HIBERNATE TUTORIAL

THE ULTIMATE GUIDE

HIBERNATE

MARTIN MOIS

Java Code Geeks
JAVA 2 JAVA DEVELOPERS RESOURCE CENTER

Hibernate Tutorial

Contents

Preface

ibernate ORM (Hibernate in short) is an object-relational mapping framework, facilitating the conversion of an object-oriented domain model to a traditional relational database. Hibernate solves the object-relational impedance mismatch problems by replacing direct persistence-related database accesses with high-level object handling functions.

Hibernate is one of the most popular Java frameworks out there. For this reason we have provided an abundance of tutorials here at Java Code Geeks, most of which can be found here: http://www.javacodegeeks.com/tutorials/java-tutorials/enterprise-java-tutorials/#Hibernate

Now, we wanted to create a standalone, reference post to provide a framework on how to work with Hibernate and help you quickly kick-start your Hibernate applications. Enjoy!

About the Author

Martin is a software engineer with more than 10 years of experience in software development. He has been involved in different positions in application development in a variety of software projects ranging from reusable software components, mobile applications over fat-client GUI projects up to larg-scale, clustered enterprise applications with real-time requirements.

After finishing his studies of computer science with a diploma, Martin worked as a Java developer and consultant for international operating insurance companies. Later on he designed and implemented web applications and fat-client applications for companies on the energy market. Currently Martin works for an international operating company in the Java EE domain and is concerned in his day-to-day work with larg-scale big data systems.

His current interests include Java EE, web applications with focus on HTML5 and performance optimizations. When time permits, he works on open source projects, some of them can be found at this github account. Martin is blogging at Martin's Developer World.

Chapter 1

Introduction

Hibernate is one of the most popular Object/Relational Mapping (ORM) framework in the Java world. It allows developers to map the object structures of normal Java classes to the relational structure of a database. With the help of an ORM framework the work to store data from object instances in memory to a persistent data store and load them back into the same object structure becomes significantly easier.

At the same time ORM solutions like Hibernate aim to abstract from the specific product used to store the data. This allows using the same Java code with different database products without the need to write code that handles the subtle differences between the supported products.

Hibernate is also a JPA provider, that means it implements the Java Persistence API (JPA). JPA is a vendor independent specification for mapping Java objects to the tables of relational databases. As another article of the Ultimate series already addresses the JPA, this article focuses on Hibernate and therefore does not use the JPA annotations but rather the Hibernate specific configuration files.

Hibernate consists of three different components:

- **Entities**: The classes that are mapped by Hibernate to the tables of a relational database system are simple Java classes (Plain Old Java Objects).
- **Object-relational metadata**: The information how to map the entities to the relational database is either provided by annotations (since Java 1.5) or by legacy XML-based configuration files. The information in these files is used at runtime to perform the mapping to the data store and back to the Java objects.
- **Hibernate Query Language (HQL)**: When using Hibernate, queries send to the database do not have to be formulated in native SQL but can be specified using Hibernate's query language. As these queries are translated at runtime into the currently used dialect of the chose product, queries formulated in HQL are independent from the SQL dialect of a specific vendor.

In this tutorial we are going through different aspects of the framework and will develop a simple Java SE application that stores and retrieves data in/from a relational database. We will use the following libraries/environments:

- maven >= 3.0 as build environment
- Hibernate(4.3.8.Final)
- H2 as relational database (1.3.176)

Chapter 2

Project setup

As a first step we will create a simple maven project on the command line:

```
mvn archetype:create -DgroupId=com.javacodegeeks.ultimate -DartifactId=hibernate
```

This command will create the following structure in the file system:

```
|-- src
|   |-- main
|   |   `-- java
|   |       `-- com
|   |           `-- javacodegeeks
|   |                       `-- ultimate
|   `-- test
|   |   `-- java
|   |       `-- com
|   |           `-- javacodegeeks
|   |                       `-- ultimate
`-- pom.xml
```

The libraries our implementation depends on are added to the dependencies section of the `pom.xml` file in the following way:

```
<properties>
        <h2.version>1.3.176</h2.version>
        <hibernate.version>4.3.8.Final</hibernate.version>
</properties>

<dependencies>
        <dependency>
                <groupId>com.h2database</groupId>
                <artifactId>h2</artifactId>
                <version>${h2.version}</version>
        </dependency>
        <dependency>
                <groupId>org.hibernate</groupId>
                <artifactId>hibernate-core</artifactId>
                <version>${hibernate.version}</version>
        </dependency>
</dependencies>
```

To get a better overview of the separate versions, we define each version as a maven property and reference it later on in the dependencies section.

Chapter 3

Basics

3.1 SessionFactory and Session

Now we cat start to implement our first O/R mapping. Let's start with a simple class that provides a `run()` method that is invoked in the application's `main` method:

```
public class Main {
        private static final Logger LOGGER = Logger.getLogger("Hibernate-Tutorial");

        public static void main(String[] args) {
                Main main = new Main();
                main.run();
        }

        public void run() {
                SessionFactory sessionFactory = null;
                Session session = null;
                try {
                        Configuration configuration = new Configuration();
                        configuration.configure("hibernate.cfg.xml");
                        ServiceRegistry serviceRegistry = new  ↩
                            StandardServiceRegistryBuilder().applySettings(configuration. ↩
                            getProperties()).build();
                        sessionFactory = configuration.buildSessionFactory(serviceRegistry) ↩
                            ;
                        session = sessionFactory.openSession();
                        persistPerson(session);
                } catch (Exception e) {
                        LOGGER.log(Level.SEVERE, e.getMessage(), e);
                } finally {
                        if (session != null) {
                                session.close();
                        }
                        if (sessionFactory != null) {
                                sessionFactory.close();
                        }
                }
        }
        ...
```

The `run()` method creates a new instance of the class `org.hibernate.cfg.Configuration` that is subsequently configured using the XML file `hibernate.cfg.xml`. Placing the configuration file in the folder `src/main/resources` of our project lets maven put it to the root of the created jar file. This way the file is found at runtime on the classpath.

As a second step the run() method constructs a ServiceRegistry that uses the previously loaded configuration. An instance of this ServiceRegistry can now be passed as an argument to the method buildSessionFactroy() of the Configuration. This SessionFactory can now be used to obtain the session needed to store and load entities to the underlying data store.

The configuration file hibernate.cfg.xml has the following content:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
        <property name="connection.driver_class">org.h2.Driver</property>
        <property name="connection.url">jdbc:h2:~/hibernate;AUTOCOMMIT=OFF</property>
        <property name="connection.username"></property>
        <property name="connection.password"></property>
        <property name="connection.pool_size">1</property>
        <property name="dialect">org.hibernate.dialect.H2Dialect</property>
        <property name="current_session_context_class">thread</property>
        <property name="cache.provider_class">org.hibernate.cache.internal.NoCacheProvider< ↩
            /property>
        <property name="show_sql">true</property>
        <property name="format_sql">true</property>
        <property name="hbm2ddl.auto">create</property>
        <mapping resource="Project.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

As we see from the example above, the configuration file defines a set of properties for the session factory. The first property connection.driver_class specifies the database driver that should be used. In our example this is the driver for the H2 database. Through the property connection.url, the JDBC-URL is specified. In our case defines that we want to use h2 and that the single database file where H2 stores its data should be located in the home directory of the user and should be named hibernate (~/hibernate). As we want to commit our transactions in the example code on our own, we also define the H2 specific configuration option AUTOCOMMIT=OFF.

