Lab on apps development for tablets, smartphones and smartwatches

Week 4: Life cycles and Wear APIs

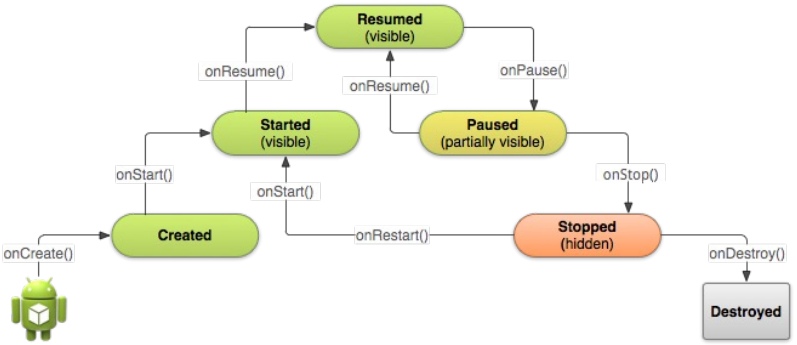
Giovanni Ansaloni

Dimitra Tatli, Riselda Kodra, Yuxuan Wang
Qunyou Liu, Amirhossein Shahbazinia, Christodoulos Kechris

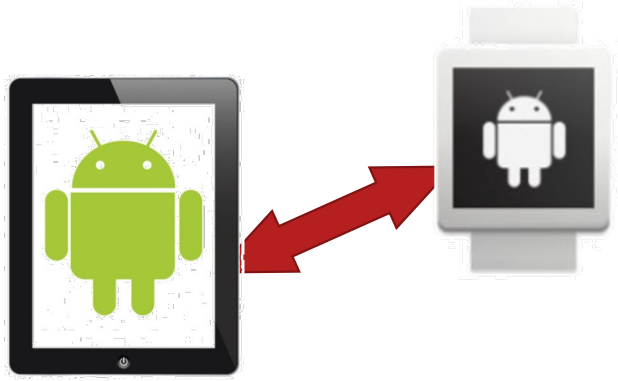
School of Engineering (STI) – Institute of Electrical and Micro Engineering (IEM)

