Programming with Android:
Notifications, Threads, Services

Luca Bedogni

Dipartimento di Scienze dell'Informazione

Università di Bologna

Outline

Notification Services: **Status Bar** Notifications

Notification Services: **Toast** Notifications

Thread Management in Android

Thread: **Handler and Looper**

Services: Local Services

Services: Remote Services

Broadcast Receivers

Android: **Where are we now …**

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?

Android: Where are we now ...

➢ Setup of the application **GUI** ✓

➢ GUI **event** management ✓

➢ Application **Menu** and **Preferences** ✓

➢ *Network functionalities* (send/receive messages) ✓

➢ Updates in **background** mode ✗

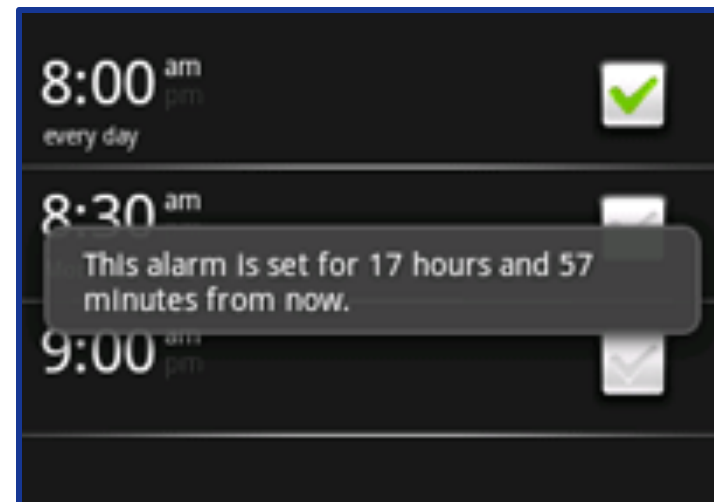➢ **Notifications** in case of message reception in background mode ✗

Android: Service Notifications Types

Service Notifications: *Mechanism to notify information to the end-user on the occurrence of specific events ....*
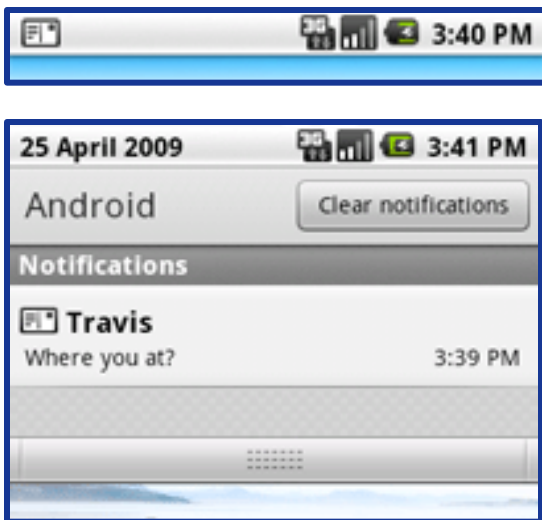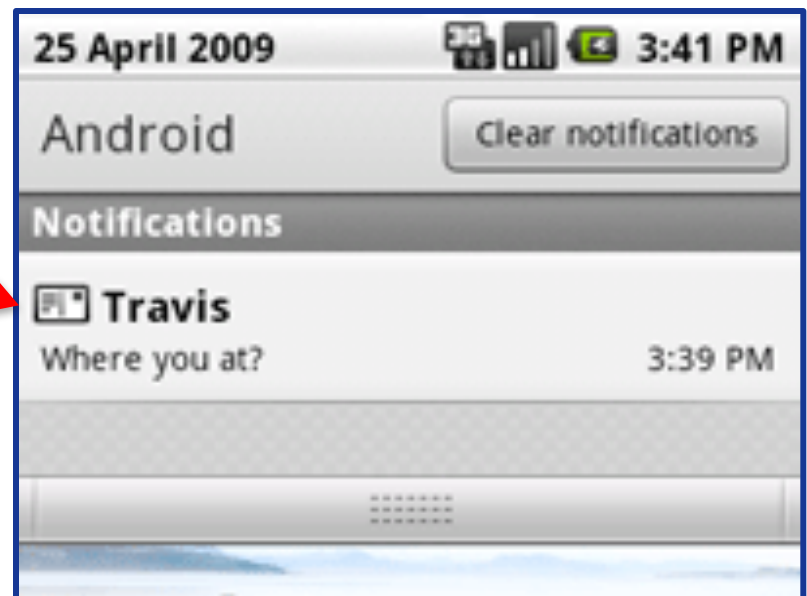

