Prerequisite knowledge

- RDBMS schema and SQL - DDL:

  `PRIMARY KEY`, `UNIQUE`, `FOREIGN KEY`, `NOT NULL`, datatypes.

- SQL, SQL - DML:

  Predicate based queries, joins.

- Transactions, ACID principle:

  Isolation level 1 - 4.

Persistence [Bauer2015]

Persistence allows an object to outlive the process that created it.

The state of the object may be stored to disk and an object with the same state re-created at some point in the future.

Java™ transient instances

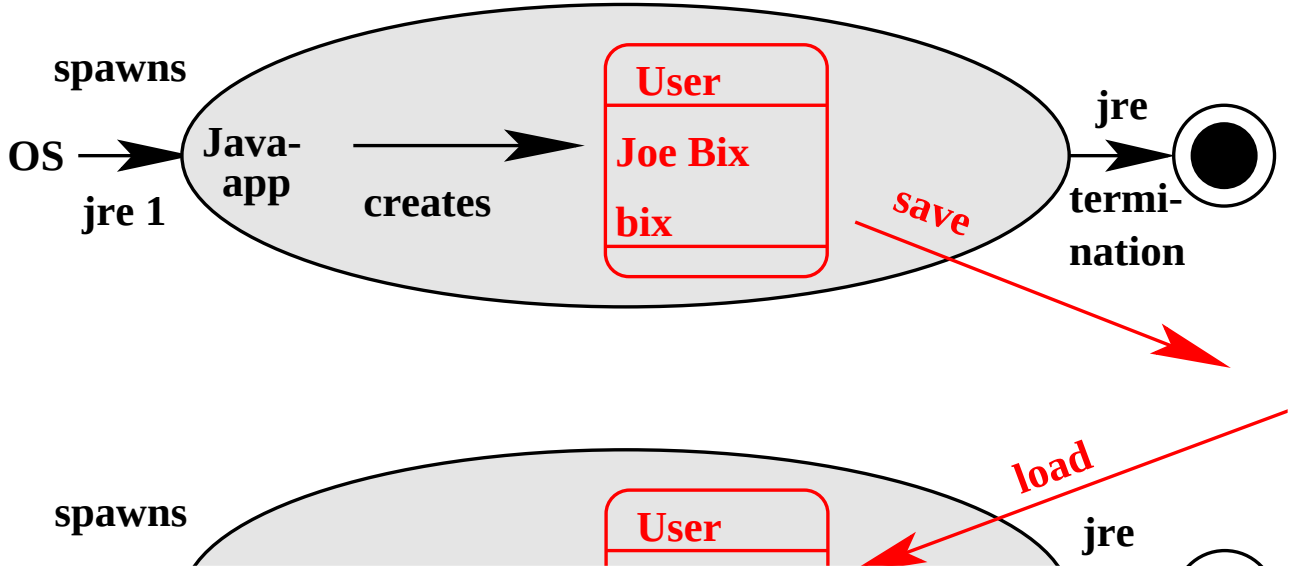
```
public class User {
  String commonName; // Common name e.g. 'Joe Bix'
  String uid;        // Unique login name e.g. 'bix'
  ...                // getters, setters and other stuff

}
//-------------------------------------
// Thread lifespan (transient instance)
User u = new User("Joe Bix", "bix");
```

RDBMS persistent records

```
CREATE TABLE User(
  commonName CHAR(80)
 ,uid CHAR(10) PRIMARY KEY
);
-- Persistent record (see Durability in ACID)
INSERT INTO User VALUES('Joe Bix', 'bix');
```

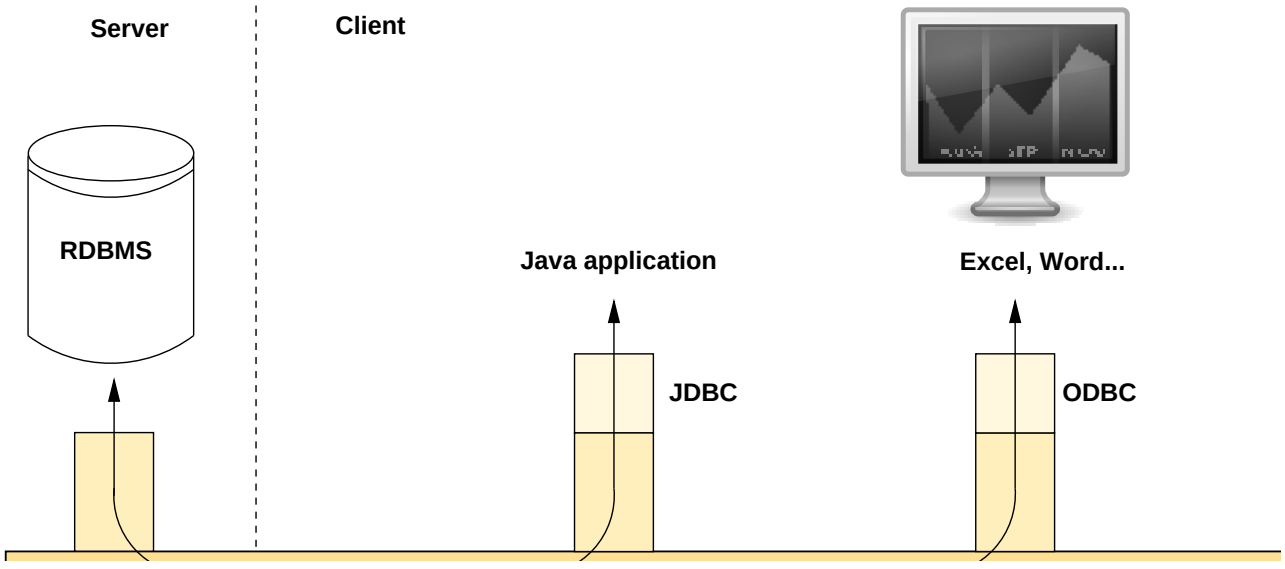
Persisting transient `User` instances



Observations

- Processes in disjoint address spaces:
  1. JRE™ runtime.
  2. RDBMS server.
- Multiple runtimes possible (PHP)
- “save” and “load” denote communications across OS boundaries.

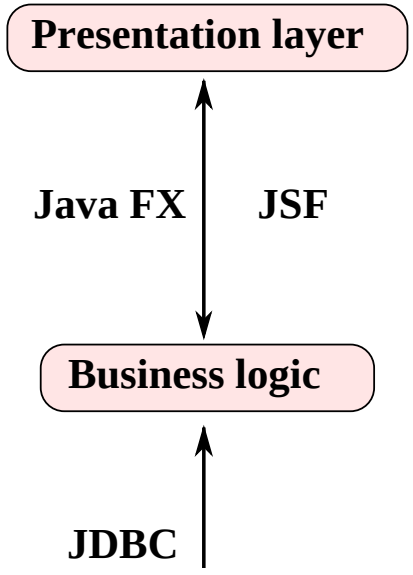
Networking between clients and database server



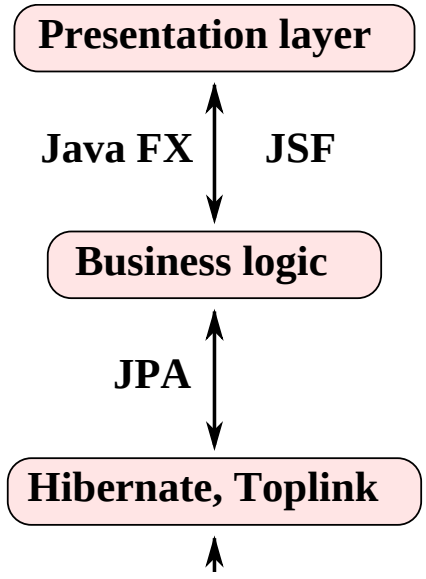
JDBC™ features

- Protocol connecting database client and server.
- Vendor dependent implementations.

