# Unit-3

**Intents and Broadcast Receivers:-Introducing Intents, Creating Intent Filters and** Broadcast Receivers. Creating and using Databases (SQLite)

# INTENTS

# INTENTS

- An Android **Intent** is an abstract description of an operation to be performed
- Intents are used to call another activity from current activity
- The another activity may be inside of the project or outside of the project .
- An activity is launched (or given something new to do) by passing an Intent object to Context.**startActivity**() or Activity.**startActivityForResult**().
- We can also **pass / return the data** from one activity to the another activity using Intents. Intents are used to navigate from one activity to another.
- Activities in Android can be invoked by any application running on the device using Intents.

## Example:

```
Intent i=new Intent(current_activity, required_activity.class);
```

**parameters:** current Activity's Context and the class of the Activity to launch

```
i.putExtra("name",value)
startActivity(i);
```

# INTENTS

- An Android **Intent** is an abstract description of an operation to be performed
- Intents are used to call another activity from current activity
- The another activity may be inside of the project or outside of the project .
- An activity is launched (or given something new to do) by passing an Intent object to Context.**startActivity**() or Activity.**startActivityForResult**().
- We can also **pass / return the data** from one activity to the another activity using Intents. Intents are used to navigate from one activity to another.
- Activities in Android can be invoked by any application running on the device using Intents.

## Example:

```
Intent i=new Intent(current_activity, required_activity.class);
```

**parameters:** current Activity's Context and the class of the Activity to launch

```
i.putExtra("name",value)
startActivity(i);
```

# Types of Intents

**Intents are two types:**

1. **Implicit Intent**
2. **Explicit Intent**

- Explicitly start a particular Service or Activity using its class name. (using Explicit Intents)
- Start an Activity or Service to perform an action with (or on) a particular piece of data. (using Implicit Intents)
- Broadcast that an event has occurred.
- You can use Intents to support interaction among any of the application components installed on an Android device, no matter which application they're a part of.
- Android broadcasts Intents to announce system events, such as changes in Internet connectivity or battery charge levels.

# Passing data to activity using Intents

- It is possible to pass data to an activity using Intents.
- Data can be placed in intent in two ways:
  - **By using Intent's putExtra( ) method.**

**Ex:**

```
//---use putExtra() to add new name/value pairs---
i.putExtra("str1", "This is a string");
i.putExtra("age1", 25);
```

  - **By using putExtras( ) method.** (a Bundle object containing data is attached to the Intent)

**Ex:**

```
//---use a Bundle object to add new name/values pairs---
Bundle extras = new Bundle();
extras.putString("str2", "This is another string");
extras.putInt("age2", 35);
//---attach the Bundle object to the Intent object---
i.putExtras(extras);
```

# Getting data passed using Intent

- String str= **getIntent().getStringExtra**("str1"); → gets the string value set using the putExtra() method.
- int i=getIntent().getIntExtra("age1", 0);
- **To retrieve Bundle object:**

```
//---get the Bundle object passed in---
Bundle bundle = getIntent().getExtras();
String str=bundle.getString("str2");
int i=bundle.getInt("age2");
```

## passing and retrieving data to acvity using setData( ) and getData( )

- Use the setData() method to set the data on which an Intent object is going to operate.

**Ex:**

```
i.setData(Uri.parse("Something passed back to main activity"));
```

- Use the getData( ) method to get the data which is passed using Intent.

**Ex:**

```
String str=i.getData( ).toString( );
```

# Returning data from activity using Intents

- To call an activity and wait for a result to be returned from it, you need to use the startActivityForResult() method.

**EX:**

**startActivityForResult(new Intent**("net.learn2develop.SecondActivity"),
request_Code);

parameter 1 → Intent object

Parameter 2 → The request code is simply an integer value that identifies an activity you are calling.

✓ onActivityResult( ) is called whenever the Intent returned a value.

```
public void onActivityResult(int requestCode, int resultCode, Intent data){
if (requestCode == request_Code) {
if (resultCode == RESULT_OK) {
Toast.makeText(this,data.getData().toString(),Toast.LENGTH_SHORT).show();
}}}
```



# Broadcasting events with Intents

- To broadcast events, construct the Intent you want to broadcast and call **sendBroadcast( ) method** to send it.
- Set the action, data, and category of your Intent in a way that lets Broadcast Receivers accurately determine their interest.

