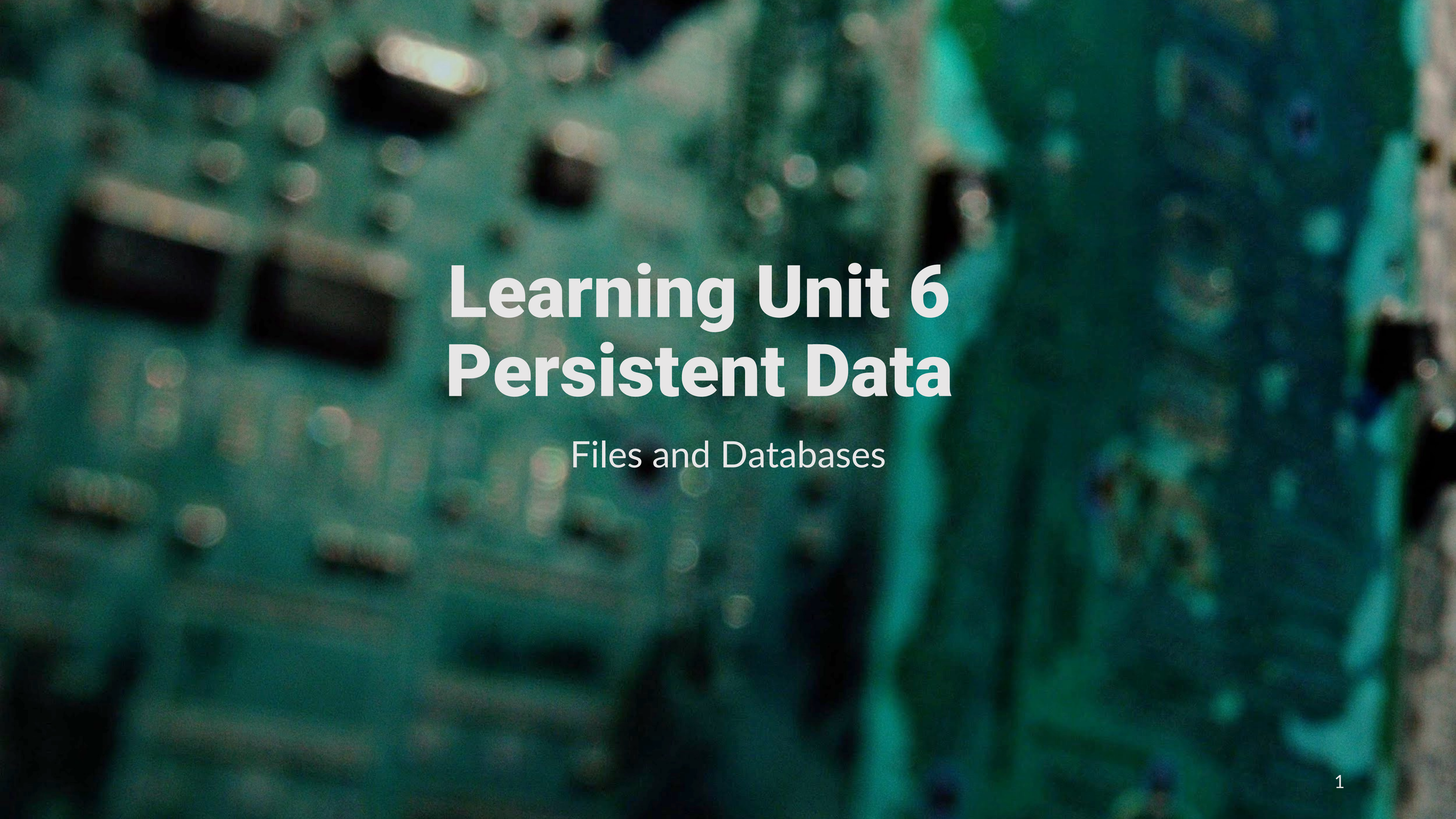
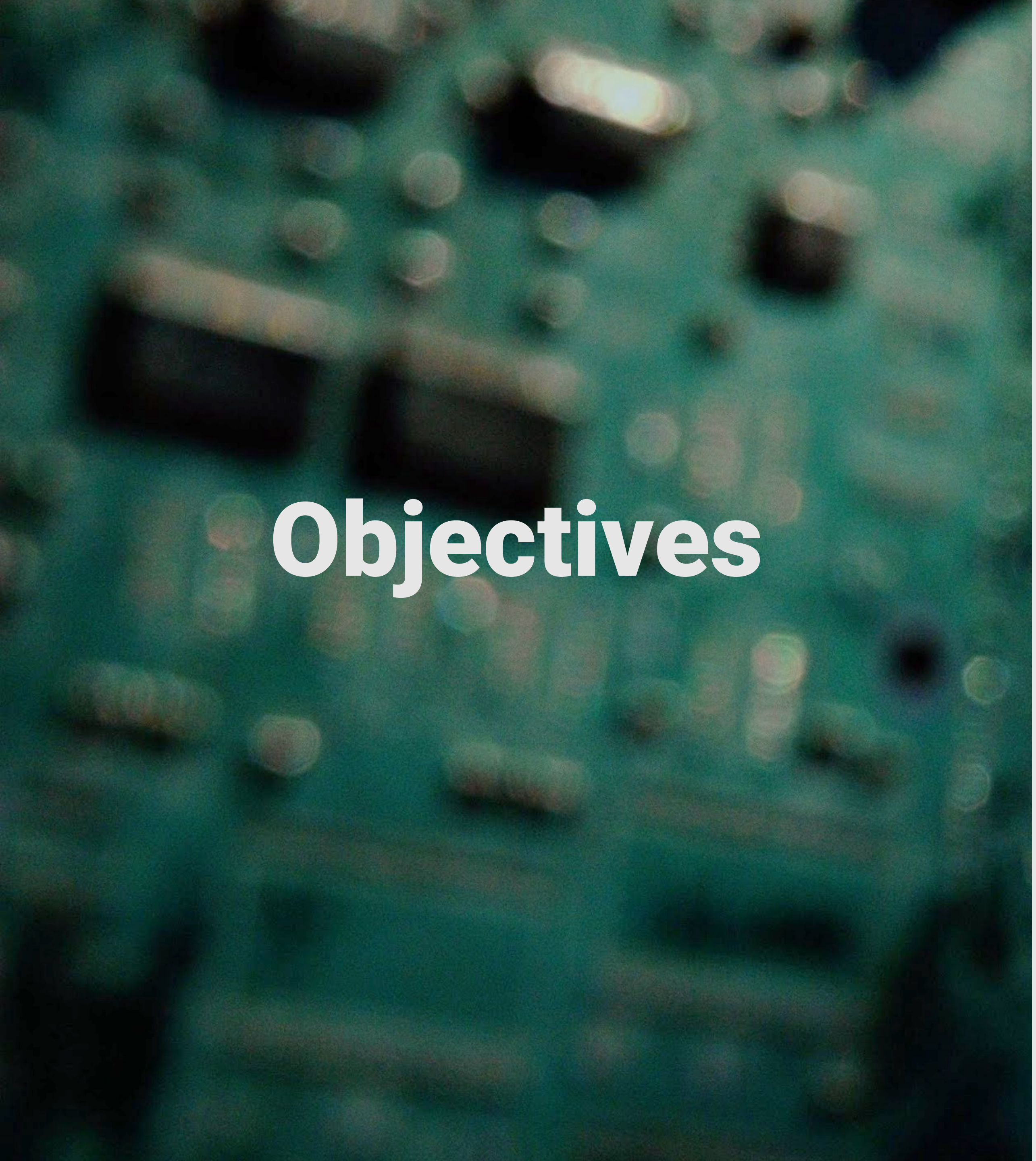


