Laboratorio di Applicazioni Mobili
Bachelor in Computer Science &
Computer Science for Management

University of Bologna

Background Operations

Federico Montori
federico.montori2@unibo.it

Table of Contents

- Notifications
- Multithreading
    - Message Passing
    - Coroutines
- Services
    - Intent Services
    - Bound Services
- Broadcast Receivers

Short Recap

TILL NOW: Android Application structured has a single Activity or as a group of Activities

- **Intents** to call other activities
- **Layout** and **Views** to setup the GUI
- **Events** to manage the interactions with the user

Activities executed only in foreground …

- What about *background activities*?
- What about *multi-threading* functionalities?
- What about *external events* handling?

Short Recap

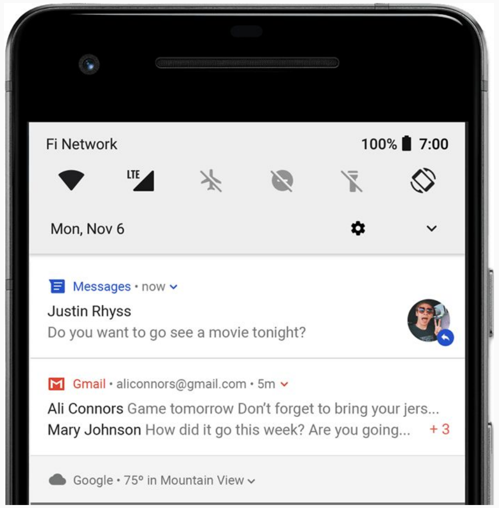
Example: What can we do for an Instant Messaging (IM) application?

- Setup of the application GUI ✅
- GUI event management ✅
- Application Menu and Preferences ✅
- Updates in background mode ❎
- Notifications of message reception in background mode ❎

Notifications

Notifications are messages from your application

- Reminders
- External events
- Timely information



Fi Network

100% 7:00

Mon, Nov 6

Messages • now

Justin Rhyss

Do you want to go see a movie tonight?

Gmail • aliconnors@gmail.com • 5m

Ali Connors Game tomorrow Don't forget to bring your jers...

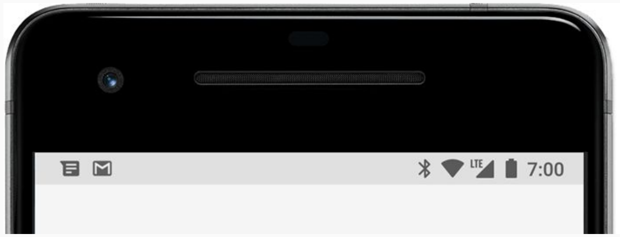
Mary Johnson How did it go this week? Are you going... + 3

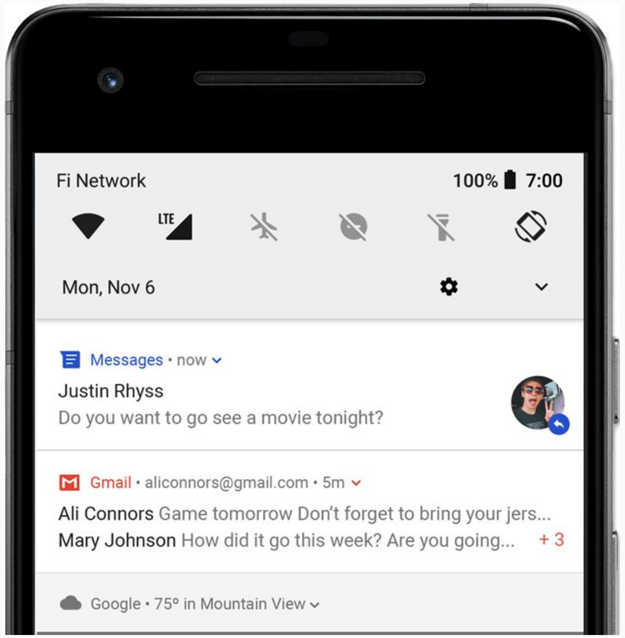
Google • 75° in Mountain View

Can serve 2 cases:

- Only informative: a message is displayed to the user
- Informative and active: by clicking on it, it is possible to open the APP or perform directly some operations

Notifications





Messages • now
Justin Rhyss
Do you want to go see a movie tonight?

Gmail • aliconnors@gmail.com • 5m
Ali Connors Game tomorrow Don't forget to bring your jers...
Mary Johnson How did it go this week? Are you going... + 3

