Java and databases

SoftEng
http://softeng.polito.it

Accessing databases in Java

- It can be extremely valuable to let Java applications access data stored in relational databases, and
  - Query existing data
  - Modify existing data
  - Insert new data
- Data can be used by
  - The algorithms running in the application
  - The end user, through a user interface

Accessing databases in Java

- A Database Management System (DBMS) needs to be installed
  - E.g., MySql, MariaDB, etc.
- A tool for testing queries on it before actually moving to Java is recommended
  - E.g., HeidiSql, TablePlus, phpMyAdmin, etc.
- They can be found (with other components) integrated in tools like XAMPP

Introduction to JDBC

SOftEng
http://softeng.polito.it

JDBC

- It is the acronym of Java Database Connectivity
- Standard library for accessing relational databases
  - Compatible with most/all different databases
  - Defined in packages java.sql and javax.sql

JDBC

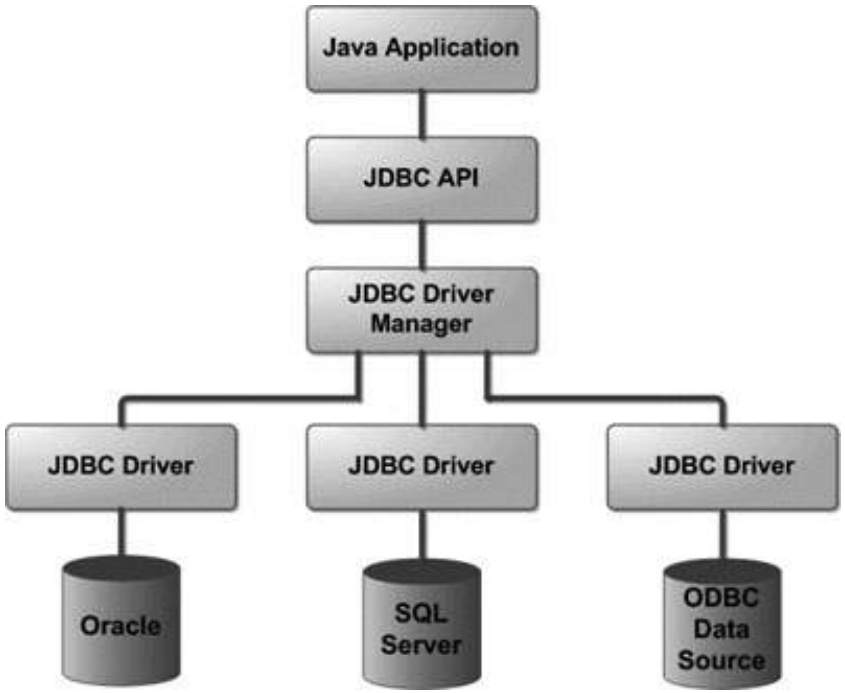
- Documentation:
  - Doc Index:
    - http://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/index.html
    - https://www.oracle.com/technetwork/java/javase/tech/index-jsp-136101.html
  - JDBC Overview:
    - http://www.oracle.com/technetwork/java/overview-141217.html
  - Tutorial:
    - http://docs.oracle.com/javase/tutorial/jdbc/basics/index.html

