Lab on apps development for tablets, smartphones and smartwatches

Week 6(b): Notifications alarms, and (7) background tasks

Dr. Marina Zapater, Prof. David Atienza

Mr. Grégoire Surrel, Ms. Elisabetta de Giovanni,
Mr. Dionisijie Sopic, Ms. Halima Najibi, Ms. Farnaz Forooghifar

Embedded Systems Laboratory (ESL) – Faculty of Engineering (STI)

Course evaluation: Mid-term and exam

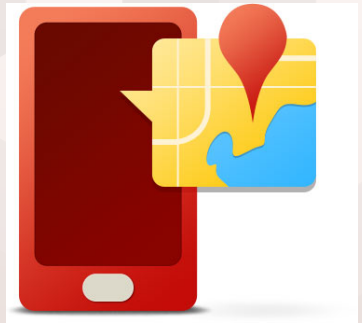
- **Mid-term exam → December 11th at 15h**
  - Theory from “Week 1” to “Week 6b” (included!)
    - (1) Basics, (2) Views and Layouts, (3) Activities and intents, (4) Lifecycle and user interface, (5) Data Management, (6) Sensors, alarms, notifications
  - Example mid-term and more info about logistics next week!

- **Oral exam:** Project evaluation → **January 24th and 25th**
  - For the moment, please book both days from 9h to 18h
  - You will be assigned a 15min slot for your exam
    - 10min presentation + 5min questions
  - If you have any incompatibility for any of those two days, send an e-mail to marina.zapater@epfl.ch

- Advanced exam for very specific cases → **December 17th 15h-17h**

Outline of the class

- **System services** and Broadcast receivers
  - **Alarms**
  - Notifications
- Background tasks
  - AsyncTask
  - AsyncTaskLoader
- Services:
- Today's lab!



System Services

- We can create our own services or use the **system services**
- There is a wide list of system services available
  - Sensor Service
  - ***Notification Service***
  - ***Alarm Service*** → **Used together with Broadcast Receivers**
  - Power Manager Service
  - Vibrator Service, Audio Service
  - Telephony Service, Connectivity Service, Wi-Fi Service
  - ...
- We can access the system services programmatically

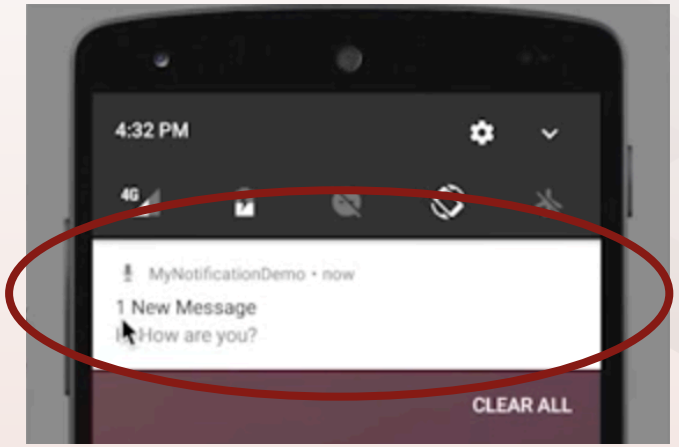
```
SensorManager sm = (SensorManager) getSystemService(SENSOR_SERVICE);
```

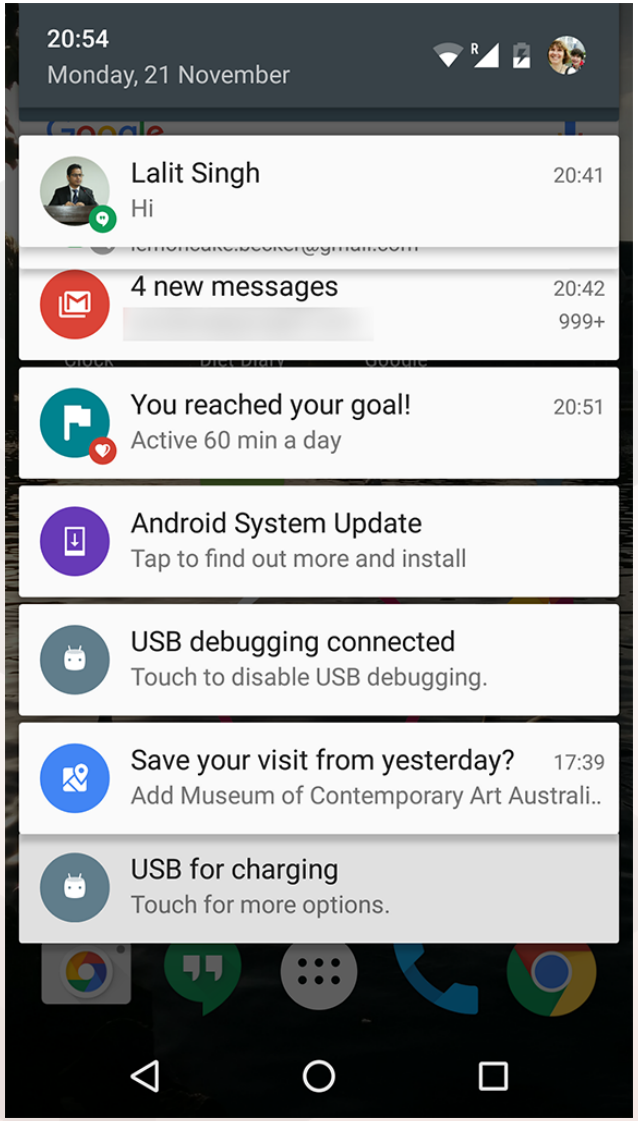
```
NotificationManager mNotifMgr = (NotificationManager) getSystemService (NOTIFICATION_SERVICE);
```

```
AlarmManager myAlarmManager = (AlarmManager) getSystemService(ALARM_SERVICE);
```

Notifications

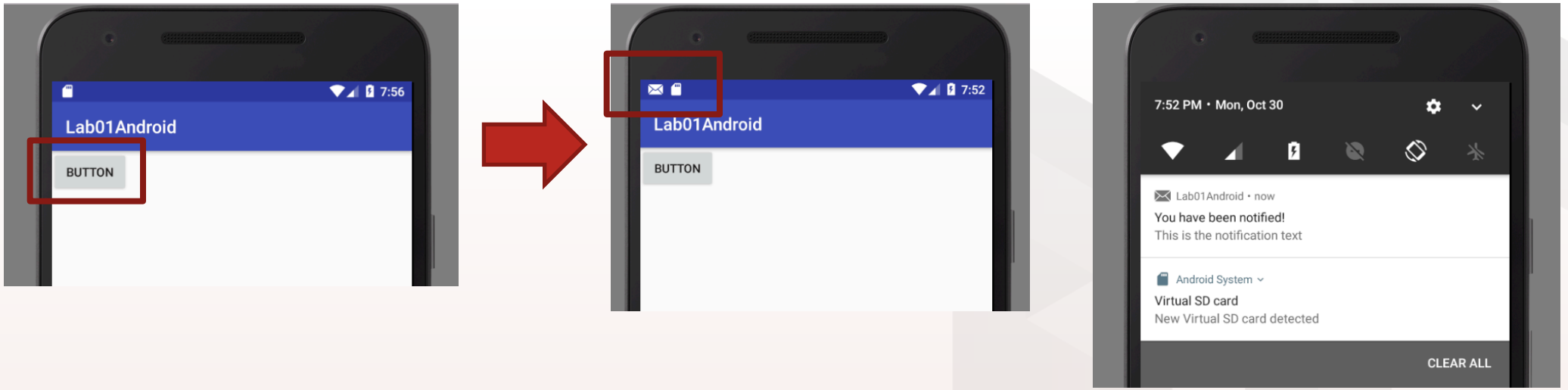
- What is a notification?
  - A message displayed outside of the regular application UI.
    - It does not interrupt operations on the foreground activity!
  - To see details, user opens the notification drawer
  - Consists on
    - Small icon
    - Title
    - Detail text





