

COS30041 Creating Secure and Scalable Software

Lecture 02 JDBC



SWIN
BUR
* NE *

SWINBURNE
UNIVERSITY OF
TECHNOLOGY

Commonwealth of Australia
Copyright Act 1968

Notice for paragraph 135ZXA (a) of the *Copyright Act 1968*

Warning

This material has been reproduced and communicated to you by or on behalf of Swinburne University of Technology under Part VB of the *Copyright Act 1968* (the Act).

The material in this communication may be subject to copyright under the Act. Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

Learning Objectives

- After studying the lecture material, you will be able to
 - Understand and describe the ideas behind Java Database Connectivity (JDBC)
 - Write Java applications that use JDBC for common programming situations

Pre-requisites

- Some concepts in Database Management System, DBMS
- The Structured Query Language, SQL

Outline

- Background: Database and SQL
- Java DataBase Connectivity, JDBC
- Programming using JDBC
- Issues related to connections
- Queries

Roadmap

- **Background: Database and SQL**
- Java DataBase Connectivity, JDBC
- Programming using JDBC
- Issues related to connections
- Queries

Some Background

■ LONG LONG TIME AGO

- ☐ Databases have their own languages
- ☐ Programs accessing database records using proprietary codes

■ PAST (not so distant):

- ☐ Databases have SQL (Structured Query Language) as a common language
- ☐ Programs accessing database records using standard SQLs – achieving better reusability

Introduction to Databases

- A **database** is an ordered collection of information from which a computer program can quickly access information
- Each **row** in a database **table** is called a **record (tuple)**
- A **record** in a database is a single complete set of related information
- Each column in a database table is called a **field (attribute)**
- **Fields** are the individual categories of information stored in a record

The diagram shows a table representing an employee directory database. Above the table, there are two boxes: 'Rows' on the left and 'Fields' on the right. Arrows point from the 'Rows' box to the first three rows of the table. Arrows point from the 'Fields' box to the first three columns of the table. The table has six columns: last_name, first_name, address, city, state, and zip. It contains five rows of employee data.

last_name	first_name	address	city	state	zip
Blair	Dennis	204 Spruce Lane	Brookfield	MA	01506
Hernandez	Louis	68 Boston Post Road	Spencer	MA	01562
Miller	Erica	271 Baker Hill Road	Brookfield	MA	01515
Morinaga	Scott	17 Ashley Road	Brookfield	MA	01515
Picard	Raymond	1113 Oakham Road	Barre	MA	01531

Employee directory database

Relational Databases

- A **relational database** consists of one or more related tables, where relationships between tables are implemented by primary/foreign keys
- A **primary key** is a field that contains a unique identifier for each record in a primary table
- A **primary key** is a type of index, which identifies records in a database to make retrievals and sorting faster
- A **foreign key** is a field in a related table that refers to the primary key in a primary table

Relationships of Tables

- One-to-One Relationship
- One-to-Many Relationship
- Many-to-Many Relationship

One-to-One Relationships

Primary key

Employees table

employee_id	last_name	first_name	address	city	state	zip
101	Blair	Dennis	204 Spruce Lane	Brookfield	MA	01506
102	Hernandez	Louis	68 Boston Post Road	Spencer	MA	01562
103	Miller	Erica	271 Baker Hill Road	Brookfield	MA	01515
104	Morinaga	Scott	17 Ashley Road	Brookfield	MA	01515
105	Picard	Raymond	1113 Oakham Road	Barre	MA	01531

Foreign key

Payroll table

employee_id	start_date	pay_rate	health_coverage	year_vested	401k
101	2002	\$21.25	none	na	no
102	1999	\$28.00	Family Plan	2001	yes
103	1997	\$24.50	Individual	na	yes
104	1994	\$36.00	Family Plan	1996	yes
105	1995	\$31.00	Individual	1997	yes

One-to-one relationship

Working with Database Management Systems

- A **relational database management system** (or RDBMS) is a system that stores and manages data in a relational format
- A **schema** is the structure of a database including its tables, fields, and relationships
- A **query** is a structured set of instructions and criteria for retrieving, adding, modifying, and deleting database information
- **Structured query language** (or SQL – pronounced as sequel) is a standard data manipulation language used among many database management systems
- **Open database connectivity** (or ODBC) : e.g. JDBC

Creating and Deleting Tables

- The `CREATE TABLE` statement specifies the table and column names and the data type for each column

```
CREATE TABLE table_name (column_name TYPE, ...);
```

- Execute the `USE` statement to select a database before executing the `CREATE TABLE` statement
- The `DROP TABLE` statement removes all data and the table definition

```
DROP TABLE table_name;
```

Example - Creating Table

```
mysql> CREATE TABLE inventory (  
    item_number int NOT NULL AUTO_INCREMENT,  
    make varchar(30) NOT NULL,  
    model varchar(30) NOT NULL,  
    price double NOT NULL,  
    quantity int NOT NULL,  
    PRIMARY KEY (item_number)  
);
```

AUTO_INCREMENT tells MySQL to go ahead and add the next available number to the `item_number` field.

