Android Developer Fundamentals

Background Tasks

Lesson 7

7.1 AsyncTask & AsyncTaskLoader

Contents

- Threads
- AsyncTask
- Loaders
- AsyncTaskLoader

Threads

The main thread

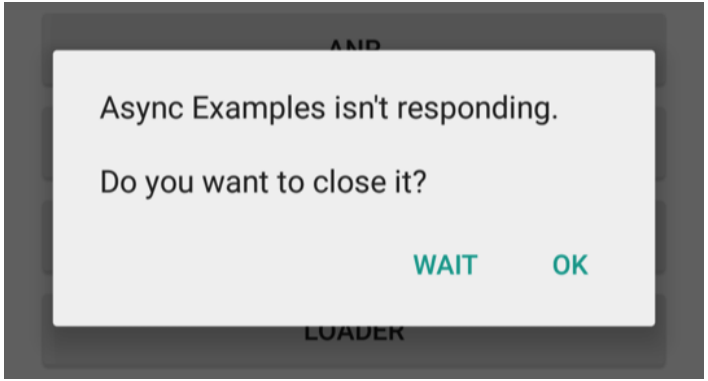
- Independent path of execution in a running program
- Code is executed line by line
- App runs on Java thread called "main" or "UI thread"
- Draws UI on the screen
- Responds to user actions by handling UI events

The Main thread must be fast

- Hardware updates screen every 16 milliseconds
- UI thread has 16 ms to do all its work
- If it takes too long, app stutters or hangs

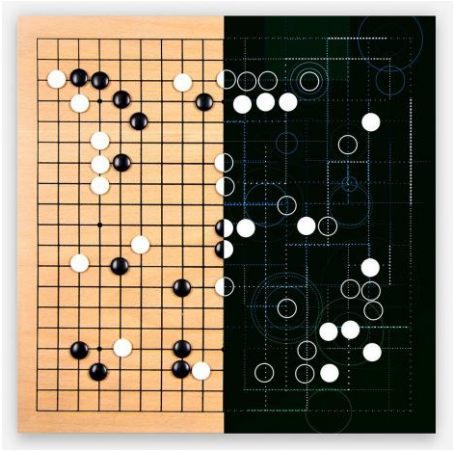
Users uninstall unresponsive apps

- If the UI waits too long for an operation to finish, it becomes unresponsive
- The framework shows an Application Not Responding (ANR) dialog



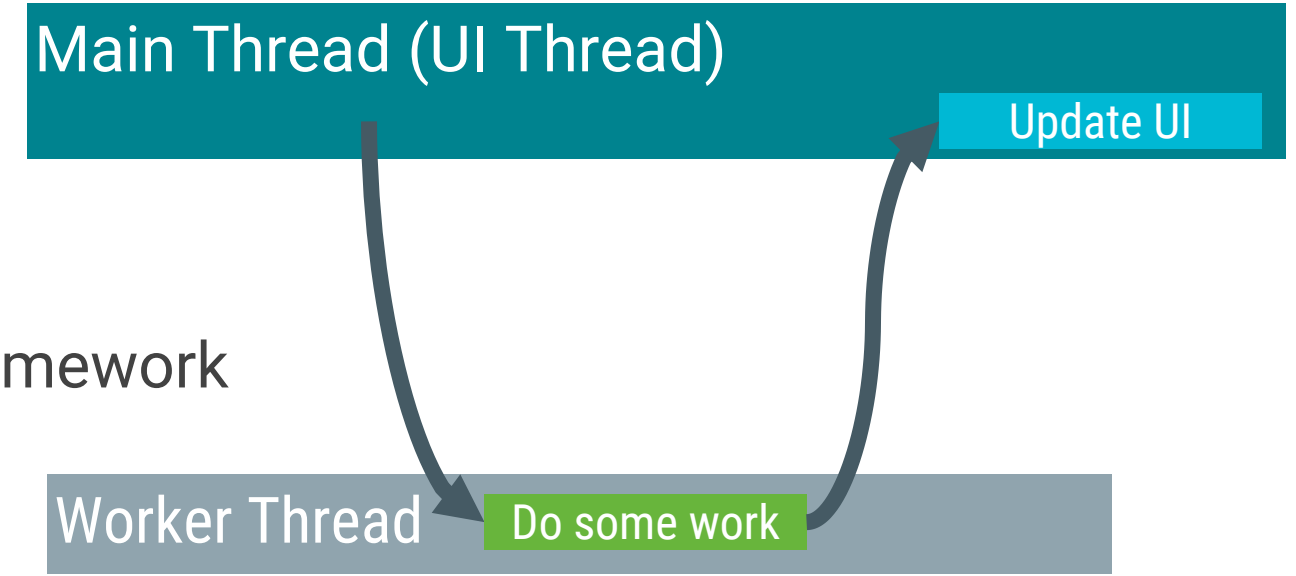
What is a long running task?

- Network operations
- Long calculations
- Downloading/uploading files
- Processing images
- Loading data



Background threads

Execute long running tasks on a **background thread**



- AsyncTask
- The Loader Framework
- Services

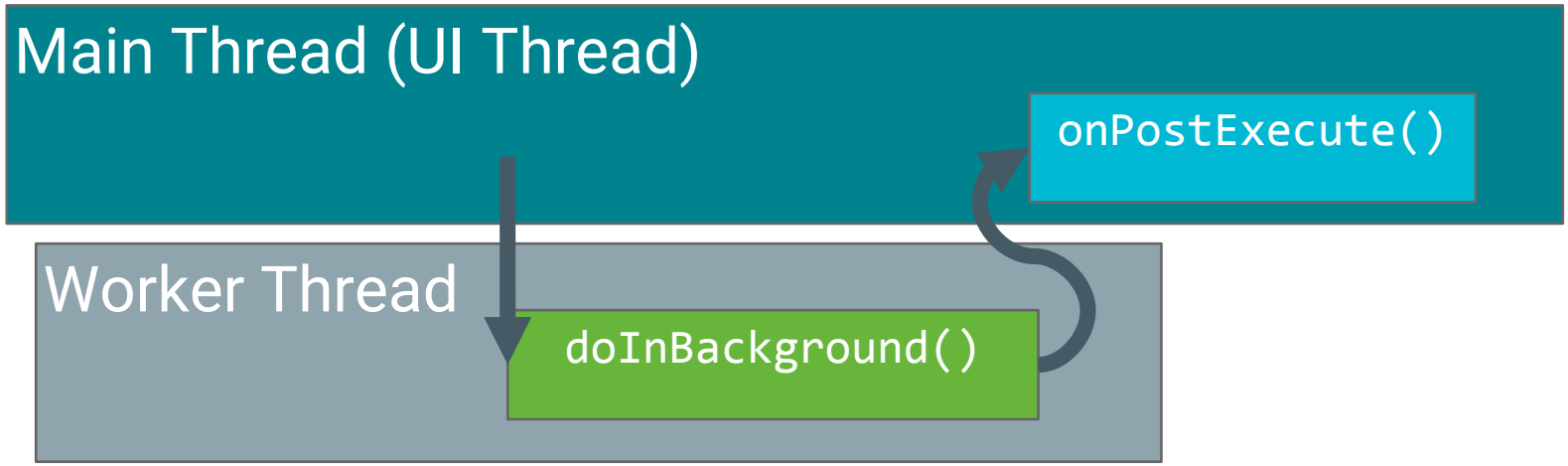
Two rules for Android threads

- Do not block the UI thread
  - Complete all work in less than 16 ms for each screen
  - Run slow non-UI work on a non-UI thread
- Do not access the Android UI toolkit from outside the UI thread
  - Do UI work only on the UI thread

AsyncTask

What is AsyncTask?

