Pracujeme s

Hibernate

Gary Mak

Obsah

1. Objektovo relačné mapovanie v JDBC 6

1.1. Inštalácia 6
1.2. Konfigurácia databázy 7
1.3. Prístup k databáze pomocou JDBC 9
1.4. Objektovo-relačné mapovanie 10

2. Základy Hibernate 14

2.1. Inštalácia a konfigurácia Hibernate 14
2.2. Konfigurácia Eclipse 15
2.3. Vytvorenie definícií mapovaní 15
2.4. Konfigurácia Hibernate 16
2.5. Získavanie a ukladanie objektov 19
2.6. Generovanie databázovej schémy pomocou Hibernate 22

3. Identifikátory objektov 24

3.1. Automaticky generované identifikátory objektov 24
3.2. Zložené (***composite***) identifikátory objektov 28

4. Asociácie typu N:1 a 1:1 33

4.1. Asociácia typu M:1 (***many-to-one***) 33
4.2. Asociácia 1:1 (***one-to-one***) 39

5. Mapovanie kolekcií 44

5.1. Mapovanie kolekcií jednoduchých typov 44
5.2. Rôzne druhy kolekcií v Hibernate 48
5.3. Triedenie kolekcií 51

6. Asociácie 1:M a M:N 53

Hibernate Tutorials
(C) Gary Mak 2006

Online: *http://www.metaarchit.com/articles.html*
Preklad: Róbert Novotný

6.1. Asociácie 1:M 53
6.2. Obojsmerné asociácie 1:N / M:1 59
6.3. Asociácia M:N 61
6.4. Používanie medzitabuľky pre asociácie 1:N 63

7. Mapovanie komponentov 64
7.1. Používanie komponentov 64
7.2. Vnorené komponenty 67
7.3. Odkazovanie sa na iné triedy v komponentoch 68
7.4. Kolekcie komponentov 70
7.5. Používanie komponentov ako kľúčov v mape 72

8. Mapovanie dedičnosti 73
8.1. Dedičnosť a polymorfizmus 73
8.2. Mapovanie dedičnosti 76

9. Dopytovací jazyk Hibernate 78
9.1. Získavanie objektov pomocou dopytov 78
9.2. Klauzula `from` 80
9.3. Dopytovanie sa na asociované objekty 80
9.4. Klauzula `where` 83
9.5. Klauzula `select` 84
9.6. Triedenie `order by` a zoskupovanie `group by` 85
9.7. Poddopyty 86
9.8. Pomenované dopyty 86

10. Kritériové dopyty 87
10.1. Používanie kritériových dopytov 87
10.2. Reštrikcie 87
10.3. Asociácie 88
10.4. Projekcie 89
10.5. Triedenie a zoskupovanie 89
10.6. Dopytovanie vzorovými objektami (***querying by example***) 90

11. Dávkové spracovanie a natívne SQL 91
11.1. Dávkové spracovanie v HQL 91

12. Dopyty s natívnym SQL 93

12.1. Pomenované SQL dopyty 95

13. Cachovanie objektov 98

13.1. Úrovne cacheovania 98
13.2. Cache druhej úrovne 99
13.3. Cacheovanie asociácii 102
13.4. Cacheovanie kolekcií 103
13.5. Cacheovanie dopytov 105

1. Objektovo relačné mapovanie v JDBC

1.1. Inštalácia

Pre potreby tohto tutoriálu budeme musieť nainštalovať niekoľko súčastí.

Inštalácia JDK

JDK je nutnou podmienkou pre spúšťanie a vytváranie programov v Jave. Zrejme ho už máte nainštalovaný a ak náhodou nie, môžete si stiahnuť poslednú verziu zo stránok Sunu (*http://java.sun.com/javase/downloads/?intcmp=1281*). Nainštalujte ju do vhodného adresára, napr. `C:\java\jdk`. Inštalácii JDK sa venuje napr. tutoriál na *http://ics.upjs.sk/~novotnyr/wiki/Java.InstalaciaJDK*.

Inštalácia vývojového prostredia Eclipse a Web Tools Platform

Eclipse je integrovaným vývojovým prostredím pre jazyk Java. Rozšírenie WTP umožňuje zjednodušiť vývoj webových a J2EE aplikácii, na čo poskytuje niekoľko nástrojov (XML editor, SQL editor a pod). Eclipse spolu s týmto rozšírením je možné si stiahnuť a nainštalovať z domovskej stránky Eclipse (*http://www.eclipse.org/downloads/*) v rámci sady *Eclipse for Java EE Developers*.

Inštalácia HSQLDB

HSQL je voľne dostupná relačná databáza s podporou SQL, ktorá je implementovaná v Jave. Stiahnuť ju možno z domovskej stránky *http://www.hsqldb.org/*. Bližšiemu popisu sa venuje napr. tutoriál na *http://ics.upjs.sk/~novotnyr/wiki/Java/HsqlDb*.

Inštalácia SQuirreL-u

SQuirrel je javovský nástroj na vizuálny návrh a prehliadanie relačných databáz. Stiahnuť si ho môžete z *http://www.squirrelsql.org/#installation*.

1.2. Konfigurácia databázy

V celom tutoriáli sa budeme venovať vývoju internetového kníhkupectva. Všetky dáta budeme ukladať do relačnej SQL databázy.

Vytvorenie inštancie databázy v HSQL

Na vytvorenie novej inštancie databázy v HSQL použijeme nasledovný príkaz. V príslušnom adresári sa vytvorí niekoľko nových súborov.

```
java -cp lib/hsqldb.jar org.hsqldb.Server
     -database.0 BookShopDB -dbname.0 BookShopDB
```

Pripojme sa k tejto databázovej inštancii pomocou SQuirreL-u. Parametre pre pripojenie sú nasledovné:

- Ovládač (JDBC Driver): HSQLDB Server
- URL adresa databázy: jdbc:hsqldb:hsql://localhost/BookShopDB
- Meno používateľa: `sa`
- Heslo: [prázdne]

Vytvorenie relačného modelu (tabuliek)

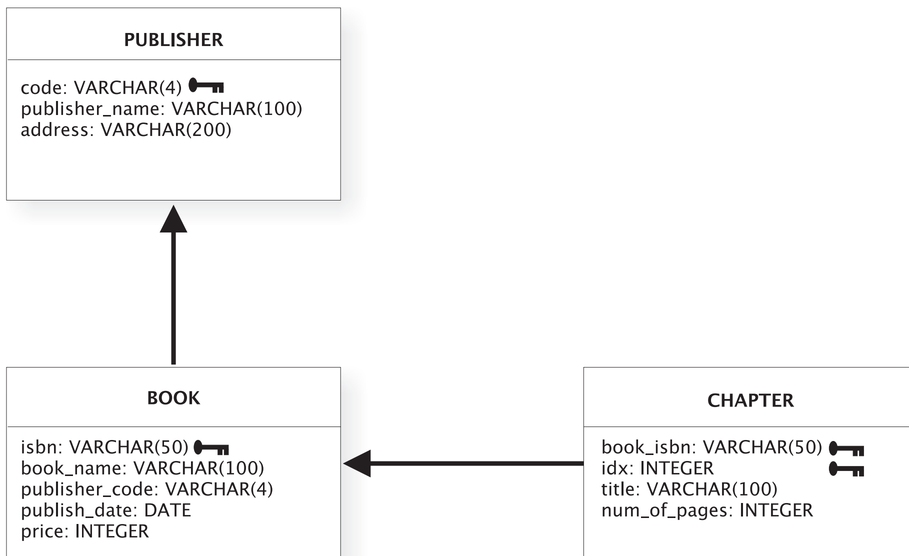
Zatiaľ je naša databázová schéma prázdna. Zrejme budeme chcieť do nej dodať niekoľko nových tabuliek. To docielime pomocou nasledovných SQL dopytov:

```
CREATE TABLE PUBLISHER (
  CODE            VARCHAR(4)    NOT NULL,
  PUBLISHER_NAME  VARCHAR(100)  NOT NULL,
  ADDRESS         VARCHAR(200),
  PRIMARY KEY (CODE)
);

CREATE TABLE BOOK (
  ISBN           VARCHAR(50)  NOT NULL,
  BOOK_NAME      VARCHAR(100) NOT NULL,
  PUBLISHER_CODE VARCHAR(4),
  PUBLISH_DATE   DATE,
  PRICE          INT,
  PRIMARY KEY (ISBN),
  FOREIGN KEY (PUBLISHER_CODE) REFERENCES PUBLISHER
);
CREATE TABLE CHAPTER (
  BOOK_ISBN    VARCHAR(50)  NOT NULL,
```

```
  IDX          INT          NOT NULL,
  TITLE        VARCHAR(100) NOT NULL,
  NUM_OF_PAGES INT,
  PRIMARY KEY (BOOK_ISBN, IDX),
  FOREIGN KEY (BOOK_ISBN) REFERENCES BOOK
);
```

Pomocou SQuirreL-u vygenerujeme z tabuliek nasledovný diagram, tzv. *relačný model (relational model)* nášho kníhkupectva.

Vložme teraz do tabuliek dáta pomocou nasledovných SQL dopytov:




```
INSERT INTO PUBLISHER (CODE, PUBLISHER_NAME, ADDRESS) VALUES
('MANN', 'Manning', 'Address for Manning');

INSERT INTO BOOK (ISBN, BOOK_NAME, PUBLISHER_CODE, PUBLISH_DATE, PRICE)
VALUES ('1932394419', 'Hibernate Quickly', 'MANN', '2005-08-01', 35);

INSERT INTO CHAPTER (BOOK_ISBN, IDX, TITLE, NUM_OF_PAGES) VALUES
('1932394419', 1, 'Why Hibernate?', 25);

INSERT INTO CHAPTER (BOOK_ISBN, IDX, TITLE, NUM_OF_PAGES) VALUES
('1932394419', 2, 'Hibernate basics', 37);
```

1.3. Prístup k databáze pomocou JDBC

Klasickým spôsobom práce s databázou v Jave je založený na využití tried tvoriacich technológiu JDBC (*Java Database Connectivity*).

Vytvorenie projektu v Eclipse

V Eclipse si vytvoríme projekt BookShop, do ktorého si pridáme knižnice JDBC ovládača pre HSQLDB. Ten sa nachádza v adresári, do ktorého sme rozbalili HSQLDB, presnejšie v podadresári `lib/hsqldb.jar`.

Inicializácia a ukončenie práce s JDBC

Prvým krokom práce s JDBC je načítanie JDBC ovládača a otvorenie spojenia s databázou. Spojenie s databázou musíme po skončení práce uzatvoriť (bez ohľadu na to, či nastala výnimka alebo nie).

```
Class.forName("org.hsqldb.jdbcDriver");
Connection connection = DriverManager.getConnection(
    "jdbc:hsqldb:hsql://localhost/BookShopDB", "sa", "");
try {
  // tu používame pripojenie na odosielanie SQL príkazov
} finally {
  connection.close();
}
```

Dopytovanie s použitím JDBC

Ak chceme napr. získať z databázy knihu, ktorej ISBN je 1932394419, môžeme na to použiť nasledovný kód:

```
PreparedStatement stmt = connection
    .prepareStatement("SELECT * FROM BOOK WHERE ISBN = ?");
stmt.setString(1, "1932394419");
ResultSet rs = stmt.executeQuery();
while (rs.next()) {
  System.out.println("ISBN : " + rs.getString("ISBN"));
  System.out.println("Book Name : " + rs.getString("BOOK_NAME"));
  System.out.println("Publisher Code : "
                          + rs.getString("PUBLISHER_CODE"));
  System.out.println("Publish Date : " + rs.getDate("PUBLISH_DATE"));
  System.out.println("Price : " + rs.getInt("PRICE"));
  System.out.println();
}
rs.close();
stmt.close();
```

Aktualizácia dát v JDBC

Zmeňme názov knihy s ISBN 1932394419 na *Hibernate Quickly 2nd Edition*. Docieliť to môžeme nasledovným fragmentom kódu:

```
PreparedStatement stmt = connection.prepareStatement(
  "UPDATE BOOK SET BOOK_NAME = ? WHERE ISBN = ?");
stmt.setString(1, "Hibernate Quickly 2nd Edition");
stmt.setString(2, "1932394419");
int count = stmt.executeUpdate();
System.out.println("Updated count : " + count);
stmt.close();
```

1.4. Objektovo-relačné mapovanie

V predošlých odsekoch sme si ukázali jednoduché príklady na prístup k relačnej databáze. Java je však objektovo-orientovaný jazyk a na reprezentáciu dôležitých prvkov nášho systému by sme mali používať objektový model.

Objektový model

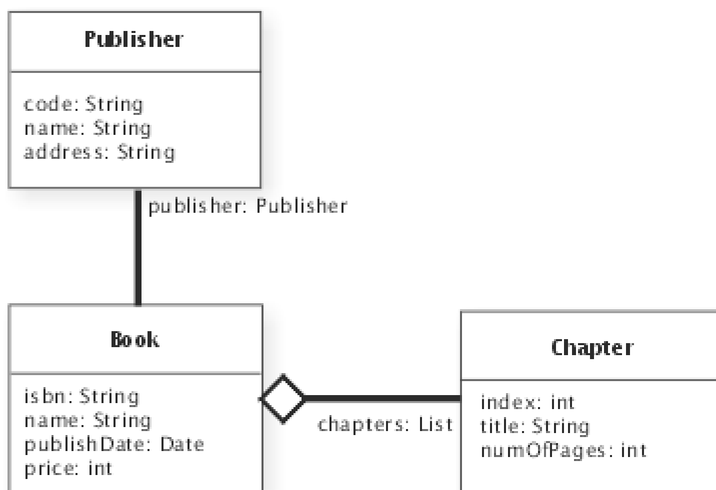
Objektový model budeme reprezentovať pomocou štandardných tried spĺňajúcich špecifikáciu JavaBeans, niekedy nazývaných POJO (*Plain Old Java Object*, jednoduchý objekt). Každá z tried musí mať implicitný (bezparametrový) konštruktor.

```
// vydavateľ
public class Publisher {
  private String code;     //kód vydavateľa
  private String name;     //názov
  private String address;  //adresa

  // gettre a settre
}

public class Book {
  private String isbn;                 //ISBN
  private String name;                 //titul
  private Publisher publisher;         //vydavateľ
  private Date publishDate;            //dátum vydania
  private int price;                   //cena
  private List<Chapter> chapters;      //zoznam kapitol

  // gettre a settre
}
```

```
public class Chapter {
  private int index;      //poradie kapitoly
  private String title;   //názov kapitoly
  private int numOfPages; //počet strán

  // gettre a settre
}
```

Tento objektový model je možné reprezentovať pomocou diagramu tried (*class diagram*) jazyka UML.

Všimnime si, že medzi objektovým a relačným modelom sú isté filozofické rozdiely. Vzťah medzi vydavateľom a knihou je v relačnom modeli reprezentovaný pomocou cudzieho kľúča v tabuľke `PUBLISHER` odkazujúceho na tabuľku `BOOK` a objektovom modeli pomocou asociácie typu M:1. Iným príkladom je kniha, ktorá má zoznam kapitol (asociácia 1:N), čo v relačnom modeli zodpovedá cudziemu kľúču v tabuľke



`CHAPTER` odkazujúcemu sa na tabuľku `KNIHA`.

Ak chceme ukladať objekty do relačnej štruktúry (resp. načítavať relačné dáta do objektov), musíme preklenúť rozpor medzi týmito dvoma prístupmi. Postupy riešiace tento problém sa súhrnne nazývajú *objektovo-relačné mapovanie* (ORM, *object/relational mapping*).

Získavanie asociovaných objektov

Predstavme si v našej aplikácii webovú stránku, ktorá obsahuje detaily o knihe – teda ISBN, názov, meno vydavateľa, adresu vydavateľa, dátum vydania a zoznam kapitol v knihe. Na získanie takýchto údajov potrebujeme ku knihe načítať objekt vydavateľa a zoznam objektov zodpovedajúcich kapitolám. Takáto skupina objektov, ktorá je navzájom poprepájaná, sa nazýva *objektový graf* (*object graph*).

```
PreparedStatement stmt = connection.prepareStatement(
  "SELECT * FROM BOOK, PUBLISHER"
  + " WHERE BOOK.PUBLISHER_CODE = PUBLISHER.CODE
  + " AND BOOK.ISBN = ?");
stmt.setString(1, isbn);
ResultSet rs = stmt.executeQuery();

Book book = new Book();

if (rs.next()) {
  book.setIsbn(rs.getString("ISBN"));
  book.setName(rs.getString("BOOK_NAME"));
  book.setPublishDate(rs.getDate("PUBLISH_DATE"));
  book.setPrice(rs.getInt("PRICE"));

  Publisher publisher = new Publisher();
  publisher.setCode(rs.getString("PUBLISHER_CODE"));
  publisher.setName(rs.getString("PUBLISHER_NAME"));
  publisher.setAddress(rs.getString("ADDRESS"));

  book.setPublisher(publisher);
}
rs.close();
stmt.close();

List chapters = new ArrayList();
stmt = connection.prepareStatement(
                 "SELECT * FROM CHAPTER WHERE BOOK_ISBN = ?");
stmt.setString(1, isbn);
rs = stmt.executeQuery();
while (rs.next()) {
  Chapter chapter = new Chapter();
  chapter.setIndex(rs.getInt("IDX"));
  chapter.setTitle(rs.getString("TITLE"));
  chapter.setNumOfPages(rs.getInt("NUM_OF_PAGES"));
  chapters.add(chapter);
}
```

```
book.setChapters(chapters);
rs.close();
stmt.close();
return book;
```

Ukladanie asociovaných objektov

Ak by sme mali stránku pomocou ktorej môžu používatelia vkladať údaje o nových knihách (vrátane informácií o vydavateľovi a kapitolách), museli by sme zabezpečiť, aby sa do databázy korektne uložili všetky objekty obsahujúce príslušné údaje.

```
PreparedStatement stmt = connection.prepareStatement(
  "INSERT INTO PUBLISHER (CODE, PUBLISHER_NAME, ADDRESS)"
  + " VALUES (?, ?, ?)");
stmt.setString(1, book.getPublisher().getCode());
stmt.setString(2, book.getPublisher().getName());
stmt.setString(3, book.getPublisher().getAddress());
stmt.executeUpdate();
stmt.close();

stmt = connection.prepareStatement(
  "INSERT INTO BOOK (ISBN, BOOK_NAME, PUBLISHER_CODE,
  + " PUBLISH_DATE, PRICE)"
  + " VALUES (?, ?, ?, ?, ?)");
stmt.setString(1, book.getIsbn());
stmt.setString(2, book.getName());
stmt.setString(3, book.getPublisher().getCode());

stmt.setDate(4, new java.sql.Date(book.getPublishDate().getTime()));
stmt.setInt(5, book.getPrice());
stmt.executeUpdate();
stmt.close();

stmt = connection.prepareStatement(
  "INSERT INTO CHAPTER (BOOK_ISBN, IDX, TITLE, NUM_OF_PAGES)"
  + "VALUES (?, ?, ?, ?)");

for (Chapter chapter : book.getChapters() {
  stmt.setString(1, book.getIsbn());
  stmt.setInt(2, chapter.getIndex());
  stmt.setString(3, chapter.getTitle());
  stmt.setInt(4, chapter.getNumOfPages());
  stmt.executeUpdate();
}
stmt.close();
```

Problémy
Toto bol jednoduchý nástrel práce s objektovo relačným mapovaním založenom na JDBC. Tento prístup je priamočiary a nevyžaduje hlbšie znalosti. V zložitejších situáciách sa však môžeme stretnúť s niekoľkými problémami:

Množstvo SQL dopytov
V prípade zložitého objektového modelu je nutné vytvoriť značný počet dopytov typu `SELECT`, `INSERT`, `UPDATE`, či `DELETE`, ktoré sa často zbytočne opakujú.