**Where:**

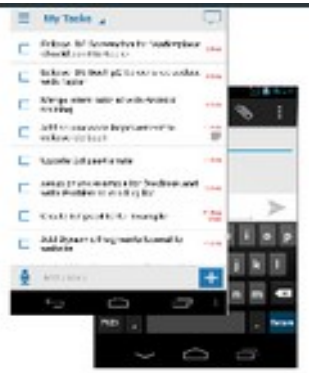
- the Intent ***action*** *string is used to identify the event being* broadcast
- action strings are constructed using the same form as Java package names.
- If you want to include data within the Intent, you can specify a URI using the Intent's data property or you can use extras.
- we can restrict the broadcasts with permissions on sender and receiver.

**Ex:**

```
Intent intent = new Intent();
intent.setAction("com.example.broadcast.MY_NOTIFICATION");
intent.putExtra("data","Notice me");
sendBroadcast(intent);
```

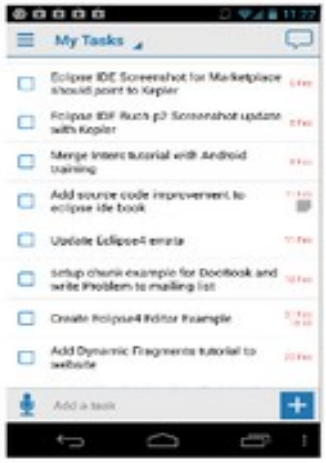
# Intents


Intent resolution by the Android system



One activity is started


Intent + resultCode provided by called activity




onActivityResult(requestCode, resultCode, intent)

RequestCode provided by Android to identify which activity

# Intent Filters

- An intent filter specifies the types of intents to which an activity, service, or **broadcast receiver** can respond to by declaring the capabilities of a component.
- Android components register intent filters either statically in the **AndroidManifest.**xml or in case of a **broadcast receiver** also dynamically via code
- how does the Android system identify the components which can react to a certain intent?

  Ans: A component can register itself via an *intent filter* for a specific action and specific data.

- An intent filter is defined by its category, action and data filters. It can also contain additional meta-data.
- If an intent is sent to the Android system, the Android platform runs a receiver determination. It uses the data included in the intent. If several components have registered for the same intent filter, the user can decide which component should be started.

# Intent Filters

- Android OS uses filters to pinpoint the set of Activities, Services, and Broadcast receivers that can handle the Intent with help of specified set of action, categories, data scheme associated with an Intent.
- You can register your Android components via intent filters for certain events. If a component does not define one, it can only be called by explicit intents.
- You will use **<intent-filter>** element in the manifest file to list down actions, categories and data types associated with any activity, service, or broadcast receiver.
- The **<data>** element specifies the data type expected by the activity to be called.
- Context.startActivity(), Context.startService(), Context.sendBroadcast() methods are used for delivering intents to Activity, Service and BroadcastReceiver components.

# Intent Filters

- Before invoking an activity, Android checks:

  1. A filter must contain at least one <action> element, otherwise it will block all intents. If more than one actions are mentioned then Android tries to match one of the mentioned actions before invoking the activity.

  2. A filter <intent-filter> may list zero, one or more than one categories. if there is no category mentioned then Android always pass this test but if more than one categories are mentioned then for an intent to pass the category test, every category in the Intent object must match a category in the filter.

  3. Each <data> element can specify a URI and a data type (MIME media type). There are separate attributes like **scheme, host, port**, and **path** for each part of the URI. An Intent object that contains both a URI and a data type passes the data type part of the test only if its type matches a type listed in the filter.

# Broadcasting Intents

- The Android system automatically sends broadcasts when various system events occur, such as when the system switches in and out of airplane mode. The system sends these broadcasts to all apps that are subscribed to receive the event.
- The table below lists the standard system broadcast intents that your app can receive in Android 11 (API level 30).
-

| | Constant | Intent Action details |
|---|---|---|
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

# Saving State & User Preferences

# Persisting state

- onSaveInstanceState( )
- onRestoreInstanceState( )

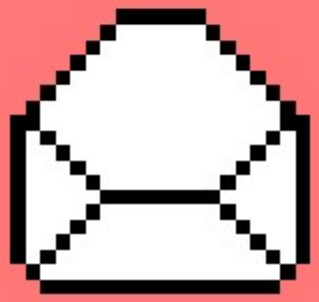
- When Activity's onSaveInstanceState( ) is called. Activity will automatically collect View's State from every single View in the View hierachy. Please note that only View that is implemented View State Saving/Restoring internally that could be collected the data from.