Class outline

- **Activities lifecycles**
  - lifecycle callbacks
  - configuration changes

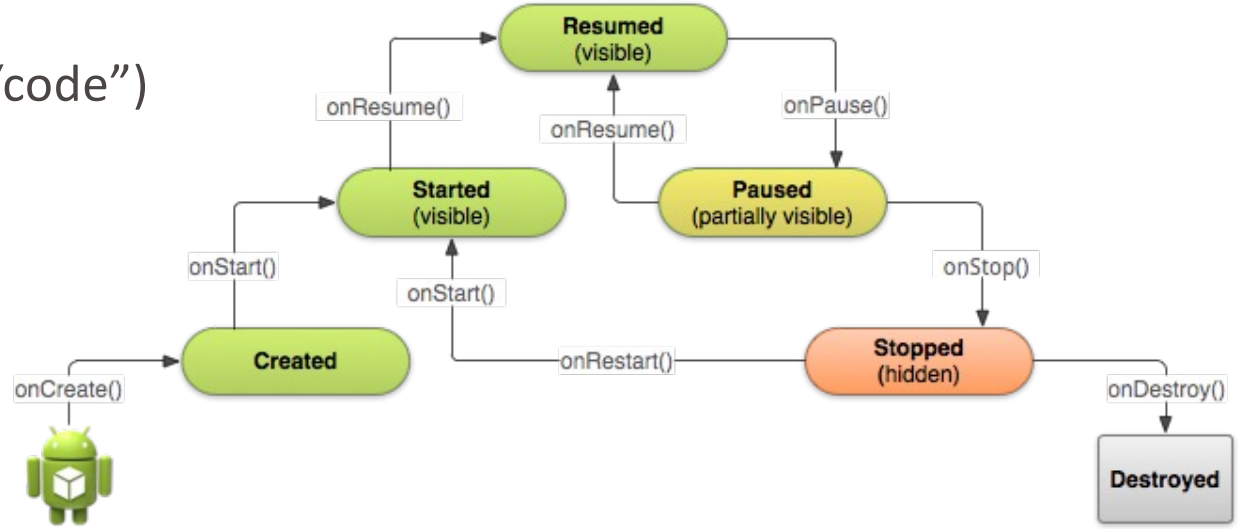


- Communicating between tablet and watch
  - Data API
  - Message API



Activity lifecycle

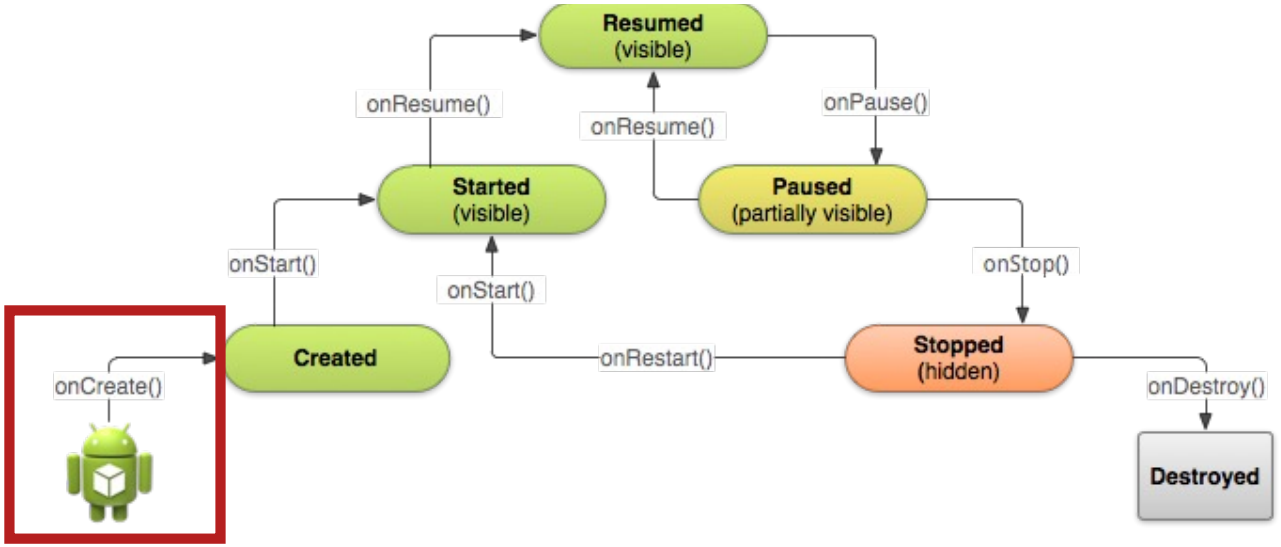
- What is the activity lifecycle?
  - A set of states the Activity can be in during its lifetime, from its creation to its destruction
- More formally:
  - A directed graph of all the states an activity can be in, and the callbacks associated with transitioning from each state to the next one
- We can add some functionality ("code") to the transition functions



Activity lifecycle: onCreate()

- onCreate()
  - Called when the Activity is created
    - for example, we tap launcher icon
  - Does all initial setup:
    - data binding, action bar, …
  - Only called once during lifetime
  - Should contain the initialization operations at the Activity level
  - Takes a Bundle with all the Activity previous state, if it exists
  - If successful, the activity is created but not visible

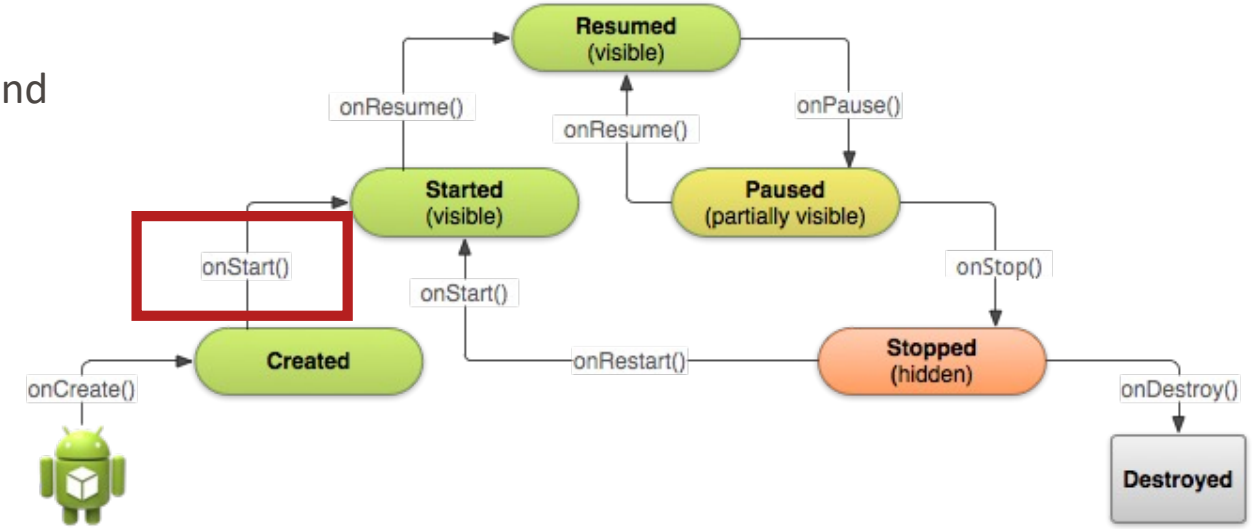
```
class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        ...    //data binding
        ...    //initialization
    }
```



Activity lifecycle: onStart()

- onStart()
  - Called when onCreate() terminates
    - Or when the activity was stopped and restarts
  - Called right before activity is visible to user
  - Followed by onResume(),
    if the Activity comes to the foreground

