

---

# »Java-Persistenz mit Hibernate«

Technology Days 2007 - Workshop

---

doubleSlash  
Net-Business GmbH

Rainer Müller  
Fon 07541/6047-100

Müllerstr. 12 B  
D-88045 Friedrichshafen

<http://doubleSlash.de>



---

## »Inhalt«

---

- Once upon a time
- Was ist Hibernate?
- Vorteile von Hibernate
- Nachteile von Hibernate
- Architektur von Hibernate
- Objektdefinitionen
- Persistenz-Lebenszyklus
- Ein einfaches Beispiel
  - Datenbankschema
  - Objektklassen
  - XML Mapping
  - Hibernate Konfigurationsdatei
  - HibernateUtil
  - Speichern. Lesen
- Ein einfaches Beispiel
  - Ändern, Löschen
  - Query mittels HQL
  - Query mittels Criteria
  - Query mittels Example
- Exception Handling
- Fallstricke
  - Lazy-Initialisierung
  - Verhalten der Session
- Tools und Vorgehensweisen
  - Top down
  - Bottom up
  - Middle out
  - Meet in the middle

## »Once upon a time«

Wie wurde Persistenz in typischen Java-Anwendungen bisher gehandhabt?

- Man musste sich selber um die Beschaffung und Verwaltung von Connections kümmern.
- Für alle Operationen (*select*, *insert*, *update*, *delete*) wurden mehr oder weniger abstrahiert Unmengen von Methoden in separaten Klassen eines Persistenz-Layers implementiert.
- Statements wurden mehr oder weniger effizient von Hand zusammen gebaut.

Hinweis: Nebstehendes Beispiel ist sehr ineffizient bezüglich Performance, da für die Insert-Operation eine eigene Connection verwendet wird.

```
public void addPerson(Person person) throws PersistenceException {
    Connection con = null;
    PreparedStatement pstmt = null;

    String str = "";
    str += "insert into person ";
    str += "(id, lastname, age) ";
    str += "values (?, ?, ?)";

    try {
        try {
            con = DriverManager.getConnection(
                this.dbUrl, this.dbUser, this.dbPwd);

            pstmt = con.prepareStatement(str);
            pstmt.setQueryTimeout(QUERYTIMEOUT);

            pstmt.setString(1, person.getId());
            pstmt.setString(2, person.getLastName());
            pstmt.setInt(3, person.getAge());

            pstmt.executeUpdate();
        }
        finally {
            if (pstmt != null) pstmt.close();
            if (con != null) con.close();
        }
    }
    catch (SQLException se) {
        PersistenceException e = new PersistenceException(
            PersistenceException.ERROR, ME, METHOD, se);
        e.addDescription("Last SQL-statement: " + str);
        throw e;
    }
}
```

---

## » Was ist Hibernate?«

---

- Hibernate ist ein O/R Persistenz-Framework.
- Es verbindet das objektorientierte Modell von Java-Anwendungen mit dem relationalen Modell von Datenbank-Systemen
  
- Quelle: [www.hibernate.org](http://www.hibernate.org)
- Aktuelle Version: 3.2.2 GA

## » Vorteile «

- Es ist das Open-Source Persistenz-Framework der Wahl mit enormer Verbreitung, sehr aktiver Entwicklung und grosser Community
- Datenbank-Unabhängigkeit  
(es werden zahlreiche DBMS und deren SQL-Dialekte unterstützt)
- Nicht intrusiv  
(keine Ableitung eigener Klassen von Hibernate-Klassen erforderlich)
- Keine Voraussetzungen an die Laufzeitumgebung  
(verwendbar in managed und non-managed Umgebungen)
- Lässt dem Entwickler grösstmögliche Freiheit bei der Anwendungserstellung und ist äusserst flexibel einsetzbar
- Konform zum JPA-Standard
- Unterstützung durch Zusatzpakete (z.B. Annotations, EntityManager) und zahlreiche Tools (z.B. für ANT und Eclipse)
- Erste Schritte bzw. einfache Anwendungen lassen sich damit sehr leicht realisieren
- Generierung von sehr effizientem SQL
- Sehr gute Literatur verfügbar

---

## »Nachteile«

---



- Komplexe Anwendungen und Problemstellungen bedürfen eines umfassenden Know-Hows
- Man sollte wissen was man tut, was man allerdings generell sollte...