Next the configuration file defines the username and password for the database connection as well as the size of the connection pool. Our sample application just executes code in one single thread, therefore we can set the pool size to one. In cases of an application that has to deal with multiple threads and users, an appropriate pool size has to be chosen.

The property dialect specifies a Java class that performs the translation into the database specific SQL dialect.

As of version 3.1, Hibernate provides a method named SessionFactory.getCurrentSession() that allows the developer to obtain a reference to the current session. With the configuration property current_session_context_class it can be configured where Hibernate should obtain this session from. The default value for this property is jta meaning that Hibernate obtains the session from the underlying Java Transaction API (JTA). As we are not using JTA in this sample, we instruct Hibernate with the configuration value thread to store and retrieve the session to/from the current thread.

For the sake of simplicity, we do not want to utilize an entity cache. Hence we set the property cache.provider_class to org.hibernate.cache.internal.NoCacheProvider.

The following two options tell Hibernate to print out each SQL statement to the console and to format it for better readability. In order to relieve us for development purposes from the burden to create the schema manually, we instruct Hibernate with the option hbm2ddl.auto set to create to create all tables during startup.

Last but not least we define a mapping resource file that contains all the mapping information for our application. The content of this file will be explained in the following sections.

As mentioned above, the session is used to communicate with the data store and actually represents a JDBC connection. This means all interaction with the connection is done through the session. It is single-threaded and provides a cache for all objects it has up to now worked with. Therefore each thread in the application should work with its own session that it obtains from the session factory.

In contrast to the session, the session factory is thread-safe and provides an immutable cache for the define mappings. For each database there is exactly one session factory. Optionally, the session factory can provide in addition to the session's first level cache an application wide second level cache.

3.2 Transactions

In the `hibernate.cfg.xml` configuration file we have configured to manage transactions on our own. Hence we have to manually start and commit or rollback every transaction. The following code demonstrates how to obtain a new transaction from the session and how to start and commit it:

```
try {
        Transaction transaction = session.getTransaction();
        transaction.begin();
        ...
        transaction.commit();
} catch (Exception e) {
        if (session.getTransaction().isActive()) {
                session.getTransaction().rollback();
        }
        throw e;
}
```

In the first step we call `getTransaction()` in order to retrieve a reference for a new transaction. This transaction is immediately started by invoking the method `begin()` on it. If the following code proceeds without any exception, the transaction gets committed. In case an exception occurred and the current transaction is active, the transaction is rolled back.

As the code shown above is the same for all upcoming examples, it is not repeated in the exact form again and again. The steps to refactor the code into a re-usable form, using for example the template pattern, are left for the reader.

3.3 Tables

Now that we have learned about session factories, sessions and transactions, it is time to start with the first class mapping. In order to have an easy start, we choose a simple class with only a few simple attributes:

```
public class Person {
        private Long id;
        private String firstName;
        private String lastName;

        public Long getId() {
                return id;
        }

        public void setId(Long id) {
                this.id = id;
        }

        public String getFirstName() {
                return firstName;
        }

        public void setFirstName(String firstName) {
                this.firstName = firstName;
        }

        public String getLastName() {
                return lastName;
        }
        public void setLastName(String lastName) {
                this.lastName = lastName;
        }
}
```

The `Person` class comes with two attributes to store the name of a person (`firstName` and `lastName`). The field `id` is used to store the object's unique identifier as a long value. In this tutorial we are going to use mapping files instead of annotations, hence we specify the mapping of this class to the table `T_PERSON` as follows:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="hibernate.entity">
    <class name="Person" table="T_PERSON">
        <id name="id" column="ID">
            <generator class="native"/>
        </id>
        <property name="firstName" column="FIRST_NAME"/>
        <property name="lastName" column="LAST_NAME"/>
    </class>
</hibernate-mapping>
```

The XML element `hibernate-mapping` is used to define the package our entities reside in (here: `hibernate.entity`). Inside this element one `class` element is provided for each class that should be mapped to a table in the database.

The `id` element specifies the name (`name`) of the class's field that holds the unique identifier and the name of the column this value is stored in (`ID`). With its child element `generator` Hibernate gets to know how to create the unique identifier for each entity. Next to the value `native` shown above, Hibernate supports a long list of different strategies.

The `native` strategy just chooses the best strategy for the used database product. Hence this strategy can be applied for different products. Other possible values are for example: `sequence` (uses a sequence in the database), `uuid` (generates a 128-bit UUID) and `assigned` (lets the application assign the value on its own). Beyond the pre-defined strategies it is possible to implement a custom strategy by implementing the interface `org.hibernate.id.IdentifierGenerator`.

The fields `firstName` and `lastName` are mapped to the columns `FIRST_NAME` and `LAST_NAME` by using the XML element `property`. The attributes `name` and `column` define the field's name in the class and the column, respectively.

The following code shows exemplary how to store a person in the database:

```
private void persistPerson(Session session) throws Exception {
        try {
                Transaction transaction = session.getTransaction();
                transaction.begin();
                Person person = new Person();
                person.setFirstName("Homer");
                person.setLastName("Simpson");
                session.save(person);
                transaction.commit();
        } catch (Exception e) {
                if (session.getTransaction().isActive()) {
                        session.getTransaction().rollback();
                }
                throw e;
        }
}
```

Next to the code to handle the transaction it creates a new instance of the class `Person` and assigns two values to the fields `firstName` and `lastName`. Finally it stores the person in the database by invoking the session's method `save()`.

When we execute the code above, the following SQL statements are printed on the console:

```
Hibernate:
    drop table T_PERSON if exists
Hibernate:
    create table T_PERSON (
        ID bigint generated by default as identity,
        FIRST_NAME varchar(255),
        LAST_NAME varchar(255),
        primary key (ID)
    )
Hibernate:
    insert
    into
        T_PERSON
        (ID, firstName, lastName, ID_ID_CARD)
    values
        (null, ?, ?, ?)
```

As we have chosen to let Hibernate drop and create the tables on startup, the first statements printed out are the `drop table` and `create table` statements. We can also see the three columns `ID`, `FIRST_NAME` and `LAST_NAME` of the table `T_PERSON` as well as the definition of the primary key (here: `ID`).

After the table has been created, the invocation of `session.save()` issues an `insert` statement to the database. As Hibernate internally uses a `PreparedStatement`, we do not see the values on the console. In case you also want to see the values that are bound to the parameters of the `PreparedStatement`, you can set the logging level for the logger `org.hibernate.type` to `FINEST`. This is done within a file called `logging.properties` with the following content (the path to the file can be given for example as a system property `-Djava.util.logging.config.file=src/main/resources/logging.properties`):

