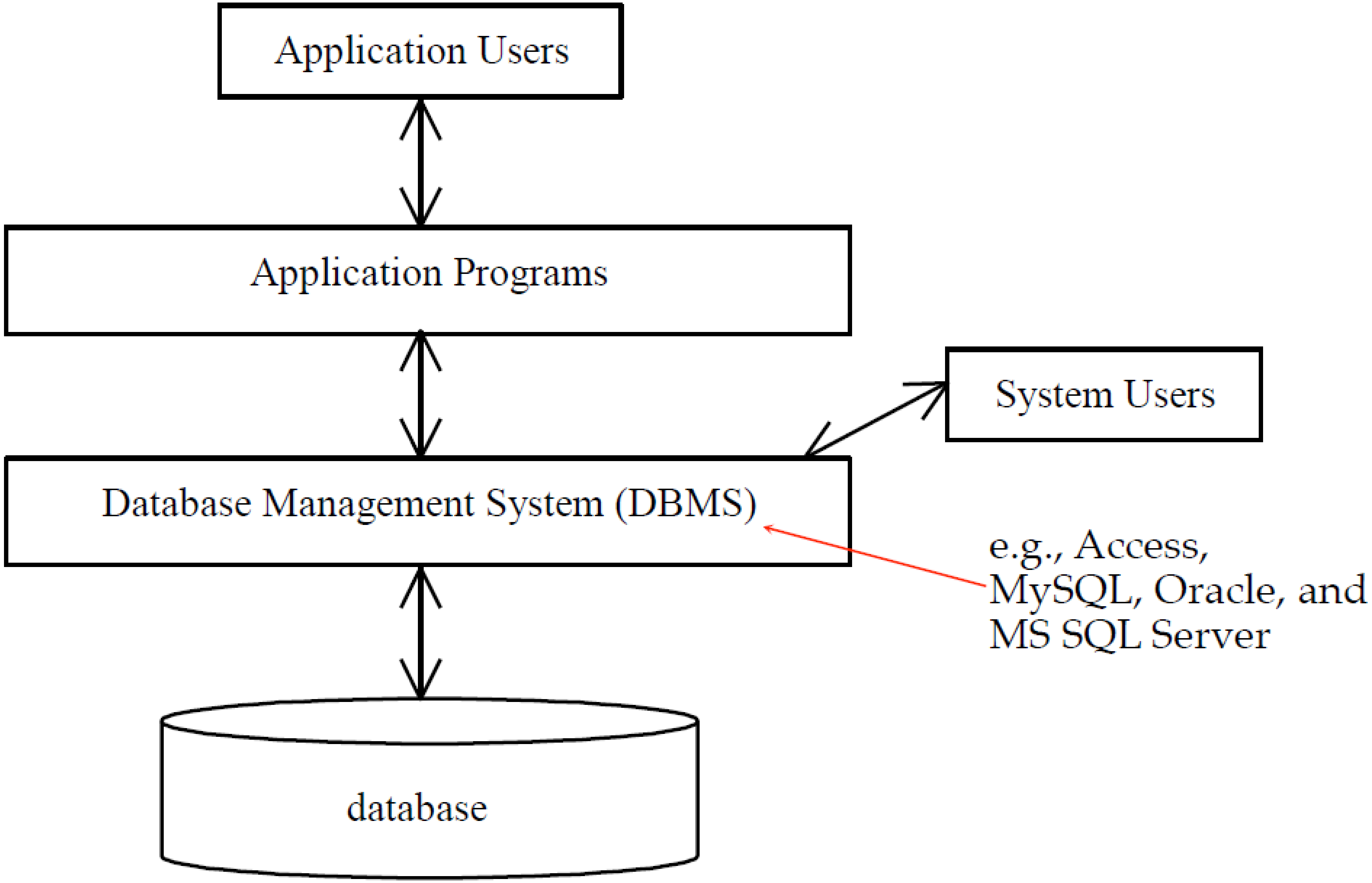
