# Object Oriented and Java Programming

## Course 5

Qiong Liu

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

October 2024

# Contents

# What You'll Learn Today

## Embedded Files: Java I/O

- Basics of JAVA I/O
- Common uses cases: txt, json, xml, bat

## External Data center: SQL Databases

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

# 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 designing a chat app: we need to store and retrieve user messages and conversations.
2. When designing a E-Commerce Apps: we need to manage products, orders, and user accounts.
3. Save user preferences and custom settings.

# Outline

# Java I/O (Embedded Files)

**Java I/O (Embedded Files)**:

- Handles local files like `JSON`, `XML`, and `.txt`.
- Used for lightweight, file-based storage.
- Common use cases:
  - TXT: Store simple, unstructured data.
  - JSON: Store structured data like user preferences.
  - XML: Configuration files for applications.
  - Logs: Writing error or debug logs.
  - Bat: Useful for setting up development environments.
- Does not require external services.

Advantages:

- Simple and easy to implement.
- Suitable for small-scale data storage.

**Limitation:** Not efficient for managing complex data relationships.

# SQL Databases (External Data Center)

Key Features:

- Operates on external databases like SQLite, MySQL.
- Best for managing structured, relational data.
- Common use cases:
  - User Accounts: Store usernames, passwords, and profiles.
  - Product Management: Maintain product catalogs in e-commerce apps.
  - Order History: Record and query user purchases.

Advantages:

- Efficient for complex queries (CRUD operations).
- Scalable for multi-user environments.
- Reliable for large-scale data.

**Limitation:** Requires setup and external service for data storage.

# When to Use Java I/O or SQL?

Decision Table:

- Use **Java I/O (Embedded Files)** for:
  - Lightweight, local data storage.
  - Simple configurations or logs (`JSON`, `XML`).
- Use **SQL Databases** for:
  - Complex, relational data with structured queries.
  - Scenarios requiring scalability and shared access.

Comparison:

| **Feature** | **Java I/O** | **SQL Database** |
|---|---|---|
| Storage Type | Local files | External service |
| Complexity | Simple | Advanced |
| Efficiency for Queries | Low | High |
| Best for | Configurations, Logs | Relational Data |

# 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

## 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.

## 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.

# File Class

## File Class – fix location of your file

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.

# File Class: Constructors

## File Class: Constructors

- Type 2: `File(String parent, String child)`

```
1 String parentPath = "C:/Users/Qiong/Documents";
2 String childPath = "example.txt";
3
4 // Create File object
5 File file = new File(parentPath, childPath);
```

- Type 3: `File(File parent, String child)`

```
1 File parentDir = new File("C:/Users/Qiong/Documents");
2 String childPath = "example.txt";
3
4 // Create File object
5 File file = new File(parentDir, childPath);
```

# File Class: Code Example

```
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());
                }
```

# 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 to deal with stream: 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.
- 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