- Once onRestoreInstanceState( ) is called, Activity will send those collected data back to the View in the View hierachy that provides the same android:id as it is collected from one by one.

•Although those View's state are automatically saved but the Activity's member variables are not. They will be destroyed along with Activity. You have to manually save in a Bundle object andrestorethem through onSaveInstanceState and onRestoreInstanceState method.

# Saving and Restoring Instance State

Activity State Saving



Instance States

View States


# Activity State Restoring

Activity State Restoring

Instance States

# Data Persistence

- Using SharedPreferences, Preferences
- Reading & Writing files in internal and external storage
- Creating and using SQLite database

SharedPreferences object:

- Used to save application data
- Writes data in (key, value) pairs to xml file
- use getSharedPreferences(prefname, mode) method.

  **mode** → MODE_PRIVATE constant indicates that the preference file can only be opened by the application that created it.
- Use edit( ) method of SharedPreferences class
- Use putFloat( ) and putString( ) methods of SharedPreferences. Editor class and commit( ) method for commiting changes.
- Information saved in the SharedPreferences obj is visible to all activities in the appl. Where as Preferences object data is visible to that activity only.

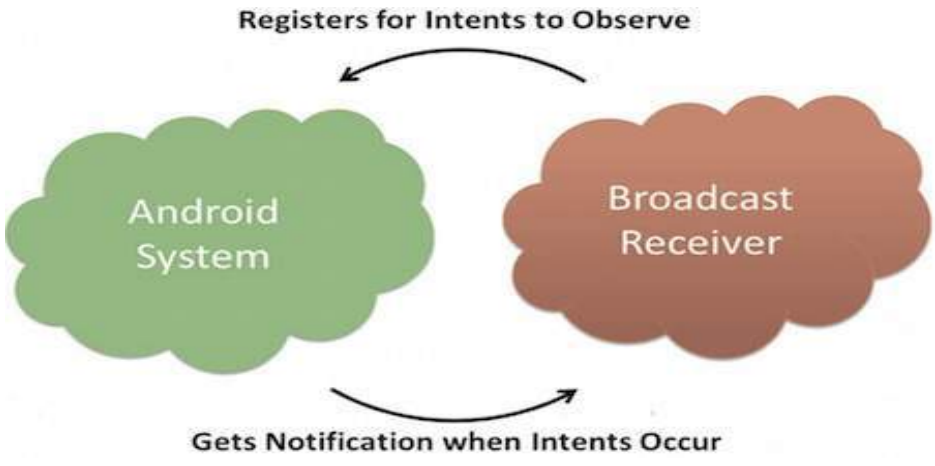
# getSharedPreferences modes



# Broadcast Receivers

# Broadcast Receivers in Android

- Broadcast Receivers simply respond to broadcast messages from other applications or from the system itself. These messages are sometime called events or intents.
- A broadcast receiver, or a receiver for short, is an application component that listens to a certain **intent broadcast**, similar to Java listeners that listen to events.
- Example: Once download is completed, app can initiate a broadcast to let other apps know that data download is completed and is available to use it.
- Generally, we use Intents to deliver broadcast events to other apps and Broadcast Receivers use status bar notifications to let the user know that broadcast event occurs.
- In android, Broadcast Receiver is implemented as a subclass of **BroadcastReceiver** and each broadcast is delivered as an Intent object.
- We can register an app to receive only a few broadcast messages based on our requirements. When a new broadcast received, the system will check for specified broadcasts have subscribed or not based on that it will routes the broadcasts to the apps.
- Using a Broadcast Receiver, applications can register for a particular event. Once the event occurs, the system will notify all the registered applications.



# Broadcast Receivers in Android

## Examples:

1. For instance, a Broadcast receiver triggers ***battery Low notification that you see on your mobile screen***.
2. Other instances caused by a Broadcast Receiver are ***new friend notifications, new friend feeds, new message*** etc. on your Facebook app.
3. In fact, you see broadcast receivers at work all the time. **Notifications like *incoming messages*, *WiFi Activated/Deactivated message*** etc. are all real-time announcements of what is happening in the Android system and the applications.

