JPA
Java Persistence API

Petr Aubrecht
CA

- Opravdovy programator zpracovava seznamy ve FORTRANu.
- Opravdovy programator pracuje s texty ve FORTRANu.
- Opravdovy programator resi zpracovani hromadnych dat (kdyz uz to dela) ve FORTRANu.
- Opravdovy programator resi umelou inteligenci ve FORTRANu.

OPRAVDOVI PROGRAMATORI NEUZIVAJI PASCAL,
Ed Post, (C) 1983

Agenda

- Persistent storage
- JDBC
- ORM
- Hibernate
- JPA
  - Basics, primary keys, N:1, M:N relations
  - Transactions
  - Query Language
- IDE support (NetBeans)

Jak ukládat data? (1)

- V Javě máme objekty, které bychom rádi uložili trvale. Jak?
  - serializace (bytestream, XML)
  - objektová databáze – znáte nějakou?
  - XML databáze
    - moderní v poslední době
    - kromě XML fanatiků je nikdo nebere vážně, výkon na úrovni parsování textového souboru, vyhledávání pomocí XPath, funguje pouze fulltext search (pro strukturované dokumenty)
  - ...

Jak ukládat data? (2)

- relační databáze
  - V podstatě jediné běžně používané řešení (prověřené, výkonné vyhledávání, bezpečné/transakce atd.)
  - přímé používání JDBC
  - ORM – object-relational mapping

Serializace

- triviálně jednoduché uložení a načtení
  - public class Abc implements Serializable

```
FileOutputStreamfos = new FileOutputStream(”out.tmp”)
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeInt(12345);
oos.writeObject(”Today”);
oos.writeObject(new Date());
oos.close();
```

Skutecne velmi dobry programator umi najit chybu i v 6MB vypisu operacni pameti, a to aniz by pouzil hexadecimalni kalkulator.

Serializace

- triviálně jednoduché uložení a načtení
- nelze se dotazovat, načítat pouze některé instance
- nevhodné pro dlouhodobé ukládání (problém s novou verzí objektů)
- použití
  - krátkodobé uložení
  - komunikace po síti, objektové zprávy (Java RMI, CORBA, JMS)
  - (un)marshaling, změna objektů kvůli rozdílům v architekturách (malý a velký indián)

Objektové databáze

- objektové databáze
  - proč: problém rozdělení složitých objektů (normalizace, dekompozice),
  - v 90. letech s nástupem OO jazyků (C++) se hovořilo o konci relačních databází
- dnešní stav
  - výkon daleko za relačními databázemi
  - neexistence všeobecně přijímaného modelu, dotazovacího jazyka, transakcí, atd.
  - dnes se prakticky nepoužívají ve prospěch relačních DB

JDBC

- Klasika, prerekvizita předmětu (otázky?)
- Co může být na následujícím kódu za problém? Např. v kódu webové stránky:

```
Class.forName("org.postgresql.Driver");
Connection conn = DriverManager
    .getConnection("jdbc:postgresql:mis0715",
    "mis0715","pwd");
Statement st = conn.createStatement();
ResultSet rs = st.executeQuery("SELECT * FROM doc");
while(rs.next()) {
    System.out.println(rs.getString("docid"));
}
conn.close();
```

Co všechno je potřeba pro práci s jedinou entitou?

- definovat třídu
- vytvořit tabulku CREATE TABLE doc (docid serial primary key...)
- CRUD
  - Create
  - Retrieve (all, by ID, related)
  - Update
  - Delete

JDBC – Create

- potřeba vyřešit primární klíče (MySQL neumí sequence)
- rozhodně používat prepared statement!
- SQL injection je strašák!

```
ResultSet idRs = conn.createStatement().executeQuery(
  "SELECT nextval('doc_docid_seq')");
idRs.next();
int newId = rdRs.getInt(1); setDocId(newId);
PreparedStatement st = conn.prepareStatement(
  "INSERT INTO doc (docid,created,type) VALUES (?,?,?)");
st.setInt(1,newId);
st.setDatetime(2,new java.sql.Date(System.currentTimeMillis());
st.setString(3,"prescr"); // or typeEdit.getText()
st.execute();
```

