Unit 3: Working in the background

Lesson 7: Background tasks

7.1: AsyncTask and AsyncTaskLoader

Contents:

There are several ways to do background processing in Android. Two of those ways are:

- You can do background processing directly, using the `AsyncTask` class.
- You can do background processing indirectly, using the `Loader` framework and then the `AsyncTaskLoader` class.

In most situations the `Loader` framework is a better choice, but it's important to know how `AsyncTask` works.
Note: The `Loader` framework provides special-purpose classes that manage loading and reloading updated data asynchronously in the background. You learn more about loaders in a later lesson.

In this chapter you learn why it's important to process some tasks in the background, off the UI thread. You learn how to use `AsyncTask`, when *not* to use `AsyncTask`, and the basics of using loaders.

The UI thread

When an Android app starts, it creates the *main thread*, which is often called the *UI thread*. The UI thread dispatches events to the appropriate user interface (UI) widgets. The UI thread is where your app interacts with components from the Android UI toolkit (components from the `android.widget` and `android.view` packages). Android's thread model has two rules:

- Do not block the UI thread.
- Do UI work only on the UI thread.

The UI thread needs to give its attention to drawing the UI and keeping the app responsive to user input. If everything happened on the UI thread, long operations such as network access or database queries could block the whole UI. From the user's perspective, the app would appear to hang. Even worse, if the UI thread were blocked for more than a few seconds (about 5 seconds currently) the user would be presented with the "application not responding" (ANR) dialog. The user might decide to quit your app and uninstall it.

To make sure your app doesn't block the UI thread:

- Complete all work in less than 16 ms for each UI screen.
- Don't run asynchronous tasks and other long-running tasks on the UI thread. Instead, implement tasks on a background thread using `AsyncTask` (for short or interruptible tasks) or `AsyncTaskLoader` (for tasks that are high-priority, or tasks that need to report back to the user or UI).

Conversely, don't use a background thread to manipulate your UI, because the Android UI toolkit is not thread-safe.

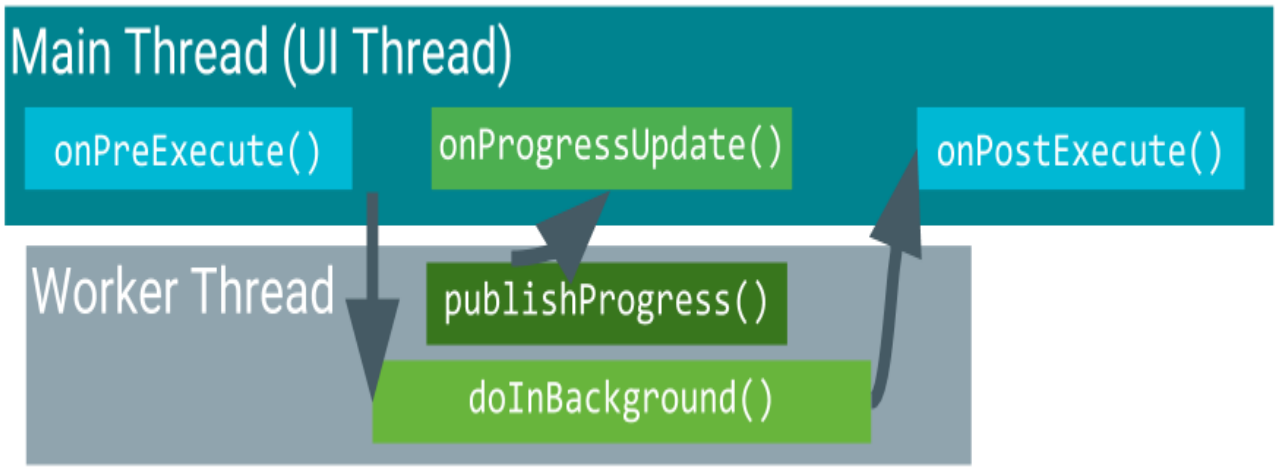
https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/

AsyncTask

A *worker thread* is any thread which is not the main or UI thread. Use the `AsyncTask` class to implement an asynchronous, long-running task on a worker thread. `AsyncTask` allows you to perform background operations on a worker thread and publish results on the UI thread without needing to directly manipulate threads or handlers.
When `AsyncTask` is executed, it goes through four steps:

1. `onPreExecute()` is invoked on the UI thread before the task is executed. This step is normally used to set up the task, for instance by showing a progress bar in the UI.
2. `doInBackground(Params...)` is invoked on the background thread immediately after `onPreExecute()` finishes. This step performs a background computation, returns a result, and passes the result to `onPostExecute()`.
   The `doInBackground()` method can also call `publishProgress(Progress...)` to publish one or more units of progress.
3. `onProgressUpdate(Progress...)` runs on the UI thread after `publishProgress(Progress...)` is invoked. Use `onProgressUpdate()` to report any form of progress to the UI thread while the background computation is executing. For instance, you can use it to pass the data to animate a progress bar or show logs in a text field.
4. `onPostExecute(Result)` runs on the UI thread after the background computation has finished. The result of the background computation is passed to this method as a parameter.

For complete details on these methods, see the `AsyncTask` reference. Below is a diagram of their calling order.



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

AsyncTask usage

To use the `AsyncTask` class, define a subclass of `AsyncTask` that overrides the `doInBackground(Params...)` method (and usually the `onPostExecute(Result)` method as well). This section describes the parameters and usage of `AsyncTask`, then shows a complete example.

AsyncTask parameters

In your subclass of `AsyncTask`, provide the data types for three kinds of parameters:

- "Params" specifies the type of parameters passed to `doInBackground()` as an array.
- "Progress" specifies the type of parameters passed to `publishProgress()` on the background thread. These parameters are then passed to the `onProgressUpdate()` method on the main thread.
- "Result" specifies the type of parameter that `doInBackground()` returns. This parameter is automatically passed to `onPostExecute()` on the main thread.

Specify a data type for each of these parameter types, or use `Void` if the parameter type will not be used. For example:

```
public class MyAsyncTask extends AsyncTask <String, Void, Bitmap>{}
```

In this class declaration:

- The "Params" parameter type is `String`, which means that `MyAsyncTask` takes one or more strings as parameters in `doInBackground()`, for example to use in a query.
- The "Progress" parameter type is `Void`, which means that `MyAsyncTask` won't use the `publishProgress()` or `onProgressUpdate()` methods.
- The "Result" parameter type is `Bitmap`. `MyAsyncTask` returns a Bitmap in `doInbackground()`, which is passed into `onPostExecute()`.

