Programming with Android: Background Operation

Federico Montori

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 …

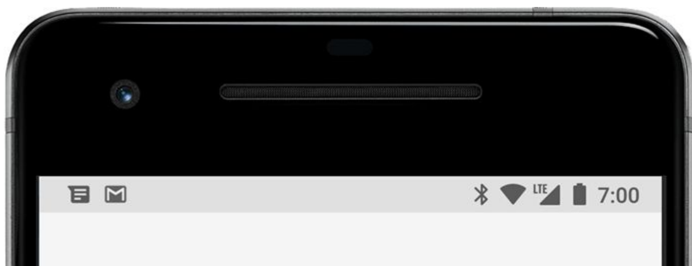
EXAMPLE: A simple application of *Instantaneous Messaging* (**IM**)

- Setup of the application **GUI** ✓
- GUI **event** management ✓
- Application **Menu** and **Preferences** ✓
- Updates in **background** mode ✗
- **Notifications** in case of message reception in background mode ✗

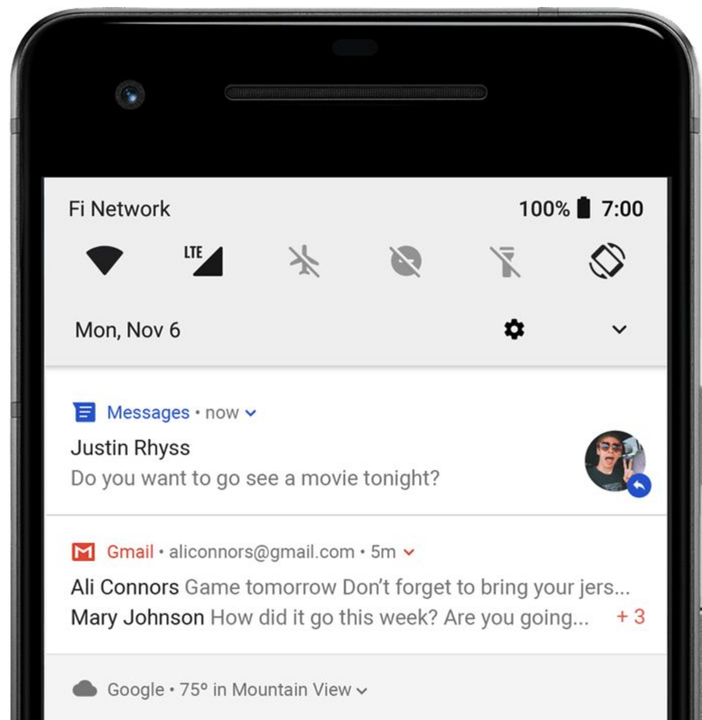
Notifications Overview

- Notifications are messages from your application
  - Reminders
  - External events
  - Timely information
- 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

Notification Types



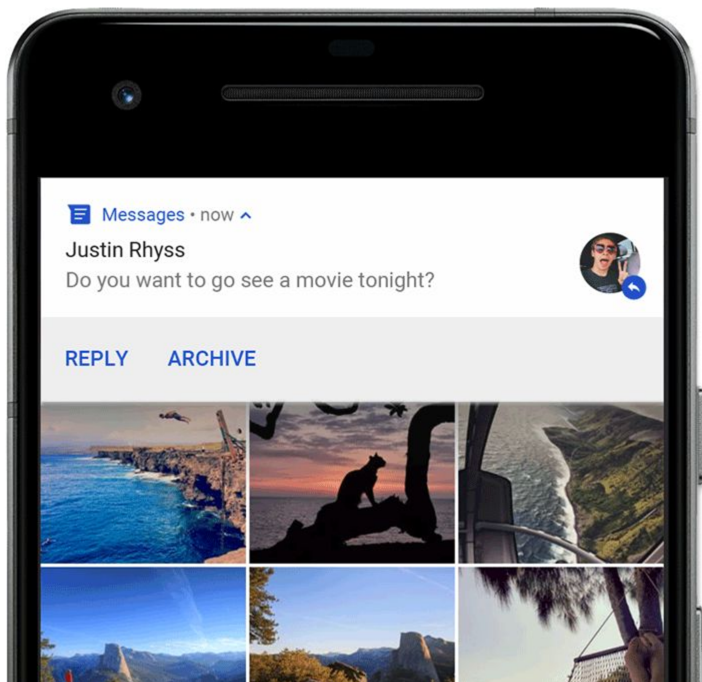
When the notification is created, its icon appears in the status bar



