Storage Options

Android provides several options for you to save persistent application data. The solution you choose depends on your specific needs, such as whether the data should be private to your application or accessible to other applications (and the user) and how much space your data requires.

Your data storage options are the following:

Shared Preferences
: Store private primitive data in key-value pairs.

Internal Storage
: Store private data on the device memory.

External Storage
: Store public data on the shared external storage.

SQLite Databases
: Store structured data in a private database.

Network Connection
: Store data on the web with your own network server.

Android provides a way for you to expose even your private data to other applications — with a content provider (/guide/topics/providers/content-providers.html). A content provider is an optional component that exposes read/write access to your application data, subject to whatever restrictions you want to impose. For more information about using content providers, see the Content Providers (/guide/topics/providers/content-providers.html) documentation.

STORAGE QUICKVIEW

- Use Shared Preferences for primitive data
- Use internal device storage for private data
- Use external storage for large data sets that are not private
- Use SQLite databases for structured storage

IN THIS DOCUMENT

Using Shared Preferences
Using the Internal Storage
Using the External Storage
Using Databases
Using a Network Connection

SEE ALSO

Content Providers and Content Resolvers

Using Shared Preferences

The `SharedPreferences` (/reference/android/content/SharedPreferences.html) class provides a general framework that allows you to save and retrieve persistent key-value pairs of primitive data types. You can use `SharedPreferences` (/reference/android/content/SharedPreferences.html) to save any primitive data: booleans, floats, ints, longs, and strings. This data will persist across user sessions (even if your application is killed).

To get a `SharedPreferences` (/reference/android/content/SharedPreferences.html) object for your application, use one of two methods:

- `getSharedPreferences()` - Use this if you need multiple preferences files identified by name, which you specify with the first parameter.
- `getPreferences()` - Use this if you need only one preferences file for your Activity. Because this will be the only preferences file for your Activity, you don't supply a name.

To write values:

1. Call `edit()` to get a `SharedPreferences.Editor`.
2. Add values with methods such as `putBoolean()` and `putString()`.
3. Commit the new values with `commit()`

User Preferences

Shared preferences are not strictly for saving "user preferences," such as what ringtone a user has chosen. If you're interested in creating user preferences for your application, see `PreferenceActivity` (/reference/android/preference/PreferenceActivity.html), which provides an Activity framework for you to create user preferences, which will be automatically persisted (using shared preferences).

To read values, use `SharedPreferences` (/reference/android/content/SharedPreferences.html) methods such as `getBoolean()`

(/reference/android/content/SharedPreferences.html#getBoolean(java.lang.String, boolean)) and getString() (/reference/android/content/SharedPreferences.html#getString(java.lang.String, java.lang.String)).

Here is an example that saves a preference for silent keypress mode in a calculator:

```
public class Calc extends Activity {
    public static final String PREFS_NAME = "MyPrefsFile";

    @Override
    protected void onCreate(Bundle state){
       super.onCreate(state);
       . . .

       // Restore preferences
       SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
       boolean silent = settings.getBoolean("silentMode", false);
       setSilent(silent);
    }

    @Override
    protected void onStop(){
       super.onStop();

      // We need an Editor object to make preference changes.
      // All objects are from android.context.Context
      SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
      SharedPreferences.Editor editor = settings.edit();
      editor.putBoolean("silentMode", mSilentMode);

      // Commit the edits!
      editor.commit();
    }
}
```

Using the Internal Storage

You can save files directly on the device's internal storage. By default, files saved to the internal storage are private to your application and other applications cannot access them (nor can the user). When the user uninstalls your application, these files are removed.

To create and write a private file to the internal storage:

1. Call openFileOutput() with the name of the file and the operating mode. This returns a FileOutputStream.
2. Write to the file with write().
3. Close the stream with close().

For example:

```
String FILENAME = "hello_file";
String string = "hello world!";

FileOutputStream fos = openFileOutput(FILENAME, Context.MODE_PRIVATE);
fos.write(string.getBytes());
fos.close();
```

MODE_PRIVATE (/reference/android/content/Context.html#MODE_PRIVATE) will create the file (or replace a

file of the same name) and make it private to your application. Other modes available are: MODE_APPEND (/reference/android/content/Context.html#MODE_APPEND), MODE_WORLD_READABLE (/reference/android/content/Context.html#MODE_WORLD_READABLE), and MODE_WORLD_WRITEABLE (/reference/android/content/Context.html#MODE_WORLD_WRITEABLE).