Example of an AsyncTask

```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
     protected Long doInBackground(URL... urls) {
         int count = urls.length;
         long totalSize = 0;
         for (int i = 0; i < count; i++) {
             totalSize += Downloader.downloadFile(urls[i]);
             publishProgress((int) ((i / (float) count) * 100));
             // Escape early if cancel() is called
             if (isCancelled()) break;
         }
         return totalSize;
     }

     protected void onProgressUpdate(Integer... progress) {
         setProgressPercent(progress[0]);
     }

     protected void onPostExecute(Long result) {
         showDialog("Downloaded " + result + " bytes");
     }
 }
```

The example above goes through three of the four basic `AsyncTask` steps:

- `doInBackground()` downloads content, a long-running task. It computes the percentage of files downloaded from the index of the `for` loop and passes it to `publishProgress()`. The check for `isCancelled()` inside the `for` loop ensures that if the task has been cancelled, the system does not wait for the loop to complete.
- `onProgressUpdate()` updates the percent progress. It is called every time the `publishProgress()` method is called inside `doInBackground()`, which updates the percent progress.
- `doInBackground()` computes the total number of bytes downloaded and returns it. `onPostExecute()` receives the returned result and passes it into `onPostExecute()`, where it is displayed in a dialog.

The parameter types used in this example are:

- `URL` for the "Params" parameter type. The `URL` type means you can pass any number of URLs into the call, and the URLs are automatically passed into the `doInBackground()` method as an array.
- `Integer` for the "Progress" parameter type.
- `Long` for the "Result" parameter type.

Executing an AsyncTask

After you define a subclass of `AsyncTask`, instantiate it on the UI thread. Then call `execute()` on the instance, passing in any number of parameters. (These parameters correspond to the "Params" parameter type discussed above).
For example, to execute the `DownloadFilesTask` task defined above, use the following line of code:

```
new DownloadFilesTask().execute(url1, url2, url3);
```

Cancelling an AsyncTask

You can cancel a task at any time, from any thread, by invoking the `cancel()` method.

- The `cancel()` method returns `false` if the task could not be cancelled, typically because it has already completed normally. Otherwise, `cancel()` returns `true`.
- To find out whether a task has been cancelled, check the return value of `isCancelled()` periodically from `doInBackground(Object[])`, for example from inside a loop as shown in the example below. The `isCancelled()` method returns `true` if the task was cancelled before it completed normally.
- After an `AsyncTask` task is cancelled, `onPostExecute()` will not be invoked after `doInBackground()` returns. Instead, `onCancelled(Object)` is invoked. The default implementation of `onCancelled(Object)` calls `onCancelled()` and ignores the result.
- By default, in-process tasks are allowed to complete. To allow `cancel()` to interrupt the thread that's executing the task, pass `true` for the value of `mayInterruptIfRunning`.

Limitations of AsyncTask

`AsyncTask` is impractical for some use cases:

- Changes to device configuration cause problems.

  When device configuration changes while an `AsyncTask` is running, for example if the user changes the screen orientation, the activity that created the `AsyncTask` is destroyed and re-created. The `AsyncTask` is unable to access the newly created activity, and the results of the `AsyncTask` aren't published.
- Old `AsyncTask` objects stay around, and your app may run out of memory or crash.
  If the activity that created the `AsyncTask` is destroyed, the `AsyncTask` is not destroyed along with it. For example, if your user exits the app after the `AsyncTask` has started, the `AsyncTask` keeps using resources unless you call `cancel()`.

When to use `AsyncTask`:

- Short or interruptible tasks.
- Tasks that don't need to report back to UI or user.
- Low-priority tasks that can be left unfinished.

For all other situations, use `AsyncTaskLoader`, which is part of the `Loader` framework described next.

Loaders

Background tasks are commonly used to load data such as forecast reports or movie reviews. Loading data can be memory intensive, and you want the data to be available even if the device configuration changes. For these situations, use *loaders*, which are classes that facilitate loading data into an activity.

Loaders use the `LoaderManager` class to manage one or more loaders. `LoaderManager` includes a set of callbacks for when the loader is created, when it's done loading data, and when it's reset.

Starting a loader

Use the `LoaderManager` class to manage one or more `Loader` instances within an activity or fragment. Use `initLoader()` to initialize a loader and make it active. Typically, you do this within the activity's `onCreate()` method. For example:

```
// Prepare the loader. Either reconnect with an existing one,
// or start a new one.
getLoaderManager().initLoader(0, null, this);
```

If you're using the Support Library, make this call using `getSupportLoaderManager()` instead of `getLoaderManager()`. For example:

```
getSupportLoaderManager().initLoader(0, null, this);
```

The `initLoader()` method takes three parameters:

- A unique ID that identifies the loader. This ID can be whatever you want.
- Optional arguments to supply to the loader at construction, in the form of a `Bundle`. If a loader already exists, this parameter is ignored.
- A `LoaderCallbacks` implementation, which the `LoaderManager` calls to report loader events. In this example, the local class implements the `LoaderManager.LoaderCallbacks` interface, so it passes a reference to itself, `this`.

The `initLoader()` call has two possible outcomes:

- If the loader specified by the ID already exists, the last loader created using that ID is reused.
- If the loader specified by the ID doesn't exist, `initLoader()` triggers the `onCreateLoader()` method. This is where you implement the code to instantiate and return a new loader.

Note: Whether `initLoader()` creates a new loader or reuses an existing one, the given `LoaderCallbacks` implementation is associated with the loader and is called when the loader's state changes. If the requested loader exists and has already generated data, then the system calls `onLoadFinished()` immediately (during `initLoader()`), so be prepared for this to happen. Put the call to `initLoader()` in `onCreate()` so that the activity can reconnect to the same loader when the configuration changes. That way, the loader doesn't lose the data it has already loaded.

Restarting a loader

When `initLoader()` reuses an existing loader, it doesn't replace the data that the loader contains, but sometimes you want it to. For example, when you use a user query to perform a search and the user enters a new query, you want to reload the data using the new search term. In this situation, use the `restartLoader()` method and pass in the ID of the loader you want to restart. This forces another data load with new input data.
About the `restartLoader()` method:

- `restartLoader()` uses the same arguments as `initLoader()`.
- `restartLoader()` triggers the `onCreateLoader()` method, just as `initLoader()` does when creating a new loader.
- If a loader with the given ID exists, `restartLoader()` restarts the identified loader and replaces its data.
- If no loader with the given ID exists, `restartLoader()` starts a new loader.

LoaderManager callbacks

The `LoaderManager` object automatically calls `onStartLoading()` when creating the loader. After that, the `LoaderManager` manages the state of the loader based on the state of the activity and data, for example by calling `onLoadFinished()` when the data has loaded.
To interact with the loader, use one of the `LoaderManager` callbacks in the activity where the data is needed:

- Call `onCreateLoader()` to instantiate and return a new loader for the given ID.
- Call `onLoadFinished()` when a previously created loader has finished loading. This is typically the point at which you move the data into activity views.
- Call `onLoaderReset()` when a previously created loader is being reset, which makes its data unavailable. At this point your app should remove any references it has to the loader's data.

The subclass of the `Loader` is responsible for actually loading the data.
Which `Loader` subclass you use depends on the type of data you are loading, but one of the most straightforward is `AsyncTaskLoader`, described next. `AsyncTaskLoader` uses an `AsyncTask` to perform tasks on a worker thread.

AsyncTaskLoader

`AsyncTaskLoader` is the loader equivalent of `AsyncTask`. `AsyncTaskLoader` provides a method, `loadInBackground()`, that runs on a separate thread. The results of `loadInBackground()` are automatically delivered to the UI thread, by way of the `onLoadFinished()` `LoaderManager` callback.

