JDBC

Java Database Connectivity

Prepared by Lei Ye

UNIVERSITY OF WOLLONGONG

Java Database Connectivity (JDBC)

- A ***database*** is an organised collection of data
- A ***relational database*** stores data in related tables
- A ***database management system*** (DBMS) provides mechanisms for storing, organising, retrieving and modifying data
  - Oracle, MS SQL Server, MySQL, PostgreSQL
  - Java DB (Apache Derby)
- ***SQL*** (Structured Query Language) is used to query and manipulate data in a database
- Java applications communicate with databases and manipulate their data using Java Database Connectivity (***JDBC***) API
  - A ***JDBC driver*** is required to enable Java applications to connect to the database in a DBMS
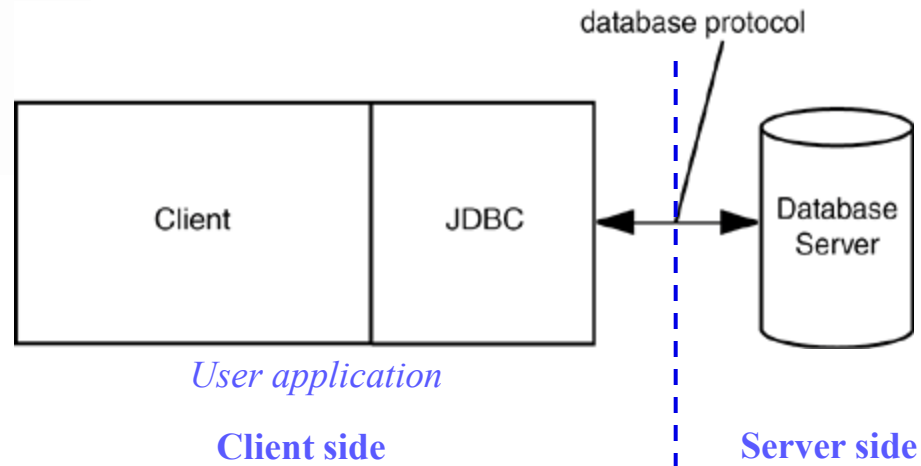
JDBC to Database Communication Path

- Programmers can write applications in Java to access any database, using standard SQL statements

- Database vendors and database tool vendors can supply the low-level drivers.

Open Database Connectivity

UNIVERSITY OF WOLLONGONG

Typical Uses of JDBC

- Traditional client/server application

User application

Client side   **Server side**

Typical Uses of JDBC

- A three-tier Application

HTTP, RMI, etc...   database protocol

```
Client                    Middle tier              JDBC        Database
(visual                   (business logic)                     Server
presentation)
```

User application   *Middleware application*
Front-end   *Middle-tier*   *Back-end*