Use AsyncTask to implement basic background tasks



Override two methods

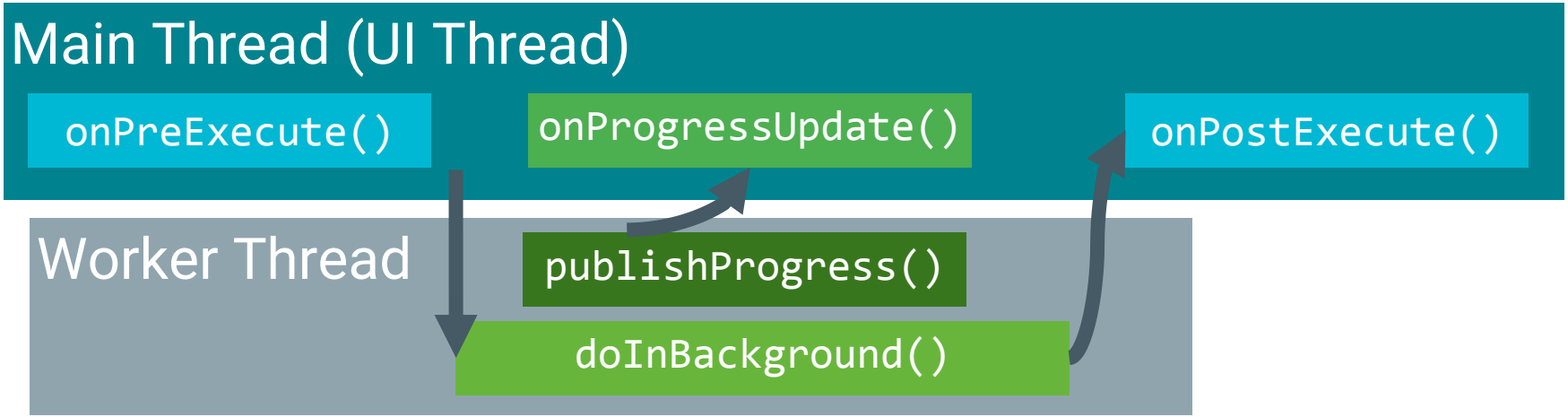
- `doInBackground()`—runs on a background thread
  - All the work to happen in the background
- `onPostExecute()`—runs on main thread when work done
  - Process results
  - Publish results to the UI

AsyncTask helper methods

- `onPreExecute()`
  - Runs on the main thread
  - Sets up the task

- `onProgressUpdate()`
  - Runs on the main thread
  - receives calls from `publishProgress()` from background thread

AsyncTask helper methods



Creating an AsyncTask

1. Subclass AsyncTask
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
        extends AsyncTask<URL, Integer, Bitmap> {...}
```

MyAsyncTask class definition

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

- String—could be query, URI for filename
- Integer—percentage completed, steps done
- Bitmap—an image to be displayed
- Use Void if no data passed

onPreExecute()

```
protected void onPreExecute() {
    // display a progress bar
    // show a toast
}
```

doInBackground()

```
protected Bitmap doInBackground(String... query) {
    // Get the bitmap
    return bitmap;
}
```

onProgressUpdate()

```
protected void onProgressUpdate(Integer... progress) {
        setProgressPercent(progress[0]);
    }
```

onPostExecute()

```
protected void onPostExecute(Bitmap result) {
    // Do something with the bitmap
}
```

Start background work

```
public void loadImage (View view) {
    String query = mEditText.getText().toString();
    new MyAsyncTask(query).execute();
   }
```

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

When to use AsyncTask

- Short or interruptible tasks
- Tasks that do not need to report back to UI or user
- Lower priority tasks that can be left unfinished
- Use AsyncTaskLoader otherwise

Loaders

What is a Loader?

- 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
- Many types of loaders available
  - AsyncTaskLoader, CursorLoader

Why use loaders?

- Execute tasks OFF the UI thread
- LoaderManager handles configuration changes for you
- Efficiently implemented by the framework
- Users don't have to wait for data to load

What is a LoaderManager?

- Manages loader functions via callbacks
- Can manage multiple loaders
  - loader for database data, for AsyncTask data, for internet data…

Get a loader with initLoader()

- Creates and starts a loader, or reuses an existing one, including its data
- Use restartLoader() to clear data in existing loader

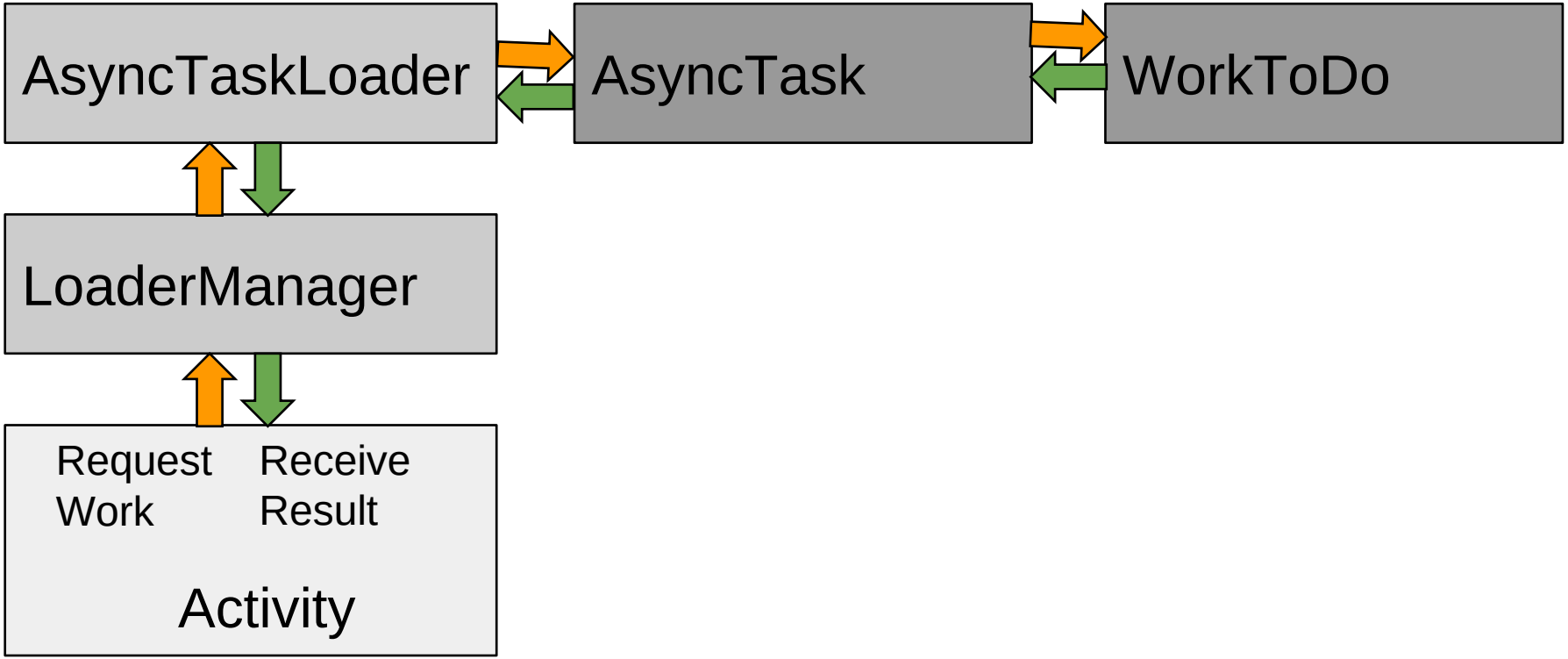
```
getLoaderManager().initLoader(Id, args, callback);

getLoaderManager().initLoader(0, null, this);

getSupportLoaderManager().initLoader(0, null, this);
```

Implementing AsyncTaskLoader

AsyncTaskLoader Overview



AsyncTask ⟶ AsyncTaskLoader

`doInBackground()` ⟶ `loadInBackground()`

`onPostExecute()` ⟶ `onLoadFinished()`

Steps for AsyncTaskLoader subclass

1. Subclass AsyncTaskLoader
2. Implement constructor
3. `loadInBackground()`
4. `onStartLoading()`

Subclass AsyncTaskLoader

```
public static class StringListLoader
        extends AsyncTaskLoader<List<String>> {

    public StringListLoader(Context context, String queryString) {
        super(context);
        mQueryString = queryString;
    }
}
```

loadInBackground()

```
public List<String> loadInBackground() {
    List<String> data = new ArrayList<String>;
    //TODO: Load the data from the network or from a database
    return data;
}
```

onStartLoading()

When `restartLoader()` or `initLoader()` is called, the LoaderManager invokes the `onStartLoading()` callback

- Check for cached data
- Start observing the data source (if needed)
- Call `forceLoad()` to load the data if there are changes or no cached data

