Working in the background:

Internet connection:

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.

Making the network calls 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.

1. 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" />
```

To access the network state of the device, your app needs an additional permission:

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

2. Performing network operations on a worker thread

Always perform network operations on a worker thread, separate from the UI thread.

Making an HTTP connection

Most network-connected Android apps use HTTP and HTTPS to send and receive data over the network.
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 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.

To construct a request URI programmatically, use the `URI.parse()` method with the `buildUpon()` and `appendQueryParameter()` methods.

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.

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`.)

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.

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.

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.

3. 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()"}
    ]
  }
}
```

4. 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).

Services

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.

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 in Started services and Bound services
3. Manage the service lifecycle.

1. Declare the service in the manifest:

To declare a service, add a `<service>` element as a child of the `<application>` element.

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

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
3. The system calls the service's `onStartCommand()` method, passing in the `Intent` supplied by the client in step 1.

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.

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.

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.

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`.

3. Service lifecycle

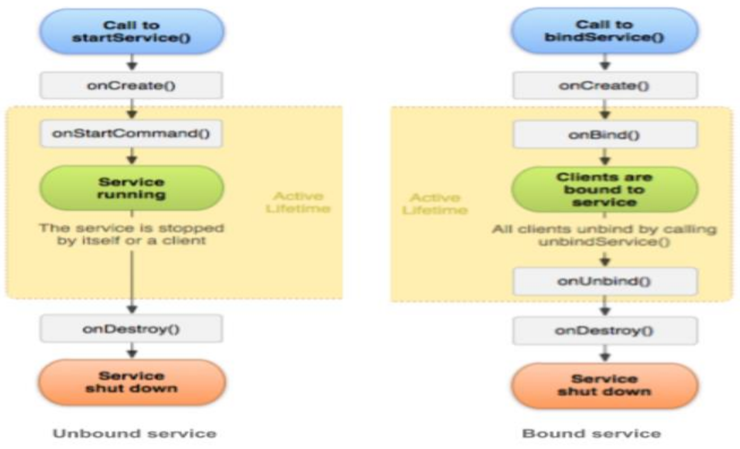
The lifecycle of a service is simpler than the `Activity` lifecycle. However, it's even more important that you pay close attention to how your service is created and destroyed. Because a service has no UI, services can continue to run in the background with no way for the user to know, even if the user 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()`).



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.

BROADCASTS

INTRODUCTION

- *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.
- There are two types of broadcasts:
  - *System broadcasts*
  - *Custom broadcasts*

SYSTEM BROADCASTS

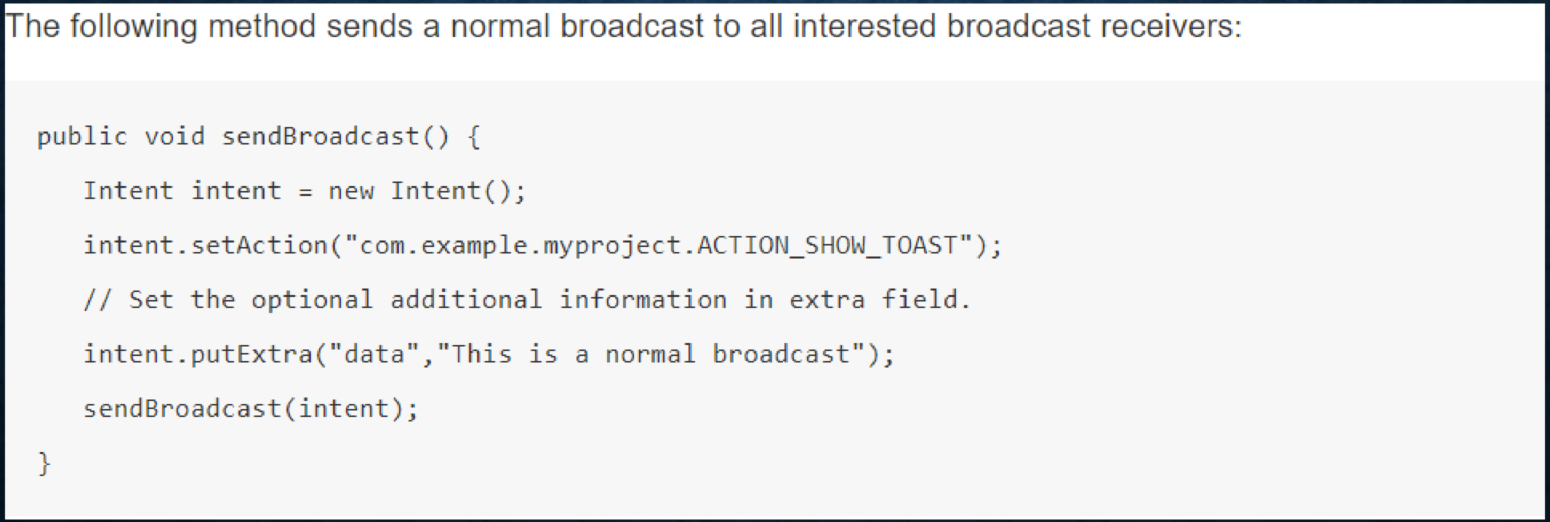
- A system broadcast is a message that the Android system sends when a system event occurs.
- The intent object's action field contains event details such as android.intent.action.HEADSET_PLUG, which is sent when a wired headset is connected or disconnected.
- Examples:
  - When the device boots, the system broadcasts a system Intent with the action ACTION_BOOT_COMPLETED.

