



**TECHNISCHE
UNIVERSITÄT
WIEN**

**VIENNA
UNIVERSITY OF
TECHNOLOGY**

Web Engineering

Datenbankprogrammierung Teil 2 (JDBC und O/R)

Studienbrief DBPR2

Version 1.2 (Juli 2005)

Jürgen Falb

Inhaltsverzeichnis

0.	Übersicht.....	3
0.1.	Lehrziele.....	3
0.2.	Lehrstoff.....	3
0.3.	Aufgaben, Übungen	3
0.4.	Voraussetzungen	3
0.5.	Literatur.....	4
1.	JDBC – Java Database Connectivity	5
1.1.	Arten von JDBC Treibern	5
1.2.	Aufbau einer Verbindung.....	6
1.3.	Beispiel einer Klasse zum Suchen von Büchern.....	6
2.	SQLJ (*).....	9
2.1.	SQLJ Pre-Compiler:.....	10
2.2.	Benannte Iteratoren und Positionsiteratoren	10
2.3.	Aufruf von Stored Procedures.....	12
3.	Das Persistenz-Framework Hibernate	13
3.1.	O/R Mapping.....	13
3.1.1.	Mapping einer einfachen Klasse	14
3.2.	Ein einfaches Beispiel.....	20
3.3.	Assoziationsabbildung	22
3.3.1.	Unidirektionale Beziehungen	22
3.3.2.	Bidirektionale Beziehungen	24
3.4.	Arbeiten mit Objekten.....	26
3.4.1.	HQL (Hibernate Query Language).....	27
3.4.2.	Criteria Queries	28
3.4.3.	Native SQL.....	29
3.5.	Anhang zur Beschreibung von Hibernate	29
4.	Lehrzielorientierte Fragen.....	31

0. Übersicht

0.1. Lehrziele

Das primäre Lehrziel der Studieneinheit **Datenbankprogrammierung Teil 2 (JDBC und O/R)** ist die Beherrschung des programmatischen Zugriffs auf Datenbanken anhand von JDBC und SQLJ. Desweiteren soll anhand des Persistenzframeworks Hibernate Verständnis für das objektrelationale Mapping hergestellt werden.

0.2. Lehrstoff

In diesem Studienbrief wird zuerst auf die Struktur und Funktionsweise von JDBC eingegangen. Daran anschließend wird das Konzept der direkten Einbindung von SQL Statements in Programme anhand von SQLJ dargestellt. Die Erläuterung beider Konzepte wird durch kurze Beispiele unterstützt. Danach wird das objektrelationale Mapping des Persistenzframeworks Hibernate erläutert.

Abschnitte, die mit einem (*) gekennzeichnet sind, dienen der freiwilligen Stoffergänzung und werden nicht geprüft.

0.3. Aufgaben, Übungen

Der letzte Abschnitt enthält eine Sammlung lehrzielorientierter Fragen zur Selbstüberprüfung. Die Beantwortung sollte nach Durcharbeiten des Studienbriefes möglich sein. Treten bei einer Frage Probleme auf, so versuchen Sie diese mit ihren Studienkollegen zu lösen. Ist das nicht möglich, können Sie mich per eMail direkt kontaktieren oder Sie stellen die entsprechende Frage in der nächsten Übungsstunde.

0.4. Voraussetzungen

Der vorliegende Kurs setzt Kenntnisse des Software-Engineering und der Netzwerkdienste voraus, sowie Objektorientierte Methoden.

Die Studieneinheit **Datenbankprogrammierung Teil 2 (JDBC und O/R)** baut auf den Studieneinheiten SQL1, SQL2 und DBPR1 auf.

0.5. Literatur

- [1] Heuer, A.; Saake, G.: „Datenbanken: Konzepte und Sprachen“, International Thomson Publishing, Bonn, 2.Auflage, 2000.
- [2] Heuer, A.; Saake, G.; Sattler, K: „Datenbanken kompakt“, mitp, Bonn, 2001.
- [3] <http://developer.java.sun.com/developer/onlineTraining/Database/JDBC20Intro/JDBC20.html>, JDBC 2.0 Tutorial.

1. JDBC – Java Database Connectivity

Dieses Kapitel hat zum Ziel die JDBC Abschnitte aus dem Kompaktbuch von Heuer und Saake zu vertiefen und mit Beispielen zu erläutern. Das Kapitel bietet keinen durchgängigen, und vollständigen Überblick über JDBC. Vielmehr vertieft es ganz konkrete Punkte zu JDBC und setzt somit die Durcharbeitung der Abschnitte im Buch von Heuer und Saake voraus.

1.1. Arten von JDBC Treibern

Um auf eine Datenbank aus Java heraus über JDBC zugreifen zu können, ist ein entsprechender JDBC Datenbanktreiber notwendig. Es gibt vier verschiedene Arten von JDBC Treibern, die je nach Anwendungsszenario und Verfügbarkeit zum Einsatz gelangen. Die folgende Abbildung illustriert die vier Arten:

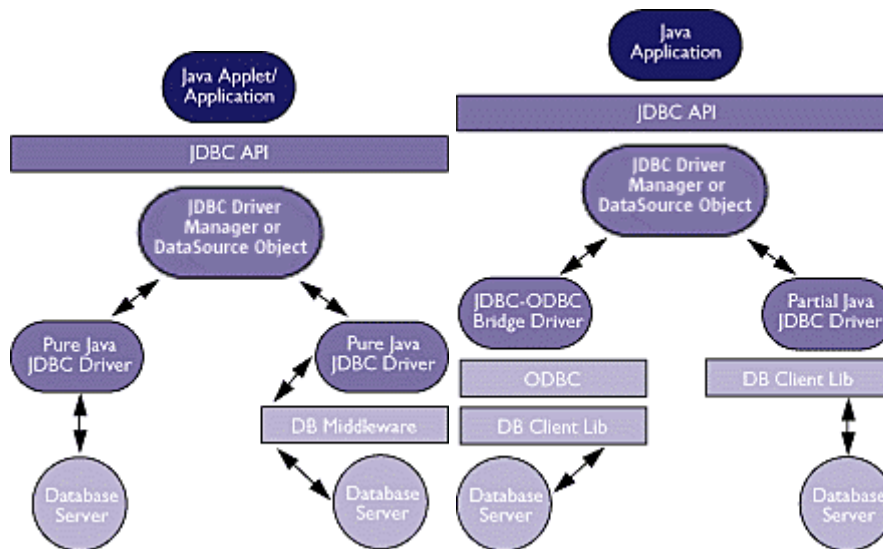


Abb. 1.1: Arten von JDBC Treibern

Die ersten beiden Treiberarten sind rein in Java implementiert und verbinden sich direkt auf die Datenbank oder eine entsprechende DB Middleware. Durch die reine Java Implementierung eignen sich diese Treiber besonders für den Einsatz in einem Client bzw. in einem Applet, da die Treiber keine zusätzlichen Bibliotheken benötigen. Ein Beispiel für so einen Treiber wäre der „JDBC Thin Treiber“ von Oracle.