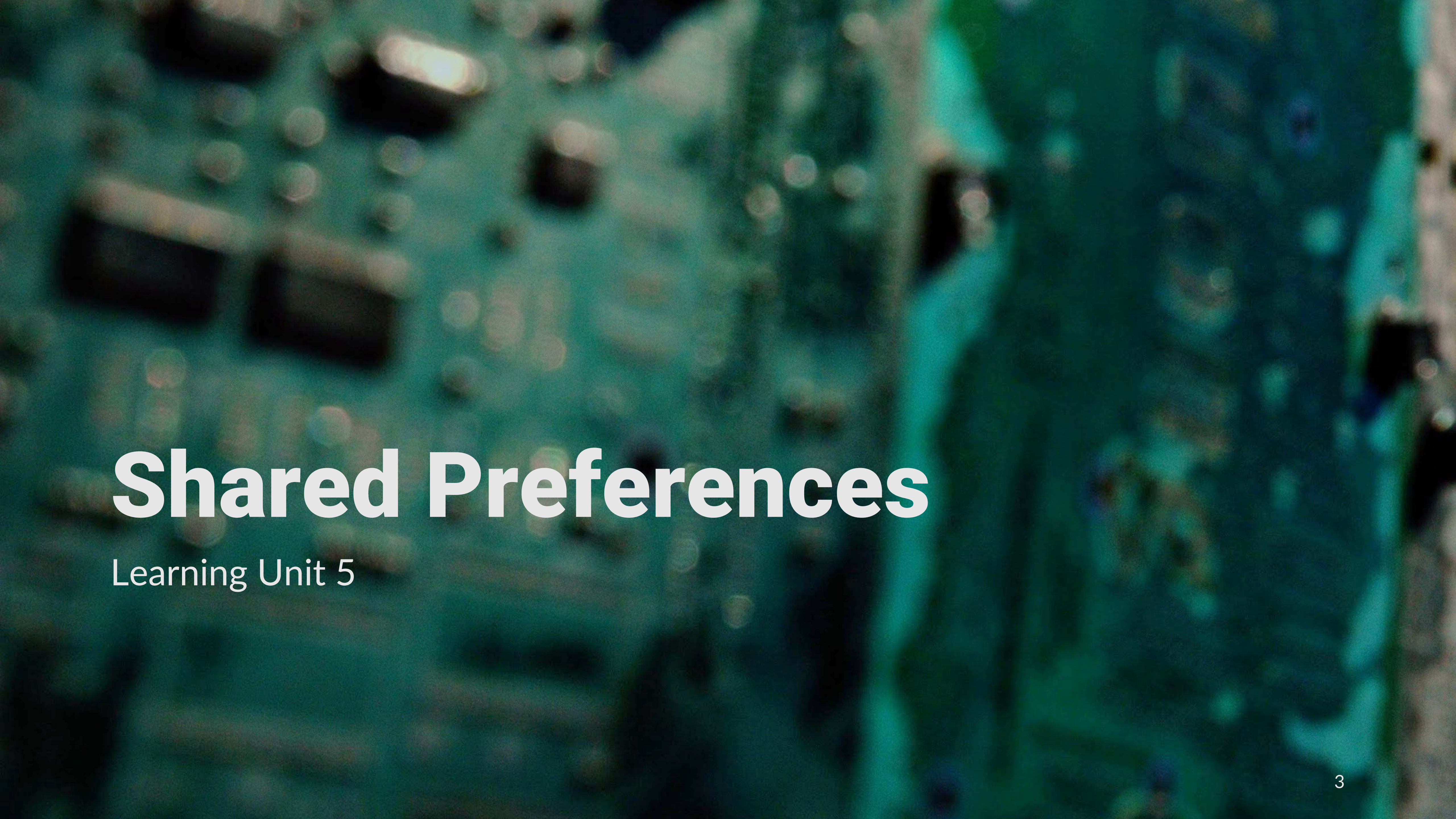
# Learning Unit 6
# Persistent Data

Files and Databases



# Objectives

1. Review the use of key-Value data.
2. Understand and use of Files.
3. Utilize Android Databases - SQLite.
4. Apply Android best practices with Persistent Data
5. Apply storing data in practical applications.

