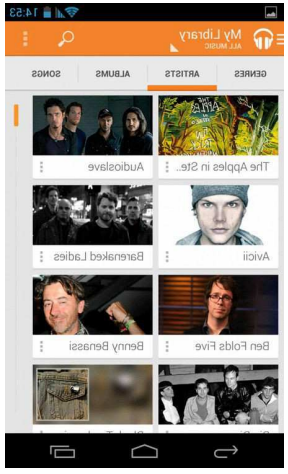
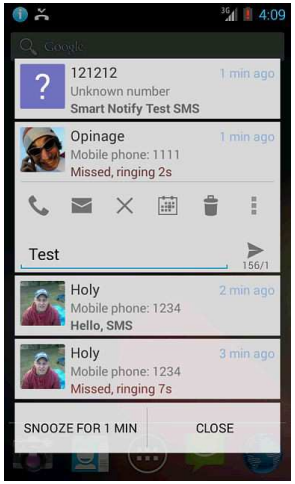
23. Services and Notifications

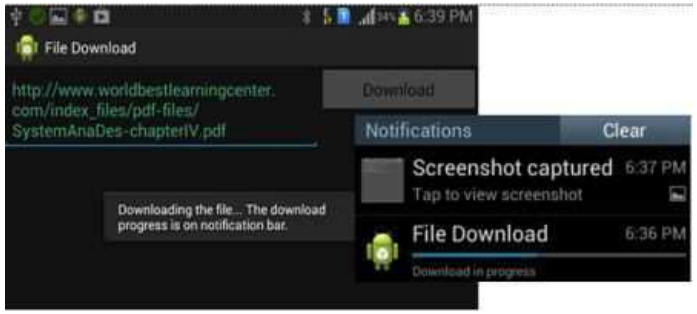
Services

- **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.