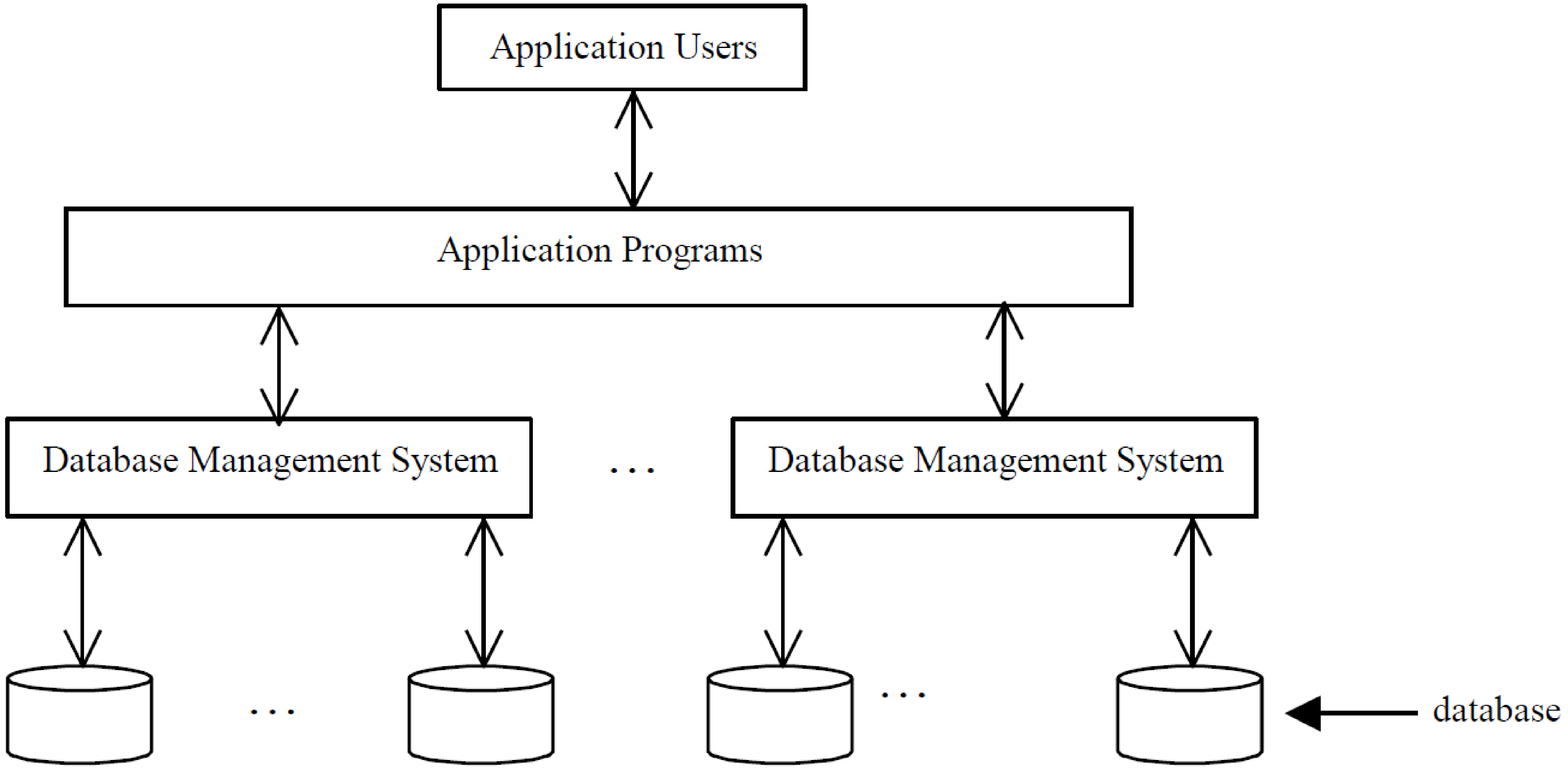
Java Database Programming