Client side   **Server side**

UNIVERSITY OF WOLLONGONG
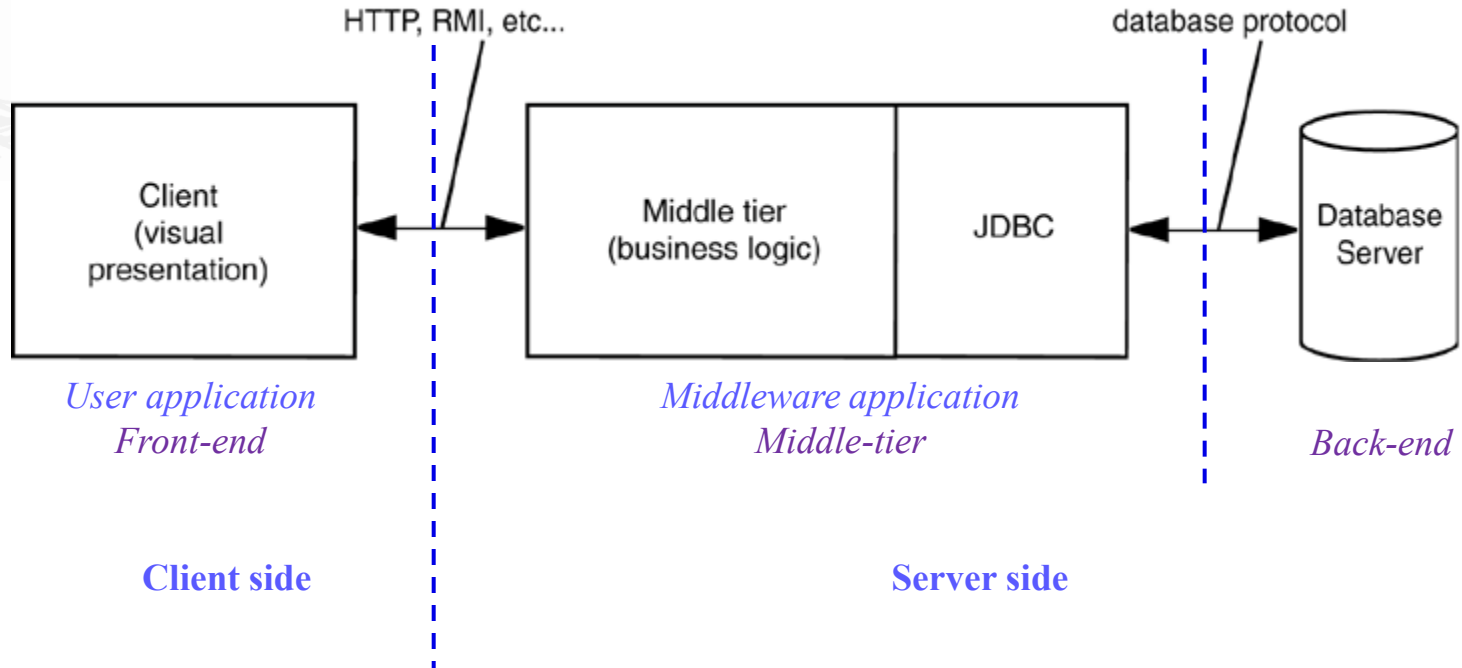
Typical Uses of JDBC

- An Embedded Application
  - The JDBC application and embedded database run in the same JVM. The JDBC application starts up the database

```
method calls          JDBC    Embedded
   to JDBC                     Database
```

User application

UNIVERSITY OF
WOLLONGONG
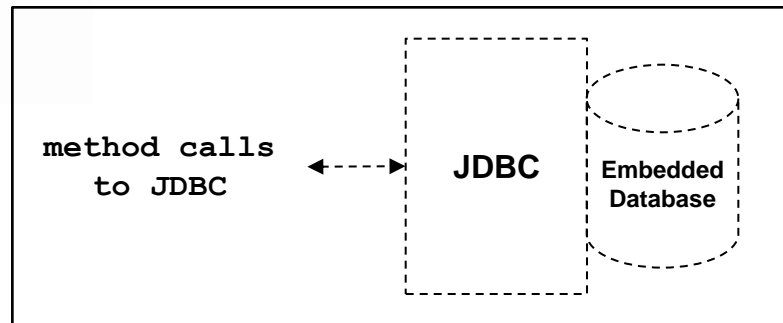
Database and SQL Basics

- Database and table
- Information Retrieval
  - SQL query
- SQL commands
  - Data manipulation language
  - Data definition language

UNIVERSITY OF
WOLLONGONG

A Table in a Database

Employees Table

Employee_Number	First_name	Last_Name	Date_of_Birth	Car_Number
10001	Axel	Washington	28-Aug-43	5
10083	Arvid	Sharma	24-Nov-54	null
10120	Jonas	Ginsberg	01-Jan-69	null
10005	Florence	Wojokowski	04-Jul-71	12
10099	Sean	Washington	21-Sep-66	null
10035	Elizabeth	Yamaguchi	24-Dec-59	null

Row
Record

Primary key

Column
Field

UNIVERSITY OF
WOLLONGONG

SQL Query

- **SELECT** statements
  - Retrieve information from a table

```sql
SELECT First_Name, Last_Name
FROM Employees
WHERE Car_Number IS NOT NULL
```