JDBC – Retrieve

- zřejmě žádný problém
- kromě vytažení podle klíče se často hledají objekty ve vztahu 1:N nebo M:N, úprava dotazu je triviální

```
PreparedStatement st = conn.prepareStatement(
  "SELECT * FROM doc WHERE docid=?");
rs.setInt(1,getDocId());
rs.executeQuery();
rs.next();
setNextTime(rs.getInt("nexttime"));
setType(rs.getString("type"));
```

JDBC - Update

- opět není problém

```
PreparedStatement st = conn.prepareStatement(
  "UPDATE doc SET nexttime=?,type=? WHERE docid=?");
rs.setInt(1,getNextTime());
rs.setInt(2,getType());
rs.setInt(3,getDocId());
rs.execute();
```

JDBC – Delete

- Tady pozor pouze při ladění, odstranění WHERE části u DELETE je poněkud drastické.

```
PreparedStatement st = conn.prepareStatement(
  "DELETE doc WHERE docid=?");
rs.setInt(1,getDocId());
rs.execute();
```

Kde je problém?

- Různé nuance v syntaxi
  - CREATE TABLE je tím známý
  - generování primárních klíčů pomocí sekvence nefunguje vždy (MySQL a jiné primitivní databáze)
  - občas odlišnosti i v syntaxi JOINu (Oracle), v zápisu transakcí ap.
- Kolikrát jsme museli řešit každý data member? (5). Typicky je v aplikaci několik set entit! Každou změnu je potřeba řešit na několika místech!

Řešení – ORM!

- Pro velké projekty je přímá práce s JDBC příliš náročná, nelze zaručit funkčnost při změně schématu (kromě změny class diagramu a tříd je potřeba změnit všechna místa, kde se s databází pracuje.
- Při používání JDBC programátoři začnou používat SQL všude (včetně prezentační vrstvy v JSP)
- Vzniká logický požadavek na automatické ukládání objektů do databáze: 1 objekt = 1 záznam v tabulce, třída = entita
- ...

Řešení – ORM! (2)

- potřeba vyřešit mapování relací 1:1, 1:N, N:1, M:N jako ukazatele a pole ukazatelů
- ruční implementace je příliš složitá, lze ji nechat na dostupné knihovně
- knihovny přistupují k properties bud přímo (přes data members), nebo pomocí set/get
- typicky existuje podpora ze strany IDE, případně nástroje dané knihovny pro automatické generování tříd/konfigurace

J2EE 1.4

- J2EE 1.4, EJB (Enterprise Java Beans) 2.1
  - velmi komplikované (jedna entita odpovídala několika třídám a XML souborům)
  - vývojář se nemusí starat o ukládání, plně v kompetenci kontejneru
  - obecné úložiště
  - řeší například distribuované transakce nad několika oddělenými zdroji dat
  - řeší load balancing (přesun EJB mezi servery v závislosti na zatížení)
  - nebudeme se jím zabývat pro extrémní složitost

Oblíbené knihovny

- Hibernate, iBatis
  - jednodušší a přímo navržené pro relační databáze
  - těší se velké oblibě v komunitě vývojářů
  - nejsou standardem, nicméně se jejich hlavní (úspěšné) nápady stávají standardem
    - validace
    - criteria

Hibernate

- Pracuje s POJO (Plain Old Java Object).
- Mapování je dáno XML souborem.
- Celková konfigurace je v hlavním XML, které popisuje připojení do databáze a seznam souborů s mapováním.