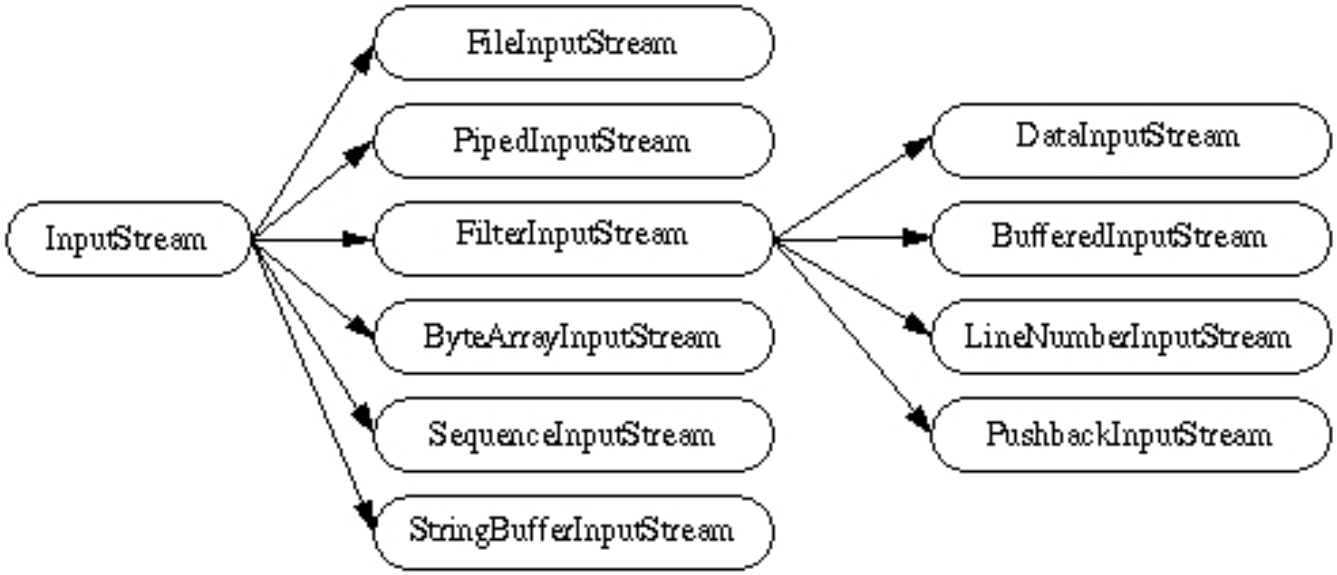



Figure: InputStream Class

## 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.

# 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.

## Error Handling

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

# 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)
```

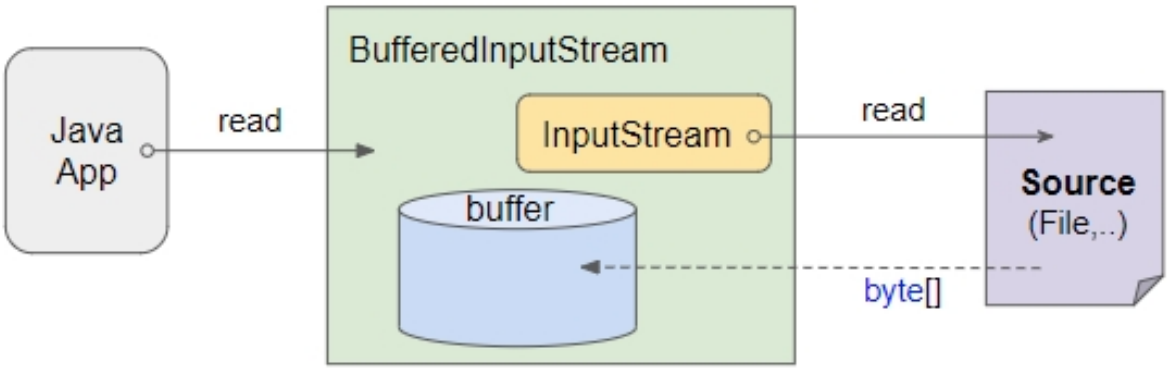



Figure: Buffer InputSteam (Similar for the Buffer OutputSteam)

# 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).

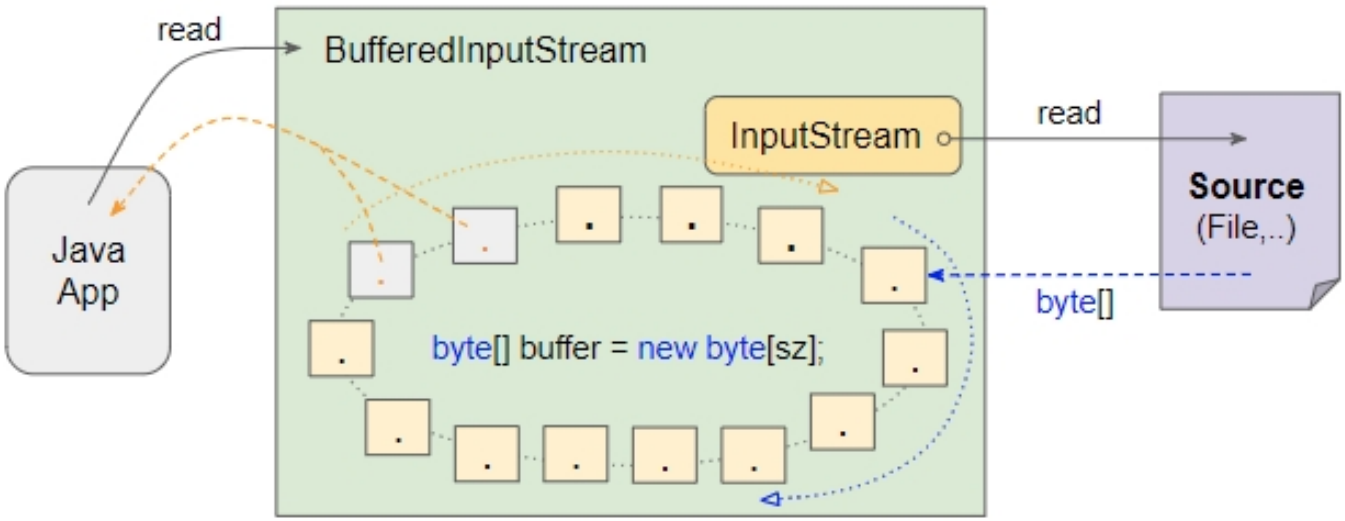



Figure: `BufferedInputStream` reads bytes from buffer array and frees the read positions. Freed positions will be used to store the newly read bytes from the origin

# bos.flush()

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

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

# 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

