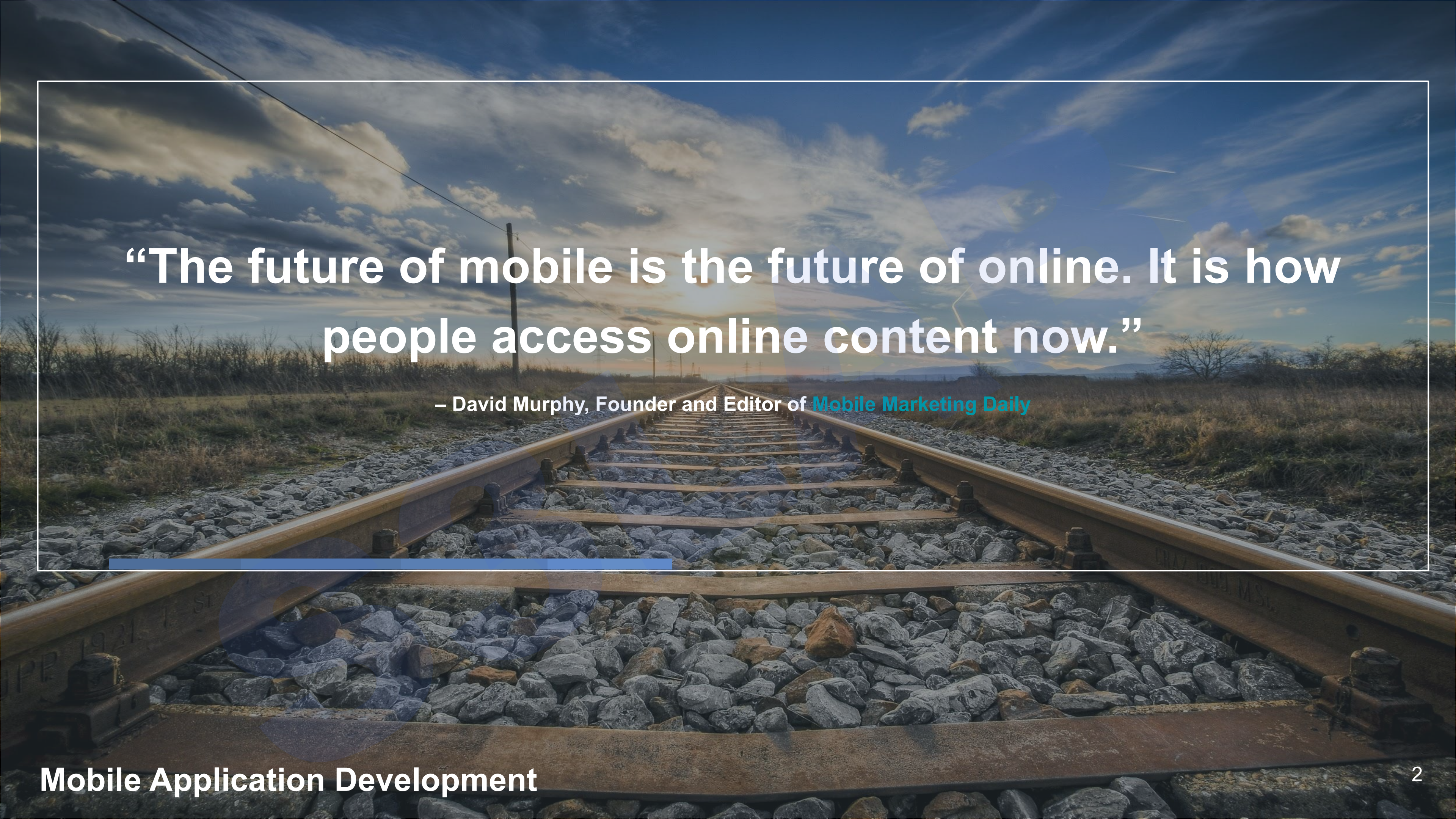
Mobile Application Development

(Background Tasks)

Instructor: Thanh Binh Nguyen

February 1st, 2020

S3Lab
Smart Software System Laboratory



“The future of mobile is the future of online. It is how people access online content now.”

– David Murphy, Founder and Editor of Mobile Marketing Daily

UI Thread

Overview

- Android app starts, it creates the **main** thread or **UI Thread**.
- The UI thread dispatches events to the appropriate user interface (UI) widgets.
- The UI thread is where your app interacts with components from the Android UI toolkit (components from the **android.widget** and **android.view** packages).
- Android thread Model has 2 rules:
  - **Do not block the UI thread**.
  - **Do UI work only on the UI thread**.
- The UI thread needs to give its attention to
  - Drawing the UI
  - Keeping the app responsive to user input.

UI Thread

Doesn’t block it

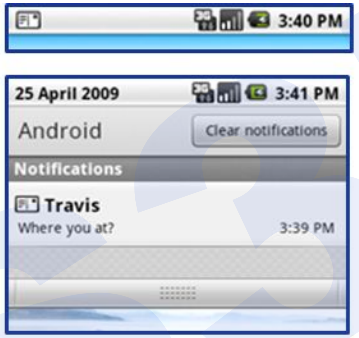
- Complete all work in **less than 16 ms** for each UI screen.
- Don't run **asynchronous tasks** and other **long-running tasks** (File operations, Network lookups, DB transactions, Complex calculations, etc...) on the UI thread

=> implement tasks on a background thread using **AsyncTask** (for short or interruptible tasks) or **AsyncTaskLoader** (for tasks that are high-priority, or tasks that need to report back to the user or UI), etc...