Prevody medzi riadkami a objektami
Pri načítavaní objektov musíte skopírovať údaje z riadkov výslednej množiny na atribúty objektu. A naopak, pri ukladaní objektu musíte namapovať hodnoty z atribútov na parametre v objekte pripraveného dopytu `PreparedStatement`.

Manuálne spravovanie asociácií
Pri získavaní objektového grafu musíme načítavať dáta z viacerých tabuliek, prípadne z tabuliek spojených pomocou `JOIN`u. Pri ukladaní objektov je potrebné aktualizovať údaje vo viacerých tabuľkách.

Závislosť na konkrétnej databáze
SQL dopyty vytvorené pre jeden konkrétny typ databázy nemusia pracovať s iným typom databázy. Niekedy sa totiž môže stať, že vaše tabuľky budete musieť premigrovať do iného databázového systému (i keď v praxi sa to stáva zriedka).

2. Základy Hibernate

2.1. Inštalácia a konfigurácia Hibernate

Inštalácia Hibernate

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

Je voľne dostupný na adrese *http://www.hibernate.org*. Navštívte túto adresu a stiahnite si *Hibernate Core*. Distribučný archív rozbaľte do ľubovoľného adresára, napr. `C:\java\hibernate-3.1`.

2.2. Konfigurácia Eclipse

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:

- ${Hibernate_Install_Dir}/hibernate3.jar
- ${Hibernate_Install_Dir}/lib/antlr.jar
- ${Hibernate_Install_Dir}/lib/asm.jar
- ${Hibernate_Install_Dir}/lib/asm-attrs.jar
- ${Hibernate_Install_Dir}/lib/cglib.jar
- ${Hibernate_Install_Dir}/lib/commons-collections.jar
- ${Hibernate_Install_Dir}/lib/commons-logging.jar
- ${Hibernate_Install_Dir}/lib/dom4j.jar
- ${Hibernate_Install_Dir}/lib/ehcache.jar
- ${Hibernate_Install_Dir}/lib/jta.jar
- ${Hibernate_Install_Dir}/lib/log4j.jar

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

Inštalácia XML šablón pre Hibernate

V okne *Preferences* otvorte *Web and XML | XML Files | XML Templates* a importujte šablóny XML, ktorá vám uľahčí vývoj v Hibernate. Šablóny si môžete stiahnuť z webových stránok kurzu na http://www2.cpttm.org.mo/training/cm192-06-2006-c/.

2.3. Vytvorenie definícií mapovaní

V prvom kroku budeme používať Hibernate 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. Samotné mapovanie je možné nastaviť v *de-*

finícii mapovania (*mapping definition*). Objekty, ktoré sú namapované na tabuľky, sa nazývajú *perzistentné objekty* (*persistent objects*), 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. Vytvoríme XML súbor definície mapovania `Book.hbm.xml`, ktorý sa bude nachádzať v rovnakom balíčku ako trieda `Book`.

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE hibernate-mapping PUBLIC
         "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
         "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Book" table="BOOK">
    <id name="isbn" type="string" column="ISBN" />
    <property name="name" type="string" column="BOOK_NAME" />
    <property name="publishDate" type="date" column="PUBLISH_DATE" />
    <property name="price" type="int" column="PRICE" />
  </class>
</hibernate-mapping>
```

Každý perzistentný objekt musí mať identifikátor, ktorý je používaný na jednoznačnú identifikáciu inštancií. V našom príklade sme zvolili do úlohy identifikátora ISBN.

2.4. Konfigurácia Hibernate

Predtým samotným používaním Hibernate (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).

K dispozícii sú tri spôsoby pre konfiguráciu – programová konfigurácia (priamo v Jave), konfigurácia pomocou XML súborov a konfigurácia pomocou súborov *properties*. Tu si ukážeme len prvé dva spôsoby (tretí spôsob je veľmi podobný XML konfigurácii).

Programová konfigurácia

Po nakonfigurovaní Hibernate potrebujeme vytvoriť *session factory* (továreň pre sedenia), čo je globálny objekt zodpovedný za manažovanie sedení s Hibernate. Sedenie (*session*) môžeme vnímať ako analógiu pripojenia k databáze (skutočnosť je taká, že v rámci sedenia sa na pozadí používa pripojenie k databáze).

```
Configuration configuration = new Configuration()
  .addResource("mo/org/cpttm/bookshop/Book.hbm.xml")

  .setProperty("hibernate.dialect", "org.hibernate.dialect.HSQLDialect")
  .setProperty("hibernate.connection.driver_class",
               "org.hsqldb.jdbcDriver")
  .setProperty("hibernate.connection.url",
               "jdbc:hsqldb:hsql://localhost/BookShopDB")
  .setProperty("hibernate.connection.username", "sa")
  .setProperty("hibernate.connection.password", "");

SessionFactory factory = configuration.buildSessionFactory();
```

Namiesto metódy `addResource()`, ktorou pridáme do konfigurácie mapovacie súbory môžeme tiež použiť `addClass()`, ktorou zaregistrujeme v Hibernate konkrétnu triedu. Hibernate potom dohľadá súbory s definíciami mapovania automaticky.

```
Configuration configuration = new Configuration()
  .addClass(mo.org.cpttm.bookshop.Book.class)

  .setProperty("hibernate.dialect", "org.hibernate.dialect.HSQLDialect")
  .setProperty("hibernate.connection.driver_class",
               "org.hsqldb.jdbcDriver")
  .setProperty("hibernate.connection.url",
               "jdbc:hsqldb:hsql://localhost/BookShopDB")
  .setProperty("hibernate.connection.username", "sa")
  .setProperty("hibernate.connection.password", "");

SessionFactory factory = configuration.buildSessionFactory();
```

V prípade, že má aplikácia veľké množstvo definícií mapovania, je možné ich zabaliť do JAR súboru a pridať do konfigurácie hromadne. JAR súbor sa musí nachádzať v `CLASSPATH` aplikácie a pridať ho možno metódou `addJar()`.

```
Configuration configuration = new Configuration()
  .addJar(new File("mapping.jar"))

  .setProperty("hibernate.dialect", "org.hibernate.dialect.HSQLDialect")
```

```
  .setProperty("hibernate.connection.driver_class",
               "org.hsqldb.jdbcDriver")
  .setProperty("hibernate.connection.url",
               "jdbc:hsqldb:hsql://localhost/BookShopDB")
  .setProperty("hibernate.connection.username", "sa")
  .setProperty("hibernate.connection.password", "");

SessionFactory factory = configuration.buildSessionFactory();
```

Konfigurácia pomocou XML

Alternatívnou možnosťou konfigurácie Hibernate je použitie XML súboru. Vytvoríme súbor `hibernate.cfg.xml` v koreňovom adresári zdrojových súborov (typicky adresár `src`; čiže na vrchole hierarchie balíčkov), čím zabezpečíme, že ho Eclipse skopíruje do koreňového adresára v `CLASSPATH`.

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE hibernate-configuration PUBLIC
 "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
 "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
    <property name="connection.driver_class">
        org.hsqldb.jdbcDriver
    </property>
    <property name="connection.url">
      jdbc:hsqldb:hsql://localhost/BookShopDB
    </property>
    <property name="connection.username">sa</property>
    <property name="connection.password"></property>
    <property name="dialect">
      org.hibernate.dialect.HSQLDialect
    </property>

    <mapping resource="mo/org/cpttm/bookshop/Book.hbm.xml" />
  </session-factory>
</hibernate-configuration>
```

Kód pre vybudovanie *session factory* sa potom zjednoduší. Nasledovný príklad načíta a nakonfiguruje továreň zo súboru `hibernate.cfg.xml` v `CLASSPATH`.

```
Configuration configuration = new Configuration().configure();
SessionFactory factory = configuration.buildSessionFactory();
```

2.5. Získavanie a ukladanie objektov

Otváranie a uzatváranie sedení

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

- nakonfigurovať *session factory* (stačí raz, pred prvým otvorením sedenia)
- otvoriť sedenie,
- vykonať požadované operácie,
- upratať po sebe, čiže uzatvoriť sedenie.

Kostra kódu vyzerá nasledovne:

```
Session session = factory.openSession();
try {
  // tu používame sedenie session na prácu s objektami
} finally {
  session.close();
}
```

Načítavanie objektov

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

```
Book book = (Book) session.load(Book.class, isbn);
```

Alebo takto:

```
Book book = (Book) session.get(Book.class, isbn);
```

Aký je rozdiel medzi `load()` a `get()`? Prvá odlišnosť sa prejaví v prípade, že v databáze sa nenachádza objekt s daným identifikátorom. Metóda `load()` vyhodí výnimku `org.hibernate.ObjectNotFoundException`, pričom `get()` vráti `null`.

Druhý spočíva v návratovom objekte – `load()` vráti proxy objekt, ktorý pristúpi k databáze až pri prvom vyvolaní niektorej z jeho metód. Naproti tomu použitie `get()` k nej pristúpi hneď.

Na dopytovanie klasickej databázy je všeobecne používaný jazyk SQL. V prípade Hibernate je k dispozícii analogický jazyk (ibaže sa dopytujeme na objekty a nie na riadky tabuľky) nazývaný *Hibernate Query Language* (HQL). Príkladom použitia HQL je nasledovný dopyt, ktorý vráti zoznam všetkých kníh v databáze:

```
Query query = session.createQuery("from Book");
List books = query.list();
```

Ak ste si istí, že výsledným zoznam obsahuje len jeden objekt, môžete použiť metódu `uniqueResult()`, ktorá vráti namiesto zoznamu danú inštanciu.

```
Query query = session.createQuery("from Book where isbn = ?");
query.setString(0, isbn);
Book book = (Book) query.uniqueResult();
```

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

Zobrazovanie SQL dopytov odosielaných Hibernate-om

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 konfiguračnom súbore XML parameter `show_sql` na `true`, Hibernate bude vypisovať SQL dopyty na štandardný výstup.

```
<property name="show_sql">true</property>
```

Na zaznamenávanie SQL príkazov a parametrov môžeme tiež použiť štandardnú knižnicu pre ladiace výpisy (*logovanie*) log4j. Vytvorme konfiguračný súbor `log4j.properties` v koreňovom adresári hierarchie balíčkov. Tento súbor je používaný na nastavenie parametrov log4j.

```
### 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
```

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.

```
Session session = factory.openSession();
Transaction tx = null;
try {
  tx = session.beginTransaction();
  // ...
  // tu používame príkazy pracujúce so sedením
  // ...

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

Ak náhodou nechcete vo svojej aplikácii používať transakcie, musíte nastaviť hodnotu konfiguračnej premennej `autocommit` na `true` (štandardná hodnota je `false`). V tom prípade sa každý aktualizačný dopyt odošle do databázy ihneď.

```
<property name="connection.autocommit">true</property>
```

Dôležitou vecou, ktorú si treba uvedomiť pri zapnutom automatickom potvrdzovaní (`autocommit = true`) je nutnosť automaticky synchronizovať stav objektov modifikovaných v rámci sedenia so stavom dát v databáze (*flush*núť) pred ukončením sedenia. Dôvodom je to, že Hibernate nerealizuje zmeny v databáze ihneď, ale kvôli vyššej výkonnosti ich zvyčajne vykonáva po skupinách.

```
session.flush();
```

Ukladanie objektov do databázy

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

```
session.save(book);
```

Ak chceme aktualizovať objekt, ktorý sa už v databáze nachádza, zavoláme metódu `update()`. Hibernate vykoná v databáze SQL príkaz `UPDATE`.

```
session.update(book);
```

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

```
session.delete(book);
```

2.6. Generovanie databázovej schémy pomocou Hibernate

V predošlom príklade sme databázové tabuľky vytvorili pred navrhnutím objektového modelu. Tento prístup je v prípade používania klasických metód na mapovanie tabuliek na objekty zvyčajným, 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.

Vytvorenie zostavovacieho súboru pre Ant

Na zjednodušenie procesu generovania databázovej schémy môžeme použiť nástroj Apache Ant.[1]

Vytvorme zostavovací súbor `build.xml` v koreňovom adresári projektu.

```
<project name="BookShop" default="schemaexport">
  <property name="build.dir" value="bin" />
  <property name="hibernate.home" value="C:/java/hibernate-3.1" />
  <property name="hsqldb.home" value="c:/hsqldb" />

  <path id="hibernate-classpath">
    <fileset dir="${hibernate.home}">
      <include name="**/*.jar" />
    </fileset>

    <fileset dir="${hsqldb.home}">
      <include name="lib/*.jar" />
    </fileset>

    <pathelement path="${build.dir}" />
  </path>

  <!--Tu definujeme antovské ciele -->
</project>
```

1 Bližšie informácie o tomto nástroji je možné nájsť na *http://ant.apache.org*, prípadne na *http://ics.upjs.sk/~novotnyr/wiki/Java/ZacinameSAntom*

Generovanie databázovej schémy pomocou `SchemaExport`

Na vygenerovanie príkazov v SQL, pomocou ktorých je možné vytvoriť databázové tabuľky použijeme externý antovský *task* `schemaexport`, ktorý nám Hibernate dá k dispozícii. Tento *task* použije konfiguračný parameter `dialect` z konfiguračného súboru `hibernate.cfg.xml` na určenie výrobcu a verzie používanej databázy.

```
<target name="schemaexport">
  <taskdef name="schemaexport"
           classname="org.hibernate.tool.hbm2ddl.SchemaExportTask"
           classpathref="hibernate-classpath" />

  <schemaexport config="${build.dir}/hibernate.cfg.xml"
                output="BookShop.sql" />
</target>
```

Aktualizácia databázovej schémy pomocou `SchemaUpdate`

Počas vývoja aplikácie sa určite nevyhneme zmenám v objektovom modeli (pridávaniu a odoberaniu tried alebo ich inštančných premenných a pod.) Základným prístúpom môže byť zrušenie zastaralej databázovej schémy a jej znovuvybudovanie vychádzajúce z novej verzie tried. To však nemusí byť efektívne. Na tento účel je dispozícii ďalší externý *task* `schemaupdate`, pomocou ktorého môžeme aktualizovať existujúcu databázovú schému.

```
<target name="schemaupdate">
  <taskdef name="schemaupdate"
           classname="org.hibernate.tool.hbm2ddl.SchemaUpdateTask"
           classpathref="hibernate-classpath" />
  <schemaupdate config="${build.dir}/hibernate.cfg.xml" text="no"/>
</target>
```

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:

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Book" table="BOOK">
    <id name="isbn" type="string">
      <column name="ISBN" length="50" />
    </id>

    <property name="name" type="string">
```

```
      <column name="BOOK_NAME" length="100" not-null="true" />
    </property>

    <property name="publishDate" type="date" column="PUBLISH_DATE" />
    <property name="price" type="int" column="PRICE" />
  </class>
</hibernate-mapping>
```

3. Identifikátory objektov

3.1. Automaticky generované identifikátory objektov

V poslednom príklade z predošlej kapitoly 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*).

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 atribút id, ktorého hodnoty budú generované automaticky:

```
public class Book {
  private long id;
  private String isbn;
  private String name;
  private Publisher publisher;
  private Date publishDate;
  private int price;
  private List chapters;

  // gettery a settery
}
```

Generovanie identifikátorov v Hibernate

V ďalšom kroku potrebujeme inštruovať Hibernate, aby pred uložením objektu do databázy vygeneroval pre daný objekt nový identifikátor. K dispozícii máme viacero spôsobov generovania; niektoré sú všeobecné a niektoré prispôsobené konkrétnym databázam.

Typickým prístupom je použitie automaticky inkrementovaných sekvencií implementovaných buď pomocou sekvencií (vrátane HSQLDB) alebo automatických generátorov (MySQL). Tento prístup je označovaný ako sequence. Okrem toho musíme upraviť definíciu stĺpca ISBN – zmeníme ho na jednoduchý atribút a pridáme doň obmedzenie na jedinečnosť a nenullovosť.