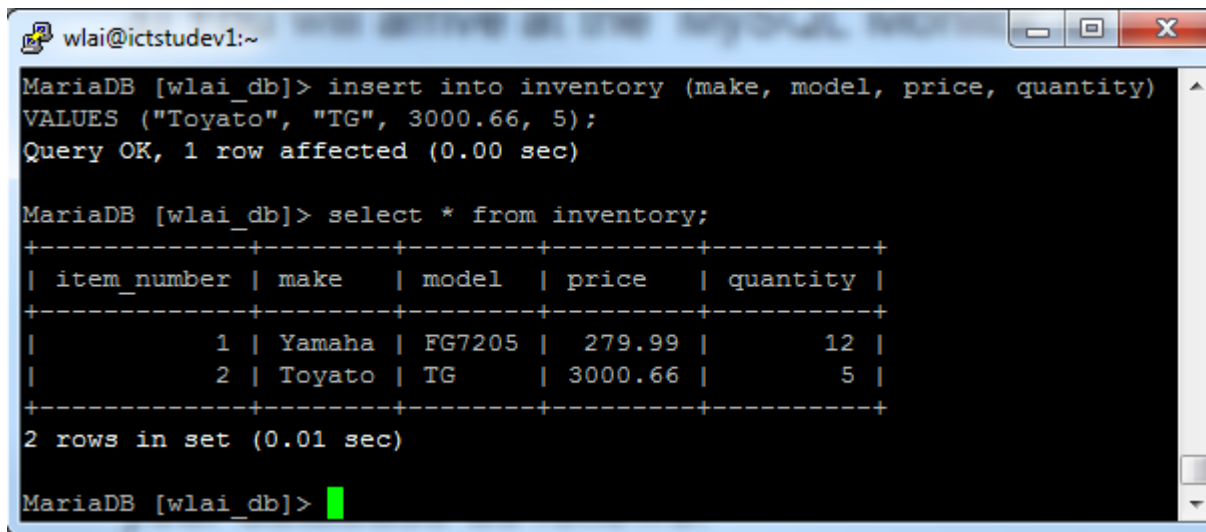
NOT NULL - we do not want this field to be NULL. So, if a user will try to create a record with a NULL value, then MySQL will raise an error.

Adding Records

- Use the `INSERT` statement to add individual records to a table

```
INSERT INTO table_name (column1, column2, ...)
VALUES (value1, value2, ...);
```

- The values entered in the `VALUES` list must be in the same order in which you defined the table fields
- Specify `NULL` in any fields for which you do not have a value



```
wlai@ictstudev1:~  
MariaDB [wlai_db]> insert into inventory (make, model, price, quantity)  
VALUES ("Toyato", "TG", 3000.66, 5);  
Query OK, 1 row affected (0.00 sec)  
  
MariaDB [wlai_db]> select * from inventory;  
+-----+-----+-----+-----+-----+  
| item_number | make   | model  | price  | quantity |  
+-----+-----+-----+-----+-----+  
|          1 | Yamaha | FG7205 | 279.99 |         12 |  
|          2 | Toyato | TG     | 3000.66 |          5 |  
+-----+-----+-----+-----+-----+  
2 rows in set (0.01 sec)  
  
MariaDB [wlai_db]>
```

Retrieving Records

- Use the `SELECT` statement to retrieve records from a table:

```
SELECT model, quantity FROM inventory;
```

- Use the asterisk (*) wildcard with the `SELECT` statement to retrieve all fields from a table
- You can also specify which records to return by using the `WHERE` keyword

```
SELECT * FROM inventory WHERE make='Martin';
```

- Use the keywords `AND` and `OR` to specify more detailed conditions about the records you want to return

```
SELECT * FROM inventory WHERE price<400;
```


Deleting Records

- Use the `DELETE` statement to delete records in a table
- The syntax for the `DELETE` statement is:

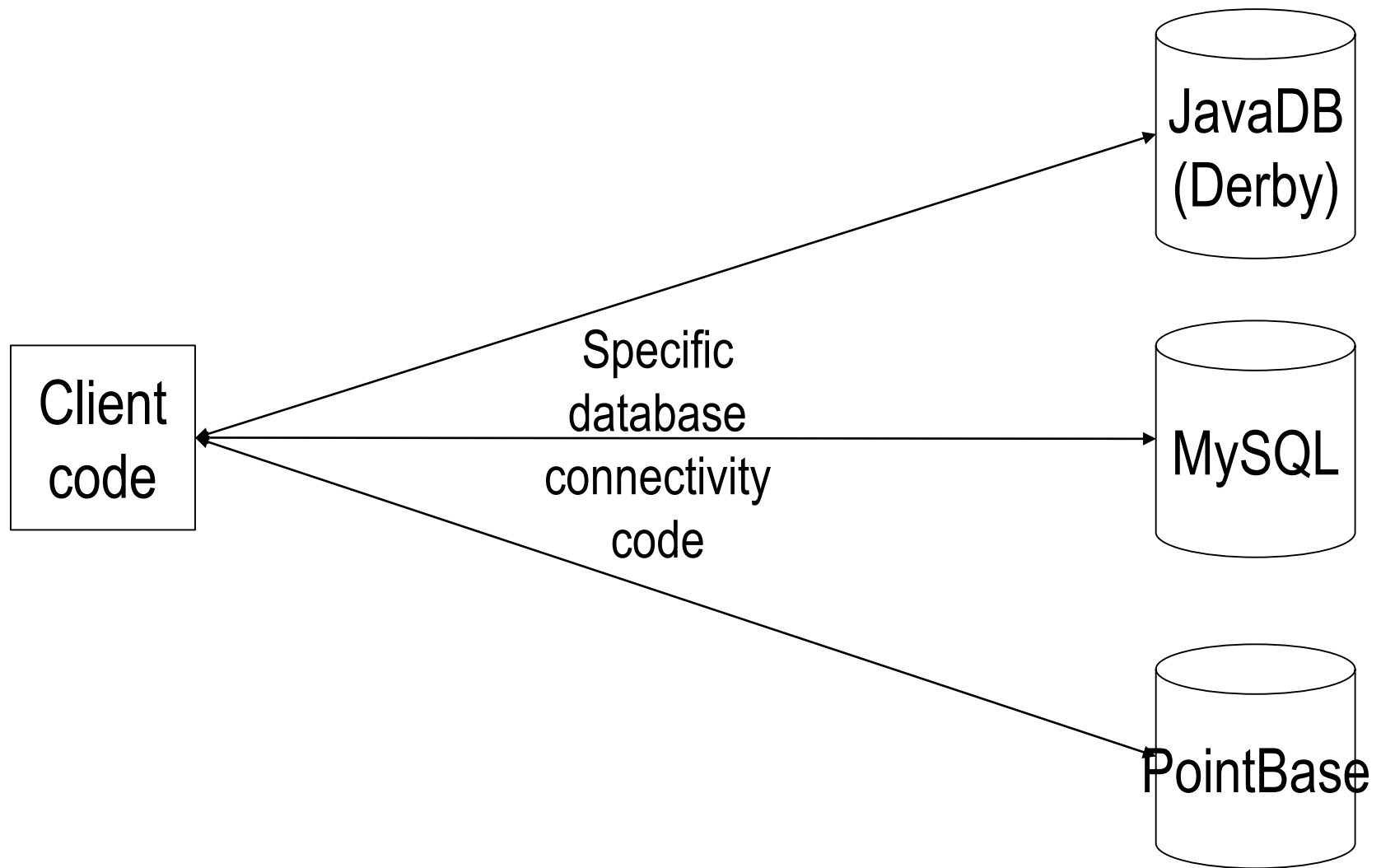
```
DELETE FROM table_name WHERE condition;
```

- The `DELETE` statement deletes all records that match the condition
- To delete all the records in a table, leave off the `WHERE` keyword