Ein JDBC-ODBC Treiber gelangt meist dann zum Einsatz, wenn keine entsprechenden JDBC-Treiber zur Verfügung stehen. Dies ist zum Beispiel beim Zugriff auf MS Access DB der Fall. Ein JDBC-ODBC Bridge Treiber ist im Java 2 SDK inkludiert.

Die letzte Treibervariante repräsentiert JDBC Treiber, die nur zum Teil in Java implementiert sind und entsprechende DB Client Bibliotheken benötigen. Diese Variante wird häufiger in Applikationsservern verwendet, da diese Treiber oft einen größeren Funktionsumfang bieten und die Mechanismen der Datenbankhersteller zum Verbindungsaufbau verwendet. Ein Beispiel für diese Art von Treibern ist der „JDBC OCI Treiber“ von Oracle, der eine entsprechende Oracle Client Installation voraussetzt. Der Treiber baut in diesem Fall die Verbindung zur DB über proprietäre Kanäle (z.B. SQL Net) auf.

1.2. Aufbau einer Verbindung

Der Aufbau einer Verbindung erfolgt in zwei Schritten:

1. Laden des Treibers
2. Herstellen einer Verbindung

Das Laden des Treibers erfolgt durch Laden der entsprechenden Java Klasse, wie in folgenden Beispielen dargestellt:

```
Class.forName("exgwe.sql.gweMySQLDriver");  
Class.forName("oracle.jdbc.driver.OracleDriver");
```

Das Herstellen einer Verbindung erfolgt über den „DriverManager“. Dazu muss ihm eine JDBC URL, ein Benutzername und ein Passwort übergeben werden:

```
DriverManager.getConnection(  
"jdbc:mysql://gutemine.ict.tuwien.ac.at:3306/test", "icss5", "icss5");  
DriverManager.getConnection(  
"jdbc:oracle:thin:@gutemine.ict.tuwien.ac.at:1521:iccc", "icss5", "icss5");
```

Wie man erkennen kann, können JDBC URLs verschiedenes Aussehen besitzen. Gemeinsam ist ihnen die Struktur „jdbc:<database>:<datasource>“ wobei die Struktur der Datenquelle von Datenbankhersteller zu Datenbankhersteller unterschiedlich ist.

1.3. Beispiel einer Klasse zum Suchen von Büchern

Das folgende Beispiel basiert auf dem Universitätsbeispiel und soll die Verwendung von JDBC verdeutlichen:

```
import java.sql.Connection;  
import java.sql.DriverManager;  
import java.sql.PreparedStatement;  
import java.sql.ResultSet;  
  
public class Buchverleih {
```

```

public void suchen(String name, String keywords, String author) {
    Connection con = null;
    PreparedStatement stmt = null;
    ResultSet result = null;
    try {
        Class.forName("oracle.jdbc.driver.OracleDriver");
        con = DriverManager.getConnection(
            "jdbc:oracle:thin:@195.245.225.70:1521:db",
            "dbsa040501", "dbsa040501");
        stmt = con.prepareStatement("select Titel, Buecher.ISBN ISBN, "+
            "Buecher.Verlagsname Verlagsname,verlagsort,buch_exemplare.auflage "+
            "Auflage,Jahr,Seiten,Preis, count(*) Anzahl "+
            "from Buecher,verlage,Buch_exemplare, Buch_versionen "+
            "where Buecher.verlagsname=Verlage.Verlagsname and Buecher.ISBN="+
            "Buch_Versionen.ISBN and Buch_versionen.auflage="+
            "buch_exemplare.auflage and Buecher.isbn=Buch_Exemplare.isbn "+
            "and (lower(Titel) like '"+name.toLowerCase()+"%') "+
            "and (('"+author+"' is null) or ('"+author.toLowerCase()+"' in "+
            "(select lower(Autor) from Buch_Autor where "+
            "Buch_Autor.isbn=Buecher.isbn))) "+
            "and (('"+keywords+"' is null) or ('"+keywords.toLowerCase()+"' "+
            "' in (select lower(Stichwort) from Buch_Stichwort where "+
            "Buch_Stichwort.isbn=Buecher.isbn))) group by Titel,Buecher.ISBN,"+
            "Buecher.Verlagsname,verlagsort,buch_exemplare.auflage,Jahr,Seiten,"+
            "Preis order by titel");
        result = stmt.executeQuery();
        stmt = con.prepareStatement("select * from buch_autor where isbn = ?");

        System.out.println("Gefundene Buecher:");
        while (result.next()) {
            System.out.println();
            System.out.println(result.getString("Titel"));
            stmt.setString(1,result.getString("ISBN"));
            System.out.print("  Autoren: ");
            ResultSet result2 = stmt.executeQuery();
            while (result2.next()) {
                System.out.print(result2.getString("Autor")+", ");
            }
            result2.close();
            System.out.println("  ISBN: "+result.getString("ISBN"));
            System.out.println("  Verlag: "+result.getString("Verlagsname"));
            System.out.println("  Ort: "+result.getString("Verlagsort"));
            System.out.println("  Auflage: "+result.getString("Auflage"));
            System.out.println("  Jahr: "+result.getString("Jahr"));
            System.out.println("  Seiten: "+result.getString("Seiten"));
            System.out.println("  Preis: "+result.getString("Preis"));
        }
    }
    catch(Exception e) {
        out.println("Error: "+e.getMessage());
    }
    finally {
        if (result != null) result.close();
        if (stmt != null) stmt.close();
        if (con != null) con.close();
    }
}
}

```

Das Beispiel zeigt die Verwendung zweier PreparedStatements. In der ersten SQL Anfrage wird eine Liste von Büchern abgefragt, die den Suchkriterien entsprechen. Mit der while-Schleife wird über das ResultSet iteriert und die Daten des Buches abgefragt. Zusätzlich wird für jedes Buch die Liste der Autoren über das zweite PreparedStatement abgefragt. Hierbei wird in bei jeder Ausführung der Parameter „ISBN“ mit Hilfe der Methode „setString()“ gesetzt.

Mit Hilfe des „try-catch“ Blockes werden alle auftretenden Fehler (inkl. SQLException) abgefangen und eine Fehlermeldung ausgegeben. In jedem Fall werden am Schluss die ResultSets, Statements und Verbindungen ordnungsgemäß geschlossen.

Sie können dieses Beispiel ausprobieren, wenn Sie Benutzername und Passwort durch Ihr eigenes ersetzen und eine passende „main()“ Methode hinzufügen.

2. SQLJ (*)

Dieses Kapitel stellt ergänzende Informationen zum Abschnitt SQLJ aus dem Kompaktbuch von Heuer und Saake zur Verfügung.