CUSTOM BROADCASTS

- Custom broadcasts are broadcasts that your app sends out. The app takes actions without launching an activity.
- For example, when you want to let other apps know that data has been downloaded to the device and is available for them to use.
- There are three ways to deliver a custom broadcast:
    - For a normal broadcast
    - For an ordered broadcast
    - For a local broadcast

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.

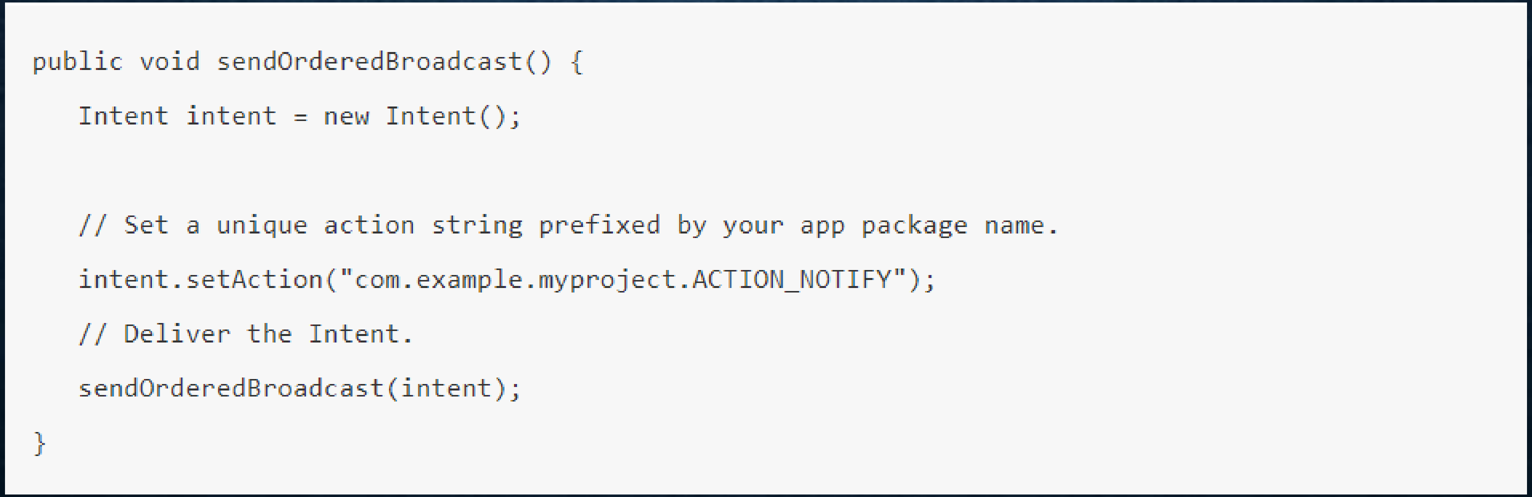


The following method sends a normal broadcast to all interested broadcast receivers:

```
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 BROADCAST

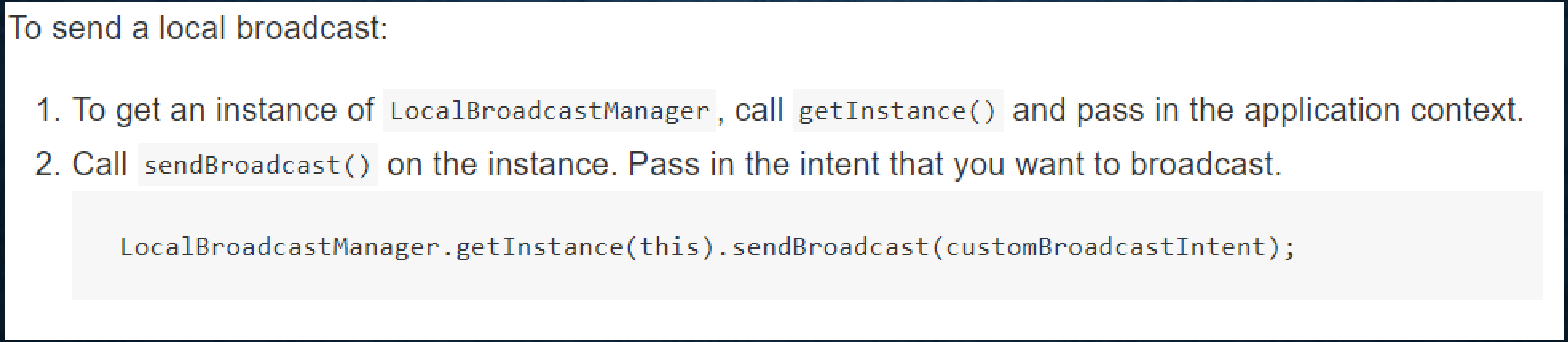
- To send a broadcast to one receiver at a time, use the sendOrderedBroadcast() method:
  - The android:priority attribute that's specified in the intent filter determines the order in which the broadcast is sent.
  - If more than one receiver with same priority is present, the sending order is random.
  - 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.