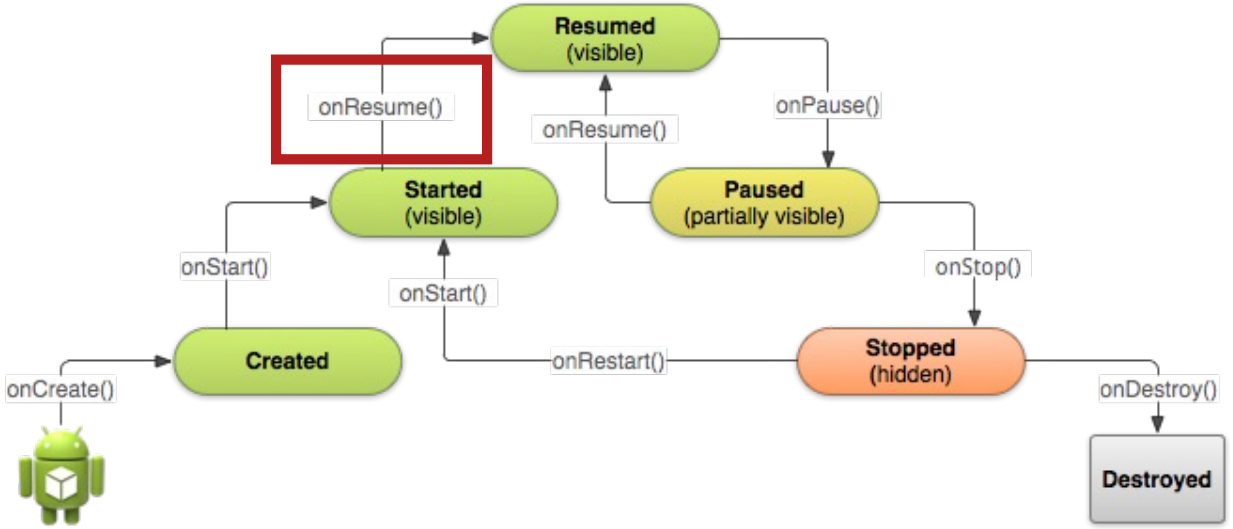
```
override fun onStart() {
    super.onStart()
    //Activity about to become visible
}
```



Activity lifecycle: onResume()

- onResume()
  - Called when the Activity is ready to get input from users
  - After onResume() successfully terminates, the Activity is **running and visible**
  - Always followed by onPause()

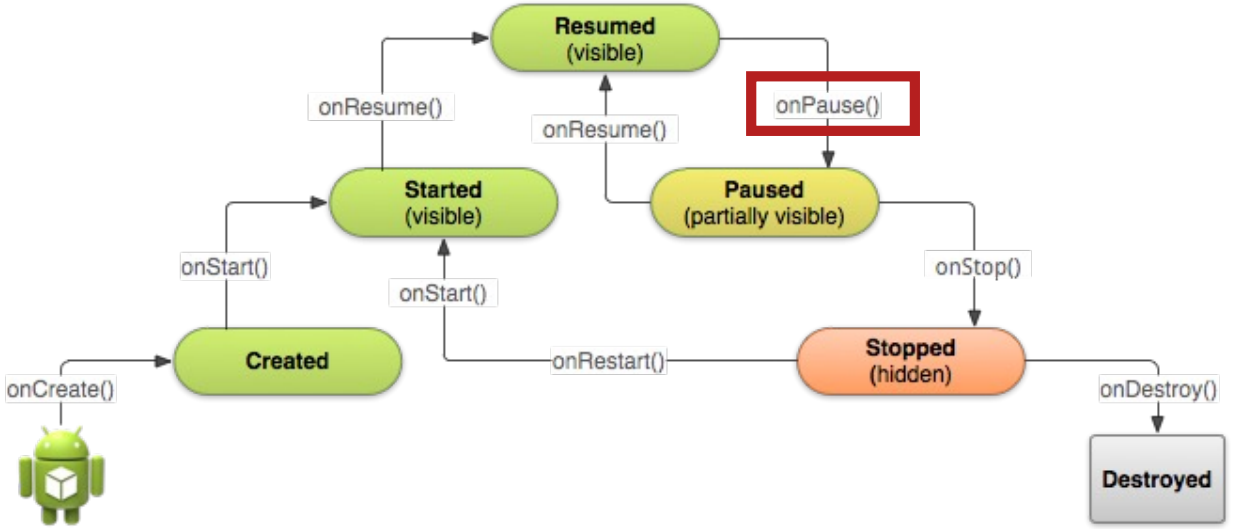
```
override fun onResume() {
    super.onResume()
    //Activity ready to interact with user
}
```



Activity lifecycle: onPause()

```
override fun onPause() {
    super.onPause()
    //Another activity is coming
    //on the foreground
}
```

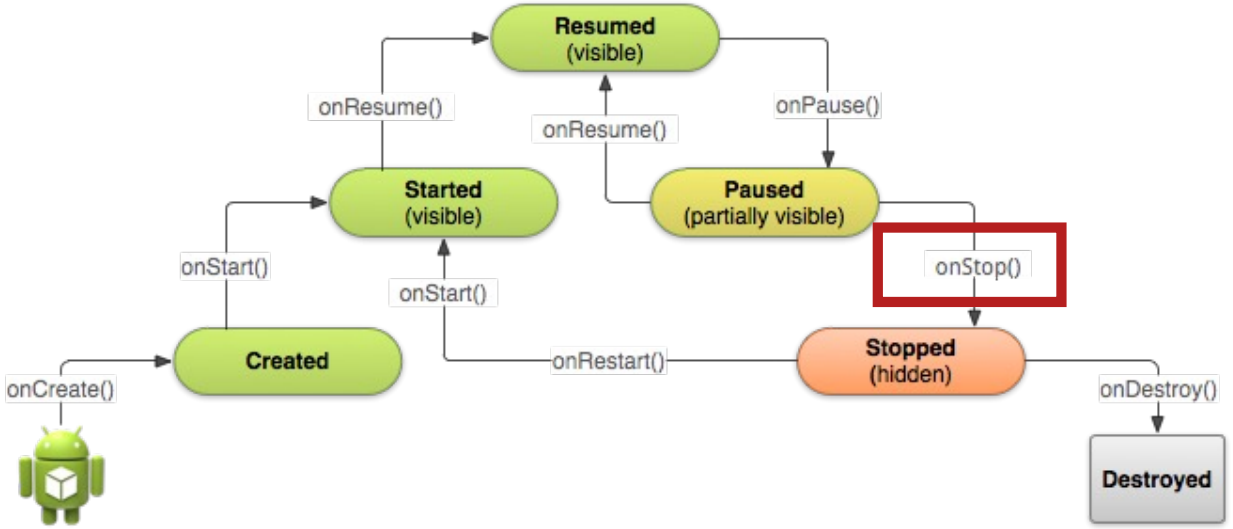
- onPause()
  - Called when another Activity comes to the foreground
  - Used e.g. to release listeners to non-GUI events
  - Should be fast, as the other Activity cannot resume until this method finishes
  - Followed by either onResume() or onStop()