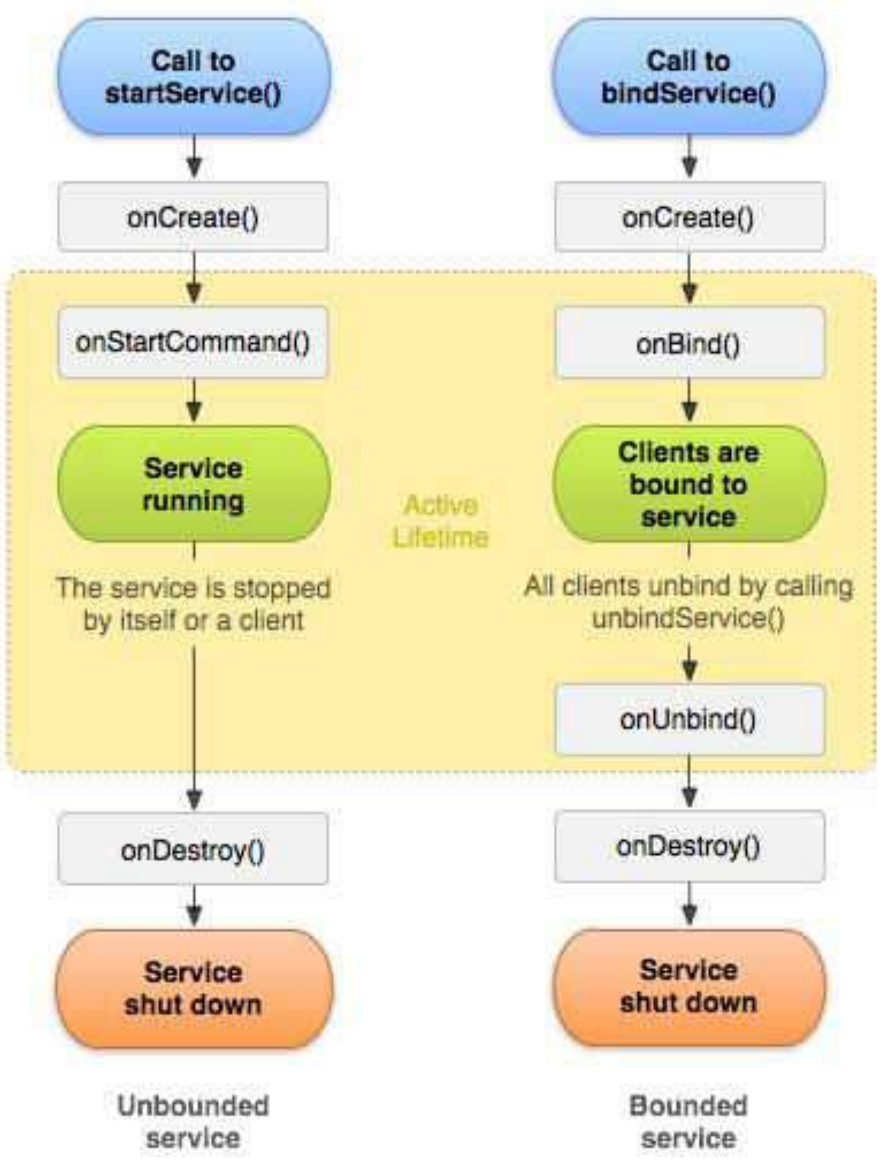


The service lifecycle

- A service is started by an app's activity using an intent.

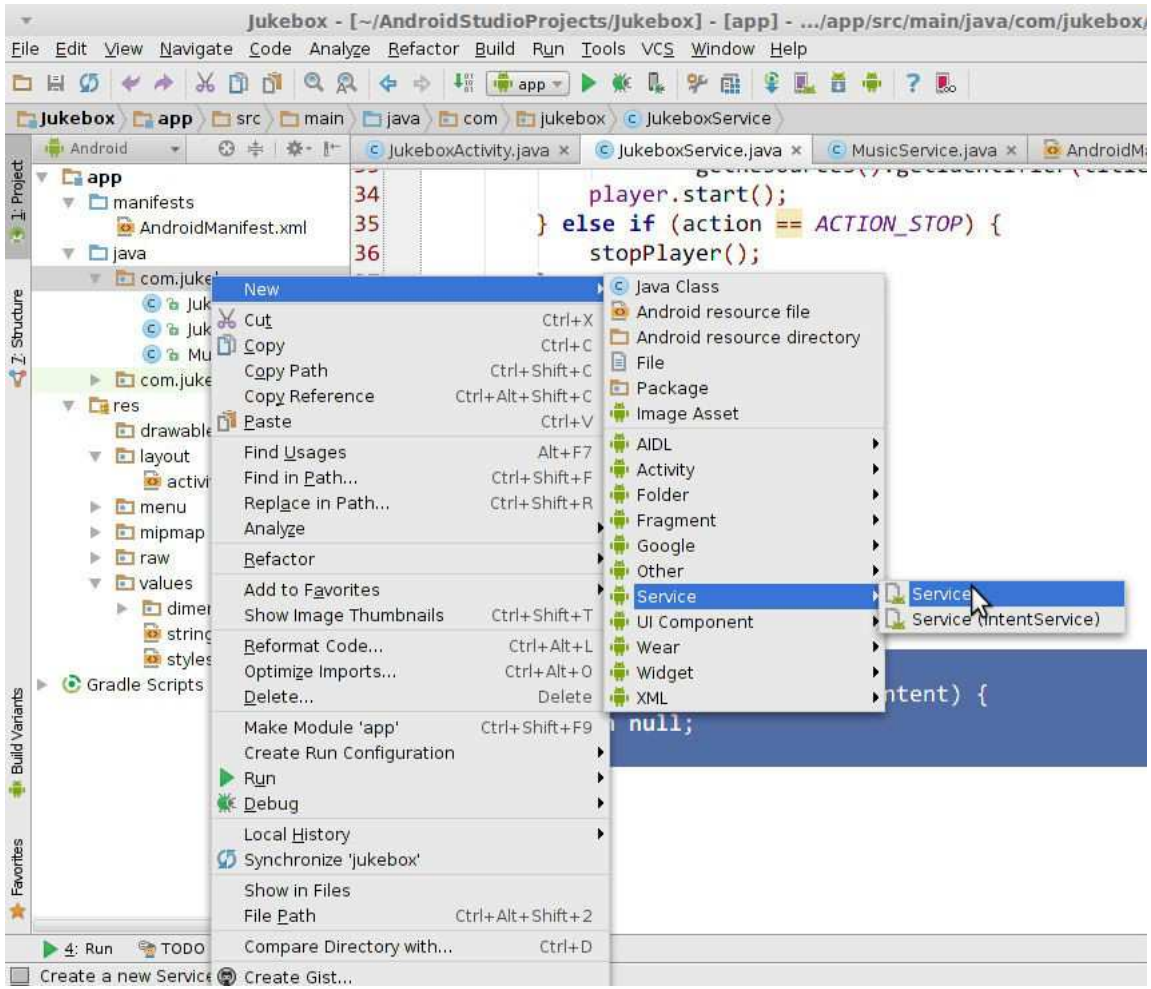
- Service operation modes:
  - **start**: The service keeps running until it is manually stopped.
    - *we'll use this one*
  - **bind**: The service keeps running until no "bound" apps are left.