- Java provides the API for developing **database applications** that works with any relational **database systems**.
- **Database systems** not only **store** data, they also provide means of **accessing**, **updating**, **manipulating**, and **analyzing** data.
- Your social security information is updated periodically
- You can register for courses online.
- You have to know SQL. SQL is the standard database language for defining and accessing databases.


Application Users
Application Programs
System Users
Database Management System (DBMS)
e.g., Access, MySQL, Oracle, and MS SQL Server
database


Application Users
Application Programs
Database Management System
. . .
Database Management System
. . .
. . .
database

SQL

- To access or write applications for database systems, you need to use the Structured Query Language (SQL).
- SQL is the universal language for accessing relational database systems.
- Application programs may allow users to access a database without directly using SQL, but these applications themselves must use SQL to access the database.

Why Java for Database Programming?

- First, Java is platform independent. You can develop platform-independent database applications using SQL and Java for any relational database systems.
- Second, the support for accessing database systems from Java is built into Java API, so you can create database applications using all Java code with a common interface.
- Third, Java is taught in almost every university either as the first programming language or as the second programming language.

Database Applications Using Java

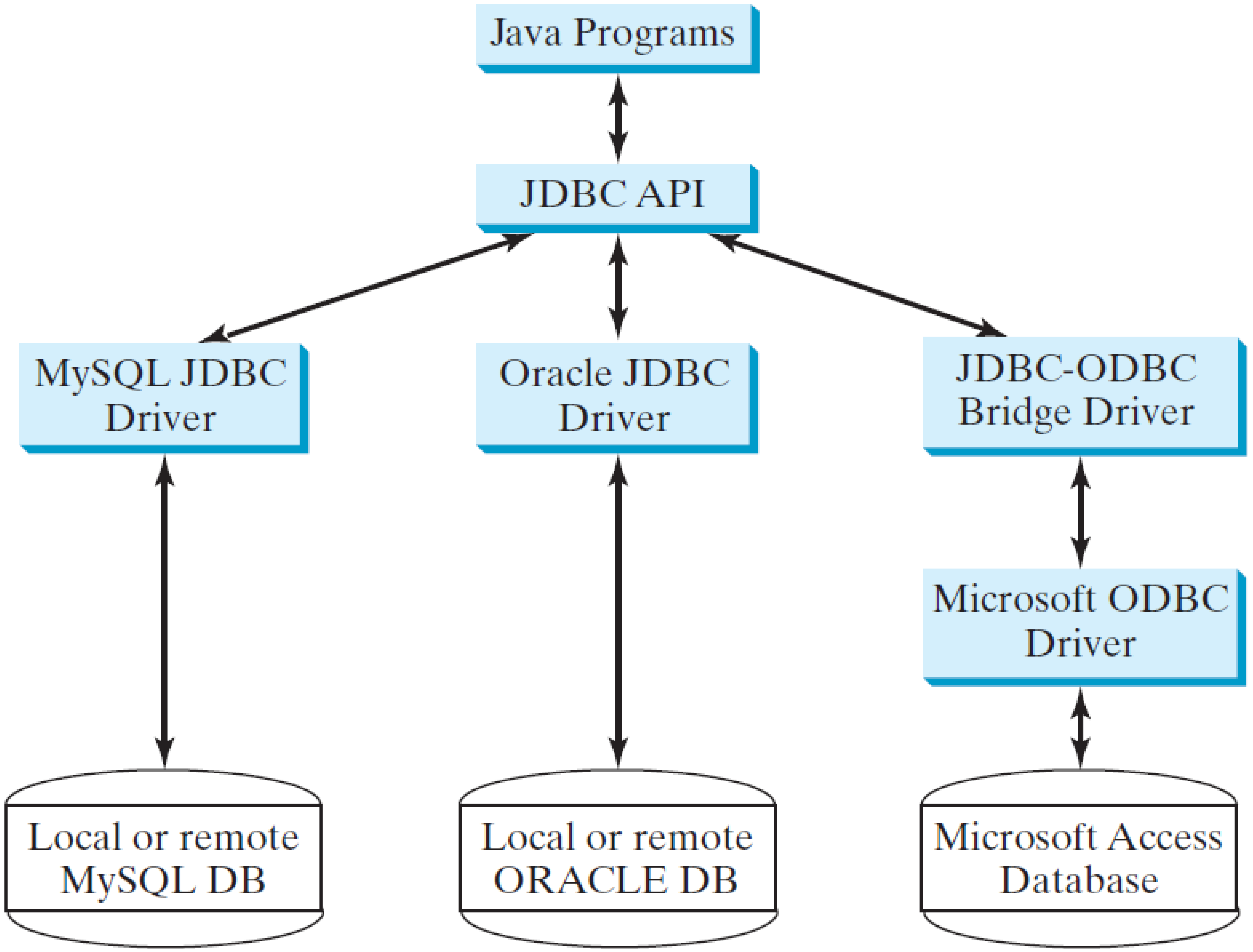
- GUI
- Client/Server
- Server-Side programming

JDBC

- JDBC is the Java API for accessing relational database.
- The Java API for developing Java database applications is called JDBC.
- JDBC is the name of a Java API that supports Java programs that access relational databases.
- JDBC is not an acronym, but it is often thought to stand for Java Database Connectivity.