Hibernate konfigurace hibernate.cfg.xml

- konfigurace připojení do db
- dialect

```
<hibernate-configuration>
 <session-factory>
  <property name="connection.url">jdbc:postgresql://localhost/k4care</p
  <property name="connection.username">k4care</property>
  <property name="connection.driver_class">org.postgresql.Driver</prope
  <property name="dialect">org.hibernate.dialect.PostgreSQLDialect</pro
  <property name="connection.password">XXXXXXXXXXXX</property>
  <property name="transaction.factory_class">org.hibernate.transaction.
  <property name="current_session_context_class">thread</property>
  <!-- this will show us all sql statements -->
  <property name="hibernate.show_sql">true</property>

  <mapping resource="net/k4care/storage/model/
    hibernate/DocumentHibernate.hbm.xml" />
  <mapping resource="net/k4care/storage/model/
    hibernate/AdministrativeDataHibernate.hbm.xml" />
...
```

Hibernate – mapovací soubor *DocumentHibernate*.hbm.xml

```
<hibernate-mapping package="net.k4care.storage.model.hibernate">
  <class name="DocumentHibernate" table="document">
    <id name="documentId" column="documentid">
      <generator class="sequence">
        <param name="sequence">document_documentid_seq</param>
      </generator>
    </id>

    <property name="documentType" column="documenttype"/>
    <property name="created" column="created"/>
    <property name="validFrom" column="validfrom"/>
    <property name="validTo" column="validto"/>
    <property name="nextTimeToConcern" column="nexttimetoconcern"/>
    <property name="content" column="content"/>
...
```

Hibernate – kód

- Vlastní kód je pak jednoduchý

```
DocumentHibernate object = new DocumentHibernate();
session.save(object);

DocumentHibernate actor = (DocumentHibernate)
                            session.get(DocumentHibernate.class,id);

session.save(actor);

session.refresh(actor);

session.delete(actor);
```

Hibernate – závěr

- usnadní mapování, nicméně pořád ještě mnoho psaní
- řeší syntaktické rozdíly (pomocí dialectu)
- je potřeba dávat pozor na “perzistentní kontext”, tedy práci uvnitř session, mimo ni dojde při pokusu o načtení dat k chybě
- výborně funguje cache (obzvlášť, je-li server na druhé straně Země)
- bohužel neudržuje v relaci 1:N obě strany, je potřeba udělat refresh()
- tak slavný, že Sun musel odpovědět výrazně lepším standardem

Hibernate =>

Rika se, ze jeden opravdovy programator zorganizoval v lodi Voyager doplneni programu pro rozpoznani, a to do nekolika stovek nevyuzitych bytu v pameti. Doplnek programu nalezl, urcil umisteni a vyfotografoval novy mesic Jupitera.

=> JPA

JPA

- JPA (Java Persistence API), JSR 220
  - důraz je kladen na jednoduchost (POJO!)
  - využití Annotation (od Javy 5)
  - smysluplné defaultní hodnoty (anotují se pouze odchylky, *definition by exception*)
  - vymyšlen původně pro EJB 3.0
  - pouze pro relační databáze
  - při zjednodušování EJB 2.1 → EJB 3.0 se JPA osamostatnilo jako knihovna, takže nyní lze použít i v J2SE
  - http://jcp.org/aboutJava/communityprocess/final/jsr220/

JPA konfigurace

- v J2SE META-INF/persistence.xml, u JEE v konf. serveru
- komu je to povědomé?

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0" xmlns=... xsi:schemaLocation=...>
  <persistence-unit name="misdemoPU" transaction-type="RESOURCE_LOCAL">
    <provider>oracle.toplink.essentials.PersistenceProvider</provider>
    <class>desktopapplication1.model.Faktura</class>
    <class>desktopapplication1.model.Zakaznik</class>
    ...
    <class>desktopapplication1.model.Sms</class>
    <properties>
      <property name="toplink.jdbc.user" value="misdemo"/>
      <property name="toplink.jdbc.password" value="data"/>
      <property name="toplink.jdbc.url" value="jdbc:postgresql://localhos
      <property name="toplink.jdbc.driver" value="org.postgresql.Driver"/
    </properties>
  </persistence-unit>
</persistence>
```

Entity manager

- stará se o připojení k databázi (vlastní Connection, ale vydá ji jen pod nátlakem)
- načte konfiguraci z XML (všimněte si, že můžeme mít několik konfigurací najednou a vybereme si pomocí persistence-unit

