

Java DataBase Connectivity

SWEN304/SWEN435

Trimester 1, 2024

Lecturer: Dr Hui Ma

Engineering and Computer Science



Plan for Java DataBase Connectivity (JDBC)

- Motivation
- Architecture
- JDBC Classes and Interfaces
 - JDBC Driver Management
 - Controlling transaction behavior
- Executing SQL statements
- Obtaining result data
- Matching data types
- Exceptions
- Closing a connection

- *Further Reading: The Java™ Tutorial:*

`http://ecs.victoria.ac.nz/technical/java/tutorial/index.html`

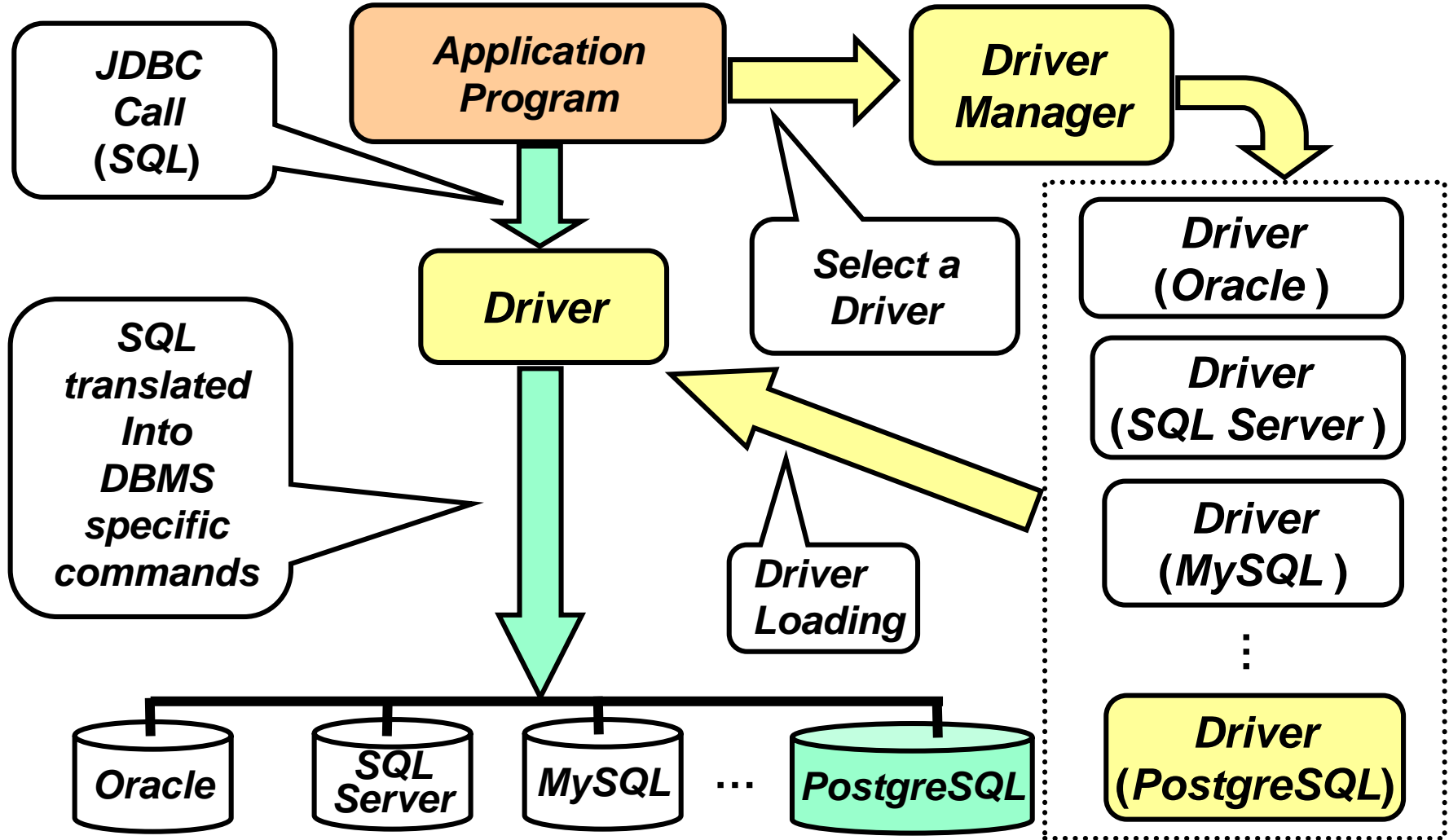
Motivation for Using JDBC

- In practice, databases are not only accessed by human users through the user interface, but also by application programs
- Application programs are written in a general purpose programming language (GPPL)
 - for example, Java, C/C++, Python, ...
- When developing applications, software engineers use SQL for data management inside their application program
- The application program is written in a GPPL with SQL statement **embedded** into it
 - therefore, the GPPL is also called the **'host' language**

