Mobile Programming

Course 4
Data storage

Qiong LIU

qiong.liu@cyu.fr
CY Tech, CY Cergy Paris University

2024 - 2025

What You'll Learn Today

Java I/O

- Basics of File Handling
- Reading and Writing Files

Database Operation

- Introduction to SQL
- Connecting to Databases
- CRUD Operations (Create, Read, Update, Delete)
- Using SQLite

Database design for Android

- Introduction to Room
- Defining Entities and DAOs
- Writing Queries

Java I/O [1]

[1] Reference: https://courses.cs.washington.edu/courses/cse341/99wi/java/tutorial/java/io/overview.html

Introduction to Java I/O

Why Use Java I/O?

- Data in arrays, variables, and objects exists only temporarily in memory.
- Once the program stops, this data is destroyed.
- To store data persistently, we need to save it in disk files.

What is Java I/O?

- Java I/O (Input/Output) provides a way to read and write data to files.
- It enables saving, loading, and processing persistent data.
- Common tasks include reading text files, writing data, and handling streams.

Input/Output Streams

Stream:

- Streams are used to read data from a source or write data to a destination.
- Streams can process different types of data (e.g., text, binary, objects).

Java I/O Classes: in package `java.io`

- `InputStream`: Reads bytes from a source (e.g., a file or network).
- `OutputStream`: Writes bytes to a destination (e.g., a file or console).
- `Reader`: Reads characters (for text data).
- `Writer`: Writes characters (for text data).

Input/Output Streams

- InputStream is an abstract class in Java for reading raw byte data.
- Base Class: ‘InputStream‘ (abstract class).
- Common Subclasses:
  - `FileInputStream`: Reads data from a file.
  - `ByteArrayInputStream`: Reads data from a byte array.
  - `StringBufferInputStream` : Reads data from a string buffer.
  - `AudioInputStream`: Reads audio data.
  - `FilterInputStream`: Provides additional functionality by wrapping other streams.

[2]

[2] More class and method can be checked at "JAVA DEVELOPMENT KIT".

Input Streams

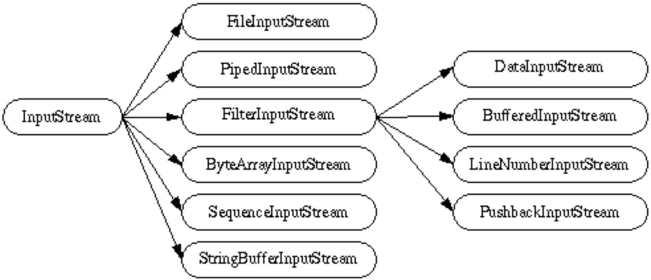


Figure: InputStream Class

Error Handling

- Almost all InputStream operations can throw an `IOException`.
- Always use `try-catch` blocks to handle these errors safely.
- Use `try-with-resources` for automatic resource management.

Input/Output Streams

- `OutputStream` is an abstract class in Java for writing raw data to a destination (e.g., files, memory, network). – package: `java.io`
- Common Subclasses:
  - `FileOutputStream`: Writes data to a file.
  - `ByteArrayInputStream`: Reads data from a byte array.
  - `AudioInputStream`: Reads audio data.
  - `FilterInputStream`: Wraps streams for additional functionality.

Output Streams

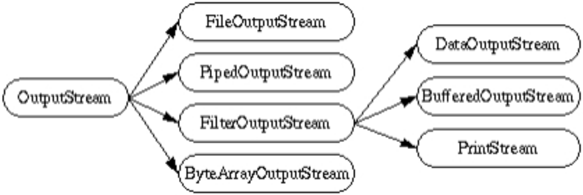


Figure: OutputStream Class

Error Handling

- Both InputStream and OutputStream methods can throw `IOException`.
- Use `try-catch` blocks or `try-with-resources` for safe resource management.

File Class

File Class

The File class represents the file or directory path in the file system. It is the only class in Java designed to directly represent disk files and directories.

Common Constructors

- `File(String pathname)`: Represents a file or directory by its path.
- `File(String parent, String child)`: Represents a file with a parent and child path.
- `File(File parent, String child)`: Combines a File object as parent with a child path.

Common Methods

- `exists()`: Checks if the file or directory exists.
- `isFile()/isDirectory()`: Checks if it is a file or directory.
- `length()`: Returns the file size in bytes.
- `canRead()/canWrite()`: Checks read or write permissions.
- `getName()`, `getPath()`, `getAbsolutePath()`: Fetches path information.
- `createNewFile()/delete()`: Creates or deletes files.