```
javax.persistence.EntityManager em;

em = javax.persistence.
  Persistence.createEntityManagerFactory("misdemoPU")
    .createEntityManager();
```

Entity manager – funkcionalita

- **find**(Class<T> entityClass, Object primaryKey)
  - najdi objekt podle primárního klíče
- **flush**() – úklid, všechno do databáze
- **merge**(T entity)
  - připoj objekt k databázi (k perzistentnímu kontextu)
- **persist**(Object entity) – uložit objekt
- **refresh**(Object entity) – znovu načíst objekt
- **remove**(Object entity) – delete
- **create*Query**(String sql,...) – položení dotazu do db

JPA minimalistický příklad

- @Entity a @Id jsou jediné dvě povinné anotace:

```
@Entity
public class Document implements Serializable {
    @Id
    private Integer documentid;

    private Date created;
    private Date validfrom;
    private Date validto;
    private Date nexttimetoconcern;
    private String content;
...
```

JPA praktický příklad

- primární klíče necháme generovat databází (bezpečná cesta), nejlépe sekvencí

```
@Entity
public class Document implements Serializable {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer documentid;

    private Date created;
    private Date validfrom;
    private Date validto;
    private Date nexttimetoconcern;
    private String content;
...
```

Požadavky na entitu

- Třída musí mít default konstruktor,
  - nesmí být final,
  - ani vnitřní nebo enum.
- Důvod je jasný – kontejner udělá proxy pro entitu (zdědí ji a nastaví listenery na gettery a settery)
- Entita MUSÍ mít primární klíč (@Id)

JPA kód

- CRUD už jednodušší být nemůže:

```
C doc = new Document(); em.persist(doc);

R doc = em.find(Document.class, 12345); //12345 je prim. klíč.

U em.persist(doc);

D em.remove(doc);
```

Nepovinné anotace

```
@Entity
@Table(name = "STUDENT")
@NamedQueries({@NamedQuery(name = "Student.findByRocnik",
query = "SELECT s FROM Student s WHERE s.rocnik = :rocnik"),...})
public class Student implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @Column(name = "STUDKEY", nullable = false)
    private String studkey;
    @Column(name = "JMENO")
    private String jmeno;
    @Column(name = "PRIJMENI")
    private String prijmeni;
    @Column(name = "ROCNIK")
    private Integer rocnik;
    @Column(name = "STIPENDIUM")
    private Float stipendium;
    @Column(name = "NAROZENI")
    @Temporal(TemporalType.TIMESTAMP)
    private Date narozeni;
```

Různé zajímavé anotace

- **@Transient**
  - data members, které nejsou persistované
- **@GeneratedValue(strategy=**
  - **Sequence**: použití sekvencí (kromě MySQL), další parametr **generator**=”jméno_sequence”, bohužel Toplink Essential ho ignoruje
  - **Identity**: auto increment
    - v Toplinku a PG implementován jako sekvence s implicitním jménem (*tabulka_id*_seq)
  - **Table**: primární klíče se ukládají do zvláštní tabulky (poslední cesta, jak zařídit unikátní klíče)
  - **Assigned**: zařídí aplikace

Vztahy (relationship)

- Unidirectional
  - je jedna strana, která obsahuje ukazatele
- Bidirectional
  - existuje vlastník a inverzní strana
  - vlastník se stará o ukazatele
  - inverzní strana je updatovaná vlastníkem
- Vlastník
  - 1:1 – strana s cizím klíčem
  - N:1 – strana s cizím klíčem
  - M:N – kterákoliv ze stran

OneToOne, 1:1

- Pravděpodobně chyba v návrhu ER, ale ve výjimečném případě lze.
- Unidirectional, jestliže manžel u manželky není zmíněn, bidirectional, jestliže ano.
- Proč je u Wife uvedeno v mappedBy “wife”?

```
@Entity
public class Husband {
...
    @OneToOne
    private Wife wife;
...
```

```
@Entity
public class Wife {
...
    @OneToOne(mappedBy="wife")
    private Husband husband;
...
```

Opravdovy programator muze, ale nemusi znat jmeno sve zeny. Zna ovsem zpameti celou kodovou tabulku ASCII ci EBCDIC.

OneToMany, 1:N

- Arabská varianta
- POZOR – vlastníkem je strana s cizím klíčem! Když upravíte druhou stranu (třeba setWives), změny se neuloží do db!