Result Set

FIRST_NAME	LAST_NAME
Axel	Washington
Florence	Wojokowski

```sql
SELECT *
FROM Employees
```

Producing a result set that includes the whole **Employee** table

UNIVERSITY OF WOLLONGONG

SQL WHERE Clauses

- String Comparison with LIKE

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Last_Name LIKE 'W%'
```

% is a wildcard

Result Set

FIRST_NAME	LAST_NAME
Axel	Washington
Florence	Wojokowski
Sean	Washington

- Numerical Comparisons

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Car_Number = 12
```

Result Set

FIRST_NAME	LAST_NAME
Florence	Wojokowski

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Employee_Number < 10100
  and Car_Number IS NULL
```

Result Set

FIRST_NAME	LAST_NAME
Arvid	Sharma
Sean	Washington
Elizabeth	Yamaguchi

Joined Tables

Cars Table

Car_Number	Make	Model	Year
5	Honda	Civic DX	1996
12	Toyota	Corolla	1999

```
SELECT  Employees.First_Name, Employees.Last_Name, Cars.Make, Cars.Model, Cars.Year
FROM    Employees, Cars
WHERE   Employees.Car_Number = Cars.Car_Number
```

Result Set

FIRST_NAME	LAST_NAME	MAKE	MODEL	YEAR
Axel	Washington	Honda	Civic DX	1996
Florence	Wojokowski	Toyota	Corolla	1999

UNIVERSITY OF
WOLLONGONG

Common SQL Commands

- Data Manipulation Language (MDL)
  - SELECT — used to query and display data from a database. The SELECT statement specifies which columns to include in the result set. The vast majority of the SQL commands used in applications are SELECT statements.

    ```
    SELECT field1, field2, …
    FROM table
    [WHERE criteria]
    [ORDER BY fields desc/asc, …]
    ```

  - INSERT — adds new rows to a table. INSERT is used to populate a newly created table or to add a new row (or rows) to an already-existing table.

    ```
    INSERT INTO table (field1, field2, …)
    VALUES (value1, value2, …)
    ```

  - DELETE — removes a specified row or set of rows from a table

    ```
    DELETE FROM table
    WHERE criteria
    ```

  - UPDATE — changes an existing value in a column or group of columns in a table

    ```
    UPDATE table
    SET (field1=value1, field2=value2, …)
    WHERE criteria
    ```

UNIVERSITY OF WOLLONGONG

Common SQL Commands

- ## Data Definition Language (DDL)

  – CREATE TABLE/DATABASE — To create a table with the column names the user provides. The user also needs to specify a type for the data in each column; to create a database

  ```
  CREATE TABLE table          CREATE DATABASE database
  (field1 type,
   field2 type,
   … )
  ```

  Data types and ranges vary from one DBMS to another

  – DROP TABLE/DATABASE — deletes all rows and removes the table definition from the database; delete a database

  ```
  DROP TABLE table            DROP DATABASE database
  ```

  – ALTER TABLE — adds or removes a column from a table. It also adds or drops table constraints and alters column attributes

  ```
  ALTER TABLE table           ALTER TABLE table
  ADD field type              DROP field
  ```

UNIVERSITY OF
WOLLONGONG

JDBC Programming Concepts

- Drivers
  - Loading/registering drivers

- Connection
  - Database URL
  - Establishing connections

- Statements
  - Creating statement object
  - Executing statements
    - Call `executeUpdate()` or `executeQuery()`

- `ResultSet` interface
  - Processing `ResultSet`

UNIVERSITY OF WOLLONGONG

Drivers

- Automatic registration (JDBC 4.0 – Java SE 6)
  - A Jar file can automatically register the driver class if it contains a file
    META-INF/servicers/java.sql.Driver
- Loading the driver class in a Java program

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

- Setting `jdbc.drivers` property with a command-line argument

  ```
  java –Djdbc.drivers="JDBCDriverClass" MyApp
  ```

- Setting system property in a Java program

