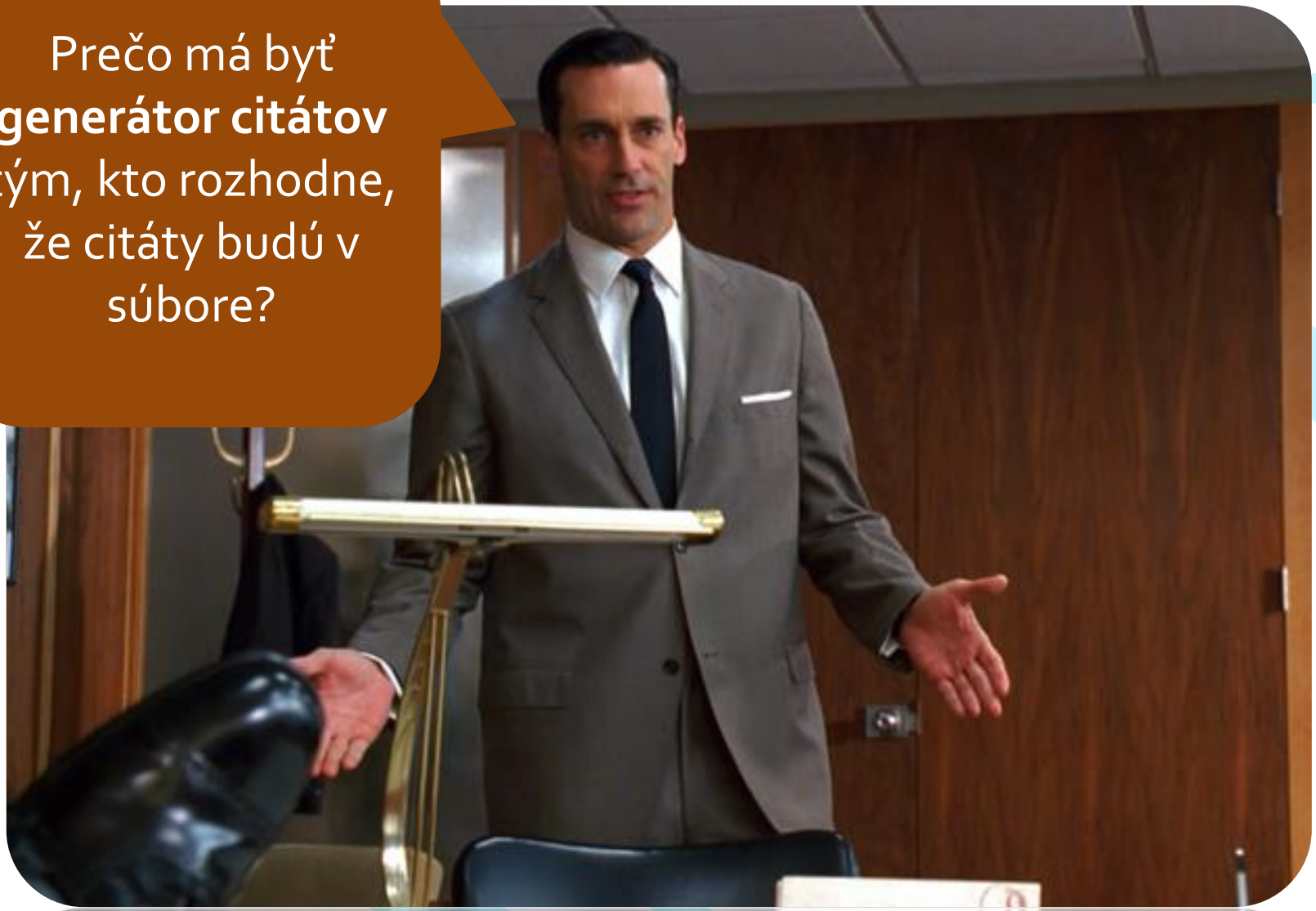
# 5. Továrne, singletony a databázy

14. 10. 2013

ÚINF/PAZ1c

(Róbert Novotný)

Prečo má byť **generátor citátov** tým, kto rozhodne, že citáty budú v súbore?



