Java Persistence API v príkladoch z Hibernate

Gary Mak a Róbert Novotný

8. apríla 2009

Obsah

1 Úvod

Java Persistence API je špecifikácia štandardizujúca objektovo-relačné mapovanie (teda mapovanie objektov na databázové tabuľky.) V súčasnosti už jestvuje dostatočný počet implementácií (Hibernate, TopLink, Kodo atď). Tutoriál prináša popis základných tried, filozofie práce a značné množstvo príkladov pre použitie JPA (v implementácii Hibernate) v aplikáciách.

Tento materiál vznikol ako voľný preklad pôvodných tutoriálov Garyho Maka k používaniu Hibernate. Tutoriál však bol značne prepracovaný a príklady, v origináli stavané pre XML mapovanie v Hibernate, boli prispôsobené všeobecnej špecifikácii JPA s použitím anotácií.

2 Základy práce s JPA

2.1 Inštalácia Hibernate

Hibernate je komplexný nástroj pre umožnenie a podporu objektovo-relačného mapovania v Java aplikáciách podporujúci špecifikáciu JPA.

Tento nástroj je open-source voľne dostupný na adrese `http://www.hibernate.org`. Navštívte túto adresu a stiahnite si súčasti *Hibernate Core*, *Hibernate Annotations* a *Hibernate Entity Manager*. Distribučný archív rozbaľte do ľubovoľného adresára, napr. `C:\java\hibernate-3.1`.

2.2 Konfigurácia Eclipse

2.2.1 Definovanie používateľskej knižnice pre Eclipse

V okne *Preferences* otvorte *Java | Build Path | User Libraries*. Pridajte vlastnú knižnicu pod názvom *Hibernate 3* a pridajte do nej nasledovné JAR súbory:

- `antlr-2.7.6.jar`
- `asm.jar`
- `asm-attrs.jar`
- `cglib-2.1.3.jar`
- `commons-collections-2.1.1.jar`

- `commons-logging-1.1.1.jar`
- `dom4j-1.6.1.jar`
- `ehcache-1.2.3.jar`
- `ejb3-persistence.jar`
- `hibernate3.jar`
- `hibernate-annotations.jar`
- `hibernate-commons-annotations.jar`
- `hibernate-entitymanager.jar`
- `javassist.jar`
- `log4j-1.2.14.jar`
- `mysql-connector-java-5.1.6-bin.jar`

Následne pridajte túto knižnicu do *build path* vo vašom projekte.

2.3 Vytvorenie základného mapovania pomocou anotácií

V prvom kroku budeme používať JPA na načítavanie inštancií kníh z databázy i na opačný proces, čiže ukladanie resp. aktualizáciu existujúcich objektov.

V predošlej časti sme spomínali, že potrebujeme preklenúť rozdiely medzi objektovým a relačným modelom. V praxi to znamená vytvorenie mapovaní medzi tabuľkami a triedami, atribútmi objektu a stĺpcami tabuľky a medzi asociáciami týkajúcimi sa objektov a kľúčmi medzi tabuľkami. Na rozdiel od klasického prístupu, kde prebieha mapovanie v XML súboroch definície mapovania (*mapping definition*), budeme v tomto prípade nastavovať tieto náležitosti pomocou anotácií. Objekty, ktoré sú namapované na tabuľky, sa nazývajú *perzistentné objekty* (*persistent objects*) alebo *entity*, pretože ich môžeme ukladať do databázy a načítavať ich z nej.

Skúsme najprv nakonfigurovať mapovanie pre triedu knihy `Book`, v ktorej budeme pre jednoduchosť predbežne ignorovať vydavateľa a kapitoly knihy.

```
@Entity
public class Book {
 @ID
 private String isbn;         //ISBN
 private String name;         //titul
 @Transient
```

```
  private Publisher publisher; //vydavatel'
  private Date publishDate;    //dátum vydania
  private int price;           //cena
  @Transient
  private List<Chapter> chapters; //zoznam kapitol

  // gettre a settre
}
```

Perzistentná trieda je anotovaná pomocou `@Entity`. Okrem toho musí mať identifikátor, ktorý je používaný na jednoznačnú identifikáciu inštancií. V našom príklade sme zvolili do úlohy identifikátora ISBN. Inštančnú premennú zodpovedajúcu identifikátoru sme označili ako `@ID`. Ignorované inštančné premenné (teda vydavateľa a zoznam kapitol) sme označili ako `@Transient`. Všetky ostatné premenné budú automaticky ukladané do databázy.

2.4 Konfigurácia JPA

Predtým samotným používaním JPA (teda pred získavaním a ukladaním objektov) musíme vykonať isté konfiguračné nastavenia. Potrebujeme namapovať perzistentné objekty (zrejme nie všetky triedy v našom projekte musia byť perzistentné), určiť typ používanej databázy a nastaviť náležitosti potrebné k pripojeniu (server, prihlasovacie meno a heslo).

Konfigurácie JPA prebieha pomocou XML súboru `persistence.xml`. Vytvoríme súbor `META-INF/persistence.xml` v adresári zdrojových súborov (typicky adresár `src`; čiže na vrchole hierarchie balíčkov), čím zabezpečíme, že ho Eclipse skopíruje do koreňového adresára v `CLASSPATH`.

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
          http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
         version="1.0">

 <persistence-unit name="books">
  <properties>
   <property name="hibernate.connection.driver_class"
          value="com.mysql.jdbc.Driver" />
   <property name="hibernate.connection.url"
          value="jdbc:mysql://localhost:80/students" />
   <property name="hibernate.connection.username"
          value="scott" />
   <property name="hibernate.connection.password"
          value="tiger" />
```

```
    </properties>
  </persistence-unit>
</persistence>
```

V rámci neho sme definovali *perzistenčnú jednotku* (*persistence unit*) s názvom books. Takýchto jednotiek môže byť v rámci systému viac (typicky sa každá nachádza v samostatnom JAR súbore so samostatným konfiguračným súborom). My sa však obmedzíme na jedinú jednotku.

Následne potrebujeme vytvoriť *továreň pre správcov entít* (*entity manager factory*), čo je globálny objekt zodpovedný za manažovanie správcov entít. *Správca entít* (*entity manager*) je fundamentálnym objektom pre prácu s JPA. Manažuje jednotlivé objekty, spravuje ich životný cyklus, prekladá volania metód na SQL príkazy, udržiava spojenie s databázou atď. Väčšina práce s JPA je založená na volaní metód tohto objektu.

Typický kód vyzerá nasledovne:

```
EntityManagerFactory entityManagerFactory
  = Persistence.createEntityManagerFactory("students");
EntityManager entityManager
  = entityManagerFactory.createEntityManager();
```

2.5 Získavanie a ukladanie objektov

2.5.1 *Správca entít*

Podobne ako v prípade JDBC, pred a po samotnom načítavaní (resp. ukladaní) objektov je potrebné vykonať niekoľko úkonov:

1. nakonfigurovať *entity manager factory* (stačí raz, pred prvým získaním inštancie správcu entít)
2. získať správcu entít,
3. vykonať požadované operácie,
4. upratať po sebe, čiže uzatvoriť správcu entít.

```
// nakonfigurovat' Entity manager factory
// (stačí raz, pred prvým získaním inštancie správcu entít)
EntityManagerFactory entityManagerFactory
  = Persistence.createEntityManagerFactory("books");
// získat' správcu entít
EntityManager entityManager
  = entityManagerFactory.createEntityManager();
try {
```

```
  // tu používame správcu entít na prácu s objektami
} finally {
  //upratat' po sebe, čiže uzatvorit' správcu entít.
  entityManager.close();
}
```

2.5.2 *Načítavanie objektov*

Ak máme k dispozícii identifikátor knihy (teda ISBN), celý objekt knihy môžeme vytiahnuť z databázy nasledovným spôsobom:

```
Book book = (Book) entityManager.find(Book.class, isbn);
```

Táto metóda zodpovedá metóde `Session#get()` z klasického Hibernate API. V prípade, že v databáze sa nenachádza objekt s daným identifikátorom, metóda vráti `null`.

Na dopytovanie klasickej databázy je všeobecne používaný jazyk SQL a v JPA je dispozícii analogický jazyk zvaný JPQL. Bližšie o ňom pojednáva časť 8, zatiaľ si ukážeme jednoduché príklady.

Nasledovný dopyt, ktorý vráti zoznam všetkých kníh v databáze:

```
Query query = entityManager.createQuery("select b from Book b");
List books = query.getResultList();
```

Ak máme istotu, že výsledný zoznam obsahuje len jeden objekt, môžeme použiť metódu `getSingleResult()`, ktorá vráti namiesto zoznamu danú inštanciu.

```
Query query
  = entityManager.createQuery("select b from Book b where isbn = ?");
query.setParameter(0, isbn);
Book book = (Book) query.getSingleResult();
```

Tento príklad zároveň ukazuje použitie parametrov v dopytoch. Namiesto otáznika sa dosadí hodnota špecifikovaná v metóde `setParameter()` na objekte dopytu (`Query`).

2.5.3 *Zobrazovanie SQL dopytov odosielaných JPA*

Pri získavaní a ukladaní objektov Hibernate automaticky generuje na pozadí (a za oponou) SQL dopyty, ktorými získava dáta z databázy. Niekedy je vhodné si nechať tieto vygenerované dopyty vypisovať a pomôcť si tým pri ladení. Ak nastavíme v súbore `persistence.xml` parameter `show_sql` na `true`, Hibernate bude vypisovať SQL dopyty na štandardný výstup.

```
<persistence-unit name="books">
  <!-- ... -->
  <property name="hibernate.show_sql" value="true" />
```

```
  <!-- ... -->
</persistence-unit>
```

Na zaznamenávanie SQL príkazov a parametrov môžeme tiež použiť štandardnú knižnicu pre ladiace výpisy (*logovanie*) log4j. Konfiguračný súbor obsahujúci nastavenia logovacích kanálov a filtre hlášok `log4j.properties` umiestnime do koreňového adresára hierarchie balíčkov.