Activity lifecycle: onStop()

- onStop()
  - Activity is no longer visible to the user
  - Could be called because:
    - the Activity is about to be destroyed
    - another Activity completely overlays the current one
  - Followed by either onRestart() if we are going to interact with user, or onDestroy() if it is going away

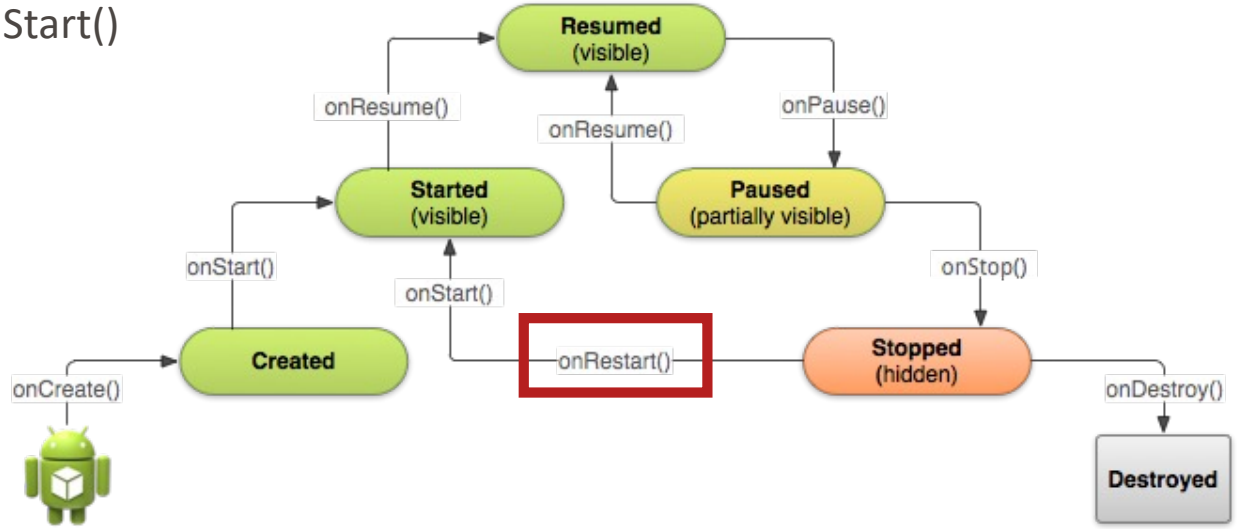
```
override fun onStop() {
    super.onStop()
    //Activity is now stopped
}
```



Activity lifecycle: onRestart()

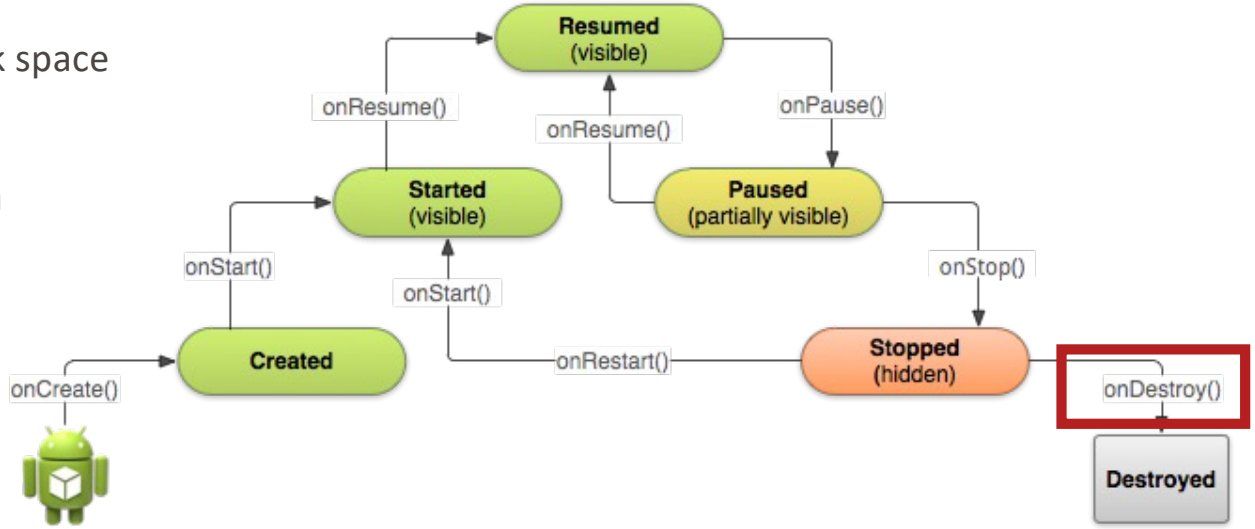
- onRestart()
  - Similar to onCreate()
  - Called after Activity has been stopped, immediately before it is started again
  - Transient state, always followed by onStart()

```
override fun onRestart() {
    super.onRestart()
    //Activity between stop and start
}
```