JDBC (Cont.)

- JDBC provides Java programmers with a **uniform interface** for accessing and manipulating relational databases.
- Using the JDBC API, applications written in the Java programming language can execute SQL statements, retrieve results, present data in a user-friendly interface, and propagate changes back to the database.
- The JDBC API can also be used to interact with multiple data sources.


Java Programs
JDBC API
MySQL JDBC Driver
Oracle JDBC Driver
JDBC-ODBC Bridge Driver
Microsoft ODBC Driver
Local or remote MySQL DB
Local or remote ORACLE DB
Microsoft Access Database

JDBC (Cont.)

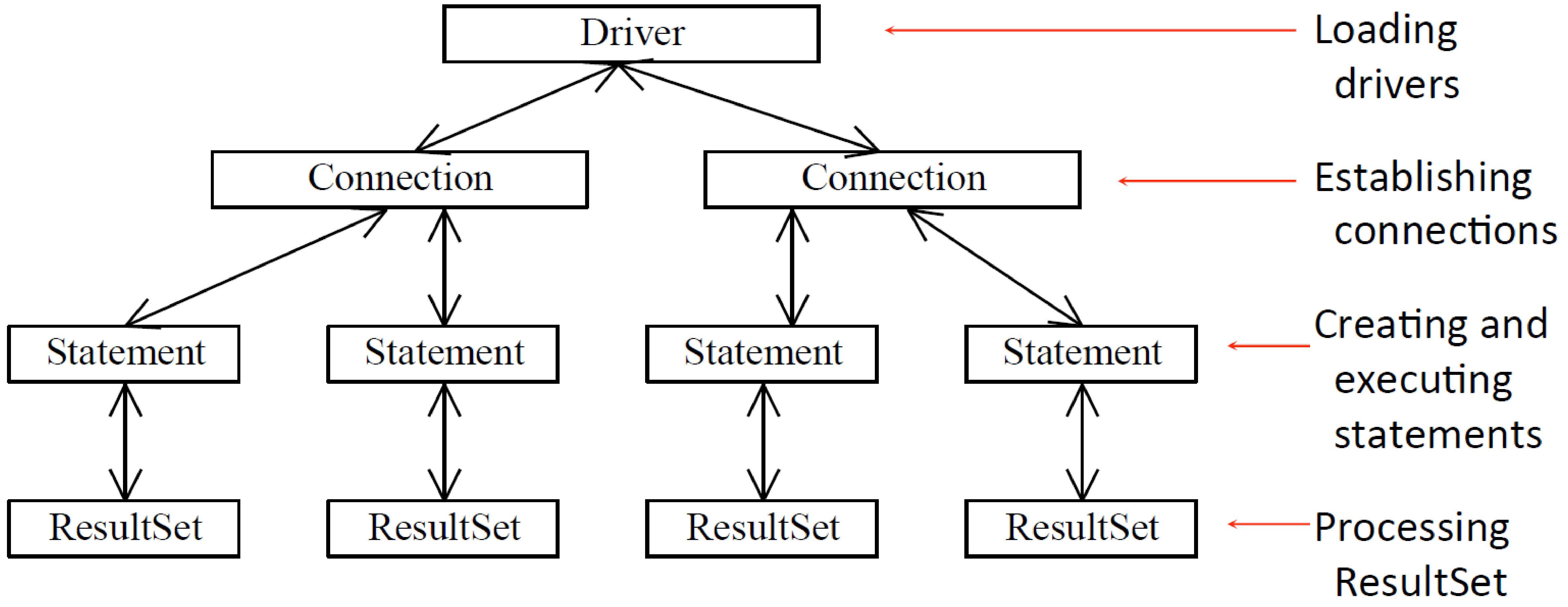
- The JDBC API is a set of Java interfaces and classes used to write Java programs for accessing and manipulating relational databases.
- Since a JDBC driver serves as the interface to facilitate communications between JDBC and a database, JDBC drivers are database specific and are normally provided by the database vendors.
- You need MySQL JDBC drivers to access the MySQL database.