```
### posielame hlásenia na štd. výstup ###
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=
  %d{yyyy-MM-dd HH:mm:ss} %5p %c{1}:%L -%m%n


### posielame hlásenia do súboru hibernate.log ###
#log4j.appender.file=org.apache.log4j.FileAppender
#log4j.appender.file.File=hibernate.log
#log4j.appender.file.layout=org.apache.log4j.PatternLayout
#log4j.appender.file.layout.ConversionPattern=
  %d{yyyy-MM-dd HH:mm:ss} %5p %c{1}:%L -%m%n

log4j.rootLogger=error, stdout

log4j.logger.org.hibernate.sql=DEBUG
log4j.logger.org.hibernate.type=DEBUG
```

2.5.4 *Ustanovenie transakcií*

Ak používame viacero aktualizačných príkazov, je temer žiadúce, aby boli vykonávané v rámci transakcie. Ak by náhodou niektorý z príkazov v transakcii zlyhal, transakciu je možné zrušiť a všetky úpravy v rámci nej je možné vrátiť späť a tým predísť nekonzistencii dát.

```
EntityManager entityManager
  = entityManagerFactory.createEntityManager();
EntityTransaction tx = null;
try {
  // získame transakciu
  tx = entityManager.getTransaction();
  // naštartujeme ju
  tx.begin();
  // ...
  // tu používame príkazy pracujúce so sedením
```

```
  // ...

  // transakcia bola úspešná, potvrdíme ju
  tx.commit();
} catch (PersistenceException e) {
  // transakcia bola zlyhala, odrolujeme ju späť
  if (tx != null) tx.rollback();
  throw e;
} finally {
  entityManager.close();
}
```

2.5.5 *Ukladanie objektov do databázy*

Ak chceme uložiť novovytvorený objekt do databázy, použijeme na to metódu `persist()`. JPA odošle do databázy SQL príkaz `INSERT`.

```
entityManager.persist(book);
```

Odstraňovať objekt z databázy je možné volaním metódy `remove()`, ktorému zodpovedá vykonanie SQL príkazu `DELETE`.

```
entityManager.remove(book);
```

2.6 Automatické generovanie databázovej schémy pomocou Hibernate

Zvyčajný postup vývoja aplikácii spočíva vo vytvorení databázových tabuliek, ktorým sa prispôsobuje objektový model. Tento prístup je v prípade používania klasických metód na mapovanie tabuliek na objekty používaný pomerne často, ale v mnohých prípadoch môže brániť použitiu niektorých techník objektovo orientovaného programovania. Hibernate umožňuje zvoliť opačný prístup – generovanie a aktualizáciu databázových tabuliek založených na existujúcom objektovom modeli a definícií mapovaní.

V konfiguračnom súbore `persistence.xml` stačí nastaviť vlastnosť:

```
<property name="hibernate.hbm2ddl.auto" value="update" />
```

Hibernate bude automaticky vytvárať tabuľky pre nové entity, aktualizovať metadáta týkajúce sa stĺpcov, či automaticky pridávať alebo odoberať nové stĺpce a to všetko na základe zmien objektového modelu.

2.6.1 *Doladenie detailov databázovej schémy*

V predošlom príklade mapovacieho súboru sme zanedbali niektoré detaily týkajúce sa používaných tabuliek – napr. maximálnu dĺžku stĺpca, či nenullové obmedzenia. Ak chceme vygenerovať databázovú schému na základe definícií mapovaní z

príkladu, je vhodné dodať do nich tieto údaje. Na tieto náležitosti slúži anotácia `@Column`:

```
@Entity
public class Book {
  @ID
  @Column(length=50)
  private String isbn;         //ISBN

  @Column(length=100, nullable=false)
  private String name;         //titul
  @Transient
  private Publisher publisher; //vydavatel'
  private Date publishDate;    //dátum vydania
  private int price;           //cena
  @Transient
  private List<Chapter> chapters; //zoznam kapitol

  // gettre a settre
}
```

Atribút `length` reprezentuje dĺžku stĺpca; pomocou `nullable` vieme povoľovať či zakazovať `NULL` hodnoty v stĺpcoch.

3 Identifikátor objektu

3.1 Automaticky generované identifikátory objektov

V predošlom príklade sme používali na jednoznačnú identifikáciu objektov kníh atribút ISBN. Hodnotu tohto identifikátora sme danej knihe museli priradiť automaticky. V tomto prípade budeme hovoriť o *priradzovanom identifikátore* (*assigned identifier*).

3.1.1 *Použitie generovaného identifikátora pre perzistentné objekty*

Predstavme si, že sa používateľ pri zadávaní ISBN a uložení knihy do databázy pomýlil. Neskôr prišiel na chybu a chcel ju opraviť. Žiaľ zistil, že v prípade perzistentných objektov už nie je možné identifikátor dodatočne zmeniť. Hibernate totiž považuje objekty s rôznymi identifikátormi za rozličné objekty (a to i v prípade, že sa ich atribúty zhodujú).

Samozrejme nás môže napadnúť vykonať zmenu nesprávnej hodnoty ISBN priamo v databáze. V našom relačnom modeli má tabuľka `BOOK` (knihy) primárny kľúč `isbn`. Tabuľka `CHAPTER` (kapitoly) obsahuje cudzí kľúč `book_isbn`, ktorý sa

odkazuje na tabuľku `BOOK`. Ak chceme opraviť chybu, musíme zmeniť hodnoty v primárnom i v cudzom kľúči.

Na základe tohto príkladu je vidieť, že používanie niektorého z atribútov používaných objektov pre úlohu identifikátora nemusí byť práve najvhodnejším prístupom (najmä v prípadoch, že hodnoty identifikátora sú zadávané používateľom). Ukážeme si, že je oveľa lepšie používať pre identifikátor nový, automaticky generovaný, atribút. Keďže samotný identifikátor nemá v samotnej aplikačnej logike žiadny iný význam, po prvom priradení sa už nebude nikdy meniť.

V prvom kroku realizácie tejto myšlienky upravíme triedu `Book`; dodáme do nej inštančnú premennú `id`, ktorú označíme ako identifikátor pomocou anotácie `@ID`.

```
@Entity
public class Book {
  @ID
  private long id;

  //...
}
```

3.1.2 *Automatické generovanie identifikátorov v JPA*

V ďalšom kroku potrebujeme inštruovať JPA, aby pred uložením objektu do databázy vygeneroval pre daný objekt nový identifikátor. K dispozícii máme viacero spôsobov generovania; niektoré sú všeobecné a niektoré špeciálne zamerané na konkrétne databázy.

Najjednoduchšie je prenechať voľbu stratégie generovania identifikátorov na databázu. To dosiahneme pridaním anotácie `@GeneratedValue` k inštančnej premennej `ID`.

Okrem toho musíme upraviť definíciu inštančnej premennej pre `ISBN` – anotujeme ju ako klasický atribút a pridáme doň obmedzenie na jedinečnosť a nenullovosť.

```
@Entity
public class Book {
  @ID
  @GeneratedValue
  private long id;

  @Column(length=50, nullable=false, unique=true)
  private String isbn;

  @Column(length=100, nullable=false)
  private String name;
```

```
    @Transient
    private Publisher publisher;
    private Date publishDate;
    private int price;

    @Transient
    private List chapters;

    // gettery a settery
}
```

3.1.3 *Životný cyklus objektov*

Objekt spravovaný JPA sa môže nachádzať v jednom viacerých globálnych stavoch, ktorý sa mení volaním príslušných metód na správcovi entít.

- nový alebo neuložený (*transient*) – predstavuje inštanciu, ktorá je mimo správy JPA. Typickým príkladom objektu v tomto stave je čerstvo vytvorená inštancia, ktorá nebola uložená metódou `persist()`. Takýto objekt nemá pridelený žiadny identifikátor.

- spravovaný (*managed*) – objekt pod správcou JPA. Nový objekt sa do tohto stavu dostane po zavolaní metódy `persist()`, v ktorej mu bude pridelený identifikátor. Zmeny na jeho inštančných premenných sú monitorované a v prípade potreby zavedené do databázy.

- odpojený (*detached*) – predstavuje objekt, ktorý bol vyňatý spod kontroly správcu entít. Spravovaný objekt sa dostane do tohto stavu najčastejšie po uzatvorení správcu entít. Odpojený objekt môžeme opäť dostať pod správu nového správcu entít zavolaním metódy `merge()`.

- odstránený (*removed*) – objekt, ktorý síce jestvuje v podobe inštancie, ale z databázy bol odstránený metódou `remove()` (inak povedané, databázový riadok zodpovedajúci tomuto objektu bol odstránený).

V predošlej sekcii sme spomínali, že nový objekt uložíme metódou `persist()`. Zmeny na inštančných premenných spravovaného objektu sú do databázy ukladané priebežne (automaticky), po `commit()`nutí transakcie alebo po zavolaní metódy `flush()`. Ak nevieme, či je daný objekt nový alebo spravovaný, vieme to zistiť zavolaním metódy `contains()` na správcovi entít, kde ho uvedieme ako parameter – metóda vracia `true` pre spravované objekty.

Ak máme perzistentný objekt, kde sa identifikátor generuje automaticky, a hodnota identifikátorového atribútu je prázdna a tento objekt odovzdáme metóde

`persist()`, správca entít najprv automaticky vygeneruje hodnotu pre identifikátor a následne vykoná SQL príkaz `INSERT`. V prípade, že hodnota identifikátorového atribútu nie je prázdna, správca entít bude považovať objekt za spravovaný. Zavolanie metódy `persist()` na spravovanej entite tak nemá žiaden efekt, s jedinou výnimkou: v prípade nastavenia kaskádneho vykonávania operácií na asociáciách sa volanie metódy propaguje na asociované entity. (Podrobnosti uvedieme v ďalších kapitolách.)

Ako však správca entít zistí, či hodnota tohto atribútu je prázdna? Jednoducho, na základe `null` hodnoty v inštančnej premennej. Predtým však budeme musieť zmeniť primitívny typ `long` na objektový protipól `java.lang.Long`.

Musíme však upraviť našu triedu:

```
@Entity
public class Book {
  @ID
  @GeneratedValue
  private Long id; // Long namiesto long
  //...
}
```

Tento problém sa môže vyskytnúť aj v prípade ostatných atribútov triedy `Book`. Povedzme, že nepoznáme cenu knihy. Akú hodnotu potom máme priradiť do príslušnej premennej `price`? Nulu? Záporné číslo? Takéto riešenia nemusia byť optimálne. Zrejme je lepšie opäť použiť namiesto primitívneho typu objekt (v tomto prípade `java.lang.Integer` namiesto `int`), a považovať `null`ovú hodnotu na reprezentáciu neznámej hodnoty.