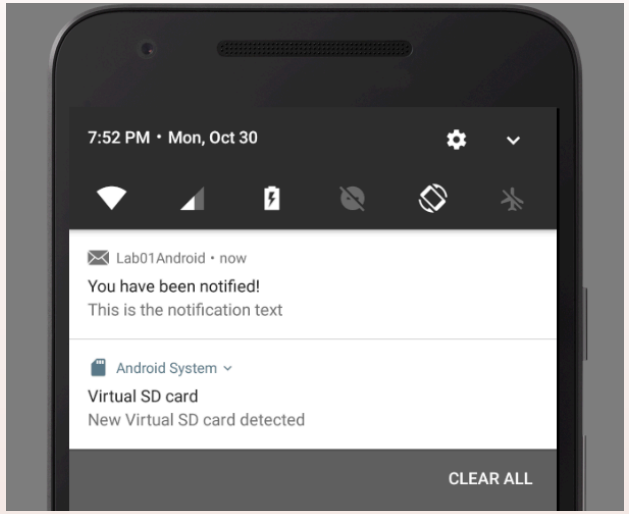
A very simple example

- Application that launches a notification when we press a button



Steps to create a Notification programmatically

1. Getting a reference to NotificationManager
   - NotificationManager is a system Service
2. Building the Notification message
   - Create notification ID!
3. Send the notification to the Notification Manager




```java
public class MainActivity extends AppCompatActivity {

    // Usually at the top of the class
    private String TAG = "MainActivity";
    private NotificationCompat.Builder mNotifyBuilder;
    private NotificationManager mNotifyManager;
    private static final int NOTIFICATION_ID=0;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
1.      //connecting to the Notification manager!
        mNotifyManager = (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
    }

    public void methodCalledOnClickButton(View view) {
2.      //I added a button to the UI, and linked button to this method with android:onClick
        //Now I will define the notification!
        mNotifyBuilder = new NotificationCompat.Builder(this);
        mNotifyBuilder.setContentTitle("You have been notified!");
        mNotifyBuilder.setContentText("This is the notification text");
        mNotifyBuilder.setSmallIcon(android.R.drawable.ic_dialog_email);

3.      Notification myNotification = mNotifyBuilder.build();
        mNotifyManager.notify(NOTIFICATION_ID, myNotification);
    }
}
```

Actions and pending intents

- Notifications must be able to perform actions on behalf of your app.
  - Include specific actions inside the Notification UI
  - Launch action when the notification is tapped
  - OK for the action to just open an Activity in your app

- … or being able to click “Reply”
  - Adding a button to notification, that will open an activity:

- In our example, we tap on the notification and the activity is re-opened
  - We use a “PendingIntent”

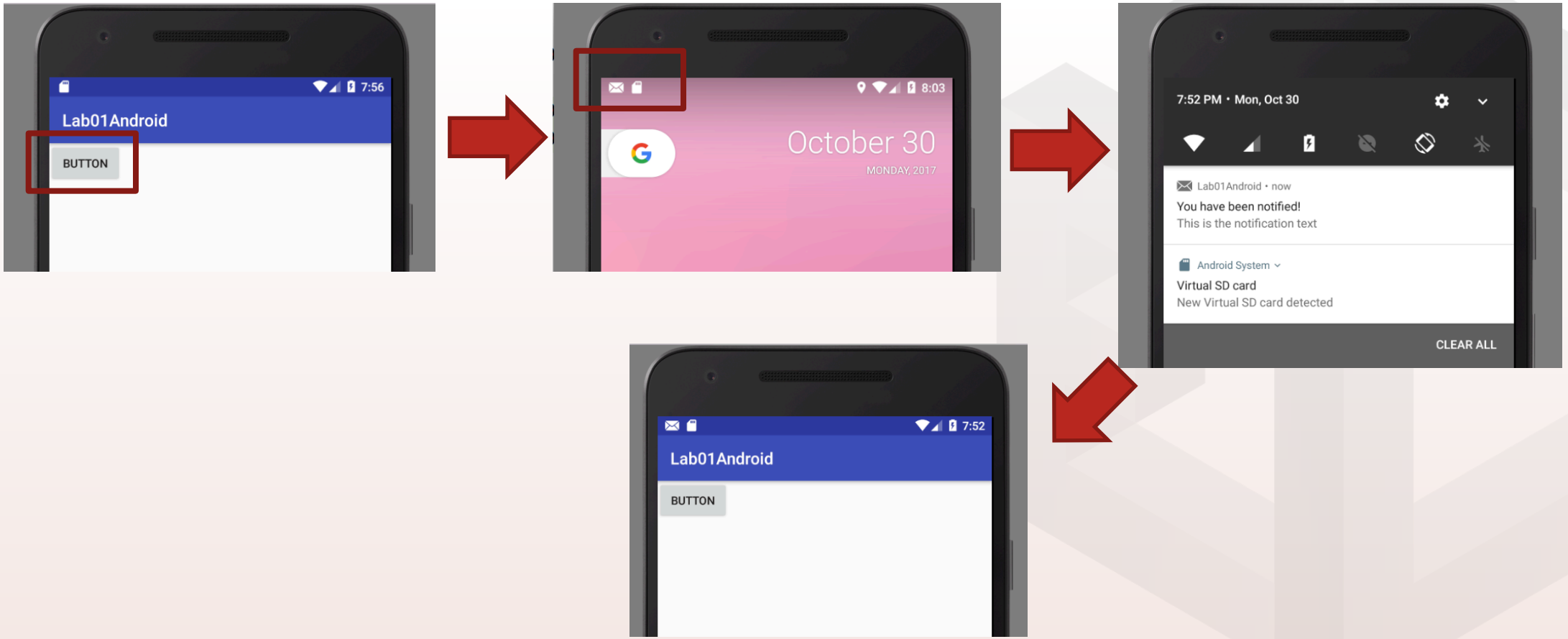
Creating a PendingIntent

1. Create an intent
2. Create a PendingIntent.
   - Types:
     1. PendingIntent.getActivity()
     2. PendingIntent.getBroadcast()
     3. PendingIntent.getService()
   - All previous methods take 4 params:
     - Application context
     - Request code
     - Intent to be delivered
     - PendingIntent flag
3. Add it to notification builder
   - And then call notify()

```
public class MainActivity extends AppCompatActivity {

    // Usually at the top of the class
    private String TAG = "MainActivity";
    private NotificationCompat.Builder mNotifyBuilder;
    private NotificationManager mNotifyManager;
    private static final int NOTIFICATION_ID=0;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        //connecting to the Notification manager!
        mNotifyManager = (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
    }


    public void methodCalledOnClickButton(View view) {
        //I added a button to the UI, and linked button to this method with android:onClick
        //Now I will define the notification!
        mNotifyBuilder = new NotificationCompat.Builder(this);
        mNotifyBuilder.setContentTitle("You have been notified!");
        mNotifyBuilder.setContentText("This is the notification text");
        mNotifyBuilder.setSmallIcon(android.R.drawable.ic_dialog_email);

        //I just added the PendingIntent so that if you click on notification,
        //this activity will open again!
        Intent notificationIntent = new Intent(this, MainActivity.class);
        PendingIntent pd = PendingIntent.getActivity(this, 1, notificationIntent, 0);
        mNotifyBuilder.setContentIntent(pd);

        //Trigger notification
        Notification myNotification = mNotifyBuilder.build();
        mNotifyManager.notify(NOTIFICATION_ID, myNotification);

        finish();
    }
}
```

How does our example change now?

- When you launch the Nofitication, the activity finishes
- When you click on the notification, you relaunch the activity!



