9.0: Data storage

Contents:

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

Your data storage options include the following:

- Shared preferences: Store private primitive data in key-value pairs. This is covered in the next chapter.
- 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.
- Room persistence library: Part of the Android Architecture Component libraries. Room caches an SQLite database locally, and automatically syncs changes to a network database
- Cloud backup: Back up your app and user data in the cloud.
- Firebase realtime database: Store and sync data with a NoSQL cloud database. Data is synced across all clients in real time, and remains available when your app goes offline.
- Custom data store: Configure the Preference APIs to store preferences in a storage location you provide.

Shared preferences

Using shared preferences is a way to read and write key-value pairs of information persistently to and from a file. Shared Preferences is covered in its own chapter.

> **Note:** The Preference APIs are used to create and store user settings for your app. Although the Preference APIs use shared preferences to store their data, they are not the same thing. You will learn more about settings and preferences in a later chapter.

Files

Android uses a file system that's similar to disk-based file systems on other platforms such as Linux. File-based operations should be familiar to anyone who has used Linux file I/O or the `java.io` package.

All Android devices have two file storage areas: "internal" and "external" storage. These names come from the early days of Android, when most devices offered built-in non-volatile memory (internal storage), plus a removable storage medium such as a micro SD card (external storage).

Today, some devices divide the permanent storage space into "internal" and "external" partitions, so even without a removable storage medium, there are always two storage spaces and the API behavior is the same whether the external storage is removable or not. The following lists summarize the facts about each storage space.

Internal storage	External storage
Always available.	Not always available, because the user can mount the external storage as USB storage and in some cases remove it from the device.
Only your app can access files. Specifically, your app's internal storage directory is specified by your app's package name in a special location of the Android file system. Other apps cannot browse your internal directories and do not have read or write access unless you explicitly set the files to be readable or writable.	World-readable. Any app can read.
When the user uninstalls your app, the system removes all your app's files from internal storage.	When the user uninstalls your app, the system removes your app's files from here only if you save them in the directory from `getExternalFilesDir()`.
Internal storage is best when you want to be sure that neither the user nor other apps can access your files.	External storage is the best place for files that don't require access restrictions and for files that you want to share with other apps or allow the user to access with a computer.

Internal storage

You don't need any permissions to save files on the internal storage. Your app always has permission to read and write files in its internal storage directory.

You can create files in two different directories:

- Permanent storage: `getFilesDir()`
- Temporary storage: `getCacheDir()`. Recommended for small, temporary files totaling less than 1MB. Note that the system may delete temporary files if it runs low on memory.

To create a new file in one of these directories, you can use the `File()` constructor, passing the `File` provided by one of the above methods that specifies your internal storage directory. For example:

```
File file = new File(context.getFilesDir(), filename);
```

Alternatively, you can call `openFileOutput()` to get a `FileOutputStream` that writes to a file in your internal directory. For example, here's how to write some text to a file:

```
String filename = "myfile";
String string = "Hello world!";
FileOutputStream outputStream;

try {
  outputStream = openFileOutput(filename, Context.MODE_PRIVATE);
  outputStream.write(string.getBytes());
  outputStream.close();
} catch (Exception e) {
  e.printStackTrace();
}
```

Or, if you need to cache some files, instead use `createTempFile()`. For example, the following method extracts the filename from a `URL` and creates a file with that name in your app's internal cache directory:

```
public File getTempFile(Context context, String url) {
    File file;
    try {
        String fileName = Uri.parse(url).getLastPathSegment();
        file = File.createTempFile(fileName, null, context.getCacheDir());
    } catch (IOException e) {
        // Error while creating file
    }
    return file;
}
```

External storage

Use external storage for files that should be permanently stored, even if your app is uninstalled, and be available freely to other users and apps, such as pictures, drawings, or documents made by your app.

Some private files that are of no value to other apps can also be stored on external storage. Such files might be additional downloaded app resources, or temporary media files. Make sure you delete those when your app is uninstalled.

Obtain permissions for external storage

To write to the external storage, you must request the `WRITE_EXTERNAL_STORAGE` permission in your Android manifest. This implicitly includes permission to read.

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

If your app needs to read the external storage (but not write to it), then you will need to declare the `READ_EXTERNAL_STORAGE` permission.

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

Always check whether external storage is mounted

Because the external storage may be unavailable—such as when the user has mounted the storage to a PC or has removed the SD card that provides the external storage—you should always verify that the volume is available before accessing it. You can query the external storage state by calling `getExternalStorageState()`. If the returned state is equal to `MEDIA_MOUNTED`, then you can read and write your files. For example, the following methods are useful to determine the storage 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;
}
```

Public and private external storage

External storage is very specifically structured by the Android system for various purposes. There are public and private directories specific to your app. Each of these file trees has directories identified by system constants.

For example, any files that you store into the public ringtone directory `DIRECTORY_RINGTONES` are available to all other ringtone apps.

On the other hand, any files you store in a private ringtone directory `DIRECTORY_RINGTONES` can, by default, only be seen by your app and are deleted along with your app.

See list of public directories for the full listing.

Getting file descriptors

To access a public external storage directory, get a path and create a file calling `getExternalStoragePublicDirectory()`.