```
protected void onStartLoading() {  forceLoad();  }
```

Implement loader callbacks in Activity

- `onCreateLoader()` — Create and return a new Loader for the given ID
- `onLoadFinished()` — Called when a previously created loader has finished its load
- `onLoaderReset()` — Called when a previously created loader is being reset making its data unavailable

onCreateLoader()

```
@Override
public Loader<List<String>> onCreateLoader(int id, Bundle args) {
    return new StringListLoader(this,args.getString("queryString"));
}
```

onLoadFinished()

Results of `loadInBackground()` are passed to `onLoadFinished()` where you can display them

```
public void onLoadFinished(Loader<List<String>> loader,
List<String> data) {
    mAdapter.setData(data);
}
```

onLoaderReset()

- Only called when loader is destroyed
- Leave blank most of the time

```
@Override
public void onLoaderReset(final LoaderList<String>> loader) { }
```

Get a loader with initLoader()

- In Activity
- Use support library to be compatible with more devices

```
getSupportLoaderManager().initLoader(0, null, this);
```

Learn more

- AsyncTask Reference
- AsyncTaskLoader Reference
- LoaderManager Reference
- Processes and Threads Guide
- Loaders Guide
- UI Thread Performance: Exceed the Android Speed Limit

What's Next?

- Concept Chapter: 7.1 C AsyncTask and AsyncTaskLoader
- Practical: 7.1 P Create an AsyncTask

END

Android Developer Fundamentals

Connect to the Internet

Lesson 7

7.2 Connect to the Internet

Steps to connect to the Internet

1. Add permissions to Android Manifest
2. Check Network Connection
3. Create Worker Thread
4. Implement background task
    a. Create URI
    b. Make HTTP Connection
    c. Connect and GET Data
5. Process results
    a. Parse Results

Permissions

Permissions in AndroidManifest

Internet

```
<uses-permission android:name="android.permission.INTERNET"/>
```

Check Network State

```
<uses-permission
    android:name="android.permission.ACCESS_NETWORK_STATE"/>
```

Manage Network Connection

Getting Network information

- ConnectivityManager
  - Answers queries about the state of network connectivity
  - Notifies applications when network connectivity changes
- NetworkInfo
  - Describes status of a network interface of a given type
  - Mobile or Wi-Fi

Check if network is available

```
ConnectivityManager connMgr = (ConnectivityManager)
            getSystemService(Context.CONNECTIVITY_SERVICE);

NetworkInfo networkInfo = connMgr.getActiveNetworkInfo();

if (networkInfo != null && networkInfo.isConnected()) {
    // Create background thread to connect and get data
    new DownloadWebpageTask().execute(stringUrl);
} else {
    textView.setText("No network connection available.");
}
```

Check for WiFi & Mobile

```
NetworkInfo networkInfo =
    connMgr.getNetworkInfo(ConnectivityManager.TYPE_WIFI);
boolean isWifiConn = networkInfo.isConnected();

networkInfo =
    connMgr.getNetworkInfo(ConnectivityManager.TYPE_MOBILE);
boolean isMobileConn = networkInfo.isConnected();
```

Worker Thread

Use Worker Thread

- AsyncTask—very short task, or no result returned to UI
- AsyncTaskLoader—for longer tasks, returns result to UI
- Background Service—later chapter

Background work

In the background task (for example in `doInBackground()`)

1. Create URI
2. Make HTTP Connection
3. Download Data

Create URI

URI = Uniform Resource Identifier

String that names or locates a particular resource

- file://
- http:// and https://
- content://

Sample URL for Google Books API

https://www.googleapis.com/books/v1/volumes?
q=pride+prejudice&maxResults=5&printType=books

Constants for Parameters

```
final String BASE_URL =
    "https://www.googleapis.com/books/v1/volumes?";
final String QUERY_PARAM = "q";
final String MAX_RESULTS = "maxResults";
final String PRINT_TYPE = "printType";
```

Build a URI for the request

```
Uri builtURI = Uri.parse(BASE_URL).buildUpon()
        .appendQueryParameter(QUERY_PARAM, "pride+prejudice")
        .appendQueryParameter(MAX_RESULTS, "10")
        .appendQueryParameter(PRINT_TYPE, "books")
        .build();
URL requestURL = new URL(builtURI.toString());
```

HTTP Client Connection

Make a connection from scratch

- Use `HttpURLConnection`
- Must be done on a separate thread
- Requires InputStreams and try/catch blocks

Create a HttpURLConnection

```
HttpURLConnection conn =
   (HttpURLConnection) requestURL.openConnection();
```

Configure connection

```
conn.setReadTimeout(10000 /* milliseconds */);
conn.setConnectTimeout(15000 /* milliseconds */);
conn.setRequestMethod("GET");
conn.setDoInput(true);
```

Connect and get response

```
conn.connect();
int response = conn.getResponseCode();

InputStream is = conn.getInputStream();
String contentAsString = convertIsToString(is, len);
return contentAsString;
```

Close connection and stream

```
} finally {
        conn.disconnect();
         if (is != null) {
             is.close();
         }
}
```

Convert Response to String

Convert input stream into a string

```
public String convertIsToString(InputStream stream, int len)
    throws IOException, UnsupportedEncodingException {

    Reader reader = null;
    reader = new InputStreamReader(stream, "UTF-8");
    char[] buffer = new char[len];
    reader.read(buffer);
    return new String(buffer);
}
```

BufferedReader is more efficient

```
StringBuilder builder = new StringBuilder();
BufferedReader reader =
    new BufferedReader(new InputStreamReader(inputStream));
String line;
while ((line = reader.readLine()) != null) {
   builder.append(line + "\n");
}
if (builder.length() == 0) {
   return null;
}
resultString = builder.toString();
```

HTTP Client Connection Libraries

Make a connection using libraries

- Use a third party library like OkHttp or Volley
- Can be called on the main thread
- Much less code

Volley

```
RequestQueue queue = Volley.newRequestQueue(this);
String url ="http://www.google.com";

StringRequest stringRequest = new StringRequest(Request.Method.GET, url,
            new Response.Listener<String>() {
    @Override
    public void onResponse(String response) {
        // Do something with response
    }
}, new Response.ErrorListener() {
    @Override
    public void onErrorResponse(VolleyError error) {}
});
queue.add(stringRequest);
```

OkHttp

```
OkHttpClient client = new OkHttpClient();
Request request = new Request.Builder()
    .url("http://publicobject.com/helloworld.txt").build();
client.newCall(request).enqueue(new Callback() {
    @Override
    public void onResponse(Call call, final Response response)
        throws IOException {
        try {
            String responseData = response.body().string();
            JSONObject json = new JSONObject(responseData);
            final String owner = json.getString("name");
        } catch (JSONException e) {}
    }
});
```

Parse Results

Parsing the results

- Implement method to receive and handle results (`onPostExecute()`)
- Response is often JSON or XML

Parse results using helper classes

- JSONObject, JSONArray
- XMLPullParser—parses XML

JSON basics

```
{
 "population":1,252,000,000,
 "country":"India",
 "cities":["New Delhi","Mumbai","Kolkata","Chennai"]
}
```

JSONObject basics

```
JSONObject jsonObject = new JSONObject(response);

String nameOfCountry = (String) jsonObject.get("country");
long population = (Long) jsonObject.get("population");
JSONArray listOfCities = (JSONArray) jsonObject.get("cities");

Iterator<String> iterator = listOfCities.iterator();
while (iterator.hasNext()) {
    // do something
}
```

Another JSON example

```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}
```

Another JSON example

Get "onclick" value of the 3rd item in the "menuitem" array