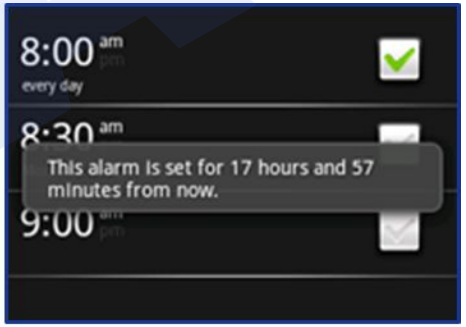
Service Notifications

2 types

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



Status Bar Notifications

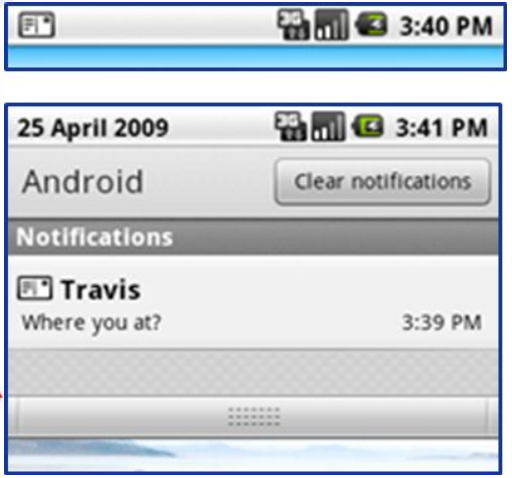


Toast Notifications

Service Notifications

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


3:40 PM

25 April 2009 3:41 PM

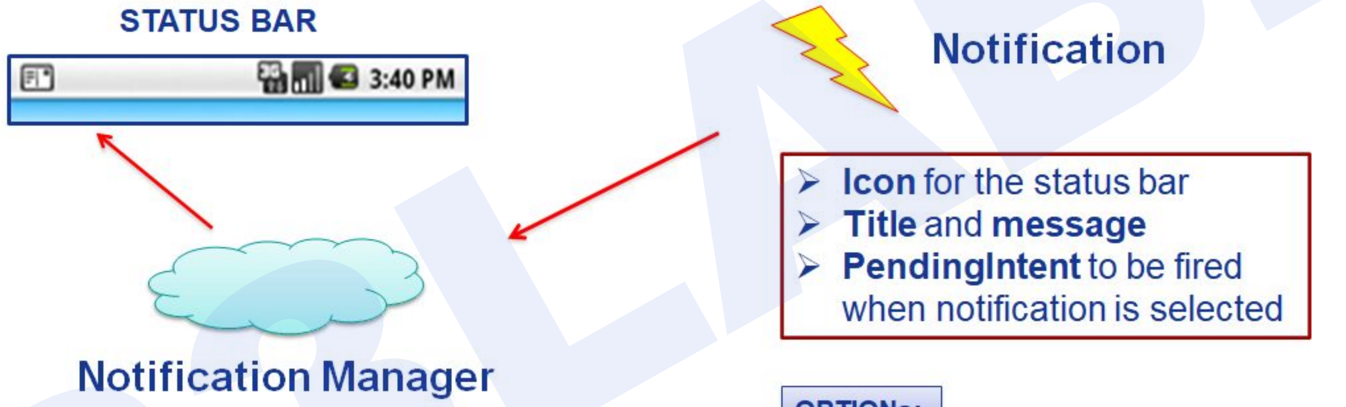
Android

Travis

Where you at? 3:39 PM

Service Notifications

Status Bar Notifications



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

Android system component
Responsible for notification management
And status bar updates

OPTIONs:

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

Service Notifications

Status Bar Notifications

- Follow these step to send a notification
  - Get a **reference** to the **Notification Manager**
    NotificationManager nm=(NotificationManager)
    getSystemService(Context.NOTIFICATION_SERVICE)
  - **Build** the Notification message
    public **Notification**(int icon, CharSequence tickerText, long when)
    public void **setLatestEvent**(Context context, CharSequence contentTitle, CharSequence contentText, PendingIntent intent)
  - **Send** the notification to the Notification Manager
    public void **notify**(int id, Notification notification)

Service Notifications

Status Bar Notifications

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);
```

Service Notifications

Status Bar Notifications

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);
```

Service Notifications

Status Bar Notifications

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;
```

Service Notifications

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;
```

Service Notifications

Status Bar Notifications

Add **vibrations** to the notification

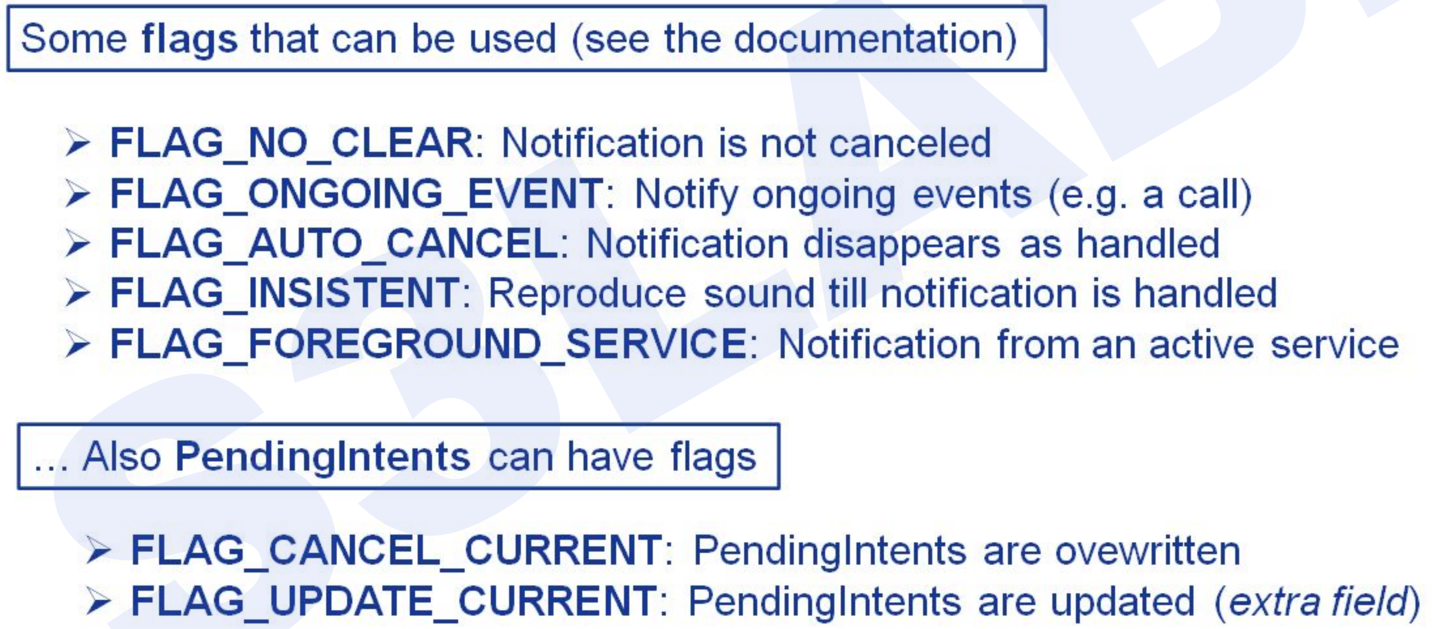
```
// Use a default vibration
notification.defaults |= Notification.DEFAULT_VIBRATE;
```