To read a file from internal storage:

1. Call openFileInput() and pass it the name of the file to read. This returns a FileInputStream.
2. Read bytes from the file with read().
3. Then close the stream with close().

> **Tip:** If you want to save a static file in your application at compile time, save the file in your project res/raw/ directory. You can open it with openRawResource() (/reference/android/content/res/Resources.html#openRawResource(int)), passing the R.raw.*<filename>* resource ID. This method returns an InputStream (/reference/java/io/InputStream.html) that you can use to read the file (but you cannot write to the original file).

Saving cache files

If you'd like to cache some data, rather than store it persistently, you should use getCacheDir() (/reference/android/content/Context.html#getCacheDir()) to open a File (/reference/java/io/File.html) that represents the internal directory where your application should save temporary cache files.

When the device is low on internal storage space, Android may delete these cache files to recover space. However, you should not rely on the system to clean up these files for you. You should always maintain the cache files yourself and stay within a reasonable limit of space consumed, such as 1MB. When the user uninstalls your application, these files are removed.

Other useful methods

getFilesDir()
: Gets the absolute path to the filesystem directory where your internal files are saved.

getDir()
: Creates (or opens an existing) directory within your internal storage space.

deleteFile()
: Deletes a file saved on the internal storage.

fileList()
: Returns an array of files currently saved by your application.

Using the External Storage

Every Android-compatible device supports a shared "external storage" that you can use to save files. This can be a removable storage media (such as an SD card) or an internal (non-removable) storage. Files saved to the external storage are world-readable and can be modified by the user when they enable USB mass storage to transfer files on a computer.

> **Caution:** External storage can become unavailable if the user mounts the external storage on a computer or removes the media, and there's no security enforced upon files you save to the external storage. All applications can read and write files placed on the external storage and the user can remove them.

Getting access to external storage

In order to read or write files on the external storage, your app must acquire the READ_EXTERNAL_STORAGE (/reference/android/Manifest.permission.html#READ_EXTERNAL_STORAGE) or WRITE_EXTERNAL_STORAGE (/reference/android/Manifest.permission.html#WRITE_EXTERNAL_STORAGE) system permissions. For example:

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

```
    ...
</manifest>
```

If you need to both read and write files, then you need to request only the `WRITE_EXTERNAL_STORAGE` (/reference/android/Manifest.permission.html#WRITE_EXTERNAL_STORAGE) permission, because it implicitly requires read access as well.

> **Note:** Beginning with Android 4.4, these permissions are not required if you're reading or writing only files that are private to your app. For more information, see the section below about saving files that are app-private (#AccessingExtFiles).

Checking media availability

Before you do any work with the external storage, you should always call `getExternalStorageState()` (/reference/android/os/Environment.html#getExternalStorageState()) to check whether the media is available. The media might be mounted to a computer, missing, read-only, or in some other state. For example, here are a couple methods you can use to check the availability:

```
/* Checks if external storage is available for read and write */
public boolean isExternalStorageWritable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state)) {
        return true;
    }
    return false;
}

/* Checks if external storage is available to at least read */
public boolean isExternalStorageReadable() {
    String state = Environment.getExternalStorageState();
    if (Environment.MEDIA_MOUNTED.equals(state) ||
        Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
        return true;
    }
    return false;
}
```

The `getExternalStorageState()` (/reference/android/os/Environment.html#getExternalStorageState()) method returns other states that you might want to check, such as whether the media is being shared (connected to a computer), is missing entirely, has been removed badly, etc. You can use these to notify the user with more information when your application needs to access the media.

Saving files that can be shared with other apps

Generally, new files that the user may acquire through your app should be saved to a "public" location on the device where other apps can access them and the user can easily copy them from the device. When doing so, you should use to one of the shared public directories, such as `Music/`, `Pictures/`, and `Ringtones/`.

To get a `File` (/reference/java/io/File.html) representing the appropriate public directory, call `getExternalStoragePublicDirectory()`

> **Hiding your files from the Media Scanner**
>
> Include an empty file named `.nomedia` in your external files directory (note the dot prefix in the filename). This prevents media scanner from reading your media files and providing them to other apps through the `MediaStore` (/reference/android/provider/MediaStore.html) content provider. However, if your files are truly private to your app, you should save them in an app-private directory (#AccessingExtFiles).