JDBC scope

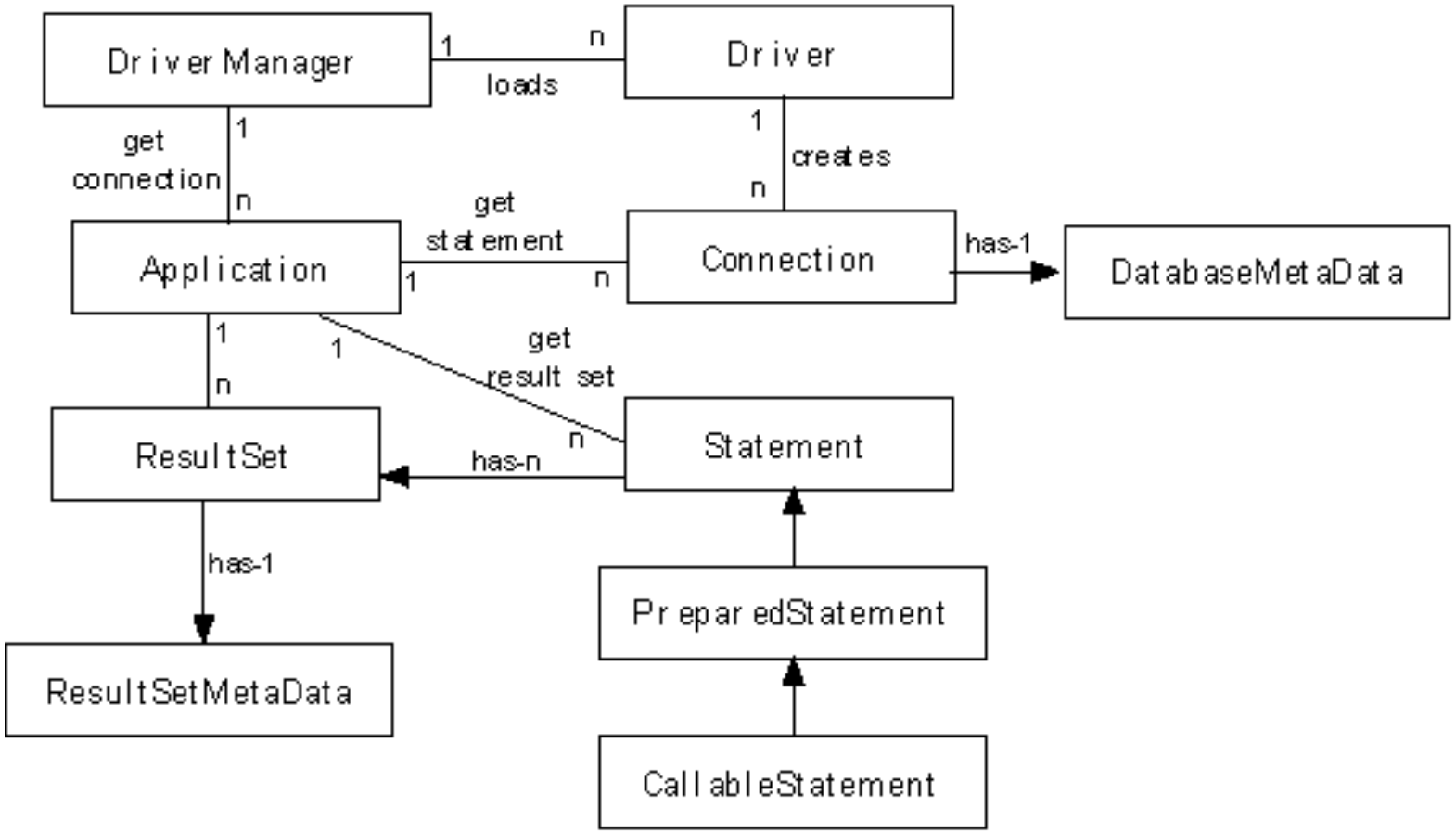
- Standardizes
  - Mechanism for connecting to DBMSs
  - Syntax for sending queries
  - Structure representing the results
- Does not standardize
  - SQL syntax: dialects, variants, extensions, ...

JDBC architecture

- Java application
- JDBC Driver Manager
  - For loading the JDBC Driver
- JDBC Driver
  - From DBMS vendor
- DBMS
  - In this case, MySQL (alternatives, Oracle, SQL Server, etc.)



Class diagram



JDBC Driver

- A Driver is a DMBS-vendor provided class, that must be available to the Java application
  - Should reside in the project's libraries
  - Should be accessible in the project's Class Path
- The application usually doesn't know the driver class name until run-time (to ease the migration to other DMBSs)
- Needs to find and load the class at run-time

JDBC Driver for MySql

- MySQL Connector/J
  - http://dev.mysql.com/downloads/connector/j/
  - **`mysql-connector-java-[version]-bin.jar`**
  - Copy or link it into project libraries
- The driver is in class
  - **`com.mysql.jdbc.Driver`**
  - …but developers don't need (want) to know that
- Documentation:
  - https://dev.mysql.com/doc/connector-j/8.0/en/

JDBC driver for MariaDB

- MariaDB Connector/J
  - https://mariadb.com/kb/en/mariadb-connector-j/
  - **`mariadb-java-client-x.x.x.jar`**
- The driver is in class
  - **`org.mariadb.jdbc.Driver`**
- Responds to JDBC URLs
  - jdbc:mariadb://…
  - jdbc:mysql://…