```
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 BROADCAST

- 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 inter-process 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.

```
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.
- To create a broadcast receiver:
  - Define a subclass of the BroadcastReceiver class and implement the onReceive() method.
  - Register the broadcast receiver, either statically or dynamically.

SUBCLASS A BROADCAST RECEIVER

- 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.

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.

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.

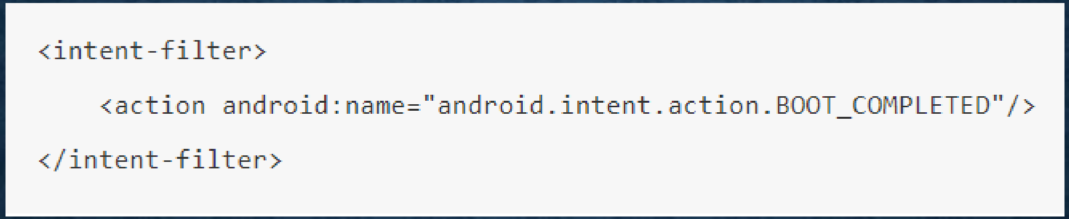
- The following code snippet shows static registration of a broadcast receiver that listens for a custom broadcast Intent with the action "ACTION_SHOW_TOAST":
  - The receiver's name is the name of the BroadcastReceiver subclass (.AlarmReceiver).
  - The receiver is not exported, meaning that no other 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>
```

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>
```

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

```
LocalBroadcastManager.getInstance(this).registerReceiver(mRece
iver,newIntentFilter(CustomReceiver.ACTION_CUSTOM_BROADCA
ST));
```

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);
```

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

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

4. 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().
5. 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.

THANK YOU ALL

Notifications

Notification is a kind of message, alert, or status of an application (probably running in the background) that is visible or available in the Android's UI elements. This application could be running in the background but not in use by the user. The purpose of a notification is to notify the user about a process that was initiated in the application either by the user or the system.

A *notification* is a message your app displays to the user outside your app's normal UI. If the device is unlocked, the user opens the *notification drawer* to see the details of the notification. If the device is locked, the user views the notification on the lock screen. The notification area, lock screen, and notification drawer are system-controlled areas that the user can view at any time.

Create and Send Notifications

Step 1 - Create Notification Builder

As a first step is to create a notification builder using *NotificationCompat.Builder.build()*. You will use Notification Builder to set various Notification properties like its small and large icons, title, priority etc.

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

Step 2 - Setting Notification Properties

Once you have **Builder** object, you can set its Notification properties using Builder object as per your requirement. But this is mandatory to set at least following −

- A small icon, set by **setSmallIcon()**
- A title, set by **setContentTitle()**
- Detail text, set by **setContentText()**

```
mBuilder.setSmallIcon(R.drawable.notification_icon);
mBuilder.setContentTitle("Notification Alert, Click Me!");
mBuilder.setContentText("Hi, This is Android Notification
Detail!");
```

You have plenty of optional properties which you can set for your notification. To learn more about them, see the reference documentation for NotificationCompat.Builder.

Step 3 - Attach Actions

This is an optional part and required if you want to attach an action with the notification. An action allows users to go directly from the notification to an **Activity** in your application, where they can look at one or more events or do further work.

The action is defined by a **PendingIntent** containing an **Intent** that starts an Activity in your application. To associate the PendingIntent with a gesture, call the appropriate method of *NotificationCompat.Builder*. For example, if you want to start Activity when the user clicks the notification text in the notification drawer, you add the PendingIntent by calling **setContentIntent()**.

A PendingIntent object helps you to perform an action on your applications behalf, often at a later time, without caring of whether or not your application is running.