# Broadcast Receivers in Android

**Steps in using Broadcast Receiver:**

1. Creating the Broadcast Receiver.
2. Registering Broadcast Receiver

- To create a receiver, you must extend the **android.content.BroadcastReceiver class or** one of its subclasses.

**1) Creating broadcast receiver:**

**Broadcast receiver class:**

```
public class MyReceiver extends BroadcastReceiver {
@Override
public void onReceive(Context context, Intent intent) {
// Implement code here to be performed when  broadcast is detected
}
```

**Example:**

```
public class MyReceiver extends BroadcastReceiver {
public void onReceive(Context context, Intent intent) {
    Toast.makeText(context, "Intent Detected.", Toast.LENGTH_LONG).show();
  }}
```

- In your class, you must provide an implementation for the **onReceive method, which gets called when an intent for which the receiver is registered is** broadcast.

# Broadcast Receivers in Android

## onReceive() method:

- In order for the notification to be sent, an onReceive() method has to be implemented. Whenever the event for which the receiver is registered occurs, onReceive() is called. For instance, in case of battery low notification, the receiver is registered to Intent.ACTION_BATTERY_LOW event. As soon as the battery level falls below the defined level, this onReceive() method is called.

public void ***onReceive(Context context, Intent intent)*** {

// Implement code here to be performed when broadcast is detected

}

- Following are the two arguments of the onReceive() method:

  **Context:** This is used to access additional information, or to start services or activities.

  **Intent:** The Intent object is used to register the receiver.

# Broadcast Receivers in Android

## 2) Registering Broadcast Receiver

There are two ways to register a Broadcast Receiver; one is Static and the other Dynamic.

1) **Static:** Use <receiver> tag in your Manifest file. (AndroidManifest.xml)

2) **Dynamic:** Use Context.registerReceiver () method to dynamically register an instance.

**Static registration:**

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

# Registering Receiver dynamically

**Dynamic registration:**

```
IntentFilter filter = new IntentFilter("com.example.Broadcast");
MyReceiver receiver = new MyReceiver();
registerReceiver(receiver, filter);
```

**Unregistering receiver:**

```
unregisterReceiver(receiver);
```

# System events defined in the android.content.Intent class

| Event | description |
|---|---|
| android.intent.action.BATTERY_CHANGED | Sticky broadcast containing the charging state, level, and other information about the battery. |
| android.intent.action.BATTERY_LOW | Indicates low battery condition on the device. |
| android.intent.action.BATTERY_OKAY | Indicates the battery is now okay after being low. |
| android.intent.action.BOOT_COMPLETED | This is broadcast once, after the system has finished booting. |
| android.intent.action.BUG_REPORT | Show activity for reporting a bug. |
| android.intent.action.CALL | Perform a call to someone specified by the data. |
| android.intent.action.CALL_BUTTON | The user pressed the "call" button to go to the dialer or other appropriate UI for placing a call. |
| android.intent.action.DATE_CHANGED | The date has changed. |
| android.intent.action.REBOOT | Have the device reboot. |

# Broadcast Receivers in Android

- **Registering broadcast receivers:**
- Static Registration → in AndroidManifest.xml file
- Dynamic Registration → using Context.registerReceiver() in onCreate():

registerReceiver(new Receiver1(), new IntentFilter(CUSTOM_INTENT));

**Sending broadcasts types:**

1. **Sending broadcast intent to interested receivers.**

void sendBroadcast(Intent intent)

2. **To interested receivers having specific permissions.**

void sendBroadcast(Intent intent, String receiverPermission)

Ex: **sendBroadcast**(new Intent(CUSTOM_INTENT),android.Manifest.permission.VIBRATE);

- When a matching broadcast is detected, the onReceive() method of the broadcast receiver is called, at which point the method has 5 seconds within which to perform any necessary tasks before returning.
- It is important to keep in mind that some system broadcast intents can only be detected by a broadcast receiver if it is registered **in code rather than in the manifest file.**



# Ordered Broadcast

- In the event that return results are required, it is necessary to use the **sendOrderedBroadcast()** method **instead of sendBroadcast().**
- With ordered broadcasts programmer has the control over the order in which a receiver can be executed.
- You can prioritize the order by giving priority attribute in the intent-filter tag of receiver tag in manifest file. More the priority number, higher the priority.