  ```
  System.setProperty("jdbc.drivers", "DBCDriverClass");
  ```

Database	Driver Class	Source
Oracle	`oracle.jdbc.driver.OracleDriver`	Vender provided
MySQL	`com.mysql.jdbc.Driver`	Vender provided
Access	`sun.jdbc.odbc.JdbcOdbcDriver`	JDK
Derby (client)	`org.apache.derby.jdbc.ClientDriver`	derbyclient.jar
Derby (embedded)	`org.apache.derby.jdbc.EmbeddedDriver`	derby.jar

UNIVERSITY OF WOLLONGONG

Connection

- Using the DriverManager Class

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

getConnection method summary

```
static Connection    getConnection(String url)
static Connection    getConnection(String url, Properties info)
static Connection    getConnection(String url, String user, String password)
```

Database	URL Pattern
Oracle	`jdbc:oracle:thin:@hostname:port#:oracleDBSID`
MySQL	`jdbc:mysql://hostname/dbname`
Access	`jdbc:odbc:dataSource`
Derby (client)	`jdbc:derby:dbname[;attribute=value]`
Derby (embedded)	`jdbc:derby:dbname[;attribute=value]`

UNIVERSITY OF WOLLONGONG

Statements

- Creating statements

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

- Executing statements

  – SQL DDL or update statements

  ```
  statement.executeUpdate("CREATE TABLE myTable(…)");
  ```

  – SQL query statements

  ```
  ResultSet resultSet = statement.executeQuery
      ("SELECT fields FROM table WHERE criteria");
  ```

UNIVERSITY OF WOLLONGONG

ResultSet Interface

- The **ResultSet** object contains a table of data representing a database result set, usually generated by executing a query statement
- The **next()** method moves to the next row
  - The initial row position is **null**
- The various **getXxx()** method retrieves values from current row

```
try {
    statement = connection.createStatement();
    ResultSet resultSet = statement.executeQuery("…");
    while (resultSet.next()) {
      …
      resultSet.getSting(1);
      resultSet.getInt("fieldLabel");
      …
}catch (SQLException e){ … }
finally {
  if (statement != null) statement.close();
}
```

<<interface>> ResultSet
+ next(): boolean
+ getXxx(fieldNumber: int): Xxx
+ getXxx(fieldLabel: String): Xxx
+ findColumn(fieldLabel: String): int
+ close(): void
+ isClose(): boolean

Xxx is a type such as **int, double, String, Date, blob**, etc.

UNIVERSITY OF WOLLONGONG

Example (since Java SE 5)

```java
private String driver ="org.apache.derby.jdbc.EmbeddedDriver";
private String url= "jdbc:derby:MyDatabase"; //"MyDatabase" is the database name
 . . .
```

(1)
```java
Class.forName(driver).newInstance();    //not necessary since JDBC 4.0 - Java SE 6
```
(2)
```java
Connection conn = DriverManager.getConnection(url + ";create=true");
                                        //create it if the database does not exist
try{
```
(3)
```java
    Statement stat = conn.createStatement();
    //necessary if you need to create a new table
    stat.executeUpdate("CREATE TABLE Greetings (Message CHAR(22))");
    stat.executeUpdate("INSERT INTO Greetings VALUES ('Hello, World!')");
    stat.executeUpdate("INSERT INTO Greetings VALUES ('Hello, World again!')");

```
(4)
```java
    ResultSet result = stat.executeQuery("SELECT * FROM Greetings");
    if (result.next())
        System.out.println(result.getString(1));
    result.close();
    stat.executeUpdate("DROP TABLE Greetings");
```
(5)
```java
} catch (SQLException sqle) {
    // handling SQLException chain
}
finally{
    try { if (stat != null) stat.close();} catch (SQLException sqle) {…}
    try { if (conn != null) conn.close();} catch (SQLException sqle) {…}
}
```

Don't forget

```
Hello, Database!
Hello, Database again!
```

UNIVERSITY OF WOLLONGONG

Example (since Java SE 7)

