

8.1: Notifications

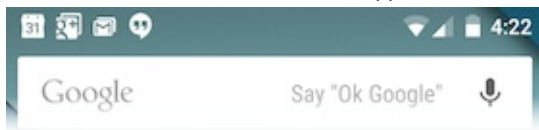
Contents:

- [Introduction](#)
- [What is a notification?](#)
- [Creating notifications](#)
- [Delivering notifications](#)
- [Reusing notifications](#)
- [Clearing notifications](#)
- [Notification compatibility](#)
- [Notification design guidelines](#)
- [Related practical](#)
- [Learn more](#)

In this chapter you learn how to create, deliver, and reuse notifications, and how to make them compatible for different Android versions.

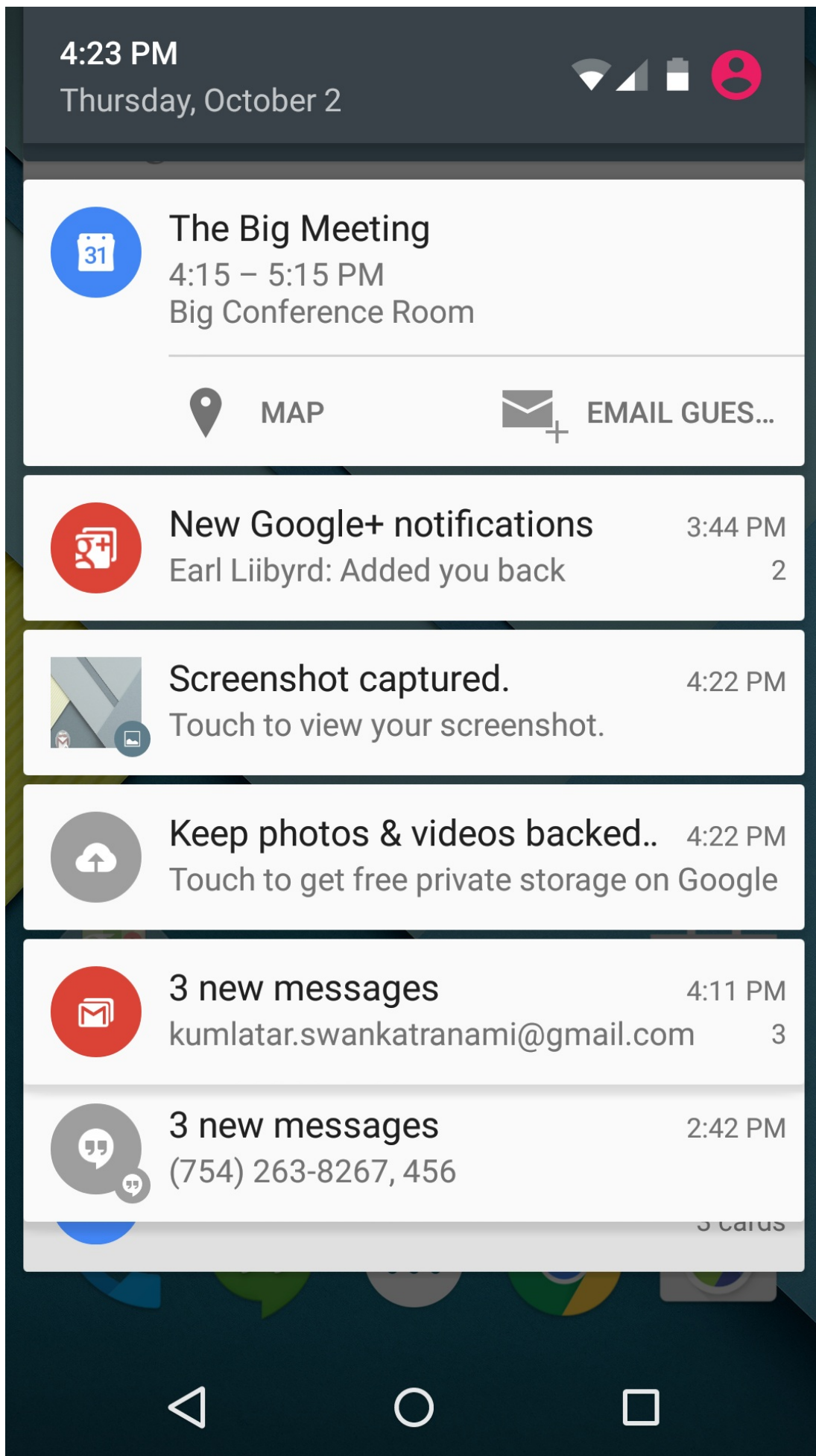
What is a notification?