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Book" table="BOOK">

    <id name="id" type="long" column="ID">
      <generator class="sequence">
        <param name="sequence">BOOK_SEQUENCE</param>
      </generator>
    </id>

    <property name="isbn" type="string">
      <column name="ISBN" length="50" not-null="true" unique="true" />
    </property>

    <property name="name" type="string">
      <column name="BOOK_NAME" length="100" not-null="true" />
    </property>

```

```
    <property name="publishDate" type="date" column="PUBLISH_DATE" />

    <property name="price" type="int" column="PRICE" />
  </class>
</hibernate-mapping>
```

Iným prístupom je použitie identifikačného stĺpca (*identity column*) tabuľky. Tento prístup je označovaný ako *identity*.

```
<id name="id" type="long" column="ID">
  <generator class="identity" />
</id>
```

Alternatívne je možné sa pri výbere prístupu spoľahnúť na Hibernate, ktorý vyberie prístup podľa používanej databázy — označujeme ho ako *native* (*natívny*).

```
<id name="id" type="long" column="ID">
  <generator class="native"/>
</id>
```

Ukladanie objektov pomocou `saveOrUpdate()`

Popri už spomínaných metódach `save()` a `update()` poskytuje Hibernate užitočnú metódu `saveOrUpdate()`. Táto metóda automaticky zistí, či má byť objekt v parametri uložený (ak ešte v databáze nejestvuje) alebo aktualizovaný (ak už existuje). Táto metóda je veľmi užitočná v prípade tranzitívnej perzistencie, o ktorej sa zmieníme neskôr.

```
session.saveOrUpdate(book);
```

Majme perzistentný objekt, v ktorom sa používa automaticky generovaný identifikátor, a hodnota identifikátorového atribútu je prázdna. Ak takýto objekt odovzdáme metóde `saveOrUpdate()`), Hibernate najprv automaticky vygeneruje hodnotu pre identifikátor a následne vykoná SQL príkaz `INSERT`. V prípade, že hodnota identifikátorového atribútu nie je prázdna, Hibernate bude považovať tento objekt za už existujúci v databáze a odošle databáze SQL príkaz `UPDATE`.

Ako však Hibernate zistí, či hodnota tohto atribútu je prázdna? V prípade našej knižnej triedy `Book` používame pre identifikátorový atribút inštančnú premennú typu `long`. Mali by sme však špecifikovať, aká hodnota atribútu bude indikovať prázdnosť – zrejme je prirodzené, aby ňou bola hodnota 0. V tomto prípade samozrejme nebudeme môcť mať objekt, ktorý má hodnotu identifikátora rovnú 0.

```
<id name="id" type="long" column="ID" unsaved-value="0">
  <generator class="native"/>
</id>
```

Iným riešením problému s „prázdnosťou" je použitie triedy `java.lang.Long` namiesto primitívneho typu `long`. Výhodou objektu typu `Long` je možnosť používať hodnotu `null` ako indikátor nedefinovaného identifikátora, a pre hodnotu identifikátora budeme môcť používať čísla z celého rozsahu typu `long`.

Musíme však upraviť našu triedu:

```
public class Book {
  private Long id; //Long namiesto long
  private String isbn;
  private String name;
  private Publisher publisher;
  private Date publishDate;
  private int price;
  private List chapters;
  // gettre a settre
}
```

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 Long id;
  private String isbn;
  private String name;
  private Publisher publisher;
  private Date publishDate;
  private Integer price; //Integer namiesto int
  private List chapters;
  // gettre a settre
}
```

3.2. 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. Každý stĺpec namapujeme na reťazcový atribút:

```
public class Customer {
  private String countryCode;
  private String idCardNo;
  private String firstName;
  private String lastName;
  private String address;
  private String email;

  // gettre a settre
}
```

Následne vytvorme definíciu mapovaní pre túto triedu. Pre zložený identifikátor (zodpovedajúci zloženému primárnemu kľúču) budeme používať element `<composite-id>`, v ktorom špecifikujeme stĺpce (kód krajiny `countryCode` a číslo karty `idCardNo`), ktoré ho budú tvoriť.

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Customer" table="CUSTOMER">
    <composite-id>
      <key-property name="countryCode" type="string"
                    column="COUNTRY_CODE" />
      <key-property name="idCardNo" type="string" column="ID_CARD_NO"/>
    </composite-id>
```

```
    <property name="firstName" type="string" column="FIRST_NAME" />
    <property name="lastName" type="string" column="LAST_NAME" />
    <property name="address" type="string" column="ADDRESS" />
    <property name="email" type="string" column="EMAIL" />
  </class>
</hibernate-mapping>
```

Keďže sme vytvorili definíciu novej perzistentnej triedy, nesmieme ju zabudnúť dodať do konfiguračného súboru `hibernate.cfg.xml`.

```
<mapping resource="mo/org/cpttm/bookshop/Customer.hbm.xml" />
```

Keď budeme používať metódy `load())`, resp. `get()`, vyvstane zrejme otázka, čomu bude zodpovedať objekt identifikátora (v prípade klasických číselných identifikátorov by sme použili jednoducho pevnú hodnotu). Aký dátový typ má mať identifikátor?

Riešením je vytvorenie nového objektu typu `Customer`, ktorému nastavíme požadované hodnoty `countryCode` a `idCardNo`. Musíme si však dať pozor na to, že trieda reprezentujúca identifikátor objektu musí byť serializovateľná (teda implementovať interfejs `java.io.Serializable`).

```
public class Customer implements Serializable {
...
}
CustomerId customerId = new Customer();
customerId.setCountryCode("mo");
customerId.setIdCardNo("1234567(8)");
Customer customer = (Customer) session.get(Customer.class, customerId);
```

Mohlo by sa zdať neprirodzené, resp. mätúce, že ako identifikátor používame celý perzistentný objekt (na získanie objektu zákazníka potrebujeme dodať objekt zákazníka). Lepším prístupom je vyňatie atribútov tvoriacich identifikátor objektu do samostatnej triedy `CustomerId`.

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

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

Následne upravíme perzistentnú triedu zákazníka `Customer` tak, aby využívala uvedenú pomocnú identifikátorovú triedu

```
public class Customer implements Serializable {

  private CustomerId id;

  private String firstName;
  private String lastName;
  private String address;
  private String email;

  // gettre a settre
}
```

Definícia mapovaní musí byť tiež upravená:

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
<class name="Customer" table="CUSTOMER">

  <composite-id name="id" class="CustomerId">
    <key-property name="countryCode"
                  type="string" column="COUNTRY_CODE" />
    <key-property name="idCardNo"
                  type="string" column="ID_CARD_NO"/>
  </composite-id>

  <property name="firstName" type="string" column="FIRST_NAME" />
  <property name="lastName" type="string" column="LAST_NAME" />
  <property name="address" type="string" column="ADDRESS" />
  <property name="email" type="string" column="EMAIL" />
</class>
</hibernate-mapping>
```

Všimnime si, že do elementu `<composite-id>` sme dodali názov atribútu a triedu.

Ak budeme chcieť získať objekt s daným identifikátorom, stačí poskytnúť príslušnej metóde inštanciu triedy `CustomerId`.

```
CustomerId customerId = new CustomerId("mo", "1234567(8)");
Customer customer = (Customer) session.get(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");
session.save(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 (http://commons.apache.org/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. Ak chceme naďalej zachovať jednoznačnosť dvojice `countryCode` a `idCardNo`, musíme tieto stĺpce v mapovaní definovať ako nenullové a dodať k nim obmedzenie na viacstĺpcovú jedinečnosť. Na to použijeme element `<properties>` pomocou ktorého môžeme zoskupiť viacero atribútov.

```
public class Customer {
  private Long id;
  private String countryCode;
  private String idCardNo;
  private String firstName;
  private String lastName;
  private String address;
  private String email;

  // gettre a settre
}
```

Súbor pre mapovanie:

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Customer" table="CUSTOMER">

  <id name="id" type="long" column="ID">
    <generator class="native"/>
  </id>

  <properties name="customerKey" unique="true">
    <property name="countryCode" type="string"
              column="COUNTRY_CODE" not-null="true" />
    <property name="idCardNo"
              type="string" column="ID_CARD_NO" not-null="true" />
  </properties>
  ...
  </class>
</hibernate-mapping>
```

4. Asociácie typu N: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 definíciu mapovania.

```
public class Publisher {
  private Long id;

  private String code;
  private String name;
  private String address;

  // gettre a settre
}
```

Definícia mapovania:

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Publisher" table="PUBLISHER">
    <id name="id" type="long" column="ID">
      <generator class="native" />
    </id>

    <property name="code" type="string">
      <column name="CODE" length="4" not-null="true" unique="true" />
    </property>

    <property name="name" type="string">
      <column name="PUBLISHER_NAME" length="100" not-null="true" />
    </property>

    <property name="address" type="string">
      <column name="ADDRESS" length="200" />
    </property>
```

```
  </class>
</hibernate-mapping>
```

Keďže sme vytvorili novú perzistentnú triedu, musíme ju dodať do konfiguračného súboru Hibernate.

```
<mapping resource="mo/org/cpttm/bookshop/Publisher.hbm.xml" />
```

V našej triede pre knihu `Book` musíme mať atribút reprezentujúci vydavateľa (`publisher`). Je typu `Publisher`.

```
public class Book {
  private Long id;
  private String isbn;
  private String name;

  private Publisher publisher;

  private Date publishDate;
  private Integer price;
  private List chapters;

  // gettre a settre
}
```

Ak budeme chcieť používať tento atribút pri mapovaní triedy na tabuľku, musíme ho zaviesť do definície mapovania pomocou elementu `<many-to-one>`. V tabuľke `BOOK` bude zavedený stĺpec `publisher_id` v ktorom budú uložené identifikátory vydavateľov pridružených ku knihám. Nesmieme zabudnúť spustiť antovský task, ktorým aktualizujeme databázovú schému (keďže sme upravili tabuľky).

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Book" table="BOOK">
    <id name="id" type="long" column="ID">
      <generator class="native"/>
    </id>

    <property name="isbn" type="string">
      <column name="ISBN" length="50" not-null="true" unique="true" />
    </property>

    <property name="name" type="string">
      <column name="BOOK_NAME" length="100" not-null="true" />
    </property>

    <property name="publishDate" type="date" column="PUBLISH_DATE" />
```

```
    <property name="price" type="int" column="PRICE" />

    <many-to-one name="publisher"
                 class="Publisher" column="PUBLISHER_ID" />
  </class>
</hibernate-mapping>
```

*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) {
  Session session = factory.openSession();
  try {
    Book book = (Book) session.get(Book.class, id);
    return book;
  } finally {
    session.close();
  }
}
```

Ak by sme sa pokúsili pristúpiť k objektu vydavateľa volaním metódy `book.getPublisher()` niekde mimo metódy `getBooks()`, nastala by výnimka.

```
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. Hibernate 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 sedenia. Keďže k vydavateľovi sme prvýkrát pristúpili až mimo metódy `getBook()` (kde už nemáme otvorené sedenie), Hibernate vyhodil výnimku.

Ak chceme, aby bol vydavateľ knihy dostupný aj mimo otvoreného sedenia, máme dve možnosti. Jednou z nich je explicitná inicializácia vydavateľa knihy. Na to môžeme použiť metódu `Hibernate.initialize()`, ktorej zavolaním vynútime načítanie inštancie vydavateľa z databázy.

```
Session session = factory.openSession();
try {
  Book book = (Book) session.get(Book.class, id);
  Hibernate.initialize(book.getPublisher());
  return book;
} finally {
  session.close();
}
```

Alternatívnym riešením je vypnutie *lazy* inicializácie. Toto však môže mať vplyv na výkonnosť, keďže s každým načítaním inštancie knihy sa bude musieť načítať aj inštancia vydavateľa.

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Book" table="BOOK">
    ...
    <many-to-one name="publisher" class="Publisher"
                 column="PUBLISHER_ID"
                 lazy="false" />
  </class>
</hibernate-mapping>
```

Rôzne prístupy pre načítavanie asociovaných objektov

Ak sa rozhodnete vypnúť *lazy* inicializáciu, máte viacero možností ako načítavať asociovaný objekt vydavateľa. Implicitný prístup je založený na vykonaní dvoch dopytoch: jedného pre knihu a druhý nezávislý pre vydavateľa, čo môže byť niekedy neefektívne.

Alternativným prístupom je použitie jediného dopytu využívajúceho spojenie tabuliek cez tabuľkové joiny. Nastaviť ho môžeme v atribúte `fetch` použitím hodnoty `join` (ak nepoužijeme tento atribút, bude sa aplikovať predošlý prístup, tzv. `select`).

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Book" table="BOOK">
    ...
   <many-to-one name="publisher" class="Publisher" column="PUBLISHER_ID"
                lazy="false"
                fetch="join" />
  </class>
</hibernate-mapping>
```

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

Prekvapenie však môže nastať pri používaní dopytov HQL:

```
Session session = factory.openSession();
try {
  Query query = session.createQuery("from Book where isbn = ?");
  query.setString(0, isbn);
  Book book = (Book) query.uniqueResult();
  return book;
} finally {
session.close();
}
```

V tomto prípade sa napriek explicitnému nastaveniu `fetch = join` vykonajú dva dopyty. Dôvodom je to, že dopyty v HQL sa prekladajú na dopyty SQL priamo a prístup pre *fetch* nakonfigurovaný v mapovacom súbore sa neberie do úvahy.

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

```
Session session = factory.openSession();
try {
  Query query = session.createQuery(
    “from Book book left join fetch book.publisher where book.isbn = ?");
  query.setString(0, isbn);
  Book book = (Book) query.uniqueResult();
  return book;
} finally {
  session.close();
}
```

Ak použijeme v HQL dopyte `left 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.

```
<many-to-one name=”publisher“ class=“Publisher“ column=“PUBLISHER_ID“
             lazy=“false“ fetch=“join“ />
```

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).

```
Session session = factory.openSession();
Transaction tx = null;
try {
  tx = session.beginTransaction();
  session.save(publisher);
  session.save(book);
  tx.commit();
} catch (HibernateException e) {
  if (tx != null) tx.rollback();
  throw e;
} finally {
  session.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 Hibernate poskytuje možnosť uložiť celú hierarchiu objektov pomocou jediného príkazom. Ak pridáme do elementu `<many-to-one>` atribút `cascade=”save-update”`, Hibernate bude operácie ukladania a aktualizácie kaskádne propagovať v smere asociácií. Inak povedané, uloženie (aktualizácia) knihy vynúti aj uloženie (aktualizáciu) vydavateľa.

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Book" table="BOOK">
    ...
    <many-to-one name="publisher" class="Publisher"
                 column="PUBLISHER_ID"
                 cascade="save-update" />
  </class>
</hibernate-mapping>
```

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

```
session.saveOrUpdate(book);
```

Okrem kaskádneho ukladania a aktualizácie môžeme tiež používať aj kaskádne odstraňovanie objektov. Stačí dodať do atribútu `cascade` hodnotu `delete`.

```
<many-to-one name="publisher" class="Publisher" column="PUBLISHER_ID"
             cascade="save-update,delete" />
```

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 vydavateľov. 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.)

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Book" table="BOOK">
    ...
    <many-to-one name="publisher" class="Publisher"
                 column="PUBLISHER_ID" />
    <join table="BOOK_PUBLISHER" optional="true">
      <key column="BOOK_ID" unique="true" />
      <many-to-one name="publisher" class="Publisher"
      column="PUBLISHER_ID" not-null="true" />
    </join>
  </class>
</hibernate-mapping>
```

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. Tento typ asociácie je nazývaný `one-to-one` (1:1). Ukážme si túto asociáciu na príklade zákazníka z predošlej kapitoly.

```
public class Customer {
  private Long id;
  private String countryCode;
  private String idCardNo;
  private String firstName;
  private String lastName;
  private String address;
  private String email;
```

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

Definícia mapovania:

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Customer" table="CUSTOMER">
    <id name="id" type="long" column="ID">
      <generator class="native"/>
    </id>
    <properties name="customerKey" unique="true">

    <property name="countryCode" type="string" column="COUNTRY_CODE"
              not-null="true" />
    <property name="idCardNo" type="string" column="ID_CARD_NO"
              not-null="true" />
    </properties>
    <property name="firstName" type="string" column="FIRST_NAME" />
    <property name="lastName" type="string" column="LAST_NAME" />
    <property name="address" type="string" column="ADDRESS" />
    <property name="email" type="string" column="EMAIL" />
  </class>
</hibernate-mapping>
```

Predstavme si teraz adresu zákazníka za samostatný objekt. 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
}
```

Definícia mapovania triedy je nasledovná:

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Address" table="ADDRESS">
```

```
    <id name="id" type="long" column="ID">
      <generator class="native" />
    </id>

    <property name="city" type="string" column="CITY" />
    <property name="street" type="string" column="STREET" />
    <property name="doorplate" type="string" column="DOORPLATE" />
  </class>
</hibernate-mapping>
```

Obe definície mapovania zaregistrujeme v konfiguračnom súbore pre Hibernate

```
<mapping resource="mo/org/cpttm/bookshop/Customer.hbm.xml" />
<mapping resource="mo/org/cpttm/bookshop/Address.hbm.xml" />
```

Teraz prejdeme k definícii mapovania asociácie 1:1. Tú môžeme realizovať tromi rôznymi spôsobmi.

Mapovanie pomocou cudzieho kľúča

Najjednoduchším spôsobom mapovania asociácie 1:1 je použitie asociácie M:1, v ktorej zavedieme obmedzenie na jedinečnosť. Konfigurácia *lazy* inicializácie, získavania asociovaných objektov a kaskádovania je identická ako v predošlom prípade.