```java
private String url= "jdbc:derby:MyDatabase";    //"MyDatabase" is the database name

1
2  try ( Connection conn = DriverManager.getConnection(url + ";create=true");
3          Statement stat = conn.createStatement();){ //create it if the database does not exist

       //necessary if you need to create a new table
       stat.executeUpdate("CREATE TABLE Greetings (Message CHAR(22))");

       stat.executeUpdate("INSERT INTO Greetings VALUES ('Hello, Database!')");
       stat.executeUpdate("INSERT INTO Greetings VALUES ('Hello, Database again!')");

4      try (ResultSet result = stat.executeQuery("SELECT * FROM Greetings")){
           while (result.next())
               System.out.println(result.getString(1));
       }
       stat.executeUpdate("DROP TABLE Greetings");
                                       //Otherwise next time, CREATE TABLE will fail
5  } catch (SQLException sqle) {
       // handling SQLException chain
   }
```

Hello, Database!
Hello, Database again!

UNIVERSITY OF WOLLONGONG

SQL Exceptions

- Each **SQLException** has a chain of **SQLException** objects that is retrieved with the **getNextException** method
- This exception chain is in addition to the "cause" chain of **Throwable** objects that every exception has
- Use a loop to enumerate all exceptions
  - **SQLException** implements **Iterable<Throwable>** interface

```
for (Throwable t : sqlException) {
        //do something with t, eg. t.printStackTrace();
}
```

- In addition, the database driver can report nonfatal conditions as warnings
  - You can retrieve warnings from connections, statements, and result sets. The SQLWarning class is a subclass of SQLException (even though a SQLWarning is not thrown as an exception). You call getSQLState and getErrorCode to get further information about the warnings. Similar to SQL exceptions, warnings are chained

```
SQLWarning w = stat.getWarning();
while (w != null) {
     //do something with w
     w = w.nextWarning();
}
```

UNIVERSITY OF WOLLONGONG

Overview of SQLException

- A description of the error
    - **SQLException.getMessage()**

- A SQLState code
    - Codes (five alphanumeric characters) standardised by ISO/ANSI and Open Group (X/Open)
    - **SQLException.getSQLState()**

- An error code
    - Codes (an integer) implementation-specific
    - **SQLException.getErrorCode ()**

- A cause
    - A **SQLException** instance might have a causal relationship
    - Recursively call the method **SQLException.getCause()**

- A reference to any chained exceptions
    - **SQLException.getNextException()**

UNIVERSITY OF
WOLLONGONG

Retrieve SQLExceptions

```java
for (Throwable e : sqle) {
        if (e instanceof SQLException) {

                e.printStackTrace(System.err);
                System.err.println("SQLState: " +
                    ((SQLException)e).getSQLState());

                System.err.println("Error Code: " +
                    ((SQLException)e).getErrorCode());

                System.err.println("Message: " + e.getMessage());

