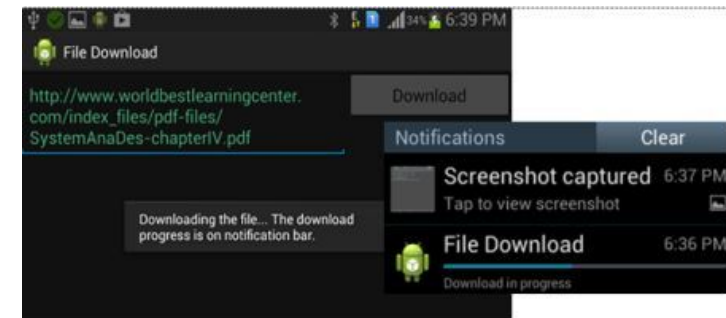
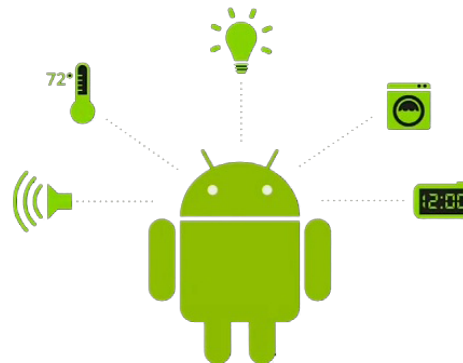
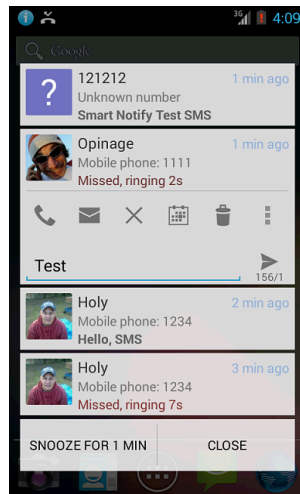
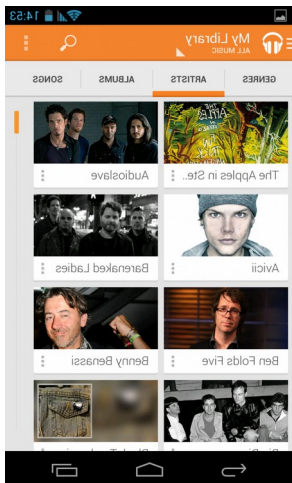