Steps for accessing a database

- Basis steps
  1. Define the connection URL
  2. Establish the connection
  3. Create a statement object
  4. Execute a query or update
  5. Process the results
  6. Close the connection

1. Define the connection URL

- The Driver Manager needs some information to connect to the DBMS
  - The database type (to call the proper Driver)
  - The server address
  - Authentication information (username/password)
  - Database/schema to connect to
- All these parameters are encoded in a string
  - The exact format depends on the Driver vendor

MySQL Connection URL format

- **jdbc:mysql://[host:port]/[database]**
  [?propertyName1][=propertyValue1]
  [&propertyName2][=propertyValue2]...
- Example:
  - **jdbc:mysql://**
  - **host:port** (usually: localhost)
  - **/database**
  - **?user=username**
  - **&password=the password** (omit for XAMPP)

2. Establish the connection

- Use **`DriverManager.getConnection`**
  - Static method that uses the appropriate driver according to the connection URL
  - And returns a **`Connection`** object
- **`Connection connection = DriverManager.getConnection(URLString)`**
- Contacts DBMS, validates user and selects the database
- On the **`Connection`** object, subsequent commands will execute queries

Example

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
 try {
   Connection conn = DriverManager.getConnection(
   "jdbc:mysql://localhost/test?user=root
   &password=secret");

   // Do something with the Connection
   ...
  } catch (SQLException ex) {
    // handle any errors
    System.out.println("SQLException: " + ex.getMessage());
    System.out.println("SQLState: " + ex.getSQLState());
    System.out.println("VendorError:" + ex.getErrorCode());
  }
```

3. Create a Statement object

- `Statement statement =`

  `connection.createStatement();`
- Creates a `Statement` object for sending SQL statements to the database
- SQL statements without parameters are normally executed using `Statement` objects
  - For efficiency and security reasons, it is always preferable to use `PreparedStatement` objects (see later)

4. Execute a query

- Use the **`executeQuery`** method of the Statement class
  - **`ResultSet executeQuery(String sql)`**
  - The sql string contains a SELECT statement
- The method returns a **`ResultSet`** object, that will be used to retrieve the query results

Example