JDBC™ in a three-tier application



JDBC™ connecting application server and database.



JDBC™ connection parameter

1. Database server type i.e. Oracle, DB2, Informix, Postgresql™, Mysql etc. due to vendor specific JDBC™ protocol implementations.
2. Server DNS name or IP number.
3. Server service's port number.
4. The database name within the given server.
5. Optional: A database user's account name and password.

Components of a JDBC™ URL

jdbc:mysql://srv.company.com:3306/foo

1. Protocol / sub protocol definition

2. Server's DNS name or IP-address

3. TCP service's port number

4. Database within

IETF Uniform Resource Identifier

https://www.ietf.org/rfc/rfc2396.txt:

```
absoluteURI   = scheme ":" ( hier_part | opaque_part )

hier_part     = ( net_path | abs_path ) [ "?" query ]

net_path      = "//" authority [ abs_path ]

abs_path      = "/"  path_segments
...
```

URL examples

- `http://www.hdm-stuttgart.de/aaa`
- `http://someserver.com:8080/someResource`

  Non-standard port 8080

- `ftp://mirror.mi.hdm-stuttgart.de/Firmen`

Sub protocol examples

Database	JDBC™ URI
PostgreSQL	jdbc:postgresql://<HOST>:<PORT>/[database]
MySQL	jdbc:mysql://[host][:port]/[database][?p1=v1]...
Oracle	jdbc:oracle:thin:[user/password]@[host][:port]:SID
DB2	jdbc:db2://<HOST>:<PORT>/[database]
Derby	jdbc:derby://[host][:port]/[database]
MS. SQL S.	jdbc:sqlserver://host[:port];user=xxx;password=xyz
Sybase	jdbc:sybase:Tds:<HOST>:<PORT>/[database]

No standard port assignments ...

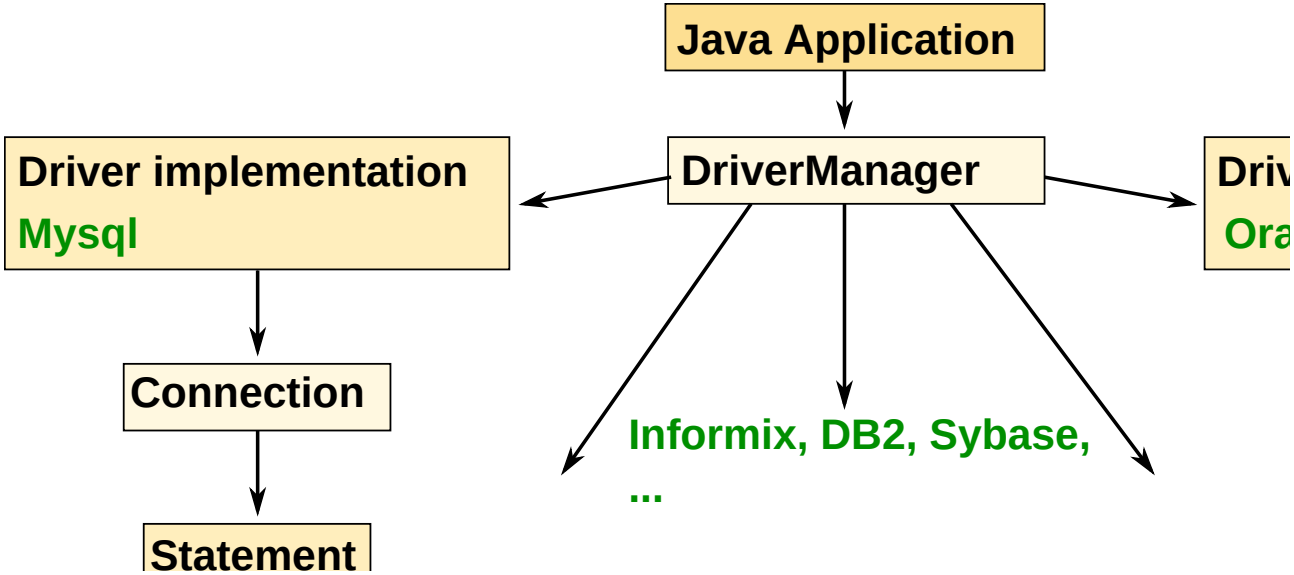
Postgresql:	5432
IBM DB2:	50000
Oracle:	1521

- No official IETF standard port assignments
- Vendor specific defaults
- Explicit port specification required

... but Postgresql made it into Linux

```
>grep postgresql /etc/services
postgresql      5432/tcp        postgres      # PostgreSQL Database
postgresql      5432/udp        postgres
```

JDBC™ architecture



`DriverManager`: Bootstrapping connections

- Bootstrapping object.
- `java.sql.DriverManager` shipped with JRE™.
- Interfacing JRE™ and JDBC™ driver.
- Provides instances of `java.sql.Connection`.
- See Interfaces and classes in JDBC™.

Example: Mysql connection implementation

- Interface `MySQLConnection` `extends` `java.sql.Connection`
- Class `ConnectionImpl` implements `MySQLConnection`

Driver libraries

- `postgresql-42.1.4.jar`
- `mysql-connector-java-x.y.z.jar`
- `ojdbc6.jar`

Driver libraries by Maven

- ```
  <groupId>postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <version>9.1-901-1.jdbc4</version>
  ```

- ```
  <groupId>com.oracle</groupId>  <!-- requires access credentials -->
  <artifactId>ojdbc7</artifactId>
  <version>12.1.0</version>
  ```

Driver unavailable

- `conn = DriverManager.getConnection(`
  `"jdbc:postgresqll://localhost/hdm", "hdmuser", "XYZ");`

- `java.sql.SQLException: No suitable driver found for`
  `jdbc:postgresqll://localhost/hdm`

  ```
   at java.sql.DriverManager.getConnection(DriverManager.java:689)
   at java.sql.DriverManager.getConnection(DriverManager.java:247)
   at de.hdm_stuttgart.mi.sda1.DatabaseTest.initDatabase(DatabaseTest.java:34)
  ...
  ```

`Connection` interface

`java.sql.Connection`

- Holding a permanent database server connection .
- Stateful protocol.
- Per connection properties: Isolation level, auto commit,...
- `rollback()` / `commit()`.