```
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) {
```

# 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) {
```

# 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:
     - 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`.

```
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

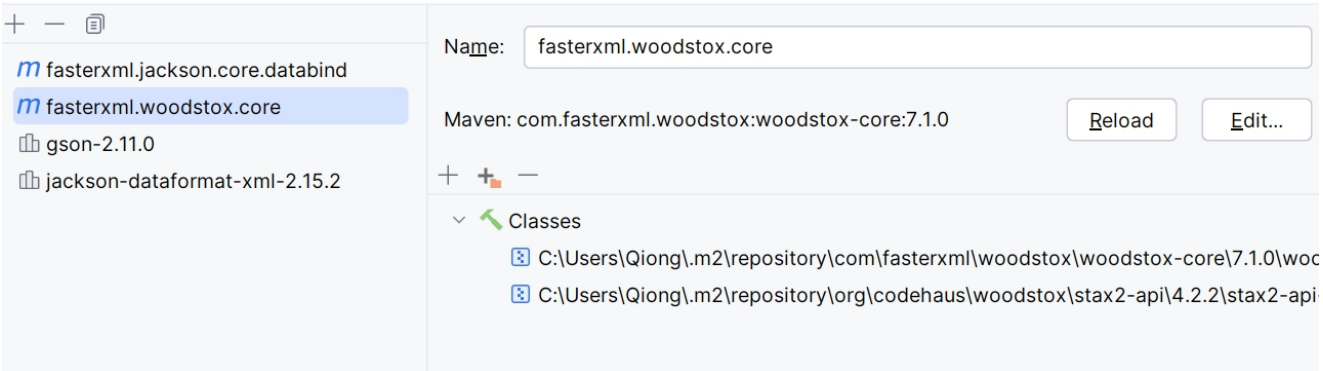
- Read JSON using `JsonParser`.
- Add or remove fields.

```
// 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`
3. you can also add Maven file to enable XML handling:



# Deal with XML file

## Create, write

```
import com.fasterxml.jackson.dataformat.xml.XmlMapper;
import java.io.File;
import java.io.IOException;

public class XmlModifyExample {
    public static void main(String[] args) throws IOException {
        XmlMapper xmlMapper = new XmlMapper();

        // Specify the XML file path
        File file = new File("/src/data.xml");
        // Check if the file exists and is valid
        if (!file.exists() || file.length() == 0) {
         System.out.println("XML file not found or is empty.
             Creating a new file with default content...");
            // Create default content
            Students defaultStudent = new Students("Default Name",
                20);
            // Write default content to file
            String defaultXml = xmlMapper.writeValueAsString(
                defaultStudent);
            System.out.println("Generated XML content:" +
```

