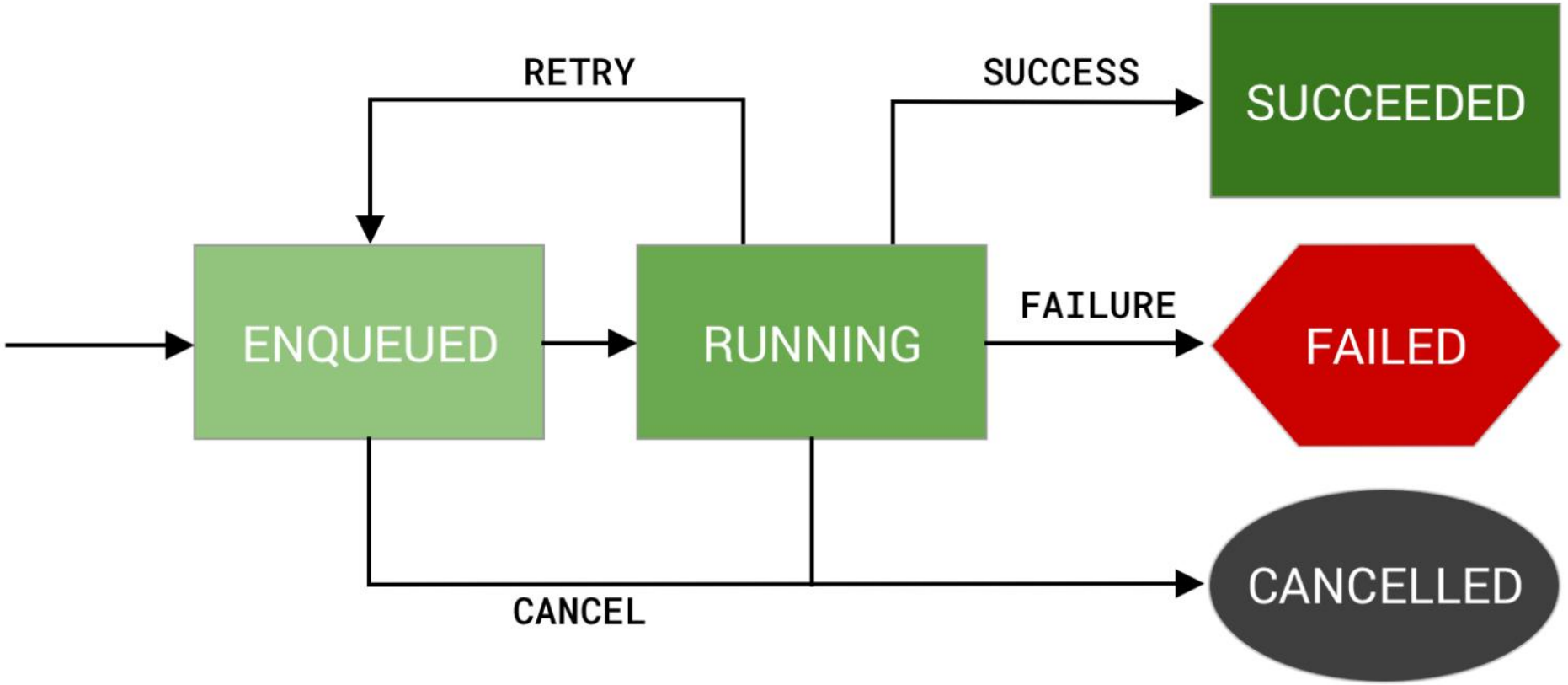
- As mentioned `WorkManager` supports retry, from `doWork()` you can return `Result.retry()`

- When we ask for retry we should define the retry policy at your request by adding :

```
setBackoffCriteria(BackoffPolicy.LINEAR,
                                10L,TimeUnit.MINUTES)
```

- Inside your worker you can check the retry count using *`runAttemptCount`*, and so you can determine when to stop retrying

Observe WorkManager Status



Observe WorkManager Status

- At any point after enqueuing work, you can check its status by querying WorkManager by its name, id or by a tag associated with it.
- Using these three methods will output a livedata that you can observe and act based on its result:

1. `workManager.getWorkInfoByIdLiveData(syncWorker.id)`
2. `workManager.getWorkInfoByTagLiveData("sync-tag")`

Lab

- Using your first products app. Connect to the API: https://dummyjson.com/products and get the products via one-time-request then output the result. Observe the work status , once it finishes, start populating your recyclerview with the incoming data