                Throwable t = sqle.getCause();
                while(t != null) {
                    System.out.println("Cause: " + t);
                    t = t.getCause();
                }
        }
}
```
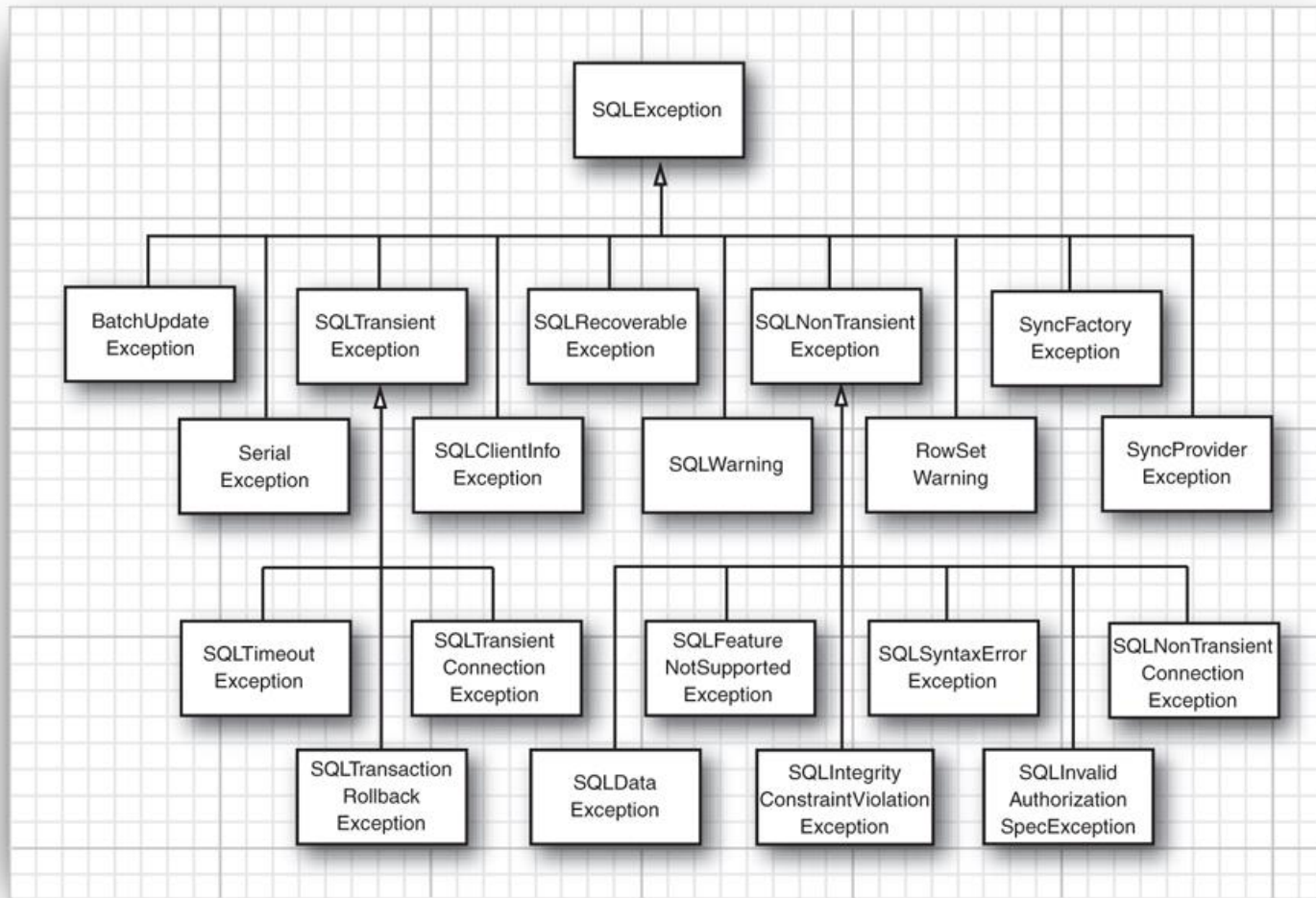
Try to query a table "MyGreeting" that does not exist:
        stat.executeQuery("SELECT * FROM MyGreetings")

```
SQLState: 42X05
Error Code: 30000
Message: Table/View 'MYGREETINGS' does not exist.
Cause: java.sql.SQLException: Table/View 'MYGREETINGS' does not exist.
Cause: ERROR 42X05: Table/View 'MYGREETINGS' does not exist.
```

SQL Exception Tree

Configure Connection Using `Properties`

- Using **`Properties`** object and properties file

```java
Properties props = new Properties();

FileInputStream in = new FileInputStream("database.properties");
props.load(in);
in.close();

String drivers = props.getProperty("jdbc.drivers");
if (drivers != null) System.setProperty("jdbc.drivers", drivers);
                    // setting system-wide properties

String url = props.getProperty("jdbc.url");
String username = props.getProperty("jdbc.username");
String password = props.getProperty("jdbc.password");