We take help of stack builder object which will contain an artificial back stack for the started Activity. This ensures that navigating backward from the Activity leads out of your application to the Home screen.

```
Intent resultIntent = new Intent(this, ResultActivity.class);
TaskStackBuilder stackBuilder = TaskStackBuilder.create(this);
stackBuilder.addParentStack(ResultActivity.class);
// Adds the Intent that starts the Activity to the top of the
stack
stackBuilder.addNextIntent(resultIntent);
PendingIntent resultPendingIntent =
stackBuilder.getPendingIntent(0,PendingIntent.FLAG_UPDATE_CURRENT
);
mBuilder.setContentIntent(resultPendingIntent);
```

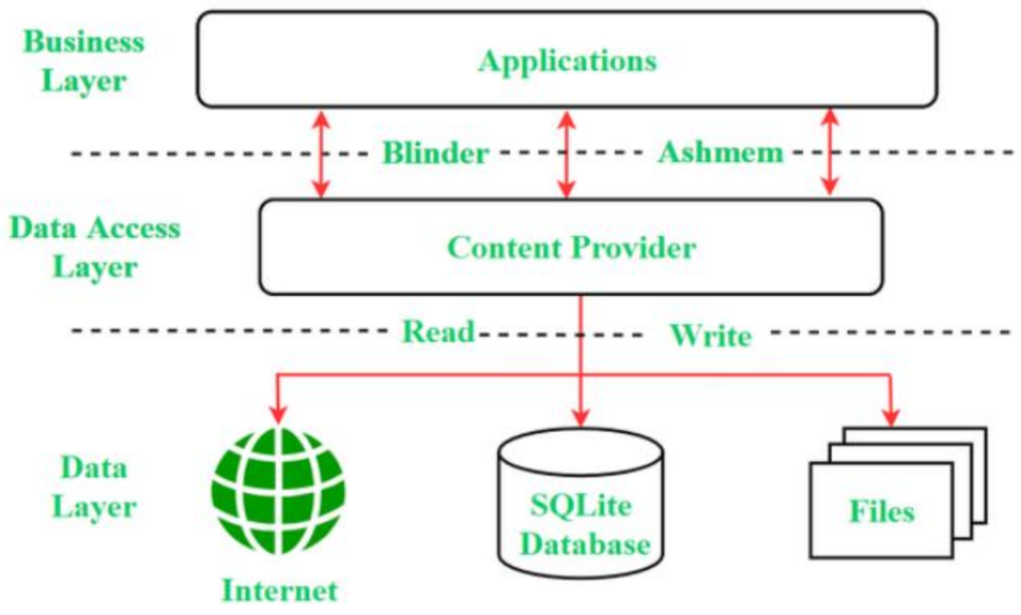
Step 4 - Issue the notification

Finally, you pass the Notification object to the system by calling NotificationManager.notify() to send your notification. Make sure you call **NotificationCompat.Builder.build()** method on builder object before notifying it. This method combines all of the options that have been set and return a new **Notification** object.

```
NotificationManager mNotificationManager = (NotificationManager)
getSystemService(Context.NOTIFICATION_SERVICE);

// notificationID allows you to update the notification later on.
mNotificationManager.notify(notificationID, mBuilder.build());
```

Content providers:

In Android, Content Providers are a very important component that serves the purpose of a relational database to store the data of applications. The role of the content provider in the android system is like a central repository in which data of the applications are stored, and it facilitates other applications to securely access and modifies that data based on the user requirements. Android system allows the content provider to store the application data in several ways. Users can manage to store the application data like images, audio, videos, and personal contact information by storing them in **SQLite Database, in files**, **or even on a network**. In order to share the data, content providers have certain permissions that are used to grant or restrict the rights to other applications to interfere with the data.



Content URI

Content URI(Uniform Resource Identifier) is the key concept of Content providers. To access the data from a content provider, URI is used as a query string.
Structure of a Content URI: *content://authority/optionalPath/optionalID*

Details of different parts of Content URI:

- **content:// –** Mandatory part of the URI as it represents that the given URI is a Content URI.
- **authority –** Signifies the name of the content provider like contacts, browser, etc. This part must be unique for every content provider.

- **optionalPath –** Specifies the type of data provided by the content provider. It is essential as this part helps content providers to support different types of data that are not related to each other like audio and video files.
- **optionalID –** It is a numeric value that is used when there is a need to access a particular record.

If an ID is mentioned in a URI then it is an id-based URI otherwise a directory-based URI.