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;
```

Service Notifications

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

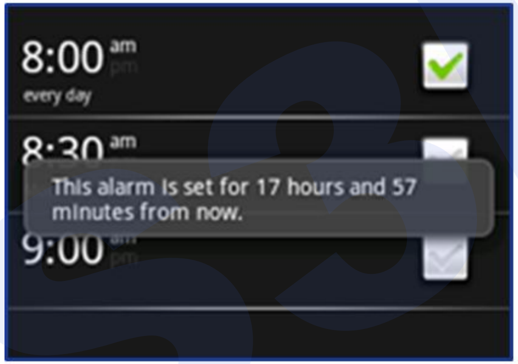
… Also **PendingIntents** can have flags

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

Service Notifications

Toast Notifications

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



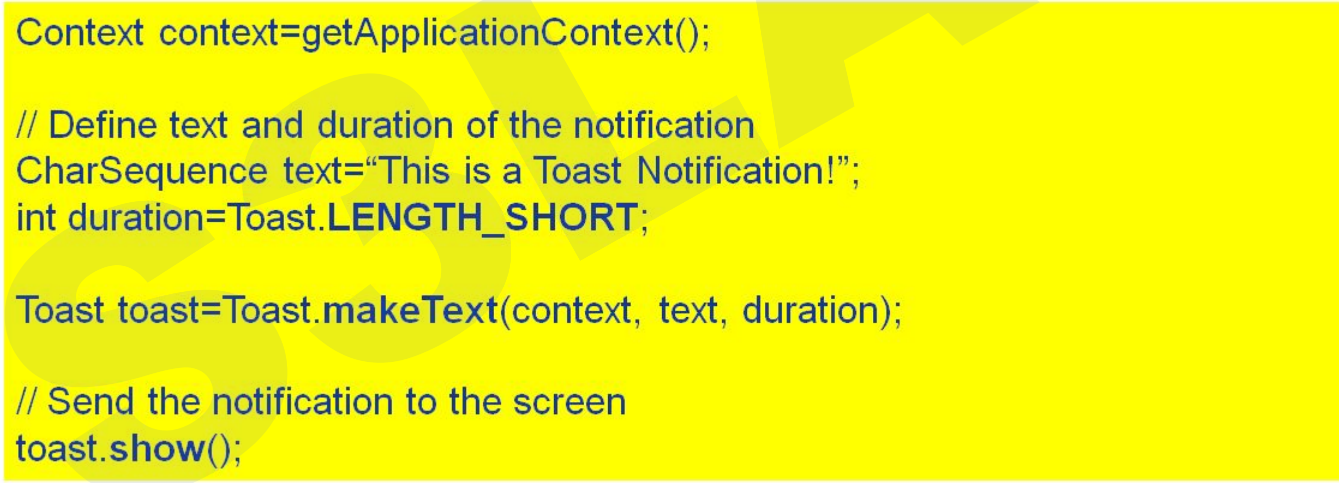
This alarm is set for 17 hours and 57 minutes from now.

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

Service Notifications

Toast Notifications

- A **Toast Notification** is a message that pops up on the surface of the window, and automatically fades out -> default or third party lib (Toasty)



```
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();
```

Thread Management

Overview

- 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.

Thread Management

Example

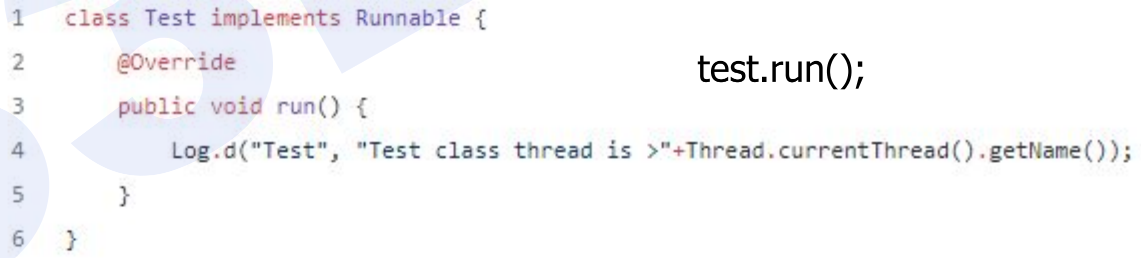
```
public class MyThread extends Thread {

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

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

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

Thread Management

Example

- (new Thread(new Runnable() {
    - public void run() {
        - String result = doLongOperation();
        - ~~updateUI(result);~~
    - }
- })).start();



