Chapitre 2

Le framework d'accès aux données Hibernate

Sommaire

- Architecture de Hibernate
- Mise en œuvre de Hibernate
- Mappage simples et mappage d'associations
- HQL et l'API Criteria
- Gestion du cache Hibernate

Architecture de Hibernate

Introduction à Hibernate

- **Hibernate a été fondé en 2001 par Gavin King, qui fait de l'équipe de développement de JBOSS.**
  - Projet open source visant à proposer un outil de mapping entre les objets et des données stockées dans une base de données relationnelle.
  - XML constitue le format de description de la correspondance entre les tables relationnelles et les classes Java.

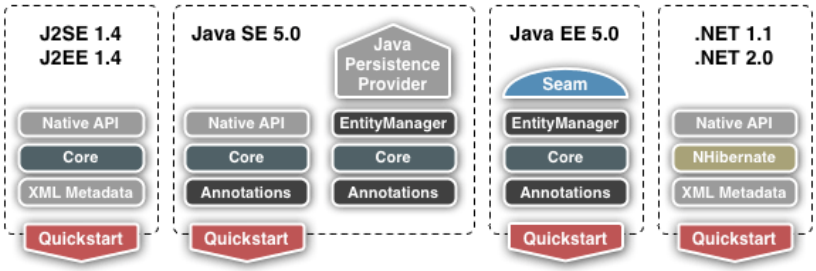
- **Hibernate a toujours suscité l'intérêt de la communauté des développeures Java/JavaEE**
  - Amélioration de la productivité
  - Amélioration de la performance
  - Amélioration de la Maintenabilité
  - Amélioration de la portabilité

- **Le site officiel du projet est : http://www.hibernate.org**

Modules de Hibernate

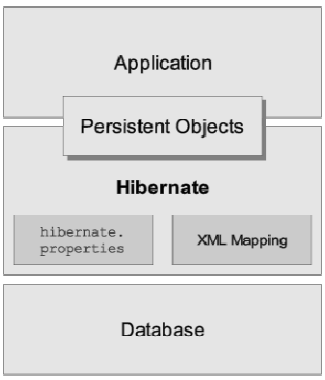
HIBERNATE

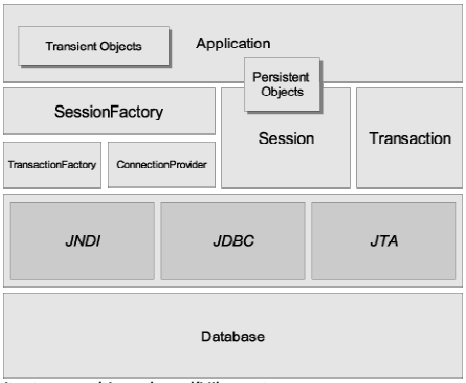
http://www.hibernate.org



Hibernate: Architecture

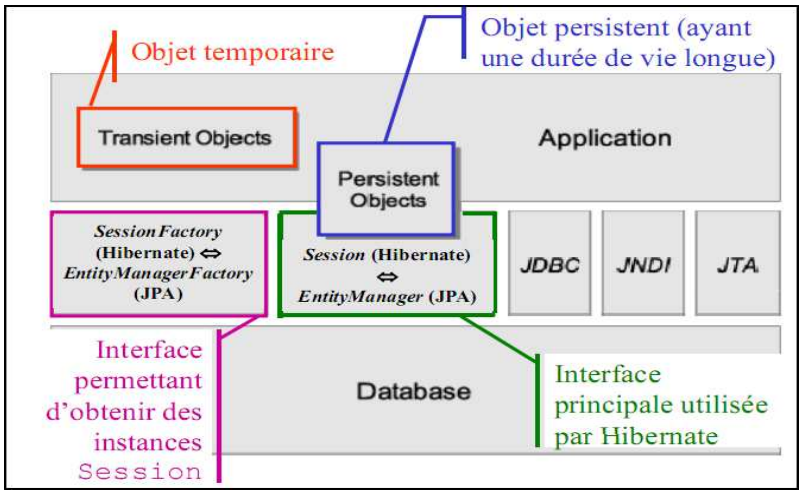
- **Hibernate utilise des API Java existantes dont JDBC, JTA (Java Transaction API) et JNDI (Java Naming and Dictory Interface).**
  - JDBC fournit un niveau rudimentaire d'abstraction des fonctionnalités communes aux bases de données relationnelles, ce qui permet à pratiquement toutes les bases de données dotées d'un pilote JDBC d'être supportées par Hibernate. JNDI et JTA permettent à Hibernate d'être intégré avec des serveurs d'applications J2EE.