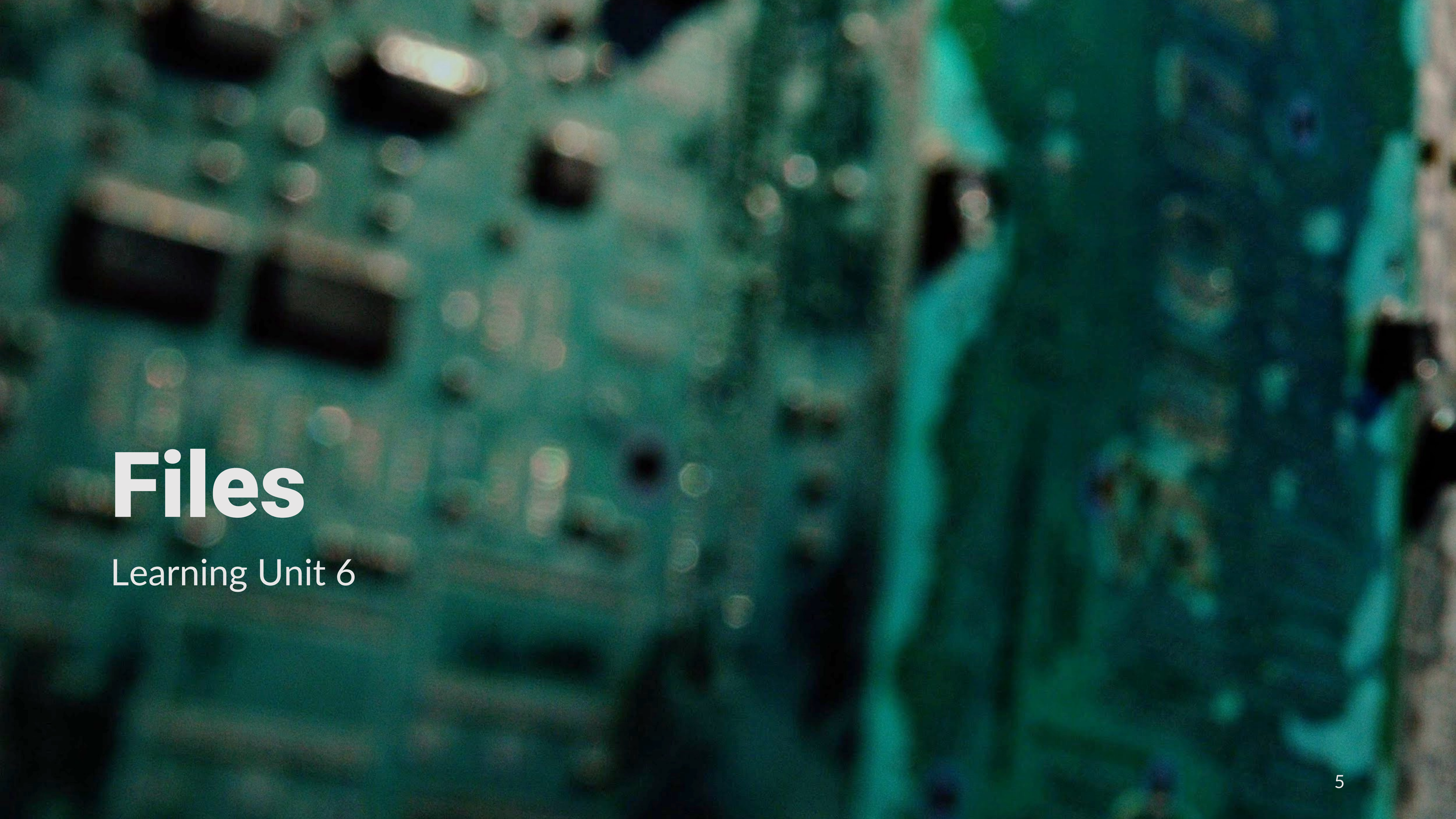


# Shared Preferences

Learning Unit 5

# Introduction to Persistent Data

- Android provides different data storage options available on Android:
  - **Shared preferences**: Store private primitive data in key-value pairs.
  - **Internal file storage**: Store app-private files on the device file system.
  - **External file storage**: Store files on the shared external file system. This is usually for shared user files, such as photos.
  - **Databases**: Store structured data in a private database.
  - **See Last unit slides**



# Files

Learning Unit 6

# Saving Files

- a file system that's similar to disk-based file systems on other platforms.
- A ‘File’ object is suited to reading or writing large amounts of data
  - good for images, network, audio
- two file storage areas: "internal" and "external" storage.

# Permissions

- Internal storage:
  - is 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 (and external if available).
- External storage:
  - To write to the external storage, you must request the WRITE_EXTERNAL_STORAGE permission in your manifest file:

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

  - Recommended to declare the READ_EXTERNAL_STORAGE permission

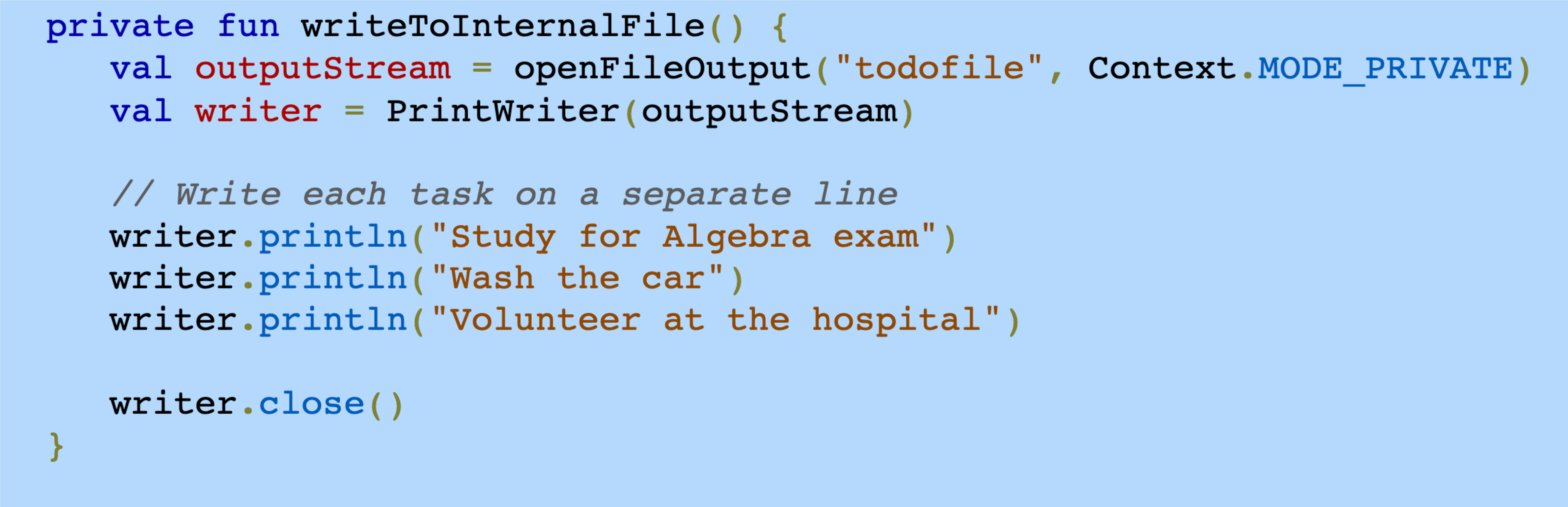
# Writing a File:

- Internal Storage:
  - acquire the appropriate directory as a File 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.

- External Storage:
  - Availability and volume need check before use:
    - getExternalStorageState()
      If the returned state is equal to MEDIA_MOUNTED, then you can read and write your files.

```
private fun isExternalStorageWritable(): Boolean {

return Environment.getExternalStorageState() ==
Environment.MEDIA_MOUNTED
}
```

# Writing a File:



```
private fun writeToInternalFile() {
    val outputStream = openFileOutput("todofile", Context.MODE_PRIVATE)
    val writer = PrintWriter(outputStream)

    // Write each task on a separate line
    writer.println("Study for Algebra exam")
    writer.println("Wash the car")
    writer.println("Volunteer at the hospital")

    writer.close()
}
```