AsyncTaskLoader usage

To define a subclass of `AsyncTaskLoader`, create a class that extends `AsyncTaskLoader<D>`, where `D` is the data type of the data you are loading. For example, the following `AsyncTaskLoader` loads a list of strings:

```
public static class StringListLoader extends AsyncTaskLoader<List<String>> {}
```

Next, implement a constructor that matches the superclass implementation:

- Your constructor takes the application context as an argument and passes it into a call to `super()`.
- If your loader needs additional information to perform the load, your constructor can take additional arguments.

In the example shown below, the constructor takes a query term.

```
public StringListLoader(Context context, String queryString) {
   super(context);
   mQueryString = queryString;
}
```

To perform the load, use the `loadInBackground()` override method, the corollary to the `doInBackground()` method of `AsyncTask`. For example:

```
@Override
public List<String> loadInBackground() {
    List<String> data = new ArrayList<String>;
    //TODO: Load the data from the network or from a database
    return data;
}
```

Implementing the callbacks

Use the constructor in the `onCreateLoader()` `LoaderManager` callback, which is where the new loader is created. For example, this `onCreateLoader()` callback uses the `StringListLoader` constructor defined above:

```
@Override
public Loader<List<String>> onCreateLoader(int id, Bundle args) {
   return new StringListLoader(this, args.getString("queryString"));
}
```

The results of `loadInBackground()` are automatically passed into the `onLoadFinished()` callback, which is where you can display the results in the UI. For example:

```
public void onLoadFinished(Loader<List<String>> loader, List<String> data) {
```

```
    mAdapter.setData(data);
}
```

The `onLoaderReset()` callback is only called when the loader is being destroyed, so you can leave `onLoaderReset()` blank most of the time, because you won't try to access the data after the loader is destroyed.

When you use `AsyncTaskLoader`, your data survives device-configuration changes. If your activity is permanently destroyed, the loader is destroyed with it, with no lingering tasks that consume system resources.

Loaders have other benefits too, for example, they let you monitor data sources for changes and reload the data if a change occurs. You learn more about the specifics of loaders in a future lesson.

Related practical

The related practical is in 7.1: AsyncTask.

Learn more

- `AsyncTask` reference
- `AsyncTaskLoader` reference
- `LoaderManager` reference
- Processes and threads overview
- Loaders guide

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

7.2: Internet connection

Contents:

Most Android apps engage the user with useful data. That data might be news articles, weather information, contacts, game statistics, and more. Often, data is provided over the network by a web API.

In this chapter you learn about network security and how to make network calls, which involves these general steps:

- Include permissions in your `AndroidManifest.xml` file.
- On a worker thread, make an HTTP client connection that connects to the network and downloads or uploads data.
- Parse the results, which are usually in JSON format.
- Check the state of the network and respond accordingly.

Network security

Network transactions are inherently risky, because they involve transmitting data that could be private to the user. People are increasingly aware of these risks, especially when their devices perform network transactions, so it's very important that your app implement best practices for keeping user data secure at all times.

Security best practices for network operations include:

- Use appropriate protocols for sensitive data. For example for secure web traffic, use the `HttpsURLConnection` subclass of `HttpURLConnection`.
- Use HTTPS instead of HTTP anywhere that HTTPS is supported on the server, because mobile devices frequently connect on insecure networks such as public Wi-Fi hotspots. Consider using `SSLSocketClass` to implement authenticated, encrypted socket-level communication.
- Don't use localhost network ports to handle sensitive interprocess communication (IPC), because other apps on the device can access these local ports. Instead, use a mechanism that lets you use authentication, for example, a `Service`.
- Don't trust data downloaded from HTTP or other insecure protocols. Validate input that's entered into a `WebView` and responses to intents that you issue against HTTP.

For more best practices and security tips, take a look at the Security Tips article.

Including permissions in the manifest

Before your app can make network calls, you need to include a permission in your `AndroidManifest.xml` file. Add the following tag inside the `<manifest>` tag:

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

When using the network, it's a best practice to monitor the network state of the device so that you don't attempt to make network calls when the network is unavailable. To access the network state of the device, your app needs an additional permission:

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

Performing network operations on a worker thread

Always perform network operations on a worker thread, separate from the UI thread. For example, in your Java code you could create an `AsyncTask` (or `AsyncTaskLoader`) implementation that opens a network connection

and queries an API. Your main code checks whether a network connection is active. If so, it runs the `AsyncTask` in a separate thread, then displays the results in the UI.
Note: If you run network operations on the main thread instead of on a worker thread, your code will throw a `NetworkOnMainThreadException` and your app will close.

Making an HTTP connection

Most network-connected Android apps use HTTP and HTTPS to send and receive data over the network. For a refresher on HTTP, visit this Learn HTTP tutorial.

Note: If a web server offers HTTPS, you should use it instead of HTTP for improved security.

The `HttpURLConnection` Android client supports HTTPS, streaming uploads and downloads, configurable timeouts, IPv6, and connection pooling. To use the `HttpURLConnection` client, build a URI (the request's destination). Then obtain a connection, send the request and any request headers, download and read the response and any response headers, and disconnect.

Building your URI

To open an HTTP connection, you need to build a request URI as a `Uri` object. A URI object is usually made up of a base URL and a collection of query parameters that specify the resource in question. For example to search for the first five book results for "Pride and Prejudice" in the Google Books API, use the following URI:
`https://www.googleapis.com/books/v1/volumes?q=pride+prejudice&maxResults=5&printType=books`
To construct a request URI programmatically, use the `URI.parse()` method with the `buildUpon()` and `appendQueryParameter()` methods. The following code builds the complete URI shown above:

```
// Base URL for the Books API.
final String BOOK_BASE_URL =
   "https://www.googleapis.com/books/v1/volumes?";

// Parameter for the search string
final String QUERY_PARAM = "q";
// Parameter to limit search results.
final String MAX_RESULTS = "maxResults";
// Parameter to filter by print type
final String PRINT_TYPE = "printType";

// Build up the query URI, limiting results to 5 printed books.
Uri builtURI = Uri.parse(BOOK_BASE_URL).buildUpon()
       .appendQueryParameter(QUERY_PARAM, "pride+prejudice")
       .appendQueryParameter(MAX_RESULTS, "5")
       .appendQueryParameter(PRINT_TYPE, "books")
       .build();
```

To convert the `Uri` object to a string, use the `toString()` method:

```
String myurl = builtURI.toString();
```

Connecting and downloading data

In the worker thread that performs your network transactions, for example within your override of the `doInBackground()` method in an `AsyncTask`, use the `HttpURLConnection` class to perform an HTTP `GET` request and download the data your app needs. Here's how:

1. To obtain a new `HttpURLConnection`, call `URL.openConnection()` using the URI that you've built. Cast the result to `HttpURLConnection`.

The URI is the primary property of the request, but request headers can also include metadata such as credentials, preferred content types, and session cookies.

2. Set optional parameters. For a slow connection, you might want a long connection timeout (the time to make the initial connection to the resource) or read timeout (the time to actually read the data).