Scrolling down the status bar reveals additional details about the notification

Some notification can also reveal further information by swiping them down

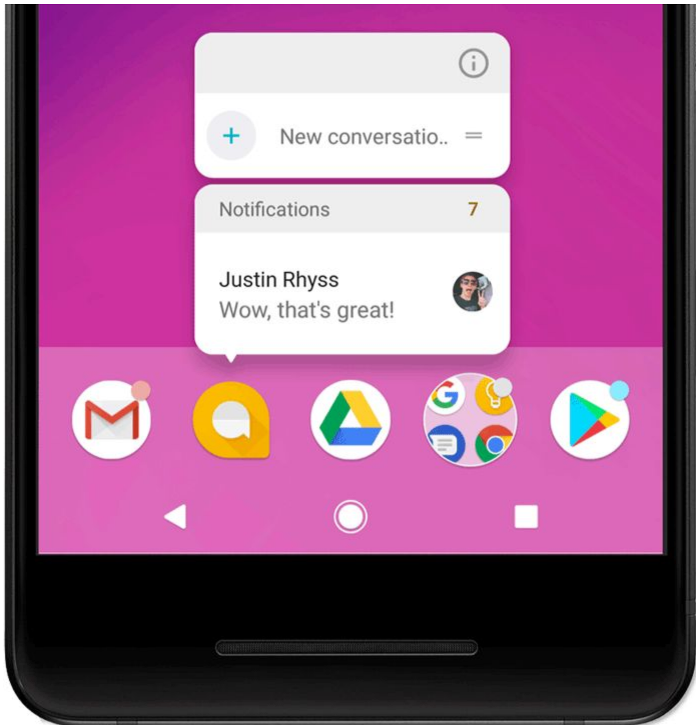
Notification Types



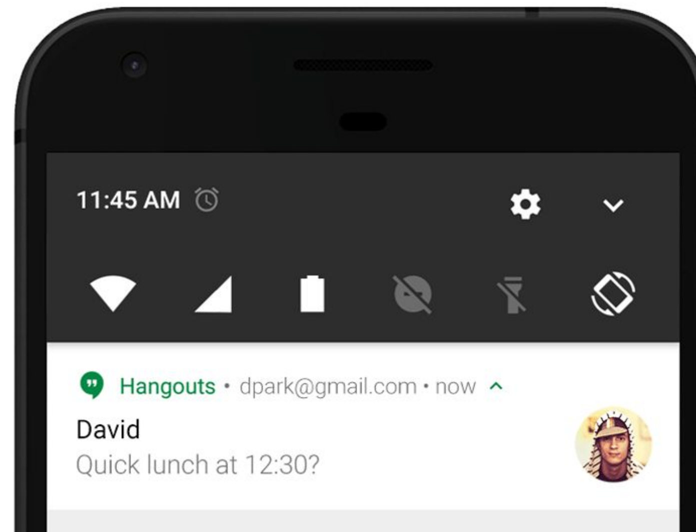
Heads up notification: useful for important information, and to notify the user while watching a full screen activity (starting from 5.0)

Notifications can also be visible in the lock screen. The developers can configure the amount of details that has to be made visible.

More notification Types



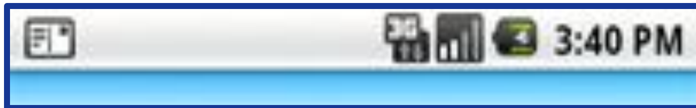
Icon badge: starting with Android 8.0. Users can get notification information about an app.

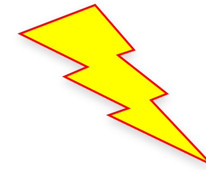


Wearables, to show the same notification on the hand-held device and wearable

Android: Status Bar Notifications

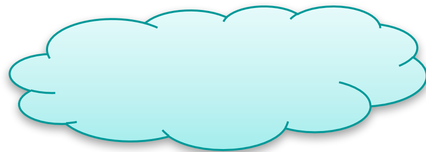
STATUS BAR





Notification

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



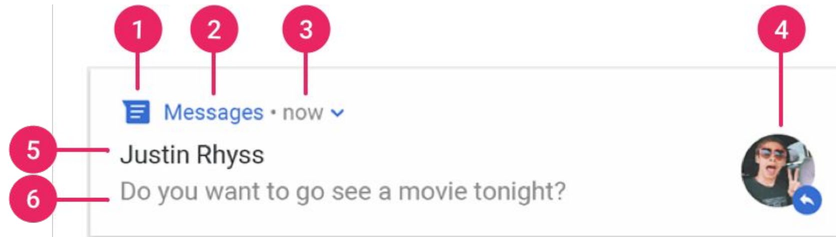
Notification Manager

Android system component
Responsible for notification management
And status bar updates