Architecture multicouches d'Hibernate
extrait du livre « Hibernate » de Gavin King et Christian Bauer.

Hibernate: Architecture



Hibernate: Interfaces centrales

- **Grâce à ces interfaces, on peut stocker et récupérer des objets persistants et contrôler des transactions.**
- **Interface Session**
  - L'interface Session est l'interface principale utilisée pour les applications Hibernate. Les instances de session sont légères et peu coûteuses à créer et à détruire.
  - C'est un objet **mono-threadé**, à durée de vie courte, qui représente une conversation entre l'application et l'entrepôt de persistance.
  - L'interface session est quelquefois appelée « gestionnaire de persistance » car il s'agit également d'une interface pour les opérations liées à la persistance comme le stockage et la récupération des objets.
- **SessionFactory**
  - L'application obtient des instances Session à partir d'une SessionFactory.
  - La sessionFactory est loin d'être légère. Elle est destinée à être partagée par de nombreux thread d'application. Il y a en général une unique SessionFactory pour l'application entière.

Hibernate: Interfaces centrales

- **Interface configuration:**
  - L'objet Configuration est utilisé pour configurer et amorcer Hibernate. L'application utilise une instance configuration pour spécifier l'emplacement de mapping et de propriétés spécifiques à Hibernate, puis pour créer la SessionFactory.
- **Interfaces Query et Criteria:**
  - L'interface Query permet de faire porter des requêtes sur la base de données et de contrôler la manière dont la requête est exécutée .
  - Une instance Query est utilisée pour lier les paramètres de requête, limiter le nombre de résultats retournés et enfin exécuter la requête.
  - Les instances Query sont légères et ne peuvent pas être utilisées en dehors de la Session qui les a crées.
  - L'interface Creteria est très similaire, elle permet de créer et d'exécuter des requêtes par critères orientés objet.
- **Interface Transaction :**
  - L'interface Transaction est une API optionnelle.
  - C'est un objet mono-threadé à vie courte utilisé par l'application pour définir une unité de travail.
  - Elle abstrait le code de l'application de l'implémentation sous-jacentes gérant les transactions qu'elles soient JDBC, JTA ou CORBA.
  - Une Session peut fournir plusieurs Transactions dans certains cas.

Mise en œuvre de Hibernate

Hibernate: Mise en œuvre

Fichiers nécessaires avec Hibernate Core :

1. **hibernate.cfg.xml : fichier de configuration globale contenant**
   - Les paramètres de connexion à la base de données (pilote, login, mot de passe,url,etc.)
   - Le dialecte SQL de la base de données
   - La gestion de pool de connexions
   - Le niveau de détails des traces etc.
2. **Pour chaque classe persistante :**
   - **ClassePersistante.java** :Implémentation POJO(Plain Old Java Objects) de la classe
   - **ClassePersistante.hbm.xml** : Fichier XML de correspondance (mapping)

Et Optionnellement :

- **ClassePersistanteHome.java** : Implémentation du DAO (Data Access Object) pour l'isolation avec la couche de persistance – Optionnel
- **ClassePersistante.sql** :Code SQL de création de la ou les relations correspondantes Optionnel – pouvant être réalisé par Hibernate