# Deal with XML file

## Modify

```
import com.fasterxml.jackson.dataformat.xml.XmlMapper;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;

public class XmlModifyExample {
    public static void main(String[] args) throws IOException {
        XmlMapper xmlMapper = new XmlMapper();

        // Specify the XML file path
        File file = new File("C:/Users/Qiong/IdeaProjects/
            Java_class/CM5/src/data.xml");
        // Read the XML file
        Students student = xmlMapper.readValue(new FileInputStream(
            file), Students.class);
        // Modify the object
        student.age = 26; // Change age
        student.name = "Alice Updated"; // Change name
        // Write the updated object back to the XML file
        xmlMapper.writeValue(file, student);
        System.out.println("XML file updated.");
```

# ObjectInputStream & ObjectOutputStream

If you need to deal with object-oriented files, like `.bat`, you can handle it by using `ObjectInputStream` and `ObjectOutputStream` from Java IO packages.

- Serialization: Converts an object into a byte stream (ObjectOutputStream).
- Deserialization: Reconstructs an object from a byte stream (ObjectInputStream).
- The object must implement the `Serializable` interface for serialization.
- Fields marked as `transient` are not serialized.
- @Override methods like: `void writeObject(Object obj)`, `Object readObject()`, `void close()`...

# serialVersionUID

`serialVersionUID`

- A unique identifier for a `Serializable` class.
- Used to ensure match during object serialization and deserialization.
- If `serialVersionUID` changes, deserialization fails with `InvalidClassException`.
- Declared as:
  - `private static final long serialVersionUID = 1L;`

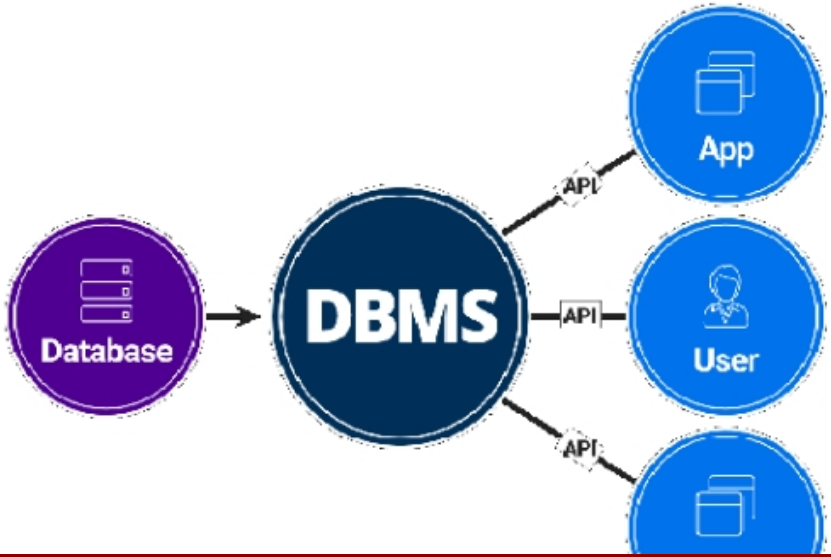
# Outline

# External Data Center:
# (Structured Query Language)SQL Databases

# 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).
   - 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

# Database Interaction in Java and Android

How to interact with databases?

- In IntelliJ (Java Development):
  - Use JDBC (Java Database Connectivity).
  - Suitable for databases like MySQL, PostgreSQL.
- In Android Development:
  - Local Databases: SQLite or Room (Jetpack Library).
  - **Remote Databases:** Use Retrofit or REST APIs to interact with servers.

# Java Database Connectivity (JDBC)

## JDBC

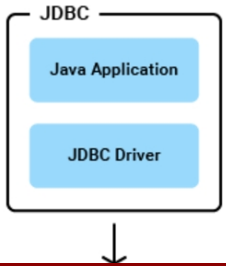
Java Database Connectivity (JDBC) is a Java API used to execute SQL statements. Acts as a **bridge** between a Java application and a database.

Key Tasks Performed by JDBC:

1. Establish a Connection: Connect to the database using a driver.
2. Send SQL Statements:
3. Process Results:

**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.
- 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 for remote databases.
- Use Firebase for apps needing real-time features or cloud storage.