Updating and cancelling notifications

- Updating a notification
  1. Issue notification with updated parameters using builder
  2. Call notify() passing in the same notification ID
  - If previous notification is still visible, system updates
  - If previous notification has been dismissed, new notification is created

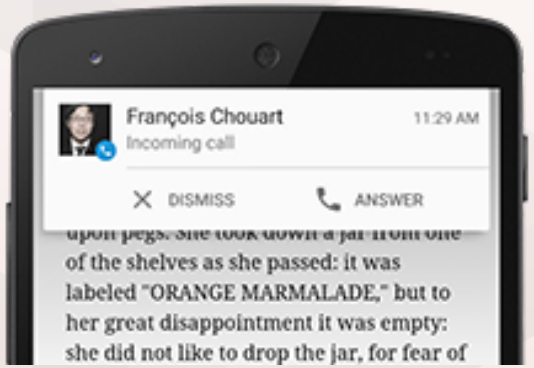
- Cancelling notifications
  - Users can simply dismiss notifications
  - User launches Content Intent with setAutoCancel() enabled
  - App calls cancel() or cancelAll() on NotificationManager
    - mNotifyManager.cancel(NOTIFICATION_ID);

Notification priorities

- Determines how the system displays the notification with respect to other notifications

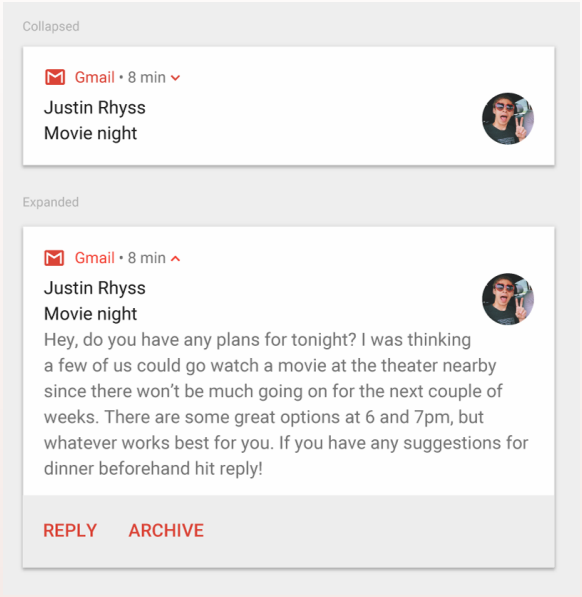
- Use NotificationCompat.Builder.setPriority()
  - .setPriority(NotificationCompat.PRIORITY_HIGH)
  - PRIORITY_MIN (-2) to PRIORITY_MAX (2)

- Priority above 0 triggers heads-up notification on top of current UI
  - Used for important notifications such as phone calls
  - Use lowest priority possible

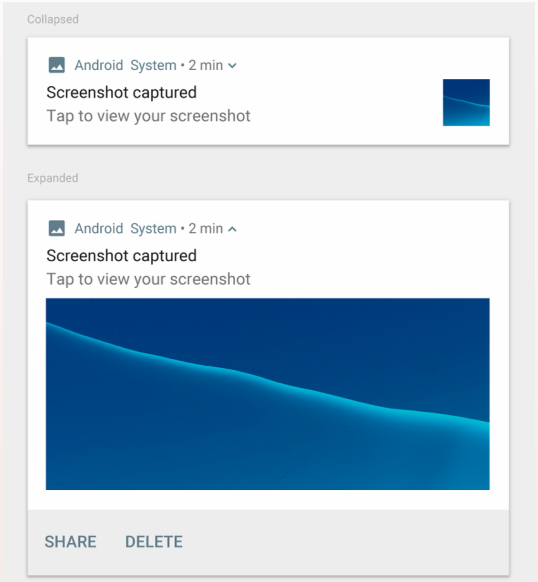


Common notification layouts

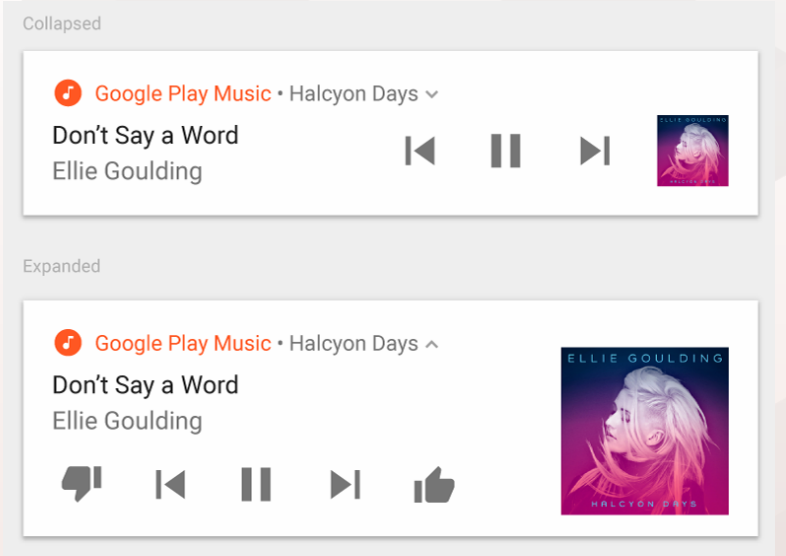
- Several different styles can be used
  - mNotifyBuilder.setStyle (new NotificationCompat.BigPictureStyle().
    bigPicture(myBitmapImage).
    setBigContentTitle("Notification!"));



NotificationCompat.BigTextStyle



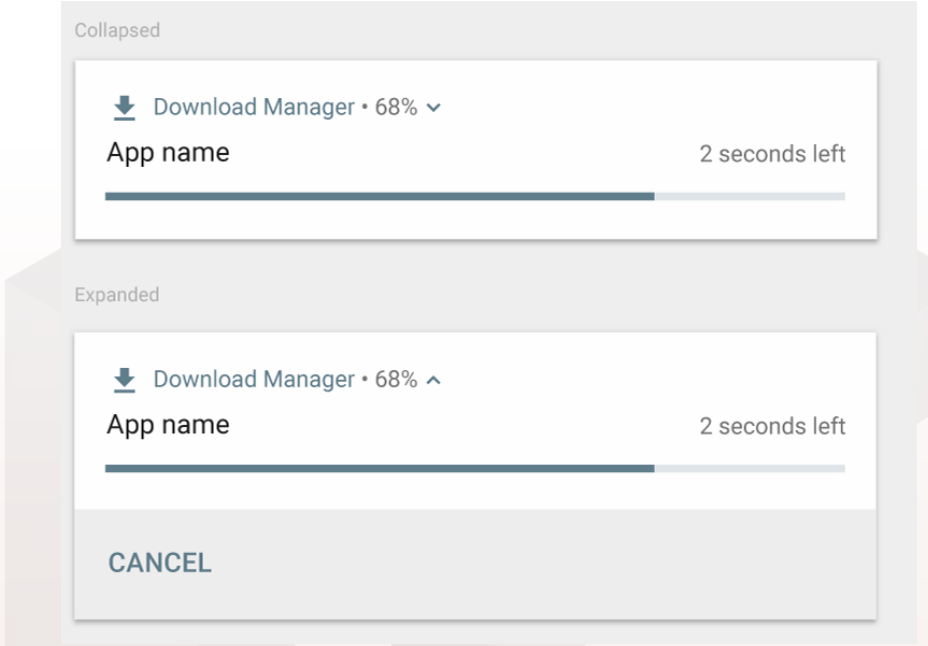
NotificationCompat.BigPictureStyle



NotificationCompat.MediaStyle

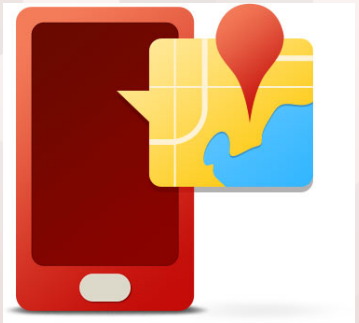
Progress bar

- Progress bar for ongoing task that can be cancelled
  - This one is not a style
  - .setProgress(100, incr, false)