```
public class Book {
  //...
  private Integer price; // Integer namiesto int
  //...
}
```

3.2 Odpojené objekty a ich opätovné spravovanie správcom entít

Dosiaľ sme v príklade používali len jediného správcu entít, v rámci ktorého boli spracované jednotlivé objekty. Jestvujú však prípady, keď jeden objekt preputuje medzi viacerými správcami entít. Príkladom je objekt, ktorý je vytiahnutý z databázy pomocou metódy `find()`, poslaný po sieti technológiou RMI, upravený a poslaný po sieti späť. Takýto objekt sa stane odpojeným (čiže nie je pod správou žiadneho správcu entít). Na to, aby sme ho mohli uložiť do databázy, ho musíme najprv dostať pod niektorého správcu entít (dostať do stavu „spravovaný“), čo do-

siahneme zavolaním metódy `merge()`. Do parametra vložíme odpojenú inštanciu a metóda vráti spravovanú inštanciu, do ktorej sa skopíruje stav odpojenej inštancie.

Príklad použitia je:

```
Book book = entityManager1.find(Book.class, 25L);
//... tu sa udeje vel'a vecí

// .. knihu dáme pod správu druhého správcu entít
Book managedBook = entityManager2.merge(book);
// managedBook je pod správcom entít
```

V prípade aplikácií, ktoré používajú len jediného správcu entít bez odpájania objektov však na používanie metódy `merge()` často nie je dôvod a je možné sa zaobísť aj bez nej.

3.3 Historické databázové schémy

3.3.1 *Zložené (composite) identifikátory objektov*

V praxi sa niekedy stáva, že v projekte musíme používať už existujúcu databázu s pevne danou štruktúrou, ktorú nemôžeme meniť, čo nás môže obmedziť pri tvorbe objektového modelu. Príkladom sú tabuľky používajúce zložené primárne kľúče (teda primárne kľúče pozostávajúce z viacerých stĺpcov). V takomto prípade si nemôžeme dovoliť pridať do tabuľky nový stĺpec pre identifikátor, ale musíme prispôsobiť mapovanie a návrh triedy.

Predstavme si „historickú“ tabuľku, ktorá bola vytvorená nasledovne:

```
CREATE TABLE CUSTOMER (
  COUNTRY_CODE VARCHAR(2) NOT NULL,
  ID_CARD_NO VARCHAR(30) NOT NULL,
  FIRST_NAME VARCHAR(30) NOT NULL,
  LAST_NAME  VARCHAR(30) NOT NULL,
  ADDRESS    VARCHAR(100),
  EMAIL      VARCHAR(30),

  PRIMARY KEY (COUNTRY_CODE, ID_CARD_NO)
);
```

Vložme do nej vzorové dáta pomocou nasledovného SQL príkazu:

```
INSERT INTO CUSTOMER
(COUNTRY_CODE, ID_CARD_NO, FIRST_NAME, LAST_NAME, ADDRESS, EMAIL)
VALUES
('mo', '1234567(8)', 'Gary', 'Mak', 'Address for Gary', 'gary@mak.com');
```

Skúsme vytvoriť na základe tejto tabuľky perzistentnú triedu. Presnejšie povedané, budeme potrebovať dve triedy: jednu s inštančnými premennými zodpovedajúcimi primárnemu kľúču a jednu regulárnu triedu zákazníka s ostatnými atribútmi.

```
public class CustomerId {
  private String countryCode;
  private String idCardNo;

  public CustomerId(String countryCode, String idCardNo) {
    this.countryCode = countryCode;
    this.idCardNo = idCardNo;
  }
}
```

Táto trieda nepotrebuje žiadne špeciálne anotácie. Ďalej vytvoríme triedu zákazníka:

```
@Entity
public class Customer {

  @Column(name="FIRST_NAME")
  private String firstName;

  @Column(name="LAST_NAME")
  private String lastName;

  @Column(name="ADDRESS")
  private String address;

  @Column(name="EMAIL")
  private String email;

  // gettre a settre
}
```

Zložený identifikátor dodáme do triedy zákazníka ako inštančnú premennú s príslušnou anotáciou:

```
public class Customer {
  @EmbeddedId
  private CustomerId customerId;
}
```

Ak chceme v mapovaní používať historické názvy stĺpcov, môžeme ich definovať v rámci anotácie `@AttributeOverrides`:

```
public class Customer {
  @AttributeOverrides({
    // inštančná premenná customerId.countryCode
    // sa mapuje na stĺpec COUNTRY_CODE
    @AttributeOverride(name = "countryCode",
```

```
    column = @Column(name="COUNTRY_CODE") ),

    // inštančná premenná customerId.idCardNo
    // sa mapuje na stĺpec ID_CARD_NO
    @AttributeOverride(name = "idCardNo",
    column = @Column(name="ID_CARD_NO") )
  })}
  private CustomerId customerId;
```

Ak budeme vyhľadávať objekt zákazníka metódou `find()`, do druhého parametra stačí dodať novú inštanciu triedy `CustomerId` s vyplnenými inštančnými premennými.

```
CustomerId customerId = new Customer();
customerId.setCountryCode("mo");
customerId.setIdCardNo("1234567(8)");

Customer customer = entityManager.find(Customer.class, customerId);
```

Ak budeme chcieť uložiť do databázy nového zákazníka, pre hodnotu identifikátora použijeme tiež novú inštanciu triedy `CustomerId`.

```
Customer customer = new Customer();
customer.setId(new CustomerId("mo", "9876543(2)"));
customer.setFirstName("Peter");
customer.setLastName("Lou");
customer.setAddress("Address for Peter");

customer.setEmail("peter@lou.com");
entityManager.persist(customer);
```

Ak chceme zaručiť správne fungovanie cacheovania hodnôt identifikátorov (čo zvyšuje výkon), musíme prekryť metódy `equals()` a `hashCode()` na našej identifikátorovej triede. (Pripomeňme, že metóda `equals()` sa používa zistenie ekvivalencie dvoch objektov a `hashCode()` na získanie hašu daného objektu.)

Na vytvorenie týchto metód môžeme použiť dva prístupy. Prvým je použitie tried `EqualsBuilder` a `HashCodeBuilder` z projektu Jakarta Commons-Lang (do projektu v Eclipse si pridáme súbor `commons-lang.jar`) nasledovným spôsobom:

```
public class CustomerId implements Serializable {
  ...
  public boolean equals(Object obj) {
    if (!(obj instanceof CustomerId)) return false;
    CustomerId other = (CustomerId) obj;

    return new EqualsBuilder()
```

```
            .append(countryCode, other.countryCode)
            .append(idCardNo, other.idCardNo)
            .isEquals();
    }

    public int hashCode() {
      return new HashCodeBuilder()
            .append(countryCode)
            .append(idCardNo)
            .toHashCode();
    }
}
```

Druhý spôsob môžeme použiť v prípade, keď používame Eclipse verzie 3.2 a novšej. Na vygenerovanie metód môžeme použiť funkciu *Source* | *Generate hashCode() and equals()*.

Ak pri perzistencii zákazníkov `Customer` nie sme obmedzení na „historickú“ relačnú schému, môžeme použiť automaticky generovaný identifikátor.

```
public class Customer {
  @ID
  @GeneratedValue
  private Long id;

  private String countryCode;
  private String idCardNo;
  private String firstName;
  private String lastName;
  private String address;
  private String email;

  // gettre a settre
}
```

4 Asociácie M:1 a 1:1

4.1 Asociácia typu M:1 (*many-to-one*)

V našej aplikácii pre internetový predaj kníh máme knihu vydávanú jedným vydavateľom. Na druhej strane jeden vydavateľ zrejme vydal viacero kníh. Takýto druh asociácie medzi entitami nazývame asociácia M:1 (*many-to-one*). Ak máme asociáciu, v ktorej vieme získať z knihy jej vydavateľa, nazývame ju *jednosmernou* (*unidirectional*). *Obojsmerná* (*bidirectional*) asociácia dokáže zabezpečiť prístup v

oboch smeroch – z knihy vieme získať vydavateľa a z vydavateľa knihu.

Ak budeme chcieť zaviesť do nášho systému vydavateľa, musíme vykonať zvyčajné kroky – vytvoriť triedu (vrátane automaticky generovaného identifikátora) a dodať do nej príslušné anotácie.

```
@Entity
public class Publisher {
  @ID
  @GeneratedValue
  private Long id;

  @Column(length=50, nullable=false, unique=true)
  private String code;

  @Column(length=100, nullable=false)
  private String name;

  @Column(length=200)
  private String address;

  // gettre a settre
}
```

V našej triede pre knihu `Book` musíme mať atribút `publisher` reprezentujúci vydavateľa, ktorý bude typu `Publisher`.

```
public class Book {
  //...
  private Publisher publisher;
  //...