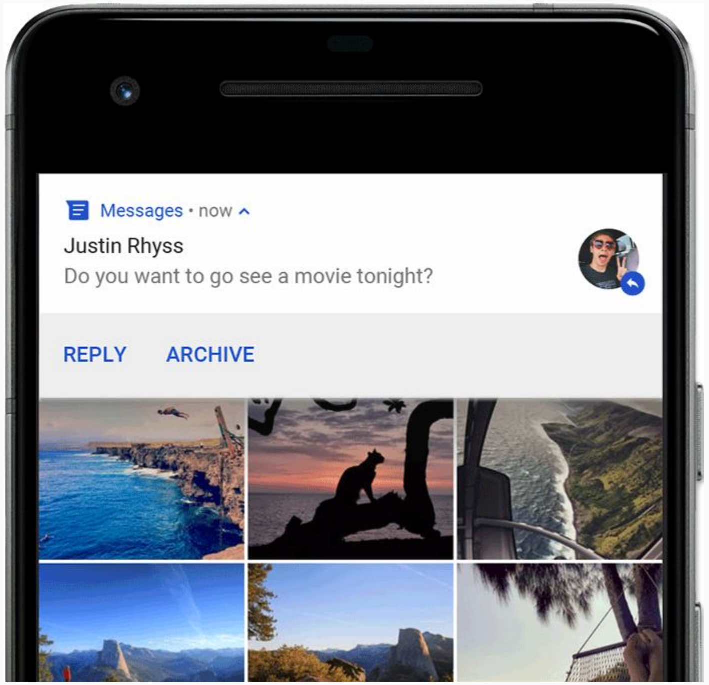
Google • 75° in Mountain View

When the notification is created, its icon appears in the status bar

Scrolling down the status bar reveals additional details about the notification

Some notification can also reveal further information by swiping them down

Notifications



Messages • now

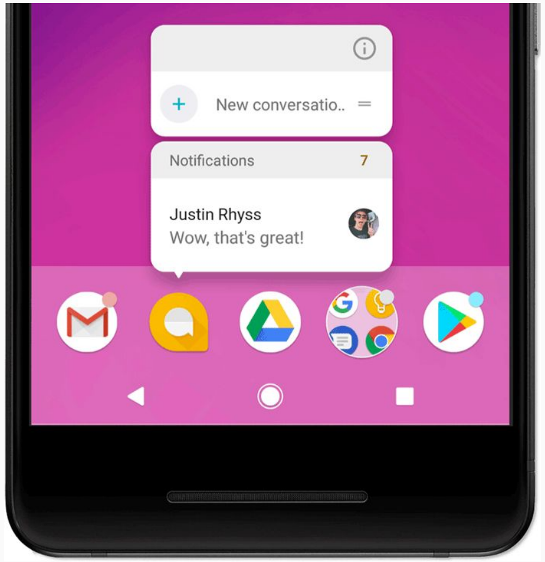
Justin Rhyss

Do you want to go see a movie tonight?

Heads up notifications: useful for important information, and to notify the user while watching a full screen activity (starting from 5.0) while providing direct actions.

Notifications can also be visible in the lock screen. Developers can configure the amount of visible details.

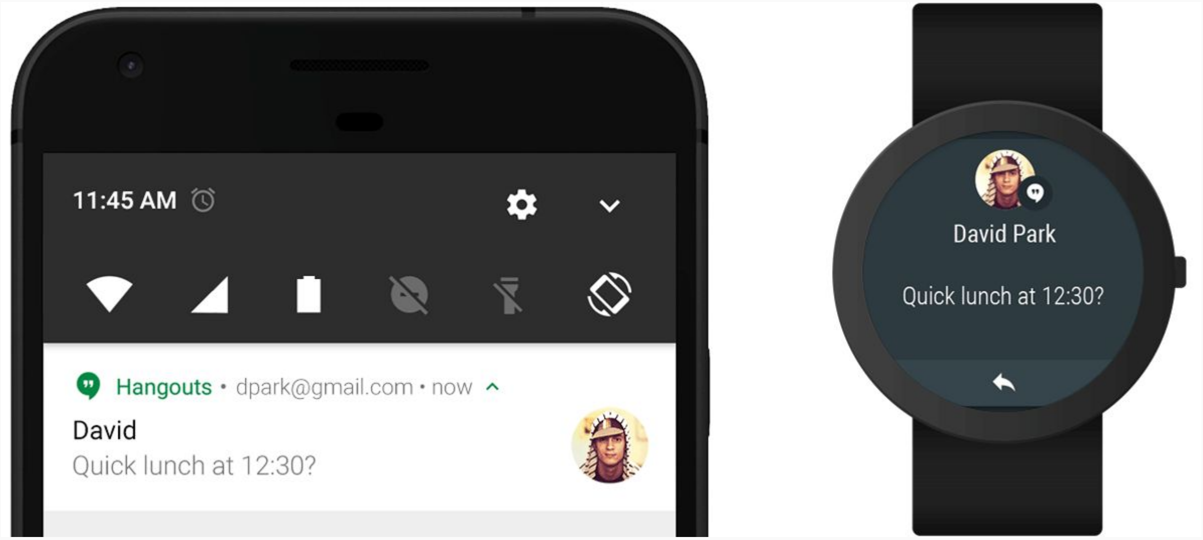
Notifications



Justin Rhyss
Wow, that's great!

Icon badge: starting with Android 8.0. Users can get notification information about an app.

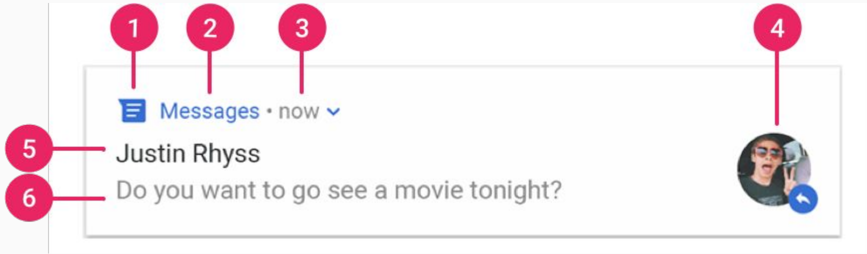
Wearables, to show the same notification on the handheld device and wearable.



David
Quick lunch at 12:30?

David Park
Quick lunch at 12:30?

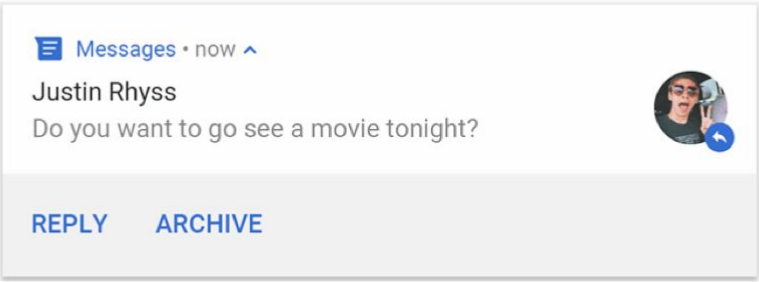
Notifications



Messages • now

Justin Rhyss

Do you want to go see a movie tonight?



Messages • now

Justin Rhyss

Do you want to go see a movie tonight?