(/reference/android/os/Environment.html#getExternalStoragePublicDirectory(java.lang.String)), passing it the type of directory you want, such as DIRECTORY_MUSIC (/reference/android/os/Environment.html#DIRECTORY_MUSIC), DIRECTORY_PICTURES (/reference/android/os/Environment.html#DIRECTORY_PICTURES), DIRECTORY_RINGTONES (/reference/android/os/Environment.html#DIRECTORY_RINGTONES), or others. By saving your files to the corresponding media-type directory, the system's media scanner can properly categorize your files in the system (for instance, ringtones appear in system settings as ringtones, not as music).

For example, here's a method that creates a directory for a new photo album in the public pictures directory:

```
public File getAlbumStorageDir(String albumName) {
    // Get the directory for the user's public pictures directory.
    File file = new File(Environment.getExternalStoragePublicDirectory(
            Environment.DIRECTORY_PICTURES), albumName);
    if (!file.mkdirs()) {
        Log.e(LOG_TAG, "Directory not created");
    }
    return file;
}
```

Saving files that are app-private

If you are handling files that are not intended for other apps to use (such as graphic textures or sound effects used by only your app), you should use a private storage directory on the external storage by calling getExternalFilesDir() (/reference/android/content/Context.html#getExternalFilesDir(java.lang.String)). This method also takes a `type` argument to specify the type of subdirectory (such as DIRECTORY_MOVIES (/reference/android/os/Environment.html#DIRECTORY_MOVIES)). If you don't need a specific media directory, pass `null` to receive the root directory of your app's private directory.

Beginning with Android 4.4, reading or writing files in your app's private directories does not require the READ_EXTERNAL_STORAGE (/reference/android/Manifest.permission.html#READ_EXTERNAL_STORAGE) or WRITE_EXTERNAL_STORAGE (/reference/android/Manifest.permission.html#WRITE_EXTERNAL_STORAGE) permissions. So you can declare the permission should be requested only on the lower versions of Android by adding the maxSdkVersion (/guide/topics/manifest/uses-permission-element.html#maxSdk) attribute:

```
<manifest ...>
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"
                     android:maxSdkVersion="18" />
    ...
</manifest>
```

> **Note**: When the user uninstalls your application, this directory and all its contents are deleted. Also, the system media scanner does not read files in these directories, so they are not accessible from the MediaStore (/reference/android/provider/MediaStore.html) content provider. As such, you **should not use these directories** for media that ultimately belongs to the user, such as photos captured or edited with your app, or music the user has purchased with your app—those files should be saved in the public directories (#SavingSharedFiles).

Sometimes, a device that has allocated a partition of the internal memory for use as the external storage may also offer an SD card slot. When such a device is running Android 4.3 and lower, the getExternalFilesDir() (/reference/android/content/Context.html#getExternalFilesDir(java.lang.String)) method provides access to only the internal partition and your app cannot read or write to the SD card. Beginning with Android 4.4, however, you can access both locations by calling getExternalFilesDirs() (/reference/android/content/Context.html#getExternalFilesDirs(java.lang.String)), which returns a File (/reference/java/io/File.html) array with entries each location. The first entry in the array is considered the primary external storage and you should use that location unless it's full or unavailable. If you'd like to access both possible locations while also supporting Android 4.3 and lower, use the support library's

(/tools/support-library/index.html) static method, `ContextCompat.getExternalFilesDirs()` (/reference/android/support/v4/content/ContextCompat.html#getExternalFilesDirs(android.content.Context, java.lang.String)). This also returns a `File` (/reference/java/io/File.html) array, but always includes only one entry on Android 4.3 and lower.

> **Caution** Although the directories provided by `getExternalFilesDir()` (/reference/android/content/Context.html#getExternalFilesDir(java.lang.String)) and `getExternalFilesDirs()` (/reference/android/content/Context.html#getExternalFilesDirs(java.lang.String)) are not accessible by the `MediaStore` (/reference/android/provider/MediaStore.html) content provider, other apps with the `READ_EXTERNAL_STORAGE` (/reference/android/Manifest.permission.html#READ_EXTERNAL_STORAGE) permission can access all files on the external storage, including these. If you need to completely restrict access for your files, you should instead write your files to the internal storage (#filesInternal).