# Trieda zdrôtovaná so závislosťou

- generátor citátov **potrebuje** DAO
- trieda si **sama** nájde a vytvorí svojho **kolaborátora / závislosť**

```
public class GeneratorCitatov
{
  private CitatDao citatDao
    = new SuborovyCitatDao();
}
```

Ale prečo?

Prečo sa má generátor citátov rozhodnúť, že chce citáty práve **zo súboru???**

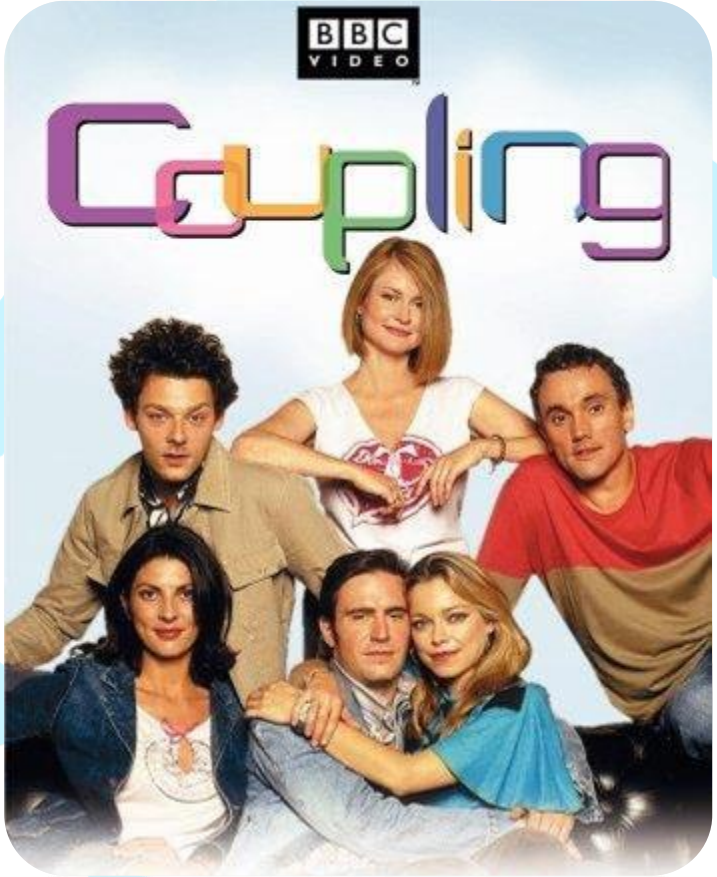
Čo ak chceme striedať implementácie?

V produkcii z databázy

V testoch z pamäte

## Prečo sa má generátor citátov rozhodnúť, že chce citáty práve **zo súboru???**

- **tight coupling**
- úzka zviazanosť tried
- priveľká vzájomná závislosť
- krehké pri zmenách



# Princípy rôznych svetov

- **Slovenský princíp:**
  - staviam dom
  - buď si betón namiešam **sám**
  - alebo poprosím niekoho, nech mi ho **vyrobí**
- **Hollywoodsky princíp:**
  - „Nevolajte nám, my sa vám ozveme.“

Trieda si kolaborátorov nevyrába sama.
Poprosí niekoho, nech ich vyrobí a dodá.

# Továreň :: Factory

Objekt, ktorý vytvára iné objekty.

- návrhový vzor!
- ak potrebujeme objekt, ktorého vytváranie je komplexný proces…
- …získame ho z **továrne**

# Továreň :: Factory

```
public class CitatDaoFactory {
  public CitatDao dajCitatDao() {
    /* tu sa vyrobí inštancia */
  }
}
```

- metóda vráti **novú inštanciu** objektu
- v jej vnútri sa rozhodne
  - akého typu bude vytváraný objekt
  - akým spôsobom sa vytvorí

# Továreň :: Factory

```
public class CitatDaoFactory {
  public CitatDao dajCitatDao() {
    /* tu sa vyrobí inštancia */
  }
}
```