SQLJ stellt die Embedded SQL Lösung für Java dar. Im folgenden ist ein Beispiel dargestellt, welches ein INSERT Statement in SQLJ realisiert und einer JDBC Lösung gegenüberstellt.

```
// SQLJ
int n;
#sql { INSERT INTO Gehalt VALUES (:n)};

// JDBC
int n;
Statement stmt =
    conn.prepareStatement(„INSERT INTO Gehalt VALUES (?)“);
stmt.setInt(1,n);
stmt.execute();
stmt.close();
```

Wie aus obigem Beispiel ersichtlich werden SQLJ Statements mit dem Schlüsselwort “#sql” eingeleitet und in geschweifte Klammern eingeschlossen. Als Hostvariablen können alle im Java-Code deklarierten Java-Variablen verwendet werden. Eine Zuordnung zwischen den Datentypen der Java-Variablen und der SQL-Datentypen kann aus der folgenden Tabelle entnommen werden:

Java Datentyp	java.sql.Types Datentyp
boolean	BIT
byte	TINYINT
short	SMALLINT
int	INTEGER
long	BIGINT
float	REAL
double	DOUBLE
Boolean	BIT
Byte	TINYINT
Short	SMALLINT
Integer	INTEGER
Double	DOUBLE
String	VARCHAR

2.1. SQLJ Pre-Compiler:

Um nun von einem Quellcode, welcher SQLJ Code beinhaltet zu ausführbaren Java Klassen zu gelangen ist ein SQLJ Pre-Compiler notwendig, der die SQLJ Statements in gültige Java Statements umgesetzt. Der Übersetzungsvorgang ist in der folgenden Abbildung dargestellt:

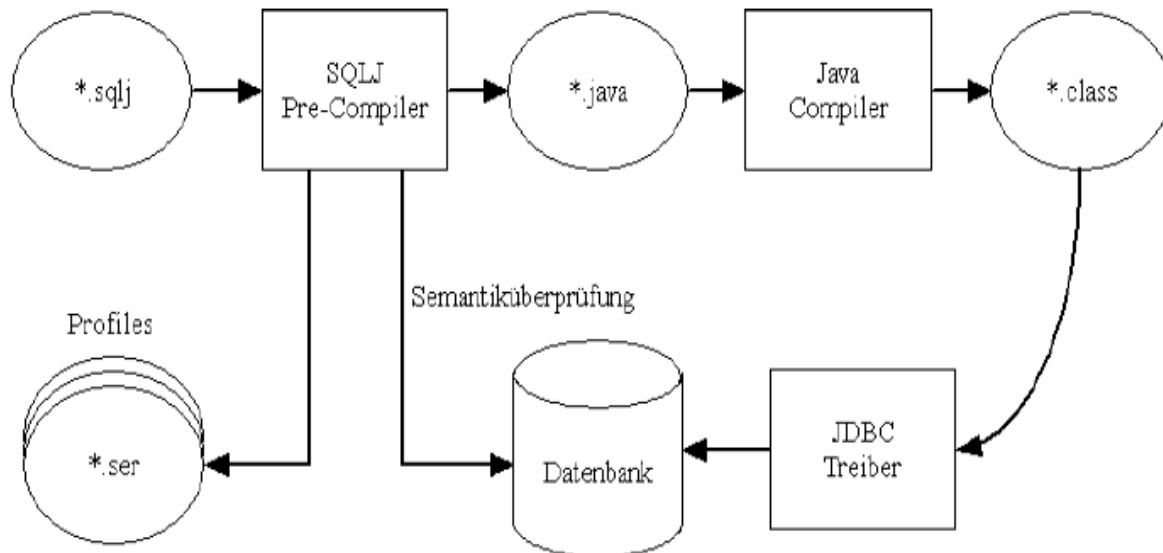


Abb. 2.1: Ablauf des Übersetzungsvorgangs

Der Vorteil von SQLJ besteht darin, dass der Pre-Compiler zusätzlich eine strengere Typüberprüfung anhand der Datenbankinformationen durchführen kann, wie in der Abbildung angedeutet. Entsprechende SQLJ Pre-Compiler sind bei den Datenbankherstellern erhältlich.

Die generierten Java-Klassen verwenden zur Laufzeit wieder JDBC um auf die Datenbank zugreifen zu können.

2.2. Benannte Iteratoren und Positionsiteratoren

Zum Abfragen von Ergebnismengen, die von SELECT Statement zurückgeliefert werden, benötigt man in SQLJ Iteratoren, die dem Cursor-Prinzip entsprechen. In SQLJ gibt es zwei Arten von Iteratoren: Benannte Iteratoren und Positionsiteratoren. Zwei Beispiele sollen die beiden Varianten von Iteratoren verdeutlichen.

Bei den benannten Iteratoren wird der Name einer Spalte einem Attributnamen bzw. einem Methodennamen zugeordnet. Eine Voraussetzung dafür ist, dass der Spaltenname in einem SELECT Statement mit dem Attributnamen in der Iteratordeklaration ident ist.

```
#sql public iterator Person_Iterator(int pid, String name, String forename);

public void showAllStoredPersons() {
    Person_Iterator listOfPersons;

    #sql listOfPersons = { SELECT pid, name, forename FROM person };

    while (listOfPersons.next()) {
        System.out.println("(" +listOfPersons.pid()+", "+listOfPersons.name()+",
            "+listOfPersons.forename()+")");
    }
    listOfPersons.close();
}
```

Abb. 2.2: Beispiel für einen benannten Iterator

Bei den Positionsiteratoren wird in der Deklaration nur der Datentyp festgelegt, nicht aber der Name. Die Voraussetzung hier ist, dass die Reihenfolge der Attribute mit der Reihenfolge im SELECT Statement übereinstimmen muss.

```
#sql public iterator listOfPerson(int, String, String);

public void showAllStoredPersons() {
    Person listOfPersons;
    int pid; String surname; String forename;

    #sql listOfPersons = { SELECT pid, surname, forename FROM person };

    while (!listOfPersons.endFetch()) {
        #sql { FETCH :listOfPersons INTO :pid, :surname, :forename };
        System.out.println("(" +pid+", "+surname+", "+forename+"");
    }
    listOfPersons.close();
}
```

Abb. 2.3: Beispiel für einen Positionsiterator

2.3. Aufruf von Stored Procedures

Stored-Procedures laufen direkt in einem Datenbanksystem. Aus SQLJ heraus können sie mit dem Befehl „CALL“ aufgerufen werden. z.B:

```
#sql {CALL bsp_prozedur (:variable1, :variable2)};
```

3. Das Persistenz-Framework Hibernate

Persistenz-Frameworks haben die Aufgabe Objekte, die im Speicher einer Anwendung aufgebaut wurden, in Datenbanken abzuspeichern. Hierbei kommen meistens relationale Datenbanken zum Einsatz, die eine Abbildung (Mapping) von der objektorientierten Welt in die relationale Welt erfordern. Dieser Studienbrief geht auf Hibernate ein, das eines der weitestverbreiteten Persistenz-Frameworks für Java darstellt.

Die entsprechende Theorie zu Persistenzframeworks ist im Studienbrief DBPR1 und im Artikel <http://www.advanced-developers.de/ndo/artikel.pdf> nachzulesen.