- 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;   // disablebinding
    }
}
```

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);
...
```

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
            ...
        }
    }
```

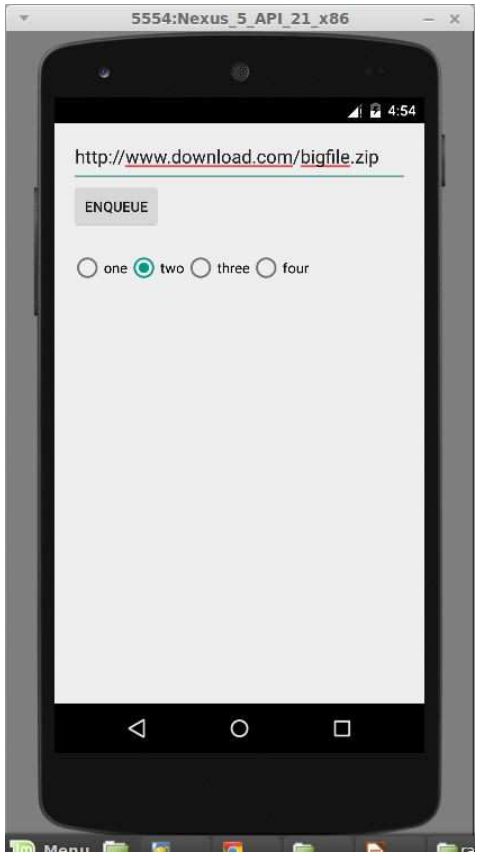
Listening for broadcasts

- 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);
    }
```

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

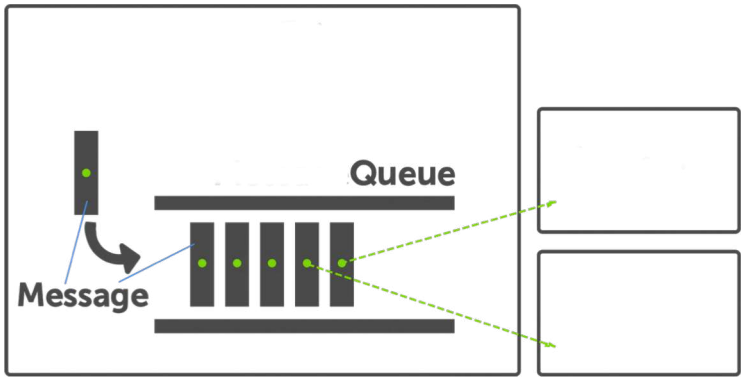
- Android provides a class called `IntentService` (subclass of `Service`) that runs all of its tasks in a single extra thread.
  - Great for a queue of long-running tasks to do one-at-a-time.
  - Instead of overriding `onStartCommand`, use `onHandleIntent` .

- Creating an intent service:

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

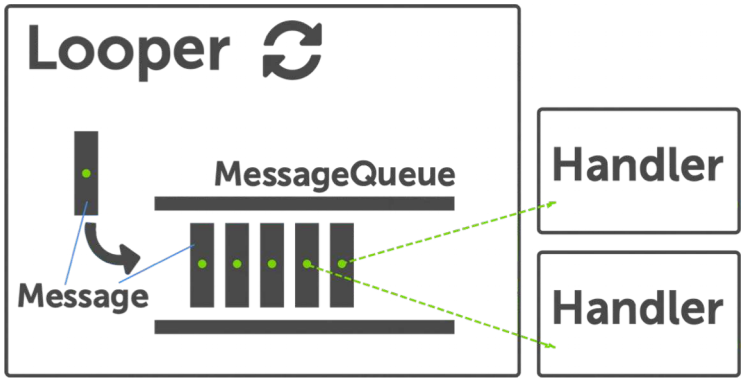
Message queues

- **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

- `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();
...
```

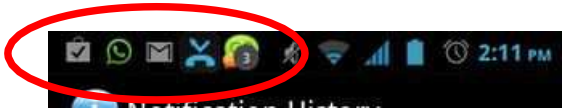
Handler

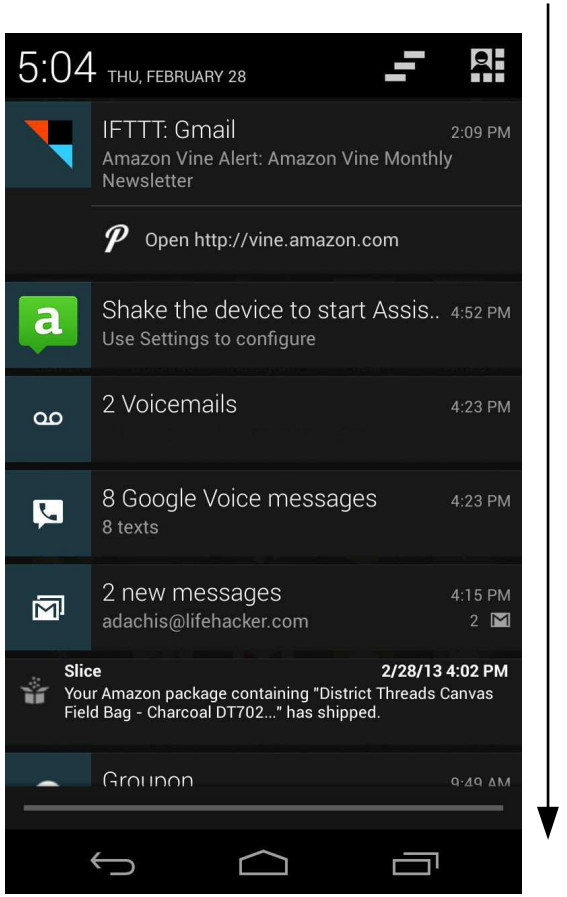
- `Handler` : Represents a single piece of code to handle one job in the job queue.
  - When you construct a handler, pass the **Looper** of the handler thread in which the job should be executed.
  - Submit a job to the handler by calling its `post` method, passing a `Runnable` object indicating the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

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

Notifications

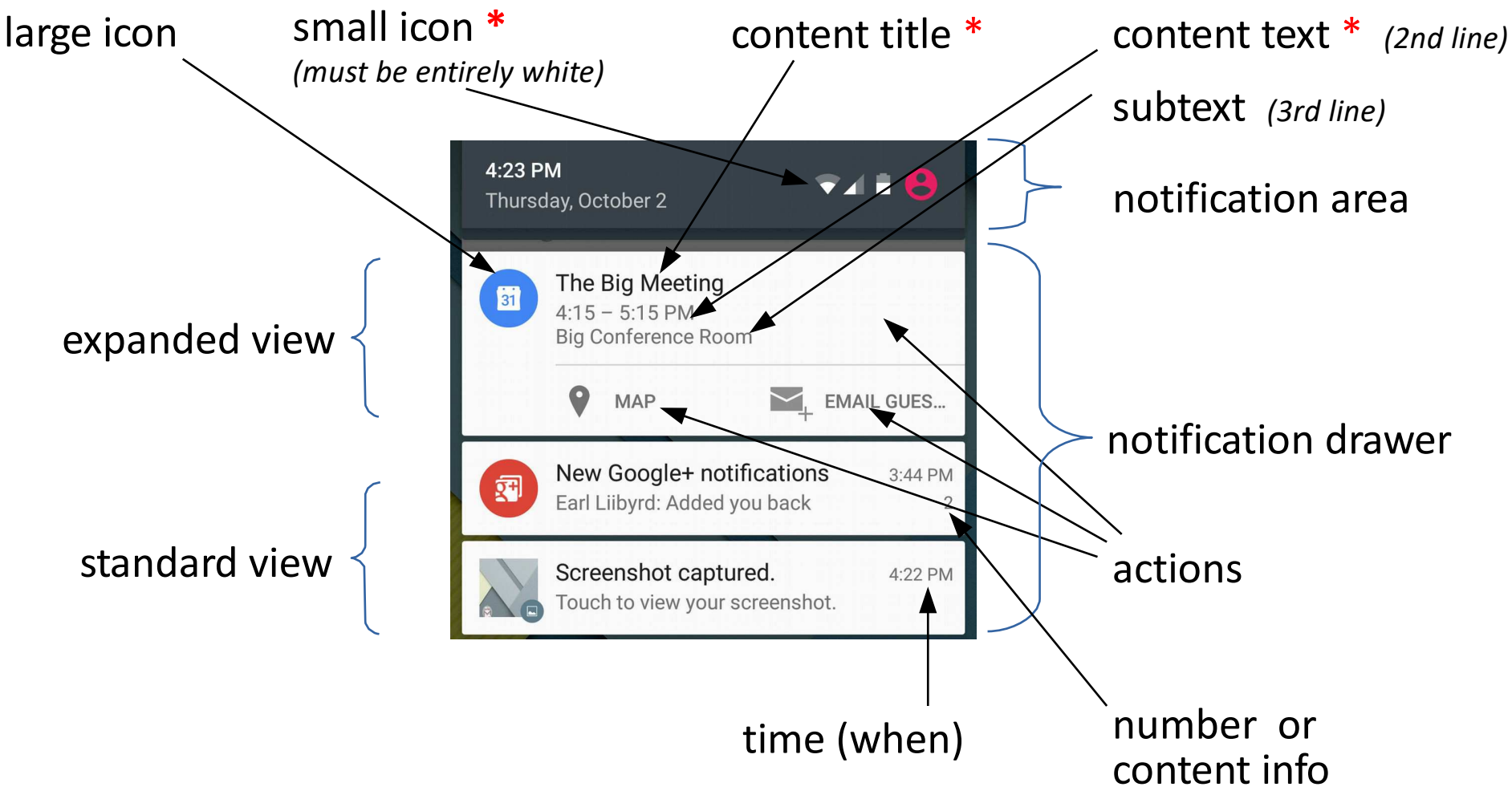
- **notification**: A message displayed to the user outside of any app's UI in a top *notification drawer* area.
  - used to indicate system events, status of service tasks, etc.





- 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



* = required

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);
```