# Sticky Broadcast

- sticky broadcasts mechanism works only with dynamic receivers. For normal receivers it works just like a normal broadcast.
- The broadcast that will stick with android, and will be re-delivered or re-broadcasted to the future requests from any dynamically registered broadcast receivers.



# Intent Filters

- An intent filter ***specifies the types of intents to which an activity, service, or broadcast receiver can respond to*** by declaring the capabilities of a component.
- Android components register intent filters either statically in the **AndroidManifest.**xml or in case of a **broadcast receiver** also dynamically via code.



# Broadcasting events using Intents

- You can also use Intents to broadcast messages *between components via the* ***sendBroadcast*** *method.*
- As a system-level message-passing mechanism, Intents are capable of sending structured messages across process boundaries. As a result, you can implement Broadcast Receivers to listen for, and respond to, these Broadcast Intents within your applications.
- Broadcast Intents are used to notify applications of system or application events, extending the event-driven programming model between applications.
- Android uses Broadcast Intents extensively to broadcast system events, such as changes in network connectivity, docking state, and incoming calls.

**Broadcasting Events:**

Within your application, construct the Intent you want to broadcast and call **sendBroadcast( )** to send it.

**or by using: sendOrderedBroadcast( ), sendStickyBroadcast( )**

# SQLite

# SQLite

- ✓ SQLite is a very light weight database which comes with Android OS.
- ✓ SQLite's memory footprint starts at about 50 kilobyte it's remains low even for bigger projects with more complex data structures

  (at about a few hundred kilobytes).

  /data/data/[package]/databases/[db-name] is the path of Sqlite db.
- ✓ **All databases are private, accessible only by the application that created them.**
- Each SQLite database is an integrated part of the application that created it. This reduces external dependencies, minimizes latency, and simplifies transaction locking and synchronization.

**Features:**

- SQLite is serverless
- Ease of use.
- Lightweight and powerful
- Low memory footprint
- SQLite stores data in one database file
- SQLite offers only a few data types
- SQLite uses manifest typing instead of static types
- SQLite has no fixed column length
- SQLite uses cross-platform database files

# SQLite Datatypes

- SQLite differs from many conventional database engines by **loosely typing each column**, meaning that column values are not required to conform to a single type; instead, **each value is typed individually** in each row. As a result, type checking isn't necessary when assigning or extracting values from each column within a row.
- **Manifest typing** means that a data type is a property of a value stored in a column, not the property of the column in which the value is stored. SQLite uses manifest typing to store values of any type in a column.
- If you come from other database systems such as MySQL and PostgreSQL, you notice that they use *static typing*. It means when you declare a column with a specific data type, that column can store only data of the declared data type.
- Different from other database systems, **SQLite uses *dynamic type system*.** In other words, a value stored in a column determines its data type, not the column's data type.
- In addition, you don't have to declare a specific data type for a column when you create a table. In case you declare a column with the integer data type, you can store any kind of data types such as text and BLOB, SQLite will not complain about this.
- To view db data use tool 'DB Browser for SQLite'

# SQLite Storage classes/Datatypes

- The following table illustrates 5 storage classes in SQLite:

| Storage Class | Meaning |
|---|---|
| NULL | NULL values mean missing information or unknown. |
| INTEGER | Integer values are whole numbers (either positive or negative). An integer can have **variable sizes such as 1, 2,3, 4, or 8 bytes.** |
| REAL | Real values are real numbers with decimal values that use **8-byte floats.** |
| TEXT | TEXT is used to store character data. The maximum length of TEXT is **unlimited.** SQLite supports various character encodings. |
| BLOB | BLOB stands for a **binary large object** that can store any kind of data. The maximum size of BLOB is, theoretically, **unlimited.** |

# ContentValues obj

- ContentValues are used to insert new rows into tables. Each ContentValues object represents a single table row as a map of column names to values.
- Database queries are returned as **Cursor** objects.
- Rather than extracting and returning a copy of the result values, Cursors are pointers to the **result set** within the underlying data.

# Cursor class

The **Cursor** class includes a number of navigation functions.