File Class: Code Example

```java
import java.io.File;

public class FileExample {
    public static void main(String[] args) {
        // Create a File object
        File file = new File("example.txt");

        // Check if the file exists
        if (file.exists()) {
            System.out.println("File exists.");
            System.out.println("Name: " + file.getName());
            System.out.println("Path: " + file.getAbsolutePath());
            System.out.println("Size: " + file.length() + " bytes");
            System.out.println("Readable: " + file.canRead());
            System.out.println("Writable: " + file.canWrite());
        } else {
            try {
                // Create a new file
                if (file.createNewFile()) {
                    System.out.println("File created: " + file.getName());
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

FileInputStream & FileOutputStream

FileInputStream & FileOutputStream

A spacial class from `InputStream`, `OutputSteam`, expecially for files.

Common Methods in FileInputStream

- `int read()`: Reads one byte of data. Returns -1 if the end of the file is reached.
- `int read(byte[] b)`: Reads up to `b.length` bytes into the array.
- `void close()`: Closes the stream and releases resources.

Common Methods in FileOutputStream

- `void write(int b)`: Writes one byte of data.
- `void write(byte[] b)`: Writes all the bytes from the array to the file.
- `void close()`: Closes the stream and releases resources.

BufferedInputStream & BufferedOutputStream

BufferedInputStream & BufferedOutputStream

These classes enhance the performance of input and output streams by adding a memory buffer. By default, a 32-byte buffer is used, but a custom size can be specified.

Constructors:

```
BufferedInputStream(InputStream in, int size)
BufferedInputStream(InputStream in)
```

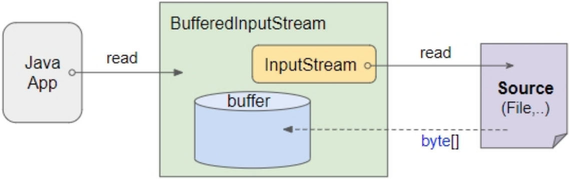


Figure: Buffer Input Steam (Similar for the Buffer Output Steam)

BufferedInputStream: Reading Process

How BufferedInputStream Works

BufferedInputStream overrides methods that inherit from its parent class, such as read(), read(byte[]), ... to ensure that they will manipulate data from the buffer rather than from the origin (e.g. file).

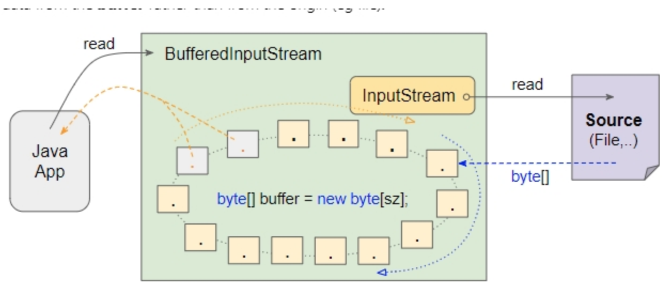


Figure: `BufferedInputStream` reads bytes from buffer array and frees the read positions. Freed positions will be used to store the newly read bytes from the origin.

What is `bos.flush()`?

- Writes buffered data to the file immediately.
- Ensures no data is left in memory.
- Use it to avoid data loss before closing the stream.

Key Point: `flush()` pushes data from memory to the file.

With vs Without Buffer

Writing Data Comparison

- **Without Buffer:**
  - Writes data directly to the file.
  - Slower due to frequent disk I/O operations.
- **With Buffer:**
  - Writes data to memory first, then flushes to the file.
  - Faster for large data because of fewer disk writes.

Key Difference: Buffered streams improve performance by reducing the number of I/O operations.

Code Example: Without Buffer

```java
import java.io.*;

public class FileStreamExampleWithoutBuffer {
    public static void main(String[] args) {
        File file = new File("C:/Users/Qiong/IdeaProjects/Java_class/CM5/src/
            FileStreamExample.txt");