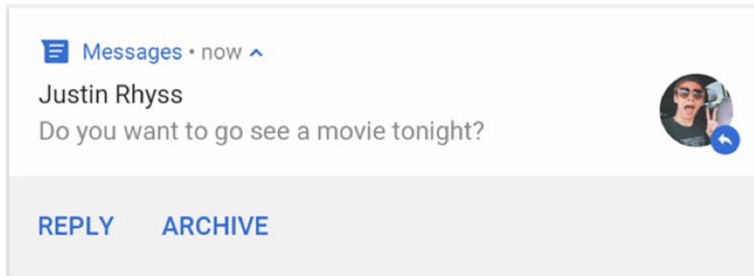
OPTIONs:

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

How a notification is made



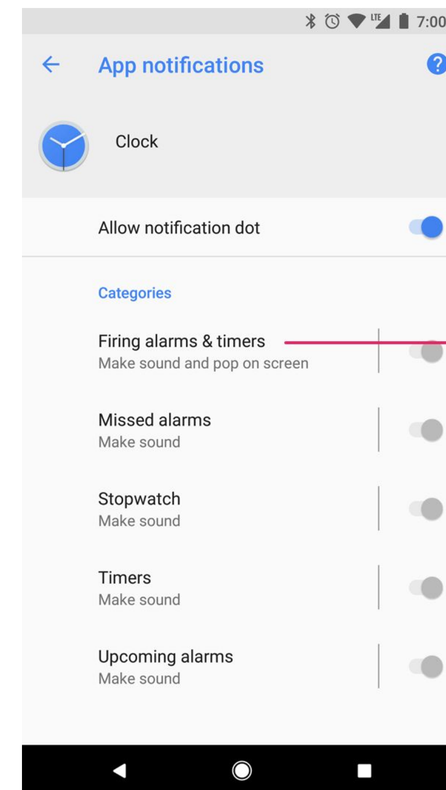
1. Small icon
2. App name
3. Timestamp
4. Optional Large Icon
5. Optional Title
6. Optional Text



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

Grouping Notification

- 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



Android: Status Bar Notifications

- For notifications, we will use the NotificationCompat module, for the newest management of notifications, still providing backwards compatibility.
- It should be already included, but still you'll need to check whether the dependency is there...

```
dependencies {
    implementation "com.android.support:support-compat:28.0.0"
}
```

Android: Status Bar Notifications

- Follow these steps to send a Notification:

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

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

or (better)

```
NotificationManagerCompat nm = NotificationManagerCompat.from(this);
```

2. **Build** the Notification message (design pattern **Builder**)

```
NotificationCompat.Builder mBuilder = new NotificationCompat.Builder(this, CHANNEL_ID);
mBuilder.setContentTitle("Picture Download").setContentText("Download in progress")
.setSmallIcon(R.mipmap.ic_launcher_round).setPriority(NotificationCompat.PRIORITY_DEFAULT);
```

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

```
notificationManager.notify(myId, mBuilder.build());
```

Android: Status Bar Notifications

- Follow these steps to send a Notification:

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

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

or (better)

```
NotificationManagerCompat nm = NotificationManagerCompat.from(this);
```

2. **Build** the Notification message (design pattern **Builder**)

```
NotificationCompat.Builder mBuilder = new NotificationCompat.Builder(this, CHANNEL_ID);
mBuilder.setContentTitle("Picture Download").setContentText("Download in progress")
.setSmallIcon(R.mipmap.ic_launcher_round).setPriority(NotificationCompat.PRIORITY_DEFAULT);
```

Ignored if older than 8.0

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

```
notificationManager.notify(myId, mBuilder.build());
```

Set by the developer at this time
Used for later modifications if needed

Android: Status Bar Notifications

Define what will happen in case the user selects the notification

- Define a **PendingIntent** (intent to be fired later by someone else)

```
Intent newIntent = new Intent(this, ReceivingActivity.class);
newIntent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK |
    Intent.FLAG_ACTIVITY_CLEAR_TASK);
newIntent.putExtra("CALLER","notifyService");
PendingIntent pendingIntent = PendingIntent.getActivity(this, 0, newIntent, PendingIntent.FLAG_IMMUTABLE);
```

A Class in your APP, such as a normal intent

```
mBuilder.setContentIntent(pendingIntent);
```

Is more a container for an intent, specifying in which context should be fired, the dev-defined request code, which intent should be fired and a set of flags.

