# Applicazioni Web 2014/15

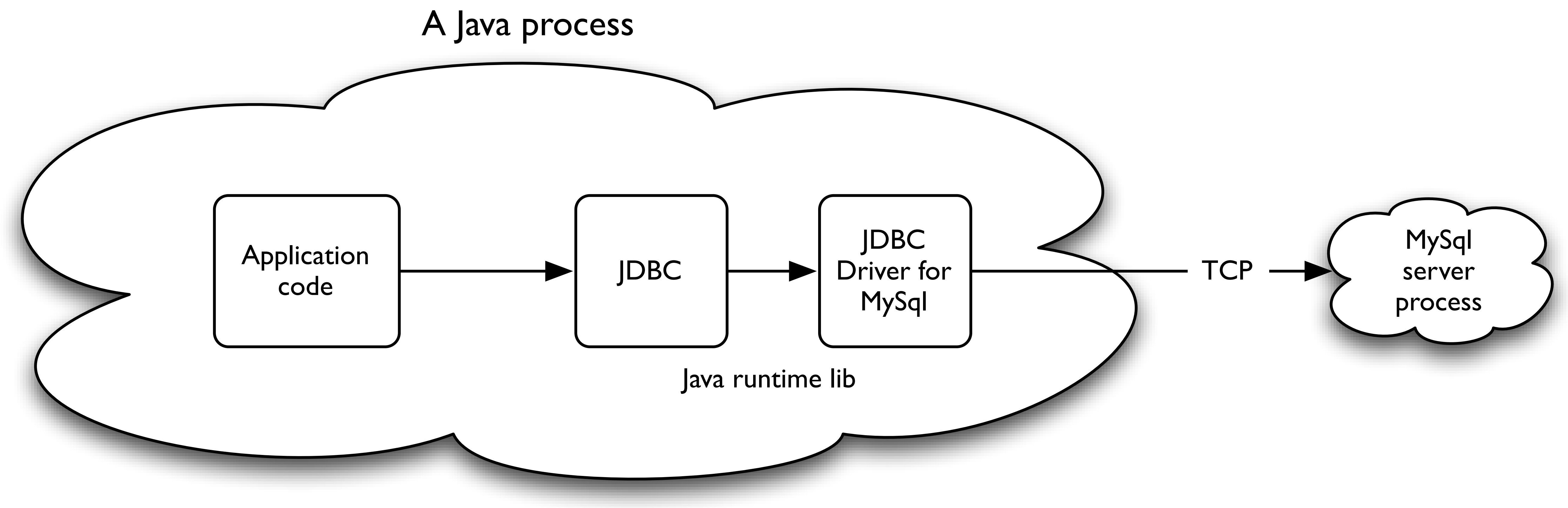
Lezione 10 - Persistenza

Matteo Vaccari
http://matteo.vaccari.name/
matteo.vaccari@uninsubria.it

## Perché usare un DB relazionale?

- Per l'accesso concorrente ai dati (e svincolare il codice applicativo dalla concorrenza)
- Per estrarre i dati in maniera veloce
- Per fare fronte a nuovi requisiti tramite una semplice riconfigurazione dello schema (cf. usare il filesystem)

# Java and JDBC



# Get a JDBC connection

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DatabaseConfiguration {

    // Should be loaded from external configuration
    final String USERNAME = "myuser";
    final String PASSWORD = "mypassword";
    final String URL = “jdbc:postgresql://localhost:5432/mydatabase";
    final String DRIVER = "org.postgresql.Driver";

    public Connection getConnection() throws ClassNotFoundException, SQLException {
        // load JDBC driver
        Class.forName(DRIVER);

        // create connection
        return DriverManager.getConnection(URL, USERNAME, PASSWORD);
    }
}
```

# Execute sql code

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

String sql = "INSERT INTO customers (name) VALUES('Mario Rossi')";

statement.executeUpdate(sql);
```

# Use a *prepared statement*

```
String sql = "INSERT INTO customers (name) VALUES (?)";
PreparedStatement statement = connection.prepareStatement(sql);
statement.setString(1, "pippo");
statement.executeUpdate();
```

## … and close the statement

```
String sql = "INSERT INTO customers (name) VALUES (?)";
PreparedStatement statement;
try {
    statement = connection.prepareStatement(sql);
    statement.setString(1, "pippo");
    statement.executeUpdate();
} finally {
    if (null != statement) {
        try {
            statement.close();
        } catch(Exception ignored) {}
    }
}
```

# ... with Java 1.7 syntax

```java
String sql = "INSERT INTO customers (name) VALUES (?)";
try (PreparedStatement statement = connection.prepareStatement(sql)) {
    statement.setString(1, "pippo");
    statement.executeUpdate();
}
```

Get rid of the "finally" block!

```java
String sql = "INSERT INTO customers (name) VAL
PreparedStatement statement;
try {
    statement = connection.prepareStatement(sq
    statement.setString(1, "pippo");
    statement.executeUpdate();
} finally {
    if (null != statement) {
        try {
            statement.close();
        } catch(Exception ignored) {}
    }
}
```

# Note

- *statement.finalize()* chiuderebbe lo statement, ma viene chiamato dal garbage collector non si sa quando
- Bisogna chiudere esplicitamente lo statement, altrimenti se abbiamo molte operazioni concorrenti alcune falliranno
- Bisogna ignorare le eccezioni in *statement.close()*, altrimenti *oscureranno* l'eventuale eccezione lanciata da *statement.executeUpdate()*