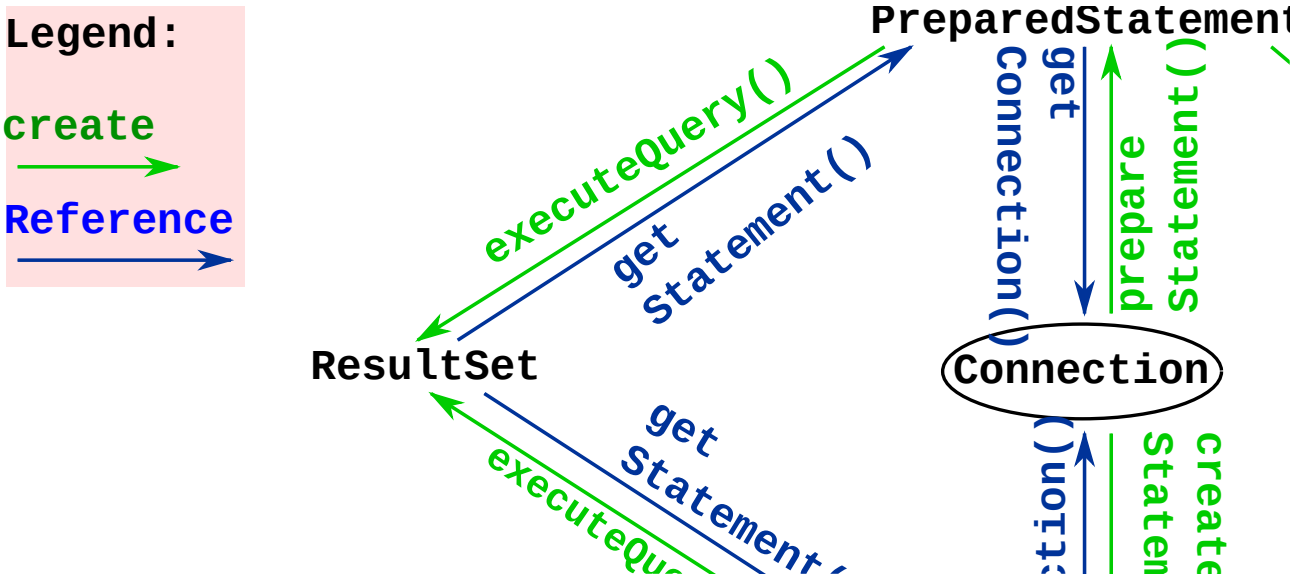
`Statement` interface

`java.sql.Statement` Two distinct operation classes:

`executeUpdate()` `INSERT`, `UPDATE`, `DELETE`: Integer return code

`executeQuery()` `SELECT`: Returning `java.sql.ResultSet`, see the section called “Read Access”.

JDBC™ instances and relationships.



Important `Connection` methods

- createStatement()
- setAutoCommit(), getAutoCommit()
- getWarnings()
- isClosed(), isValid(int timeout)
- rollback(), commit()
- close()

Important `Statement` methods

- executeUpdate(String sql)
- getConnection()
- getResultSet()
- close() and isClosed()

JDBC™ and threading.

From JDBC and Multithreading:

> “Because all Oracle JDBC API methods are synchronized, if two threads try to use the connection object simultaneously, then one will be forced to wait until the other one finishes its use.”

Consequence:

- Use one `java.sql.Connection` per thread.
- Use connection pooling e.g. c3po.

JDBC™ connection pooling

```
try (final Connection conn =
       C3P0DataSource.getInstance().getConnection()) {

  final PreparedStatement pstmt = conn.create...;
    ...
  pstmt.executeUpdate();
  // Auto close connection, back to pool.
} catch (SQLException e) {
  e.printStackTrace();
}
```

pom.xml driver **runtime** scope

```
...
<dependency>
  <groupId>postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <version>9.1-901-1.jdbc4</version>
  <scope>runtime</scope>
</dependency> ...
```

Related exercises

Exercise 1: Why `<scope>runtime</scope>`?

Person table

```
CREATE TABLE Person (
    name CHAR(20)
   ,email CHAR(20) UNIQUE
)
```

Objective: insert person record

- Java™ application executing:

```
INSERT INTO Person VALUES('Jim', 'jim@foo.org')
```

- No database read required (No `java.sql.ResultSet`).
- Success / failure related database return parameter.

JDBC™ backed data insert

```
// Step 1: Open connection to database server
final Connection conn = DriverManager.getConnection (
  "jdbc:postgresql://localhost/hdm",          // Connection parameter URL
  "hdmuser",                                  // Username
  "XYZ");                                     // Password

// Step 2: Create a Statement instance
final Statement stmt = conn.createStatement();

// Step 3: Execute the desired INSERT
final int updateCount = stmt.executeUpdate(
  "INSERT INTO Person VALUES('Jim', 'jim@foo.org')");

// Step 4: Give feedback to the end user
System.out.println("Successfully inserted " + updateCount + " dataset(s)");
```

Result

- Execution yields:

```
Successfully inserted 1 dataset(s)
```

- Note: The database server returns the number of inserted / modified / deleted datasets.

Two JDBC™ configurations

1. IDE level.

2. Project level (Maven).

Related exercises

Exercise 2: Exception on inserting objects

Figure 20.34, “JDBC™ backed data insert” deficiencies

Missing exception handling:

```
public static void main(String[] args)
  throws SQLException { ...
```

Hard coded connection parameters:

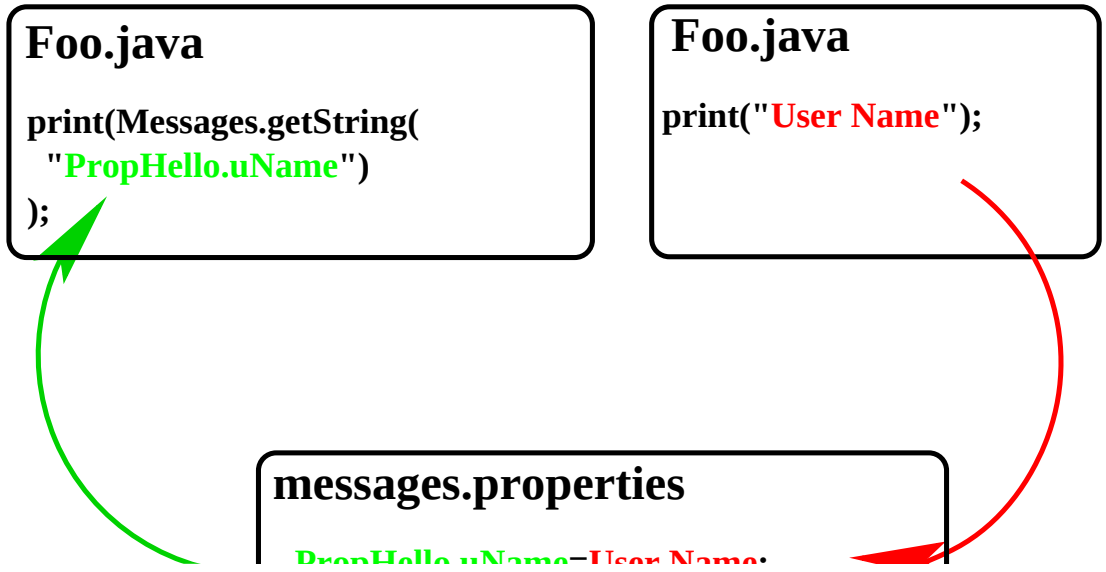
```
... = DriverManager.getConnection (
  "jdbc:postgresql://localhost/hdm", //JDBC URL
  "hdmuser",                         // Username
  "XYZ")                             // Password
```

Why properties?

- Connection parameter changes require recompilation!
- Parameters should be configurable.

    Possible solution: Java™ properties.

