Lesson 8: Alarms and Schedulers

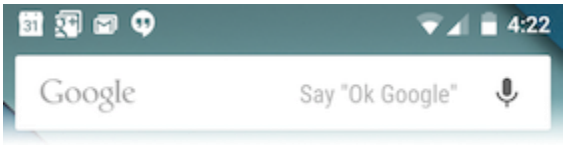
8.1: Notifications

Contents

In this chapter you learn how to create, deliver, and reuse notifications. You also learn how to make notifications compatible with different Android versions.

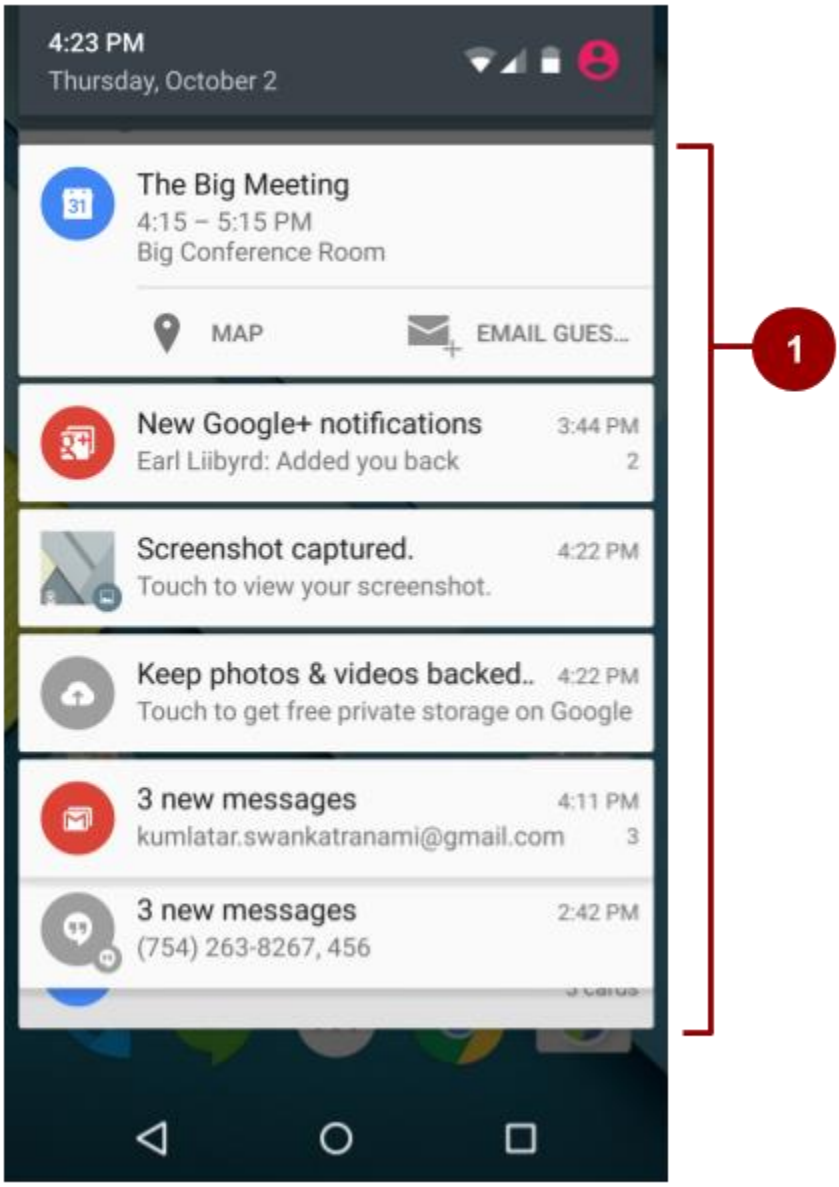
What is a notification?

A *notification* is a message your app displays to the user outside your app's normal UI. When your app tells the system to issue a notification, the notification appears to the user as an icon in the *notification area*, on the left side of the status bar.



If the device is unlocked, the user opens the *notification drawer* to see the details of the notification. If the device is locked, the user views the notification on the lock screen. The notification area, lock screen, and notification drawer are system-

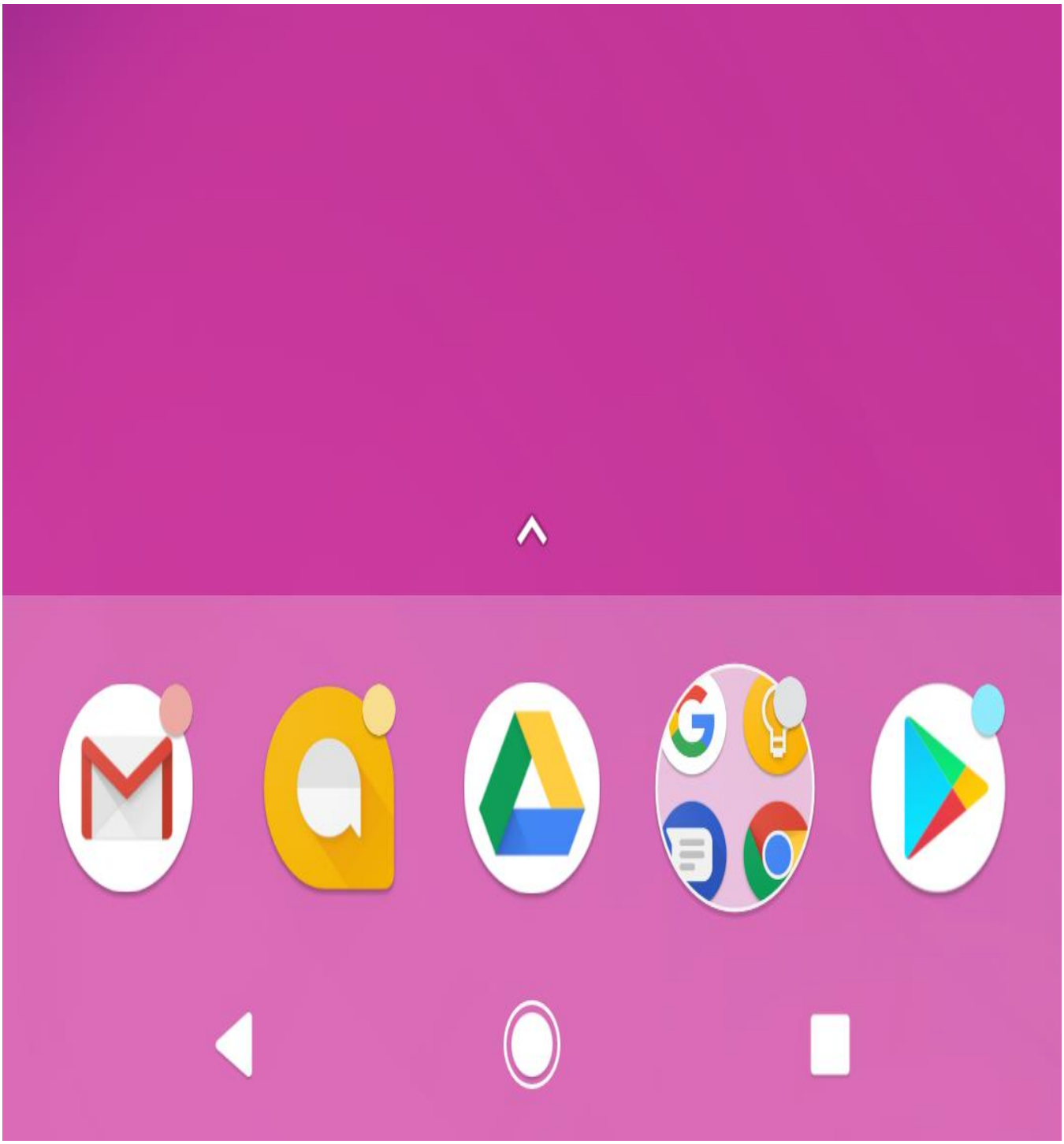
controlled areas that the user can view at any time.