The Java Database Connectivity (JDBC) API

- **Java Database Connectivity (JDBC)** is the standard application program interface (API) for accessing databases from a Java program
 - it allows us to embed SQL statements into Java code
- JDBC is supported by all relevant DBMS, including all major commercial and open source DBMS
 - Interactions of an application program with a specific DBMS are accomplished through a DBMS-specific **JDBC driver**
- Application programs using JDBC are independent of the particular DBMS that is used
 - Independence holds on source code & executable code level
 - Independence is a huge advantage when developing applications for multiple DBMSs
 - Independence is achieved by an **extra level** of indirection

Accessing Databases from Application Programs



JDBC Architecture (1)

- When using JDBC the following play a role :
 - Application programs
 - A database system (DBMS plus databases)
 - A DBMS-specific JDBC driver
 - A driver manager

JDBC Architecture (2)

- Application programs ...
 - Dynamically load the JDBC **drivers** needed,
 - Initiate a **connection** with a database,
 - Set transaction **boundaries** (BEGIN,..., {COMMIT | ROLLBACK}),
 - Acquire **locks**,
 - Submit **SQL** statements,
 - **Receive** data,
 - **Process** data,
 - **Process** error messages,
 - Decide whether to **commit** or **roll-back** a transaction,
 - Disconnect from the database to **terminate** a session

JDBC Architecture (3)

- The DBMS ...
 - Processes data manipulation commands,
 - Returns results to the application program
- The JDBC driver ...
 - Establishes **connection** with a database,
 - Submits data manipulation **requests**,
 - Accepts **results** returned by the DBMS,
 - Translates DBMS specific **data types** into Java data types,
 - Translates **error** messages
- The driver manager ...
 - Loads and supervises available JDBC drivers for various DBMS
 - Calls JDBC drivers to connect to a database

JDBC Classes

- JDBC is a collection of Java classes and interfaces
 - All these are put together in the `java.sql` package
- It contains methods for:
 - **Connecting** to a remote database,
 - **Executing** SQL statements,
 - **Iterating** over a set of tuples from a SQL statement,
 - **Transaction** management,
 - **Exception** handling

JDBC DriverManager class

- JDBC provides the `DriverManager` class
- Among others, it defines methods to enable dynamic addition and deletion of JDBC drivers :
 - `registerDriver()`
 - `deregisterDriver()`
- The first step in **connecting to a database** (managed by some DBMS) is to load a suitable JDBC driver for that particular DBMS
- Any current JDBC drivers that are found in the class path are automatically loaded

Registering a JDBC Driver

- However, drivers prior to JDBC 4.0 must be loaded manually with the method `Class.forName`

```
public static native Class.forName (String name)  
throws ClassNotFoundException
```

- This creates a `Driver` object for the respective JDBC driver
- Example: to load a JDBC driver for PostgreSQL use
 - `Class.forName ("org.postgresql.Driver") ;`
- If you are unsure which JDBC drivers will be available, load them manually

Establishing a Connection

- To connect to a database in a DBMS, the `getConnection` method of the `DriverManager` object is called
 - `Connection con = DriverManager.getConnection(url, [userId], [password]);`
 - This method requires a URL to the database
- In the application program, the method call starts a **session** with the database by creating a `Connection` object

Specifying the URL for the Connection

- The URL `url` to the database is of the form
`jdbc:[drivertype]:[database]`
- Herein,
 - `jdbc` is a constant,
 - `[drivertype]` is the type of the database we want to connect (e.g. `postgresql`), and
 - `[database]` is the address of the database in form
`//hostname[:portnumber]/database_name`
- Example:
`jdbc:postgresql://db.ecs.vuw.ac.nz/007_jdbc`

Connection Interface

- The interface `java.sql.Connection` has a number of classes and methods that are used:
 - To control **transactional** behavior of a `Connection` object,
 - To create and execute **SQL** statements,
 - To iterate over the **result** returned by a DBMS, and
 - To finish interaction with a database by **closing** the connection
- After acquiring a connection (with the name say `con`) and before it is closed, the same connection can be used for executing several transactions

Controlling Transaction Behavior-Start

- By default, a Connection **automatically** commits changes after executing each SQL statement

- The method

```
public abstract void setAutoCommit(  
boolean autoCommit) throws SQLException
```

is applied onto a `Connection` object

- To designate the **start** of a transaction (BEGIN point), we assign a value `false` to `autoCommit`

```
con.setAutoCommit(false);
```

Controlling Transaction Behavior-End

- A transaction is **terminated** using:

- Either

```
public abstract void commit() throws  
SQLException
```

- or

```
public abstract void rollback() throws  
SQLException
```

- And (after any of them)

```
con.setAutoCommit(true)
```

on the `Connection` object

What Next?