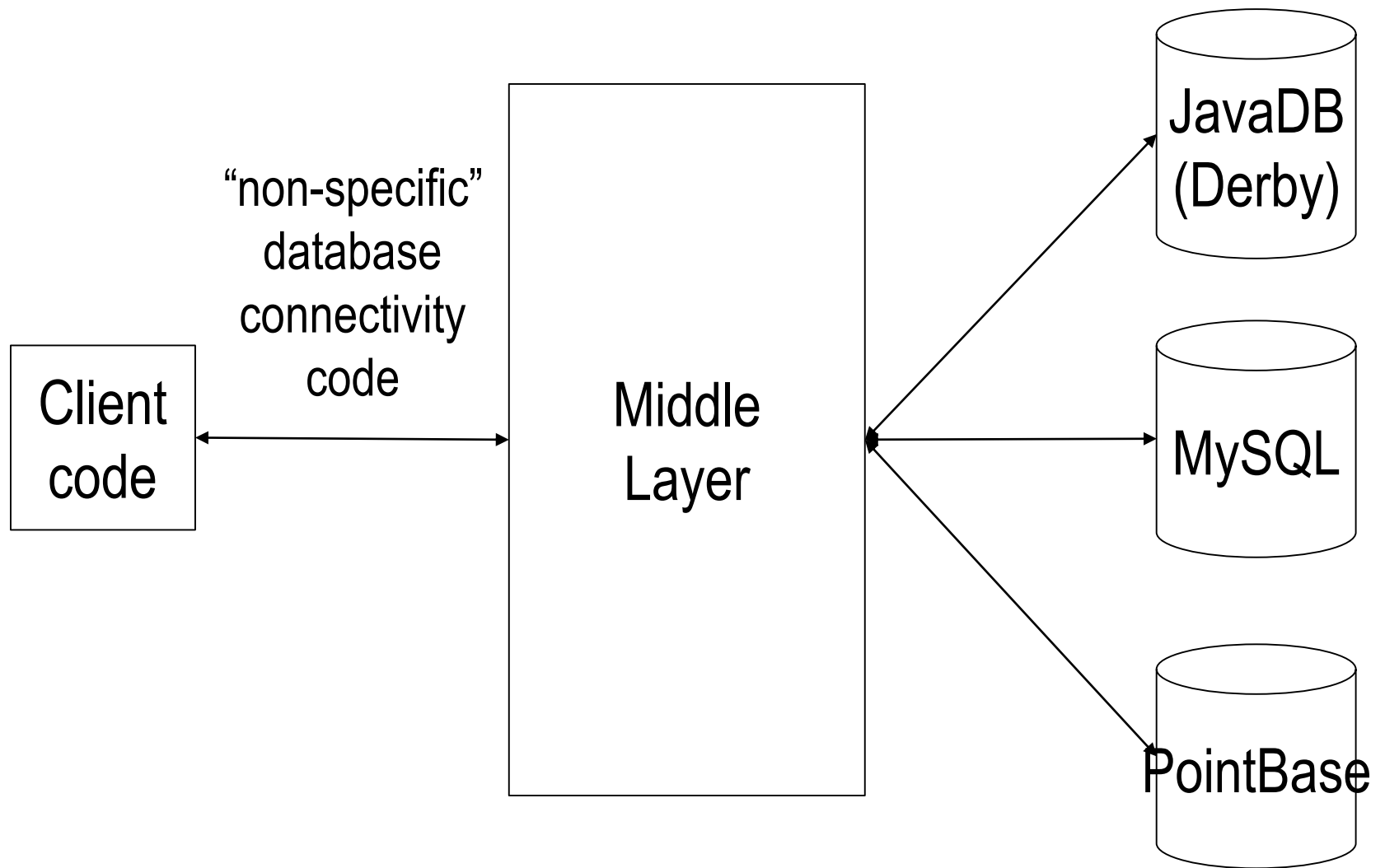
Roadmap

- Background: Database and SQL
- **Java DataBase Connectivity, JDBC**
- Programming using JDBC
- Issues related to connections
- Queries