JDBC-ODBC bridge driver

- For the Access database, use the JDBC-ODBC bridge driver included in the JDK.
- ODBC is a technology developed by Microsoft for accessing databases on the Windows platform.
- An ODBC driver is preinstalled on Windows. The JDBC-ODBC bridge driver allows a Java program to access any ODBC data source.

Developing Database Applications Using JDBC

- The JDBC API consists of classes and interfaces for establishing connections with databases, sending SQL statements to databases, processing the results of SQL statements, and obtaining database metadata.
- Four key interfaces are needed to develop any database application using Java: **Driver**, **Connection**, **Statement**, and **ResultSet**.


Driver
Connection
Connection
Statement
Statement
Statement
Statement
ResultSet
ResultSet
ResultSet
ResultSet
Loading drivers
Establishing connections
Creating and executing statements
Processing ResultSet

Process Overview

- A JDBC application loads an appropriate driver using the **Driver interface**, connects to the database using the **Connection interface**, creates and executes SQL statements using the **Statement interface**, and processes the result using the **ResultSet interface** if the statements return results.
- Note that some statements, such as SQL data definition statements and SQL data modification statements, do not return results.

JDBC Application

- The JDBC interfaces and classes are the building blocks in the development of Java database programs.
- A typical Java program takes the following steps to access a database.
    1. Loading drivers.
    2. Establishing connections.
    3. Creating statements.
    4. Executing statements.
    5. Processing results.

Loading Drivers

- An appropriate driver must be loaded using the statement shown below before connecting to a database.

```
Class.forName("JDBCDriverClass");
```

- A driver is a class that implements the **java.sql.Driver** interface.
- If your program accesses several different databases, all their respective drivers must be loaded.

Loading Drivers (Cont.)

- The JDBC-ODBC driver for Access is bundled in JDK.
- The most recent platform independent version of MySQL JDBC driver is in **mysql-connector-java-5.1.40-bin.jar**.
- This file is contained in a ZIP file downloadable from https://dev.mysql.com/downloads/connector/j/.
- If you use an IDE such as Eclipse, you need to add these jar files into the library in the IDE.
- `com.mysql.jdbc.Driver` is a class in **mysql-connector-java-5.1.41-bin.jar**

Establishing Connections

- To connect to a database, use the static method **getConnection(databaseURL)** in the **DriverManager** class, as follows:

```
Connection connection =
DriverManager.getConnection(databaseURL);
```

- **databaseURL** is the unique identifier of the database on the Internet.
- MySQL:

  jdbc:mysql://hostname/dbname

ODBC Establishing Connections

- For an ODBC data source, the **databaseURL** is **jdbc:odbc:dataSource**.
- Suppose a data source named ExampleMDBDataSource has been created for an Access database.
- The following statement creates a **Connection** object:

  Connection connection = DriverManager.getConnection

  ("**jdbc:odbc:ExampleMDBDataSource**");

Establishing Connections – Example

- The **databaseURL** for a MySQL database specifies the host name and database name to locate a database.
- For example, the following statement creates a **Connection** object for the local MySQL database **database1** with username ***mehrnaz*** and password ***12345***:

  Connection connection = DriverManager.getConnection

  (**"jdbc:mysql://localhost/database1"**, **“mehrnaz"**, **"12345"**);

Creating Statements

- If a **Connection** object can be envisioned as a cable linking your program to a database, an object of **Statement** can be viewed as a cart that delivers SQL statements for execution by the database and brings the result back to the program.
- Once a **Connection** object is created, you can create statements for executing SQL statements as follows:

```
Statement statement =
connection.createStatement();
```

Executing Statements

- SQL data definition language (DDL) and update statements can be executed using **executeUpdate(String sql)**

  statement.executeUpdate

  (**"CREATE TABLE Temp (col1 CHAR(5), col2 CHAR(5))"**);

- An SQL query statement can be executed using **executeQuery(String sql)**. The result of the query is returned in **ResultSet**.

  ResultSet resultSet = statement.executeQuery

  (**"SELECT first_name, last_name FROM users WHERE "**

  + "last_name = ‘zhian'");

Processing Results

- The **ResultSet** maintains a table whose current row can be retrieved.
- The initial row position is **null**.
- You can use the **next** method to move to the next row and the various getter methods to retrieve values from a current row.
- For example, the following code displays all the results from the preceding SQL query.

```
while (resultSet.next())

  System.out.println(resultSet.getString(1) +
      " " + resultSet.getString(2) + " " +
      resultSet.getString(3));
```

Processing Results (Cont.)