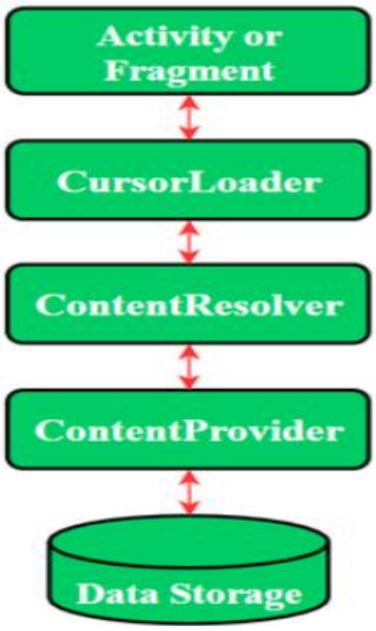
Operations in Content Provider

Four fundamental operations are possible in Content Provider namely **Create**, **Read**, **Update**, and **Delete**. These operations are often termed as **CRUD operations**.

- **Create:** Operation to create data in a content provider.
- **Read:** Used to fetch data from a content provider.
- **Update:** To modify existing data.
- **Delete:** To remove existing data from the storage.

Working of the Content Provider

UI components of android applications like Activity and Fragments use an object **CursorLoader** to send query requests to **ContentResolver.** The ContentResolver object sends requests (like create, read, update, and delete) to the **ContentProvider** as a client. After receiving a request, ContentProvider process it and returns the desired result.



Creating a Content Provider

Following are the steps which are essential to follow in order to create a Content Provider:

- Create a class in the same directory where the that **MainActivity** file resides and this class must extend the ContentProvider base class.
- To access the content, define a content provider URI address.
- Create a database to store the application data.
- Implement the **six abstract methods** of ContentProvider class.
- Register the content provider in **AndroidManifest.xml** file using **<provider> tag**.

query()

A method that accepts arguments and fetches the data from the desired table. Data is retired as a cursor object.

insert()

To insert a new row in the database of the content provider. It returns the content URI of the inserted row.

update()

This method is used to update the fields of an existing row. It returns the number of rows updated.

delete()

This method is used to delete the existing rows. It returns the number of rows deleted.

getType()

This method returns the Multipurpose Internet Mail Extension(MIME) type of data to the given Content URI.

onCreate()

As the content provider is created, the android system calls this method immediately to initialise the provider.

SQLite Database

Features

- Offline Applications
- Permanent Storage
- SQLite Tools
- Small Size around 0.5 MB
- Entire Database in a single file
- Server less applications

Column Data Type

- NULL – null value
- INTEGER - signed integer, stored in 1, 2, 3, 4, 6, or 8 bytes depending on the magnitude of the value
- REAL - a floating point value, 8-byte IEEE floating point number.
- TEXT - text string, stored using the database encoding (UTF-8, UTF-16BE or UTF-16LE).
- BLOB. The value is a blob of data, stored exactly as it was input.