Hibernate: hibernate.cfg.xml

Exemple de fichier de configuration hibernate.cfg.xml :

```
<?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">org.postgresql.Driver
     </property>
    <property name="hibernate.connection.password">passwd</property>
    <property name="hibernate.connection.url">jdbc:postgresql:BDTest2</property>
    <property name="hibernate.connection.username">login</property>
   <!-- SQL dialect -->
   <property name="dialect">org.hibernate.dialect.PostgreSQLDialect</property>
   <!-- Enable Hibernate's automatic session context management -->
   <property name="current_session_context_class">thread</property>
   <!-- Disable the second-level cache  -->
   <property  name="cache.provider_class">org.hibernate.cache.NoCacheProvider
    </property>
   <!-- Echo all executed SQL to stdout -->
   <property name="show_sql">true</property>
   <!-- Drop and re-create the database schema on startup -->
   <property name="hbm2ddl.auto">create</property>
   <mapping resource="events/Event.hbm.xml"/>
   <mapping resource="events/Person.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

Hibernate: Obtention de la session Hibernate

```
private static Configuration configuration;
private static SessionFactory sessionFactory;
private Session s;
try {
      // étape 1
      configuration = new Configuration();
      // étape 2
      sessionFactory =
      configuration.configure().buildSessionFactory();
      // étape 3
      s = sessionFactory.openSession();
} catch (Throwable ex) {
log.error("Building SessionFactory failed.", ex);
throw new ExceptionInInitializerError(ex);
}
```

Consultation du fichier hibernate.cfg.xml présent dans le classpath de l'application

Analyse du fichier de mapping

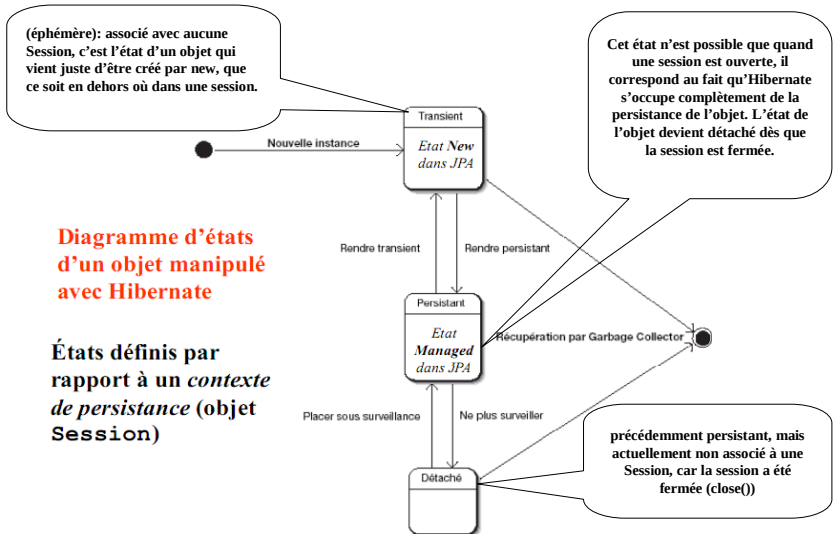
Les fichiers hbm de mappage

- **Le mappage se fait avec les fichiers .hbm ou avec les annotations**
- **Les métadonnées de mappage définissent :**
  - Le mappage Attributs (Classes) / Colonnes (tables)
  - Mappage de la clé primaire Primary Key mapping & generation Scheme
  - Associations
  - Collections
  - Caching Settings
  - Custom SQL
  - …

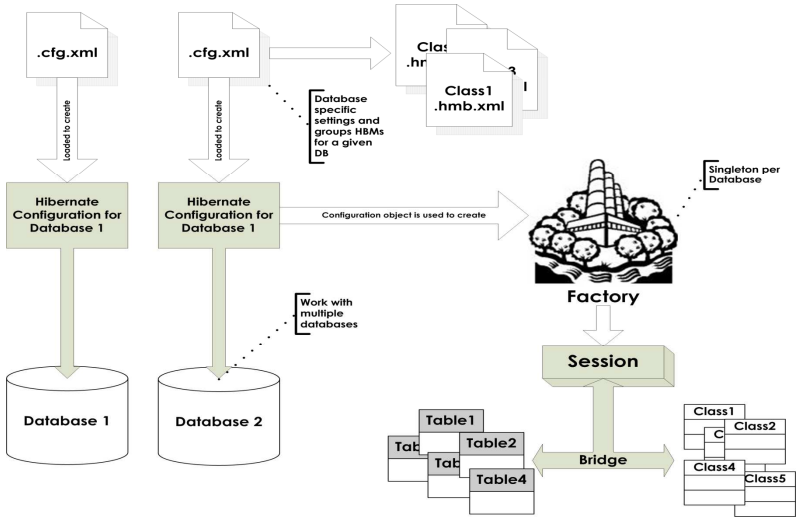
Exemple de fichier de mappage (hbm)