Die Dokumentation zu Hibernate v3.0 finden Sie unter http://www.hibernate.org/hib_docs/v3/reference/en/html/.

Dieses Kapitel bietet eine Einführung in Hibernate. Als erstens wird erklärt, wie Hibernate das objektrelationale Mapping realisiert. Im Abschnitt 3.2 wird das Konzept anhand eines Beispiels erläutert. Abschnitt 3.3 zeigt, wie sich einzelne Relationen abbilden lassen. Hierbei wird sowohl auf unidirektionale als auch auf bidirektionale Relationen eingegangen. 3.4 erklärt wie man in Hibernate Objekte abfragt, manipuliert und persistiert.

3.1. O/R Mapping

Dieser Abschnitt beschreibt wie Hibernate das Mapping zwischen der objektorientierten und der relationalen Welt löst. Für das Mapping gibt es drei Spezifikationsmöglichkeiten:

1. Definition einer Mapping-Datei
2. Metadata Annotation des Quellcodes (mithilfe von XDoclet Markup oder JDK 5.0 Annotation)
3. Reiner Quellcode

In der *ersten* Variante liegt ein explizites Mapping in Form einer Datei vor auf welches wir später im Detail eingehen werden.

In der *zweiten* Variante wird die notwendige Abbildungsinformation als Metainformation zu den einzelnen Teilen des Quellcodes (Attribute, Methoden) hinzugefügt. Im Folgenden ist ein Beispiel aufgezeigt, welches die Mapping-Information für einen Customer enthält:

```
@Entity(access = AccessType.FIELD)
public class Customer implements Serializable {

    @Id;
```

```

Long id;

String firstName;
String lastName;
Date birthday;

@Transient
Integer age;

@Dependent
private Address homeAddress;

@OneToMany(cascade=CascadeType.ALL, targetEntity="Order")
@JoinColumn(name="CUSTOMER_ID")
Set orders;

// Getter/setter and business methods
}

```

Die Metainformationen sind hier durch das Zeichen „@“ gekennzeichnet. So gibt zum Beispiel `@Id` an, dass das Attribut „id“ der Primärschlüssel der Entität `Customer` (gekennzeichnet durch `@Entity`) ist. `@Transient` gibt zum Beispiel an, dass das Attribut „age“ nicht persistiert werden soll, da es jedes mal aus dem Geburtsdatum und dem heutigen Datum berechnet werden kann. Auf die anderen Metainformationen wird in späteren Kapiteln dieses Studienbriefs eingegangen.

In der *dritten* Variante – nur Spezifikation des Quellcodes – versucht Hibernate mithilfe von Java Reflection und Heuristiken ein passendes Mapping zu erstellen. Diese Variante ist sinnvoll für einfache Mappings und zur Erstellung eines Mapping-Grundgerüsts.

Für diesen Studienbrief gehen wir im Folgenden von der ersten Variante aus, sprich wir definieren ein explizites Mapping für jede zu persistierende Klasse. Der Aufbau eines solchen Mappings wird im folgenden Abschnitt anhand eines Beispiels im Detail betrachtet.

Aufbauend auf einem definierten Mapping erlaubt Hibernate die Generierung von:

- SQL Schemata für die entsprechende Datenbank
- und die Generierung von Quellcode.

3.1.1. Mapping einer einfachen Klasse

Zur Erläuterung des Mappings zwischen einer Klasse und einer Entität gehen wir von einer einfachen Java-Klasse aus, die aus der Hibernate Dokumentation stammt. Die Klasse `Cat` beschreibt eine Katze durch ihr Geburtsdatum, ihre Farbe, ihr Geschlecht, ihr Gewicht und ihren Wurf. Weiters besitzt jede Katze eine Referenz auf ihre Mutter und auf ihre eventuellen Kinder. Die entsprechende Java Klasse ist im Folgenden dargestellt:

```
package eg;
```

```
import java.util.Set;
import java.util.Date;

public class Cat {
    private Long id; // identifier

    private Date birthdate;
    private Color color;
    private char sex;
    private float weight;
    private int litterId;

    private Cat mother;
    private Set kittens = new HashSet();

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

    void setBirthdate(Date date) {
        birthdate = date;
    }
    public Date getBirthdate() {
        return birthdate;
    }

    void setWeight(float weight) {
        this.weight = weight;
    }
    public float getWeight() {
        return weight;
    }

    public Color getColor() {
        return color;
    }
    void setColor(Color color) {
        this.color = color;
    }

    void setSex(char sex) {
        this.sex=sex;
    }
    public char getSex() {
        return sex;
    }

    void setLitterId(int id) {
        this.litterId = id;
    }
    public int getLitterId() {
        return litterId;
    }

    void setMother(Cat mother) {
        this.mother = mother;
    }
    public Cat getMother() {
        return mother;
    }
    void setKittens(Set kittens) {
        this.kittens = kittens;
    }
    public Set getKittens() {
```

```

    return kittens;
}

// addKitten not needed by Hibernate
public void addKitten(Cat kitten) {
    kitten.setMother(this);
    kitten.setLitterId( kittens.size() );
    kittens.add(kitten);
}
}

```

Die Klasse „Cat“ besitzt weiters noch eine Methode „addKitten“, die von Hibernate für die Abbildung auf die entsprechenden Tabellen nicht benötigt wird.

Als nächstes wollen wir uns direkt der Mapping-Datei widmen. Die Mapping-Datei ist eine XML-Datei, die für jede zu persistierende Klasse erstellt werden muß. Im Folgenden ist die Mapping-Datei für unsere Klasse „Cat“ dargestellt:

```

<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="eg">

    <class name="Cat"
        table="cats"
        discriminator-value="C">

        <id name="id">
            <generator class="native"/>
        </id>

        <discriminator column="subclass"
            type="character"/>

        <property name="weight"/>

        <property name="birthdate"
            type="date"
            not-null="true"
            update="false"/>

        <property name="color"
            type="eg.types.ColorUserType"
            not-null="true"
            update="false"/>

        <property name="sex"
            not-null="true"
            update="false"/>

        <property name="litterId"
            column="litterId"
            update="false"/>

        <many-to-one name="mother"
            column="mother_id"
            update="false"/>

        <set name="kittens"
            inverse="true"
            order-by="litter_id">
            <key column="mother_id"/>
        </set>
    </class>
</hibernate-mapping>

```

```

        <one-to-many class="Cat" />
    </set>

    <subclass name="DomesticCat"
        discriminator-value="D">

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

    </subclass>

</class>

</hibernate-mapping>

```

<hibernate-mapping>

Dieses Tag stellt das Root-Tag einer Mapping Datei dar. Es kann einige allgemeine Zusatzinformationen beinhalten, wie z.B. das Package der Java Klasse(n), den Namen des Datenbankschemas bzw. Datenbankkatalogs oder die Zugriffsmethode, die Hibernate verwenden soll um auf Attribute der Klasse zuzugreifen (z.B. direkt oder über get/set-Methoden). Eine detaillierte Beschreibung dieses und der folgenden Tags finden Sie in [Abschnitt 5.1](#) der Hibernate Dokumentation.