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Customer" table="CUSTOMER">
    ...
    <many-to-one name="address" class="Address" column="ADDRESS_ID"
                 unique="true" cascade="save-update,delete" />
  </class>
</hibernate-mapping>
```

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;

  private Customer customer;

  // gettre a settre
}
```

Mapovanie:

```
<hibernate-mapping>
  <class name="Address" table="ADDRESS">
    ...
    <one-to-one name="customer" class="Customer" property-ref="address" />
  </class>
</hibernate-mapping>
```

Mapovanie pomocou primárneho kľúča

Druhým spôsobom mapovania asociácií 1:1 je nastavenie rovnakého identifikátora pre oba objekty v asociácii. Zákazníka môžeme považovať za „hlavný" objekt, ktorý bude mať automaticky generovaný identifikátor. Identifikátor „vedľajšej" triedy `Address` namapujeme ako "cudzí" s odkazom identifikátor zákazníka. Každému objektu adresy potom bude priradený taký istý identifikátor, aký má zákazník s danou adresou. Atribút `constrained` znamená, že identifikátor adresy je cudzím kľúčom k identifikátoru zákazníka.

Mapovanie zákazníka:

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Customer" table="CUSTOMER">
    <id name="id" type="long" column="ID">
      <generator class="native"/>
    </id>
    ...
    <one-to-one name="address" class="Address"
                cascade="save-update,delete" />
  </class>
</hibernate-mapping>
```

Mapovanie adresy:

```
<hibernate-mapping>
  <class name="Address" table="ADDRESS">
    <id name="id" column="ID">
      <!-- Cudzí identifikátor špecifikovaný v atribúte "customer" -->
      <generator class="foreign">
        <param name="property">customer</param>
      </generator>
    </id>
    ...
    <one-to-one name="customer" class="Customer" constrained="true" />
  </class>
</hibernate-mapping>
```

Ak chceme namapovať obojsmernú asociáciu, nadeklarujeme `<one-to-one>` v triedach na jej oboch stranách. Ak chceme asociáciu len v jednom smere, vynecháme `<one-to-one>` na strane zákazníka, čím dosiahneme, že z objektu adresy budeme vedieť pristúpiť k objektu zákazníka. Ak chceme dosiahnuť opačný prístup, vieme vzájomne vymeniť deklarácie `<id>` a `<one-to-one>` medzi triedami zákazníka a adresy.

*Použitie medzitabuľky (**join table**)*

Posledným prístupom k mapovaniu asociácií 1:1 je použitie medzitabuľky (tak ako v prípade asociácií M:1). Rozdiely oproti prístupu M:1 nie sú žiadne, niekedy sa však môže stať, že potrebujeme obmedzenie na jedinečnosť v `<key>` a `<many-to-one>`). V praxi sa tento prístup sa používa len zriedka.

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Customer" table="CUSTOMER">
    ...
    <many-to-one name="address" class="Address" column="ADDRESS_ID"
                 unique="true" cascade="save-update" />

    <join table="CUSTOMER_ADDRESS" optional="true">
      <key column="CUSTOMER_ID" unique="true" />
      <many-to-one name="address" class="Address" column="ADDRESS_ID"
                   not-null="true" unique="true"
                   cascade="save-update,delete" />
    </join>
  </class>
</hibernate-mapping>
```

Ak chceme mať obojsmernú asociáciu, do definície mapovania v triede `Address` pridáme mapovanie pre opačný smer (s tým, že vynecháme atribút pre kaskádovanie).

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Address" table="ADDRESS">
    ...
   <one-to-one name="customer" class="Customer" property-ref="address" />

    <join table="CUSTOMER_ADDRESS" optional="true">
      <key column="ADDRESS_ID" unique="true" />
      <many-to-one name="customer" class="Customer" column="CUSTOMER_ID"
                   not-null="true" unique="true" />
    </join>
  </class>
</hibernate-mapping>
```

Ak sa pokúsite uložiť takýto typ objektovej štruktúry do databázy, získate chybu. Dôvodom je to, že Hibernate sa pokúsi uložiť obe strany asociácie nezávisle. V prípade zákazníka sa do tabuľky CUSTOMER_ADDRESS vloží jeden riadok a v prípade adresy ten istý riadok, čo je však porušenie obmedzenia na jedinečnosť.

Ak sa chceme vyhnúť dvojitému mapovaniu toho istého vzťahu, musíme označiť jednu zo strán asociácie ako *inverznú* (inverse = "true"). V tom prípade bude Hibernate pri ukladaní ignorovať príslušnú stranu vzťahu, pretože bude vedieť, že asociácia už bola namapovaná v opačnom smere.

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Address" table="ADDRESS">
    ...
    <join table="CUSTOMER_ADDRESS" optional="true" inverse="true">

      <key column="ADDRESS_ID" unique="true" />
      <many-to-one name="customer" class="Customer"
                   column="CUSTOMER_ID"
                   not-null="true" unique="true"     </join>
  </class>
</hibernate-mapping>
```

5. Mapovanie kolekcií

5.1. Mapovanie kolekcií jednoduchých typov

V našom internetovom kníhkupectve sme sa rozhodli zobrazovať používateľom informácie o kapitolách jednotlivých kníh, čím im možno uľahčíme rozhodovanie pred kúpou. Každá kniha obsahuje viacero kapitol, ktoré budeme reprezentovať ako reťazce (pre jednoduchosť budeme používať len názov kapitoly) uložené v zozname java.util.List (neskôr si ukážeme použitie ďalších druhov kolekcií).

```
public class Book {
  private Long id;
  private String isbn;
  private String name;
  private Publisher publisher;
  private Date publishDate;
  private Integer price;
  private List<String> chapters;

  // gettre a settre
}
```

Dosiaľ sme mapovali buď atribúty objektov na tabuľkové stĺpce alebo vzťahy medzi objektami na asociácie. Mapovanie kolekcií je založené na mapovaní objektov z Java Collections API na kolekcie podporované v Hibernate.

Najjednoduchším typom kolekcie je multimnožina alebo *bag* (multimnožina môže obsahovať viacero rovnakých prvkov a zanedbáva sa v nej poradie prvkov). Zodpovedajúcim dátovým typom v Jave je zoznam `java.util.List`.

Príkladom definície mapovania je

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Book" table="BOOK">
    <id name="id" type="long" column="ID">
      <generator class="native"/>
    </id>

    <property name="isbn" type="string">
      <column name="ISBN" length="50" not-null="true" unique="true" />
    </property>

    <property name="name" type="string">
      <column name="BOOK_NAME" length="100" not-null="true" />
    </property>

    <property name="publishDate" type="date" column="PUBLISH_DATE" />
    <property name="price" type="int" column="PRICE" />

    <many-to-one name="publisher" class="Publisher"
                 column="PUBLISHER_ID"
                 lazy="false" fetch="join"
                 cascade="save-update,delete" />

    <bag name="chapters" table="BOOK_CHAPTER">
      <key column="BOOK_ID" />
      <element column="CHAPTER" type="string" length="100" />
    </bag>
  </class>
</hibernate-mapping>
```

V tomto mapovaní budeme používať na ukladanie informácií o kapitolách samostatnú tabuľku `BOOK_CHAPTER`, ktorá bude obsahovať dva stĺpce `book_id` a `chapter`. Každá kapitola bude v tejto tabuľke uložená ako jeden záznam.

Po úprave mapovacieho súboru nezabudnite spustiť antovský task pre aktualizáciu databázovej schémy.

Lazy *inicializácia kolekcií*

Skúsme získať z databázy inštanciu knihy a to vrátane zoznamu kapitol (ktorý sme namapovali na *bag*):

```
Session session = factory.openSession();
try {
  Book book = (Book) session.get(Book.class, id);
  return book;
} finally {
  session.close();
}
```

Ak však pristúpime k zoznamu kapitol pomocou `book.getChapters()` mimo otvoreného sedenia, vyvolá sa výnimka:

```
for(String chapter : book.getChapters()) {
  System.out.println(chapter);
}
```

Príčinou tejto výnimky je *lazy* inicializácia, ktorá vynúti načítavanie príslušnej kolekcie až pri prvom prístupe k nej. Používať ju však možno len v rámci otvoreného sedenia, čo sme v príklade nedodržali. Riešením je explicitná inicializácia príslušnej kolekcie, čím sa jednotlivé prvky načítajú ihneď.

```
Session session = factory.openSession();
try {
  Book book = (Book) session.get(Book.class, id);
  Hibernate.initialize(book.getChapters());
  return book;
} finally {
  session.close();
}
```

Lazy načítavanie prvkov kolekcie je možné vypnúť nastavením `lazy="false"` na príslušnom elemente kolekcie. Podobne ako v prípade asociácií je však treba zvážiť dopad na výkonnosť.

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Book" table="BOOK">
    ...
    <bag name="chapters" table="BOOK_CHAPTER" lazy="false">
      <key column="BOOK_ID" />
```

```
      <element column="CHAPTER" type="string" length="100" />
    </bag>
  </class>
</hibernate-mapping>
```

Rôzne prístupy k načítavaniu kolekcií

Implicitným režimom pre načítavanie kolekcie je `select`, pri ktorom Hibernate odošle do databázy dva dopyty: jeden pre načítanie knihy a druhý pre načítanie zoznamu kapitol. Ak nastavíme režim na `join`, Hibernate použije jediný SQL dopyt `SELECT`, v ktorom sa použije spojenie dvoch tabuliek cez `JOIN`.

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Book" table="BOOK">
    ...
    <bag name="chapters" table="BOOK_CHAPTER" lazy="false" fetch="join">
      <key column="BOOK_ID" />
      <element column="CHAPTER" type="string" length="100" />
    </bag>
  </class>
</hibernate-mapping>
```

Ak používame dopytovanie pomocou HQL, prístup `fetch` nastavení v mapovaní sa ignoruje. Na načítanie prvkov kolekcie sa potom musí použiť syntax využívajúca direktívu `left join`. Tá umožňuje vynútiť načítanie prvkov i v prípade, že máme *lazy* kolekciu.

```
Session session = factory.openSession();
try {
  Query query = session.createQuery(
    "from Book book left join fetch book.chapters where book.isbn = ?");
  query.setString(0, isbn);

  Book book = (Book) query.uniqueResult();
  return book;
} finally {
  session.close();
}
```

Pri výbere používaného prístupu je treba obzvlášť zvážiť výkonnostný dopad. V prípade `fetch`-prístupu totiž platí, že veľkosť množiny objektov vo výsledku je vynásobená veľkosťou kolekcie (v našom prípade počet kníh krát počet kapitol). V prípade, že je kolekcia veľká, je lepšie zvoliť `select`-prístup.

5.2. Rôzne druhy kolekcií v Hibernate

V predošlej časti sme si ukázali použitie multimnožiny `<bag>` v Hibernate. Okrem toho však existujú ďalšie druhy kolekcií, pričom každá má iné vlastnosti.

Multimnožina

Multimnožina (*bag*) je neusporiadaná kolekcia, ktorá môže obsahovať duplicitné prvky. To znamená, že ak uložíte do databázy multimnožinu, po jej následnom načítaní z databázy sa poradie jej prvkov nemusí zachovať. V Jave žiaľ nejestvuje trieda pre multimnožinu. Namiesto nej môžeme použiť zoznam `java.util.List`.

Na ukladanie kolekcie kapitol použijeme tabuľku `BOOK_CHAPTER` s dvoma stĺpcami: `book_id` a `chapter`. Každá kapitola sa uloží ako jeden riadok tabuľky.

```
public class Book {
  ...
  private List<String> chapters;
}
```

Definícia mapovaní:

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Book" table="BOOK">
    ...
    <bag name="chapters" table="BOOK_CHAPTER">
      <key column="BOOK_ID" />
      <element column="CHAPTER" type="string" length="100" />
    </bag>
  </class>
</hibernate-mapping>
```

Množina `set`

Množina je multimnožina, ktorá nepovoľuje duplicitné objekty. Ak pridáte do množiny prvok, ktorý v nej už existuje, nestane sa nič (resp. nový prvok nahradí starý). Poradie prvkov v množine nie je určené (môžeme ich však zoraďovať, čo si ukážeme neskôr). V Hibernate zodpovedá množine element `<set>` a v Jave trieda `java.util.Set`.

Štruktúra tabuliek je rovnaká ako v prípade multimnožiny / *bag*u. Medzitabuľka teda obsahuje dva stĺpce `book_id` a `chapter`.

```
public class Book {
  ...
  private Set chapters;
}
```

Definícia mapovania:

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Book" table="BOOK">
    ...
    <set name="chapters" table="BOOK_CHAPTER">
      <key column="BOOK_ID" />
      <element column="CHAPTER" type="string" length="100" />
    </set>
  </class>
</hibernate-mapping>
```

Zoznam `list`

Zoznam `<list>` je indexovaná kolekcia, kde sa uchováva aj poradie (resp. index) prvkov. To znamená, že pri načítavaní z databázy je možné zachovávať poradie prvkov. Zodpovedajúcim typom v Jave je `java.util.List` – na rozdiel od multimnožiny sa však do databázy ukladá aj index prvku v zozname.

Štruktúra tabuliek je takmer rovnaká ako v prípade multimnožiny. Do tabuľky `BOOK_CHAPTER` však potrebujeme pridať ešte jeden stĺpec: `chapter_index` typu `int` v ktorom bude uložený index prvku.

```
public class Book {
  ...
  private List chapters;
}
```

Definícia mapovania:

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Book" table="BOOK">
    ...
    <list name="chapters" table="BOOK_CHAPTER">
      <key column="BOOK_ID" />
      <list-index column="CHAPTER_INDEX"/>
      <element column="CHAPTER" type="string" length="100" />
    </list>
  </class>
</hibernate-mapping>
```

Pole `array`

Pole `<array>` má rovnaké použitie ako zoznam `<list>`. Jediným rozdielom je prislúchajúci dátový typ v Jave, ktorým nie je `java.util.List`, ale pole príslušného typu. Táto kolekcia sa používa len veľmi zriedka (azda len v prípadoch, keď musíme mapovať „historické" triedy). Vo väčšine prípadov je užitočnejšie použiť zoznam, ktorý na rozdiel od poľa môže byť dynamicky zväčšovaný či zmenšovaný.

Štruktúra tabuliek je rovnaké ako v prípade zoznamu, čiže máme tri stĺpce: `book_id`, `chapter` a `chapter_index`.

```
public class Book {
  ...
  private String[] chapters;
}
```

Definícia mapovania

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Book" table="BOOK">
    ...
    <array name="chapters" table="BOOK_CHAPTER">
      <key column="BOOK_ID" />
      <list-index column="CHAPTER_INDEX" />
      <element column="CHAPTER" type="string" length="100" />
    </array>
  </class>
</hibernate-mapping>
```

Mapa `map`

Na mapu sa dá dívať ako na zoznam položiek, ktoré sú tvorené dvojicou kľúč-hodnota. Hodnoty prvkov vieme vyhľadávať na základe kľúča, a navyše kľúče i hodnoty môžu byť ľubovoľného typu.

Z istého pohľadu je mapa veľmi podobná zoznamu; na zoznam sa môžeme pozerať ako na mapu, kde sú kľúče tvorené celými číslami.

Štruktúra tabuliek je podobná ako v prípade zoznamu. Máme tri stĺpce: `book_id`, `chapter` a `chapter_index`. Dátovým typom stĺpca s indexom však nie je celé číslo, ale reťazec.

```
public class Book {
  ...
  private Map chapters;
}
```

Definícia mapovaní:

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Book" table="BOOK">
    ...

    <map name="chapters" table="BOOK_CHAPTER">
      <key column="BOOK_ID" />
      <map-key column="CHAPTER_KEY" type="string" />
      <element column="CHAPTER" type="string" length="100" />
    </map>

  </class>
</hibernate-mapping>
```

5.3. Triedenie kolekcií

Niekedy je užitočné, aby boli naše kolekcie po načítaní z databázy automaticky usporiadané. Hibernate podporuje dva prístupy na triedenie kolekcií.

Triedenie v pamäti

Prvým spôsobom je použitie triediacich tried a algoritmov poskytovaných Javou. Triedenie kolekcie sa uskutoční v pamäti po načítaní dát z databáz. Treba vedieť, že pre veľké kolekcie môže byť tento spôsob pamäťovo náročný. Tento spôsob triedenia je podporovaný len množinami `<set>` a mapami `<map>`.

Ak chceme, aby bola kolekcia zotriedená na základe „prirodzeného usporiadania", nastavíme atribút `sort` na hodnotu `natural`. Na porovnávanie prvkov bude Hibernate používať metódu `compareTo()` definovanú v interfejse `java.lang.Comparable`. Tento interfejs implementuje väčšina základných dátových typov (reťazce, celé čísla, reálne čísla a pod.)

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Book" table="BOOK">
    ...
    <set name="chapters" table="BOOK_CHAPTER" sort="natural">
      <key column="BOOK_ID" />
      <element column="CHAPTER" type="string" length="100" />
    </set>
  </class>
</hibernate-mapping>
```

Ak vám nevyhovuje prirodzené usporiadanie, je možné vytvoriť a nastaviť vlastný spôsob porovnávania prvkov. Stačí vytvoriť novú triedu implementujúcu interfejs `java.lang.Comparator` s prekrytou metódou `compare()`. Použitie tejto triedy možno nastaviť v atribúte `sort` v mapovaní kolekcie.