# CS 193A

## Services and Notifications

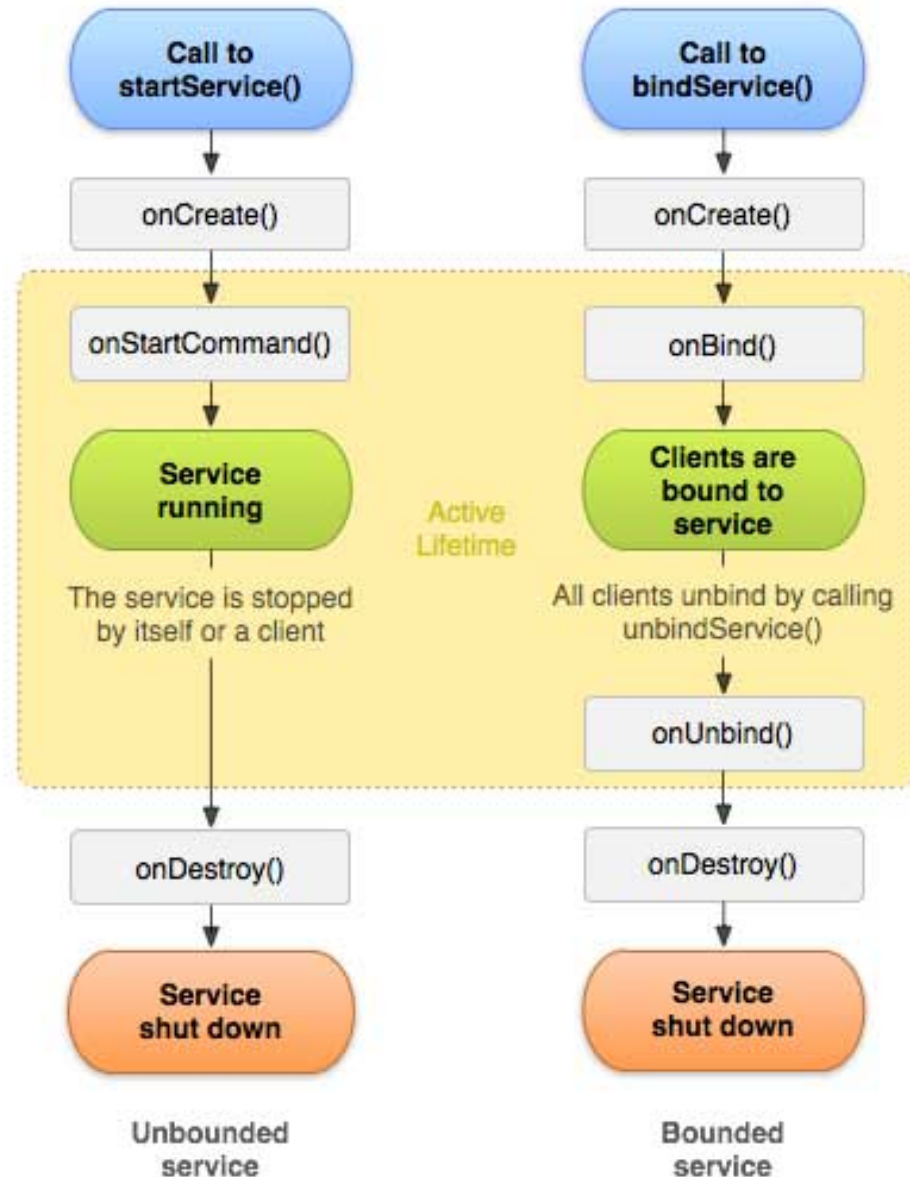
# Services

- **service:** A background task used by an app.
  - Example: Google Play Music plays the music using a service.
  - Example: Web browser runs a downloader service to retrieve a file.
  - Example: Chat app listens for new messages to come in and alerts the user, even if the user is not actively using the chat app.
  - Services are useful for long-running tasks, and/or providing functionality that can be used by other applications.



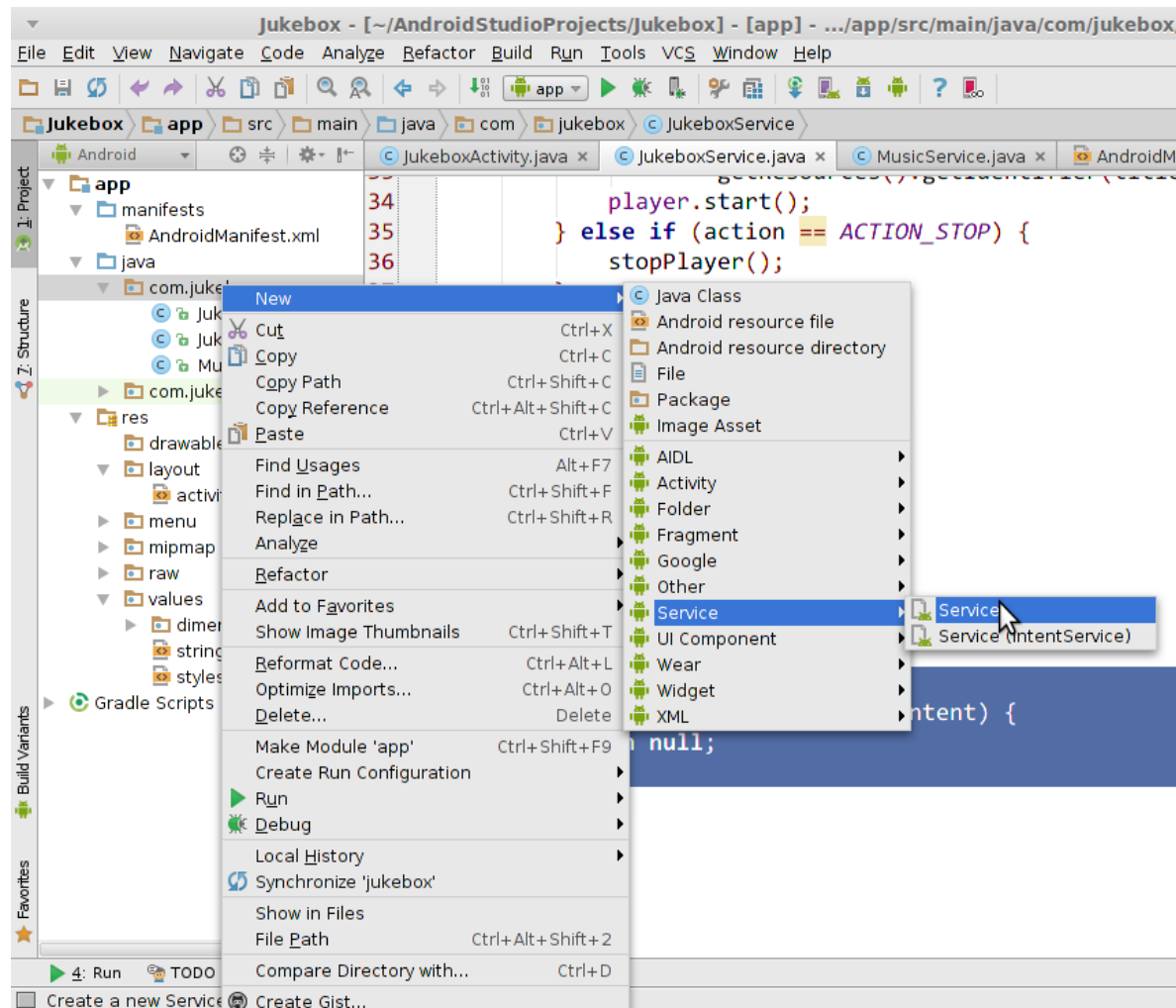
# The service lifecycle

- A service is started by an app's activity using an intent.
- Service operation modes:
  - **start**: The service keeps running until it is manually stopped.
    - *we'll use this one*
  - **bind**: The service keeps running until no "bound" apps are left.
- Services have similar methods to activities for lifecycle events.
  - onCreate, onDestroy