Outline of the class

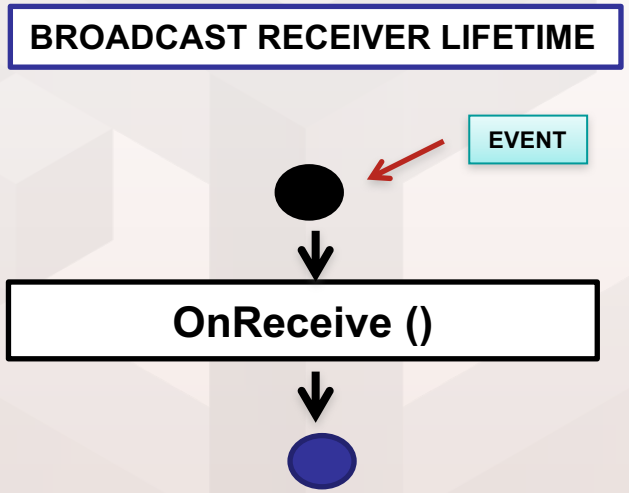
- System services and **Broadcast receivers**
  - Alarms
  - **Notifications**
- Background tasks
  - AsyncTask
  - AsyncTaskLoader
- Services:
- Today's lab!



Broadcast Receiver

- A **Broadcast Receiver** is a component that is activated only when specific events occur (i.e. SMS arrival, phone call, etc.)
  - Responds even when your app is closed
  - Independent from any activity
  - When an event is delivered to onReceive(), it has 5 seconds to be executed
  - Afterwards, then the receiver is destroyed

- System broadcasts: delivered under certain events:
  - After the system ends booting:
    - android.intent.action.BOOT_COMPLETED
  - When the WiFi state changes:
    - android.intent.action.WIFI_STATE_CHANGED



Steps for creating Broadcast Receivers

1. Creating a new class that extends BroadcastReceiver
2. Implementing the onReceive() method
   - Handling the event
3. Registration of the Broadcast Receiver to the event
   - Event → **Intent**
   - Registration through **XML** code
     - Statically, in the Android Manifest
   - Registration through **Java** code
     - Dynamically, with registerReceiver()

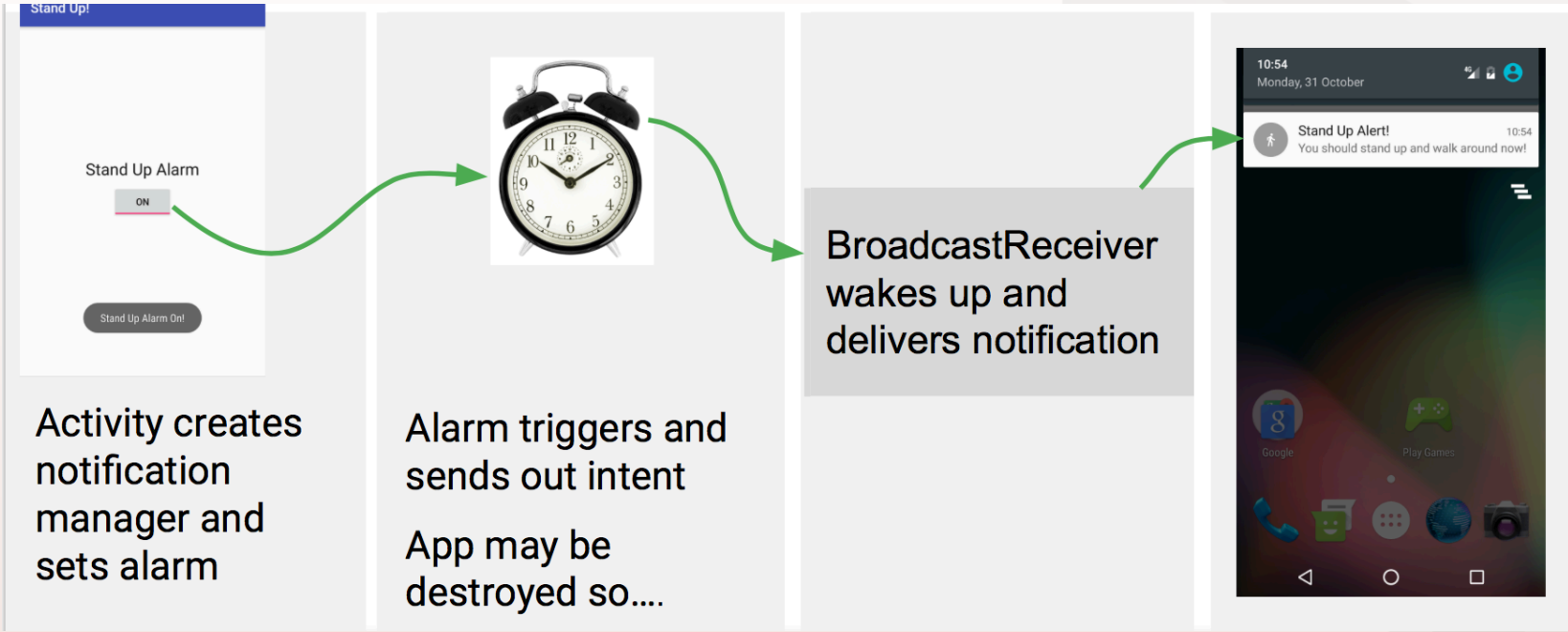
Let's see how we use them with Alarms and Notifications!

```
public class StandUp extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {


    }
}
```

```
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="Lab01Android"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity"...>

        <receiver
            android:name=".StandUp"
            android:enabled="true"
            android:exported="true">
            <intent-filter>
                <action android:name="com.example.zapater.myown.receiver.Message"></action>
                <category android:name="android.intent.category.DEFAULT"></category>
            </intent-filter>
        </receiver>

        <activity android:name=".StandUpActivity"></activity>
    </application>

</manifest>
```

Alarms

- What is an Alarm in Android? → **Not an actual alarm clock**
  - Schedules something to happen at a set time
    - Fire intents at set times or intervals → Goes off once or recurring
  - App does not need to run for alarm to be active
  - Use with BroadcastReceiver to start services and other operations



Types of Alarms and behaviors

- Measuring time
  - Elapsed Real Time: time since system boot → Use when possible!
    - Independent of time zone
    - Used to measure intervals and relative time
    - Elapsed time includes time device was asleep
  - Real Time Clock (RTC)—UTC (wall clock) time
    - When time of day at local time zone matter

- Wake up behavior
  - Wakes up device if screen is off
    - Use only for critical operations
    - Can drain battery!
  - Does not wake up device
    - Fires next time device is awake
    - Polite ☺

	Elapsed Real Time (ERT)—since system boot	Real Time Clock (RTC)—time of day matters
Do not wake up device	`ELAPSED_REALTIME`	`RTC`
Wake up	`ELAPSED_REALTIME_WAKEUP`	`RTC_WAKEUP`

Alarm Best Practices

- Imagine an app with millions of users
  - Server sync operation based on clock time
  - Every instance of app syncs at 11:00 p.m.

Load on the server could result in high latency or even "denial of service"!!!