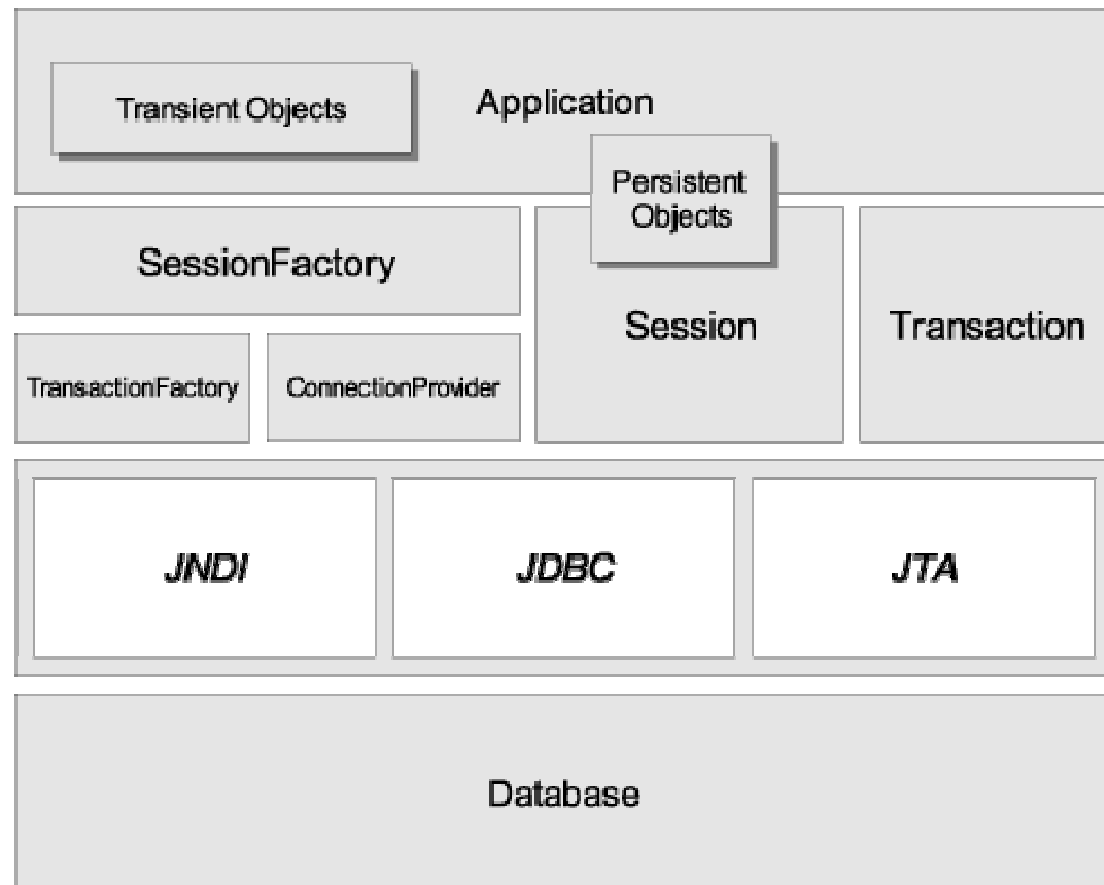
## »Architektur von Hibernate«

Nebenstehendes Schaubild zeigt die Architektur von Hibernate.

Als Anwendungsentwickler hat man in der Regel mit folgenden Komponenten bzw. Objekten zu tun:

- `SessionFactory`
- `Session`
- `Transaction`

Über die `SessionFactory` erhält man eine `Session`. Darüber wird eine `Transaction` gestartet, in der man seine Operationen durchführt.



---

## »Objektdefinitionen (1)«

---

- SessionFactory

Ein thread-safe Cache von kompilierten Mappings für eine einzelne Datenbank. Dient als Factory für `Session`-Objekte und kann optional einen Second-Level Cache mit persistenten Objekten enthalten, welche über mehrere Transaktionen hinweg wiederverwendet werden können.

- Session

Ein single-threaded, kurzlebiges Objekt, welches eine komplette Konversation zwischen der Anwendung und der Datenbank repräsentiert. Kapselt eine JDBC-Connection. Dient als Factory für `Transaction`-Objekte. Enthält den nicht deaktivierbaren First-Level Cache mit persistenten Objekten, welcher beim Navigieren in einem Objekt-Graphen genutzt wird.



---

## »Objektdefinitionen (2)«

---

- Transaction

Ein single-threaded, kurzlebiges Objekt, welches von der Anwendung zur Spezifizierung atomarer Arbeitseinheiten genutzt wird. Abstrahiert die Anwendung von darunter liegenden JDBC, JTA oder CORBA Transaktionen. In einer *Session* können mehrere Transactions erfolgen, jedoch niemals ineinander verschachtelt. Jede Operation muss zwingend in einer *Transaction* erfolgen, auch wenn es sich um eine rein lesende Operation handelt.