1. Small icon
2. App name
3. Timestamp
4. Optional Large Icon
5. Optional Title
6. Optional Text

Starting with Android 7.0, users can perform simple actions directly in the Notification

Notifications

Notifications can also be updated

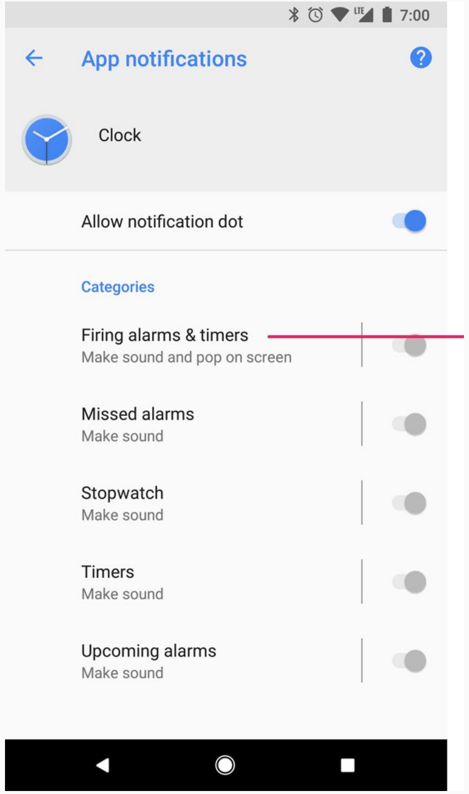
- Notifications should be updated if they refer to the same content that has just changed

If more than one notification is needed for the same app, they can be grouped together

- Starting with Android 7.0

Starting with Android 8.0

- Notification MUST also set a channel
  - To let users have more control about which kind of notification they want to see
  - Can control them through system settings
- Channels have also an associated priority



7:00

App notifications

Clock

Allow notification dot

Categories

Firing alarms & timers
Make sound and pop on screen

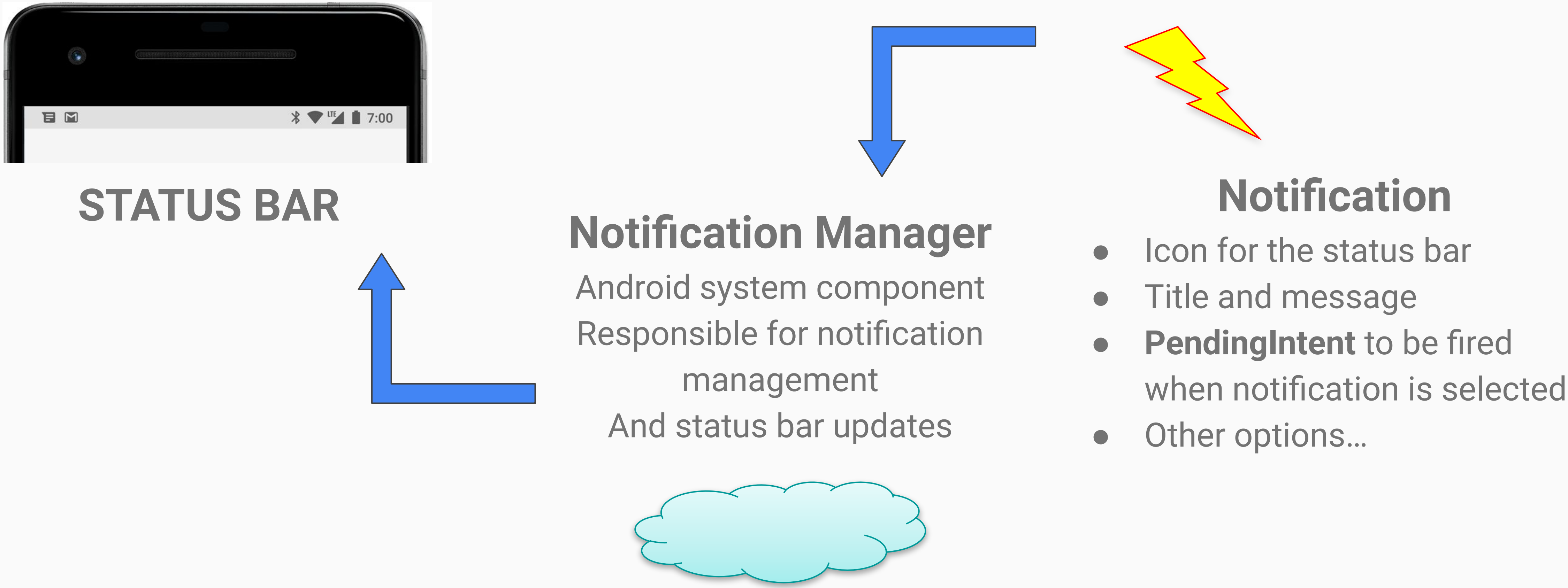
Missed alarms
Make sound

Stopwatch
Make sound

Timers
Make sound

Upcoming alarms
Make sound

Notifications



STATUS BAR

Notification Manager

Android system component
Responsible for notification management
And status bar updates

Notification

- Icon for the status bar
- Title and message
- **PendingIntent** to be fired when notification is selected
- Other options…

Notifications

These are the to send a notification:

Step 1: Get a reference to the **NotificationManager**

```
val notificationManager = getSystemService(Context.NOTIFICATION_SERVICE) as NotificationManager
```

or better:

```
val notificationManager = NotificationManagerCompat.from(this)
```

This is a **System Service** which we have to invoke to tell the operating system that we are doing things that may affect the world outside our app.

Notifications

Step 2: Create a notification channel (since Android 8)

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
    val CHANNEL_ID = "My Channel Id"
    val importance = NotificationManager.IMPORTANCE_DEFAULT
    val channel = NotificationChannel(CHANNEL_ID, "MyChannelName", importance)
    channel.description = "My description"
    val notificationManager = NotificationManagerCompat.from(this)
    notificationManager.createNotificationChannel(channel)
}
```

Notification channels are mandatory since API 26, for lower versions running the app, the channel will just be ignored.

Notifications

Step 3: Build the notification message (design pattern **Builder**)

```
val builder = NotificationCompat.Builder(this, CHANNEL_ID)
        .setSmallIcon(androidx.core.R.drawable.notification_bg)
        .setContentTitle("Remember that you will die!")
        .setContentText("Let me explain a number of reasons why this is the case, blah, blah, blah...")
        .setPriority(NotificationCompat.PRIORITY_DEFAULT)
```

ignored if API version < 26

Step 4: Commit the building process and fire the notification.

```
val myNotficationId = 0
notificationManager.notify(myNotficationId, builder.build())
```

set by the developer, used for later modification

Notifications

What happens if the user taps on the notification?

Define a **Pending Intent** (a container for an intent to be fired by someone else).