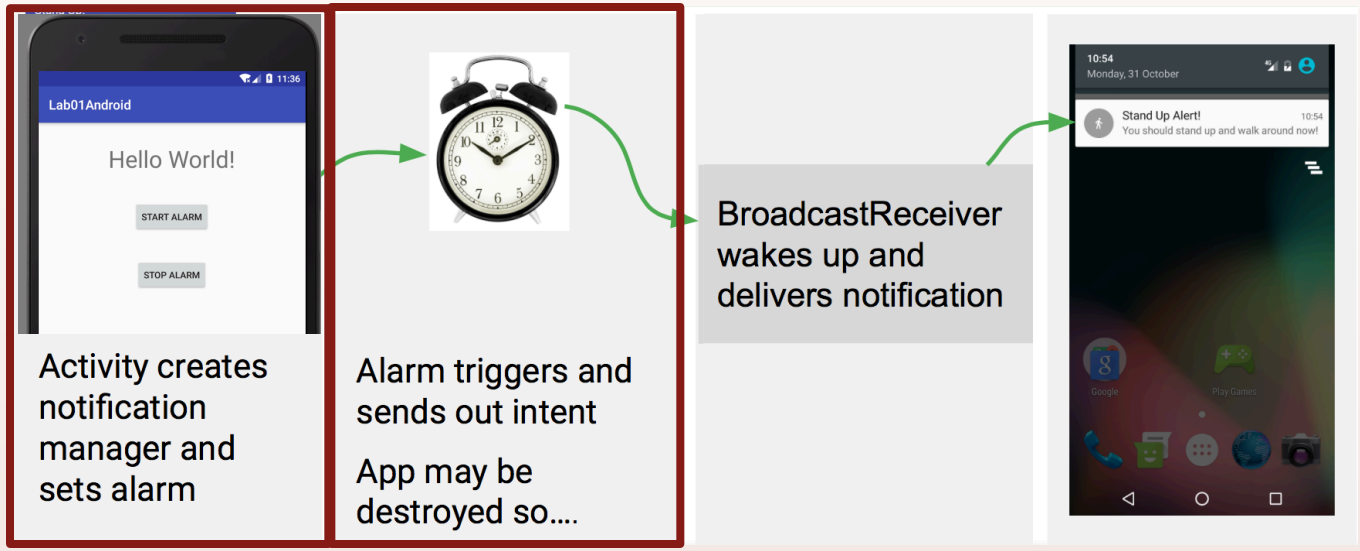
- Add randomness to network requests on alarms
  - Minimize alarm frequency
  - Use ELAPSED_REALTIME, not clock time, if you can

- Saving battery
  1. minimize waking up the device
  2. Use Inexact alarms → setInexactRepeating()
     - Android synchronizes multiple repeating alarms and fires them at the same time

Steps to create an Alarm: MainActivity

1. Creating AlarmManager on the Activity
   - Provides access to system alarms and services
2. We use a PendingIntent that we associate to a broadcast receiver
   - We set a repeating alarm with 4 parameters:
     - 1) Type of Alarm, 2) Time to trigger,
       3) Interval for repeating, 4) the PendingIntent

1. 2.



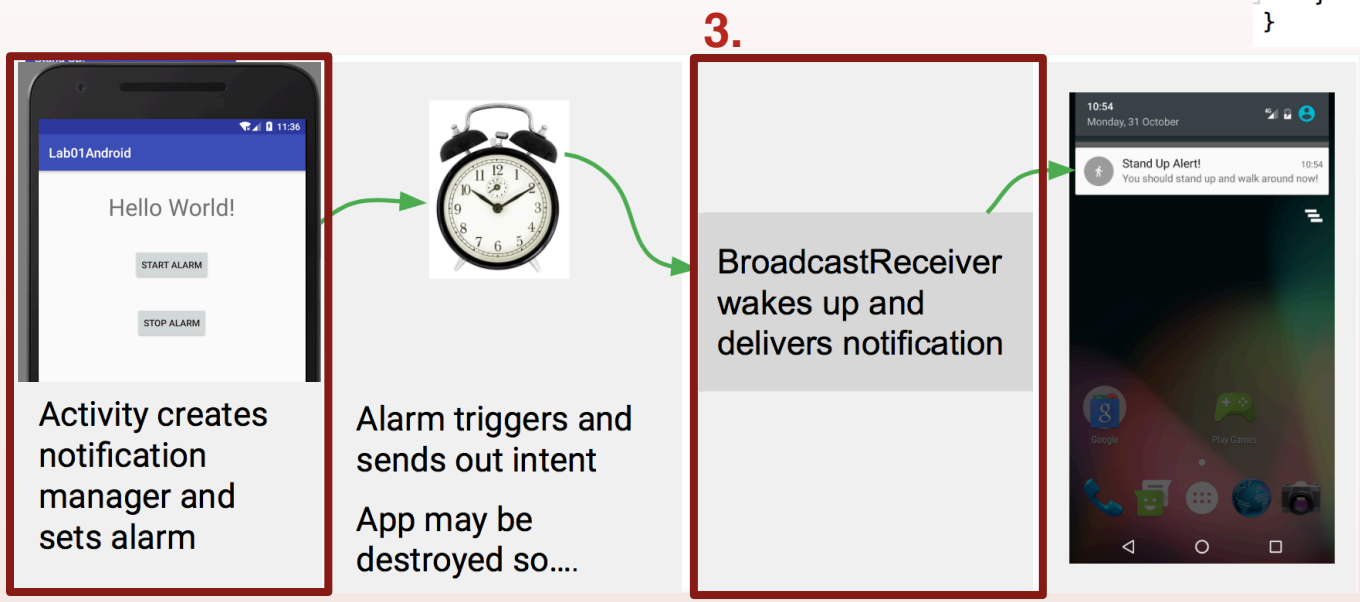

```java
public class MainActivity extends AppCompatActivity {

    // Usually at the top of the class
    private String TAG = "MainActivity";

1.  AlarmManager myAlarmManager;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        //Creating Alarm Manager
        myAlarmManager = (AlarmManager) getSystemService(ALARM_SERVICE);
    }

2.  public void startSomething(View view) {
        Intent i1 = new Intent();
        i1.setAction("com.example.zapater.myown.receiver.Message");
        i1.addCategory("android.intent.category.DEFAULT");
        PendingIntent pd = PendingIntent.getBroadcast(this, 0, i1, 0);

        //Setting the Alarm -> the alarm will repeat itself every 30sec
        myAlarmManager.setRepeating(AlarmManager.RTC_WAKEUP,
                System.currentTimeMillis(), 1000 * 5, pd);

    }

    public void stopSomething(View view) {
        //Cancelling the alarm
        Intent i1 = new Intent();
        i1.setAction("com.example.zapater.myown.receiver.Message");
        i1.addCategory("android.intent.category.DEFAULT");
        PendingIntent pd = PendingIntent.getBroadcast(this, 0, i1, 0);

        myAlarmManager.cancel(pd);
    }
}
```

Steps to create an Alarm: Broadcast receiver

3. Create a BroadcastReceiver

- The broadcast receiver uses a notification manager
  - And in this case calls another activity (example)
- Remember to add the intent-filter to the manifest.xml file

```
public class StandUp extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {
        //Create notification Code
        NotificationManagerCompat mNotifyManager = NotificationManagerCompat.from(context);

        NotificationCompat.Builder myNotif = new NotificationCompat.Builder(context);
        myNotif.setContentTitle("Stand UP notification");
        myNotif.setContentText("You need to stand up now!");
        myNotif.setSmallIcon(android.R.drawable.ic_dialog_alert);

        Intent i1 = new Intent(context, StandUpActivity.class);
        PendingIntent pd = PendingIntent.getActivity(context, 0, i1, 0);
        myNotif.setContentIntent(pd);
        myNotif.setAutoCancel(true);

        mNotifyManager.notify(1,myNotif.build());

    }
}
```




```
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="Lab01Android"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity"...>

        <receiver
            android:name=".StandUp"
            android:enabled="true"
            android:exported="true">
            <intent-filter>
                <action android:name="com.example.zapater.myown.receiver.Message"></action>
                <category android:name="android.intent.category.DEFAULT"></category>
            </intent-filter>
        </receiver>

        <activity android:name=".StandUpActivity"></activity>
    </application>

</manifest>
```