```
@Entity
public class Husband {
...
    @OneToMany(mappedBy="husband")
    private Collection<Wife> wives;
...
```

```
@Entity
public class Wife {
...
    @ManyToOne
    private Husband husband;
...
```

ManyToMany, M:N

- Trošku pozměníme téma: budeme počítat i bývalé

```
@Entity
public class Husband {
...
    @ManyToMany
    @JoinTable( name="marriage",
        joinColumns=
            @JoinColumn(name="husb_id", referencedColumnName="id"),
        inverseJoinColumns=
            @JoinColumn(name="wife_id", referencedColumnName="id")
    )
    private Collection<Wife> wives;
...
```

```
@Entity
public class Wife {
...
    @ManyToMany(mappedBy=”wives”)
    private Collection<Husband> husbands;
...
```

Složený primární klíč

- Vřele nedoporučuji, bohužel nutnost při dekompoziční tabulce s dalšími parametry. Pak lze zvážit změnu na entitu.

```
@Embeddable
public class EmployeePK implements Serializable {
    private String name;
    private long id;
...
}

@Entity
public class Employee implements Serializable {
    @EmbeddedId
    EmployeePK primaryKey;
...
```

Dědění

- Velký rozdíl mezi databází a OOP – bohatý strom dědění.
- JPA nabízí 2 strategie (popsané u kořenové entity)
  - JOINED: každý potomek má vlastní tabulku s odkazem na tabulku předka
  - SINGLE_TABLE (default): všechny objekty v jedné tabulce

```
@Entity
@Inheritance(strategy=JOINED)
public class Human {
    @Id
    private int humanId;
    ...
}
```

```
@Entity
public class Man extends Human {
    ...
}
```

Dědění – jaká je to vlastně třída?

- Jak se v tabulce předka pozná, že jde vlastně o potomka?
- @DiscriminatorColumn
  - name (default=”DTYPE”) – jméno sloupce, který určuje, o jaký půjde typ (typicky jméno entity)
  - discriminatorType (default=String) – jakého typu bude sloupec (může být i int)
- @DiscriminatorValue(value="identifikace") – co za třídu

```
@Entity
@Inheritance(strategy=SINGLE_TABLE)
@DiscriminatorColumn(name="type", discriminatorType=STRING, length=3)
@DiscriminatorValue(value="WMN")
public class Woman {
    ...
}
```

Kaskáda

- Kontrolní otázka – znáte CASCADE UPDATE/DELETE?
- Podobnou funkcionalitu nabízí i JPA.
- Možné hodnoty cascade:
  - PERSIST – je-li uložen tento objekt, uloží se i odkazované
  - REFRESH, MERGE – dtto pro refresh, merge
  - REMOVE – cascade delete
  - ALL - obvious

```
@Entity
public class Pharaon {
...
    @OneToMany(cascade=REMOVE, mappedBy="husband")
    private Collection<Wife> wives;
...
```

Jeden nebo dva objekty?

- Na jeden objekt je možné dostat se více způsoby, jak to synchronizovat?
  - jednoduše – je to ten samý objekt
  - em.find(E.class, 1234) == em.find(E.class, 1234) == related.getE()

Dotazování

- Hledání podle klíče je potřeba, ale samozřejmě nestačí.
- Protože se mnoho dat drží v paměti, byla by škoda toho nevyužít. Proto se používá jazyk EJB QL, který toho sice mnoho neumí, ale zato ho JPA umí parsovat a někdy vrátit odpověd i bez spolupráce s databází.

```
query = entityManager.createQuery("SELECT z FROM Zakaznik z");

query.getResultList();
nebo
query.getSingleResult();
nebo
query.executeUpdate();
```

Dotazování – named query

- Pěkná myšlenka je přesunout dotazy k entitě (dotazy vytváří někdo, kdo entitě rozumí)

```
@Entity
@NamedQueries({@NamedQuery(
  name = "Zakaznik.findByIdZakaznik",
  query = "SELECT z FROM Zakaznik z "
          " WHERE z.idZakaznik = :idZakaznik")})
public class Zakaznik {
...
}
```