```
.handlers = java.util.logging.ConsoleHandler
.level = INFO

java.util.logging.ConsoleHandler.level = ALL
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter

org.hibernate.SQL.level = FINEST
org.hibernate.type.level = FINEST
```

Setting the logger `org.hibernate.SQL` has the same effect as setting the property `show_sql` in the Hibernate configuration file to `true`.

Now you can see the following output and therewith the actual values on the console:

```
DEBUG:
    insert
    into
        T_PERSON
        (ID, FIRST_NAME, LAST_NAME, ID_ID_CARD)
    values
        (null, ?, ?, ?)
TRACE: binding parameter [1] as [VARCHAR] - [Homer]
TRACE: binding parameter [2] as [VARCHAR] - [Simpson]
TRACE: binding parameter [3] as [BIGINT] - [null]
```

Chapter 4

Inheritance

An interesting feature of O/R mapping solutions like Hibernate is the usage of inheritance. The user can chose how to map superclass and subclass to the tables of a relational database. Hibernate supports the following mapping strategies:

- **Single table per class**: Both superclass and subclass are mapped to the same table. An additional column marks whether the row is an instance of the superclass or subclass and fields that are not present in the superclass are left empty.
- **Joined subclass**: This strategy uses a separate table for each class whereas the table for the subclass only stores the fields that are not present in the superclass. To retrieve all values for an instance of the subclass, a join between the two tables has to be performed.
- **Table per class**: This strategy also uses a separate table for each class but stores in the table for the subclass also the fields of the superclass. With this strategy one row in the subclass table contains all values and in order to retrieve all values no join statement is necessary.

The approach we are going to investigate is the "Single Table per class" approach. As a subclass of person we choose the class `Geek`:

```
public class Geek extends Person {
        private String favouriteProgrammingLanguage;

        public String getFavouriteProgrammingLanguage() {
                        return favouriteProgrammingLanguage;
        }

        public void setFavouriteProgrammingLanguage(String favouriteProgrammingLanguage) {
                this.favouriteProgrammingLanguage = favouriteProgrammingLanguage;
        }
}
```

The class extends the already known class `Person` and adds an additional field named `favouriteProgrammingLanguage`. The mapping file for this use case looks like the following one:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="hibernate.entity">
    <class name="Person" table="T_PERSON">
        <id name="id" column="ID">
            <generator class="native"/>
        </id>
        <discriminator column="PERSON_TYPE" type="string"/>
        <property name="firstName" column="FIRST_NAME"/>
        <property name="lastName" column="LAST_NAME"/>
        <subclass name="Geek" extends="Person">
            <property name="favouriteProgrammingLanguage" column="FAV_PROG_LANG"/>
        </subclass>
    </class>
</hibernate-mapping>
```

The first difference is the introduction of the `discriminator` column. As mentioned above this column stores the information of which type the current instance is. In our case we call it `PERSON_TYPE` and let for better readability a string denote the actual type. Per default Hibernate takes just the class name in this case. To save storage one can also use a column of type integer.

Beyond the discriminator we have also added the `subclass` element that informs Hibernate about the new Java class `Geek` and its field `favouriteProgrammingLanguage` which should be mapped to the column `FAV_PROG_LANG`.

The following sample codes shows how to store instances of type `Geek` in the database:

```
session.getTransaction().begin();
Geek geek = new Geek();
geek.setFirstName("Gavin");
geek.setLastName("Coffee");
geek.setFavouriteProgrammingLanguage("Java");
session.save(geek);
geek = new Geek();
geek.setFirstName("Thomas");
geek.setLastName("Micro");
geek.setFavouriteProgrammingLanguage("C#");
session.save(geek);
geek = new Geek();
geek.setFirstName("Christian");
geek.setLastName("Cup");
geek.setFavouriteProgrammingLanguage("Java");
session.save(geek);
session.getTransaction().commit();
```

Executing the code shown above, leads to the following output:

```
Hibernate:
    drop table T_PERSON if exists
Hibernate:
    create table T_PERSON (
        ID bigint generated by default as identity,
        PERSON_TYPE varchar(255) not null,
        FIRST_NAME varchar(255),
        LAST_NAME varchar(255),
        FAV_PROG_LANG varchar(255),
        primary key (ID)
    )
Hibernate:
    insert
    into
        T_PERSON
        (ID, FIRST_NAME, LAST_NAME, FAV_PROG_LANG, PERSON_TYPE)
    values
        (null, ?, ?, ?, 'hibernate.entity.Geek')
```

In contrast to the previous example the table `T_PERSON` now contains the two new columns `PERSON_TYPE` and `FAV_PROG_LANG`. The column `PERSON_TYPE` contains the value `hibernate.entity.Geek` for geeks.

In order to investigate the content of the `T_PERSON` table, we can utilize the Shell application shipped within the H2 jar file:

```
> java -cp h2-1.3.176.jar org.h2.tools.Shell -url jdbc:h2:~/hibernate
...
```

```
sql> select * from t_person;
ID | PERSON_TYPE              | FIRST_NAME | LAST_NAME | FAV_PROG_LANG
1  | hibernate.entity.Person | Homer      | Simpson   | null
2  | hibernate.entity.Geek   | Gavin      | Coffee    | Java
3  | hibernate.entity.Geek   | Thomas     | Micro     | C#
4  | hibernate.entity.Geek   | Christian  | Cup       | Java
```

As discussed above, the column `PERSON_TYPE` stores the type of the instance whereas the column `FAV_PROG_LANG` contains the value `null` for instances of the superclass `Person`.

Changing the mapping definition in a way that it looks like the following one, Hibernate will create for the superclass and the subclass a separate table:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="hibernate.entity">
    <class name="Person" table="T_PERSON">
        <id name="id" column="ID">
            <generator class="native"/>
        </id>
        <property name="firstName" column="FIRST_NAME"/>
        <property name="lastName" column="LAST_NAME"/>
        <joined-subclass name="Geek" table="T_GEEK">
            <key column="ID_PERSON"/>
            <property name="favouriteProgrammingLanguage" column="FAV_PROG_LANG"/>
        </joined-subclass>
    </class>
</hibernate-mapping>
```

The XML element `joined-subclass` tells Hibernate to create the table `T_GEEK` for the subclass `Geek` with the additional column `ID_PERSON`. This additional key column stores a foreign key to the table `T_PERSON` in order to assign each row in `T_GEEK` to its parent row in `T_PERSON`.

Using the Java code shown above to store a few geeks in the database, results in the following output on the console:

```
Hibernate:
    drop table T_GEEK if exists
Hibernate:
    drop table T_PERSON if exists
Hibernate:
    create table T_GEEK (
        ID_PERSON bigint not null,
        FAV_PROG_LANG varchar(255),
        primary key (ID_PERSON)
    )
Hibernate:
    create table T_PERSON (
        ID bigint generated by default as identity,
        FIRST_NAME varchar(255),
        LAST_NAME varchar(255),
        primary key (ID)
    )
Hibernate:
    alter table T_GEEK
        add constraint FK_p2ile8qooftvytnxnqtjkrbsa
        foreign key (ID_PERSON)
        references T_PERSON
```

Now Hibernate creates two tables instead of one and defines a foreign key for the table T_GEEK that references the table T_PERSON. The table T_GEEK consists of two columns: ID_PERSON to reference the corresponding person and FAV_PROG_LANG to store the favorite programming language.

Storing a geek in the database now consists of two insert statements:

```
Hibernate:
    insert
    into
        T_PERSON
        (ID, FIRST_NAME, LAST_NAME, ID_ID_CARD)
    values
        (null, ?, ?, ?)
Hibernate:
    insert
    into
        T_GEEK
        (FAV_PROG_LANG, ID_PERSON)
    values
        (?, ?)
```

The first statement inserts a new row into the table T_PERSON, while the second one inserts a new row into the table T_GEEK. The content of these two tables look afterwards like this:

```
sql> select * from t_person;
ID | FIRST_NAME | LAST_NAME
1  | Homer      | Simpson
2  | Gavin      | Coffee
3  | Thomas     | Micro
4  | Christian  | Cup

sql> select * from t_geek;
ID_PERSON | FAV_PROG_LANG
2         | Java
3         | C#
4         | Java
```

Obviously the table T_PERSON only stores the attributes of the superclass whereas the table T_GEEK only stores the field values for the subclass. The column ID_PERSON references the corresponding row from the parent table.

The next strategy under investigation is "table per class". Similar to the last strategy this one also creates a separate table for each class, but in contrast the table for the subclass contains also all columns of the superclass. Therewith one row in such a table contains all values to construct an instance of this type without the need to join additional data from the parent table. On huge data set this can improve the performance of queries as joins need to find additionally the corresponding rows in the parent table. This additional lookup costs time that is circumvented with this approach.

To use this strategy for the above use case, the mapping file can be rewritten like the following one:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="hibernate.entity">
    <class name="Person" table="T_PERSON">
        <id name="id" column="ID">
            <generator class="sequence"/>
        </id>
        <property name="firstName" column="FIRST_NAME"/>
        <property name="lastName" column="LAST_NAME"/>
        <union-subclass name="Geek" table="T_GEEK">
            <property name="favouriteProgrammingLanguage" column="FAV_PROG_LANG"/>
        </union-subclass>
    </class>
</hibernate-mapping>
```

The XML element `union-subclass` provides the name of the entity (`Geek`) as well as the name of the separate table (`T_GEEK`) as attributes. As within the other approaches, the field `favouriteProgrammingLanguage` is declared as a property of the subclass.

Another important change with regard to the other approaches is contained in the line that defines the id generator. As the other approaches use a `native` generator, which falls back on H2 to an identity column, this approach requires an id generator that creates identities that are unique for both tables (`T_PERSON` and `T_GEEK`).

The identity column is just a special type of column that automatically creates for each row a new id. But with two tables we have also two identity columns and therewith the ids in the `T_PERSON` table can be the same as in the `T_GEEK` table. This conflicts with the requirement that an entity of type `Geek` can be created just by reading one row of the table `T_GEEK` and that the identifiers for all persons and geeks are unique. Therefore we are using a sequence instead of an identity column by switching the value for the `class` attribute from `native` to `sequence`.

Now the DDL statements created by Hibernate look like the following ones:

```
Hibernate:
    drop table T_GEEK if exists
Hibernate:
    drop table T_PERSON if exists
Hibernate:
    drop sequence if exists hibernate_sequence
Hibernate:
    create table T_GEEK (
        ID bigint not null,
        FIRST_NAME varchar(255),
        LAST_NAME varchar(255),
        FAV_PROG_LANG varchar(255),
        primary key (ID)
    )
Hibernate:
    create table T_PERSON (
        ID bigint not null,
        FIRST_NAME varchar(255),
        LAST_NAME varchar(255),
        primary key (ID)
    )
Hibernate:
    create sequence hibernate_sequence
```

The output above clearly demonstrates that the table `T_GEEK` now contains next to `FAV_PROG_LANG` also the columns for the superclass (`FIRST_NAME` and `LAST_NAME`). The statements do not create a foreign key between the two tables. Please also note that now the column `ID` is no longer an identity column but that instead a sequence is created.

The insertion of a person and a geek issues the following statements to the database:

```
Hibernate:
    call next value for hibernate_sequence
Hibernate:
    insert
    into
        T_PERSON
        (FIRST_NAME, LAST_NAME, ID)
    values
        (?, ?, ?, ?)
Hibernate:
    call next value for hibernate_sequence
Hibernate:
    insert
    into
```

```
        T_GEEK
        (FIRST_NAME, LAST_NAME, FAV_PROG_LANG, ID)
    values
        (?, ?, ?, ?, ?)
```

For one person and one geek we have obviously only two insert statements. The table `T_GEEK` is completely filled by one insertion and contains all values of an instance of `Geek`:

```
sql> select * from t_person;
ID | FIRST_NAME | LAST_NAME
1  | Homer      | Simpson

sql> select * from t_geek;
ID | FIRST_NAME | LAST_NAME | FAV_PROG_LANG
3  | Gavin      | Coffee    | Java
4  | Thomas     | Micro     | C#
5  | Christian  | Cup       | Java
```

Chapter 5

Relationships

Up to now the only relationship between two tables we have seen was the "extends" one. Next to the mere inheritance Hibernate can also map relationships that are based on lists where the one entity has a list of instances of another entity. The following types of relationships are distinguished:

- **One to one**: This denotes a simple relationship in which one entity of type A belongs exactly to one entity of type B.
- **Many to one**: As the name indicates, this relationship encompasses the case that an entity of type A has many child entities of type B.
- **Many to many**: In this case there can be many entities of type A that belong to many entities of type B.

To understand these different types of relationships a little better, we will investigate them in the following.

5.1 OneToOne

As an example for the "one to one" case we add the following class to our entity model:

```
public class IdCard {
        private Long id;
        private String idNumber;
        private Date issueDate;
        private boolean valid;

        public Long getId() {
                return id;
        }

        public void setId(Long id) {
                this.id = id;
        }

        public String getIdNumber() {
                return idNumber;
        }

        public void setIdNumber(String idNumber) {
                this.idNumber = idNumber;
        }

        public Date getIssueDate() {
                return issueDate;
        }
        public void setIssueDate(Date issueDate) {
                this.issueDate = issueDate;
        }

        public boolean isValid() {
                return valid;
        }

        public void setValid(boolean valid) {
                this.valid = valid;
        }
}
```

An identity card as an internal unique identifier as well as an external idNumber, an issue date and a boolean flag that indicates if the card is valid or not.