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



bar.

To see the details of the notification, the user opens the *notification drawer*, or views the notification on the lock screen if the device is locked. The notification area, the lock screen, and the notification drawer are system-controlled areas that the user can view at any time.



The screenshot shows an "open" notification drawer. The the status bar isn't visible, because the notification drawer is open.

This process is described below.

Creating notifications

You create a notification using the `NotificationCompat.Builder` class. (Use `NotificationCompat` for the best backward compatibility. For more information, see [Notification compatibility](#).) The builder classes simplify the creation of complex objects.

To create a `NotificationCompat.Builder`, pass the application context to the constructor:

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

Setting notification components

When using `NotificationCompat.Builder`, you must assign a small icon, text for a title, and the notification message. You should keep the notification message shorter than 40 characters and not repeat what's in the title. For example:

```
NotificationCompat.Builder mBuilder =
    new NotificationCompat.Builder(this)
        .setSmallIcon(R.drawable.notification_icon)
        .setContentTitle("Dinner is ready!")
        .setContentText("Lentil soup, rice pilaf, and cake for dessert.");
```

You also need to set an `Intent` that determines what happens when the user clicks the notification. Usually this `Intent` results in your app launching an `Activity`.

To make sure the system delivers the `Intent` even when your app isn't running when the user clicks the notification, wrap the `Intent` in a `PendingIntent` object, which allows the system to deliver the `Intent` regardless of the app state.

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 clicks on the notification, use `PendingIntent.getActivity()`, passing 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 take 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 application.

For example:

```
Intent contentIntent = new Intent(this, ExampleActivity.class);
PendingIntent pendingContentIntent = PendingIntent.getActivity(this, 0,
    contentIntent, PendingIntent.FLAG_UPDATE_CURRENT);
mBuilder.setContentIntent(pendingContentIntent);
```

To learn more about `PendingIntent`, see the [PendingIntent documentation](#).

Optional components

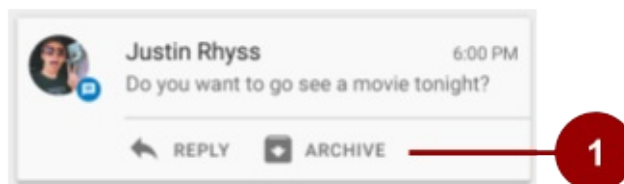
You can use various options with notifications, including:

- Notification actions
- Priorities
- Expanded layouts
- Ongoing notifications

For other options you can use with notifications, see the [NotificationCompat.Builder](#) reference.

Notification actions

A *notification action* is an action that the user can take on the notification. The action is made available via an action button on the notification. Like the `Intent` that determines what happens when the user clicks the notification, a notification action uses a `PendingIntent` to complete the action. The Android system usually displays a notification action as a button adjacent to the notification content. Starting with Android 4.1 (API level 16), notifications support icons embedded below the



content text, as shown in the screenshot below.

1. This notification has two actions that the user can take, "Reply," or "Archive." Each has an icon.

To add a notification action, use the `addAction()` method with the `NotificationCompat.Builder` object. Pass in the icon, the title string and the `PendingIntent` to trigger when the user taps the action. For example:

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

To ensure that an action button's functionality is always available, follow the instructions in the [Notification compatibility](#) section, below.

Notification priority

Android allows you to assign a priority level to each notification to influence how the Android system will deliver it. Notifications have a priority between `MIN` (`-2`) and `MAX` (`2`) that corresponds to their importance. The following table shows the available priority constants defined in the `Notification` class.

Priority Constant	Use
PRIORITY_MAX	For critical and urgent notifications that alert the user to a condition that is time-critical or needs to be resolved before they can continue with a time-critical task.
PRIORITY_HIGH	Primarily for important communication, such as messages or chats.
PRIORITY_DEFAULT	For all notifications that don't fall into any of the other priorities described here.
PRIORITY_LOW	For information and events that are valuable or contextually relevant, but aren't urgent or time-critical.
PRIORITY_MIN	For nice-to-know background information. For example, weather or nearby places of interest.

To change the priority of a notification, use the `setPriority()` method on the `NotificationCompat.Builder` object, passing in one of the above constants.

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

Notifications can be intrusive. Using notification priority correctly is the first step in making sure that your users don't uninstall your app because it's too distracting.