`message.properties` string externalization



Properties code sketch

Properties key / value file `resources/jdbc.properties`

```
jdbcurl=jdbc:postgresql://localhost/hdm
username=hdmuser
password=XYZ
```

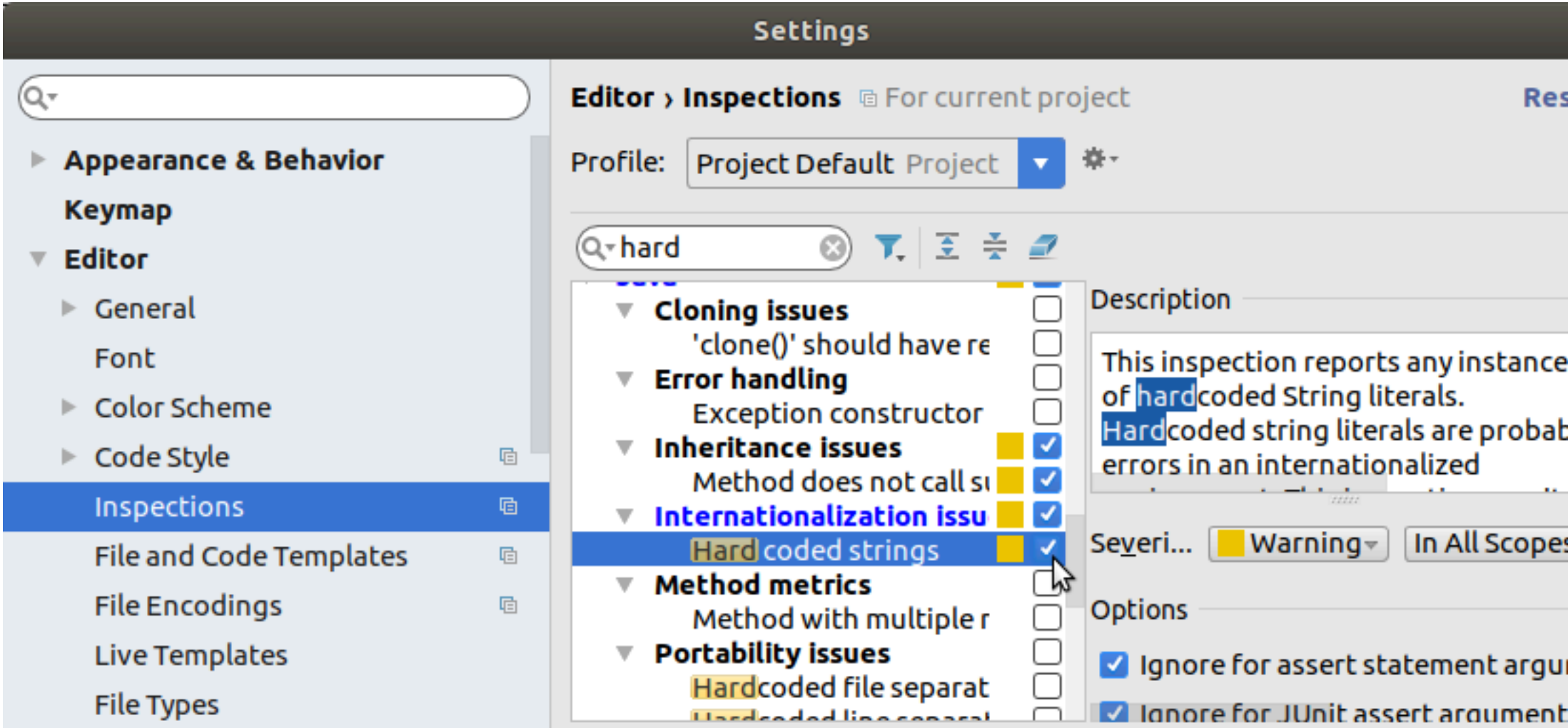
ResourceBundle reading properties

```
// resources/jdbc.properties
ResourceBundle jdbcProperties = ResourceBundle.getBundle("jdbc");
```

Using ResourceBundle

```
... Connection conn = DriverManager.getConnection(
      jdbcProperties.getString("jdbcurl"),
      jdbcProperties.getString("username"),
      jdbcProperties.getString("password"));
```

Intellij IDEA settings, preconditions



Database related unit test phases

1. **Set up:** Test preparation.
   - Open database connection
   - Create a required schema.
   - Optional: Insert initial data.
2. **Test:** Execute JDBC™ CRUD / SELECT operations.
3. **Tear down:**
   - Drop schema
   - Close database connection.

Implementing unit tests

```
public class InsertTest {
  static private Connection conn;
  static private Statement stmt;

  @BeforeClass ❶ static public void initDatabase() throws SQLException {
    conn = DriverManager.getConnection(
      SimpleInsert.jdbcProperties.getString("jdbcurl"),
      SimpleInsert.jdbcProperties.getString("username"),...);
    ScriptUtils.executeSqlScript(conn, new ClassPathResource("schema.sql"));
    stmt = conn.createStatement();}

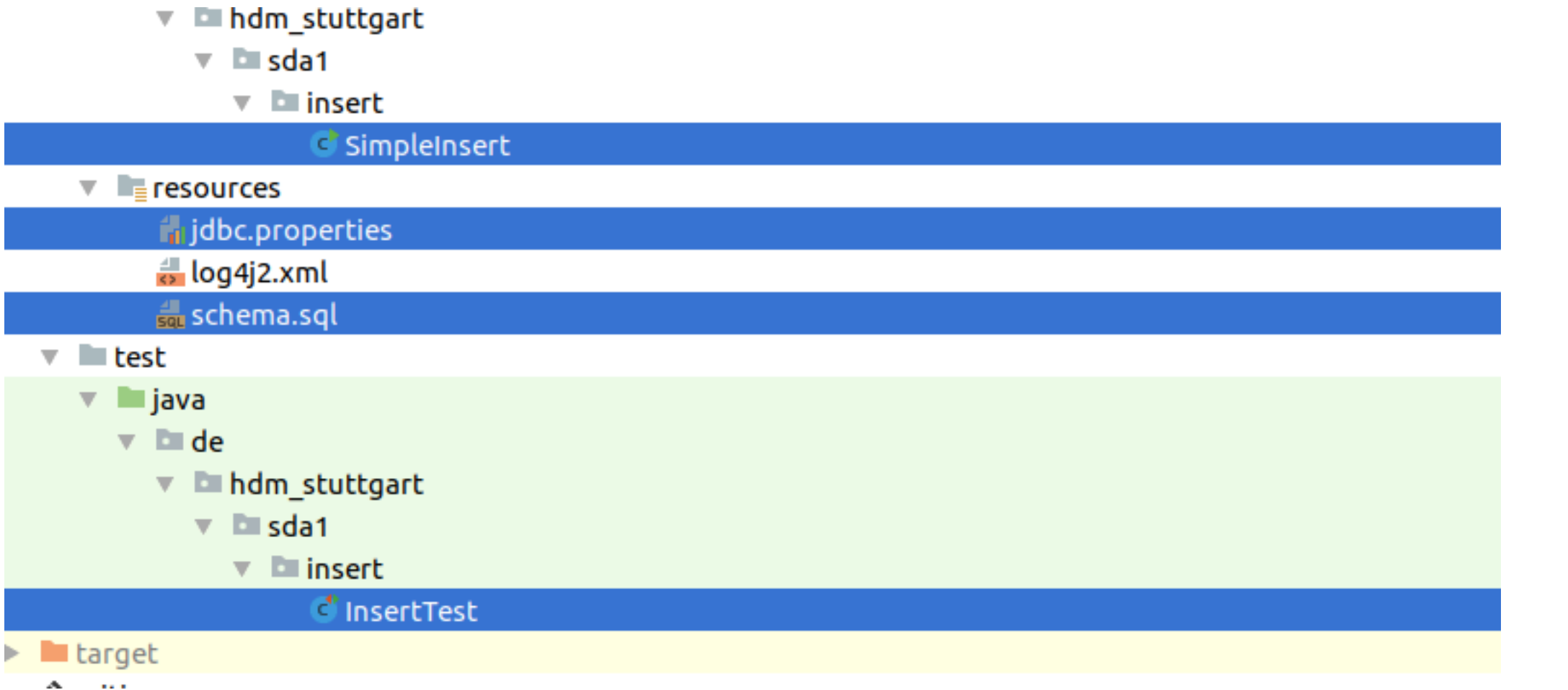
  @Test ❷
  public void test_010_insertJill() throws SQLException {
    Assert.assertEquals(1, SimpleInsert.insertPerson(
         stmt, "Jill", "jill@programmer.org"));
 }
@AfterClass ❸ static public void releaseDatabase()
   throws SQLException {conn.close();}
```

Spring is your friend

Getting **ScriptUtils.executeSqlScript(...)** to work:

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-jdbc</artifactId>
  <version>5.3.1</version>
  <scope>test</scope>
</dependency>
```

Project layout



Closing connections

```
final Connection conn = DriverManager.getConnection(...);
... // CRUD operations
conn.close(); // Important! Wanna use a connection pool instead?
```

Employ `AutoCloseable`

Using try-with-resources statement.