```
String query = "SELECT id, name FROM user";
ResultSet resultSet = statement.executeQuery(query);
```

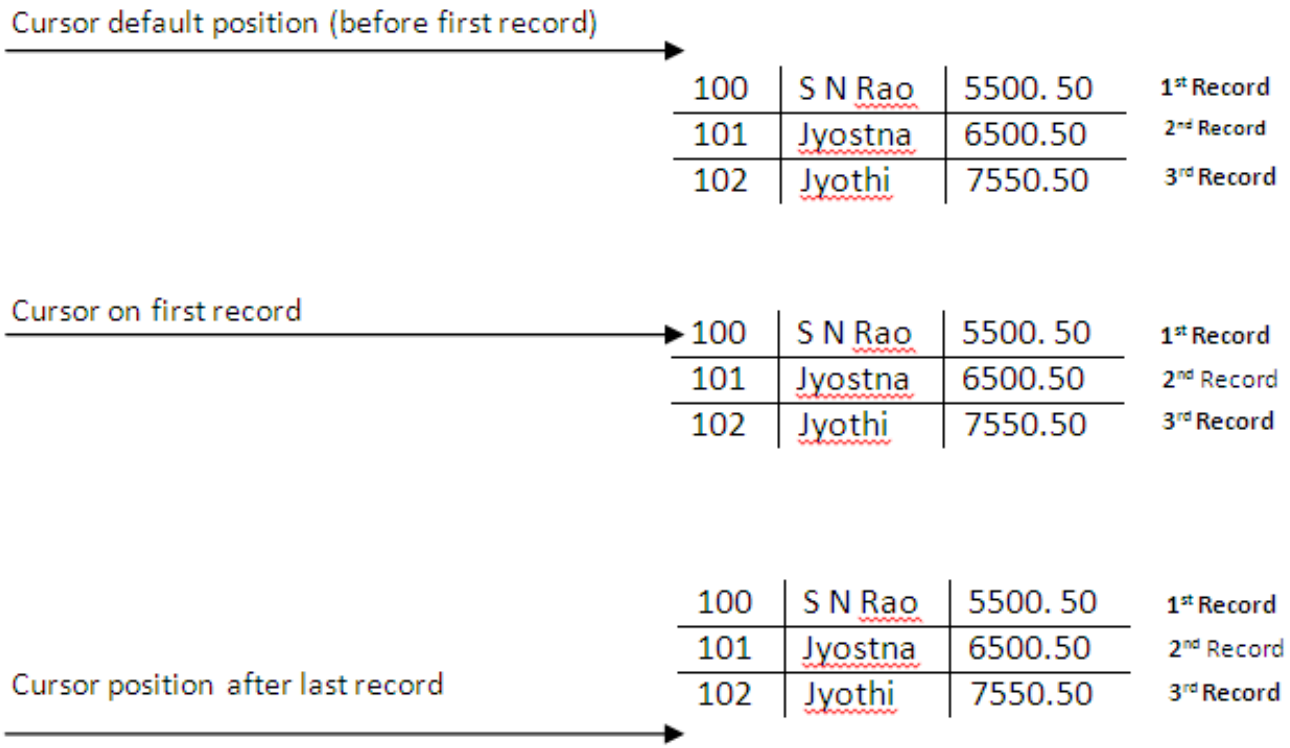
Other execute methods

- **`int executeUpdate(String sql)`**
  - For INSERT, UPDATE, or DELETE statements
  - For other SQL statements that don't return a resultset (e.g., CREATE TABLE)
  - Returns either the row count for INSERT, UPDATE or DELETE statements, or 0 for SQL statements that return nothing
- **`boolean execute(String sql)`**
  - For general SQL statements

5. Process the result

- The **`ResultSet`** object implements a "cursor", that can iterate over the query results
  - Data are available one row at a time
    - Method **`ResultSet.next()`** goes to the next row
  - The column values (for the selected row) are available through **`getXXX`** methods
    - **`getInt`**, **`getString`**, **`getBoolean`**, **`getDate`**, **`getDouble`**, ...
  - Data types are converted from SQL types to Java types

Cursor


Cursor default position (before first record)
100 S N Rao 5500. 50 1st Record
101 Jyostna 6500.50 2nd Record
102 Jyothi 7550.50 3rd Record
Cursor on first record
100 S N Rao 5500. 50 1st Record
101 Jyostna 6500.50 2nd Record
102 Jyothi 7550.50 3rd Record
Cursor position after last record
100 S N Rao 5500. 50 1st Record
101 Jyostna 6500.50 2nd Record
102 Jyothi 7550.50 3rd Record

ResultSet.getXXX methods

- **`XXX`** is the desired datatype
  - Must be compatible with the column type
  - **`String`** is almost always acceptable
- Two versions
  - **`getXXX(int columnIndex)`**
    - number of column to retrieve (starting from 1 !!!)
  - **`getXXX(String columnName)`**
    - name of column to retrieve
    - Always preferred

ResultSet navigation methods

- **`boolean next()`**
  - Moves the cursor down one row from its current position
  - A resultset cursor is initially positioned before the first row:
    - the first call to the method next makes the first row the current row
    - the second call makes the second row the current row, …

Other navigation methods

- Query cursor position
  - `boolean isFirst()`
  - `boolean isLast()`
  - `boolean isBeforeFirst()`
  - `boolean isAfterLast()`

Other navigation methods

- Move cursor
  - `void beforeFirst()`
  - `void afterLast()`
  - `boolean first()`
  - `boolean last()`
  - `boolean absolute(int row)`
  - `boolean relative(int rows)`
  - `boolean previous()`

Example

```
while( resultSet.next() )
{
    System.out.println(
       resultSet.getInt("ID") + " - " +
       resultSet.getString("name") ) ;
}
```

Datatype conversions (MySQL)

MySQL Data Types	Corresponding Java types
`CHAR, VARCHAR, BLOB, TEXT, ENUM, and SET`	`java.lang.String, java.io.InputStream, java.io.Reader, java.sql.Blob, java.sql.Clob`
`FLOAT, REAL, DOUBLE PRECISION, NUMERIC, DECIMAL, TINYINT, SMALLINT, MEDIUMINT, INTEGER, BIGINT`	`java.lang.String, java.lang.Short, java.lang.Integer, java.lang.Long, java.lang.Double, java.math.BigDecimal`
`DATE, TIME, DATETIME, TIMESTAMP`	`java.lang.String, java.sql.Date, java.sql.Timestamp`

6. Close the connection

- Additional queries may be done on the same connection
  - Each returns a different **`ResultSet`** object, unless one re-uses it
  - When no longer needed, **`ResultSet`** resources can/should be freed by "closing" with **`resultSet.close()`**
- When no additional queries are needed, close the connection to the database:
  - **`connection.close();`**

Parametric queries

- SQL queries may depend on user input data
  - Example: find item whose code is specified by the user
  - Method 1: String interpolation (e.g., with concatenation)

