ICT-605
Mobile Application and Game Development

Pintu Chandra Paul

Lecturer

Dept. of ICT

Comilla University

References

1. Beginning Android Programming with Android Studio by J. F. DiMarzio .
2. https://www.javatpoint.com/android-tutorial
3. https://www.tutorialspoint.com/android/index.htm

DATA PERSISTENCE

- Persisting data is an important topic in application development because users typically expect to reuse data in the future. For Android, there are primarily three basic ways of persisting data:

  - A lightweight mechanism known as *shared preferences* to save small chunks of data
  - Traditional file systems
  - A relational database management system through the support of SQLite databases

DATA STORAGE

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.

Shared Preferences

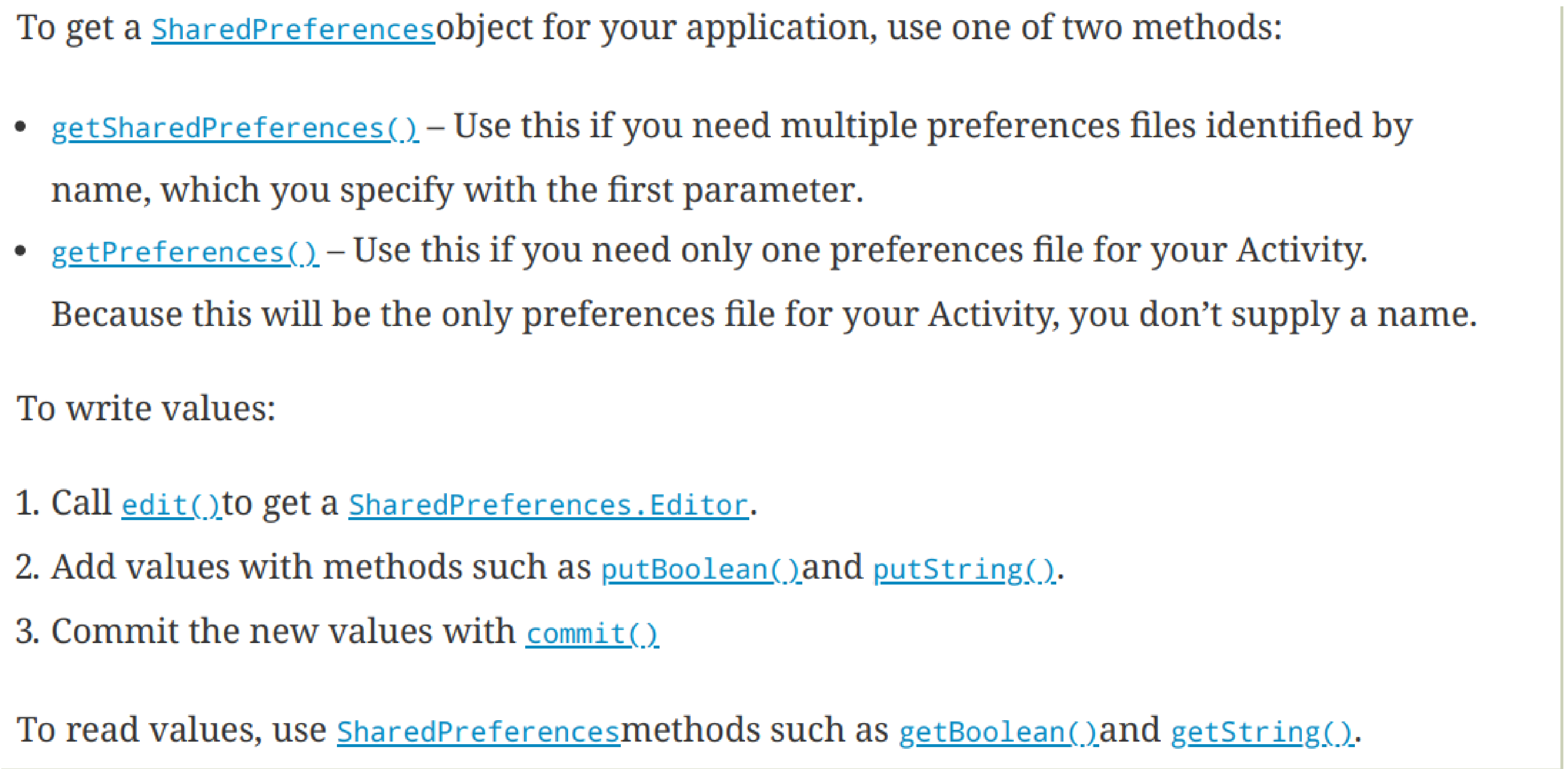
Using Shared Preferences

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

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`, which provides an Activity framework for you to create user preferences, which will be automatically persisted (using shared preferences).

Shared Preferences



To get a `SharedPreferences`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()`

To read values, use `SharedPreferences`methods such as `getBoolean()`and `getString()`.