- The **getString(1)**, **getString(2)**, and **getString(3)** methods retrieve the column values for **first_name**, **last_name**, and **email**, respectively.
- Alternatively, you can use **getString("first_name")**, **getString("last_name")**, and **getString("email")** to retrieve the same three column values.
- The first execution of the **next()** method sets the current row to the first row in the result set, and subsequent invocations of the **next()** method set the current row to the second row, third row, and so on, to the last row.

Developing JDBC Programs

Loading drivers

Establishing connections

Creating and executing statements

Processing ResultSet

Statement to load a driver:

```
Class.forName("JDBCDriverClass");
```

A driver is a class. For example:

Database	Driver Class	Source
Access	sun.jdbc.odbc.JdbcOdbcDriver	Already in JDK
MySQL	com.mysql.jdbc.Driver	Website
Oracle	oracle.jdbc.driver.OracleDriver	Website

The JDBC-ODBC driver for Access is bundled in JDK.

MySQL driver class is in mysqljdbc.jar

Oracle driver class is in classes12.jar

To use the MySQL and Oracle drivers, you have to add mysqljdbc.jar and classes12.jar in the classpath using the following DOS command on Windows:

```
classpath=%classpath%;c:\book\mysqljdbc.jar;c:\book\classes12.jar
```

Developing JDBC Programs

Loading drivers

Establishing connections

Creating and executing statements

Processing ResultSet

```
Connection connection = DriverManager.getConnection(databaseURL);
```

Database	URL Pattern
Access	jdbc:odbc:dataSource
MySQL	jdbc:mysql://hostname/dbname
Oracle	jdbc:oracle:thin:@hostname:port#:oracleDBSID

See Supplement IV.D for creating an ODBC data source

Examples:

For Access:

```
    Connection connection = DriverManager.getConnection
     ("jdbc:odbc:ExampleMDBDataSource");
```

For MySQL:

```
    Connection connection = DriverManager.getConnection
     ("jdbc:mysql://localhost/test");
```

For Oracle:

```
    Connection connection = DriverManager.getConnection
     ("jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl",  "scott", "tiger");
```

Developing JDBC Programs

Loading drivers

Establishing connections

Creating and executing statements

Processing ResultSet

Creating statement:

```
Statement statement = connection.createStatement();
```

Executing statement (for update, delete, insert):

```
statement.executeUpdate
  ("create table Temp (col1 char(5), col2 char(5))");
```

Executing statement (for select):

```
// Select the columns from the Student table
ResultSet resultSet = statement.executeQuery
 ("select firstName, mi, lastName from Student where lastName "
   + " = 'Smith'");
```

Developing JDBC Programs

Loading drivers

Establishing connections

Creating and executing statements

Processing ResultSet

Executing statement (for select):

```
    // Select the columns from the Student table
    ResultSet resultSet = stmt.executeQuery
     ("select firstName, mi, lastName from Student where lastName "
       + " = 'Smith'");
```

Processing ResultSet (for select):

```
    // Iterate through the result and print the student names
    while (resultSet.next())
     System.out.println(resultSet.getString(1) + " " + resultSet.getString(2)
       + ". " + resultSet.getString(3));
```

methods

- The methods for executing SQL statements are execute, executeQuery, and executeUpdate, each of which accepts a string containing a SQL statement as an argument. This string is passed to the database for execution.
- The execute method should be used if the execution produces multiple result sets, multiple update counts, or a combination of result sets and update counts.

methods (Cont.)

- The executeQuery method should be used if the execution produces a single result set, such as the SQL select statement.
- The executeUpdate method should be used if the statement results in a single update count or no update count, such as a SQL INSERT, DELETE, or UPDATE, statement.

Prepared Statement

- The Prepared Statement interface is designed to execute dynamic SQL statements and SQL-stored procedures with IN parameters.
- These SQL statements and stored procedures are precompiled for efficient use when repeatedly executed.

```
PreparedStatement pstmt =
connection.prepareStatement
"INSERT INTO users (first_name,
last_name, email) + VALUES (?, ?, ?)");
```