# Reading a File

- In order to read from the file, call the openFileInput()
  - method with the name of the file.
  It returns an instance of FileInputStream.

```
private fun readFromInternalFile(): String {
    val inputStream = openFileInput("todofile")
    val reader = inputStream.bufferedReader()
    val stringBuilder = StringBuilder()
    val lineSeparator = System.getProperty("line.separator")

    // Append each task to stringBuilder
    reader.forEachLine { stringBuilder.append(it).append(lineSeparator) }

    return stringBuilder.toString()
}
```

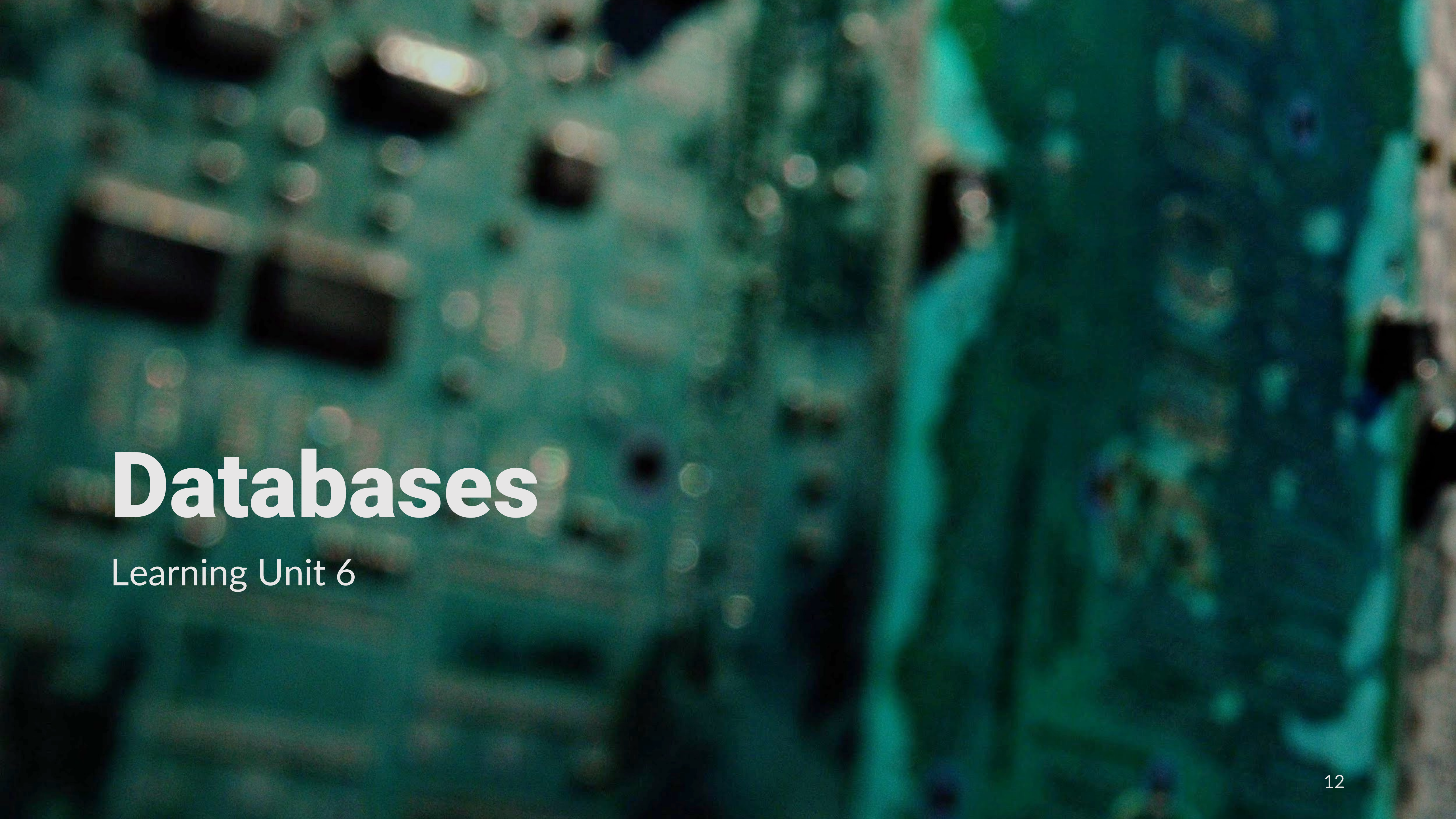
# Delete a File

- 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() on itself.

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

- If the file is saved on internal storage, you can also ask the Context to calling deleteFile():

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



# Databases

Learning Unit 6

# SQLite

- Embedded RDBMS
- ACID Compliant
- Size – about 257 Kbytes
- Not a client/server architecture
  - Accessed via function calls from the application
- Writing (insert, update, delete) locks the database,  queries can be done  in parallel

# SQLite

- Datastore – single, cross platform file (kinda like an MS Access DB)
  - Definitions
  - Tables
  - Indicies
  - Data

# SQLite Data Types

- This is quite different than the normal SQL data types so please read:

  https://www.sqlite.org/docs.html

# Storage classes

- **NULL** – null value
- **INTEGER** - signed integer, stored in 1, 2, 3, 4, 6, or 8 bytes depending on the magnitude of the value
- **REAL** - a floating point value,  8-byte IEEE floating point number.
- **TEXT** - text string, stored using the database encoding (UTF-8, UTF-16BE or UTF-16LE).
- **BLOB**. The value is a blob of data, stored exactly as it was input.