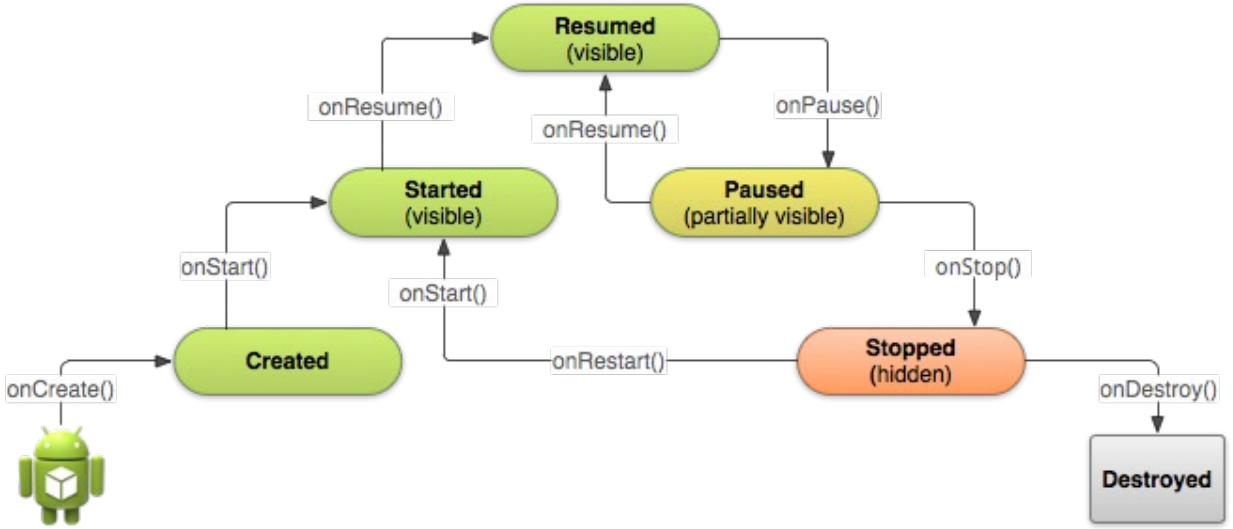
Activity lifecycle: onDestroy()

- onDestroy()
  - The Activity is about to be destroyed (final call before destruction)
  - Could happen because:
    - Configuration changes
    - finish() method is called
    - The Android system need some stack space
  - The system may destroy Activities without calling this function
    - Save data on onPause() or onStop()



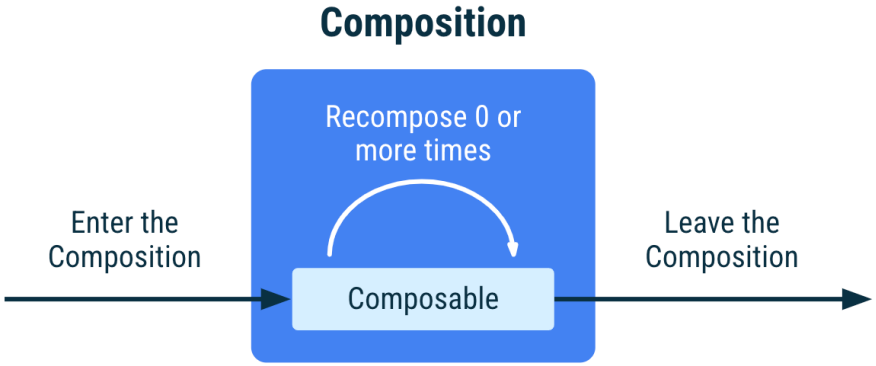
Activity lifecycle loops

- **Entire lifetime**
  - Between onCreate() and onDestroy()
  - Setup of global state in onCreate()
  - Release remaining resources in onDestroy()
- **Visible lifetime**
  - Between onStart() and onStop()
  - Maintain resources that have to be shown to the user
- **Foreground lifetime**
  - Between onResume() and onPause()
  - Code should be light