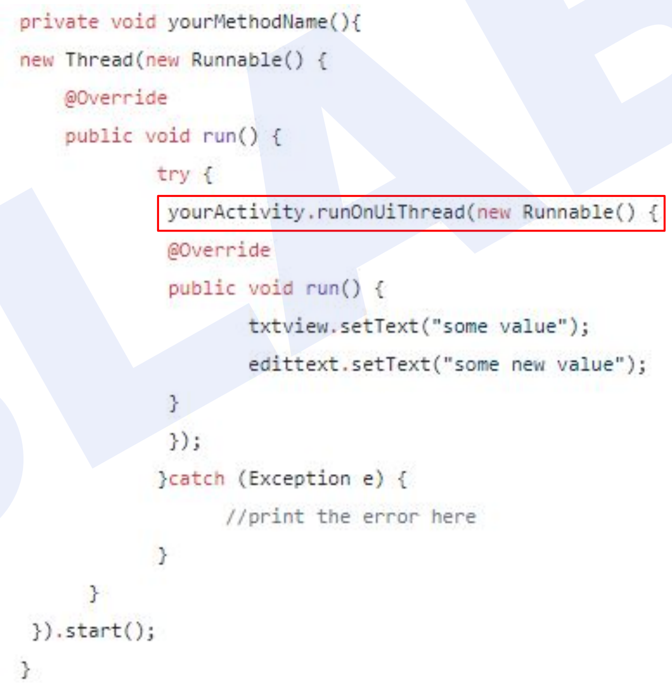
```
class Test implements Runnable {
    @Override
    public void run() {
        Log.d("Test", "Test class thread is >"+Thread.currentThread().getName());
    }
}
```

```
Test test = new Test();

test.run();
```

Thread Management

Example - Update UI

runOnUiThread() or **Handler**



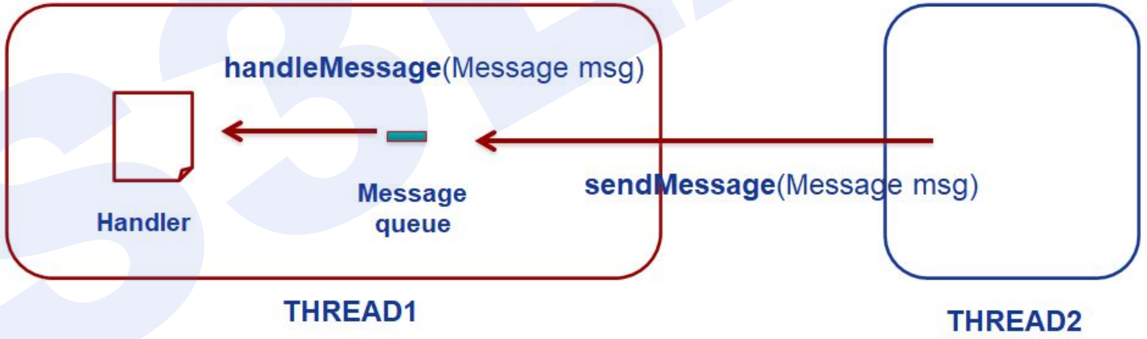
```
private void yourMethodName(){
new Thread(new Runnable() {
    @Override
    public void run() {
            try {
             yourActivity.runOnUiThread(new Runnable() {
             @Override
             public void run() {
                    txtview.setText("some value");
                    edittext.setText("some new value");
             }
             });
            }catch (Exception e) {
                  //print the error here
            }
      }
 }).start();
}
```