```
JSONObject data = new JSONObject(responseString);
JSONArray menuItemArray =
        data.getJSONArray("menuitem");
JSONObject thirdItem =
        menuItemArray.getJSONObject(2);
String onClick = thirdItem.getString("onclick");
```

Learn more

- Connect to the Network Guide
- Managing Network Usage Guide
- HttpURLConnection reference
- ConnectivityManager reference
- InputStream reference

What's Next?

- Concept Chapter: 7.2 C Connect to the Internet

Practical:

- 7.2 P Connect to the Internet with AsyncTask and AsyncTaskLoader

END

Android Developer Fundamentals

Background Tasks

Lesson 7

7.3 Broadcast Receivers

Contents

- Broadcast intents
- Broadcast receivers
- Implementing broadcast receivers
- Custom broadcasts
- Security
- Local broadcasts

Broadcast Intents

Broadcast vs. Activity

Use implicit intents to send broadcasts or start activities

Sending broadcasts

- Use sendBroadcast()
- Can be received by any application registered for the intent
- Used to notify all apps of an event

Starting activities

- Use startActivity()
- Find a single activity to accomplish a task
- Accomplish a specific action

Broadcast Receivers

What is a broadcast receiver?

- Listens for incoming intents sent by sendBroadcast()
  - In the background
- Intents can be sent
  - By the system, when an event occurs that might change the behavior of an app
  - By another application, including your own

Broadcast receiver always responds

- Responds even when your app is closed
- Independent from any activity
- When a broadcast intent is received and delivered to onReceive(), it has 5 seconds to execute, and then the receiver is destroyed

System broadcasts

- Automatically delivered when certain events occur
- After the system completes a boot
  - `android.intent.action.BOOT_COMPLETED`
- When the wifi state changes
  - `android.net.wifi.WIFI_STATE_CHANGED`

Custom broadcasts

- Deliver any custom intent as a broadcast
  - `sendBroadcast()` method—asynchronous
  - `sendOrderedBroadcast()`—synchronously
  - `android.example.com.CUSTOM_ACTION`

sendBroadcast()

- All receivers of the broadcast are run in an undefined order
- Can be at the same time
- Efficient
- Use to send custom broadcasts

sendOrderedBroadcast()

- Delivered to one receiver at a time
- Receiver can propagate result to the next receiver or abort the broadcast
- Control order with android:priority of matching intent filter
- Receivers with same priority run in arbitrary order

Implementing Broadcast Receivers

Steps for creating a broadcast receiver

1. Subclass `BroadcastReceiver`
2. Implement `onReceive()` method
3. Register to receive broadcast
    - Statically, in AndroidManifest
    - Dynamically, with `registerReceiver()`

File > New > Other > BroadcastReceiver

```
public class CustomReceiver extends BroadcastReceiver {
    public CustomReceiver() {
    }

    @Override
    public void onReceive(Context context, Intent intent) {
        // TODO: This method is called when the BroadcastReceiver
        //  is receiving an Intent broadcast.
        throw new UnsupportedOperationException("Not yet implemented");
    }
}
```

Register in Android Manifest

- <receiver> element inside <application>
- <intent-filter> registers receiver for specific intents

```
<receiver
   android:name=".CustomReceiver"
   android:enabled="true"
   android:exported="true">
      <intent-filter>
 <action android:name="android.intent.action.BOOT_COMPLETED" />
      </intent-filter>
</receiver>
```

Register dynamically

- In onCreate() or onResume()
- Use registerReceiver() and pass in the intent filter
- Must unregister in onDestroy() or onPause()

```
registerReceiver(mReceiver, mIntentFilter)
unregisterReceiver(mReceiver)
```

Available intents

- ACTION_TIME_TICK
- ACTION_TIME_CHANGED
- ACTION_TIMEZONE_CHANGED
- ACTION_BOOT_COMPLETED
- ACTION_PACKAGE_ADDED
- ACTION_PACKAGE_CHANGED
- ACTION_PACKAGE_REMOVED
- ACTION_PACKAGE_RESTARTED
- ACTION_PACKAGE_DATA_CLEARED
- ACTION_PACKAGES_SUSPENDED
- ACTION_PACKAGES_UNSUSPENDED
- ACTION_UID_REMOVED
- ACTION_BATTERY_CHANGED
- ACTION_POWER_CONNECTED
- ACTION_POWER_DISCONNECTED
- ACTION_SHUTDOWN

Implement onReceive()

```
@Override
public void onReceive(Context context, Intent intent) {
   String intentAction = intent.getAction();
   switch (intentAction){
       case Intent.ACTION_POWER_CONNECTED:
           break;
       case Intent.ACTION_POWER_DISCONNECTED:
           break;
   }
}
```

Custom Broadcasts

Custom broadcasts

- Sender and receiver must agree on unique name for intent (action name)
- Define in activity and broadcast receiver

```
private static final String ACTION_CUSTOM_BROADCAST =
        "com.example.android.powerreceiver.ACTION_CUSTOM_BROADCAST";
```

Send custom broadcasts

```
Intent customBroadcastIntent =
    new Intent(ACTION_CUSTOM_BROADCAST);

LocalBroadcastManager.getInstance(this)
    .sendBroadcast(customBroadcastIntent);
```

Destroy!

```
@Override
protected void onDestroy() {
    LocalBroadcastManager.getInstance(this)
            .unregisterReceiver(mReceiver);
    super.onDestroy();
}
```

Security

Security

- Receivers cross app boundaries
- Make sure namespace for intent is unique and you own it
- Other apps can send broadcasts to your receiver—use permissions to control this
- Other apps can respond to broadcast your app sends
- Access permissions can be enforced by sender or receiver

Controlling permission sender

- `void sendBroadcast (Intent intent,`
  `String receiverPermission)`

- Receivers must request permission with `<uses-permission>` in AndroidManifest.xml

Controlling permission receiver

- `registerReceiver(BroadcastReceiver, IntentFilter, String, android.os.Handler)`
- or in `<receiver>` tag

- Senders must request permission with `<uses-permission>` in AndroidManifest.xml

Local Broadcast Manager

Local Broadcast Manager

- For broadcasts only in your app
- No security issues since no cross-app communication

```
LocalBroadcastManager.sendBroadcast()

LocalBroadcastManager.registerReceiver()
```

Register local broadcast manager

```
LocalBroadcastManager.getInstance(this)
   .registerReceiver( mReceiver,
        new IntentFilter(ACTION_CUSTOM_BROADCAST));
```

Learn more

- BroadcastReceiver Reference
- Intents and Intent Filters Guide
- LocalBroadcastManager Reference
- Manipulating Broadcast Receivers On Demand

What's Next?

- Concept Chapter: 7.3 C Broadcast Receivers
- Practical: 7.3 P Broadcast Receivers

END

Android Developer Fundamentals

Background Tasks

Lesson 7

7.4 Services

Contents

- Services for long tasks
- IntentService

Services is an advanced topic

- Services are complex
- Many ways of configuring a service
- This lesson has introductory information only
- Explore and learn for yourself if you want to use services

Services for Long Tasks

What is a service?

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

What are services good for?

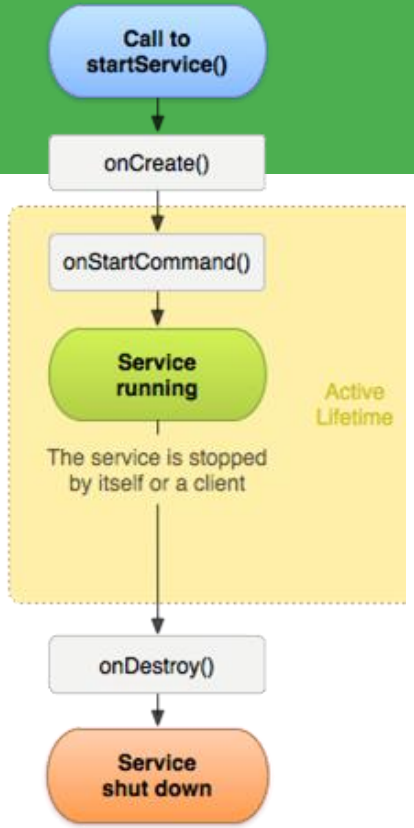
- Network transactions
- Play music
- Perform file I/O
- Interact with a content provider

Characteristics of services

- Started with an Intent
- Can stay running when user switches applications
- Lifecycle—which you must manage
- Other apps can use the service—manage permissions
- Runs in the main thread of its hosting process

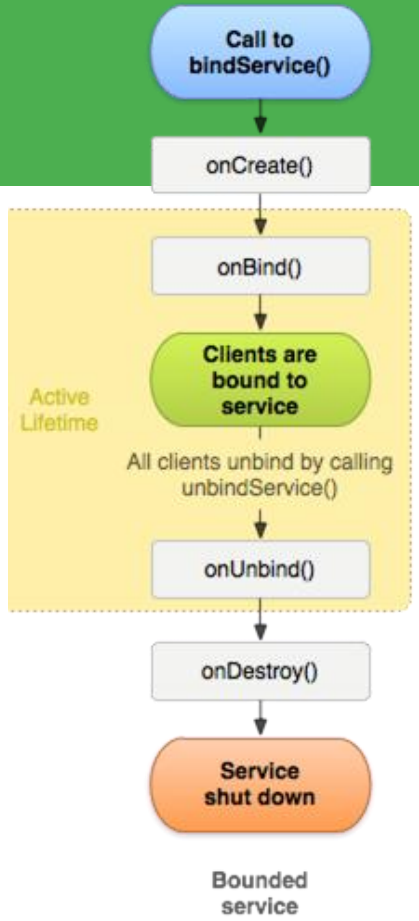
Forms of services: started

- Started with startService()
- Runs indefinitely until it stops itself
- Usually does not update the UI



Forms of services: 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



Services and threads

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

Offload CPU-intensive work to a separate thread within the service

Updating the app

If the service can't access the UI, how do you update the app to show the results?

Use a broadcast receiver!

Foreground services

Runs in the background but requires that the user is actively aware it exists—e.g. music player using music service

- Higher priority than background services since user will notice its absence—unlikely to be killed by the system
- Must provide a notification which the user cannot dismiss while the service is running

Creating a service

- `<service android:name=".ExampleService" />`
- Manage permissions
- Subclass IntentService or Service class
- Implement lifecycle methods
- Start service from activity
- Make sure service is stoppable

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

IntentService

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

IntentService Limitations

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

IntentService Implementation

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

    @Override
  protected void onHandleIntent(Intent intent) {
      try {
          // Do some work
      } catch (InterruptedException e) {
          Thread.currentThread().interrupt();
      }
  } // When this method returns, IntentService stops the service, as appropriate.
}
```