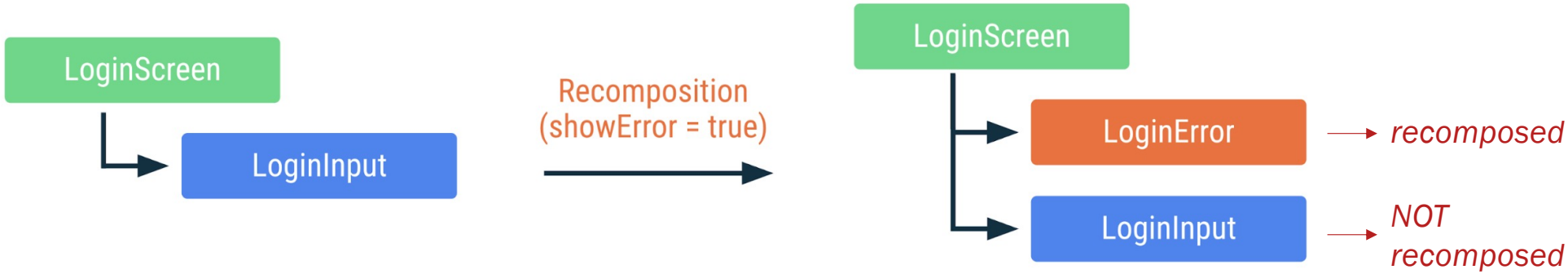


Composable Lifecycle

- Composables also have a lifecycle

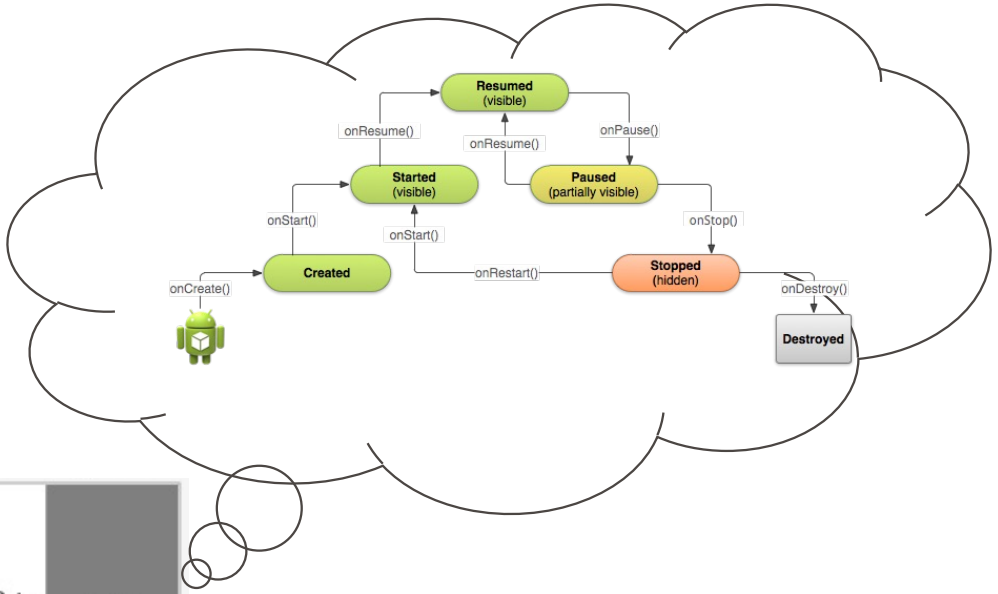


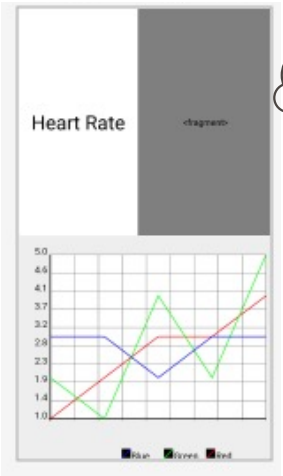
- Only composable functions affected by recomposition are called



Lifecycle-aware Composable

- Composable can be made Lifecycle-aware
  - Capture and react to Activity-level lifecycle events
- Examples
  - Enable/Disable sensors
  - Notify devices





Lifecycle-aware Composable

- All activity lifecycle events can be captured by composables with LifecycleEventEffect
  - specific lifecycle event as a parameter

```
@Composable
fun HomeScreen( ) {
   LifecycleEventEffect(Lifecycle.Event.ON_RESUME) {
     //do something when activity resumes
   }
   // …
}
```

- Specialized functions: LifecycleResumeEffect / LifecycleStartEffect
  - require the implementation of the dual lifecycle callback (onPause / onStop)

```
@Composable
fun HomeScreen(0) {
  LifecycleResumeEffect {
     // add ON_RESUME code here
     onPauseOrDispose {
        // add ON_PAUSE code here
     }
}
```

Configuration changes

- Configuration changes invalidate the current layout or other resources
  - When does configuration change?
    - Rotates the device
    - Chooses different system language (e.g. English to French)

- On a configuration change, AndroidOS:
  1. Shuts down activity calling: onPause() → onStop() → onDestroy()
  2. Then starts it over calling: onCreate() → onStart() → onResume()

- State is lost during a configuration change!

Saving Activity state

- Composable state data can be saved with rememberSaveable → Lecture2

```
@Composable
fun Counter(modifier: Modifier = Modifier) {
   var count by rememberSaveable { mutableStateOf(0) }
   …
}
```

- For Activity data, implement onSaveInstanceState()
  - called after onStop()

```
override fun onSaveInstanceState(outState: Bundle) {
    super.onSaveInstanceState(outState)
    //Add information for saving something to the outState bundle
    outState.putInt("oneStoredValue",myValue )
}
```