Thread Management

Communication between Thread

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



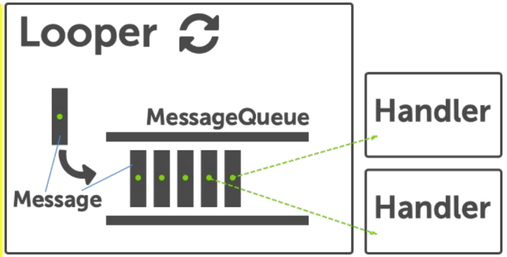
Thread Management

Communication between Thread

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

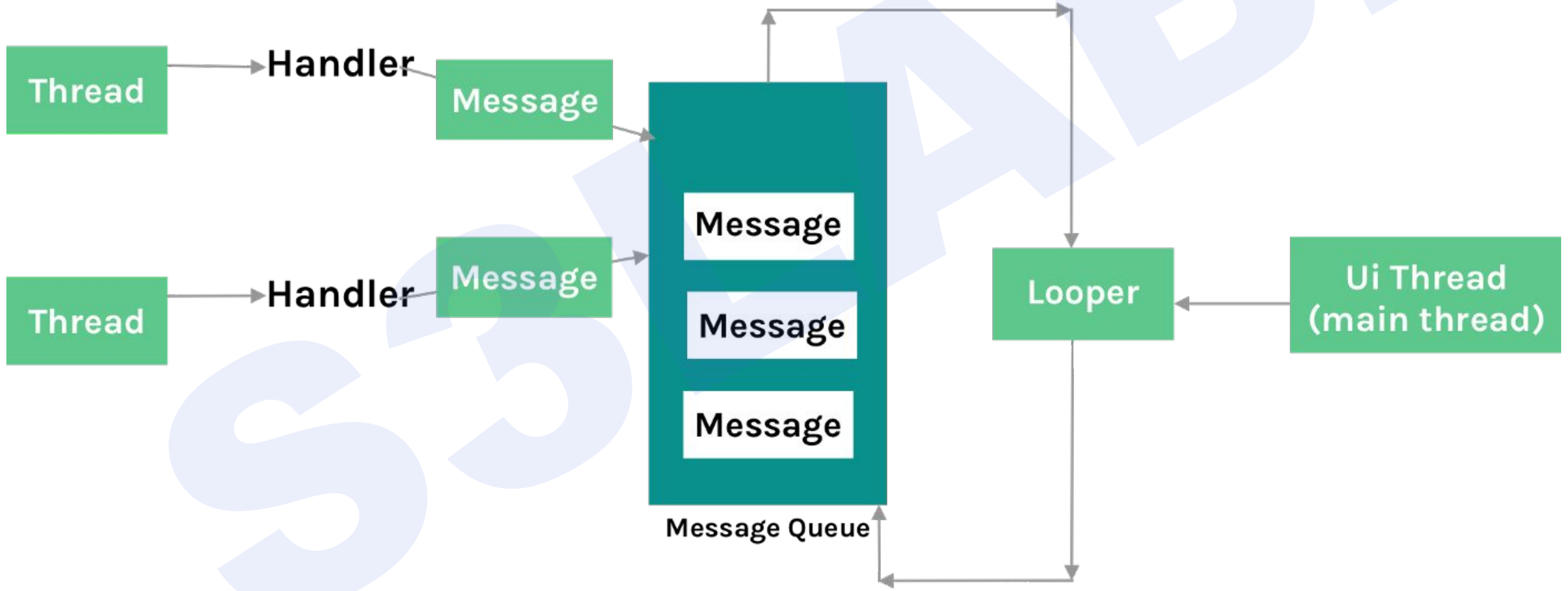
HOW? Use **Looper** objects …

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



Thread Management

Communication between Thread and UI



Thread Management

Communication between Thread and UI

```
Handler handler = new Handler(Looper.getMainLooper());
handler.postDelayed(new Runnable() {
    @Override
    public void run() {
        // update the ui from here
    }
},1000);
```

```
private void updateUIByHandler() {
final Handler myHandler = new Handler() {
@Override
public void handleMessage(Message msg) {
updateUI((String) msg.obj);
}

};
(new Thread(new Runnable() {

public void run() {
Message msg = myHandler.obtainMessage();//get message object

msg.obj = doLongOperation(1000);

myHandler.sendMessage(msg);//send message to handle it
}
})).start();
}
```

Services

Overview

- 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
  - 3 types of services: **Foreground**, **Background** (Started Service) and **Bound** Service.

  ⇒ **A Service provides a robust environment for background tasks …**

Services

Declare Service

- Declaring a service in the manifest

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

Services

Foregound 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:
  - 1. Create a **Notification** object
  - 2.Call **startForeground**(id, notification) from onStartCommand()
  - Call **stopForeground**() to stop the Service.

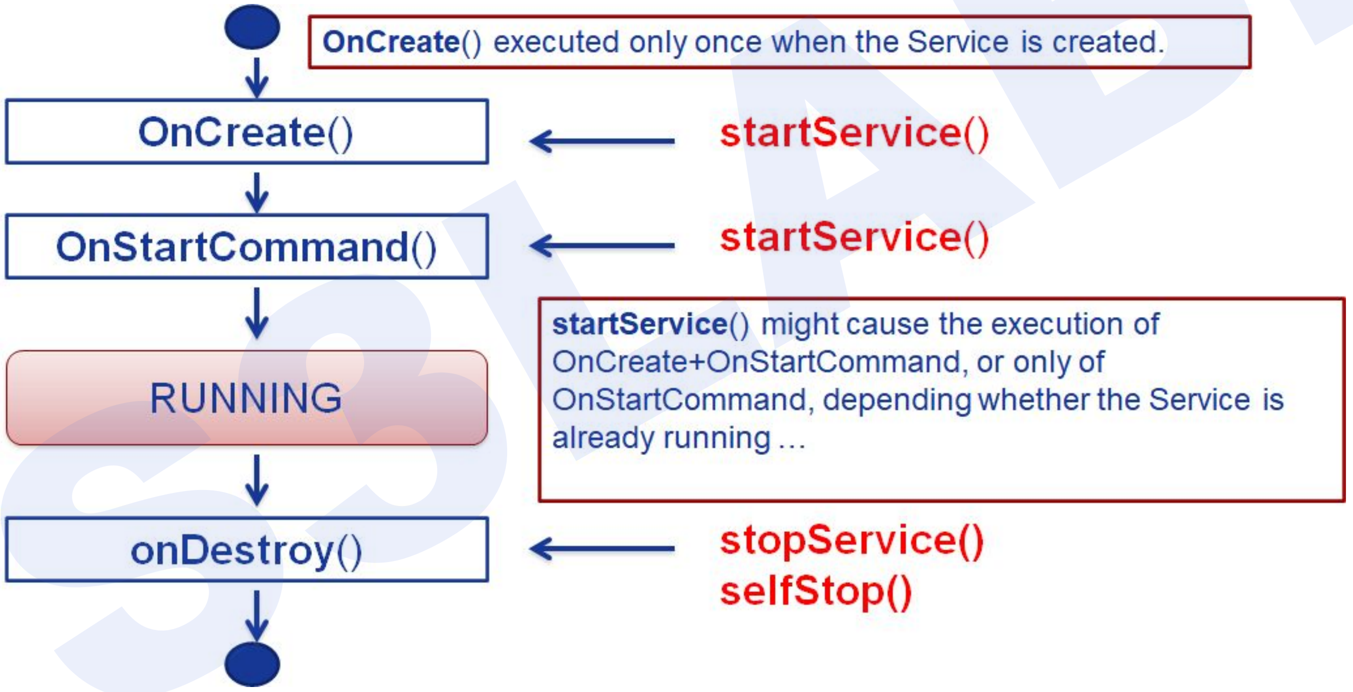
Services

Started or Background Service

- A Service is started when an application component starts it by calling **startService**(Intent).
- Once started, a Service runs in **background** indefinitely, 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)