```
val newIntent: Intent = Intent(this, MainActivity.javaClass)
newIntent.flags = Intent.FLAG_ACTIVITY_NEW_TASK or Intent.FLAG_ACTIVITY_CLEAR_TASK
newIntent.putExtra("caller", "notification")
val pendingIntent: PendingIntent = PendingIntent.getActivity(
        this, 0, newIntent, PendingIntent.FLAG_IMMUTABLE
) // getActivity is just like startActivity for instantaneous Intents
```

requestCode, set by the developer,

Then add it to your notification builder...

```
builder.setContentIntent(pendingIntent)
```

Notifications

What happens if the user taps on the notification?

Define a **Pending Intent** (a container for an intent to be fired by someone else).

```
val newIntent: Intent = Intent(this, MainActivity.javaClass)
newIntent.flags = Intent.FLAG_ACTIVITY_NEW_TASK or Intent.FLAG_ACTIVITY_CLEAR_TASK
newIntent.putExtra("caller", "notification")
val pendingIntent: PendingIntent = PendingIntent.getActivity(
        this, 0, newIntent, PendingIntent.FLAG_IMMUTABLE
) // getActivity is just like startActivity for instantaneous Intents
```

Or add it as a button!

- A maximum of three buttons can be added, or media controls…
  - For more information and possibilities go to https://developer.android.com/training/notify-user/build-notification

```
builder.addAction(androidx.core.R.drawable.notification_action_background, "PRESS ME", pendingIntent)
```

Notifications

There is a whole world about notifications and ever-evolving ways to build them (e.g. grouping, media, progress bars, in-notification reply, …). For a complete course: https://developer.android.com/guide/topics/ui/notifiers/notifications

It is although very important to know and implement some best practices:

- The Notification UI, once built, runs on a **different system thread** held by a RemoteView object.
- Building a notification may be long and could block the UI. It’s always better to do it on a worker thread (see later).
- Don’t tease the user with too many notifications...

Multithreading

IMPORTANT:

"In general, any task that takes more than a few milliseconds should be delegated to a background thread. Common long-running tasks include things like decoding a bitmap, accessing storage, working on a machine learning (ML) model, or performing network requests."

Multithreading

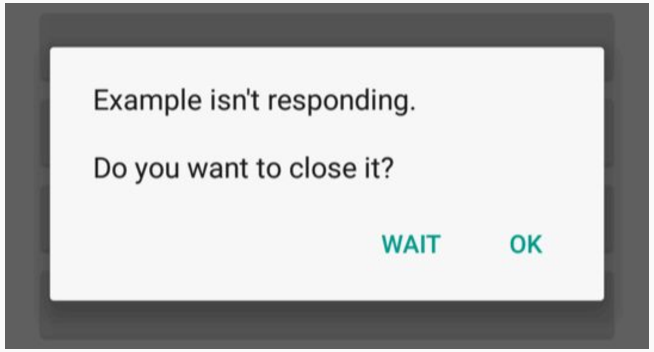
By default, all components of the same application run in the same process and thread (called “**Main Thread**” or “**UI Thread**”).

- In *Manifest.xml*, it is possible to specify the process in which a component (activity, service, receiver, provider) should run through the attribute android:process.
- Processes might be killed by the system to reclaim memory.
  - Processes’ hierarchy to decide the importance of a process.
  - Five types: Foreground, Visible, Service, Background, Empty.
  - more at: https://developer.android.com/guide/components/activities/process-lifecycle

Multithreading

By default, all components of the same application run in the same process and thread (called “main thread” or “UI” thread).

- In certain rare cases they do not correspond (only in context of some system applications)
- Main Thread is responsible for drawing stuff, queuing events and calling their callbacks functions …
- Sometimes this may yield poor performances when performing other operations (database transactions, networking…) and freezes the UI
  - If the UI freezes for more than 5 secs it will be very very unpleasant



Example isn't responding.

Do you want to close it?

Multithreading

Android natively supports a multi-threading environment.

An Android application can be composed of multiple concurrent threads.

How to create a thread in Android?

- Threads and Runnables { what really happens under the hood }
- Coroutines (Kotlin only) { what is more convenient to use }

We also need to manage callbacks and/or allow message passing

Multithreading - Threads

Let us start with legacy Java Threads (here used in Kotlin).

```
Thread(
        Runnable {
                // Do your stuff... for example:
                var counter = 1000
                while (counter > 0) {
                        Thread.sleep(10)
                        counter = counter - 1
                }
        }).start()
```

Threads implement a Runnable, a SAM interface that specifies a behavior in the method **run()**.

This piece of code executes the body within a separate thread.

Multithreading - Threads

A thread pool is a managed collection of threads that runs tasks in parallel from a queue. New tasks are executed on existing threads as those threads become idle.

- Be sure to instantiate the pool only once in your application.

```
val executorService : ExecutorService = Executors.newFixedThreadPool(4)
```

- An ExecutorService (or an Executor implementing it) takes in input a Runnable

```
executorService.execute {
      // Do your stuff…
}
```

Multithreading - Threads

The UI or main thread is in charge of dispatching events to the user interface widgets, and of drawing the elements of the UI.

- Do not block the UI thread.
- Do not access the Android UI components from outside the UI thread.

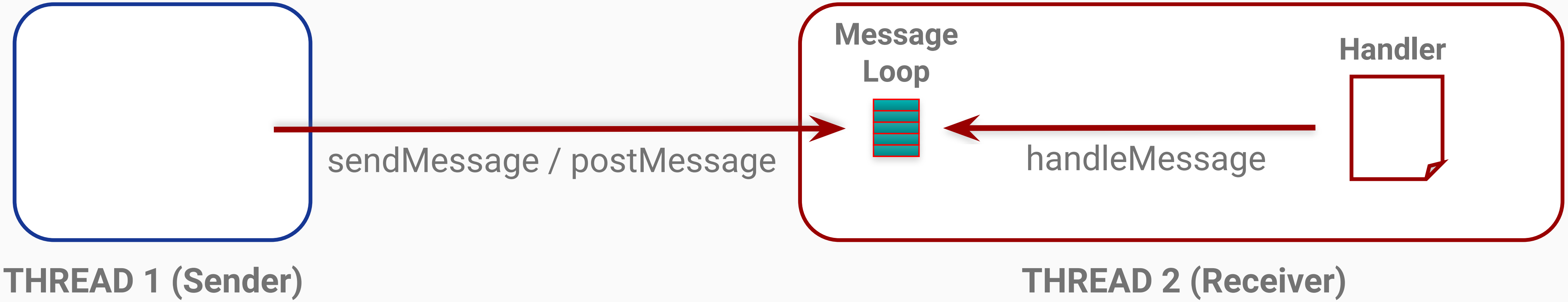
QUESTION: How to update the UI components from worker threads?
Threads need to communicate!

Message passing

Multithreading - Message Passing

Message Passing like mechanisms for Thread communication in OS.

Message Loop → Queue of messages associated to a thread.
Handler → Object that processes incoming messages within a thread.
Message → Parcelable Object that can be sent/received by a thread.



Multithreading - Message Passing

Receiver Side: use Looper and Handler objects