# Adding a service in Android Studio

- right-click your project's Java package
- click New → Service → **Service**



# Service class template

```
public class ServiceClassName extends Service {  
    /* this method handles a single incoming request */  
    @Override  
    public int onStartCommand(Intent intent, int flags, int id) {  
        // unpack any parameters that were passed to us  
        String value1 = intent.getStringExtra("key1");  
        String value2 = intent.getStringExtra("key2");  
  
        // do the work that the service needs to do ...  
  
        return START_STICKY;    // stay running  
    }  
  
    @Override  
    public IBinder onBind(Intent intent) {  
        return null;    // disable binding  
    }  
}
```

# AndroidManifest.xml changes

- To allow your app to use the service, add the following to your app's `AndroidManifest.xml` configuration:

(Android Studio does this for you if you use the New Service option)

- the `exported` attribute signifies whether other apps are also allowed to use the service (`true=yes`, `false=no`)
- note that you must write a dot ( `.` ) before the class name below!

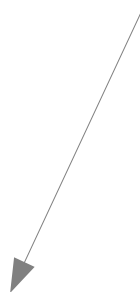
```
<application ...>
```

```
  <service
```

```
    android:name=".ServiceClassName"
```

```
    android:enabled="true"
```

```
    android:exported="false" />
```



# Starting a service

- In your Activity class:

```
Intent intent = new Intent(this, ServiceClassName.class);  
intent.putExtra("key1", "value1");  
intent.putExtra("key2", "value2");  
startService(intent);    // not startActivity!
```

- or if the same code is launched from a fragment:

```
Intent intent = new Intent(getActivity(),  
                           ServiceClassName.class);  
...
```

- or if using the Stanford library, in your SimpleActivity class:  
startService(*ServiceClassName*.class, key1, value1, ...);

# Intent actions

- Often a service has several "**actions**" or commands it can perform.
  - Example: A music player service can play, stop, pause, ...
  - Example: A chat service can send, receive, ...
- Android implements this with set/getAction methods in Intent.
  - In your Activity class:

```
Intent intent = new Intent(this, ServiceClassName.class);
intent.setAction("action");
intent.putExtra("key1", "value1");
startService(intent);
```
  - In your Service class:

```
String action = intent.getAction();
if (action.equals("action")) { ... }
```



# Broadcasting a result

- When a service has completed a task, it can notify the app by "sending a broadcast" which the app can listen for:
  - As before, set an **action** in the intent to distinguish different kinds of results.

```
public class ServiceClassName extends Service {  
    @Override  
    public int onStartCommand(Intent tent, int flags, int id) {  
        // do the work that the service needs to do ...  
        ...  
        // broadcast that the work is done  
        Intent done = new Intent();  
        done.setAction("action");  
        done.putExtra("key1", value1); ...  
        sendBroadcast(done);  
  
        return START_STICKY;    // stay running  
    }  
}
```

# Receiving a broadcast

- Your activity can hear broadcasts using a BroadcastReceiver.
  - Extend BroadcastReceiver with the code to handle the message.
  - Any extra parameters in the message come from the service's intent.

```
public class ActivityClassName extends Activity {  
    ...  
  
    private class ReceiverClassName extends BroadcastReceiver {  
        @Override  
        public void onReceive(Context context, Intent intent) {  
            // handle the received broadcast message  
            ...  
        }  
    }  
}
```

# Listening for broadcasts

- Set up your activity to be notified when certain broadcast actions occur.
  - You must pass an **intent filter** specifying the action(s) of interest.

```
public class ActivityClassName extends Activity {  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        ...  
        IntentFilter filter = new IntentFilter();  
        filter.addAction("action");  
        registerReceiver(new ReceiverClassName(), filter);  
    }  
}
```