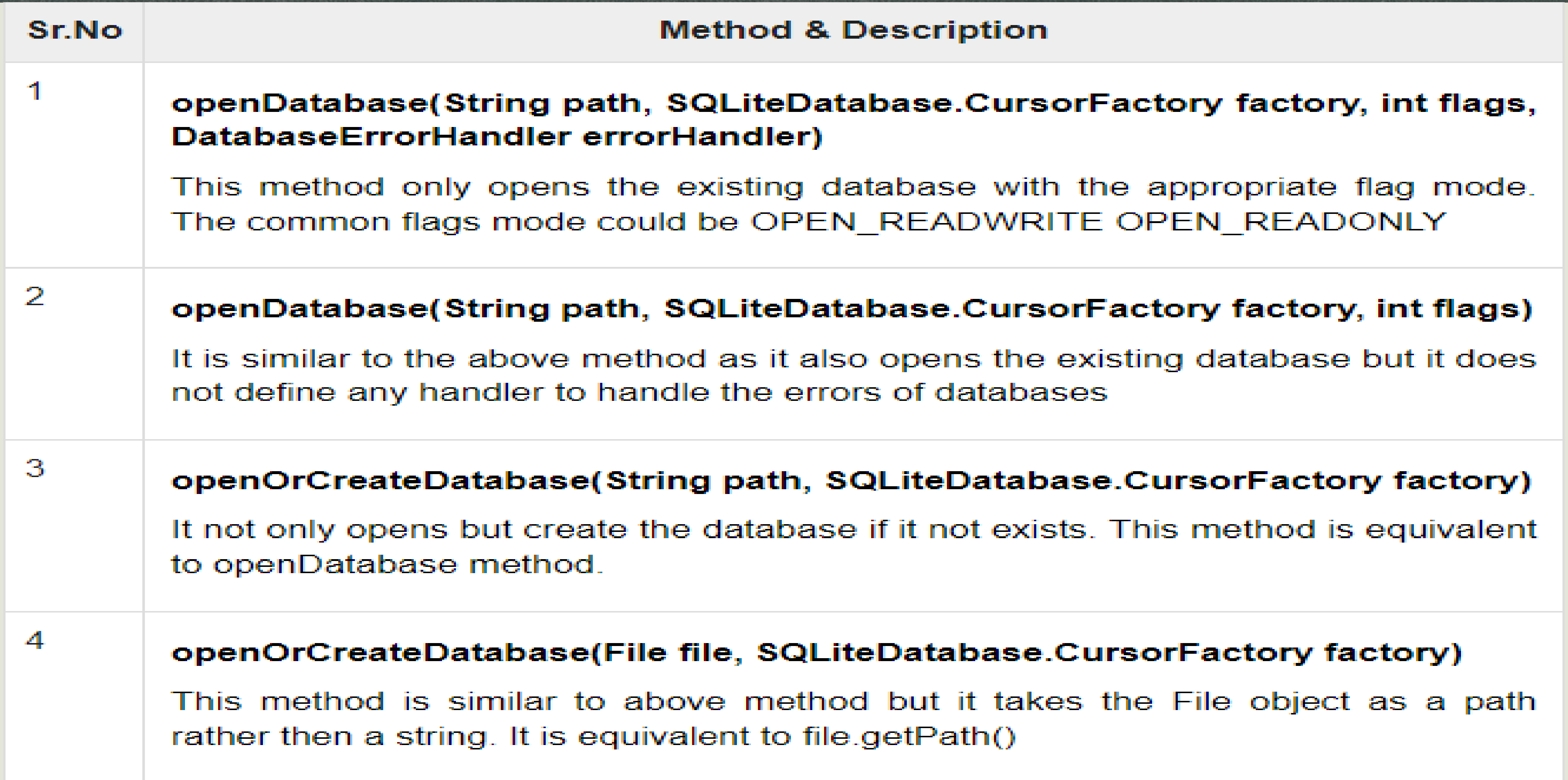
SQLite Classes

- SQLiteCloseable - An object created from a SQLiteDatabase that can be closed.
- SQLiteCursor - A Cursor implementation that exposes results from a query on a SQLiteDatabase.
- SQLiteDatabase - Exposes methods to manage a SQLite database.
- SQLiteOpenHelper - A helper class to manage database creation and version management.
- SQLiteProgram - A base class for compiled SQLite programs.
- SQLiteQuery - A SQLite program that represents a query that reads the resulting rows into a CursorWindow.
- SQLiteQueryBuilder - a convenience class that helps build SQL queries to be sent to SQLiteDatabase objects.
- SQLiteStatement - A pre-compiled statement against a SQLiteDatabase that can be reused.

SQLite Database

- Contains the methods for: creating, opening, closing, inserting, updating, deleting and quering an SQLite database
- These methods are similar to JDBC but more method oriented than what we see with JDBC
- SQLiteDatabase db = openOrCreateDatabase ("my_sqlite_database.db" , MODE_PRIVATE , null);

SQLite Database



Sr.No	Method & Description
1	**openDatabase(String path, SQLiteDatabase.CursorFactory factory, int flags, DatabaseErrorHandler errorHandler)** This method only opens the existing database with the appropriate flag mode. The common flags mode could be OPEN_READWRITE OPEN_READONLY
2	**openDatabase(String path, SQLiteDatabase.CursorFactory factory, int flags)** It is similar to the above method as it also opens the existing database but it does not define any handler to handle the errors of databases
3	**openOrCreateDatabase(String path, SQLiteDatabase.CursorFactory factory)** It not only opens but create the database if it not exists. This method is equivalent to openDatabase method.
4	**openOrCreateDatabase(File file, SQLiteDatabase.CursorFactory factory)** This method is similar to above method but it takes the File object as a path rather then a string. It is equivalent to file.getPath()

Defining a Schema or Contract

- Schema: a formal declaration of how the database is organized. The schema is reflected in the SQL statements that you use to create your database.
- A contract class is a container for constants that define names for URIs, tables, and columns. The contract class allows you to use the same constants across all the other classes in the same package. This lets you change a column name in one place and have it propagate throughout your code.

Defining a Schema or Contract

- public final class FeedReaderContract {
- private FeedReaderContract() {}
- public static class FeedEntry implements BaseColumns {
-         public static final String TABLE_NAME = "entry";
-         public static final String COLUMN_NAME_TITLE = "title";
-         public static final String COLUMN_NAME_SUBTITLE = "subtitle";
-     }
- }

Creating Database using an SQL helper

- The SQLiteOpenHelper class contains a useful set of APIs for managing your database.
- When you use this class to obtain references to your database, the system performs the potentially long-running operations of creating and updating the database only when needed and not during app startup.
- All you need to do is call getWritableDatabase() or getReadableDatabase().
- Ex: public class FeedReaderDbHelper extends SQLiteOpenHelper {}
- To access your database, instantiate your subclass of SQLiteOpenHelper:
- FeedReaderDbHelper dbHelper = new FeedReaderDbHelper(getContext());

Inserting to the Database

- The first argument for insert() is simply the table name.
- The second argument tells the framework what to do in the event that the ContentValues is empty (i.e., you did not put any values). If you specify the name of a column, the framework inserts a row and sets the value of that column to null. If you specify null, like in this code sample, the framework does not insert a row when there are no values.
- The insert() methods returns the ID for the newly created row, or it will return -1 if there was an error inserting the data.

