# Applicazioni Web 2013/14

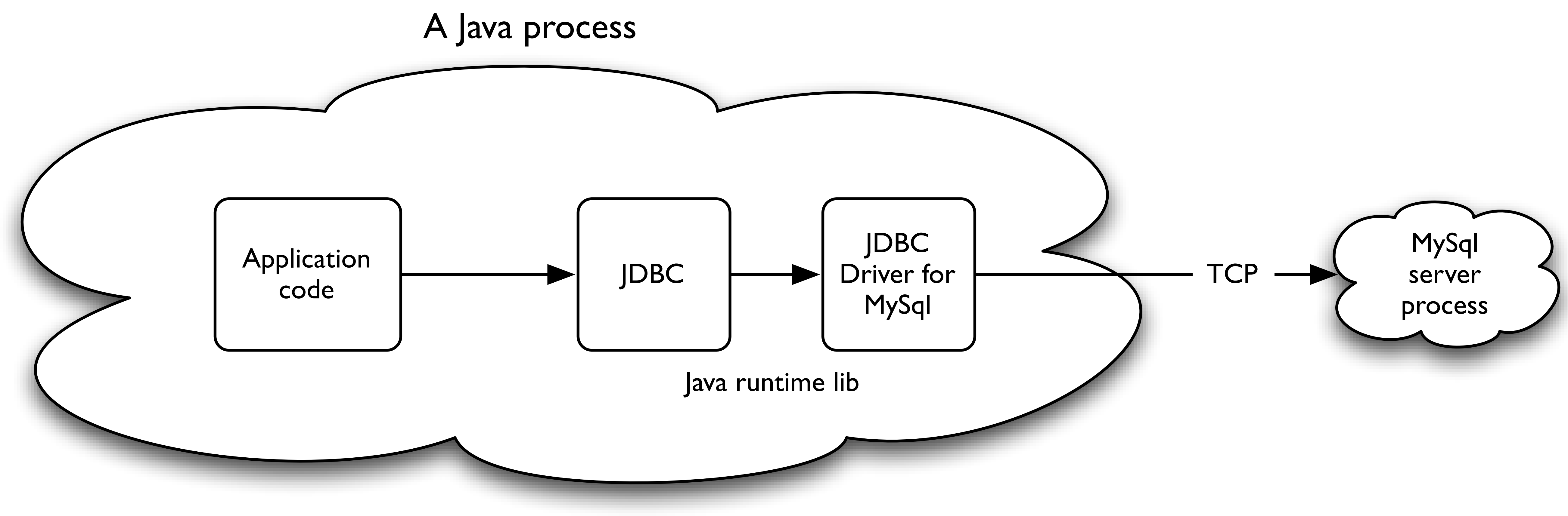
Lezione 9 - Persistenza

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 DatabaseConnector {

    // Should be loaded from external configuration
    final String USERNAME = "myuser";
    final String PASSWORD = "mypassword";
    final String URL = "jdbc:mysql://localhost/mydatabase";
    final String DRIVER = "com.mysql.jdbc.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*

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

# … and close the statement

```
PreparedStatement statement;
try {
    String sql = "INSERT INTO customers (name) VALUES (?)";
    statement = connection.prepareStatement(sql);
    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 objects

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

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

## *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

## *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

```
# define key information
src=sql                       # sql sources directory
dbname=tai_chat               # name of development db
dbname_test=tai_chat_test # name of test db
dbuser=tai_chat               # name of application user
dbpassword=tai_chat           # password of application user

# ask mysql root password
read -s -p "mysql root password? (type return for no password) " MYSQL_ROOT_PASSWORD
if [ "$MYSQL_ROOT_PASSWORD" != "" ]; then
    MYSQL_ROOT_PASSWORD=-p$MYSQL_ROOT_PASSWORD
fi

# drop and create databases
mysqladmin -uroot $MYSQL_ROOT_PASSWORD drop $dbname
mysqladmin -uroot $MYSQL_ROOT_PASSWORD --force drop $dbname_test
mysqladmin -uroot $MYSQL_ROOT_PASSWORD create $dbname
mysqladmin -uroot $MYSQL_ROOT_PASSWORD create $dbname_test

# create application user and give grants
echo "grant all on $dbname.* to '$dbuser'@localhost identified by '$dbpassword';" \
     | mysql -uroot $MYSQL_ROOT_PASSWORD $dbname
echo "grant all on $dbname_test.* to '$dbuser'@localhost identified by '$dbpassword';" \
      | mysql -uroot $MYSQL_ROOT_PASSWORD $dbname_test

# load schema
cat $src/???_*.sql | mysql -u$dbuser -p$dbpassword $dbname
cat $src/???_*.sql | mysql -u$dbuser -p$dbpassword $dbname_test
```

## 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

```
Database database = new MysqlDatabase(
   "localhost", "blog_test", "blog_user", "password");

@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*

```
public void execute(String sql, Object ... params) {
    Connection connection = getConnection();
    PreparedStatement statement = null;
    try {
        statement = connection.prepareStatement(sql);
        for (int i=0; i<params.length; i++) {
            statement.setObject(i+1, params[i]);
        }
        statement.executeUpdate();
    } catch (SQLException e) {
        throw new RuntimeException(e);
    } finally {
        safelyClose(statement);
    }
}

// example
String sql = "insert into my_table (foo, bar) values (?, ?)";
executor.execute(sql, "a string", 123);
```

```java
private void safelyClose(Statement statement) {
    if (null != statement) {
        try {
            statement.close();
        } catch (SQLException ignored) {}
    }
}
```

# 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) {...}
}
```