...or (better) implement ViewModels → *next lecture*

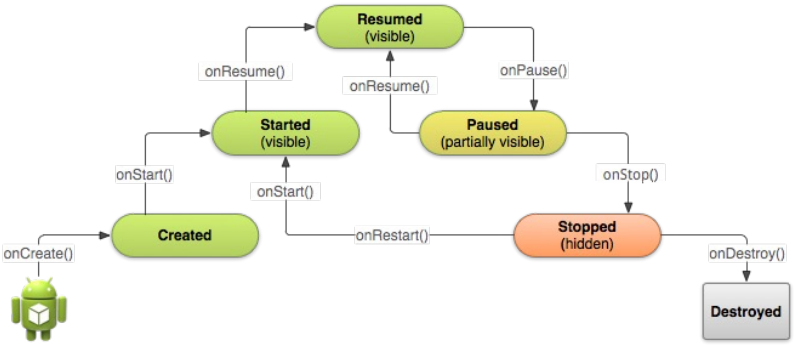
Restoring Activity State

- Retrieving the saved data from the onCreate() argument
  - bundle will return saved values, or null if not present

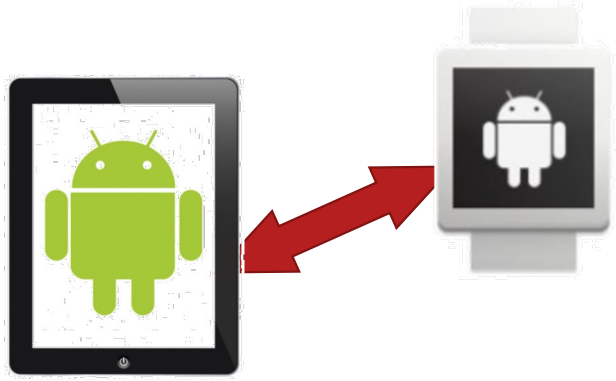
```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    myValue = savedInstanceState?.getInt("oneStoredValue") ?: 0
    ...
}
```

Class outline

- Activities lifecycles
  - lifecycle callbacks
  - configuration changes



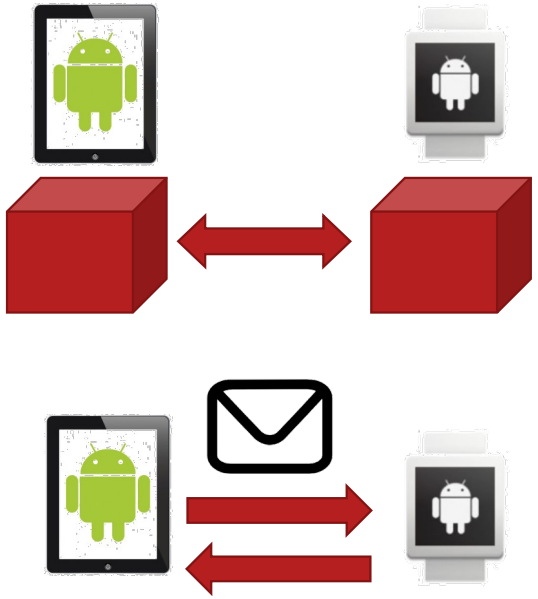
- **Communicating between tablet and watch**
  - Data API
  - Message API



Communicating between tablet and watch

- Communication enabled by WearOS among connected/paired nodes
  - WearOS app must be installed on tablet/smartphone
  - Tablets/smartphone paired with smartwatch via WearOS app
  - For development, tablets/smartphone and smartwatch must be compiled from the same laptop/desktop
  - More details here:
    - https://support.google.com/wearos/answer/6056630?hl=en&co=GENIE.Platform%3DAndroid

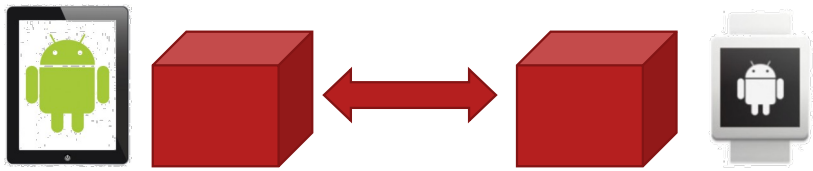
Android Wear API



- Two communication modalities:
  - *Data*
    - From one node to **all connected nodes**
    - Synchronizes data, similar to shared memory
    - Good for structured data
    - Synchronization event queue
  - *Messages*
    - From **one node to another**
    - Good for one-way requests
    - Best effort
- Available by updating the projects' gradle

```
dependencies {
    ...
    implementation 'com.google.android.gms:play-services-wearable:17.0.0'
}
```

Data API



- Data API provide storage with automatic synchronization
  - Data API requests stored in a queue
  - Objects can be incapsulated to send binary data (e.g., images)
- On the sending side, dataClient is used to interface with the Data API
- On the receiving side, OnDataChangedListener() callbacks are executed when receiving data

Initiating a Data synchronization

- Declare and initialize the dataClient

```
class MainActivity : ComponentActivity() {
   private lateinit var dataClient: DataClient
   ...
   override fun onCreate(...) {
      ...
      dataClient = Wearable.getDataClient(this)
      ...}
}
```

- Define the Data for syncing
  - Payload → what is being sent
  - Path → unique string (starting with a forward slash)

- Send the Data to the destination

Sending Data with Data API

1. Data Items are implemented using a PutDataMapRequest object and its embedded Datamap
   - .create() → path
   - .datamap → payload
   - .setUrgent() → send as soon as possible

```
private var COUNT_KEY: String = "COUNT_KEY"
private lateinit var dataClient: DataClient

private fun increaseCounter() {
    val putDataMapRequest: PutDataMapRequest = PutDataMapRequest.create("/count")

1   putDataMapRequest.dataMap.putInt(COUNT_KEY, count)
    putDataMapRequest.setUrgent()

    val countDataRequest: PutDataRequest =  putDataMapRequest.asPutDataRequest()
    dataClient.putDataItem(countDataRequest)
}
```