- továreň môže vyrobiť **vždy novú inštanciu**
- alebo recyklovať len jednu dokola
  - návrhový vzor **singleton**
- alebo mať **pool** objektov, ktoré znovupoužíva

# Továreň :: Factory

```
public class CitatDaoFactory {
  public CitatDao dajCitatDao() {
    /* tu sa vyrobí inštancia */
  }
}
```

- sama továreň sa rozhodne, či vráti
  - **súborový** DAO
  - **databázový** DAO
  - alebo **iný** DAO implementujúci interfejs

# Ako použiť továreň v kóde?

Milá továreň,
som trieda, ktorá generuje náhodné citáty.
Daj mi, prosím, nejakú inštanciu CitatDao.

Ako však trieda uvidí továreň?

# Trieda potrebuje továreň

- ako ju získa?
- podľa pravidla „čo je závislosť, to je inštančná premenná”:

```
public class GeneratorCitatov
{
  private CitatDaoFactory citatDaoFactory
    = new CitatDaoFactory();
}
```

Ufff, ufff…

# Ako uvidí trieda továreň?

- problém so závislosťou triedy na inej triede sme vylepšili **továrňou**…

*Každý problém v informatike možno vyriešiť pridaním ďalšej úrovne nepriameho prístupu / abstrakcie.*
*-- Butler Lampson, 1972*

- ale máme ten istý problém v bledomodrom!

# Ako uvidí trieda továreň?

Vyrobíme továreň na továreň!

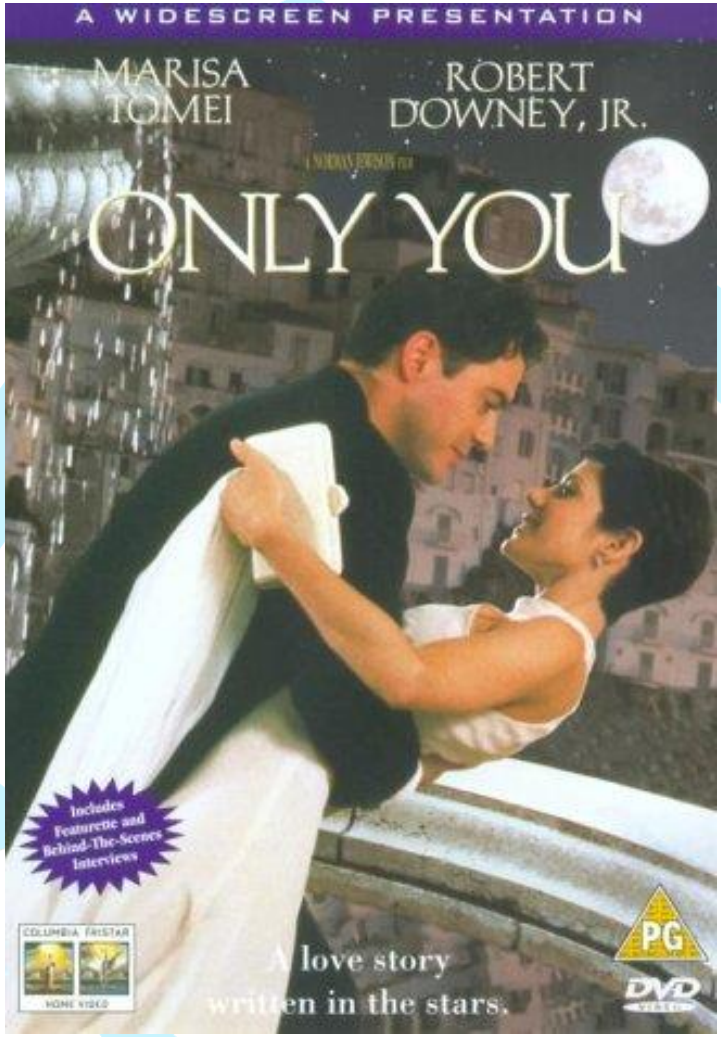
Koľko inštancií továrne naozaj potrebujeme?