Learn more

- Services Guide
- Running a Background Service

What's Next?

- Concept Chapter: 7.4 C Services
- No practical

END

Android Developer Fundamentals

Background Tasks

Lesson 8

8.1 Notifications

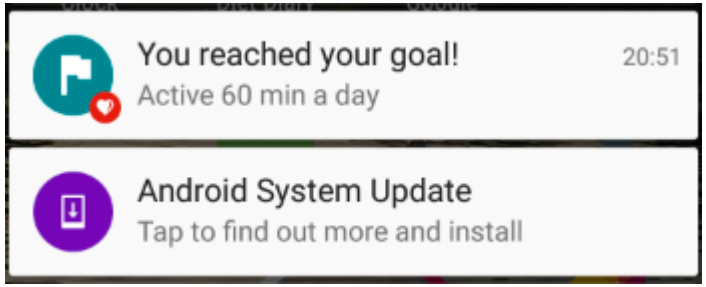
Contents

- Notifications
- Creating Notifications
- Actions and Pending Intents
- Priority and Defaults
- Common Layouts
- Managing Notifications

What Are Notifications?

What is a notification?

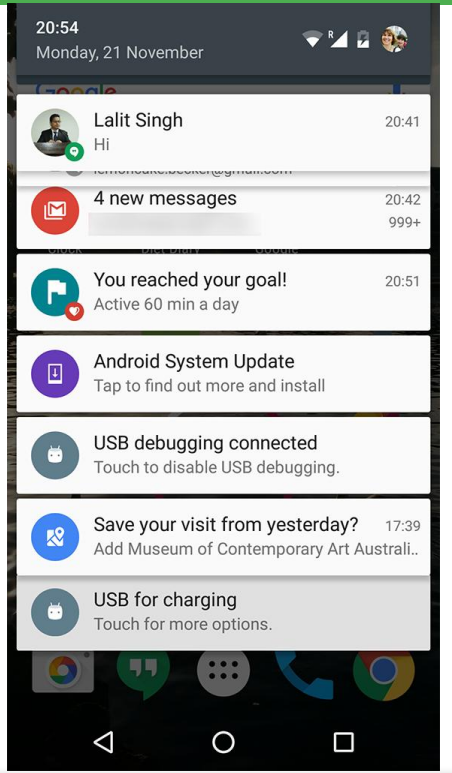
Message displayed to user outside regular app UI



- Small icon
- Title
- Detail text

How are notifications used?

- Android issues a notification that appears as icon
- To see details, user opens the notification drawer
- User can view notifications any time in the notification drawer



Creating Notifications

Two classes

`NotificationCompat.Builder`

- Specifies UI and actions
- NotificationCompat.Builder.build() creates the Notification

`NotificationManager` / `NotificationManagerCompat`

- NotificationManager.notify() issues the notification

Define variables

```
private NotificationCompat.Builder mNotifyBuilder;

private NotificationManager mNotifyManager;

private static final int NOTIFICATION_ID = 0
```

Instantiate NotificationManager

- In onCreate() of activity

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

Build and send notification

- Define notification and set required attributes

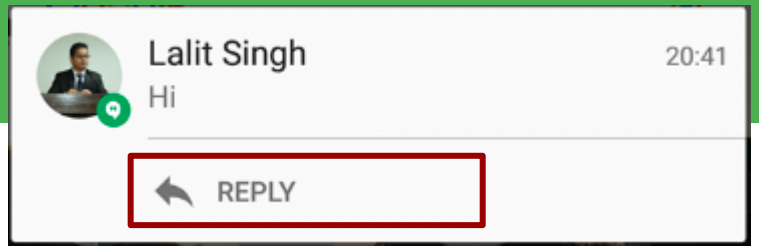
```
mNotifyBuilder = new NotificationCompat.Builder(this)
        .setContentTitle("You've been notified!")
        .setContentText("This is your notification text.")
        .setSmallIcon(R.drawable.ic_android_black_24dp);

Notification myNotification = mNotifyBuilder.build();

mNotifyManager.notify(NOTIFICATION_ID,  myNotification);
```

Actions and Pending Intents

User actions



Notifications must be able to perform actions on behalf of your application

- 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

Pending intents

- A PendingIntent is a description of an intent and target action to perform with it

- Give a PendingIntent to another application to grant it the right to perform the operation you have specified as if the other application was yourself

Pending Intents for Notifications

- Content intent is activated when the notification is tapped
  - setContent()

- Actions are choices shown to user
  - setAction()

Methods to create a PendingIntent

Use method that corresponds to intent

- PendingIntent.getActivity()
- PendingIntent.getBroadcast()
- PendingIntent.getService()

Methods all take four arguments

1. Application context
2. Request code—constant integer id for the pending intent
3. Intent to be delivered
4. PendingIntent flag determines how the system handles multiple pending intents from same app

Step 1: Create intent

```
Intent notificationIntent =
    new Intent(this, MainActivity.class);
```

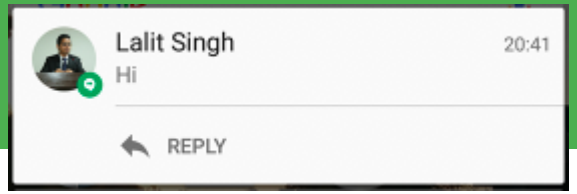
Step 2: Create PendingIntent

```
PendingIntent notificationPendingIntent =
    PendingIntent.getActivity(
                this,
                NOTIFICATION_ID,
                notificationIntent,
                PendingIntent.FLAG_UPDATE_CURRENT);
```

Step 3: Add to notification builder

```
.setContentIntent(notificationPendingIntent);
```

Add action buttons



- Use NotificationCompat.Builder.addAction()
  — pass in icon, caption, PendingIntent

```
.addAction(R.drawable.ic_color_lens_black_24dp,
           "R.string.label",
           notificationPendingIntent);
```

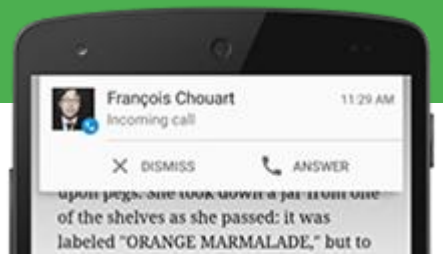
Priority and Defaults

Notification priority

- Determines how the system displays the notification with respect to other notifications
- Use NotificationCompat.Builder.setPriority()
  —pass in priority level

```
.setPriority(NotificationCompat.PRIORITY_HIGH)
```