# android.database.sqlite

- Contains the SQLite database management classes that an application would use to manage its own private database.

# android.database.sqlite - Classes

- SQLiteCloseable - An object created from a SQLiteDatabase that can be closed.
- SQLiteCursor - A Cursor implementation that exposes results from a query on a SQLiteDatabase.
- SQLiteDatabase - Exposes methods to manage a SQLite database.
- SQLiteOpenHelper - A helper class to manage database creation and version management.
- SQLiteProgram -  A base class for compiled SQLite programs.
- SQLiteQuery - A SQLite program that represents a query that reads the resulting rows into a CursorWindow.
- SQLiteQueryBuilder - a convenience class that helps build SQL queries to be sent to SQLiteDatabase objects.
- SQLiteStatement - A pre-compiled statement against a SQLiteDatabase that can be reused.

# android.database.sqlite.SQLiteDatabase

- Contains the methods for: creating, opening, closing, inserting, updating, deleting and quering an SQLite database
- These methods are similar to JDBC but more method oriented than what we see with JDBC (remember there is not a RDBMS server running)

# Room Database

- The ***Room persistence library*** provides an abstraction layer over SQLite, allowing developers to write significantly simpler code to interact with a SQLite database.

- Room is an ORM library. An ***object-relational mapping (ORM) library*** is software that converts objects in an object-oriented programming language into tables and queries in a relational database.

- Some knowledge of relational databases and SQL is helpful when using Room.

# Room compoenents

- Room defines three components:

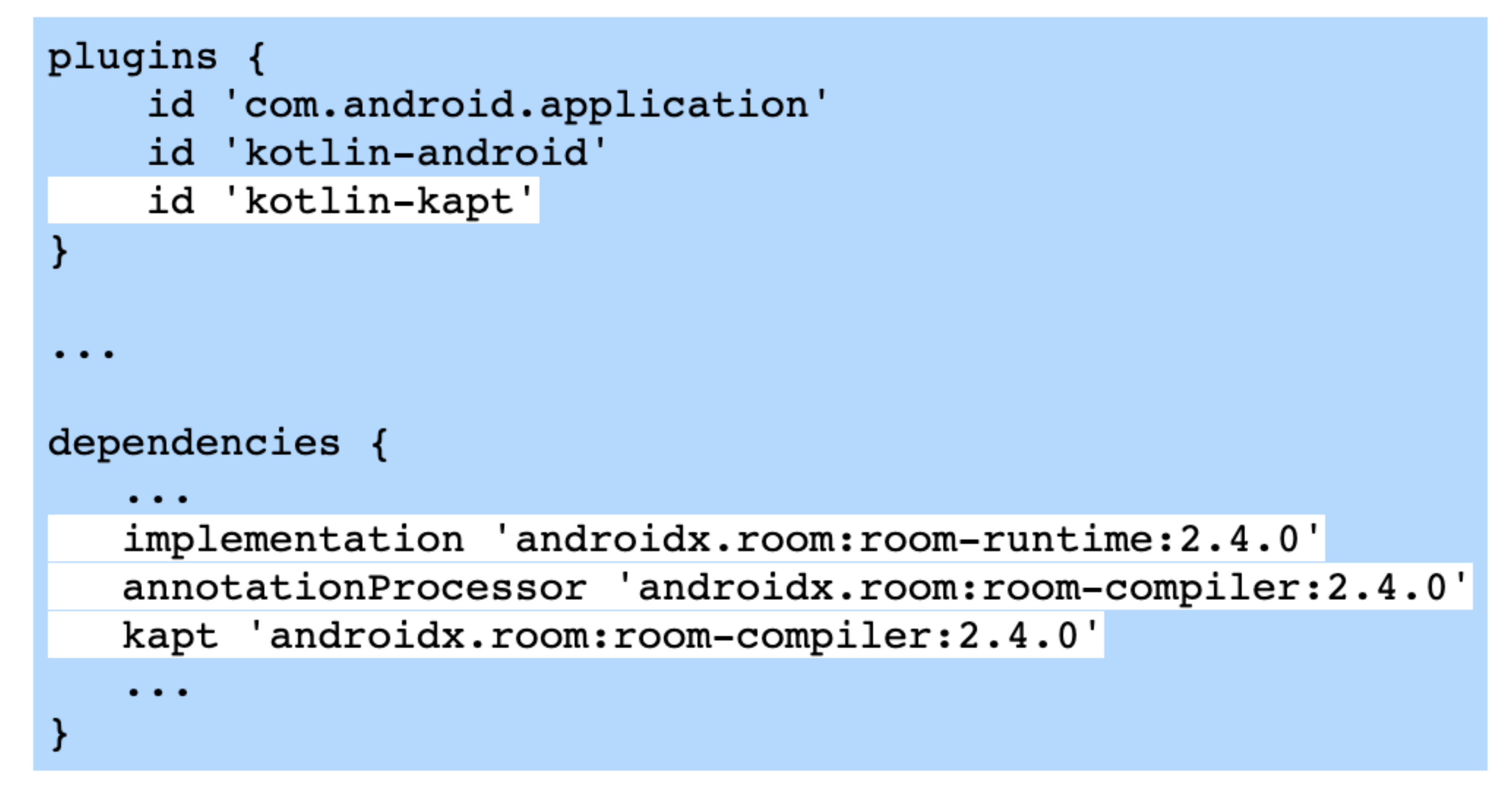
1. ***Entity*** is a class annotated with @Entity that defines the columns and keys of a database table.