  // gettre a settre
}
```

Dosiaľ bol tento atribút označený ako `@Transient`, čiže pri perzistencii sa ignoroval. Ak ho chceme používať pri mapovaní triedy na tabuľku, musíme ho anotovať príslušnou asociačnou anotáciou. Asociácii M:1 zodpovedá anotácia `@ManyToOne`, ktorú dodáme k inštančnej premennej.

```
public class Book {
  //...
  @ManyToOne
  private Publisher publisher;
  //...

  // gettre a settre
```

```
}
```

4.1.1 *Inicializácia asociácií podľa potreby (lazy initialization)*

Predpokladajme, že máme metódu na získanie objektu knihy z databázy na základe jej identifikátora. Keďže sme pridali mapovanie pre asociáciu M:1, pri načítaní knihy by sa mal automaticky načítať aj objekt vydavateľa (aby sme k nemu mohli pristupovať pomocou gettera `getPublisher()`).

```
public Book getBooks(Long id) {
  EntityManager entityManager
    = entityManagerFactory.createEntityManager();
  try {
    Book book = entityManager.find(Book.class, id);
    return book;
  } finally {
    entityManager.close();
  }
}
```

Ak by sme sa pokúsili pristúpiť k objektu vydavateľa cez `book.getPublisher()` niekde mimo metódy `getBooks()`, vyvolali by sme výnimku `LazyInitialization-Exception`.

```
System.out.println(book.getName());
System.out.println(book.getPublisher().getName());
```

Ak by sme tento kód použili vo vnútri `try` bloku v tele metódy `getBooks()`, všetko by bolo v poriadku. Aká je však príčina vyhodenej výnimky? Pri načítavaní objektu knihy by sme možno očakávali, že sa zároveň načíta i asociovaný vydavateľ. To však nie je vždy pravda. Správca entít totiž načítava asociované objekty až pri prvom prístupe k nim. Tento prístup sa nazýva *inicializácia podľa potreby* (*lazy initialization*, doslovne *lenivá inicializácia*) a v prípade jej správneho použitia sa zamedzí vykonávaniu zbytočných databázových dopytov a teda zvýši výkon.

Problém nastáva v situácii, keď chceme pristupovať k asociovanému objektu mimo otvoreného správcu entít. Keďže k vydavateľovi sme prvýkrát pristúpili až mimo metódy `getBook()` (kde už nemáme otvoreného správcu entít), JPA vyhodil výnimku.

Ak chceme, aby bol vydavateľ knihy dostupný aj mimo otvoreného správcu entít, máme možnosť vypnúť *lazy* inicializáciu. To však môže mať vplyv na výkonnosť, keďže s každým načítaním inštancie knihy sa bude musieť načítať aj inštancia vydavateľa.

```
@ManyToOne(fetch=FetchType.EAGER)
private Publisher publisher;
```

Ak si pozrieme SQL dopyty, ktoré generuje Hibernate na pozadí, ukáže sa, že pri získavaní inštancie knihy sa odošle len jediný SQL dopyt s použitím joinu.

Prekvapenie však môže nastať pri používaní dopytov JPQL:

```
EntityManager em = emf.createEntityManager();
try {
  Query query = em.createQuery(
    "select book from Book as book where book.isbn = ?");
  query.setParameter(0, isbn);
  Book book = (Book) query.getSingleResult();
  System.out.println(book.getPublisher());
} finally {
  entityManager.close();
}
```

V tomto prípade sa vykonajú dva SQL dopyty, pretože JPQL dopyty sa prekladajú na dopyty SQL priamo.

Ak chceme zvoliť pri používaní HQL dopytov prístup využívajúci joiny, musíme použiť nasledovnú syntax:

```
EntityManager em = emf.createEntityManager();
try {
  Query query = em.createQuery(
    "select book from Book book " +
    "join fetch book.publisher where isbn = ?");
  query.setParameter(0, isbn);
  Book book = (Book) query.getSingleResult();
  System.out.println(book.getPublisher());
} finally {
  entityManager.close();
}
```

Ak použijeme v JPQL dopyte `join fetch`, vynútime tým inicializáciu príslušnej asociácie a to i v prípade, že je načítavaná v *lazy* režime. To môžeme s výhodou použiť na inicializáciu objektov v *lazy* asociáciách, a teda docieliť možnosť ich používania aj mimo otvoreného sedenia.

4.1.2 *Kaskádovanie operácií na asociáciách*

Ak vytvoríme novú inštanciu knihy a priradíme jej novú inštanciu vydavateľa a budeme chcieť uložiť túto knihu do databázy, vyvstane otázka, či s uložením knihy sa uloží aj vydavateľ. Odpoveď je záporná – pri ukladaní knihy by nastala výnimka. To znamená, že objekty musíme ukladať postupne a navyše v správnom poradí (najprv vydavateľa a potom knihu).

```
EntityManager em = emf.createEntityManager();
```

```
EntityTransaction tx = null;
try {
  tx = em.getTransaction();
  tx.begin();

  em.persist(publisher);
  em.persist(book);

  tx.commit();

} catch (PersistenceException e) {
  if (tx != null) tx.rollback();
  throw e;
} finally {
  em.close();
}
```

V tomto jednoduchom prípade je ukladanie objektov po poradí ešte zvládnuteľné. V prípade zložitých asociácií (objekt odkazujúci na iný objekt, ktorý môže odkazovať ešte na iný objekt) by ich ukladanie bolo značne komplikované. Našťastie v JPA existuje možnosť uložiť celú hierarchiu objektov pomocou jediného príkazu. Ak pridáme do anotácie `@ManyToOne` atribút `cascade`, JPA bude zvolené operácie ukladania a aktualizácie kaskádne propagovať v smere asociácií.

Inak povedané, uloženie (aktualizácia) knihy vynúti aj uloženie (aktualizáciu) vydavateľa.

```
@ManyToOne(fetch=FetchType.EAGER,
        cascade={CascadeType.PERSIST})
private Publisher publisher;
```

Kaskádovanie ukladania/aktualizácie je užitočné aj v prípade, že ukladáme zložitú štruktúru objektov, ktorá pozostáva z nových i existujúcich objektov. Ak použijeme metódu `persist()`, správca entít sa automaticky rozhodne, ktoré objekty má uložiť a ktoré aktualizovať.

```
entityManager.persist(book);
```

Okrem kaskádneho ukladania môžeme tiež používať aj kaskádne odstraňovanie objektov. Stačí dodať do atribútu `cascade` hodnotu `CascadeType.REMOVE`.

```
@ManyToOne(fetch=FetchType.EAGER,
        cascade={CascadeType.PERSIST,
              CascadeType.REMOVE})
private Publisher publisher;
```

4.1.3 *Používanie medzitabuľky pre asociácie M:1*

V predošlých prípadoch použitia asociácie M:1 sme používali na jej reprezentáciu stĺpec `publisher_id` v tabuľke `BOOK`. Inou možnosťou je zvoliť odlišný návrh tabuliek – môžeme vytvoriť novú tabuľku `BOOK_PUBLISHER` obsahujúci asociáciu medzi knihou a vydavateľom. Táto tabuľka sa zvykne nazývať *join table*. Voliteľný atribút `optional=true` indikuje, že riadok do tejto tabuľky sa vloží len v prípade, keď objekt v asociácii nie je nullový. (Zodpovedá to asociácii, kde na opačnej strane nemusí byť žiadny objekt, napr. kniha, ktorá nemá vydavateľa.)

```
@ManyToOne(fetch=FetchType.EAGER,
        cascade={CascadeType.PERSIST, CascadeType.MERGE},
        optional=true)
@JoinTable(name="book_publisher",
   joinColumns    = @JoinColumn(name="book_id",
                            referencedColumnName="id"),
   inverseJoinColumns = @JoinColumn(name="publisher_id",
                            referencedColumnName="id",
                            nullable=false)
  )
private Publisher publisher;
```

4.2 Asociácia 1:1 (*one-to-one*)

V predošlej časti sme rozoberali asociácie M:1. Teraz si ukážeme, ako je možné obmedziť asociáciu tak, aby na každej strane asociácie mohol byť maximálne jeden objekt. Tomuto typu asociácie zodpovedá anotácia `OneToOne` (1:1). Ukážme si túto asociáciu na príklade zákazníka z predošlého dielu.

```
public class Customer {
  @ID
  @GeneratedValue
  private Long id;

  @Column(nullable=false)
  private String countryCode;

  @Column(nullable=false)
  private String idCardNo;

  private String firstName;

  private String lastName;
```

```
  private String address;

  private String email;

  // gettre a settre
}
```

Predstavme si teraz adresu zákazníka ako samostatnú triedu. Medzi zákazníkom a adresou je zrejme asociácia 1:1 (zákazník má jednu adresu a na jednej adrese je jeden zákazník). Vytvorme teda triedu pre adresu `Address` a namapujeme ju na databázovú tabuľku zvyčajným spôsobom. Mapovanie asociácie zatiaľ vynecháme.

```
public class Customer {
  ...

  private Address address;
}

public class Address {
  private Long id;
  private String city;
  private String street;
  private String doorplate;

  // gettre a settre
}
```

Adresu môžeme priamo anotovať ako entitu:

```
@Entity
public class Address {
  @ID
  @GeneratedValue
  private Long id;

  private String city;

  private String street;

  private String doorplate;

  // gettre a settre
}
```

Teraz prejdeme k definícii mapovania asociácie 1:1. Zjavným spôsobom je použitie anotácie `@OneToOne`. Konfigurácia *lazy* inicializácie, získavania asociovaných

objektov a kaskádovania je identická ako v predošlom prípade.

```
@Entity
public class Customer {
  //...

  @OneToOne
  private Address address;

  //...

}
```

Týmto sme namapovali jednosmernú asociáciu od zákazníka k adrese. Ak by sme chceli mať obojsmernú asociáciu, jej opačný smer namapujeme ako asociáciu 1:1, v ktorom sa odkážeme na atribút `address` v zákazníkovi `Customer`.