## Reading data from a table

```
Statement statement = connection.createStatement();
ResultSet resultSet = statement.executeQuery("SELECT * FROM customers");

while (resultSet.next()) {
    String s = resultSet.getString("name");
}
```

# ... and close all objects

```
Statement statement;
ResultSet resultSet;
try {
    statement = connection.createStatement();
    resultSet = statement.executeQuery("SELECT foo,bar FROM my_table");

    while (resultSet.next()) {
        String s = resultSet.getString("foo");
        // do something with s
    }
} finally {
    if (null != resultSet) {
        try {
            resultSet.close();
        } catch(Exception ignored) {}
    }
    if (null != statement) {
        try {
            statement.close();
        } catch(Exception ignored) {}
    }
}
```

## … with Java 1.7 syntax

```
try (
    Statement statement = connection.createStatement();
    ResultSet resultSet = statement.executeQuery("SELECT foo FROM my_table");
    )
{
   while (resultSet.next()) {
      String s = resultSet.getString("foo");
      // do something with s
   }
}
```

## *Usare uno script per generare il database*

- Crea ***due database***, uno per unit test e uno per sviluppo
- Però prima li cancella se esistono
- Carica lo schema dei dati
- Crea un ***utente applicativo*** e gli dà i diritti *minimi* che gli servono

## *Usare uno script per generare il database, perchè?*

- Bisogna sempre automatizzare tutto
- Mette i colleghi in grado di partire velocemente
- Cristallizza le informazioni necessarie per installare l'applicazione
- Se ho lo script, modificare lo schema costa poco

```bash
#!/bin/bash
#
# Purpose: create all needed databases for the application,
# loading the schema and the test data

# define key information
project=????
dbpassword="secret"

# no customization needed beyond this line
db_development=${project}_development
db_test=${project}_test
dbuser=$project

# Stop at the first error
set -e

# Go to the main project directory
cd "$(dirname $0)/.."

# if we're on Linux
if uname -a | grep -qi linux; then
  # if the postgres user for the current login does not exist
  if ! psql -tAc "select 3 + 4" template1 > /dev/null 2> /dev/null; then
    # then create the postgres user with superuser privileges
    sudo -u postgres createuser --superuser $(whoami)
  fi
fi

# create user
createuser --no-superuser --createdb --no-createrole $dbuser || true
```

```
# create databases
for db in $db_development $db_test; do
    echo doing $db

    dropdb --if-exists $db
    createdb $db
    psql -tAc "ALTER USER $dbuser WITH PASSWORD '$dbpassword'" $db

    # load all sql scripts in database
    cat src/main/sql/???_*.sql src/main/sql/seed.sql | psql $db

    # grant all privileges on all tables to our user
    for table in $(psql -tAc "select relname from pg_stat_user_tables" $db); do
      psql -tAc "GRANT ALL PRIVILEGES ON TABLE $table TO $dbuser " $db
    done
done

echo "OK"
```

## Astrarre il database

# Una semplice interfaccia al DB

```
public interface Database {
    void execute(String sql, Object ... params);

    Map<String, Object> selectOneRow(String sql, Object ... params);

    List<Map<String, Object>> selectMultipleRows(String sql, Object ... params);
}


database.execute(
  “UPDATE users SET email = ? WHERE user_id = ?”, “foo@bar.com” , 1234);
```

# L'implementazione del "database" astratto

```java
Database database = new Database(new DatabaseConfiguration(
"postgres://myproject:secret@localhost:5432/myproject_test"));

@Test
public void selectsOneRow() throws Exception {
   List<DatabaseRow> rows = database.select("select 2+2");
   assertEquals(1, rows.size());
   assertEquals(new Long(4), rows.get(0).getLong(0));
}

@Test
public void selectsMoreRows() throws Exception {
   List<DatabaseRow> rows = database.select("(select 2) union (select 3)");
   assertEquals(2, rows.size());
   assertEquals("2", rows.get(0).getString(0));
   assertEquals("3", rows.get(1).getString(0));
}
```

# Il metodo *execute*

```java
public void execute(String sql, Object... params) {
    try (
            Connection connection = getConnection();
            PreparedStatement statement = prepareStatement(sql, connection, params);
        )
    {
        statement.execute();
        connection.commit();
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}

private PreparedStatement prepareStatement(String sql, Connection connection, Object... params) {
    PreparedStatement statement = connection.prepareStatement(sql);
    for (int i = 0; i < params.length; i++) {
        statement.setObject(i + 1, params[i]);
    }
    return statement;
}


// example
database.execute("update people set name = ? where city = ?", "pippo", "topolinia");
```

# Astrarre la persistenza

The *Repository* pattern:
"*A mechanism for encapsulating*

1. *storage,*
2. *retrieval, and*
3. *search*

*which emulates a collection of objects*"

-- Eric Evans, *Domain Driven Design*

# Example *repository*

```
public interface PictureRepository {
    // storage
    void add(Picture picture);
    void update(Picture picture);

    // retrieval
    Picture findOne(Object pictureId);
    List<Picture> findAll();

    // search
    List<Picture> findAllByAuthor(String authorName);
    List<Picture> findAllByYearRange(int startYear, int endYear);
}

public class DatabasePictureRepository implements PicturesRepository {
    public DatabasePictureRepository(Database database) {...}
}
```