- moveToFirst — Moves the cursor to the fi rst row in the query result
- moveToNext — Moves the cursor to the next row
- moveToPrevious — Moves the cursor to the previous row
- getCount — Returns the number of rows in the result set
- getColumnIndexOrThrow — Returns the zero-based index for the column with the specified name (throwing an exception if no column exists with that name)
- getColumnName — Returns the name of the specifi ed column index
- getColumnNames — Returns a string array of all the column names in the current Cursor
- moveToPosition — Moves the cursor to the specified row
- getPosition — Returns the current cursor position

# SQLiteOpenHelper

- A helper class to manage database creation and version management.
- SQLiteOpenHelper is an abstract class used to implement the best practice pattern for **creating, opening, and upgrading databases**.
- By implementing an SQLiteOpenHelper, you hide the logic used to decide if a database needs to be created or upgraded before it's opened, as well as ensure that each operation is completed efficiently.
- To access a database using the **SQLiteOpenHelper**, call **getWritableDatabase** or **getReadableDatabase** to open and obtain a writable or read-only instance of the underlying database, respectively.
- If the database doesn't exist, the helper executes its onCreate handler. If the database version has changed, the onUpgrade handler will fire.

# SQLiteOpenHelper

- SQLiteOpenHelper (Context context, String name, SQLiteDatabase.CursorFactory factory, int version)

- Create a helper object to create, open, and/or manage a database. This method always returns very quickly. The database is not actually created or opened until one of getWritableDatabase() or getReadableDatabase() is called.

**context** → Context to use to open or create the database
**name** → name of the database file, or null for an in-memory database
**factory** → to use for creating cursor objects, or null for the default
**version** → number of the database (starting at 1)

# SQLiteOpenHelper

- When a database has been successfully opened, the SQLiteOpenHelper will cache it, so you can (and should) use these methods each time you query or perform a transaction on the database, rather than caching the open database within your application.

**Opening or creating DB:**

```
SQLiteDatabase db = context.openOrCreateDatabase(DATABASE_NAME,
Context.MODE_PRIVATE,null);
```

**Note:**

- It's good practice to defer creating and opening databases until they're needed, and to cache database instances after they're successfully opened to limit the associated efficiency costs.

# SQLiteOpenHelper

- **SQLiteDatabase getReadableDatabase ( )**
  - ✓ Create and/or open a database
  - ✓ SQLiteDatabase is returned which is valid until getWritableDatabase() or close() is called.
  - ✓ Throws SQLiteException if the database cannot be opened.

**SQLiteDatabase getWritableDatabase( )**

- Create and/or open a database that will be used for reading and writing.
- The first time this is called, the database will be opened and onCreate(SQLiteDatabase), onUpgrade(SQLiteDatabase, int, int) and/or onOpen(SQLiteDatabase) will be called.
- Once opened successfully, the database is cached, so you can call this method every time you need to write to the database.
- Returns SQLiteDatabase which is valid until close( ) is called.

# SQLiteOpenHelper

| Method | Description |
|---|---|
| void **close()** | Close any open database object. |
| String **getDatabaseName()** | Return the name of the SQLite database being opened, as given to the constructor. |
| SQLiteDatabase **getReadableDatabase()** | Create and/or open a database. |
| SQLiteDatabase **getWritableDatabase()** | Create and/or open a database that will be used for reading and writing. |
| abstract void **onCreate**(SQLiteDatabase db) | Called when the database is created for the first time. |
| void **onOpen**(SQLiteDatabase db) | Called when the database has been opened. |
| void **setWriteAheadLoggingEnabled**(boolean enabled) | Enables or disables the use of **write-ahead logging** for the database. Write-ahead logging cannot be used with read-only databases so the value of this flag is ignored if the database is |

# SQLiteDatabase class

| Method | Description |
|---|---|
| void **beginTransaction**() | Begins a transaction in EXCLUSIVE mode. Transactions can be nested. When the outer transaction is ended all of the work done in that transaction and all of the nested transactions will be committed or rolled back. The changes will be rolled back if any transaction is ended without being marked as clean (by calling setTransactionSuccessful). Otherwise they will be committed. |
| void **beginTransactionNonExclusive()** | Begins a transaction in IMMEDIATE mode. |
| static SQLiteDatabase **create(**SQLiteDatabase.CursorFactory factory) | Create a memory backed SQLite database. |
| int **delete**(String table, String whereClause, String[] whereArgs) | Convenience method for deleting rows in the database. |
| static boolean **deleteDatabase**(File file) | Deletes a database including its journal file and other auxiliary files that may have been created by the database engine. |