```
public class Address {

  private Long id;

  private String city;

  private String street;

  private String doorplate;

  @OneToOne(mappedBy="address")
  private Customer customer;

  // gettre a settre
}
```

Anotácia `@OneToOne` hovorí, že jedna adresa bude namapovaná na jedného zákazníka. Atribút `mappedBy` hovorí, že táto asociácia je opačnou stranou k asociácii, ktorá už bola namapovaná v inštančnej premennej `address` na opačnej strane, teda v triede `Customer`. Ak by sme tento atribút neuviedli, mali by sme dve jednosmerné asociácie typu 1:1, kde by vznikol problém s manažovaním objektov (nastal by prípad, keby jeden objekt bol do databázy ukladaný dvakrát – raz za každú stranu asociácie.)

5 Asociácie 1:M a M:N

5.1 Asociácie 1:M

V predošlom príklade sme považovali každú kapitolu za jeden reťazec, ktorý sme ukladali v zozname. Tento prístup teraz zovšeobecníme a rozšírime. Každú kapitolu budeme reprezentovať ako samostatný perzistentný objekt. Keďže jedna kniha môže mať viacero kapitol, asociáciu z knihy ku kapitole budeme nazývať asociáciou 1:M (*one-to-many*). Túto asociáciu najprv namapujeme ako jednosmernú (z knihy budeme vedieť získať zoznam kapitol) a neskôr ju rozšírime na obojsmernú.

Pripomeňme, že trieda kapitoly `Chapter` v našej aplikácii ešte nemá definované anotácie pre perzistenciu. Pristúpime teda k ich vytvoreniu. V rámci nich priradíme kapitole automaticky generovaný identifikátor.

```
@Entity
public class Chapter {
  @Id
  @GeneratedValue
  private Long id;

  private int chapterIndex; //poradie kapitoly

  @Column(length=100)
  private String title; //názov

  private int numOfPages; //počet strán

  // gettre a settre
}
```

V našej triede pre knihu `Book` sme už definovali kolekciu, v ktorej budú uložené kapitoly. Ak používame Javu 5 a novšiu, stačí zmeniť dátový typ zo `Stringu` na `Chapter` v generiku. Otázka je, aký typ kolekcie máme zvoliť pri mapovaní. Keďže kniha nezvykne obsahovať duplicitné kapitoly, stačí použiť množinu `Set`.

Rovnako musíme poopraviť definíciu mapovania. Keďže ukladáme kapitolové objekty (a nie základné dátové typy), namiesto `@CollectionOfElements` budeme používať asociáciu 1:M, ktorej zodpovedá anotácia `@OneToMany`.

```
@Entity
public class Book {
  //..

  // množina kapitol
  @OneToMany
  private Set<Chapter> chapters = new HashSet<Chapters>();
```

```
  //...
}
```

Keďže ku kapitolám budeme pristupovať sekvenčne, je rozumné zoradiť ich podľa atribútu `chapterIndex`. Najjednoduchšou a najefektívnejšou cestou je prenechať triedenie na databázu, čo môžeme dosiahnuť dodaním anotácie `@OrderBy`, kde navyše uvedieme meno atribútu objektu (nie stĺpca!), podľa ktorého bude kolekcia utriedená.

```
@OneToMany
@OrderBy("chapterIndex")
private Set<Chapter> chapters = new HashSet<Chapters>();
```

Ak by sme chceli pristupovať ku kolekcii kapitol aj pomocou indexu (napr. získať desiatu kapitolu), množina `Set` by už nevyhovovala. Namiesto neho je v tomto prípade vhodnejší zoznam `List`. Celočíselné indexy takéhoto zoznamu budú potom ukladané v samostatnom stĺpci tabuľky pre kapitoly.

```
@OneToMany
@OrderBy("chapterIndex")
private List<Chapter> chapters = new ArrayList<Chapter>();
```

5.1.1 *Inicializácia podľa potreby a prístupy k načítavaniu asociácií*

Pre asociáciu 1:M môžeme, tak ako v ostatných prípadoch, nakonfigurovať *lazy* inicializáciu.

```
@OneToMany(fetch=FetchType.EAGER)
@OrderBy("chapterIndex")
private List<Chapter> chapters = new ArrayList<Chapter>();
```

V JPQL môžeme použiť na načítanie asociovaných objektov klauzulu `join fetch`, ktorá je tiež užitočná v prípade, keď chceme vynútiť inicializáciu asociovanej *lazy* kolekcie.

```
Query query = em.createQuery(
  "select book from Book book join fetch book.chapters " +
  "where book.isbn = ?");
query.setParameter(0, isbn);

Book book = (Book) query.getSingleResult();
return book;
```

5.1.2 *Kaskádne vykonávanie operácií na asociovaných kolekciách*

Kaskádovaniu operácií na kolekciách sme sa v prípade kolekcií jednoduchých typov nevenovali, pretože to nemalo zmysel (napr. reťazec sa do databázy ukladá, resp. aktualizuje automaticky, tak ako akýkoľvek iný jednoduchý atribút). Podobne ako v prípade ostatných typov asociácií môžeme špecifikovať operácie, ktoré sa majú propagovať na asociované objekty, resp. na prvky v asociovanej kolekcii. Inak povedané, môžeme nastaviť, že príslušná operácia (napr. vkladanie) na danom objekte sa bude vykonávať aj na objektoch v rámci kolekcie.

Do anotácie `@OneToMany` vieme dodať atribút `cascade`, v ktorom vymenujeme operácie, ktoré sa majú propagovať.

```
@OneToMany(cascade={CascadeType.MERGE, CascadeType.PERSIST})
@OrderBy("chapterIndex")
private Set<Chapter> chapters = new HashSet<Chapter>();
```

Na uloženie zložitej štruktúry objektov potom stačí jediné zavolanie metódy `persist()`. Prezrime si vygenerované výpisy a prezreli si SQL dopyty, ktoré boli vygenerované:

```
insert into Book (isbn, name, price, publishDate)
values (?, ?, ?, ?)

insert into Chapter (chapterIndex, numOfPages, title)
values (?, ?, ?)

insert into Chapter (chapterIndex, numOfPages, title)
values (?, ?, ?)

insert into Book_Chapter (Book_id, chapters_id)
values (?, ?)

insert into Book_Chapter (Book_id, chapters_id)
values (?, ?)
```

5.2 Obojsmerné asociácie 1:N / M:1

Predstavme si situáciu, že do nášho kníhkupectva budeme chcieť dopracovať stránku, ktorá bude zobrazovať podrobné informácie o kapitole knihy. Je asi jasné, že budeme potrebovať vedieť, ku ktorej bude zobrazovaná kapitola patriť.

To dosiahneme pridaním inštančnej premennej `book` (odkazujúcej sa na knihu) v triede kapitoly `Chapter`, čím zároveň vytvoríme asociáciu typu M:1. Získali sme teda už dve asociácie medzi knihou a kapitolou: kniha-kapitoly (1:N) a kapitola-kniha (M:1). Tie predstavujú dohromady jednu obojsmernú asociáciu.

```
@Entity
public class Chapter {

  // ...
  @ManyToOne
  private Book book;

  // ...
}
```

Ak vytvoríme novú inštanciu knihy s dvoma kapitolami, uložíme ju do databázy a prezrieme si vygenerované SQL dopyty, uvidíme analogický výpis ako v predošlom prípade.

5.3 Asociácia M:N

Posledným typom asociácie, ktorý vysvetlíme, je typ M:N (*many-to-many*). Spomeňme si na príklad zákazníka a jeho adresy z časti opisujúcej asociáciu 1:1. Rozšírme tento príklad tak, aby demonštroval použitie asociácie M:N.

Niektorí zákazníci môžu mať viacero adries (napr. domácu, pracovnú, adresu prechodného a trvalého pobytu atď.) A naopak, zamestnanci jednej spoločnosti budú mať zrejme rovnakú pracovnú adresu. Na mapovanie takéhoto vzťahu môžeme použiť anotáciu `@ManyToMany`. Definícia asociácie M:N pomocou tejto anotácie je veľmi podobná definícii `@OneToMany`. Pre reprezentáciu takejto asociácie bude vytvorená medzitabuľka, ktorá bude obsahovať cudzie kľúče pre obe asociované strany.

```
@Entity
public class Customer {
  //...

  @ManyToMany(cascade={CascadeType.PERSIST, CascadeType.MERGE})
  private Set<Address> addresses = new HashSet<Address>();

  // gettre a settre
}
```

Tým sme vybavili (zatiaľ jednosmernú) asociáciu od zákazníka k adrese. Ak chceme túto asociáciu povýšiť na obojsmernú, pridáme opačné definície k mapovaniu objektu adresy.

```
public class Address {
  @ManyToMany
  private Set<Customer> customers = new HashSet<Customer>();

```

```
  // gettre a settre
}
```

Ak sa však pokúsite uložiť do databázy komplexnú štruktúru využívajúcu uvedené mapovanie, stretnete sa s chybovým hlásením. Dôvodom je to, že správca entít sa pokúsi uložiť obe strany asociácie nezávisle. Pri zákazníkovi vloží do tabuľky `CUSTOMER_ADDRESS` niekoľko riadkov. Tie isté riadky sa však pokúsi vložiť aj v prípade ukladania adresy, čo spôsobí porušenie obmedzenia na jedinečnosť.

Ak chceme zabrániť duplicitnému ukladaniu asociácie, musíme opäť označiť jednu zo strán ako inverznú. Ako už bolo spomenuté vyššie, inverzná asociácia sa bude pri ukladaní objektu ignorovať (Hibernate totiž bude predpokladať, že sa uloží v rámci opačného konca). Inak povedané, dve jednosmerné asociácie splynú do jedinej obojsmernej. To dosiahneme uvedením atribútu `mappedBy` s označením názvu inštančnej premennej na opačnej strane asociácie, ktorá obsahuje anotáciu.