```
String query = "SELECT * FROM items
                    WHERE code='"+userCode+"'" ;
```

  - Method 2: use prepared statements
    - Always preferable
    - See later

Prepared statements and callable statements

SoftEng
http://softeng.polito.it

Problems with statements

```
String user;
String sql = "select * from users where
                                    username='" + user + "'" ;
```

- Problems:
  - Security
  - Performance

Security risk

- SQL injection: syntax errors or privilege escalation
- Example
  - Username: `'; delete * from users ; --`
  - `select * from users where username='';`
    `delete * from users ; -- '`
- Must detect or *escape* all dangerous characters!
  - Will never be perfect…
- Never trust user-entered data!

Performance limitation

- Performance issues
  - Query must be re-parsed and re-optimized every time
  - Complex queries require significant set-up overhead
- When the same query is repeated (even with different data), parsing and optimization wastes CPU time in the DBMS server
  - Increased response-time latency
  - Decreased scalability of the system

Prepared statements

- Idea: separate statement creation from statement execution
  - At creation time: define SQL syntax (template), with placeholders for variable quantities (parameters)
  - At execution time: define actual quantities for placeholders (parameter values), and run the statement
- Replace `Statement` with `PreparedStatement`

Prepared statements

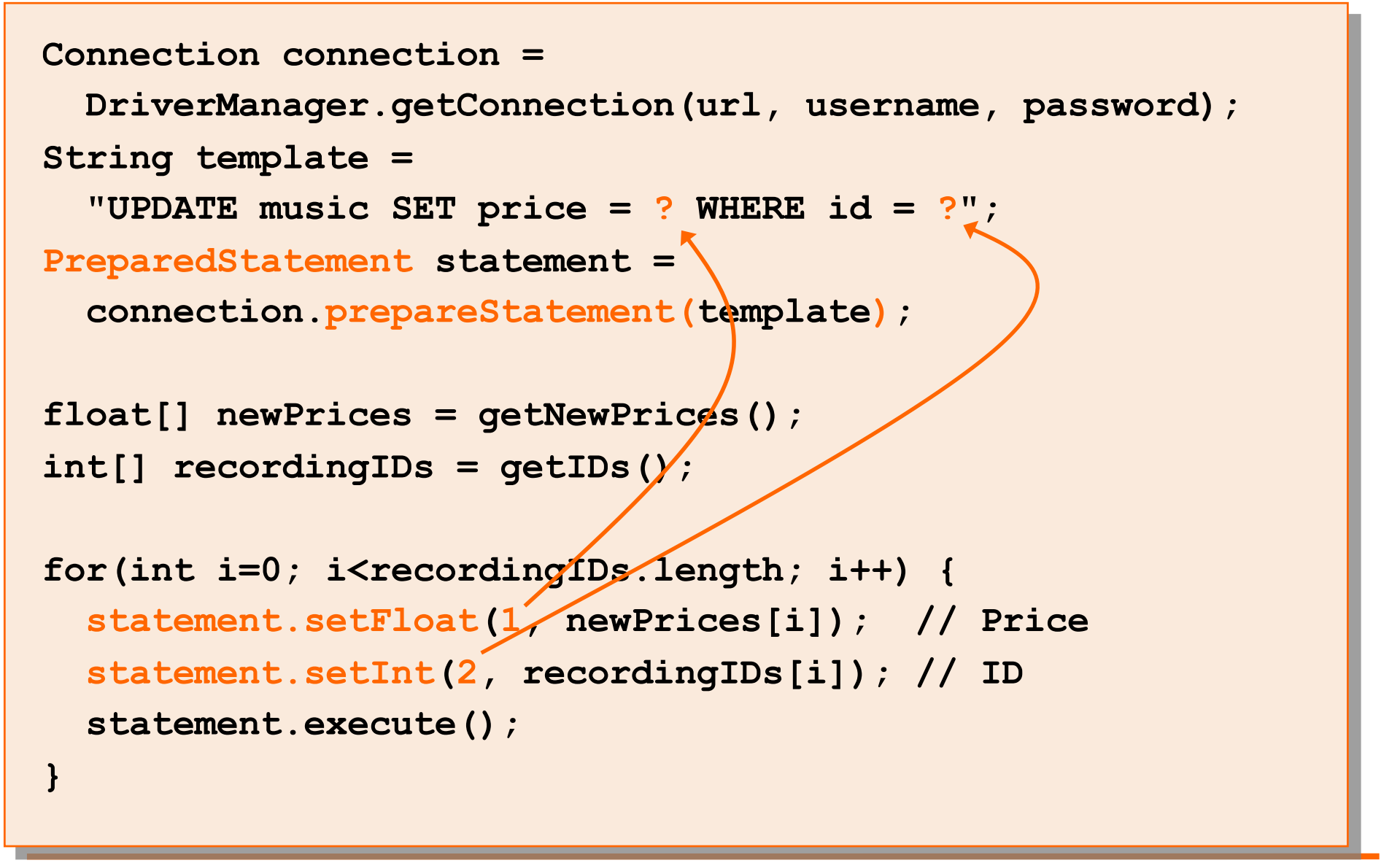
- Prepared statements can be re-run many times
- Parameter values are automatically
  - Converted according to their Java type
  - Escaped, if they contain dangerous characters
  - Handle non-character data (serialization)

Example