Cancelling and Clearing Alarms

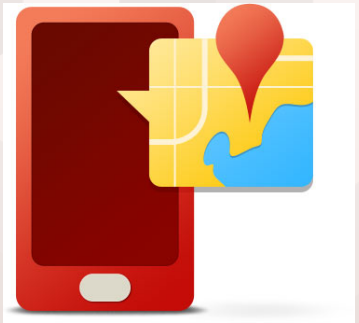
- Call cancel() on the Alarm Manager
  - pass in the PendingIntent
    - alarmManager.cancel(alarmPendingIntent);

- Alarms and Reboots
  - Alarms are cleared when device is off or rebooted
  - Use a BroadcastReceiver registered for the BOOT_COMPLETED event and set the alarm in the onReceive() method

```
    public void stopSomething(View view) {
        //Cancelling the alarm
        Intent i1 = new Intent();
        i1.setAction("com.example.zapater.myown.receiver.Message");
        i1.addCategory("android.intent.category.DEFAULT");
        PendingIntent pd = PendingIntent.getBroadcast(this, 0, i1, 0);

        myAlarmManager.cancel(pd);
    }
```

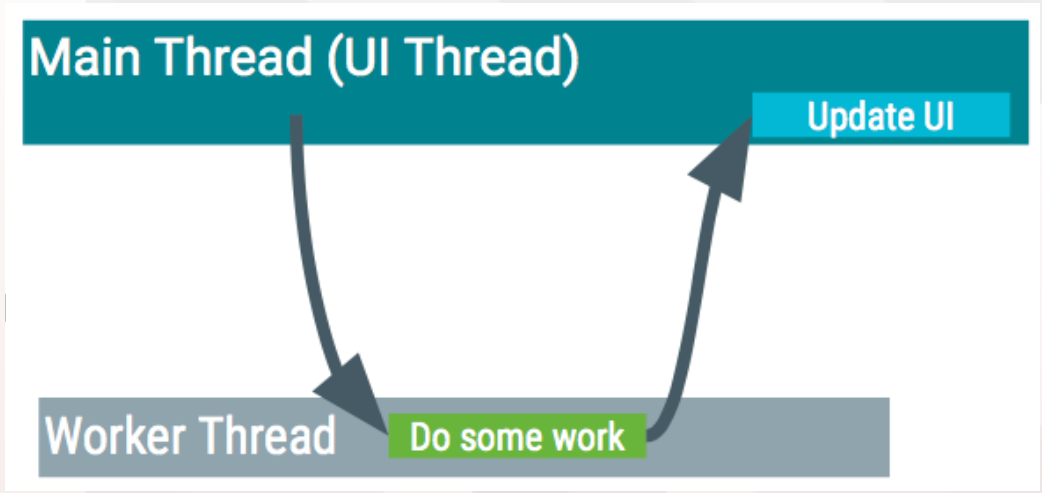
Outline of the class

- System services and Broadcast receivers
  - Alarms
  - Notifications
- **Background tasks**
  - AsyncTask
  - AsyncTaskLoader
- Services:
- Today's lab!



Background threads

- We execute the long-running tasks on a **background thread**
- "Three" types: 2 tasks + services
  - AsyncTask
    - very short task or low priority
    - no result returned to UI
    - can be interrupted
  - AsyncTaskLoader
    - for longer tasks
    - returns result to UI
  - Services
- Two rules:
  - Never block the UI thread (16ms!)
  - Do not access the Android UI toolkit from outside the UI thread

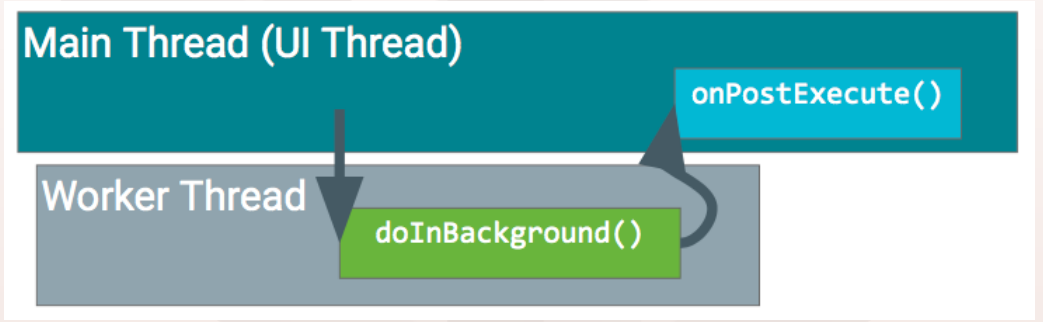


AsyncTask class

- Used to implement basic background task
  - Create a new object of class AsyncTask to take care of a task → for instance, Room DB!

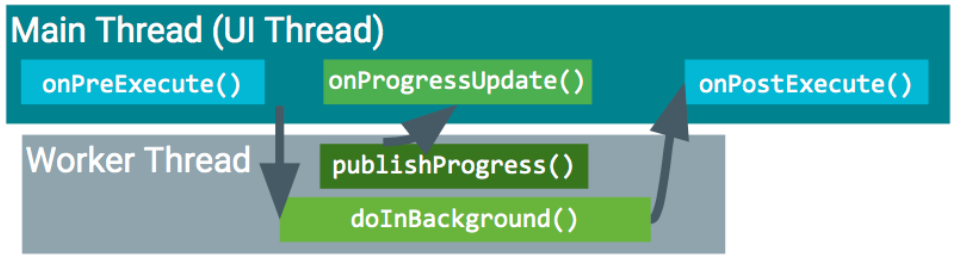
- We need to Override two methods:
  - doInBackground() → runs on background thread
    - Does all the work that needs to happen on the background
  - onPostExecute() → runs on main thread when work is done
    - Processes results and publishes them to the UI

- Two helper methods:
  - onPreExecute():
    - Runs on main thread and sets up task
  - onProgressUpdate():
    - Runs on main thread
    - Receives calls from background thread



Creating an AsyncTask

1. Extending from the AsyncTask class
2. Provide data type sent to doInBackground()
3. Provide data type of progress units for onProgressUpdate()
4. Provide data type of result for onPostExecute()




```
private class MyAsyncTask
1.  extends AsyncTask<String, Integer, Bitmap> {...}
```

2. doInBackground()
3. onProgressUpdate()
4. onPostExecute()

Example: today's lab

```
public class MyAsyncTask extends AsyncTask<Void, Void, SensorData>{

    protected SensorData doInBackground(Void... params) {
        SensorData sensorData = new SensorData();
            //Do something, like storing in DB!
            return sensorData;
        }

        @Override
        protected void onPostExecute(SensorData sensorData) {
            //Update a TextView, for example!
            if (R.id.textView != 1) {
                //do something
            }
        }

}
```

Limitations of AsyncTask

- When device configuration changes, Activity is destroyed…
  - AsyncTask cannot connect to Activity anymore
  - New AsyncTask created for every config change
  - Old AsyncTasks stay around
- App may run out of memory or crash!