<class>

Das class-Tag beschreibt das Mapping einer Klasse auf eine Tabelle. Die wichtigsten Parameter sind (alle sind optional):

- `name`: spezifiziert den Namen der Java-Klasse
- `table`: gibt den Namen der Tabelle an. Der Standardwert ist gleich der Klassenname
- `discriminator-value`: dient zur Unterscheidung der einzelnen Unterklassen. Dieser Wert wird für Tabellenspaltennamen herangezogen, die zu einzelnen Unterklassen gehören.

Einige weitere Parameter dienen zu Beschreibung von Teilen von SQL Statements (z.B. WHERE oder CHECK Klauseln) um korrekte Abfragen und Schemadefinitionen zu erstellen.

<id>

Das id-Tag spezifiziert den Primärschlüssel der Klasse und der Tabelle und kann nur aus einem Attribut bestehen. Für Primärschlüssel, die aus mehreren Attributen bestehen, kann das `composite-id`-Tag verwendet werden. Die wichtigsten Parameter des id-Tags sind:

- `name`: Name des identifizierenden Properties. z.B. `id`
- `column`: ist anzugeben, wenn der Spaltenname vom Propertynamen unterschiedlich ist.

Jedes `id`-Tag kann ein so genanntes `generator`-Tag beinhalten, welches zur Erzeugung von eindeutigen IDs herangezogen werden kann.

<generator>

Das `generator`-Tag dient der Erzeugung eindeutiger IDs. Der wichtigste Parameter ist der `class`-Parameter, der den Algorithmus zur ID-Erzeugung angibt. Hibernate unterstützt 11 verschiedene Algorithmen. Die wichtigsten sind:

- `increment`: Die Id wird vom Prozess und nicht von der DB vergeben. z.B. mittels `SELECT MAX(id)+1 FROM cat`. Es wird nicht garantiert, dass die ID eindeutig ist, wenn mehrere Prozesse gleichzeitig zugreifen.
- `identity`: unterstützt `identity` Spalten wie sie z.B. in MySQL verwendet werden.
- `sequence`: unterstützt Sequences, wie sie z.B. Oracle verwendet.
- `hilo`: generiert IDs unter Verwendung einer zusätzlichen Tabelle mittels eines hi/lo Algorithmus
- `native`: wählt je nach Datenbank einen der oberen drei Algorithmen aus
- `foreign`: Hierbei wird die ID einer anderen Tabelle verwendet. Dieser Algorithmus ist vor allem für 1:1 Beziehungen interessant.

<composite-id>

Das `composite-id`-Tag dient zur Erstellung von Primärschlüsseln aus mehreren Attributen. Es hat allerdings den Nachteil, dass ein persistiertes Objekt nicht einfach mit der `load()`-Methode geladen werden kann, da einzelne Attribute ja das Objekt identifizieren. D.h., dass in diesem Fall zu erst eine Instanz der Java-Klasse mithilfe von „new“ erstellt werden muss, dann die Schlüsselattribute gesetzt werden müssen, und zuletzt können die restlichen Daten aus der Tabelle geladen werden.

<discriminator>

Das `discriminator`-Tag definiert eine Spalte, die entscheidet, welche Unterklasse instanziiert werden soll. Sie beinhaltet, die bei den einzelnen Klassen angegebenen `discriminator-values` zur Unterscheidung. Im obigen Beispiel kann diese Spalte den Wert „C“ für eine herkömmliche Katze oder den Wert „D“ für eine `DomesticCat` beinhalten. Es ist zu beachten, dass nicht jedes Vererbungsmapping eine Diskriminatorspalte benötigt. (siehe `subclass`-Tag für Vererbungsmapping in Hibernate).

<property>

Das `property`-Tag deklariert nun ein persistentes Klassenattribut. Die wichtigsten Parameter sind:

- `name`: Name des Properties
- `column`: Name der Tabellenspalte
- `type`: Datentyp
- `update`: kann `true` oder `false` sein, je nachdem ob das Attribut mittels SQL UPDATE geändert werden darf.
- `insert`: kann `true` oder `false` sein, je nachdem ob das Attribut in SQL INSERT Anweisungen eingefügt werden darf.
- `not-null`: generiert eine not-null Spalte
- `unique`: stellt Eindeutigkeit des Attributs sicher.

<many-to-one>, <one-to-one>

Spezifizieren Attribute die zur Herstellung einer Beziehung zu einem anderen persistenten Objekt dienen. Die Parameter sind im Wesentlichen dieselben, wie beim `property`-Tag. Der wichtigste zusätzliche Parameter ist der `class`-Parameter, der die Klasse angibt auf die referenziert wird. Im `Cat`-Beispiel ist dieser Parameter weggelassen, da das `many-to-one` Attribut „`mother`“ wieder auf die eigene Klasse referenziert (Selbstbeziehung)

<set>

Dieses Tag dient zur Abbildung von Attributen, die eine Menge von Werten darstellen. Die wesentlichste Anwendung ist bei m:n und 1:n Beziehungen.

<subclass>, <joined-subclass>, <union-subclass>

Diese drei Tags dienen zum Mapping der Vererbungshierarchie auf die relationale Struktur. Hibernate unterstützt drei verschiedene Konzepte:

- Eine Tabelle für die gesamte Klassenhierarchie (subclass-Tag)
- Eine Tabelle pro Unterklasse (joined-subclass-Tag)
- Eine Tabelle für jede konkrete Klasse (union-subclass-Tag)

In Hibernate ist es auch möglich, für jeden Vererbungszeitweig eine unterschiedliche Strategie zu verwenden (Allerdings darf in der Wurzelklasse nicht gemischt werden).

Jedes der drei subclass-Tags kann wieder alle anderen Tags zur Spezifikation des Mappings der Unterklasse beinhalten.

3.2. Ein einfaches Beispiel

Anhand eines Beispiels soll hier erläutert werden, wie man Hibernate konfiguriert und zu einem ausführbaren Programm gelangt. Es soll an dieser Stelle gleich gesagt werden, dass die hier vorgestellte Möglichkeit nur eine von vielen ist. Hibernate erlaubt mehrere Konfigurationsmöglichkeiten, die an unterschiedliche Technologien angepasst sind (z.B. EJB, Web Applications, ...).

Die einfachste Variante ist die Spezifikation aller Konfigurationsparameter in einer hibernate.cfg.xml Datei.