```
Runnable {
        Looper.prepare() // Instantiate the message queue
        val handler : Handler =
                Handler(Looper.myLooper()!!) { msg ->
                        // Handle here the message
                }
        Looper.loop() // Have it ready for receiving
}
```

```
val handler : Handler = Handler(Looper.myLooper()!!)
```

Threads not always have loopers by default (use HandlerThread)

If the receiver does not know what to do upon receiving, but still wants to receive messages, then the handler can be empty.

Multithreading - Message Passing

Receiver Side: use Looper and Handler objects

```
class LooperThread() : Thread("Custom Thread") {
    lateinit var handler: Handler
    override fun run() {
        Looper.prepare() // Initialize the message queue
        handler = object : Handler(Looper.myLooper()!!) {
            override fun handleMessage(msg: Message){
            // Handle the message
            }
        }
        Looper.loop() // Have it ready for receiving
    }
}
```

Threads not always have loopers by default (use HandlerThread)

If the receiver does not know what to do upon receiving, but still wants to receive messages, then the handler can be empty.

Multithreading - Message Passing

Sender Side: obtain a reference to the receiver’s handler and send a message

```
val looperThread = LooperThread()
looperThread.start()
```

Send a message to be handled by handleMessage if receiver knows what to do

```
var message: Message = looperThread.handler.obtainMessage()
message.arg1 = 0        // Some custom body
looperThread.handler.sendMessage(message)
```

Send a runnable if the sender knows what to instruct

```
looperThread.handler.post {
    // Orders for the receiver
}
```

Multithreading - Message Passing

The main usage for message passing is to pass the result of the thread operation to the UI thread. Message loop is implicitly defined for the UI thread: if you get it you can create an empty Handler and post task for the UI thread.

```
val mainHandler = HandlerCompat.createAsync(Looper.getMainLooper())
mainHandler.post{
        // Run on UI thread
}
```

This has all been wrapped by the following:

```
runOnUiThread {
        // Run on UI thread
}
```

Multithreading - Coroutines

"A coroutine is an instance of a suspendable computation. It is conceptually similar to a thread, in the sense that it takes a block of code to run that works concurrently with the rest of the code. However, a coroutine is not bound to any particular thread. It may suspend its execution in one thread and resume in another one."

Coroutines can be thought of as light-weight threads, but there is a number of important differences that make their real-life usage very different from threads.

https://kotlinlang.org/docs/coroutines-basics.html

Multithreading - Coroutines

- **Coroutine Scope**
  - An environment that keeps track of the coroutines it creates and offers ways to interact with them (cancel, suspend, resume…). It launches them, but does not run them.
- **Coroutine Context**
  - A set of metadata about the coroutine, including the dispatcher, the element that runs the coroutine in a specific type of thread.
- **Coroutine Job**
  - A handle to a coroutine, it basically stores a reference to a running coroutine into a variable.

Multithreading - Coroutines

Create a coroutine with **launch** for legacy code (the caller knows what to put in a coroutine).

```
/* Create a scope that fires coroutines in the main (context) thread by default */
val scope: CoroutineScope = CoroutineScope(Dispatchers.Main)

fun myBlockingFunction () { /* Blocking Code */ }

/* launch a coroutine in a different context (IO threads) - you can omit it */
val job: Job = scope.launch (Dispatchers.IO) {
        delay(100)     // function that blocks the coroutine, not the thread
        myBlockingFunction()
}
```

One problem here is that the caller needs to know what is blocking…

Multithreading - Coroutines

- **Dispatchers.Main** - Use this dispatcher to run a coroutine on the main Android thread. This should be used only for interacting with the UI and performing quick work. Examples include calling suspend functions, running Android UI framework operations, and updating LiveData objects.
- **Dispatchers.IO** - This dispatcher is optimized to perform disk or network I/O outside of the main thread. Examples include using the Room component, reading from or writing to files, and running any network operations.
- **Dispatchers.Default** - This dispatcher is optimized to perform CPU-intensive work outside of the main thread. Example use cases include sorting a list and parsing JSON.

Multithreading - Coroutines

Create a **main-safe** function, the caller does not need to know where it runs.

```
/* The suspend keyword forces the caller to call the function within a coroutine */
suspend fun myBlockingFunction (): String {
      return withContext(Dispatchers.IO) { /* Blocking Code */ }
}

/* launch a coroutine in the main thread */
scope.launch {
      delay(100)
      val result = myBlockingFunction() // execute it in a IO thread and wait here until it finishes
      /* Do stuff with result in the main thread */
}
```

This setup makes the main thread **suspend** the coroutine until the IO thread has returned, without blocking the UI. It is good for synchronous operations.

Multithreading - Coroutines

Within a coroutine you can change the context if you need to update the UI.

```
/* launch a coroutine in the main thread */
scope.launch {
	withContext(Dispatchers.IO) { /* Do your database operations */ }
	withContext(Dispatchers.Main) { /* Update UI */ }
}
```

Alternatively, you can update your LiveData from a worker thread...