```
public class Address {
  @ManyToMany(mappedBy="addresses")
  private Set<Customer> customers = new HashSet<Customer>();

  // gettre a settre
}
```

6 Mapovanie komponentov

6.1 Čo sú komponenty a ich používanie

V našom internetovom kníkupectve si môže zákazník vystaviť objednávku na nákup niekoľkých kníh. Zamestnanci jeho objednávku spracujú a požadovaný tovar odošlú na zadanú adresu. Zákazník môže špecifikovať viacero kontaktných údajov (napr. inú adresu doručenia v prípade pracovného dňa a inú v prípade pracovného voľna). Dodajme do aplikácie novú perzistentnú triedu pre objednávku `Order` s príslušnými anotáciami.

```
@Entity
public class Order {
  @Id
  @GeneratedValue
  private Long id;

  private Book book;       //kniha
  private Customer customer; //zákazník
  private String weekdayRecipient; //adresát v pracovný deň
  private String weekdayPhone; //telefón v pracovný deň
```

```
  private String weekdayAddress; //adresa v pracovný deň
  private String holidayRecipient; //adresát cez víkendy a sviatky
  private String holidayPhone; //telefón cez víkendy a sviatky
  private String holidayAddress; //adresa cez víkendy a sviatky

  // gettre a settre
}
```

Pokročilým programátorom by sa mohlo zdať, že trieda `Order` nie je navrhnutá práve optimálne, pretože sa môže ľahko stať, že údaje pre pracovné dni a sviatky sa budú uvádzať duplicitne (ak má zákazník rovnaké kontaktné údaje pre všedný deň i sviatok). Z hľadiska objektovo orientovaného programovania bude lepšie vytvoriť samostatnú triedu `Contact`. Objednávka potom bude obsahovať dva kontakty – jeden pre pracovný a jeden pre všedný deň.

```
public class Contact {
  private String recipient; //adresát
  private String phone; //telefón
  private String address; //adresa

  // gettre a settre
}

public class Order {
  ...
  private Contact weekdayContact; //kontakt cez všedný deň
  private Contact holidayContact; //kontakt cez víkend

  // gettre a settre
}
```

Týmto sme dokončili úpravy v objektovom modeli. Teraz prejdeme k úpravam definície mapovaní. Na základe postupov uvedených v minulých dieloch môžeme špecifikovať kontakt ako novú perzistentnú triedu a použiť asociáciu 1:1 (pomocou anotácie `@OneToOne` medzi objednávkou `Order` a kontaktom `Contact`.

Mapovanie pre kontakt:

```
@Entity
public class Contact {
  @Id
  @GeneratedValue
  private Long id;

  private String recipient; //adresát
  private String phone; //telefón
```

```
  private String address; //adresa

  // gettre a settre
}
```

Mapovanie pre objednávku:

```
@Entity
public class Order {
  ...
  @OneToOne
  private Contact weekdayContact; //kontakt cez všedný deň

  @OneToOne
  private Contact holidayContact; //kontakt cez víkend
  // gettre a settre
}
```

V tomto prípade sa však ukáže, že modelovanie kontaktu ako samostatnej perzistentnej triedy nemusí byť veľmi výhodné. Osamotená inštancia kontaktu totiž nemá veľký zmysel, ak nevieme, ku ktorej objednávke patrí. Hlavným účelom triedy pre kontakty je len logické zoskupovanie niekoľkých hodnôt. Je teda zbytočné narábať s ňou ako so samostatnou perzistentnou triedu majúcou identifikátor.

Triedy takéhoto druhu sa v JPA nazývajú *komponentami*. Na ich mapovanie sa používa anotácia `@Embeddable`. Najprv upravíme triedu `Contact`:

```
@Embeddable
public class Contact {
  private String recipient; //adresát
  private String phone; //telefón
  private String address; //adresa

  // gettre a settre
}
```

Následne dodáme anotácie do kontaktu:

```
@Entity
public class Order {
  ...
  @Embedded
  private Contact weekdayContact; //kontakt cez všedný deň

  @Embedded
  private Contact holidayContact; //kontakt cez víkend
```

```
  // gettre a settre
}
```

Všetky stĺpce namapované v komponentoch sa nachádzajú v rovnakej tabuľke ako rodičovský objekt `Order`. Komponenty nemajú identifikátor a existujú len v prípade, ak existujú ich rodičia. Ich hlavným účelom je zoskupovanie niekoľkých atribútov v samostatnom objekte. V tomto prípade sme zoskupili dáta z troch a troch stĺpcov do dvoch komponentov.

6.2 Vnorené komponenty

Komponenty je možné vnárať, čiže je možné vytvárať komponenty v komponentoch. Atribúty týkajúce sa telefónneho čísla môžeme vytiahnuť do samostatnej triedy a vložiť ju do komponentu pre kontakt.

```
@Embeddable
public class Phone {
  private String areaCode; //predvol'ba
  private String telNo; //telefón

  // gettre a settre
}

@Embeddable
public class Contact {
  /* telefón je odteraz komponentom
  private String phone;
  */
  @Embedded
  private Phone phone;

  // gettre a settre
}
```

6.3 Odkazovanie sa na iné triedy v komponentoch

6.3.1 *Odkaz na rodiča*

Ak chceme evidovať v komponente odkaz na jeho rodiča, môžeme použiť neštandardnú anotáciu `@org.hibernate.annotations.Parent`.

```
public class Contact {
  @Parent
  private Order order; //odkaz na rodiča
```

```
  private String recipient;
  private Phone phone;
  private String address;

  // gettre a settre
}
```

6.3.2 *Asociácie medzi komponentami a triedami*

Komponent môže zoskupovať nielen základné atribúty, ale i asociácie M:1 a 1:N. Predpokladajme, že budeme chcieť asociovať adresu z objednávky s adresou, ktorá sa nachádza v samostatnej tabuľke. Použijeme na to anotáciu `@ManyToOne`.

```
public class Contact {
  private Order order;
  private String recipient;
  private Phone phone;
  /* adresa je odteraz reprezentovaná ako objekt
  private String address;
  */
  @ManyToOne
  private Address address; //adresa

  // gettre a settre
}
```

6.4 Kolekcie komponentov

Predpokladajme, že chceme náš systém objednávok vylepšiť ešte viac. Dajme zákazníkovi možnosť špecifikovať viacero kontaktných údajov pre prípad, že nie je jasné, ktorý z nich bude v čase odoslania objednávky správny. Zamestnanci sa pokúsia odoslať zásielku postupne na každý uvedený kontakt, až kým nebudú úspešní. Kontakty budeme udržiavať v množine `java.util.Set`.

```
public class Order {
  ...
  /* kontakty odsunieme do množiny
  private Contact weekdayContact;
  private Contact holidayContact;
  */
  private Set<Contact> contacts = new HashSet<Contact>();

  // gettre a settre
```

```
}
```

Na mapovanie množiny komponentov môžeme použiť opäť neštandardnú anotáciu z Hibernate `@CollectionOfElements`. Pre jednoduchosť budeme používať predošlú verziu triedy `Contact`:

```
@Embeddable
public class Contact {
  private String recipient;
  private String phone;
  private String address;

  // gettre a settre

}
```

Mapovanie bude vyzerať nasledovne:

```
public class Order {
  //...
  @CollectionOfElements
  private Set<Contact> contacts = new HashSet<Contact>();

  // gettre a settre
}
```

Okrem kolekcií komponentov môžeme používať aj kolekciu vnorených komponentov a prípadných asociácií.

Triedu `Contact` a príslušné anotácie môžeme potom upraviť nasledovne:

```
@Embeddable
public class Contact {
  private String recipient;

  @Embedded
  private Phone phone;

  @Embedded
  private Address address;

  // gettre a settre
}
```

7 Mapovanie dedičnosti

7.1 Dedičnosť a polymorfizmus

Skúsme si rozšíriť naše internetové knihkupectvo o predaj CDčiek a DVDčiek. Vytvoríme najprv triedu pre disk (`Disc`) a príslušné anotácie.

```
@Entity
public class Disc {
  @Id
  @GeneratedValue
  private Long id;

  private String name;
  private Integer price;

  // gettre a settre
}
```

7.1.1 *Dedičnosť*

V obchode budeme predávať dva druhy diskov: audio (CDčka) a video (DVD). Každý druh disku má atribúty, ktoré prináležia len jemu (napr. CDčko má počet skladieb a DVD má režiséra). Z hľadiska objektovo-orientovaného návrhu by sme mohli namodelovať CD a DVD ako dve samostatné podtriedy disku, čím zaprezentujeme hierarchiu „is-a“ („je“).

Vzťah medzi podtriedou (t. j. CDčkom, resp. DVDčkom) k rodičovskej triede (t. j. disku) sa nazýva *dedičnosť* (*inheritance*). Hierarchia rodičovských tried a podtried sa nazýva *hierarchia tried* (*class hierarchy*). Pripomeňme, že v Jave dosiahneme dedičnosť použitím kľúčového slova `extends`. Trieda `Disc` sa nazýva *nadtriedou* (*superclass*) alebo *rodičovskou triedou* (*superclass*) tried `AudioDisc` (CD) a `VideoDisc` (DVD).

```
public class AudioDisc extends Disc {
  private String singer; //spevák
  private Integer numOfSongs; //počet skladieb

  // gettre a settre
}

public class VideoDisc extends Disc {
  private String director; //režisér
  private String language; //jazyk
```

```
  // gettre a settre
}
```

V relačnom modeli však nejestvuje pojem dedičnosti. To znamená, že na ukladanie vzťahov založených na dedičnosti musíme použiť niektorý z mapovacích mechanizmov. Hibernate tento proces do značnej miery uľahčuje a dáva k dispozícii viacero prístupov, ktorými je ho možné dosiahnuť.

7.1.2 *Polymorfizmus*

Na nájdenie všetkých diskov v databáze (či už CD alebo DVD) môžeme použiť nižšie uvedený dopyt. Takýto dopyt sa nazýva *polymorfný dopyt* (*polymorphic query*).