Status Bar Notifications

Toast Notifications

Android: Status Bar Notifications

➢ Used by *background services* to notify the occurrence of an event that requires a **response** … without *interrupting the operations of the foreground activities*!

➢ Display an **icon** on the Status Bar (top screen)

➢Display a **message** in the Notification Window

➢Fire an **event** in case the user selects the notification

Android: Status Bar Notifications

STATUS BAR

Notification

Notification Manager

Android system component
Responsible for notification management
And status bar updates
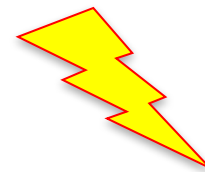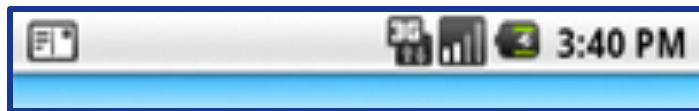
> ➢ **Icon** for the status bar
> ➢ **Title** and **message**
> ➢ **PendingIntent** to be fired
>   when notification is selected

OPTIONs:

> ➢ Ticket-text message
> ➢ Alert-sound
> ➢ Vibrate setting
> ➢ Flashing LED setting
> ➢ Customized layout

➤ Follow these steps to send a Notification:

1. Get a **reference** to the **Notification Manager**

   NotificationManager nm=(NotificationManager) **getSystemService**(Context.NOTIFICATION_SERVICE)

2. **Build** the Notification message

public **Notification**(int icon, CharSequence tickerText, long when)

public void **setLatestEvent**(Context context, CharSequence contentTitle, CharSequence contentText, PendingIntent intent)

3. **Send** the notification to the Notification Manager

public void **notify**(int id, Notification notification)

Build the notification object

```
// Specificy icon, ticket message and time
Notification notification = new Notification(R.drawable.icon, "This is a very basic Notification to catch your attention!", System.currentTimeMillis());
```

Define what will happen in case the user selects the notification

```
// Build an explicit intent to NotificationActivity
Intent intent = new Intent(this, NotificationActivity.class);
PendingIntent pIntent = PendingIntent.getActivity(this, 0, intent, PendingIntent.FLAG_CANCEL_CURRENT);
```

Add (optional) flags for notification handling

```
// Specificy that notification will disappear when handled
notification.flags |= Notification.FLAG_AUTO_CANCEL;
```

Send the notification to the Notification Manager

```
// Set short and long message to be displayed on the notification window
// Set the PendingIntent
notification.setLatestEventInfo(this, "Notification", "Click to launch
NotificationActivity", pIntent);
notificationManager.notify(SIMPLE_NOTIFICATION_ID, notification);
```

Add a **sound** to the notification

```
// Use a default sound
notification.defaults |= Notification.DEFAULT_SOUND;
```

Pass an **URI** to the sound field to set a different sound

```
notification.sound = Uri.parse(file://sdcard/path/ringer.mp3);
```

Use FLAG_INSISTENT to play the sound till notification is handled

```
notification.flags |= Notification.FLAG_INSISTENT;
```

Android: Status Bar Notifications

Add **flashing lights** to the notification

```
// Use a default LED
notification.defaults |= Notification.DEFAULT_LIGHTS;
```

Define *color and pattern* of the flashing lights

```
notification.ledARGB = 0xff00ff00;
notification.ledOnMS = 300;
notification. ledOffMS = 1000;
notification.flags |= Notification.FLAG_SHOW_LIGHTS;
```

Android: Status Bar Notifications

Add **vibrations** to the notification

```
// Use a default vibration
notification.defaults |= Notification.DEFAULT_VIBRATE;
```