   To change the request method to something other than `GET`, use the `setRequestMethod()` method. If you won't use the network for input, call the `setDoInput()` method with an argument of `false`. (The default is `true`.) For more methods you can set, see the `HttpURLConnection` and `URLConnection` reference documentation.
3. Open an input stream using the `getInputStream()` method, then read the response and convert it into a string. Response headers typically include metadata such as the response body content type and length, modification dates, and session cookies. If the response has no body, `getInputStream()` returns an empty stream.
4. Call the `disconnect()` method to close the connection. Disconnecting releases the resources held by a connection so they can be closed or reused.

These steps are shown in the Request example, below.

Uploading data

If you're uploading (posting) data to a web server, you need to upload a *request body*, which holds the data to be posted. To do this:

1. Configure the connection so that output is possible by calling `setDoOutput(true)`. By default, `HttpURLConnection` uses HTTP `GET` requests. When `setDoOutput` is `true`, `HttpURLConnection` uses HTTP `POST` requests instead.
2. Open an output stream by calling the `getOutputStream()` method.

For more about posting data to the network, see "Posting Content" in the `HttpURLConnection` documentation.

Note: All network calls must be performed in a worker thread and not on the UI thread.

Request example

The following example sends a request to the URL built in the Building your URI section, above. The request obtains a new `HttpURLConnection`, opens an `InputStream`, reads the response, converts the response into a string, and closes the connection.

```
private String downloadUrl(String myurl) throws IOException {
    InputStream inputStream = null;
    // Only display the first 500 characters of the retrieved
    // web page content.
    int len = 500;

    try {
        URL url = new URL(myurl);
        HttpURLConnection conn =
          (HttpURLConnection) url.openConnection();
        conn.setReadTimeout(10000 /* milliseconds */);
        conn.setConnectTimeout(15000 /* milliseconds */);
        // Start the query
        conn.connect();
        int response = conn.getResponseCode();
        Log.d(DEBUG_TAG, "The response is: " + response);
        inputStream = conn.getInputStream();

        // Convert the InputStream into a string
        String contentAsString =
           convertInputToString(inputStream, len);
        return contentAsString;

    // Close the InputStream and connection
    } finally {
          conn.disconnect();
        if (inputStream != null) {
            inputStream.close();
        }
    }
}
```

Converting the InputStream to a string

An `InputStream` is a readable source of bytes. Once you get an `InputStream`, it's common to decode or convert it into the data type you need. In the example above, the `InputStream` represents plain text from the web page located at https://www.googleapis.com/books/v1/volumes?q=pride+prejudice&maxResults=5&printType=books
The `convertInputToString` method defined below converts the `InputStream` to a string so that the activity can display it in the UI. The method uses an `InputStreamReader` instance to read bytes and decode them into characters:

```
// Reads an InputStream and converts it to a String.
public String convertInputToString(InputStream stream, int len)
         throws IOException, UnsupportedEncodingException {
    Reader reader = null;
    reader = new InputStreamReader(stream, "UTF-8");
    char[] buffer = new char[len];
    reader.read(buffer);
    return new String(buffer);
}
```

Note: If you expect a long response, wrap your `InputStreamReader` inside a `BufferedReader` for more efficient reading of characters, arrays, and lines. For example: ``` reader = new BufferedReader(new InputStreamReader(stream, "UTF-8")); ```

Parsing the results

When you make web API queries, the results are often in JSON format. Below is an example of a JSON response from an HTTP request. It shows the names of three menu items in a popup menu and the methods that are triggered when the menu items are clicked:

```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}
```

To find the value of an item in the response, use methods from the `JSONObject` and `JSONArray` classes. For example, here's how to find the `"onclick"` value of the third item in the `"menuitem"` array:

```
JSONObject data = new JSONObject(responseString);
JSONArray menuItemArray = data.getJSONArray("menuitem");
JSONObject thirdItem = menuItemArray.getJSONObject(2);
String onClick = thirdItem.getString("onclick");
```

Managing the network state

Making network calls can be expensive and slow, especially if the device has little connectivity. Being aware of the network connection state can prevent your app from attempting to make network calls when the network isn't available.

Sometimes it's also important for your app to know what kind of connectivity the device has: Wi-Fi networks are typically faster than data networks, and data networks are often metered and expensive. To control when certain tasks are performed, monitor the network state and respond appropriately. For example, you may want to wait until the device is connected to Wifi to perform a large file download.

To check the network connection, use the following classes:

- `ConnectivityManager` answers queries about the state of network connectivity. It also notifies apps when network connectivity changes.
- `NetworkInfo` describes the status of a network interface of a given type (currently either mobile or Wi-Fi).

The following code snippet tests whether Wi-Fi and mobile are connected. In the code:

- The `getSystemService()` method gets an instance of `ConnectivityManager` from the context.
- The `getNetworkInfo()` method gets the status of the device's Wi-Fi connection, then its mobile connection. The `getNetworkInfo()` method returns a `NetworkInfo` object, which contains information about the given network's connection status (whether that connection is idle, connecting, and so on).
- The `networkInfo.isConnected()` method returns `true` if the given network is connected. If the network is connected, it can be used to establish sockets and pass data.

```
private static final String DEBUG_TAG = "NetworkStatusExample";
ConnectivityManager connMgr = (ConnectivityManager)
            getSystemService(Context.CONNECTIVITY_SERVICE);
NetworkInfo networkInfo =
            connMgr.getNetworkInfo(ConnectivityManager.TYPE_WIFI);
boolean isWifiConn = networkInfo.isConnected();
networkInfo =
            connMgr.getNetworkInfo(ConnectivityManager.TYPE_MOBILE);
boolean isMobileConn = networkInfo.isConnected();
Log.d(DEBUG_TAG, "Wifi connected: " + isWifiConn);
Log.d(DEBUG_TAG, "Mobile connected: " + isMobileConn);
```

Related practical

The related practical is in 7.2: AsyncTask and AsyncTaskLoader.

Learn more

- Connect to the network
- Manage network usage
- `HttpURLConnection` reference
- `ConnectivityManager` reference
- `InputStream` reference

7.3 Broadcasts

Contents:

In this chapter you learn about broadcasts and broadcast receivers. *Broadcasts* are messaging components used for communicating across different apps, and also with the Android system, when an event of interest occurs. *Broadcast receivers* are the components in your Android app that listen for these events and respond accordingly.

Broadcasts

Broadcasts are messages that the Android system and Android apps send when events occur that might affect the functionality of other apps. For example, the Android system sends an event when the system boots up, when power is connected or disconnected, and when headphones are connected or disconnected. Your Android app can also broadcast events, for example when new data is downloaded.

In general, broadcasts are messaging components used for communicating across apps when events of interest occur. There are two types of broadcasts:

- *System broadcasts* are delivered by the system.
- *Custom broadcasts* are delivered by your app.

System broadcasts

A *system broadcast* is a message that the Android system sends when a system event occurs. System broadcasts are wrapped in `Intent` objects. The intent object's action field contains event details such as `android.intent.action.HEADSET_PLUG`, which is sent when a wired headset is connected or disconnected. The intent can also contain more data about the event in its extra field, for example a `boolean` extra indicating whether a headset is connected or disconnected.
Examples:

- When the device boots, the system broadcasts a system `Intent` with the action `ACTION_BOOT_COMPLETED`.
- When the device is disconnected from external power, the system sends a system `Intent` with the action field `ACTION_POWER_DISCONNECTED`.