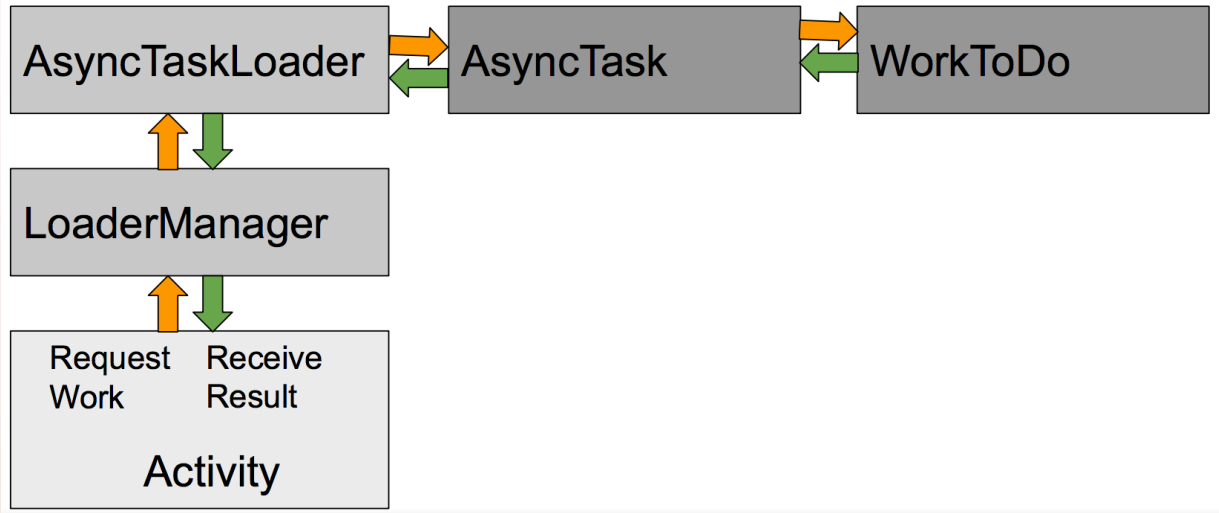
- For long-running tasks… we use AsyncTaskLoader

AsyncTaskLoader

- Provides asynchronous loading of data
  - **Reconnects to Activity after configuration change**
  - Can monitor changes in data source and deliver new data
  - Callbacks implemented in Activity

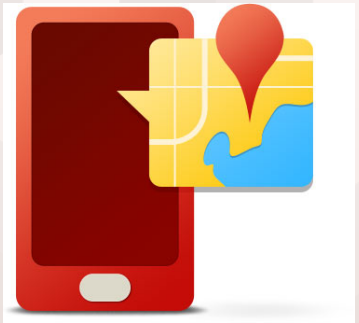
- Similarities between AsyncTask and AsyncTaskLoader
  - doInBackground() → loadInBackground()
  - onPostExecute() → onLoadFinished()

- To implement it:
  1. Subclass AsyncTaskLoader
  2. Implement the constructor
  3. loadInBackground()
  4. onStartLoading()



Outline of the class

- System services and Broadcast receivers
  - Alarms
  - Notifications
- Background tasks
  - AsyncTask
  - AsyncTaskLoader
- **Services:**
- Today's lab!



Quick recap: Services

- A **Service** is an application that can perform long-running operations in background and does not provide an user interface
- A Service provides a robust environment for background tasks
- Activity vs. service
  - Activity: UI, can be disposed when it loses visibility
  - Service: No UI, disposed when it terminates
- We use them for
  - Network transactions, playing music, perform file I/O, interact with content providers…

Types of services

- **Started services:**
  - Foreground service:
    - Performs an operation noticeable by the user
      - Example: music player using music service
    - Runs in the background but requires that the user is actively aware of it.
    - Provide a notification that **the user cannot dismiss** while the service is running

  - Background service:
    - Performs an operation that isn't directly noticed by the user

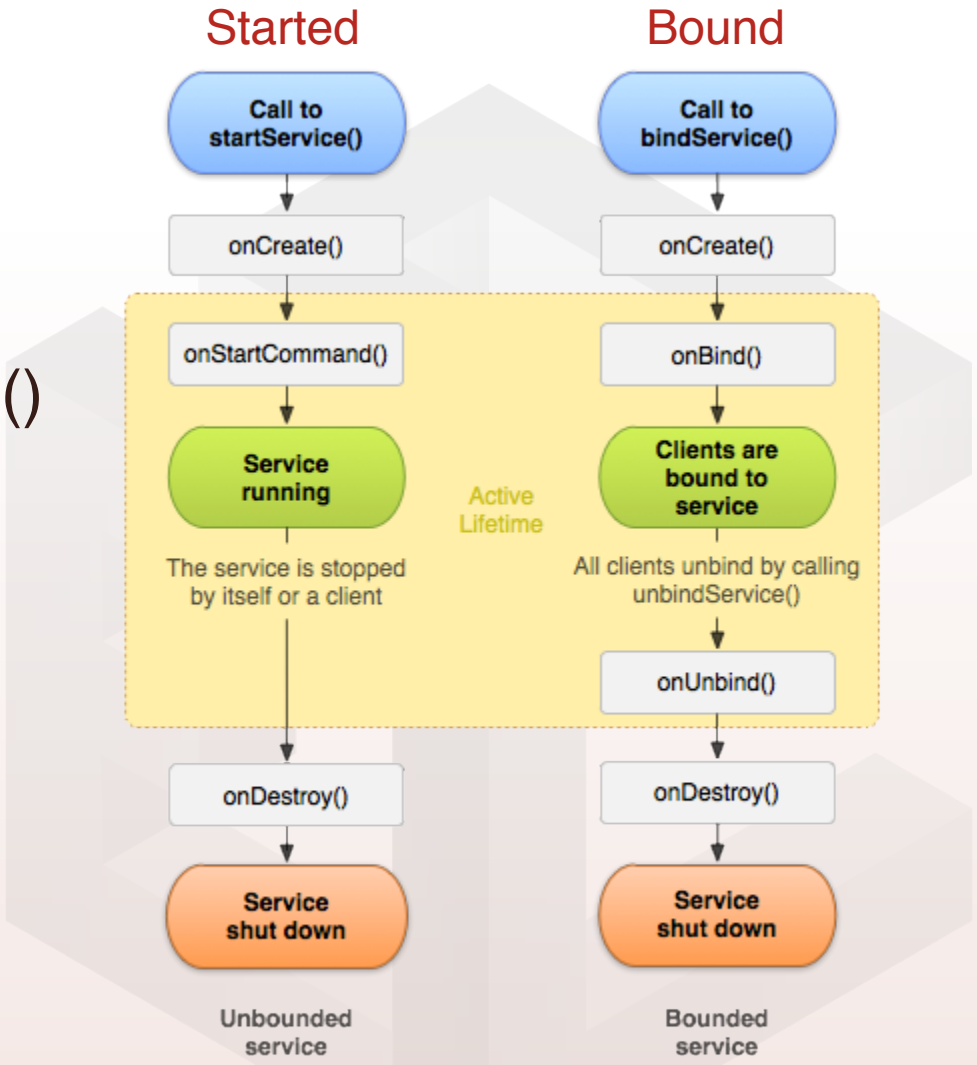
- **Bound services:**
  - A service is *bound* when an application component binds to it by calling bindService().