Stačí jedna!

# Ako dosiahnuť jedinú inštanciu?

- návrhový vzor **Singleton**!
- zaistí, že v systéme bude len jediná inštancia danej triedy

návrhový vzor = design pattern = recept starej matere



# Singleton v Jave

```
public enum CitatDaoFactory {
  INSTANCE;

  public CitatDao getCitatDao() {
     /*...*/
  }
}
```

```
CitatDao citatDao
      = CitatDaoFactory.INSTANCE.getCitatDao();
```

# Enumy v Jave

- pre premenné s hodnotou z pevne danej sady

```
public enum Smer {
      HORE, DOLE, VLAVO, VPRAVO;
}
```

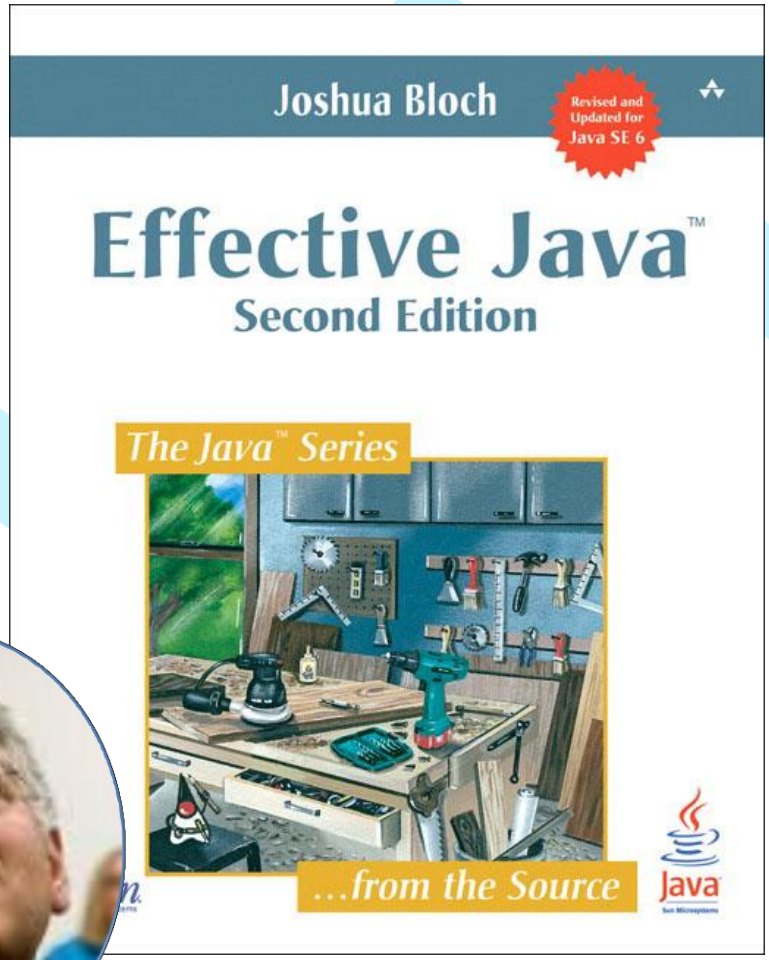
```
Smer smer = Smer.HORE
```

```
figurka.posun(5, Smer.VPRAVO)
```

- vhodné namiesto konštantných Stringov
  - nemožno zadať hodnotu mimo rozsahu

# Enumy pre singletony

- vytvoríme **enum** s jedinou hodnotou (**INSTANCE**)
- krátky zápis
- vláknovo bezpečné
- odsúhlasené autoritou



# Singleton v Jave

- použitie **enum**-u je ťažký syntaktický hack Javy
- ale zaistí, že trieda **CitatDaoFactory** bude naozaj **len jedna**
- **INSTANCE** = verejná inštančná premenná

Pojednanie o korektnosti iných prístupov je zábava pre hardcore programátorov...

# Migrácia rýchlo a ľahko

- z pamäťovej databázy na súbor?