        // Write to file without buffering
        try (FileOutputStream fos = new FileOutputStream(file,true)) {
            fos.write("Hello, I am going to add a new scentence.".getBytes())
                ;
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Read from file without buffering
        try (FileInputStream fis = new FileInputStream(file)) {
            int data;
            while ((data = fis.read()) != -1) {
                System.out.print((char) data);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Code Example: With buffer

```
import java.io.*;

public class FileStreamExampleWithBuffer {

    public static void main(String[] args) {
        File file = new File("C:/Users/Qiong/IdeaProjects/Java_class/CM5/src/
            FileStreamExample.txt");
        // Write to file without buffering
        try (FileOutputStream fos = new FileOutputStream(file,true);
            BufferedOutputStream bos = new BufferedOutputStream(fos)){

            bos.write("Hello, I am going to add a new scentence again.".
                getBytes());
            bos.flush();
        } catch (IOException e) {
            e.printStackTrace();
        }
        // Read from file without buffering
        try (FileInputStream fis = new FileInputStream(file)) {
            int data;
            while ((data = fis.read()) != -1) {
                System.out.print((char) data);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

Deal with JSON file

In Android programming, JSON and XML are the most commonly used data formats.

- JSON:
    - Simple, fast, and modern.
    - Preferred for most Android apps.
    -
    - Interacting with web APIs.
- XML:
    - Verbose but powerful.
    - Still used for legacy systems or configuration files.

Create and Modify JSON File in Java

Before enable JSON format, we need to download `Gson` library

1. Download Gson Library:`https://search.maven.org/artifact/com.google.code.gson/gson/2.11.0/jar?eh=`
2. Add JAR to Project:
   - IntelliJ IDEA:
     - Right-click project → `Open Module Settings`.
     - Go to `Libraries` → Add JAR file.
   - Gradle (Optional):
     - Add: `implementation 'com.google.code.gson:gson:2.8.9'`

Create a JSON File

Create a JSON File

- Use `JsonObject` to store data.
- Write it to a file using `FileWriter`.

```java
import com.google.gson.*;

public class JsonFileExample {
    public static void main(String[] args) {
        String filePath = "C:/Users/Qiong/IdeaProjects/Java_class/CM5/src/dataJson.json";
        Gson gson = new Gson();

        // Step 1: Create a JSON file
        JsonObject jsonObject = new JsonObject();
        jsonObject.addProperty("name", "Alice");
        jsonObject.addProperty("age", 25);

        try (FileWriter writer = new FileWriter(filePath)) {
            gson.toJson(jsonObject, writer); //
            System.out.println("JSON file created: " + filePath);
        } catch (IOException e) {
            e.printStackTrace();
        }
        }}
```

Modify a JSON File

Code to Modify JSON

- Read JSON using `JsonParser`.
- Add or remove fields.

```java
// step 2: read and print the JSON file, put it inside the previous class
    brace.
try (FileReader reader = new FileReader("data.json")) {
    JsonObject jsonObject = JsonParser.parseReader(reader).getAsJsonObject();

    jsonObject.addProperty("city", "New York"); // Add
    jsonObject.remove("age"); // Remove

    try (FileWriter writer = new FileWriter("data.json")) {
        new Gson().toJson(jsonObject, writer);
        System.out.println("JSON updated.");
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

Deal with XML file

Steps to Use XML in Java, similiar to json

1. Download Jackson XML Library: `https://github.com/FasterXML/jackson-dataformat-xml`.
2. Add JAR to Project:
    - IntelliJ IDEA:
        - Right-click project → `Open Module Settings`.
        - Go to `Libraries` → Add JAR file.
    - Gradle (Optional):
        - Add: `implementation 'com.fasterxml.jackson.dataformat:jackson-dataformat-xml:2.15.2'`

Database Operation.

Database Operations

Database is everywhere:

- Data Storage: Save user data, app settings, and application states.
- Performance Optimization: Efficiently query and manipulate data for faster app performance.
- Offline Support: Ensure app functionality even without an internet connection.

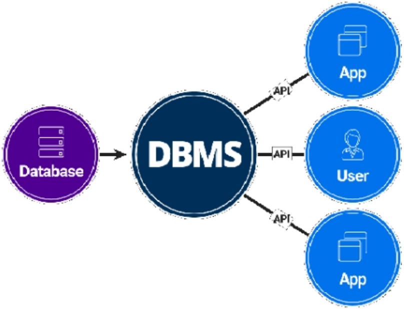
Application Scenarios:

1. When you design a chat app: we need to store and retrieve user messages and conversations.
2. When you design a E-Commerce Apps: we need to manage products, orders, and user accounts.
3. Further, we always save user preferences and custom settings.

Components of a Database System

Database System Components:

- Database (DB): Stores data in an organized format.
- Database Management System (DBMS): Software to manage and interact with the database.
- Application System (AS): The end-user application that accesses the database.
- Database Administrator (DBA): Responsible for database maintenance and security.