```
try (final Connection conn = DriverManager.getConnection(...) {
  ... // CRUD operations
} catch (SQLException e) {...}
```

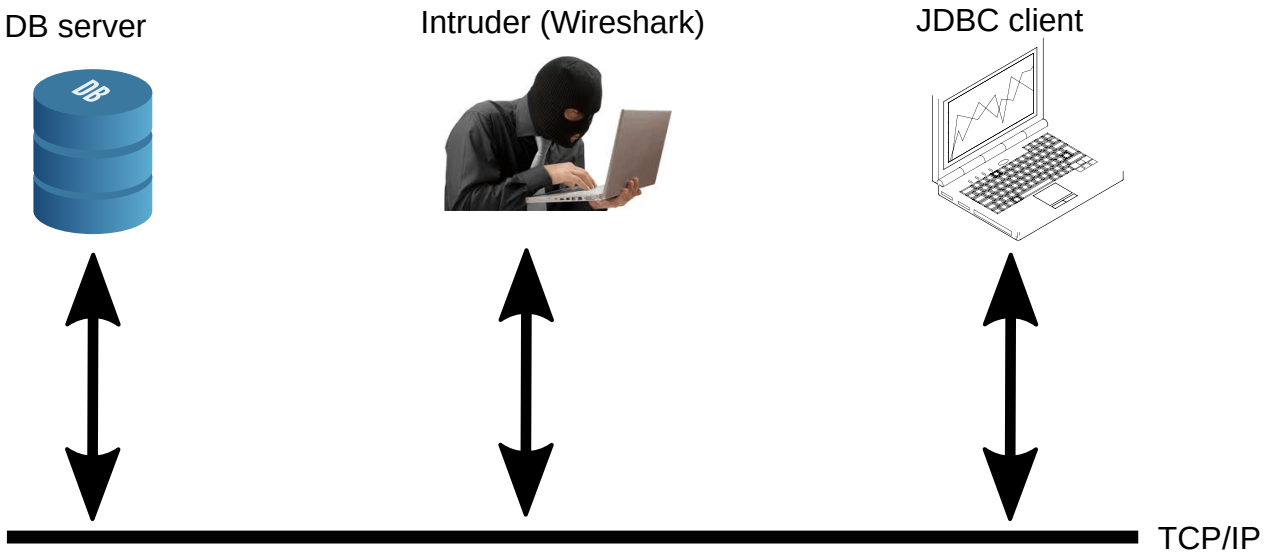
Related exercises

Exercise 3: Interactive inserts, connection properties, error handling and unit tests
Exercise 4: Interfaces and classes in JDBC™
Exercise 5: Closing JDBC™ connections
Exercise 6: Driver dispatch mechanism

Sniffing a JDBC™ connection by an intruder.

Sniffing a JDBC™ connection by an intruder.

Sniffing a JDBC™ connection by an intruder.



Setting up Wireshark

- Database server and JDBC™ client on same machine.
- Connecting to the `loopback` (lo) interface only.

  (Sufficient since client connects to `localhost`)
- Capture packets of type TCP having port number 3306.

Capturing results

```
[...
5.5.24-0ubuntu0.12.04.1.%...X*e?I1ZQ.................e,F[yoA5$T[N.mysql_native_password.
 A..........!.....................hdmuser ❶......U.>S.%..~h...!.xhdm...........j..../*

 ... INSERT INTO Person VALUES('Jim', 'jim@foo.org') ❷6...
  .&.#23000Duplicate entry 'jim@foo.org' for key 'email' ❸
```

❶ `username` initiating database connection.
❷ `INSERT(...)` statement.
❸ Resulting error message sent back to the client.

Password?

Mysql™ security

What about the missing password?

Making MySQL Secure Against Attackers:

When you connect to a MySQL server, you should use a password.

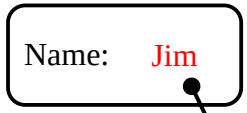
The password is not transmitted in clear text over the connection.

Mysql™ security

- Data exchange client to server nearly fully disclosed.
- Mysql mitigates the attack type's severity
- Possible solutions:
  - Encrypted tunnel between client and server: like e.g. ssh port forwarding or VPN.
  - Use JDBC™ driver supporting TLS.
- Irrelevant e.g. within DMZ.

Assembling SQL

GUI Window



Assembling SQL

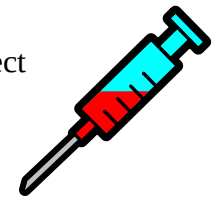
INSERT INTO Person VALUES('Jim')

SQL injection principle

GUI Window

Name: Jim'); DROP TABLE Person; INSERT INTO Person VALUES('Joe

SQL inject



INSERT INTO Person
VALUES(' ')

SQL injection principle

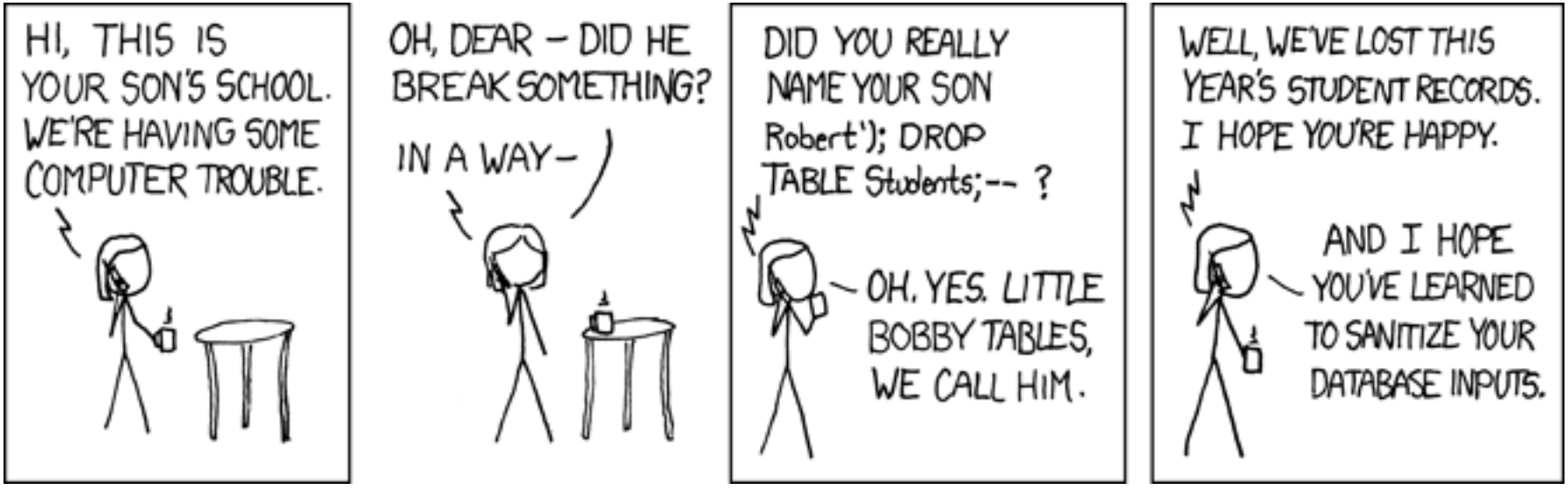
GUI Window

Name: Jim'); DROP TABLE Person; INSERT INTO Person VALUES('Joe