Android: Status Bar Notifications

Define what will happen in case the user selects the notification **button**(s)

- Define a **PendingIntent** (intent to be fired later by someone else)

```
Intent newIntent = new Intent(this, ReceivingActivity.class);
newIntent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK |
      Intent.FLAG_ACTIVITY_CLEAR_TASK);
newIntent.putExtra("CALLER","notifyService");
PendingIntent pendingIntent = PendingIntent.getActivity(this, 0, newIntent, PendingIntent.FLAG_IMMUTABLE);
```

A Class in your APP, such as a normal intent

Like a delayed “startActivity”

```
mBuilder.addAction(R.drawable.ic_notification, "PRESS ME", pendingIntent);
```

A maximum of three buttons can be added, also media controls etc...

For more information and possibilities go to

https://developer.android.com/training/notify-user/build-notification

Android: Status Bar Notifications

Add (optional) flags for notification handling

```
mBuilder.setAutoCancel(true)
```

- Notification goes away when tapped

Send the notification to the Notification Manager

```
notificationManager.notify(0, mBuilder.build());
```

Add a **long text** and make the notification expandable

```
mBuilder.setStyle(new NotificationCompat.BigTextStyle()
            .bigText("Much longer text that cannot fit one line..."))
```

Android: Status Bar Notifications

Add a **sound** to the notification

```
mBuilder.setSound(URI sound);
```

Add **flashing lights** to the notification

```
mBuilder.setLights(0xff00ff00, 300, 100);
```

This sets a green led
The LED flashes for 300ms and turns it off for 100ms

Add a **vibration pattern** to the notification

```
mBuilder.setVibrate(long [])
mBuilder.setVibrationPattern(long []) // From API 26
```

Android: Notification Channels

Set Notification Channels from Android 8.0 (API 26)

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
    CharSequence name = getString(R.string.channel_name);
    String description = getString(R.string.channel_description);
    int importance = NotificationManager.IMPORTANCE_DEFAULT;
    NotificationChannel channel =
            new NotificationChannel(CHANNEL_ID, name, importance);
    channel.setDescription(description);
    NotificationManager notificationManager =
            getSystemService(NotificationManager.class);
    notificationManager.createNotificationChannel(channel);
}
```

Android: Notifications best practices

There is a whole world about notifications and complicated (and ever-evolving) ways to build them (*e.g.* grouping, media, progress bars, in-notification reply, …). For a complete course go to:

- 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...

Android: Background Work

PLEASE, PLEASE, PLEASE, KEEP THIS IN MIND:

“

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.

“

Android: Processes and Threads

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

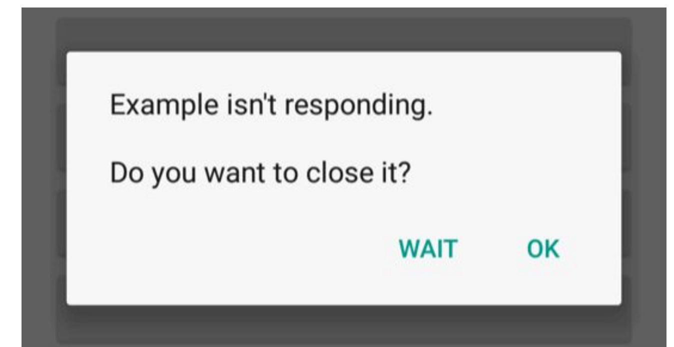
- In **Manifest.xml**, it is possible to specify the process in which a component (activity, service, receiver, provider) should run through the attribute **android:process**.

- 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

Android: Thread Management

- 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
- I the UI freezes for more than 5 secs it will be very very unpleasant



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
  - implementing the **Runnable** interface
  - AsyncTask <**deprecated**>
  - **Coroutines** (Kotlin only)

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

Android: Thread Management

```
public class MyThread extends Thread {

    public MyThread() {
        super (“My Thread”);
    }

    public void run() {
        // do your stuff
    }
}
```

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

Android: Thread Pool

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.