```
public class ChapterComparator implements Comparator {
  public int compare(Object o1, Object o2) {
  // ak o1 a o2 nie sú rovnakého typu, hoď výnimku
  // ak o1 je menšie než o2, vráť záporné číslo
  // ak o1 je identické s o2, vráť nulu
  // ak o1 je väčšie než o2, vráť kladné číslo
  ...
  }
}
```

Definícia mapovania:

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Book" table="BOOK">
    ...
    <set name="chapters" table="BOOK_CHAPTER"
         sort="mo.org.cpttm.bookshop.ChapterComparator">
      <key column="BOOK_ID" />
      <element column="CHAPTER" type="string" length="100" />
    </set>
  </class>
</hibernate-mapping>
```

Triedenie v databáze

Ak je kolekcia veľká, je efektívnejšie prenechať triedenie priamo na databázu. V definícii mapovania vieme špecifikovať podmienku `order-by`, ktorá (podobne ako v SQL) určí stĺpec, podľa ktorého sa zotredia riadky výsledného dopytu. Je treba dať pozor na to, že hodnota v `order-by` musí byť názvom stĺpca v databázovej tabuľke a nie názov atribútu objektu!

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Book" table="BOOK">
    ...
    <set name="chapters" table="BOOK_CHAPTER" order-by="CHAPTER">
      <key column="BOOK_ID" />
      <element column="CHAPTER" type="string" length="100" />
    </set>
  </class>
</hibernate-mapping>
```

V podmienke order-by nie sme obmedzení len na stĺpce, ale môžeme použiť akýkoľvek výraz, ktorý je platný v ORDER BY časti SQL dopytu. Môžeme teda použiť viacero názvov stĺpcov oddelených čiarkami, používať asc a desc na zmenu poradia triedenia alebo uviesť celý poddopyt (ak to povoľuje databáza). Hibernate skopíruje hodnotu z order-by do výsledného vygenerovaného SQL do klauzuly ORDER BY.

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Book" table="BOOK">
    ...
    <set name="chapters" table="BOOK_CHAPTER"
      order-by="CHAPTER desc">

      <key column="BOOK_ID" />
      <element column="CHAPTER" type="string" length="100" />
    </set>
  </class>
</hibernate-mapping>
```

6. Asociácie 1:M a M:N

6.1. 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 mapovanie na tabuľku. Pristúpime teda k jeho vytvoreniu. V rámci neho priradíme kapitole automaticky generovaný identifikátor.

```
public class Chapter {
  private Long id;
  private int index; //poradie kapitoly
  private String title; //názov
  private int numOfPages; //počet strán

  // gettre a settre
}
```

Mapovanie bude vyzerať nasledovne:

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Chapter" table="CHAPTER">
    <id name="id" type="long" column="ID">
      <generator class="native" />
    </id>

    <property name="index" type="int" column="IDX" not-null="true" />

    <property name="title" type="string">
      <column name="TITLE" length="100" not-null="true" />
    </property>

    <property name="numOfPages" type="int" column="NUM_OF_PAGES" />

  </class>
</hibernate-mapping>
```

Novovytvorenú definíciu mapovania novej perzistentnej triedy musíme dodať do konfiguračného súboru `hibernate.cfg.xml`.

```
<mapping resource="mo/org/cpttm/bookshop/Chapter.hbm.xml" />
```

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 `String`u na `Chapter` v generiku. Otázka je, akú hibernateovský typ kolekcie máme zvoliť pri mapovaní. Kniha nezvykne obsahovať duplicitné kapitoly, preto môžeme použiť množinu `<set>`.

```
public class Book {
  private Long id;
  private String isbn;
  private String name;
  private Publisher publisher;
  private Date publishDate;
  private Integer price;
  // množina kapitol
  private Set<Chapter> chapters;

  // gettre a settre
}
```

Rovnako musíme poopraviť definíciu mapovania. Keďže ukladáme kapitolové objekty (a nie základné dátové typy), namiesto `<element>`u budeme používať asociáciu 1:M, ktorej zodpovedá element `<one-to-many>`.

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Book" table="BOOK">
    <id name="id" type="long" column="ID">
      <generator class="native" />
    </id>
    <property name="isbn" type="string">
      <column name="ISBN" length="50" />
    </property>
    <property name="name" type="string">
      <column name="BOOK_NAME" length="100" not-null="true"
              unique="true" />
    </property>
    <property name="publishDate" type="date" column="PUBLISH_DATE" />
    <property name="price" type="int" column="PRICE" />
    <set name="chapters" table="BOOK_CHAPTER">
      <key column="BOOK_ID" />
      <element column="CHAPTER" type="string" length="100" />
      <one-to-many class="Chapter" />
    </set>
  </class>
</hibernate-mapping>
```

Keďže ku kapitolám budeme pristupovať sekvenčne, je rozumné zoradiť ich podľa atribútu `index` (resp. podľa zodpovedajúceho stĺpca `IDX`). Najjednoduchšou a najefektívnejšou cestou je prenechať triedenie na databázu. To dosiahneme dodaním `order-by=”IDX”` do definície mapovania.

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Book" table="BOOK">
    ...
    <set name="chapters" order-by="IDX">
      <key column="BOOK_ID" />
      <one-to-many class="Chapter" />
    </set>
  </class>
</hibernate-mapping>
```

Ak by sme chceli pristupovať ku kolekcii kapitol aj pomocou indexu kapitoly (napr. získať desiatu kapitolu), typ `<set>` by už nevyhovoval.Namiesto neho je v tomto prí-

pade vhodnejší zoznam `<list>`. Celočíselné indexy takéhoto zoznamu budú potom ukladané v stĺpci `IDX` v tabuľke `CHAPTER`.

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Book" table="BOOK">
    ...
    <list name="chapters">
      <key column="BOOK_ID" />
      <list-index column="IDX" />
      <one-to-many class="Chapter" />
    </list>
  </class>
</hibernate-mapping>
```

Inicializácia podľa potreby a prístupy k načítavaniu asociácií

Pre asociáciu 1:M môžeme, tak ako v ostatných prípadoch, nakonfigurovať *lazy* inicializáciu a používaný prístup pre *fetch*.

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Book" table="BOOK">
    ...
    <set name="chapters" lazy="false" fetch="join">
      <key column="BOOK_ID" />
      <one-to-many class="Chapter" />
    </set>
  </class>
</hibernate-mapping>
```

V HQL môžeme použiť na načítanie asociovaných objektov klauzulu `left join fetch`, ktorá je tiež užitočná v prípade, keď chceme vynútiť inicializáciu asociovanej kolekcie i v prípade, že je namapovaná ako *lazy*. (`left join fetch` je jeden z efektívnych spôsobov, akými je možné nainicializovať *lazy* asociácie a kolekcie všetkých objektov vrátených dopytom).

```
Session session = factory.openSession();
try {
  Query query = session.createQuery(
    "from Book book left join fetch book.chapters where book.isbn = ?");
  query.setString(0, isbn);

  Book book = (Book) query.uniqueResult();
  return book;
} finally {
  session.close();
}
```

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/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 elementu `<set>` dodáme atribút `cascade`, v ktorom vymenujeme operácie, ktoré sa majú propagovať.

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Book" table="BOOK">
    ...
    <set name="chapters" order-by="IDX"
         cascade="save-update,delete">

      <key column="BOOK_ID" />
      <one-to-many class="Chapter" />
    </set>
  </class>
</hibernate-mapping>
```

Na uloženie zložitej štruktúry objektov môžeme využiť jediné volanie metódy `saveOrUpdate()`. Ak by sme ukladali knihu s dvoma kapitolami a prezreli si SQL dopyty, ktoré vygeneroval Hibernate, uvideli by sme výpisy, ktoré na prvý pohľad vyzerajú zmätočne:

```
insert into BOOK (ISBN, BOOK_NAME, PUBLISH_DATE,
                  PRICE, PUBLISHER_ID, ID)
values (?, ?, ?, ?, ?, null)

insert into CHAPTER (IDX, TITLE, NUM_OF_PAGES, ID)
values (?, ?, ?, null)

insert into CHAPTER (IDX, TITLE, NUM_OF_PAGES, ID)
values (?, ?, ?, ?)

uupdate CHAPTER set BOOK_ID=? where ID=?

update CHAPTER set BOOK_ID=? where ID=?
```

Podľa ladiacich výpisov boli vykonané tri dopyty `INSERT` a dva dopyty `UPDATE`. Prečo sa však vykonali aj `UPDATE`y, keď sú podľa všetkého zbytočné?

Ak zavoláme metódu `saveOrUpdate` a odovzdáme jej triedu knihy s asociovanými kapitolami, Hibernate vykoná nasledovné úkony:

- Uloží alebo aktualizuje objekt knihy. V našom prípade ho uloží, keďže naša kniha je nová a jej identifikátor je `null`.
- Kaskádne vykoná operáciu `saveOrUpdate()` na každej kapitole danej knihy. V našom prípade bude každá kapitola uložená ako nový objekt, pretože má `null`ový identifikátor.
- Uloží asociáciu 1:M. Každému riadku tabuľky `CHAPTER` bude priradený identifikátor, teda hodnota v stĺpci `BOOK_ID`.

Dva `UPDATE` dopyty sa zdajú byť zbytočnými, keďže identifikátor `book_id` v tabuľke `CHAPTER` sme sa mohol nastaviť už v `INSERT`och. Tento problém by sme vyriešili obojsmernou asociáciou, čo spravíme neskôr.

Venujme sa teraz chvíľu inej situácii. Čo ak by sme chceli odstrániť z knihy tretiu kapitolu? To môžeme dosiahnuť nasledovným kódom, v ktorom budeme iterovať kolekciu kapitol. Ak nájdeme tretí element, odstránime ho. Výsledkom bude `null`ová hodnota v stĺpci `book_id` v riadku tretej kapitoly, čiže stav, keď kapitola nepatrí žiadnej knihe.

```
for (Iterator iter = book.getChapters().iterator(); iter.hasNext();) {
  Chapter chapter = (Chapter) iter.next();
  if (chapter.getIndex() == 3) {
    iter.remove();
  }
}
```

Má takéto chovanie zmysel? Takto odstránený objekt kapitoly sa stane „bezprízorným” a zbytočne len zaberá miesto v databáze a pamäti. Asi je preto rozumnejšie ho odstrániť.

Objekt kapitoly nepatriacej žiadnej knihe sa tiež nazýva *sirota* (*orphan*). Siroty môžeme odstraňovať z databázy použitím vhodného prístupu v kaskádovaní. Nastavenie `delete-orphan` v definícii kaskádovania na danej kolekcii zabezpečí automatické odstránenie bezprízorného objektu z databázy.

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Book" table="BOOK">
    ...
    <set name="chapters" order-by="IDX"
      cascade="save-update,delete,delete-orphan">

      <key column="BOOK_ID" />
      <one-to-many class="Chapter" />
    </set>
  </class>
</hibernate-mapping>
```

6.2. 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 atribútu book (odkazujúceho 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.

```
public class Chapter {
  private Long id;
  private int index;
  private String title;
  private int numOfPages;

  private Book book;

  // gettre a settre
}
```

Mapovanie bude vyzerať nasledovne:

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Chapter" table="CHAPTER">
    ...

    <many-to-one name="book" class="Book" column="BOOK_ID" />

  </class>
</hibernate-mapping>
```

Ak vytvoríme novú inštanciu knihy s dvoma kapitolami, uložíme ju do databázy a prezrieme si vygenerované SQL dopyty, uvidíme nasledovný výpis:

```
insert into BOOK (ISBN, BOOK_NAME, PUBLISH_DATE,
                  PRICE, PUBLISHER_ID, ID)
values (?, ?, ?, ?, ?, null)
insert into CHAPTER (IDX, TITLE, NUM_OF_PAGES, BOOK_ID, ID)
values (?, ?, ?, ?, null)

insert into CHAPTER (IDX, TITLE, NUM_OF_PAGES, BOOK_ID, ID)
values (?, ?, ?, ?, null)

update CHAPTER set BOOK_ID=? where ID=?
update CHAPTER set BOOK_ID=? where ID=?
```

Aj v tomto výpise je vidieť, že sa vykonalo päť SQL príkazov. Rozdiel oproti jednosmernej asociácii je v zahrnutí stĺpca `book_id` do `INSERT`ov nad tabuľkou `CHAPTER`. Posledné dva príkazy `UPDATE` sa tým stávajú nadbytočnými. Ak pridáme do definície mapovania kolekcie nastavenie `inverse="true"`, Hibernate už nebude príslušnú kolekciu ukladať. Bude totiž predpokladať, že táto kolekcia sa uloží v rámci ukladania opačného konca asociácie (teda pri ukladaní či aktualizovaní knihy). Ak je na opačnom konci asociácie nastavené kaskádovanie operácií, prejaví sa to na prvkoch tejto kolekcie zvyčajným spôsobom.

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Book" table="BOOK">
    ...
    <set name="chapters" order-by="IDX"
         cascade="save-update,delete,delete-orphan"
         inverse="true">
      <key column="BOOK_ID" />
      <one-to-many class="Chapter" />
    </set>
  </class>
</hibernate-mapping>
```

6.3. 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 asociáciu *many-to-many*.

```
public class Customer {
  private Long id;
  private String countryCode;
  private String idCardNo;
  private String firstName;
  private String lastName;
  private String email;

  private Set addresses;

  // gettre a settre
}
```

Definícia asociácie M:N pomocou `<many-to-many>` je veľmi podobná definícii `<one-to-many>`. Pre reprezentáciu takejto asociácie musíme použiť medzitabuľku, ktorá bude obsahovať cudzie kľúče pre obe asociované strany.

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Customer" table="CUSTOMER">
    <id name="id" type="long" column="ID">
      <generator class="native" />
    </id>
    <properties name="customerKey" unique="true">
      <property name="countryCode" type="string" column="COUNTRY_CODE"
              not-null="true" />
      <property name="idCardNo" type="string" column="ID_CARD_NO"
              not-null="true" />
    </properties>
    <property name="firstName" type="string" column="FIRST_NAME" />
    <property name="lastName" type="string" column="LAST_NAME" />
    <property name="email" type="string" column="EMAIL" />

    <set name="addresses" table="CUSTOMER_ADDRESS"
      cascade="save-update,delete">

      <key column="CUSTOMER_ID" />
      <many-to-many column="ADDRESS_ID" class="Address" />
    </set>
  </class>
</hibernate-mapping>
```

Tým sme vybavili (zatiaľ jednosmernú) asociáciu zákazník->adresa. Ak chceme túto asociáciu povýšiť na obojsmernú, pridáme opačné definície k mapovaniu objektu adresy, pričom použijeme rovnakú medzitabuľku.

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

  // gettre a settre
}
```

Mapovanie bude vyzerať nasledovne:

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Address" table="ADDRESS">
    <id name="id" type="long" column="ID">
      <generator class="native" />
    </id>

    <property name="city" type="string" column="CITY" />
    <property name="street" type="string" column="STREET" />
    <property name="doorplate" type="string" column="DOORPLATE" />

    <set name="customers" table="CUSTOMER_ADDRESS">
      <key column="ADDRESS_ID" />
      <many-to-many column="CUSTOMER_ID" class="Customer" />
    </set>
  </class>
</hibernate-mapping>
```

Ak sa však pokúsite uložiť do databázy komplexnú štruktúru využívajúcu uvedené mapovanie, stretnete sa s chybovým hlásením. Dôvodom je to, že Hibernate 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).

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Address" table="ADDRESS">
    ...
    <set name="customers" table="CUSTOMER_ADDRESS"
         <!-- asociácia je už namapovaná na opačnej strane -->
         inverse="true">

      <key column="ADDRESS_ID" />
      <many-to-many column="CUSTOMER_ID" class="Customer" />
    </set>
  </class>
</hibernate-mapping>
```

6.4. Používanie medzitabuľky pre asociácie 1:N

Medzitabuľku sme používali už v prípade asociácií M:1 a môžeme ju použiť aj v prípade 1:N. Takúto asociáciu namapujeme pomocou elementu `<many-to-many>`, v ktorom nastavíme `unique=”true”`. Tým stanovíme obmedzenie na jedinečnosť cudzieho kľúča `BOOK_ID`, čiže v konečnom dôsledku obmedzíme asociáciu M:N na typ 1:N.

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Book" table="BOOK">
    ...
    <set name="chapters" table="BOOK_CHAPTER"
         cascade="save-update,delete,delete-orphan">

      <key column="BOOK_ID" />
      <many-to-many column="CHAPTER_ID" class="Chapter"
                    unique="true" />
    </set>
  </class>
</hibernate-mapping>
```

Ak chceme používať obojsmernú dvojasociáciu 1:N / M:1 založenú na medzitabuľke, v smere M:1 ju definujeme zvyčajným spôsobom. Navyše nesmieme zabudnúť označiť jeden zo smerov asociácie ako inverzný. V tomto prípade sme sa rozhodli označiť za inverzný smer kapitola-kniha, ale nič nebráni použitiu opačného riešenia.

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Chapter" table="CHAPTER">
    ...
    <join table="BOOK_CHAPTER" optional="true" inverse="true">
      <key column="CHAPTER_ID" unique="true" />
      <many-to-one name="book" class="Book" column="BOOK_ID"
```

```
                  not-null="true" />
    </join>
  </class>
</hibernate-mapping>
```

7. Mapovanie komponentov

7.1. Používanie komponentov

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`.

```
public class Order {
  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
}
```

Následne vytvoríme definíciu mapovania pre túto perzistentnú triedu, pričom jej atribúty namapujeme zvyčajným spôsobom.

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Order" table="BOOK_ORDER">
    <id name="id" type="long" column="ID">
      <generator class="native" />
    </id>
    ...
    <property name="weekdayRecipient"
              type="string" column="WEEKDAY_RECIPIENT" />
    <property name="weekdayPhone"
              type="string" column="WEEKDAY_PHONE" />
```

```
    <property name="weekdayAddress"
              type="string" column="WEEKDAY_ADDRESS" />
    <property name="holidayRecipient"
              type="string" column="HOLIDAY_RECIPIENT" />
    <property name="holidayPhone"
              type="string" column="HOLIDAY_PHONE" />
    <property name="holidayAddress"
              type="string" column="HOLIDAY_ADDRESS" />
  </class>
</hibernate-mapping>
```

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 predošlých kapitolách môžeme špecifikovať kontakt ako novú perzistentnú triedu a použiť asociáciu 1:1 (najjednoduchšie je použiť `<many-to-one>`, kde nastavíme `unique=”true”`) medzi objednávkou `Order` a kontaktom `Contact`.