Zmením **JEDEN** riadok!

```
public enum CitatDaoFactory {
  INSTANCE;

  public CitatDao dajCitatDao() {
     return new PamatovyCitatDao();
     return new SuborovyCitatDao();
  }
}
```

Celý projekt sa magicky zmení

# Výhody továrne?

- továreň môže vracať iné objekty **v produkcii** a iné **v testoch**
  - **testy**: testujeme oproti vymyslenej databáze, alebo voči súborom
  - **produkcia**: oproti distribuovanej relačnej databáze
- môže vracať buď **tú istú inštanciu** alebo rozličné objekty v závislosti od **potreby**

# RELAČNÉ DATABÁZY SKRZ SPRING

# Relačné databázy

- **dominantný** spôsob ukladania dát
  - finančné, osobné, ...
- bohatý **výskum**
  - relačná algebra
- tona **implementácií**
  - komerčné: Oracle, DB2
  - open-source: MySQL, PostGreSQL

# JDBC = Java API pre SQL

# Problémy s JDBC

- funkčné, ale ťažkopádne
- mnoho opakujúceho sa kódu
- nutnosť korektne obsluhovať a uvoľňovať prostriedky
  - v tutoriáloch sa v mene jednoduchosti ignorujú závažné chyby

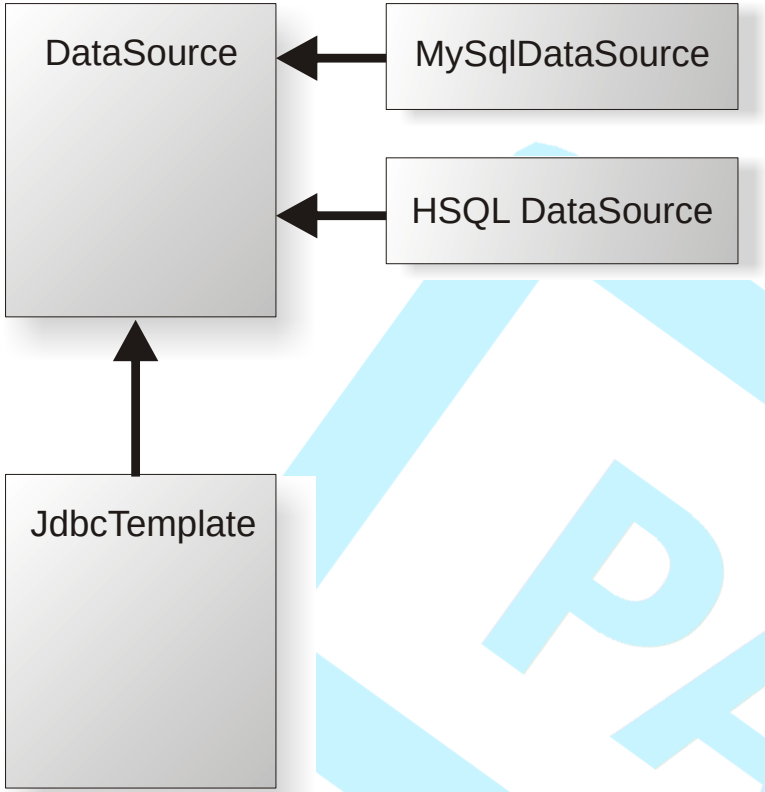
Takmer každý rozsiahlejší projekt si vytvorí vlastnú nadstavbu nad JDBC.

# Spring Framework

„Budujme lepšie enterprise aplikácie“

- adresuje mnohoročné skúsenosti z vývoja veľkých aplikácií
  - webaplikácie, transakcie, messaging…
- vznikal **zdola**
  - od roku 2001, prichádza 4.0