Saving cache files

To open a `File` (/reference/java/io/File.html) that represents the external storage directory where you should save cache files, call `getExternalCacheDir()` (/reference/android/content/Context.html#getExternalCacheDir()). If the user uninstalls your application, these files will be automatically deleted.

Similar to `ContextCompat.getExternalFilesDirs()` (/reference/android/support/v4/content/ContextCompat.html#getExternalFilesDirs(android.content.Context, java.lang.String)), mentioned above, you can also access a cache directory on a secondary external storage (if available) by calling `ContextCompat.getExternalCacheDirs()` (/reference/android/support/v4/content/ContextCompat.html#getExternalCacheDirs(android.content.Context)).

> **Tip:** To preserve file space and maintain your app's performance, it's important that you carefully manage your cache files and remove those that aren't needed anymore throughout your app's lifecycle.

Using Databases

Android provides full support for SQLite (http://www.sqlite.org/) databases. Any databases you create will be accessible by name to any class in the application, but not outside the application.

The recommended method to create a new SQLite database is to create a subclass of `SQLiteOpenHelper` (/reference/android/database/sqlite/SQLiteOpenHelper.html) and override the `onCreate()` (/reference/android/database/sqlite/SQLiteOpenHelper.html#onCreate(android.database.sqlite.SQLiteDatabase)) method, in which you can execute a SQLite command to create tables in the database. For example:

```
public class DictionaryOpenHelper extends SQLiteOpenHelper {

    private static final int DATABASE_VERSION = 2;
    private static final String DICTIONARY_TABLE_NAME = "dictionary";
    private static final String DICTIONARY_TABLE_CREATE =
                "CREATE TABLE " + DICTIONARY_TABLE_NAME + " (" +
                KEY_WORD + " TEXT, " +
                KEY_DEFINITION + " TEXT);";

    DictionaryOpenHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(DICTIONARY_TABLE_CREATE);
```

```
    }
}
```

You can then get an instance of your `SQLiteOpenHelper` (/reference/android/database/sqlite/SQLiteOpenHelper.html) implementation using the constructor you've defined. To write to and read from the database, call `getWritableDatabase()` (/reference/android/database/sqlite/SQLiteOpenHelper.html#getWritableDatabase()) and `getReadableDatabase()` (/reference/android/database/sqlite/SQLiteOpenHelper.html#getReadableDatabase()), respectively. These both return a `SQLiteDatabase` (/reference/android/database/sqlite/SQLiteDatabase.html) object that represents the database and provides methods for SQLite operations.

You can execute SQLite queries using the `SQLiteDatabase` (/reference/android/database/sqlite/SQLiteDatabase.html) `query()` (/reference/android/database/sqlite/SQLiteDatabase.html#query(boolean, java.lang.String, java.lang.String[], java.lang.String, java.lang.String[], java.lang.String, java.lang.String, java.lang.String, java.lang.String)) methods, which accept various query parameters, such as the table to query, the projection, selection, columns, grouping, and others. For complex queries, such as those that require column aliases, you should use `SQLiteQueryBuilder` (/reference/android/database/sqlite/SQLiteQueryBuilder.html), which provides several convienent methods for building queries.

> Android does not impose any limitations beyond the standard SQLite concepts. We do recommend including an autoincrement value key field that can be used as a unique ID to quickly find a record. This is not required for private data, but if you implement a content provider (/guide/topics/providers/content-providers.html), you must include a unique ID using the `BaseColumns._ID` (/reference/android/provider/BaseColumns.html#_ID) constant.

Every SQLite query will return a `Cursor` (/reference/android/database/Cursor.html) that points to all the rows found by the query. The `Cursor` (/reference/android/database/Cursor.html) is always the mechanism with which you can navigate results from a database query and read rows and columns.

For sample apps that demonstrate how to use SQLite databases in Android, see the Note Pad (/resources/samples/NotePad/index.html) and Searchable Dictionary (/resources/samples/SearchableDictionary/index.html) applications.

Database debugging

The Android SDK includes a `sqlite3` database tool that allows you to browse table contents, run SQL commands, and perform other useful functions on SQLite databases. See Examining sqlite3 databases from a remote shell (/tools/help/adb.html#sqlite) to learn how to run this tool.

Using a Network Connection

You can use the network (when it's available) to store and retrieve data on your own web-based services. To do network operations, use classes in the following packages:

- `java.net.*`
- `android.net.*`