# Broadcasts in Stanford library

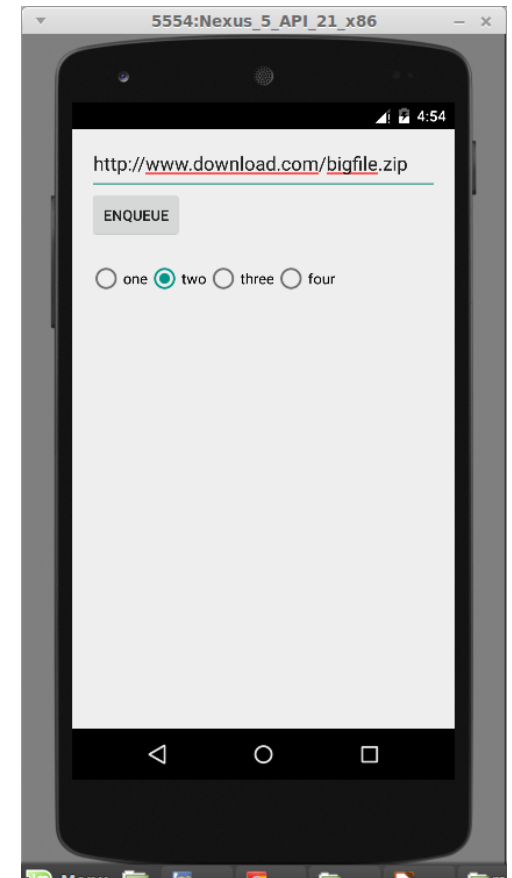


- A `SimpleActivity` can act as a broadcast receiver.
  - No need for intent filter or separate broadcast receiver class.
  - Just override the `onBroadcastReceived` method.

```
public class ActivityClassName extends SimpleActivity {  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        // register for any broadcasts you want to receive  
        // (no need for IntentFilter or BroadcastReceiver class)  
        registerReceiver("action1", "action2", ..., "actionN");  
    }  
  
    @Override  
    public void onBroadcastReceived(Intent intent) {  
        ...  
    }  
}
```

# Services and threading

- By default, a service lives in the same process and thread as the app that created it.
  - This is not ideal for long-running tasks.
  - If the service is busy, the app's UI will freeze up.
  - Example: If the Downloader app at right tries to download a large/slow file, the radio buttons and other UI elements will not respond during the download.
- To make the service and app more independent and responsive, the service should handle tasks in **threads**.



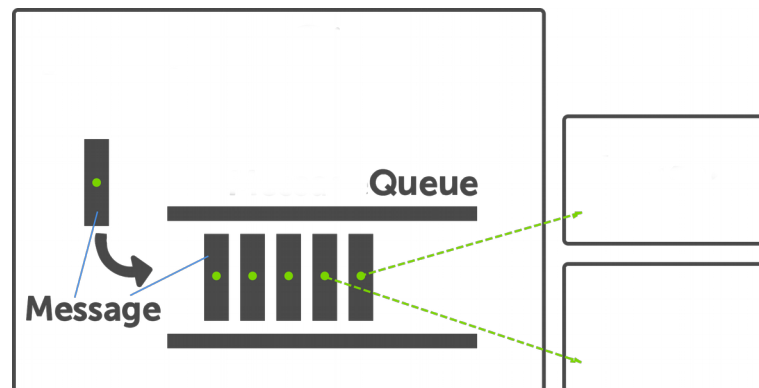
# IntentService

- Android provides a class called `IntentService` (subclass of `Service`) that runs all of its tasks in a single extra thread.
  - Great for a queue of long-running tasks to do one-at-a-time.
  - Instead of overriding `onStartCommand`, use `onHandleIntent` .
- Creating an intent service:

```
public class Name extends IntentService {  
    @Override  
    protected void onHandleIntent(Intent intent) {  
        ...  
    }  
}
```

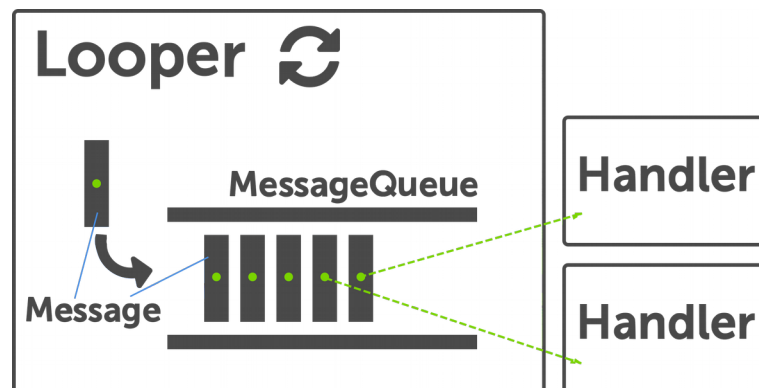
# Message queues

- **job (or message) queue:** Common pattern in Android services.
  - New jobs come in to the service via the app's intents.
  - Jobs are "queued up" in some kind of structure to be processed.
  - Handlers (usually in threads) process jobs in the order they came in.
  - As jobs finish, results are broadcast back to the app.



# Android thread helper classes

- Android provides several classes to help implement multi-threaded job/message queues:
  - `Looper`, `Handler`, `HandlerThread`, `AsyncTask`, `Loader`, `CursorLoader`, ...
  - *advantages*: easier to submit/finish jobs; easier synchronization; able to be canceled; support for thread pooling; better handling of Android lifecycle issues; ...