System broadcasts aren't targeted at specific recipients. Interested apps must register a component to "listen" for these events. This listening component is called a broadcast receiver.

As the Android system evolves, significant changes are made to improve the system's performance. For example, starting from Android 7.0, the system broadcast actions `ACTION_NEW_PICTURE` and `ACTION_NEW_VIDEO` are not supported. Apps can no longer receive broadcasts about these actions, regardless of the `targetSDK` version on the device. This is because device cameras take pictures and record videos frequently, so sending a system broadcast every time one of these actions occurred would strain a device's memory and battery.
To get the complete list of broadcast actions that the system can send for a particular SDK version, check the `broadcast_actions.txt` file in your SDK folder, at the following path: `Android/sdk/platforms/android-xx/data`, where `xx` is the SDK version.

Custom broadcasts

Custom broadcasts are broadcasts that your app sends out. Use a custom broadcast when you want your app to take an action without launching an activity. For example, use a custom broadcast when you want to let other apps know that data has been downloaded to the device and is available for them to use. More than one broadcast receiver can be registered to receive your broadcast.

To create a custom broadcast, define a custom `Intent` action.
Note: When you specify the action for the `Intent`, use your unique package name (for example `com.example.myproject`) to make sure that your intent doesn't conflict with an intent that is broadcast from a different app or from the Android system.

There are three ways to deliver a custom broadcast:

- For a normal broadcast, pass the intent to `sendBroadcast()`.
- For an ordered broadcast, pass the intent to `sendOrderedBroadcast()`.
- For a local broadcast, pass the intent to `LocalBroadcastManager.sendBroadcast()`.

Normal, ordered, and local broadcasts are described in more detail below.

Normal broadcasts

The `sendBroadcast()` method sends broadcasts to all the registered receivers at the same time, in an undefined order. This is called a *normal broadcast*. A normal broadcast is the most efficient way to send a broadcast. With normal broadcasts, receivers can't propagate the results among themselves, and they can't cancel the broadcast.
The following method sends a normal broadcast to all interested broadcast receivers:

```
public void sendBroadcast() {
   Intent intent = new Intent();
   intent.setAction("com.example.myproject.ACTION_SHOW_TOAST");
   // Set the optional additional information in extra field.
   intent.putExtra("data","This is a normal broadcast");
   sendBroadcast(intent);
}
```

Ordered broadcasts

To send a broadcast to one receiver at a time, use the `sendOrderedBroadcast()` method:

- The `android:priority` attribute that's specified in the intent filter determines the order in which the broadcast is sent.
- If more than one receiver with same priority is present, the sending order is random.
- The `Intent` is propagated from one receiver to the next.
- During its turn, a receiver can update the `Intent`, or it can cancel the broadcast. (If the receiver cancels the broadcast, the `Intent` can't be propagated further.)

For example, the following method sends an ordered broadcast to all interested broadcast receivers:

```
public void sendOrderedBroadcast() {
   Intent intent = new Intent();

   // Set a unique action string prefixed by your app package name.
   intent.setAction("com.example.myproject.ACTION_NOTIFY");
   // Deliver the Intent.
   sendOrderedBroadcast(intent);
}
```

Local broadcasts

If you don't need to send broadcasts to a different app, use the `LocalBroadcastManager.sendBroadcast()` method, which sends broadcasts to receivers within your app. This method is efficient, because it doesn't involve interprocess communication. Also, using local broadcasts protects your app against some security issues.
To send a local broadcast:

1. To get an instance of `LocalBroadcastManager`, call `getInstance()` and pass in the application context.
2. Call `sendBroadcast()` on the instance. Pass in the intent that you want to broadcast.

```
3.  LocalBroadcastManager.getInstance(this).sendBroadcast(customBroadcastIntent);
```

Broadcast receivers

Broadcast receivers are app components that can register for system events or app events. When an event occurs, registered broadcast receivers are notified via an `Intent`. For instance, if you are implementing a media app and you're interested in knowing when the user connects or disconnects a headset, register for the `ACTION_HEADSET_PLUG` intent action.
Use broadcast receivers to respond to messages that have been broadcast from apps or from the Android system. To create a broadcast receiver:

1. Define a subclass of the `BroadcastReceiver` class and implement the `onReceive()` method.
2. Register the broadcast receiver, either statically or dynamically.

These steps are described below.

Subclass a BroadcastReceiver

To create a broadcast receiver, define a subclass of the `BroadcastReceiver` class. This subclass is where `Intent` objects are delivered if they match the intent filters you register for.
Within your subclass:

- Implement the `onReceive()` method, which is called when the `BroadcastReceiver` object receives an `Intent` broadcast .
- Inside `onReceive()`, include any other logic that your broadcast receiver needs.

Example: Create a broadcast receiver

In this example, the `myReceiver` class a subclass of `BroadcastReceiver`. If the incoming broadcast intent has the `ACTION_SHOW_TOAST` action, the `myReceiver` class shows a toast message:

```
//Subclass of the BroadcastReceiver class.
private class myReceiver extends BroadcastReceiver {
   // Override the onReceive method to receive the broadcasts
   @Override
   public void onReceive(Context context, Intent intent) {
      //Check the Intent action and perform the required operation
        if (intent.getAction().equals(ACTION_SHOW_TOAST)) {
            CharSequence text = "Broadcast Received!";
            int duration = Toast.LENGTH_SHORT;

            Toast toast = Toast.makeText(context, text, duration);
            toast.show();
       }
   }
}
```

The `onReceive()` method is called when your app receives a registered `Intent` broadcast. The `onReceive()` method runs on the main thread unless it is explicitly asked to run on a different thread in the `registerReceiver()` method.

The `onReceive()` method has a timeout of 10 seconds. After 10 seconds, the Android system considers your receiver to be blocked, and the system might show the user an "application not responding" error. For this reason, you shouldn't implement long-running operations in `onReceive()`.

Important: Don't use asynchronous operations to run a long-running operation in your `onReceive()` implementation, because once your code returns from `onReceive()`, the system considers the `BroadcastReceiver` component to be finished. If `onReceive()` started an asynchronous operation, the system would stop the `BroadcastReceiver` process before the asynchronous operation had a chance to complete.

In particular:

- Don't try to show a dialog from within a `BroadcastReceiver`. Instead, display a notification using the `NotificationManager` API.
- Don't try to bind to a service from within a `BroadcastReceiver`. Instead, use `Context.startService()` to send a command to the service.

If you need to perform a long-running operation inside `BroadcastReceiver`, use `WorkManager` to schedule a job. When you schedule a task with `WorkManager`, the task is guaranteed to run. `WorkManager` chooses the appropriate way to run your task, based on such factors as the device API level and the app state.

Register your broadcast receiver and set intent filters

There are two types of broadcast receivers:

- Static receivers, which you register in the Android manifest file.
- Dynamic receivers, which you register using a context.