5 notification priority levels



- `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

Notification defaults

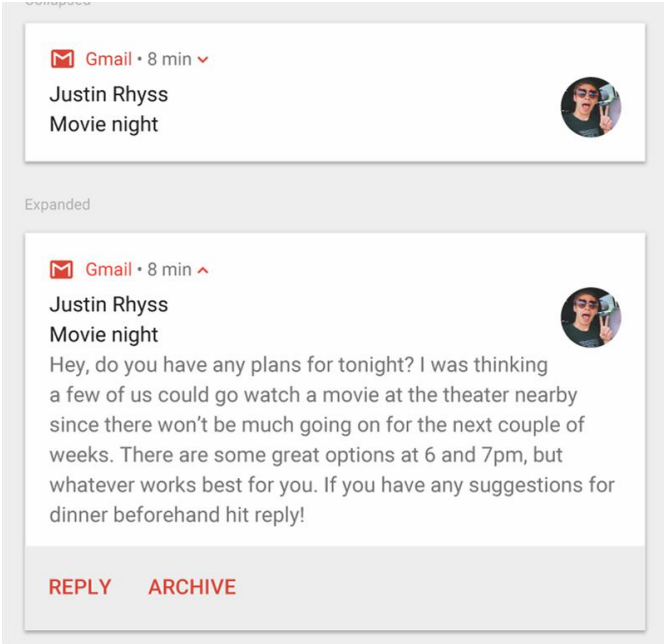
- Set sounds, vibration, and LED color pattern

```
.setDefaults(NotificationCompat.DEFAULT_ALL)
```

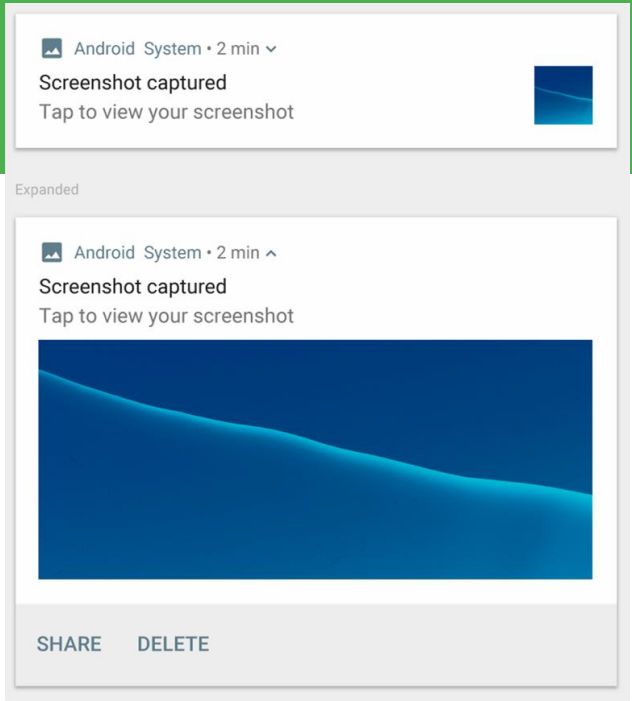
Common Layouts

Big text

- More text than will fit in standard view
- `NotificationCompat.BigTextStyle`



Big image

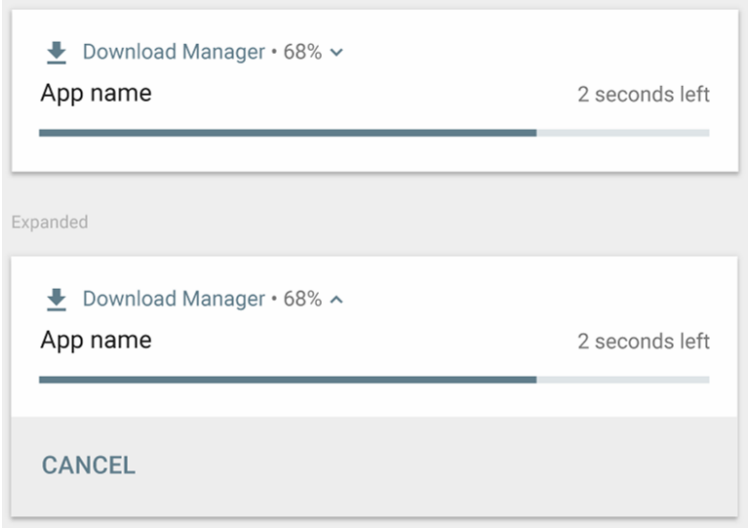


- Include image with notification

- `NotificationCompa.BigPictureStyle`

Progress bar

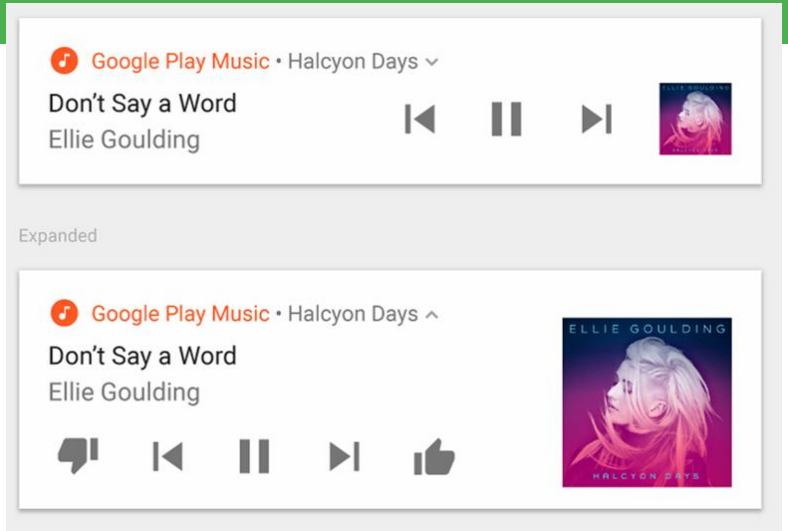
- Progress bar for ongoing task that can be cancelled
- Not a style



- `.setProgress(100, incr, false)`

Media

- Actions for controlling media such as music with image for album cover



- [NotificationCompat.MediaStyle]

Setting styles

```
mNotifyBuilder
.setStyle(new NotificationCompat.BigPictureStyle()
        .bigPicture(myBitmapImage)
        .setBigContentTitle("Notification!"));
```

Managing Notifications

Updating notifications

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

Canceling notifications

- Users can dismiss notifications
- User launches Content Intent with setAutoCancel() enabled
- App calls cancel() or cancelAll() on NotificationManager

```
mNotifyManager.cancel(NOTIFICATION_ID);
```

Design guidelines

- If your app sends too many notifications, users will disable notifications or uninstall the app

- Notifications design guide

Learn more

- Notifications
- Notification Design Guide
- NotificationCompat.Builder
- NotificationCompat.Style

What's Next?

- Concept Chapter: 8.1 C Notifications
- Practical: 8.1 P Notifications

The End

Android Developer Fundamentals

Background Tasks

Lesson 8

8.2 Alarm Manager

Contents

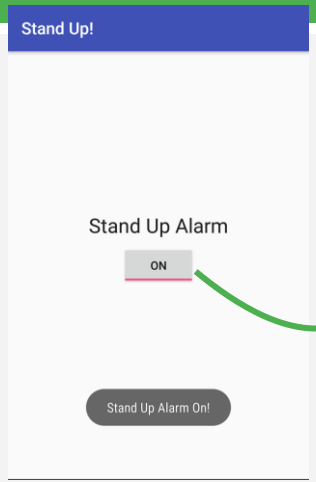
- What are Alarms
- Alarms Best Practices
- Alarm Manager
- Scheduling Alarms
- More Alarm Considerations

What Are 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
- Can be based on a real-time clock or elapsed time
- App does not need to run for alarm to be active

How alarms work with components

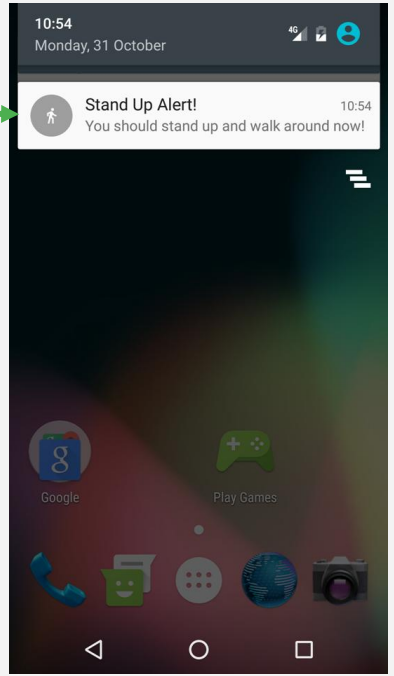


Activity creates notification manager and sets alarm

Alarm triggers and sends out intent

App may be destroyed so….