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

<hibernate-configuration>

    <!-- a SessionFactory instance -->
    <session-factory>

        <!-- properties -->
        <property name="connection.driver_class">
            oracle.jdbc.driver.OracleDriver
        </property>
        <property name="connection.url">
            jdbc:oracle:thin:@195.245.225.70:1521:db
        </property>
```

```

<property name="connection.username">dbsa040501</property>
<property name="connection.password">dbsa040501</property>
<property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
<property name="show_sql">false</property>
<property name="transaction.factory_class">
    org.hibernate.transaction.JDBCTransactionFactory
</property>

<!-- mapping files -->
<mapping resource="eg/Cat.hbm.xml"/>

</session-factory>
</hibernate-configuration>

```

Die Konfigurationsdatei beinhaltet im Wesentlichen, die Datenbankverbindung und die einzelnen zu verwendenden Mapping-Dateien.

In diesem Fall kann Hibernate im Programm mit einer einfachen Zeile initialisiert werden:

```
SessionFactory sf = new Configuration().configure().buildSessionFactory();
```

Ein einfaches Programm zum Erstellen und wieder Laden einer Instanz kann nun folgendermaßen aussehen:

```

public static void main(String[] args) {
    try {
        Session session = HibernateUtil.currentSession();

        Transaction tx = session.beginTransaction();

        Cat princess = new Cat();
        princess.setBirthdate(new Date());
        princess.setColor(Color.WHITE);
        princess.setSex('F');
        princess.setWeight(7.4f);

        session.save(princess);
        tx.commit();

        tx = session.beginTransaction();

        Query query = session.createQuery("select c from Cat as c where c.sex = :sex");
        query.setCharacter("sex", 'F');
        for (Iterator it = query.iterate(); it.hasNext();) {
            Cat cat = (Cat) it.next();
            System.out.println("Female Cat: " + cat.getColor().getColor() + "/"
                + cat.getWeight());
        }

        tx.commit();

        HibernateUtil.closeSession();
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}

```

Zuerst wird eine aktuelle Session von der SessionFactory bezogen. Danach wird eine Transaktion gestartet. Jetzt wird ein neues Objekt der Klasse „Cat“ erstellt. Nachdem dieses bearbeitet wurde, wird es mit der save ()-Methode persistiert und die Transaktion mit commit abgeschlossen. In einer weiteren Transaktion werden nun Instanzen der Klasse Cat geladen deren Geschlecht weiblich ist. Dazu beinhaltet die Abfrage die Hostvariable „:sex“,

die mittels der Methode `setCharacter()` auf „F“ gesetzt wird. Anschließend werden über einen Iterator alle gefundenen Instanzen ausgegeben. (Bei jedem Neustart dieses Programms kommt eine neue Instanz der Klasse `Cat` hinzu).

3.3. Assoziationsabbildung

Hibernate unterstützt zwei Arten von Beziehungsmapping, unidirektionale und bidirektionale Beziehungen. Diese Unterscheidung betrifft die Navigationsrichtung zwischen den beiden in Beziehung stehenden Klassen. So kann bei einer unidirektionalen Beziehung von einem Objekt der Klasse A auf Objekte der Klasse B zugegriffen werden, aber nicht umgekehrt.

3.3.1. Unidirektionale Beziehungen

Hibernate unterstützt verschiedene Arten unidirektionaler Beziehungen. Beziehungen, die eine Assoziationstabelle verwenden und Beziehungen, die keine verwenden. Hierbei kann noch zwischen 1:1, 1:n und m:n Beziehungen unterschieden werden, wobei die m:n Beziehung nur mit Hilfe einer Assoziationstabelle realisiert werden kann. Im Folgenden sind die Definitionen der wichtigsten Mappings und der dazugehörigen Tabellen aufgelistet:

n:1 Mapping:

```
<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>

  <many-to-one name="address"
    column="addressId"
    not-null="true"/>
</class>

<class name="Address">
  <id name="id" column="addressId">
    <generator class="native"/>
  </id>
</class>
```

```
create table Person ( personId bigint not null primary key, addressId bigint not
null )
create table Address ( addressId bigint not null primary key )
```

Diese Variante ist die am häufigsten vorkommende Mapping-Variante. Dabei wird das „address“ Attribut der Klasse `Person` auf ein `Address`-Objekt gemappt. Mehrere Personen können dabei auf dasselbe Adressenobjekt verweisen. Es ist aber nicht möglich vom Adressenobjekt auf die dazugehörigen Personenobjekte zu schließen.

1:1 Mapping:

```

<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>

  <many-to-one name="address"
    column="addressId"
    unique="true"
    not-null="true"/>
</class>

<class name="Address">
  <id name="id" column="addressId">
    <generator class="native"/>
  </id>
</class>

```

Das 1:1 Mapping ist identisch, bis auf den zusätzlichen „unique“ Parameter im „many-to-one“ Tag der Klasse Person. In der Tabelle Person ist die Spalte „addressId“ eindeutig.

1:n Mapping mit Hilfe einer Assoziationstabelle

```

<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>

  <set name="addresses" table="PersonAddress">
    <key column="personId"/>
    <many-to-many column="addressId"
      unique="true"
      class="Address"/>
  </set>
</class>

<class name="Address">
  <id name="id" column="addressId">
    <generator class="native"/>
  </id>
</class>

```

```

create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId not null, addressId bigint not null primary
key )
create table Address ( addressId bigint not null primary key )

```

In diesem Beispiel beinhaltet jede Person eine Menge an Adressen, wobei jede Adresse nur zu einer Person gehören kann. Für dieses Mapping ist ein „set“-Tag zu inkludieren, welches die Menge der Adressen darstellt. Diese Menge wird in der Tabelle PersonAddress abgebildet. Der Mengenschlüssel „personId“ ist weiters der Fremdschlüssel auf die Tabelle Person. Da es kein „one-to-many“ Tag gibt wird mithilfe des Tags „many-to-many“ und dem Parameter „unique“ jedes Adressattribut jeder Person auf ein Adressobjekt gemappt. In der Datenbank geschieht dieses Mapping über die zusätzliche Tabelle PersonAddress.

m:n Mapping mit Hilfe einer Assoziationstabelle

```

<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>
  <set name="addresses" table="PersonAddress">
    <key column="personId"/>
    <many-to-many column="addressId"
      class="Address"/>
  </set>
</class>

<class name="Address">
  <id name="id" column="addressId">
    <generator class="native"/>
  </id>
</class>

```

```

create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bigint not null,
primary key (personId, addressId )
create table Address ( addressId bigint not null primary key )

```

Dieses Mapping ist ident zu obigem deklariert bis auf den fehlenden Parameter „unique“.

Weitere unidirektionale Mappingmöglichkeiten finden Sie in der Hibernate Dokumentation in [Abschnitt 7.2 und 7.3](#).

3.3.2. Bidirektionale Beziehungen

Hibernate unterstützt alle obigen Beziehungen auch in bidirektionaler Form. Damit kann von jeder Seite zur anderen Seite über die Objekte navigiert werden. Z.B. kann von einer Person auf alle seine Adressobjekte zugegriffen werden und umgekehrt kann vom Adressobjekt auf die dazugehörige Person im Falle eines 1:n Mappings zugegriffen werden.

Ebenso können im bidirektionalen Fall, 1:1 und 1:n (n:1) Mappings wieder mit oder ohne Assoziationstabelle realisiert werden.

1:n Mapping / n:1 Mapping