```
ExecutorService executorService = Executors.newFixedThreadPool(4);
```

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

- A Single Abstract Method (SAM) interface

```
executorService.execute(new Runnable() {
        @Override
        public void run() {
            // do your stuff
        }
    });         // See also Lambda notation
```

Android: Thread Management

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.

QUESTIONS:

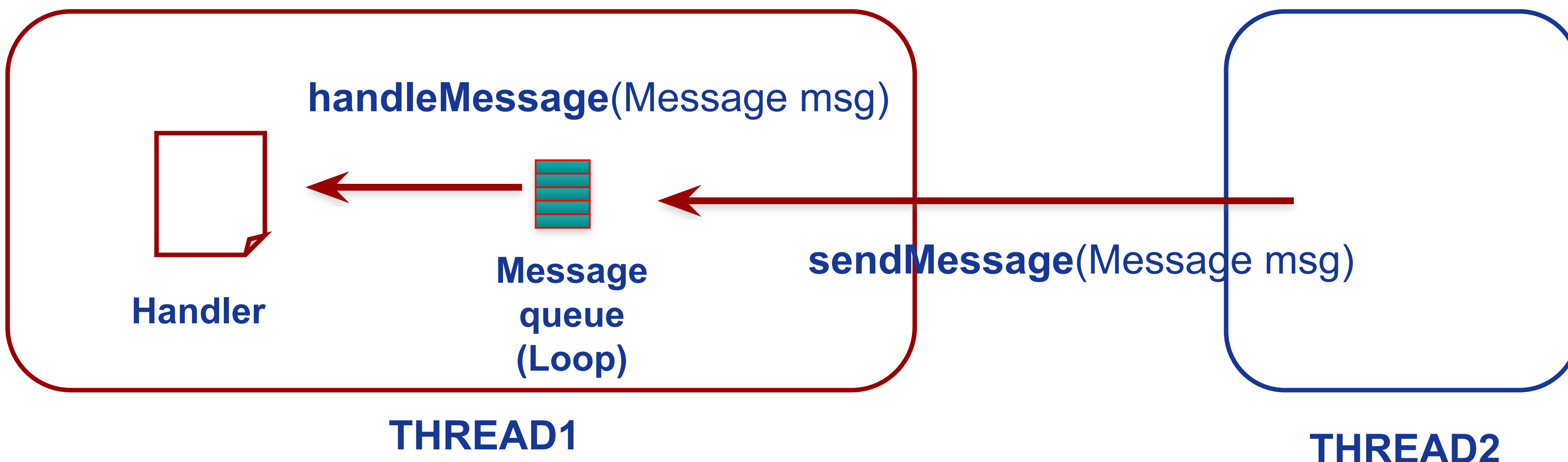
How to update the UI components from worker threads?

- Handlers and Loopers
- AsyncTask was the historical solution (now **deprecated**)
- Observables (will see it somewhere else)

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



Android: Thread Management

Message loop must be explicitly defined for worker threads.

HOW? Use **Looper** and **Handler** objects …

```
public void run() {
        Looper.prepare();                              // Instantiate the queue
        handler = new Handler(Looper.myLooper()) {
            @Override
            public void handleMessage(Message msg) {
                // handle here the message
            }
        }
        Looper.loop();                                 // Have it ready for receiving
}
```

You can use **HandlerThread** that has a Looper by default.

Android: Thread Management

You need then to target the thread's **Handler** with a message

HOW? Let's imagine the thread of the previous slide is called **mThread**

```
mThread.start();
Handler mHandler = mThread().handler;    // Assuming you can get the handler
```

You can send it a message to be handled by the **handleMessage** ....