Mapovanie pre kontakt:

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Contact" table="CONTACT">
    <id name="id" type="long" column="ID">
```

```
      <generator class="native" />
    </id>
    <property name="recipient" type="string" column="RECIPIENT" />
    <property name="phone" type="string" column="PHONE" />
    <property name="address" type="string" column="ADDRESS" />
  </class>
</hibernate-mapping>
```

Mapovanie pre objednávku:

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Order" table="BOOK_ORDER">
    ...
    <many-to-one name="weekdayContact"
                 class="Contact" column="CONTACT_ID"
                 unique="true" />
    <many-to-one name="holidayContact"
                 class="Contact" column="CONTACT_ID"
                 unique="true" />
  </class>
</hibernate-mapping>
```

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.

Takéto triedy sa v Hibernate nazývajú *komponentami*.

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Order" table="BOOK_ORDER">
    ...
    <component name="weekdayContact" class="Contact">
      <property name="recipient"
                type="string" column="WEEKDAY_RECIPIENT" />
      <property name="phone" type="string" column="WEEKDAY_PHONE" />
      <property name="address" type="string" column="WEEKDAY_ADDRESS" />
    </component>

    <component name="holidayContact" class="Contact">
      <property name="recipient"
                type="string" column="HOLIDAY_RECIPIENT" />
      <property name="phone" type="string" column="HOLIDAY_PHONE" />
      <property name="address" type="string" column="HOLIDAY_ADDRESS" />
```

```
    </component>
  </class>
</hibernate-mapping>
```

Všimnime si, že nemapujeme žiadnu novú triedu. 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.

7.2. 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.

```
public class Phone {
  private String areaCode; //predvoľba
  private String telNo;    //telefón

  // gettre a settre
}

public class Contact {
  private String phone;
  private Phone phone;

  // gettre a settre
}
```

Mapovanie:

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Order" table="BOOK_ORDER">
    ...
    <component name="weekdayContact" class="Contact">
      <property name="recipient"
                type="string" column="WEEKDAY_RECIPIENT" />
      <property name="phone"
                type="string" column="WEEKDAY_PHONE" />
      <property name="phone" type="string" column="WEEKDAY_PHONE" />
      <component name="phone" class="Phone">
        <property name="areaCode"
                  type="string" column="WEEKDAY_PHONE_AREA_CODE" />
        <property name="telNo"
```

```
                    type="string" column="WEEKDAY_PHONE_TEL_NO" />
      </component>

      <property name="address" type="string" column="WEEKDAY_ADDRESS" />
    </component>

    <component name="holidayContact" class="Contact">
      ...
    </component>
  </class>
</hibernate-mapping>
```

7.3. Odkazovanie sa na iné triedy v komponentoch

Odkaz na rodiča

Ak chceme evidovať v komponente odkaz na jeho rodiča, môžeme použiť element `<parent>`.

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

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

  // gettre a settre
}
```

Mapovanie bude vyzerať nasledovne

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Order" table="BOOK_ORDER">
    ...
    <component name="weekdayContact" class="Contact">

      <!-- odkaz na rodiča Order -->
      <parent name="order" />

      <property name="recipient"
                type="string" column="WEEKDAY_RECIPIENT" />

      <component name="phone" class="Phone">
        <property name="areaCode"
                  type="string" column="WEEKDAY_PHONE_AREA_CODE" />
        <property name="telNo" type="string"
```

```
                  column="WEEKDAY_PHONE_TEL_NO" />
      </component>

      <property name="address" type="string" column="WEEKDAY_ADDRESS" />
    </component>
  </class>
</hibernate-mapping>
```

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.

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

  // gettre a settre
}
```

Mapovanie bude vyzerať nasledovne:

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Order" table="BOOK_ORDER">
    ...
    <component name="weekdayContact" class="Contact">
      ...
      <property name="address" type="string" column="WEEKDAY_ADDRESS" />
      <many-to-one name="address"
                   class="Address" column="WEEKDAY_ADDRESS_ID" />
    </component>
  </class>
</hibernate-mapping>
```

7.4. 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 {
  ...
  private Contact weekdayContact;
  private Contact holidayContact;
  private Set contacts;

  // gettre a settre
}
```

Na mapovanie množiny komponentov je v Hibernate k dispozícii element `<composite-element>`. Pre jednoduchosť budeme používať predošlú verziu triedy `Contact`:

```
public class Contact {
  private String recipient;
  private String phone;
  private String address;
  // gettre a settre
}
```

Mapovanie bude vyzerať nasledovne:

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Order" table="BOOK_ORDER">
    ...
    <set name="contacts" table="ORDER_CONTACT">
      <key column="ORDER_ID" />
      <composite-element class="Contact">
        <property name="recipient" type="string" column="RECIPIENT" />
        <property name="phone" type="string" column="PHONE" />
        <property name="address" type="string" column="ADDRESS" />
      </composite-element>
    </set>
  </class>
</hibernate-mapping>
```

Okrem kolekcií komponentov môžeme používať aj kolekciu vnorených komponentov a prípadných asociácií. Vnorené komponenty v kolekcii namapujeme pomocou `<nested-composite-element>`.

Triedu `Contact` môžeme teda upraviť nasledovne:

```
public class Contact {
```

```
  private String recipient;
  private Phone phone;
  private Address address;
  // gettre a settre
}
```

Mapovanie bude vyzerať nasledovne:

```
<hibernate-mapping package="mo.org.cpttm.bookshop"
  <class name="Order" table="BOOK_ORDER">
    ...
    <set name="contacts" table="ORDER_CONTACT">

      <key column="ORDER_ID" />

      <composite-element class="Contact">
        <property name="recipient" type="string" column="RECIPIENT" />
        <property name="phone" type="string" column="PHONE" />

        <nested-composite-element name="phone" class="Phone">
          <property name="areaCode"
                    type="string" column="PHONE_AREA_CODE" />
          <property name="telNo" type="string" column="PHONE_TEL_NO" />
        </nested-composite-element>

        <property name="address" type="string" column="ADDRESS" />

        <many-to-one name="address"
                     class="Address" column="ADDRESS_ID" />
      </composite-element>
    </set>
  </class>
</hibernate-mapping>
```

7.5. Používanie komponentov ako kľúčov v mape

Predpokladajme, že chceme rozšíriť našu kolekciu kontaktov na typ `java.util.Map`, kde použijeme časové obdobia ako kľúče v mape. Zákazník si potom môže určiť najvhodnejší kontakt pre príslušné časové obdobie. Vytvoríme triedu pre obdobie `Period`, ktorá bude obsahovať počiatočný a koncový dátum obdobia. Táto trieda bude používaná ako kľúč do mapy a namapujeme ju´ako komponent. Nemá totiž zmysel, aby to bola samostatná perzistentná trieda s identifikátorom.

```
public class Period {
  private Date startDate;
```

```
  private Date endDate;

  // gettre a settre
}
```

Definícia mapovania:

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Order" table="BOOK_ORDER">
    ...
    <map name="contacts" table="ORDER_CONTACT">

      <key column="ORDER_ID" />

      <composite-map-key class="Period">
        <key-property name="startDate" type="date"
          column="START_DATE" />
        <key-property name="endDate" type="date"
          column="END_DATE" />
      </composite-map-key>

      <composite-element class="Contact">
        <property name="recipient" type="string" column="RECIPIENT" />

        <nested-composite-element name="phone" class="Phone">
          <property name="areaCode"
                    type="string" column="PHONE_AREA_CODE" />

          <property name="telNo" type="string" column="PHONE_TEL_NO" />
        </nested-composite-element>

        <many-to-one name="address"
                     class="Address" column="ADDRESS_ID" />
      </composite-element>
    </map>
  </class>
</hibernate-mapping>
```

Ak chceme zaručiť správne fungovanie objektu `Period` v úlohe kľúča v mape, musíme prekryť metódy `equals()` a `hashCode()`.

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

```
      return new EqualsBuilder()
        .append(startDate, other.startDate)
        .append(endDate, other.endDate)
        .isEquals();
    }

    public int hashCode() {
      return new HashCodeBuilder()
        .append(startDate)
        .append(endDate)
        .toHashCode();
    }
  }
```

Alternatívne môžeme použiť zabudovanú funkciu v Eclipse **Source | Generate hashCode() and equals().**

8. Mapovanie dedičnosti

8.1. Dedičnosť a polymorfizmus

Skúsme si rozšíriť naše internetové kníhkupectvo o predaj CDčiek a DVDčiek. Vytvoríme najprv triedu pre disk (`Disc`) a definíciu mapovania.

```
public class Disc {
  private Long id;
  private String name;
  private Integer price;

    // gettre a settre
}
```

Definícia mapovania:

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Disc" table="DISC">
    <id name="id" type="long" column="ID">
      <generator class="native"/>
    </id>

    <property name="name" type="string" column="NAME" />

    <property name="price" type="int" column="PRICE" />
```

```
</class>
</hibernate-mapping>
```

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 zareprezentujeme 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.

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*).

```
Session session = factory.openSession();
try {
  Query query = session.createQuery("from Disc");
  List discs = query.list();
  return discs;
```

```
} finally {
  session.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*).

```
public class Reservation {
  private Long id;

  private Disc disc;

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

Definícia mapovania bude vyzerať nasledovne:

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Reservation" table="RESERVATION">
    <id name="id" type="long" column="ID">
      <generator class="native" />
    </id>

    <!-- viac rezervacii k jednemu disku -->
    <many-to-one name="disc" class="Disc" column="DISC_ID" />

    ...
  </class>
</hibernate-mapping>
```

8.2. Mapovanie dedičnosti

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

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

Tento prístup bude používať jedinú tabuľku, 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.

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Disc" table="Disc">
```

```
    <id name="id" type="long" column="ID">
      <generator class="native"/>
    </id>

    <discriminator column="DISC_TYPE" type="string" />
    ...

    <subclass name="AudioDisc" discriminator-value="AUDIO">
      <property name="singer" type="string" column="SINGER" />
      <property name="numOfSongs" type="int" column="NUM_OF_SONGS" />
    </subclass>

    <subclass name="VideoDisc" discriminator-value="VIDEO">
      <property name="director" type="string" column="DIRECTOR" />
      <property name="language" type="string" column="LANGUAGE" />
    </subclass>

  </class>
</hibernate-mapping>
```

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 ne`null`ovosť.

*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.

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Disc" table="DISC">
    ...
    <joined-subclass name="AudioDisc" table="AUDIO_DISC">
      <key column="DISC_ID" />
      <property name="singer" type="string" column="SINGER" />
      <property name="numOfSongs" type="int" column="NUM_OF_SONGS" />
    </joined-subclass>

    <joined-subclass name="VideoDisc" table="VIDEO_DISC">
      <key column="DISC_ID" />
```

```
      <property name="director" type="string" column="DIRECTOR" />
      <property name="language" type="string" column="LANGUAGE" />
    </joined-subclass>

  </class>
</hibernate-mapping>
```

Tento prístup nezakazuje používanie `NOT NULL` stĺpcov, ale je zase menej efektívny. Pri načítavaní objektu sa totiž musí vykonať viacero dopytov s *join*ami. Ak sa používajú polymorfné dopyty a asociácie, musí sa vzájomne poprepájať viacero tabuliek.

*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.

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Disc" table="DISC">

    <id name="id" type="long" column="ID">
      <generator class="sequence">
        <param name="sequence">DISC_SEQUENCE</param>
      </generator>
    </id>
    ...
    <union-subclass name="AudioDisc" table="AUDIO_DISC">
      <property name="singer"     type="string" column="SINGER" />
      <property name="numOfSongs" type="int"    column="NUM_OF_SONGS" />
    </union-subclass>

    <union-subclass name="VideoDisc" table="VIDEO_DISC">
      <property name="director" type="string" column="DIRECTOR" />
      <property name="language" type="string" column="LANGUAGE" />
    </union-subclass>

  </class>
```

```
</hibernate-mapping>
```

9. Dopytovací jazyk Hibernate

9.1. 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 Hibernate vieme väčšinu aktualizačných úkonov realizovať pomocou poskytovaných tried a metód (`save()`, `saveOrUpdate` 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 *Hibernate Query Language* (HQL).

HQL je databázovo nezávislý jazyk, ktorý sa počas behu prekladá do SQL. Samotná syntax HQL je zameraná na prácu s objektami a ich atribútmi, pričom sa snaží odtieniť používateľa od pojmov používaných v databázach. Na HQL sa dá dívať ako na 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 query = session.createQuery("from Book");
List books = query.list();
```

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 = session.createQuery("from Book");
query.setFirstResult(20);
query.setMaxResults(10);

List books = query.list();
```

Ďalším užitočným nastavením v dopyte je *fetch size*, ktorý indikuje JDBC ovládaču počet riadkov, ktorý sa má získať vo výsledku v rámci jednej požiadavky na databázu.

```
Query query = session.createQuery("from Book");
query.setFetchSize(100);
```

```
List books = query.list();
```

Namiesto metódy `list()` je niekedy výhodné použitie metódy `uniqueResult()`, ktorá namiesto zoznamu vráti len jeden objekt (v prípade ak sme si istí, že výsledný zoznam bude mať len jeden prvok). Ak je výsledok prázdny, metóda vráti `null`.

V HQL dopytoch môžeme používať parametre a to rovnakým spôsobom ako v prípade SQL dopytov.

```
Query query = session.createQuery("from Book where isbn = ?");
query.setString(0, "1932394419");

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

V uvedenom príklade sme použili otáznik `?` ako zástupný symbol pre parameter a jeho hodnotu sme nastavili pomocou indexu (indexy parametrov v HQL začínajú od nuly na rozdiel od JDBC, kde začínajú jednotkou). 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 = session.createQuery("from Book where isbn = :isbn");
query.setString("isbn", "1932394419");

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

V ďalších sekciách sa budeme venovať jazyku HQL podrobnejšie. Pri skúšaní dopytov odporúčame sledovať vygenerované SQL dopyty. Niekedy nám to pomôže pri optimalizovaní HQL dopytov.

9.2. Klauzula `from`

Základnou klauzulou a jedinou povinnou časťou dopytov v HQL je `from`. Nasledovný HQL dopyt získa všetky knihy, ktorých názov obsahuje je `Hibernate Quickly`. Všimnite si, že `name` je názvom atribútu knihy a nie databázovým stĺpcom.

```
from Book
where name = 'Hibernate Quickly'
```

Podobne ako v SQL je možné používať aliasy, čo sprehľadňuje situáciu v prípade dopytovania sa na viacero objektov. Pri pomenovaniach by sme mali dodržiavať menné konvencie známe z Javy (názvy tried začínajú prvým veľkým písmenom, názvy inštancií malým). Kľúčové slovo `as` je nepovinné.

```
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`).

```
from Book book, Publisher publisher
```

9.3. Dopytovanie sa na asociované objekty

V HQL 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. Každý dvojica bude obsahovať objekt knihy a objekt vydavateľa.

```
from Book book join book.publisher 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.

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

*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*).

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

```
from Book book
where book.publisher.name = 'Manning'
```

```
from Book book
where book.chapters.title = 'Hibernate Basics'
```

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.

```
from Book book
where book.chapters.title = 'Hibernate Basics'
and book.chapters.numOfPages = 25
```

Preto treba postupovať pri implicitných asociáciách kolekcií opatrne. Kolekcia, na ktorú sa v dopyte odkazujeme viackrát, by mala byť v dopyte asociovaná explicitne.

```
from Book book join book.chapters chapter
where chapter.title = 'Hibernate Basics' and chapter.numOfPages = 25
```

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.

```
from Book book join book.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`.

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

Okrem tohto typu existuje ešte *pravé spojenie* (*right join*) a *úplné spojenie* (*full join*). Ich význam je analogický ako v SQL a používajú sa len zriedka.

Ignorovanie duplicitných objektov vo výsledku

Nasledovný HQL dopyt môže byť použitý na získanie kníh a asociovaných kapitol, kde platí, že aspoň jedna kapitola knihy v sebe obsahuje *Hibernate*. Vo výsledku budú dvojice kníh a kapitol

```
from Book book join book.chapters chapter
where chapter.title like '%Hibernate%'
```

Ak použijeme modifikáciu s implicitným napojením, vo výsledku sa zjavia len objekty kníh. Ale na veľké prekvapenie zistíme, že zoznam obsahuje duplicitné knihy. Počet duplikátov bude závisieť od počtu kapitol, ktoré majú v názve *Hibernate*.

```
from Book book
where book.chapters.title like '%Hibernate%'
```

Podľa dokumentácie je toto správanie normálne, pretože počet prvkov vo výsledku je zhodný s počtom riadkov vo výsledku vrátenom databázou v `ResultSet`e. Na odfiltrovanie duplicitných objektov so zachovaním poradia prvkov v zozname môžeme použiť triedu `LinkedHashSet`.

```
Query query = session.createQuery(
       "from Book book where book.chapters.title like '%Hibernate%'");
List books = query.list();
Set uniqueBooks = new LinkedHashSet(books);
```

Asociácie v HQL 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.

```
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:

```
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`.

```
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 pomocoui `join fetch` sa zahrnú už do prvého SQL dopytu.

9.4. Klauzula `where`

V HQL 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`.

```
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.

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

Vo `where` klauzule je možné používať aj implicitné napájanie objektov. Ako už bolo spomenuté, pri viacnásobnom použití jednej kolekcie je často výhodnejšie použiť explicitné napojenie, čím sa predíde duplicitnému napájaniu rovnakých tabuliek v SQL.