```

<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>

  <many-to-one name="address"
    column="addressId"
    not-null="true"/>
</class>

<class name="Address">
  <id name="id" column="addressId">
    <generator class="native"/>
  </id>

  <set name="people" inverse="true">
    <key column="addressId"/>
    <one-to-many class="Person"/>
  </set>
</class>

```

Im Gegensatz zu den unidirektionalen Mappings ist in diesem Beispiel in beiden Klassen ein Beziehungsmapping eingebunden. Dieses Beispiel stellt wieder eine n:1 Beziehung dar. Ein Adressobjekt kann in mehreren Personenobjekten enthalten sein. Das Mapping der Klasse Person in diesem Beispiel ist ident zu dem Mapping im Beispiel unidirektionales n:1 Mapping. Zusätzlich ist im Mapping der Klasse Address ein 1:n Mapping mit Hilfe des „set“ Tags angegeben. Der wesentliche Unterschied zu einem unidirektionalen 1:n Mapping besteht darin, dass mit dem Parameter „inverse“ angegeben wird, dass es sich um das andere Ende einer bidirektionalen Beziehung handelt.

1:1 Mapping

```
<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>

  <many-to-one name="address"
    column="addressId"
    unique="true"
    not-null="true"/>
</class>

<class name="Address">
  <id name="id" column="addressId">
    <generator class="native"/>
  </id>

  <one-to-one name="person"
    property-ref="address"/>
</class>
```

Das 1:1 Mapping ist in der einen Richtung wieder ident zum unidirektionalen 1:1 Mapping. In der anderen Richtung ist im Mapping der Klasse Address ein „one-to-one“ Tag hinzuzufügen, dessen „property-ref“ Parameter auf das Attribut der Klasse Person zeigt, über welches die Person in Beziehung mit der Adresse steht.

m:n Mapping mit Hilfe einer Assoziationstabelle

```
<class name="Person">
  <id name="id" column="personId">
    <generator class="native"/>
  </id>

  <set name="addresses">
    <key column="personId"/>
    <many-to-many column="addressId"
      class="Address"/>
  </set>
</class>

<class name="Address">
  <id name="id" column="addressId">
    <generator class="native"/>
  </id>

  <set name="people" inverse="true">
    <key column="addressId"/>
    <many-to-many column="personId">
```

```

class="Person"/>
</set>
</class>

```

Das bidirektionale m:n Mapping ist ebenfalls nach demselben Schema aufgebaut. Es entspricht dem unidirektionalen m:n Mapping, wobei wieder in der `Address` Klasse das m:n Mapping in die andere Richtung angegeben ist. Es ist dabei egal ob der „inverse“ Parameter in der Klasse `Person` oder der Klasse `Address` auftaucht.

Weitere bidirektionale Möglichkeiten des Mappings können wieder in der Hibernate Dokumentation gefunden werden.

3.4. Arbeiten mit Objekten

In Hibernate können Objekte mit denen gearbeitet wird, drei verschiedene Zustände besitzen:

- **Transient:** sind jene Objekte, die mit „new“ erzeugt wurden und noch keiner Hibernate Session zugeordnet sind. Beendet sich die Anwendung, so gehen diese Objekte verloren.
- **Persistent:** sind jene Objekte, die einer Hibernate Session zugeordnet sind. Diese Objekte haben eine Repräsentierung in der Datenbank (sie sind z.B. gerade gespeichert oder geladen worden). Solche Objekte werden von Hibernate automatisch mit der Datenbank synchronisiert, wenn sie sich ändern.
- **Detached:** sind jene Objekte, die auch nach dem Schließen einer Hibernate Session weiter existieren. Wird erneut eine Session geöffnet, können diese Objekte wieder „attached“ werden.

Die wichtigsten Funktionen, die Hibernate zum Arbeiten mit Objekten anbietet sind:

- **session.save():** Mit dieser Methode können transiente Objekte persistiert werden.
- **session.load():** erlaubt das erneute Laden von persistenten Objekten. Dazu muss die Klasse und der Primärschlüssel angegeben werden.
- **session.refresh():** erlaubt das erneute Laden aller in der Session befindlichen Objekte. Die kann notwendig sein, wenn sich Attribute aufgrund von Datenbank-Triggern ändern.
- **session.flush():** persistiert alle Änderungen von Objekten, die sich in der Session befinden.

- **session.update():** erlaubt das erneute Anbinden eines detached'en Objektes an eine neue Session. Es darf in der neuen Session allerdings kein Objekt mit derselben ID existieren. In diesem Fall kann die merge() Methode verwendet werden.
- **session.merge():** erlaubt das erneute Anbinden eines detached'en Objektes an eine neue Session. Existiert in der Session bereits ein Objekt mit derselben ID, so werden die beiden Objekte zusammengefügt.
- **session.saveOrUpdate():** persistiert entweder ein transientes Objekt, wenn es noch keine ID hat, oder fügt andernfalls das Objekt erneut einer Session hinzu.
- **session.delete():** dient zum Löschen eines persistenten Objekts. Es wird dadurch aus der der Datenbank und der Session entfernt. Es kann allerdings als transientes Objekt in der Anwendung weiter existieren.
- **session.replicate():** erlaubt das replizieren der Objekte in der Session auf eine andere Datenbank.

Die folgenden Anschnitte zeigen, wie Objekte außer mit der load() Methode mittels Abfragen geladen werden können

3.4.1. HQL (Hibernate Query Language)

HQL ist syntaktisch ähnlich zu SQL aber voll objektorientiert. Eine einfache HQL Abfrage könnte wie folgt aussehen:

```
select c from eg.Cat as c where c.sex = 'F'
```

Diese Abfrage liefert alle weiblichen Instanzen der Klasse Cat und aller abgeleiteten Klassen (z.B. DomesticCat).

Hibernate Queries erlauben die Verwendung jeder Java Klasse in der Abfrage. Die folgende Abfrage würde alle persistenten Objekte zurückliefern, da in Java jedes Objekt von java.lang.Object abgeleitet ist (Anm: der Select Teil ist bei HQL optional).

```
from java.lang.Object o
```

HQL erlaubt desweiteren auch Aggregatfunktionen:

```
select avg(cat.weight), sum(cat.weight), max(cat.weight), count(cat)  
from eg.Cat cat
```

Diese Abfrage würde anstatt persistenter Objekte, drei Zahlen zurückliefern.

Ebenso können anstatt von ganzen Instanzen auch einzelnen Attribute selektiert werden:

```
select cat.name from eg.DomesticCat cat  
where cat.name like 'fri%'
```

Die obige Abfrage würde anstatt von Instanzen eine Liste von Strings mit den Namen aller DomesticCats zurückliefern, die mit "fri" beginnen.

Weiters unterstützt HQL verschiedene Operatoren, wie z.B. „<“, „and“ oder „between“, sowie „group by“, „order by“ und Subqueries.

Ausführen einer HQL Query:

Eine HQL Query kann mit createQuery() erstellt und wie folgt ausgeführt werden:

- **list():** liefert eine Liste zurück, die alle Ergebnistupel der Abfrage enthält. Die Liste kann mit Standard Java List Methoden bearbeitet werden.