Sending Data with Data API

2. Data Items are sent as a PutDataRequest
   - using dataClient

```
private var COUNT_KEY: String = "COUNT_KEY"
private lateinit var dataClient: DataClient

private fun increaseCounter() {
    val putDataMapRequest: PutDataMapRequest = PutDataMapRequest.create("/count")

    putDataMapRequest.dataMap.putInt(COUNT_KEY, count)
    putDataMapRequest.setUrgent()

2   val countDataRequest: PutDataRequest =  putDataMapRequest.asPutDataRequest()
    dataClient.putDataItem(countDataRequest)
}
```

Receiving Data with Data API

- The other side of the data connection is notified of data changes by implementing a listener

1. Implement the DataClient.OnDataChangedListener interface

```
class MainActivity : Activity(), DataClient.OnDataChangedListener{
```

2. registering / unregistering listener
   - in onResume() and onPause()

```
override fun onResume() {
    super.onResume()
    Wearable.getDataClient(this).addListener(this)
}

override fun onPause() {
    super.onPause()
    Wearable.getDataClient(this).removeListener(this)
}
```

Receiving Data with Data API

3. Provide the onDataChanged() callback

look at pending events

check path

retrieve payload

```
override fun onDataChanged(dataEvents: DataEventBuffer) {
    dataEvents.forEach { event ->
        // DataItem changed
        if (event.type == DataEvent.TYPE_CHANGED) {
            val recItem = event.dataItem
            if (recItem.uri.path?.compareTo("/count") == 0) {
                val count =
                  DataMapItem.fromDataItem(recItem).dataMap.getInt(COUNT_KEY)
                updateCount(count) //private fun doing something with "count"
            }
        } else if (event.type == DataEvent.TYPE_DELETED) {
            // DataItem deleted
        }
    }
}
```

Exchange Assets with Data API

- used to exchange large amounts of data
  https://developer.android.com/training/wearables/data-layer/assets
  - As before, attach the Asset to the dataMap and send

```
val resId =  R.drawable.<someImageResource>
val myAsset =
        createAssetFromBitmap(BitmapFactory.decodeResource(resources, resId))
...
putDataMapRequest.dataMap.putAsset(ASSET_KEY, myAsset)
...
val putDataRequest: PutDataRequest =  putDataMapRequest.asPutDataRequest()
dataClient.putDataItem(putDataRequest)
```

- Convert image to bytes

```
private fun createAssetFromBitmap(bitmap: Bitmap) : Asset{
    val myStream = ByteArrayOutputStream()
    bitmap.compress(Bitmap.CompressFormat.PNG, 100, myStream)
    return Asset.createFromBytes(myStream.toByteArray())
}
```

Receiving Assets

- Use getAsset() to retrieve the Asset from the dataMap

```
override fun onDataChanged(dataEvents: DataEventBuffer) {
    dataEvents.forEach { event ->
        if (event.type == DataEvent.TYPE_CHANGED) {
            val recItem = event.dataItem
            if (recItem.uri.path?.compareTo("/asset_data") == 0) {
                val myAsset =
                    DataMapItem.fromDataItem(recItem).dataMap.getAsset(COUNT_KEY)
                loadBitmapFromAsset(myAsset)
                ...
```

- Retrieve the data stream and convert it to the proper format
  - e.g.: bitmap image

```
fun loadBitmapFromAsset(asset: Asset): Bitmap? {
    val assetInputStream: InputStream? =
            Tasks.await(Wearable.getDataClient(this).getFdForAsset(asset))
                    ?.inputStream
    return BitmapFactory.decodeStream(assetInputStream)
}
```

Messages

- Messages implement a one-way communication
  - sent immediately
  - small payload
- Sender gets a list of all connected nodes

```
private fun getNodes(): Collection<String> {
    return
Tasks.await(Wearable.getNodeClient(requireActivity()).connectedNodes).map { it.id }
}
```

  - Restricting the list of nodes to specific capabilities
    https://developer.android.com/training/wearables/data-layer/messages
- Receiver implements the Message API interface

```
class MainActivity : Activity(), MessageClient.OnMessageReceivedListener{
```

Messages

- Sending a message to nodes

```
for (nodeId in getNodes()){
    Wearable.getMessageClient(requireActivity()).sendMessage(
        nodeId,
        MESSAGE_PATH,
        message.toByteArray())
}
```

node → nodeId

path → MESSAGE_PATH

payload → message.toByteArray()

Messages

- Receiving messages in onMessageReceived() callback

```
override fun onMessageReceived(messageEvent: MessageEvent) {
    if (messageEvent.path == MESSAGE_PATH) {
        receivedMessageString = messageEvent.data.toString()
    }
}
```

- Receiver registers/unregisters listener in onResume(), onPause()

```
override fun onResume() {
    super.onResume()
    Wearable.getMessageClient(this).addListener(this)
}

override fun onPause() {
    super.onPause()
    Wearable.getMessageClient(this).addListener(this)
}
```

Today's Lab

- Share login information between tablet and watch using Wear API
- Share login credentials among Screens using NavHost arguments on tablet app

Questions?