Inserting to the Database

- SQLiteDatabase db = dbHelper.getWritableDatabase();
- ContentValues values = new ContentValues();
- values.put(FeedEntry.COLUMN_NAME_TITLE, title);
- values.put(FeedEntry.COLUMN_NAME_SUBTITLE, subtitle);
- long newRowId = db.insert(FeedEntry.TABLE_NAME, null, values);

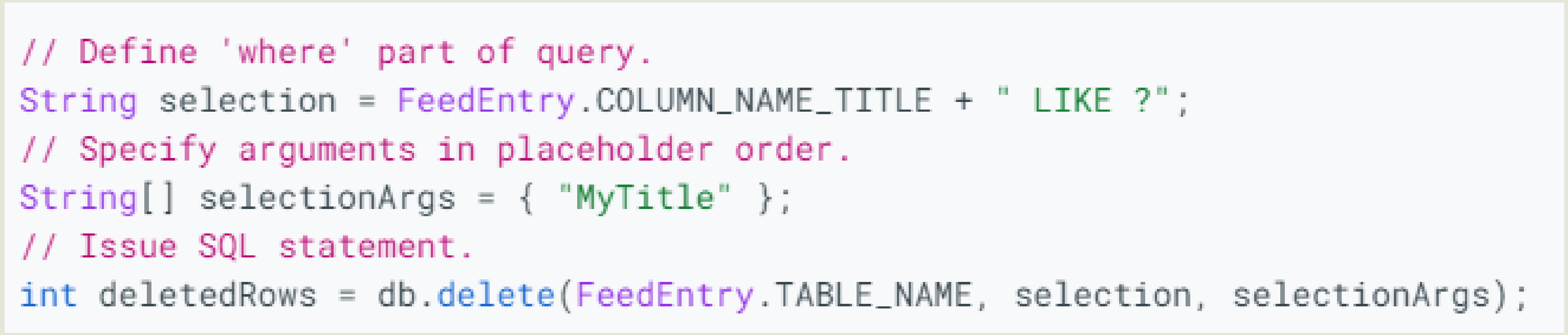
Read information from database

- To read from a database, use the query() method, passing it your selection criteria and desired columns. The method combines elements of insert() and update(), except the column list defines the data you want to fetch , rather than the data to insert. The results of the query are returned to you in a Cursor object.
- To look at a row in the cursor, use one of the Cursor move methods, which you must always call before you begin reading values.
- For each row, you can read a column's value by calling one of the Cursor get methods, such as getString() or getLong().
- For each of the get methods, you must pass the index position of the column you desire

```
SQLiteDatabase db = dbHelper.getReadableDatabase();

// Define a projection that specifies which columns from the database
// you will actually use after this query.
String[] projection = {
    BaseColumns._ID,
    FeedEntry.COLUMN_NAME_TITLE,
    FeedEntry.COLUMN_NAME_SUBTITLE
    };

// Filter results WHERE "title" = 'My Title'
String selection = FeedEntry.COLUMN_NAME_TITLE + " = ?";
String[] selectionArgs = { "My Title" };

// How you want the results sorted in the resulting Cursor
String sortOrder =
    FeedEntry.COLUMN_NAME_SUBTITLE + " DESC";

Cursor cursor = db.query(
    FeedEntry.TABLE_NAME,   // The table to query
    projection,             // The array of columns to return (pass null to get all)
    selection,              // The columns for the WHERE clause
    selectionArgs,          // The values for the WHERE clause
    null,                   // don't group the rows
    null,                   // don't filter by row groups
    sortOrder               // The sort order
    );
```

Delete information from a database

- To delete rows from a table, you need to provide selection criteria that identify the rows to the delete() method.
- The mechanism works the same as the selection arguments to the query() method
- The return value for the delete() method indicates the number of rows that were deleted from the database.