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

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

Hibernate poskytuje možnosť pre zistenie veľkosti kolekcie. Jestvujú dva prístupy: buď použitím špeciálneho atribútu `size` alebo použitím funkcie `size()`. Hibernate použije na zistenie veľkosti kolekcie SQL poddopyt `SELECT COUNT(...)`.

```
from Book book
where book.chapters.size > 10

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

9.5. Klauzula `select`

V predošlých príkladoch sme sa dopytovali na celé inštancie perzistentných objektov. V prípade potreby sa však môžeme dopytovať aj na jednotlivé atribúty. Slúži na to klauzula `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
```

Výsledky je tiež možné poskytovať v podobe kolekcií, teda zoznamov a máp. Výsledkom takéhoto dopytu bude zoznam kolekcií:

```
select new list(book.isbn, book.name, book.publisher.name)
from Book book
```

V prípade mapy potrebujeme špecifikovať pre každý atribút pomocou kľúčového slova `as` kľúč do mapy.

```
select new map(
  book.isbn as bookIsbn, book.name as bookName,
  book.publisher.name as publisherName)
from Book book
```

9.6. 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`).

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

V HQL 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
```

9.7. Poddopyty

V HQL 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`.

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

9.8. Pomenované dopyty

Dopyty v HQL 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 ktorejkoľvek definícii mapovania. Kvôli väčšej prehľadnosti je však odporúčané uviesť všetky pomenované dopyty na jednom mieste, napr. v definícii mapovania `NamedQuery.hbm.xml`. Ak je ich veľmi veľa, je lepšie ich logicky kategorizovať do viacerých súborov. Odporúča sa tiež zaviesť jednotný postup pre pomenovávani dopytov.

Každý pomenovaný dopyt musí mať jednoznačné meno. Ak chceme v znení dopytu používať znaky, ktoré by mohli byť v konflikte so špeciálnymi znakmi v XML, môžeme použiť blok `<![CDATA[ ... ]]>`.

```
<hibernate-mapping>
  <query name="Book.by.isbn">
    <![CDATA[
       from Book where isbn = ?
   ]]>
  </query>
</hibernate-mapping>
```

Ak sa chceme odkazovať v kóde na pomenovaný dopyt, použijeme metódu `getNamedQuery()` na objekte sedenia:

```
Query query = session.getNamedQuery("Book.by.isbn");
query.setString(0, "1932394419");
Book book = (Book) query.uniqueResult();
```

10. Kritériové dopyty

10.1. Používanie kritériových dopytov

V predošlej kapitole sme si ukázali použitie jazyka HQL na dopytovanie sa po perzistentných objektoch. Hibernate poskytuje k HQL alternatívu nazývanú *kritériové dopyty* (*criteria queries*). V tomto prípade ide o sadu tried a metód, pomocou ktorých vieme vybudovať dopyt spôsobom blízkym objektovo orientovanému programovaniu.

Niekedy je totiž potrebné vybudovať dopyt dynamicky (napr. v prípade pokročilého vyhľadávania či filtrovania). Tradičným spôsobom je vybudovanie dopytu HQL

alebo SQL spájaním reťazcov tvoriacich čiastkové dopyty. Problémom tohto prístupu je náročnejšia udržovateľnosť, pretože fragmenty dopytu je niekedy ťažké čítať a upravovať.

Príkladom kritériových dopytov je nasledovný kód:

```
Criteria criteria = session.createCriteria(Book.class)
criteria.add(Restrictions.eq("name", "Hibernate Quickly"));
List books = criteria.list();
```

Ten zodpovedá dopytu v HQL:

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

Väčšina metód v triede `Criteria` vracia inštanciu samého seba, takže výsledný objekt môžeme vybudovať nasledovnou skrátenou formou:

```
Criteria criteria = session.createCriteria(Book.class)
                     .add(Restrictions.eq("name", "Hibernate Quickly"));
List books = criteria.list();
```

10.2. Reštrikcie

Do kritériového dopytu vieme pridať jednu alebo viac reštrikcií, ktorými odfiltrujeme nežiadúce objekty. Ide o analógiu `where` klauzuly v HQL. Trieda `Restrictions` ponúka množstvo metód na tvorbu a skladanie reštrikcií. Viacero pridaných reštrikcií bude považovaných za elementy v logickej konjunkcii.

```
Criteria criteria = session.createCriteria(Book.class)
                     .add(Restrictions.like("name", "%Hibernate%"))
                     .add(Restrictions.between("price", 100, 200));
List books = criteria.list();
```

Uvedený kritériový dopyt zodpovedá nasledovnému dopytu v HQL:

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

V prípade potreby môžeme používať reštrikcie oddelené logickou disjunkciou:

```
Criteria criteria = session.createCriteria(Book.class)
                      .add(Restrictions.or(
                            Restrictions.like("name", "%Hibernate%"),
                            Restrictions.like("name", "%Java%")
                           )
```

```
                        )
                       .add(Restrictions.between(
                              "price",
                              100,
                              200));
List books = criteria.list();
```

Zodpovedajúci dopyt v HQL je nasledovný:

```
from Book book
where (book.name like '%Hibernate%' or book.name like '%Java%')
      and (book.price between 100 and 200)
```

10.3. Asociácie

V HQL sme sa mohli odkazovať na atribút asociovaného objektu priamo, čím sme vyvolali implicitný join. Platí rovnaká vec aj pre kritériové dopyty?

```
Criteria criteria = session.createCriteria(Book.class)
  .add(Restrictions.like("name", "%Hibernate%"))
  .add(Restrictions.eq("publisher.name", "Manning"));
List books = criteria.list();
```

Ak sa pokúsite vykonať vyššieuvedený kód, Hibernate vyhodí výnimku s hlásením, že nie je možné zistiť hodnotu atribútu `publisher.name`. Dôvodom je to, že v kritériových dopytoch nie je možné používať implicitné joiny. Ak sa chcete dopytovať na atribút v asociovanom objekte kritériom, potrebujete pre tento atribút vytvoriť nový kritériový poddopyt.

```
Criteria criteria = session.createCriteria(Book.class)
  .add(Restrictions.like("name", "%Hibernate%"))
  .createCriteria("publisher")
    .add(Restrictions.eq("name", "Manning"));

List books = criteria.list();
```

Tento kritérový dopyt zodpovedá HQL dopytu:

```
from Book book
where book.name like '%Hibernate%' and book.publisher.name = 'Manning'
```

V kritériových dopytoch je tiež možné dynamicky špecifikovať režim pre *fetch*:

```
Criteria criteria = session.createCriteria(Book.class)
  .add(Restrictions.like("name", "%Hibernate%"))
  .setFetchMode("publisher", FetchMode.JOIN)
  .setFetchMode("chapters", FetchMode.SELECT);
List books = criteria.list();
```

10.4. Projekcie

Ak chcete vo výsledku získať len niektoré atribúty, môžete na to použiť triedu Projections.

```
Criteria criteria = session.createCriteria(Book.class)
  .setProjection(Projections.property("name"));

List books = criteria.list();
```

čo je analogické HQL dopytu:

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

Trieda Projections v sebe obsahuje aj statické metódy na prístup k agregačným funkciám. Nasledovný dopyt vráti priemernú cenu knihy:

```
Criteria criteria = session.createCriteria(Book.class)
  .setProjection(Projections.avg("price"));
List books = criteria.list();
```

Analogický HQL dopyt je

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

10.5. Triedenie a zoskupovanie

Výsledky dopytu je možné dynamicky triediť, či už vzostupne alebo zostupne. Dopyt vráti knihy zotriedené podľa názvu a potom podľa dátumu vydania:

```
Criteria criteria = session.createCriteria(Book.class)
  .addOrder(Order.asc("name"))
  .addOrder(Order.desc("publishDate"));

List books = criteria.list();
```

Analogickým dopytom v HQL je

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

Kritériové dopyty poskytujú ďalej podporu pre zoskupovanie (*grouping*). Je možné zoskupiť niekoľko atribútov naraz, čo sa prejaví v GROUP BY klauzule vo výslednom SQL dopyte.

Nasledovný dopyt vráti zoznam dvojprvkových polí s priemernou cenou kníh vydaných v daný dátum.

```
Criteria criteria = session.createCriteria(Book.class)
  .setProjection(Projections.projectionList()
    .add(Projections.groupProperty("publishDate"))
    .add(Projections.avg("price"))
);

List books = criteria.list();
```

Analogickým dopytom v HQL je

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

10.6. Dopytovanie vzorovými objektami (*querying by example*)

Kritériové dopyty môžu byť konštruované i na základe vzorového objektu. Takýto objekt by mal byť inštanciou perzistentnej triedy, na ktorú sa chcete dopytovať. Všetky `null`ové atribúty sú štandardne ignorované.

```
Book book = new Book();
book.setName("Hibernate");
book.setPrice(100);

Example exampleBook = Example.create(book);
Criteria criteria = session.createCriteria(Book.class)
  .add(exampleBook);

List books = criteria.list();
```

Analogickým dopytom v HQL je

```
from Book book
where book.name = 'Hibernate' and book.price = 100
```

Vo vzorových objektov je možné vynechať neželané atribúty či použiť zástupné znaky známe z SQL funkcie `LIKE`.

```
Book book = new Book();
book.setName("%Hibernate%");
book.setPrice(100);

Example exampleBook = Example.create(book)
  .excludeProperty("price") //ignorujeme cenu
  .enableLike();            //povolíme zástupné znaky v LIKE
```

```
Criteria criteria = session.createCriteria(Book.class)
  .add(exampleBook);

List books = criteria.list();
```

To zodpovedá HQL dopytu:

```
from Book book
where book.name like '%Hibernate%'
```

11. Dávkové spracovanie a natívne SQL

11.1. Dávkové spracovanie v HQL

V našom internetovom kníhkupectve sa plánuje veľká reklamná akcia. Všetky knihy, ktoré obsahujú v názve *Hibernate* sa zlacnia o 10 dolárov. Podľa dosiaľ dosiahnutých vedomostí by bolo jedným z riešení získanie zoznamu všetkých kníh z databázy, postupnému prechádzaniu tohto zoznamu, úprave každej knihy a jej aktualizácii v databáze.

```
Session session = factory.openSession();
Transaction tx = null;
try {
  tx = session.beginTransaction();

  Query query = session.createQuery("from Book where name like ?");
  query.setString(0, "%Hibernate%");

  List<Book> books = (List<Book>) query.list();
  for(Book book : books) {
    Book book = (Book) iter.next();
    book.setPrice(new Integer(book.getPrice().intValue() -10));
    session.saveOrUpdate(book);
  }
  tx.commit();
} catch (HibernateException e) {
  if (tx != null) tx.rollback();
  throw e;
} finally {
  session.close();
}
```

Je zrejme vidieť, že takýto spôsob aktualizácie dát je značne neefektívny. V prípade klasického SQL by sme neodosielali pre každú knihu jeden aktualizačný dopyt, ale použili by sme jediný hromadný `UPDATE`. V HQL je možný analogický postup s podobnou syntaxou, namiesto názvov stĺpcov však použijeme názvy atribútov objektov (`price` je atribútom zodpovedajúcim cene v triede `Book`).

```
Session session = factory.openSession();
Transaction tx = null;
try {
  tx = session.beginTransaction();
  Query query = session.createQuery(
    "update Book set price = price - ? where name like ?");
  query.setInteger(0, 10);
  query.setString(1, "%Hibernate%");

  int count = query.executeUpdate();

  tx.commit();
} catch (HibernateException e) {
  if (tx != null) tx.rollback();
  throw e;
} finally {
session.close();
}
```

Popri dávkovej aktualizácii je možné realizovať aj dávkové odstraňovanie objektov pomocou `delete`. Poznamenajme, že kľúčové slovo `from` je nepovinné.

```
Session session = factory.openSession();
Transaction tx = null;
try {
  tx = session.beginTransaction();

  Query query = session.createQuery(
                  "delete from Book where name like ?");
  query.setString(0, "%Hibernate%");
  int count = query.executeUpdate();

  tx.commit();
} catch (HibernateException e) {
  if (tx != null) tx.rollback();
  throw e;
} finally {
  session.close();
}
```

12. Dopyty s natívnym SQL

Predstavme si situáciu, v ktorej chceme získať z dát rôzne štatistické údaje — napr. najpredávanejšie knihy. Môže sa však stať, že tento proces je komplexný alebo je založený na vlastnostiach a postupoch striktne zviazaných s konkrétnou databázou. V tomto prípade nám čisté HQL nepomôže a budeme sa musieť *znížiť* na úroveň SQL a dopytovať sa pomocou neho.

V našom príklade nebudeme demonštrovať tieto zložité štatistické výpočty. Pre jednoduchosť si vytvoríme pohľad (*view*) s výsledkami dopytu. Pohľad `TOP_SELLING_BOOK` (najpredávanejšie knihy) bude obsahovať tabuľku `BOOK` spojenú s tabuľkou `PUBLISHER`.

```
CREATE VIEW TOP_SELLING_BOOK (
  ID, ISBN, BOOK_NAME, PUBLISH_DATE, PRICE,
  PUBLISHER_ID, PUBLISHER_CODE, PUBLISHER_NAME, PUBLISHER_ADDRESS
) AS
SELECT
  book.ID, book.ISBN, book.BOOK_NAME, book.PUBLISH_DATE, book.PRICE,
  book.PUBLISHER_ID, pub.CODE, pub.PUBLISHER_NAME, pub.ADDRESS
FROM BOOK book
LEFT OUTER JOIN PUBLISHER pub ON book.PUBLISHER_ID = pub.ID
```

Skúsme najprv vytiahnuť z databázy len inštancie kníh (a na chvíľu ignorovať asociovaných vydavateľov). Na to použijeme SQL, ktoré vráti vo výsledku stĺpce týkajúce sa knihy. Metóda `addEntity()` slúži na špecifikovanie dátového typu pre objekty vo výsledku.

```
String sql = "SELECT  ID, ISBN, BOOK_NAME,
                      PUBLISH_DATE, PRICE, PUBLISHER_ID
              FROM    TOP_SELLING_BOOK
              WHERE   BOOK_NAME LIKE ?";

Query query = session.createSQLQuery(sql)
                     .addEntity(Book.class)
                     .setString(0, "%Hibernate%");

List books = query.list();
```

Keďže všetky stĺpce vo výsledku majú rovnaké názvy ako názvy stĺpcov v definícii mapovania knihy, môžeme v klauzule `select` použiť jednoduchší zápis `{book.*}`. Hibernate nahradí symbol `{book.*}` zoznamom stĺpcov z definície mapovania knihy.

```
String sql = "SELECT {book.*}
              FROM   TOP_SELLING_BOOK book
              WHERE  BOOK_NAME LIKE ?";

Query query = session.createSQLQuery(sql)
                     .addEntity("book", Book.class)
                     .setString(0, "%Hibernate%");

List books = query.list();
```

Skúsme teraz brať do úvahy aj asociovaných vydavateľov. Keďže nie všetky názvy stĺpcov v pohľade sú zhodné s názvami v definícii mapovania, musíme ich namapovať v `SELECT` dopyte explicitne pomocou aliasov. Metóda `addJoin()` je použitá na špecifikovanie asociácie s napojeným objektom.

```
String sql = "SELECT {book.*},
                     book.PUBLISHER_ID as {publisher.id},
                     book.PUBLISHER_CODE as {publisher.code},
                     book.PUBLISHER_NAME as {publisher.name},
                     book.PUBLISHER_ADDRESS as {publisher.address}
              FROM   TOP_SELLING_BOOK book
              WHERE  BOOK_NAME LIKE ?";

Query query = session.createSQLQuery(sql)
                     .addEntity("book", Book.class)
                     .addJoin("publisher", "book.publisher")
                     .setString(0, "%Hibernate%");

List books = query.list();
```

Ak sa chceme dopytovať len na jednoduché hodnoty, použijeme na určenie dátového typu výsledku metódu `addScalar()`.

```
String sql = "SELECT max(book.PRICE) as maxPrice
              FROM TOP_SELLING_BOOK book
              WHERE BOOK_NAME LIKE ?";

Query query = session.createSQLQuery(sql)
                     .addScalar("maxPrice", Hibernate.INTEGER)
                     .setString(0, "%Hibernate%");

Integer maxPrice = (Integer) query.uniqueResult();
```

12.1. Pomenované SQL dopyty

Dosiaľ sme naše dopyty uvádzali priamo v kóde. Alternatívnym miestom je definícia mapovania, v ktorej uvedieme dopyt pod vhodným logickým označením, na ktoré sa budeme odkazovať. Dátový typ pre návratovú hodnotu a špecifikáciu pripojených objektov môžeme nastaviť pomocou elementov `<return>` a `<return-join>`.