On the other side of the relation the person gets a new field named `idCard` that references the card of this person:

```
public class Person {
        ...
        private IdCard idCard;

        ...

        public IdCard getIdCard() {
                return idCard;
        }

        public void setIdCard(IdCard idCard) {
                this.idCard = idCard;
        }
```

To map this relation using the Hibernate specific mapping file, we change it in the following way:

```
<hibernate-mapping package="hibernate.entity">
    <class name="IdCard" table="T_ID_CARD">
        <id name="id" column="ID">
            <generator class="sequence"/>
        </id>
    </class>
    <class name="Person" table="T_PERSON">
        <id name="id" column="ID">
            <generator class="sequence"/>
        </id>
        <property name="firstName" column="FIRST_NAME"/>
        <property name="lastName" column="LAST_NAME"/>
        <many-to-one name="idCard" column="ID_ID_CARD" unique="true"/>
        <union-subclass name="Geek" table="T_GEEK">
            <property name="favouriteProgrammingLanguage" column="FAV_PROG_LANG"/>
        </union-subclass>
    </class>
</hibernate-mapping>
```

First of all we add a new `class` element for the new class, specifying the name of the class and its corresponding table name (here: `T_ID_CARD`). The field `id` becomes the unique identifier and should be filled with the value of a sequence.

On the other hand the `Person` mapping now contains the new XML element `many-to-one` and references with its attribute `name` the field of the class `Person` that stores the reference to the `IdCard`. The optional attribute `column` lets us specify the exact name of the foreign key column in the table `T_PERSON` that links to the person's id card. As this relation should be of type "one to one" we have to set the attribute `unique` to `true`.

Executing this configuration results in the following DDL statements (please note that in order to reduce the number of tables we have switched back to the "single table per class" approach where we have only one table for superclass and subclass):

```
Hibernate:
    drop table T_ID_CARD if exists
Hibernate:
    drop table T_PERSON if exists
Hibernate:
    drop sequence if exists hibernate_sequence
Hibernate:
    create table T_ID_CARD (
        ID bigint not null,
        ID_NUMBER varchar(255),
        ISSUE_DATE timestamp,
        VALID boolean,
        primary key (ID)
    )
Hibernate:
    create table T_PERSON (
        ID bigint not null,
        PERSON_TYPE varchar(255) not null,
        FIRST_NAME varchar(255),
        LAST_NAME varchar(255),
        ID_ID_CARD bigint,
        FAV_PROG_LANG varchar(255),
        primary key (ID)
    )
Hibernate:
    alter table T_PERSON
        add constraint UK_96axqtck4kc0be4ancejxtu0p  unique (ID_ID_CARD)
Hibernate:
    alter table T_PERSON
        add constraint FK_96axqtck4kc0be4ancejxtu0p
        foreign key (ID_ID_CARD)
        references T_ID_CARD
Hibernate:
    create sequence hibernate_sequence
```

What has changed with regard to the previous examples is that the table `T_PERSON` now contains an additional column `ID_ID_CARD` that is defined as foreign key to the table `T_ID_CARD`. The table `T_ID_CARD` itself contains as expected the three columns `ID_NUMBER`, `ISSUE_DATE` and `VALID`.

The Java code to insert a person together with its id card looks like the following one:

```
Person person = new Person();
person.setFirstName("Homer");
person.setLastName("Simpson");
session.save(person);
IdCard idCard = new IdCard();
idCard.setIdNumber("4711");
idCard.setIssueDate(new Date());
person.setIdCard(idCard);
session.save(idCard);
```

Creating an instance of `IdCard` is straight-forward, please also note that the reference from `Person` to `IdCard` is set in the last but one line. Both instances are passed to Hibernate's `save()` method.

Looking at the code above in more detail, one might argue why we have to pass both instances to the `save()` method of the session. This point is justified, as Hibernate allows to define that certain operation should be "cascaded" when processing a complete entity graph. To enable cascading for the relationship to the `IdCard` we can simply add the attribute `cascade` to the `many-to-one` element in the mapping file:

```
<many-to-one name="idCard" column="ID_ID_CARD" unique="true" cascade="all"/>
```

Using the value `all` tells Hibernate to cascade all types of operations. As this is not always the preferred way to handle relationships between entities, one can also select only specific operations:

```
<many-to-one name="idCard" column="ID_ID_CARD" unique="true" cascade="save-update,refresh"/ ↩
    >
```

The example above demonstrates how to configure the mapping such that only calls to `save()`, `saveOrUpdate()` and `refresh` (re-reads the state of the given object from the database) are cascaded. Calls to the Hibernate methods `delete()` or `lock()` would for example not be forwarded.

Using on of the two configurations above, the code to store a person together with its id card can be rewritten to the following one:

```
Person person = new Person();
person.setFirstName("Homer");
person.setLastName("Simpson");
IdCard idCard = new IdCard();
idCard.setIdNumber("4711");
idCard.setIssueDate(new Date());
person.setIdCard(idCard);
session.save(person);
```

Instead of using the method `save()` one could also use in this use case the method `saveOrUpdate()`. The purpose of the method `saveOrUpdate()` is that it can be also used to update an existing entity. A subtle difference between both implementations is the fact that the `save()` methods returns the created identifier of the new entity:

```
Long personId = (Long) session.save(person);
```

This is helpful when writing for example server side code that should return this identifier to the caller of the method. On the other hand the method `update()` does not return the identifier as it assumes that the entity has already been stored to the data store and therefore must have an identifier. Trying to update an entity without an identifier will throw an exception:

```
org.hibernate.TransientObjectException: The given object has a null identifier: ...
```

Therefore `saveOrUpdate()` helps in cases where one wants to omit code that decides whether the entity has already been stored or not.

5.2 OneToMany

Another relation that appears frequently during O/R mappings is the “one to many” relation. In this case a set of entities belongs to one entity of another type. In order to model such a relation we add the class `Phone` to our model:

```
public class Phone {
        private Long id;
        private String number;
        private Person person;

        public Long getId() {
                return id;
        }

        public void setId(Long id) {
                this.id = id;
        }

        public String getNumber() {
                return number;
        }
        public void setNumber(String number) {
                this.number = number;
        }

        public Person getPerson() {
                return person;
        }

        public void setPerson(Person person) {
                this.person = person;
        }
}
```

As usual the entity `Phone` has an internal identifier (`id`) and a field to store the actual phone number. The field `person` stores a reference back to the person who owns this phone. As one person can have more than one phone, we add a `Set` to the `Person` class that collects all phones of one person:

```
public class Person {
        ...
        private Set<Phone> phones = new HashSet<Phone>();
        ...
        public Set<Phone> getPhones() {
                return phones;
        }

        public void setPhones(Set<Phone> phones) {
                this.phones = phones;
        }
}
```

The mapping file has to be updated accordingly:

```
<hibernate-mapping package="hibernate.entity">
        ...
    <class name="Phone" table="T_PHONE">
        <id name="id" column="ID">
            <generator class="sequence"/>
        </id>
        <property name="number" column="NUMBER"/>
        <many-to-one name="person" column="ID_PERSON" unique="false" cascade="all"/>
    </class>
    <class name="Person" table="T_PERSON">
        <id name="id" column="ID">
            <generator class="sequence"/>
        </id>
        <discriminator column="PERSON_TYPE" type="string"/>
        <property name="firstName" column="FIRST_NAME"/>
        <property name="lastName" column="LAST_NAME"/>
        <many-to-one name="idCard" column="ID_ID_CARD" unique="true" cascade="all"/>
                <subclass name="Geek" extends="Person">
            <property name="favouriteProgrammingLanguage" column="FAV_PROG_LANG"/>
        </subclass>
    </class>
</hibernate-mapping>
```

The listing above shows the definition of the mapping for the class `Phone`. Next to the usual identifier (`id`) that is generated using a sequence and the field `number` this definition also contains out the `many-to-one` element. In contrast to the "one to one" relation we have seen before, the attribute `unique` is set to `false`. Beyond that the attribute `column` defines the name of the foreign key column and the value of the attribute `cascade` how Hibernate should cascade operations on this relation.