```
INSERT INTO Person
VALUES('Jim'); DROP TABLE Person; INSERT INTO Person VALUES('Joe')
```

Preventing traffic tickets

Trouble at school



SQL injection impact

- **Heartland Payment Systems data breach**
- **March 2008**
- **134 million credit cards exposed through SQL injection to i**
- **Cost: At least $129 million**

The vulnerability to SQL injection was well understood and s

SQL injection relevance, [Clarke2009]

Many people say they know what SQL injection is, but all they have heard about or experienced are trivial examples.

SQL injection is one of the most devastating vulnerabilities to impact a business, as it can lead to exposure of all of the sensitive information stored in an application's database, including handy information such as user's names, passwords, names, addresses, phone numbers, and credit card details.

Related exercises

Exercise 7: Attack from the dark side

Handling injection attacks, part 1

Keep the database server from interpreting user input completely.

This is the preferred way eliminating security issues completely as being discussed in the section called “`java.sql.PreparedStatement`”.

May not be possible in legacy applications due to required efforts.

Handling injection attacks, part 2

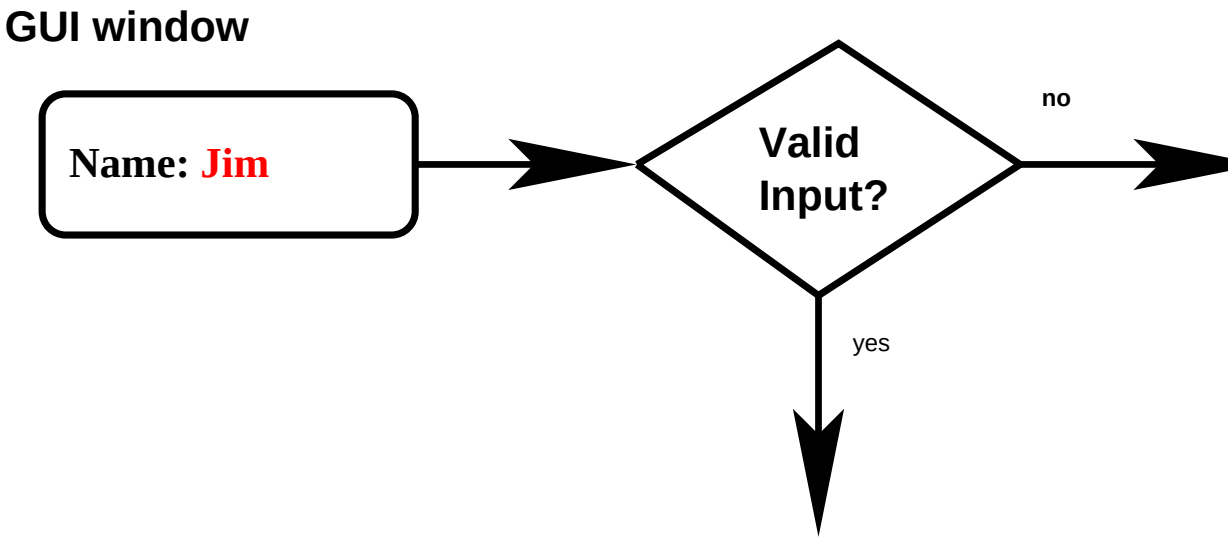
Let the application check user input beforehand	Malicious user input is being rejected from being embedded into SQL statements.

Input filtering

Regular expression matching user names.

Regular expression	User input
`[a-zA-Z]+`	Matches “`Jennifer`”
	Does not match “`DROP TABLE Users`”

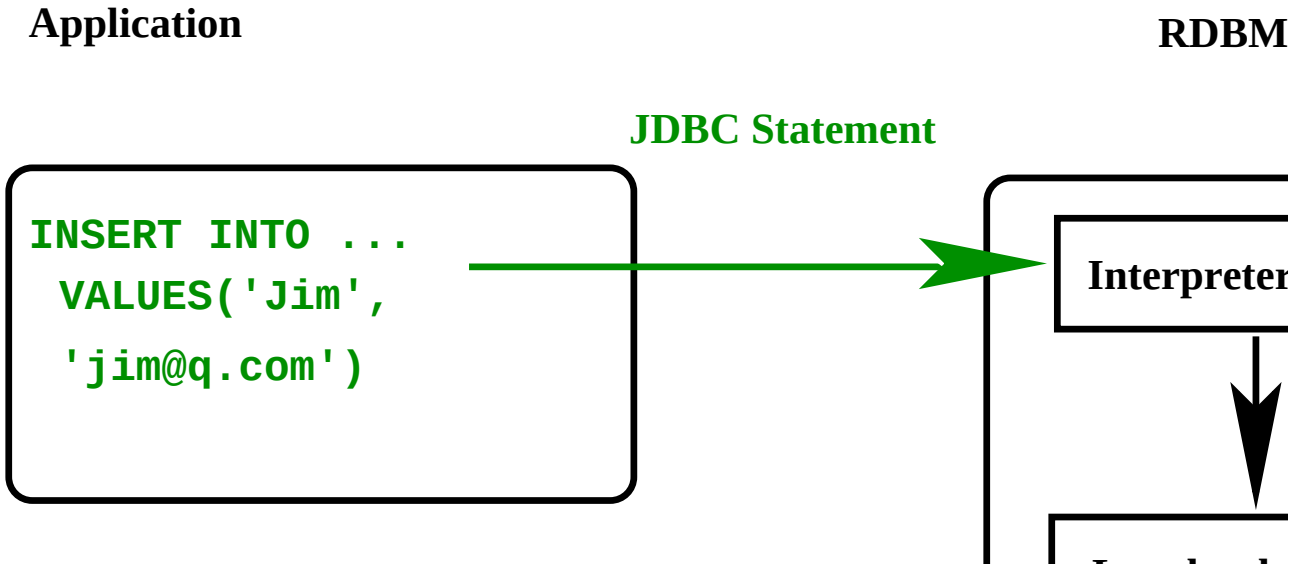
Validating user input prior to dynamically composing SQL statements.



Related exercises

Exercise 8: Using regular expressions in Java™
Exercise 9: Input validation by regular expressions

SQL statements in Java™ applications get parsed at the database server



Two questions

1. What happens when executing thousands of SQL statements having identical structure?

2. Is this architecture adequate with respect to security concerns?

Addressing performance

```
INSERT INTO Person VALUES ('Jim', 'jim@q.org')
INSERT INTO Person VALUES ('Eve', 'eve@y.org')
INSERT INTO Person VALUES ('Pete', 'p@rr.com')
...
```

Wasting time parsing SQL over and over again!

Addressing performance mitigation