```
List cats = session.createQuery("from Cat as cat where cat.birthdate < ?")
    .setDate(0, date)
    .list();
```

- **uniqueResult():** kann verwendet werden, wenn die Abfrage nur ein Tupel zurückliefert.

```
Cat mother = (Cat) session.createQuery("select cat.mother from Cat as cat
    where cat = ?")
    .setEntity(0, fritzi)
    .uniqueResult();
```

- **iterate():** erlaubt das iterieren über die Ergebnismenge, ohne dass alle Daten auf einmal von der Datenbank abgefragt werden müssen.

```
Iterator iter = sess.createQuery("from Cat as cat where cat.birthdate <
    '1.1.2005'")
    .iterate();
while ( iter.hasNext() ) {
    Cat cat = (Cat) iter.next(); // fetch the object
    // do something with the object
}
```

Liefert eine HQL Query mehrere Objekte innerhalb einer Zeile zurück, so werden diese in ein Objektarray verpackt:

```
Iterator kittensAndMothers = sess.createQuery(
    "select kitten, mother from Cat kitten join kitten.mother mother")
    .list()
    .iterator();
while ( kittensAndMothers.hasNext() ) {
    Object[] tuple = (Object[]) kittensAndMothers.next();
    Cat kitten = tuple[0];
    Cat mother = tuple[1];
    ....
}
```

Dasselbe gilt auch für Abfragen, die mehrere skalare Werte zurückliefern, wie die obige Abfrage mit den Aggregatfunktionen.

3.4.2. Criteria Queries

Criteria Queries erlauben das programmtechnische Zusammenbauen einer Abfrage mit Hilfe eines Criteria Objektes, wie im folgenden Beispiel dargestellt:

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "F%")
    .addOrder( Order.asc("name") )
    .addOrder( Order.desc("age") )
    .setMaxResults(50)
    .list();
```

Hierbei werden die einzelnen Bedingungen, Joins und Sortierungen durch Aufruf von Methoden zu dem Criteria Objekt hinzugefügt. Das obige Beispiel selektiert alle Katzen deren Namen mit „F“ beginnt und sortiert sie aufsteigend nach Name und anschließend absteigend nach dem Alter. Weiters wird die maximale Anzahl an Datensätzen (Instanzen) auf 50 beschränkt.

Die Ausführung der Criteria Query ist ident zu HQL.

3.4.3. Native SQL

Native SQL Queries werden in Hibernate hauptsächlich im Mapping File verwendet um spezielle Statements zum Bearbeiten und Abfragen der Daten bereitzustellen. Der vollständigkeit halber sollen Sie hier anhand eines Beispiels erwähnt werden, wie Sie in einem Programm verwendet werden können:

```
String sql = "select cat.originalId as {cat.id}, " +
    "cat.mateid as {cat.mate}, cat.sex as {cat.sex}, " +
    "cat.weight*10 as {cat.weight}, cat.name as {cat.name} " +
    "from cat_log cat where {cat.mate} = :catId"
List loggedCats = sess.createSQLQuery(sql)
    .addEntity("cat", Cat.class)
    .setLong("catId", catId)
    .list();
```

Das SQL Statement hat zusätzlich Platzhalter in geschwungenen Klammern zu enthalten. Diese definieren, welches Attribut der Abfrage auf welches Attribut der zurückzuliefernden Klasse gemappt werden soll (z.B. cat.name → cat.name). Zusätzlich muß durch Aufruf der Methode „addEntity“ auf der Query festgelegt werden um welchen Typ von Objekt es sich handelt.

Weitere Informationen zu den Abfragen sind in der Hibernate Dokumentation in den Kapiteln [14](#), [15](#) und [16](#) zu finden.

3.5. Anhang zur Beschreibung von Hibernate

Im Anhang finden Sie hier die Beschreibung der zusätzlichen Klasse `HibernateUtil`, die im Beispiel „Cat“ verwendet wurde:

```
public class HibernateUtil {
    private static Log log = LogFactory.getLog(HibernateUtil.class);
    private static final SessionFactory sessionFactory;
    static {
```

```

        try {
            // Create the SessionFactory
            sessionFactory = new Configuration().configure().buildSessionFactory();
        } catch (Throwable ex) {
            // Make sure you log the exception, as it might be swallowed
            log.error("Initial SessionFactory creation failed.", ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static final ThreadLocal session = new ThreadLocal();

    public static Session currentSession() throws HibernateException {
        Session s = (Session) session.get();
        // Open a new Session, if this Thread has none yet
        if (s == null) {
            s = sessionFactory.openSession();
            session.set(s);
        }
        return s;
    }

    public static void closeSession() throws HibernateException {
        Session s = (Session) session.get();
        session.set(null);
        if (s != null)
            s.close();
    }
}

```

Diese Klasse erzeugt im Klassenkonstruktor eine neue `SessionFactory`, die von allen Objekten in allen Threads dieser Virtual Machine verwendet werden kann. Die Methode `currentSession()` stellt ein Singleton Pattern dar. Die Methode sieht nach ob es für den aktuellen Thread eine Session gibt oder nicht. Wenn es keine gibt, wird eine neue erzeugt und in der „`session`“ Variable abgespeichert. Die Methode `closeSession()` schließt die Session des Threads und entfernt sie aus der „`session`“ Variable.

4. Lehrzielorientierte Fragen

1. Erläutern Sie die prinzipiellen Schritte, die in JDBC notwendig sind um eine SQL Anfrage durchzuführen. Beschreiben Sie weiters den Aufbau einer JDBC URL.
2. Erläutern Sie die Bedeutung und Verwendungsweise der drei Arten von Statements in JDBC.
3. Welche Möglichkeiten der Transaktionssteuerung gibt es in JDBC und wie werden Sie verwendet.
4. Geben Sie ein Beispiel für eine SQLJ Anweisung an, die Hostvariablen verwendet. Welche Besonderheit haben Hostvariablen in SQLJ?
5. Wie wird bei SQLJ eine Verbindung zur Datenbank hergestellt?
6. Beschreiben Sie die Vorgangsweise beim Zugriff auf die Ergebnismenge einer SQL Anfrage unter Verwendung von SQLJ. Welche zwei Arten von Iteratoren gibt es? Geben Sie ein Beispiel an.
7. Wozu braucht man Persistenz-Frameworks?
8. Auf welche Arten kann eine Vererbungshierarchie auf das relationale Modell abgebildet werden?
9. Welche drei Zustände kann ein Objekt typischerweise in einem Persistenzframework einnehmen?
10. Erklären Sie den Unterschied zwischen den drei Abfragesprachen HQL, Criteria Queries, und native SQL.
11. Wo ist der Unterschied zwischen uni- und bidirektionalen Assoziationsabbildungen beim Persistenzmapping?
12. Auf welche Arten kann ein Mapping bei Hibernate definiert werden?