Saving Files

Android uses a file system that's similar to disk-based file systems on other platforms. This lesson describes how to work with the Android file system to read and write files with the `File` (/reference/java/io/File.html) APIs.

A `File` (/reference/java/io/File.html) object is suited to reading or writing large amounts of data in start-to-finish order without skipping around. For example, it's good for image files or anything exchanged over a network.

This lesson shows how to perform basic file-related tasks in your app. The lesson assumes that you are familiar with the basics of the Linux file system and the standard file input/output APIs in `java.io` (/reference/java/io/package-summary.html).

THIS LESSON TEACHES YOU TO

1. Choose Internal or External Storage
2. Obtain Permissions for External Storage
3. Save a File on Internal Storage
4. Save a File on External Storage
5. Query Free Space
6. Delete a File

YOU SHOULD ALSO READ

- Using the Internal Storage
- Using the External Storage

Choose Internal or External Storage

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). 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:

- It's always available.
- Files saved here are accessible by only your app by default.
- When the user uninstalls your app, the system removes all your app's files from internal storage.

Internal storage is best when you want to be sure that neither the user nor other apps can access your files.

External storage:

- It's not always available, because the user can mount the external storage as USB storage and in some cases remove it from the device.
- It's world-readable, so files saved here may be read outside of your control.
- 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()`.

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.

> **Tip:** Although apps are installed onto the internal storage by default, you can specify the `android:installLocation` (/guide/topics/manifest/manifest-element.html#install) attribute in your manifest so your app may be installed on external storage. Users appreciate this option when the APK size is very large and they have an external storage space that's larger than the internal storage. For more information, see App Install Location (/guide/topics/data/install-location.html).

Obtain Permissions for External Storage

To write to the external storage, you must request the WRITE_EXTERNAL_STORAGE (/reference/android/Manifest.permission.html#WRITE_EXTERNAL_STORAGE) permission in your manifest file (/guide/topics/manifest/manifest-intro.html):

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

Caution: Currently, all apps have the ability to read the external storage without a special permission. However, this will change in a future release. If your app needs to read the external storage (but not write to it), then you will need to declare the READ_EXTERNAL_STORAGE (/reference/android/Manifest.permission.html#READ_EXTERNAL_STORAGE) permission. To ensure that your app continues to work as expected, you should declare this permission now, before the change takes effect.

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

However, if your app uses the WRITE_EXTERNAL_STORAGE (/reference/android/Manifest.permission.html#WRITE_EXTERNAL_STORAGE) permission, then it implicitly has permission to read the external storage as well.

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

Save a File on Internal Storage

When saving a file to internal storage, you can acquire the appropriate directory as a File (/reference/java/io/File.html) by calling one of two methods:

getFilesDir()
: Returns a File representing an internal directory for your app.

getCacheDir()
: Returns a File representing an internal directory for your app's temporary cache files. Be sure to delete each file once it is no longer needed and implement a reasonable size limit for the amount of memory you use at any given time, such as 1MB. If the system begins running low on storage, it may delete your cache files without warning.

To create a new file in one of these directories, you can use the File() (/reference/java/io/File.html#File(java.io.File, java.lang.String)) constructor, passing the File (/reference/java/io/File.html) 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() (/reference/android/content/Context.html#openFileOutput(java.lang.String, int)) to get a FileOutputStream (/reference/java/io

/FileOutputStream.html) 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, you should instead use createTempFile() (/reference/java/io/File.html#createTempFile(java.lang.String, java.lang.String)). For example, the following method extracts the file name from a URL (/reference/java/net/URL.html) 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;
}
```

Note: Your app's internal storage directory is specified by your app's package name in a special location of the Android file system. Technically, another app can read your internal files if you set the file mode to be readable. However, the other app would also need to know your app package name and file names. 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. So as long as you use MODE_PRIVATE (/reference/android/content/Context.html#MODE_PRIVATE) for your files on the internal storage, they are never accessible to other apps.

Save a File on External Storage

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() (/reference/android/os/Environment.html#getExternalStorageState()). If the returned state is equal to MEDIA_MOUNTED (/reference/android/os/Environment.html#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;
}
```

Although the external storage is modifiable by the user and other apps, there are two categories of files you might save here:

Public files
: Files that should be freely available to other apps and to the user. When the user uninstalls your app, these files should remain available to the user.

  For example, photos captured by your app or other downloaded files.

Private files
: Files that rightfully belong to your app and should be deleted when the user uninstalls your app. Although these files are technically accessible by the user and other apps because they are on the external storage, they are files that realistically don't provide value to the user outside your app. When the user uninstalls your app, the system deletes all files in your app's external private directory.

  For example, additional resources downloaded by your app or temporary media files.

If you want to save public files on the external storage, use the `getExternalStoragePublicDirectory()` (/reference/android/os/Environment.html#getExternalStoragePublicDirectory(java.lang.String)) method to get a `File` (/reference/java/io/File.html) representing the appropriate directory on the external storage. The method takes an argument specifying the type of file you want to save so that they can be logically organized with other public files, such as `DIRECTORY_MUSIC` (/reference/android/os/Environment.html#DIRECTORY_MUSIC) or `DIRECTORY_PICTURES` (/reference/android/os/Environment.html#DIRECTORY_PICTURES). For example:

```
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;
}
```

If you want to save files that are private to your app, you can acquire the appropriate directory by calling `getExternalFilesDir()` and passing it a name indicating the type of directory you'd like. Each directory created this way is added to a parent directory that encapsulates all your app's external storage files, which the system deletes when the user uninstalls your app.

For example, here's a method you can use to create a directory for an individual photo album:

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

If none of the pre-defined sub-directory names suit your files, you can instead call `getExternalFilesDir()` and pass `null`. This returns the root directory for your app's private directory on the external storage.

Remember that `getExternalFilesDir()` creates a directory inside a directory that is deleted when the user uninstalls your app. If the files you're saving should remain available after the user uninstalls your app—such as when your app is a camera and the user will want to keep the photos—you should instead use `getExternalStoragePublicDirectory()`.

Regardless of whether you use `getExternalStoragePublicDirectory()` for files that are shared or `getExternalFilesDir()` for files that are private to your app, it's important that you use directory names provided by API constants like `DIRECTORY_PICTURES`. These directory names ensure that the files are treated properly by the system. For instance, files saved in `DIRECTORY_RINGTONES` are categorized by the system media scanner as ringtones instead of music.

Query Free 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. This information is also useful to avoid filling the storage volume above a certain threshold.

However, the system does not guarantee that you can write as many bytes as are indicated by `getFreeSpace()`. If the number returned is a few MB more than the size of the data you want to save, or if the file system is less than 90% full, then it's probably safe to proceed. Otherwise, you probably shouldn't write to storage.

> **Note:** 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` (/reference/java/io/IOException.html) if one occurs. You may need to do this if you don't know exactly how much space you need. For example, if you change the file's encoding before you save it by converting a PNG image to JPEG, you won't know the file's size beforehand.

Delete a File

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()` (/reference/java/io/File.html#delete()) on itself.

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

If the file is saved on internal storage, you can also ask the `Context` (/reference/android/content/Context.html) to locate and delete a file by calling `deleteFile()` (/reference/android/content/Context.html#deleteFile(java.lang.String)):

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

> **Note:** When the user uninstalls your app, the Android system deletes the following:
- All files you saved on internal storage
- All files you saved on external storage using `getExternalFilesDir()`.

> However, you should manually delete all cached files created with `getCacheDir()` (/reference/android/content/Context.html#getCacheDir()) on a regular basis and also regularly delete other files you no longer need.