The [IntentService](https://developer.android.com/reference/android/app/IntentService.html) class provides a straightforward structure for running an operation on a single background thread. This allows it to handle long-running operations without affecting your user interface's responsiveness. Also, an [IntentService](https://developer.android.com/reference/android/app/IntentService.html) isn't affected by most user interface lifecycle events, so it continues to run in circumstances that would shut down an [AsyncTask](https://developer.android.com/reference/android/os/AsyncTask.html)

An [IntentService](https://developer.android.com/reference/android/app/IntentService.html) has a few limitations:

* It can't interact directly with your user interface. To put its results in the UI, you have to send them to an [Activity](https://developer.android.com/reference/android/app/Activity.html).
* Work requests run sequentially. If an operation is running in an [IntentService](https://developer.android.com/reference/android/app/IntentService.html), and you send it another request, the request waits until the first operation is finished.
* An operation running on an [IntentService](https://developer.android.com/reference/android/app/IntentService.html) can't be interrupted.

Notice that the [<service>](https://developer.android.com/guide/topics/manifest/service-element.html) element doesn't contain an intent filter. The [Activity](https://developer.android.com/reference/android/app/Activity.html) that sends work requests to the service uses an explicit [Intent](https://developer.android.com/reference/android/content/Intent.html), so no filter is needed. This also means that only components in the same app or other applications with the same user ID can access the service.

This [Intent](https://developer.android.com/reference/android/content/Intent.html) can optionally contain data for the [IntentService](https://developer.android.com/reference/android/app/IntentService.html) to process. You can send an [Intent](https://developer.android.com/reference/android/content/Intent.html) to an [IntentService](https://developer.android.com/reference/android/app/IntentService.html) from any point in an [Activity](https://developer.android.com/reference/android/app/Activity.html) or [Fragment](https://developer.android.com/reference/android/app/Fragment.html).

Create and Send a Work Request to an IntentService

To create a work request and send it to an [IntentService](https://developer.android.com/reference/android/app/IntentService.html), create an explicit [Intent](https://developer.android.com/reference/android/content/Intent.html), add work request data to it, and send it to [IntentService](https://developer.android.com/reference/android/app/IntentService.html) by calling [startService()](https://developer.android.com/reference/android/content/Context.html" \l "startService(android.content.Intent)).

The next snippets demonstrate this:

1. Create a new, explicit [Intent](https://developer.android.com/reference/android/content/Intent.html) for the [IntentService](https://developer.android.com/reference/android/app/IntentService.html) called RSSPullService.

/\*  
 \* Creates a new Intent to start the RSSPullService  
 \* IntentService. Passes a URI in the  
 \* Intent's "data" field.  
 \*/  
mServiceIntent = new Intent(getActivity(), RSSPullService.class);  
mServiceIntent.setData(Uri.parse(dataUrl));

1. Call [startService()](https://developer.android.com/reference/android/content/Context.html" \l "startService(android.content.Intent))

// Starts the IntentService  
getActivity().startService(mServiceIntent);

Notice that you can send the work request from anywhere in an Activity or Fragment. For example, if you need to get user input first, you can send the request from a callback that responds to a button click or similar gesture.

Once you call [startService()](https://developer.android.com/reference/android/content/Context.html" \l "startService(android.content.Intent)), the [IntentService](https://developer.android.com/reference/android/app/IntentService.html) does the work defined in its [onHandleIntent()](https://developer.android.com/reference/android/app/IntentService.html" \l "onHandleIntent(android.content.Intent)) method, and then stops itself.

**Note :** A single [BroadcastReceiver](https://developer.android.com/reference/android/content/BroadcastReceiver.html) can handle more than one type of broadcast [Intent](https://developer.android.com/reference/android/content/Intent.html) object, each with its own action. This feature allows you to run different code for each action, without having to define a separate [BroadcastReceiver](https://developer.android.com/reference/android/content/BroadcastReceiver.html) for each action. To define another [IntentFilter](https://developer.android.com/reference/android/content/IntentFilter.html) for the same[BroadcastReceiver](https://developer.android.com/reference/android/content/BroadcastReceiver.html), create the [IntentFilter](https://developer.android.com/reference/android/content/IntentFilter.html) and repeat the call to [registerReceiver()](https://developer.android.com/reference/android/support/v4/content/LocalBroadcastManager.html" \l "registerReceiver(android.content.BroadcastReceiver, android.content.IntentFilter)).

**Note :** Sending an broadcast [Intent](https://developer.android.com/reference/android/content/Intent.html) doesn't start or resume an [Activity](https://developer.android.com/reference/android/app/Activity.html). The [BroadcastReceiver](https://developer.android.com/reference/android/content/BroadcastReceiver.html) for an [Activity](https://developer.android.com/reference/android/app/Activity.html) receives and processes [Intent](https://developer.android.com/reference/android/content/Intent.html) objects even when your app is in the background, but doesn't force your app to the foreground. If you want to notify the user about an event that happened in the background while your app was not visible, use a [Notification](https://developer.android.com/reference/android/app/Notification.html). *Never* start an [Activity](https://developer.android.com/reference/android/app/Activity.html) in response to an incoming broadcast [Intent](https://developer.android.com/reference/android/content/Intent.html).

## Keep the Screen On

Certain apps need to keep the screen turned on, such as games or movie apps. The best way to do this is to use the [FLAG\_KEEP\_SCREEN\_ON](https://developer.android.com/reference/android/view/WindowManager.LayoutParams.html#FLAG_KEEP_SCREEN_ON) in your activity (and only in an activity, never in a service or other app component). For example:

public class MainActivity extends Activity {  
  @Override  
  protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity\_main);  
    **getWindow().addFlags(WindowManager.LayoutParams.FLAG\_KEEP\_SCREEN\_ON);**  
  }

The advantage of this approach is that unlike wake locks (discussed in [Keep the CPU On](https://developer.android.com/training/scheduling/wakelock.html#cpu)), it doesn't require special permission, and the platform correctly manages the user moving between applications, without your app needing to worry about releasing unused resources.

Another way to implement this is in your application's layout XML file, by using the [android:keepScreenOn](https://developer.android.com/reference/android/R.attr.html" \l "keepScreenOn) attribute:

<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    android:layout\_width="match\_parent"  
    android:layout\_height="match\_parent"  
    **android:keepScreenOn="true">**  
    ...  
</RelativeLayout>

## Keep the CPU On

If you need to keep the CPU running in order to complete some work before the device goes to sleep, you can use a [PowerManager](https://developer.android.com/reference/android/os/PowerManager.html) system service feature called wake locks. Wake locks allow your application to control the power state of the host device.

PowerManager powerManager = (PowerManager) getSystemService(POWER\_SERVICE);  
WakeLock wakeLock = powerManager.newWakeLock(PowerManager.PARTIAL\_WAKE\_LOCK,  
        "MyWakelockTag");  
wakeLock.acquire();

To release the wake lock, call [wakelock.release()](https://developer.android.com/reference/android/os/PowerManager.WakeLock.html" \l "release()). This releases your claim to the CPU. It's important to release a wake lock as soon as your app is finished using it to avoid draining the battery.

# **Scheduling Repeating Alarms**

Alarms (based on the [AlarmManager](https://developer.android.com/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](https://developer.android.com/reference/android/os/Handler.html) class in conjunction with [Timer](https://developer.android.com/reference/java/util/Timer.html) and [Thread](https://developer.android.com/reference/java/lang/Thread.html). This approach gives Android better control over system resources.