# Spring JDBC: základné triedy




- **javax.sql.DataSource**
  - nízkoúrovňová komunikácia s databázou
- **JdbcTemplate**
  - spúšťanie SQL dopytov
  - mapovanie výsledkov na Java objekty

```
public enum JdbcTemplateFactory {
  INSTANCE;
  private JdbcTemplate jdbcTemplate;
  private JdbcTemplateFactory() {
    MysqlDataSource dataSource = new MysqlDataSource();
    dataSource.setDatabaseName("hydra_test");
    dataSource.setUser("root");
    dataSource.setPassword("hamoomah");
    jdbcTemplate = new JdbcTemplate(dataSource);
  }
  public JdbcTemplate getJdbcTemplate() {
    return jdbcTemplate;
  }
}
```

implementácia z MySQL

# JdbcTemplate: získanie celého čísla

```
int rows =
  jdbcTemplate.queryForInt("SELECT COUNT(*) FROM citat");
```

- z dátového zdroja sa získa pripojenie **Connection**
- vytvorí sa dopyt **Statement**
- vykonaním dopytu sa získa výsledok **ResultSet**
- iterovaním cez **ResultSet** sa spracuje výsledok
- uvoľnia sa prostriedky
  - zatvorí sa **ResultSet**, **Statement** a **Connection**
- ošetria sa výnimky **SQLException**

# JdbcTemplate: pridávanie / aktualizácia

```
String sql
    = "UPDATE citat SET text = ?, autor = ? "
    + "WHERE id = ?"

int rows = jdbcTemplate.update(sql,
                citat.getText(),
                citat.getAutor(),
                citat.getId());
```

Za parametre v ? sa dosadia hodnoty

# JdbcTemplate: mapovanie riadkov na objekty

- priama podpora mapovania objektov podľa mennej konvencie

```
class Citat {
  Long id
  String autor;
  String text;
}
```

Java

```
CREATE TABLE citat
  id INTEGER NOT NULL,
  autor VARCHAR(64);
  text VARCHAR(255);
)
```

SQL

# JdbcTemplate: mapovanie riadkov na objekty

```
BeanPropertyRowMapper<Citat> mapper =
  BanPropertyRowMapper.newInstance(Citat.class)
```

- každý riadok z výsledku dopytu automaticky prevedie na objekt
  - podľa mennej konvencie
  - názvy stĺpcov mapované na inštančné premenné
  - automatická konverzia štandardných dátových typov
- inštanciu použijeme pri **query()**

# JdbcTemplate: mapovanie riadkov na objekty

- ak neladíme s konvenciou, mapujme po svojom

```
public class BeerMapper implements RowMapper<Citat> {
  public Citat mapRow(ResultSet rs, int rowNum)
    throws SQLException
    {
      Citat citat = new Citat();
      citat.setId(rs.getInt("id"));
      citat.setText(rs.getString("text"));
      citat.setAutor(rs.getString("autor"));
      return citat;
    }
  }
}
```

# Dopytovanie s mapovaním

```
List<Citat> citaty = jdbcTemplate.query(
                                        "SELECT * FROM citat",
                                        mapper);
```

- pre každý riadok z výslednej tabuľky sa zavolá metóda **mapRow()**
  - mapovač vráti inštanciu citátu
- metóda **query()** vyzbiera všetky inštancie a vráti ich ako zoznam

# SimpleJdbcInsert

- elegantné vkladanie objektov cez INSERT
- automaticky sa vybuduje INSERT do tabuľky
- stĺpce sa prevezmú z **params**
- možno ich však prispôsobiť cez **withColumnNames()**

```
SimpleJdbcInsert insert
  = new SimpleJdbcInsert(dataSource);
insert.setTableName("beer");
//...
SqlParameterSource params
  = new BeanPropertySqlParameterSource(beer);
insert.execute(params);
```

# SimpleJdbcInsert

- v prípade INSERTov často vraciame vygenerovaný kľúč
- ak máme autogenerované primárne kľúče
- metóda **executeAndReturnKey()**

```
java.lang.Number key = insert.executeAndReturnKey(params);
```

- z **Number** si vieme vytiahnuť číselnú hodnotu podľa príslušného typu

# Sumár

- veľmi jednoducho vieme pokryť **základné úlohy** s JDBC
- stačia na to **dve** triedy
- vieme vybudovať ďalšiu implementáciu DAO
- a zmenou v továrni zmigrovať projekt

# Otázky?