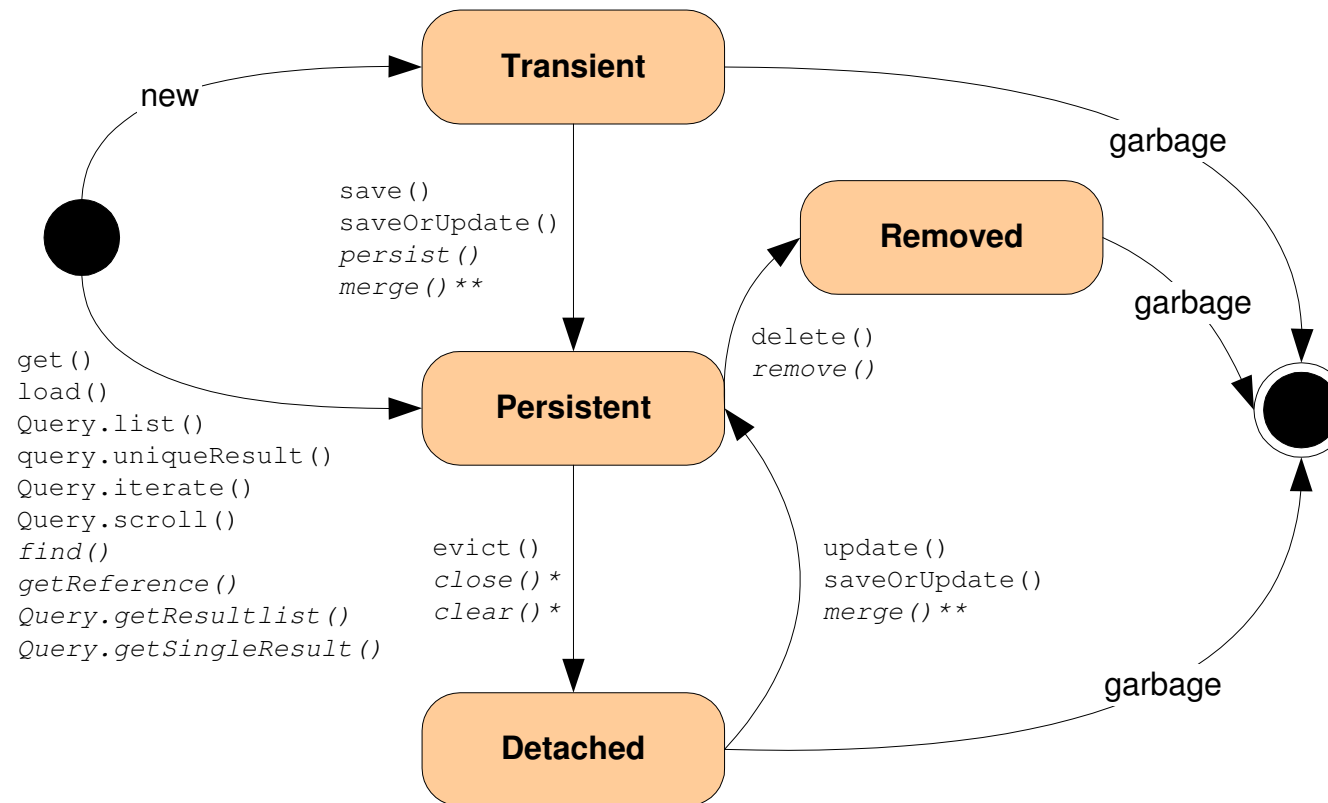
---

## »Objektdefinitionen (3)«

---

- Transiente Objekte  
Neu erzeugte Instanzen persistenter Klassen (für welche ein Hibernate-Mapping definiert ist), welche (noch) nicht über eine `Session` persistiert wurden. Es gibt somit in der Datenbank (noch) keine Entsprechung.
- Persistente Objekte  
Instanzen persistenter Klassen (z.B. gewöhnliche JavABeans/POJOs), welche aktuell mit einer `Session` verbunden sind und in der Datenbank eine Entsprechung haben.
- Detached Objekte  
Persistente Objekte, die aus einer `Session` entfernt wurden, oder deren zugehörige `Session` geschlossen wurde. Diese Objekte haben somit eine Entsprechung in der Datenbank, ihr Zustand kann jedoch veraltet sein.

## »Persistenz-Lebenszyklus«



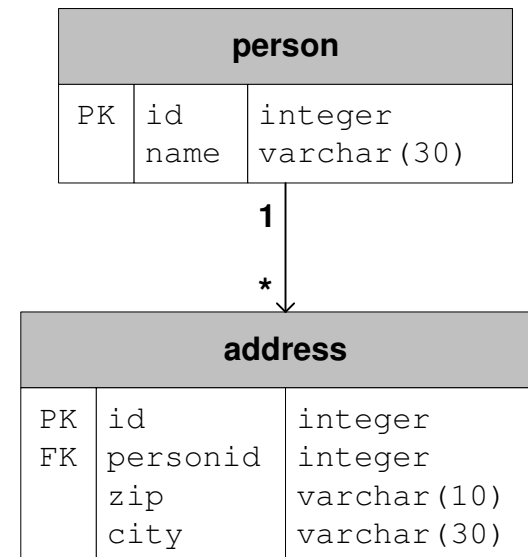
\* Hibernate Core API & JPA, betrifft alle Instanzen im Persistenz-Kontext (Session).

\*\* Merging liefert als Ergebnis eine persistente Instanz, das Original bleibt unverändert.

## » Ein einfaches Beispiel – Datenbankschema«

Nebenstehend ist das zu unserem einfachen Beispiel zugehörige Datenbank-Schema zu sehen. Dass es im Rahmen dieses Beispiels zu allererst gezeigt wird muss nicht bedeuten, dass es zu allererst erstellt werden muss (siehe „Verfahrensweisen“).

- In unserem Beispiel haben wir eine 1:N Beziehung zwischen Person und Adresse. Eine Person kann mehrere Adressen haben, z.B. einen Erstwohnsitz und einen Zweitwohnsitz.