Define *the vibration pattern*

```
// Set two vibrations, one starting at time 0 and with duration equal to 100ms
long[] vibrate={0,100,200,300};
notification.vibrate = vibrate;
```

Android: Status Bar Notifications

Some **flags** that can be used (see the documentation)

- ➤ **FLAG_NO_CLEAR**: Notification is not canceled
- ➤ **FLAG_ONGOING_EVENT**: Notify ongoing events (e.g. a call)
- ➤ **FLAG_AUTO_CANCEL**: Notification disappears as handled
- ➤ **FLAG_INSISTENT**: Reproduce sound till notification is handled
- ➤ **FLAG_FOREGROUND_SERVICE**: Notification from an active service

… Also **PendingIntents** can have flags

- ➤ **FLAG_CANCEL_CURRENT**: PendingIntents are ovewritten
- ➤ **FLAG_UPDATE_CURRENT**: PendingIntents are updated (*extra field*)

A **Toast Notification** is a message that pops up on the surface of the window, and automatically fades out.

> ➤ Typically created by the *foreground* activity.
>
> ➤ *Display* a message text and then fades out
>
> ➤ **Does not accept events**! (use *Status Bar Notifications* instead)

Android: **Toast Notifications**

A **Toast Notification** is a message that pops up on the surface of the window, and automatically fades out.

```
Context context=getApplicationContext();

// Define text and duration of the notification
CharSequence text="This is a Toast Notification!";
int duration=Toast.LENGTH_SHORT;

Toast toast=Toast.makeText(context, text, duration);

// Send the notification to the screen
toast.show();
```

Android: **Processes** and **Threads**

➢ By default, <u>all components of the same application run in the same process and thread</u> (called "**main** thread" or "**UI**" thread).

➢ In **Manifest.xml**, it is possible to specify the process in which a component (e.g. an activity) 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.

Android: Thread Management

➤ Android natively supports a **multi-threading** environment.

➤ An Android application can be composed of multiple *concurrent* threads.

➤ How to create a thread in Android? … Like in Java!

  ➤ extending the **Thread** class   **OR**
  ➤ implementing the **Runnable** interface
  ➤ **run**() method executed when MyThread.**start**() is launched.

```
public class MyThread extends Thread {

    public MyThread() {
        super ("My Threads");
    }

    public void run() {
        // do something
    }
}
```

```
myThread m=new MyThread();
m.start();
```

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

➢ <u>Do not block the UI thread</u>.

➢ <u>Do not access the Android UI components from outside the UI thread.</u>

QUESTIONS:

How to update the UI components from worker threads?

Android: AsyncTask

AsyncTask is a Thread helper class (Android only).

✧ Computation running on a **background** thread.
✧ Results are published on the **UI** thread.

RULES

➢ AsyncTask must be created on the UI thread.
➢ AsyncTask can be executed only once.
➢ AsyncTask must be canceled to stop the execution.

private class MyTask extends **AsyncTask**<Par, Prog, Res>

Par → type of parameters sent to the AsyncTask
Prog → type of progress units published during the execution
Res → type of result of the computation

EXAMPLES

private class MyTask extends AsyncTask<Void,Void,Void>

private class MyTask extends AsyncTask<Integer,Void,Integer>

Android: AsyncTask

The UI Thread invokes the **execute** method of the AsyncTask:

(new Task()).**execute**(param1, param2 … paramN)

After **execute** is invoked, the task goes through four steps:

1.onPreExecute() → invoked on the UI thread
2.doInBackground(Params…) →computation of the AsyncTask
  ◇ can invoke the publishProgress(Progress…) method
3.onProgressUpdate(Progress …) → invoked on the UI thread
4.onPostExecute(Result) → invoked on the UI thread

Android: Thread Management

Message-passing like mechanisms for Thread communication.

MessageQueue → Each thread is associated a queue of messages
Handler → Handler of the message associated to the thread
Message → Parcelable Object that can be sent/received

handleMessage(Message msg)

Handler

Message queue

sendMessage(Message msg)

THREAD1

THREAD2

Message loop is <u>implicitly defined</u> for the **UI** thread … but it must be <u>explicitly defined</u> for worker threads.