```
query = entityManager.
   createNamedQuery("Zakaznik.findByIdZakaznik");
```

Dotazování – native query

- Někdy je potřeba položit dotaz příliš složitý, takže nelze využít EJB-QL.
- Odpověď dostaneme jako list of lists:

```
Query q = entityManager.createNativeQuery(
   "select frrmm, sum(kc) from fv group by frrmm order by frrmm");
List res = q.getResultList();
Adding it to a JTable is pretty straightforward:
String[] cols = {”frrmm”,”sumkc”};
jTable1.setModel(new DefaultTableModel(
    new Vector(res),
    new Vector(Arrays.asList(cols))));
```

Dotazování – native query + mapování

- Lépe – požádáme o mapování (přeci jenom, je 21. století...). Výsledkem je seznam objektů, nikoliv pole polí:

```
@Entity
public class StatRow implements java.io.Serializable {
@Id
@Column(name="frrmm")
int frrmm;
double kcsum;

Použití
Query q = entityManager.createNativeQuery(
    "select cast(frrmm as int) as frrmm, sum(kc) as kcsum “
    +” from fv group by frrmm order by frrmm",StatRow.class);
List res = q.getResultList();
```

Transakce

- V nejjednodušším případě průzračně jednoduché:

```
entityManager.getTransaction().begin();
... neco se deje...
entityManager.getTransaction().commit();
v pripade chyby:
entityManager.getTransaction().rollback();


if(entityManager.getTransaction().isActive())...
```

Automatické vytváření tabulek

- je možno nechat databázi (tabulky) vygenerovat
- <property name="toplink.ddl-generation" value="**create-tables**"/>
  - **create-tables**: vždy se pokusí tabulky vygenerovat (což napodruhé a dále selže), vytvoří tedy pouze neexistující tabulky (ovšem neupraví již stávající tabulky)
  - **drop-and-create-tables**: vždy tabulky smaže a znova vytvoří
  - **none** (neuvede se): žádné pokusy o vytvoření tabulek se nekonají

Lazy loading, cache

- JPA implementace typicky nenačítá s objektem všechny odkazované objekty, to by snadno skončilo načtením celé databáze.
- Pro kolekce použije své vlastní implementace, které jsou na začátku prázdné. Teprve při pokusu získat z nich data se objekty načtou z databáze a zpřístupní.
- Pozor, aby se to dělo v perzistentním kontextu (u Hibernate, Toplink s tím neměl problém).
- JPA si drží seznam již načtených objektů (podle jména entity a primárního klíče) a pokud je vyžadován, vrátí ho z cache, nenačítá z databáze.

Používání v kódu

- Poznáte v tomto kódu databázovou aplikaci?

```
void fillWithCalls(Zakaznik zakaznik) {
  clientsNameLabel.setText(zakaznik.getJmeno());
  hovorList.clear();
  for(Telefon tlf : zakaznik.getTelefonCollection()) {
    for(Hovor hovor : tlf.getHovorCollection()) {
      hovorList.add(hovor);
    }
  }
}
```

- JPA implementuje collections a v případě přístupu na ně vytvoří SQL dotazy a kolekce naplní.

JPA implementace

- JPA je pouze API, existuje pro něj více implementací:Hibernate, Oracle Toplink/Essentials, OpenJPA, a další.
- Každá implementace má svá specifika, je lepší si zvolit jednu a tu používat.
- Doporučuje se používat technologie od jednoho providera (tedy JPA dodávané s použitým aplikačním serverem).

Vylepšení JPA 2.0

- Sorted lists – pomocí @OrderBy lze říct, jak mají být objekty v kolekcích (@OneToMany) uspořádány.
- Element Collection:

public **enum FeatureType** { AC, CRUISE, PWR, BLUETOOTH, TV, ... }