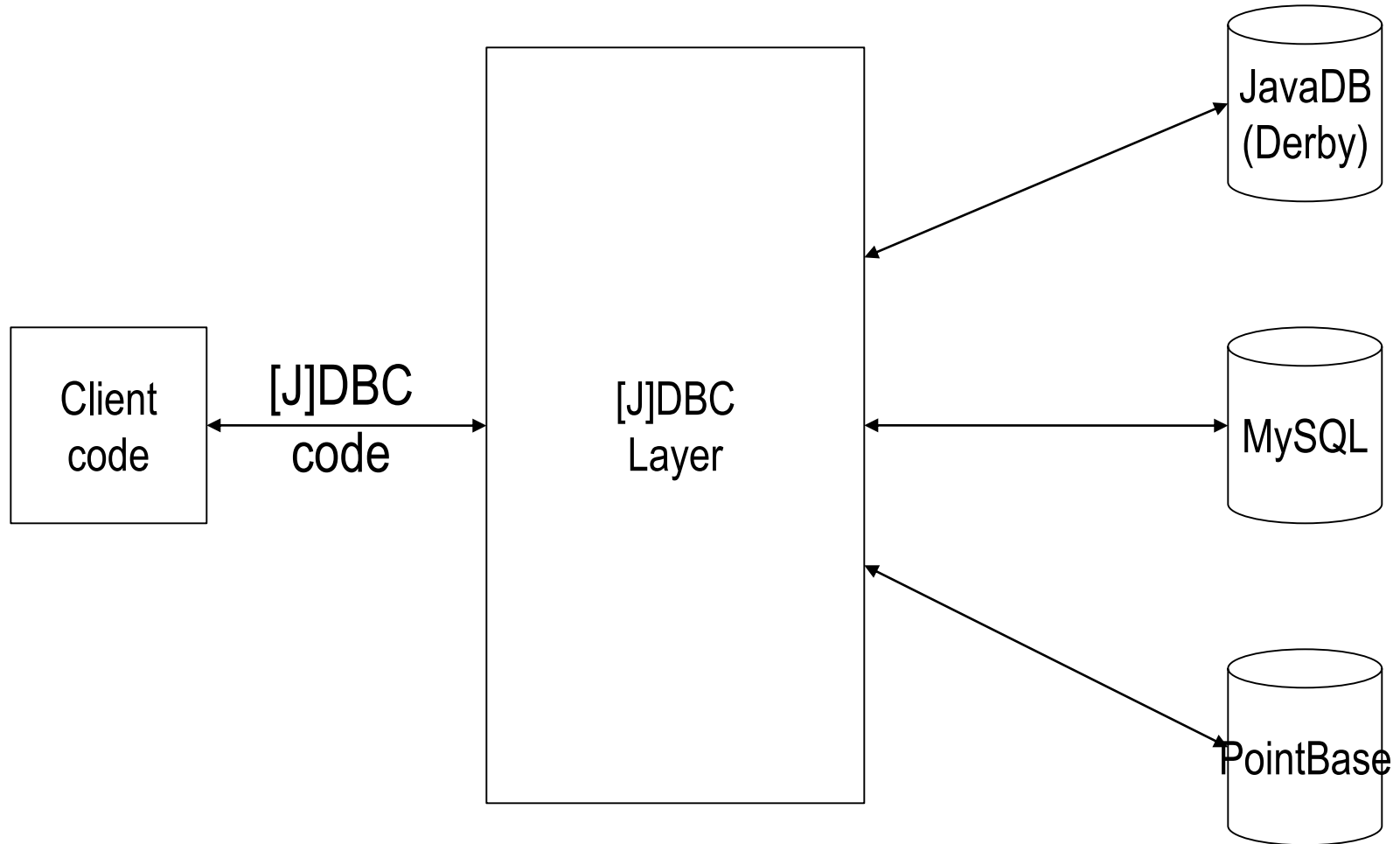
Database Connectivity – Option 1



Database Connectivity – Option 2



[Java] DataBase Connectivity, [J]DBC



JDBC – two parts

■ The JDBC API

- ☐ Standard Java API
- ☐ Use those APIs to program database requests
- ☐ Written by programmers to make database requests

■ The JDBC drivers

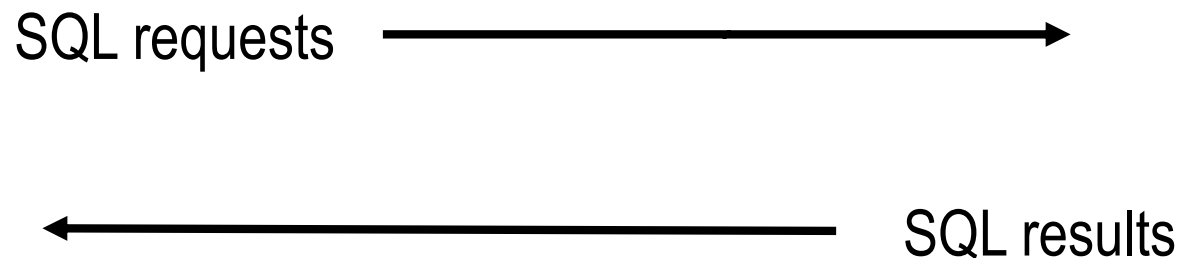
- ☐ Written by database providers to translate JDBC API to something that their database systems can understand
- ☐ Note: each provider has their own drivers
- ☐ Example: “Java DB Driver” provided by “Java DB” (aka Derby)