Types of Databases

1. Hierarchical Database: Organizes data in a tree-like structure.
   - Example: IBM Information Management System (IMS).
   - Use Case: Banking systems for storing account details.
2. Network Database: Represents data as records connected by links.
   - Example: Integrated Data Store (IDS).
   - File Type: ‘.db‘ or proprietary formats.
   - Use Case: Telecom databases for managing connections and call data.
3. Relational Database (RDBMS): Uses tables with rows and columns; most common type.
   - Examples: MySQL, PostgreSQL, Oracle Database.
   - File Types: ‘.sql‘, ‘.db‘, ‘.sqlite‘, ‘.accdb‘ (for Access), ‘.mdb‘ (for older Access).
   - Use Case: E-commerce platforms for managing products, orders, and customers.
4. Object-Oriented Database: Stores data as objects, similar to programming languages.
   - Examples: ObjectDB, db4o.
   - File Type: ‘.odb‘, ‘.bin‘ (binary serialized objects), or custom formats.
   - Use Case: Multimedia applications for managing complex objects like videos and images.

Java Database Connectivity

Java Database Connectivity (JDBC)

- To develop an application, we use JDBC to interact with the database.
- JDBC enables us to:
  - Retrieve records matching specific criteria.
  - Add, update, or delete data from the database.

Java Database Connectivity (JDBC)

What is JDBC?

- Java Database Connectivity (JDBC) is a Java API used to execute SQL statements.
- Acts as a **bridge** between a Java application and a database.

Key Tasks Performed by JDBC:

1. Establish a Connection: Connect to the database using a driver.
2. Send SQL Statements: Execute SQL queries or updates.
3. Process Results: Handle and utilize data retrieved from the database.

Pay attention:

- JDBC does not directly access the database.
- It relies on database vendor-specific **drivers** to establish communication.

JDBC (The Choice of Different Drivers)

- SQLite:
  - Use Android's built-in `android.database.sqlite` API.
  - Ideal for local storage within the app.
  - No additional setup is required.
- MySQL:
  - Use `MySQL Connector/J`.
  - Suitable for apps that need to interact with a remote MySQL server.
  - Commonly used in server-side components for data storage.
- PostgreSQL:
  - Use `PostgreSQL JDBC Driver`.
  - Preferred for advanced features like JSON support and data integrity.
  - Suitable for robust and scalable remote databases.
- Firebase:
  - Use the official `Firebase SDK` instead of JDBC.
  - Provides real-time database synchronization.
  - Great for chat applications or apps requiring real-time updates.

Summary:

- Choose SQLite for local data, and MySQL/PostgreSQL for remote databases.
- Use Firebase for apps needing real-time features or cloud storage.

Classes and Interfaces Commonly Used in JDBC

Classes and Interfaces Commonly Used in JDBC.

- `DriveManager` class
- `Connection` interface
- `Statement` interface
- `PreparedStatement` interface
- `ResultSet` interfance

DriverManager Class Example

```java
import java.sql.Connection;
import java.sql.DriverManager;

public class DatabaseExample {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/mydatabase";
        String username = "root";
        String password = "password";

        try {
            // Load the MySQL JDBC driver
            Class.forName("com.mysql.cj.jdbc.Driver");

            // Set a login timeout (in seconds)
            DriverManager.setLoginTimeout(10); // 10 seconds

            // Establish a connection
            Connection conn = DriverManager.getConnection(url, username,
                 password);
            System.out.println("Database connected successfully!");

            // Close the connection
            conn.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Connection & Statement Interface Example

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
import java.sql.ResultSet;

public class JDBCExample {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/mydatabase";
        String username = "root";
        String password = "password";

        try (Connection conn = DriverManager.getConnection(url, username,
            password);
             Statement stmt = conn.createStatement()) {

            // Execute a query
            ResultSet rs = stmt.executeQuery("SELECT * FROM Users");

            // Process the results
            while (rs.next()) {
                System.out.println("User: " + rs.getString("name"));
            }

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

JDBC Practice Exercise: Student Management

Task Description:

- Create a class named JDBC4.
- Implement methods to perform database operations on a table named Students.
- The Students table has the following columns:
  - id (INT, Primary Key)
  - name (VARCHAR)
  - age (INT)

Requirements:

1. Initialize a database connection (initConnection).
2. Close the database connection (closeConnection).
3. Query all student records (queryAllStudents).
4. Add a new student record (addStudent).
5. Update a student's name based on their ID (updateStudentName).
6. Delete a student record based on their ID (deleteStudent).

Expected Output:

- Students added to the table.
- Updated student names displayed correctly.
- Deleted student records no longer appear in queries.

JDBC Practice Exercise: Student Management

Hints:

- Use JDBC classes like `DriverManager`, `Connection`, and `PreparedStatement`.
- Use the SQL statements: `SELECT`, `INSERT`, `UPDATE`, and `DELETE`.
- Ensure database credentials and table are correctly set up.

JDBC4 Class Overview

Step 1: Initialize Database Connection