```
<hibernate-mapping    package="tn.insat.genielogiciel4.domain">
   <class name="Address">
      <id name="Id" column="PK_ID" type="integer">
         <generator class="identity" />
      </id>
      <property name="StreetAddress" />
      ...
      <property name="AptNumber" />
   </class>
</hibernate-mapping>
```


Cycle d'une vie d'une entité Hibernate



The big picture



Utilisation des annotations

- **Les méta-données de mappage O/R peuvent être décrites avec des annotations.**
  - Ceci réduit les artéfacts du projet en éliminant les fichiers hbm
  - Les métadonnées des Annotations Hibernate sont conformes au annotations standard JPA
- **Pour utiliser Hibernate Annotations, il faut télécharger les libraries Hibernate Annotations et Hibernate EntityManager**
- **Les classes persistantes sont annotés @Entity. L'annotation @Table annotation indique le nom de la table dans la BDD.**
  - Les classes mappées doivent avoir une annotation indiquant la propriété utilisée comme clé : @Id

```
@Entity
@Table(name = "employee")
public class Employee implements Serializable {
  public Employee() {   }
  @Id
  @Column(name = "id")
  Integer id;

  @Column(name = "name")
  String name;

  public Integer getId() {
    return id;
  }

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

  public String getName() {
    return name;
  }

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

}
```

Annotations Overview

- **Core**
  - @Entity
  - @Table
  - @Id
  - @Basic
  - @Column
  - @Transient
  - @Enumerated
  - @Temporal
  - @Type
- **Relationships**
  - @ManyToOne
  - @OneToOne
  - @OneToMany
  - @JoinColumn
- **Inheritance**
  - @MappedSuperclass
  - @Inheritance
  - @DiscriminatorColumn
  - @DiscriminatorValue

Mappage de classes simples

Create the Location class

Location class

Location
id: int
name: String
address: String

LOCATIONS table (Hibernate can auto-generate this)

LOCATIONS		
PK	id	INTEGER
	name	VARCHAR(80)
	address	VARCHAR(160)

Write the Location class

```
package eventmgr.domain;
public class Location {
    private int id;
    private String name;
    private String address;

    public Location() { }  // a no argument constructor

    public int getId( ) { return id; }
    public void setId( int id ) { this.id = id; }
    public String getName( ) { return name; }
    public void setName( String n ) { this.name = n; }
    public String getAddress( ) { return address; }
    public void setAddress( String a ) { address = a; }
}
```

Use JavaBean conventions in Persistent object classes.

Hibernate can access private methods

```
package eventmgr.domain;
public class Location {
    private int id;
    private String name;
    private String address;

    public Location() { }

    public int getId( ) { return id; }
    private void setId( int id ) { this.id = id; }
    public String getName( ) { return name; }
    private void setName( String n ) { this.name = n; }
    public String getAddress( ) { return address; }
    private void setAddress( String a ) { address = a; }
}
```

OK to use "private" or "protected" for *mutators*.

Hibernate can access private data, too

```
public class Location {
   private int id;
   private String name;

   private void setName( String name ) {
        if ( name.length() < 3 )
                new RuntimeException("name too short");
        ...
   }
```

Some mutator methods contain data validity checks or other complicated logic.

to tell Hibernate to set the field values directly (don't use the "set" method) in the class mapping file write. The Default value is **property**