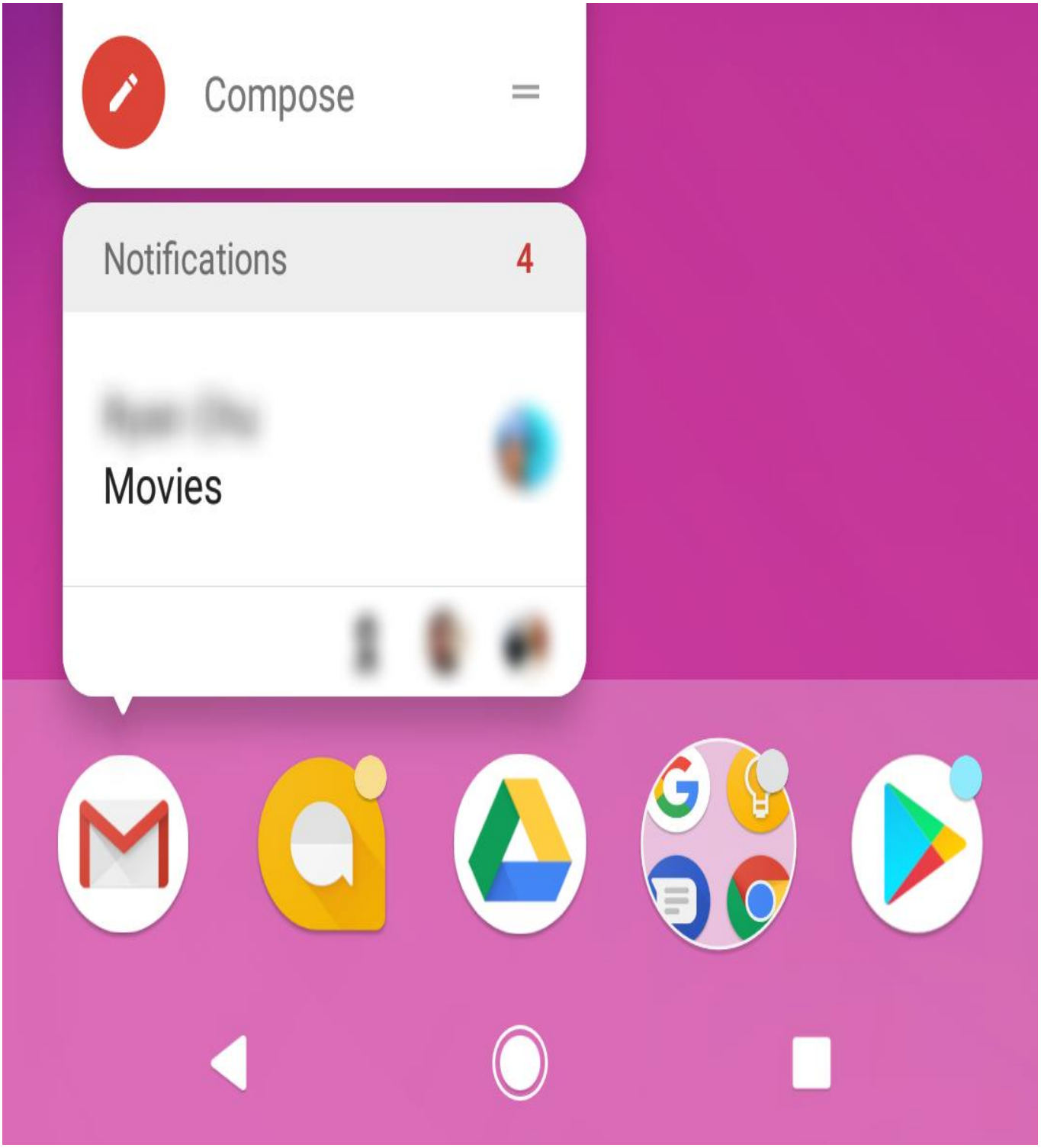
1. An "open" notification drawer. The status bar isn't visible in this screenshot, because the notification drawer is open.

App icon badge

In supported launchers on devices running Android 8.0 (API level 26) and higher, an app icon changes its appearance slightly when the app has a new notification to show to the user. The app icon shows a colored *badge*, also known as a *notification dot*, as shown on four of the five app icons in the screenshot below.



To see the notification for an app with a notification dot, the user long-presses the app icon. The notification menu appears, as shown below, and the user dismisses the notification or acts on it from the menu. This is similar to the way the user interacts with a notification in the notification drawer.



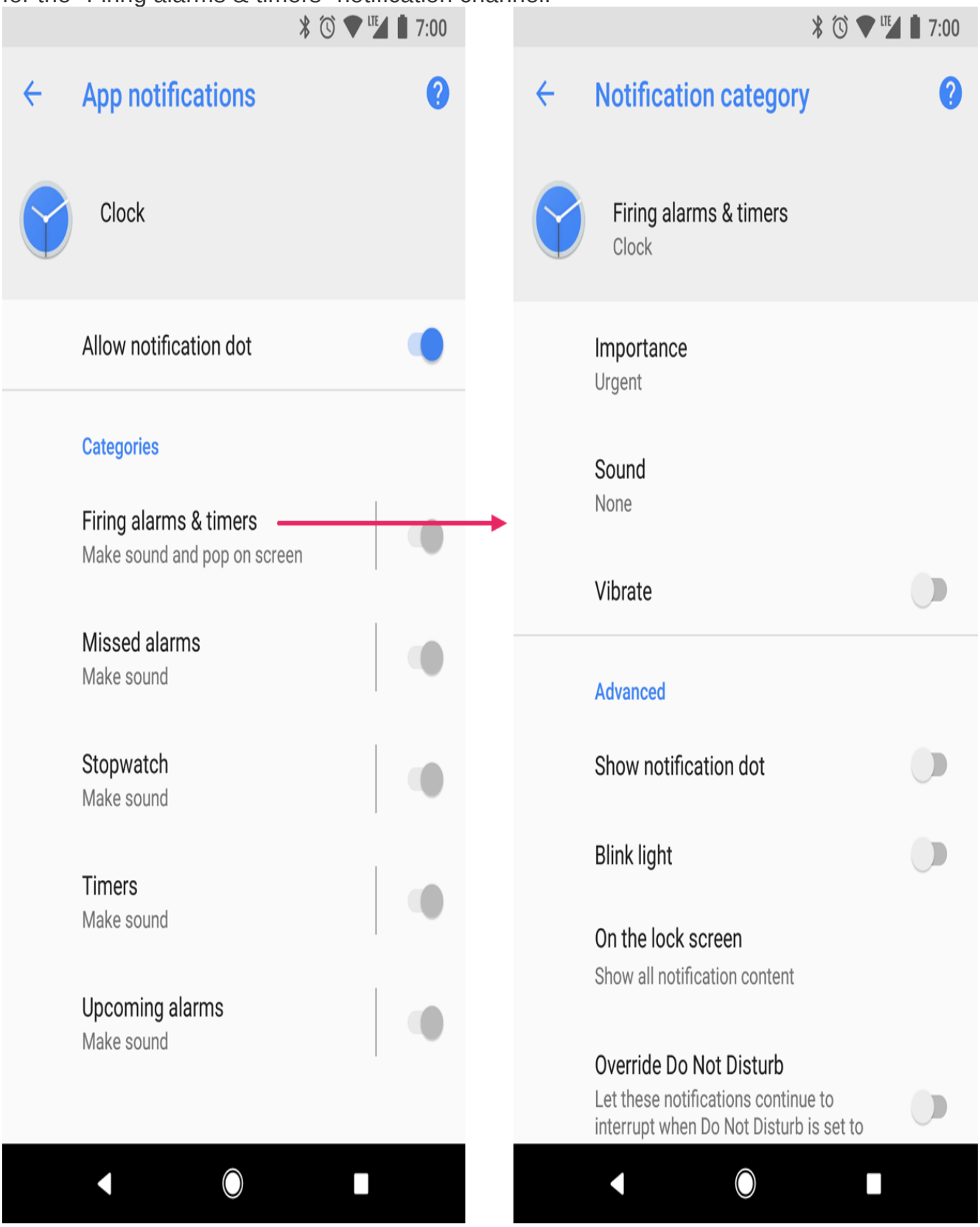
Notification channels

In the Settings app on an Android-powered device, users can adjust the notifications they receive. Starting with Android 8.0 (API level 26), you can assign each of your app's notifications to a *notification channel*. Each notification channel represents a type of notification, and you can group several notifications in each channel.