# Service with thread

```
public class ServiceClassName extends Service {  
    /* this method handles a single incoming request */  
    @Override  
    public int onStartCommand(Intent intent,  
                              int flags, int id) {  
        // unpack any parameters that were passed to us  
        String value1 = intent.getStringExtra("key1");  
  
        Thread thread = new Thread(new Runnable() {  
            public void run() {  
                // do the work that the service needs to do  
            }  
        });  
        thread.start();  
  
        return START_STICKY;    // stay running  
    }  
}
```

# HandlerThread ([link](#))

- HandlerThread : just a thread that has some internal data representing a queue of jobs to perform.
  - Looper : Lives inside a handler thread and performs a long-running while loop that waits for jobs and processes them. ([link](#))
  - You can give new jobs to the handler thread to process via its looper.

```
HandlerThread hThread = new HandlerThread("name");  
hThread.start();
```

```
Looper looper = hThread.getLooper();
```

```
...
```

# Handler ([link](#))

- Handler : Represents a single piece of code to handle one job in the job queue.
  - When you construct a handler, pass the **Looper** of the handler thread in which the job should be executed.
  - Submit a job to the handler by calling its post method, passing a Runnable object indicating the code to run.

```
Handler handler = new Handler(looper);
handler.post(new Runnable() {
    public void run() {
        // the code to process the job
        ...
    }
});
```

# Library SimpleService



- The library also has a `SimpleService` class.
  - Has a few convenience methods for handling intents and broadcasts.

```
public class Name extends SimpleService {  
    public int onStartCommand(Intent intent, int flags, int id) {  
        ...  
    }
```

- Also available: `SimpleIntentService`
  - A subclass of `IntentService`.

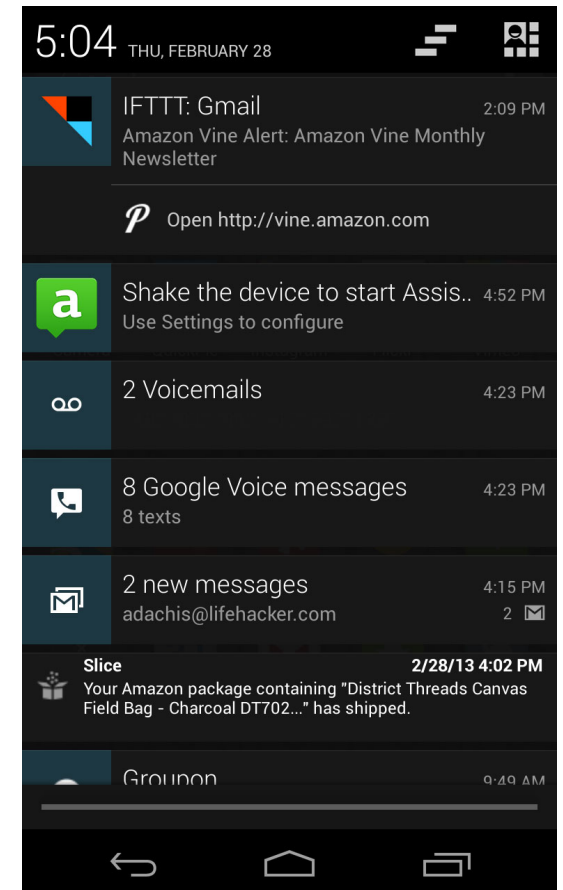
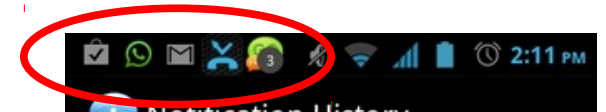
```
public class Name extends SimpleIntentService { ...  
    @Override  
    protected void onHandleIntent(Intent intent) { ... }
```