- Note: Although services are separate from the UI, they still run on the main thread by default (except IntentService)

Started vs. bounded Services

- Started
  - Started with startService()
  - Runs indefinitely (even if the component who started it is destroyed)
  - … until it stops itself with selfStop()
  - or is terminated by another via stopService()

- Bound
  - Offers a client-server interface that allows components to interact with the service
  - Clients send requests and get results
  - Started with bindService()
  - Ends when all clients unbind



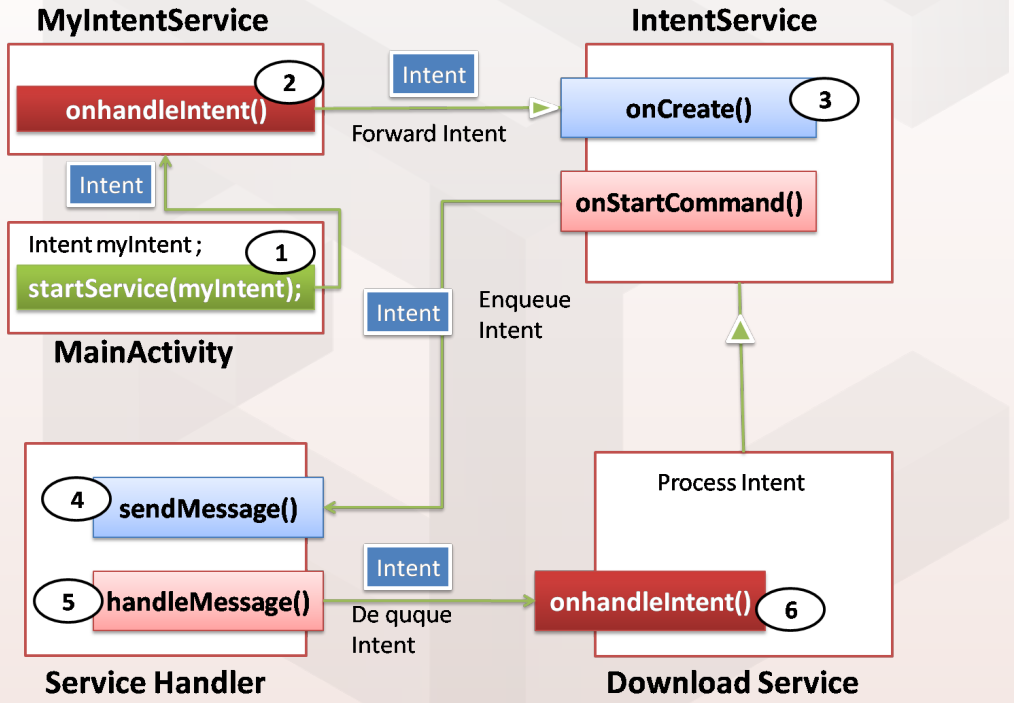
IntentService

- Simple service with simplified lifecycle
- Uses worker threads to fulfill requests
- Stops itself when done
- Ideal for one long task on a single background thread

- We simply create the class and implement the onHandleIntent()

- Limitations:
  - Cannot interact with the UI
  - Can only run one request at a time
  - Cannot be interrupted

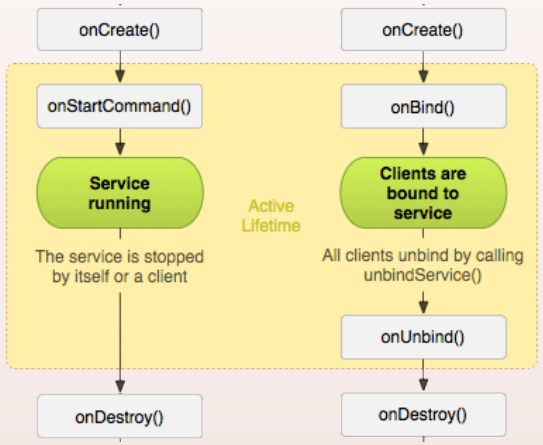
```
public class HelloIntentService extends IntentService {
    public HelloIntentService() { super("HelloIntentService");}

    @Override
    protected void onHandleIntent(Intent intent) {
        // Do some work
        // When this method returns, IntentService stops the service
    }

}
```



Creating a service

1. Declaring the service in the manifest
2. Giving the right permissions
3. Extending from IntentService or Service class
4. Implement lifecycle methods
5. Start service from activity
6. Make sure service is stoppable

4.



In Manifest.xml:

1.

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

Create Service class

3a.

```
public class HelloService extends Service {
  private Looper mServiceLooper;
  private ServiceHandler mServiceHandler;
```

3b.

```
public class HelloIntentService extends IntentService {
```

In MainActivity (or other activity):

5.

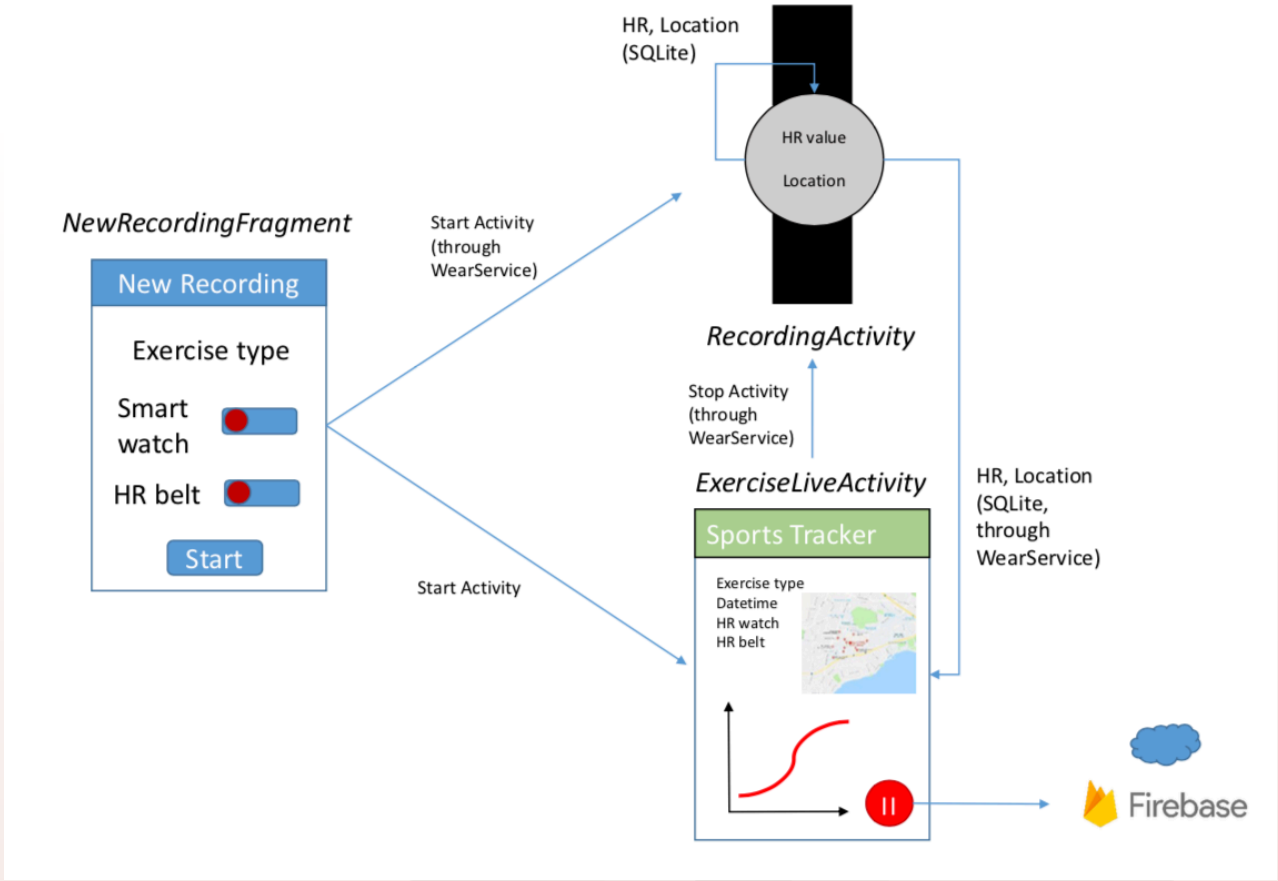
```
Intent intent = new Intent(this, HelloService.class);
startService(intent);
```

Stopping a service

- A **started service** must manage its own lifecycle
- If not stopped, will keep running and consuming resources
- The service must stop itself by calling stopSelf()
- Another component can stop it by calling stopService()
- **Bound service** is destroyed when all clients unbound
- **IntentService** is destroyed after onHandleIntent() returns

Today's lab

- Room Library → DB
  - AsyncTask
- Shared Prefs.
- Alarms
- Notifications



Questions?