```
INSERT INTO Person VALUES
  ('Jim', 'jim@q.org'),
  ('Eve', 'eve@y.org'),
  ('Pete', 'p@rr.com') ... ;
```

Dealing with large record counts even this option may become questionable.

Restating the SQL injection problem

The database server's interpreter may interpret an attacker's malicious code among with intended SQL.

- User input is being interpreted by the database server's interpreter.
- User input filtering my be incomplete / tedious.

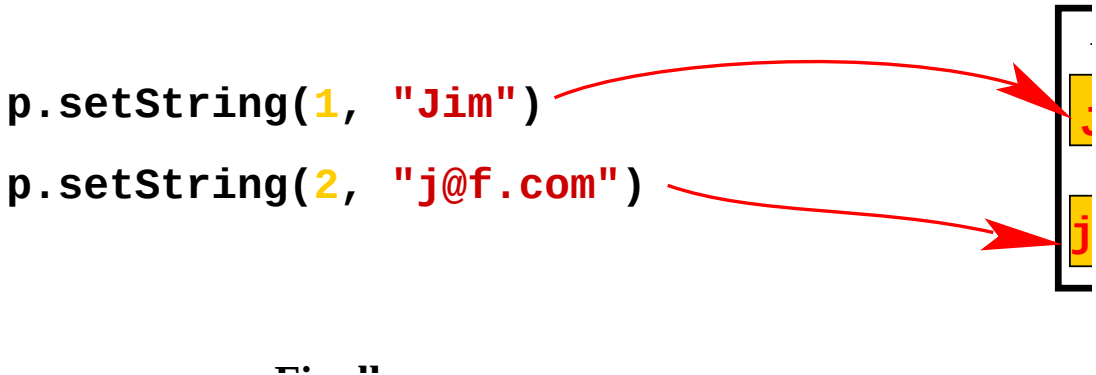
Solution: Use `java.sql.PreparedStatement`

- User input being excluded from parsing.
- Allows for reuse per record.

PreparedStatement principle.

```
INSERT INTO ...VALUES (?, ?)
```

PreparedStatement



Three phases using parameterized queries

1. `PreparedStatement` instance creation: Parsing SQL statement possibly containing place holders.

2. Set values of all placeholder values: SQL values are not being parsed.

3. Execute the statement.

Steps 2. and 3. may be repeated without re-parsing the underlying SQL statement thereby saving database server resources.

PreparedStatement example

```
final Connection conn = DriverManager.getConnection (...

final PreparedStatement pStmt = conn.prepareStatement(
  "INSERT INTO Person VALUES(?, ?)");❶

pStmt.setString(1, "Jim");❷
pStmt.setString(2, "jim@foo.org");❸

final int updateCount = pStmt.executeUpdate();❹

System.out.println("Successfully inserted " + updateCount + " dataset(s)");
```

Injection attempt example

```
Jim', 'jim@c.com');DROP TABLE Person;INSERT INTO Person VALUES('Joe
```

Attacker's injection text simply becomes part of the database server's content.

Problem solved!

Limitation: No dynamic table support!

- `SELECT birthday from Persons`

- ```
  PreparedSatatement statement =
    connection.prepareStatement("SELECT ? ❶ from ?" ❷);
  statement.setString(1, "birthday") ❸;
  statement.setString(2, "Persons") ❹;
  ResultSet rs = statement.executeQuery() ❺;
  ```

In a nutshell: **Only attribute value literals may be parameterized.**
  ```

Related exercises

Exercise 10: Prepared Statements to keep the barbarians at the gate

JDBC™ read and write

- `CREATE` / `UPDATE` / `DELETE`

  client modifies database server data:

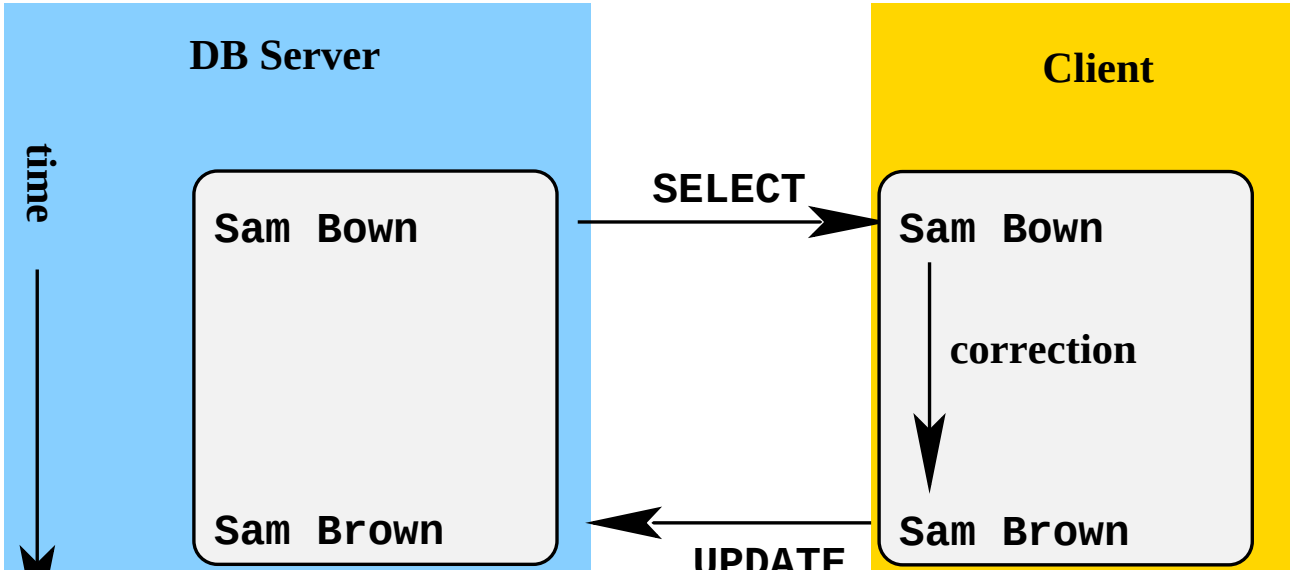
  ```
  int result = statement.executeUpdate("UPDATE Person ...");
  ```

- `SELECT`

  client receives copies of database server data:

  ```
  ResultSet result = statement.executeQuery("SELECT ... FROM Person ...");
  ```

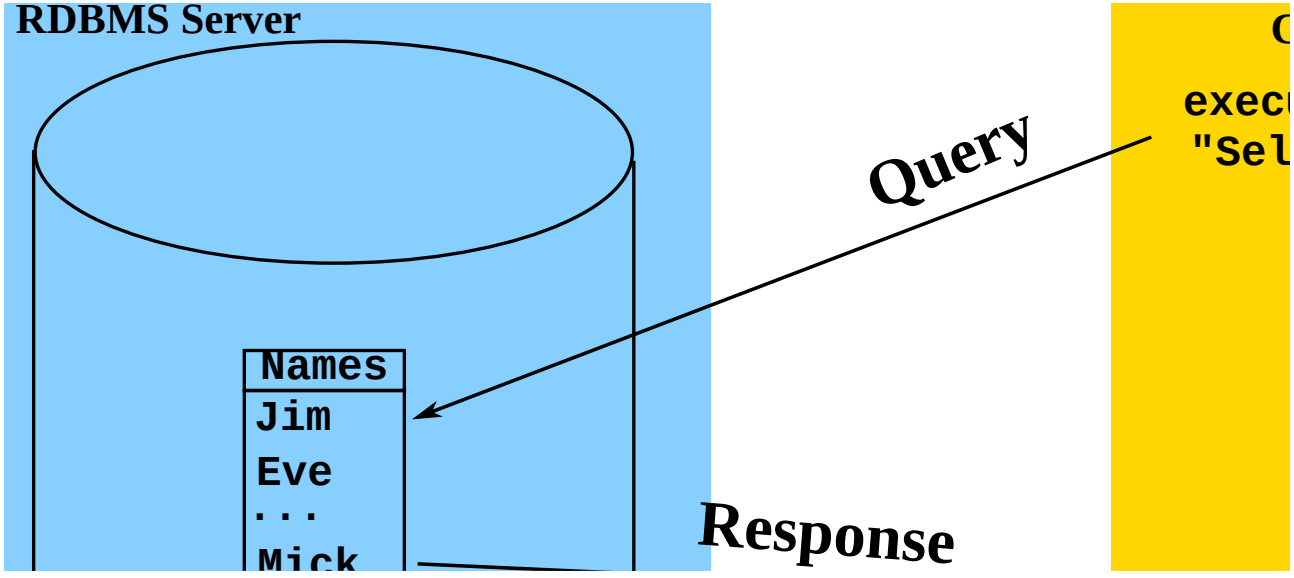
Server / client object's life cycle



JDBC™ record container

- No standard Collections container e.g. `java.util.List`.
- “Own” collection `java.sql.ResultSet` holding transient database object copies.