```
scope.launch {
	withContext(Dispatchers.IO) {
		/* Do your database operations */
		myLiveData.postValue(result) // Always watch out for race conditions though…
	}
}
```

Multithreading - Coroutines

Similar to Javascript promises, you can use **async** calls without suspending…

```
suspend fun myBlockingFunction () { withContext(Dispatchers.IO) { /* Blocking Code */ } }
```

async and **await** can be called only within a coroutine scope…

```
scope.launch {
      val deferred: Deferred<Unit> =
            async { myBlockingFunction() }
      /* CODE BLOCK A */
      deferred.await()
      /* CODE BLOCK B */
}
```

The coroutine does not suspend upon calling myBlockingFunction, it calls it asynchronously, then executes CODE BLOCK A, waits for myBlockingFunction to finish and executes CODE BLOCK B.

You can use also awaitAll() on a list of async Deferreds for parallelization.

Multithreading - Coroutines

- Lightweight: You can run many coroutines on a single thread due to support for suspension, which doesn't block the thread where the coroutine is running.
- Fewer memory leaks: Use structured concurrency to run operations within a scope.
- Built-in cancellation support: Cancellation is propagated automatically through the running coroutine hierarchy.

Read the full documentation about coroutines at:
https://developer.android.com/kotlin/coroutines

Services

A **Service** is a component that can perform long-running operations in background and does not provide a user interface.
Can be thought as the **dual** of an Activity.

- **Activity** → UI, can be disposed when it loses visibility
- **Service** → No UI, disposed when it terminates or when it is terminated by other components

Declare it in the manifest

```
<service android:name=".ExampleService" />
```

Services

A Service provides only a robust environment where to host separate threads of our application, but it is not a separate Thread… why should we use it then?

There are several reasons, but a very prominent one is:

Because if nothing else holds the main thread (i.e. no activity is running or stopped), then a Service is the only component that can keep the main thread alive.

Services

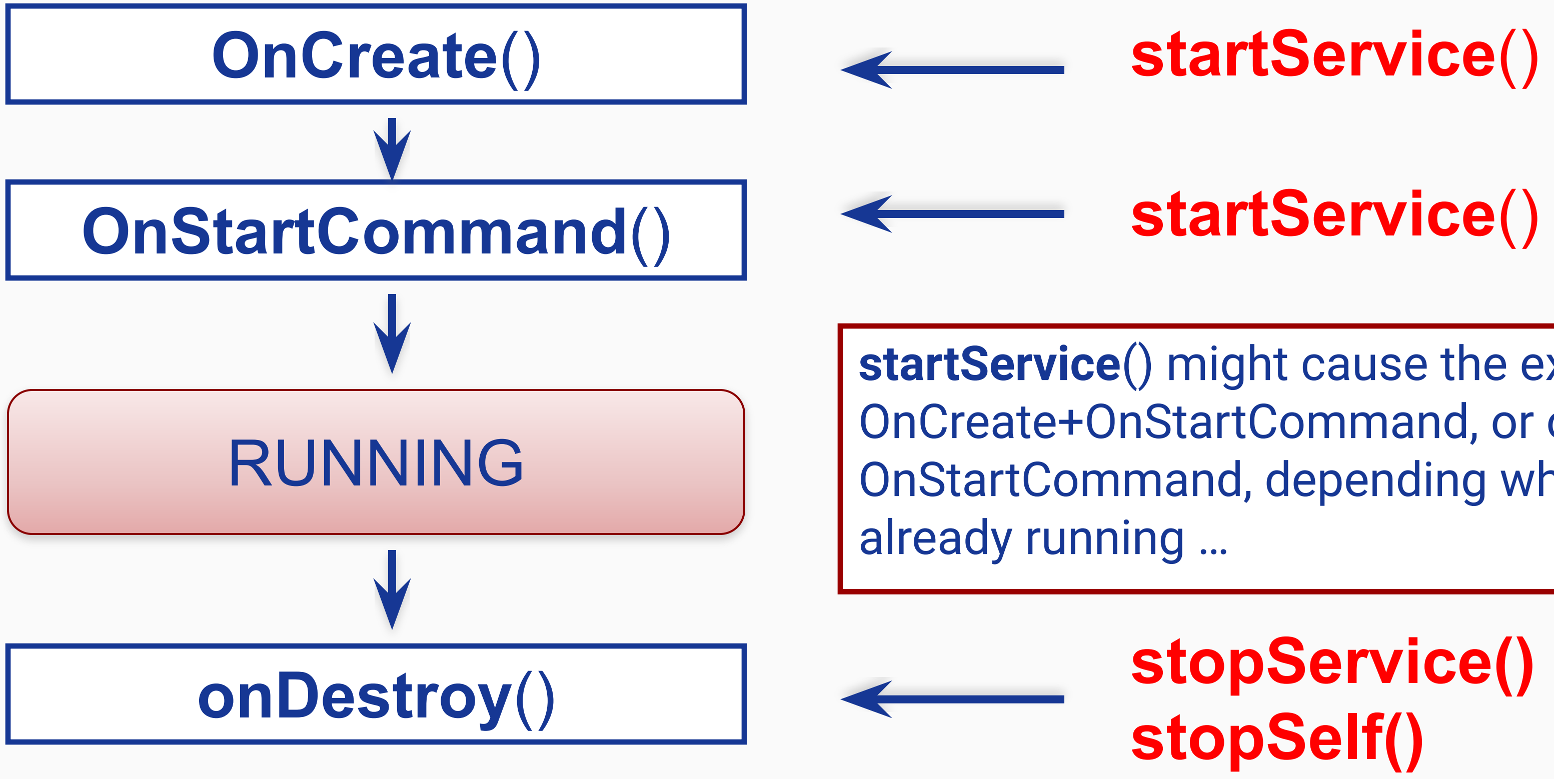
A Service is started when an application component starts it by calling **startService(Intent)**.

Once started, a Service can run in background, even if the component that started it is destroyed.

Termination of a Service:

1. **stopSelf**() → self-termination of the service
2. **stopService**(Intent) → terminated by others
3. System-decided termination (i.e. memory shortage)

Services



OnCreate() executed only once when the Service is created.

startService() might cause the execution of OnCreate+OnStartCommand, or only of OnStartCommand, depending whether the Service is already running …

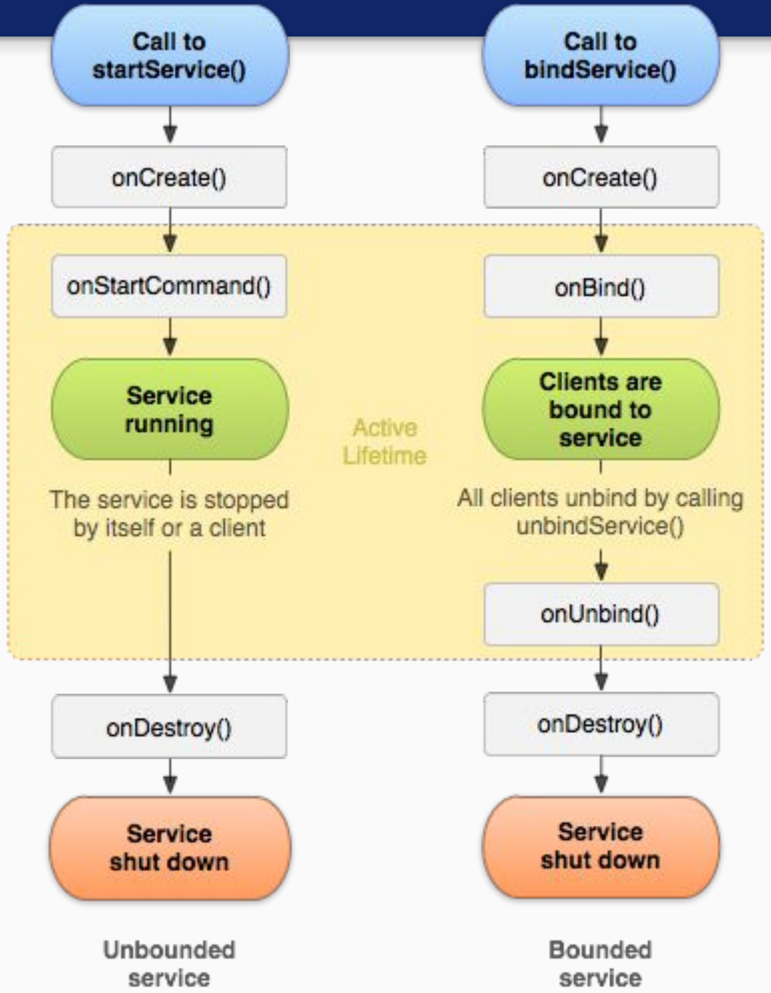
Services

Tell what we should do if the Service is killed by the system through the return flag in the onStartCommand():

- START_STICKY: recreate the service with a null intent
- START_NOT_STICKY: do not bother recreating it
- START_REDELIVER_INTENT: recreate the service and resent the same intent