HOW? Use **Looper** objects …

```
public void run() {
        Looper.prepare();
        handler=new Handler() {
                public void handleMessage(Message msg) {
                        // do something
                }
        }
        Looper.loop();
```

Android: Services

A **Service** is an application that can perform *long-running operations in background* and *does not provide a user interface*.

➢ **Activity** → UI, can be disposed when it loses visibility

➢ **Service** → No UI, disposed when it terminates or when it is terminated by other components

A Service provides a robust environment for background tasks …

Android: 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. **selfStop**() → self-termination of the service

    2. **stopService**(Intent) → terminated by others

    3. System-decided termination (i.e. memory shortage)

Android: Service Lifetime

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

```
    ●
    │
    ▼
┌─────────────────────┐
│   OnCreate()        │  ◄──────  startService()
└─────────────────────┘
    │
    ▼
┌─────────────────────┐
│ OnStartCommand()    │  ◄──────  startService()
└─────────────────────┘
    │
    ▼
┌─────────────────────┐
│     RUNNING         │
└─────────────────────┘
    │
    ▼
┌─────────────────────┐
│   onDestroy()       │  ◄──────  stopService()
└─────────────────────┘                selfStop()
    │
    ▼
    ●
```

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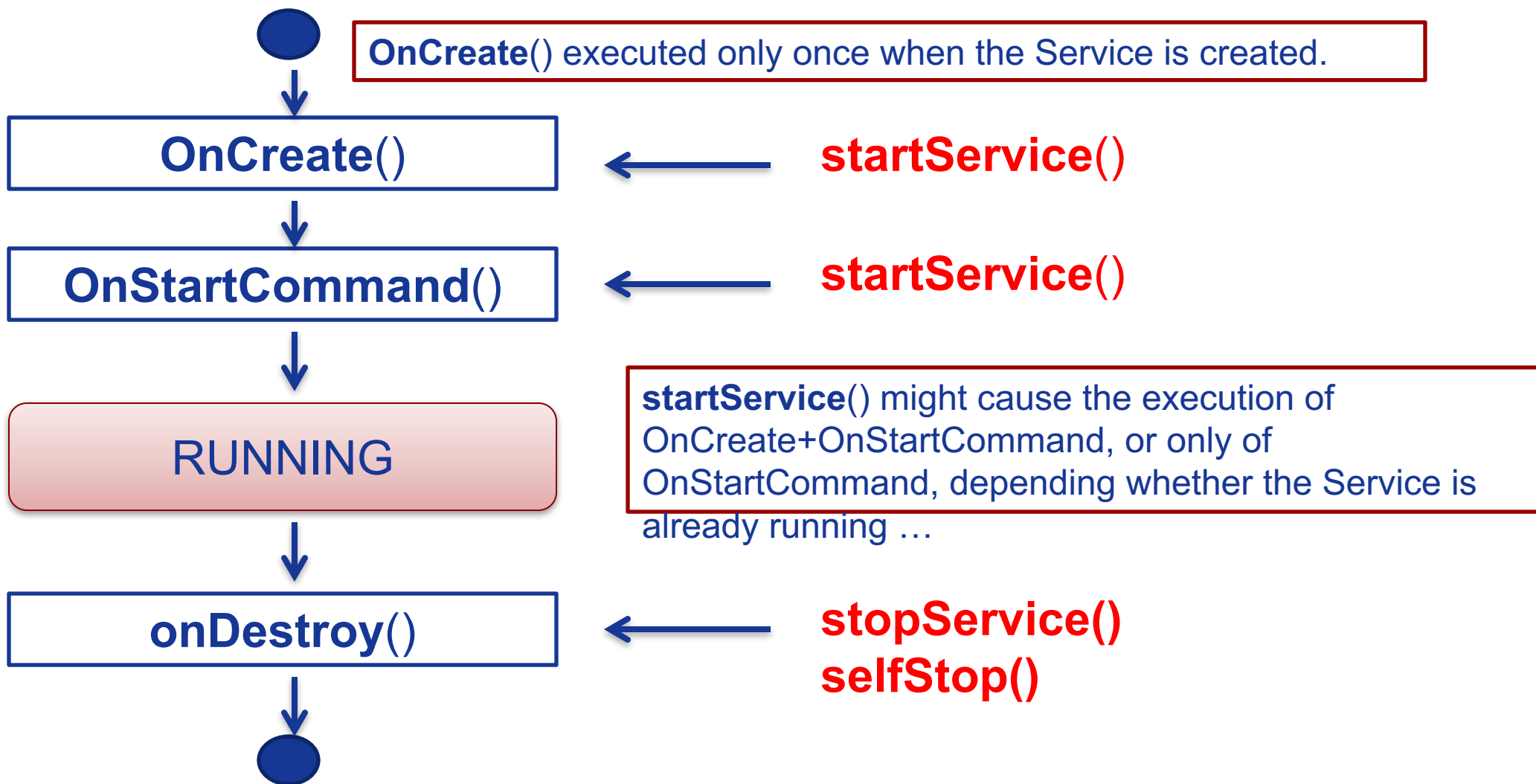
Android: Foreground Services