Reading data from a database server.



Names and birth dates of friends

```
CREATE TABLE Friends (
   id INTEGER NOT NULL PRIMARY KEY
  ,nickname char(10)
  ,birthdate DATE
);
```

```
INSERT INTO Friends VALUES
   (1, 'Jim', '1991-10-10')
  ,(2, 'Eve', '2003-05-24')
  ,(3, 'Mick','2001-12-30')
  ;
```

Accessing friend's database records

```
final Connection conn = DriverManager.getConnection (...);
final Statement stmt = conn.createStatement();
// Step 3: Creating the client side JDBC container holding our data records
final ResultSet data = stmt.executeQuery("SELECT * FROM Friends"); ❶

// Step 4: Dataset iteration
while (data.next()) { ❷
   System.out.println(data.getInt("id") ❸
       + ", " + data.getString("nickname") ❹
       + ", " + data.getString("birthdate")); ❺
}
```

Important `ResultSet` states

New: `resultSet = statement.executeQuery(...)`	Caution: Data not yet accessible!
Cursor positioned: `resultSet.next()` returning `true`	Data accessible until `resultSet.next()`returns `false`.
Closed: `resultSet.next()` returning `false`	Caution: Data not longer accessible!

JDBC™ to Java™ type conversions

JDBC™ Type	Java™ type
`CHAR`, `VARCHAR`, `LONGVARCHAR`	`String`
`NUMERIC`, `DECIMAL`	`java.math.BigDecimal`
`BIT`	`boolean`
`TINYINT`	`byte`
`...`	`...`

Shamelessly copied from JDBC Types Mapped to Java Types.

Java™ to JDBC™ type conversions

Java™ Type	JDBC™ type
`String`	`CHAR`, `VARCHAR`, `LONGVARCHAR`
`java.math.BigDecimal`	`NUMERIC`
`boolean`	`BIT`
`...`	`...`

Shamelessly copied from Java Types Mapped to JDBC Types.

Error prone type accessors!

```
int getInt(int columnIndex)
double getDouble(int columnIndex)
Date getDate(int columnIndex)
...
```

Polymorphic accessor

```
Object getObject(int columnIndex)
```

Best SQL to Java type match.

Access by column name

```
final int id =
  resultSet.getInt("id");
final String nickName =
  resultSet.getString("nickname");
final Date birthDate =
  resultSet.getDate("birthdate");
```

```
CREATE TABLE Friends (
   id INTEGER NOT NULL PRIMARY KEY
  ,nickname char(10)
  ,birthdate DATE
);
```

Caveat: May impact performance.

Access by column index

```
final int id =
  resultSet.getInt(1);
final String nickName =
  resultSet.getString(2);
final Date birthDate =
  resultSet.getDate(3);
```

```
CREATE TABLE Friends (
   id INTEGER NOT NULL PRIMARY KEY
  ,nickname char(10)
  ,birthdate DATE
);
```

Caveat: Error prone on schema evolution.

Related exercises

Exercise 11: Getter methods and type conversion

Problem: `null` value ambiguity

```
final int count = resultSet.getInt("numProducts");
```

Problem: Two possibilities in case of `count == 0`:

1. DB attribute numProducts is 0 (zero).
2. DB attribute numProducts is `null`.

Resolving `null` value ambiguity

```
final int count = resultSet.getInt("numProducts");

if (resultSet.wasNull()) {
...
} else {
...
}
```

See `wasNull()`.

Related exercises

Exercise 12: Handling NULL values.

Problem: Dynamic driver configuration

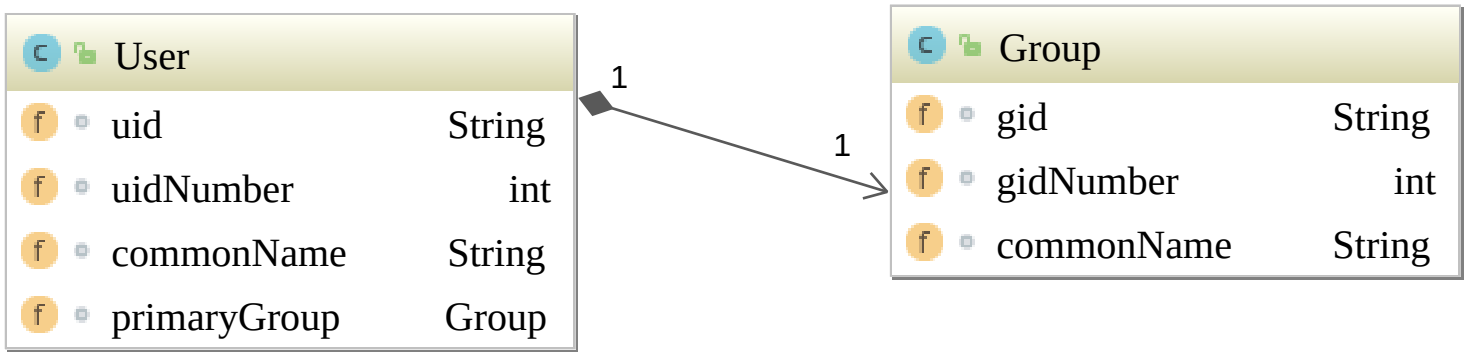
```
server=db.somedomain.org
port=3306
...
driver=mariadb-java-client-3.3.3.jar
```

- Driver file `mariadb-java-client-3.3.3.jar` shall be loaded at runtime.
- Cannot be packaged by manufacturer.
- Problem: Class loader and security

Shim driver (facade)

```
import java.sql.Driver;
  ...
public class DriverShim implements Driver {
    private Driver driver;
    DriverShim(Driver driver) {
        this.driver = driver;
    }
    @Override
    public Connection connect(String s, Properties properties) throws SQLException {
        return driver.connect(s, properties);
    }
    @Override
    public boolean acceptsURL(String u) throws SQLException {
        return driver.acceptsURL(u);
    }
...
}
```

Users and groups



Isolation level

- **Level 0**: Prevent other transactions from changing data that has already been modified by an uncommitted transaction.

  Other transactions can read uncommitted data resulting in “dirty reads”.

- **Level 1**: Prevents dirty reads. (Default on many RDBMS)
- **Level 2**: prevents non-repeatable reads.
- **Level 3**: Data read by one transaction is valid until the end of that transaction, preventing phantom rows.

JDBC™ Isolation level

- Transaction unsupported: Connection.TRANSACTION_NONE
- Level 0: Connection.TRANSACTION_READ_COMMITTED
- Level 1: Connection.TRANSACTION_READ_UNCOMMITTED
- Level 2: Connection.TRANSACTION_REPEATABLE_READ
- Level 2: Connection.TRANSACTION_SERIALIZABL

Setting the isolation level

- `connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);`
- See `Connection.TRANSACTION_READ_COMMITTED` and `setTransactionIsolation`.
- Note: Setting will become effective when starting next transaction.

Related exercises

Exercise 13: Isolation level 1 vs. 2
Exercise 14: JDBC™ and transactions
Exercise 15: Aborted transactions