If you target only lower-end devices, you don't need to implement notification channels to display notifications, but it's good practice to target the latest available SDK, then check the device's SDK version before building the notification channel.

Notification channels are called **Categories** in the user-visible Settings app. For example, the screenshot on the left shows the notification settings for the Clock app, which has five notification channels. The screenshot on the right shows the settings

for the "Firing alarms & timers" notification channel.



When you create a notification channel in your code, you set *behavior* for that channel, and the behavior is applied to all of the notifications in the channel. For example, your app might set the notifications in a channel to play a sound, blink a light, or vibrate. Whatever behavior you set for a notification channel, the user can change it, and they can turn off notifications from your app altogether.

Note: If your app targets Android 8.0 (API level 26) or higher, you *must* implement one or more notification channels. If your `targetSdkVersion` is set to 25 or lower, when your app runs on Android 8.0 (API level 26) or higher, it behaves the same as it would on devices running Android 7.1 (API level 25) or lower.

Creating a notification channel

To create a notification channel instance, use the `NotificationChannel` constructor. Specify an ID that's unique within your package, a user-visible channel name, and an importance for the channel:

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
    NotificationChannel notificationChannel =
         new NotificationChannel(CHANNEL_ID, "Mascot Notification",
         NotificationManager.IMPORTANCE_DEFAULT);
}
```

Important: Before you use the notification-channel APIs, check the SDK version. Notification channels are only available in Android 8.0 (API level 26) and higher. Notification channels are not available in the Android Support Library.

Set the importance level

The `NotificationChannel` constructor, which is available in Android 8.0 (API level 26) and higher, requires an `importance` level. The channel's `importance` determines the instrusiveness of the notifications posted in that channel. For example, notifications with a higher importance might make sound and show up in more places than notifications with a lower importance. There are five `importance` levels, ranging from `IMPORTANCE_NONE(0)` to `IMPORTANCE_HIGH(4)`.

To support Android 7.1 (API level 25) or lower, you must also set a priority for each notification. To set a priority, use the `setPriority()` method with a priority constant from the `NotificationCompat` class.

```
mBuilder.setPriority(NotificationCompat.PRIORITY_HIGH);
```

On devices running Android 8.0 and higher, all notifications, regardless of priority and importance level, appear in the notification drawer and as app icon badges. After a notification is created and delivered, the user can change the notification channel's importance level in the Android Settings app.

The following table shows how the user-visible importance level maps to the notification-channel `importance` level and the priority constants.

User-visible importance level	Importance (Android 8.0 and higher)	Priority (Android 7.1 and lower)
Urgent Notifications make a sound and appear as heads-up notifications.	`IMPORTANCE_HIGH`	`PRIORITY_HIGH` or `PRIORITY_MAX`
High Notifications make a sound.	`IMPORTANCE_DEFAULT`	`PRIORITY_DEFAULT`
Medium Notifications make no sound.	`IMPORTANCE_LOW`	`PRIORITY_LOW`
Low Notifications make no sound and do not appear in the status bar.	`IMPORTANCE_MIN`	`PRIORITY_MIN`

Configure the initial settings

Configure the notification channel object with initial settings such as an alert sound, a notification light color, and an optional user-visible description.

```
notificationChannel.enableLights(true);
notificationChannel.setLightColor(Color.RED);
notificationChannel.enableVibration(true);
notificationChannel.setDescription("Notification from Mascot");
```

Starting from Android 8.1 (API Level 27), apps can only make a notification alert sound once per second. Alert sounds that exceed this rate aren't queued and are lost. This change doesn't affect other aspects of notification behavior, and notification messages still post as expected.

Create the notification channel

To create the notification channel, pass the instance of `NotificationChannel` to the `createNotificationChannel()` method from the `NotificationManager` class.

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
NotificationManager mNotificationManager =
        (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
mNotifyManager.createNotificationChannel(notificationChannel);
}
```

Check the SDK version before using `createNotificationChannel()`, because this API is only available on Android 8.0 and higher and in support packages.
Once you create a notification and notification channel and submit it to the `NotificationManager`, you cannot change the importance level using code. However, the user can change their preferences for your app's channels using the Android Settings app on their device.

Creating notifications

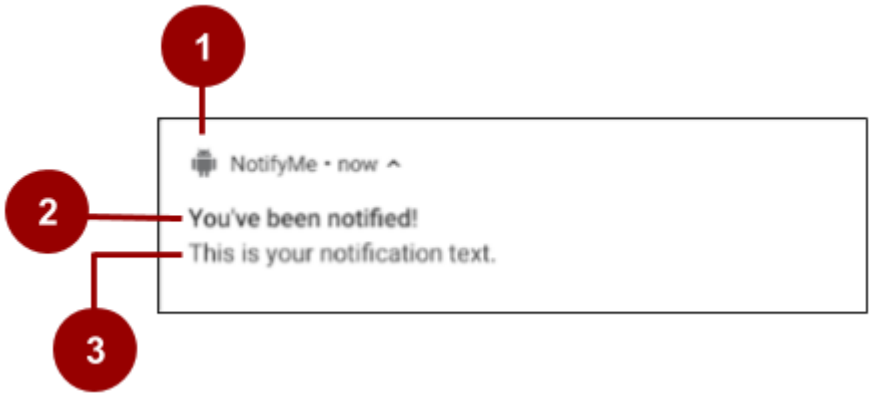
You create a notification using the `NotificationCompat.Builder` class. (`NotificationCompat` from the Android Support Library provides compatibility back to Android 4.0, API level 14. For more information, see Notification compatibility, below.) The builder classes simplify the creation of complex objects.
To create a `NotificationCompat.Builder`, pass the application context and notification channel ID to the constructor:

```
NotificationCompat.Builder mBuilder = new NotificationCompat.Builder(this,
CHANNEL_ID);
```

The `NotificationCompat.Builder` constructor takes the notification channel ID as one of its parameters. This parameter is only used by Android 8.0 (API level 26) and higher. Lower versions of Android ignore it.

Set notification contents

You can assign components to the notification like a small icon, a title, and the notification message.



In the screenshot above:

1. A small icon, set by `setSmallIcon()`. This is the only content that's required.
2. A title, set by `setContentTitle()`.
3. The body text, set by `setContentText()`. This text must be fewer than 40 characters, and it should not repeat what is in the title.

NotificationCompat.Builder mBuilder =

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

Set the intent for the notification's tap action

Every notification must respond when it is tapped, usually by launching an `Activity` in your app. To launch an `Activity` in your app, set a content intent using the `setContentIntent()` method, passing in the `Intent` wrapped in a `PendingIntent` object. When your app uses a `PendingIntent`, the system can launch the `Activity` in your app on your behalf.
To instantiate a `PendingIntent`, use one of the following methods, depending on how you want the contained `Intent` to be delivered:

- To launch an `Activity` when a user taps the notification, use `PendingIntent.getActivity()`. Pass in an explicit `Intent` for the `Activity` you want to launch. The `getActivity()` method corresponds to an `Intent` delivered using `startActivity()`.
- For an `Intent` passed into `startService()`, for example a service to download a file, use `PendingIntent.getService()`.
- For a broadcast `Intent` delivered with `sendBroadcast()`, use `PendingIntent.getBroadcast()`.

Each of these `PendingIntent` methods takes the following arguments:

- The application `context`.
- A request code, which is a constant integer ID for the `PendingIntent`.
- The `Intent` to be delivered.
- A `PendingIntent` flag that determines how the system handles multiple `PendingIntent` objects from the same app.