## »Ein einfaches Beispiel – Objektklassen (1)«

- Erstellung von Objektklassen, fast genau so wie man das tun würde, wenn keine Persistierung nötig wäre.
- Seitens Hibernate wird nur vorausgesetzt, dass ein `public` Konstruktor ohne Parameter vorhanden ist.
- Üblicherweise werden für alle Objektattribute Getter- und Setter-Methoden implementiert, welche man auch einfach von Eclipse generieren lassen kann. Diese sind für Hibernate jedoch nicht zwingend erforderlich.
- Das Attribut `id` ist in diesem Beispiel der Primärschlüssel einer Person in der DB.

```
package de.doubleslash.demo.model

public class Person {
    private Integer id;
    private String name;
    private Set<Address> addresses;

    public Person() {}

    public Integer getId() {
        return this.id;
    }

    protected void setId(Integer id) {
        this.id = id;
    }

    public String getName() {
        return this.name;
    }

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

    ...
}
```

## »Ein einfaches Beispiel – Objektklassen (2) «

- Für die Verwaltung (*add*, *remove*) der Adressen einer Person definieren wir zwei Hilfsmethoden, welche uns etwas Programmierarbeit abnehmen.
- Unser Beispiel sieht eine unidirektionale Verknüpfung von `Person` und `Address` Objekten vor, d.h. über das `Person` Objekt kann man eine Kollektion von `Address` Objekten bekommen, nicht jedoch von einem `Address` Objekt das damit verknüpfte `Person` Objekt.
- Bei einer bidirektionalen Verknüpfung müssen beide Objekte korrekt miteinander verknüpft werden.

```
...  
  
public Set<Address> getAddresses() {  
    return this.addresses;  
}  
  
protected void setAddresses(Set<Address> addresses) {  
    this.addresses = addresses;  
}  
  
public void addAddress(Address address) {  
    if (address == null) {  
        throw new IllegalArgumentException(  
            "Address is null!");  
    }  
    this.addresses.add(address);  
}  
  
public void removeAddress(Address address) {  
    if (address == null) {  
        throw new IllegalArgumentException(  
            "Address is null!");  
    }  
    if (!this.addresses.remove(address)) {  
        throw new IllegalArgumentException(  
            "Address is not assigned!");  
    }  
}  
}
```

## »Ein einfaches Beispiel – Objektklassen (3) «

- Dies ist die `Address` Objektklasse. Diese enthält neben Getter- und setter-Methoden für die Attribute keine zusätzlichen Methoden.

```
package de.doubleslash.demo.model

public class Address {
    private Integer id;
    private String zip;
    private String city;

    public Address() {}

    public Integer getId() {
        return this.id;
    }

    protected void setId(Integer id) {
        this.id = id;
    }

    public String getZip() {
        return this.zip;
    }

    public void setZip(String zip) {
        this.zip = zip;
    }

    // + Getter / Setter fuer city
}
```

## »Ein einfaches Beispiel – XML Mapping«

- Es wird jeweils ein XML-Mapping für die Objektklassen Person und Address erstellt. Diese können auch per Tool anhand der Java-Klassen generiert werden. Alternativ kann man das Mapping direkt in den Java-Klassen mittels Annotations vornehmen.

```
<?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="de.doubleslash.demo.model">
  <class name="Address" table="address">
    <id name="id" type="integer" column="id">
      <generator class="native"/>
    </id>

    <property name="zip" type="string" column="zip"/>
    <property name="city" type="string" column="city"/>
  </class>
</hibernate-mapping>
```

```
<?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="de.doubleslash.demo.model">
  <class name="Person" table="person">
    <id name="id" type="integer" column="id">
      <generator class="native"/>
    </id>

    <!-- object attribut name -->
    <property name="name" type="string" column="name"/>

    <!-- uni-directional one-to-many association to Address -->
    <set
      name="addresses"
      table="address"
      access="field"
      cascade="save-update, delete"
    >
      <key column="personid" not-null="true"/>
      <one-to-many class="Address"/>
    </set>
  </class>
</hibernate-mapping>
```



## »Ein einfaches Beispiel – Hibernate Konfigurationsdatei«

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

<hibernate-configuration>
  <session-factory>
    <!-- Database connection settings -->
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="hibernate.connection.url">jdbc:mysql://horizon:3306/demodb</property>
    <property name="hibernate.connection.username">user</property>
    <property name="hibernate.connection.password">geheim</property>

    <!-- SQL dialect -->
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLInnoDBDialect</property>

    <!-- JDBC connection pool (built-in) -->
    <!--property name="connection.pool_size">10</property-->

    <!-- Disable the second-level cache -->
    <property name="cache.provider_class">org.hibernate.cache.NoCacheProvider</property>

    <!-- Enable Hibernate's automatic session context management -->
    <property name="current_session_context_class">thread</property>

    <!-- Mapping files -->
    <mapping resource="Person.hbm.xml"/>
    <mapping resource="Address.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

## »Ein einfaches Beispiel - HibernateUtil«

- Um uns die Arbeit zu erleichtern, erstellen wir noch die Hilfsklasse `HibernateUtil`, von welcher man über eine statische Methode überall und jederzeit die immer benötigte `SessionFactory` bekommen kann.
- Diese sorgt auch bei ihrem Laden durch den Classloader, d.h. bei erstmaligem Zugriff auf diese Klasse dafür, dass die Hibernate-Konfiguration (`hibernate.cfg.xml`) und die XML-Mappings (`Person.hbm.xml` und `Address.hbm.xml`) eingelesen werden.

```
package de.doubleslash.demo.util;

import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {
    private static SessionFactory sessionFactory;
    private static Configuration configuration;

    static {
        try {
            configuration = new Configuration();
            sessionFactory = configuration.configure()
                .buildSessionFactory();
        }
        catch (Throwable t) {
            log.error("Build of SessionFactory failed.", t);
            throw new ExceptionInInitializerError(t);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
}
```

## » Ein einfaches Beispiel - Speichern«

Es ist soweit: Die eigentliche Anwendung kann implementiert werden. Wir werden uns im weiteren Verlauf die Operationen *speichern*, *lesen*, *ändern* und *löschen* für unsere zuvor erstellten Objekte `Person` und `Address` ansehen.