Static receivers

Static receivers are also called *manifest-declared receivers*. To register a static receiver, include the following attributes inside the `<receiver>` element in your `AndroidManifest.xml` file:

- `android:name`
  The value for this attribute is the fully classified name of the `BroadcastReceiver` subclass, including the package name. To use the package name that's specified in the manifest, prefix the subclass name with a period, for example `.AlarmReceiver`.
- `android:exported` (optional)
  If this boolean value is set to `false`, other apps cannot send broadcasts to your receiver. This attribute is important for security.
- `<intent-filter>`
  Include this nested element to specify the broadcast `Intent` actions that your broadcast receiver component is listening for.
  The following code snippet shows static registration of a broadcast receiver that listens for a custom broadcast `Intent` with the action "`ACTION_SHOW_TOAST`":

- The receiver's `name` is the name of the `BroadcastReceiver` subclass (`.AlarmReceiver`).
- The receiver is not exported, meaning that no other apps can deliver broadcasts to this app.
- The intent filter checks whether incoming intents include an action named `ACTION_SHOW_TOAST`, which is a custom `Intent` action defined within the app.

```
<receiver
 android:name=".AlarmReceiver"
 android:exported="false">
 <intent-filter>
     <action android:name=
          "com.example.myproject.intent.action.ACTION_SHOW_TOAST"/>
 </intent-filter>
</receiver>
```

Note: For Android 8.0 (API level 26) and higher, static receivers can't receive most implicit broadcasts. (*Implicit* broadcasts are broadcasts that don't target your app specifically.) Even if you register for these broadcasts in the manifest, the Android system won't deliver them to your app. However, you can still use a dynamic receiver to register for these broadcasts.

A few broadcasts, such as `ACTION_BOOT_COMPLETED` and `ACTION_TIMEZONE_CHANGED`, are excepted from this restriction. You can declare them in the manifest, no matter what the target API level. To learn more, see the complete list of implicit broadcast exceptions.

Intent filters

An *intent filter* specifies the types of intents that a component can receive. When the system receives an `Intent` as a broadcast, it searches the broadcast receivers based on the values specified in receivers' intent filters.
To create an intent filter statically, use the `<intent-filter>` element in the Android manifest. An `<intent-filter>` element has one required element, `<action>`. The Android system compares the `Intent` action of the incoming broadcast to the filter action's `android:name` strings. If any of the names in the filter match the action name in the incoming broadcast, the broadcast is sent to your app.
This `<intent-filter>` listens for a system broadcast that's sent when the device boots up. Only `Intent` objects with an action named `BOOT_COMPLETED` match the filter:

```
<intent-filter>
    <action android:name="android.intent.action.BOOT_COMPLETED"/>
</intent-filter>
```

If no intent filters are specified, the broadcast receiver can only be activated with an explicit broadcast `Intent` that specifies the component by name. (This is similar to how explicit intents are used to launch activities by their class names.)
To learn more about using intent filters, see the Intent resolution section of the `Intent` guide.

Dynamic receivers

Dynamic receivers are also called *context-registered receivers*. You register a dynamic receiver using an application context or an `Activity` context. A dynamic receiver receives broadcasts as long as the registering context is valid:

- If you use the application context to register your receiver, your app receives relevant broadcasts as long as your app is running in either the foreground or the background.
- If you use an `Activity` context to register your receiver, your app receives relevant broadcasts until that `Activity` is destroyed.

To use the context to register your receiver dynamically:

1. Create an `IntentFilter` and add the `Intent` actions that you want your app to listen for. You can add more than one action to the same `IntentFilter` object.

```
IntentFilter intentFilter = new IntentFilter();
filter.addAction(Intent.ACTION_POWER_CONNECTED);
filter.addAction(Intent.ACTION_POWER_DISCONNECTED);
```

When the device power is connected or disconnected, the Android system broadcasts
the `Intent.ACTION_POWER_CONNECTED` and `Intent.ACTION_POWER_DISCONNECTED` intent actions.

5. Register the receiver by calling `registerReceiver()` method on the context. Pass in the `BroadcastReceiver` object and the `IntentFilter` object.

```
mReceiver = new AlarmReceiver();
this.registerReceiver(mReceiver, intentFilter);
```

In this example the `Activity` context (`this`) is used to register your receiver. So the app will receive the broadcast as long as the `Activity` is running.

Local broadcasts

You must register local receivers dynamically, because static registration in the manifest is not possible for a local broadcasts.

To register a receiver for local broadcasts:

1. Get an instance of `LocalBroadcastManager` by calling the `getInstance()` method.
2. Call `registerReceiver()`, passing in the receiver and an `IntentFilter` object.

```
LocalBroadcastManager.getInstance(this)
    .registerReceiver(mReceiver,
            new IntentFilter(CustomReceiver.ACTION_CUSTOM_BROADCAST));
```

Unregister the receiver

To save system resources and avoid leaks, unregister dynamic receivers when your app no longer needs them, or before the `Activity` or app is destroyed. This is also true for local broadcast receivers, because they're registered dynamically.
To unregister a normal broadcast receiver:

1. Call `unregisterReceiver()` and pass in your `BroadcastReceiver` object:

```
unregisterReceiver(mReceiver);
```

To unregister a local broadcast receiver:

3. Get an instance of the `LocalBroadcastManager`.
4. Call `LocalBroadcastManager.unregisterReceiver()` and pass in your `BroadcastReceiver` object:

```
LocalBroadcastManager.getInstance(this)
    .unregisterReceiver(mReceiver);
```

Where you call these `unregisterReceiver()` methods depends on the desired lifecycle of your `BroadcastReceiver` object:

7. Sometimes the receiver is only needed when your activity is visible, for example to disable a network function when the network is not available. In these cases, register the receiver in `onResume()` and unregister the receiver in `onPause()`.
8. You can also use the `onStart()`/`onStop()` or `onCreate()`/`onDestroy()` method pairs, if they are more appropriate for your use case.

Restricting broadcasts

An unrestricted broadcast can pose a security threat, because any registered receiver can receive it. For example, if your app uses a normal broadcast to send an implicit `Intent` that includes sensitive information, an app that contains malware could receive that broadcast. Restricting your broadcast is strongly recommended. Ways to restrict a broadcast:

- If possible, use a `LocalBroadcastManager`, which keeps the data inside your app, avoiding any security leaks. You can only use `LocalBroadcastManager` if you don't need interprocess communication or communication with other apps.
- Use the `setPackage()` method and pass in the package name. Your broadcast is restricted to apps that match the specified package name.
- Enforce access permissions on the sender side, on the receiver side, or both.

To enforce a permission when *sending* a broadcast:

- Supply a non-null permission argument to `sendBroadcast()`. Only receivers that request this permission using the `<uses-permission>` tag in their `AndroidManifest.xml` file can receive the broadcast.

To enforce a permission when *receiving* a broadcast:

- If you register your receiver dynamically, supply a non-null permission to `registerReceiver()`.
- If you register your receiver statically, use the `android:permission` attribute inside the `<receiver>` tag in your `AndroidManifest.xml`.