BroadcastReceiver wakes up and delivers notification



Benefits of alarms

- App does not need to run for alarm to be active
- Device does not have to be awake
- Does not use resources until it goes off
- Use with BroadcastReceiver to start services and other operations

Measuring time

- Elapsed Real Time—time since system boot
  - Independent of time zone and locale
  - Use for intervals and relative time
  - Use whenever possible
  - Elapsed time includes time device was asleep

- Real Time Clock (RTC)—UTC (wall clock) time
  - When time of day at locale matter

Wakeup behavior

- Wakes up device CPU if screen is off
  - Use only for time critical operations
  - Can drain battery

- Does not wake up device
  - Fires next time device is awake
  - Is polite

Types of alarms

	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`

Alarms Best Practices

If everybody syncs at the same time...

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"

Alarm Best Practices

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

Battery

- Minimize waking up the device
- Use inexact alarms
  - Android synchronizes multiple inexact repeating alarms and fires them at the same time
  - Reduces the drain on the battery.
  - Use setInexactRepeating() instead of setRepeating()

When not to use an alarm

- Ticks, timeouts, and while app is running—Handler
- Server sync—SyncAdapter with Cloud Messaging Service
- Inexact time and resource efficiency—JobScheduler

AlarmManager

What is AlarmManager

- AlarmManager provides access to system alarm services
- Schedules future operation
- When alarm goes off, registered Intent is broadcast
- Alarms are retained while device is asleep
- Firing alarms can wake device

Get an AlarmManager

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

Scheduling Alarms

What you need to to schedule an alarm

1. Type of alarm
2. Time to trigger
3. Interval for repeating alarms
4. PendingIntent to deliver at the specified time

   (just like notifications)

Schedule a single alarm

- set()—single, inexact alarm
- setWindow()—single inexact alarm in window of time
- setExact()—single exact alarm

More power saving options AlarmManager API 23+

Schedule a repeating alarm

- setInexactRepeating()
  - repeating, inexact alarm

- setRepeating()
  - Prior to API 19, creates a repeating, exact alarm
  - After API 19, same as setInexactRepeating()

setInexactRepeating()

```
setInexactRepeating(
    int alarmTtype,
    long triggerAtMillis,
    long intervalMillis,
    PendingIntent operation)
```

Create an inexact alarm

```
alarmManager.setInexactRepeating(
    AlarmManager.ELAPSED_REALTIME_WAKEUP,
        SystemClock.elapsedRealtime()
            + AlarmManager.INTERVAL_FIFTEEN_MINUTES,
        AlarmManager.INTERVAL_FIFTEEN_MINUTES,
        notifyPendingIntent);
```

More Alarm Considerations

Checking for an existing alarm

```
boolean alarmExists =
 (PendingIntent.getBroadcast(this,
        0, notifyIntent,
        PendingIntent.FLAG_NO_CREATE) != null);
```

Doze and Standby

- Doze—completely stationary, unplugged, and idle device
- Standby—unplugged device on idle apps
- Alarms will not fire
- API 23+

User visible alarms

- setAlarmClock()
- System UI may display time/icon
- Precise
- Works when device is idle
- App can retrieve next alarm with getNextAlarmClock()
- API 21+

Cancel an alarm

- 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

Learn more

- Schedule Repeating Alarms Guide
- AlarmManager reference
- Choosing an Alarm Blog Post
- Scheduling Alarms Presentation
- Optimizing for Doze and Standby

What's Next?

- Concept Chapter: 8.2 C Scheduling Alarms
- Practical: 8.2 P Alarm Manager

END

Android Developer Fundamentals

Background Tasks

Lesson 8

8.3 Transferring Data Efficiently & Job Scheduler

Contents

- Transferring Data Efficiently
- Job Scheduler
  - JobService
  - JobInfo
  - JobScheduler

Transferring Data Efficiently

Transferring data uses resources

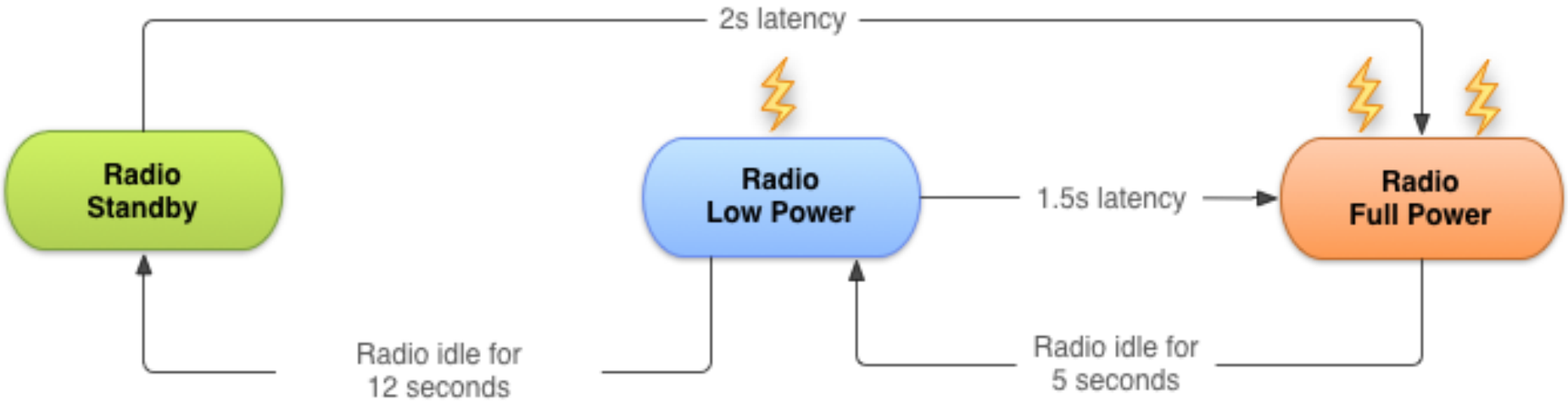
- Wireless radio uses battery
  - Device runs out of battery
  - Need to let device charge

- Transferring data uses up data plans
  - Costing users real money (for free apps…)

Wireless radio power states

- Full power—Active connection, highest rate data transfer
- Low power—Intermediate state that uses 50% less power
- Standby—Minimal energy, no active network connection