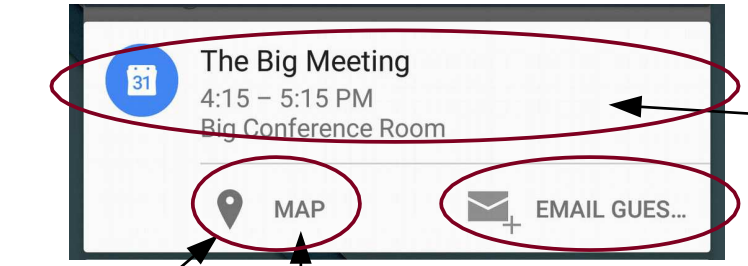
Channel: Android Oreo and above

```
NotificationManager notification_manager = (NotificationManager)
        this.getSystemService(this.NOTIFICATION_SERVICE);

if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
    String channel_id = "3000";
    CharSequence name = "Channel Name";
    String description = "Chanel Description";
    int importance = NotificationManager.IMPORTANCE_LOW;

    NotificationChannel mChannel = new NotificationChannel(channel_id,
name, importance);
    mChannel.setDescription(description);
    mChannel.enableLights(true);
    mChannel.setLightColor(Color.BLUE);
    notification_manager.createNotificationChannel(mChannel);
    notification_builder = new NotificationCompat.Builder(this,
channel_id);
} else {
    notification_builder = new NotificationCompat.Builder(this);
}
```