```
class MyService: Service() {
  override fun onBind(intent: Intent?): IBinder? { /* Not bound */ return null }
  override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {
     super.onStartCommand(intent, flags, startId)
     /* Do your stuff */
     return START_STICKY
  }
}
```

Services



Call to startService()

onCreate()

onStartCommand()

Service running

The service is stopped by itself or a client

onDestroy()

Service shut down

Unbounded service

Call to bindService()

onCreate()

onBind()

Clients are bound to service

All clients unbind by calling unbindService()

onUnbind()

onDestroy()

Service shut down

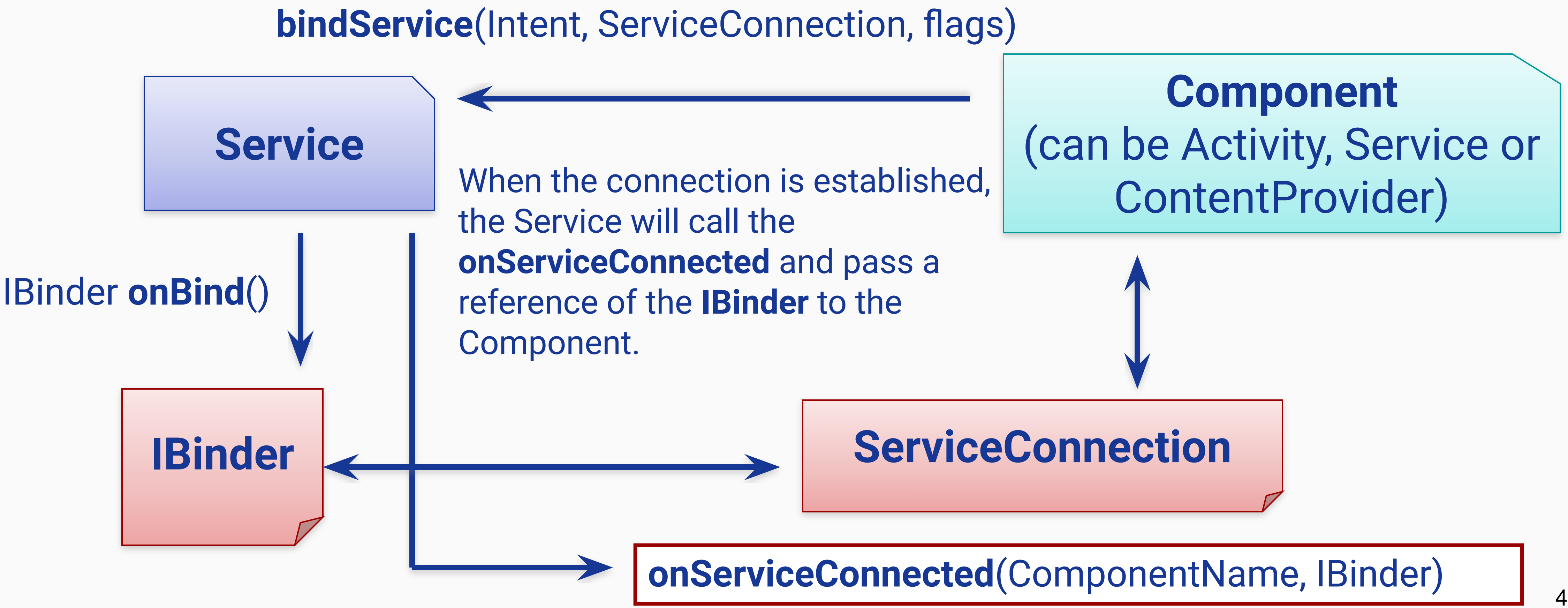
Bounded service

Active Lifetime

Bound Services: Services can either be started with **startService**() or bound to a component through **bindService**(): in the second case the binding lifecycle takes over.

- Bound services end when all the bound components unbind
- These two lifecycles are not separated: a component can bind to a started service.
    - in such case unbinding kills, stopping does not.

Services



bindService(Intent, ServiceConnection, flags)

Service

Component
(can be Activity, Service or ContentProvider)

IBinder **onBind**()

When the connection is established, the Service will call the **onServiceConnected** and pass a reference of the **IBinder** to the Component.

IBinder

ServiceConnection

onServiceConnected(ComponentName, IBinder)

Services

When creating a Service, an **IBinder** must be created to provide an Interface that clients can use to interact with the Service … HOW?

1. Extending the **Binder** class (local Services only)
   - Extend the Binder class and return it from **onBind**()
   - Only for a Service used by the same application

2. Using the Android Interface Definition Language (AIDL)
   - Allow to access a Service from different applications.

Services

Example **Service Side**