Zu Demonstrationszwecken werden dabei direkt die Methoden der `Session` verwendet, welche üblicherweise in einer DAO-Klasse weggekapselt sind. Zudem wird das Exception-Handling vereinfacht dargestellt.

- Nebenstehendes Beispiel zeigt die Operation *speichern*.

```
public class StoreDemo {
    public static void main(String[] args) {
        Address address = new Address();
        address.setZip("88045");
        address.setCity("Friedrichshafen");

        Person person = new Person();
        person.setName("Müller");
        person.addAddress(address);

        SessionFactory sf = HibUtil.getSessionFactory();
        Session session = null;
        Transaction tx = null;

        try {
            session = sf.getCurrentSession();
            tx = session.beginTransaction();

            session.save(person); // Person & Address -> DB

            tx.commit();
        }
        catch (Exception e) {
            rollback(tx);
        }
    }
}
```

## » Ein einfaches Beispiel - Lesen«

- Nebenstehendes Beispiel zeigt die Operation *lesen*.
- Das gewünschte `Person` Objekt wird dabei mittels angenommenerweise bekannter ID aus der Datenbank gelesen.
- Da von Hibernate standardmässig nur angeforderte Objekte geholt werden und wir dies im XML-Mapping nicht anderst spezifiziert haben, werden die Adressen der Person beim Zugriff darauf in einem zweiten select nachgeladen. Dies nennt man Lazy-Initialisierung.

```
package de.doubleslash.demo;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;

public class ReadDemo {
    public static void main(String[] args) {
        SessionFactory sf = HibUtil.getSessionFactory();
        Session session = null;
        Transaction tx = null;

        try {
            session = sf.getCurrentSession();
            tx = session.beginTransaction();

            Person person = session.get(Person.class, 1);

            Set<Address> addresses = person.getAddresses();

            tx.commit();
        }
        catch (Exception e) {
            rollback(tx);
        }
    }
}
```

## » Ein einfaches Beispiel - Ändern«

- Nebenstehendes Beispiel zeigt die Operation *ändern*.
- Das gewünschte `Person` Objekt wird dabei mittels angenommenerweise bekannter ID aus der Datenbank gelesen.
- Da wir im XML-Mapping spezifiziert haben, dass die Operationen *save* und *update* auf verknüpfte `Address` Objekte kaskadiert werden sollen (`cascade=save-update`), genügt das Speichern der `Person` um auch automatisch die Änderungen an allen Adressen in die Datenbank zu persistieren.

```
public class ChangeDemo {
    public static void main(String[] args) {
        SessionFactory sf = HibUtil.getSessionFactory();
        Session session = null;
        Transaction tx = null;

        try {
            session = sf.getCurrentSession();
            tx = session.beginTransaction();

            Person person = session.get(Person.class, 1);
            person.setName("Meier");

            Address address = person.getAddresses().get(0);
            address.setZip("88046");

            session.saveOrUpdate(person);

            tx.commit();
        }
        catch (Exception e) {
            rollback(tx);
        }
    }
}
```

## » Ein einfaches Beispiel - Löschen«

- Nebenstehendes Beispiel zeigt die Operation *löschen*.
- Das zu löschende `Person` Objekt muss zuvor gelesen werden, bevor es gelöscht werden kann. dies erfolgt wieder anhand der angenommenerweise bekannten ID.
- Da wir im XML-Mapping spezifiziert haben, dass die Operation *delete* auf verknüpfte `Address` Objekte kaskadiert werden soll (`cascade=delete`), genügt das Löschen der `Person` um auch automatisch alle Adressen aus der Datenbank zu löschen.

```
public class DeleteDemo {
    public static void main(String[] args) {
        SessionFactory sf = HibUtil.getSessionFactory();
        Session session = null;
        Transaction tx = null;

        try {
            session = sf.getCurrentSession();
            tx = session.beginTransaction();

            Person person = session.get(Person.class, 1);
            session.delete(person);

            tx.commit();
        }
        catch (Exception e) {
            rollback(tx);
        }
    }
}
```

## » Ein einfaches Beispiel – Query mittels HQL «

- Nebenstehendes Beispiel zeigt wie Objekte mit bestimmten Eigenschaften mittels HQL aus der Datenbank gelesen werden können.
- Zu diesem Zweck erzeugen wir ein HQL Query mit unserem Selektionskriterium. HQL ist von der Syntax stark an SQL angelehnt. Es ist jedoch eine eigenständige, mächtige, objektorientierte Abfragesprache.
- Nicht gesetzte Attribute werden standardmässig nicht berücksichtigt.
- In dem Beispiel werden alle Personen aus der Datenbank gelesen, welche eine Adresse mit dem Ort „Friedrichshafen“ zugeordnet haben.

```
public class HqlDemo {
    public static void main(String[] args) {
        SessionFactory sf = HibUtil.getSessionFactory();
        Session session = null;
        Transaction tx = null;
        Query query = null;

        try {
            session = sf.getCurrentSession();
            tx = session.beginTransaction();

            query = session.createQuery("from Person p"
                + "where p.address.city = :city");
            query.setString("Friedrichshafen");

            List<Person> persons = query.list();

            tx.commit();
        }
        catch (Exception e) {
            rollback(tx);
        }
    }
}
```