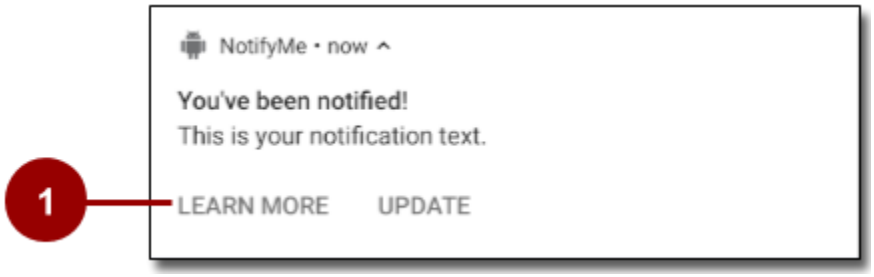
The following snippet shows how to create a basic `Intent` to open an `Activity` when the user taps the notification:

```
// Create an explicit intent for an Activity in your app
Intent contentIntent = new Intent(this, ExampleActivity.class);
PendingIntent pendingContentIntent = PendingIntent.getActivity(this, 0,
                        contentIntent, PendingIntent.FLAG_UPDATE_CURRENT);

// Set the intent that will fire when the user taps the notification
mBuilder.setContentIntent(pendingContentIntent);
```

Add notification action buttons

Notification *action buttons* allow the user to perform an app-related task without launching the app. The system typically displays action buttons adjacent to the notification content. A notification can have up to three notification action buttons.



1. "Learn more" and "Update" action buttons

Using action buttons, you can let the user perform a variety of actions, beyond just launching an `Activity` after the user taps the notification itself. For example, you can use action buttons to let the user start a background task to upload a file, place a phone call, snooze an alarm, or play music. For Android 7.0 (API level 24) and higher, you can use an action button to let the user reply to a message directly from a notification.

Adding an action button is similar to setting up the notification's default tap action: pass a `PendingIntent` to the `addAction()` method in the `NotificationCompat.Builder` class. But this action should not replicate what happens when the user taps the notification itself.

The following code shows how to add an action button using the `addAction()` method with the `NotificationCompat.Builder` object, passing in the icon, the title string for the label, and the `PendingIntent` to trigger when the user taps the action button.

```
mBuilder.addAction(R.drawable.car, "Get Directions", mapPendingIntent);
```

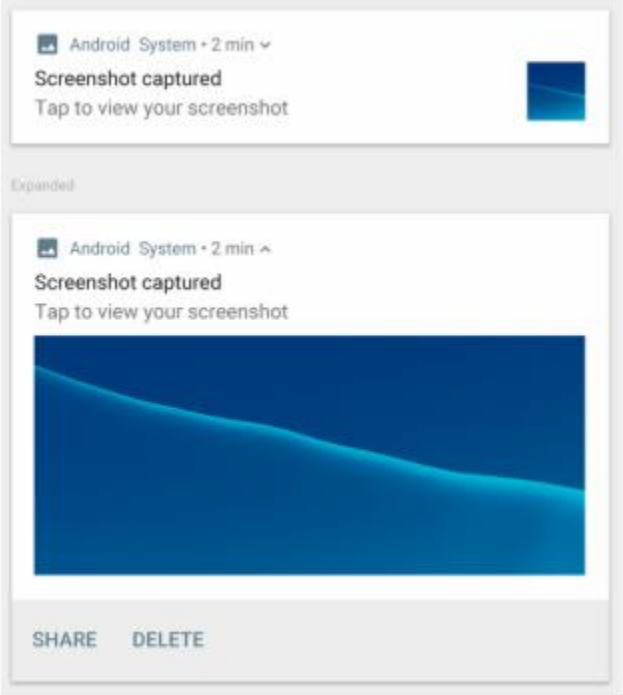
Starting from Android 7.0, icons are not displayed in notifications. Instead, more room is provided for the notification labels themselves. But notification action icons are still required, and they are used on older versions of Android and on devices such as Android Wear.

Expandable notifications

Notifications in the notification drawer appear in two main layouts, *normal view* (which is the default) and *expanded view*. Expanded view notifications were introduced in Android 4.1. Use them sparingly, because they take up more space and attention than normal view layouts.

To create notifications that appear in an expanded layout, use one of these helper classes to set the style object to the `setStyle()` method:

- Use `NotificationCompat.BigTextStyle` for large-format notifications that include a lot of text.
- Use `NotificationCompat.InboxStyle` for displaying a list of summary lines, for example for incoming messages or emails.
- Use `NotificationCompat.MediaStyle` for media playback notifications.
- Use `NotificationCompat.MessagingStyle` to display sequential messages in an ongoing conversation. This style currently applies only on devices running Android 7.0 and higher. On lower devices, these notifications are displayed in the supported style.
- Use `NotificationCompat.BigPictureStyle` for large-format notifications that include large image attachments, as shown in the screenshot below.



For example, here's how you'd set the `BigPictureStyle` on a notification:

```
NotificationCompat notif = new NotificationCompat.Builder(mContext, channelId)
    .setContentTitle("New photo from " + sender.toString())
    .setContentText(subject)
    .setSmallIcon(R.drawable.new_post)
    .setLargeIcon(aBitmap)
    .setStyle(new NotificationCompat.BigPictureStyle()
        .bigPicture(aBigBitmap)
        .setBigContentTitle("Large Notification Title"))
    .build();
```

To learn more about implementing expanded styles, see the `NotificationCompat.Style` documentation.

Ongoing notifications

Ongoing notifications are notifications that the user can't dismiss. Use ongoing notifications for background tasks that the user actively engages with, for example playing music. You can also use ongoing notifications to show tasks that are occupying the device, for example file downloads, sync operations, and active network connections.

Ongoing notifications can be a nuisance to your users, because users can't cancel them, so use them sparingly.

To make a notification ongoing, set `setOngoing()` to `true`. Your app must explicitly cancel ongoing notifications by calling `cancel()` or `cancelAll()`.

Delivering notifications

Use the `NotificationManager` class to deliver notifications:

1. To create an instance of `NotificationManager`, call `getSystemService()`, passing in the `NOTIFICATION_SERVICE` constant.
2. To deliver the notification, call `notify()`.

Pass these two values in the `notify()` method:

- A notification ID, which is used to update or cancel the notification.
- The `NotificationCompat` object that you created using the `NotificationCompat.Builder` object.

The following example creates a `NotificationManager` instance, then builds and delivers a notification:

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

//Builds the notification with all the parameters
NotificationCompat.Builder notifyBuilder =
                new NotificationCompat.Builder(this, PRIMARY_CHANNEL)
       .setContentTitle(getString(R.string.notification_title))
       .setContentText(getString(R.string.notification_text))
       .setSmallIcon(R.drawable.ic_android)
       .setContentIntent(notificationPendingIntent)
       .setPriority(NotificationCompat.PRIORITY_HIGH)
 .setDefaults(NotificationCompat.DEFAULT_ALL);