Best practices

- In your intent actions, prefix `String` constants with your app's package name. Otherwise, you may conflict with other apps' intents. The `Intent` namespace is global.
- Restrict broadcast receivers, as described above.
- Prefer dynamic receivers over static receivers.
- Do not start an `Activity` from a broadcast receiver—use a notification instead. Starting an activity from a broadcast receiver causes a bad user experience if more than one receiver is listening for the same broadcast event.
- Never perform a long running operation in the broadcast receiver's `onReceive(Context, Intent)` method, because the method runs on the main UI thread. Consider using a `JobScheduler` or a `WorkManager` instead.

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

Related practical

The related practical is 7.3: Broadcast receivers.

Learn more

- Broadcasts overview
- `BroadcastReceiver` reference
- Intents and Intent Filters guide
- `LocalBroadcastManager` reference

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

7.4: Services

Contents:

In this chapter you learn about the different types of services, how to use them, and how to manage their lifecycles within your app.

What is a service?

A *service* is an app component that performs long-running operations, usually in the background. Unlike an `Activity`, a service doesn't provide a user interface (UI). Services are defined by the `Service` class or one of its subclasses.
A service can be *started*, *bound*, or both:

- A *started service* is a service that an app component starts by calling `startService()`.
  Use started services for tasks that run in the background to perform long-running operations. Also use started services for tasks that perform work for remote processes.
- A *bound service* is a service that an app component binds to itself by calling `bindService()`.
  Use bound services for tasks that another app component interacts with to perform interprocess communication (IPC). For example, a bound service might handle network transactions, perform file I/O, play music, or interact with a database.

A service runs in the main thread of its hosting process—the service doesn't create its own thread and doesn't run in a separate process unless you specify that it should.

If your service is going to do any CPU-intensive work or blocking operations (such as MP3 playback or networking), create a new thread within the service to do that work. By using a separate thread, you reduce the risk of the user seeing

"application not responding" (ANR) errors, and the app's main thread can remain dedicated to user interaction with your activities.

In Android 8.0 (Oreo, API 26) or higher, the system imposes some new restrictions on running background services when the app itself isn't in the foreground. For details about these restrictions, see Background services and API 26.

To implement any kind of service in your app, do the following steps:

1. Declare the service in the manifest.
2. Extend a `Service` class such as `IntentService` and create implementation code, as described in Started services and Bound services, below.
3. Manage the service lifecycle.

Declaring services in the manifest

As with activities and other components, you must declare all services in your Android manifest. To declare a service, add a `<service>` element as a child of the `<application>` element. For example:

```
<manifest ... >
  ...
  <application ... >
      <service android:name="ExampleService"
               android:exported="false" />
      ...
  </application>
</manifest>
```

To block access to a service from other apps, declare the service as private. To do this, set the `android:exported` attribute to `false`. This stops other apps from starting your service, even when they use an explicit intent.

Started services

How a service starts:

1. An app component such as an `Activity` calls `startService()` and passes in an `Intent`. The `Intent` specifies the service and includes any data for the service to use.
2. The system calls the service's `onCreate()` method and any other appropriate callbacks on the main thread. It's up to the service to implement these callbacks with the appropriate behavior, such as creating a secondary thread in which to work.
3. The system calls the service's `onStartCommand()` method, passing in the `Intent` supplied by the client in step 1. (The *client* in this context is the app component that calls the service.)

Once started, a service can run in the background indefinitely, even if the component that started it is destroyed. Usually, a started service performs a single operation and does not return a result to the caller. For example, it might download or upload a file over the network. When the operation is done, the service should stop itself by calling `stopSelf()`, or another component can stop it by calling `stopService()`.
For instance, suppose an `Activity` needs to save data to an online database. The `Activity` starts a companion service by passing an `Intent` to `startService()`. The service receives the intent in `onStartCommand()`, connects to the internet, and performs the database transaction. When the transaction is done, the service uses `stopSelf()` to stop itself and is destroyed. (This is an example of a service you want to run in a worker thread instead of the main thread.)

IntentService

Most started services don't need to handle multiple requests simultaneously, and if they did, it could be a complex and error-prone multi-threading scenario. For these reasons, the `IntentService` class is a useful subclass of `Service` on which to base your service:

- `IntentService` automatically provides a worker thread to handle your `Intent`.
- `IntentService` handles some of the boilerplate code that regular services need (such as starting and stopping the service).
- `IntentService` can create a work queue that passes one intent at a time to your `onHandleIntent()` implementation, so you don't have to worry about multi-threading.

Note:`IntentService` is subject to the new restrictions on background services in Android 8.0 (API 26). For this reason, Android Support Library 26.0.0 introduces a new `JobIntentService` class, which provides the same functionality as `IntentService` but uses jobs instead of services when running on Android 8.0 or higher.

To implement `IntentService`:

1. Provide a small constructor for the service.
2. Create an implementation of `onHandleIntent()` to do the work that the client provides.

Here's an example implementation of `IntentService`:

```
public class HelloIntentService extends IntentService {
  /**
   * A constructor is required, and must call the
   * super IntentService(String) constructor with a name
   * for the worker thread.
   */
  public HelloIntentService() {
      super("HelloIntentService");
```

```
    }

    /**
     * The IntentService calls this method from the default
     * worker thread with the intent that started the service.
     * When this method returns, IntentService stops the service,
     * as appropriate.
     */
    @Override
    protected void onHandleIntent(Intent intent) {
        // Normally we would do some work here, like download a file.
        // For our sample, we just sleep for 5 seconds.
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            // Restore interrupt status.
            Thread.currentThread().interrupt();
        }
    }
}
```

Bound services

A service is "bound" when an app component binds to it by calling `bindService()`. A bound service offers a client-server interface that allows components to interact with the service, send requests, and get results, sometimes using interprocess communication (IPC) to send and receive information across processes. A bound service runs only as long as another app component is bound to it. Multiple components can bind to the service at once, but when all of them unbind, the service is destroyed.
A bound service generally does not allow components to start it by calling `startService()`.

Implementing a bound service

To implement a bound service, define the interface that specifies how a client can communicate with the service. This interface, which your service returns from the `onBind()` callback method, must be an implementation of `IBinder`.
To retrieve the `IBinder` interface, a client app component calls `bindService()`. Once the client receives the `IBinder`, the client interacts with the service through that interface.
There are multiple ways to implement a bound service, and the implementation is more complicated than a started service. For complete details about bound services, see Bound Services.

Binding to a service

To bind to a service that is declared in the manifest and implemented by an app component, use `bindService()` with an explicit `Intent`.
Caution: Do not use an implicit intent to bind to a service. Doing so is a security hazard, because you can't be certain what service will respond to your intent, and the user can't see which service starts. Beginning with Android 5.0 (API level 21), the system throws an exception if you call `bindService()` with an implicit `Intent` .