Services

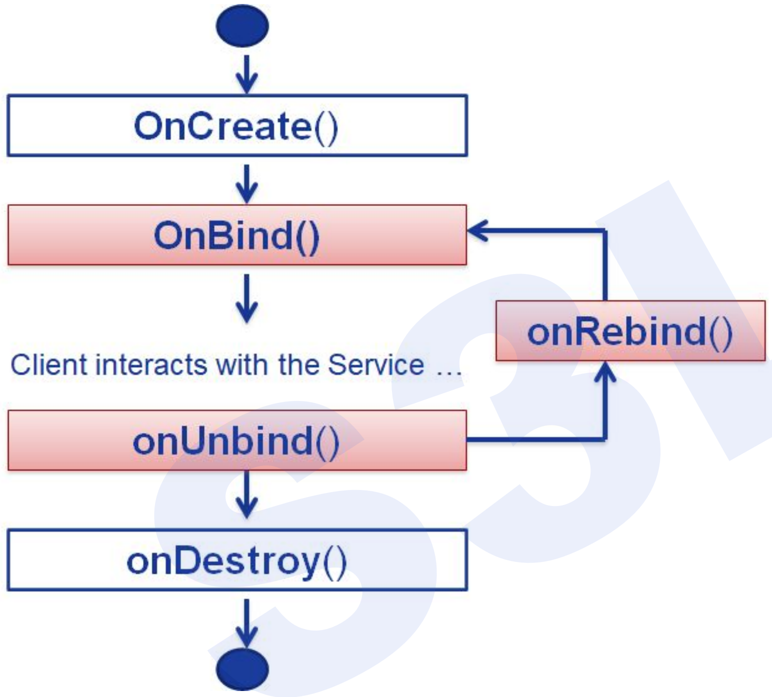
Started or Background Service



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

Services

Bound Services

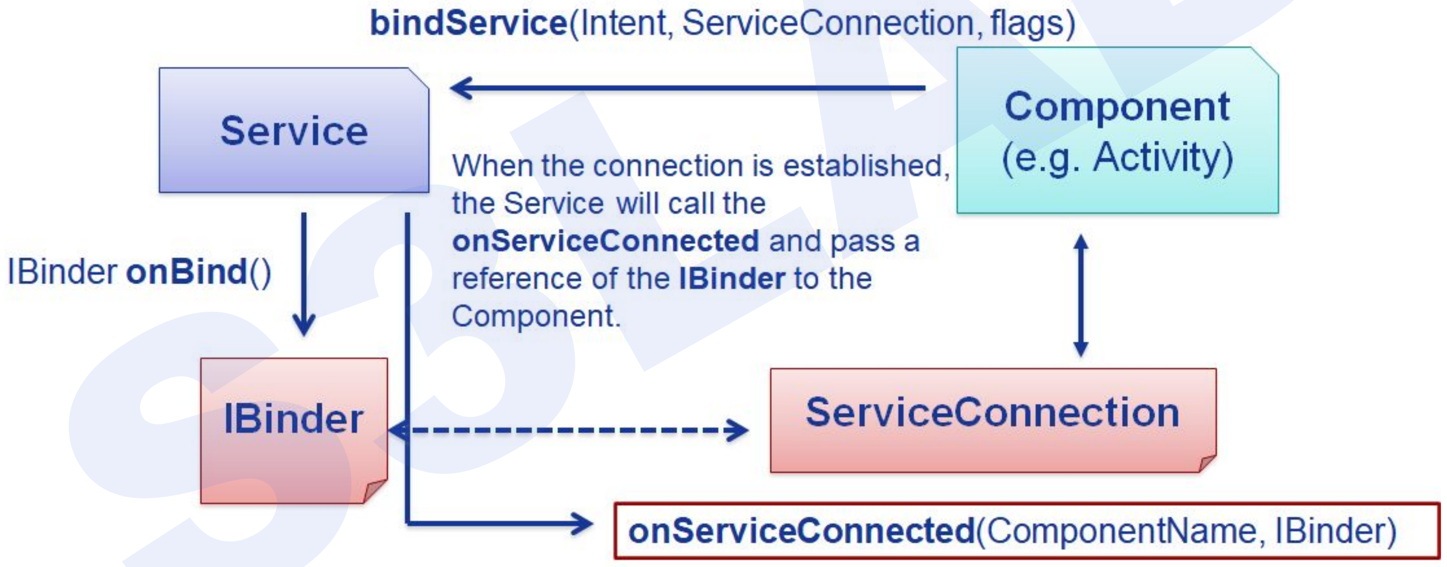


- A **Bound** Service allows components (e.g. Activity) to **bind** to the services, **send** requests, **receive** response.
- A **Bound** Service can serve components running on different processes (**IPC**).

Services

Bound Services

- Through the IBinder, the Component can send requests to the Service ...



bindService(Intent, ServiceConnection, flags)
Service
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 onBind()
IBinder
ServiceConnection
onServiceConnected(ComponentName, IBinder)

Services

Bound Services

- When creating a Service, an **IBinder** must be created to provide an Interface that clients can use to interact with the Service … HOW?
  - **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
  - **Using** the **A**ndroid **I**nterface **D**efinition **L**anguage (**AIDL**)
    - Allow to access a Service from different applications.

Services

Bound Services

```
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;
        }
    }
}
```

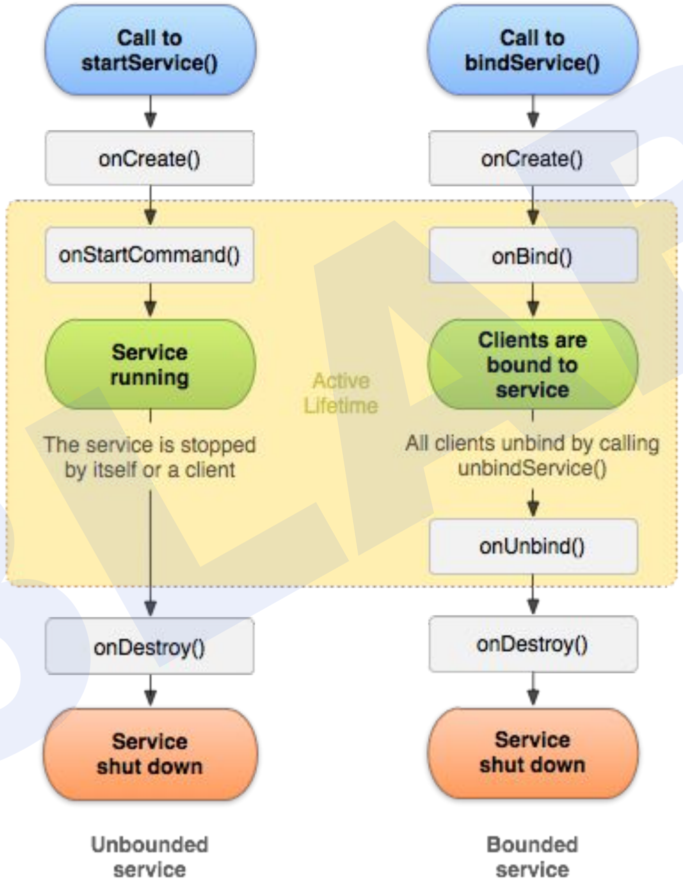
Services

Bound Services

```
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) {
        }

    };
```