- Executing SQL statements
 - Statement object
 - PreparedStatement object
- Obtaining result data
- Matching data types
- Exceptions
- Closing a connection

Executing SQL Statements

- JDBC supports three different ways of executing SQL statements:
 - **Statement,**
 - **PreparedStatement, and**
 - `CallableStatement`

Statement Class and It's Subclasses

- The **Statement** class is the base class of the three classes used to submit queries to a DBMS
 - Its objects are used to send such SQL queries to a DBMS that are executed with no repetition within a transaction and that have no parameters
- **PreparedStatement** objects are used for SQL statements with parameters or for those that are executed multiple times (in a loop)
 - SQL statements of PreparedStatement objects may be precompiled yielding better performance
- `CallableStatement` objects are used with stored procedures and are out of the scope of the course

Submitting a SQL Query to a DBMS

- The following steps should be performed in order to submit a SQL statement to a DBMS either using a Statement (**S**) or PreparedStatement (**PS**) object:

1. Define a SQL query as a string
2. Create a S or PS object

If the SQL statement is one of CREATE, INSERT, DELETE, UPDATE, or SET type:

3. Apply **executeUpdate()** method onto the S or PS object

Else (the SQL statement is of the **SELECT** type):

3. Create a **ResultSet** object
4. Feed into the `ResultSet` object the return value of applying **executeQuery()** method onto the S or PS object

Statement Objects With `executeUpdate`

```
String insert="INSERT INTO Grades " +
"VALUES (007007,'C305','A+')";
```

```
Statement stmt=con.createStatement();
int return_value =
stmt.executeUpdate(insert);
```

- For INSERT, DELETE, and UPDATE queries, the return value will be the **number** of tuples affected
- For CREATE or SET, the return value should be **0**

ResultSet Object

- The `executeQuery()` method returns an object of the type `set` (or `superset`)
- This set object should be assigned to an object of the `ResultSet` class
- The `ResultSet` class has the `next()` method that allows traversing the set in a tuple at a time fashion
- Initially, the `ResultSet` object is positioned `before` the first tuple of the result
- The method `next()` returns `true` if there is a `next` tuple in the result, otherwise `false`
- After executing `next()`, the `ResultSet` object contains a pointer to the current tuple

Statement Objects With `executeQuery`

```
String select="SELECT * FROM Grades"  
+ "WHERE StudentId=007007";  
  
Statement stmt =  
con.createStatement();  
  
ResultSet rs =  
stmt.executeQuery(select);  
  
while (rs.next()) {  
    // extracting data from rs tuples  
    // data processing  
}
```

Extracting Data from the Result

- To match Java and database data types, JDBC specifies mappings and provides accessor methods in the `ResultSet` class

```
int j_studId;  
String j_courseId;  
String j_grade;  
while (rs.next()) {  
    j_studId=rs.getInt("StudentId");  
    j_courseId=rs.getString("CourseId");  
    j_grade=rs.getString(3)  
    // 3 is the column number in the result  
}
```


PreparedStatement **With** executeUpdate

```
String insert="INSERT INTO Grades VALUES (?, ?, ?)";
PreparedStatement prstmt =
    con.prepareStatement(insert);
boolean end=false;
while(!end) {
    // suppose j_studId, j_courseId, j_grade, and end
    //are dynamically initialized to desired values
    prstmt.setInt(1, j_studId);
    prstmt.setString(2, j_courseId);
    prstmt.setString(3, j_grade);
    int return_value = prstmt.executeUpdate();

    ...
}
```

PreparedStatement **with** executeQuery

```
String select= "SELECT * FROM Grades  
WHERE StudentId = ?";
```

```
PreparedStatement pstmt =  
    con.prepareStatement(select);
```

```
// suppose j_studId is initialized on the  
// desired value
```

```
pstmt.setInt(1, j_studId);
```

```
ResultSet rs =  
pstmt.executeQuery();
```

```
while(rs.next()) {  
  
}
```

Closing a Connection

- Before exiting from an application program all connections acquired should be closed by applying

```
public abstract void close() throws  
SQLException
```

method on each of them

Exceptions

- Most of the methods in `java.sql` can throw an exception of the type `SQLException` if an error occurs
- In addition to inherited `getMessage()` method, `SQLException` class has two additional methods for providing error information:
 - `public String getSQLState()` that returns an SQL state identifier according to SQL:1999 standard, and
 - `public int getErrorCode()` that retrieves a vendor specific error code
- Each JDBC method that throws an exception has to be placed inside a **try** block followed by a **catch** block

Code to See Exceptions

```
try{
/* Code that could generate an
      exception goes here. If an
      exception is generated, the
catch  block below will print out
      information about it*/
}
catch (SQLException ex) {
    System.println(ex.getMessage());
    System.println(ex.getSQLState());
    System.println(ex.getErrorCode());
}
```

Summary

- JDBC Transactions are executed by:
 - Acquiring a Driver,
 - Constructing a connection object,
 - Establishing transaction boundaries
 - Submitting SQL statements,
 - Retrieving results,
 - Processing either results returned or exception errors,
 - Committing or roll-backing transactions, and
 - Disconnecting from databases to terminate interaction
- **Statement** objects - SQL statements have no parameters,
- **PreparedStatement** objects - SQL statements have parameters