```
<hibernate-mapping default-access="field">
   ...
```

Schema to create Locations table

This works for MySQL.

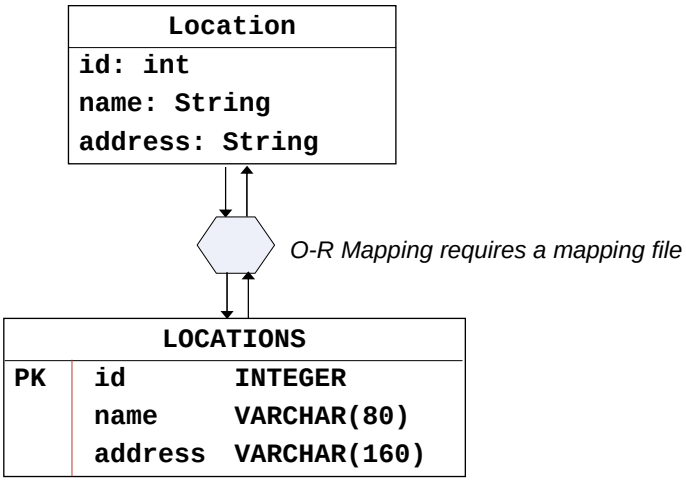
Hibernate can generate table schema at runtime.

```
CREATE TABLE locations (
    id          INTEGER NOT NULL,
    name        VARCHAR(80) NOT NULL,
    address     VARCHAR(160),
    PRIMARY KEY(id)
) DEFAULT CHARSET=utf8 ;
```

```
mysql> use eventmgr ;
mysql> source tableschema.sql ;
```


O-R Mapping for the Location class

Map between object attributes and table columns.



Mapping File Location.hbm.xml

An XML file describing how to map object to table.

Filename: `Location.hbm.xml`

```
<!DOCTYPE hibernate-mapping PUBLIC … >

<hibernate-mapping package="eventmgr.domain">
   <class name="Location" table="LOCATIONS">
      <id name="id" column="id">
         <!-- let hibernate choose id for new entities -->
         <generator class="native"/>
      </id>
      <property name="name" column="name" not-null="true"/>
      <property name="address"/>
   </class>
</hibernate-mapping>
```


Mapping File Explained

```
<hibernate-mapping>
   <class name="eventmgr.domain.Location"
          table="LOCATIONS"
          options... >
      <id name="id"
          column="id"
          unsaved-value="0">
         <generator class="native"/>
      </id>
      object attribute mappings
   </class>
</hibernate-mapping>
```

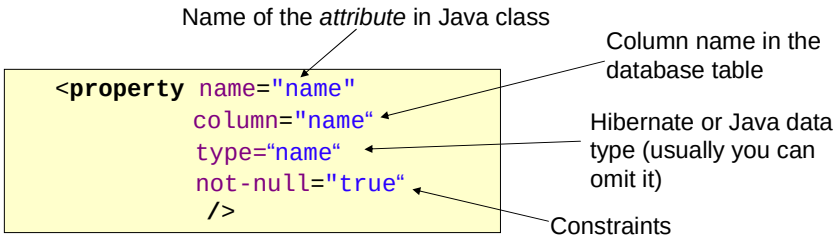
root element of a hibernate mapping

mapping for one class

Every persistent class needs an "**identifier**" attribute and column.

The identifier is used to establish object identity (obj1 == obj2) and locate the table row for a persisted object. The **id** is usually the Primary Key of the table.

Attribute Mapping: <property …/>



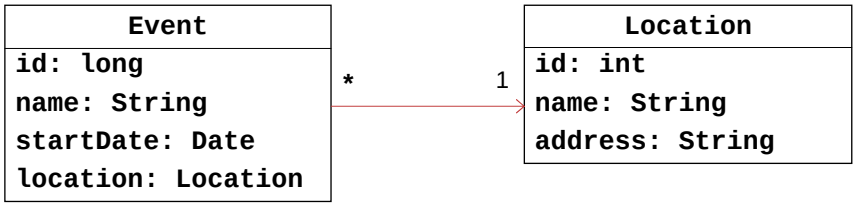

```
<property name="name"
          column="name"
          type="name"
          not-null="true"
          />
```

You omit elements if Hibernate can guess the value itself:

```
<property name="address" column="ADDRESS" type="string"/>
<!-- omit data type and Hibernate will determine it -->
<property name="address" column="ADDRESS"/>
<!-- omit column if same as attribute (ignoring case)-->
<property name="address"/>
```

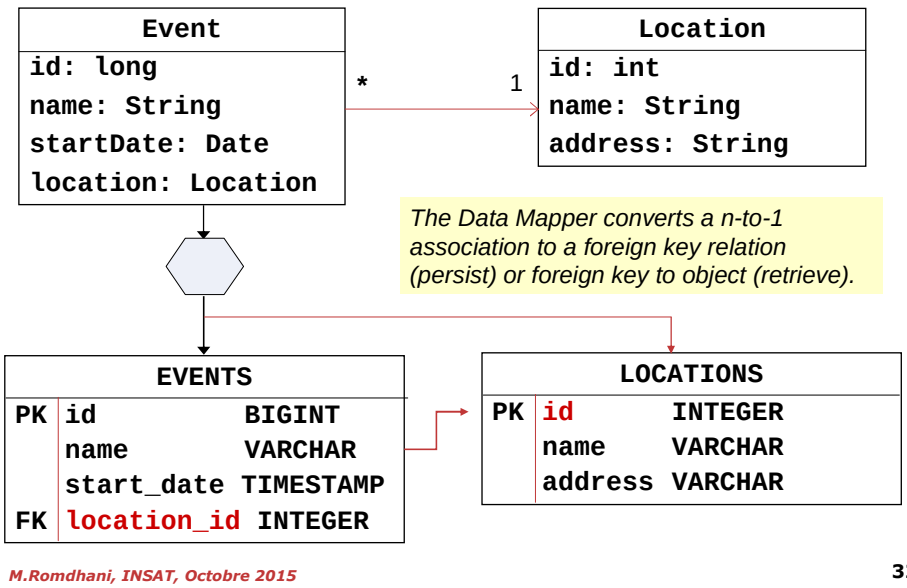

Mappage d'associations

Mapping a Class with Associations



Simplified version of Event class.

O-R Mapping of n-to-1 Associations



Mapping for Event (Event.hbm.xml)

Use `<many-to-one name="attribute" ... />`

to map a reference to another object.

```
<!DOCTYPE hibernate-mapping PUBLIC ... remainder omitted >
<hibernate-mapping package="eventmgr.domain">
   <class name="Event" table="EVENTS">
      ...

      <property name="startDate" column="start_date"
                type="timestamp" />
      <many-to-one name="location"  column="location_id"
                class="Location" />
   </class>
</hibernate-mapping>
```

you can omit class (Hibernate can determine itself)

Test: Save an Event

```
public static void testSave() {
  Location loc1 = new Location( );
  loc1.setName("Kasetsart University");
  loc1.setAddress("90 Pahonyotin Rd, Bangkok");

  Event event = new Event( );
  event.setName("Java Days");
  event.setLocation( loc1 );
  event.setStartDate( new Date(108,Calendar.JULY, 1) );

  Session session = HibernateUtil.getCurrentSession();
  Transaction tx = session.beginTransaction();
  session.saveOrUpdate( event );
  tx.commit();
  System.out.println("Event saved");
}
```

Did you get an Error?

The Location doesn't exist in database (*transient object*).

```
Exception in thread "main“
     org.hibernate.TransientObjectException:
   object references an unsaved transient instance
- save the transient instance before flushing:
eventmgr.domain.Location
```


Persisting the Event location

Solutions:

1. save location during the transaction (manual save)
2. tell Hibernate to ***cascade*** the save operation (automatically save Location)

```
<many-to-one name="location"  column="location_id"
             class="Location"
             cascade="save-update" />
```

cascade="none"		don't cascade operations (the default)
	"all"	cascade all operations (be careful)
	"save-update"	cascade save and updates
	"delete-orphan"	cascade all, delete unreferenced orphan children

Test: Retrieve an Event

```
public static void testRetrieve() {
  System.out.println("Retrieving event");
  Session session = HibernateUtil.getCurrentSession();
  Transaction tx = session.beginTransaction();

  Query query = session.createQuery(
       "from Event e where e.name= :name");
  query.setParameter("name", "Java Days");
  // query.list() returns objects, cast to List<Location>
  List<Event> list = (List<Event>)query.list( );
  tx.commit();

  for(Event e : list ) out.printf("%d %s %s\n",
       e.getId(), e.getName(), e.getLocation().getName()
  );
}
```


Lazy Instances and Proxies