```
File path = Environment.getExternalStoragePublicDirectory(
            Environment.DIRECTORY_PICTURES);
File file = new File(path, "DemoPicture.jpg");
```

To access a private external storage directory, get a path and create a file calling `getExternalFilesDir()`.

```
File file = new File(getExternalFilesDir(null), "DemoFile.jpg");
```

Querying storage space

If you know ahead of time how much data you're saving, you can find out whether sufficient space is available without causing an `IOException` by calling `getFreeSpace()` or `getTotalSpace()`. These methods provide the current available space and the total space in the storage volume, respectively.

You aren't required to check the amount of available space before you save your file. You can instead try writing the file right away, then catch an `IOException` if one occurs. You may need to do this if you don't know exactly how much space you need.

Deleting files

You should always delete files that you no longer need. The most straightforward way to delete a file is to have the opened file reference call `delete()` on itself.

```
myFile.delete();
```

If the file is saved on internal storage, you can also ask the `Context` to locate and delete a file by calling `deleteFile()`:

```
myContext.deleteFile(fileName);
```

As a good citizen, you should also regularly delete cached files that you created with `getCacheDir()`.

Interacting with files: summary

Once you have the file descriptors, use standard java.io file operators or streams to interact with the files. This topic is not Android-specific, so it's not covered here.

SQLite database

Saving data to a database is ideal for repeating or structured data, such as contact information. Android provides an SQL-like database for this purpose.

The SQLite Primer chapter gives you an overview on using an SQLite database.

Room persistence library

The Room persistence library provides an abstraction layer over SQLite to allow fluent database access while harnessing the full power of SQLite.

The library helps you create a cache of your app's data on a device that's running your app. This cache, which serves as your app's single source of truth, allows users to view a consistent copy of key information within your app, regardless of whether users have an internet connection.

You will learn more about Room in the Room, LiveData, and ViewModel chapter. For more information see the Room training guide.

Other storage options

Android provides additional storage options that are beyond the scope of this introductory course. If you'd like to explore them, see the Learn More section below.

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

Backing up app data

Users often invest significant time and effort creating data and setting preferences within apps. Preserving that data for users if they replace a broken device or upgrade to a new one is an important part of ensuring a great user experience.

Auto backup for Android 6.0 (API level 23) and higher

For apps whose target SDK version is Android 6.0 (API level 23) and higher, devices running Android 6.0 and higher automatically backup app data to the cloud. The system performs this automatic backup for nearly all app data by default, and does so without you writing any additional app code.

When a user installs your app on a new device, or re-installs your app on one (for example, after a factory reset), the system automatically restores the app data from the cloud. The automatic backup feature preserves the data your app creates on a user device by uploading it to the user's Google Drive account and encrypting it. There is no charge to you or the user for data storage, and the saved data does not count towards the user's personal Google Drive quota. Each app can store up to 25MB. Once its backed-up data reaches 25MB, the app no longer sends data to the cloud. If the system performs a data restore, it uses the last data snapshot that the app had sent to the cloud.

Automatic backups occur when the following conditions are met:

- The device is idle.
- The device is charging.
- The device is connected to a Wi-Fi network.
- At least 24 hours have elapsed since the last backup.

You can customize and configure auto backup for your app. See Configuring Auto Backup for Apps.

Backup API for Android 5.1 (API level 22) and lower

For users with previous versions of Android, you need to use the Backup API to implement data backup. In summary, this requires you to:

1. Register for the Android Backup Service to get a Backup Service Key.
2. Configure your Manifest to use the Backup Service.
3. Create a backup agent by extending the BackupAgentHelper class.
4. Request backups when data has changed.

More information and sample code:

- Using the Backup API
- Data Backup

Firebase

Firebase is a mobile platform that helps you develop apps, grow your user base, and earn more money. Firebase is made up of complementary features that you can mix-and-match to fit your needs.

Some features are Analytics, Cloud Messaging, Notifications, and the Test Lab.

For data management, Firebase offers a Realtime Database.

- Store and sync data with a NoSQL cloud database.
- Connected apps share data
- Hosted in the cloud
- Data is stored as JSON
- Data is synchronized in real time to every connected client
- Data remains available when your app goes offline

See the Firebase home for more information.

Custom data store for preferences

By default, the `Preference` class stores its values into the `SharedPreferences` interface, which is the recommended way to persist user preferences. However, providing a custom data store to your preferences can be useful if your app stores the preferences in a cloud or local database, or if the preferences are device-specific.

On devices running Android 8.0 (API level 26) or higher, you can achieve this by providing any `Preference` object with your implementation of the `PreferenceDataStore` interface.

See Setting up a custom data store for more information.