## » Ein einfaches Beispiel – Query mittels Criteria«

- Nebenstehendes Beispiel zeigt wie Objekte mit bestimmten Eigenschaften mittels eines **Criteria**s aus der Datenbank gelesen werden können.
- Zu diesem Zweck erzeugen wir ein **Criteria** Objekt entsprechend unseres Selektionskriteriums.
- In dem Beispiel werden alle Personen aus der Datenbank gelesen, welche eine Adresse mit einem Ort haben, dessen Name auf „hafen“ endet.

```
public class ExampleDemo {
    public static void main(String[] args) {
        SessionFactory sf = HibUtil.getSessionFactory();
        Session session = null;
        Transaction tx = null;
        Criteria crit = null;

        try {
            session = sf.getCurrentSession();
            tx = session.beginTransaction();

            crit = session.createCriteria(Person.class);
            crit.createCriteria(Address.class)
                .add(Restrictions.like("city", "%hafen"));
            List<Person> persons = crit.list();

            tx.commit();
        }
        catch (Exception e) {
            rollback(tx);
        }
    }
}
```



## » Ein einfaches Beispiel – Query mittels Example«

- Nebenstehendes Beispiel zeigt wie Objekte mit bestimmten Eigenschaften mittels eines Examples aus der Datenbank gelesen werden können.
- Zu diesem Zweck erzeugen wir ein `Person` Objekt und setzen dessen Attribute entsprechend der gewünschten Selektionskriterien.
- Nicht gesetzte Attribute werden standardmässig nicht berücksichtigt.
- In dem Beispiel werden alle Personen aus der Datenbank gelesen, welche eine Adresse mit dem Ort „Friedrichshafen“ zugeordnet haben.

```
public class ExampleDemo {
    public static void main(String[] args) {
        SessionFactory sf = HibUtil.getSessionFactory();
        Session session = null;
        Transaction tx = null;
        Criteria crit = null;

        try {
            session = sf.getCurrentSession();
            tx = session.beginTransaction();

            Address address = new Address();
            address.setCity("Friedrichshafen");

            Example example = Example.create(address);

            crit = session.createCriteria(Person.class);
            crit.createCriteria("address").add(example);
            List<Person> persons = crit.list();

            tx.commit();
        }
        catch (Exception e) {
            rollback(tx);
        }
    }
}
```

## »Exception Handling«

- Hibernate wirft seit Version 3.x ausschliesslich **ungeprüfte** Exceptions welche von `RuntimeException` abgeleitet sind.
- Alle von Hibernate geworfenen Exceptions sind **fatal**, d.h. die aktuelle `Transaction` muss rückgängig gemacht werden (**roll back**) und die aktuelle `Session` geschlossen werden.
- Von Hibernate geworfene Exceptions sollten nicht zu Validierungszwecken genutzt werden. Probleme: Schlechte Skalierbarkeit, Arbeitseinheit wird ungültig.

```
SessionFactory sf = HibernateUtil.getSessionFactory();
Session session = null;
Transaction tx = null;

try {
    session = sf.openSession();
    tx = session.beginTransaction();

    // do something

    tx.commit();
}
catch (RuntimeException e) {
    try {
        tx.rollback();
    }
    catch (RuntimeException ex) {
        log.error("Couldn't roll back transaction", ex);
    }

    throw ex;
}
finally {
    session.close();
}
```

## »Fallstrick Lazy-Initialisierung (1)«

Ein klassischer Fehler bei Hibernate-Anwendungen ist das Auftreten einer `LazyInitialisationException`.

- Standardmässig werden beim Laden persistenter Objekte damit in Relation stehende andere Objekte nicht ebenfalls aus der Datenbank geladen.
- Stattdessen enthält das geladene persistente Objekt einen Proxy, welcher das in Relation stehende Objekt erst beim Zugriff darauf nachträglich aus der Datenbank lädt.
- Dies funktioniert nur, solange man sich innerhalb einer gültigen `Session` befindet.

```
public class Person {
    private Integer id;
    private String name;
    private Address address;

    public Address getAddress() {
        return this.address;
    }
}
```

```
SessionFactory sf = HibernateUtil.getSessionFactory();
Session session = null;
Transaction tx = null;

PersonDAO dao = DAOFactory.getPersonDAO();
Person person = null;

try {
    session = sf.getCurrentSession();
    tx = session.beginTransaction();

    person = dao.getPerson(4711);

    tx.commit();
}
catch (Exception e) {
    rollback(tx);
}

person.getAddress(); // <- LazyInitialisationException
```

## »Fallstrick Lazy-Initialisierung (2)«

### Lösungsmöglichkeit 1:

```
SessionFactory sf = HibUtil.getSessionFactory();
Session session = null;
Transaction tx = null;

PersonDAO dao = DAOFactory.getPersonDAO();
Person person = null;
Address address = null;

try {
    session = sf.getCurrentSession();
    tx = session.beginTransaction();

    person = dao.getPerson(4711);
    address = person.getAddress(); // <- keine
                                // Exception

    tx.commit();
}
catch (Exception e) {
    rollback(tx);
}

address.getStreet(); // <- keine Exception
```

