Scheduling Repeating Alarms

Alarms (based on the `AlarmManager` (/reference/android/app/AlarmManager.html) class) give you a way to perform time-based operations outside the lifetime of your application. For example, you could use an alarm to initiate a long-running operation, such as starting a service once a day to download a weather forecast.

Alarms have these characteristics:

- They let you fire Intents at set times and/or intervals.
- You can use them in conjunction with broadcast receivers to start services and perform other operations.
- They operate outside of your application, so you can use them to trigger events or actions even when your app is not running, and even if the device itself is asleep.
- They help you to minimize your app's resource requirements. You can schedule operations without relying on timers or continuously running background services.

> **Note:** For timing operations that are guaranteed to occur *during* the lifetime of your application, instead consider using the `Handler` (/reference/android/os/Handler.html) class in conjunction with `Timer` (/reference/java/util/Timer.html) and `Thread` (/reference/java/lang/Thread.html). This approach gives Android better control over system resources.

THIS LESSON TEACHES YOU TO

1. Understand the Trade-offs
2. Set a Repeating Alarm
3. Cancel an Alarm
4. Start an Alarm When the Device Boots

TRY IT OUT

Download the sample

Scheduler.zip

VIDEO
The App Clinic: Cricket

VIDEO
DevBytes: Efficient Data Transfers

Understand the Trade-offs

A repeating alarm is a relatively simple mechanism with limited flexibility. It may not be the best choice for your app, particularly if you need to trigger network operations. A poorly designed alarm can cause battery drain and put a significant load on servers.

A common scenario for triggering an operation outside the lifetime of your app is syncing data with a server. This is a case where you might be tempted to use a repeating alarm. But if you own the server that is hosting your app's data, using Google Cloud Messaging (/google/gcm/index.html) (GCM) in conjunction with sync adapter (/training/sync-adapters/index.html) is a better solution than `AlarmManager` (/reference/android/app/AlarmManager.html). A sync adapter gives you all the same scheduling options as `AlarmManager` (/reference/android/app/AlarmManager.html), but it offers you significantly more flexibility. For example, a sync could be based on a "new data" message from the server/device (see Running a Sync Adapter (/training/sync-adapters/running-sync-adapter.html) for details), the user's activity (or inactivity), the time of day, and so on. See the linked videos at the top of this page for a detailed discussion of when and how to use GCM and sync adapter.

Best practices

Every choice you make in designing your repeating alarm can have consequences in 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 syncs at 11:00 p.m., the load on the server could result in high latency or even "denial of service." Follow these best practices in using alarms:

- Add randomness (jitter) to any network requests that trigger as a result of a repeating alarm:
  - Do any local work when the alarm triggers. "Local work" means anything that doesn't hit a server or require the data from the server.
  - At the same time, schedule the alarm that contains the network requests to fire at some random period of time.

- Keep your alarm frequency to a minimum.
- Don't wake up the device unnecessarily (this behavior is determined by the alarm type, as described in Choose an alarm type).
- Don't make your alarm's trigger time any more precise than it has to be.

  Use `setInexactRepeating()` (/reference/android/app/AlarmManager.html#setInexactRepeating(int, long, long, android.app.PendingIntent)) instead of `setRepeating()` (/reference/android/app/AlarmManager.html#setRepeating(int, long, long, android.app.PendingIntent)). When you use `setInexactRepeating()` (/reference/android/app/AlarmManager.html#setInexactRepeating(int, long, long, android.app.PendingIntent)), 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 Android 4.4 (API Level 19), all repeating alarms are inexact. Note that while `setInexactRepeating()` (/reference/android/app/AlarmManager.html#setInexactRepeating(int, long, long, android.app.PendingIntent)) is an improvement over `setRepeating()` (/reference/android/app/AlarmManager.html#setRepeating(int, long, long, android.app.PendingIntent)), 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 discussed above.

- Avoid basing your alarm on clock time if possible.

  Repeating alarms that are based on a precise trigger time don't scale well. Use `ELAPSED_REALTIME` (/reference/android/app/AlarmManager.html#ELAPSED_REALTIME) if you can. The different alarm types are described in more detail in the following section.

Set a Repeating Alarm

As described above, repeating alarms are a good choice for scheduling regular events or data lookups. A repeating alarm has the following characteristics:

- A alarm type. For more discussion, see Choose an alarm type.
- A trigger time. If the trigger time you specify is in the past, the alarm triggers immediately.
- The alarm's interval. For example, once a day, every hour, every 5 seconds, and so on.
- A pending intent that fires when the alarm is triggered. When you set a second alarm that uses the same pending intent, it replaces the original alarm.