COMMON **MISTAKES**

➢ A **Service** provides only a **robust environment** where to host separate threads of our application.

- ◇ A Service is not a separate process.

- ◇ A Service is not a separate Thread (i.e. it runs in the main thread of the application that hosts it).

- ◇ A Service does nothing except executing what listed in the **OnCreate**() and **OnStartCommand**() methods.

- ◇ Behaviors of Local/Bound Services can be different.

➢ A **Foreground** Service is a service that is continuously active in the <u>Status Bar</u>, and thus it is not a good candidate to be killed in case of low memory.

➢ The Notification appears between **ONGOING** pendings.

➢ To create a Foreground Service:

1. Create a **Notification** object

2. Call **startForeground**(id, notification) from onStartCommand()

➢ Call **stopForeground**() to stop the Service.

Android: Service Lifetime


```
        ●
        │
        ▼
┌─────────────────┐
│   OnCreate()    │
└─────────────────┘
        │
        ▼
┌─────────────────┐
│ OnStartCommand()│
└─────────────────┘
        │
        ▼
┌─────────────────┐
│    RUNNING      │
└─────────────────┘
        │
        ▼
┌─────────────────┐
│   onDestroy()   │
└─────────────────┘
        │
        ▼
        ●
```
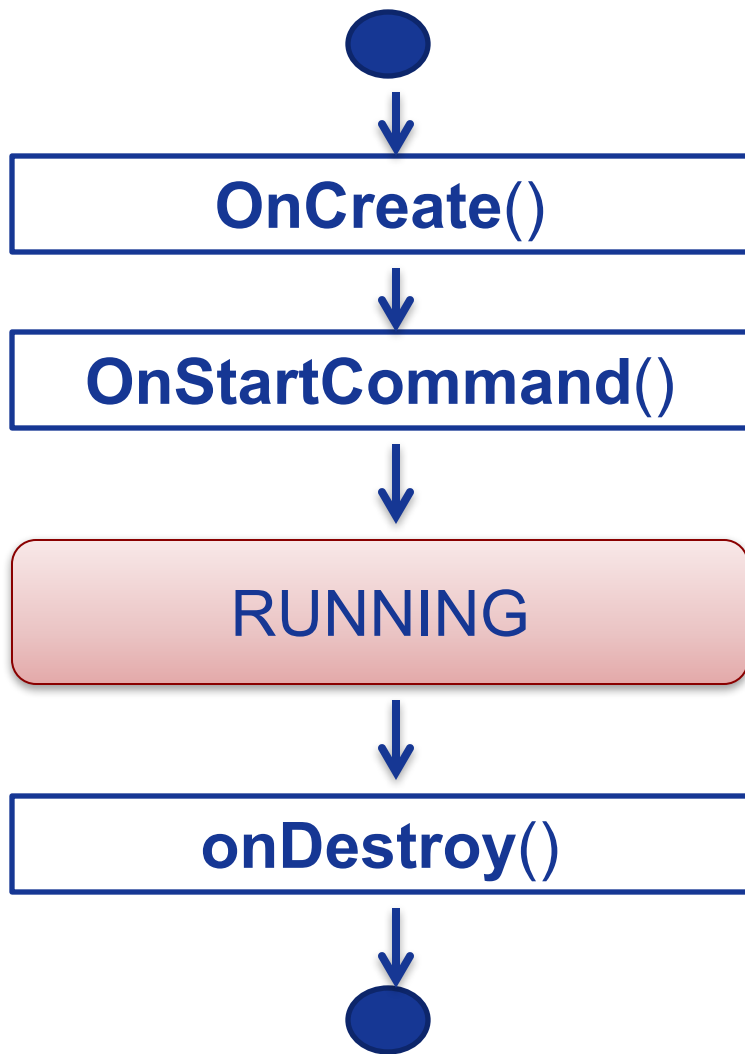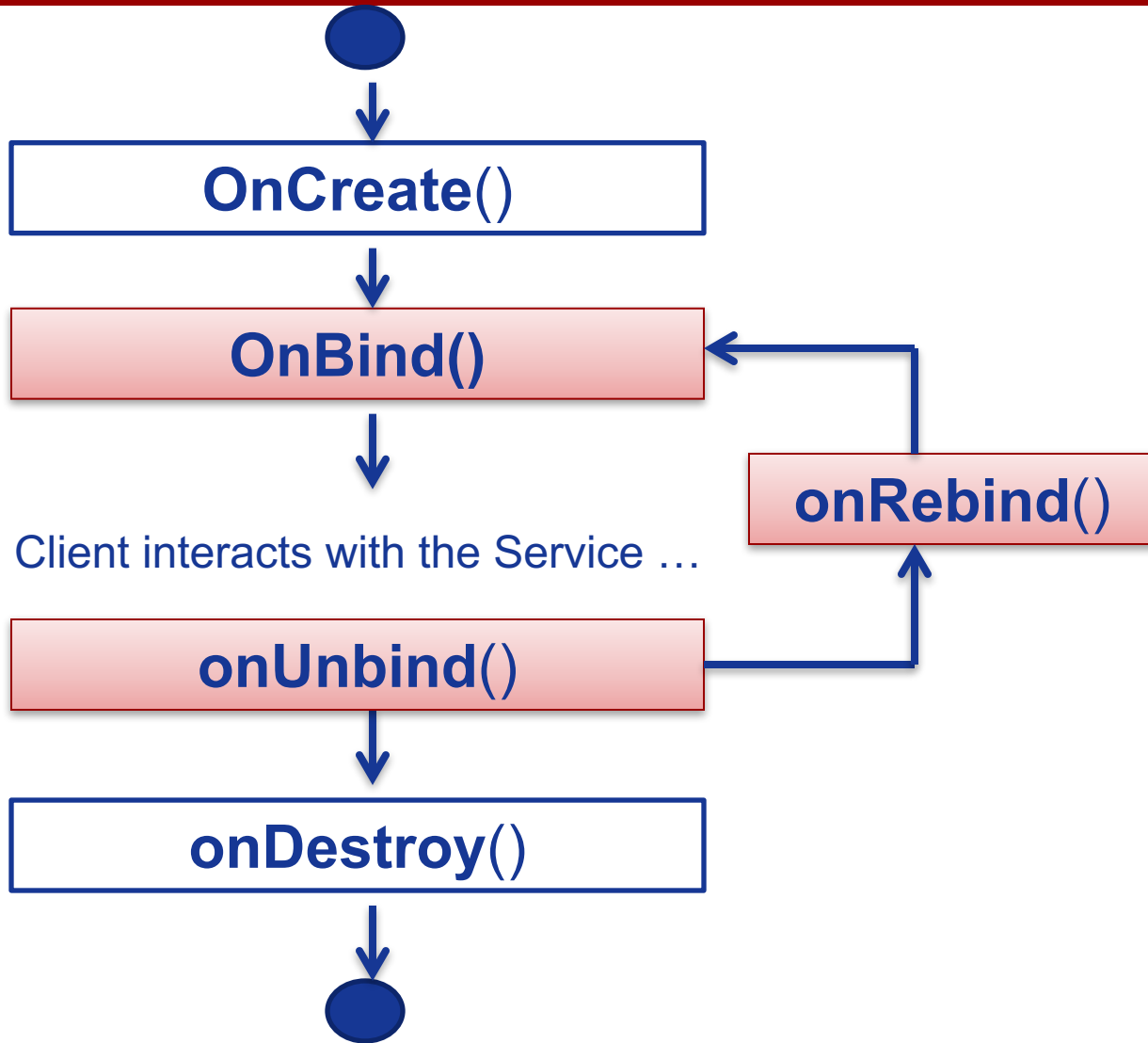
Two Types of **Services**:

1. **Local** Services: Start-stop lifecycle as the one shown.