# CS 193A

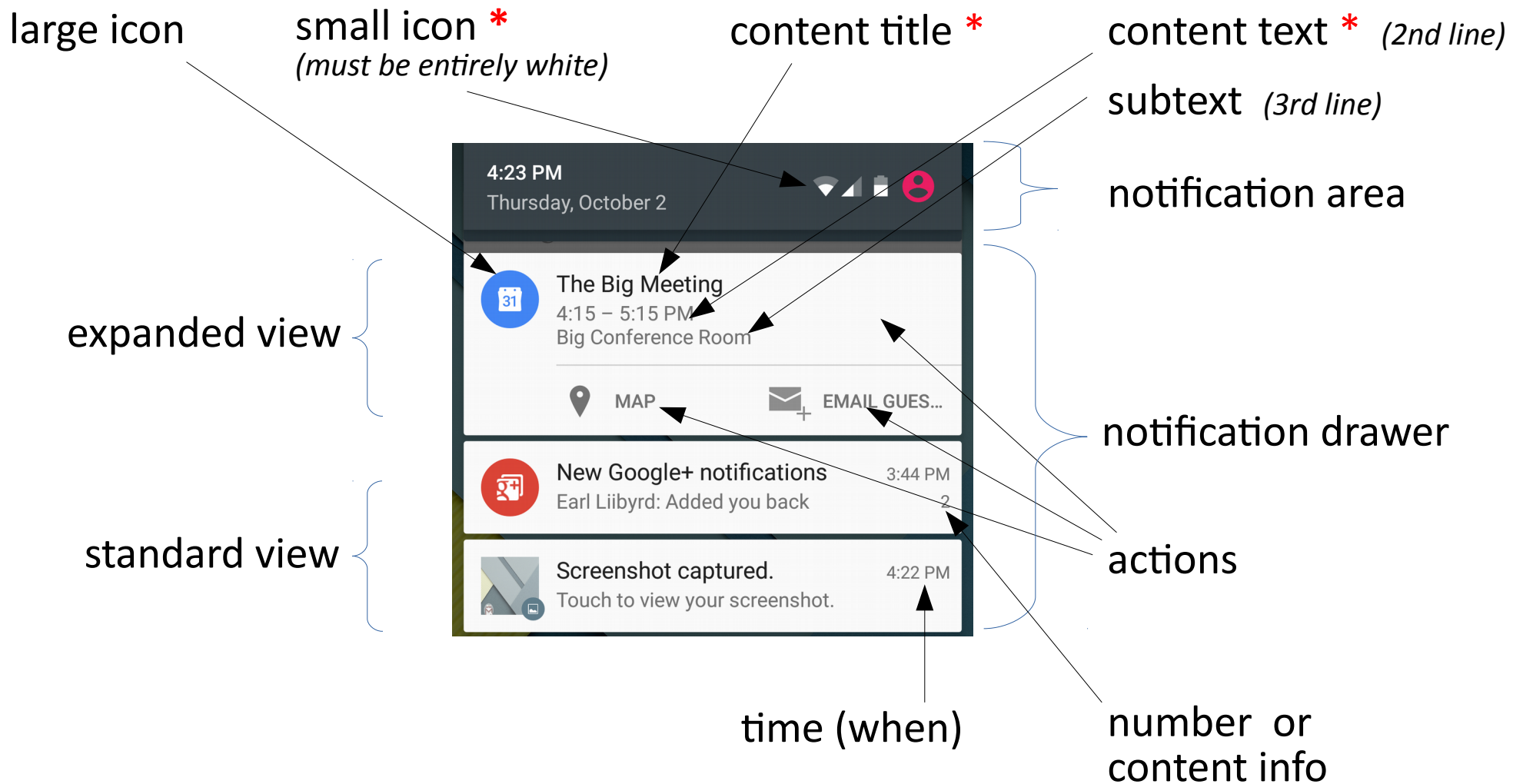
## Notifications

# Notifications

- **notification:** A message displayed to the user outside of any app's UI in a top *notification drawer* area.
  - used to indicate system events, status of service tasks, etc.
- notifications can have:
  - **icons** (small, large)
  - a **title**
  - a detailed **description**
  - one or more associated **actions** that will occur when clicked
  - ...



# Anatomy of a notification



# Creating a Notification

- Create a notification using a `Notification.Builder`.
- Use `NotificationManager` to send out the notification.

```
Notification.Builder builder = new Notification.Builder(this)
    .setContentTitle("title")
    .setContentText("text")
    .setAutoCancel(true)
    .setSmallIcon(R.drawable.icon);
Notification notification = builder.build();
```

```
NotificationManager manager = (NotificationManager)
    getSystemService(Context.NOTIFICATION_SERVICE);
manager.notify(ID, notification);
```



# Notification.Builder methods ([link](#))

Method	Description
<code>setAutoCancel(<i>boolean</i>)</code>	whether to hide when clicked
<code>setColor(<i>int</i>)</code>	background color
<code>setContentIntent(<i>Intent</i>)</code>	intent for action to run when clicked
<code>setContentText("text")</code>	detailed description
<code>setContentTitle("title")</code>	large heading text
<code>setGroup("name")</code>	group similar notifications together
<code>setLargeIcon(<i>Bitmap</i>)</code>	image for big icon
<code>setLights(<i>argb</i>, <i>onMS</i>, <i>offMS</i>)</code>	blinking lights!
<code>setNumber(<i>n</i>)</code>	a number at right of notification
<code>setOngoing(<i>boolean</i>)</code>	is this a long-term notif. that can't be dismissed?
<code>setPriority(<i>priority</i>)</code>	from <code>PRIORITY_MIN</code> to <code>PRIORITY_MAX</code>
<code>setProgress(<i>max</i>, <i>prog</i>, <i>bool</i>)</code>	sets a progress bar to <i>prog</i> out of <i>max</i>
<code>setSmallIcon(<i>id</i>)</code>	image file for icon
<code>setSound(<i>uri</i>)</code>	a sound to play
<code>setStyle(<i>style</i>)</code>	sets an expanded style when dragged down
<code>setSubText("text")</code>	third line of text (under content text)
<code>setTicker("text")</code>	text to scroll across top bar
<code>setVibrate(<i>pattern</i>)</code>	makes notification vibrate
<code>setVisibility(<i>vis</i>)</code>	whether notification should show on lock screen
<code>setWhen(<i>ms</i>)</code>	timestamp of notification