```
Message m = mThread.handler.obtainMessage();  // new message for mHandler
m.arg1 = “Argument for the message”;
mThread.handler.sendMessage(m);
```

OR something to execute on the thread that owns the handler

```
mThread.handler.post(new Runnable() {
        @Override
        public void run() {    /* Something to do  */      }        });
```

Android: Thread Management

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

```
Handler mainThreadHandler =
                    HandlerCompat.createAsync(Looper.getMainLooper());

mainThreadHandler.post(new Runnable() {
          @Override
          public void run() {    /* Run on UI thread */       }        });
```

OR you can skip all this magic by only using

```
runOnUiThread(new Runnable() {
          @Override
          public void run() {    /* Run on UI thread */       }        });
```

Android: AsyncTask <deprecated>

AsyncTask is a Thread helper class (Android only).

- Computation running on a **background** thread.
- Results are published on the **UI** thread.
- Should be used for short operations

RULES

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

Android: AsyncTask <deprecated>

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

Must be subclassed to be used

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 MyAsyncTask extends AsyncTask<Void,Void,Void>
```

```
private class MyAsyncTask extends AsyncTask<Integer,Void,Integer>
```

Android: AsyncTask <deprecated>

EXECUTION of the ASYNCTASK

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

```
(new MyAsyncTask()).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: Services

A **Service** is a component 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 …

Register it in the manifest

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

Android: 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.
  - Wanna perform potentially blocking operations? Use Threads!

Android: Services

COMMON **MISTAKES**

- 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? Well, several reasons but the main we can think of 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.

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. **stopSelf**() → 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

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

onDestroy() ← **stopService()** **selfStop()**

Android: Intent Service

- Created for simple services
  - Does not handle multiple request simultaneously
  - Creates by default a HandlerThread where it runs
- Handles one Intent at a time
  - Through onHandleIntent()
  - Stops after the handling ended

```
public class myIntentService extends IntentService {

    public myIntentService() { super(" myIntentService");  }

    @Override
    protected void onHandleIntent(Intent intent) {          // doSomething     }
}
```

Android: Intent Service

- Start simple, if you don't need more:

  - Remember: it's a subclass of Service made simple.
  - It actually runs the body of onHandleIntent() on a separate thread by default...
  - You can override the other methods, but be sure to always return the super call.
  - onHandleIntent() is performed within the onStartCommand().
  - destroyed after the Intent has been handled.

Android: More Complex Services

- If we want to create a more complex Service that handles multiple stuff then we might want to:
  - Run a **HandlerThread** and get its **Looper**
  - Implement a **Handler** in our Service that will run tasks in the thread when received by the Looper.
    - The handler may also handle the stopping of both the thread and the service.
  - Tell what we should do if the Service is killed by the system
    - Look for the return flag in the onStartCommand():
      - START_STICKY | START_NOT_STICKY | START_REDELIVER_INTENT

Android: IntentService implementation (example)

```
public class HelloService extends Service {
  private Looper serviceLooper;
  private ServiceHandler serviceHandler;

  private final class ServiceHandler extends Handler {
      public ServiceHandler(Looper looper) {
          super(looper);
      }
      @Override
      public void handleMessage(Message msg) {
              // DO YOUR STUFF
              Thread.currentThread().interrupt();
          }
          // Stop the service using the startId
          stopSelf(msg.arg1);
      }
  }

  @Override
  public void onCreate() {
    HandlerThread thread = new
        HandlerThread("ServiceStartArguments",
        Process.THREAD_PRIORITY_BACKGROUND);
      thread.start();
      serviceLooper = thread.getLooper();
      serviceHandler = new ServiceHandler(serviceLooper);
  }

  @Override
  public int onStartCommand(Intent intent, int flags, int
startId) {

      Message msg = serviceHandler.obtainMessage();
      msg.arg1 = startId;
      serviceHandler.sendMessage(msg);
      // If killed we restart with a NULL intent
      return START_STICKY;
  }