Having executed the above configuration will print out the following DDL statements:

```
...
Hibernate:
    drop table T_PERSON if exists
Hibernate:
    drop table T_PHONE if exists
...
Hibernate:
    create table T_PERSON (
        ID bigint not null,
        PERSON_TYPE varchar(255) not null,
        FIRST_NAME varchar(255),
        LAST_NAME varchar(255),
        ID_ID_CARD bigint,
        FAV_PROG_LANG varchar(255),
        primary key (ID)
    )
Hibernate:
    create table T_PHONE (
        ID bigint not null,
        NUMBER varchar(255),
        ID_PERSON bigint,
        primary key (ID)
    )
...
Hibernate:
    alter table T_PHONE
        add constraint FK_dvxwd55q1bax99ibyw4oxa8iy
        foreign key (ID_PERSON)
        references T_PERSON
...
```

Next to the table `T_PERSON` Hibernate now also creates the new table `T_PHONE` with its three columns `ID`, `NUMBER` and `ID_PERSON`. As the latter column stores the reference to the `Person`, Hibernate also adds a foreign key constraint to the table `T_PHONE` that points to the column `ID` of the table `T_PERSON`.

In order to add a phone number to one of the existing persons, we first load a specific person and then add the phone:

```
session.getTransaction().begin();
List<Person> resultList = session.createQuery("from Person as person where person.firstName ←
     = ?").setString(0, "Homer").list();
for (Person person : resultList) {
        Phone phone = new Phone();
        phone.setNumber("+49 1234 456789");
        session.persist(phone);
        person.getPhones().add(phone);
        phone.setPerson(person);
}
session.getTransaction().commit();
```

This example shows how to load a person from the data store by using Hibernate's Query Language (HQL). Similarly to SQL this query consists of a from and a where clause. The column `FIRST_NAME` is not referenced by using its SQL name. Instead the name of the Java field/property is used. Parameters like the first name can be passed into the query by using the `setString()` method.

In the following the code iterates over the found persons (should be only one) and creates a new instance of `Phone` that is added to the set of phones of the found person. The link back from the phone to the person is also set before the transaction is committed. Having executed this code, the database looks like the following one:

```
sql> select * from t_person where first_name = 'Homer';
ID | PERSON_TYPE             | FIRST_NAME | LAST_NAME | ID_ID_CARD | FAV_PROG_LANG
1  | hibernate.entity.Person | Homer      | Simpson   | 2          | null
```

```
sql> select * from t_phone;
ID | NUMBER          | ID_PERSON
6  | +49 1234 456789 | 1
```

The result sets of the two select statements above shows that the row in `T_PHONE` is connected to the selected row in `T_PERSON` as it contains the id of the person with first name "Homer" in its column `ID_ID_PERSON`.

5.3 ManyToMany

The next interesting relationship to look at is the "many to many" relation. In this case many entities of type A can belong to many entities of type B and vice versa. In practice this is for example the case with geeks and projects. One geek can work in multiple projects (either simultaneously or sequentially) and one project can consist of more than one geek. Therefore the new entity `Project` is introduced:

```
public class Project {
        private Long id;
        private String title;
        private Set<Geek> geeks = new HashSet<Geek>();

        public Long getId() {
                return id;
        }

        public void setId(Long id) {
                this.id = id;
        }

        public String getTitle() {
                return title;
        }

        public void setTitle(String title) {
                this.title = title;
        }

        public Set<Geek> getGeeks() {
                return geeks;
        }

        public void setGeeks(Set<Geek> geeks) {
                this.geeks = geeks;
        }
}
```

It consists next to the identifier (`id`) of a title and a set of geeks. On the other side of the relation the class `Geek` has a set of projects:

```
public class Geek extends Person {
        private String favouriteProgrammingLanguage;
        private Set<Project> projects = new HashSet<Project>();

        public String getFavouriteProgrammingLanguage() {
                        return favouriteProgrammingLanguage;
        }

        public void setFavouriteProgrammingLanguage(String favouriteProgrammingLanguage) {
                this.favouriteProgrammingLanguage = favouriteProgrammingLanguage;
        }
        public Set<Project> getProjects() {
                return projects;
        }

        public void setProjects(Set<Project> projects) {
                this.projects = projects;
        }
}
```

To support this kind of relation the mapping file has to be changed in the following way:

```
<hibernate-mapping package="hibernate.entity">
        ...
    <class name="Project" table="T_PROJECT">
        <id name="id" column="ID">
            <generator class="sequence"/>
        </id>
        <property name="title" column="TITLE"/>
        <set name="geeks" table="T_GEEKS_PROJECTS">
            <key column="ID_PROJECT"/>
            <many-to-many column="ID_GEEK" class="Geek"/>
        </set>
    </class>
    <class name="Person" table="T_PERSON">
        <id name="id" column="ID">
            <generator class="sequence"/>
        </id>
        <discriminator column="PERSON_TYPE" type="string"/>
        <property name="firstName" column="FIRST_NAME"/>
        <property name="lastName" column="LAST_NAME"/>
        <many-to-one name="idCard" column="ID_ID_CARD" unique="true" cascade="all"/>
        <subclass name="Geek" extends="Person">
            <property name="favouriteProgrammingLanguage" column="FAV_PROG_LANG"/>
            <set name="projects" inverse="true">
                <key column="ID_GEEK"/>
                <many-to-many column="ID_PROJECT" class="Project"/>
            </set>
        </subclass>
    </class>
</hibernate-mapping>
```

First of all we see the new class `Project` that is mapped to the table `T_PROJECT`. Its unique identifier is stored in the field `id` and its field `title` is stored in the column `TITLE`. The XML element `set` defines the one side of the mapping: the items inside the set `geeks` should be stored in a separate table named `T_GEEKS_PROJECTS` with the columns `ID_PROJECT` and `ID_GEEK`. On the other side of the relation the XML element `set` inside the `subclass` for `Geek` defines the inverse relation (`inverse="true"`). On this side the field in the class `Geek` is called `projects` and the reference class is `Project`.

The resulting statements to create the tables look like this:

```
...
Hibernate:
    drop table T_GEEKS_PROJECTS if exists
Hibernate:
    drop table T_PROJECT if exists
...
Hibernate:
    create table T_GEEKS_PROJECTS (
        ID_PROJECT bigint not null,
        ID_GEEK bigint not null,
        primary key (ID_PROJECT, ID_GEEK)
    )
```

```
Hibernate:
    create table T_PROJECT (
        ID bigint not null,
        TITLE varchar(255),
        primary key (ID)
    )
...
Hibernate:
    alter table T_GEEKS_PROJECTS
        add constraint FK_2kp3f3tq46ckky02pshvjngaq
        foreign key (ID_GEEK)
        references T_PERSON
Hibernate:
    alter table T_GEEKS_PROJECTS
        add constraint FK_36tafu1nw9j5o51d21xm5rqne
        foreign key (ID_PROJECT)
        references T_PROJECT
...
```

These statements create the new tables `T_PROJECT` as well as `T_GEEKS_PROJECTS`. The table `T_PROJECT` consists of the columns `ID` and `TITLE` whereby the values in the column `ID` are referred to in the new table `T_GEEKS_PROJECTS` in its column `ID_PROJECT`. The second foreign key on this table points to the primary key of `T_PERSON`.