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

database.properties

```
jdbc.drivers=org.apache.derby.jdbc.ClientDriver
jdbc.url=jdbc:derby:MyDatabase;create=true
jdbc.username=derbyuser
jdbc.password=secret
```

UNIVERSITY OF
WOLLONGONG

PreparedStatement

- **Statement** interface executes static SQL statements without parameters

- **PreparedStatement** interface (extending **Statement**) executes a precompiled SQL statement with/without parameters
    – Efficient for repeated executions

```
Statement preparedStatement = conn.prepareStatement
        ("SELECT fields FROM table WHERE field1=? and field2=?");


prepareStatement.setX(int parameterIndex, X value);        // X is a type
ResultSet resultSet = prepareStatement.executeQuery();
```

UNIVERSITY OF
WOLLONGONG

Scrollable and Updatable Result Sets

- Scrollable: Move forward and backward in the result set;
- Updatable: edit the **ResultSet** and have the changes automatically reflected in the database.
- Create different statement object:

```
Statement stat = conn.createStatement(type, concurrency)
PreparedStatement stat = conn.prepareStatement(command, type, concurrency)
```

- ResultSet Types

Value	Explanation
TYPE_FORWARD_ONLY	The result set cannot be scrolled; its cursor moves forward only. (Default)
TYPE_SCROLL_INSENSITIVE	The result can be scrolled and is insensitive to database changes.
TYPE_SCROLL_SENSITIVE	The result can be scrolled and is sensitive to database changes.

- ResultSet Concurrency

Value	Explanation
CONCUR_READ_ONLY	The ResultSet object cannot be used to update the database. (Default)
CONCUR_UPDATABLE	The ResultSet object can be used to update the database; not all queries return updatable result sets; use the getConcurrency() to find out.

UNIVERSITY OF WOLLONGONG

Examples: Scrollable `ResultSet`

```java
Statement stat = conn.createStatement(
                ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
ResultSet rs = stat.executeQuery(query);



rs.previous();    // scroll backward

rs.relative(n);   // move forward (n>0) or backward (n<0)

rs.absolute();    // move to a particular row number

int currentRow = rs.getRow(); // get the current row number
```

UNIVERSITY OF
WOLLONGONG

Examples: Updatable `ResultSet`

```java
Statement stat = conn.createStatement(
                ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stat.executeQuery(query);


/* update row values */
double price = rs.getDouble("Price");
rs.updateDouble("Price", price + increase);   // new price in ResultSet
rs.updateRow();  // send all updates in the row to database; must be done before
                    moving to another row otherwise this row's updates are
                    discarded
. . .
/* add a new row */
rs.moveToInsertRow();  // move to the special insert row
rs.updateString("Title", title);
. . .                   // build up a new row using updateXxx()
rs.updateDouble("Price", price + increase);
rs.insertRow();  // send the new row to the database
rs.moveToCurrentRow();// move back to the position
                    // before the call to moveToInsertRow()

. . .
/* delete the row under the cursor */
rs.deleteRow();  // immediately remove the row from ResultSet and the database
```

UNIVERSITY OF WOLLONGONG

Row Sets

- Major drawback of `ResultSet`: need to keep the database connection

- `RowSet`: not have to tied to a database connection
  - extending `ResultSet`

- `RowSet` object contains tabular data in a way more flexible and easier to use than a `ResultSet`

Value	Explanation
`JdbcRowSet`	Connected scrollable and updatable `RowSet` and turning the `ResultSet` into a JavaBean component
`CachedRowSet`	Disconnected operations of extension of `JdbcRowSet`; populate itself with `ResultSet`; reconnect to write changes back to the database
`WebRowSet`	Extension of `CachedRowSet`; can be saved to an XML file
`FilteredRowSet/JoinRowSet`	Extension of `WebRowSet`; equivalent to SQL SELECT/JOIN on row set without having to make a database connection

UNIVERSITY OF
WOLLONGONG

Reading and Writing LOBs

- Many database can store *large objects* (**LOB**s) such as images or other data
- In SQL, binary LOBs are called **BLOB**s and character LOBs are called **CLOB**s
- Use stream I/O to read and write LOBs