Choose an alarm type

One of the first considerations in using a repeating alarm is what its type should be.

There are two general clock types for alarms: "elapsed real time" and "real time clock" (RTC). Elapsed real time uses the "time since system boot" as a reference, and real time clock uses UTC (wall clock) time. This means that elapsed real time is suited to setting an alarm based on the passage of time (for example, an alarm that fires every 30 seconds) since it isn't affected by time zone/locale. The real time clock type is better suited for alarms that are dependent on current locale.

Both types have a "wakeup" version, which says to wake up the device's CPU if the screen is off. This ensures that the alarm will fire at the scheduled time. This is useful if your app has a time dependency—for example, if it has a limited window to perform a particular operation. If you don't use the wakeup version of your alarm type, then all the repeating alarms will fire when your device is next awake.

If you simply need your alarm to fire at a particular interval (for example, every half hour), use one of the elapsed real time types. In general, this is the better choice.

If you need your alarm to fire at a particular time of day, then choose one of the clock-based real time clock types. Note, however, that this approach can have some drawbacks—the app may not translate well to other locales, and if the user changes the device's time setting, it could cause unexpected behavior in your app. Using a real time clock alarm type also does not scale well, as discussed above. We recommend that you use a "elapsed real time" alarm if you can.

Here is the list of types:

- `ELAPSED_REALTIME`—Fires the pending intent based on the amount of time since the device was booted, but

doesn't wake up the device. The elapsed time includes any time during which the device was asleep.

- ELAPSED_REALTIME_WAKEUP—Wakes up the device and fires the pending intent after the specified length of time has elapsed since device boot.
- RTC—Fires the pending intent at the specified time but does not wake up the device.
- RTC_WAKEUP—Wakes up the device to fire the pending intent at the specified time.

ELAPSED_REALTIME_WAKEUP examples

Here are some examples of using ELAPSED_REALTIME_WAKEUP (/reference/android/app/AlarmManager.html#ELAPSED_REALTIME_WAKEUP).

Wake up the device to fire the alarm in 30 minutes, and every 30 minutes after that:

```
// Hopefully your alarm will have a lower frequency than this!
alarmMgr.setInexactRepeating(AlarmManager.ELAPSED_REALTIME_WAKEUP,
        AlarmManager.INTERVAL_HALF_HOUR,
        AlarmManager.INTERVAL_HALF_HOUR, alarmIntent);
```

Wake up the device to fire a one-time (non-repeating) alarm in one minute:

```
private AlarmManager alarmMgr;
private PendingIntent alarmIntent;
...
alarmMgr = (AlarmManager)context.getSystemService(Context.ALARM_SERVICE);
Intent intent = new Intent(context, AlarmReceiver.class);
alarmIntent = PendingIntent.getBroadcast(context, 0, intent, 0);

alarmMgr.set(AlarmManager.ELAPSED_REALTIME_WAKEUP,
        SystemClock.elapsedRealtime() +
        60 * 1000, alarmIntent);
```

RTC examples

Here are some examples of using RTC_WAKEUP (/reference/android/app/AlarmManager.html#RTC_WAKEUP).

Wake up the device to fire the alarm at approximately 2:00 p.m., and repeat once a day at the same time:

```
// Set the alarm to start at approximately 2:00 p.m.
Calendar calendar = Calendar.getInstance();
calendar.setTimeInMillis(System.currentTimeMillis());
calendar.set(Calendar.HOUR_OF_DAY, 14);

// With setInexactRepeating(), you have to use one of the AlarmManager interval
// constants--in this case, AlarmManager.INTERVAL_DAY.
alarmMgr.setInexactRepeating(AlarmManager.RTC_WAKEUP, calendar.getTimeInMillis(),
        AlarmManager.INTERVAL_DAY, alarmIntent);
```

Wake up the device to fire the alarm at precisely 8:30 a.m., and every 20 minutes thereafter:

```
private AlarmManager alarmMgr;
private PendingIntent alarmIntent;
...
alarmMgr = (AlarmManager)context.getSystemService(Context.ALARM_SERVICE);
Intent intent = new Intent(context, AlarmReceiver.class);
alarmIntent = PendingIntent.getBroadcast(context, 0, intent, 0);

// Set the alarm to start at 8:30 a.m.
```

```
Calendar calendar = Calendar.getInstance();
calendar.setTimeInMillis(System.currentTimeMillis());
calendar.set(Calendar.HOUR_OF_DAY, 8);
calendar.set(Calendar.MINUTE, 30);

// setRepeating() lets you specify a precise custom interval--in this case,
// 20 minutes.
alarmMgr.setRepeating(AlarmManager.RTC_WAKEUP, calendar.getTimeInMillis(),
        1000 * 60 * 20, alarmIntent);
```