In order to insert a project with a few geeks that can program in Java into the data store, the following code can be used:

```
session.getTransaction().begin();
List<Geek> resultList = session.createQuery("from Geek as geek
        where geek.favouriteProgrammingLanguage = ?").setString(0, "Java").list();
Project project = new Project();
project.setTitle("Java Project");
for (Geek geek : resultList) {
        project.getGeeks().add(geek);
        geek.getProjects().add(project);
}
session.save(project);
session.getTransaction().commit();
```

The initial query selects all geeks that have "Java" as their favorite programming language. Then a new instance of `Project` is created and all geeks that are in the result set of the query are added to the project's set of geeks. On the other side of the relation the project is added to the set of projects for the geek. Finally the project is stored and the transaction gets committed.

After having executed this code, the database looks like the following:

```
sql> select * from t_person;
ID | PERSON_TYPE               | FIRST_NAME | LAST_NAME | ID_ID_CARD | FAV_PROG_LANG
1  | hibernate.entity.Person | Homer      | Simpson   | 2          | null
3  | hibernate.entity.Geek   | Gavin      | Coffee    | null       | Java
4  | hibernate.entity.Geek   | Thomas     | Micro     | null       | C#
5  | hibernate.entity.Geek   | Christian  | Cup       | null       | Java

sql> select * from t_project;
ID | TITLE
7  | Java Project

sql> select * from t_geeks_projects;
ID_PROJECT | ID_GEEK
7          | 5
7          | 3
```

The first select reveals that only the two geeks with id 3 and 5 have denoted that Java is their favorite programming language. Hence the project with title "Java Project" (id: 7) consist of the two geeks with ids 3 and 5 (last select statement).

5.4 Component

Object-Oriented design rules suggest to extract commonly used fields to a separate class. The `Project` class above for example misses still a start and end date. But as such a period of time can be used for other entities as well, we can create a new class called `Period` that encapsulates the two fields `startDate` and `endDate`:

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

        public Date getStartDate() {
                return startDate;
        }

        public void setStartDate(Date startDate) {
                this.startDate = startDate;
        }

        public Date getEndDate() {
                return endDate;
        }

        public void setEndDate(Date endDate) {
                this.endDate = endDate;
        }
}

public class Project {
        ...
        private Period period;
        ...
        public Period getPeriod() {
                return period;
        }

        public void setPeriod(Period period) {
                this.period = period;
        }
}
```

But we do not want that Hibernate creates a separate table for the period as each `Project` should only have exactly one start and end date and we want to circumvent the additional join. In this case Hibernate can map the two fields in the embedded class `Period` to the same table as the class `Project`:

```
<hibernate-mapping package="hibernate.entity">
    ...
    <class name="Project" table="T_PROJECT">
        <id name="id" column="ID">
            <generator class="sequence"/>
        </id>
        <property name="title" column="TITLE"/>
        <set name="geeks" table="T_GEEKS_PROJECTS">
            <key column="ID_PROJECT"/>
            <many-to-many column="ID_GEEK" class="Geek"/>
        </set>
        <component name="period">
            <property name="startDate" column="START_DATE"/>
            <property name="endDate" column="END_DATE"/>
        </component>
    </class>
    ...
```

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

The way how to map this embedded class to fields of the table T_PROJECT is to use the component element and provide the name of the field in the Project class for the name attribute. The two fields of the class Period are then just declared as properties of the component.

This results in the following DDL statement:

```
...
Hibernate:
    create table T_PROJECT (
        ID bigint not null,
        TITLE varchar(255),
        START_DATE timestamp,
        END_DATE timestamp,
        primary key (ID)
    )
...
```

Although the fields for START_DATE and END_DATE are located in a separate class, Hibernate adds them to the table T_PROJECT. The following code creates a new project and adds a period to it:

```
Project project = new Project();
project.setTitle("Java Project");
Period period = new Period();
period.setStartDate(new Date());
project.setPeriod(period);
...
session.save(project);
```

This results in the following data situation:

```
sql> select * from t_project;
ID | TITLE        | START_DATE              | END_DATE
7  | Java Project | 2015-01-01 19:45:12.274 | null
```

To load the period together with the project no additional code has to be written, the period is automatically loaded and initialized:

```
List<Project> projects = session.createQuery("from Project as p where p.title = ?")
        .setString(0, "Java Project").list();
for (Project project : projects) {
        System.out.println("Project: " + project.getTitle() + " starts at " + project. ↩
            getPeriod().getStartDate());
}
```

Just in case all fields of the period have been set to NULL in the database, Hibernate also sets the reference to Period to null.

Chapter 6

User-defined Data Types

When working for example with a legacy database it can happen that certain columns are modelled in a different way than Hibernate would map them. The `Boolean` data type for example is mapped on an H2 database to the type `boolean`. If the original development team has decided to map boolean values using a string with the value "0" and "1", Hibernate allows to implement user-defined types that are used for the mapping.

Hibernate defines the interface `org.hibernate.usertype.UserType` that has to be implemented:

```
public interface UserType {
    int[] sqlTypes();
    Class returnedClass();
    boolean equals(Object var1, Object var2) throws HibernateException;
    int hashCode(Object var1) throws HibernateException;
    Object nullSafeGet(ResultSet var1, String[] var2, SessionImplementor var3, Object var4) ↩
         throws HibernateException, SQLException;
    void nullSafeSet(PreparedStatement var1, Object var2, int var3, SessionImplementor var4 ↩
        ) throws HibernateException, SQLException;
    Object deepCopy(Object var1) throws HibernateException;
    boolean isMutable();
    Serializable disassemble(Object var1) throws HibernateException;
    Object assemble(Serializable var1, Object var2) throws HibernateException;
    Object replace(Object var1, Object var2, Object var3) throws HibernateException;
}
```

Simple implementations for those methods that are not specific for our problem are shown below:

```
@Override
public boolean equals(Object x, Object y) throws HibernateException {
        if (x == null) {
                return y == null;
        } else {
                return y != null && x.equals(y);
        }
}

@Override
public int hashCode(Object o) throws HibernateException {
        return o.hashCode();
}

@Override
public Object deepCopy(Object o) throws HibernateException {
        return o;
}
```

```
@Override
public boolean isMutable() {
        return false;
}

@Override
public Serializable disassemble(Object o) throws HibernateException {
        return (Serializable) o;
}

@Override
public Object assemble(Serializable cached, Object owner) throws HibernateException {
        return cached;
}