Shared Preferences Example

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);
    }
```

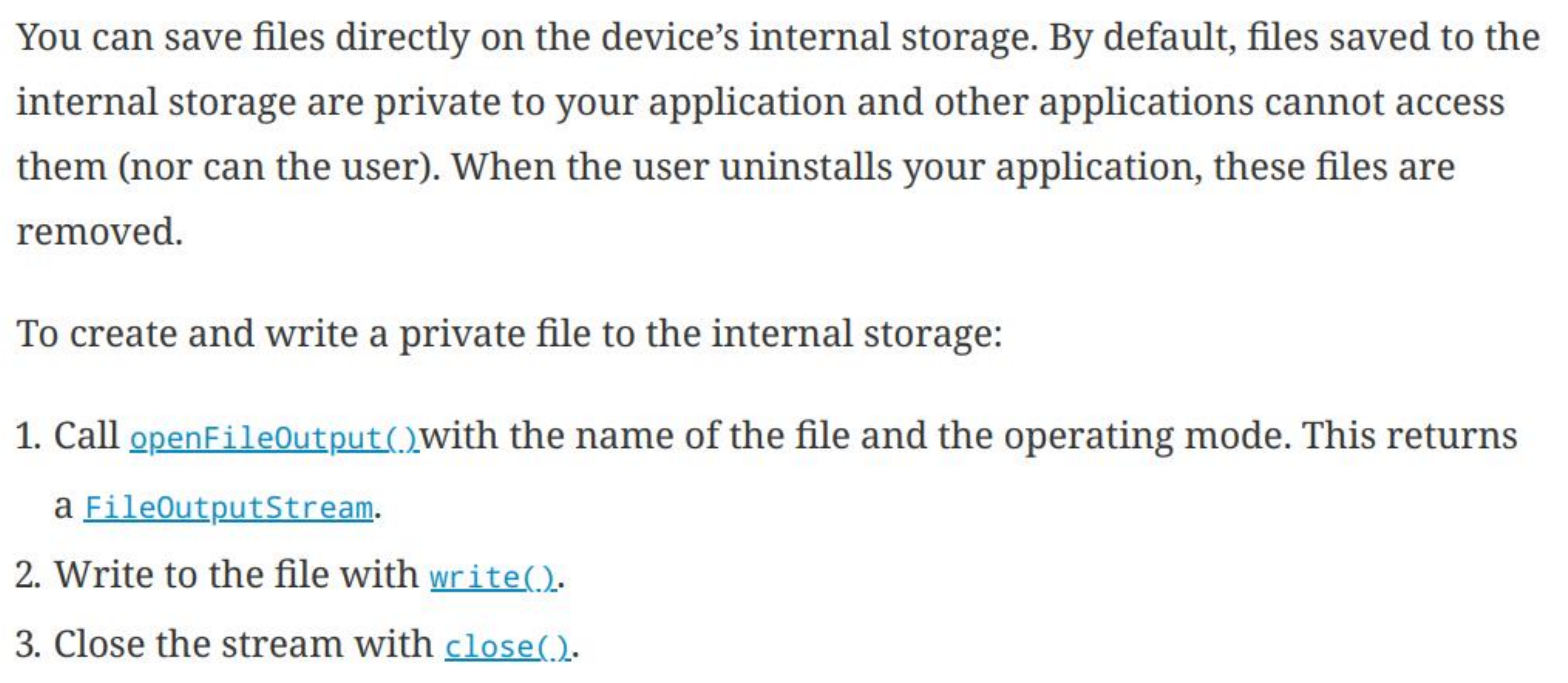
Shared Preferences Example

```
    @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();
    }
}
```

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()`.

Internal Storage

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`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`, `MODE_WORLD_READABLE`, and `MODE_WORLD_WRITEABLE`.

Internal Storage

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()`.

Saving cache files

If you'd like to cache some data, rather than store it persistently, you should use `getCacheDir()`to open a `File`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.

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 files can disappear 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.

Choosing Best Storage

The previous sections described three main ways to save data in your Android applications: the `SharedPreferences` object, internal storage, and external storage. Which one should you use in your applications? Here are some guidelines:

- If you have data that can be represented using name/value pairs, then use the `SharedPreferences` object. For example, if you want to store user preference data such as username, background color, date of birth, or last login date, then the `SharedPreferences` object is the ideal way to store this data. Moreover, you don't really have to do much to store data this way. Simply use the `SharedPreferences` object to store and retrieve it.
- If you need to store ad-hoc data then using the internal storage is a good option. For example, your application (such as an RSS reader) might need to download images from the web for display. In this scenario, saving the images to internal storage is a good solution. You might also need to persist data created by the user, such as when you have an application that enables users to take notes and save them for later use. In both of these scenarios, using the internal storage is a good choice.
- There are times when you need to share your application data with other users. For example, you might create an Android application that logs the coordinates of the locations that a user has been to, and subsequently, you want to share all this data with other users. In this scenario, you can store your files on the SD card of the device so that users can easily transfer the data to other devices (and computers) for use later.

Android Database

Android provides full support for SQLite 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`and override the`onCreate()`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);";
```

Android Database

```
    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`implementation using the constructor you've defined. To write to and read from the database, call `getWritableDatabase()`and `getReadableDatabase()`, respectively. These both return a `SQLiteDatabase`object that represents the database and provides methods for SQLite operations.

Android Database

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 acontent provider, you must include a unique ID using the BaseColumns._IDconstant.

You can execute SQLite queries using the SQLiteDatabasequery()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, which provides several convienent methods for building queries.

Every SQLite query will return a Cursorthat points to all the rows found by the query. The Cursoris always the mechanism with which you can navigate results from a database query and read rows and columns.

Task

- *Implement android SQLite database on your app.*

The END