Decide how precise your alarm needs to be

As described above, choosing the alarm type is often the first step in creating an alarm. A further distinction is how precise you need your alarm to be. For most apps, `setInexactRepeating()` (/reference/android/app/AlarmManager.html#setInexactRepeating(int, long, long, android.app.PendingIntent)) is the right choice. When you use this method, Android synchronizes multiple inexact repeating alarms and fires them at the same time. This reduces the drain on the battery.

For the rare app that has rigid time requirements—for example, the alarm needs to fire precisely at 8:30 a.m., and every hour on the hour thereafter—use `setRepeating()` (/reference/android/app/AlarmManager.html#setRepeating(int, long, long, android.app.PendingIntent)). But you should avoid using exact alarms if possible.

With `setInexactRepeating()` (/reference/android/app/AlarmManager.html#setInexactRepeating(int, long, long, android.app.PendingIntent)), you can't specify a custom interval the way you can with `setRepeating()` (/reference/android/app/AlarmManager.html#setRepeating(int, long, long, android.app.PendingIntent)). You have to use one of the interval constants, such as `INTERVAL_FIFTEEN_MINUTES` (/reference/android/app/AlarmManager.html#INTERVAL_FIFTEEN_MINUTES), `INTERVAL_DAY` (/reference/android/app/AlarmManager.html#INTERVAL_DAY), and so on. See `AlarmManager` (/reference/android/app/AlarmManager.html) for the complete list.

Cancel an Alarm

Depending on your app, you may want to include the ability to cancel the alarm. To cancel an alarm, call `cancel()` (/reference/android/app/AlarmManager.html#cancel(android.app.PendingIntent)) on the Alarm Manager, passing in the `PendingIntent` (/reference/android/app/PendingIntent.html) you no longer want to fire. For example:

```
// If the alarm has been set, cancel it.
if (alarmMgr!= null) {
    alarmMgr.cancel(alarmIntent);
}
```

Start an Alarm When the Device Boots

By default, all alarms are canceled when a device shuts down. To prevent this from happening, you can design your application to automatically restart a repeating alarm if the user reboots the device. This ensures that the `AlarmManager` (/reference/android/app/AlarmManager.html) will continue doing its task without the user needing to manually restart the alarm.

Here are the steps:

1. Set the `RECEIVE_BOOT_COMPLETED` permission in your application's manifest. This allows your app to receive the `ACTION_BOOT_COMPLETED` that is broadcast after the system finishes booting (this only works if the app has already been launched by the user at least once):

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

2. Implement a `BroadcastReceiver` to receive the broadcast:

```
public class SampleBootReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {
        if (intent.getAction().equals("android.intent.action.BOOT_COMPLETED")
            // Set the alarm here.
        }
    }
}
```

3. Add the receiver to your app's manifest file with an intent filter that filters on the `ACTION_BOOT_COMPLETED` action:

```
<receiver android:name=".SampleBootReceiver"
        android:enabled="false">
    <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED"></action>
    </intent-filter>
</receiver>
```

Notice that in the manifest, the boot receiver is set to `android:enabled="false"`. This means that the receiver will not be called unless the application explicitly enables it. This prevents the boot receiver from being called unnecessarily. You can enable a receiver (for example, if the user sets an alarm) as follows:

```
ComponentName receiver = new ComponentName(context, SampleBootReceiver.class)
PackageManager pm = context.getPackageManager();

pm.setComponentEnabledSetting(receiver,
        PackageManager.COMPONENT_ENABLED_STATE_ENABLED,
        PackageManager.DONT_KILL_APP);
```

Once you enable the receiver this way, it will stay enabled, even if the user reboots the device. In other words, programmatically enabling the receiver overrides the manifest setting, even across reboots. The receiver will stay enabled until your app disables it. You can disable a receiver (for example, if the user cancels an alarm) as follows:

```
ComponentName receiver = new ComponentName(context, SampleBootReceiver.class)
PackageManager pm = context.getPackageManager();

pm.setComponentEnabledSetting(receiver,
        PackageManager.COMPONENT_ENABLED_STATE_DISABLED,
        PackageManager.DONT_KILL_APP);
```