Typical client invocations

- Establish connection to DBMS server
- Send SQL requests to server
- Receive results from server
- Close connection

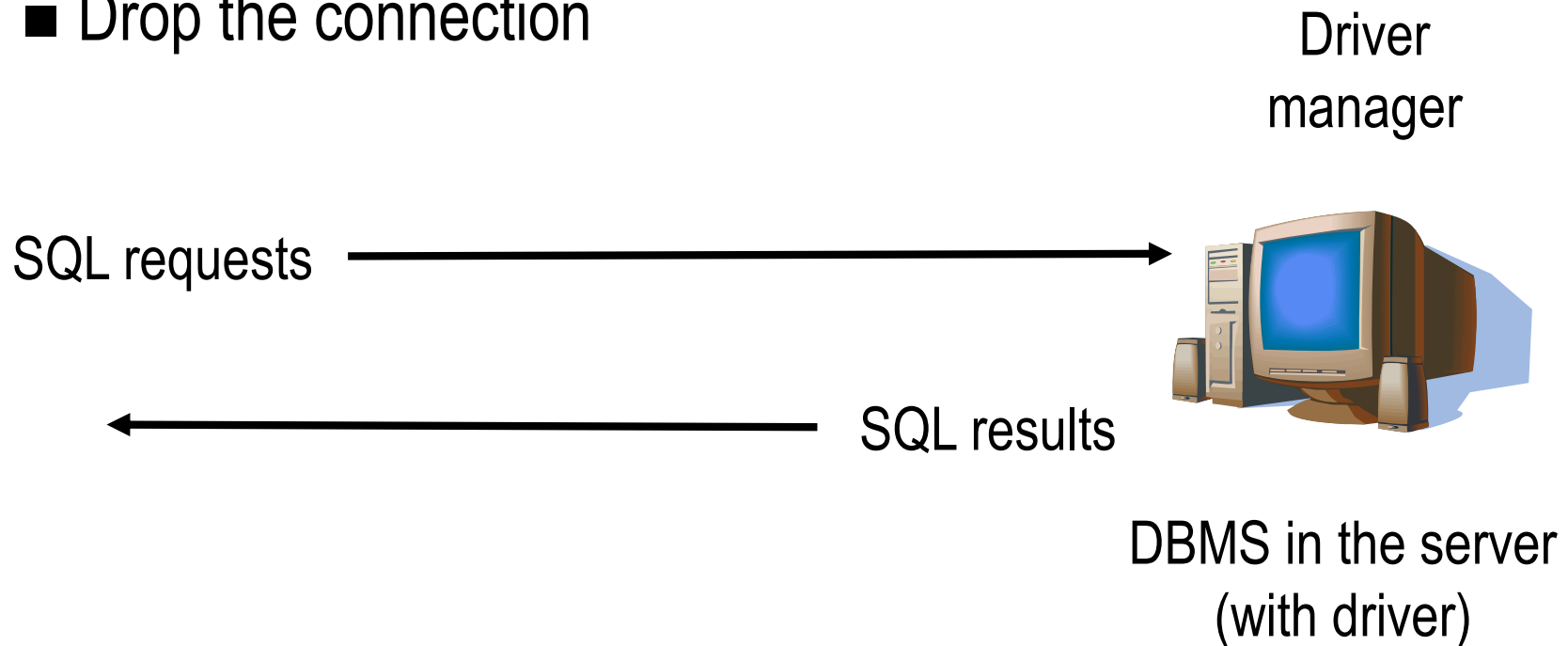


Remote / Local Client
(using API)



Typical server responses

- Authenticate connection request
- Process the SQL request from the client
- Send the results back to client
- Drop the connection



[Java] Database Connectivity – Advantages

- Highly portable and interoperable
- A developer can create the related applications without targeting a specific DBMS
- Vendor independent (?)
- Platform independent (?)

Why JDBC?