In the next,
we try to establish the link between MySQL and your Java project!

# Step 1: Download and Install MySQL

**Steps to download and install MySQL:**

1. Go to the official MySQL website: https://dev.mysql.com/downloads/.
2. Download the appropriate version of MySQL Installer for your operating system.
3. Install MySQL:
   - Choose **Server Only** or **Developer Default** installation.
   - Configure the root password during setup.
4. Ensure the MySQL server is running.

**Verify Installation:**

- Open the MySQL Command Line Client.
- Login using the command: `mysql -u root -p`.
- To check your username: `SELECT user, host FROM mysql.user;`
- `SELECT * FROM users;`, show your table.

# Step 2: Configure MySQL Database and Table

**Steps to create the database and table:**

1. Open MySQL Command Line Client or Workbench.
2. Execute the following commands:

```
CREATE DATABASE user_management;

USE user_management;

CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    age INT NOT NULL
);
```

- Create a database named `user_management`.
- Create a table named `users` to store user information.
- The keyword `NOT NULL` ensures that a column cannot have empty (null) values.

# Step 3: Download and Configure MySQL JDBC Driver

## Method 1: Add MySQL JDBC driver to IntelliJ:

1. Go to the Maven Repository: https://search.maven.org/.
2. Search for `mysql-connector-java`.
3. Add the dependency to your `pom.xml` file:

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.40</version>
</dependency>
```

## Method 2: Manual Alternative (we will use this way):

- Download the JAR file from MySQL Connector/J.
- Add the JAR to IntelliJ's `Project Structure > Libraries`.

You now have all the tools to manage an external database using a Java project. Let's create several classes to handle database connections and modifications:

- `DatabaseConnection`: establishes a connection to the database.
- `UserDAO`: provides methods such as `addUser`, `updateUser`, `deleteUser`, etc.
- `Main`: tests the functionality of your code.

# DatabaseConnection

Database Connection Class:

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

public class DatabaseConnection {
    private static final String URL = "jdbc:mysql://localhost:3306/user_management";
    private static final String USERNAME = "root";
    private static final String PASSWORD = "your_password";

    public static Connection getConnection() throws SQLException {
        return DriverManager.getConnection(URL, USERNAME, PASSWORD);
    }
}
```

**Note:** Replace `your_password` with your actual MySQL password, which I do not know.

# UserDAO

## Operation Class:

```java
import java.sql.*;
import java.util.ArrayList;
import java.util.List;

public class UserDAO {
    // add user
    public void createUser(String name, String email, int age) throws SQLException {
        String sql = "INSERT INTO users (name, email, age) VALUES (?, ?, ?)";
        try (Connection connection = DatabaseConnection.getConnection();
             PreparedStatement statement = connection.prepareStatement(sql)) {
             // your code
        }
    }
    // delete user
    public void deleteUser(int id) throws SQLException {
        String sql = "DELETE FROM users WHERE id = ?";
        try (Connection connection = DatabaseConnection.getConnection();
             PreparedStatement statement = connection.prepareStatement(sql)) {
            statement.setInt(1, id);
            statement.executeUpdate();
            System.out.println("User deleted successfully.");
        }
    }
}
```

# Main

## Main Class (test class):

```
import java.sql.SQLException;
public class Main {
    public static void main(String[] args) {
        UserDAO userDAO = new UserDAO();

        try {
            // creat user
            userDAO.createOrUpdateUser("Alice", "alice@example.com", 30);
            userDAO.createOrUpdateUser("Bob", "bob@example.com", 25);
            // ask user
            System.out.println("All users:");
            userDAO.getAllUsers().forEach(System.out::println);
            // update user
            userDAO.updateUser(1, "Alice Updated", 32);
            // delete user
            userDAO.deleteUser(2);

        } catch (SQLException 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. Query all student records (`queryAllStudents`).
3. Add a new student record (`addStudent`).
4. Delete a student record based on their ID (`deleteStudent`).

## Expected Output:

- Students added to the table.
- Updated student names displayed correctly.