```java
import java.sql.*;
public class JDBC4 {
    private static final String URL = "jdbc:mysql://localhost:3306/mydata";
    private static final String USER = "root";
    private static final String PASSWORD = "password";

    private Connection conn;

    // 1. Initialize database connection
    public void initConnection() {
        try {
            conn = DriverManager.getConnection(URL, USER, PASSWORD);
            System.out.println("Database connection initialized.");
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
    // 2. Close database connection
    public void closeConnection() {
        try {if (conn != null) {
                conn.close();
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
```

Query and Add Operations

Method: Query All Students

```java
public void queryAllStudents() {
    String sql = "SELECT * FROM Students";
    try (Statement stmt = conn.createStatement();
         ResultSet rs = stmt.executeQuery(sql)) {

        System.out.println("ID\tName\tAge");
        while (rs.next()) {
            int id = rs.getInt("id");
            String name = rs.getString("name");
            int age = rs.getInt("age");
            System.out.println(id + "\t" + name + "\t" + age);
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

Method: Add a New Student

```java
public void addStudent(int id, String name, int age) {
    String sql = "INSERT INTO Students (id, name, age) VALUES (?, ?, ?)";
    try (PreparedStatement pstmt = conn.prepareStatement(sql)) {
        pstmt.setInt(1, id);
        pstmt.setString(2, name);
        pstmt.setInt(3, age);
        int rows = pstmt.executeUpdate();
```

Update and Delete Operations

Method: Update Student Name

```java
public void updateStudentName(int id, String newName) {
    String sql = "UPDATE Students SET name = ? WHERE id = ?";
    try (PreparedStatement pstmt = conn.prepareStatement(sql)) {
        pstmt.setString(1, newName);
        pstmt.setInt(2, id);
        int rows = pstmt.executeUpdate();
        System.out.println(rows + " student(s) updated.");
    } catch (SQLException e) {
        e.printStackTrace();
    }}
```

Method: Delete a Student

```java
public void deleteStudent(int id) {
    String sql = "DELETE FROM Students WHERE id = ?";
    try (PreparedStatement pstmt = conn.prepareStatement(sql)) {
        pstmt.setInt(1, id);
        int rows = pstmt.executeUpdate();
        System.out.println(rows + " student(s) deleted.");
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

Main Method for Testing

Main Method: Test All Operations

```java
public static void main(String[] args) {
    JDBC4 jdbc = new JDBC4();

    // Initialize connection
    jdbc.initConnection();

    // Add new students
    jdbc.addStudent(1, "Alice", 20);
    jdbc.addStudent(2, "Bob", 22);

    // Query all students
    jdbc.queryAllStudents();

    // Update a student's name
    jdbc.updateStudentName(1, "Alicia");

    // Delete a student
    jdbc.deleteStudent(2);

    // Close connection
    jdbc.closeConnection();
}
```

Database design for Android

Data storage

Android provides several options to save persistent application data.

Your data storage options are the following:

- Traditional files: Internal (private data on the device memory) or external (public data on shared external storage, e.g. SD card)
- Shared Preferences: Store small amount of private data in key-value pairs.
- SQLite Databases: Store structured data in a private database.
- Network: Store data on the web with your own network server.

Internal vs. External storage

Android devices have two file storage areas: "internal" and "external" storage.

Internal storage:

- Always available.
- By default, files saved here are accessible by your app only (unless you specify otherwise)
- When the user uninstalls your app, the system removes all your app's files from internal storage.

External storage:

- 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 `getExternalFileDir()`.

Internal storage: Files

- 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.
- When the user uninstalls your application, these files are removed.

Write data