@Override
public Object replace(Object original, Object target, Object owner) throws ↩
    HibernateException {
        return original;
}
```

The interesting part of the `UserType` are the methods `nullSafeGet()` and `nullSafeSet()`:

```
@Override
public Object nullSafeGet(ResultSet resultSet, String[] strings,
        SessionImplementor sessionImplementor, Object o) throws HibernateException, ↩
            SQLException {
        String str = (String) StringType.INSTANCE.nullSafeGet(resultSet, strings[0], ↩
            sessionImplementor, o);
        if ("1".equals(str)) {
                return Boolean.TRUE;
        }
        return Boolean.FALSE;
}

@Override
public void nullSafeSet(PreparedStatement preparedStatement, Object value,
        int i, SessionImplementor sessionImplementor) throws HibernateException, ↩
            SQLException {
        String valueToStore = "0";
        if (value != null) {
                Boolean booleanValue = (Boolean) value;
                if (booleanValue.equals(Boolean.TRUE)) {
                        valueToStore = "1";
                }
        }
        StringType.INSTANCE.nullSafeSet(preparedStatement,valueToStore, i, ↩
            sessionImplementor);
}
```

The method `nullSafeGet()` uses Hibernate's `StringType` implementation to extract the string representation of the boolean value from the `ResultSet` of the underlying query. If the returned string equals "1" the method returns "true", otherwise it returns "false".

Before an `insert` statement can be executed, the boolean value passed in as parameter `value` has to be "decoded" into either the string "1" or the string "0". The method `nullSafeSet()` then uses Hibernate's `StringType` implementation to set this string value on the `PreparedStatement`.

Finally we have to tell Hibernate which kind of object is returned from `nullSafeGet()` and which type of column it should use for this type:

```
@Override
```

```
public int[] sqlTypes() {
        return new int[]{ Types.VARCHAR };
}

@Override
public Class returnedClass() {
        return Boolean.class;
}
```

Having implemented the `UserType` interface, an instance of this class can now be given to the `Configuration`:

```
Configuration configuration = new Configuration();
configuration.configure("hibernate.cfg.xml");
configuration.registerTypeOverride(new MyBooleanType(), new String[]{"MyBooleanType"});
...
```

`MyBooleanType` is here our implementation of the `UserType` interface, whereas the `String` array defines how to reference this type in the mapping file:

```
<hibernate-mapping package="hibernate.entity">
    <class name="IdCard" table="T_ID_CARD">
        <id name="id" column="ID">
            <generator class="sequence"/>
        </id>
        <property name="idNumber" column="ID_NUMBER"/>
        <property name="issueDate" column="ISSUE_DATE"/>
        <property name="valid" column="VALID" type="MyBooleanType"/>
    </class>
    ...
</hibernate-mapping>
```

As can be seen from the snippet above, the new type "MyBooleanType" is used for the boolean property of the table `T_ID_CARD`:

```
sql> select * from t_id_card;
ID | ID_NUMBER | ISSUE_DATE              | VALID
2  | 4711      | 2015-03-27 11:49:57.533 | 1
```

Chapter 7

Interceptors

A project may come with the requirement that for each entity/table the timestamp of its creation and its last update should be tracked. Setting these two values for each entity on all insert and update operations is a fairly tedious task. Therefore Hibernate offers the ability to implement interceptors that are called before an insert or update operation is performed. This way the code to set the creation and update timestamp can be extracted to a single place in the code base and does not have to be copied to all locations where it would be necessary.

As an example we are going to implement an audit trail that tracks the creation and update of the `Project` entity. This can be done by extending the class `EmptyInterceptor`:

```
public class AuditInterceptor extends EmptyInterceptor {

        @Override
        public boolean onSave(Object entity, Serializable id, Object[] state,
                String[] propertyNames, Type[] types) {
                if (entity instanceof Auditable) {
                        for ( int i=0; i < propertyNames.length; i++ ) {
                                if ( "created".equals( propertyNames[i] ) ) {
                                        state[i] = new Date();
                                        return true;
                                }
                        }
                        return true;
                }
                return false;
        }

        @Override
        public boolean onFlushDirty(Object entity, Serializable id, Object[] currentState,
                Object[] previousState, String[] propertyNames, Type[] types) {
                if (entity instanceof Auditable) {
                        for ( int i=0; i < propertyNames.length; i++ ) {
                                if ( "lastUpdate".equals( propertyNames[i] ) ) {
                                        currentState[i] = new Date();
                                        return true;
                                }
                        }
                        return true;
                }
                return false;
        }
}
```

As the class `EmptyInterceptor` already implements all methods defined in the interface `Interceptor`, we only have to override the methods `onSave()` and `onFlushDirty()`. In order to easily find all entities that have a field `created` and lastUpdate we extract the getter and setter methods for these entities into a separate interface called Auditable:

```
public interface Auditable {
        Date getCreated();
        void setCreated(Date created);
        Date getLastUpdate();
        void setLastUpdate(Date lastUpdate);
}
```

With this interface it is easy to check whether the instance passed into the interceptor is of type Auditable. Unfortunately we cannot modify the entity directly through the getter and setter methods but we have to use the two arrays propertyNames and state. In the array propertyNames we have to find the property created (lastUpdate) and use its index to set the corresponding element in the array state (currentState).

Without the appropriate property definitions in the mapping file Hibernate will not create the columns in the tables. Hence the mapping file has to be updated:

```
<hibernate-mapping>
        ...
    <class name="Project" table="T_PROJECT">
        <id name="id" column="ID">
            <generator class="sequence"/>
        </id>
        <property name="title" column="TITLE"/>
        <set name="geeks" table="T_GEEKS_PROJECTS">
            <key column="ID_PROJECT"/>
            <many-to-many column="ID_GEEK" class="Geek"/>
        </set>
        <component name="period">
            <property name="startDate" column="START_DATE"/>
            <property name="endDate" column="END_DATE"/>
        </component>
        <property name="created" column="CREATED" type="timestamp"/>
        <property name="lastUpdate" column="LAST_UPDATE" type="timestamp"/>
    </class>
        ...
</hibernate-mapping>
```

As can be seen from the snippet above, the two new properties created and lastUpdate are of type timestamp:

```
sql> select * from t_person;
ID | PERSON_TYPE              | FIRST_NAME | LAST_NAME | CREATED                 |  ↩
    LAST_UPDATE | ID_ID_CARD | FAV_PROG_LANG
1  | hibernate.entity.Person | Homer      | Simpson   | 2015-01-01 19:45:42.493 | null  ↩
           | 2           | null
3  | hibernate.entity.Geek   | Gavin      | Coffee    | 2015-01-01 19:45:42.506 | null  ↩
           | null        | Java
4  | hibernate.entity.Geek   | Thomas     | Micro     | 2015-01-01 19:45:42.507 | null  ↩
           | null        | C#
5  | hibernate.entity.Geek   | Christian  | Cup       | 2015-01-01 19:45:42.507 | null  ↩
           | null        | Java
```

Chapter 8

Download

This was a tutorial on JBoss Hibernate.

Download You can download the full source code of this tutorial here: **hibernate-tutorial-sources**.