Services

Bound Services



Broadcast Receiver

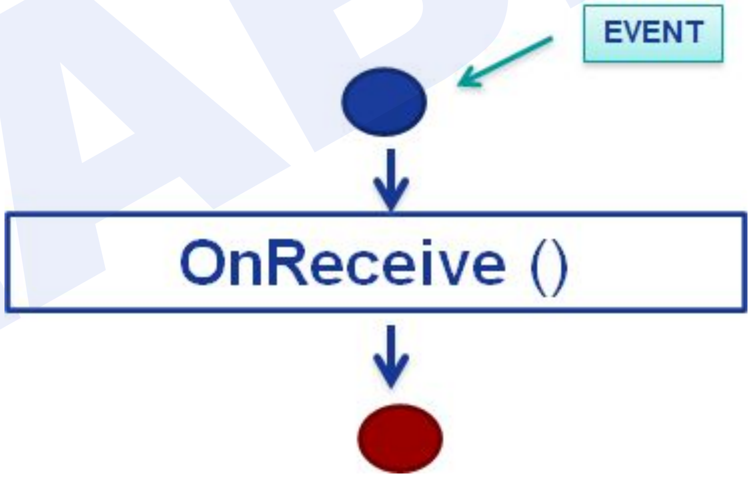
Overview

- 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.

Broadcast Receiver

Lifetime

- Single-state component …
- **onReceive()** is invoked when the registered event occurs
- After handling the event, the Broadcast Receiver is **destroyed**.

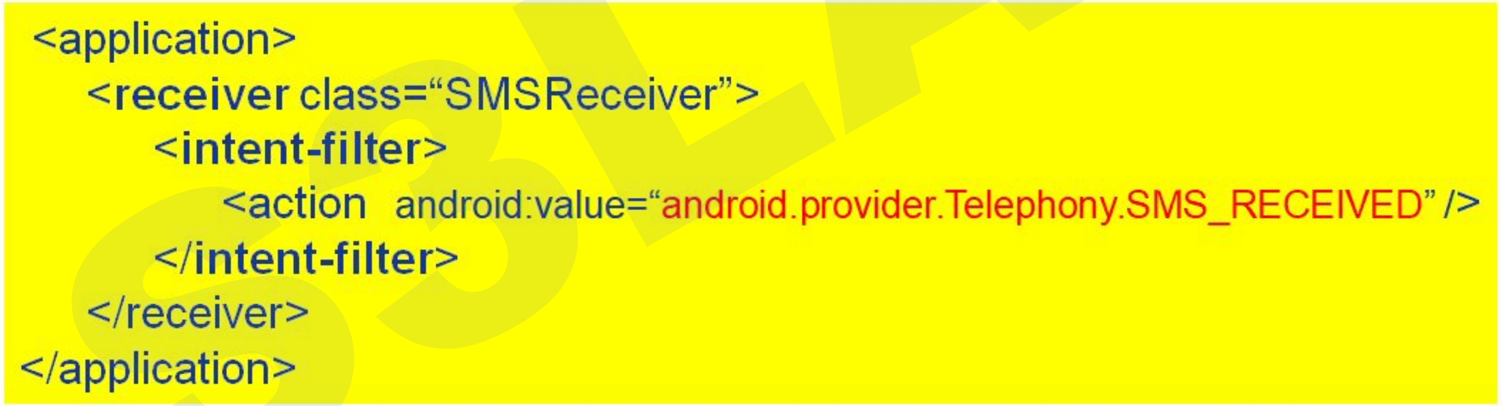


Broadcast Receiver

Lifetime

- **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>
```

Broadcast Receiver

Lifetime

- **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);
}
```

Broadcast Receiver

Lifetime

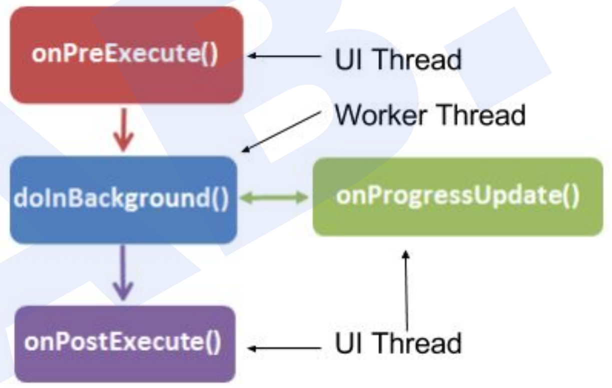
- How to send the **Intents** handled by **Broadcast Receivers**?
- void **sendBroadcast**(Intent intent)

  … 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!

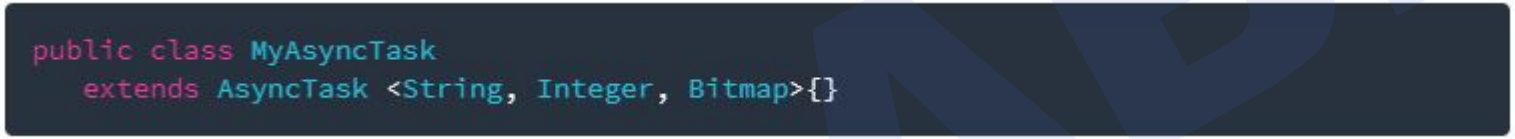
AsyncTask

- **onPreExecute**
    - is invoked before the execution.
- **onPostExecute**
    - is invoked after the execution.
- **doInBackground**
    - the main operation. Write your heavy operation here.
- **onProgressUpdate**
    - Indication to the user on progress. It is invoked every time **publishProgress**() is called.