```
class LocalService: Service() {

    inner class SimpleBinder: Binder() {
        fun getService(): LocalService { return this@LocalService }
    }
    private val binder = SimpleBinder()

    override fun onBind(intent: Intent?): IBinder? { return binder }

    fun apiFunction() { /* Stuff for clients */ }
}
```

Services

Example **Client Side** (e.g. from an Activity)

```
private lateinit var localService: LocalService
val serviceConnection: ServiceConnection = object : ServiceConnection{
      override fun onServiceConnected(className: ComponentName, service: IBinder) {
            localService = (service as LocalService.SimpleBinder).getService()
      }
      override fun onServiceDisconnected(name: ComponentName?) { /* Whatever */ }
}

bindService(
      Intent(this, LocalService::class.java), serviceConnection, BIND_AUTO_CREATE
      )
/* Now we can call localService.apiFunction() */
```

Services

Foreground Services: A Foreground Service is a service that is continuously active in the Status Bar, and thus it is not a good candidate to be killed in case of low memory. Its Notification appears between ONGOING pendings.

To create a Foreground Service:

- Create a **Notification** object
- Call ServiceCompat.**startForeground**(id, notification) within **onStartCommand**()
- Call **stopForeground**() to bring it to the background.

Note that you need FOREGROUND_SERVICE permission

Broadcast Receivers

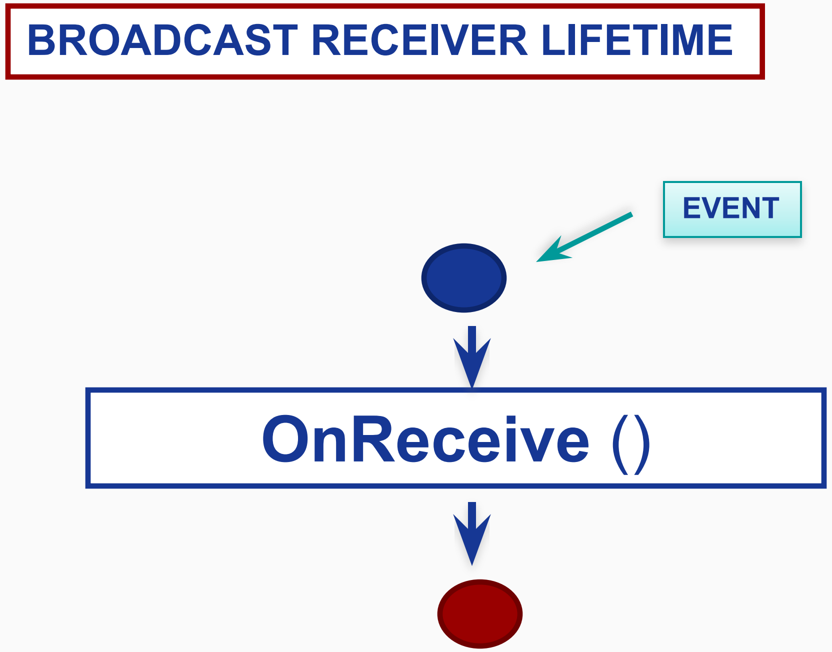
A **Broadcast Receiver** is a component that is activated only when specific events occur (i.e. SMS arrival, phone call, etc).

The Event is an Intent

Registration of the Broadcast Receiver to the event using an IntentFilter:

- Registration through XML code (Manifest-declared) as you would do for Activities and Services
- Registration through Java/Kotlin code (Context-declared)
  - In this case it listens for events only within a context.

Broadcast Receivers



- Single-state component …
- onReceive() is invoked when the registered event occurs
- After handling the event, the Broadcast Receiver is destroyed.
- It runs in the **Main Thread** by default.

Broadcast Receivers

- Registration in the context code:

```
val broadcastReceiver =
    object: BroadcastReceiver() {
    override fun onReceive
        (context: Context?, intent: Intent?) {
        /* Do your stuff */
    }
}
```

This example lacks permission requests for brevity

```
override fun onResume() {
    super.onResume()
    registerReceiver
        (broadcastReceiver, IntentFilter(
        "android.provider.Telephony.SMS_RECEIVED
        "))
}

override fun onPause() {
    super.onPause()
    unregisterReceiver(broadcastReceiver)
}
```

Broadcast Receivers

- Registration in the manifest:

```
 <application>
      <receiver class=“SMSReceiver”>
            <intent-filter>
                  <action  android:value=“android.provider.Telephony.SMS_RECEIVED” />
            </intent-filter>
      </receiver>
</application>
```

The receiver here can be activated even it the app is closed, but onReceive must be short enough! In this case it is always better to:

- Run the BroadcastReceiver within a sticky service
- Start a

Broadcast Receivers

How to send Intents for broadcast Receivers?

- sendBroadcast(intent: Intent)
  - No order of reception is specified for all registered receivers
- sendOrderedBroadcast(intent: Intent, permit: String)
  - reception order given by the android:priority field

sendBroadcast() and startActivity() work on different contexts!

Questions?

federico.montori2@unibo.it