2. **Remote/Bound** Services: Bound to application components. Allow interactions with them, send requests, get results, IPC facilities.

Android: Bound Service

```
        ( )
         |
         v
   ┌─────────────────┐
   │   OnCreate()    │
   └─────────────────┘
         |
         v
   ┌─────────────────┐
   │    OnBind()     │ ◄────────┐
   └─────────────────┘          │
         |                 ┌────────────┐
         v                 │ onRebind() │
Client interacts with      └────────────┘
the Service …                   ▲
   ┌─────────────────┐          │
   │   onUnbind()    │ ─────────┘
   └─────────────────┘
         |
         v
   ┌─────────────────┐
   │   onDestroy()   │
   └─────────────────┘
         |
         v
        ( )
```

➢ A **Bound** Service allows components (e.g. Activity) to **bind** to the services, **send** requests, **receive** response.

➢ A **Bound** Service can serve components running on different processes (**IPC**).
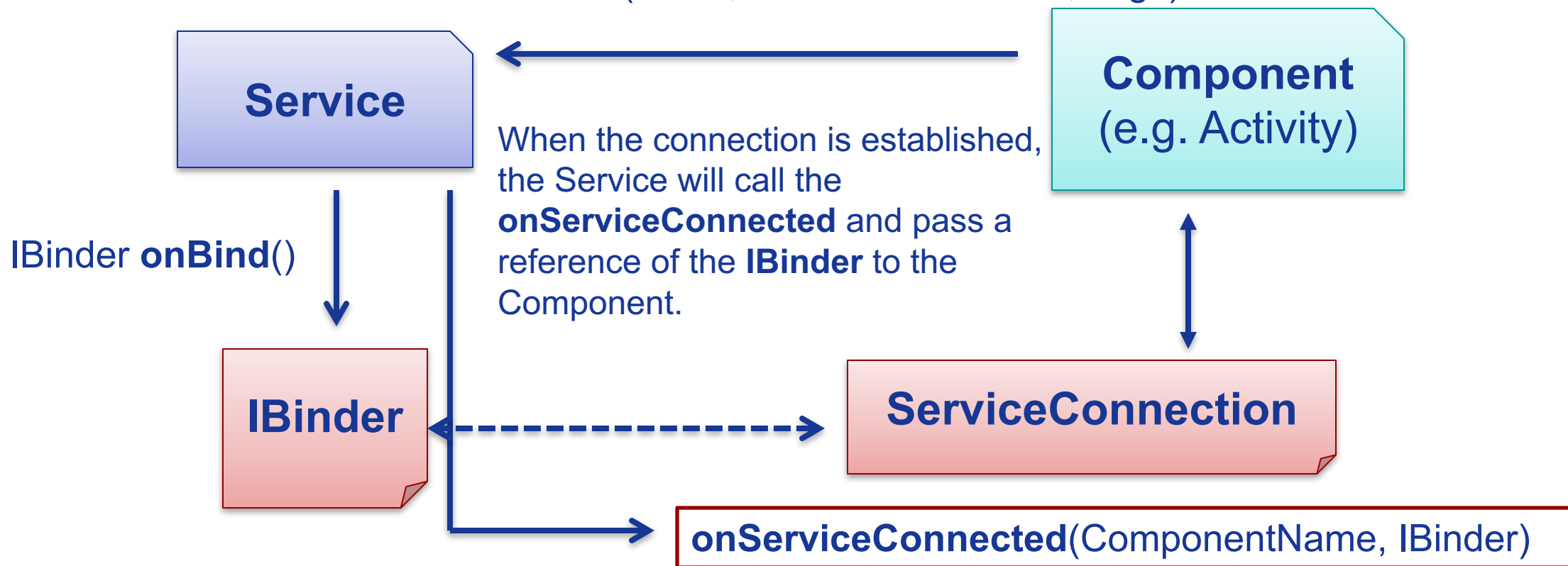
Android: Bound Service