AsyncTask

Create SubClass

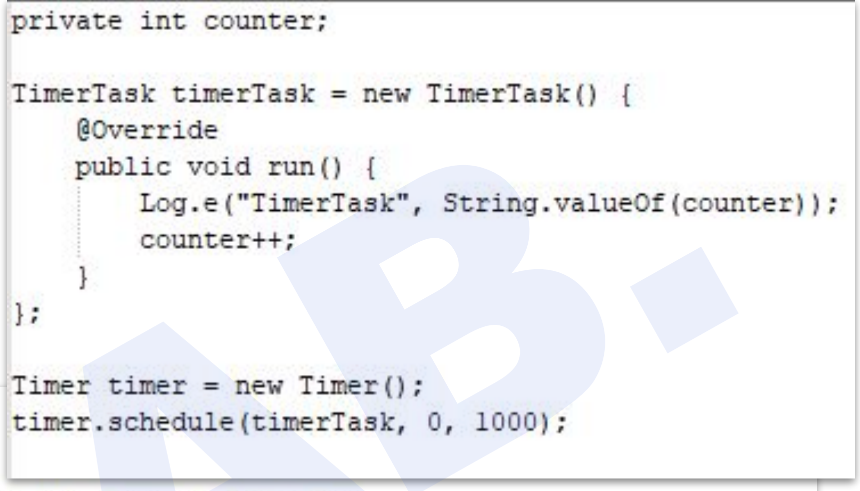


```
public class MyAsyncTask
    extends AsyncTask <String, Integer, Bitmap>{}
```

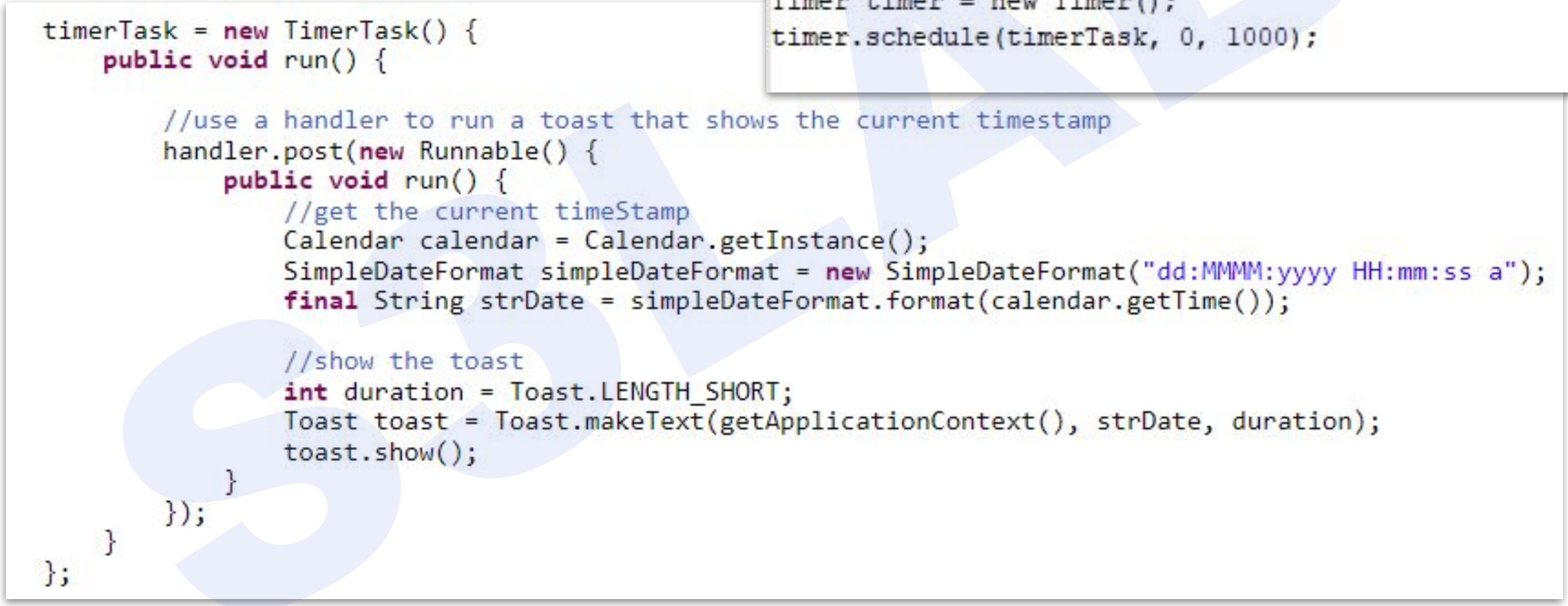
- A **String** as a parameter in **doInBackground**(), to use in a query, for example.
- An **Integer** for **onProgressUpdate**(), to represent the percentage of job complete
- A **Bitmap** for the result in **onPostExecute**(), indicating the query result.

Timer

- Like thread like timer in UI update.



```
private int counter;

TimerTask timerTask = new TimerTask() {
    @Override
    public void run() {
        Log.e("TimerTask", String.valueOf(counter));
        counter++;
    }
};

Timer timer = new Timer();
timer.schedule(timerTask, 0, 1000);
```



```
timerTask = new TimerTask() {
    public void run() {

        //use a handler to run a toast that shows the current timestamp
        handler.post(new Runnable() {
            public void run() {
                //get the current timeStamp
                Calendar calendar = Calendar.getInstance();
                SimpleDateFormat simpleDateFormat = new SimpleDateFormat("dd:MMMM:yyyy HH:mm:ss a");
                final String strDate = simpleDateFormat.format(calendar.getTime());

                //show the toast
                int duration = Toast.LENGTH_SHORT;
                Toast toast = Toast.makeText(getApplicationContext(), strDate, duration);
                toast.show();
            }
        });
    }
};
```

Homeworks

- Create an example which using background processing
  - Add a complicated operator to the existed calculator
  - Improve the last restful api homework
- References
  - https://www.tutlane.com/tutorial/android/android-progress-notification-with-examples
  - https://codelabs.developers.google.com/codelabs/android-training-notifications/index.html
  - https://codelabs.developers.google.com/codelabs/android-training-create-asynctask/index.html
  - https://codelabs.developers.google.com/codelabs/android-training-asynctask-asynctaskloader/index.html
  - https://codelabs.developers.google.com/codelabs/android-training-broadcast-receivers/index.html
  - https://codelabs.developers.google.com/codelabs/android-training-job-scheduler/index.html

Q & A

Thank you for listening

"Coming together is a beginning;
Keeping together is progress;
Working together is success."
- HENRY FORD