- Programs developed with Java programming language and JDBC are
 - ☐ platform independent
 - ☐ vendor independent

Databases and JDBC

- With JDBC, Java continues to support its highly portable and interoperable features
- A developer can create JDBC applications without targeting a specific DBMS
- Java classes in the form of JDBC drivers operate as a software layer between the Java program using JDBC and the actual database

Common Architecture of JDBC Applications

- Two tier: Client \leftrightarrow DBMS server
 - Disadvantage: The client depends on the actual database
- Three tier: Client \leftrightarrow **Business Server** \leftrightarrow DBMS server
 - A middle tier usually used for handling **business rules**
 - Client $\leftarrow X \rightarrow$ DBMS server
 - Advantage: The client is totally independent of the database

Roadmap

- Background: Database and SQL
- Java DataBase Connectivity, JDBC
- **Programming using JDBC**
- Issues related to connections
- Queries

Programming using JDBC

- 1. Register the database driver for each database that you want to send query to
- 2. Make the connection for each database
- 3. Execute the SQL statements and get the results
 - Two types of SQL statements

Prog. JDBC – 1 Register the database driver

- Use `System.setProperty()` method to set the “jdbc.driver” property

Settings for different databases

- JavaDB / Derby –

`org.apache.derby.jdbc.ClientDriver`

- Example:

```
System.setProperty("jdbc.drivers",  
    "org.apache.derby.jdbc.ClientDriver")
```

- PointBase –

`com.pointbase.jdbc.jdbcUniversalDriver`

- MySQL database – Swinburne mercury (Linux Server)

`com.mysql.jdbc.Driver`

Prog. JDBC – 2 Make the connection

■ Use

```
DriverManager.getConnection(  
    url, username, password  
)
```

to create a **Connection** object for each database

- url : the url of the database that you want to connect
- username, password: login credentials

Prog. JDBC – 2 Make the connection (cont'd)

■ The Database URL – vendor specific

- Syntax: `jdbc:subprotocol_name:other_stuff`
- “subprotocol_name” – the specific driver used by JDBC to connect to the database
 - Example: `odbc` if the `jdbc-odbc` bridge driver is being used
- “other_stuff” – Normally, the data source of the database
 - JavaDB / Derby –
`jdbc:derby://localhost:1527/sun-appserv-samples[;create=true]*`
 - Pointbase –
`jdbc:pointbase:server://localhost:1527/sun-appserv-samples`

*Note: “create=true” – create one if it does not exist

Make the connection - example

```
String url = "jdbc:derby://localhost/sun-appserv-samples;create=true";
```

```
String username = "APP";
```

```
String password = "APP";
```

```
Connection dbConn;
```

```
dbConn=DriverManager.getConnection(url, username, password);
```

```
// For MySQL database – Swinburne mercury (Linux Server):
```

```
// String url = "jdbc:mysql://feenix-mariadb.swin.edu.au/s1234567_db
```

```
// String username = "s1234567";
```

```
//String password = "030795";
```

Prog. JDBC – 3. Execute SQL Statement

After the database connection object is established,

■ Prepare the SQL statement (choose an appropriate one)

- ☐ Static – Use the connection object's `createStatement()` method to create a `Statement` object
- ☐ Precompiled – Use the connection object's `prepareStatement()` method to create a `PreparedStatement` object

■ Call the appropriate method

- ☐ `executeUpdate()` : to update the database
- ☐ `executeQuery()` : to perform query
- ☐ `execute()` : can do both

Prog. JDBC – Two Types of SQL Statement

[Assume `dbConn` is the database connection obtained in 2 above]

■ Static SQL statements

- `USE Statement object`

- `Statement s =
 dbConn.createStatement (...) ;`

■ Precompiled SQL statements with parameters

- `USE PreparedStatement object`

- `PreparedStatement s =
 dbConn.prepareStatement (...) ;`

■ See code example (EX-JDBC.zip)

- `CustomerDB.java`

Programming JDBC – Example

- WANT: A program to send SQL requests to a database
- NEED to perform the following
 - ☐ Establish the connection
 - ☐ Create the database table
 - ☐ Add records
 - ☐ Modify particular records
 - ☐ Delete particular records
 - ☐ Query records
 - ☐ Destroy the database table
- See the NetBeans project ED-JDBC (Lab 2)

Create a table - example

```
Statement stmt = null;
```

```
stmt = dbConn.createStatement();
```

```
stmt.execute("CREATE TABLE MYUSER ( "  
    + " UserId CHAR(6) PRIMARY KEY, "  
    + " Name CHAR(30), Password CHAR(6), Email CHAR(30), "  
    + " Phone CHAR(10), Address CHAR(60), "  
    + " SecQn CHAR(60), SecAns CHAR(60))");
```

Destrory a table - example

```
Statement stmt = null;  
stmt = dbConn.createStatement();  
stmt.execute("DROP TABLE MYUSER");
```

See sample code in
ED-JDBC

Or

```
String destroyTableSQL = "DROP TABLE MYUSER ";  
stmt.execute(destroyTableSQL);
```