Peeking

Notifications with a priority of `HIGH` or `MAX` can *peek*, which means they slide briefly into view on the user's current screen, no matter what apps the user is using. Note that on devices running Android 6.0 and higher, users can block peeking by changing the device's "App notification" settings. This means you can't rely on notifications peeking, even if you set them up that way.

To create a notification that can peek:

1. Set the priority to `HIGH` or `MAX`.
2. Set a sound or light pattern using the `setDefaults()` method on the builder, passing the `DEFAULTS_ALL` constant. This gives the notification a default sound, light pattern, and vibration.

```
NotificationCompat.Builder mBuilder =
    new NotificationCompat.Builder(this)
        .setSmallIcon(R.drawable.notification_icon)
        .setContentTitle("My notification")
        .setContentText("Hello World!")
        .setPriority(PRIORITY_HIGH)
        .setDefaults(DEFAULTS_ALL);
```

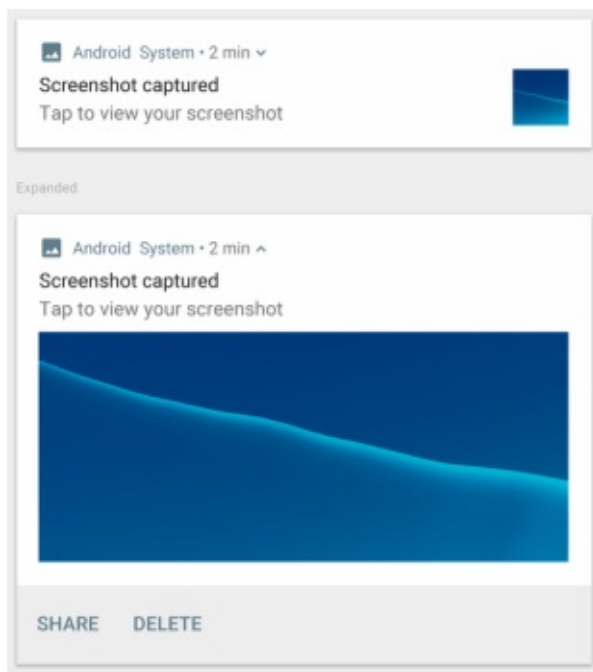
Expanded view layouts

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:

- Use `NotificationCompat.BigTextStyle` for large-format notifications that include a lot of text.
- Use `NotificationCompat.InboxStyle` for large-format notifications that include a list of up to five strings.
- Use `Notification.MediaStyle` for media playback notifications. There is currently no `NotificationCompat` version of this style, so it can only be used on devices with Android 4.1 or above. See the [Notification compatibility](#) section for more information.

- Use `NotificationCompat.BigPictureStyle`, shown in the screenshot below, for large-format notifications that include a



large image attachment.

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

```
NotificationCompat notif = new NotificationCompat.Builder(mContext)
    .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 can't be dismissed by the user. Your app must explicitly cancel them by calling `cancel()` or `cancelAll()`. Creating multiple ongoing notifications is a nuisance to your users since they are unable to cancel the notification. Use ongoing notifications sparingly.

To make a notification ongoing, set `setOngoing()` to `true`. Use ongoing notifications to indicate background tasks that the user actively engages with (such as playing music) or tasks that occupy the device (such as file downloads, sync operations, and active network connections).

Delivering notifications

Use the `NotificationManager` class to deliver notifications:

1. Call `getSystemService()`, passing in the `NOTIFICATION_SERVICE` constant, to create an instance of `NotificationManager`.
2. Call `notify()` to deliver the notification. In the `notify()` method, pass in these two values:
 - 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)
    .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());
```

Reusing notifications

When you need to issue a notification multiple times for the same type of event, you can update a previous notification by changing some of its values, adding to it, 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.

Clearing notifications

Notifications remain visible until one of the following happens:

- If the notification can be cleared, it disappears when the user dismisses it individually or by using "Clear All."
- If you called `setAutoCancel()` when you created the notification, the notification disappears when the user clicks it.
- If you call `cancel()` for a specific notification ID, the notification disappears.
- If you call `cancelAll()`, all the notifications you've issued disappear.

Because [ongoing notifications](#) can't be dismissed by the user, your app must cancel them by calling `cancel()` or `cancelAll()`.

Notification compatibility

To ensure the best compatibility, create notifications with `NotificationCompat` and its subclasses, particularly `NotificationCompat.Builder`.