### Lösungsmöglichkeit 2:

```
SessionFactory sf = HibUtil.getSessionFactory();
Session session = null;
Transaction tx = null;

PersonDAO dao = DAOFactory.getPersonDAO();
Person person = null;
Address address = null;

try {
    session = sf.getCurrentSession();
    tx = session.beginTransaction();

    person = dao.getPerson(4711);
    Hibernate.initialize(person.getAddress());

    tx.commit();
}
catch (Exception e) {
    rollback(tx);
}

person.getAddress(); // <- keine Exception
address.getStreet(); // <- keine Exception
```

---

## »Fallstrick Lazy-Initialisierung (3)«

---

### Weitere Möglichkeit:

- Deaktivierung des Lazy-Modus auf Objektebene oder Relationsebene in den XML Mapping-Metadaten oder per Annotationen im Java-Code

### Weitere Möglichkeiten die damit in Zusammenhang stehen, jedoch primär auf Performance-Optimierung ausgerichtet sind:

- Prefetching von abhängigen Objekten per Batch-Selektion
- Wahl eines geeigneten Fetch-Modus auf Objektebene oder Relationsebene in den XML Mapping-Metadaten oder per Annotationen im Java-Code
- Wahl eines geeigneten Fetch-Modus selektiv für einzelne HQL-Abfragen oder Criteria-Abfragen

## »Fallstrick Verhalten der Session (1)«

Ein weiterer Fallstrick bei Hibernate-Anwendungen ist die standardmässige Verhaltensweise der `Session`.

- Standardmässig verbleiben alle innerhalb einer `Session` geladenen persistenten Objekte solange in der `Session`, bis diese durch den GC entsorgt wird.
- Standardmässig wird vor jeder Selektion ein **flush** der `Session` durchgeführt. Dabei werden alle darin enthaltenen Objekte überprüft und erforderlichenfalls mit der Datenbank abgeglichen.
- Resultat davon sind unter Umständen Performance- und Speicherprobleme

```
SessionFactory sf = HibernateUtil.getSessionFactory();
Session session = null;
Transaction tx = null;

PersonDAO dao = DAOFactory.getPersonDAO();
Set<Person> persons = null;

List<Integer> allIds = null;
List<Integer> someIds = null;

try {
    session = sf.getCurrentSession();
    tx = session.beginTransaction();

    allIds = dao.getAllPersonIds(); // <- 30.000 Stück
    int count = (int) Math.ceil(allIds.size() / 100);

    for (int i=0; i<count; i++) {
        someIds = getSomeIds(allIds, i, 100); // 100 Stück
        persons = dao.getPersonsByIds(someIds);
        exportToExcel(persons);
    }

    tx.commit();
}
catch (Exception e) {
    rollback(tx);
}
```

## »Fallstrick Verhalten der Session (2)«

Eine mögliche Lösung kann wie in nebenstehendem Beispiel gezeigt aussehen.

- Durch Umstellen des Flush-Modus wird das automatische Flushen der `Session` unterbunden und somit darin enthaltene Objekte nicht mehr vor jeder Selektion überprüft und erforderlichenfalls in die Datenbank weggeschrieben.
- Nicht mehr benötigte Objekte werden durch Leeren der gesamten `Session` aus dem Arbeitsspeicher entfernt.
- Diese Lösung funktioniert nur, wenn die persistenten Objekte bei der Verarbeitung nicht verändert werden.

```
SessionFactory sf = HibernateUtil.getSessionFactory();
Session session = null;
Transaction tx = null;

PersonDAO dao = DAOFactory.getPersonDAO();
Set<Person> persons = null;

List<Integer> allIds = null;
List<Integer> someIds = null;

try {
    session = sf.getCurrentSession();
    session.setFlushMode(FlushMode.MANUAL); // <- Lösung
    tx = session.beginTransaction();

    allIds = dao.getAllPersonIds(); // <- 30.000 Stück
    int count = (int) Math.ceil(allIds.size() / 100);

    for (int i=0; i<count; i++) {
        someIds = getSomeIds(allIds, i, 100); // 100 Stück
        persons = dao.getPersonsByIds(someIds);
        exportToExcel(persons);
        session.clear(); // <- Lösung
    }

    tx.commit();
}
catch (Exception e) {
    rollback(tx);
}
```

---

## »Tools und Vorgehensweisen«

---

Hibernate unterstützt den Entwickler durch diverse Tools bei der Generierung von:

- Domain Model Java-Klassen
- Data Access Java-Klassen
- SQL DDL-Skripte
- XML Mapping-Metadaten

Tools stehen zur Verfügung als:

- Eclipse-Plugins
- ANT-Tasks

Bei der Entwicklung von Anwendungen mit Hibernate gibt es folgende möglichen Vorgehensweisen

- Top down
- Bottom up
- Middle out
- Meet in the middle



» Vorgehensweise Top down«

Ausgangsbasis: Existierende **Java-Klassen** entsprechend eines Domain Models.