```
Connection connection =
  DriverManager.getConnection(url, username, password);
String template =
  "UPDATE music SET price = ? WHERE id = ?";
PreparedStatement statement =
  connection.prepareStatement(template);

float[] newPrices = getNewPrices();
int[] recordingIDs = getIDs();

for(int i=0; i<recordingIDs.length; i++) {
  statement.setFloat(1, newPrices[i]);  // Price
  statement.setInt(2, recordingIDs[i]); // ID
  statement.execute();
}
```

Example

```
Connection connection =
  DriverManager.getConnection(url, username, password);
String template =
  "UPDATE music SET price = ? WHERE id = ?";
PreparedStatement statement =
  connection.prepareStatement(template);

float[] newPrices = getNewPrices();
int[] recordingIDs = getIDs();

for(int i=0; i<recordingIDs.length; i++) {
  statement.setFloat(1, newPrices[i]);  // Price
  statement.setInt(2, recordingIDs[i]); // ID
  statement.execute();
}
```



Prepared statements

- Easier to write
  - Data type conversion done by JDBC library
- Secure (no SQL injection possible)
  - Quoting is done by JDBC library
- More efficient
  - Query re-use
  - Parameter values sent in binary form
- The bottom line:
  - Always prefer prepared statements!

Callable statements

- Many DBMSs allow defining “stored procedures”, directly defined at the DB level
- Stored procedures are SQL queries (with parameters), or sequences of queries
  - Language for defining stored procedures is DBMS-dependent: not portable!
- Calling stored procedures
  - Use class **`CallableStatement`** in JDBC

Data Access Object (DAO)

SoftEng
http://softeng.polito.it

Problems

- Database code involves a lot of “specific” knowledge
  - Connection parameters
  - SQL commands
  - The structure of the database