Keep in mind that not all notification features are available for every Android version, even though the methods to set them are in the support library class `NotificationCompat.Builder`. For example, expanded view layouts for notifications are only available on Android 4.1 and higher, but action buttons depend on expanded view layouts. This means that if you use notification action buttons, they don't show up on devices running anything before Android 4.1.

To solve this:

- Don't rely on notification action buttons to carry out a notification's action; instead make the action available in an Activity. You may want to add a new Activity to do this.

For example, if you set a notification action that provides a control to stop and start media playback, first implement this control in an Activity in your app.

- Have the Activity start when users click the notification. To do this:
 1. Create a `PendingIntent` for the Activity.
 2. Call `setContentIntent()` to add the `PendingIntent` to 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 click the notification.

Notification design guidelines

Notifications always interrupt the user. As such they must be short, timely, and most of all, relevant.

- **Relevant:** Ask yourself whether this information is essential for the user. What happens if they don't get the notification? For example, scheduled calendar events are likely 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. Now, challenge yourself to say it with fewer.

Give users the power to choose:

- Provide settings in your app that allow users to choose the kinds of notifications they want to receive, and how they want to receive them.

In addition to these basic principles, notifications have their own design guidelines:

- To learn how to design notifications and their interactions, see the [Material Design notification patterns](#) documentation.
- To learn how to design notifications and their interactions for older Android versions, see [Notifications, Android 4.4 and Lower](#).
- For important details about Material Design changes introduced in Android 5.0 (API level 21), see the [Material Design training](#).

Related practical

The related exercises and practical documentation is in [Android Developer Fundamentals: Practicals](#).

- [Notifications](#)

Learn more

Guides

- [Notifications](#)
- [Notification design guide](#)

Reference

- [NotificationCompat.Builder reference](#)
- [NotificationCompat.Style reference](#)

8.2: Scheduling Alarms

Contents:

- [Introduction](#)
- [Alarm types](#)
- [Alarm best practices](#)
- [Scheduling an alarm](#)
- [Checking for an existing alarm](#)
- [Canceling an alarm](#)
- [User-visible alarms \("alarm clocks"\)](#)
- [Related practical](#)
- [Learn more](#)

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'll learn how to use alarms to schedule tasks for specific times, whether or not your app is running at the time the alarm is set to 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 intents at set times or intervals. You can use alarms with broadcast receivers to start services and perform other operations.
- Alarms operate outside your app, so 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 an alarm:

- For timing events such as ticks and timeouts, and for timed operations that are guaranteed to happen during the lifetime of your app, use the `Handler` class with `Timer` and `Thread`. This approach gives Android better control over system resources than if you used alarms.
- For server sync operations, use `SyncAdapter` with the [Google Cloud Messaging Service](#).
- For tasks that can wait until conditions are favorable, such as when the device is connected to WiFi and is charging (for example, updating weather information or news stories), you might not want to use alarms. For these tasks on API 21+ devices, consider using `JobScheduler`, which you will learn about in an upcoming lesson.

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. Elapsed real-time alarms aren't affected by time zones, so they 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, waking the device's CPU if the screen is off. Use this alarm instead of `ELAPSED_REALTIME` if your app has a time dependency, for example if it has a limited window during which to perform an operation.

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 up the device. All repeating alarms fire when your device is next awake.
- `RTC_WAKEUP` : Fires the pending intent at the specified time, waking the device's CPU if the screen is off.

Alarm best practices

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

To avoid this problem and others, follow these best practices:

- Add randomness (jitter) to network requests that trigger as a result of a repeating alarm. Here's one way to do this:
 - Schedule an exact alarm that performs any local work. "Local work" means anything 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. Usually this second alarm is set by whatever component receives the `PendingIntent` from the first alarm. (You can also set this alarm at the same time as you set the first alarm.)
- Keep your alarm frequency to a minimum.
- Don't wake up 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 these two arguments:

- A `type` argument, which is how you specify the [alarm type](#):

- `ELAPSED_REALTIME` or `ELAPSED_REALTIME_WAKEUP`, described in [Elapsed real-time alarms](#) above.
- `RTC` or `RTC_WAKEUP`, described in [Real-time clock \(RTC\) alarms](#) above.
- 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`, which means that 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 of time, 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.

To use alarms with Doze and App Standby:

- If you need an alarm that fires while a device is in Doze or App Standby mode without waiting for a maintenance window, use `setAndAllowWhileIdle()` for inexact and `setExactAndAllowWhileIdle()` for exact alarms, or set a [user-visible alarm](#) (API 21+).
- 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.