*Gute Vorgehensweise wenn noch kein Datenbank-Schema existiert.*

1. Generierung der **Mapping-Metadaten** anhand der Java-Klassen  
Tool: *java2hbm*

2. Generierung des **Datenbank-Schema** anhand der Mapping-Metadaten  
Tool: *hbm2ddl*



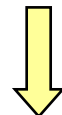
```
public class Person {
    private Integer id;
    private String firstName;
    private String lastName;
    private int age;

    // Getter-/Setter Methoden
}
```



```
<?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="de.doubleslash.demo.model">
  <class name="Person" table="person">
    <id name="id" type="integer" column="id">
      <generator class="native"/>
    </id>
    <property name="firstName" type="string"
      column="firstname" length="30"/>
    <property name="lastName" type="string"
      column="lastname" length="30" not-null="true"/>
    <property name="age" type="integer" column="age"/>
  </class>
</hibernate-mapping>
```




```
CREATE TABLE person (
  id                INTEGER NOT NULL AUTO_INCREMENT,
  firstname         VARCHAR(30),
  lastname         VARCHAR(30) NOT NULL,
  age              INTEGER
);
```

» *Vorgehensweise Bottom up* «


Ausgangsbasis: Existierendes Datenmodell und **Datenbank-Schema**.

*Gute Vorgehensweise wenn eine Anwendung zu einem bestehenden Datenbank-Schema entwickelt werden soll.*



1. Generierung von **Metadaten** anhand des Datenbank-Schemas  
Tool: *?ddl2hbm?*
2. Generierung der **Mapping-Metadaten** anhand der Metadaten  
Tool: *hbm2hbmxml*
3. Generierung der Domain-Model **Java-Klassen** und optional **DAO-Klassen** anhand der Mapping-Metadaten  
Tool: *hbm2java*



```
CREATE TABLE person (  
  id                INTEGER NOT NULL AUTO_INCREMENT,  
  firstname         VARCHAR(30),  
  lastname          VARCHAR(30) NOT NULL,  
  age               INTEGER  
);
```



```
<?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="de.doubleslash.demo.model">  
  <class name="Person" table="person">  
    <id name="id" type="integer" column="id">  
      <generator class="native"/>  
    </id>  
    <property name="firstName" type="string"  
      column="firstname" length="30"/>  
    <property name="lastName" type="string"  
      column="lastname" length="30" not-null="true"/>  
    <property name="age" type="integer" column="age"/>  
  </class>  
</hibernate-mapping>
```



```
public class Person {  
  private Integer id;  
  private String firstName;  
  private String lastName;  
  private int age;  
  
  // Getter-/Setter Methoden  
}
```

» *Vorgehensweise Middle out* «

Ausgangsbasis: Existierende XML **Mapping-Metadaten** entsprechend eines Domain Models.

*Mögliche Vorgehensweise wenn man sich gut mit dem Mapping auskennt und dieses als zentralen Pflegepunkt nutzen möchte.*



```
<?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="de.doubleslash.demo.model">
  <class name="Person" table="person">
    <id name="id" type="integer" column="id">
      <generator class="native"/>
    </id>
    <property name="firstName" type="string"
      column="firstname" length="30"/>
    <property name="lastName" type="string"
      column="lastname" length="30" not-null="true"/>
    <property name="age" type="integer" column="age"/>
  </class>
</hibernate-mapping>
```



1. Generierung der Domain-Model **Java-Klassen** und optional **DAO-Klassen** anhand der Mapping-Metadaten  
Tool: *hbm2java*



```
public class Person {
    private Integer id;
    private String firstName;
    private String lastName;
    private int age;

    // Getter-/Setter Methoden
}
```



2. Generierung des **Datenbank-Schema** anhand der Mapping-Metadaten  
Tool: *hbm2ddl*



```
CREATE TABLE person (
    id                INTEGER NOT NULL AUTO_INCREMENT,
    firstname         VARCHAR(30),
    lastname          VARCHAR(30) NOT NULL,
    age               INTEGER
);
```

## » Vorgehensweise Meet in the middle «

Ausgangsbasis: Existierende **Java-Klassen** entsprechend eines Domain Models und ein damit nicht übereinstimmendes **Datenbank-Schema**.

*In diesem Fall können die Hibernate-Tools kaum Unterstützung leisten.*

1. Manuelle Refaktorisierung der Java-Klassen und/oder des Datenbank-Schema.

2. Manuelle Erstellung der XML **Mapping-Metadaten**



```
public class Person {
    private Integer id;
    private String firstName;
    private String lastName;
    private int age;

    // Getter-/Setter Methoden
}
```



```
CREATE TABLE employee (
    name          VARCHAR(50) NOT NULL,
    gender        INTEGER NOT NULL,
    birthdate     DATE NOT NULL,
    salary        NUMBER(7,2) NOT NULL
);
```



```
<?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="de.doubleslash.demo.model">
    <class name="Person" table="person">
        ??? <!-- Abhaengig von den Refaktorisierungsmassnahmen -->
    </class>
</hibernate-mapping>
```

# Herzlichen Dank für Ihr Interesse.

Ihr IT-Partner für  
individuelle Softwarelösungen



doubleSlash Net-Business GmbH  
Müllerstr. 12 B / D-88045 Friedrichshafen  
<http://doubleSlash.de> / [info@doubleSlash.de](mailto:info@doubleSlash.de)