```
<hibernate-mapping package="mo.org.cpttm.bookshop">

  <sql-query name="TopSellingBook.by.name">
    <!-- návratovou hodnotou je Book, pričom alias tabuľky
         s atribútmi je book -->
    <return alias="book" class="Book" />
    <!-- alias tabuľky s asociovanými stĺpcami je book
         a budeme ho mapovať na atribút book.publisher -->
    <return-join alias="book" property="book.publisher"/>
      <![CDATA[

       SELECT  {book.*},
               book.PUBLISHER_ID as {publisher.id},
               book.PUBLISHER_CODE as {publisher.code},
               book.PUBLISHER_NAME as {publisher.name},
               book.PUBLISHER_ADDRESS as {publisher.address}
       FROM    TOP_SELLING_BOOK book
       WHERE   BOOK_NAME LIKE ?
       ]]>

  </sql-query>
</hibernate-mapping>
```

V kóde sa potom na dopyt odkážeme nasledovným spôsobom:

```
Query query = session.getNamedQuery("TopSellingBook.by.name")
                     .setString(0, "%Hibernate%");
List books = query.list();
```

Pomenovávať môžeme aj dopyty vracajúce jednoduché hodnoty. V tomto prípade špecifikujeme návratovú hodnotu v elemente `<return-scalar>`.

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  ...
  <sql-query name="TopSellingBook.maxPrice.by.name">
    <return-scalar column="maxPrice" type="int" />
```

```
    <![CDATA[
     SELECT max(book.PRICE) as maxPrice
     FROM TOP_SELLING_BOOK book
     WHERE BOOK_NAME LIKE ?
   ]]>
  </sql-query>
</hibernate-mapping>
```

Príklad Java kódu:

```
Query query = session.getNamedQuery("TopSellingBook.maxPrice.by.name")
                     .setString(0, "%Hibernate%");
Integer maxPrice = (Integer) query.uniqueResult();
```

V pomenovanom dopyte môžeme zoskupiť elementy `<return>` a `<return-join>` do tzv. *mapovania výsledku* (*result set mapping* reprezentovanom elementom `<resultset>`. Na toto mapovanie výsledku sa potom môžeme odkazovať z viacerých dopytov (namiesto jeho explicitného uvádzania v každom dopyte zvlášť). Odkazovaný dopyt uvedieme ako hodnotu atribútu v elemente `resultset-ref`.

```
<hibernate-mapping package="mo.org.cpttm.bookshop">

  <resultset name="bookPublisher">
    <return alias="book" class="Book" />
    <return-join alias="book" property="book.publisher" />
  </resultset>
  <sql-query name="TopSellingBook.by.name"
             resultset-ref="bookPublisher">
    <![CDATA[
     SELECT  {book.*},
             book.PUBLISHER_ID as {publisher.id},
             book.PUBLISHER_CODE as {publisher.code},
             book.PUBLISHER_NAME as {publisher.name},
             book.PUBLISHER_ADDRESS as {publisher.address}
     FROM  TOP_SELLING_BOOK book
     WHERE BOOK_NAME LIKE ?
     ]]>
  </sql-query>
</hibernate-mapping>
```

Vo výslednom mapovaní výsledku môžeme ďalej namapovať každý databázový stĺpec na atribút objektu. To by zjednodušilo náš SQL dopyt o časť, v ktorej mapujeme stĺpce na atribúty vydavateľa.

```
<hibernate-mapping package=”mo.org.cpttm.bookshop”>
  <resultset name="bookPublisher">
    <return alias="book" class="Book" />
    <return-join alias="book" property="book.publisher">
        <!-- PUBLISHER_ID sa namapuje na atribút id -->
      <return-property name="id" column="PUBLISHER_ID" />
      <return-property name="code" column="PUBLISHER_CODE" />
      <return-property name="name" column="PUBLISHER_NAME" />
      <return-property name="address" column="PUBLISHER_ADDRESS" />
    </return-join>
  </resultset>
  <sql-query name="TopSellingBook.by.name"
           resultset-ref="bookPublisher">
    <![CDATA[
     SELECT  {book.*},
         book.PUBLISHER_ID as {publisher.id},
         book.PUBLISHER_CODE as {publisher.code},
         book.PUBLISHER_NAME as {publisher.name},
         book.PUBLISHER_ADDRESS as {publisher.address}

     FROM    TOP_SELLING_BOOK book
     WHERE   BOOK_NAME LIKE ?
    ]]>

  </sql-query>
</hibernate-mapping>
```

Ak v mapovaní výsledku uvedieme mapovanie všetkých stĺpcov, klauzulu `SELECT` môžeme zjednodušiť na formu `SELECT *`.

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  ...
  <sql-query name="TopSellingBook.by.name"
       resultset-ref="bookPublisher">
    <![CDATA[

     SELECT *
     FROM TOP_SELLING_BOOK book
     WHERE BOOK_NAME LIKE ?

    ]]>
  </sql-query>
</hibernate-mapping>
```

13. Cachovanie objektov

13.1. Úrovne cacheovania

Základnou ideou cacheovania perzistentných objektov je ukladanie inštancií, ktoré boli načítané z databázy, do medzipamäte (**cache**). Každé ďalšie načítanie objektu potom nevyžaduje prístup k databáze, ale len jeho získanie z medzipamäte cache. Cache sú väčšinou uložené v pamäti alebo na lokálnom disku a prístup k nim je rýchlejší než k databáze. V prípade správneho používania môže cache značne zvýšiť výkon našej aplikácie.

Keďže Hibernate je vysokovýkonným nástrojom objektovo-relačného mapovania, podpora cacheovania objektov nie je ničím prekvapivým a to dokonca na viacerých úrovniach. Vyskúšajme si nasledovný príklad: získajme v rámci jedného sedenia objekt s rovnakým identifikátorom dvakrát. Koľkokrát pristúpi Hibernate k databáze?

```
Session session = factory.openSession();
try {
  Book book1 = (Book) session.get(Book.class, id);
  Book book2 = (Book) session.get(Book.class, id);
} finally {
  session.close();
}
```

Ak si prezrieme SQL dopyty odosielané do databázy, zistíme, že sa vykoná len jeden dopyt. Hibernate načítal objekt a uložil ho do cache. Pri druhom prístupe už nepristupoval k databáze, ale na získanie daného objektu bola použitá cache. Tento spôsob cachovania reprezentuje cache prvej úrovne, ktorá je platná v rámci jedného sedenia. Čo však v prípade prístupu k rovnakému objektu v dvoch sedeniach?

```
Session session1 = factory.openSession();
try {
  Book book1 = (Book) session1.get(Book.class, id);
} finally {
  session1.close();
}

Session session2 = factory.openSession();
try {
  Book book2 = (Book) session2.get(Book.class, id);
} finally {
  session2.close();
}
```

V tomto prípade odošle Hibernate dva SQL dopyty. Hibernate totiž v tomto prípade nedokáže načítavať cacheované objekty načítavané v rôznych sedeniach. Ak chceme dosiahnuť podporu takéhoto cacheovania, musíme použiť cache druhej úrovne, ktorá je platná v rámci jednej inštancie továrne na sedenia.

13.2. Cache druhej úrovne

Ak chceme zapnúť cache druhej úrovne, máme na výber viacero možností týkajúcich sa poskytovateľa. Hibernate podporuje viacero implementácii cache, napr. EHCache, OSCache, SwarmCache či JBossCache. V nedistribuovanom prostredí bude stačiť **EHCache**, ktorá je zároveň štandardne používanou implementáciou pre Hibernate. Nastavenie vykonáme v konfiguračnom súbore `hibernate.cfg.xml`.

```
<hibernate-configuration>
  <session-factory>
    ...
    <property name="cache.provider_class">
      org.hibernate.cache.EhCacheProvider
    </property>
    ...
  </session-factory>
</hibernate-configuration>
```

Implementáciu EHCache môžeme nakonfigurovať v súbore ehcache.xml, ktorý uložíme do koreňového adresára hierarchie balíčkov (teda do adresára so zdrojákmi). Môžeme v ňom špecifikovať viacero regiónov platnosti cache (cache regions), kde každý región obsahuje iný typ objektov. Parameter `eternal="false"` indikuje, že platnosť objektov vyprší po uplynutí príslušnej doby. Túto dobu je možné nastaviť v atribútoch `timeToIdleSeconds` and `timeToLiveSeconds`.

```
<ehcache>
  <diskStore path="java.io.tmpdir" />
  <defaultCache maxElementsInMemory="10000"
                eternal="false"
                timeToIdleSeconds="120" timeToLiveSeconds="120"
                overflowToDisk="true" />

  <cache name="mo.org.cpttm.bookshop.Book" maxElementsInMemory="10000"
         eternal="false"
         timeToIdleSeconds="300" timeToLiveSeconds="600"
         overflowToDisk="true" />
</ehcache>
```

Ak chceme sledovať aktivity týkajúce sa práce s cache počas behu, môžeme pridať do konfiguračného súboru log4j `log4j.properties` nasledovný riadok

```
log4j.logger.org.hibernate.cache=debug
```

V ďalšom kroku potrebujeme povoliť cache pre jednotlivé perzistentné triedy. Cacheované objekty budú uložené v regióne, ktorý má rovnaký názov ako perzistentná trieda, teda `mo.org.cpttm.bookshop.Book`.

Pri cacheovaní perzistentnej triedy môžeme zvoliť viacero variantov cacheovania. Ak sú perzistentné triedy určené len na čítanie a nebudú nikdy modifikované, najefektívnejším typom používania je read-only cache.

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Book" table="BOOK">
    <cache usage="read-only" />
    ...
  </class>
</hibernate-mapping>
```

Ak povolíme cacheovania kníh, po načítaní inštancie knihy s rovnakým identifikátorom v dvoch rôznych sedeniach sa odošle len jeden dopyt.

```
Session session1 = factory.openSession();
try {
  Book book1 = (Book) session1.get(Book.class, id);
} finally {
  session1.close();
}

Session session2 = factory.openSession();
try {
  Book book2 = (Book) session2.get(Book.class, id);
} finally {
  session2.close();
}
```

Ak však upravíme v niektorom zo sedení objekt knihy a pokúsime sa odoslať zmeny pomocou `flush()`, vyhodí sa výnimka upozorňujúca na aktualizáciu objektu, ktorý je určený len na čítanie:

```
Session session1 = factory.openSession();
try {
  Book book1 = (Book) session1.get(Book.class, id);
  book1.setName("New Book");
  session1.save(book1);
```

```
    session1.flush();
  } finally {
    session1.close();
  }
```

Ak chceme používať aktualizovateľné objekty, musíme zvoliť iný typ cacheovania a to read-write. Objekt, ktorý má byť aktualizovaný sa pred uložením do databázy najprv odstráni z cache.

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Book" table="BOOK">
    <cache usage="read-write" />
    ...
  </class>
</hibernate-mapping>
```

V niektorých prípadoch (napr. v prípade aktualizácie databázy inými aplikáciami) môže nastať potreba odstraňovať objekty z cache manuálne. Na to je možné použiť metódy poskytované továrňou na sedenia. Ukončiť platnosť objektu v cache je možné po jednom (`evict()`) alebo hromadne (všetky inštancie) pomocou `evictEntity()`.

```
factory.evict(Book.class);
factory.evict(Book.class, id);

factory.evictEntity("mo.org.cpttm.bookshop.Book");
factory.evictEntity("mo.org.cpttm.bookshop.Book", id);
```

13.3. Cacheovanie asociácii

Možno sa pýtate, či sa v prípade cacheovania objektu knihy vložia do cache aj asociované objekty (napr. vydavatelia).

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Book" table="BOOK">
    <cache usage="read-write" />
    ...
    <many-to-one name="publisher"
                 class="Publisher" column="PUBLISHER_ID" />
</class>
</hibernate-mapping>
```

Ak inicializujete asociáciu v dvoch rôznych sedeniach, ukáže sa, že asociované objekty načítajú dvakrát. Dôvodom je to, že Hibernate neukladá v regióne knihy celého vydavateľa, ale len jeho identifikátor.

```
Session session1 = factory.openSession();
try {
  Book book1 = (Book) session1.get(Book.class, id);
  Hibernate.initialize(book1.getPublisher());
} finally {
  session1.close();
}

Session session2 = factory.openSession();
try {
  Book book2 = (Book) session2.get(Book.class, id);
  Hibernate.initialize(book2.getPublisher());
} finally {
session2.close();
}
```

Ak chceme cacheovať objekty vydavateľov (v ich vlastnom regióne), musíme cacheovanie povoliť v definícii mapovania triedy `Publisher`. To realizujeme podobne ako v prípade knihy. Cacheované objekty vydavateľov budú ukladané v regióne `mo.org.cpttm.bookshop.Publisher`.

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Publisher" table="PUBLISHER">
    <cache usage="read-write" />
    ...
  </class>
</hibernate-mapping>
```

13.4. Cacheovanie kolekcií

Ak chceme cacheovať aj kapitoly knihy, musíme ho povoliť v rámci triedy Chapter. Cacheované kapitoly budú uložené v regióne `mo.org.cpttm.bookshop.Chapter`.

Mapovanie pre knihu:

```
<hibernate-mapping package="mo.org.cpttm.bookshop">
  <class name="Book" table="BOOK">
    <cache usage="read-write" />
    ...
    <set name="chapters" table="BOOK_CHAPTER">
      <key column="BOOK_ID" />
      <one-to-many class="Chapter" />
    </set>
  </class>
</hibernate-mapping>
```

Mapovanie pre kapitolu:

```
<hibernate-mapping package=”mo.org.cpttm.bookshop”>
  <class name=”Chapter” table=”CHAPTER”>
    <cache usage=”read-write” />
    ...
    <many-to-one name=”book” class=”Book” column=”BOOK_ID” />
  </class>
</hibernate-mapping>
```

Skúsme teraz inicializovať kolekciu v dvoch nezávislých sedeniach. Zrejme očakávame, že v druhom sedení sa nevyvolá žiaden SQL dopyt:

```
Session session1 = factory.openSession();
try {
  Book book1 = (Book) session1.get(Book.class, id);
  Hibernate.initialize(book1.getChapters());
} finally {
  session1.close();
}

Session session2 = factory.openSession();
try {
  Book book2 = (Book) session2.get(Book.class, id);
  Hibernate.initialize(book2.getChapters());
} finally {
  session2.close();
}
```

Ak si prezrieme vygenerované SQL dopyty, zistíme, že sa stále vykonáva jeden dopyt. Na rozdiel od asociácií M:1 sa kolekcie implicitne do cache neukladajú. Cacheovanie kolekcie musíme explicitne povoliť v rámci daného elementu kolekcie. Kolekcie kníh sa budú cacheovať v regióne `mo.org.cpttm.bookshop.Book.chapters`.

Metóda cacheovania kolekcie je nasledovná:

- Ak sa v kolekcii nachádzajú jednoduché hodnoty, do cache sa uložia priamo.
- Ak sa v kolekcii nachádzajú perzistentné objekty, v regióne kolekcie sa uložia len identifikátory objektov a samotné perzistentné objekty budú cacheované v ich regiónoch.

```
<hibernate-mapping package=”mo.org.cpttm.bookshop”>
  <class name=”Book” table=”BOOK”>
    <cache usage=”read-write” />
    ...
```

```
    <set name="chapters" table="BOOK_CHAPTER">

      <cache usage="read-write" />

      <key column="BOOK_ID" />
      <one-to-many class="Chapter" />
    </set>
  </class>
</hibernate-mapping>
```

Ak chceme kolekciu z cache odstrániť (alebo chceme odstrániť všetky kolekcie v rámci regiónu), môžeme na to použiť príslušné metódy volané na továrni na sedenia.

```
// všetky kolekcie
factory.evictCollection("mo.org.cpttm.bookshop.Book.chapters");
// len jedna kolekcia
factory.evictCollection("mo.org.cpttm.bookshop.Book.chapters", id);
```

Ak máme obojsmernú asociáciu 1:N/M:1, metóda `evictCollection()` by sa mala zavolať na kolekcii po aktualizácii jedného z koncov asociácie.

```
Session session1 = factory.openSession();
try {
  Book book1 = (Book) session1.get(Book.class, id);

  Chapter chapter = (Chapter) book1.getChapters().iterator().next();
  chapter.setBook(null);

  session1.saveOrUpdate(chapter);
  session1.flush();

  factory.evictCollection("mo.org.cpttm.bookshop.Book.chapters", id);
} finally {
  session1.close();
}

Session session2 = factory.openSession();
try {
  Book book2 = (Book) session2.get(Book.class, id);
  Hibernate.initialize(book2.getChapters());
} finally {
  session2.close();
}
```

13.5. Cacheovanie dopytov

Popri cacheovaní objektov načítavaných v rámci sedenia je možné cacheovať aj HQL dopyty. V nasledovnom príklade sa nachádza dopyt, ktorý sa vykonáva v rámci dvoch sedení:

```
Session session1 = factory.openSession();
try {
  Query query = session1.createQuery("from Book where name like ?");
  query.setString(0, "%Hibernate%");

  List books = query.list();
} finally {
  session1.close();
}

Session session2 = factory.openSession();
try {
  Query query = session2.createQuery("from Book where name like ?");
  query.setString(0, "%Hibernate%");

  List books = query.list();
} finally {
  session2.close();
}
```

Štandardne je cacheovanie HQL dopytov vypnuté. Povoliť ho môžeme v konfiguračnom súbore Hibernate.

```
<hibernate-configuration>
  <session-factory>
    ...
    <property name="cache.use_query_cache">true</property>
    ...
  </session-factory>
</hibernate-configuration>
```

Konkrétny dopyt môžeme cacheovať nastavením `setCacheable(true)` pred jeho vykonaním. Výsledok dopytu sa uloží do cache v regióne `org.hibernate.cache.StandardQueryCache`.

Ako prebieha cacheovanie výsledkov? Ak dopyt vracia jednoduché hodnoty, do cache sa uložia tieto hodnoty priamo. Ak dopyt vracia perzistentné objekty, do regiónu dopytu sa uložia identifikátory objektov a perzistentné objekty sa uložia do regiónu, ktorý im prislúcha.

```
Session session1 = factory.openSession();
try {
  Query query = session1.createQuery("from Book where name like ?");
  query.setString(0, "%Hibernate%");
  query.setCacheable(true);
  List books = query.list();
} finally {
  session1.close();
}

Session session2 = factory.openSession();
try {
  Query query = session2.createQuery("from Book where name like ?");
  query.setString(0, "%Hibernate%");
  query.setCacheable(true);
  List books = query.list();
} finally {
  session2.close();
}
```

V prípade potreby je možné danému dopytu zmeniť región cacheovania. Týmto spôsobom je možné rozložiť medzipamäte cache do viacerých regiónov a zredukovať počet medzipamätí v jednom regióne.

```
...
query.setCacheable(true);
query.setCacheRegion("mo.org.cpttm.bookshop.BookQuery");
```