Scheduling a repeating alarm

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

- `setRepeating()`: Prior to Android 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 hits 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 either 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 is a [clock-based alarm](#) that wakes the device when the alarm is sent.
- 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, you may want to disable the ability to set another alarm if one already exists.

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` already exists. Otherwise, the request returns `null` .

2. Check whether the `PendingIntent` is `null` :
 - If it's `null` , the alarm has not yet been set.
 - If it's not `null` , the `PendingIntent` already 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()`.

Alarms clocks set with `setAlarmClock()` work even when the device or app is idle (similar to `setExactAndAllowWhileIdle()`), which gets you as close to an exact wake up call as possible.

Related practical

The related exercises and practical documentation is in [Android Developer Fundamentals: Practicals](#).

- [Alarm Manager](#)

Learn more

- [Schedule Repeating Alarms Guide](#)
- [AlarmManager reference](#)
- [Choosing an Alarm Blog Post](#)
- [Scheduling Alarms Presentation](#)

8.3: Transferring Data Efficiently

Contents:

- [Wireless radio state](#)
- [Bundling network transfers](#)
- [Prefetching](#)
- [Monitor connectivity state](#)
- [Monitor battery state](#)
- [JobScheduler](#)
- [Related practical](#)
- [Learn more](#)

Transferring data is an essential part of most Android applications, but it can negatively affect battery life and increase data usage costs. 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

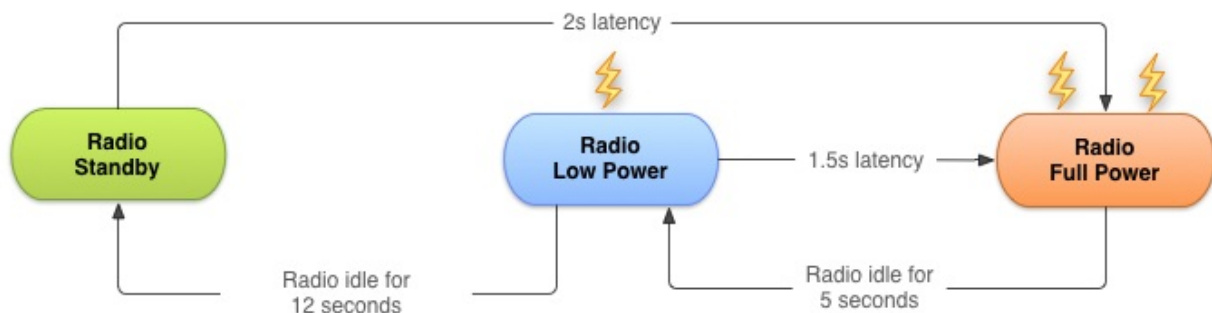
A fully active wireless radio consumes significant power. To conserve power when not in use, the radio transitions between different energy states. However, there is a trade-off between conserving power and the time it takes to power up when needed.

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

1. **Full power:** Used when a connection is active. Allows the device to transfer data at its highest possible rate.
2. **Low power:** An intermediate state that uses about 50% less battery.
3. **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 the lower energy states.



The radio state machine on each device, particularly the associated transition delay ("tail time") and startup latency, vary based on the wireless radio technology employed (2G, 3G, LTE, etc.) and 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, based on [data provided by AT&T](#). 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.