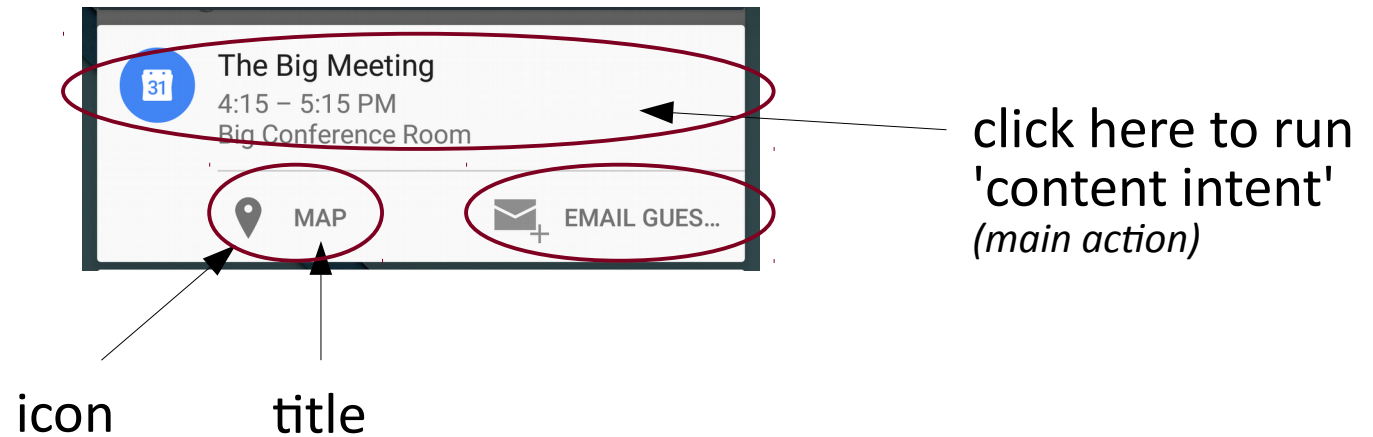
# Notification with action

- Normally when the user clicks on a notification, an action should occur. (direct the user to a particular app / activity, etc.)
  - To achieve this, use an intent inside your notification.
  - Must wrap it inside a "pending intent" object.

```
Notification.Builder builder = ...;
Intent intent = new Intent(this, ActivityClassName.class);
intent.putExtra("key1", "value1");
...
PendingIntent pending = PendingIntent.getActivity(
    this, 0, intent, 0);
builder.setContentIntent(pending);

Notification notification = builder.build();
...
```

# Anatomy of a Notif. Action



# Multiple actions (**link**)

- You can supply additional actions to a notification.
  - Build an `Action` object, then call `addAction` to add it.
  - The actions will appear underneath the expanded notification.

```
Notification.Action action =  
    new Notification.Action.Builder(iconID, "title", PendingIntent)  
    .build();
```

```
Notification.Builder builder = new Notification.Builder(this)  
    .setContentTitle("title")  
    .set ...  
    .addAction(action);  
Notification notification = builder.build();  
...
```

# Library: SimpleNotification



- Stanford library class `SimpleNotification` extends `Notification.Builder` with convenience methods:

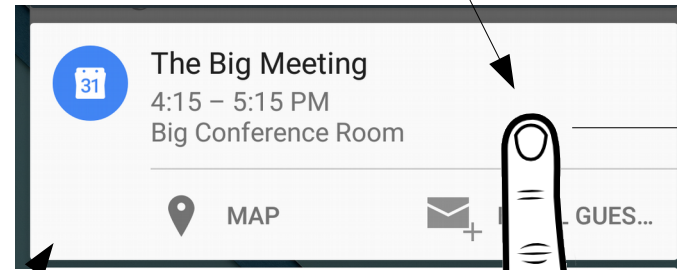
`send()`                      - combines `build()` with `NotificationManager`  
`setIntent(...)`        - simpler syntax for a pending intent  
`addAction(...)`       - simpler syntax for an action

// example

```
SimpleNotification.with(this)
    .setContentTitle("title")
    .setContentText("text")
    .setSmallIcon(R.drawable.icon)
    .setIntent(MyActivity.class, parameters)
    .addAction(iconID1, "title1", MyActivity1.class, params)
    .addAction(iconID2, "title2", MyActivity2.class, params)
    .send();
```