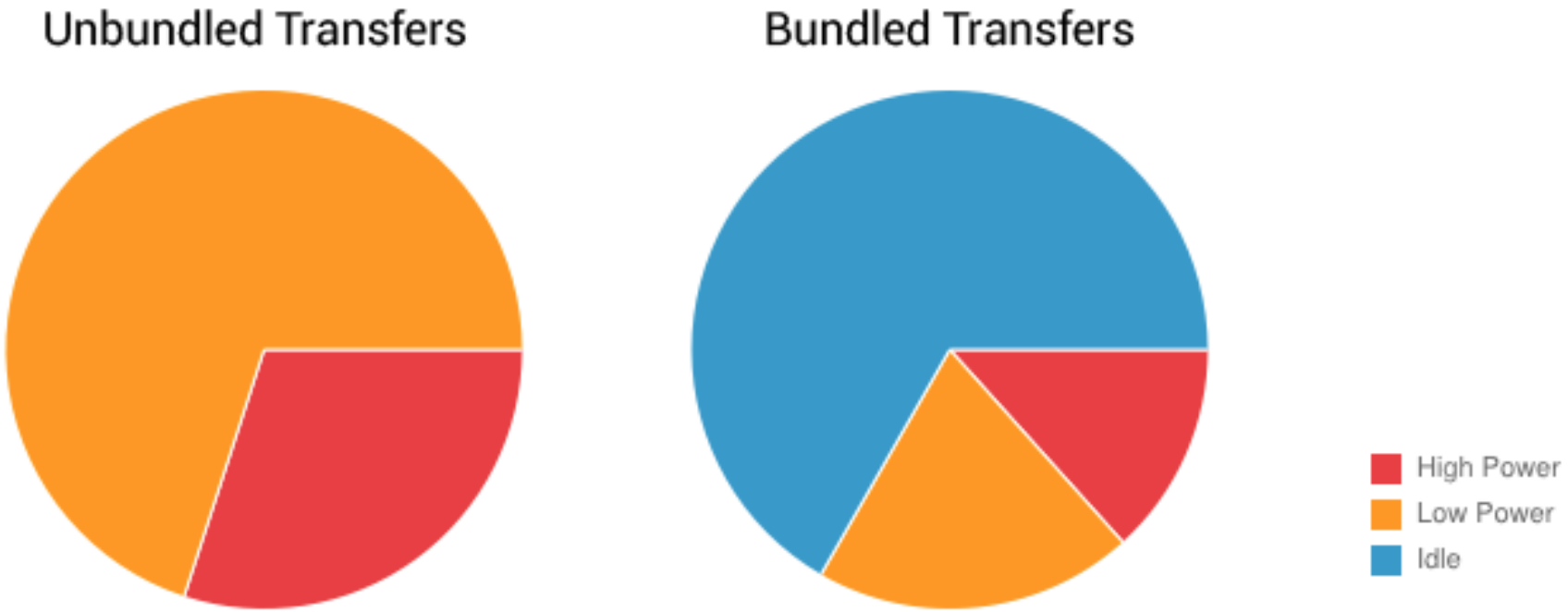
Wireless radio state transitions for 3G



Bundle network transfers

- For a typical 3G device, every data transfer session, the radio draws energy for almost 20 seconds
- Send data for 1s every 18s—radio mostly on full power
- Send data in bundles of 3s—radio mostly idle
- Bundle your data transfers

Bundled vs. unbundled



Prefetch data

- Download all the data you are likely to need for a given time period in a single burst, over a single connection, at full capacity
- If you guess right, reduces battery cost and latency
- If you guess wrong, may use more battery and data bandwidth

Monitor connectivity state

- WiFi radio uses less battery and has more bandwidth than wireless radio
- Use ConnectivityManager to determine which radio is active and adapt your strategy

Monitor battery state

- Wait for specific conditions to initiate battery intensive operation
- BatteryManager broadcasts all battery and charging details in a broadcast Intent
- Use a BroadcastReceiver registered for battery status actions

Job Scheduler

What is Job Scheduler

- Used for intelligent scheduling of background tasks
- Based on conditions, not a time schedule
- Much more efficient than AlarmManager
- Batches tasks together to minimize battery drain
- API 21+ (**no support library**)

Job Scheduler components

- JobService—Service class where the task is initiated
- JobInfo—Builder pattern to set the conditions for the task
- JobScheduler—Schedule and cancel tasks, launch service

JobService

JobService

- JobService subclass
- Override
  - onStartJob()
  - onStopJob()
- **Runs on the main thread**

onStartJob()

- Implement work to be done here
- Called by system when conditions are met
- Runs on main thread
- **Off-load heavy work to another thread**

onStartJob() returns a boolean

FALSE—Job finished

TRUE

- Work has been off-loaded
- Must call **jobFinished()** from the worker thread
- pass in JobParams object from onStartJob()

onStopJob()

- Called if system has determined execution of job must stop
- … because requirements specified no longer met
- For example, no longer on WiFi, device not idle anymore
- Before jobFinished(JobParameters, boolean)
- Return TRUE to reschedule

Basic JobService code

```
public class MyJobService extends JobService {
    private UpdateAppsAsyncTask updateTask = new UpdateAppsAsyncTask();
    @Override
    public boolean onStartJob(JobParameters params) {
        updateTask.execute(params);
        return true; // work has been offloaded
    }
    @Override
    public boolean onStopJob(JobParameters jobParameters) {
       return true;
    }
}
```

Register your JobService

```
<service
    android:name=".NotificationJobService"
    android:permission=
        "android.permission.BIND_JOB_SERVICE"/>
```

JobInfo

JobInfo

- Set conditions of execution

- JobInfo.Builder object

JobInfo builder object

- Arg 1: Job ID
- Arg 2: Service component
- Arg 3: JobService to launch

```
JobInfo.Builder builder = new JobInfo.Builder(
        JOB_ID,
        new ComponentName(getPackageName(),
        NotificationJobService.class.getName()));
```

Setting conditions

setRequiredNetworkType(int networkType)

setBackoffCriteria(long initialBackoffMillis, int backoffPolicy)

setMinimumLatency(long minLatencyMillis)

setOverrideDeadline(long maxExecutionDelayMillis)

setPeriodic(long intervalMillis)

setPersisted(boolean isPersisted)

setRequiresCharging(boolean requiresCharging)

setRequiresDeviceIdle(boolean requiresDeviceIdle)

setRequiredNetworkType()

`setRequiredNetworkType(int networkType)`

- NETWORK_TYPE_NONE—Default. No network required
- NETWORK_TYPE_ANY
- NETWORK_TYPE_NOT_ROAMING
- NETWORK_TYPE_UNMETERED

setBackOffCriteria()

`setBackoffCriteria(long initialBackoffMillis, int backoffPolicy)`

- How soon to reschedule if task fails
- Arg 1: Initial time to wait after the task fails—default 30s
- Arg 2: How long to wait subsequently
  - BACKOFF_POLICY_LINEAR
  - BACKOFF_POLICY_EXPONENTIAL
  - Default {30 seconds, Exponential}

setMinimumLatency()

`setMinimumLatency(long minLatencyMillis)`

- Minimum milliseconds to wait before completing task

setOverrideDeadline()

`setOverrideDeadline(long maxExecutionDelayMillis)`

- Maximum milliseconds to wait before running the task, even if other conditions aren't met

setPeriodic()

`setPeriodic(long intervalMillis)`

- Repeats task after a certain amount of time
- Pass in repetition interval
- Mutually exclusive with minimum latency and override deadline conditions
- Task is not guaranteed to run in the given period

setPersisted()

`setPersisted(boolean isPersisted)`

- Sets whether the job is persisted across system reboots
- Pass in True or False
- Requires RECEIVE_BOOT_COMPLETED permission

setRequiresCharging()

`setRequiresCharging(boolean requiresCharging)`

- Whether device must be plugged in
- Pass in true or false
- Defaults to false

setRequiresDeviceIdle()

`setRequiresDeviceIdle(boolean requiresDeviceIdle)`

- Whether device must be in idle mode
- Idle mode is a loose definition by the system, when device is not in use, and has not been for some time
- Use for resource-heavy jobs
- Pass in true or false. Defaults to false

JobInfo code

```
JobInfo.Builder builder = new JobInfo.Builder(
        JOB_ID, new ComponentName(getPackageName(),
        NotificationJobService.class.getName()))
        .setRequiredNetworkType(JobInfo.NETWORK_TYPE_UNMETERED)
        .setRequiresDeviceIdle(true)
        .setRequiresCharging(true);
JobInfo myJobInfo = builder.build();
```

JobScheduler

Scheduling the job

1. Obtain a JobScheduler object form the system
2. Call schedule() on JobScheduler, with JobInfo object

```
mScheduler =
    (JobScheduler)getSystemService(JOB_SCHEDULER_SERVICE);
mScheduler.schedule(myJobInfo);
```

Resources

- Transferring Data Without Draining the Battery Guide
- Optimizing Downloads for Efficient Network Access Guide
- Modifying your Download Patterns Based on the Connectivity Type Guide
- JobScheduler Reference
- JobService Reference
- JobInfo Reference
- JobInfo.Builder Reference
- JobParameters Reference
- Presentation on Scheduling Tasks

What's Next?

- Concept Chapter: 8.3 C Transferring Data Efficiently
- Practical: 8.3 P Job Scheduler

END