Service lifecycle

The lifecycle of a service is simpler than the `Activity` lifecycle. However, it's even more important that you pay close attention to how your service is created and destroyed. Because a service has no UI, services can continue to run in the background with no way for the user to know, even if the user switches to another app. This situation can potentially consume resources and drain the device battery. Similar to an `Activity`, a service has lifecycle callback methods that you can implement to monitor changes in the service's state and perform work at the appropriate times. The following skeleton service implementation demonstrates each of the lifecycle methods:

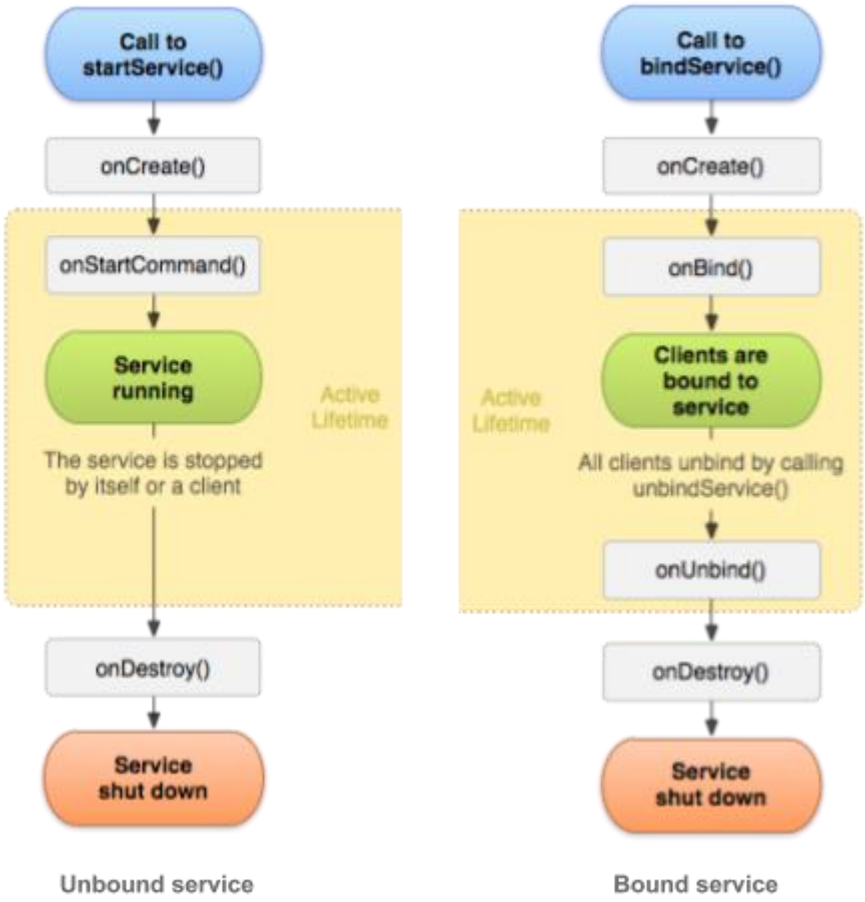
```
public class ExampleService extends Service {
    // indicates how to behave if the service is killed.
    int mStartMode;

    // interface for clients that bind.
    IBinder mBinder;

    // indicates whether onRebind should be used
    boolean mAllowRebind;

    @Override
    public void onCreate() {
        // The service is being created.
    }

    @Override
    public int onStartCommand(Intent intent,
        int flags, int startId) {
        // The service is starting, due to a call to startService().
        return mStartMode;
    }

    @Override
    public IBinder onBind(Intent intent) {
        // A client is binding to the service with bindService().
        return mBinder;
    }

    @Override
    public boolean onUnbind(Intent intent) {
```

```
        // All clients have unbound with unbindService()
        return mAllowRebind;
    }

    @Override
    public void onRebind(Intent intent) {
        // A client is binding to the service with bindService(),
        // after onUnbind() has already been called
    }

    @Override
    public void onDestroy() {
        // The service is no longer used and is being destroyed
    }
}
```

Lifecycle of started services vs. bound services

A bound service exists only to serve the app component that's bound to it, so when no more components are bound to the service, the system destroys it. Bound services don't need to be explicitly stopped the way started services do (using `stopService()` or `stopSelf()`).

The diagram below shows a comparison between the started and bound service



lifecycles.

Foreground services

While most services run in the background, some run in the foreground.
A *foreground service* is a service that the user is aware is running. Although both Activities and Services can be killed if the system is low on memory, a foreground service has priority over other resources.

For example, a music player that plays music from a service should be set to run in the foreground, because the user is aware of its operation. The notification in the status bar might indicate the current song and allow the user to launch an `Activity` to interact with the music player.
To request that a service run in the foreground, call `startForeground()` instead of `startService()`. This method takes two parameters: an integer that uniquely identifies the notification and the `Notification` object for the status bar notification. This notification is *ongoing*, meaning that it can't be dismissed. It stays in the status bar until the service is stopped or removed from the foreground.
For example:

```
Intent notificationIntent = new Intent(this, ExampleActivity.class);
PendingIntent pendingIntent =
        PendingIntent.getActivity(this, 0, notificationIntent, 0);

Notification notification =
          new Notification.Builder(this, CHANNEL_DEFAULT_IMPORTANCE)
    .setContentTitle(getText(R.string.notification_title))
    .setContentText(getText(R.string.notification_message))
    .setSmallIcon(R.drawable.icon)
    .setContentIntent(pendingIntent)
    .setTicker(getText(R.string.ticker_text))
    .build();

startForeground(ONGOING_NOTIFICATION_ID, notification);
```

Note: The integer ID for the notification you give to `startForeground()` must not be 0.

To remove the service from the foreground, call `stopForeground()`. This method takes a boolean, indicating whether to remove the status bar notification. This method doesn't stop the service. However, if you stop the service while it's still running in the foreground, then the notification is also removed.

Background services and API 26

Services running in the background can consume device resources, potentially using device battery and resulting in a worse user experience. To mitigate this problem, the system now applies limitations on started services running in the background, beginning with Android version 8.0 (Oreo, API 26) or higher. These limitations don't affect foreground services or bound services.

Here are a few of the specific changes:

- The `startService()` method now throws an `IllegalStateException` if an app targeting API 26 tries to use that method in a situation when it isn't permitted to create background services.
- The new `startForegroundService()` method starts a foreground service from an app component. The system allows apps to call this method even while the app is in the background. However, the app must call that service's startForeground() method within five seconds after the service is created.

While an app is in the foreground, it can create and run both foreground and background services freely. When an app goes into the background, it has several minutes in which it is still allowed to create and use services. At the end of that time, the app is considered to be idle. The system stops the app's background services, just as if the app had called the services' `Service.stopSelf()` methods.
Android 8.0 and higher does not allow a background app to create a background service. Android 8.0 introduces the new method `startForegroundService()` to start a new service in the foreground.

After the system has created the service, the app has five seconds to call the service's `startForeground()` method to show the new service's user-visible notification. If the app does not call `startForeground()` within the time limit, the system stops the service and declares the app to be ANR (Application Not Responding).
For more information on these changes, see Background Execution Limits.

Scheduled services

For API level 21 and higher, you can launch services using the `JobScheduler` API, and with the background service restrictions in API 26 this may be a better alternative to services altogether. To use `JobScheduler`, you need to register jobs and specify their requirements for network and timing. The system schedules jobs for execution at appropriate times.
The `JobScheduler` interface provides many methods to define service-execution conditions. For details, see the `JobScheduler` reference.
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.

Learn more

- Services overview
- Background Execution Limits