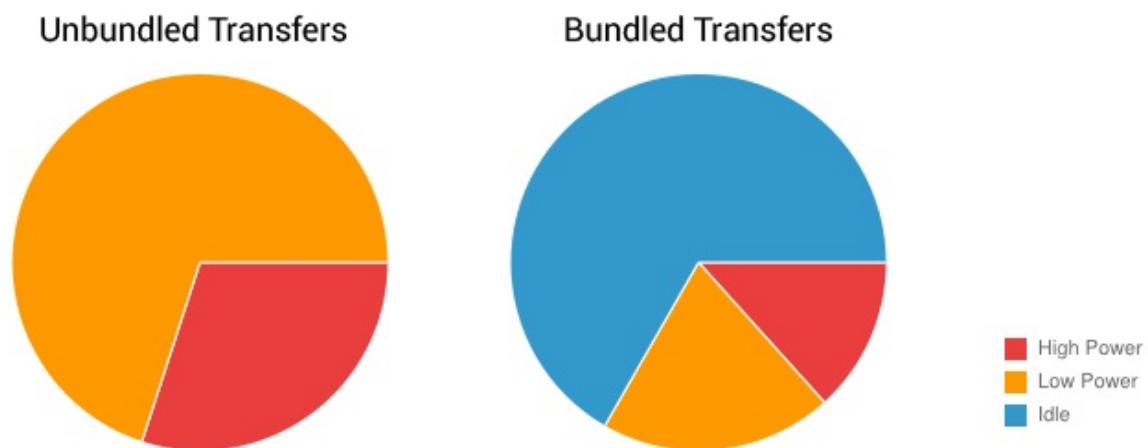
Bundling network transfers

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, it remains at full power for the duration of your transfer, followed by 5 seconds of tail time, followed by 12 seconds at the low energy state before turning off. So, 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.
- By comparison, the same app bundling transfers for 3 seconds of every minute keeps the radio in the high power state for only 8 seconds, and 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 it 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. Or, if a user is watching a video, fetching the next minutes of the video is also a good guess.

Prefetching data is an effective way to reduce the number of independent data transfer sessions. 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 not only conserve battery life, but also improve latency for the user, lower the required bandwidth, and reduce download times.

Prefetching has trade-offs. If you download too much or the wrong data, you might increase battery drain. And if you download at the wrong time, users may 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 depends on the size of the data being downloaded and the likelihood of it being used. As 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) before the potential cost of

downloading unused data matches the potential savings of not downloading that data to begin with.

Generally speaking, it's good practice to prefetch data such that you only need to initiate another download every 2 to 5 minutes, and on the order of 1 to 5 megabytes.

Following this principle, large downloads—such as video files—should be downloaded in chunks at regular intervals (every 2 to 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, perhaps less intuitively, because the tail-time battery cost is relatively higher, it is also more efficient to keep the radio active for longer periods during each transfer session to reduce the frequency of updates.
- *WiFi radio* uses significantly less battery than wireless and offers greater bandwidth.

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` registered for the battery status actions:

```

<receiver android:name=".PowerConnectionReceiver">
    <intent-filter>
        <action android:name="android.intent.action.ACTION_POWER_CONNECTED"/>
        <action android:name="android.intent.action.ACTION_POWER_DISCONNECTED"/>
    </intent-filter>
</receiver>

```

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

```
"android.intent.action.BATTERY_LOW"
"android.intent.action.BATTERY_OKAY"
```

JobScheduler

Constantly monitoring the connectivity and battery status of the device can be a challenge, and 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 this 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:

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

Note: `JobScheduler` is only available from API 21+. There is no backwards compatible version for prior API releases. If your app targets devices with earlier API levels, you might find the [FirebaseJobDispatcher](#) a useful alternative.

1. 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 available constraints with their respective `set()` methods and class constants:

- **Backoff/Retry policy:** Determines when how the task should be rescheduled if it fails. Set this condition using the `setBackoffCriteria()` method, which takes two arguments: the initial time to wait after the task fails, and the backoff strategy. The backoff strategy argument can be one of two constants: `BACKOFF_POLICY_LINEAR` or `BACKOFF_POLICY_EXPONENTIAL`. This defaults to {30 seconds, Exponential}.
- **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 exclusive with the minimum latency and override deadline conditions: setting `setPeriodic()` with one of them results in an error.
- **Persisted:** Sets whether the job is persisted across system reboots. For this condition to work, your app must hold the `RECEIVE_BOOT_COMPLETED` permission. Set this condition using the `setPersisted()` method, passing in a boolean that indicates whether or not to persist the task.
- **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 or not 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 or not 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` .

2. 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()

The system calls `onStopJob()` if it determines that you must stop execution of your job even before you've call `jobFinished()` . This happens if the requirements that you specified when you scheduled the job are no longer met.

Examples:

- If you request WiFi with `setRequiredNetworkType()` but the user turns off WiFi while your job is executing, the system calls `onStopJob()` .
- If you specify `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.

3. 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. Typically, if you don't specify a deadline on your job, the system can run it at any time, depending on the current state of the `JobScheduler` object's internal queue; however, it 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 exercises and practical documentation is in [Android Developer Fundamentals: Practicals](#).

- [JobScheduler](#)

Learn more

- [Transferring Data Without Draining the Battery Guide](#)
- [Optimizing Downloads for Efficient Network Access Guide](#)
- [Modifying your Download Patterns Based on the Connectivity Type Guide](#)
- [JobScheduler Reference](#)
- [JobService Reference](#)
- [JobInfo Reference](#)
- [JobInfo.Builder Reference](#)
- [JobParameters Reference](#)
- [Presentation on Scheduling Tasks](#)
- [What are the different networks and speeds?](#)