  @Override
  public void onDestroy() {
        Toast.makeText(this, "service done",
              Toast.LENGTH_SHORT).show();
  }
}
```

Android: Foreground Services

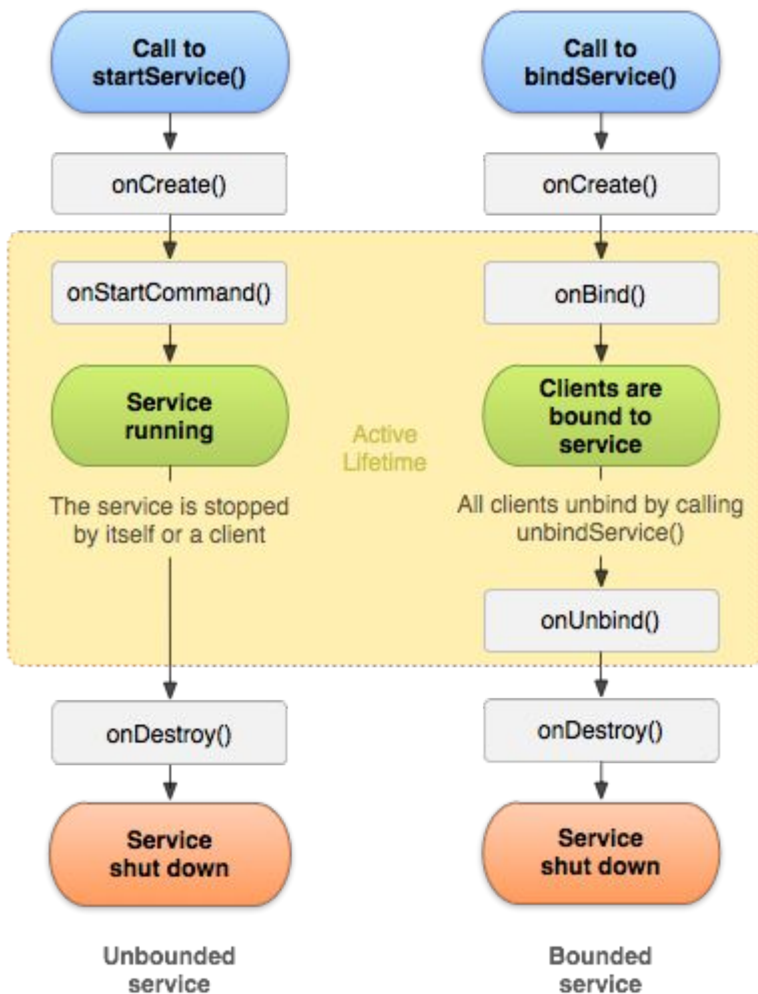
- A **Foreground** Service is a service that is continuously active in the Status Bar, and thus it is not a good candidate to be killed in case of low memory.

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

- To create a Foreground Service:
  - Create a **Notification** object
  - Call **startForeground**(id, notification) from onStartCommand()
  - Call **stopForeground**() to bring it to the background.

Note that you need FOREGROUND_SERVICE permission

Services and BoundServices



- 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.

Android: Bound Service

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

bindService(Intent, ServiceConnection, flags)

Service

Component
(e.g. Activity)

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)

Android: Bound Service

- 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.

Android: Bound Service

```
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;
    }

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

Android: Bound Service

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

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

        @Override
        public void onServiceDisconnected(ComponentName arg0) {
        }
    };

… bindService(new  Intent(this,LocalService.class), mConnection, BIND_AUTO_CREATE);
… lService.myFunction();
```

Android: Broadcast Receiver

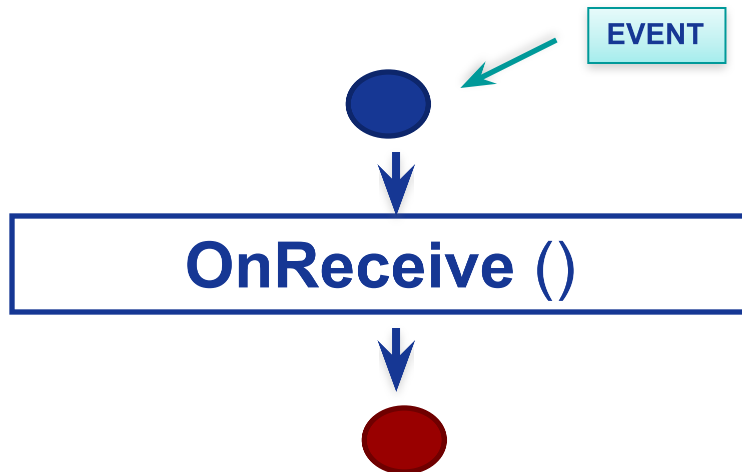
A **Broadcast Receiver** is a component that is activated only when specific events occur (i.e. SMS arrival, phone call, etc).

The Event is an **Intent**

- **Registration** of the Broadcast Receiver to the event …
  - Registration through **XML** code (Manifest-declared)
  - Registration through **Java** code (Context-declared)
- **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).



- 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>
```

Android: Broadcast Receiver

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);
}
```

Android: Broadcast 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() work on different contexts!

Android: Broadcast Receiver

onReceive() should be short enough. If you need more time to process the reaction it may be a good idea to:

- Trigger an IntentService within the onReceive().
- Register the receiver in the context of a long-running Service.
- Use WorkManager (will see it later).