# Dismissing a Notification

if set to '**auto cancel**' mode,  
disappears when you click it

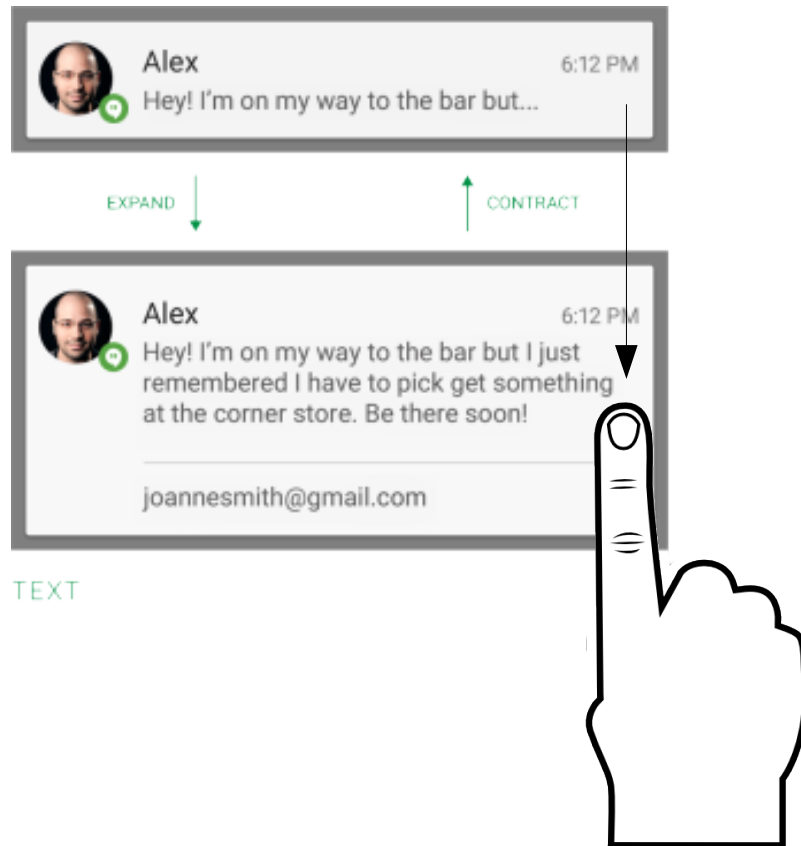


swipe to cancel  
(unless it's '*ongoing*')

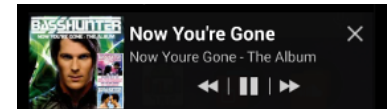
if set to '**ongoing**',  
cannot be canceled/dismissed  
(grr!)

# Expanding a Notification (**link**)

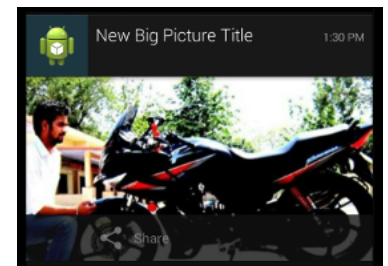
if the user drags a notification downward,  
it can show an "expanded" layout view



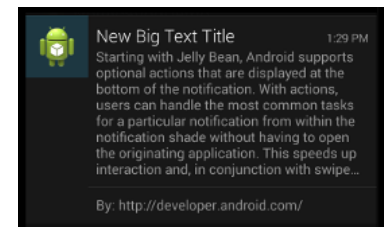
## MediaStyle



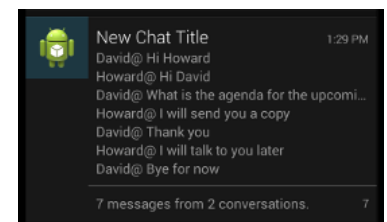
## BigPictureStyle



## BigTextStyle



## InboxStyle



# Expanded notification styles

- To make the notification expandable, use `setStyle` along with one of the `Notification.Style` subclasses.

```
Notification.BigTextStyle style =  
    new Notification.BigTextStyle()  
        .bigText("text")  
        .setBigContentTitle("title");
```

```
Notification.Builder builder = new Notification.Builder(this)  
    .set ...  
    .setStyle(style);
```

```
Notification notification = builder.build();  
...
```



# Notification.Style subclasses ([link](#))

## Methods Common to All

`s.setBigContentTitle("title")`

## Description

replacement title text

`s.setSummaryText("text")`

truncated first line of preview text to show before expanded

## BigTextStyle Method

`s.bigText("text")`

## Description

longer content text to display

## BigPictureStyle Method

`s.bigLargeIcon(bitmap)`

## Description

icon to show when expanded

`s.bigPicture(bitmap)`

big image to show in center of notification

## InboxStyle Method

`s.addLine("text")`

## Description

add a line to the message digest area

## MediaStyle Method

`s.setCancelButtonIntent(PendingIntent)`

## Description

set action for when Cancel button is pressed

`s.setMediaSession(token)`

additional playback information for system UI

`s.setShowActionsInCompactView(actions)`

actions to display even when not expanded

`s.setShowCancelButton(boolean)`

whether a top-right cancel button should appear

# Other stuff

- Want a **custom layout** for your expanded view?
  - check out RemoteViews and setContent
- Should your notification show up on the **lock screen**?
  - look into setVisibility
- Does your app generate lots of **similar notifications**?
  - group/update them by reusing IDs or addPerson
- Is your notification displaying a **long task** like a download?
  - check out setProgress
- What **state** will the app have when user clicks notification?
  - may want to make a custom activity stack with TaskStackBuilder
- need a nice **icon** for your notification?
  - get Google's material design icons at <https://design.google.com/icons/>