- Bad practice to “mix” this low-level information with main application code
  - Reduces portability and maintainability
  - Creates more complex code
- What it a better code organization?

Goals

- Encapsulate database access into separate classes, distinct from application ones
  - All other classes should be shielded from database details (hiding)
- Database access should be indepentent from application needs
  - Potentially reusable in different parts of the application
- Develop a reusable development pattern that can be easily applied to different situations

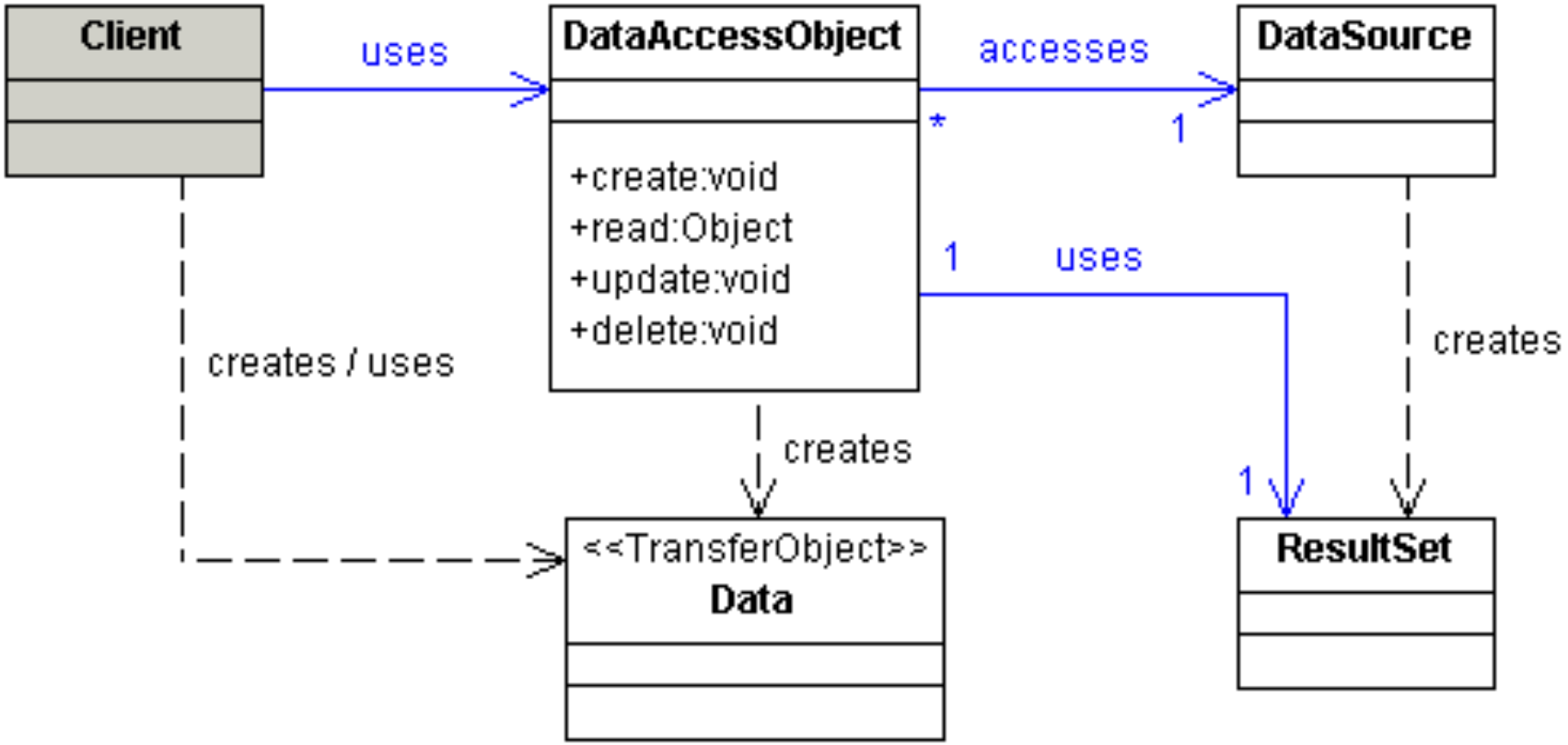
Data Access Object (DAO)

- "Client" classes:
  - Application code that needs to access the database
  - Ignorant of database details (connection, queries, schema, ...)
- "DAO" classes:
  - Encapsulate all database access code (JDBC)
  - The only ones that will ever contact the database
  - Ignorant of the goal of the Client

Data Access Object (DAO)

- Low-level database classes: DriverManager, DataSource, ResultSet, etc
  - Used by DAO (only!) but invisible to Client
- "Transfer Object" (TO) or "Data Transfer Object" (DTO) classes
  - Contain data sent from Client to DAO and/or returned by DAO to Client
  - Represent the data model, as seen by application
  - Ignorant of DAO, ignorant of database, ignorant of Client

DAO class diagram



DAO (class) design criteria

- DAO has no state
  - No instance variables (except **`Connection`**, in case)
- DAO manages one *kind* of data
  - Uses a small number of DTO classes and interacts with a small number of DB tables
  - If you need more, create many DAO classes
- DAO offers CRUD methods
  - Create, Read, Update, Delete
- DAO may offer search methods
  - Returning collections of DTO