> Through the **IBinder**, the Component can send requests to the Service …

bindService(Intent, ServiceConnection, flags)

Service

IBinder **onBind**()

Component
(e.g. Activity)

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)

Android: Bound Service

➢ When creating a Service, <u>an **IBinder** must be created to provide an Interface that clients can use to interact with the Service</u> … 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

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

```java
public class LocalService extends Service {
    // Binder given to clients
    private final IBinder sBinder=(IBinder) new SimpleBinder();

    @Override
    public IBinder onBind(Intent arg0) {
        // TODO Auto-generated method stub
        return sBinder;
    }

    class SimpleBinder extends Binder {
        LocalService getService() {
            return LocalService.this;
        }
    }
}
```

```java
public class MyActivity extends Activity {
    LocalService IService;
    private ServiceConnection mConnection=new ServiceConnection() {

        @Override
        public void onServiceConnected(ComponentName arg0, IBinder bind) {
            SimpleBinder sBinder=(SimpleBinder) bind;
            IService=sBinder.getService();
            ….
        }

        @Override
        public void onServiceDisconnected(ComponentName arg0) {
        }
        … bindService(new  Intent(this,LocalService.class),mConnection,BIND_AUTO_CREATE);

    };
```

Android: Broadcast Receiver

A **Broadcast Receiver** is a component that is activated only when specific events occur (i.e. SMS arrival, phone call, etc).
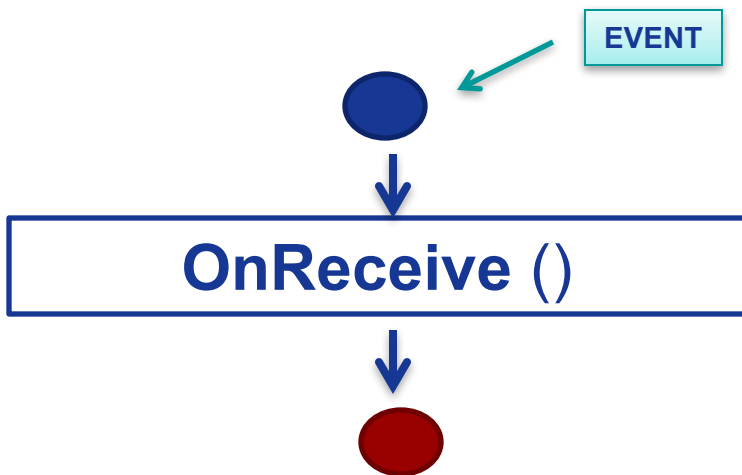
➢ **Registration** of the Broadcast Receiver to the event …

1. Event → **Intent**
2. Registration through **XML** code
3. Registration through **Java** code

➢ **Handling** of the event.

Android: Broadcast Receiver

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

BROADCAST RECEIVER LIFETIME

EVENT

OnReceive ()

➢ Single-state component …

➢ **onReceive**() is invoked when the registered event occurs

➢ After handling the event, the Broadcast Receiver is **destroyed**.

Android: Broadcast Receiver

➢**Registration** of the Broadcast Receiver to the event …
XML Code: ➔ modify the **AndroidManifest**.xml

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

➢ **Registration** of the Broadcast Receiver to the event …
    In **Java** → **registerReceiver**(BroadcastReceiver, IntentFilter)

```
receiver=new BroadcastReceiver() { … }

protected void onResume() {
    registerReceiver(receiver, new IntentFilter(Intent.ACTION_TIME_TICK));
}


protected void onPause() {
    unregisterReceiver(receiver);
}
```

How to send the **Intents** handled by **Broadcast Receivers**?

➢void **sendBroadcast**(Intent intent)
… No order of reception is specified

➢void **sendOrderedBroadcast**(Intent intent, String permit)
… reception order given by the android:priority field

sendBroadcast() and startActivity() <u>work on different contexts!</u>