```
  Transaction tx = session.beginTransaction();
  Query query = session.createQuery(
       "from Event e where e.name=:name");
  query.setParameter("name", "Java Days");
  List<Event> list = (List<Event>)query.list( );
  tx.commit();

  for(Event e : list ) out.printf("%d %s %s\n",
       e.getId(), e.getName(), e.getLocation().getName()
  );
```

Error: *LazyInstantiationException*

- Hibernate uses *lazy instantiation* and proxy objects
- Hibernate *instantiates* the location object when it is first accessed
- We closed the transaction *before* accessing the location

Two Solutions

1. **Modify our code: getLocation( ) before closing the session.**

```
List<Event> list = (List<Event>)query.list( );
for(Event e : list ) out.printf("%d %s %s\n",
     e.getId(), e.getName(), e.getLocation().getName()
);
tx.commit( );
```

2. **Tell Hibernate not to use lazy instantiation of Location objects (in `Location.hbm.xml`)**

```
<class name="Location" table="LOCATIONS" lazy="false">
        ...
</class>
```


Mappage des relations d'héritage

Stratégies de mappage des relations d'héritage

- Table Per Concrete Class
- Table Per sub-class
- Table Per Hierarchy

Table Per Concrete Class

```
public class CDAccount extends Account {
        Integer term;

}
```

accounts

id	balance	created_on

cd_accounts

id	balance	created_on	term

- Mapping strategy #1: map them as two completely unrelated classes
- Mapping strategy #2: `<union-subclass>`
  - Polymorphic query

Table Per Subclass

```
<joined-subclass name="CDAccount" table="cd_accounts">
    <key column="account_id"/>
    <property name="term"/>
</joined-subclass>
```

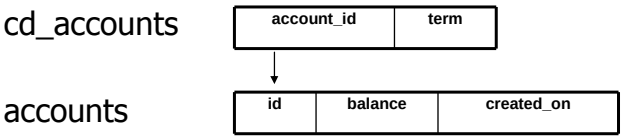


Table Per Hierarchy

```
<discriminator column="account_type" type="string"/>

<subclass name="CDAccount" discriminator-value="CD">
    <property name="term"/>
</subclass>
```

accounts

id	account_type	balance	created_on	term

Le langage HQL

Hibernate Query Language

- **HQL is a language for talking about "sets of objects"**
- **Make SQL be object oriented**
  - Classes and properties instead of tables and columns
  - Polymorphism
  - Associations
  - *Much* less verbose than SQL
- **Full support for relational operations**
  - Inner/outer/full joins, cartesian products
  - Projection
  - Aggregation (max, avg) and grouping
  - Ordering
  - Subqueries
  - SQL function calls

Hibernate Query Language

Simplest HQL Query:

```
from AuctionItem
```

i.e. get all the AuctionItems:

```
List allAuctions = session.createQuery("from AuctionItem")
    .list();
```


Hibernate Query Language

More realistic example:

```
select item
from AuctionItem item
    join item.bids bid
where item.description like 'hib%'
    and bid.amount > 100
```

i.e. get all the AuctionItems with a Bid worth > 100 and description that begins with "hib"

Hibernate Query Language

Projection:

```
select item.description, bid.amount
from AuctionItem item
    join item.bids bid
where bid.amount > 100
order by bid.amount desc
```

i.e. get the description and amount for all the AuctionItems with a Bid worth > 100

Hibernate Query Language

Aggregation:

```
select max(bid.amount), count(bid)
from AuctionItem item
    left join item.bids bid
group by item.type
order by max(bid.amount)
```

Hibernate Query Language

Runtime fetch strategies:

```
from AuctionItem item
     left join fetch item.bids
     join fetch item.successfulBid
where item.id = 12

AuctionItem item = session.createQuery(…)
     .uniqueResult(); //associations already fetched
item.getBids().iterator();
item.getSuccessfulBid().getAmount();
```


Using Named Parameters in Query

```
// use the existing session factory
Session session = sessionFactory.openSession();