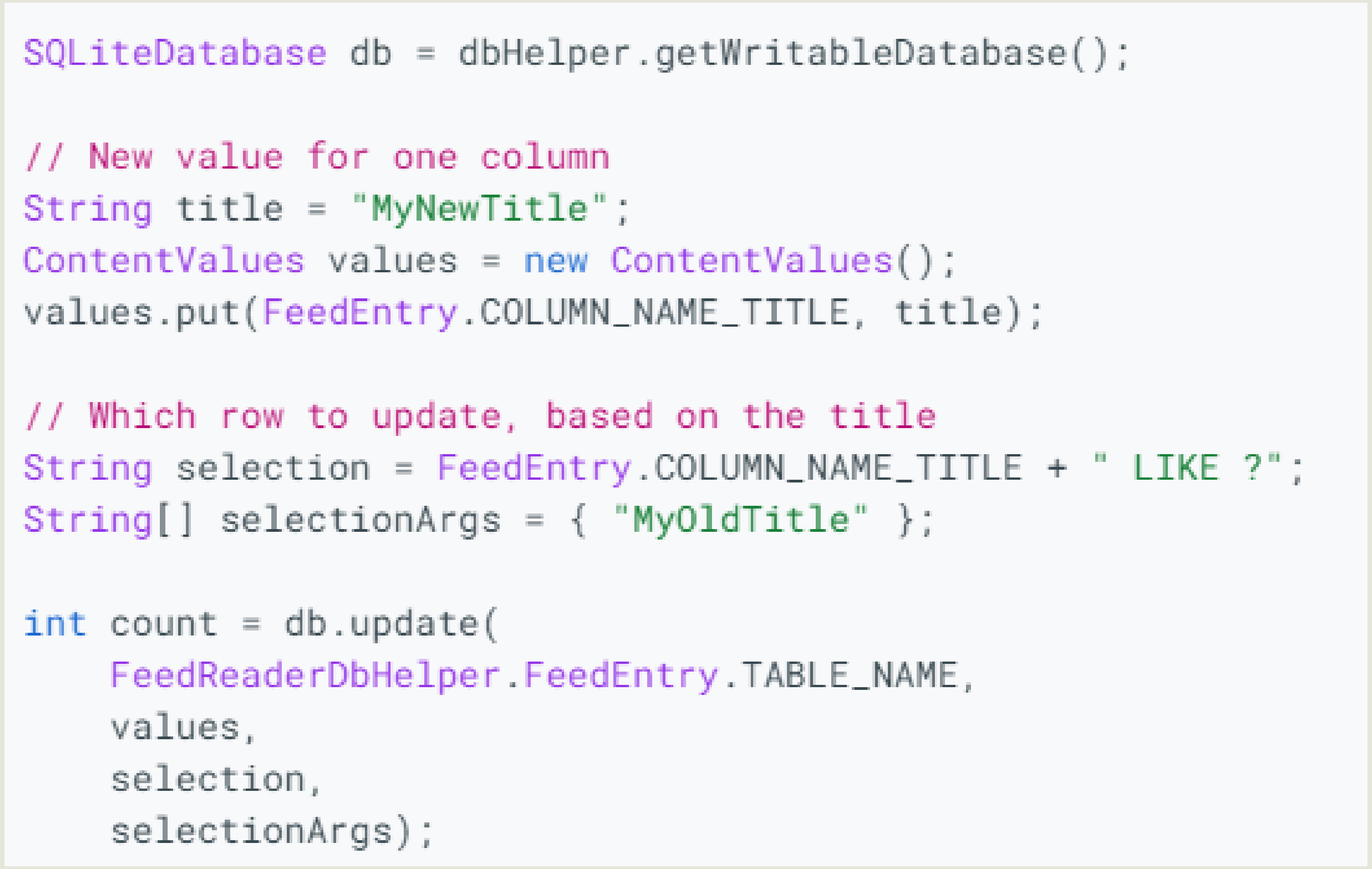


```
// Define 'where' part of query.
String selection = FeedEntry.COLUMN_NAME_TITLE + " LIKE ?";
// Specify arguments in placeholder order.
String[] selectionArgs = { "MyTitle" };
// Issue SQL statement.
int deletedRows = db.delete(FeedEntry.TABLE_NAME, selection, selectionArgs);
```

Update a database

- When you need to modify a subset of your database values, use the update() method.
- Updating the table combines the ContentValues syntax of insert() with the WHERE syntax of delete().
- The return value of the update() method is the number of rows affected in the database.

Update a database



```java
SQLiteDatabase db = dbHelper.getWritableDatabase();

// New value for one column
String title = "MyNewTitle";
ContentValues values = new ContentValues();
values.put(FeedEntry.COLUMN_NAME_TITLE, title);

// Which row to update, based on the title
String selection = FeedEntry.COLUMN_NAME_TITLE + " LIKE ?";
String[] selectionArgs = { "MyOldTitle" };

int count = db.update(
    FeedReaderDbHelper.FeedEntry.TABLE_NAME,
    values,
    selection,
    selectionArgs);
```