//Delivers the notification
mNotifyManager.notify(NOTIFICATION_ID, notifyBuilder.build());
```

Updating and reusing notifications

Sometimes you need to issue a notification multiple times for the same type of event. In this situation, you can update a previous notification by changing some of the notification's values, adding to the notification, or both.

To reuse an existing notification:

1. Update a `NotificationCompat.Builder` object and build a `Notification` object from it, as when you first created and built the notification.
2. Deliver the notification with the same ID you used previously.

Important: If the previous notification is still visible, the system updates it from the contents of the `Notification` object. If the previous notification has been dismissed, a new notification is created and displayed.

Clearing notifications

Notifications remain visible until one of the following happens:

- If the notification can be cleared, it disappears when the user dismisses it by swiping it or by using "Clear All".
- If you called `setAutoCancel()` when you created the notification, the notification cancels itself automatically. When the user taps the notification, the notification is removed from the status bar.
- If you call `cancel()` on the `Notification` object for a specific notification ID, the notification is removed from the status bar.
- If you call `cancelAll()` on the `Notification` object, all the notifications you've issued are removed from the status bar.

Because the user can't cancel ongoing notifications, your app must cancel them by calling `cancel()` or `cancelAll()` on the `Notification` object.

Notification compatibility

To ensure the best compatibility, create notifications with `NotificationCompat` and its subclasses. In particular, use `NotificationCompat.Builder` from the Android Support library.
Keep in mind that not all notification features are available for every Android version, even though the methods to set them are in the `NotificationCompat.Builder` class. For example, expanded view layouts for notifications are only available on Android 4.1 and higher, but action buttons depend on expanded view layouts. If you use notification action buttons, they don't show up on devices running Android versions lower than 4.1.
To solve this:

- Don't rely only on notification action buttons to carry out a notification's action. Instead, give your users a way to perform the same functionality from inside your app.

  For example, if you set a notification action that lets the user stop and start media playback, first implement this functionality in an `Activity` in your app.
- Have the `Activity` start when the user taps the notification.
- Use `addAction()` to add features to the notification as needed. Remember that any functionality you add also has to be available in the `Activity` that starts when users tap the notification.

Notification design guidelines

Notifications always interrupt the user, so they should be short, timely, and most of all, relevant.

- **Relevant:** Ask yourself whether this information is essential for the user. What happens if the user doesn't get the notification? For example, scheduled calendar events are probably relevant.
- **Timely:** Notifications need to appear when they are useful. For example, notifying the user when it's time to leave for an appointment is useful.
- **Short:** Use as few words as possible.

For more notification design guidelines, see the Material Design spec for notifications.

https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/

Related practical

The related practical is in 8.1: Notifications

Learn more

Guides:

- Notifications Overview
- Material Design spec for notifications

Reference:

- `NotificationCompat.Builder` reference
- `NotificationCompat.Style` reference

Background Tasks

8.2 Alarms

Contents

You already know how to use broadcast receivers to make your app respond to system events even when your app isn't running. In this chapter, you learn how to use alarms to trigger tasks at specific times, whether or not your app is running when the alarms go off. Alarms can either be single use or repeating. For example, you can use a repeating alarm to schedule a download every day at the same time.

To create alarms, you use the `AlarmManager` class. Alarms in Android have the following characteristics:

- Alarms let you send an `Intent` at set times or intervals. You can use alarms with broadcast receivers to start services and perform other operations.
- Alarms operate outside your app. You can use them to trigger events or actions even when your app isn't running, and even if the device is asleep.
- When used correctly, alarms can help you minimize your app's resource requirements. For example, you can schedule operations without relying on timers or continuously running background services.

When *not* to use alarms:

- Don't use alarms for timing events such as ticks and timeouts, or for timed operations that are guaranteed to happen during the lifetime of your app. Instead, use the `Handler` class with `Timer` and `Thread`. This approach gives Android better control over system resources than if you used alarms.
- Don't use alarms for server sync operations. Instead, use `SyncAdapter` with Firebase Cloud Messaging.
- You might not want to use alarms for tasks that can wait until conditions are favorable, such as when the device is connected to Wi-Fi and is charging. (These tasks include things like updating weather information or news stories.) For these tasks on devices running API 21 or higher, consider using `JobScheduler`, which you learn about in an upcoming lesson.
- Don't use alarms for tasks that have to happen even if the device is idle. When a device is in Doze mode (idle), scheduled alarms are deferred until the device exits Doze. To guarantee that alarms execute even if the device is idle, use `setAndAllowWhileIdle()` or `setExactAndAllowWhileIdle()`. Another option is to use the new `WorkManager` API, which is built to perform background work either once or periodically. For details, see Schedule tasks with WorkManager.

Alarm types

There are two general types of alarms in Android: *elapsed real-time alarms* and *real-time clock (RTC)* alarms, and both use `PendingIntent` objects.

Elapsed real-time alarms

Elapsed real-time alarms use the time, in milliseconds, since the device was booted. Time zones don't affect elapsed real-time alarms, so these alarms work well for alarms based on the passage of time. For example, use an elapsed real-time alarm for an alarm that fires every half hour.

The `AlarmManager` class provides two types of elapsed real-time alarm:

- `ELAPSED_REALTIME`: Fires a `PendingIntent` based on the amount of time since the device was booted, but doesn't wake the device. The elapsed time includes any time during which the device was asleep. All repeating alarms fire when your device is next awake.
- `ELAPSED_REALTIME_WAKEUP`: Fires the `PendingIntent` after the specified length of time has elapsed since device boot. If the screen is off, this alarm wakes the device's CPU. If your app has a time dependency, for example if your app has a limited window during which to perform an operation, use this alarm instead of `ELAPSED_REALTIME`.

Real-time clock (RTC) alarms

Real-time clock (RTC) alarms are clock-based alarms that use Coordinated Universal Time (UTC). Only choose an RTC alarm in these types of situations:

- You need your alarm to fire at a particular time of day.
- The alarm time is dependent on current locale.

Apps with clock-based alarms might not work well across locales, because they might fire at the wrong times. And if the user changes the device's time setting, it could cause unexpected behavior in your app.

The `AlarmManager` class provides two types of RTC alarm:

- `RTC`: Fires the pending intent at the specified time but doesn't wake the device. All repeating alarms fire when your device is next awake.
- `RTC_WAKEUP`: Fires the pending intent at the specified time. If the screen is off, this alarm wakes the device's CPU.

Alarm best practices

Alarms affect how your app uses (or abuses) system resources. For example, imagine a popular app that syncs with a server, and imagine that the sync operation is based on clock time. If every instance of the app connects to the server at the same time, the load on the server can delay responses or even create a "denial of service" condition.

To avoid this problem, add randomness (jitter) to network requests that trigger as a result of a repeating alarm. Here's one way to add randomness:

- Schedule an exact alarm that performs any local work. "Local work" is any task that doesn't contact a server over a network or require data from that server.
- Schedule a separate alarm that contains the network requests, and have this alarm fire after a random period of time. The component that receives the `PendingIntent` from the first alarm usually sets this second alarm. (You can also set this alarm at the same time as you set the first alarm.)

Other best practices:

- Keep your alarm frequency to a minimum.
- Don't wake the device unnecessarily.
- Use the least precise timing possible to allow the `AlarmManager` to be the most efficient it can be. For example, when you schedule a repeating alarm, use `setInexactRepeating()` instead of `setRepeating()`. For details, see Scheduling a repeating alarm, below.
- Avoid basing your alarm on clock time and use `ELAPSED_REALTIME` for repeating alarms whenever possible. Repeating alarms that are based on a precise trigger time don't scale well.

Scheduling an alarm

The `AlarmManager` class gives you access to the Android system alarm services. `AlarmManager` lets you broadcast an `Intent` at a scheduled time, or after a specific interval.
To schedule an alarm:

1. Call `getSystemService(ALARM_SERVICE)` to get an instance of the `AlarmManager` class.
2. Use one of the `set...()` methods available in `AlarmManager` (as described below). Which method you use depends on whether the alarm is elapsed real time, or RTC.

All the `AlarmManager.set...()` methods include a `type` argument and a `PendingIntent` argument:

- Use the `type` argument to specify the alarm type. Alarms can be elapsed real-time alarms (`ELAPSED_REALTIME` or `ELAPSED_REALTIME_WAKEUP`) or real-time clock alarms (`RTC` or `RTC_WAKEUP`).
- A `PendingIntent` object, which is how you specify which task to perform at the given time.

Scheduling a single-use alarm

To schedule a single alarm, use one of the following methods on the `AlarmManager` instance:

- `set()`: For devices running API 19+, this method schedules a single, inexactly timed alarm, meaning that the system shifts the alarm to minimize wakeups and battery use. For devices running lower API versions, this method schedules an exactly timed alarm.
- `setWindow()`: For devices running API 19+, use this method to set a window of time during which the alarm should be triggered.
- `setExact()`: For devices running API 19+, this method triggers the alarm at an exact time. Use this method only for alarms that must be delivered at an exact time, for example an alarm clock that rings at a requested time. Exact alarms reduce the OS's ability to minimize battery use, so don't use them unnecessarily.

Here's an example of using `set()` to schedule a single-use alarm:

```
alarmMgr.set(AlarmManager.ELAPSED_REALTIME,
             SystemClock.elapsedRealtime() + 1000*300,
             alarmIntent);
```

In this example:

- The `type` is `ELAPSED_REALTIME`, so this is an elapsed real-time alarm. If the device is idle when the alarm is sent, the alarm does not wake the device.
- The alarm is sent 5 minutes (300,000 milliseconds) after the method returns.
- `alarmIntent` is a `PendingIntent` broadcast that contains the action to perform when the alarm is sent.

Note: For timing operations like ticks and timeouts, and events that happen more often than once a minute, it's easier and more efficient to use a `Handler` rather than an alarm.

Doze and App Standby

API 23+ devices sometimes enter Doze or App Standby mode to save power:

- *Doze* mode is triggered when a user leaves a device unplugged and stationary for a period, with the screen off. During short "maintenance windows," the system exits Doze to let apps complete deferred activities, including firing standard alarms, then returns to Doze. Doze mode ends when the user returns to their device.
- *App Standby* mode is triggered on idle apps that haven't been used recently. App Standby mode ends when the user returns to your app or plugs in the device.

Some alarms can wait for a maintenance window, or until the device comes out of Doze or App Standby mode. For these alarms, use the standard `set()` and `setExact()` methods to optimize battery life.
If you need an alarm that fires during Doze mode or App Standby mode without waiting for a maintenance window:

- For inexact alarms, use `setAndAllowWhileIdle()` and for exact alarms, use `setExactAndAllowWhileIdle()`, or
- Set a user-visible alarm (API 21 and higher).

Scheduling a repeating alarm

You can also use the `AlarmManager` to schedule repeating alarms, using one of the following methods:

- `setRepeating()`: On devices running Android versions lower than 4.4 (API Level 19), this method creates a repeating, exactly timed alarm. On devices running API 19 and higher, `setRepeating()` behaves exactly like `setInexactRepeating()`.
- `setInexactRepeating()`: This method creates a repeating, inexact alarm that allows for batching. When you use `setInexactRepeating()`, Android synchronizes repeating alarms from multiple apps and fires them at the same time. This reduces the total number of times the system must wake the device, thus reducing drain on the battery. As of API 19, all repeating alarms are inexact.

To decrease possible battery drain:

- Schedule repeating alarms to be as infrequent as possible.
- Use inexact timing, which allows the system to batch alarms from different apps together.

Note: while `setInexactRepeating()` is an improvement over `setRepeating()`, it can still overwhelm a server if every instance of an app reaches the server around the same time. Therefore, for network requests, add some randomness to your alarms, as described in Alarm best practices. If you really need exact repeating alarms on API 19+, set a single-use alarm with `setExact()` and set the next alarm once that alarm has triggered. This second alarm is set by whatever component receives the `PendingIntent`—usually a service or a broadcast receiver.

Here's an example of using `setInexactRepeating()` to schedule a repeating alarm:

```
alarmMgr.setInexactRepeating(AlarmManager.RTC_WAKEUP,
                calendar.getTimeInMillis(),
                AlarmManager.INTERVAL_FIFTEEN_MINUTES,
                alarmIntent);
```

In this example:

- The `type` is `RTC_WAKEUP`, which means that this alarm is clock-based, and it wakes the device.
- The first occurrence of the alarm is sent immediately, because `calendar.getTimeInMillis()` returns the current time as UTC milliseconds.
- After the first occurrence, the alarm is sent approximately every 15 minutes.

If the method were `setRepeating()` instead of `setInexactRepeating()`, and if the device were running an API version lower than 19, the alarm would be sent *exactly* every 15 minutes.

Possible values for this argument are `INTERVAL_DAY`, `INTERVAL_FIFTEEN_MINUTES`, `INTERVAL_HALF_DAY`, `INTERVAL_HALF_HOUR`, `INTERVAL_HOUR`.

- `alarmIntent` is the `PendingIntent` that contains the action to perform when the alarm is sent. This `Intent` typically comes from `IntentSender.getBroadcast()`.

Checking for an existing alarm

It's often useful to check whether an alarm is already set. For example, if an alarm exists, you may want to disable the ability to set another alarm.

To check for an existing alarm:

1. Create a `PendingIntent` that contains the same `Intent` used to set the alarm, but this time, use the `FLAG_NO_CREATE` flag.
   With `FLAG_NO_CREATE`, a `PendingIntent` is only created if one with the same `Intent` exists. Otherwise, the request returns `null`.
2. Check whether the `PendingIntent` is `null`:
   If the `PendingIntent` is `null`, the alarm has not yet been set. If the `PendingIntent` is not `null`, the `PendingIntent` exists, meaning that the alarm has been set.
   For example, the following code returns `true` if the alarm contained in `alarmIntent` already exists:

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

Canceling an alarm

To cancel an alarm, use `cancel()` and pass in the `PendingIntent`. For example:

```
alarmManager.cancel(alarmIntent);
```

User-visible alarms ("alarm clocks")

For API 21+ devices, you can set a user-visible alarm clock by calling `setAlarmClock()`. Apps can retrieve the next user-visible alarm clock that's set to go off by calling `getNextAlarmClock()`.
Alarm clocks set with `setAlarmClock()` work even when the device or app is idle, similar to the way `setExactAndAllowWhileIdle()` works. Using `setAlarmClock()` gets you as close to an exact-time wake-up call as possible.

Related practical

The related practical is in 8.2: The alarm manager.

Learn more

Android developer documentation:

- Schedule repeating alarms
- `AlarmManager` reference
- Choosing an alarm

Video:

- Alarms and Syncing and Tasks, Oh My! (Big Android BBQ 2015)

8.3 Efficient data transfer

Contents

Transferring data is an essential part of most Android apps, but it can affect battery life and increase data usage. Using the wireless radio to transfer data is potentially one of your app's most significant sources of battery drain.

Users care about battery drain because they would rather use their mobile device without it connected to the charger. And users care about data usage, because every bit of data transferred can cost them money.

In this chapter, you learn how your app's networking activity affects the device's radio hardware so you can minimize the battery drain associated with network activity. You also learn how to wait for the proper conditions to accomplish resource-intensive tasks.

Wireless radio state

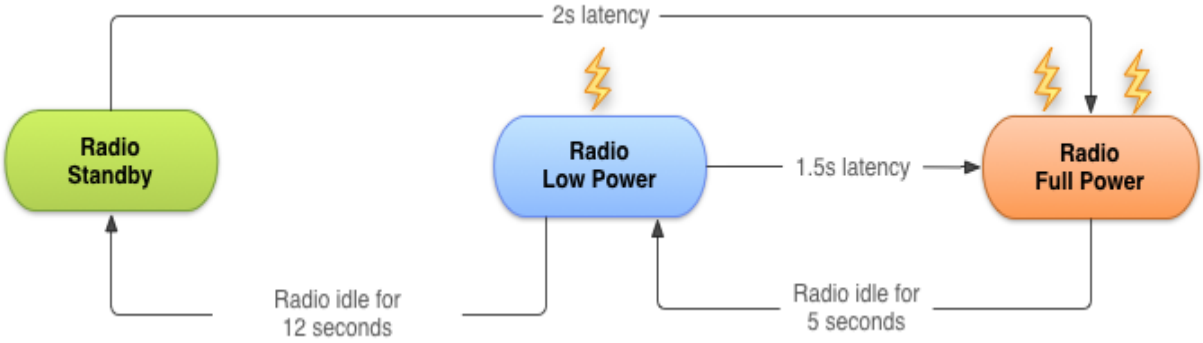
A fully active wireless radio consumes significant power. To conserve power when not in use, the radio transitions between different energy states. However, there is a trade-off between conserving power and the time it takes to power up when needed.

For a typical 3G network the radio has these three energy states:

- **Full power:** Used when a connection is active. Allows the device to transfer data at its highest possible rate.
- **Low power:** An intermediate state that uses about 50% less battery.
- **Standby:** The minimal energy state, during which no network connection is active or required.

While the low and standby states use much less battery, they also introduce latency to network requests. Returning to full power from the low state takes around 1.5 seconds, while moving from standby to full can take over 2 seconds.

Android uses a state machine to determine how to transition between states. To minimize latency, the state machine waits a short time before it transitions to lower energy states.



The radio state machine on each device, particularly the transition delay ("tail time") and startup latency, vary based on the wireless radio technology employed, for example 2G, 3G, or LTE. The state machine is defined and configured by the carrier network over which the device is operating.

This chapter describes a representative state machine for a typical 3G wireless radio. However, the general principles and resulting best practices are applicable for all wireless radio implementations.

As with any best practices, there are trade-offs that you need to consider for your own app development.

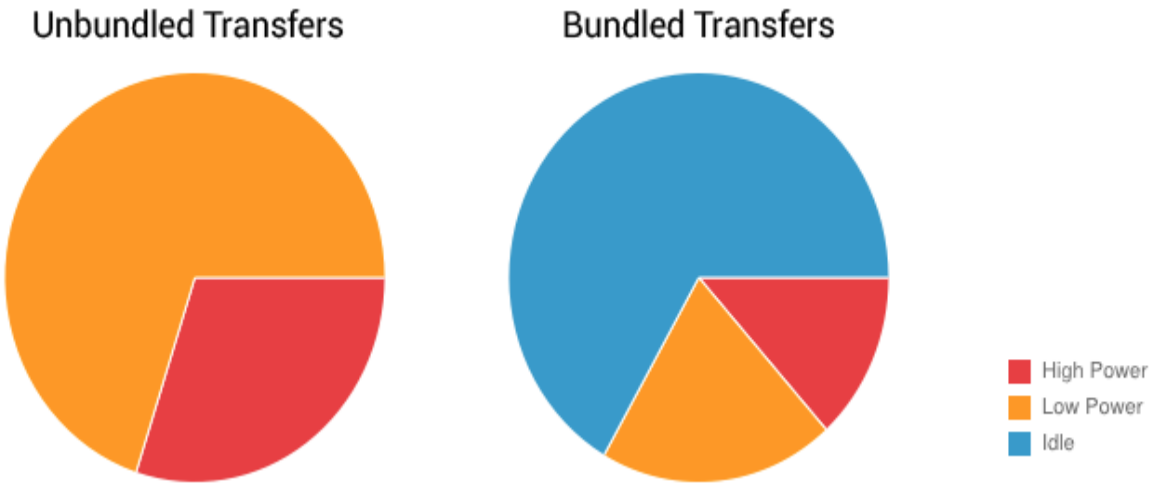
Bundling network transfers

Every time you create a new network connection, the radio transitions to the full power state. In the case of the 3G radio state machine described above, the radio remains at full power for the duration of your transfer. Then there are 5 seconds of tail time, followed by 12 seconds at the low energy state. Then the radio turns off. For a typical 3G device, every data transfer session causes the radio to draw power for almost 20 seconds.

What this means in practice:

- An app that transfers unbundled data for 1 second every 18 seconds keeps the wireless radio always active.
- An app that transfers bundled data for 3 seconds every minute keeps the radio in the high power state for only 8 seconds. Then the radio is in the low power state for an additional 12 seconds.

The second example allows the radio to be idle for 40 seconds out of every minute, resulting in a massive reduction in battery consumption.



It's important to bundle and queue up your data transfers. You can bundle transfers that are due to occur within a time window and make them all happen simultaneously, ensuring that the radio draws power for as little time as possible.

Prefetching

To *prefetch* data means that your app takes a guess at what content or data the user will want next, and fetches the data ahead of time. For example, when the user looks at the first part of an article, a good guess is to prefetch the next part. If a user is watching a video, fetching the next minutes of the video is also a good guess.

Prefetching allows you to 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. This reduces the number of radio activations required to download the data. As a result, you conserve battery life, improve latency for the user, lower the required bandwidth, and reduce download times.

Prefetching has trade-offs. If you download too much data or the wrong data, you might increase the drain on the battery. If you download at the wrong time, users might end up waiting.

Optimizing prefetching data is an advanced topic not covered in this course, but the following guidelines cover common situations:

- How aggressively you prefetch data depends on the size of the data being downloaded and the likelihood of the data being used. Here's a rough guide, based on the state machine described above: for data that has a 50% chance of being used within the current user session, you can typically prefetch for around 6 seconds (approximately 1-2 MB). After this point, the potential cost of downloading unused data matches the potential savings of not downloading that data to begin with.
- It's good practice to prefetch data in such a way that you only need to initiate downloads of 1-5 MB every 2-5 minutes. For example, for a large video file, you would download a chunk of the data every 2-5 minutes, effectively prefetching only the video data likely to be viewed in the next few minutes.

Prefetching example

Many news apps attempt to reduce bandwidth by downloading headlines only after a category has been selected, full articles only when the user wants to read them, and thumbnails just as they scroll into view.

Using this approach, the radio is forced to remain active for the majority of a news-reading session as users scroll headlines, change categories, and read articles. Not only that, but the constant switching between energy states results in significant latency when switching categories or reading articles.

Here's a better approach:

1. Prefetch a reasonable amount of data at startup, beginning with the first set of news headlines and thumbnails. This ensures a quick startup time.
2. Continue with the remaining headlines, the remaining thumbnails, and the article text for each article from the first set of headlines.

Monitor connectivity state

Devices can network using different types of hardware:

- *Wireless radios* use varying amounts of battery depending on technology, and higher bandwidth consumes more energy. Higher bandwidth means you can prefetch more aggressively, downloading more data during the same amount of time. However, because the tail-time battery cost is relatively higher, it's also more efficient to keep the radio active for longer periods during each transfer session to reduce the frequency of updates.
- *Wi-Fi radio* uses significantly less battery than wireless and offers greater bandwidth.

Note: Perform data transfers when connected over Wi-Fi whenever possible.

You can use the `ConnectivityManager` to determine the active wireless radio and modify your prefetching routines depending on network type:

```
ConnectivityManager cm =
   (ConnectivityManager)getSystemService(Context.CONNECTIVITY_SERVICE);
TelephonyManager tm =
    (TelephonyManager)getSystemService(Context.TELEPHONY_SERVICE);

NetworkInfo activeNetwork = cm.getActiveNetworkInfo();
int PrefetchCacheSize = DEFAULT_PREFETCH_CACHE;

switch (activeNetwork.getType()) {
    case (ConnectivityManager.TYPE_WIFI):
        PrefetchCacheSize = MAX_PREFETCH_CACHE;
        break;
    case (ConnectivityManager.TYPE_MOBILE): {
        switch (tm.getNetworkType()) {
           case (TelephonyManager.NETWORK_TYPE_LTE |
                  TelephonyManager.NETWORK_TYPE_HSPAP):
                PrefetchCacheSize *= 4;
                break;
            case (TelephonyManager.NETWORK_TYPE_EDGE |
                  TelephonyManager.NETWORK_TYPE_GPRS):
               PrefetchCacheSize /= 2;
               break;
            default: break;
        }
        break;
      }
  default: break;
  }
```

The system sends out broadcast intents when the connectivity state changes, so you can listen for these changes using a `BroadcastReceiver`.

Monitor battery state

To minimize battery drain, monitor the state of your battery and wait for specific conditions before initiating a battery-intensive operation.

The `BatteryManager` broadcasts all battery and charging details in a broadcast `Intent` that includes the charging status.
To check the current battery status, examine the broadcast intent:

```
IntentFilter ifilter = new IntentFilter(Intent.ACTION_BATTERY_CHANGED);
Intent batteryStatus = context.registerReceiver(null, ifilter);
// Are we charging / charged?
int status = batteryStatus.getIntExtra(BatteryManager.EXTRA_STATUS, -1);
boolean isCharging = status == BatteryManager.BATTERY_STATUS_CHARGING ||
                     status == BatteryManager.BATTERY_STATUS_FULL;

// How are we charging?
int chargePlug = batteryStatus.getIntExtra(BatteryManager.EXTRA_PLUGGED, -1);
boolean usbCharge = chargePlug == BatteryManager.BATTERY_PLUGGED_USB;
boolean acCharge = chargePlug == BatteryManager.BATTERY_PLUGGED_AC;
```

If you want to react to changes in the battery charging state, use a `BroadcastReceiver` to register for the battery status actions in your code:

```
public void registerBatteryChargingStateReceiver()
 {

     // Create Receiver Object
     BroadcastReceiver receiver = new MyPowerReceiver();

     //Create Intent Filter
     IntentFilter intentFilter = new IntentFilter();

     intentFilter.addAction(Intent.ACTION_POWER_CONNECTED);
     intentFilter.addAction(Intent.ACTION_POWER_DISCONNECTED);

     // Register broadcast receiver
     this.registerReceiver(receiver, intentFilter);

 }
```

Broadcast intents are also delivered when the battery level changes in a significant way:

Broadcast action	Constant value
`Intent.ACTION_BATTERY_LOW`	`"android.intent.action.BATTERY_LOW"`
`Intent.ACTION_BATTERY_OKAY`	`"android.intent.action.BATTERY_OKAY"`

https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/

https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/

JobScheduler

Constantly monitoring the connectivity and battery status of the device can be a challenge. It requires using components such as broadcast receivers, which can consume system resources even when your app isn't running. Because transferring data efficiently is such a common task, the Android SDK provides a class that makes efficient data transfer much easier: `JobScheduler`.
Introduced in API level 21, `JobScheduler` allows you to schedule a task around specific conditions, rather than a specific time as with `AlarmManager`.
`JobScheduler` has three components:

- `JobInfo` uses the builder pattern to set the conditions for the task.
- `JobService` is a wrapper around the `Service` class where the task is actually completed.
- `JobScheduler` schedules and cancels tasks.

Note: `JobScheduler` is only available on devices running API 21+, and is not available in the support library. If your app targets devices with earlier API levels, look into the backwards compatible `WorkManager`, a new API currently in alpha, that allows you to schedule background tasks that need guaranteed completion (regardless of whether the app process is around or not). `WorkManager` provides JobScheduler-like capabilities to API 14+ devices, even those without Google Play Services.

JobInfo

Set the job conditions by constructing a `JobInfo` object using the `JobInfo.Builder` class. The `JobInfo.Builder` class is instantiated from a constructor that takes two arguments: a job ID (which can be used to cancel the job), and the `ComponentName` of the `JobService` that contains the task. Your `JobInfo.Builder` must set at least one, non-default condition for the job. For example:

```
JobScheduler scheduler = (JobScheduler) getSystemService(JOB_SCHEDULER_SERVICE);
ComponentName serviceName = new ComponentName(getPackageName(),
NotificationJobService.class.getName());
JobInfo.Builder builder = new JobInfo.Builder(JOB_ID, serviceName);
builder.setRequiredNetworkType(NETWORK_TYPE_UNMETERED);
JobInfo jobInfo = builder.build();
```

Note: See the related practical for a complete example.

The `JobInfo.Builder` class has many `set()` methods that allow you to determine the conditions of the task. Below is a list of few available constraints with their respective `set()` methods and class constants:

- **Minimum Latency:** The minimum amount of time to wait before completing the task. Set this condition using the `setMinimumLatency()` method, which takes a single argument: the amount of time to wait in milliseconds.
- **Override Deadline:** The maximum time to wait before running the task, even if other conditions aren't met. Set this condition using the `setOverrideDeadline()` method, which is the maximum time to wait in milliseconds.
- **Periodic:** Repeats the task after a certain amount of time. Set this condition using the `setPeriodic()` method, passing in the repetition interval. This condition is mutually e
- xclusive with the minimum-latency and override-deadline conditions: setting `setPeriodic()` with one of them results in an error.
- **Required Network Type:** The kind of network type your job needs. If the network isn't necessary, you don't need to call this function, because the default is `NETWORK_TYPE_NONE`. Set this condition using the `setRequiredNetworkType()` method, passing in one of the following constants: `NETWORK_TYPE_NONE`, `NETWORK_TYPE_ANY`, `NETWORK_TYPE_NOT_ROAMING`, `NETWORK_TYPE_UNMETERED`.
- **Required Charging State:** Whether the device needs to be plugged in to run this job. Set this condition using the `setRequiresCharging()` method, passing in a boolean. The default is `false`.
- **Requires Device Idle:** Whether the device needs to be in idle mode to run this job. "Idle mode" means that the device isn't in use and hasn't been for some time, as loosely defined by the system. When the device is in idle mode, it's a good time to perform resource-heavy jobs. Set this condition using the `setRequiresDeviceIdle()`method, passing in a `boolean`. The default is `false`.

JobService

Once the conditions for a task are met, the framework launches a subclass of `JobService`, which is where you implement the task itself. The `JobService` runs on the UI thread, so you need to offload blocking operations to a worker thread. Declare the `JobService` subclass in the Android Manifest, and include the `BIND_JOB_SERVICE` permission:

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

In your subclass of `JobService`, override two methods, `onStartJob()` and `onStopJob()`.

onStartJob()

The system calls `onStartJob()` and automatically passes in a `JobParameters` object, which the system creates with information about your job. If your task contains long-running operations, offload the work onto a separate thread. The `onStartJob()` method returns a `boolean`: `true` if your task has been offloaded to a separate thread (meaning it might not be completed yet) and `false` if there is no more work to be done.
Use the `jobFinished()` method from any thread to tell the system that your task is complete. This method takes two parameters: the `JobParameters` object that contains information about the task, and a boolean that indicates whether the task needs to be rescheduled, according to the defined backoff policy.

onStopJob()

If the system determines that your app must stop execution of the job, even before `jobFinished()` is called, the system calls `onStopJob()`. This happens if the requirements that you specified when you scheduled the job are no longer met. Examples:

- If your app requests Wi-Fi with `setRequiredNetworkType()` but the user turns Wi-Fi while your job is executing, the system calls `onStopJob()`.
- If your app specifies `setRequiresDeviceIdle()` but the user starts interacting with the device while your job is executing, the system calls `onStopJob()`.

You're responsible for how your app behaves when it receives `onStopJob()`, so don't ignore it. This method returns a `boolean`, indicating whether you'd like to reschedule the job based on the defined backoff policy, or drop the task.

JobScheduler

The final part of scheduling a task is to use the `JobScheduler` class to schedule the job. To obtain an instance of this class, call `getSystemService(JOB_SCHEDULER_SERVICE)`. Then schedule a job using the `schedule()` method, passing in the `JobInfo` object you created with the `JobInfo.Builder`. For example:

```
mScheduler.schedule(myJobInfo);
```

The framework is intelligent about when you receive callbacks, and it attempts to batch and defer them as much as possible. If you don't specify a deadline on your job, the system can run the job at any time, depending on the current state of the `JobScheduler` object's internal queue. However, the job might be deferred as long as until the next time the device is connected to a power source.
To cancel a job, call `cancel()`, passing in the job ID from the `JobInfo.Builder` object, or call `cancelAll()`. For example:

```
mScheduler.cancelAll();
```

Related practical

The related practical is 8.3: JobScheduler.

Learn more

Android developer documentation:

- Transferring data without draining the battery
- Optimize downloads for efficient network access
- Modify your download patterns based on the connectivity type
- `JobScheduler` reference
- `JobService` reference
- `JobInfo` reference
- `JobInfo.Builder` reference
- `JobParameters` reference

Video:

- Alarms and Syncing and Tasks, Oh My! (Big Android BBQ 2015)