  – Retrieve BLOB

```
Blob blob = resultSet.getBlob(fieldIndex);
InputStream in = blob.getInputStream();
```

  – Store BLOB

```
Blob clob = Connection.createBlob();
OutputStream out = blob.setBinaryStream(offset);  //offset=0
. . . // write to out

PrepareStatement stat = connection.prepareStatement("INSERT INTO field VALUES (?, ?)");
stat.set(1, id);
stat.set(2, blob);
stat.executeUpdate();
```

UNIVERSITY OF
WOLLONGONG

Transactions

- A transaction is a set of one or more SQL statements that make up a logical unit of work

- The transaction can be *commit*ted when all has gone well. Or, if an error has occurred in one of them, it can be *rolled back* as if none of the commands had been issued

- Major reason: *database integrity*

  - Scenario

    One statement to debit one account and another statement to credit another account.

    Do you like the system to fail after debiting your account but before crediting the other account?

- By default, a database connection is in *autocommit* mode, and each SQL command is committed to the database as soon as it is executed. Once a command is committed, you cannot roll it back.

UNIVERSITY OF WOLLONGONG

Example

```java
//Turn off autocommit mode
conn.setAutoCommit(false);

//Create a statement object in the normal way
Statement stat = conn.createStatement();

//Call executeUpdate any number of times
stat.executeUpdate(command1);
stat.executeUpdate(command2);
stat.executeUpdate(command3);
. . .

//Call the commit method when all commands have been executed
conn.commit();

//Call the rollback method if an error occurred
conn.rollback();    // Typically a rollback is issued
                    // when a transaction was interrupted
                    // by a SQLException.
```

UNIVERSITY OF
WOLLONGONG

SQL vs Java Data Types

SQL Data Type	Java Data Type
`INTEGER or INT`	`int`
`SMALLINT`	`short`
`NUMERIC(m,n), DECIMAL(m,n) or DEC(m,n)`	`java.math.BigDecimal`
`FLOAT(n)`	`double`
`REAL`	`float`
`DOUBLE`	`double`
`CHARACTER(n) or CHAR(n)`	`String`
`VARCHAR(n)`	`String`
`BOOLEAN`	`boolean`
`DATE`	`java.sql.Date`
`TIME`	`java.sql.Time`
`TIMESTAMP`	`java.sql.Timestamp`
`BLOB`	`java.sql.Blob`
`CLOB`	`java.sql.Clob`
`ARRAY`	`java.sql.Array`

UNIVERSITY OF WOLLONGONG

DataSources in Enterprise Infrastructure

- When a JDBC application is deployed in an enterprise environment, the management of database connections is integrated with the Java Naming and Directory Interface (JNDI)

- The properties of data sources across the enterprise can be stored in a *directory*. Using a directory allows for centralized management of user names, passwords, database names, and JDBC URLs

- Reason:

  - `DriverManager` etc require programs to have built-in constants for things like URL of the database and the name of the driver class
  - This makes it harder to switch database being used

UNIVERSITY OF WOLLONGONG

DataSources

- DataSources

  – View them as little data structures that a program can load at runtime

  – These structures contain all the information needed to connect to database

  – Relatively easy to substitute different data structure and so change database used.

  – Program simply has a symbolic name for the datasource structure that it wants to use

Establish Connection

```
Context jndiContext = new InitialContext();
DataSource source =
        (DataSource) jndiContext.lookup("java:comp/env/jdbc/MyDatabase");
Connection conn = source.getConnection();
```

- The `DriverManager` is no longer involved. Instead, the JNDI service locates a data source

UNIVERSITY OF
WOLLONGONG

Supporting Infrastructure

- But where do these data structures get loaded from?
  - That symbolic name in the program – how does it get mapped to a record with the right `datasource` structure

- Supporting infrastructure
  - a directory service where can lookup symbolic name and find resource
  - JNDI – naming and directory services are required

UNIVERSITY OF WOLLONGONG