```
@Entity public class Vehicle {
        @ElementCollection
        @CollectionTable(name="VEH_OPTNS")
        @Column(name="FEAT")
        Set<FeatureType> optionalFeatures;
```

EJB QL

- CASE, NULLIF, COALESCE
  - práce s NULL ve sloupci
- Criteria API
  - vytváření dotazů dynamicky (programovaný složitější SELECT)
- Support for pessimistic locking (em.lock(obj1, READ))
  - vhodné pro případy, kdy dochází často ke kolizím
  - 6 režimů (3 optimistické, 3 pesimistické)
- <validation-mode> v persistence.xml určuje, jak budou využívány validace

NetBeans podpora

- Dnešní IDE podporují JPA.
- V NetBeans zaregistrujte databázi v záložce Services. Poté ve svém projektu zvolte v okně Projects z kontextového menu New – Entity Classes from Database. Zvolte svoji databázi, založte Persistence Unit, zvolte své tabulky a package a NB vytvoří entity tak, jak jsou definované tabulky v databázi. Umí i relace N:1 (ne M:N).
- Do vygenerovaných entit je potřeba přidat @GeneratedValue, aby si databáze doplňovala primární klíče sama.

Příklad z NetBeans

- Nyní bychom měli plně porozumnět všemu, co NetBeans implicitně vygenerují:

```
@Entity
@Table(name = "STUDENT")
@NamedQue;ries({@NamedQuery(name = "Student.findByRocnik",
query = "SELECT s FROM Student s WHERE s.rocnik = :rocnik"),...})
public class Student implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @Column(name = "STUDKEY", nullable = false)
    private String studkey;
    @Column(name = "JMENO")
    private String jmeno;
    @Column(name = "PRIJMENI")
    private String prijmeni;
    @Column(name = "STIPENDIUM")
    private Float stipendium;
    @Column(name = "NAROZENI")
    @Temporal(TemporalType.TIMESTAMP)
    private Date narozeni;
```

Jaký je vlastně vztah JPA a Hibernate?

- Úspěch Hibernate donutil Sun přijít s ještě lepším standardem. Místo XML používá JPA anotace, což značně zjednodušuje zápis.
- Hibernate lze použít jako implementaci JPA.
- Hibernate se naučil používat anotace i bez JPA.
- JPA může používat XML podobné tomu z Hibernate, které pak má přednost a dá se tak použít k přebití nastavení, i pokud nemáte zdrojový kód.
- Doufám, že teď je to jasné :-)
- Aby to bylo jasnější – Sun k tvorbě JPA přizval Gavina Kinga, autora Hibernate.

Domácí úkol

- Vytvořte 2 tabulky v relaci 1:N (např. Auto a Majitel), každá bude mít alespoň 5 atributů (jméno, adresa, telefon, číslo motoru, platnost technické kontroly)
  - zkuste více typů, samé stringy jsou nuda
- Napište 2 programy, jeden za použití JDBC a druhý JPA:
  - program vytvoří jednu instanci (např. majitele) a k němu dvě odpovídající (tedy auta) a uloží je do db
  - vypíše seznam aut i s majiteli
  - změní atributy – majiteli adresu a autům prodlouží platnost kontroly o 2 roky
  - všechny tři objekty zruší
- Spočítejte řádky programů a dobu strávenou psaním

Náměty k diskuzi

- Složené primární klíče
  - problémy se složitými joiny přes více tabulek
  - nemožnost použít ve vnořených dotazech
  - problémy s dotazy v EJBQL
- Primární klíče
  - přirozené: co v případě změny logiky a ztráty jedinečnosti? Často vede na složené klíče.
  - umělé: podpora v databázi, těžší ruční procházení databáze, klíč nic nevypovídá, potřeba vygenerovat

Linky na JPA

- http://java.sun.com/javaee/5/docs/tutorial/doc/
  - bible, součástí zkoušky bude doslovná znalost každého slova
- http://java.sun.com/javaee/technologies/persistence.jsp
  - JPA
- http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html
  - pěkné příklady
- http://java.cz
  - podcasty (pokud unesete lehký tón) a linky
  - články českých odborníků (Dagi, Roumen, Filemon)

Ještě jeden, jestli si ho zaslouží!

Žena pošle svého manžela, programátora, na nákup se slovy:

"Kup jeden chleba a kdyby ještě měli rohlíky, tak jich vem deset."

Programátor odešel do obchodu, a protože ještě měli rohlíky, koupil deset chlebů.

Přesně podle zadání.