```
EntityManager em = entityManagerFactory.createEntityManager();
try {
  Query query = em.createQuery("from Disc");
  List<Disc> discs = query.getResultList();
  return discs;
} finally {
  entityManager.close();
}
```

Zaveďme teraz podporu pre rezervovanie tovaru pred jeho kúpou. Vytvoríme triedu rezervácie `Reservation` a definovať asociáciu M:1 s triedou `Disc`. Konkrétna trieda reprezentujúca špecifický druh disku (`AudioDisc` alebo `VideoDisc`) sa zvolí až za behu. Takáto asociácia sa preto nazýva *polymorfnou* (*polymorphic association*).

```
@Entity
public class Reservation {
  @Id
  @GeneratedValue
  private Long id;

  @ManyToOne
  private Disc disc;

  private Customer customer;
  private int quantity;
}
```

7.2 Mapovanie dedičnosti

Hibernate ponúka tri rôzne prístupy k mapovaniu vzťahov založených na dedičnosti. Každý z prístupov má svoje výhody a nevýhody.

7.2.1 *Jedna tabuľka pre všetky podtriedy (Table per class hierarchy)*

Tento prístup bude založený na jedinej tabuľke, v ktorej budú uložené všetky atribúty triedy a všetkých jej podtried. V tabuľke bude špeciálny stĺpec zvaný *diskriminátor* (*discriminator*), ktorý bude slúžiť na odlíšenie skutočného typu objektu.

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
public class AudioDisc extends Disc {
  private String singer; //spevák
  private Integer numOfSongs; //počet skladieb

  // gettre a settre
}

@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
public class VideoDisc extends Disc {
  private String director; //režisér
  private String language; //jazyk

  // gettre a settre
}
```

Hlavná výhoda tohto prístupu je jej jednoduchosť a efektivita a to hlavne v prípade polymorfných dopytov a asociácii. Tabuľka totiž obsahuje dáta o všetkých atribútoch všetkých inštancií a pri dopytovaní nie je potrebné používať spojenia (*join*).

Žiaľ, veľkou nevýhodou je fakt, že žiadny z atribútov vyskytujúcich sa v triede a všetkých podtriedach nesmie mať na stĺpci obmedzenie na `null`ovosť.

7.2.2 *Tabuľka pre rodičovskú triedu + tabuľka pre každú z podtried (Table per subclass)*

Tento prístup používa ukladá rodičovskú triedu do jednej tabuľky a každú z podtried namapuje na samostatnú tabuľku. V tabuľkách podtried sa bude nachádzať cudzí kľúč odkazujúci sa na rodiča. To sa dá predstaviť ako asociácia 1:1 medzi podtriedou a rodičovskou triedou.

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class AudioDisc extends Disc {
  private String singer; //spevák
  private Integer numOfSongs; //počet skladieb

  // gettre a settre
```

```
}

@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class VideoDisc extends Disc {
  private String director; //režisér
  private String language; //jazyk

  // gettre a settre
}
```

Tento prístup nezakazuje používanie `NOT NULL` stĺpcov, ale je zase menej efektívny. V SQL dopyte vygenerovanom pri načítavaní objektu sa totiž musí použiť viacero *join*ov. Ak sa používajú polymorfné dopyty a asociácie, v SQL dopyte sa musí vzájomne poprepájať viacero tabuliek, čo môže mať vplyv na efektivitu a výkonnosť.

7.2.3 *Tabuľka pre každú triedu (*Table per concrete class*)*

Posledný prístup je založený na vytvorení tabuľky pre každú z tried vyskytujúcich sa v hierarchii. V každej z tabuliek sa bude nachádzať stĺpec pre každý atribút danej triedy – či už zvyčajný alebo zdedený. Poznamenajme, že v tomto prípade nebudeme môcť používať generovanie identifikátorov pomocou identity, keďže identifikátor musí byť jedinečný vzhľadom na všetky tabuľky tried. Teda namiesto natívneho generátora budeme používať generátor sekvencií.

Tento prístup nie je veľmi efektívny pre polymorfné dopyty a asociácie, pretože na získanie objektu je potrebné prejsť viacero tabuliek.

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public class AudioDisc extends Disc {
  private String singer; //spevák
  private Integer numOfSongs; //počet skladieb

  // gettre a settre
}

@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public class VideoDisc extends Disc {
  private String director; //režisér
  private String language; //jazyk

  // gettre a settre
}
```

8 Dopytovanie pomocou jazyka JPQL

8.1 Získavanie objektov pomocou dopytov

Ak sa prístup k databáze používame JDBC, na dopytovanie a aktualizáciu dát máme k dispozícii jedinú možnosť – jazyk SQL. V ňom pracujeme s tabuľkami, stĺpcami a záznamami. V rámci JPA vieme väčšinu aktualizačných úkonov realizovať pomocou poskytovaných tried a metód (`persist()`, `merge()` atď). Pri dopytovaní by sme si však s dostupnými metódami často nevystačili (alebo by sa dopytovanie veľmi skomplikovalo). Na tento účel je lepšie použiť dopytovací jazyk nazývaný *Java Persistence Query Language* alebo tiež JPQL[1]. Tento jazyk je značne inšpirovaný dopytovacím jazykom *Hibernate Query Language*, čo je jazyk inšpirovaný *Hibernate Query Language*[2].

JPQL je databázovo nezávislý jazyk, ktorý sa počas behu prekladá do SQL. Samotná syntax je zameraná na prácu s objektami a ich atribútmi, pričom sa snaží odtieniť používateľa od pojmov používaných v databázach. Istým spôsobom možno vnímať tento jazyk ako objektovo orientovaný variant SQL.

V predošlých častiach sme už používali niekoľko základných dopytov v tomto jazyku. Pripomeňme si príklad, v ktorom sme získali zoznam všetkých kníh. Dopyt vykonáme zavolaním metódy `list()`, ktorá vráti zoznam všetkých inštancií kníh v databáze.

```
Query q = entityManager.createQuery("select book from Book as book");
List<Book> books = q.getResultList();
```

Interfejs `Query` (dopyt) poskytuje dve metódy na získanie podmnožiny výsledku určenej počiatočným záznamom (indexovaným od nuly) a počtom záznamov, ktoré sa majú vrátiť. Tieto metódy sú užitočné, ak chceme výsledky prezentovať po stránkach.

```
Query query
  = entityManager.createQuery("select book from Book as book");
query.setFirstResult(20);
query.setMaxResults(10);

List<Book> books = query.getResultList();
```

[1] Materiály sa nevedia vcelku dohodnúť na správnej skratke. Dokumentácia k *Hibernate Entity Manager* uvádza skratku EJB-QL (keďže tento jazyk je naozaj rozšírením dopytovacieho jazyka známeho z EJB 2.0), *Hibernate Annotations* zase JPA-QL, ale samotní autori špecifikácie nedefinujú žiadny oficiálny akronym. Preto sa budeme držať skratky vytvorenej z oficiálneho názvu.

[2] Ak používame správcu entít z Hibernate, môžeme používať v rámci dopytov oba druhy jazykov. Pre zachovanie portability je však lepšie používať JPQL.

Namiesto metódy `getResultList()` možno použiť i metódu `getSingleResult()`, ktorá namiesto zoznamu vráti len jeden objekt (ak máme istotu, že výsledný zoznam bude mať len jeden prvok). Ak je výsledok prázdny, metóda vráti `null`.

V JPA-QL dopytoch môžeme používať parametre a to podobným spôsobom ako v prípade SQL dopytov.

```
Query query = entityManager.createQuery(
  "select book from Book as book where book.isbn = ?1");
query.setParameter(1, "1932394419");

Book book = (Book) query.getSingleResult();
```

V uvedenom príklade sme použili otáznik `?1` ako zástupný symbol pre prvý parameter a jeho hodnotu sme nastavili pomocou indexu (indexy parametrov v JPQL začínajú od jednotky podobne ako v JDBC). Tento prístup sa nazýva *pozičným*. Alternatívnym prístupom je využitie *pomenovaných* parametrov, ktoré sú prehľadnejšie. Navyše pomenovaný parameter sa v dopyte môže vyskytnúť na viacerých miestach.

```
Query query = entityManager.createQuery(
  "select book from Book as book where book.isbn = :isbn");
query.setParameter("isbn", "1932394419");

Book book = (Book) query.getSingleResult();
```

Pomenovaný parameter začína dvojbodkou nasledovanou názvom parametra.

V ďalších sekciách sa budeme venovať jazyku JPQL podrobnejšie. Pri skúšaní dopytov odporúčame sledovať vygenerované SQL dopyty, čo nám to pomôže pri ich optimalizovaní.