See sample code in
EX-JDBC

Sample code:
EX-JDBC.zip – download from Canvas
ED-JDBC – Lab 2

Add a record - example

SQL:

```
INSERT INTO table_name (column1, column2, ...)
VALUES (value1, value2, ...);
```

```
int ID=100001;
```

```
String name = "Peter";
```

```
PreparedStatement stmt=dbConn.prepareStatement("insert into EMP values(?,?)");
```

```
stmt.setInt(1,ID);//1 specifies the first parameter in the query
```

```
stmt.setString(2,name);
```

```
int i=stmt.executeUpdate();
```

```
System.out.println(i+" records inserted");
```


execute, executeUpdate, executeQuery

- **execute** method can be used with any type of SQL statements and it returns a boolean.
- **executeUpdate** method execute sql statements that insert/update/delete data at the database. This method return int value representing number of records affected; Returns 0 if the query returns nothing. The method accepts only non-select statements.
- **executeQuery** method execute statements that returns a result set by fetching some data from the database. It executes only select statements.

Roadmap

- Background: Database and SQL
- Java DataBase Connectivity, JDBC
- Programming using JDBC
- **Issues related to connections**
- Queries

Issues related to Connections

- When the *DriverManager's getConnection()* method is executed, it will try to find a loaded driver that can use the protocol specified in the database URL by iterating through the available drivers currently registered with the device manager
- The *Connection* object creates a specific connection with a specific database and enables you to use the JDBC facilities to manage SQL queries

Issues related to Connections

- You can execute queries and action statements and commit or roll back transactions
- It is possible to have different connections to different databases at the same time and transfer data between them
- On completion of the use of the Connection, the connection **MUST** be closed with the `close()` method
- Otherwise, the connection will not be released and hence the database's performance will be degraded

Roadmap

- Background: Database and SQL
- Java DataBase Connectivity, JDBC
- Programming using JDBC
- Issues related to connections
- **Queries**

Queries

- Use the `executeQuery(String)` method of the `Statement` or `PreparedStatement` class
- The `String` contains the SQL command for the query
- The same `Statement` object can be used for many different, unrelated queries
- The results of the query are returned as a `ResultSet` object which is created by the execution of the query
- Example

```
ResultSet rs = s.executeQuery("SELECT *  
FROM Customer");
```

Queries (cont'd)

- To examine a `ResultSet` object, an iterative loop can be set up as follows:

```
ResultSet rs = s.executeQuery("SELECT * FROM Customer");  
while (rs.next()) {  
    // examine a row from the results  
    ...  
}
```

- A `ResultSet` object is initially positioned *before* its first row
- Execution of `next()` advances the cursor to point to the next row
- A `ResultSet` object is automatically closed when the `Statement` / `PreparedStatement` object that generates it is closed or re-executed

Queries (cont'd)

- Use various accessor methods to inspect individual rows
- Such methods return the value of the column with a given column number or name

`Xxxxx getXxxxx(int columnNumber)`

or

`Xxxxx getXxxxx(String columnName)`

where `Xxxxx` is a type such as `int`, `double`, `String`, `Date`, etc

- See the jdk documentation for details

Queries (cont'd)

■ Example (accessor methods)

```
String custName = rs.getString(1); //Column number 1  
int custAge = rs.getInt("Age"); //Column name "Age"
```

```
e.g. int empID=rs.getInt(1);  
      String empName=rs.getSring(2);
```

- Java will make reasonable type conversions when the type of the `getXxxxx()` method doesn't match the type of the column, but you need to check the documentation for specifics here
- Unfortunately there are significant variations between the SQL types supported by different database products

Queries (cont'd)

- Even when different databases support SQL types with the same semantics, they may give those types different names
- For example, most of the major databases support an SQL data type for *large binary values*, but
 - Oracle calls this type LONG RAW,
 - Sybase calls it IMAGE,
 - Informix calls it BYTE, and
 - DB2 calls it LONG VARCHAR FOR BIT DATA.

Queries (cont'd)

Some Basic SQL Type and their corresponding Java types (extract from Table 4.6 in CJV2)

SQL data type	Java data type
INTEGER or INT	int
FLOAT(n)	double
REAL	float
DOUBLE	double
CHARACTER(n) or CHAR(n)	String
BOOLEAN	boolean
ARRAY	java.sql.Array
DATE	java.sql.Date
TIME	java.sql.Time

References

- [CJV2] C.S. Horstmann and G. Cornell (2005) *Core Java 2 Vol. II – Advanced Features*, Prentice-Hall
 - Chapter 4