```
    File directory = getFilesDir();
    File file = new File(directory, FILENAME);
    FileWriter fw = new FileWriter(file);
    PrintWriter writer = new PrintWriter(fw);
    writer.println("hello world!");
    writer.close();
```

Internal storage: Files

- 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.
- When the user uninstalls your application, these files are removed.

Read data

```
    File directory = getFilesDir();
    File file = new File(directory, FILENAME);
    FileReader fr = new FileReader(file);
    BufferedReader inStream = new BufferedReader(fr);
    StringBuilder stringBuilder = new StringBuilder();
    String inString;
    while ((inString = inStream.readLine()) != null){
        stringBuilder.append(inString);}
    inStream.close();
    String fileContents = stringBuilder.toString();
    Toast.makeText(this, "Data: "+fileContents, Toast.LENGTH_SHORT)
        .show();
```

Shared Preferences

- Simple way to store a small amount of data
- Private by default but can be shared with other applications
- Key-value pairs of simple data types
- boolean, float, int, long, and string
- XML file

Shared Preferences: example

```
SharedPreferences sp =
    getApplicationContext().getSharedPreferences("mypref"
        , Context.MODE_PRIVATE);

// put data
SharedPreferences.Editor editor = sp.edit();
editor.putString("MY_NAME", "John");
editor.commit();

// get data
String name = sp.getString("MY_NAME", "no name");
```

where `"no name"` is the default name.

SQLite

As previously discussed, working with a database requires selecting an appropriate driver. In the following sections, we will focus on using SQLite.

SQLite:

- Use Android's built-in `android.database.sqlite` API.
- Ideal for local storage within the app.
- No additional setup is required.

Content Provider

- A typical SQLite database is **private** to the application which creates it. If you want to share data with other applications you should use a **ContentProvider**.
- A content provider is a component that exposes read/write access to application data, subject to whatever restrictions you want to impose.
- Android itself includes content providers that manage data such as audio, video, images, and personal contact information.

SQLite storage types

- 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 (Binary Large Object) - the value is a blob of data, stored exactly as it was input.

class SQLiteDatabase

- Similar to JDBC (Java Database Connectivity)
- Contains the methods for: creating, opening, closing, inserting, updating, deleting and querying an SQLite database

Create a Database

Create a subclass of SQLiteOpenHelper and override `onCreate()`:

```java
public class DBHelper extends SQLiteOpenHelper {
    private static final String DATABASE_NAME = "shopping_list.db";
    private static final int DATABASE_VERSION = 1;
    public DBHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL("CREATE TABLE IF NOT EXISTS shopping_items (_id
            INTEGER PRIMARY KEY AUTOINCREMENT, item_name TEXT)");
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int
        newVersion) {
        db.execSQL("DROP TABLE IF EXISTS shopping_items");
        onCreate(db);
    }
}
```

Action queries

Every time you write to the database

- Grab an instance of your SQLiteOpenHelper
- Call `getWritableDatabase()`
- This returns a SQLiteDatabase object that represents the database and provides methods for SQLite operations.
- When your app is destroyed, close database by calling `close()`

```
    MyDatabaseHelper helper = new MyDatabaseHelper();
    SQLiteDatabase db = helper.getWritableDatabase();
    ...
    db.insert(...); // or update or delete
    ...
    db.close();
```

Action query: Insert

- long insert(String table, String nullColumnHack, ContentValues values)
- Returns the row ID of the newly inserted row, or -1 if an error occurred

```
ContentValues values = new ContentValues();
values.put("item_name", itemName);
long newRowId = db.insert("shopping_items", null, values
    );
```

Action query: Update

- int update(String table, ContentValues values, String whereClause, String[ ] whereArgs)
- Returns the number of rows affected

```
ContentValues values = new ContentValues();
values.put("item_name", newItemName);
String[] selectionArgs = { String.valueOf(itemId) };
int rowsAffected = db.update("shopping_items", values, "
    _id=?", selectionArgs);
```

Action query: Delete

- int delete(String table, String whereClause, String[] whereArgs)
- Returns the number of rows affected

```
// Delete row with id = 1
String[] selectionArgs = { String.valueOf(1) };
db.delete("shopping_items", "_id = ?", selectionArgs);
```

Exercise: More on shopping list

Write an interactive shopping List. Enter text in Dialog windows.

- Action: "Edit name"
- Floating Action Button: "Add item"
- User name stored in the shared preferences
- Jobs stored in a SQLite database

Add the following features.

- Long click deletes an item
- The user can dump the current jobs to a text file ("export"), and 'export"), and restore them at a later stage ("import").