# SQLiteDatabase class

| Method | Description |
|---|---|
| void **execSQL**(String sql) | Execute a single SQL statement that is NOT a SELECT or any other SQL statement that returns data. |
| void **execSQL**(String sql, Object[] bindArgs) | Execute a single SQL statement that is NOT a SELECT/INSERT/UPDATE/DELETE. |
| long **getMaximumSize()** | Returns the maximum size the database may grow to. |
| final String **getPath()** | Gets the path to the database file. |
| int **getVersion()** | Gets the database version. |
| long **insert**(String table, String nullColumnHack, ContentValues values) | Convenience method for inserting a row into the database. |
| long **insertOrThrow**(String table, String nullColumnHack, ContentValues values) | Convenience method for inserting a row into the database. |

# SQLiteDatabase class

| Method | Description |
|---|---|
| long **insertWithOnConflict**(String table, String nullColumnHack, ContentValues initialValues, int conflictAlgorithm) | General method for inserting a row into the database. |
| boolean **isOpen()** | Returns true if the database is currently open. |
| boolean **isReadOnly**() | Returns true if the database is opened as read only. |
| static SQLiteDatabase **openDatabase**(String path, SQLiteDatabase.CursorFactory factory, int flags) | Open the database according to the flags OPEN_READWRITE OPEN_READONLY CREATE_IF_NECESSARY and/or NO_LOCALIZED_COLLATORS. |
| static SQLiteDatabase **openDatabase**(String path, SQLiteDatabase.CursorFactory factory, int flags, DatabaseErrorHandler errorHandler) | Open the database according to the flags OPEN_READWRITE OPEN_READONLY CREATE_IF_NECESSARY and/or NO_LOCALIZED_COLLATORS. |

# SQLiteDatabase class

| Method | Description |
|---|---|
| static SQLiteDatabase **openOrCreateDatabase**(File file, SQLiteDatabase.CursorFactory factory) | Equivalent to openDatabase(file.getPath(), factory, CREATE_IF_NECESSARY). |
| static SQLiteDatabase **openOrCreateDatabase**(String path, SQLiteDatabase.CursorFactory factory, DatabaseErrorHandler errorHandler) | Equivalent to openDatabase(path, factory, CREATE_IF_NECESSARY, errorHandler). |
| static SQLiteDatabase **openOrCreateDatabase**(String path, SQLiteDatabase.CursorFactory factory) | Equivalent to openDatabase(path, factory, CREATE_IF_NECESSARY). |
| Cursor **query**(boolean distinct, String table, String[] columns, String selection, String[] selectionArgs, String groupBy, String having, String orderBy, String limit) | Query the given URL, returning a Cursor over the result set. |

# SQLiteDatabase class

| Method | Description |
|---|---|
| Cursor **query**(String table, String[] columns, String selection, String[] selectionArgs, String groupBy, String having, String orderBy, String limit) | Query the given table, returning a Cursor over the result set. |
| Cursor **query**(String table, String[] columns, String selection, String[] selectionArgs, String groupBy, String having, String orderBy) | Query the given table, returning a Cursor over the result set. |
| Cursor **rawQuery**(String sql, String[] selectionArgs, CancellationSignal cancellationSignal) | Runs the provided SQL and returns a Cursor over the result set. |
| Cursor **rawQuery**(String sql, String[] selectionArgs) | Runs the provided SQL and returns a Cursor over the result set. |

# SQLiteDatabase class

| Method | Description |
|---|---|
| long **replace**(String table, String nullColumnHack, ContentValues initialValues) | Convenience method for replacing a row in the database. |
| long **replaceOrThrow**(String table, String nullColumnHack, ContentValues initialValues) | Convenience method for replacing a row in the database. |
| void **setForeignKeyConstraintsEnabled**(boolean enable) | Sets whether foreign key constraints are enabled for the database. |
| long **setMaximumSize**(long numBytes) | Sets the maximum size the database will grow to. |
| void **setVersion**(int version) | Sets the database version. |
| int **update**(String table, ContentValues values, String whereClause, String[] whereArgs) | Convenience method for updating rows in the database. |

# Transaction Example

```
SQLiteDatabase db = getDatabase();
db.beginTransaction();
try {     // insert/update/delete
          // insert/update/delete
          // insert/update/delete
db.setTransactionSuccessful(); }
finally {
  db.endTransaction();
 }
```

# SQLiteOpenHelper