8.1.1 *Klauzula `select X from Y as X`*

Základnou klauzulou, ktorá je povinná v každom dopyte JPQL je `select-from-as`. Nasledovný JPQL dopyt získa všetky knihy, ktorých názov obsahuje `Hibernate Quickly`. Všimnite si, že `name` je názvom atribútu knihy a nie databázovým stĺpcom. `Book` je zase názvom entity (perzistentnej triedy) a `book` premennou, na ktorú sa budú postupne viazať inštancie entity.

```
select book
from Book as book
where book.name = 'Hibernate Quickly'
```

V klauzule `from` môžeme uviesť aj viacero tried. V tom prípade bude výsledok obsahovať zoznam polí objektov. V nasledovnom príklade budeme mať karteziánsky súčin všetkých kníh a všetkých vydavateľov (neuviedli sme totiž žiadnu podmienku `where`).

```
Query query = entityManager.createQuery(
  "select book, publisher from Book, Publisher publisher");
List<Object[]> results = query.getResultList();
for (Object[] objects : results) {
  System.out.print(objects[0]); //typu Book
  System.out.print(objects[1]); //typu Publisher
  System.out.println();
}
```

8.2 Dopytovanie sa na asociované objekty

V JPQL sa môžeme dopytovať aj na asociované objekty a to pomocou klauzuly `join`. Nasledovný dopyt nájde všetky knihy od vydavateľa *Manning*. Výsledkom bude zoznam dvojíc objektov: presnejšie zoznam dvojprvkových polí objektov. Každá dvojica bude obsahovať objekt knihy a objekt vydavateľa.

```
select book, publisher
from Book as book
join book.publisher as publisher
where publisher.name = 'Manning'
```

Takýmto spôsobom je možné pripájať objekty v akomkoľvek type asociácie. Môžeme tak napríklad kapitoly ku knihám (kniha a kapitola sú v asociácii 1:N), čo ukazuje nasledovný dopyt, ktorý vráti všetky knihy obsahujúce kapitolu *Hibernate Basics*. Vo výsledku bude opäť zoznam dvojprvkových polí objektov a každé pole bude obsahovať knihu a kolekciu kapitol.

```
select book, chapters
from Book book
join book.chapters as chapters
where chapter.title = 'Hibernate Basics'
```

8.2.1 *Implicitné napájanie objektov (implicit joins)*

Vo vyššie uvedených príkladoch sme na napojenie asociovaného objektu používali `join`. Takýto spôsob napájania sa nazýva *explicitným* (*explicit join*). V mnohých prípadoch však môžeme používať zjednodušený zápis, kde sa na napájaný objekt môžeme odkazovať priamo cez bodkovú notáciu. Takýto spôsob sa nazýva *implicitným napájaním* (*implicit join*).

Predošlý dopyt môžeme prepísať tak, aby používal implicitný *join*. Rozdiel bude vo výsledku. Predošlý príklad vracal dvojice kníh a vydavateľov, ale nasledovný dopyt vráti zoznam kníh. V klauzule `from` sme totiž špecifikovali len jeden objekt.

```
select book
from Book book
```

```
where book.publisher.name like '%Wiley%'
```

Pri dopytovaní sa na asociované kolekcie je treba dať pozor na to, že implicitné napájanie sa prekladá na spojenie tabuliek cez `JOIN` a to pri každom jeho výskyte. To znamená, že ak budeme implicitne pripájať danú kolekciu dvakrát, tak vo výslednom SQL dopyte sa objavia dva `JOIN`y na príslušnú tabuľku. Preto treba postupovať pri implicitných asociáciách kolekcií opatrne. Kolekcia, na ktorú sa v dopyte odkazujeme viackrát, by mala byť v dopyte asociovaná explicitne.

```
select book, chapter from Book book
join book.chapters as chapter
where chapter.title like 'Hibernate Basics'
    and chapter.numOfPages < 25
```

8.2.2 *Rôzne prístupy k napájaniu objektov*

Ak použijeme na získanie kníh a ich vydavateľov nižšie uvedený príklad, ukáže sa, že vo výsledku sa nebudú nachádzať knihy bez vydavateľa. Tento typ napájania sa nazýva *vnútorným* (*inner join*) a je štandardne používaným v prípade, ak neuvedieme explicitne žiadny iný typ.

```
select book, publisher
from Book book
join book.publisher publisher
```

Ak chceme získať všetky knihy (vrátane tých, ktoré nemajú vydavateľa), budeme musieť použiť *ľavé spojenie* (*left join*) uvedením klazuly `left join` alebo `left outer join`.

```
select book, publisher from Book book
left join book.publisher publisher
```

8.2.3 *Asociácie v JPQL dopytoch*

Ak chcete vynútiť načítavanie *lazy* asociácie, môžeme na to použiť direktívu `join fetch`. Na rozdiel od klasického `join` však `join fetch` nepridá pripájaný objekt do výsledného zoznamu.

Dopyt využívajúci `join` vráti dvojice kníh a vydavateľov.

```
select book, publisher
from Book book
join book.publisher publisher
```

Na druhej strane dopyt s `join fetch` inicializuje asociáciu medzi knihou a vydavateľom, ale vo výsledku sa zjavia len knihy:

```
select book
from Book book
join fetch book.publisher publisher
```

Predošlý dopyt je typu `inner join` a teda nevráti knihy, ktoré nemajú vydavateľa. Ak chcete pristupovať aj ku knihám bez vydavateľa, použite `left join fetch`.

```
select book
from Book book
left join fetch book.publisher publisher
```

Direktívu `join fetch` môžeme používať aj na predzískanie niektorých atribútov priamo v prvom dopyte. Môže sa totiž stať, že na získanie objektu je potrebné odoslať niekoľko SQL dopytov po sebe. Atribúty získané pomocou `join fetch` sa zahrnú už do prvého SQL dopytu.

8.3 Klauzula `where`

V JPQL môžeme používať klauzulu `where`, ktorá umožňuje odfiltrovať výsledné objekty podobným spôsobom ako v SQL. V prípade zložených podmienok môžeme používať spojky `and`, `or` a `not`.

```
select book
from Book book
where book.name like '%Hibernate%'
and book.price between 100 and 200
```

Ak chceme zistiť, či asociovaný objekt je alebo nie je `null`ový, môžeme použiť direktívu `is null` alebo `is not null`. Poznamenajme, že `null`ovosť kolekcie sa týmto spôsobom zistiť nedá.

```
select book
from Book book
where book.publisher is not null
```

V nasledovnom príklade používame spojku `in`, ktorá určuje príslušnosť prvku do kolekcie. Príklad vráti knihy, ktorých vydavateľ sa volá *Manning* alebo *OReilly*.

```
from Book book
where book.publisher.name in ('Manning', 'OReilly')
```

K dispozícii je tiež možnosť zisťovať veľkosť kolekcie a to použitím funkcie `size()`. JPA použije na zistenie veľkosti kolekcie SQL poddopyt `SELECT COUNT(...)`.

```
select book
from Book book
where size(book.chapters) > 10
```

8.4 Ďalšie použitie klauzuly `select`

V predošlých príkladoch sme sa dopytovali na celé inštancie perzistentných objektov. V prípade potreby sa však môžeme dopytovať aj na jednotlivé atribúty. Použiť na to možno klauzulu `select`. Nasledovný príklad vráti zoznam všetkých názvov kníh.

```
select book.name
from Book book
```

V klauzule `select` je možné používať aj agregačné funkcie známe z SQL: `count()`, `sum()`, `avg()`, `max()`, `min()`. Tie sa preložia na ich SQL ekvivalenty.

```
select avg(book.price)
from Book book
```

V klauzule `select` môžeme používať aj implicitné napojenia. Okrem toho máme k dispozícii kľúčové slovo `distinct`, ktoré odfiltruje duplicitné záznamy.

```
select distinct book.publisher.name
from Book book
```

Ak sa chceme dopytovať na viacero atribútov, použijeme na ich oddelenie čiarku. Výsledný zoznam bude obsahovať polia objektov. Nasledovný dopyt vráti zoznam trojprvkových polí objektov:

```
select book.isbn, book.name, book.publisher.name
from Book book
```

V klauzule `select` vieme vytvárať aj inštancie vlastných typov, ktoré budú obsahovať dáta z výsledku. Vytvorme napríklad triedu `BookSummary` so sumárom informácií o knihe: bude v nej obsiahnuté ISBN, názov knihy a názov vydavateľa. Takáto vlastná trieda musí mať definovaný konštruktor so všetkými požadovanými parametrami.

```
public class BookSummary {
  private String bookIsbn;
  private String bookName;
  private String publisherName;

  public BookSummary(String bookIsbn, String bookName,
                String publisherName) {
    this.bookIsbn = bookIsbn;
    this.bookName = bookName;
    this.publisherName = publisherName;
  }

  // gettre a settre
}
```

Túto triedu môžeme použiť v dopyte nasledovným spôsobom:

```
select
  new mo.org.cpttm.bookshop.BookSummary(book.isbn, book.name,
                                book.publisher.name)
from Book book
```

8.5 Triedenie (`order by`) a zoskupovanie (`group by`)

Výsledný zoznam je možné utriediť pomocou klauzuly `order by`. Je možné špecifikovať viacero atribútov a poradie triedenia (vzostupne `asc` či zostupne `desc`).

```
select book
from Book book
order by book.name asc, book.publishDate desc
```

V JPQL sú podporované aj klauzuly `group by` a `having`. Do SQL sa preložia priamo:

```
select book.publishDate, avg(book.price)
from Book book
group by book.publishDate
```

Alebo:

```
select book.publishDate, avg(book.price)
from Book book
group by book.publishDate
having avg(book.price) > 10
```

8.6 Poddopyty

V JPQL je možné používať aj poddopyty. Treba dať pozor na to, že neuvážené poddopyty môžu byť značne neefektívne. Vo väčšine prípadov je možné transformovať poddopyty na ekvivalentné dopyty typu `select-from-where`.

```
select expensiveBook
from Book expensiveBook
where expensiveBook.price > (
  select avg(book.price) from Book book
)
```

8.7 Pomenované dopyty

Dopyty v JPQL môžu byť uvedené v definícii mapovania pod logickým názvom, na základe ktorého je možné sa na ne odkazovať. Voláme ich potom *pomenované dopyty* (*Named Queries*). Možno ich uvádzať v príslušnej triede pomocou anotácie `@NamedQuery`. Odporúča sa zaviesť jednotný postup pre pomenovávani dopytov.

```
@NamedQuery(
name = "findBookByIsbn",
query = "select book from Book book where book.isbn = ?1")
```

Na pomenovaný dopyt sa môžeme v kóde odkázať cez metódu `getNamedQuery()` na objekte správcu entít:

```
Query query = em.createNamedQuery("findBookByIsbn");
query.setParameter(1, isbn);
```