Notification.Builder methods

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

Anatomy of a Notif. Action



click here to run 'content intent' *(main action)*

icon

title

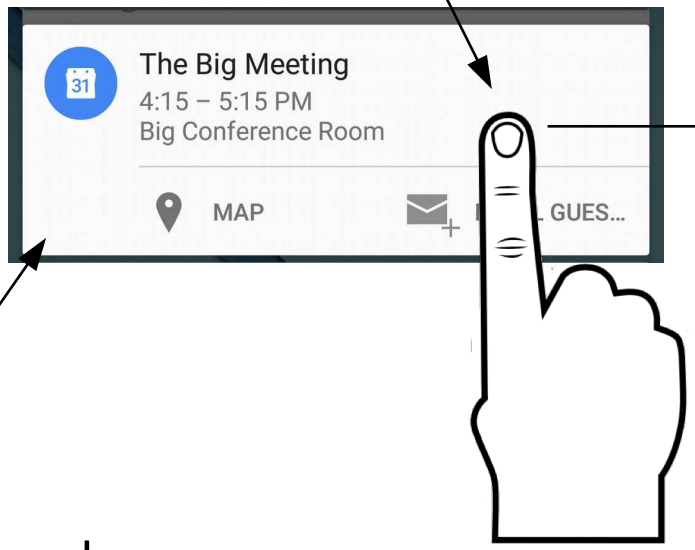
Multiple actions

- 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();
...
```

Dismissing a Notification

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

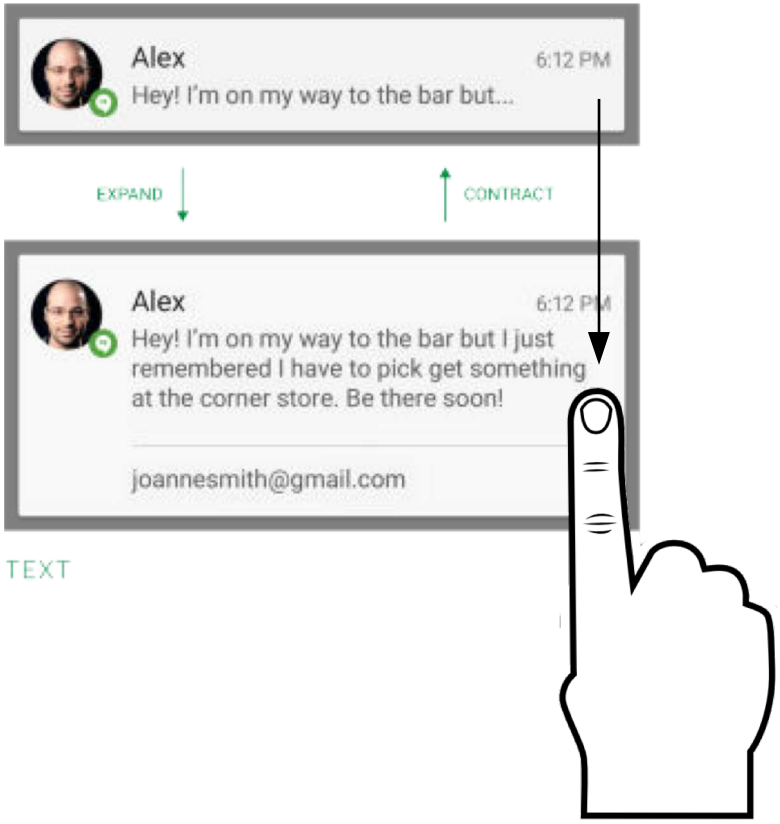


swipe to cancel
(unless it's 'ongoing')

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

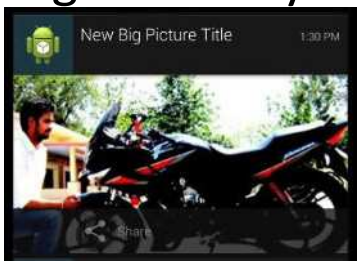
Expanding a Notification

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



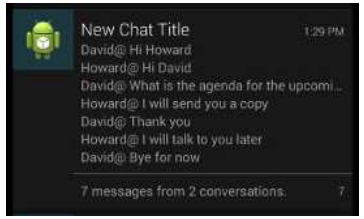
MediaStyle

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

Methods Common to All	Description
`s.setBigContentTitle("title")`	replacement title text
`s.setSummaryText("text")`	truncated first line of preview text to show before expanded

BigTextStyle Method	Description
`s.bigText("text")`	longer content text to display

BigPictureStyle Method	Description
`s.bigLargeIcon(bitmap)`	icon to show when expanded
`s.bigPicture(bitmap)`	big image to show in center of notification

InboxStyle Method	Description
`s.addLine("text")`	add a line to the message digest area

MediaStyle Method	Description
`s.setCancelButtonIntent(PendingIntent)`	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/