// Hibernate Query Language (HQL) can use named params
Query query = session.createQuery(
                    "from Event where name=:name");
query.setParameter( "name", "Java Days");
List events = query.list( );

out.println("Upcoming Java Days events: ");
for( Object obj : events ) {
          Event event = (Event) obj;
          String name = event.getName( );
          Location loc = event.getLocation( );
          ...
```

L'API Criteria

Hibernate Criteria Query API

- Hibernate 3.0 provides three full-featured query facilities:
  1. Hibernate Query Language
  2. Hibernate Criteria Query API
  3. Hibernate Native Query
- The Criteria interface allows to create and execute object-oriented queries. It is powerful alternative to the HQL but has own limitations. Criteria Query is used mostly in case of multi criteria search screens, where HQL is not very effective.

Restrictions

- Simple and powerful query building:

```
Criteria criteria = session.createCriteria(PaymentModelImpl.class);
criteria.add(Restrictions.like("description", "Комплект", MatchMode.START));

criteria.add(Restrictions.or(Restrictions.gt("amount", new Long(200)),
Restrictions.like("description", "маса", MatchMode.ANYWHERE)));

criteria.createCriteria("merchant").add(Restrictions.eq("url",
"http://www.insat.rnu.tn/"));

List<PaymentModel> result = criteria.list();
```


Find by Example

- A search prototype entity can be used to model restrictions on the properties set:

```
PaymentModel example = new OnlineShoppingModelImpl();
example.setDescription("example");

Criteria criteria = session.createCriteria(PaymentModelImpl.class);

criteria.add(Example.create(example).ignoreCase().enableLike(MatchMode.ANY
WHERE));
List<PaymentModel> result = criteria.list();
```

Detached Criteria

- **A detached criteria can be:**
  - serialized to a byte stream
  - sent over the wire

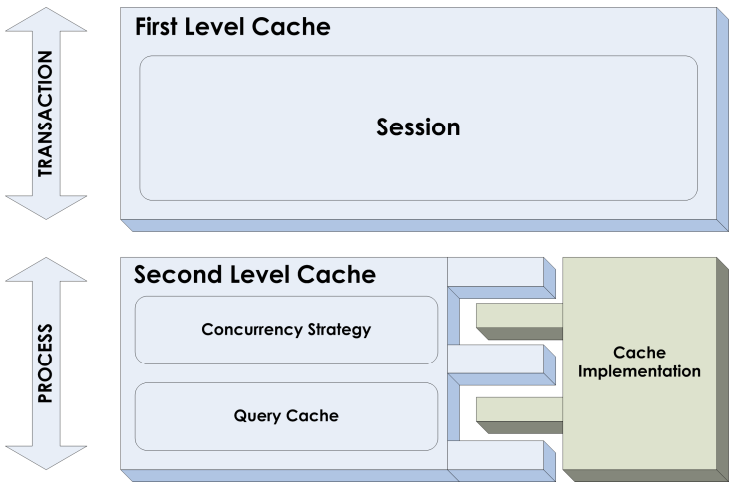
```
DetachedCriteria dc = DetachedCriteria.forClass(PaymentModelImpl.class);
dc.add(Restrictions.gt("amount", new Long(500)));
dc.add(Restrictions.eq("customer.id", customer.getId()));

byte[] bytes = SerializationHelper.serialize(dc);
```


Gestion du cache Hibernate

Caching in Hibernate

- **1st Level**
  - Transaction scoped cache provided by the Session.
  - Always available
- **2nd Level**
  - Process scoped cache
  - Shared by all Sessions
  - Optional
  - Pluggable implementation, can be clustered
- **Query**
  - Use with care, profile first

Caching in Hibernate



Caching in Hibernate

- **Caching setting in HBM for Read Only**

```
<class name="ReferenceData"
        table="REFDATA">
   <cache usage="read-only" />
..
</class>
```


Caching in Hibernate

- **Caching setting in HBM for Read/Write**
- **Profile the application before applying**

```
<class name="ReadMostly"
        table="READMOSTLY">
   <cache usage="nonstrict-read-write" />
..
</class>
```

Caching in Hibernate

- **Query Caching, most difficult to get right**
- **Analyze/Profile First. Let usage determine which queries to cache**

```
String queryString = "from ...";
List hits = session
        .createQuery(queryString)
        .setObject("user", user)
        .setCacheable(true)
        .list();
```