2. ***DAO*** (***Data Access Object***) is an interface annotated with @Dao that defines methods for selecting, updating, inserting, and deleting entities in a database.

3. ***Database*** is an abstract class annotated with @Database that inherits from the RoomDatabase class and provides DAOs for accessing the database.

# Room dependences

Room dependencies for app module's build.gradle file.

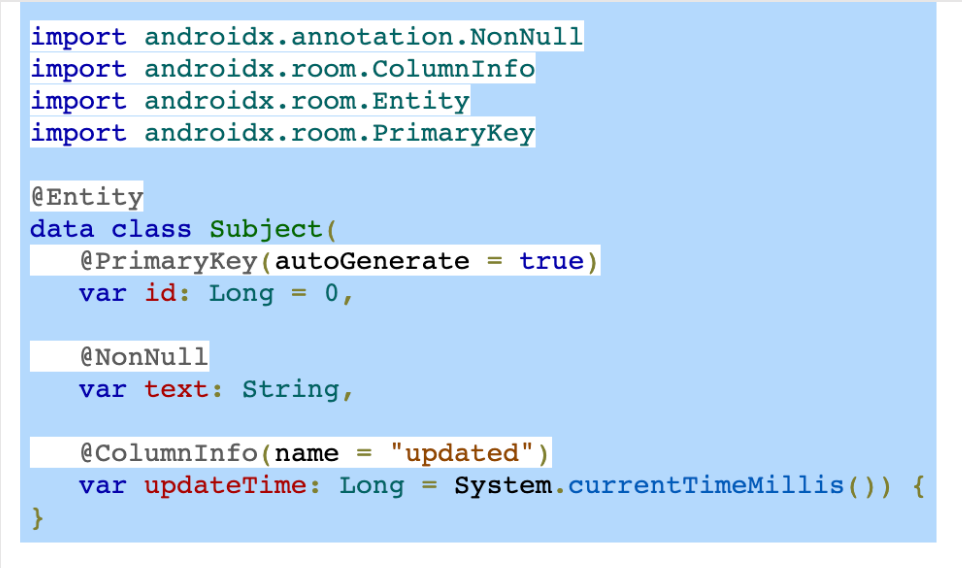


```
plugins {
     id 'com.android.application'
     id 'kotlin-android'
     id 'kotlin-kapt'
}

...

dependencies {
    ...
    implementation 'androidx.room:room-runtime:2.4.0'
    annotationProcessor 'androidx.room:room-compiler:2.4.0'
    kapt 'androidx.room:room-compiler:2.4.0'
    ...
}
```

# Entities

- A SQLite table is created for each entity class, and the entity's fields define the table columns. The figure below defines the Subject entity. Several annotations are used:

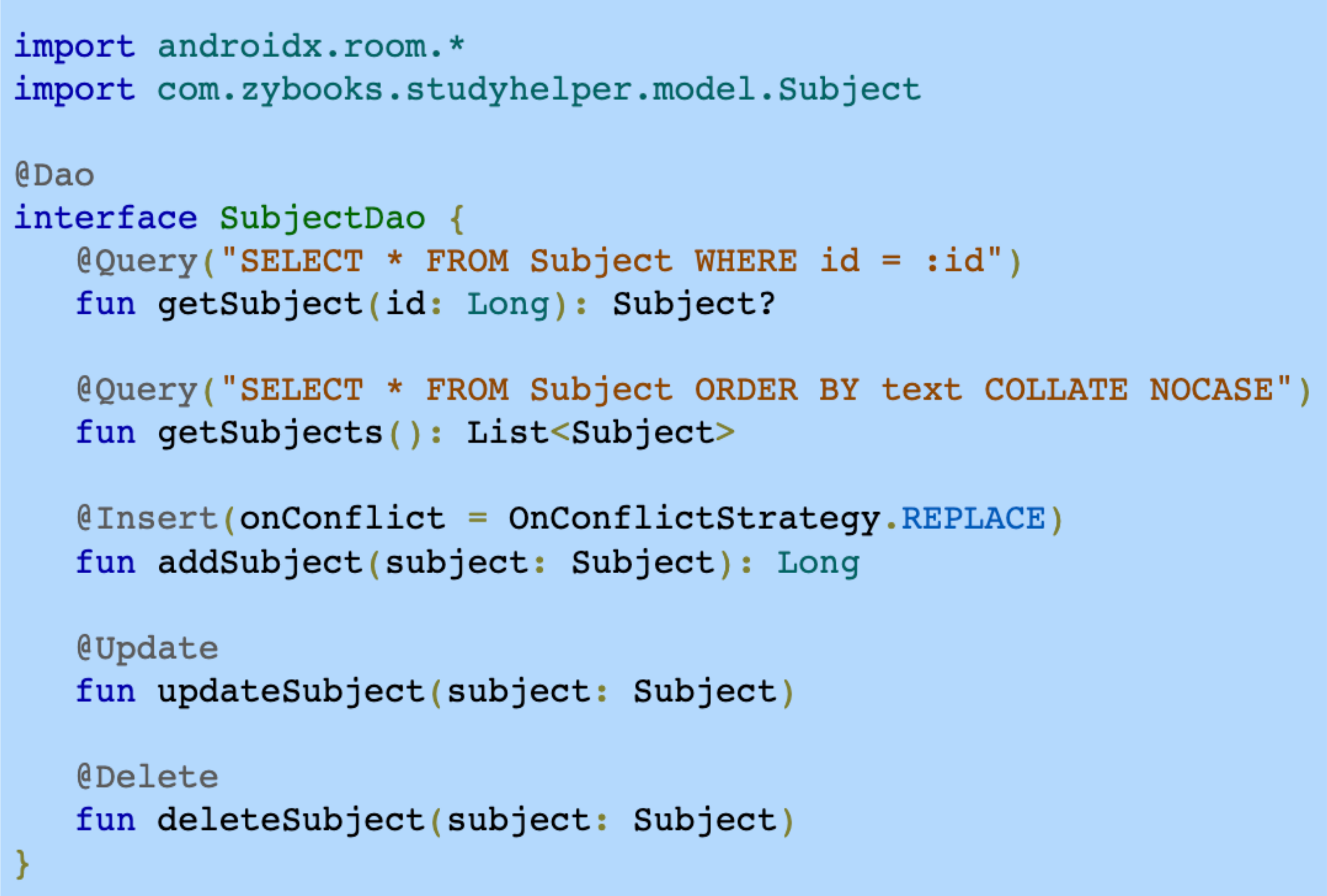
- ***@Entity*** designates the entity class. The class name is used to name the table unless the optional tableName property specifies a different table name.
- ***@PrimaryKey*** designates which field is the table's primary key. An entity must have at least one field annotated with @PrimaryKey. Typically the primary key is an integer or long field. Setting the autoGenerate property to true makes SQLite automatically generate unique numbers for the primary key.
- ***@NonNull*** indicates the field should not be null. SQLite does not allow a primary key to be null.
- ***@ColumnInfo*** with the name property specifies a column name for a field. If @ColumnInfo is not present, the field's name is used to name the column.



```
import androidx.annotation.NonNull
import androidx.room.ColumnInfo
import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity
data class Subject(
    @PrimaryKey(autoGenerate = true)
    var id: Long = 0,

    @NonNull
    var text: String,

    @ColumnInfo(name = "updated")
    var updateTime: Long = System.currentTimeMillis()) {
}
```

# Data Access Objects

- The ***@Dao*** annotation designates a DAO's public interface that defines methods to select, insert, update, and delete database entities.
  - Room implements the interface automatically, writing all the code necessary to interact with SQLite.
- ***@Query*** designates a database query, usually a SELECT statement, to be executed. The query can bind parameters from the abstract method.
  - Ex: The @Query for getSubject() has an :id parameter that matches the id parameter in getSubject().
- The return value for a @Query method matches the data returned by the SELECT statement.
  Ex: getSubjects() returns List<Subject> as the SELECT statement selects multiple rows from the table.
- ***@Insert*** designates an insert query, which inserts a new entity into the database using an INSERT statement. The onConflict property indicates what the database should do if the entity being inserted already exists.
  - The return value for an @Insert method is a long when the INSERT statement inserts a row with an auto-incremented ID. The new ID is returned by the @Insert method.
- ***@Update*** designates an update query, which updates an existing entity in the database using an UPDATE statement.
- ***@Delete*** designates a delete query, which deletes an entity from the database using a DELETE statement.



```
import androidx.room.*
import com.zybooks.studyhelper.model.Subject

@Dao
interface SubjectDao {
   @Query("SELECT * FROM Subject WHERE id = :id")
   fun getSubject(id: Long): Subject?

   @Query("SELECT * FROM Subject ORDER BY text COLLATE NOCASE")
   fun getSubjects(): List<Subject>

   @Insert(onConflict = OnConflictStrategy.REPLACE)
   fun addSubject(subject: Subject): Long

   @Update
   fun updateSubject(subject: Subject)

   @Delete
   fun deleteSubject(subject: Subject)
}
```

# Room database

- The Room database class is an abstract class that inherits from ***RoomDatabase***, the base class for all Room databases. The ***@Database*** annotation designates the Room database class and uses the property entities to name the database's entities and version to name the database version number.

```
import androidx.room.Database
import androidx.room.RoomDatabase
 import com.zybooks.studyhelper.model.Question
import com.zybooks.studyhelper.model.Subject

@Database(entities = [Question::class, Subject::class], version = 1)
   abstract class StudyDatabase : RoomDatabase() {
   abstract fun subjectDao(): SubjectDao
 }
```

# Thank you

- In this learning unit we discussed:
    - Shared Preferences
    - Files
        - Internal
        - External
    - Databases